Shlaer-Mellor 法を利用した統計的テストのための利用モデルの作成法

高木智彦[†] 古川善吾[††] 山崎敏範[††]

統計的テスト法においてソフトウェアの利用方法を特性付ける利用モデルを，Shlaer-Mellor 法を用いて系統的に作成する方法を提案する．統計的テスト法は，ソフトウェアの信頼性を評価するためのテスト手法である．実運用を反映したテストケースを無際限に生成するために，利用モデルが用いられる．これは，ソフトウェアに対して可能なすべての操作を定義するステートマシン（利用の構造）に，遷移確率をはじめとした各種統計量（利用の分布）を関連付けたものである．Shlaer-Mellor 法を用いて利用モデルを作成することには 2 つの利点がある．1 つはソフトウェアの仕様から手続き的に利用の構造を作成することが可能な点である．他方は，実運用または試験運用を行うソフトウェアに対して，利用の分布を導出するための探針を機械的に挿入できる点である．以上によって，少ない手間で正確な利用モデルを作成できるので，より効果的な信頼性評価が可能となる．本手法を在庫管理問題に実際に適用し，利用モデルを半自動的に作成できること，利用モデルを用いたテスト駆動系を容易に作成できることが分かった．同時に，いくつかの課題を抽出できた．

Constructing a Usage Model for Statistical Testing using Shlaer-Mellor Method

Tomohiko Takagi,[†] Zengo Furukawa[††] and Toshinori Yamasaki[††]

This paper shows that a usage model that characterizes software usage on statistical testing is systematically constructed by using Shlaer-Mellor method. Statistical testing is the method for evaluating software reliability. The usage model, which is composed of a usage structure and usage distribution, generates innumerable testcases that reflect actual software usage. The usage model construction with Shlaer-Mellor method brings statistical testing two advantages; a usage structure can be readily developed with software specifications, and the probes that develop usage distribution can be readily inserted into the software in actual or trial operations. This method develops reasonable usage models at low cost and results in effective reliability estimation. From the result of applying this method to the stock control problem, usage models may be semiautomatically constructed, test-driver which executes the tested program with test data is effectively developed, and some problems for effective statistical testing are found.

1. はじめに

統計的テスト法[23)~25)]は，ランダムに生成されるテストケースを用いてソフトウェアの信頼性を評価するテスト手法である．実運用を反映したテストケースを無際限に生成するために，利用モデル（usage model）が用いられる．これは，ソフトウェアに対して可能なすべての操作を定義するステートマシンに，遷移確率をはじめとした各種統計量を関連付けたものである．前者を利用の構造（usage structure），後者を利用の分布（usage distribution）という．利用モデルと同様の概念は，オペレーショナルプロファイル（operational profile）[12),13)]として知られている．統計的テストの目的は欠陥の発見では必ずしもないが，結果として，利用環境において顕在化しやすい（すなわち信頼性に与える影響が相対的に大きい）欠陥を発見することができる．

統計的テストは，入力条件の生成から信頼性の評価に至るまで，利用モデルに基づいて行われるため，利用モデルは重要な成果物である．しかしながら，作成には手間がかかる．たとえば，Guen ら[6)]は，自動車，航空宇宙，通信分野で統計的テストの適用実験を行っ

† 香川大学大学院工学研究科
Graduate School of Engineering, Kagawa University
†† 香川大学工学部
Faculty of Engineering, Kagawa University

た結果，利用モデルの遷移の数が 1,000LOC（Lines Of Codes）あたり 8.6〜346.9 個であったと報告している．特に，大規模な利用モデルをすべて人手によって構築することは，手間だけでなく確度の点からも現実的ではない．

そこで本研究は，利用モデルの構築を特定のソフトウェア分析設計方法論に関連付けることを試みる．一般的に，分析設計方法論にはそれ特有の方法を強制する側面がある．よって，その方法に従う限り，系統的に利用モデルを構築することが可能となるはずである．いい換えれば，開発プロジェクトごとに利用モデルの構築方法を検討する必要がなくなると同時に，開発方法論準拠の CASE（Computer Aided Software Engineering）ツールによって自動化が促進されるため，利用モデル構築の作業品質の安定化や手間の削減が期待できる．

開発方法論はこれまで様々なものが提案され，また実用化されているが，本研究では Shlaer-Mellor 法[8),17),18)] を対象とする．これはオブジェクト指向分析方法論の 1 つであり，ビジネス系や組み込み系など様々な分野において適用可能である．Shlaer-Mellor 法では，まず概念的実体（conceptual entity）をクラスとして定義する．概念的実体とは，現実世界の問題領域における有形物，役割，出来事，相互作用などのことである．振舞いやライフサイクルを持つ概念的実体に対してはさらにステートマシンを定義する．ステートマシンを備えた概念的実体はイベントの送受信を行い，イベント受信によって状態遷移することができる．ステートマシンの各状態は，状態遷移直後に実行される動作を持つ．この動作はプロシージャ（procedure）と呼ばれる．なお，Shlaer-Mellor 法の UML（Unified Modeling Language）[14)] へのマッピングは Executable UML[8)] として知られており，MDA（Model Driven Architecture）[3)] を実現する方法論の 1 つとして注目されている．本研究は，プロシージャの記述に ADFD（Action Data Flow Diagram）ではなくアクション言語（Object Action Language）を用いるなど，Executable UML に従うものとするが，MDA については前提としない．

本研究が Shlaer-Mellor 法に着目した理由は以下の 2 点である．

- Shlaer-Mellor 法は概念的実体のライフサイクルや振舞いをステートマシン図として定義する．利用の構造はこれの延長として手続き的に定義できる．
- Shlaer-Mellor 法は他の分析設計方法論と比べて，設計から実装を導出することに注力している．このことは，利用の構造を半自動で実装し，さらに，利用の分布を記録するための探針をソフトウェアに自動挿入することを可能にする．

本稿の目的は，Shlaer-Mellor 法を用いて利用モデルを系統的に作成できることを示すことである．まず 2 章で，利用モデル構築に関する従来方法の問題点を明らかにする．次に 3 章で利用の構造を作成する方法について，また 4 章で利用の分布を導出する方法について説明する．5 章では，在庫管理システムの問題文を導入して利用モデルの作成事例を紹介する．最後に 6 章で考察を行う．

