アスペクト指向プログラミングへのモデル検査手法の適用

鵜 林 尚 靖†　玉 井 哲 雄††

アスペクト指向プログラミング (AOP) とは，同期制御，資源共有，性能最適化など複数のオブジェクトにまたがる関心事をアスペクトというモジュール概念を用いてオブジェクトとは独立に記述するプログラミング方式である．オブジェクトとアスペクトは織り込み (weaver) と呼ばれる言語処理系により 1 つのプログラムとして合成される．しかし，AOP では同期制御など処理的にクリティカルな部分がアスペクトとして個別に記述されることが多く，織り込みにより合成された結果が本当に正しいかどうかを確かめることは必ずしも容易ではない．本論文では，このような問題を解決するため，モデル検査と呼ばれる手法を用いて，合成されたプログラムがデッドロックなどの予期しない動作を引き起こすかどうかを自動的に検査するアプローチを提示する．さらに，モデル検査を効率的に適用するための手段として，AOP のメカニズムを利用した検査フレームワークを提案する．この検査フレームワークを適用することにより，複数のオブジェクトにまたがる検査の仕様をアスペクトとしてプログラム本体から分離して記述することが可能になる．

Aspect-Oriented Programming with Model Checking

Naoyasu Ubayashi† and Tetsuo Tamai††

Aspect-Oriented Programming(AOP) is a programming paradigm such that crosscutting concerns including synchronization policies, resource sharing and performance optimizations over objects are modularized as aspects that are separated from objects. A compiler, called weaver, weaves aspects and objects together into a program. In AOP, however, it is not easy to verify the correctness of a woven program because crucial behaviors are strongly influenced by aspect descriptions. In order to deal with such problem, this paper proposes an automatic verification approach using model checking that verifies whether the woven program contains unexpected behaviors such as deadlocks. AOP-based checking framework is proposed in order to use model checking tools efficiently. Using this checking framework, checking items that crosscut over objects can be described as an aspect and separated from a program body.

1. はじめに

アスペクト指向プログラミング (Aspect-Oriented programming. 以下 AOP)[18] は，同期制御，資源共有，性能最適化など複数のオブジェクトにまたがる関心事をアスペクトというモジュール概念を用いてオブジェクトとは独立に記述するプログラミング方式である．従来のオブジェクト指向プログラミングは各々のオブジェクトに求められる要件を機能モジュールとしてカプセル化する目的には優れているが，性能最適化など複数のオブジェクトにまたがる要件を表現するのには必ずしも向いていない．性能最適化のためのコードを追加しようとすると，多くの場合，コードが複数のオブジェクトに分散してしまい，見通しの悪いプログラムになりがちである．AOP はこのような問題意識の下で考え出されたプログラミング方式で，通常の機能はオブジェクトとして，性能最適化などの局面機能はアスペクトとして記述する．1 つのプログラムは複数のオブジェクトとアスペクトから構成されるが，これらを合成するのが織り込み (weaver) と呼ばれる言語処理系である．このように，AOP では複数のオブジェクトにまたがる関心事を見通しよく記述することが可能になるが，一方では，記述が複数のオブジェクトとアスペクトに分散するため，これらを実際に合成したとき，全体として矛盾なく動作するかどうかを確認することが難しくなる．また，アスペクトとして記述されるものは同期制御や性能チューニングなどが中心になるため，通常のプログラミングよりもバグが混入しやすいという側面もある．

本論文では，このような問題を解決するため，モデ

† (株) 東芝 SI 技術開発センター
Systems Integration Technology Center, Toshiba Corporation
†† 東京大学大学院 情報学環
Interfaculty Initiative in Information Studies, Graduate School, University of Tokyo

ル検査 (model checking)[7] と呼ばれる手法を用いて，合成されたプログラムがデッドロックなどの予期しない動作を引き起こすかどうかを静的かつ自動的に検査するアプローチを提示する．モデル検査は，元来，回路設計や通信プロトコルなど有限状態並行システムの正当性を自動的に検査するための技術として発展したが，近年はソフトウェアシステムへの応用が進みつつある．特に，複数の並行プロセスにまたがるような大域的な性質を検査するのに効果がある．AOP の場合も複数のオブジェクトにまたがる性質を検査する必要があり，モデル検査のアプローチは有効であると予想される．本論文では以下の 2 つの観点から AOP へのモデル検査手法の適用について述べ，その有効性を確かめる．

(1) モデル検査手法を利用したプログラムの検査
(2) AOP による検査フレームワークの実現

(1) では，複数のアスペクトとオブジェクトを合成した結果が仕様を満たすかどうかをモデル検査手法で検査するアプローチを提示する．(2) では，モデル検査をさらに効率的に適用するための手段として，AOP のメカニズムを利用した検査フレームワークを提案する．この検査フレームワークを適用することにより，複数のオブジェクトにまたがる検査の仕様をアスペクトとしてプログラム本体から分離して記述することが可能になる．

まず 2 節で AOP の概要と現状の問題点について述べ，つづく 3 節でモデル検査手法を利用した解決方法を提案する．4 節では AOP で作成したプログラムの検査にモデル検査手法を適用した例を紹介する．5 節では AOP ベースの検査フレームワークを提案する．6 節で今後の研究課題について述べ，最後に 7 節でまとめを行う．

2. AOP の考え方とその課題

2.1 AOP の背景

プログラミングにおいてモジュール化は非常に重要な概念である．モジュール化によりプログラムを一枚岩のコードの塊ではなく，あるまとまりを持ったモジュールの集合として記述できる．うまくモジュール化されたプログラムは全体の見通しが良く，理解性，保守性，変更容易性などに優れる．一方，実用的で高品質なプログラムには効率性や安全性が求められる．分りやすくかつ効率性や安全性にも優れるプログラムが開発できれば問題はないが，通常，両者は相反することが多い．分りやすいプログラムを記述すれば性能が出なかったり，性能を重視したプログラムを記述すれば逆にプログラムの構造が失われたりする．

一般的にシステムに求められる要件には機能的要件と非機能的要件の 2 種類がある．現在，手続き型，関数型，オブジェクト指向など様々なプログラミング言語が存在するが，これらが提供する「手続き」「関数」「オブジェクト」などのモジュール化機構は機能的要件を表現する分には優れているが，性能最適化などの非機能的要件については直接的には表現できない．たとえば，プログラムに性能チューニングを施すと，基本機能を表現するコードに性能改善のためのコードが幾個所にも絡み合いコード全体の見通しが悪くなるという現象は，この問題に起因することが多い．

