ソフトウェアテストシンポジウム 2005
JaSST'05: Japan Symposium on Software Testing 2005

統計的テスト法による回帰テスト自動化に関する考察

高木 智彦† 古川 善吾‡ 山崎 敏範‡

香川大学 〒761-0396 香川県高松市林町 2217-20

E-mail: †s04d454@stmail.eng.kagawa-u.ac.jp, ‡{zengo, yamasaki}@eng.kagawa-u.ac.jp

あらまし 自動回帰テストにおいて，系統的なテスト法とは別のアプローチとして統計的テスト法を導入することを提案する．本稿の手法は，被テストプログラムの前バージョンに対してランダム生成した入力条件を適用することにより，厳密な期待出力を自動的に生成する．特定の運用環境におけるシステムの信頼性推定や系統的テストの補完，ストレステストなどの目的で，状態マシンをもつ種々の粒度のオブジェクトに対して適用できる．

キーワード 統計的テスト，回帰テスト，自動テスト，状態遷移図

Automated Regression Testing using a Statistical Testing Method

Tomohiko TAKAGI† Zengo FURUKAWA‡ Toshinori YAMASAKI‡

Kagawa University 2217-20 Hayashi-cho, Takamatsu City, Kagawa 761-0396, JAPAN

E-mail: †s04d454@stmail.eng.kagawa-u.ac.jp, ‡{zengo, yamasaki}@eng.kagawa-u.ac.jp

Abstract This paper proposes that the statistical testing method, which is a different approach from systematic ones, is applied to automated regression testing. This method inputs random-generated test data into the previous version program of the tested one to generate expected output automatically. It can be applied to the objects that have a state machine, with the purpose of estimating the reliability of system under actual usage, complementing systematic testing and stress testing.

Keyword Statistical Testing, Regression Testing, Automated Testing, State Transition Diagram

1. はじめに

近年，テスト作業の効率化やプログラムの信頼性向上に対する要求が高まっており，xUnit をはじめとしたテスティングフレームワークや記録再生方式の自動テストツールなどが開発現場に浸透しつつある．これらの多くは回帰テストを自動化するものであり，テスト担当者はあらかじめテストケースを設計しておくことによって，プログラムの更新を繰り返す場合のテスト工数を短縮することができる．テストケースは，コードや機能を特定のテスト基準で網羅するための系統的方法に基づいて，一つ一つ手作業で設計されるのが一般的である．

本稿では，回帰テストにおける異なったアプローチとして，ブラックボックステストでありランダムテストの一種である統計的テスト法[1-6]の導入を提案する．統計的テストでは，ユーザの利用の分布を利用モデル（マルコフ連鎖）として定義し，利用モデル上の確率に比例して非決定的にテストケースを生成する．そしてテスト結果はテストモデル（マルコフ連鎖）として要約し，最終的にプログラムの信頼性を評価する．この手法には，テストケースを確率的に際限なく生成できるという，従来の系統的なテストにはない特徴がある．この点を応用することによって，特定の運用環境におけるシステムの信頼性推定や系統的テストの補完，ストレステストなどを行うことができる．特に数学的なアルゴリズムを含まない制御系システムに対して有効であり，ほとんどのテスト工程において導入が可能である．

従来から指摘されているランダムテストの難しさの一つに，入力条件に対する期待出力の導出方法がある．すなわち，入力条件をランダムに生成できたとしても，それに対応する厳密な期待出力を自動的に導出できない，という指摘である[7]．本研究においては，前バージョンのプログラムに対してランダム生成した入力条件を適用し，そこから得られた出力を期待出力とする．この方法は，前バージョンを基準とした出力結果の相違しか検出できないという制限があるが，その反面，現実的なコストの範囲で大量のテストケース設計を行うことが期待できる．

本稿では，まず 2 節で本手法の目的やテスト対象，手順について示す．特に手順については，仕様書の更新の有無に基づいて 2 通りの方法を紹介する．そして 3 節では，2 節で示した方法に関して，回帰テストに適用可能な一般論を含めた考察を行う．

2. 方法

2.1. 概要

本手法は，統計的テスト法を用いて回帰テストを自動化する．これにより，開発者に依らない視点からプログラムをテストする．ランダムに生成された入力条件に対応する厳密な期待出力を生成するために，被テストプログラムの前バージョン（基準プログラム）を用いる．