2. 従来方法の問題

本章では，利用モデル（利用の構造と利用の分布）の構築に関する従来方法の問題点を明らかにする．

利用の構造は，ソフトウェアに対して可能なすべての操作を定義するステートマシンであり，統計的テストの初期の研究では，ユーザの操作を節点（または弧）とする単純なグラフとして提案された[23)〜25)]．その後，UML に基づく表記法[19),21),29)] や，テストの目的に応じて利用モデルの粒度を選択できるように階層化する表記法[22)] などが示された．利用の分布の導出方法については，(a) 運用現場のデータに基づく，(b) 予想あるいは期待される利用法に基づく，(c) 各状態における遷移の発火頻度を一様とすることなどが提案されている[23),24)]．この中で，現実の利用方法に関する特徴を最も正確に反映できるのは (a) であり，可能な限りこの方法を選択するべきであるとされる．運用現場のデータは，運用現場を人手をかけて調査したり，ソフトウェアの実行履歴を取得したりすることによって得られる．前者は，ソフトウェアの運用や業務に関する手続き，取り決め，また，ソフトウェアの運用や業務の過程で発生する情報などを分析する．この方法の問題点は，手間がかかることのほか，必要な情報のすべてを得ることが困難なことである．たとえば，誤操作に関しては，重大な結果を引き起こした場合でもない限り，記録として残らない．あるいはユーザから聞き取りを行うことも可能であるが，客観的で明確な情報を得ることは通常困難である．なぜなら，ユーザは当該ソフトウェアを用いて業務を適切に処理することには高い関心を持っているが，自身のソフトウェアの使い方についてはほとんど関心を持っておらず，普通は意識しないからである．一方，ソフトウェアの実行履歴を取得する方法に関しては，主に以下の 2 つの方法が提案されている．

探針の挿入 探針は実行履歴を記録するためのルーチンである．開発者がプログラムを分析して探針の適切な挿入位置を見つけ，手作業で挿入する．たとえば，Shukla ら[19] は，Java の API（Application Program Interface）に探針を挿入し，メソッドのコールシーケンスを取得する手法を示している．また，類似したものとして，カバレッジ情報を取得するための探針をテストツールによって挿入し，イベントの送受信に対応するステートメントの累積実行回数を得る方法が提案されている[28]．

汎用記録ツールの利用 古川ら[27] は，Windows 上で稼動する SPY++プログラム[9] や X Window System の記録プロトコル[5] など，ユーザの操作を記録するための汎用ツールを利用することを提案している．また Musa[12] は，手間はかかるが記録ツールを自作できると述べている．

上記方法の問題は，利用の構造に対してプログラムコードや実行履歴を人手によって対応付けなければならない点である．たとえば，SPY++プログラムを用いれば,「どの GUI コンポーネントのどの座標でいつ何の入力が発生したか」といった詳細な記録をとることができる．しかし「個々の入力が利用の構造のどの遷移に対応しているか」までは明らかでない．開発者にとってこの対応関係は自明であるかもしれないが，たとえばギガ Byte 規模の実行履歴をすべて人手で分析することは現実的でない．これを自動処理するためには，開発者は対応関係を明示的に定義したうえで，専用の分析ツールを作成する必要がある．しかし，Guen ら[6] は，利用モデルの遷移の数が 1,000LOC あたり 8.6〜346.9 個であったと報告している．先述の作業は開発者にとって過大な負担となる場合があり，結果の正しさについても保証できない．

統計的テストは信頼性を評価するために大量のテストケースを実行しなければならないため，自動化は必須の条件である．しかし，そのためのツールは一般に普及しておらず，独自に用意する必要がある．たとえば，Guen らは統計的テストツール MaTeLo を作成した．これは，利用モデルの編集，テストケースの生成と実行，各種レポート機能を備えるが，本章で示した問題にどの程度対応可能かは明確でない．また，このようなツールを作成するコストは大きいため，可能な限り既存の開発環境で対応できることが望ましい．

3. 利用の構造

本研究が想定する統計的テストの概要を**図 1** に示す．本稿が扱う範囲はプロセス 3〜6 である（プロセス番号は実行順序を意味するものではない）．

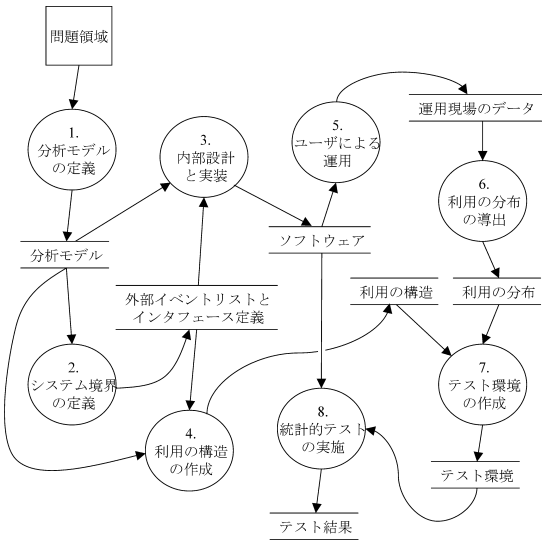

図 1 本研究が想定する統計的テストの概要（DFD による表現）
Fig. 1 Overview of statistical testing of this study (described in DFD).

まず本章において，利用の構造を作成する方法について述べる．

3.1 外部実体のステートマシン図

Shlaer-Mellor 法の視点では，利用の構造は外部実体（external entity）のステートマシンである．外部実体とは，ソフトウェアと何らかの相互作用を行う人や外部システムのことであり，ソフトウェアの仕様である分析モデル（図 1 参照）において，明示的に 1 つ以上定義される．一般的に外部実体はアクタ（actor）とも呼ばれ，ユースケース図やコラボレーション図で人型アイコンとして表す．

利用の構造はステートマシン図によって定義する．ステートマシン図は，主に次の要素から構成される．

状態 外部実体のライフサイクルの段階のことであり，状態番号と状態名でラベル付けした角丸長方形（状態ボックス）として表現する．各状態は独自のプロシージャを持つ．

遷移 外部実体が，ある状態のときにイベントを受信した結果，新しい状態に変化する場合に定義する．遷移元の状態ボックスから遷移先の状態ボックスへの矢印に，イベント名とそのイベントパラメータでラベル付けすることによって表現する．

プロシージャ 状態遷移直後に実行する状態の動作（entry アクション）であり，アクション言語を用いて状態ボックス内に表現する．ソフトウェアへのイベント送信はプロシージャで定義する．

以下は，本研究が提唱する，ステートマシン図としての利用の構造を作成する手順である．

Step 1. 外部実体とソフトウェアの間で交わされるイベントを識別する．外部実体からソフトウェアに送信するイベントは，ソフトウェアに対する外部実体の操作を表すもので，これを外部イベント（external event）と呼ぶ．Shlaer-Mellor 法では外部イベントリスト（図 1 参照）を作成するので，これをそのまま利用する．外部イベントリストは，外部イベント名，外部イベントパラメータ，送信元の外部実体，送信先の概念的実体などを示した一覧表である．一方，ソフトウェアから外部実体に送信するイベントについては，分析モデルですでに定義されるステートマシン図において，送信先に外部実体を指定するイベント生成アクションを検索する．これはアクション言語で「generate 〈 イベント名 〉 to 〈 外部実体名 〉;」と記述されるが，〈 外部実体名 〉には接頭辞「EE_」を付ける決まりになっているので，容易に抽出できる．

