修 士 論 文

# ”例示によるプログラミング手法”を用いた GUIテストケーススクリプト自動生成

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

筒井 圭一朗

2003 年 3 月

修 士 論 文

# ”例示によるプログラミング手法”を用いた GUIテストケーススクリプト自動生成


指導教官　権藤克彦 助教授

審査委員主査　権藤克彦 助教授
審査委員　片山卓也 教授
審査委員　敷田幹文 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

110082 筒井 圭一朗



提出年月: 2003 年 2 月

## 概 要

本研究では、例示プログラミング手法を応用したGUIテストケース自動生成法を提案する。GUIアプリケーションのテストケース生成は非常にコストが高く、テストケース自動生成ツールへの期待が非常に高まっている。

既存のテストケース自動生成手法は、扱いが簡単だが単純なテストケースしか作成できなかったり、カバレッジの高いテストケースが生成できるが特定の操作のテストが困難である等の問題点を抱えている。

例示プログラミングは、ユーザのGUI操作から一連のパターンを認識し、自動的にプログラム化する手法である。例示プログラミングでは、ユーザはアプリケーションを操作するだけで、システムがその操作シーケンスから特定のパターンを認識し、推論機構を用いて一般化を行う。このようにして、ユーザが意識しなくとも定型操作パターンのプログラムが生成され、ユーザビリティを向上することができる。

本研究では、GUIのテストケース生成に例示プログラミングを応用することで、テスト担当者のテストケース生成の労力を抑えたまま、特定の操作のテストケースを生成し、かつカバレッジの高いテストケースを生成することを提案する。

GUI操作の構造的な特徴は、大きく分けると、同一ウィンドウ内でのイベント発生と他ウィンドウへの移動イベント発生の2種類になる。

本研究のシステムは、テスト担当者が対象のアプリケーション上で行った、GUIの操作シーケンスから操作の構造的な特徴を認識する。

そこで、テスト担当者の操作シーケンスからそれら2種類のグループ別にパターン抽出を行い、ヒューリスティックな推論を適用することで、可読性と再利用性の向上を目的としたテストケーススクリプトに変換させた。また、抽出したパターンにゆらぎテンプレートを適用し一定のゆらぎを与えることでテストケース作成者が行った操作に関連するテストパターンを補完しテストカバレッジの高いテストケーススクリプトを生成させた。

生成効果は部分的なものであったが、テスト担当者は通常のテストオペレーションを行うだけで、可動性と再利用性のある一般化されたものと、高いテストカバレッジをもつテストケースを生成できる自動生成器がコスト削減を支援できると考えることができる。

また、生成されたテストケースの実行環境を実装するとともに、GUIの「見た目」の不具合を検出できるGUI自動テストシステムの構築についての検討と実行環境構築のガイドラインを示す。

# 目 次

# 図 目 次

# 表 目 次

# 第1章　はじめに

## 1.1　背景

今日のGUIソフトウェアの普及とともに、GUIの複雑化、大規模化が進んでおり、GUIのテスト工程のコストは莫大なものとなっている。そのため、GUIテスト自動化ツールへの期待は非常に高い。

多くのGUI自動テストツールは、テストケースの実行時間短縮に成果を上げているが、テストケースの自動生成に関しては、依然として研究の余地がある[3, 4, 5, 7]。

その理由として、GUIの特徴そのものがテストケース生成を行いにくいものであることが挙げられる[20, 3, 4, 5, 6, 7, 9, 16]。イベントドリブン、GUIのオブジェクトの多くの静的/動的な属性、そして無限の操作パターンの組み合わせ、これらの特徴がテストケースの量を莫大なものとし、生成するテストパターンの有用性の判断を困難にしているからである。

既存のテストケース自動生成法は、3つのカテゴリに大別できる。

1. Capture/Replay手法[13, 15, 23]

   テスト担当者が対象のGUIアプリケーションを操作した操作シーケンスそのものをテストケースとして出力するものである。この手法は、シンプルで強力ではあるが、実際の操作したパターンしか出力しないため、テストカバレッジを達成するためには、多くの操作を必要とする。