テスト終了条件は「基準プログラムと同程度の信頼性を達成する」ことである．したがって，安定した基準プログラムが存在しなければならない．例えば，次のような工程に対するテストとして導入できる：

- リファクタリング，チューニング
- コンパイラによる高度の最適化
- 他の実行環境への移植，複数の実行環境への対応
- バージョンアップ

本手法の目的として以下のものがある：

- 特定の運用環境におけるシステムの信頼性推定
- 従来の系統的テストの補完
- ストレステスト

テスト対象は，状態マシンをもつ種々の粒度のオブジェクト（システム，サブシステム，コンポーネントなど）であり，テストに先立って状態遷移図によりモデル化される必要がある．

本テスト手順は主に 2 種類に分けることができる；すなわち，被テストプログラムが基準プログラムと共通の仕様書に基づいている場合と，被テストプログラムの仕様書が基準プログラムのものから更新されている場合である．前者の方が実現が容易であるが，適用範囲は限定される．各手順の詳細は，それぞれ 2.2 節と 2.3 節に後述する．基準プログラムを用意できない場合の議論は 3.2 節で行う．また，被テストプログラムや基準プログラムに対応する仕様書を用意できない場合は，本手法の適用範囲外である．この問題を解決する方法については 3.3 節を参照されたい．

なお，具体的なテスト環境の実現方法は，個々の開発環境や開発組織に依存するところが大きく，また，後の議論を一般的なものとするためにも言及しない．

2.2. 共通の仕様書に基づく場合の手順

被テストプログラムが基準プログラムと共通の仕様書に基づいている場合，本手法は以下の 5 つのステップから構成される．図 1 に成果物の関連を示す．

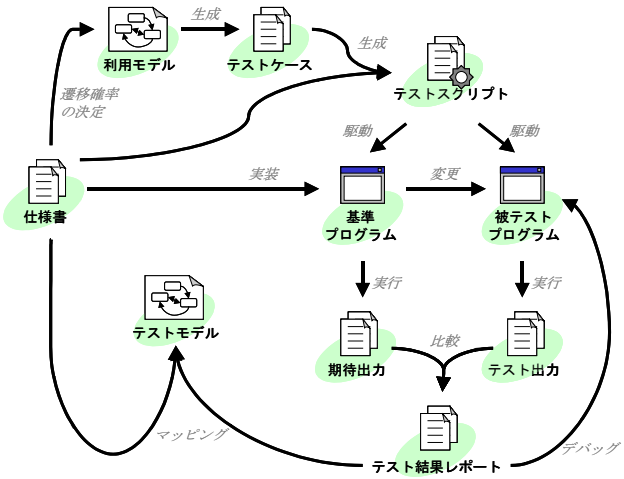

図 1. 共通の仕様書に基づく場合の成果物の関連

Step 1. 利用モデルの作成

利用モデルとは，仕様化工程で作成された状態遷移図上に遷移確率を記入したものである．遷移確率は次に例示する方法によって決定することができる：

A) 利用現場のデータに基づく方法[3,8-10]. プログラムのログや業務の生データ，人手による作業手順などから遷移確率を算出する．

B) 開発者の推定（期待）に基づく方法[3].「悪意がなく操作に習熟したユーザならどう使うか？」を考える．

C) 各状態における出力遷移に一様な遷移確率を割り当てる方法[3]. 利用モデルのエントロピーが最大になる．

D) プレテストの結果から各遷移の誤り確率を推定し，コストや予算から最適なテスト配分を決定する方法[6].

E) プロファイラを用いてプログラムのボトルネックを明らかにし，負荷が大きくなるように遷移確率を割り当てる方法．

F) 状態や遷移に関連する何らかのメトリックスに応じて遷移確率を割り当てる方法．

本来，統計的テストはプログラムの信頼性を推定するところに主な目的がある．この限りにおいては上記 A が最良の方法であるが，他の目的や様々な状況に対応するためには，複数の選択肢を用意しておく必要がある．

利用モデルの作成コストを抑え，再利用を促進するためには，すべての遷移確率を数理計画法における制約条件に簡約する[5]. 例えば，「遷移 a の遷移確率 P_a は遷移 b の遷移確率 P_b の 2 倍である」ことを制約条件として表現すると，$P_a - 2P_b = 0$ となる．

