仮想マシンの中間言語に基づく回帰テスト選択手法

孝 壽 俊 彦[†] 高 田 眞 吾[†] 土 居 範 久[†,††]

回帰テストでは，ソフトウェアに対して行った変更が，予期しない故障を引き起こしていないか検査するために，変更前に通過したテストケースを再実行する．しかしすべてのテストケースを再実行すると非常にコストがかかるため，テストケースを選択的に再実行するための研究が行われてきた．特に再実行しても故障を発見しないテストケースのみを省略する手法は，安全な回帰テスト選択手法と呼ばれる．しかしながら，従来の手法はソースコードの解析に基づいていた．近年，仮想マシンを利用するプログラミング言語によるソフトウェアの開発が，急速に広まりつつある．たとえば Java や Microsoft .Net Framework で開発されたソフトウェアは，プラットフォーム独立な中間言語にコンパイルされる．このような中間言語も，回帰テスト時のテストケースの選択に利用することが可能である．これは，特に Microsoft .Net Framework のように，C#や VisualBasic など，複数のプログラミング言語を対象としている環境では重要である．そこで本論文では，仮想マシンの中間言語に基づく安全な回帰テスト選択手法を提案する．本論文では，特に Microsoft .Net Framework を例として取り上げる．提案手法を評価した結果，回帰テストのコストを平均 40.4%も節約することができた．

Regression Test Selection Based on Intermediate Code for Virtual Machines

Toshihiko Koju,[†] Shingo Takada[†] and Norihisa Doi[†,††]

Regression testing is testing applied to software that has been modified. It basically entails re-testing the software with previous test cases to confirm that the modifications made to the software do not have an adverse effect. But re-executing all test cases is normally cost prohibitive, and thus much research has been done on selecting test cases from a test suite without compromising the reliability of the software. These regression test selection techniques find test cases that will not detect any bugs in the modified software. However, these techniques are based on analysis of source code. Recent programming environments have seen a proliferation of virtual machines. For example, programs written in Java and with the Microsoft .Net Framework are compiled into a platform-independent intermediate code. Such code could also be used for regression test selection. This especially holds for the Microsoft .Net Framework which handles various programming languages, such as Visual Basic and C#. Thus, this paper presents a safe regression test selection technique for virtual machine based programs. We especially target the Microsoft .Net Framework. Evaluation on 10 different examples resulted in an average of a 40.4% decrease in the cost of regression testing.

1. 序　　論

ソフトウェアを変更した際，その変更が予期しない故障を引き起こす可能性がある．そのため，以前通過したテストケースを再実行する必要がある．この種のテストを回帰テストと呼ぶが，これには非常にコストがかかる．たとえば，1 つのテストケースの実行に数千ドルもかかったという例がある[5]．また個々のテストケースの実行にかかるコストが低い場合でも，変更が頻繁に行われる場合には，そのたびに回帰テストも実行されることになるので，最終的な総コストは非常に高くなる．

このため，回帰テストのコスト削減を目的とした研究が多くなされてきた[1),2),5),7),9)]．同時に多くの実証的研究が，それらの有効性を指摘してきた[4),8),10)]．提案された手法の多くは，テストケースを選択的に再実行するというものである．特に，再実行しても故障を検出しないテストケースのみを省略する手法は，**安全な**回帰テスト選択手法と呼ばれる．安全な回帰テスト選択手法では，回帰テストの効果を損なうことなく，そのコストのみを削減することが可能である．

近年，仮想マシンを利用するプログラミング言語に

† 慶應義塾大学
Keio University
†† 中央大学
Chuo University

よるソフトウェアの開発が，急速に広まりつつある．たとえば Java や Microsoft .Net Framework で開発されたソフトウェアは，プラットフォーム独立な中間言語にコンパイルされる．このような中間言語も，回帰テスト時のテストケースの選択に利用することが可能である．

しかし既存の回帰テスト選択手法は，ソースコードの解析に基づくものであった．たとえば Harrold らは Java を対象に，動的束縛や例外処理といった機能も考慮に入れた，最初の安全な回帰テスト選択手法を提案した[5]．オリジナルのソースコードを解析する手法は，Java のように，中間言語が特定の高級言語のためだけに定義されている場合には，大きな問題にならない．しかし Microsoft .Net Framework のように，オリジナルのソースコードが Visual Basic や C#など，様々な高級言語で記述されている場合には，それぞれに異なった手法が必要となってしまう．中間言語に基づいた手法なら，それらをすべて同様に扱うことが可能である．さらに中間言語は，異なった高級言語間で協調を行うために使われる場合もある．たとえば Visual Basic で記述されたクラスを，C#で継承する場合などである．このような場合には，中間言語に基づいた手法が必須となる．

本論文では，仮想マシンの中間言語に基づく安全な回帰テスト選択手法を提案する．提案手法は，Harrold らの手法[5] を基盤としている．本論文では，特に Microsoft .Net Framework を例として取り上げ，Microsoft Intermediate Language（MSIL）に基づく回帰テスト選択システムの実装と，その評価についても述べる．なお，この仮想マシンの仕様は，European Computer Manufacturers Association（ECMA）によって，Common Language Infrastructure（CLI）として標準化されており，そこでは MSIL を Common Intermediate Language（CIL）と呼んでいる☆．

以下，2 章で Harrold らの手法の概要について述べ，次に 3 章で提案手法の詳細について述べる．そして 4 章ではその評価について述べ，最後に 5 章で結論と今後の課題について述べる．

2. 安全な回帰テスト選択手法

安全な回帰テスト選択手法とは，回帰テストの効果を少しも損なうことなく，テストケースの選択を行う手法である．回帰テストにおいて，テストケースの選択は，そのコストを管理するために非常に有効な手法である．本章では，提案手法が基盤としている Harrold らの手法[5] について述べる．

Harrold らの手法における処理の流れは，次のとおりである：

(1) 変更前のプログラムに対してテストスィートを実行し，そのカバレッジ情報を取得する．
(2) 変更前と変更後のプログラムから，Control Flow Graph（CFG）を作成する．
(3) CFG を比較して Dangerous Edge を検出する．
(4) カバレッジ情報と Dangerous Edge を比較して，テストケースを選択する．

Harrold らは，実際には CFG を拡張した Java Interclass Graph（JIG）を用いた．以下では JIG と Dangerous Edge について述べる．

