ソフトウェア工学 96−16
（1994．1．21）

オブジェクト指向プログラムのテスト法に関する一考察

古川善吾＊　梅田修一＊＊　片山徹郎＊＊　伊東栄典＊＊　牛島和夫＊＊

＊ 九州大学 情報処理教育センター
＊＊ 九州大学 工学部 情報工学科

オブジェクト指向言語で記述されたプログラム（オブジェクト指向プログラム）のテスト法を提案する。オブジェクト指向プログラムのテスト基準として、(1) 従来の手続き型言語で用いられている制御フローテスト基準、(2) オブジェクトの操作のための関数や上位のオブジェクトから継承した操作のための関数を含めた導出操作類関数をすべて少なくとも１回実行することを要求する導出操作類関数テスト基準、(3) 導出操作類関数の呼び出しと導出操作類関数の宣言との対をすべて少なくとも１回実行することを要求する導出操作類関数呼び出しテスト基準、を提案する。これらを実用化するためには、オブジェクト指向プログラムの普及に効果があったと同様に支援ツールが必要であり、プログラムの解析およびテスト実行時の計測が可能であることを示す。

Testing Criteria for Programs written in an Object Orientied Programming Language

Zengo FURUKAWA*, Shuichi UMEDA**, Tetsuro KATAYAMA**,
Eisuke ITOU** and Kazuo USHIJIMA**

* Educational Center for Information Processing, Kyushu University.
** Department of Computer Science and Communication Engineering,
Kyushu University.

6-10-1 Hakozaki, Fukuoka 812, Japan.

This paper proposes and discusses testing criteria for programs written in an object oriented programming language (object oriented programs). Three testing criteria are specified for the programs. The first is the path testing criterion on a control flow graph for each object. The second is the method testing criterion which requires all methods in a program are executed at least once in testing. The last is the call testing criterion which requires all pairs of callings and declarations for a method are executed at least once. These criteria are based on a source code of an object oriented program. A supporting tool is necessary for the testing criteria as development environment for object oriented programs.

1. はじめに

1980 年代を通してオブジェクト指向パラダイムが浸透し、プログラミング言語だけでなく、ソフトウェア開発における現実世界の分析やソフトウェアの設計法としてオブジェクト指向分析 / 設計が利用されている。オブジェクト指向パラダイムは、柔軟性に富み、支援ツールの充実によってプログラムの開発が容易になることが期待できる。しかしながら、実用的なソフトウェアをオブジェクト指向言語で記述する場合には、開発工程のすべてに渡る支援ツールの必要性が認識されてきている[15]。また、オブジェクト指向言語で記述されたプログラム (オブジェクト指向プログラム) の信頼性についてもこれまで充分に議論されていない。そこで、実用的なソフトウェアをオブジェクト指向言語で開発するには、ソフトウェアの信頼性を確保するためのテストが必要である。本論文では、オブジェクト指向プログラムのテスト法について検討する。ここで検討する結果は、オブジェクト指向プログラムだけでなく、オブジェクト指向分析や設計で得られた仕様の信頼性向上にも役立てることができる。

ソフトウェアのテストは、実際に入力データを与えて被テストプログラムを実行し、誤りがあれば発見し、誤りがなければプログラムの正しさに対する確信を増大させるためのものである。プログラムの正しさを、テストによって証明することはできない。そのために、テストを実施したときにプログラムをどの程度テストしたかを被覆率として計測し、テスト充分性を評価するテスト充分性評価技法が用いられている。これまでに提案されている、手続き型言語で記述したプログラムのテスト充分性評価技法は必ずしも充分なものとは言い難いけれども、テストの程度を定量的に表すことができるので、実際の開発現場で用いられている。

そこで、オブジェクト指向プログラムについてもこのテスト充分性評価技法を開発する。まず、対象となるオブジェクト指向プログラムの記述言語としては、Smalltalk-80 [17] や C++ [16] などがある。今回は、C++ をとりあげる。C++ は、既存の C 言語にオブジェクト指向のための機能を追加した言語である。そのために、既存言語からの制約を受けているという欠点がある。一方、C 言語で蓄積された資産を受け継ぐことができるという利点がある。そのために、開発現場での使用が多くなることが期待され、テストが必要となる度合も高くなる。