Step 2. 外部イベントを手がかりとして状態を識別し，プロシージャを定義する．状態は外部イベントの生成アクションを 1 つ以上含まなければならない．1 つの状態では，途中でソフトウェアからの応答イベントを必要としない限り，複数の外部イベントを連続して生成することができる．また，if 文を用いて複数の外部イベントの中から選択的に生成することができる．その際，if 文の条件式では，イベントパラメータやユーザ定義の変数などのほか，利用の分布を参照するオブジェクト（4.1 節で導入する）を用いて各外部イベントに対する生成条件を厳密に定義する．

Step 3. 各状態に対して，外部実体が受信しうるすべてのイベントを考察して遷移を作成する．すべての状態は再帰可能でなければならない．もし，再帰可能でない（すなわち到達不可能状態や不活性状態が存在する）場合は，自らに対するイベント生成アクションを記述することにより，必要な遷移を追加する．最後に，利用の構造が要求仕様どおりにソフトウェアとイベント送受信できることを確認する必要がある．たとえば，利用の構造における全遷移およびアクションの全分岐について，網羅的に検証を行う．

3.2 CASE ツールによる実装支援

利用の構造は，自動テストのための駆動系として実行可能でなければならない．なぜなら，ソフトウェアの信頼性を評価する（実運用をシミュレーションする）ために，利用の分布を反映したテストケースを大量に生成し，実行する必要があるからである．これをテスト担当者が手作業で行うとした場合，非現実的な作業量を強いられるだけでなく，担当者個人の先入観によってテストケースが偏向する可能性がある．より正確な信頼性を現実的な手数で評価するために，人間の介入を排除して完全にランダムな方法でテストを実行する必要がある．

Shlaer-Mellor 法では通常，短期間で信頼性の高い実装を行うために，ソースコード生成器を備えた CASE ツールを用いてモデルからソースコードを生成する．本稿の手法では，ソフトウェアだけでなく利用の構造についてもこの方法に従って実装する．ステートマシン図としての利用の構造を作成する手順を前節で示したが，この利用の構造から，テスト対象のソフトウェアに特化した自動テストの駆動系を作成することができる．この場合，ソフトウェアの開発環境がそのまま利用できるため，統計的テストの特別なツールを別途作成する必要がなく，コスト削減が期待できる．

ステートマシン図からのソースコード生成には，様々なバリエーションが存在する．ここですべてを紹介することはできないが，一般に認知されている方法として以下の 2 つがある．

ステートパターンの応用 状態をクラスとして実装し（状態クラス），イベントやプロシージャは状態クラスにカプセル化する[26]．

タイプコードと状態テーブル 現在状態と受信イベントの各タイプコードの組合せを用いて，実行すべきプロシージャと遷移先の状態を決定するための状態テーブルを作成する[1]．

プロシージャの生成は，アクション言語に対応した CASE ツールを用いれば可能である．しかし現時点では一般的でないため，ほとんどの場合は人手で実装することになる．利用の構造とソフトウェアの間におけるイベント通信の実装に関しては，インタフェース定義（図 1 参照）に基づく．

4. 利用の分布

本章では，利用の分布を導出する方法を述べる．

4.1 外部イベントの生成特性

利用の分布とは，外部イベントの 2 種類の生成特性のことである．

1 つは，プロシージャが外部イベントを選択的に生成する場合の生成確率である．これは，たとえば「プロシージャ P において，外部イベント E を x% の確率で生成する」「プロシージャ A における外部イベ

ント E_A の生成確率は，プロシージャ B における外部イベント E_B の生成確率の y 倍である」などである．これは連立方程式や数理計画法の制約条件[15] を用いて表現できる．

もう 1 つは，外部イベントパラメータの値の分布である．外部イベントパラメータは，外部実体からの入力データそのものである．これについては，入力定義域上の分布関数として定義したり，また異常値をとりうる場合はその傾向についても調査したりする．ソフトウェアの振舞いに与える影響が大きいものは，パラメータ間の相関関係を分析するなど，より詳細な分析が必要である．

本研究では，上記の外部イベントの生成特性を利用の構造のプロシージャから簡便に参照するために，特別なオブジェクト usage_dist を考案した．その属性の用途ととりうる値を以下に示す．

usage_dist.〈外部イベント名〉 当該プロシージャにおいて外部イベントを生成するかどうかを決定するために，利用の分布に基づいて求められる真偽値を参照する．生成する場合に true，生成しない場合に false の値をとる．

usage_dist.〈外部イベントパラメータ名〉 外部イベントパラメータの値を決定するために，利用の分布に基づいて求められる値を参照する．これは，指定したパラメータのデータ型に合致する値のほか，異常値をとることもある．

4.2 イベントシーケンス

本研究の手法では，利用の分布（外部イベントの生成特性）を導出するために，実運用中または試験運用中のソフトウェアから実行履歴としてイベントシーケンスを取得する．

イベントシーケンスは，ソフトウェアが，外部実体の特定のインスタンス（外部実体インスタンス）との間で行ったイベント送受信の履歴である．これは，イベント名とイベントパラメータの値で構成される．本研究では，イベントシーケンスにおけるイベントの「送信」と「受信」の関係を反転させ，外部実体の履歴と見なす．反転させたイベントシーケンスを用いて，対応する利用の構造をトレースすることによって，その外部実体における外部イベントの生成特性を導出できる．

イベントシーケンス内に外部イベントパラメータとして外部実体の識別子を含む場合は，そのイベントシーケンスに関わった外部実体インスタンスを特定できる．識別子とは，インスタンスを一意に識別するためのクラスの属性のことであり，クラス図において「{I}」というタグが付される．イベントシーケンスに関わった外部実体インスタンスを特定することで，外部実体インスタンスごとに利用の分布を導出できるようになる．

イベントシーケンスには 2 つの特長がある．1 つは，利用の分布の標本として必要な情報のみを含むため，ディスクスペースや処理負荷などのソフトウェアへの影響を少なくできることである．もう 1 つは，利用の構造に直接対応付け可能な形式であるため，機械的な方法で利用の分布を導出できることにある．イベントシーケンスを記録するためには，ソフトウェア中の適切な箇所に探針を挿入する必要がある．探針に関する議論は次節で行う．