2. 組み合わせを用いた手法[19, 17, 21, 22]

   組み合わせを用いて操作パターンを網羅的に生成する手法である。ポピュラーなものとしては、重要な操作ポイントを記述したテキストを基に可能性のある全ての操作パターンを生成するものである。この手法の問題点は、そのテキストの記述に時間がかかることと重要な操作ポイントを指定し損なう可能性があることである。その他の手法としては、初期状態の画面と操作後の画面を指定すると有限ステートマシンを用いて、2画面間における全ての操作のパターンを生成するというものである。しかし、2画面間で発生する操作を全て生成するために、テストケースの生成量が莫大なものとなってしまい、テストの実行にコストがかかるという問題点がある。また、特定の機能テストを行いたい場合には、適用しにくいという面もある。

3. Planningを行う手法[3, 4, 7]

   人間が指示した生成プラン(ルール)に基づいてテストケースを自動生成するもので

ある。プランはあらかじめ人間がプランニングツールを用いて詳細に指示する必要があり、その指示方法も簡単とは言えず、テスト作成者の学習時間負荷が高くなりがちな面がある。

3 つの手法とも決定的なものとは言えず、現状では次の 3 通りの方法が GUI テスト自動化用のテストケースを生成する主なものとなっている。

1. テストケーススクリプトを手動で記述する
2. 操作イベントをキャプチャして、それをテストケースとする
3. キャプチャしたテストケースをスクリプトに再編集する

多くの場合、単純な Capture/Replay 手法による生成と手動による作成の併用によって行われており、その作業コストは莫大なものとなっている。

近年、例示プログラミング (Programming By Demonstration, PBD) の様々な研究成果が発表され注目されている [1, 2, 26, 31, 18]。

例示プログラミングは、ユーザの GUI 操作から一連のパターンを認識し、自動的にプログラム化する手法である。例示プログラミングでは、ユーザがアプリケーションを操作するだけで、システムがその操作シーケンスから特定のパターンを認識し、推論機構を用いて一般化を行う。このようにして、ユーザが意識しなくとも定型操作パターンのプログラムが生成され、ユーザビリティを向上することができる。

この手法を応用することで、3 つのテストケース自動生成手法の長所を生かしつつ、欠点を補うことが可能になる。それは、テスト担当者が対象のアプリケーションを操作するだけで、システムがその操作パターンを一般化したり、その操作に関するパターンを組み合わせてテストカバレッジの高いテストケースを生成することができることを意味する。

## 1.2 研究の目的

本研究の目的は、テスト担当者の GUI 操作からその構造パターンを自動認識し、汎化推論や関連操作の組み合わせによって、GUI テストケースを生成するアルゴリズムの提示、および GUI テストケース自動生成器への応用とその効果について評価を行うことである。そこで、新しい生成法を実現する手段として、例示プログラミング手法に注目した。

例示プログラミング手法は、ユーザの GUI 操作から一連のパターンを認識し、人間の意図を推論することで、自動的にプログラム化を行う。本研究では、テスト担当者の操作シーケンスから GUI の構造特徴を認識抽出し、推論を行うことで、2 つの目的を持つテストケーススクリプトを生成する。

- 一般化テストケーススクリプトの生成
  単純な時系列でしかない操作シーケンスから、一般的な構造を推論し、テスト担

当者が編集しやすいテストケースを生成する。これにより、テストケース生成器が対応しきれないパターンを生成したり、ループピング処理コードを追記して負荷テストに利用したりといったことが容易になる。

- ゆらぎテストケーススクリプトの生成
  テスト担当者の操作パターンを代表的な操作パターンと見なし、その操作シーケンスにテンプレートを適用することで、一連のテストができるテストケースを生成する。これにより、ある機能操作の操作バリエーションによる不具合の検出が可能になる。

このようにして、テストケース生成コストを抑えたまま、再利用性のあるテストケーススクリプトとテストカバレッジの高いテストケーススクリプトを自動生成することを目指す。最終的に、これらのアルゴリズムを組み込んだ自動生成器の実装、評価を行う。また、生成されたテストケースの実行環境を実装するとともに、GUI の「見た目」の不具合を検出できる GUI 自動テストシステムの構築についての検討と実行環境構築のガイドラインを示す。