2.1 Java Interclass Graph（JIG）

従来の Control Flow Graph（CFG）では，Java の持つ様々な機能や，内部コードと外部コードとの間の相互作用などを，適切に表現できなかった．そのため従来の CFG を拡張して，Java Interclass Graph（JIG）を定義した．ここで**外部**コードとは，ライブラリなどの変更されないと見なせるコードのことであり，逆に**内部**コードとは，開発中のコードのことである．同様に，内部コードのクラスやメソッドを内部クラスや内部メソッドと呼び，外部コードのクラスやメソッドを外部クラスや外部メソッドと呼ぶ．

JIG の主な拡張点は，以下の 3 点である：

- 内部コードからのメソッド呼び出し
- 外部コードからのメソッド呼び出し
- 例外処理

例外処理については，3.3.4 項で述べる．

内部コードからのメソッド呼び出しは，*Call Edge* という特殊な枝で表す．呼び出し先が動的束縛によって決定される場合は，呼び出される可能性のある各メソッドに対して Call Edge を結び，その動的束縛を引き起こすクラスの名前をラベル付けする．**図 1** に例を示す☆☆．変数 p は，実際にはクラス A，B，C，D，E，F，G のインスタンスを格納している可能性がある．`p.foo()` というメソッド呼び出しは，動的束縛により，クラス A，B，C，D，E，G のインスタンスに対しては `A.foo()` を，クラス F のインスタンスに対しては `F.foo()` を呼び出す．そのため `p.foo()` ノードと `A.foo()`，`F.foo()` ノードとの間に，それぞれに対応する Call Edge を作成する．

外部コードについては，*External Code Node*

☆ http://msdn.microsoft.com/net/ecma/

☆☆ 簡単のため，一部のメソッドをクラスの外側に記述している．

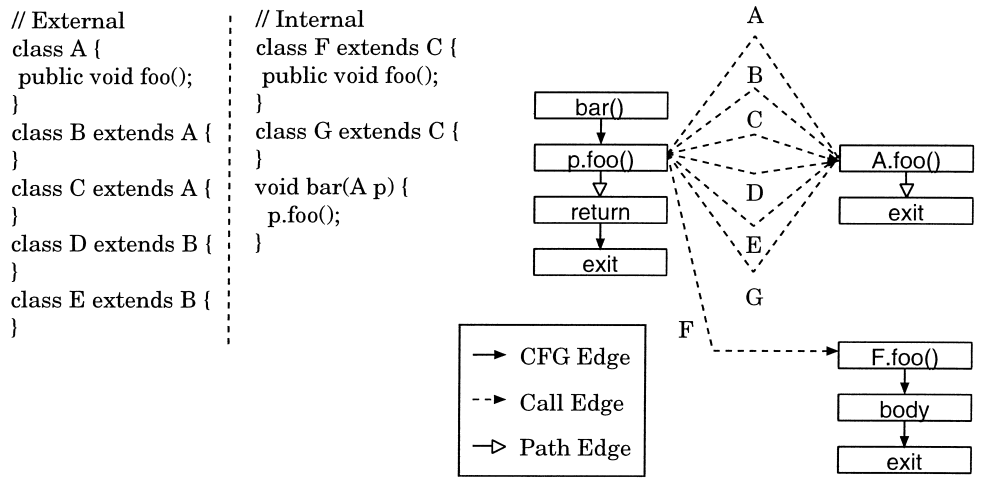


図 1 内部コードからのメソッド呼び出しの例
Fig. 1 An example of a method call from internal code.

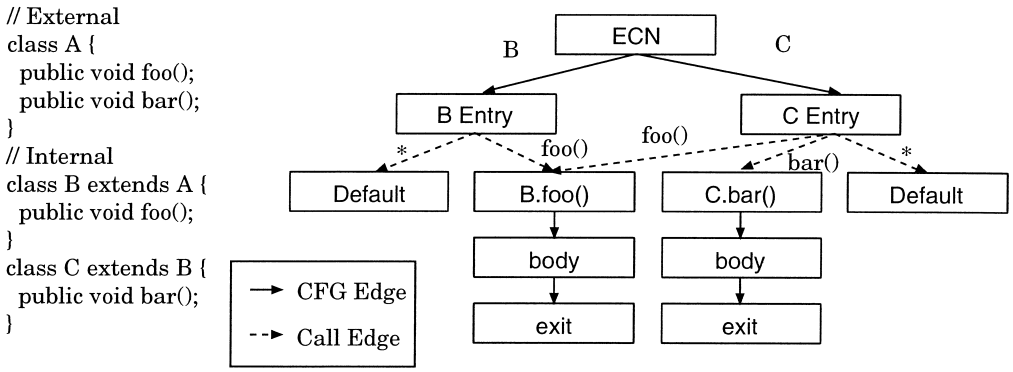


図 2 外部コードからのメソッド呼び出しの例
Fig. 2 An example of a method call from external code.

（*ECN*）という特殊なノードでモデル化する．そして外部コードからアクセス可能な各内部クラスの Entry Node に対して，ECN から CFG Edge を結ぶ．ここで外部コードからアクセス可能な内部クラスとは，外部メソッドを内部メソッドでオーバライドしているクラスとする．そこで次に，そのような各内部メソッドに対して，そのクラスの Entry Node から Call Edge を結ぶ．また内部メソッドでオーバライドされていない外部メソッドは，*Default* という特殊なノードで表す．そして Default に対して，そのクラスの Entry Node から '*' とラベル付けした Call Edge を結ぶ．**図 2** に例を示す．内部クラス B，C 中で外部メソッドをオーバライドしており，外部コードからアクセス可能と見なせるので，`B Entry` と `C Entry` という 2 つの Entry Node を作成し，そこと `B.foo()`，`C.bar()`，`Default` との間に適切な Call Edge を結ぶ．

2.2 Dangerous Edge の検出

Dangerous Edge とは，変更前のプログラムの JIG において，変更部分への制御の流れを表す枝のことである．そのため Dangerous Edge を実行しているテストケースのみ，再実行すればよい．いい換えれば，それらのテストケースの振舞いは変更前と変わっている可能性があるので，選択しなければならない．ただし，ラベル名が '*' である枝が Dangerous Edge となった場合には，そのクラスのインスタンスを生成するすべてのテストケースを再実行する必要がある．

Dangerous Edge の検出は，JIG を深さ優先順に比較する従来の手法[7] に基づいて行った．

3. 仮想マシンの中間言語に基づく安全な回帰テスト選択手法

本章では，仮想マシンの中間言語に基づく安全な回帰テスト選択手法を，Microsoft .Net Framework を

例に述べる．処理の全体的な流れそのものは，2 章で述べた Harrold らの手法[5] と同じである．しかし，MSIL 特有の課題や効率を考慮したため，提案手法の詳細は異なる．以下，提案手法が仮定していることを述べた後，提案手法の安全性について述べ，CFG および Dangerous Edge に関わる問題を克服するための提案事項を述べる．