2.2 AOP とは

AOP はこのような問題点を解決するためのプログラミング技術で，以下のような要件をアスペクトと呼ぶモジュール概念を用いて分離する[3]．これらの要件は複数のオブジェクトにまたがるため，横断的関心事 (crosscutting concerns) と呼ばれる．

- エラーチェック戦略
- デザインパターン
- 同期制御
- 資源共有
- 分散にかかわる関心事
- 性能最適化

AOP では，システムに求められる機能を素直に記述したオブジェクト群と横断的関心事を記述したアスペクト群を織り込み (weaver) と呼ばれる言語処理系で合成することにより，「分りやすくかつ効率性や安全性にも優れる」プログラムを生成する．アスペクトとオブジェクトの繋ぎ目 (フックの類) のことを，AOP では合流点 (join point) と呼ぶ．織り込みは合流点の指定により，アスペクトとオブジェクトを織り合わせる．

AOP の目標である「横断的関心事の分離 (separation of crosscutting concerns)」と同じ方向性をもつ研究として AOP の他に，SOP(Subject Oriented Programming)[12]，Composition Filter[1]，ロールモデル[27][17][23][24][25]，Adaptive Programming[10] などがある[9]．自己反映計算 (reflection)[28][29] もこれらの研究との関わりが深い．自己反映計算とは，通常のプログラム記述はベースレベルに，プログラムの振る舞いを規定するものをメタレベルに記述するプログラミング方式である．横断的関心事をメタレベルに記述することにより，通常のオブジェクト機能と分離した形で表現できる．しかし，自己反映計算は強力すぎて，通常のプログラマが用いるには難しいという面がある．AOP では合流点の切り口を提供し，自己反映計

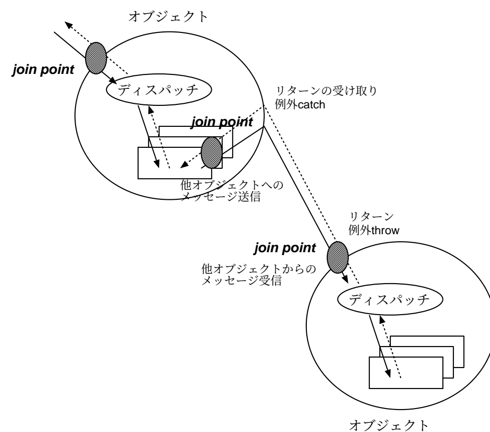


図 1　AspectJ における合流点
Fig. 1　Join point in AspectJ

算が持つ機能をプログラマが容易に使用できるようにしている．AOP では，自己反映計算は一番低レベルの AOP として位置付けられている[18]．

2.3　AOP 言語 AspectJ

AOP をサポートする言語はいくつか存在するが，その中の 1 つとして Xerox Parc で開発している Java ベースの AOP 言語 AspectJ[3] がある．AspectJ のプログラムは，通常のクラス定義群と aspect 構文で定義されるアスペクト定義群から構成される．AspectJ はオブジェクト指向言語をベースにした汎用的な AOP 言語を目的としており，合流点 (pointcut 構文で記述) についてもオブジェクト指向言語がもつメカニズムに基づいたものになっている．たとえば，合流点として，メッセージ送信/受信や例外 throw/catch などの切り口を指定できる (図 1[3])．アスペクト記述のための主な言語要素として，合流点の他に introduction 構文と before/after 構文がある．introduction 構文は既存クラスに新規メソッドを追加する際に使用される．一方，before/after 構文は合流点の前後にコードを追加する際に使用される．たとえば，合流点として「あるメッセージの受信点」を設定した場合，そのメッセージに対応するメソッドに制御が移る前に実行したいコードを before 構文を使用して記述できる．

例: エラーロギング

通常，エラーログの処理はプログラム中の様々な場所に分散して出現する．そのため，もしエラーログの処理に変更が発生すれば，多くの箇所にプログラム変更が発生しまいバグが混入する原因となる．以下のプログラム☆はエラーログ処理を PublicErrorLogging アスペクトとして分離したものである．合流点として publicInterface が定義されているが，これは mypackage の中のクラスのインスタンスが public メソッドに対応するメッセージを受信する点を設定したものである．2 つの after 構文は各々publicInterface に対応するメソッド中でリターンまたは例外 throw した場合にメッセージを表示したりログを書き出したりするコードを追加したものである．

```
package mypackage;

//  クラス
class Foo { public void m1() {} }
class Bar extends Foo {
    public  void m1()  { super.m1(); }
    private void m2()  {}
}
class Log { void write(Object o) {} }

//  アスペクト
aspect PublicErrorLogging {
  static Log log = new Log();

  // 合流点の指定
  pointcut publicInterface ():
    call(public * *(..)) && target(mypackage..*);

  // 既存メソッドの修正
  after() returning (Object o): publicInterface() {
      System.out.println(thisJoinPoint);
  }

  after() throwing (Error e): publicInterface() {
    log.write(e);
  }
}
```

☆ [3] から入手したプログラムに若干の変更を加えたもの．

2.4　AOP の課題

AOP はモジュール化において最も重要な概念の 1 つである「関心事の分離」という面で優れた言語メカニズムを提供するが，その一方で，プログラムの正しさを検査する面で 1 つの問題点を提起する．プログラマから見た場合，AOP によるプログラミングはあくまでもアスペクトとオブジェクトの記述である．それらを合成して実行プログラムを生成するのは織り込みの役割である．すなわち，プログラマはオブジェクトとアスペクトの記述のみに専念すればよい．ところが，作成したプログラムを実際にテストする段階になると，プログラマは頭を切替えて織り込みにより合成された実行プログラムの動きをイメージする必要がある．テストの対象となるのはオブジェクトやアスペクトそのものではなく，それらが合成された結果だからである．このように AOP ではプログラミング段階とテスト段階では大きなギャップが発生する．また，AOP では同期制御や性能最適化など処理的にクリティカルな部分がアスペクトとして個別に記述される場合が多く，織り込みにより合成された結果が本当に正しいかどうかを確かめることは必ずしも容易ではない．特に対象プログラムが並行環境下における同期制御や性能最適化を扱っている場合，検査行為はさらに困難になる．並行プログラムは実行ケースが多岐にわたるため，実行

時テストだけでプログラムの正しさを検査することは通常難しい．AOP の場合，プログラムの記述がオブジェクトとアスペクトに分離されるため，検査の困難度は更に増すことになる．

また，別の問題として，個々のオブジェクトやアスペクトには問題なくても，それらを合成した結果にバグが潜む可能性がある．このようなバグは，織り込みがアスペクトを合成する際の順番の違いにより発生したりしなかったりする．たとえば，アスペクト X とアスペクト Y が存在する場合，X，Y の順に織り合わせれば問題なくとも，Y，X の順に織り合わせるとバグが生じるかもしれない．通常の AOP 言語では織り込みによる合成方針は実装依存であることが多く，このことが，AOP によるプログラミングの検査をより困難にしている面がある[20]．

例： エラーロギングの場合

前節のエラーロギングのプログラムで，以下の内容は成立するであろうか?