本論文では、第 2 章でテストとテスト充分性評価技法について簡単にまとめた後、第 3 章でオブジェクト指向プログラムの特徴述べ、第 4 章でオブジェクト指向プログラムのモデル化を行い、第 5 章でオブジェクト指向プログラムのテスト充分性評価技法の提案と実現可能性や信頼性、支援ツールについて議論し、第 7 章でまとめる。

2. ソフトウェアテスト法

ソフトウェアのテストの枠組みを Goulay がテストシステムとして提案した[7]。テストシステムでは、テストで使用したテストデータの集合 T によって仕様に対するプログラムの正当性を評価する述語 *corr* が用いられる。この *corr* 述語を被覆率で定義するのがテスト充分性評価技法である。テスト充分性評価技法において被覆率は、プログラムの正当性を定量的に表す指標として用いられる。

事前にソースコードから抽出した測定対象と、テストデータを与えて被テストプログラムを実行するテスト実施において実行された測定対象との割合、として被覆率を定義する。この被覆率を用いたテスト充分性評価技法を開発するためには、一般に以下のことが必要である。

i) プログラム言語や仕様記述言語の解釈モデルを開発する。解釈モデルに従ってプログラムあるいは仕様から抽出したモデル (グラフと呼ぶ) は、被覆率を計算するためにプログラムを実行する前に確定していなければならない、すなわち、静的に決定可能でなければならない。また、解釈モデルは、それに従ってプログラムのテスト充分性を評価するので、プログラムあるいは仕様の特徴を表していなければならない。例えば、逐次処理プログラムでは実行文を節点とし、制御の移行を枝とする制御フローモデルを解釈モデルとして用いている。一方、Ada 並行処理プログラムでは、逐次的に実行されるタスクと、タスクの間の同期や通信を行うランデブーあるいは共有変数のデータフローとでモデル化している[5], [6]。

ii) 解釈モデルにおいてテスト時の被覆率を測定するための測定対象を定義する。測定対象は、グラフから決定可能で、プログラムを実行したときに、測定対象が実行されたことが計測可能でなければならない。例えば、逐次処理プログラムでは制御フローグラフの節点 (C_0 被覆率) や枝 (C_1 被覆率) を測定対象としている。また、Ada 並行処理プログラムでは、エントリ呼び出しとアクセプト文の対 (ランデブー通路被覆率) や共有変数のデータフロー (共有変数データフロー被覆率) を測定対象としている。

iii) 被覆率が 100 % になったとき (これを達成することをテスト基準と呼ぶ) の信頼性 (テスト基準が満

たされたときに、存在するならば必ず発見できる不良) を明確にする。これは、開発したテスト充分性評価技法の有効性を定性的に示すものである。C_0 テスト基準は、用いたデータによっては計算不良が存在するならば発見できるという意味で部分的に信頼できる。C_1 テスト基準は領域不良や路選択不良に対して部分的に信頼できる [8]。また、ランデブー通路テスト基準や共有変数テスト基準は通信不良に対して部分的に信頼できる [5], [6]。

iv) テスト基準が保証する信頼性が有効であるか否かを実際のプログラム開発現場で実証する。項番 iii) では、テスト基準の有効性を定性的に検討する必要があることを述べた。しかしながら、テスト基準が有効であることを示すためには、実際のソフトウェア開発現場での不良の発生状況や発見状況に基づいて定量的に有効性を確認する必要がある。C_0 や C_1 テスト基準については、Howden が文献上のプログラムにおいて約 60 % の不良を発見できることを示した [9]。Ada 並行処理プログラムについては、まだ、定量的な評価は行なわれていない [5], [6]。