3.1 仮　　定

提案手法は，Harrold らの手法[5] と同様に，次の 2 点が成り立つことを仮定する．

- テストコードについて：外部コードは内部コード（についての情報）をいっさい利用せず，また内部コードも外部コードもリフレクションを利用しないものとする．この仮定により，内部コードと外部コードの間の相互作用を，大幅に限定することができる．特に外部コードからの内部クラスのメソッドに対する呼び出しを，動的束縛かコールバック関数の機能を利用したものに限定できる．通常，外部コードはライブラリなので，この仮定は成り立つ．
- 実行について：同じコードの断片を，同じ状態と入力を与えて実行した場合，つねに同じパスが実行され，同じ状態と出力が得られるものとする．またスレッド間の相互作用などによる非決定性は，いっさい生じないものとする．この仮定により，プログラムの変更されていない部分のみを実行しているテストケースは，安全に省略することができる．

3.2 安　全　性

実行についての仮定により，プログラムの変更点を CFG の Dangerous Edge として正しく検出することができれば，安全な回帰テスト選択手法を実現できることになる．そのため，従来の CFG や JIG に対する提案手法の拡張部分については，その拡張部分が，対象としている機能に関連した変更点を，Dangerous Edge として正しく検出できることも示す．これにより，それぞれの拡張部分の安全性も示すことができる．

3.3 CFG の作成

CFG を作成する際には，動的束縛におけるクラスと束縛されるメソッドの組合せを，効率良く知る必要がある．またメソッド呼び出しについても，内部コードからのメソッド呼び出しと，外部コードからのメソッド呼び出しを，区別して扱う必要がある．さらに例外処理による制御の遷移も，正確に扱う必要がある．本節では，従来のクラス階層の解析方法を改良し，また CFG を拡張することにより，これらの問題を解決する手法について述べる．

3.3.1 クラス階層の解析

CFG を作成する際には，動的束縛におけるクラスと束縛されるメソッドの組合せを知る必要がある．

MSIL のモジュールは，以下の情報を持っている：モジュール内で定義しているクラス，その親クラスとインタフェース，モジュールから参照しているモジュールとクラス．そのため，Dean らの手法[3] を利用して，参照をたどりながらすべてのモジュールとクラスを解析することにより，完全なクラス階層木を作ることは可能である．

しかしこの手法では，変更を検出するためには必要のない部分まで，解析を行わなければならない．これはクラス階層木に，外部コードで定義，参照しているすべてのクラスを追加するためである．あらかじめ階層木を用意することもできるが，そのサイズは効率に大きな影響を与えることになる．

クラス階層木は，3.3.2 項と 3.3.3 項で，動的束縛に関連した CFG を作成するために用いる．そのためクラス階層木には，それらの CFG で Dangerous Edge を正しく検出するために必要な情報があれば十分である．ここで外部クラスの継承関係や，外部クラスに対して束縛されるメソッドは，変わることがない．また内部クラスの継承関係や，内部クラスに対して束縛されるメソッドは，内部クラスとその先祖の情報から取得することができる．そのため内部クラスとその先祖のみを解析対象とすれば，変更される可能性のある，内部クラスと束縛されるメソッドの組合せを取得することができる．

図 3 にクラス階層木の例を示す．クラス階層は，図 1 と同一のものとする．クラス F と G のみが内部クラスであり，その他は外部クラスとなっている．たとえばクラス A のインスタンスに対してメソッド `foo()` が呼び出される場合を考えると，変更される可能性のある，内部クラスと束縛されるメソッドの組合せは，次の 2 種類であることが分かる：(1) クラス F のインスタンスに対して，メソッド `F.foo()` が呼び出される，(2) クラス G のインスタンスに対して，メソッド `A.foo()` が呼び出される．この場合，クラス B，D，E は解析から省略することができる．ここでクラス G は内部クラスであるが，そのメソッド `foo()` は内部メソッドではなく，クラス A で定義された外部メソッドであることに注意したい．

また内部クラスで実装しているインタフェースについても，同様の解析を行うことができる．

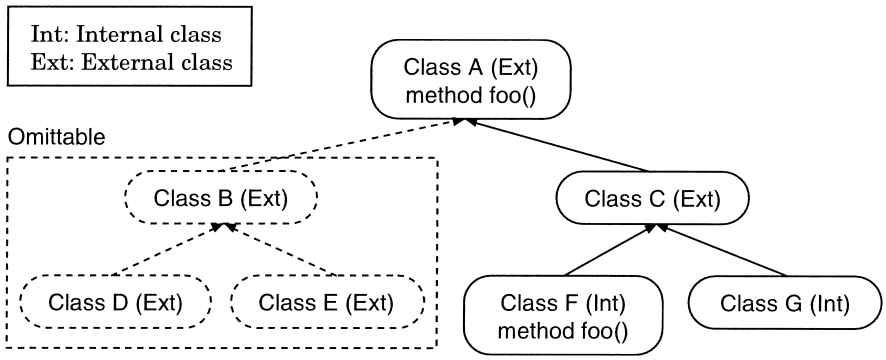


図 3 クラス階層木の例
Fig. 3 An example of a class hierarchy tree.

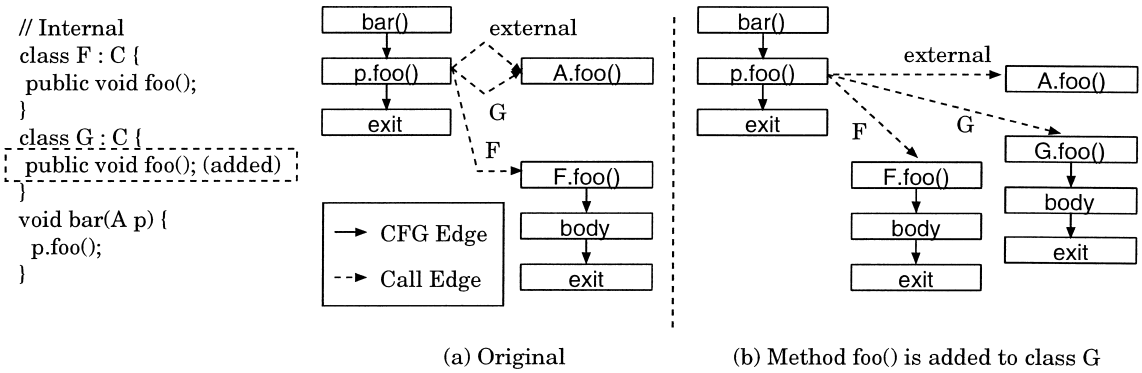


図 4 内部コードからのメソッド呼び出しの例
Fig. 4 Examples of method calls from internal code.