Step 2. テストスクリプトの生成

利用モデルに従ってテストケースを生成する．ここでのテストケースとは，状態遷移図上のパスとそれに対応する具体的な入力条件のことである．この段階で厳密な期待出力は含まれていない．

次に，テストケースと仕様書（状態遷移図に限定しない）を用いて，被テストプログラムと基準プログラムを駆動するためのスクリプト（テストスクリプト）を生成する．仕様書は，各プログラムを駆動するためのインタフェースや基本検証項目，例えば OCL（Object Constraint Language）[11]に関する情報などを提供する．

Step 3. 期待出力とテスト出力の生成

厳密な期待出力を導出するために，テストスクリプトを基準プログラムに適用する．テストスクリプトには基本検証項目が組み込まれているので，期待出力にバグが混入するのをある程度は防ぐことができる．そして，被テストプログラムにもテストスクリプトを適用し，テスト出力を記録する．

プログラムは，テストに役立つ内部的なデータを出力するように制御できることが望ましい．

Step 4. テスト結果レポートの生成

テスト出力と期待出力を比較し，テスト結果レポートを生成する．また，テストスクリプトの基本検証項目が検出したエラーを記録する．

テスト出力と期待出力の不一致は必ずしもプログラムのバグを意味しない．バグ以外のあらかじめ想定できる不一致については，その条件を明示することによって無視することも可能になる．もし不一致が検出されたなら，必要に応じて人手で原因を究明し，テスト結果レポートに記録する．不一致の原因としては以下のものが存在する：

- 新しいバグの発生．基準プログラムには存在しなかったバグが，被テストプログラムに作り込まれた．本テストの目的はこれを発見することにある．
- 以前のバグの修正．基準プログラムのバグが被テストプログラムにおいて修正された．
- 実行環境の動的条件の差異．例えば，テストを実行した時刻や並行動作するタスク，通信トラフィックなど．一方，マシンのハードウェア構成や OS などの静的条件については，Step 3 で揃えることができれば問題にならない．
- 仕様の変化．影響がテストスクリプトにまで及ぶ場合は，2.3 節の方法に従う必要がある．

Step 5. テストモデルの生成と分析

最後に，テスト結果レポートを状態遷移図に記入することによってテストモデルを生成し，信頼性を評価する．基準プログラムが既に十分な信頼性を実現している場合においては，被テストプログラムそのものの信頼性として評価し，出荷基準に達したか否かを判断することができる．もし，基準プログラムが十分な信頼性を達成していなければ，基準プログラムに加えた変更に対するテスト十分性の指標とする．本テストは，目標とする信頼性を達成した時点で終了する．

テストモデルは，ソフトウェア信頼性モデルにおけるデータ領域モデル[12]の一種であり，マルコフ連鎖の理論[2,13]を適用することによって信頼性を求められる．通常はステップ数（状態遷移回数）に基づいた尺度として表されるが，以下の方法によれば実時間で表現することもできる：

- 被テストプログラムに探針を挿入することによって，状態毎，遷移毎の平均実行時間を計測し，テストモデルを重み付けする方法．
- テスト開始時刻，エラー発生時刻，エラー回復時刻，テスト終了時刻から計算する方法．

テストモデルにおける信頼性の評価精度に関する議論は 3.6 節を参照されたい．

2.3. 仕様書が更新された場合の手順

被テストプログラムの仕様書が基準プログラムのものから更新されている場合，作業内容は複雑になる．図 2 は成果物の関連を表している．以下にステップ毎の詳細を述べる．