(1) クラス Bar のメソッド m1 で例外が発生した場合はログが採られるか?

(2) クラス Bar のメソッド m2 で例外が発生した場合はログが採られるか?

1 は真であるが，2 は偽である．この例ではアスペクトが 1 つしかなく，合流点の指定も簡単であるため，真偽を確かめることは容易であるが，通常はもっと複雑になる．このような場合，織り込みによる合成イメージを頭の中でシミュレーションすることは必ずしも容易でない．

3. モデル検査による解決方法

本節では，2 節で提起した問題を解決するため，モデル検査の手法を AOP に応用する方法を提案する．

3.1 モデル検査とは

モデル検査とは，ある性質を与えたとき，対象となる構造 (システム) がその性質を満たすかどうかを形式的に調べる手法である[7]．ある状態 s において構造 M が論理式φを満たすとき，M のことをφのモデルと呼び，M,s |= φと表記する．多くの場合，M は有限状態遷移マシンで，φは時相論理式 (temporal logic formula) で記述される．モデル検査により，有限状態遷移マシン M の状態空間を探索して，状態 s から時相論理式φを満たす状態に到達する遷移パスが存在するかどうかを機械的かつ自動的に調べることが可能になる．代表的な時相論理として，CTL*(Computation Tree Logic)，CTL，LTL(Linear Temporal Logic) などがある．CTL および LTL は，CTL*に制限を付けた論理である．CTL*式は，以下のパス限量子 (path quantifier) と時相演算子 (temporal operator) から構成される．

表 1 検査ツール
Table 1 Checking tools

ツール名	モデル記述言語	検査仕様の指定方法
SPIN	PROMELA	LTL 式，表明
Java PathFinder	Java	表明
Bandera	Java	LTL 式
ESC/Java	Java	表明

- パス限量子: A(“for all computation paths”), E(“for some computation path”)
- 時相演算子: X(“next time”), F(“in the future”), G(“globally”), U(“until), R(“release”)

例： エラーロギングの場合

M をエラーロギングの構造，s を初期状態とする．以下は,「クラス Bar のメソッド m1 で例外発生の状態になると (ThrowingException-Bar-m1)，その後のすべてのパスでいつかはエラーロギングを行う (ErrorLogging-Bar-m1)」ということが常に成立する「s を初期状態とするパス」が構造 M の中に存在することを示している．

```
M,s |= AG(ThrowingException-Bar-m1
          -> AF ErrorLogging-Bar-m1)
```

3.2 主な検査ツール

検査ツールとして有名なものに，SPIN[15]，Java PathFinder (以下 Jpf)[13][14]，Bandera[5][8] などがある．SPIN では，対象となるシステムを形式的なプログラミング言語 PROMELA で記述し，LTL に基づいたモデル検査を行う．SPIN は長い歴史をもち非常に活用実績の多いモデル検査系であるが，モデルの作成に大きな労力を必要とするという問題がある．モデル作成に際して，PROMELA という独自言語を使用しなければならないからである．これに対し，Jpf や Bandera など最近のモデル検査系では Java プログラムをそのものをモデルとして取り扱う方式を採っている．そのため，プログラマは検査のために特別なモデルを作成する必要はない．一方，モデル検査の範疇には属さないが，定理証明をベースとする静的解析ツールとして，ESC/Java(Extented Static Checker for Java)[22] などがある．ESC/Java は，Java プログラムを静的解析し，実行時に発生するエラーを検出するツールである．ESC/Java では検査したい仕様を pragma と呼ばれる特別な形式のコメントとして指定する．Jpf，Bandera，ESC/Java では Java プログラムは実行せず，あくまでもソースレベルで静的かつ自

動的に検査を行う．表1は主な検査ツールについてまとめたものである．

検査仕様の指定方法には，コマンドから LTL 式を与える方式と，モデルの中に表明 (assertion) を追加する方式の2通りがある．表明とは，システムの状態をチェックするもので，モデル記述の中に埋めこまれる．SPIN の場合は両方式をサポートしているが，Jpf では表明方式のみ，Bandera では LTL 式方式のみサポートしている．表明方式は，プログラマに時相論理の知識を求めない分，使いやすい．プログラマは調べたい性質を表明として挿入すれば済む．Jpf では表明は以下のような Verify クラス (一部) を使用して記述される[14]．Verify クラスの本体は実行時のみ意味を持ち，モデル検査時には意味を持たない．デッドロックについては表明の有無に関係なく検出する．

```
class Verify {
  public static void assert(boolean b) {
    if (!b) System.out.println(
                "*** assertion broken");
  }
  public static void assert(String s,boolean b) {
    if (!b) System.out.println(
                "*** assertion broken : " + s);
  }
}
```

表明方式の短所はプログラム全体に関わる大域的な性質を指定するのには必ずしも向いていない点である．これに対し，LTL 式方式は時相論理式の知識が求められるものの，大域的な性質を指定するのに優れている．

4. 適 用 例

本節では，AOP で作成されたプログラムの検査にモデル検査が有効であることを確認する．検査対象のプログラム例として，並行環境下で動作するバッファ経由の Producer/Consumer 問題☆を取り上げる．

4.1 例題プログラム

例題プログラムは，図2に示すように，Producer がデータをバッファに put し，それを Consumer が get するという簡単なものである．バッファは循環的に使用されるものとする．Producer はバッファに空きがあれば順にデータを put する．バッファが満杯になると，Consumer がバッファからデータを取り出して空きができるまで wait する．Producer は空きができた段階でデータを put する．一方，Consumer はバッファからデータを順に get し，もしバッファが空であれば，Producer がデータを put するまで wait する．