3.3.2 内部コードからのメソッド呼び出し

メソッド呼び出しで利用されるクラスは，内部クラスの場合と外部クラスの場合がある．内部クラスに対しては，文献 5) と同様に，クラス名をラベル付けした Call Edge を作成する．3.3.1 項で述べたとおり，呼び出し先のメソッドは，内部メソッドの場合と外部メソッドの場合がある．内部メソッドの場合は，Call Edge の行き先をその CFG とする．また外部メソッドの場合は，Call Edge の行き先をそれを表すノードとする．

外部クラスに対しては，3.3.1 項で述べたクラス階層木からは，完全な情報を取得することができない．これは内部クラスの先祖でない外部クラスを，階層木から省略したためである．しかし 3.3.1 項で述べたとおり，外部クラスに対して束縛されるメソッドは変わることがない．つまり個々の外部クラスに対して Call Edge を作成したとしても，それらは決して Dangerous Edge にはならない．そこでメソッド呼び出しで利用される可能性のあるすべての外部クラスをまとめ，*external* とラベル付けした Call Edge を作成する．そして Call Edge の行き先を，（一連の）外部メソッドを表す特殊なノードとする．このようにしても，内部クラスに対する Call Edge はすべて存在しているので，内部コードからのメソッド呼び出しに関連した Dangerous Edge はすべて正しく検出できる．

図 4 (a) に内部コードからのメソッド呼び出しの例を示す．これはクラス G に，メソッド `foo()` を追加する前のものである☆．またクラス階層木は，図 3 と同一のものとする．変数 `p` は，図 3 の任意のクラスのインスタンスになりえる．そのためノード “p.foo()” は，内部クラス F，G とその他の外部クラスに対応する，3 本の Call Edge（ラベルはそれぞれ F，G，external）を持っている．

図 4 (b) に，クラス G にメソッド `foo()` を追加し，クラス A の定義をオーバライドした例を示す．2 章で述べたとおり，Dangerous Edge は変更前後の CFG を比較することにより検出するので，図 4 の場合，ノー

☆ 本研究の CFG は MSIL を対象としているが，本論文では簡単のため元のソースコードを示す．

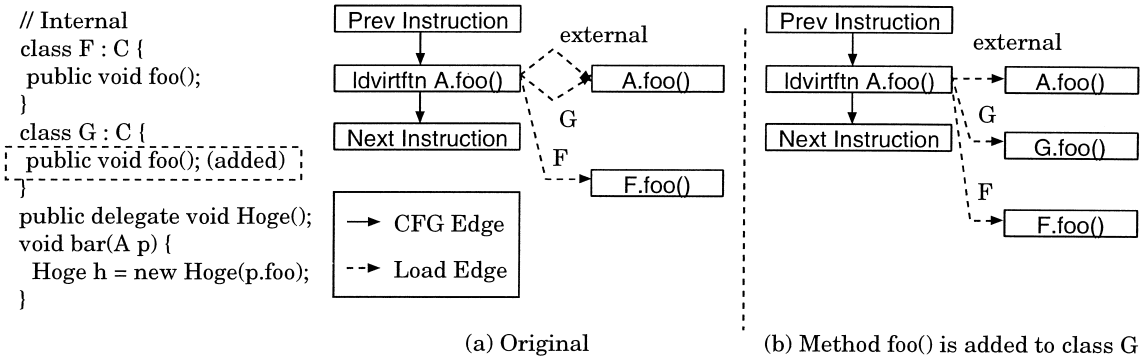


図 5 ldvirtftn 命令の例
Fig. 5 Examples of the instructions ldvirtftn.

ド “p.foo()” の持つ，G というラベルの Call Edge が Dangerous Edge となる．

3.3.3 外部コードからのメソッド呼び出し

テストコードに関する仮定により，外部コードからの内部クラスのメソッドに対する呼び出しを，動的束縛かコールバック関数の機能を利用したものに限定できる．動的束縛を利用した呼び出しには，2 章で述べた ECN を用いた手法を利用することができる．しかしその手法では，コールバック関数の機能を利用した呼び出しに対処できない．そこで本項では，コールバック関数の機能，特にデリゲートの扱いについて述べる．

3.3.3.1 デリゲートとは

デリゲートとは，Microsoft .Net Framework において，コールバック関数の機能を提供するメカニズムである．これは多くのオブジェクト指向言語で利用されている関数ポインタに似ているが，最大の違いはそれがタイプセーフであるということである．タイプセーフなコードとは，適切に定義された，許可されている方法でだけ，型にアクセスするコードのことである．デリゲートを利用した呼び出しでは，動的束縛を利用した呼び出しとは異なり，外部メソッドをオーバライドしていない内部メソッドでも，外部コードから呼び出すことができる．

Microsoft .Net Framework の仮想マシンは，デリゲートをサポートするために，3 種類の命令を定義している：`ldftn`, `ldvirtftn`, `calli`. `ldftn` と `ldvirtftn` 命令は，メソッドへの参照を取得するために用いる．`calli` 命令は，メソッドの間接呼び出しを行うために用いる．しかし `calli` 命令は，仕様上の制約によりタイプセーフではない．そのため C#などのコンパイラでは，`calli` 命令を直接生成する代わりに，仮想マシンが実装するメソッドを通じて間接呼び出しを行う．

3.3.3.2 デリゲートの扱い

デリゲートによって呼び出されるメソッドを静的に決定し，その変更を検出するためには，外部コードも含めた詳細なデータフロー解析を行う必要がある．しかし実行に関する仮定により，すべての `ldftn` と `ldvirtftn` 命令によって取得されるメソッドへの参照が変わらず，またプログラムの他の部分でも，変更されていない部分のみを実行していれば，変更前と同じメソッドが呼び出されることを保証できる．そのためここでは，`ldftn` と `ldvirtftn` 命令によって取得されるメソッドへの参照の変更を，Dangerous Edge として正しく検出する方法について述べる．

内部コードに関しては，これらの命令に対して，直接 CFG を作成することができる．まず *Load Edge* という特殊な枝を定義する．内部クラスに対しては，クラス名をラベル付けした Load Edge を作成する．そして Load Edge の行き先を，参照を取得されるメソッドを表すノードとする．また外部クラスに対しては，それらをまとめ，*external* とラベル付けした Load Edge を作成する．そして Load Edge の行き先を，(一連の) 外部メソッドを表す特殊なノードとする．3.3.2 項と同様に，内部クラスに対する Load Edge はすべて存在しているので，内部コードでのメソッドへの参照の取得に関連した Dangerous Edge は，すべて正しく検出できる．