3. オブジェクト指向プログラムの特徴

オブジェクト指向言語で記述されたプログラム (オブジェクト指向プログラム) は、開発・実行環境を備えたプログラミング言語 Smalltlk-80 の普及に応じて広まってきた [17]。その特徴は、自律的に動作するオブジェクトとオブジェクトの間でやり取りされるメッセージである。オブジェクトがメッセージを交換しながら協調的に問題を解決していく。これは、現実世界の反映としても優れたモデル化であり、オブジェクト指向分析や設計技法が確立されてきている [1]。また、C++ のように既存プログラミング言語の拡張においてオブジェクト指向の考え方を取り入れた言語がある [16], [3]。オブジェクト指向プログラムは、以下のような基本概念を持っている [2]。C++ を例に取りながら説明する。

i) データ抽象化機能

データの抽象化は、データだけでなくデータに対する操作 (Method) を一体化した自律的な単位として定義することによって行われる。この自律的な単位がオブジェクトである。オブジェクトはソースコードとして記述されるクラスと、実際に動作する実体 (Instance) として構成される。C++ では、操作をクラスの操作関数 (Member) として定義する。

クラスは、情報隠蔽の単位であり、公開する (public) データや操作を宣言することができる。さらに、一部のオブジェクトに対してのみ公開するために、親密関数 (friend) や保護された操作関数 (protected) を宣言することができる。このような機能は、既存の Ada 言語などでも一部実現されている [19]。さらに、同一の関数名あるいは演算子名で引き数の型が異なる多重定義が可能である。

ii) クラスの階層化と継承機能

クラスの定義においてその上位のクラスを明示的に宣言できる。このクラスの上下関係は、クラスの階層構造を構成する。さらに、上位クラスの操作を下位クラスが継承することができる。このクラス階層と操作の継承によって、既存のクラスに機能を付加した新たなクラスを効率よく作成できる。C++ では、上位クラスを基底クラス、下位クラスを派生クラスと呼ぶ。派生クラスは 2 つ以上の基底クラスを持つことができる (多重継承)。さらに、基底クラスの操作関数が自動的に派生クラスの同じ名前の操作を呼び出す仮想関数の機能がある。

これまでの言語では、ソースコード上での包含関係で宣言の有効範囲を定めていたのに対して、明示的に宣言することによって柔軟な階層構造を持つことが可能になった。さらに、下位クラスを追加する場合に上位クラスを変更することなく、追加できるようになった。さらに、操作の継承によってプログラムの記述量を減らすことができる。一方、階層の明示的な宣言は、複雑な階層や多数の操作を持つことが可能になり、混入する不良が増大することが懸念され、開発環境による支援が必要となる。Smalltalk-80 では、言語の開発と同時に環境を開発することによってこの問題点を克服している。

iii) クラスからの実体 (Instance) 生成機能

型としてのクラスから実行するための実体を動的に生成する。この機能は、既存のプログラミング言語のいくつかにも見られる機能である。例えば、Ada は自律的に動作するタスクの実体をタスク型から動的に生成することができる。操作の実行はこの作成された実体の上で行われる。実体の生成時に実行するプロローグと削除時に行うエピローグをそれぞれ生成関数 (Constructor) と削除関数 (Destructor) としてクラスに宣言することができる。C++ では生成関数は、操作関数ではない。操作関数は実体に対して呼び出すことができるものであるので、実体を作成するために必要な生成関数を操作関数とすると矛盾が発生する。そこで **new** 演算子によって明示的に実体の生成を行う。同様に削除関数を呼び出すために **delete** 演算

子を用いる。また、操作関数の継承は、同じ名前で行われるが、削除関数については、派生クラスで定義されていなければ基底クラスの削除関数を呼び出す。クラスが宣言している操作関数や親密関数、生成関数、削除関数を一括して「操作類関数」と呼ぶことにする。

iv) 分散協調型計算モデル

オブジェクト指向プログラムでは、自律的な機能を持ったクラスの実体がメッセージと呼ばれる操作の呼び出しをやり取りしながら問題を解決する。C++ では、実体を指定した操作関数の呼び出しを行う。実体の実行を並行に行うための機構は直接言語仕様の中に含まれてはいない。操作類関数の定義として C 言語とオペレーティングシステムを用いて並行処理を実現する必要がある。最初に C++ を開発した AT & T の C++ では、コルーチンを実現するための task ライブラリが準備されている [3]。