☆ Jpf ユーザマニュアル[14] の例題を一部修正したもの．なお，本節で述べる検査内容や反例は Jpf ユーザマニュアルに基づいている．

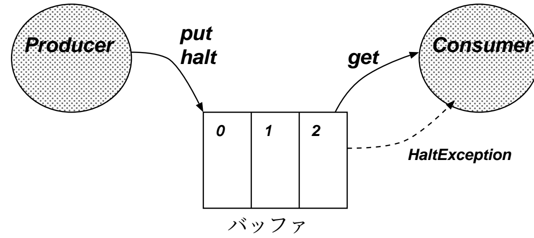


図 2 Producer/Consumer 問題
Fig. 2 Producer/Consumer problem

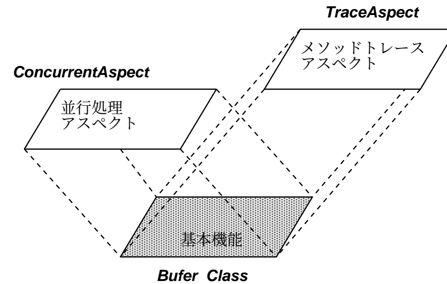


図 3 AOP ベースの Buffer 記述
Fig. 3 Buffer description based on AOP

そしてデータが入った段階で get する．Producer は全データをバッファに書き終わると halt フラグを立てる．バッファにデータが存在せず，かつ halt フラグが立っている状態で Consumer がデータを get しようとすると HaltException が throw される．なお，例題では，簡単のため，Producer と Consumer の数をそれぞれ1つに限定する．

付録 A は AspectJ を使用して例題プログラムを記述したものである．ここでは，図3に示すように，ConcurrentAspect(並行処理に対応するためのアスペクト)，TraceAspect(メソッドトレースのためのアスペクト) の2つがアスペクトとして Buffer クラスから分離して記述されている．Buffer クラスにはバッファとしての本来機能しか記述されていない．このように付録 A の記述はバッファに関わる様々な関心事が分離されている．図4は，付録 A のプログラム構造を模式化したものである．ここでは，重要なクラスとアスペクトを抜きだして図式化した．図中の矢印は織り込みによる合成を示している．ConcurrentAspect, TraceAspect のコードが Buffer クラスや Producer クラスそして Consumer クラスに挿入される様子が分かる．

4.2 モデル検査による検査

例題プログラムは小さいがマルチスレッド下で動作するため，その振る舞いは見た目ほどには単純ではな

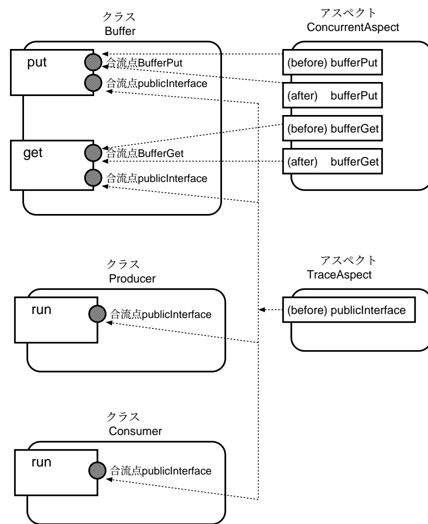


図 4 プログラム構造
Fig. 4 Program structure

い．特に付録 A ではプログラムが 6 個のクラスと 2 個のアスペクトに分離されるため，プログラム全体としての振る舞いを理解するのは必ずしも容易ではない．実際，図 5 に示した手順で検査を行うと，Consumer クラスで指定した表明 (Jpf の表記法を使用) で違反が発生する．時相論理の観点から言うと，例題プログラムの状態空間には，初期状態から表明を満たす状態 (Producer が put したデータ数と Consumer が get したデータ数が一致する状態) に到達しない遷移パスが存在することを意味する．実際，$if(halted)$ (ConcurrentAspect の 32 行目) の箇所で不具合が発生することが分かる．

(1) Producer がバッファの位置 0，1，2 に 3 つの値を put し，wait メソッドを呼び出す．
(2) Consumer が位置 0 の値を取り出し，Producer に notify する．その後，Consumer は残り 2 つの値を取り出す．
(3) Producer は位置 0，1，2 に各々4 番目から 6 番目の値を put し，最後に halted フラグをセットする．
(4) Consumer は get メソッドを呼び出すが，この場合，$if(halted)$ が成立し HaltException が throw されてしまう．Consumer は 4 番目から 6 番目の値を get できず，Consumer クラスに指定したアサーションが成立しなくなる．

$if(halted)$ ではなく $if(usedSlots == 0)$ が正しいコードである．このようにアスペクトには処理上クリティカルな部分が記述される場合が多く，モデル検査はこのような問題の解決に有効である．実際，上記の反例は実行時に常に起きるわけではない．もし Producer が halted フラグをセットする前に Consumer が 4 番目から 6 番目の値を get してしまえば，Consumer クラスに指定した表明は成立する．実行時テストだけでは今回の例題のようなバグを完全に取り除くことは難しい．

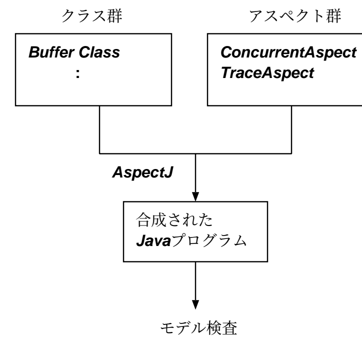


図 5 検査手順
Fig. 5 Checking procedure

5. AOP ベース検査フレームワーク

前節までの説明により AOP の検査にモデル検査が有効であることが示されたが，ここでは，さらにモデル検査を効率的に適用するための手段として，AOP のメカニズムを利用した検査フレームワークを提案する．ただし，ここで提案する検査フレームワークはモデル検査以外の検査，たとえば，実行時検査などにも応用可能である．

5.1 基本的な考え方

付録 A では，検査のための表明は Consumer クラスに直接埋め込まれたが，この方法だと，検査のたびにソースコードを修正しなければならない．通常，検査したい事項は複数のオブジェクトやアスペクトにまたがることが多く，検査内容そのものもアスペクトとして捉えることができる．付録 B は，検査のための記述をアスペクトとして分離したものである．この方式には以下のような長所がある．

- 検査内容が本来機能から分離され，プログラムの見通しがよくなる．
- 複数オブジェクトにまたがる検査内容を 1 つのアスペクトにカプセル化できる．
- 検査のたびに，プログラム本体を修正しなくて良い．単に検査用アスペクトを合成の際に指定するだけで済む．
- 検査したい視点ごとにアスペクトを定義することができる．各視点単位に検査 (単体検査) を行う