図 5 (a) に `ldvirtftn` 命令の例を示す．ここでは簡単のため，CFG から他の MSIL 命令を省略している．またクラス階層木は，図 3 と同一のものとする．ノード “ldvirtftn A.foo()” では，変数 `p` からメソッド `foo()` への参照を取得する．変数 `p` は，図 3 の任意のクラスのインスタンスになりうる．そのためノード “ldvirtftn A.foo()” は，内部クラス F，G とその他の外部クラスに対応する，3 本の Load Edge（ラベ

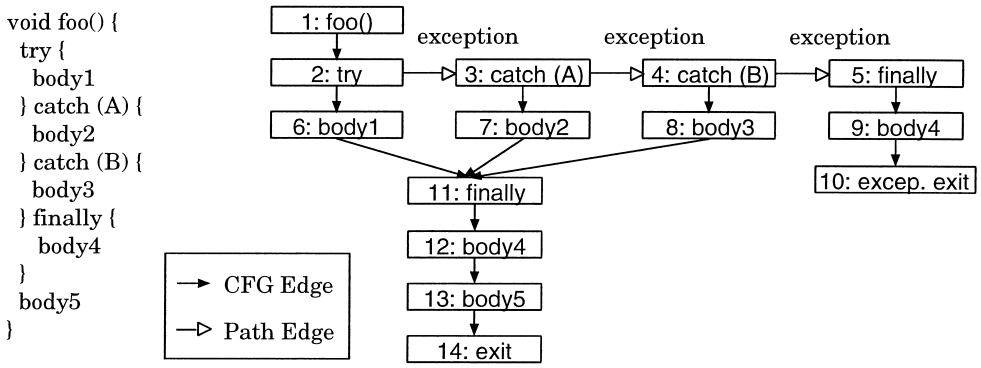


図 6 例外処理の例
Fig. 6 An example of exception handling.

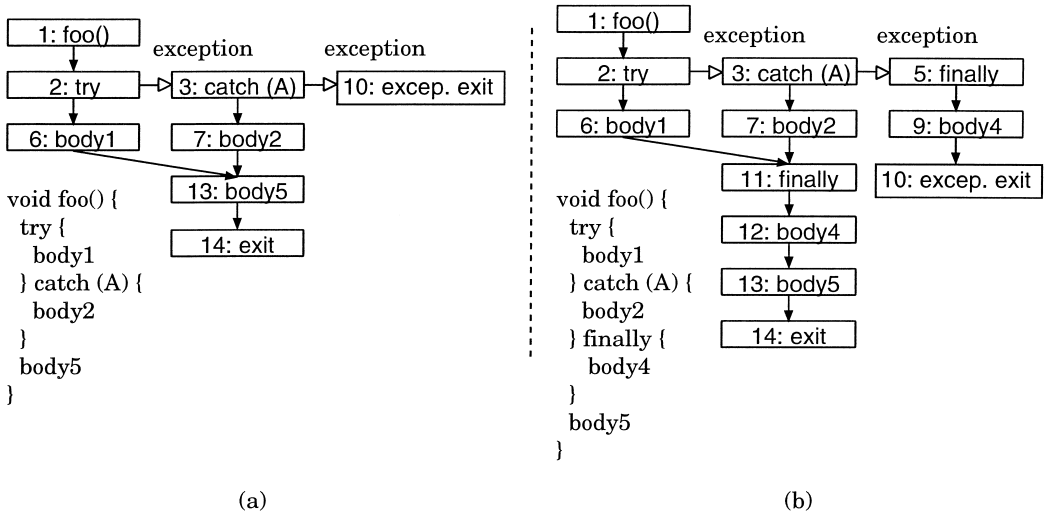


図 7 図 6 の変更例
Fig. 7 Examples of modifications to Fig. 6.

ルはそれぞれ F，G，external）を持っている．

図 5 (b) に，クラス G にメソッド `foo()` を追加した例を示す．図 5 の 2 つのグラフを比較することにより，ノード “ldvirtftn A.foo()” の持つ，G というラベルの Load Edge が Dangerous Edge となる．

一方，外部コードに関しては，同様の解析を行うことができない．そこで外部コードで取得される，デリゲートとして呼び出されていないメソッドへの参照は，変わっても実行結果に影響を与えないものと見なす．すると外部コードで取得される，デリゲートとして呼び出されているメソッドへの参照のみ，検査すればよいことになる．ここでテストコードについての仮定により，外部コードで取得される内部メソッドへの参照は，動的束縛を利用した呼び出しと同様に，外部メソッドをオーバライドしているものに限定できる．そのため ECN を用いた手法を利用することにより，これらの命令で取得される参照が変わる可能性を，Dangerous Edge として正しく検出できる．つまり実際に呼び出される内部メソッドを検査することによって，変更を検出するのである．

3.3.4 例外処理

例外処理を扱う際には，例外の処理中に再度例外が発生した場合の問題に，注意する必要がある．さもなければ，必要以上のテストケースを選択する可能性がある．

たとえば Harrold らは，例外の発生による *try*, *catch*, *finally* ブロック間の制御の遷移を，*Path Edge* という特殊な枝で表した[5]．まず各 try，catch，finally ブロックの CFG を作成し，それらの先頭ノード間を Path Edge で結ぶ．また finally ブロックがある場合は，finally ブロックを複製し，その先頭のノードに対して，各 try，catch ブロックの末尾のノードから CFG