## 1.3 アプローチ

例示プログラミングを GUI テストケース自動生成に応用するためには、まずユーザ (テスト担当者) の GUI 操作シーケンスから、テストケースを生成するのに有用な特徴となるパターンを認識し抽出する必要がある。

本システムでは、GUI が機能的に階層的な設計をされていることと、ユーザはその機能的階層上の機能を用いて操作を行っていること、そしてテスト担当者も機能的階層を意識してテストケースを作成していることに注目し、同一画面内での操作シーケンスと画面遷移を伴う操作シーケンスをテストケース生成の特徴とした。

そして、それらの特徴を利用して 2 つの目的を持つテストケーススクリプトを生成する。

**一般化テストケーススクリプト生成**: パターンの種別や出現頻度から汎化推論を行い、制御構造化やサブルーチン化そしてコメント追加をしたスクリプトを生成する。本システムでは、可読性や再利用性といった観点においてヒューリスティックな推論によって汎化を行う。

**ゆらぎテストケーススクリプト生成**: ユーザの GUI 操作シーケンスをある機能操作の代表的な操作パターンと見なし、そのパターンに対してゆらぎテンプレートを適用してスクリプト生成を行う。テンプレートには、ある機能操作の操作バリエーションによる不具合をテストする際に、テストケース作成者がよく行うテスト操作プランを記述してある。

テストケース自動生成器の処理の流れは以下の通り。

1. ユーザ (テスト担当者) の GUI 操作シーケンスのキャプチャリング
2. 特徴抽出
3. 汎化推論およびゆらぎテンプレート適用
4. テストケーススクリプト生成

## 1.4 成果

提案した例示プログラミングを GUI テストケース自動生成を実装し、評価実験用アプリケーションにおいてテストケーススクリプト生成実験を行った。実験結果より以下に示す効果を得ることができた。

- 一般化テストケーススクリプト生成実験
  サブルーチン化による機能呼び出しの影響範囲の明確化やループ化による繰り返し部分の明確化といった可読性と再利用性が向上した。
- ゆらぎテストケーススクリプト生成実験
  元の操作シーケンスからゆらぎテンプレートのテスト操作プランに則ったスクリプトが生成され、元の操作シーケンスの 1.5 から 3 倍のイベントが発生された。

このことから、テスト対象のアプリケーションを実際に操作するだけで、生成システムが自動的に可読性を高めた再利用性のあるテストケーススクリプトを生成したり、高いカバレッジを達成しつつ有用なテストケーススクリプトを優先的に生成できる生成器がコスト削減を支援可能と考える。

しかし、いくつかの問題点があることも判明した。以下にそれらを示す。

- 粒度の細かいサブルーチンが生成されやすく、多くのサブルーチン呼び出しが発生する。
- 推論ルールに合致しないループ化が行われないため、サブルーチン呼び出しの羅列発生により可読性が下がるときがある。
- テストカバレッジの評価が困難で、ゆらぎテンプレートの有効度が十分に検証できない。

また、GUI の「見た目」の不具合を検出するためのテストケース実行環境構築のガイドラインの検討を行った。そして、それに基づいて操作イベント毎のウィンドウのスクリーンショットをテストログ出力できるテストケース実行環境を実装し、実際に不具合を検出できることを確認できた。

本研究を通して、汎化推論の改良の検討点を得るだけでなく GUI テストオラクルの設定法やテストロギングタイミングの問題などの GUI テスト自動化ツールを構築する上で考慮すべき点を整理するといった有益な議論ができた。

## 1.5 本論文の構成

本論文の構成は以下の通り。

**第 1 章** 本研究の背景と目的として、GUI テストにおいてテストケース作成コストが高いことを示し、その解決案として、例示プログラミングを用いたテストケースの自動生成法を提案し、研究のアプローチと成果の概略を示した。

**第 2 章** GUI テストの現状と問題点、および GUI テストカバレッジについて述べ、GUI テスト自動化の概要と利点、および自動化可能な GUI テスト種類についてまとめた後、GUI テストを自動化するにあたっての現実的な問題点である、テストケース生成コストについて述べる。