ことも，それらを合成した全体で検査 (組み合わせ検査) を行うことも可能である．

検査のための表明はプログラムに対する仕様と見なすことができるので，プログラムの中に直接埋め込んだ方が分かりやすいという意見もあるが，これには少なからず問題点がある．確かに，プログラムに対する検査ストーリが 1 つしかない場合については問題は少ない．しかしながら，検査ストーリが複数ある場合は (通常はこのケースに該当するがほとんどである)，それらをすべて検査対象プログラム中に埋め込んでしまうと，どの表明がどの検査ストーリに関係しているのか，また，一つの検査ストーリにおいて，どの表明とどの表明が結び付いているのか理解することが困難となる．また，検査にあたって複数の検査ツールを併用する場合，ツールごとに異なる形式の表明がコード中に乱立してしまいプログラムの理解を困難にしてしまう．このような問題の解決策として，検査仕様をアスペクトとして分離する方式は有効である．

検査仕様をアスペクトとして検査対象プログラムから分離独立して記述するには，検査で必要となる性質が合流点から捕捉できなくてはならない．逆に言えば，検査のためのコードを検査対象プログラムに付加するためのフックを AOP 言語が合流点として提供できなければ，検査仕様をアスペクトとして分離できないことを意味する．ここでは，検査にはどのような合流点が必要となるのか，そして，そのような合流点を利用した検査フレームワークをどのように構築すれば良いのかについて述べる．

5.2　検査に必要な合流点

AspectJ[4] では，契約による設計 (Design by Contract)[19] に基づいたプログラミングスタイルが可能である．このスタイルでは，メソッド起動に関連する合流点の前 (before) に事前条件を検査するコードを，後 (after) に事後条件を検査するコードを記述する．これにより，メソッドが要求仕様を満たす形で実行されたか否かが確認できる．メソッド起動の前後に事前条件や事後条件を設定する方法は機能的な面で検査を行うのに有効である．しかしながら，これだけですべての検査を行うのは難しい．たとえば，安全性や生存性といった性質はメソッド起動に関わる事前条件や事後条件で必ずしも表現されない．これらの性質はメソッドの実行過程により影響を受けるため，状態の変化などを捕らえないと十分な検査を行うことはできない．たとえば，不正な実行が可能になるような状態に遷移する可能性があるかなどが検査仕様として記述できる必要がある．そのためには，メソッド起動以外に状態の変化に関わる合流点を利用した検査仕様の記述を可能にする必要がある．

表 2　AOP ベース検査フレームワークで使用する合流点
Table 2　Join points used in AOP-based checking frameworks

観点	事前条件/事後条件	合流点	コード修正 (advice)
メソッド起動	事前条件	call(シグニチャ)	before
	事後条件	call(シグニチャ)	after
状態遷移	事前条件	set(シグニチャ)	before
	事後条件	set(シグニチャ)	after
特定条件	事前条件	if(ブール式)	before
	事前条件	if(ブール式)	after

ここでは AspectJ が提供する合流点の観点から，AOP ベースの検査フレームワークについて考えてみることにする．表 2 は検査に利用可能な合流点を示したものであり，大きく 3 つのパターンに分類される．第 1 のパターンは前述したメソッド起動に関わる合流点であり call 構文により指定される．第 2 のパターンはフィールドセットに関わる合流点であり set 構文により指定される．この合流点は状態遷移を捕捉するのに使用される．第 3 のパターンは特定条件の変更に関わる合流点であり if 構文により指定される．

合流点には静的に決まるものと動的に決まるものとがある．静的な合流点は機織りによってアスペクトとオブジェクトを織り合わせる際に決まるが，動的な合流点は実行時でないと決まらない．AspectJ を例にとると，call(メソッド起動) は静的な合流点であるが，cflow(プログラムの制御) などは動的に決まる合流点である．静的な合流点のみを用いた検査フレームワークはモデル検査にも実行時検査にも適用可能であるのに対し，動的な合流点を用いた検査フレームワークは実行時検査のみにしか利用できない．これは，モデル検査などの静的検査ツールを AOP に適用する際の限界でもある．

5.3　検査フレームワーク

AOP ベース検査フレームワークは複数の検査用アスペクトから構成される．個々の検査用アスペクトは，表 2 に示した合流点の前後に事前条件と事後条件を設定することにより目的とする検査を実施する．以下は AspectJ による検査用アスペクトの記述例である．

```
aspect MethodFunctionCheck {
   // 合流点を設定
   pointcut invokeMethodPointcut(Foo o):
     call(public void bar()) && target(o);
   before(Foo o): invokeMethodPointcut(o){
     // 事前条件を記載
   }
   after(Foo o): invokeMethodPointcut(o){
     // 事後条件を記載
   }
}
```

検査用アスペクトは検査ストーリ単位に用意することになるが，検査ストーリ間に類似性がある場合は，スーパクラスに相当する検査用アスペクトを定義することになる．アスペクト間に継承が使えるかどうかはどの AOP 言語を用いるかに依存するが，AspectJ では継承は可能になっている．継承を用いて検査用アスペクトを体系化することにより，オブジェクト指向フレームワークに習った検査フレームワークを構築することが可能になる．

検査フレームワークは検査に必要なる枠組を与えるが，残念ながらこれのみですべての場合を網羅することはできない．たとえば，検査フレームワークのみではデッドロックそのものは検出できない．このような場合は，先に示した Jpf のように表明の有無に関わらずデッドロック等を検出できる検査ツールを併用する必要がある．検査ツールには各々長所と短所があるので，目的に応じてこれらを組み合わせて利用するとより効果の高い検査が実現できる．ただし，検査ツールにより表明の記述方式は異なるので，どのツールでどのような項目を検査するのかを明確にした上で，各々について検査用アスペクトを記述する必要がある．検査時は必要なアスペクトのみを検査対象プログラムと合成すればよい．アスペクトで検査内容を記述すると，利用する検査ツールが変わっても検査対象のコードには変更は及ばない．このことは，検査に AOP を適用する利点の一つでもある．図 6 は複数の検査ツールを同時に利用した場合の検査フレームワークを図示したものである．

前述のように，AOP ベース検査フレームワークはモデル検査だけでなく実行時検査にも適用可能である．モデル検査用のアスペクトと実行時テスト用のアスペクトを使い分けることにより，総合的な検査環境を構築することが可能になる．