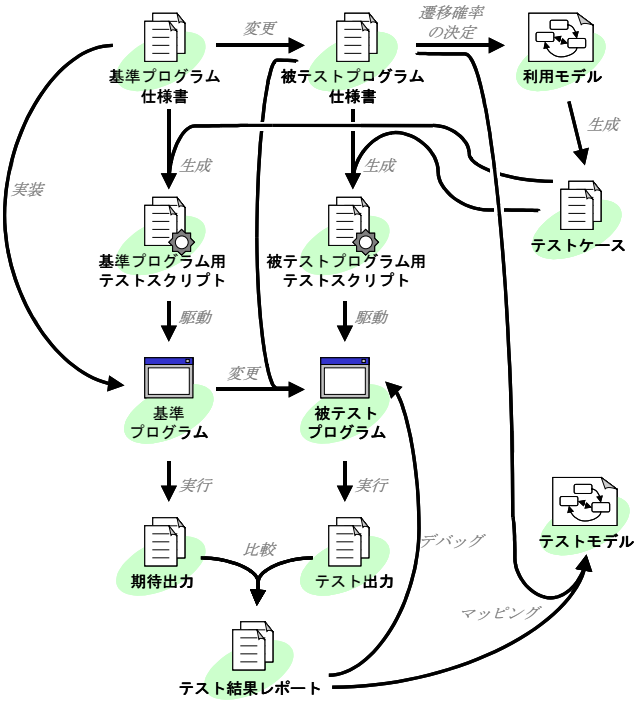

図 2. 仕様書が更新された場合の成果物の関連

Step 1. 利用モデルの作成

被テストプログラムの仕様書から利用モデルを作成する．遷移確率を決定するには，2.2 節の Step 1 で紹介した方法を用いる．基準プログラムの利用モデルが存在する場合は，その制約条件の一部を再利用することによって手間を省くことができる．ただし，仕様の更新が制約条件に与える影響について十分検討しなければならない．

Step 2. テストスクリプトの生成

被テストプログラムの利用モデルに従ってテストケースを生成する．次に，基準プログラムを駆動するためのテストスクリプトと被テストプログラムを駆動するためのテストスクリプトを生成する．これらは各仕様書および生成したテストケースに基づく．

ここで問題となるのは，基準プログラム用テストスクリプトの生成における，テストケースと基準プログラム仕様書の整合性である．被テストプログラムの機能集合を F_{test}，基準プログラムの機能集合を $F_{standard}$ とすると，$F_{test} \cap F_{standard}$ についてはテストケースの実行ルーチンを生成し，二つのプログラムでの実行結果によって不一致を発見できる．しかし，$F_{test} - F_{standard}$ については，基準プログラムでのテストケースの実行が不可能である．そのため，被テストプログラムをテストケースによって駆動した結果を通常のテストと同様に判定する必要がある．また，$F_{standard} - F_{test}$ は実行をスキップするか，あるいは直近の $F_{test} \cap F_{standard}$ に接続するための最低限の処理を行うルーチンを生成する．

Step 3. 期待出力とテスト出力の生成

厳密な期待出力を導出するために，基準プログラム用テストスクリプトを基準プログラムに適用する．$F_{test} - F_{standard}$ における期待出力は取得できない．

さらに，被テストプログラムの出力を得るために，被テストプログラム用テストスクリプトを適用する．

Step 4. テスト結果レポートの生成

$F_{test} \cap F_{standard}$ におけるテスト出力と期待出力を比較し，不一致を検出する．そして不一致の原因がエラーであれば，テスト結果レポートに記録する．さらに，テストスクリプトの基本検証項目が F_{test} において検出したエラーを記録する．

Step 5. テストモデルの生成と分析

被テストプログラムの状態遷移図にテスト結果レポートを記入し，テストモデルを生成，分析する．

3. 考察

3.1. 系統的なテストとの比較

統計的テストはランダムテストの一種である．系統的なテストとランダムテストの優劣に関しては従来から議論が行われているが，意見の一致するところではない[7]. 本研究は, 双方の手法の特徴を理解した上で, 回帰テストに両方を導入することを提案する. これは, 以下に示すように互いが補完の関係にあるためである：

- カバレッジ

  系統的なテストの基本方針は，プログラムを詳細に分析することによって，可能な限り少量のテストケースで高いカバレッジを達成することである．残存バグの多いテスト工程初期においては効果があり，少量のテストケースで網羅的にバグを洗い出すことができる．これは，多様なユーザが利用するプログラムにおいて一定の信頼性を確保するのに有効である．ただし，高い信頼性を達成するために厳しいテスト基準を用いるとテストケース数が大幅に増加するので，コスト的な限界がある．

  一方，ランダムテストでは，コードや機能を網羅できる保証はない．特に統計的テストにおいては，利用モデルのエントロピーが小さいほど，テストケースに片寄りが生じやすい（これは，利用モデルにおいて平均訪問時間[13]が大きい機能ほど高いカバレッジが期待できる，ともいえる）．ただし，テストケースを確率的に際限なく生成できるという特徴は，系統的なテストにはない．この点に着目することによって，特定の運用環境における信頼性推定や系統的テストの補完，ストレステストなどを行うことができる．