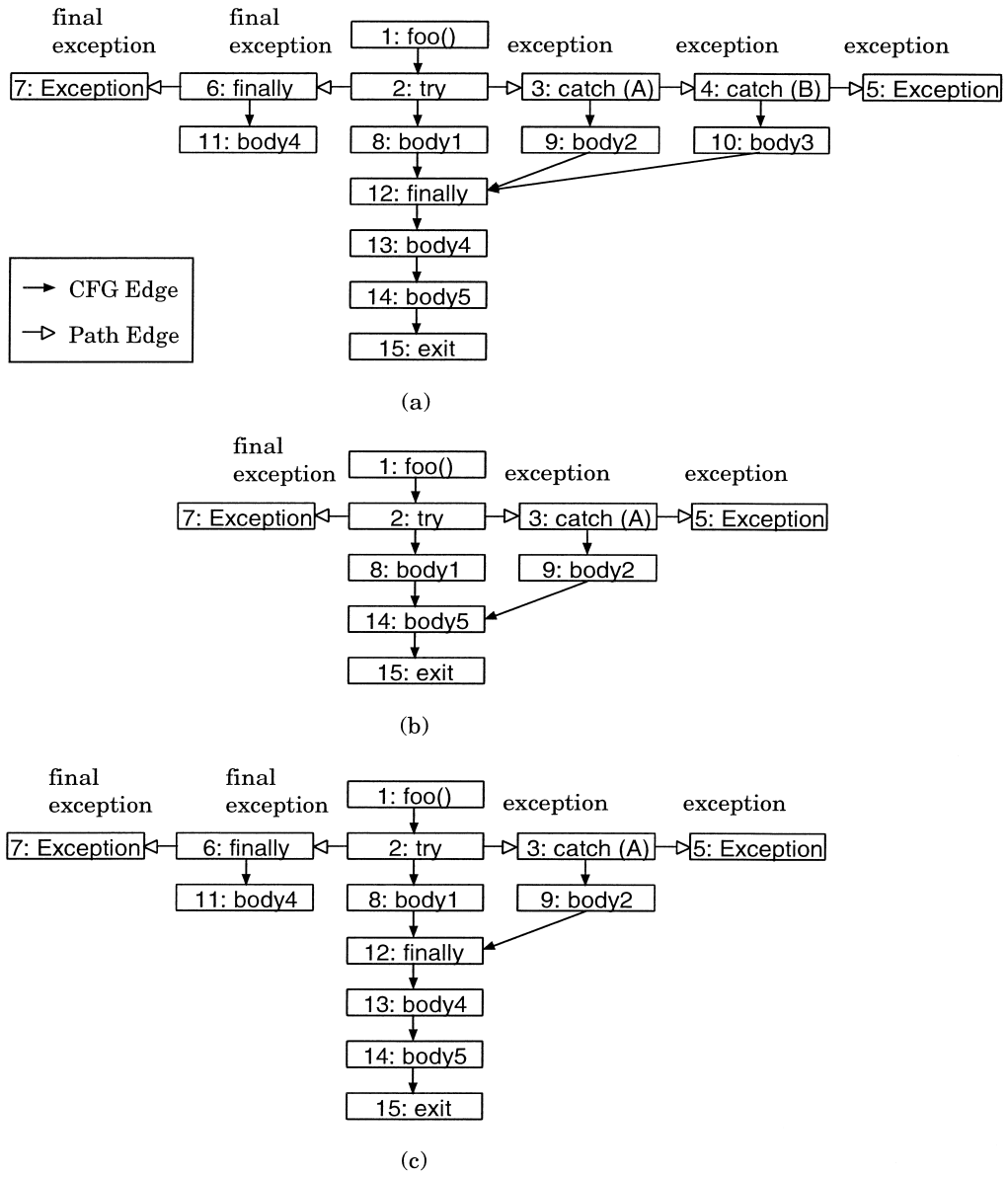


図 8 提案手法での例外処理の例
Fig. 8 Examples of exception handling according to our approach.

Edge を結ぶ．**図 6** に例を示す．そこでは，ノード 6 で例外が発生したとすると，枝 (2, 3) の Path Edge を通って，ノード 3 に遷移するものと考える．

しかし，図 6 のコードと JIG に対して，**図 7** (a), (b) のような，2 種類の変更を行ったとする．図 7 (a) では最後の catch ブロックと finally ブロックの削除を行い，(b) では最後の catch ブロックのみ削除を行っている．どちらの場合も，図 6 の枝 (3, 4) が Dangerous Edge として検出される．ここでノード 7 で例外が発生するか，捕捉した例外を再投入するような，ノード 7 に対するテストケースを考えてみる．この実行のパスは，(1, 2)，(2, 6)，(2, 3)，(3, 7)，(3, 4)，(4, 5)，(5, 9)，(9, 10) となる．図 7 (a)，(b) のどちらの変更に対しても，枝 (3, 4) が Dangerous Edge なので，このテストケースは選択される．しかし実際には，(b) では再テストする必要がない．なぜなら，ノード 7 はノード 2 の try ブロックの外側なので，そこで発生した例外は，ノード 4 の catch ブロックでは決して処理されないためである．これは，このような例外の伝播を，JIG が正確に表現していないことが原因である．

そこで提案手法では，try ノードの両側に，catch ブロックと finally ブロックを分けて配置することによって，例外処理を表現することにする．さらに変更点を正しく検出するために，処理されなかった例外を表す，“Exception” とラベル付けした特殊なノードを 2 つ作成する．そして最後の catch ブロックと finally ブロック（なければ try ブロック）を，Exception ノードと Path Edge で結ぶ．

図 8 に提案手法に従って書き直した CFG を示す．図 8 (a)，(b)，(c) が，それぞれ図 6，図 7 (a)，(b) に対応している．図 8 (a) と (b) を比べると，枝 (3, 4) と (2, 6) が Dangerous Edge となる．しかし図 8 (a) と (c) を比べると，枝 (3, 4) のみが Dangerous Edge となる．ここでノード 9 で例外が発生するか，捕捉した例外を再投入するような，ノード 9 に対するテストケースを考える（これは図 6 のノード 7 にあたる）．この実行のパスは，(1, 2)，(2, 8)，(2, 3)，(3, 9)，(2, 6)，(6, 11)，(6, 7) となる．そのため図 8 (b) ではテストケースが選択されるが，(c) では選択されない．

提案手法が Harrold らの手法と異なるのは，catch ブロックを最後まで走査したが，例外を捕捉できなかった場合と，catch ブロックで例外を捕捉したが，その catch ブロック中で再度例外が発生した場合の処理である．どちらの場合でも，実際のプログラムでは，finally ブロックがあればその finally ブロックに遷移し，なければ例外を外側に伝播する．前者の場合には，最後の catch ブロックの次に新たに catch ブロックを作成するか，finally ブロックを作成・削除することで，例外による遷移を変更できる．また後者の場合には，finally ブロックを作成・削除することで，例外による遷移を変更できる．

提案手法では，前者の場合の，最後の catch ブロックの次に新たに catch ブロックを作成するという変更を，catch ブロック側の Exception ノードへの Path Edge において，Dangerous Edge として正しく検出することができる．たとえば図 8 (a) では，枝 (4, 5) がこれにあたる．またどちらの場合でも，finally ブロックを作成・削除するという変更は，try ノードから出ている finally ブロック側への Path Edge において，Dangerous Edge として正しく検出することができる．たとえば図 8 (a) では，枝 (2, 6) がこれにあたる．

3.4 Dangerous Edge の検出

Dangerous Edge は，従来の手法[5),7)] と同じように，変更前後の CFG を深さ優先順に比較していくことで検出する．しかし MSIL のコードは辞書的に比較できないため，比較方法については変更が必要となる．