5.4 検査フレームワークを利用した検査プロセス

AOP ベース検査フレームワークを利用した場合の検査プロセスについて述べることにする．前述のように，AOP ベース検査フレームワークでは，検査ストーリごとに検査アスペクトを作成する．検査ストーリには機能性の検査もあるし，機能では捉えられないような性質，たとえば，安全性や生存性などもある．機能面の検査ストーリはユースケース等から導くことが可能であるが，機能以外の性質はユースケースでは捉えきれない．ドメインエンジアリングの分野では上流段階でフィーチャモデル (feature model) を作成するという手法がある[9]．フィーチャとはシステムに求められる特徴や性質を表現したものであり，この中にはユースケースで表現できるような機能の他に非機能的な性質も含まれる．AOP ベース検査フレームワークでは，検査プロセスの最初にこのフィーチャモデルの作成プロセスを置くことにする．

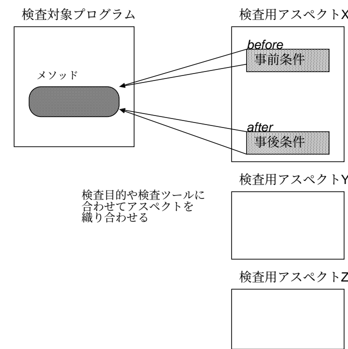


図 6 複数のツールを利用した AOP ベース検査フレームワーク
Fig. 6 AOP based checking framework with multiple checkers

AOP ベース検査フレームワークによる検査プロセスのもう一つの特徴は，XP(eXtreme Programming)[6] との親和性である．XP ではプログラム修正のたびに単体テストを実施し，修正による不具合が生じていないかを常に確認することにより品質を高めているが，そのためのツールとして Java に対応した JUnit などの単体テストフレームワークを提案している．AOP ベース検査フレームワークも，AOP を対象にした単体テストフレームワークと捉えることができる．また，検査ストーリをアスペクトとして分離して記述することに，XP でいうテストファースト (test first) の実行が容易になる．テストファーストでは，プログラムの作成に先立ちテストを書くというプラックティスである．これにより，検査ができないようなプログラムを作成するのを防ぐことができる．最初にフィーチャモデルの作成を行い，それに続いて個々のフィーチャに対応する検査用アスペクトを記述するというプロセスを採用することにより，テストファーストのプラックティスを実践することが可能になる．実際には，検査用アスペクトの作成とプログラムの作成はインクリメンタルに行われる．たとえば，合流点の設定をどう

するかなどは，反復の過程で具体化されていく．検査用アスペクトは検査対象プログラムに依存しない形で記述されるべきであるが，既に存在するプログラムに対して検査用アスペクトを全く独立に記述することは必ずしも容易ではない．実際には検査用アスペクトにストーリ性を持たせ，対象プログラムから独立して記述するには，対象プログラム側の記述にも工夫が必要とされる．検査対象プログラム内でのモジュール独立性，検査用アスペクト同士のモジュール独立性に加えて，検査対象プログラムと検査用アスペクトの間にもモジュール独立性が求められる．このようなモジュール独立性は，ここで示したような反復過程を経て実現されるものである．

AOP ベース検査フレームワークに基づく検査プロセスをまとめると以下のようになる．

(1) フィーチャモデルを作成し，システムに求められる機能的要件と非機能的要件をフィーチャとして抽出する．

(2) 各フィーチャに対応する検査ストーリを作成する．

(3) 各検査ストーリ単位に検査用アスペクトを作成する．

(4) システムを実装するプログラムを作成する．このプログラムの作成と検査用アスペクトの作成はインクリメンタルに行われる．合流点の設定は反復の過程で詳細化，明確化される．

(5) 単体検査を行う．単体検査は，プログラムと検査用アスペクトを個々に織り合わせた結果に対し行う．検査にあたっては，複数の検査ツールを適用することができる．

(6) 組み合わせ検査を行う．組み合わせ検査も，単体検査同様，プログラムと検査用アスペクトを織り合わせた結果に対し行う．ただし，単体検査とは異なり，すべての検査用アスペクトを織り合わせる．

(7) 最後に実行時検査を実施する．組み合わせ検査まではモデル検査などの静的ツールを使用して実施できるが，最後は実行時検査を実施することが望ましい．

6. 今後の課題

本論文では，AOP の検査にモデル検査を適用する方法を提案したが，モデル検査には性能やメモリ効率などの面で従来から問題が指摘されてきた．たとえば，Jpf や Bandera などの検査ツールではソースプログラムそのものをモデルとしており，状態空間のサイズが爆発しやすいという問題が生じる．これに対し，Bandera では，プログラムスライシングなどの技術を使用して，Java プログラムから LTL 式で記述された検査仕様に関連する有限状態モデルのみを抽出するアプローチを採っている．これにより，モデルのサイズを大幅に小さくすることが可能になる．このように，モデル検査に関わる問題点は改善されつつあり，今後，実用化段階に入ってくるものと考えられる．

モデル検査や AOP ベース検査フレームワークの考え方は AOP だけではなく通常のオブジェクト指向開発へも適用可能である．オブジェクト指向においても継承の問題があり，どのメソッドが実際に実行されるのかソースコードを見ただけでは分かりにくい場合がある．特にフレームワークを利用した場合，作成したアプリケーションが本当に正しく動作をするのかを確かめることは容易ではない．また，アプリケーション開発にコンポーネントを利用した場合も，そのコンポーネントが正しく動作するのかを確かめることも必ずしも容易ではない．コンポーネントは，それが利用されるコンテキストにより動作が異なる可能性があるからである．たとえば，コンポーネント C がメソッド m1 と m2 を持っており，m1 → m2 の順番にメソッドが起動された場合は正常に動作しても，m2 → m1 の順番に起動された場合は異常な動作をするかもしれない．フレームワークにしてもコンポーネント間の連携にしても，横断的関心事の 1 つと考えられ，本論文の手法は有効だと思われる．