**第 3 章** GUI テストケース自動生成に求められる条件を述べた後、既存の手法の紹介とそれぞれの手法についての検討をする。どのような生成器が有効であるかを検討した結果である、テスト対象のアプリケーションを実際に操作するだけで、生成システムが自動的に可読性を高めた再利用性のあるテストケースを生成したり、高いカバレッジを達成しつつ有用なテストケースを優先的に生成できる生成器が有効であるという結論を述べる

**第 4 章** 本研究の応用技術である例示プログラミング手法について述べる。応用している。例示プログラミングは、プログラムの振る舞いや効果の例を与えることによってプログラムを生成するアプローチの 1 つであることを述べる。

**第 5 章** テストケース作成者の GUI 操作シーケンスを基に汎化推論とゆらぎテンプレート適用によって、一般化テストケーススクリプトおよびゆらぎテストケーススクリプトを生成する GUI テストケース自動生成器の提案とその実現について述べる。

**第 6 章** 自動生成したテストケースを実行するための環境の設計とその実装について述べる。本研究のテストケース実行環境は、テストケーススクリプトの処理系、および GUI の「見た目」の状態遷移をテストするために必要なテストオラクルの取得と比較の機能を含んでいる。

**第 7 章** 航空旅客券を予約するための GUI アプリケーションを作成してテスト対象とし、いくつかのテストケースからテストケーススクリプトを生成させた結果を示す。

**第 8 章** 評価実験の結果や本研究を通じて得ることができた情報について議論し、例示プログラミングが GUI テストケース自動生成器に有用であることを示す。そして、特徴抽出法の発展性や、実装した自動生成したテストケースが GUI の「見た目」の状態遷移をテストできることについて述べる。

**第 9 章** 本論文についてのまとめを行い、結論として、例示プログラミング手法を GUI のテストケーススクリプト自動生成に応用することについて部分的ではあるが有用性であることを述べ、最後に将来への課題をまとめる。

# 第2章　GUIテストと自動化

本章では、まずはじめに、GUI テストの概要を理解するために、GUI テストの現状と問題点、および GUI テストカバレッジについて述べる。その後、GUI テスト自動化の概要と利点、および自動化可能な GUI テスト種類についてまとめた後、GUI テストを自動化するにあたっての現実的な問題点である、テストケース生成コストについて述べる。

## 2.1　GUIテストの現状

GUI を持つソフトウェアの普及により、開発者は、ユーザからソフトウェアの複雑性を隠しつつ、ユーザビリティを提供できるようになった。また、様々な GUI ツールキットや開発フレームワークの登場で、開発者からも複雑性を隠すことが可能となり、開発者はインターフェイスシステムの設計を始めから行う必要性は薄れてきている。

しかし、GUI が一般的なインターフェイスとなることで、テスト担当者の心配事は大きくなってきている。「きちんとボタンは押せるのか？」「このボタンは押せないべきなのでは？」「設定通りにウィンドウが描画されるのか？」「フォントは正しいのか？サイズは？色は？」そして、これらのテスト操作を一体何回繰り返せば良いのだろうか？と。

従来のインターフェイスは、ユーザにモーダル[1] な操作体系を押し付けてきた。そのため、操作性の柔軟性はそれほど高くはなかった。しかし、そのことは開発者やテスト担当者がユーザの操作手順を予想しやすいことを意味する。そして、様々なテスト技法が開発され、ソフトウェアの品質保証に有効性を示してきた [8, 10]。

だが、GUI では必ずしも従来のテスト技法が有効とは限らない [20, 3, 4, 5, 6, 7, 9, 16]。それは、GUI がモードレス[2] な操作体系を基本としているために、柔軟な操作性が、その操作手順の組み合わせの総数を莫大なものとしてしまっているからである。

それらの莫大な操作手順にパステストを適用するには、重要な操作手順に絞ったり、メニューマップ[3] で図解分析することである程度対応できるが、それでも人海戦術に依存している面が少なくない。