4.3 明示的なステートマシン

探針は，イベントシーケンスを記録するためにソフトウェアに挿入されるルーチンである．ソフトウェアが外部実体に対してイベント送受信を行うたびに，そのイベントに関する情報を記録する．したがって，外部実体とのイベント通信に 1 対 1 で対応する基本ブロックに探針を挿入する必要がある．

Shlaer-Mellor 法で作成されるソフトウェアに対して探針を挿入することは容易である．なぜなら，3.2 節でも述べたように，分析モデルのステートマシン図からソースコードを生成するからである．このようにして実装されるプログラムを，本稿では明示的なステートマシン（explicit state machine）と呼ぶ．明示的なステートマシンは以下の原則に基づくため，イベント送受信に対応する基本ブロックの存在が保証される[21]．

- ステートマシン図のすべてのモデル要素（状態，遷移，プロシージャ）が，そのまま，プログラム要素（クラス，メソッド，ステートメントなど）に変換され，1 対 1 で対応する．
- ステートマシン図のすべてのモデル要素が，それらに対応するプログラム要素と，実行順序に関して適合する．

明示的なステートマシンは，生成元のステートマシン図と明確に関連付いているため，探針の挿入はきわめて単純な作業となる．アーキタイプ言語[8] に代表されるような，ソースコード生成規則を定義する仕組みがソースコード生成器に用意されている場合は，自動的に探針を挿入することができる．

探針を挿入した状態でソフトウェアをリリースすることが可能であれば，実際の運用を通してイベントシーケンスを取得できるので，保守やバージョンアップを行う際に利用の分布を導出できる．また，それが不可能である場合や新規にソフトウェアを作成した場

合は，試験運用時に取得する．試験運用では，ユーザがソフトウェアを普段利用する状態に設定して現実の使い方を試すので，実運用に近いイベントシーケンスを取得することが期待できる．特に，長期間試験運用を実施する場合では，実運用との誤差を小さくすることが可能である．試験運用の機会は，通常の開発プロセスにおいて多く得られないかもしれないが，統計的テストは，ソフトウェアの信頼性を評価するために開発プロセスの最終段階で実施するので，それまでに利用の分布が完成すればよい．たとえば，アルファテストやベータテスト，受け入れテストなど，ユーザ参加のテストプロセスにおいてイベントシーケンスを取得しておけば，正式リリース直前や，以降の保守，バージョンアップ時に統計的テストを実施できる．アジャイル開発プロセスでは，従来より早い段階で実行可能なプログラムを作成するので，試験運用を行う機会が得られやすい．

なお，探針は，ソフトウェアに与える影響が許容範囲に収まるように設計しなければならない．処理速度を損なわないためには，個々のイベント送受信に関する記録はバッファに対して行う．そしてソフトウェアの実行が終了した時点で，まとめてファイルに書き出すようにするべきである．また，ディスクスペースに制限がある場合は，使用頻度の高い文字（列）に短いビット列を割り当てることでファイルサイズを圧縮できる．ただし，これらの点に留意したとしても，レスポンスタイムを重視するリアルタイムシステムや，リソースの制限がある組み込みシステムなどでは，探針を利用できない場合がある．

5. ケーススタディ

5.1 受付係システム

本研究では，ケーススタディとして，酒類卸売業者の受付係の仕事を自動化するシステム（受付係システム）を考察することにした．問題文を**図 2** に示す．これは，文献 30) において紹介された在庫管理システムの問題を一部変更したものである．

まず受付係システムの分析モデルを作成した．分析モデルは，ユースケース，ドメインチャート，クラス図，ステートマシン図，および各図に対する説明文などから構成される．完成した分析モデルは，本稿とほぼ同じ体裁で 20 ページを超える文量がある．本稿では利用モデルの作成に関わる部分のみを掲載することにするが，受付係システムとしての概要は，アプリケーションドメインのクラス図（**図 3**）に表れている．ステレオタイプ ≪event≫ が示されるボックスでは，当

ある酒類卸売業者の倉庫では，ビン詰めの酒を積載したコンテナが毎日数個搬入される．1 つのコンテナには，その容量の許す限り複数の銘柄を混載できる．扱い銘柄は約 100 種類ある．倉庫係は，コンテナを受け取ってそのまま倉庫に保管し，積荷票を受付係へ手渡す．また受付係から出庫指示を受けると，在庫品を古いものから出庫することになっている．コンテナの積載品は別のコンテナに詰め替えられたり，別の場所で保管されることはない．空になったコンテナはすぐに搬出される．移送や倉庫保管中に酒類の損失は生じない．

さて，受付係は契約先の小売店から毎日数十件の出庫依頼を受け，その都度倉庫係へ出庫指示書を出すことになっている．出庫依頼は出庫依頼票によるものとし，1 件の依頼では 1 銘柄のみに限られている．在庫量が不足の場合には，その旨依頼者である小売店に連絡する．同時に在庫補充依頼票に記入し，酒造業者に不足品の注文を行う．そして当該品の在庫量が回復した時点で，倉庫係に出庫指示をする．また空になる予定のコンテナを知らせることになっている．

受付係が扱う帳票の内容は次の通りである．

- 積荷票：コンテナ ID，搬入日時，{ 注文番号，品名，数量 } の繰り返し
- 出庫依頼票：注文番号，注文日時，品名，数量，送り先名
- 出庫指示書：注文番号，品名，送り先名，{ コンテナ ID，数量，空コンテナ搬出マーク } の繰り返し
- 在庫補充依頼票：注文番号，注文日時，品名，数量

受付係の仕事を自動化する計算機プログラム（受付係システム）を作成せよ．なお，この課題は現実的でない部分もあるので，入力データのエラー処理などは簡略に扱ってよい．

図 2 酒類卸売業者の受付係問題（文献 30) より引用，一部変更）
Fig. 2 Problem of receptionists of a wholesaler.

該クラスのステートマシンが受信するイベントを列挙している．

本研究ではこの分析モデルに基づき，受付係システムをクライアントサーバプログラムとして実装した．使用した開発環境は，J2SE（Java 2 SDK Standard Edition）[20] と，本研究において試作中の支援システムである．支援システムは，ステートマシン図を編集する機能や，ステートマシン図にステートパターンなどを適用してソースコードを生成する機能を備えている．受付係システムは，支援システムが生成したソースコードを基に，明示的なステートマシンとして実装した．完成した時点でのソースコード規模は約 4,000 行で，クラス数は約 100 である．

5.2 利用の構造の作成

受付係システムの外部実体として，小売店，倉庫係，酒造業者が存在する．本節では，小売店の利用の構造を作成するための方法を示す．他の外部実体についても同様に作成できるのでそれらは省略する．なお，受付係システムに対する小売店の利用形態として，以下を想定した．