本論文で提案した検査手法はプログラミング段階のものであるが，これを設計段階にも応用しようという考え方は当然成立し得る．実際，AOP の考え方をより上位の設計段階に応用しようという試みもある．このような試みは AOD(Aspect-Oriented Design) と呼ばれており，たとえば，これをソフトウェアアーキテクチャの記述に応用する研究もなされている[21)]．更にソフトウェアアーキテクチャを ADL(Architecture Description Language) で記述し，その正しさをモデル検査を用いて調べる研究もなされている[2)]．ところが，AOP については本論文で取り上げた AspectJ のような支援環境があるが，AOD についてはまだ環境が整備されていない．したがって，AOD で設計しても，その内容を手作業で AspectJ などのプログラミング言語に変換しなければならない．また，織り込みによる合成が可能なのはプログラミング段階であるため，設計段階で内容が本当に正しいかどうかを検査するには，設計者自身が合成結果を頭の中で思い浮かべるしか方法はない．今後は AOD の段階で設計内容を検査

できるような環境が求められてくると思われるが，これに関連して，Nelson らは横断的関心事から構成されるシステムの性質を形式的に検証する試みを行っている[20]．Nelson らは形式言語として Z をベースとした Alloy[16] とラベル付き遷移システム (LTS: Labeled Transition Systems) を採用している．

本論文のアプローチは，オブジェクトとアスペクトを織り込みにより合成した結果に対してモデル検査を適用するものであった．これに対して，オブジェクトとアスペクトは「関心事の分離」の視点から別々のモジュールとして分離されているのだから，これらは「別々に検査できるべき」という考え方も成立し得る．実際，プログラムの規模が大きくなった場合，このようなモジュール単位の検査は重要であり，今後の研究課題の 1 つと言える[11]．これについては，たとえば，アスペクト記述からオブジェクトスタブを生成し，これらを合成した結果に対し，モデル検査を適用するなどの方法が考えられる．

7. おわりに

本論文では，AOP へのモデル検査手法の適用について，1) モデル検査手法を利用したプログラムの検査と 2)AOP を利用した検査フレームワークの実現の 2 つの面から考察した[26]．本論文で提案したアプローチを実用レベルで利用するには未だ克服しなければならない課題がいくつか存在するが，1 つの方向性としてその有効性が確認できた．

参考文献

1) Akist, M. and Tripathi, A.: Data Abstraction Mechanisms in Sina/ST, *Proc. OOPSLA'88*, pp.265-275, 1988.
2) Allen, R., Garlan, D., and Ivers, J.: Formal Modeling and Analysis of the HLA Component Integration Standard, *Proc. FSE'98*, pp.70-79, 1998.
3) AspectJ. http://aspectj.org/.
4) The AspectJ Programming Guide, 2001.
5) Bandera. http://www.cis.ksu.edu/ santos/bandera/.
6) Beck, K.: Extreme Programming Explained: Embrace Change, Addison-Wesley, 1999.
7) Clarke, E., Grumberg, O., and Peled, D.: Model Checking, The MIT Press, 1999.
8) Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., and Zheng, H.: Bandera: Extracting Finite-state Models from Java Source Code, *Proc. ICSE 2000*, 2000.
9) Czarnecki, K. and Eisenecker, U.W.: Generative Programming, Addison-Wesley, 2000.
10) Demeter Project. http://www.ccs.neu.edu/research/demeter/.
11) Elrad, T., Aksits, M., Kiczales, K., Lieberherr, K., and Ossher, H.: Discussing aspects of AOP, *COMMUNICATIONS OF THE ACM*, vol.44, no.10, pp.33-38, 2001.
12) Harrison, W. and Ossher, H.: Subject-oriented Programming, *Proc. OOPSLA'93*, pp.411-428, 1993.
13) Havelund, K., and Pressburger, T.: Model checking Java programs using Java PathFinder, *International Journal on Software Tools for Technology Transfer*, 1999.
14) Havelund, K.: Java PathFinder User Guide, 1999.
15) Holzmann, G.J. and Smith, M.H.: The Model Checker SPIN, *IEEE trans. SE*, vol.23, no.5, pp.279-295, 1997.
16) Jackson, D.: Aloca: the alloy constraint analyzer, *Proceedings of ICSE 2000*, 2000.
17) Kendall, E.A.: Role Model Designs and Implementations with Aspect-oriented Programming, *Proc. OOPSLA'99*, pp.353-369, 1999.
18) Kiczales, G., Lamping, J., Mendhekar A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J.: Aspect-Oriented Programming, *Proc. ECOOP'97*, Lecture Notes in Computer Science, Springer, vol.1241, pp.220-242, 1997.
19) Meyer, B.: Object-Oriented Software Construction, 2nd Edition, Prentice Hall, 2000.
20) Nelson, T., Cowan, D. and Alencar, P.: Supporting Formal Verification of Crosscutting Concerns, *Proc. REFLECTION 2001*, Lecture Notes in Computer Science, Springer, vol.2192, pp.153-169, 2001.
21) Noda, N. and Kishi, T.: Aspect-Oriented Design: An Approach to Designing Qaulity Attributes, *Proc. APSEC'99*, pp.230-237, 1999.
22) K. Rustan M. Leino, Greg Nelson, and James B. Saxe: ESC/Java User's Manual, 2000.
23) Tamai, T.: Objects and roles: modeling based on the dualistic view, *Information and Software Technology*, vol.41, no.14, pp.1005-1010, 1999.
24) Ubayashi, N. and Tamai, T.: An Evolutional Cooperative Computation Based on Adaptation to Environment, *Proc. APSEC'99*, IEEE Computer Society, pp.334-341, 1999.
25) Ubayashi, N. and Tamai, T.: Separation of Concerns in Mobile Agent Applications, *Proc. REFLECTION 2001*, Lecture Notes in Computer Science, Springer, vol.2192, pp.89-109, 2001.

26) Ubayashi, N. and Tamai, T.: Aspect-Oriented Programming with Model Checking, *Proc. AOSD 2002 (to appear)*, 2002.
27) VanHilst, M. and Notkin, D.: Using Role Components to Implement Collaboration-Based Designs, *Proc. OOPSLA'96*, pp.359-369, 1996.
28) 渡部卓雄: リフレクション, コンピュータソフトウェア, Vol.11, No.3, pp.5-14, 1994.
29) Zimmermann, C.: Advances on Object-Oriented Metalevel Architectures and Reflection, CRC Press, 1996.