- テストケースのバグ発見能力

  系統的なテストでは，過去の経験や知識に基づいてバグを効果的に検出するためのテストケースを設計できる反面，設計者の思い込みによってテストが偏向する可能性がある．

  一方，ランダムテストは系統的なテストに比べてバグ発見能力は一般的に劣る．その原因として，テストケースが重複したり，バグが潜在する可能性の低い機能をテストしたりすることが考えられる．しかし，設計者には無駄や意味がないと思われるテストケースが思いがけずバグを検出することがある．

3.2. 基準プログラムが用意できない場合の問題

基準プログラムが用意できない場合においては，次に示す 2 つの問題が生じる．

まず 1 つ目は,厳密な期待出力を導出できないため,被テストプログラムの出力を厳密に検証することが困難な点である．ただし，テストスクリプトが備える基本検証項目との矛盾やプログラムのクラッシュ，誤り状態への遷移などは検出できる．

2 つ目の問題は，運用環境における信頼性推定を指向した利用モデルの作成が困難な点である．基準プログラムが存在しないということは，運用経験が全くないことを意味する．したがって，開発時にユーザの利用の分布を正しく求めることは困難である．この点に関して文献[8,9]では，統計的テスト実施前にプログラムをユーザが試用することによって，要求仕様の確認を行うと同時に利用モデルのための運用データを取得することを提案している．運用データは，スケルトンコードジェネレータを応用することによって仕様書に直接マッピング可能な形式で取得することができる．また，Shukla ら[10]は API（Application Program Interface）の実行シーケンスを用いてコンポーネントレベルの利用モデルを構築する方法を提案している．

3.3. MDA との関連

MDA（Model Driven Architecture）[14]は，すべての開発工程を仕様書に基づいて進めるための広範な技術体系であり,本研究では以下の点で重要な意味がある:

- 仕様書

  本研究ではテストスクリプトやテストケースを仕様書に基づいて生成するので，被テストプログラムや基準プログラムに対応する最新かつ詳細な仕様書が用意できなければならない．しかし現実の開発プロジェクトにおいては，実装段階で生じた変更が仕様書に反映されないことがある．この問題を解決するためには，仕様書とソースコードの間で同期を取る仕組みが必要である．MDA において PSM（Platform Specific Model）から完全なソースコードを生成する場合は，このような問題は発生しない．

- テストの目的

  PSM は基本的にプログラムと同程度の情報をもっているので，仕様書を正確に反映したソースコードを生成できる．そのため，プログラマが直接ソースコードを編集しない限り，仕様書との不一致を発見するためのテストは必要ない．一方，システムの信頼性を評価するためのテストやストレステストについては実際に実行可能なプログラムに対して行う必要があり，重要性は失われない．

3.4. 仕様書の検証

本稿のテスト手法は仕様書に基づいたものであり，仕様書そのものの検証（仕様書がユーザの要求を正確に反映していることの検証)を行うことは困難である．1 つの方法としては，利用モデルから生成されたシーケンス（状態遷移図上の実行系列）をユーザが一つ一つ確認することによって要求と仕様の不一致を発見することが考えられる[15]．ユーザの要求は本来曖昧なものであり，また状態遷移図も実際の利用場面を想像するための素材としては十分なものではないが，表現形式をシーケンスに変換することで理解が容易になることが期待できる．この場合，$F_{test} - F_{standard}$ のみに着目すれば，ユーザが検証すべきシーケンスの数を抑えることができる．

3.5. テスト範囲の絞り込み

テストの予算が限られている場合，テスト範囲を絞り込む必要に迫られることがある．本手法は，理論的には，コードの変更箇所に限定してテストを実行し，信頼性を評価することができる．例えば，コードの変更箇所に対応する状態や遷移を 1 回以上通過する利用モデルを作成すればよい．ただし，プログラムスライス等を用いて変更による影響範囲を正確に特定し，仕様書にマーキングする必要がある．この作業は自動化が困難であるため，手作業によるコストと確度が問題となる可能性がある．

3.6. 信頼性の評価精度

信頼性は，プログラムにおけるバグと利用の分布によって決まる．したがって，信頼性の評価精度は以下の 2 点に依存する：

- テストのバグ認識精度

  テストのバグ認識精度とは「プログラムの出力に潜在するバグの結果や兆候をテストが発見する能力の度合い」のことであり，テスト結果の検証機能やテストケースの確度に依存する．本手法は，基準プログラムから期待出力を生成するため，テストケースの確度に限界がある．例えば，基準プログラムのバグをそのまま被テストプログラムが継承している場合，そのバグは隠れる可能性が高い．ただし，基準プログラムに多数のユーザによる長期間の運用実績がある場合は，基準プログラムの信頼性に対して（テストケースの確度に対して）確証を得ることができる．