想定 小売店は受付係システムを利用するにあたって，小売店ごとに割り当てられた ID を入力してログインする必要がある．ログインに成功すると，小売店は希望の銘柄とその数量を入力する．入力が完了すると依頼内容が酒類卸売業者に送信され，

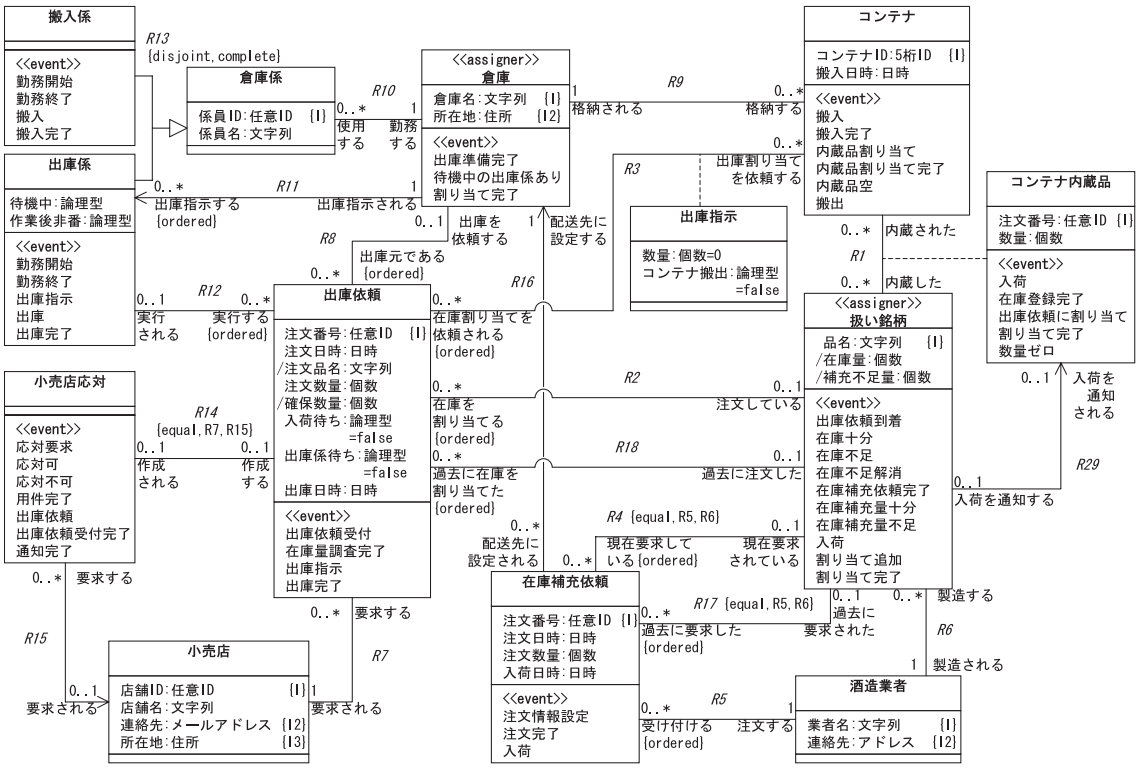

図 3 アプリケーションドメインのクラス図
Fig. 3 Class diagram in the application domain.

依頼内容と納品時期に関する情報がフィードバックされる．出庫依頼は繰り返し行うことができる．用件が完了すれば，小売店はログアウトする．

まず，ステートマシン図としての利用の構造を作成するために，外部実体とソフトウェアの間のイベント通信を明らかにする．外部実体としての小売店は，概念的実体である小売店応対（図 3 参照）とイベント通信を行うことが，分析モデルの構築時においてすでに分かっている．小売店応対のステートマシン図を**図 4** に示す．図中の網掛け部分は，小売店とのイベント送受信を示している．小売店から小売店応対に送信する外部イベントとしては,「応対要求」「出庫依頼」「用件完了」の 3 つがある．これらは，外部イベントリストから明らかである．また，小売店応対から小売店に送信するイベントは,「応対可」「応対不可」「出庫依頼受付完了」の 3 つである．これらについては，イベント送信先に小売店「EE_RetailStore」が指定されているものを検索することによって，すべて機械的に抽出できる．

次に状態を識別し，プロシージャを定義する．ここで，外部イベント「出庫依頼」「用件完了」は小売店がログイン中に選択的に生成できる必要がある．そこで「ログイン中」「ログアウト中」という 2 つの状態を作成して,「ログアウト中」では「応対要求」を生成し,「ログイン中」では「出庫依頼」「用件完了」を usage_dist に基づいて選択的に生成するようにした．

最後に，これらの状態に対し，小売店応対から受信する 3 つのイベントを考察して遷移を追加した．なお,「ログイン中」から「ログアウト中」に遷移するための受信イベントが存在しないため，外部イベント「用件完了」を小売店自身に対しても生成できるようにしている．

完成したステートマシン図としての小売店の利用の構造を**図 5** に示す．これに基づき，Java 言語によって利用の構造を自動テストの駆動系として実装した．ソースコード行数は約 1,000 行である．ソースコードの約 30% は，支援システムによってステートマシン図から生成した．

以上は，筆者 1 人で 10 日間程度の作業であった．

5.3 利用の分布の導出

前述のように，受付係システムは明示的なステートマシンとして実装した．そのソースコードの一部は，支援システムが分析モデルのステートマシン図から生成したが，その際，イベントシーケンスを記録する

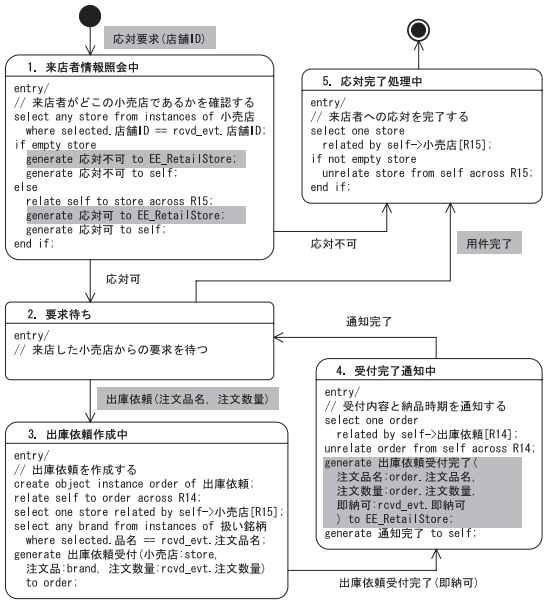

図 4 小売店応対のステートマシン図
Fig. 4 State machine diagram for the serving of a retail store.