付 録

A. Producer/Consumer 問題の記述

Buffer クラス

```
 1: class Buffer {
 2:   static final int SIZE = 3;
 3:   Object[] array = new Object[SIZE];
 4:   int putPtr = 0;
 5:   int getPtr = 0;
 6:
 7:   public synchronized void put(Object x){
 8:     array[putPtr] = x;
 9:     putPtr = (putPtr + 1) % SIZE;
10:   }
11:
12:   public synchronized Object get()
13:                throws HaltException {
14:     Object x = array[getPtr];
15:     array[getPtr] = null;
16:     getPtr = (getPtr + 1) % SIZE;
17:     return x;
18:   }
19: }
```

Concurrent アスペクト

```
 1: aspect ConcurrentAspect
 2:       of eachobject(target(Buffer)) {
 3:   int usedSlots = 0;
 4:   boolean halted = false;
 5:
 6:   // --- put メソッドの修正 ---
 7:   pointcut bufferPut(Buffer b):
 8:     call(public * put(..)) && target(b);
 9:
10:   before(Buffer b): bufferPut(b){
11:     while (usedSlots == Buffer.SIZE)
12:       try{
13:         b.wait();
14:       }catch(InterruptedException ex){};
15:   }
16:
17:   after(Buffer b): bufferPut(b){
18:     if (usedSlots == 0) b.notifyAll();
19:     usedSlots++;
20:   }
21:
22:   // --- get メソッドの修正 ---
23:   pointcut bufferGet(Buffer b):
24:     call(public * get(..)) && target(b);
25:
26:   before(Buffer b): bufferGet(b){
27:     while (usedSlots == 0 && !halted)
28:       try{
29:         b.wait();
30:       }catch(InterruptedException ex){};
31:
32:     if (halted){  /*** BUG!! ***/
33:       HaltException he = new HaltException();
34:       b.throw(he);
35:     }
36:   }
37:
38:   after(Buffer b): bufferGet(b){
39:     if (usedSlots == Buffer.SIZE) b.notifyAll();
40:     usedSlots--;
41:   }
42:
43:   // --- halt メソッドの追加 (Introduction) ---
44:   public synchronized void Buffer.halt() {}
45:
46:   pointcut bufferHalt(Buffer b):
47:     call(public * halt(..)) && target(b);
48:
49:   before(Buffer b): bufferHalt(b){
50:     halted = true;
51:     b.notifyAll();
52:   }
53: }
```

Trace アスペクト

```
1: aspect TraceAspect {
2:   pointcut publicInterface(): call(public * *(..));
3:
4:   before(): publicInterface(){
5:     System.out.println(
6:       "[Trace public methods] "
7:          + thisJoinPoint.toLongString());
8:   }
9: }
```

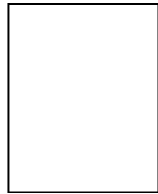
Main クラス他

```
 1: // -----------------
 2: // The Main class
 3: // -----------------
 4: class Main {
 5:   public static void main(String[] args) {
 6:     Buffer b = new Buffer();
 7:     Producer p = new Producer(b);
 8:     Consumer c = new Consumer(b);
 9:   }
10: }
11:
12: // ---------------------
13: // The Attribute class
14: // ---------------------
15: class Attribute {
16:   public int attr;
17:   public Attribute(int attr) {
18:     this.attr = attr;
19:   }
20: }
21:
22: class AttrData extends Attribute {
23:   public int data;
24:   public AttrData(int attr,int data) {
25:     super(attr);
26:     this.data = data;
27:   }
28: }
29:
30: // ---------------------
31: // The Producer class
32: // ---------------------
33: class Producer extends Thread {
34:   static final int COUNT = 6;
35:   private Buffer buffer;
36:
37:   public Producer(Buffer b) {
38:     buffer = b; this.start();
39:   }
40:
41:   public void run() {
42:     for (int i = 0; i != COUNT; i++) {
43:       AttrData ad = new AttrData(i,i*i);
44:       buffer.put(ad);
45:       yield();
46:     }
47:     buffer.halt();
48:   }
49: }
50:
51: // ---------------------
52: // The Consumer class
53: // ---------------------
54: class Consumer extends Thread {
55:   private Buffer buffer;
56:
57:   public Consumer(Buffer b) {
58:     buffer = b;
59:     this.start();
60:   }
61:
62:   public void run() {
63:     int count = 0;
64:     AttrData[] received = new AttrData[10];
65:     try{
66:       while (count != 10){
67:         received[count] = (AttrData)buffer.get();
68:         count++;
69:       }
70:     }catch(HaltException e){};
71:
72:     Verify.assert("count != COUNT",
73:           c.count == Producer.COUNT);
74:     for (int i = 0; i != c.count; i++){
75:       Verify.assert("wrong value received",
76:             c.received[i].attr == i);
77:     }
78:   }
79: }
```

B. 検査仕様の分離

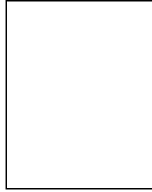
```
 1: aspect VerifyAspect {
 2:   pointcut consumerRun(Consumer c):
 3:     call(public void run()) && target(c);
 4:
 5:   after(Consumer c): consumerRun(c){
 6:     Verify.assert("count != COUNT",
 7:            c.count == Producer.COUNT);
 8:     for (int i = 0; i != c.count; i++){
 9:       Verify.assert("wrong value received",
10:              c.received[i].attr == i);
11:     }
12:   }
13: }
```

(平成 2001 年 9 月 30 日受付)
(平成 2002 年 3 月 19 日採録)



鵜林 尚靖（正会員）

1960 年生．1982 年広島大学理学部数学科卒．1996 年筑波大学大学院経営システム科学専攻修士課程修了．1999 年東京大学大学院総合文化研究科広域科学専攻広域システム科学系博士課程修了．博士 (学術)．1982 年より (株) 東芝に勤務．現在，同社 SI 技術開発センターに所属．ソフトウェア工学，プログラミングモデル/言語に興味をもつ．日本ソフトウェア科学会，電子情報通信学会各会員．



玉井 哲雄（正会員）

1948 年生．1970 年東京大学工学部計数工学科卒業．1972 年同大学院工学系研究科計数工学専攻修士課程修了．同年 (株) 三菱総合研究所入社．1985 年同社人工知能開発室室長．1989 年筑波大学大学院経営システム科学専攻助教授．1994 年東京大学教養学部教授，1996 年東京大学大学院総合文化研究科教授，2000 年同大学院情報学環教授，現在に至る．工学博士．ソフトウェアの仕様技術，検証技術，進化プロセスのモデル化，等の研究及びそれらの技術の実際的な問題への適用に従事．著書に「ソフトウェアのテスト技法」(共立出版) など，訳書に「ソフトウェア博物誌」(トッパン) などがある．日本ソフトウェア科学会，電子情報通信学会，日本オペレーショズリサーチ学会，人工知能学会，ACM, IEEE 各会員．