C++ で記述したオブジェクト指向プログラムの例として、抽象木操作プログラムを用いる [3]。抽象木操作プログラムは、仮想関数を用いたプログラムである。

4. オブジェクト指向プログラムのモデル化

テストの対象となるオブジェクト指向プログラムの解釈モデルとして、これまでに状態遷移図や交信図という動作のモデル、オブジェクトの階層という静的なモデルが提案されている [15], [18]。これらのモデルは、オブジェクト指向概念に基づくプログラムや仕様の記述やデバッグ、プログラムの解釈を助けるためのものである。それらを基にテストのための解釈モデルとして以下のものを考える。

i) 制御フローモデル

制御フローモデルは従来のプログラムにおいて用いられているモデルであり、実行文を節点、制御の移行を枝とするモデルである。オブジェクト指向プログラムにおいても 1 つのオブジェクトの中の実行は逐次的に行われるので、オブジェクト内部の実行のモデルとして制御フローモデルを用いる。C++ プログラムでは、宣言を任意の場所に置くことができ、しかも、宣言と同時に初期設定を行うことが可能であるので、宣言文も節点とする (初期設定を行わない宣言についても記憶領域の確保を行うので節点とする)。

ii) 導出操作類関数モデル

オブジェクトの操作は、オブジェクトの中で宣言されているものだけでなく、上位クラスから継承した操作がある。1 つのオブジェクトのテスト対象としては、継承された操作も含める必要がある。特に、オブジェクト指向プログラムでよく用いられるライブラリの提供に当たっては、提供した操作のすべてをテストする必要がある。C++ では間接的な呼び出しも含めて呼び出されるのは操作類関数であるので、継承したものも含めた操作関数を「導出操作類関数」と呼ぶことにする。

例えば、抽象木操作プログラムのクラス階層 (図 1. 参照) に基づいて定義されている導出操作類関数の一覧は、表 1. に示すとおりである。

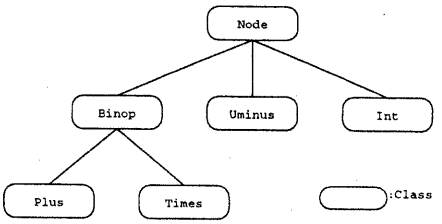

図 1: 抽象木操作プログラムのクラス階層

表 1: 抽象木操作プログラムの導出操作類関数

クラス	生成関数	削除関数	操作関数
Node	Node()	~Node()	eval()
Binop	Binop()	~Binop()	Node::eval()
Plus	Plus()	Binop::~Binop()	eval()
Times	Times()	Binop::~Binop()	eval()
Uminus	Uminus()	~Uminus()	eval()
Int	Int()	Node::~Node()	eval()

iii) 導出操作類関数呼び出しモデル

実体の動作に関しては、状態遷移図やオブジェクト交信図がオブジェクト指向プログラムのモデルとして提案されている [18]。しかしながら、テストのためには充分とは言い難い。そこで、まず実体階層を考える。実体階層とは、生成された実体を下位に、生成した実体を上位においた図である。C++ では、従来の C 言語の関数をそのまま用いることができる (主プログラム main() は必ず C の関数である) のでこの C の関数を静的実体 (Static Instance) と呼び、一方、クラスの実体として明示的に **new** 演算子で作成された実体を動的実体 (Dynamic Instance) と呼ぶ。抽象木操作プログラムの実体階層を図 2. に示す。ただし、動

的実体が複数個生成される場合 1 つのみ示した。
実体階層の作成手順は以下のとおりである。

(1) 静的実体が生成する実体は実体要素である。
(2) 実体要素が生成する実体は実体要素である。
(3) 静的実体が生成する実体の実体要素とその宣言を含んでいる実体とは生成関係がある。
(4) 動的に生成された実体の実体要素とそれを生成する実体要素とは生成関係がある。
(5) 生成関係にループがある場合、ループを 1 回繰り返した時点で実体要素の生成を停止する。