MSIL の命令の比較方法は，そのオペランドの種類によって決定される．大半の命令は次の 2 種類なので，容易に比較することができる：

(1) オペランドを持たないか，オペランドを数値的に比較できるもの．
(2) オペランドからシンボルなどの情報を取得して，それらを比較できるもの．

ただしローカル変数にアクセスする命令と分岐命令は，注意して扱わなければならない．

MSIL では，メソッドのローカル変数は，インデックスによって識別される．そして MSIL でのローカル変数のインデックスとソースコードでのローカル変数は，1 対 1 で対応しているわけではない．またそのインデックスは，ソースコードの変更にともない，頻繁に変更される．そのため変更前と変更後のプログラムのローカル変数のインデックスどうしを，矛盾なく対応付ける必要がある．

そこで CFG を比較する際に，変更前と変更後のプログラムの同じ命令で，同じ型のローカル変数にアクセスを行っていたら，それらのインデックスどうしを対応付けることにする．このとき，それらがすでに別のローカル変数のインデックスに対応付けられているなど，何らかの矛盾が発生したら，変更が行われたものと判断する．

分岐命令と，`ldftn`, `ldvirtftn` 命令については，そのオペランドを比較する必要がない．これは CFG に，それらの情報が含まれているためである．

4. 評　　価

3 章で述べた提案手法に基づいて，回帰テスト選択システムを実装した．本章では，回帰テストのコスト削減における，提案手法の有効性に対する評価について述べる．

4.1 対　　象

実験対象には，Mono Project[☆] が開発している.Net Framework クラスライブラリを利用した．Mono Project は，Microsoft 版と互換性のある，Unix 版の.Net Framework の開発を目標としており，その成果はオープンソースとして公開されている．またテストケースには，Microsoft が自社のクラスライブラリとともに配布しているものを利用した[☆☆]．実験では，

☆ http://www.go-mono.com
☆☆ http://msdn.microsoft.com/downloads/default.asp?url=/downloads/sample.asp?url=/MSDN-FILES/027/002/097/msdncompositedoc.xml

表 1 実験対象の概要
Table 1 Summary of software subjects.

クラス	バージョン	LOC	LOC（中間言語）	メソッド数	テストケース数
ArrayList	1.29 - 1.38	2399	5185	140	51
BitArray	1.1 - 1.8	500	1549	36	24
Hashtable	1.6 - 1.15	1047	3075	92	15
Queue	1.3 - 1.11	350	1325	37	11
SortedList	1.1 - 1.10	1134	3475	118	29
Math	1.4 - 1.13	375	1705	61	54
BinaryWriter	1.1 - 1.7	207	958	29	22
MemoryStream	1.1 - 1.9	403	1275	27	27
StreamWriter	1.10 - 1.19	219	772	21	8
TimeSpan	1.1 - 1.8	551	1760	55	35

表 2 評価結果
Table 2 Evaluation results.

クラス	全 TC の実行時間（msec）	解析時間（msec）	選択 TC（%）	選択 TC の実行時間（msec）	節約時間（%）
ArrayList	3773	2308	31.2	1193	7.2
BitArray	1941	384	32.7	626	48.0
Hashtable	1358	517	42.2	599	17.8
Queue	906	248	30.7	271	42.8
SortedList	2340	963	25.3	576	34.2
Math	3649	1377	5.3	190	57.1
BinaryWriter	2283	510	40.9	975	35.0
MemoryStream	2344	520	33.3	745	46.0
StreamWriter	874	116	56.9	507	28.7
TimeSpan	4863	706	24.5	873	67.5
平均	2411	795	26.6	643	40.4

TC はテストケースを指す.

システムに合わせてテストドライバを用意した.

実験対象は，すべて DLL ファイルである．そしてテストケースは，それらを利用する実行ファイルである．それぞれのテストケースは内部で様々な検査を行い，その結果を自動的に判断し出力する．これらのテストケースはすべて C#で記述されており，その総 LOC は約 50,000 行であった．またコンパイル済みのテストケースを逆アセンブルした結果，それらの中間言語での総 LOC は約 194,000 行となった．**表 1** に実験対象の概要を示す．なお表 1 の LOC，LOC（中間言語），メソッド数は，表 1 の最新バージョンでの数値を示している．たとえば ArrayList の LOC，LOC（中間言語），メソッド数は，バージョン 1.38 のものである．

4.2 結果と考察

表 2 に評価結果を示す．それぞれの数値は，各実験対象において，そのすべてのバージョンに対する評価の平均である．たとえば BitArray クラスでは，テストケースの選択と実行を 7 回行った．それぞれは，バージョン 1.2，1.3，1.4，1.5，1.6，1.7，1.8 への変更に対するものである．実験に用いたコンピュータの構成は，次のとおりである：CPU Athlon XP 1900+, Main Memory 512 MB，OS Windows XP SP1.

提案手法は，次の 7 段階に分けられる：

(1) 変更前のプログラムの CFG の作成.
(2) 変更前のプログラムの書き換え.
(3) すべてのテストケースのカバレッジ情報の取得.
(4) 変更後のプログラムの CFG の作成.
(5) Dangerous Edge の検出.
(6) テストケースの選択.
(7) 選択したテストケースの実行.

評価では，提案手法による回帰テストのコストとして，(4)，(5)，(6)，(7) の処理時間の合計を用いる．(1)，(2)，(3) も，プログラムに変更を加えるたびに必要な処理である．しかしすべて変更前のプログラムに対する処理なので，プログラムに変更を行っている間に自動的に実行可能であり，実際に回帰テストを行う段階ではすでに終了しているものと仮定した．そのため，提案手法による回帰テストのコストからは省略している．表 2 で，“解析時間” と “選択 TC の実行時間” の合計が，提案手法による回帰テストのコストにあたる．“解析時間” は (4)，(5)，(6) の処理時間の合計であり，“選択 TC の実行時間” は (7) の処理時間である．また “節約時間” は，提案手法を適用することで，すべてのテストケースを実行する場合と

比べて節約できた時間の割合である．

表 2 より，提案手法を適用することで，再実行が必要なテストケースの数を，テストスィート全体の 26.6%にまで削減できることが分かる．また変更前と変更後のプログラムにおいて，選択されなかったテストケースを実際に実行し，その実行結果（出力）を手動で比較してみた．その結果，それらのテストケースの実行結果は変更前と変更後で変わっておらず，確かに再実行する必要はないことが分かった．

しかし実験対象によって，選択されたテストケースの割合に大きなばらつきが見られた．特に変更が大きかった場合や，コンストラクタなど頻繁に使われるメソッドに対して変更が行われた場合に，より多くのテストケースが選択される傾向が見られた．これは文献 5) でも見られた傾向である．逆に Math のように，静的メソッドしか持たないクラスでは，より少ないテストケースが選択された．