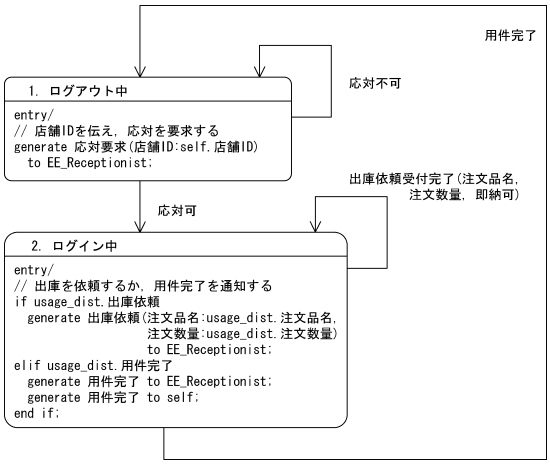

図 5 小売店の利用の構造
Fig. 5 Usage structure for a retail store.

ための探針を挿入した．小売店応対におけるイベントシーケンスの例を**図 6** に示す．このイベントシーケンスは，外部実体インスタンスの識別を可能にする外部イベントパラメータの値「ST001」(小売店の識別子) を含んでいる．したがって，小売店ごとに利用の分布を導出することができる．実運用や試験運用において大量に取得したイベントシーケンスは利用の構造に読み込まれ，オブジェクト usage_dist の属性値を決定するための統計量（すなわち利用の分布）として参照される．

receive 応対要求 (ST001) -> generate 応対可 -> receive 出庫依頼 (大吟醸さぬき,36) -> generate 出庫依頼受付完了 (大吟醸さぬき,36,true) -> receive 出庫依頼 (本醸造やしま,50) -> generate 出庫依頼受付完了 (本醸造やしま,50,false) -> receive 用件完了

図 6 小売店応対におけるイベントシーケンスの例
Fig. 6 Example of an event sequence on the serving of a retail store.

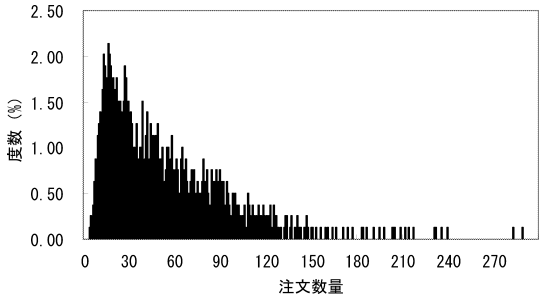

図 7 外部イベントパラメータの分布の例
Fig. 7 Example of distribution of external event parameter.

受付係システムは実験用として作成したため，実際のビジネスの現場における利用の分布を明らかにすることはできない．そこで，筆者らが実運用を想定しつつ受付係システムを実行した．これによって利用の分布の導出を試行し，実際に導出できることを確認した．

本節では例として，小売店の利用の構造（図 5 参照）のための利用の分布を 2 つ示す．

まず 1 つ目として，外部イベントの生成確率について述べる．小売店応対のステートマシン（図 4）の状態「要求待ち」では，外部実体である小売店から，外部イベント「出庫依頼」「用件完了」のいずれかを受信する．仮に，実運用または試験運用において図 6 のイベントシーケンスが得られたとすると,「出庫依頼」を 2 回,「用件完了」を 1 回受信したことが分かる．これは，外部実体である小売店が「出庫依頼」を 2 回,「用件完了」を 1 回送信したということである．この場合，小売店の利用の構造において「usage_dist. 出庫依頼」は約 66% で true に，また「usage_dist. 用件完了」は約 33% で true になる．

2 つ目の例は，外部イベントパラメータの値の分布である．これも，イベントシーケンスから導出できる．たとえば，外部イベント「出庫依頼」には「注文数量」というイベントパラメータが付随している．実運用または試験運用においてイベントシーケンスを取得できれば，**図 7** に示すような，注文数量の値の分布を作成できる．注文数量は 1 以上 300 以下を定義域とする整数型のパラメータであり，定義域内の各値（あるいは

定義域を等間隔で分割したときの各区間）の度数として表現できる．利用の構造の「ログイン中」のプロシージャでは,「出庫依頼」を生成する際に「usage_dist. 注文数量」を参照することによって，図 7 の分布に基づいた値を得る．

5.4 テストの実施

以上により作成した利用モデルを用いれば，実際の利用を反映した入力条件を自動的に生成し，受付係システムに適用できる．テストの実施に関しては本稿の主題ではないが，利用モデルを作成する目的や意義を明確にするとともに，自動テストの駆動系として機能することを示すために，本節で簡単に言及する．

筆者らは別の研究[29] において，回帰テストのためのテスト環境を構築した．統計的テストでは多くの入力条件を用いるので，テスト結果の判定を人手で行うと時間がかかるが，回帰テストの場合，改訂前のソフトウェアの出力を期待出力とすることによって，改訂後のソフトウェアのテストを自動化できる．改訂後のソフトウェアとして，受付係システムのミュータントプログラムを 10 個作成し，これらをそれぞれ 3 分間自動テストした．その結果，203 個のテストケースが実行され，9 個のミュータントプログラムで欠陥を検出することができた．残り 1 個は，テストケースが欠陥を実行しなかったために検出できなかった．しかしながら，実行した欠陥については見逃すことはなかったため，生成した期待出力が機能したことを確認できた．なお，203 個のテストケースのうち，163 個は小売店の利用モデルが生成したものであった．1 つのテストケースは小売店の 1 回分の利用に相当するため，仮に小売店の 1 日あたりの利用回数を n とすると，このテストは $163/n$ 日分の運用と見積もることができる．テスト環境はテストの履歴を記録するので，ソフトウェアの信頼性を自動的に算出できる．

上記の回帰テストは実装済みのソフトウェアに対するものであったが，同様のことは仕様そのものに対しても理論上可能である．アクション言語で記述した仕様は実行可能であり，実際に Project Technology 社製 BridgePoint [16] には，仕様の動的テストを行うための機能が備えられている．

6. 考　　察

この章では，ケーススタディの評価を行い，さらにステートマシンを用いた従来のテスト法と本研究との関連について明らかにする．

6.1 ケーススタディの評価

利用の構造の作成　ステートマシン図としての利用の構造は，3.1 節の手順によって受付係システムの仕様から手続き的に作成できた．そしてこれに基づき，自動テストを行うための駆動系を 10 日程度で作成することができた．駆動系は Java 言語で約 1,000 行のプログラムであり，約 30%はソースコード生成法によって自動的に作成した．

利用の分布の導出　Shlaer-Mellor 法で作成したソフトウェアに対して，機械的に探針を挿入することができた．探針は，ソフトウェアの実運用または試験運用時において，イベントシーケンスを自動的に記録する．このシーケンスを自動的に変換することによって利用の分布を導出した．これは実際のソフトウェアの利用状況（あるいはそれに近い状況）を反映するので，より現実に近い信頼性を求めることができる．