この実体階層は、プログラムの動作の一面を表したものである。この実体階層に基づいてプログラムは実行される。しかしながら、実体の実際の作成状況そのものではない。実際に作成される実体の数は、ソースコードに規定されている個数あるいは入力データに応じた個数である。そのために、実際に作成される実体個数ではなく、作成される可能性を代表して 1 つだけ示す。これによって、被覆率測定のための測定対象の個数が不定になることを防止する。一方、動的に生成される実体を 1 つだけに限定することには問題があり、その対策については片山らが検討している [13]。

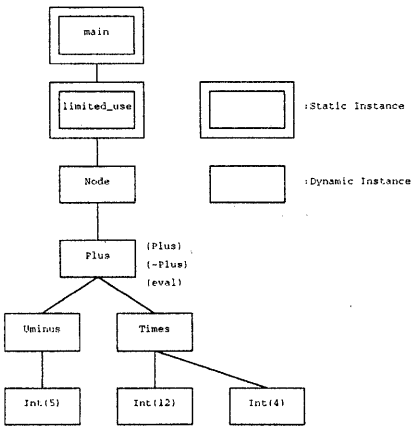

図 2: 抽象木操作プログラムの実体階層

この実体階層の上で導出操作類関数呼び出しテスト基準を定義する。すなわち、各実体からの導出操作類関数の呼び出しと導出操作類関数の宣言との対をテストの対象とする。実体毎の呼ばれる関数は導出操作類関数であり、呼び出しは実体階層全体の導出操作類関数となる可能性がある。これは、クラスポインタによって実体に関する情報が実体間で渡されることが可能であるためである。

5. オブジェクト指向プログラムのテスト充分性評価

5.1 テスト充分性評価技法

オブジェクト指向プログラムのテスト充分性評価技法としては 第 4 章で述べたオブジェクト指向プログラムのモデルに基づいて以下の 3 つを考える。

i) 制御フローテスト基準
制御フローグラフの節点や枝を測定対象として被覆率を計測する。逐次処理プログラムの場合と同様に C_0, C_1 被覆率と呼ぶ。ただし、クラスに対して制御フローグラフを作成し、計測は実体に対して行うので、計数を何に対して行うかを決める必要がある。ここでは、クラスのソースコードに累積する。クラスのソースコードに累積することによって、 (1) 節点や枝の数を静的に決定できる、 (2) 被覆率が 100 % にならない場合、被テストプログラムの追加の実行によって被覆率を増大させることができるという利点がある。一方、 (3) 実体の細かい情報を取得できない、という欠点がある。

ii) 導出操作類関数テスト基準
導出操作類関数を測定対象として被覆率を計測する。このテスト基準は、呼び出される可能性のある導出操作類関数を少なくとも 1 回呼び出すことを要求する。第 4 章で述べたように、 1 つのプログラムの中で使われるか否かに関わりなく、呼び出すことを要求する。オブジェクト指向プログラムでは、クラスの集合をライブラリとして登録しておき、ライブラリを用いてプログラムを構築したり、追加構築によってプログラムを開発する。その場合、信頼性を確保するためにこのテスト基準を用いる。

iii) 導出操作類関数呼び出しテスト基準
実体階層上で導出類関数の呼び出しと、導出操作類関数の宣言との対を測定対象として被覆率を計測する。

このテスト基準の有効性として以下のことが期待できる。

(1) 制御フローテスト基準では節点や枝の数が多すぎる場合、測定の単位を導出操作類関数に

することにって一般に測定対象の数を減らすことができる。

(2) 並行処理機能を C プログラムとオペレーティングシステムとによって並行処理機能を導入した場合、実体の数への配慮が必要になる。その場合、第 4 章で述べた実体階層図の上でこの基準を考えるので、C_0 被覆率に含まれない。すなわち、クラスから生成した実体の間で導出操作類関数の呼び出しのループが構成されるとデッドロックが発生する。このような場合、制御フローテスト基準では呼び出し側のみを測定対象にする。すなわち、呼び出される導出操作類関数が 2 つ以上あるときに呼び出しと呼び出される操作類関数とのすべての組合せを実行する保証は、制御フローテスト基準では得られない。Ada におけるランデブー通路 [5] と同様の考え方が必要である。