[1] モードとは、システムが持っている状態のことを指す。モーダルは、その状態をユーザに示すことである。特にユーザインターフェイスでは、ある機能に依存した操作しかユーザに許さない状態を指す。

[2] モーダルとは反対に、ユーザがモードに縛られず自由な操作ができる。

[3] どのメニューでどの選択肢を選ぶとどこに遷移するか図解したもの。ウィンドウの状態遷移を机上で辿ったり、見通しのよいテスト設計ができる。[10]

## 2.2 GUIテストの困難さ

GUIテストでは、GUI発明以前の従来のテスト技術を適用しにくいことが多い[4]。その理由は、GUIが持つ性質によるところが大きい。

以下に、GUIテストを困難にしているいくつかのGUI独特の問題を示す。

### 2.2.1 イベントドリブン [20]

現在のGUIのほとんどの実装には、イベントドリブンが用いられている。イベントドリブン式GUIでは、ユーザに非常に自由な選択肢を提供しているため、アプリケーションのどの時点においても、フォーカスのあるウィンドウ内のすべてのウィジット[5]を押下できるし、同じアプリケーションの他のウィンドウをアクティブにして操作を行うこともできる。また、複数のアプリケーションが動作可能なOS上では、そのウィンドウは他のアプリケーションが所有していることもある。

ユーザの選択肢が自由ということは、アプリケーションのコードがあらゆるタイミングにおいても、次のイベントを処理しなければならないことを意味している。もしも、全くハンドリングされていないイベントがあれば、アプリケーションは全く何もしないか、あるいはクラッシュする可能性もある。

たとえ全てのイベントがハンドリングされていたとしても、その呼び出し順はユーザに依存するため、ハンドラのコード間での不具合がいつ起きるのかを予想することは不可能である。例えば、ボタン押下イベントの多くは、アプリケーションのあるウィジットのフォーカスを全く関連のないウィジットに移動させることがある。そのような場合、前のオブジェクトのハンドラに不具合があったとすると、次のウィジットには直接の影響を与えないものの、潜在的な問題を引きずったままアプリケーションは実行を続け、最終的に大きな不具合を引き起こす。

このように、ユーザの選択肢が多いことは、アプリケーション内のウィジット間の潜在的な繋がりが複雑に絡み合っていることを示し、それらを1つ1つテストすることの困難さは容易に想像できる。

---

[4] しかし、インターフェイスと関連が低いビジネスロジックやアプリケーションロジックに関しては、従来のテスト技術は非常に有効である。代表的な技術には、パステスト、構文テスト、ユニットテスト等がある。

[5] ウィジットとは、GUIのユーザインターフェイスの各要素のことを指す。ボタンやテキスト入力フィールド、スクロールバーだけでなく、ダイアログボックスやウィンドウ自体がウィジットの場合もある。本論文では、ダイアログボックスやウィンドウは、単に「ウィンドウ」と呼んでいる。Unix系OSの用語であり、Microsoft Windows© においてはコントロールと呼ばれる。

### 2.2.2 GUI のオブジェクトの多くの静的/動的な属性

GUI の各構成要素 (デスクトップ、ウィンドウ、ボタン、そして他のグラフィカルなウィジット) は、ほとんどの場合、あるクラスのオブジェクトになっている。そして、それらのオブジェクトは、なんらかの属性値を持っている。以下に典型的な属性例を示す [20]。

- アクティブ属性: ユーザからのイベントを受け取るかどうかの情報
- 外観属性: キャプション、フォントタイプ、フォントサイズ、位置、大きさ、表示/非表示
- 値属性: テキストストリング、ラジオボタンオン/オフ、真/偽、数値、リストボックスやコンボボックスの各要素

これらの属性は、静的に定義されるものと動的に定義されるものに分けられる。静的な属性は、アプリケーションがビルドアップされる毎に、1 回テスト[6] すれば良いが、動的な属性は、その値が変更される度にウィンドウに反映しているかどうかを確認しなければならない。しかも、1 つのオブジェクトにつき、数個から十数個の属性がある。