テストの実施　今回のケーススタディでは，回帰テストの実施を例として取り上げた．改訂前のソフトウェアの出力を期待出力とすることで，自動テストが可能になった．実際にテスト環境を構築し，自動回帰テストを試行したところ，10 個の欠陥のうち 1 個は検出できなかった．ここで行ったテストは $163/n$ 日分の運用に相当するので，仮にこの欠陥を修正せずにリリースしたとすると，$163/n$ 日程度は欠陥が顕在化しない可能性があることを示したことになる．テスト環境はテストの履歴を記録するので，信頼性を自動的に算出できる．

6.2 ステートマシンを用いた従来のテスト法

ステートマシンを用いたテスト法は従来から数多く研究されている．

Miller ら[10] は，Z 言語によって形式的に記述したソフトウェア仕様をテストするための方法を提案している．Z 言語では，ステートマシンの遷移は操作の呼び出しに対応する．一方，状態空間は状態変数のとりうる値の集合に対応するため，きわめて大きく，通常人手での操作は困難である．そこで，テスト担当者が任意に選択するテスト手法に基づいて状態空間をパーティションに分割し，抽象化したものをテスト用ステートマシンとして利用する．テストケースはテスト用ステートマシンの全状態または全遷移を網羅するように設計される．また，Murray ら[11] は，Object-Z 言語で記述したソフトウェア仕様からステートマシンを作成し，クラスをテストするためのテストケースを生成する方法を示している．ステートマシンやテストケースの作成方法は先述の Miller らのものと類似している．しかし，Miller らは仕様そのものをテストするた

めに何らかの非形式的仕様からステートマシンを作成するのに対して，Murray らは実装したプログラムをテストするために形式的仕様に基づいてステートマシンを作成するという点で両者は本質的に異なる．

Hartmann ら[7] は，コンポーネントベースシステムの結合テスト法を提案している．個々のコンポーネントの振舞いはステートマシンによって定義されるので，それらを順次結合しながらテストケースを生成，実行する．結合したステートマシン（複数のステートマシンの積）は一般的にプロダクトステートマシン（product state machine）と呼ばれ，他のテスト法でも利用される[2]．テストケースは，単体テストでは少なくとも全遷移を，また結合テストではコンポーネントどうしの通信に対応する遷移を網羅すべきであるとしている．

テストケースを生成する際，ステートマシンの少なくとも全遷移を網羅すべきと主張する論文が多い一方で，全遷移網羅では状態遷移関係の欠陥のすべては検出できないという意見もある．Fujiwara ら[4] は，出力遷移（トリガイベントと出力動作で特徴付けられる）によって各状態を識別するべきとの考えに基づき，Wp 法と呼ばれるテストケース作成方法を提案している．

以上に例示した従来手法に対して，本研究の手法は次の 3 点で特徴的である．

テストの目的 統計的テスト法はソフトウェア信頼性を評価することが目的であるため，テスト打ち切り基準として MTTF（Mean Time To Failure）をはじめとした信頼性尺度が用いられる☆．これに対して，従来のテスト法は欠陥を発見することが目的であるため，「作ったものは少なくとも 1 度はテストすべき」という考えに基づいてカバレッジを評価する．カバレッジの基準は信頼性に対する明確な根拠がないため，統計的テスト法において必須ではない．

期待出力 従来手法の多くは適合性テストに属する．適合性テストは，プログラムが仕様（ステートマシン）どおりに状態遷移することを確認することが主な目的であるため，出力（内部的な振舞い）のアルゴリズムやその実装を対象とするような期待出力を通常用意しない[2]．これに対して本手法は，正確な信頼性を評価するために，状態遷移関係の欠陥に限らず，あらゆる種類の欠陥に対応できる必要がある．この点については前章で，改訂前のソフトウェアの出力を期待出力とすれば解決できることを示した．

状態の概念 従来の手法（特に Z 言語による形式的手法）は状態変数の値の変化に着目してステートマシンを作成するのに対して，本研究はオブジェクトのライフサイクルに着目している．前者と後者は必ずしも等価ではない．前者の方が厳密で形式的であるため，適合性テストには適している．しかし，本手法は入力条件の生成を主な目的としているため，そのような厳密さは必ずしも必要としない．ライフサイクルに着目したステートマシン図は少ない手間で容易に作成できるため，大規模ソフトウェアにも対応できるという利点がある．

7. おわりに

Shlaer-Mellor 法を用いて，利用モデル（利用の構造と利用の分布）を系統的に作成するための手法を提案した．

利用の構造は，Shlaer-Mellor 法によって作成したソフトウェアの仕様から手続き的に作成できる．本研究では，利用の構造から利用の分布を参照するための特別なオブジェクトをアクション言語に導入した．利用の構造は，テスト対象のソフトウェアに特化した自動テストのための駆動系として実装する．その際，Shlaer-Mellor 法準拠の既存の開発環境によって半自動生成が可能であり，コスト削減が期待できる．

一方，利用の分布はプログラムの実行履歴から導出する．Shlaer-Mellor 法では，ソフトウェアを明示的なステートマシンとして実装するという点に本研究は着目した．明示的なステートマシンにはイベントの送受信に対応する基本ブロックが存在するので，イベントシーケンスを記録するための探針を機械的に挿入することが可能である．探針は，実運用時や試験運用時において利用の分布の導出に必要な情報のみを記録するため，ソフトウェアへの影響を少なくできる．取得した大量のイベントシーケンスは，利用の分布として利用の構造から参照可能な形式に自動変換するため，人手による作業を必要としない．このようにして得られた利用の分布は実際の利用状況を反映するため，統計的テストにおいて正確なソフトウェアの信頼性を求めることができる．

さらに，受付係システムのための利用モデルの作成事例を示した．利用モデルが系統的に作成可能であり，また，回帰テストの駆動系として機能することを確認できた．

今後の課題として，以下の点があげられる．

☆ 例外的な統計的テスト法も存在する．Thevenod-Fosse ら[22] は，ソフトウェア全体をまんべんなくテストするために利用の分布を変化させながらテストケースを生成することを提案している．この方法では，利用モデルが利用環境を反映しないため，信頼性を評価できない．

利用の構造用モデルコンパイラの作成 利用の構造用モデルコンパイラは，Executable UML で記述したモデルに基づいて利用の構造を自動生成する CASE ツールである．これは，作成コストを抑えるために必要である．

利用の構造への制約条件の組み込み ここでいう制約条件とは，ある時点においてプログラムが満たすべき普遍的な条件のことである．これを，アクション言語や OCL（Object Constraint Language）[14] で定義し，利用の構造の一部としてソースコード生成できる必要がある．