抽象木操作プログラムでは、仮想関数が用いられているので、操作関数 eval、生成関数、削除関数は、抽象構造木にしたがって呼び出される。

5.2 議論

本節では、オブジェクト指向プログラムの測定可能性および信頼性について検討する。

5.2.1 被覆率の測定方法 (測定可能性)

オブジェクト指向プログラムの被覆率を計測するためには、(1) ソースコードから測定対象の抽出、(2) 被テストプログラムの実行時に測定対象の実行の計測、が可能でなければならない。抽出と計測は以下のとおりに行う。

i) 測定対象の抽出

5.1 節で述べたテスト法では、第 4 章で述べたモデルに従って、測定対象をソースコードから抽出する。測定対象は、文、制御の移行、導出操作類関数、導出操作類関数呼び出し、である。これらは、C++ プログラムのコンパイル時に認識されているので、抽出が可能である。

さらに、実体階層を抽出しなければならない。実体階層は、第 4 章で述べた手順で構築できる。

ii) 実行の計測

被テストプログラムの実行時の測定対象を計測するためには、いくつかの方法がある [4]。その中で、実現の容易性からソースコード変換方式を用いる。ソースコードの変更には、基本的には、Ada 並行処理プログラムのランデブー通路の被覆率測定のための変換方式 [5] を用いる。また、5.3 節で議論する C 並行処理プログラムにおける被覆率測定ツール [12] での実現方式をそのまま用いることが考えられる。ただし、計測のためのソースコードの変換のためには、(1) 計測の集計方式、(2) 計測のためのプログラムの変換の影響、を検討しておく必要がある。

(1) 計測結果は、前節で述べたように、基本的にソースコードに集計する。これによって、(a) テスト充分性評価の目的である信頼性の確認が可能になる、(b) ソースコードとの対応が容易になる、(c) ソースコードは 1 つであるので出力の増大を防止できる、という効果がある。さらに、片山らが検討した実体数を考慮したテスト充分性評価 [13] の場合には、実行時における実体の識別が必要となる。

(2) プログラムの変換による影響は、ランデブー通路の被覆率測定で述べたと同様に、(a) 実行時間の増大、(b) 並行性の減少、(c) プログラムの規模の拡大による記憶領域不足という異常終了の発生、が引き起こされる可能性がある。しかしながら、異常終了以外は、変換前のプログラムで発生する可能性のある動作を制限はするけれども、変更していない。

5.2.2 信頼性

ソフトウェアのテストでプログラムが正しいことを証明するためにはすべての実行順序とすべての入力データとを実現する「全数テスト」が必要であるけれども、現実的ではない。そこで、テスト法に対応した発見可能な不良を明確にする必要がある。 Howden は、テスト法に従ったテストを実施したとき、プログラムが正しいかあるいはある種の不良を含むときその不良をすべて発見できるならばそのテスト法が不良に対して「信頼できる」と定義した [8]。さらに、テストデータの値によっては不良を発見できるならば、「部分的に信頼できる」という。

「信頼できる」を上記のように定義すれば、オブジェクト指向プログラムのテスト充分性評価技法の信頼性は、以下のとおりである。

i) 構文不良やオブジェクトの不完全性、矛盾についてはコンパイラおよび静的解析ツールが信頼できる。特に、変数の型の宣言や導出操作類関数の宣言についての型チェックはプログラムの信頼性を向上させるのに効果があることは、これまでの手続き型言語においてと同様である。

ii) 制御フローテスト基準と導出操作類関数テスト基準は、Howden が解析している [8] ように計算不良、定義域不良、部分路不良に対して信頼できるかあるいは部分的に信頼できる。

iii) 導出操作類関数呼び出しテスト基準は、関数の呼び出し時の引き数の解釈の間違いである通信不良について信頼できる、あるいは部分的に信頼できる。引き数の型の不一致は、項番 i) によってコンパイラあるいは静的解析が信頼できるので、動的な解釈についての通信不良を発見するための方法が必要であり、導出操作類関数呼び出しテスト基準が効果的である。

iv) C++ プログラムでは言語仕様上では並行処理プログラムを記述できないけれども、C++ のライブラリあるいはオペレーティングシステムの機能を用いれば並行処理が可能である。そのために、同期不良が発生する可能性がある。同期不良に対しては、導出操作類関数呼び出しテスト基準では対応が困難であり、C 並行処理プログラムに対して検討したと同様の順序列テスト基準 [11] が必要となる。