- 利用モデルの精度

  利用モデルの精度とは「実運用におけるユーザの利用の分布を利用モデルが正確に反映している

度合い」のことである．利用モデルの精度と信頼性の評価精度には相関があるが，利用モデルの精度が高いほど多くのバグが発見できるわけではない．

4. おわりに

回帰テストにおいて，系統的テストとは別のアプローチとして統計的テスト法を導入することを提案した．本手法は，安定した基準プログラムに対してランダム生成した入力条件を適用することにより，厳密な期待出力を自動的に生成する．特定の運用環境におけるシステムの信頼性推定や系統的テストの補完，ストレステストなどの目的で，状態マシンをもつ種々の粒度のオブジェクトに対して適用できる．

仕様書は，被テストプログラムや基準プログラムをテストケースに基づいて駆動するための条件が記述されている必要がある．本手法の限界は，被テストプログラムが基準プログラムよりも高い信頼性を達成することが困難な点である．これは，期待出力が基準プログラムから生成されるためである．しかし，統計的テストには，テストケースを確率的に際限なく生成できるという,従来の系統的なテストにはない特徴がある．それゆえ，従来方法だけでは見落とす可能性のあるバグが，本手法によって効果的に検出されることが期待できる．

今後の研究においては，本稿の内容に基づいたテスト環境を構築し，パイロットプロジェクトに適用することによって有効性の評価を行う予定である．

文　献

[1] J. A. Whittaker and J. H. Poore, “Markov Analysis of Software Specifications”, ACM Transactions on Software Engineering and Methodology, Vol.2, No.1, pp.93-106, January 1993.

[2] J. A. Whittaker and M. G. Thomason, “A Markov Chain Model for Statistical Software Testing”, IEEE Transactions on Software Engineering, Vol.20, No.10, pp.812-824, October 1994.

[3] G. H. Walton, J. H. Poore, and C. J. Trammell, “Statistical Testing of Software Based on a Usage Model”, Software Practice and Experience, Vol.25, No.1, pp.97-108, January 1995.

[4] D. P. Kelly and R. S. Oshana, “Improving software quality using statistical testing techniques”, Information and Software Technology, Vol.42, No.12, pp.801-807, 2000.

[5] J. H. Poore, G. H. Walton and J. A. Whittaker, “A constraint-based approach to the representation of software usage models”, Information and Software Technology, Vol.42, No.12, pp.825-833, 2000.

[6] K. Sayre and J. H. Poore, “Partition testing with usage models”, Information and Software Technology, Vol.42, No.12, pp.845-850, 2000.

[7] B. Beizer, “ソフトウェアテスト技法”, 日経 BP 出版センター, 1994.

[8] 高木智彦, 古川善吾, “ソースコード生成による利用モデルの作成と分析”, ソフトウェアテストシンポジウム 2004 予稿集, pp.44-49.

[9] T. Takagi and Z. Furukawa, “Constructing a Usage Model for Statistical Testing with Source Code Generation Methods”, In Proceedings of 11th Asia-Pacific Software Engineering Conference, pp.448-454, 2004.

[10] R. Shukla, D. Carrington and P. Strooper, “Systematic Operational Profile Development for Software Components”, In Proceedings of 11th Asia-Pacific Software Engineering Conference, pp.528-537, 2004.

[11] “OMG Unified Modeling Language Specification”, Object Management Group, March 2003, http://www.uml.org/.

[12] 山田茂, “ソフトウェア信頼性モデル”, 日科技連出版社, 実践ソフトウェア開発工学シリーズ 12, 1994.

[13] 渡部隆一, “マルコフ・チェーン”, 共立出版, 数学ワンポイント双書 31, 1979.

[14] D. S. Frankel, “MDA モデル駆動アーキテクチャ”, SiB access, 2003.

[15] 高木智彦, 古川善吾, “UML 状態図を用いたテストケース作成支援システムの試作”, 情報処理学会研究報告, Vol.2002, No.23, pp.79-86, 2002.