現場での長期にわたる適用実験 本手法が統計的テストの効果を向上させることを実証するためには，実際の開発現場において長期の適用実験を行う必要がある．そして，統計的テストによる信頼性が利用現場での実際の信頼性と一致するか，あるいは，利用現場からの欠陥報告件数が従来よりも減少することを示さなければならない．

参 考 文 献

1) Beizer, B.: *Software Testing Techniques*, 2nd edition, Van Nostrand Reinhold (1990).
2) Binder, R.V.: *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley (1999).
3) Frankel, D.S.: *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley & Sons (2003).
4) Fujiwara, S., Bochmann, G.V., Khendek, F., Amalou, M. and Ghedamsi, A.: Test Selection Based on Finite State Models, *IEEE Trans. Softw. Eng.*, Vol.17, No.6, pp.591–603 (1991).
5) Gildea, S.: *Record Extension Protocol Specification*, X Consotium (1995).
6) Guen, H.L. and Thelin, T.: Practical Experiences with Statistical Usage Testing, *Proc. 11th Annual International Workshop on Software Technology and Engineering Practice*, pp.87–93 (2003).
7) Hartmann, J., Imoberdorf, C. and Meisinger, M.: UML-based Integration Testing, *Proc. International Symposium on Software Testing and Analysis*, Vol.25, pp.60–70 (2000).
8) Mellor, S.J. and Balcer, M.J.: *Executable UML: A Foundation for Model-Driven Architecture*, Addison-Wesley (2002).
9) Microsoft Co.: Microsoft SPY++ (1997).
10) Miller, T. and Strooper, P.: A Framework and Tool Support for the Systematic Testing of Model-based Specifications, *ACM Trans. Software Engineering and Methodology*, Vol.12, No.4, pp.409–439 (2003).
11) Murray, L., Carrington, D., MacColl, I., McDonald, J. and Strooper, P.: Formal Derivation of Finite State Machines for Class Testing, *Proc. 11th International Conference of Z Users*, pp.42–59 (1998).
12) Musa, J.D.: Operational Profiles in Software Reliability Engineering, *IEEE Software*, Vol.10, No.2, pp.14–32 (1993).
13) Musa, J.D.: The Operational Profile, *Computer and system sciences*, NATO ASI Series F, Vol.154, pp.333–344 (1996).
14) Object Management Group: Unified Modeling Language. http://www.uml.org/
15) Poore, J.H., Walton, G.H. and Whittaker, J.A.: A constraint-based approach to the representation of software usage models, *Information and Software Technology*, Vol.42, No.12, pp.825–833 (2000).
16) Project Technology Co.: BridgePoint. http://www.projtech.com/
17) Shlaer, S. and Mellor, S.J.: *Object-Oriented Systems Analysis: Modeling the World in Data*, Prentice-Hall (1988).
18) Shlaer, S. and Mellor, S.J.: *Object Lifecycles: Modeling the World in States*, Prentice-Hall (1992).
19) Shukla, R., Carrington, D. and Strooper, P.: Systematic Operational Profile Development for Software Components, *Proc. 11th Asia-Pacific Software Engineering Conference*, pp.528–537 (2004).
20) Sun Microsystems Co.: Sun Developer Network: Java 2 Platform Standard Edition. http://java.sun.com/
21) Takagi, T. and Furukawa, Z.: Constructing a Usage Model for Statistical Testing with Source Code Generation Methods, *Proc. 11th Asia-Pacific Software Engineering Conference*, pp.448–454 (2004).
22) Thevenod-Fosse, P. and Waeselynck, H.: STATEMATE Applied to Statistical Software Testing, *Proc. International Symposium on Software Testing and Analysis*, Vol.18, pp.99–109 (1993).
23) Walton, G.H., Poore, J.H. and Trammell, C.J.: Statistical Testing of Software Based on a Usage Model, *Software Practice and Experience*, Vol.25, No.1, pp.97–108 (1995).
24) Whittaker, J.A. and Poore, J.H.: Markov Analysis of Software Specifications, *ACM Trans. Software Engineering and Methodology*, Vol.2, No.1, pp.93–106 (1993).

25) Whittaker, J.A. and Thomason, M.G.: A Markov Chain Model for Statistical Software Testing, *IEEE Trans. Softw. Eng.*, Vol.20, No.10, pp.812–824 (1994).
26) アリジョハル，田中二郎：オブジェクト指向方法論（OMT）に基づく動的モデルからの Java コード生成，情報処理学会論文誌，Vol.39, No.11, pp.3084–3096 (1998).
27) 古川善吾，川野朋彦，伊東栄典：並行動作の統計的テスト法に関する一考察，情報処理学会研究報告，Vol.98, No.20, pp.173–178 (1998).
28) 高木智彦，古川善吾：プログラムの実行履歴を用いた利用モデル作成の一手法，情報科学技術フォーラム一般講演論文集第 1 分冊，pp.209–210 (2003).
29) 高木智彦，古川善吾，山崎敏範：統計的回帰テストのための期待出力の導出方法，情報処理学会研究報告，Vol.2005, No.75, pp.97–102 (2005).
30) 二村良彦，雨宮真人，山崎利治，淵 一博：新しいプログラミング・パラダイムによる共通問題の設計，情報処理，Vol.26, No.5, pp.458–459 (1985).

(平成 17 年 10 月 3 日受付)
(平成 18 年 5 月 9 日採録)

高木 智彦（学生会員）

2002 年香川大学工学部卒業．2004 年同大学院工学研究科修士課程修了．現在，同大学院博士後期課程に在学し，ソフトウェア工学，特にソフトウェアテスト法に興味を持つ．電子情報通信学会会員．

古川 善吾（正会員）

1975 年九州大学工学部卒業．1977 年同大学院工学研究科修士課程修了．同年日立製作所システム開発研究所勤務．1986 年九州大学工学部情報工学科助手，1990 年九州大学工学部講師，1992 年九州大学情報処理教育センター助教授，1998 年香川大学工学部教授，現在に至る．博士（工学）．ソフトウェア工学，特にソフトウェアテスト法，分散システム/インターネットの運用管理等の研究に従事．電子情報通信学会，ソフトウェア科学会，ACM，IEEE-CS，ISOC 各会員．

山崎 敏範（正会員）

1966 年大阪大学工学部卒業．1968 年同大学院工学研究科修士課程修了．同年関西大学工学部電子工学科助手，1970 年香川大学教育学部助手，教授，附属高松中学校長併任，1993 年スイスベルン大学情報科学研究所客員研究員を経て，1998 年香川大学工学部教授，現在に至る．工学博士．教育工学，パターン計測・認識等の研究に従事．電子情報通信学会，教育システム情報学会，物理学会，システム制御情報学会各会員．