5.3 ツール化

オブジェクト指向プログラムの普及に当たっては、支援ツールの存在が大きな役割を果たしている。また、ソフトウェアのテスト法は、テスト充分性の評価や被テストプログラムの実行結果の確認といった大量のデータを扱い、しかも、緊張を強いる作業があり支援ツールが不可欠である。本節では、オブジェクト指向プログラムのテストにおける支援ツールについて述べる。言語に依存する部分については、C++ を対象にする。

i) オブジェクト指向プログラミング言語 C++ は、C 言語の拡張である。そのために、C プログラムのための支援ツールを利用することが可能である。C++ の言語プロセッサである g++ は、C++ のソースコードを入力して C を出力する前処理系 (preprosessor) であるので、実行部分については C の支援ツールを利用することが可能である。しかしながら、ソースコードと関連する部分については新たな支援ツールの開発が必要である。

ii) オブジェクト指向プログラムに対しては、最初に述べたように CASE ツールが発達している [15]。この CASE ツールとの連携が必要である。そのために、ソフトウェア開発におけるリポジトリ機能の実現が重要である。テストに必要なリポジトリは、以下のデータを維持管理する必要がある。

(1) プログラムのソースコード

(2) テストケースやテストデータ

(3) 被テストプログラムのテストデータに対する実行結果

(4) テスト充分性評価データ

iii) 実際のテスト作業を支援するツールとしては以下のものが考えられる。

(1) テスト充分性評価ツール : 5.2 節で述べたように、被テストプログラムから測定対象を抽出し、テストの実施における測定対象の実行を計測する。C++ プログラムの計測には C プログラムの計測ツールを利用することができる。

(2) テスト実施支援ツール : 被テストプログラムにテストデータを与えて実行するテストの実施を支援する。このツールも C プログラムの支援ツールを利用することができる。また、デバッガとして開発されているツールを利用することができる。ただし、並行処理プログラムについては、実行の非決定性があるために、強制実行ツールが必要になる [14]。

(3) テストケース、テストデータ作成ツール : 実際にプログラムに入力するテストデータを作成したり、テストデータの条件と期待出力を規定したテストケースを作成する。テストケースの作成では、本論文で議論したテスト基準を満足するテストケースを自動的に作成する機能が必要である。一方、テストデータ作成は、完全には行えないので人手の介在が必要になる。また、 項番 iii) の (1) で述べたテスト充分性評価ツールで被覆率を計測したときに、テスト基準を満足させるためには被覆率を増大させるためのテストデータが必要であり、その作成を支援する必要がある。

6. おわりに

本論文では、オブジェクト指向プログラムのテスト充分性評価技法について議論した。テスト基準としては、(1) 従来の手続き型言語で用いられている制御フローテスト基準、(2) オブジェクトの操作関数や継承した操作関数を含めた導出操作類関数をすべて少なくとも 1 回実行することを要求する導出操作類関数テスト基準、(3) 導出操作類関数の呼び出しと導出操作類関数の宣言との対をすべて少なくとも 1 回実行することを要求する導出操作類関数呼び出しテスト基準、を提案し

た。今回は、C++ に基づいてテスト基準を検討したけれども、他のオブジェクト指向プログラミング言語で記述されたプログラムにおいても同様のテスト基準を開発することが可能である。ただし、並行処理の記述についてはさらに検討する必要がある。

これらのテスト基準の開発および検討においては、テスト充分性評価技法の構築手順 (メタテスト法) を明確にして実行した。ただし、メタテスト法のテスト充分性評価技法の実際の開発現場での有効性の実証についてはまだ行っていない。さらに、支援ツールについてその実現方法および他のプログラム開発支援ツールとの関連性について議論した。