すべての実験は，最適化を有効にした状態と，無効にした状態の両方で行った．しかし選択されたテストケースの割合に関しては，その影響は特に見られなかった．

また表 2 より，回帰テストを行うのに必要な時間は，解析時間を考慮に入れても，すべてのテストケースを実行する場合と比べて，40.4%も節約できることが分かる．今回の実験対象は DLL ファイルであり，テストケースはそれらを利用する実行ファイルであった．そのため解析時間には，実験対象だけでなく，テストケースの解析にかかる時間も含まれている．表 1 と表 2 より，解析時間が実験対象の LOC だけでなく，テストケース数の影響も受けていることが分かる．通常，テスト対象は実行ファイルであり，テストケースはその入力であるため，テストケースを解析する必要はない．そのような場合には，テストケース数が多くなるにつれて，提案手法の効果もさらに増していくものと考えられる．

本実験では，実験に利用したテストケースのコスト自体がそれほど高くなかったために，実際に節約できた時間は合計で 10 秒程度であった．本実験で利用したテストケースは，全部で 276 個であったが，どれも 1 つのクラスだけを対象とした単体テストを行うものであった．しかし現実には，単体テストだけではなく，統合テストやシステムテストの段階でも回帰テストは必要となる．たとえば統合テストの段階における回帰テストでは，統合が進むにつれてテストケース数が非常に多くなり，すべてのテストケースを実行することは困難となる場合がある[6]．このような場合には，提案手法を利用することで，回帰テストの効果をまったく損なうことなく，大幅なコスト削減を達成できるものと考えられる．またテストケースのコストがそれほど高くない場合でも，提案手法を利用することで，回帰テストをより頻繁に実行できるようになるという利点がある．たとえば，提案手法を開発環境に統合し，変更をほんの少し加えるたびに回帰テストを実行する，という使い方も可能である．

5. 結　　論

本論文では，仮想マシンの中間言語に基づく回帰テスト選択手法を，Microsoft .Net Framework を例に提案した．従来の Harrold らによる手法などは，Microsoft .Net Framework に対しては，CFG の作成，および Dangerous Edge の検出に MSIL 特有の課題が存在するため，そのまま適用することができない．また本論文では，提案手法に基づいた回帰テスト選択システムの実装と，その評価についても述べた．そこでは再実行が必要なテストケースの数を，テストスィート全体の 26.6%にまで減らすことができた．また回帰テストを行うために必要な時間は，解析時間を考慮に入れても，40.4%も節約することができた．これらの結果は，回帰テストのコスト削減において，提案手法が非常に有効であることを示している．

今後の課題としては，解析の効率化や，非決定的な振舞いをするソフトウェア，特にスレッドを利用したソフトウェアへの対応などがあげられる．

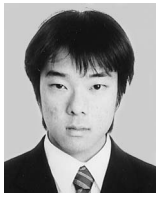
謝辞 本研究は，文部科学省科学技術振興調整費「環境情報獲得のための高信頼性ソフトウェアに関する研究」の支援による．

参 考 文 献

1) Ball, T.: On the Limit of Control Flow Analysis for Regression Test Selection, *Proc. ISSTA 1998*, pp.134–142 (1998).
2) Chen, Y.F., Rosenblum, D. and Vo, K.P.: Testtube: A System for Selective Regression Testing, *Proc. ICSE 16*, pp.211–222 (1994).
3) Dean, J., Grove, D. and Chambers, C.: Optimizations of Object-Oriented Programs Using Static Class Hierarchy Analysis, *Proc. ECOOP '95*, pp.77–101 (1995).
4) Graves, T.L., Harrold, M.J., Kim, J., Porter, A. and Rothermel, G.: An Empirical Study of Regression Test Selection Techniques, *ACM TOSEM*, Vol.10, No.2, pp.184–208 (2001).
5) Harrold, M.J., Jones, J.T., Li, D.L., Orso, A., Pennings, M., Sinha, S., Spoon, S. and

Gujarathi, A.: Regression Test Selection for Java Software, *Proc. OOPSLA 2001*, pp.312–326 (2001).
6) Pressman, R.S.: *Software Engineering: A Practioner's Approach, Fourth Edition*, McGraw-Hill (1997).
7) Rothermel, G. and Harrold, M.J.: A Safe, Efficient Regression Test Selection Technique, *ACM TOSEM*, Vol.6, No.2, pp.173–210 (1997).
8) Rothermel, G. and Harrold, M.J.: Empirical Studies of A Safe Regression Test Selection Technique, *IEEE Trans. Softw. Eng.*, Vol.24, No.6, pp.401–419 (1998).
9) Vokolos, F.I. and Frankl, P.G.: Pythia: A Regression Test Selection Tool Based on Textual Differencing, *Proc. International Conference on Reliability, Quality, and Safety of Software Intensive Systems* (1997).
10) Vokolos, F.I. and Frankl, P.G.: Empirical Evaluation of the Textual Differencing Regression Testing Technique, *Proc. ICSM 1998*, pp.44–53 (1998).

(平成 15 年 12 月 8 日受付)
(平成 16 年 6 月 8 日採録)



孝壽 俊彦

2003 年慶應義塾大学理工学部卒業．同年同大学大学院理工学研究科修士課程入学，現在に至る．ソフトウェア工学に関する研究に従事．

高田 眞吾（正会員）

1990 年慶應義塾大学理工学部卒業．1992 年同大学大学院理工学研究科修士課程修了．1995 年同博士課程修了．博士（工学）．同年奈良先端科学技術大学院大学情報科学研究科助手．1999 年より慶應義塾大学理工学部情報工学科専任講師．ソフトウェア工学，情報検索等の研究に従事．電子情報通信学会，日本ソフトウェア科学会，ACM，IEEE CS 各会員．

土居 範久（正会員）

1969 年慶應義塾大学大学院博士課程単位取得退学．慶應義塾大学理工学部教授を経て，2003 年より中央大学理工学部教授，慶應義塾大学名誉教授．工学博士．現在，文部科学省科学技術・学術審議会委員，総務省情報通信審議会委員，世界科学会議（International Council for Science：ICSU）Priority Area Assessment Panel of Scientific Data and Information メンバー，科学技術振興機構（JST）社会技術システムミッションプログラム II「情報セキュリティ」研究統括，特定非営利活動法人日本セキュリティ監査協会会長，国際計算機学会（ACM）日本支部長，等．専門はソフトウェアを中心とした計算機科学．