今後の課題としては、以下のことがある。

i) オブジェクト指向プログラムテスト充分性評価ツールの開発。

ii) テスト充分性評価技法の実際の開発現場での有効性の実証。これを行うためには、支援ツールの開発が必要である。

iii) メタテスト法の確立。メタテスト法は、これまで、逐次処理プログラムのテスト充分性評価技法、並行処理プログラムのテスト充分性評価技法を確立する中で明確になったものである。今回、オブジェクト指向プログラムのテスト充分性評価技法の確立をこの手順で行った。プログラムの開発環境が提唱されて以来、プログラミング言語やプログラミングパラダイムには、言語処理系だけでなく開発支援ツールの必要性が認識されている。テスト支援ツールは開発環境の中で、下流工程の重要な構成要素の 1 つである。メタテスト法によってプログラミング言語の設計時に、テスト支援ツールの機能を検討できるので、メタテスト法の確立は、開発環境の構築に寄与することができる。

参考文献

[1] Analysis and Modeling in Software Development, CACM, Vol.35, No.9, 1992.

[2] 中所武司: ソフトウェア危機とプログラミングパラダイム, 啓学出版, 1992.

[3] Dewhurst, Stephen C., Stark, Kathy T. : Programming in C++ (「C++ 言語入門」小山裕司監訳, アスキー出版局, 1990).

[4] 古川善吾, 牛島和夫: 並行処理プログラムのテスト法に関する一考察, 日本ソフトウェア科学会第 6 回大会論文集, pp.185-188, 1989.

[5] 古川善吾, 牛島和夫: ランデブー通路を用いた Ada 並行処理プログラムのテスト充分性評価, 電子情報通信学会論文誌, Vol.J75-DI, No.5, pp.288-296, 1992.

[6] 古川善吾, 有村耕治, 牛島和夫: 並行処理プログラムにおける共有変数のデータフローテスト基準, 情報処理学会論文誌, Vol.33, No.11, pp.1394-1401, 1992.

[7] Gourlay, J.S. : A Mathematial Framework for the Investigation of Testing, IEEE Trans. Soft. Eng., Vol.SE-9, No.6, pp.686-709, 1983.

[8] Howden, W.E. : Reliability of the Path Analysis Testing Strategy, IEEE Trans. Soft. Eng., Vol.SE-2, pp.208-215, 1976.

[9] Howden, W.E. : Theoretical and Empirical Studies of Program Testing, IEEE Trans. Soft. Eng., Vol.SE-4, No.4, pp.293-298, 1978.

[10] 伊東栄典, 川口豊, 古川善吾, 牛島和夫: C 並行処理プログラムのプロセス間通信に関するテスト充分性評価について, 情報処理学会ソフトウェア工学研究会, No.90, pp.9-16, 1993.

[11] 伊東栄典, 川口豊, 古川善吾, 牛島和夫: 同期誤りに対する順序列テスト基準の信頼性評価について, 平成 5 年度電気関係学会九州支部連合大会論文集, 1993.

[12] 川口豊, 伊東栄典, 古川善吾, 牛島和夫: C 並行処理プログラムのテスト充分性評価システムの試作, 情報処理学会全国大会論文集, 1993.

[13] 片山徹郎, 菰田敏行, 古川善吾, 牛島和夫: タスク型を含んだ並行処理プログラムのテスト法について, 情報処理学会研究会報告, 93-SE-93, pp.181-188, 1993.

[14] 菰田敏行, 片山徹郎, 古川善吾, 牛島和夫: Ada 並行処理プログラムのテストケース作成とその強制実行に関する一考察, 第 20 回 Japan SIGAda, pp.9-15, 1993.

[15] 中谷多哉子: オブジェクト指向 CASE はなぜオブジェクト指向分析なのか, 情報処理学会研究報告, 93-SE-93, pp.133-141, 1993.

[16] Stroustrup, Bjarne : The C++ Programming Language (「プログラミング言語 C++」, 斎藤信男訳), トッパン, 1988.

[17] 館野昌一, 及川一成, 田制貴俊, 川西真木: 基礎からの Smalltalk-80 – オブジェクト指向のプログラミング –, 工学社, 1987.

[18] 植野直樹: 階層化を考慮したオブジェクト交信図, 情報処理学会研究報告, 93-SE-93, pp.143-150, 1993.

[19] United States D.o.D. : Reference Manual for the Ada Programing Language, 1983.