平均 5 個の属性を持つオブジェクトを 50 個使用しているアプリケーションの場合、テスト項目数は少なくとも 250 項目であり、そのうち動的な属性に関しては、それぞれ複数のテスト項目を用意しなければならない。さらに、それ以上の属性を持つオブジェクトを使用しているアプリケーションの場合、そのテスト項目数は、格段に増える。

### 2.2.3 無限の入力ドメイン [20]

GUI アプリケーションでは、ユーザはポインティングデバイスを用いて、フォーカスのあるウィンドウ内のすべてのウィジットを任意のタイミングで選択できる。

例えば、「氏名」「住所」「電話番号」等のテキスト入力フィールドウィジットを 10 個持つ住所録ウィンドウがあるとする。それらのタブオーダと呼ばれるデフォルトの入力順が定義されているが、GUI では、その入力順はユーザが自由に変更してもよいため、[7] その入力順の総数は、$10! = 3628800$ 通り である。

あるフィールドを入力中に、ユーザが他のフィールドの入力ミスに気づいて再入力する場合を考慮すると、その総数は無限となる。テキスト入力フィールドに限らず、他のウィジットでも同様に無限の入力ドメインの問題を持っている。

---

[6] 1 回と言っても、属性が文字列の場合、その用法やスペルが正しいのか？フォントタイプは統一されているか？といったことに注意する必要がある。他の属性についてもそれぞれ注意すべき点がある。

[7] タブオーダが固定されているアプリケーションも存在する。このようなアプリケーションは、一見グラフィカルなデザインであっても、厳密には GUI と区別されるべきである。

### 2.2.4 多くの入り口と出口 [20]

GUI のイベントドリブンの性質により、アプリケーション内のあるコード群 (クラスメソッド、関数、コールバック関数、ハンドラ等) が呼び出されるタイミングは複数存在する。この事は、機能レベルでも同様である。

ある機能を呼び出すために、そのボタンを押下する方法以外にも、メニュー呼び出し、ショートカット呼び出し[8] というような複数の呼び出し法が挙げられる。もしも、設計した GUI がそのような複数の呼び出し法を提供している場合、各機能毎に各呼び出し法のテストをしなければならない。

同様に、機能から抜ける方法もいくつか考えることができる。ある機能を使用中に、「OK」「キャンセル」「閉じる」等のボタン押下で機能を終了するだけでなく、突然メニュー呼び出しやショートカット呼び出しで他の機能へ移行したり、他のウィンドウへ移動してしまうこともある。

このように機能への入り口と出口は 1 つではないため、テストを行う際にはさまざまなシチュエーションを考慮しなければならない。

### 2.2.5 GUI テストカバレッジ問題 [6]

GUI テストを行う際にテスト担当者が悩ましく思うことの 1 つに、「この GUI のテスト作業はどの程度完了したのだろうか？」というものがある。

GUI のテストでは、従来のテストカバレッジを用いて、テスト作業の進捗度を測る目安を設定することが困難である [6]。その理由として以下が挙げられる。

- ホワイトボックステストができないことがある

  構文カバレッジ、分岐カバレッジ、パスカバレッジ等のテストカバレッジを用いるテスト技法では、プログラムのコードを必要とする。しかし、ウィジットの多くは、コンパイルされライブラリに格納されているコードのインスタンスとして、ウィンドウにレイアウトされている。

  そのためプログラムのコードは、テストカバレッジの指標として常に利用できるわけではなく、従来のテストカバレッジを GUI の指標とすることは、必ずしも妥当性を主張できるとは言い難い。[9]

- GUI への入力値は、イベントシーケンスである

  ユーザが GUI を操作することで発生するイベントシーケンスは、ユーザ毎に異なり、しかもユーザは機能間を好きなタイミングで自由に移動可能である。そのため、操作シーケンスの数は莫大になってしまう。

  このようなことから、ある程度固定されたシーケンスや入力値を想定しているト

[8] ファンクションキー押下やキーボードショートカットのように深い階層の機能を直接呼び出す。

[9] プログラム全体の品質を評価する際の議論である。ライブラリ内のプログラムの品質を信用できると仮定すれば、アプリケーション部分のコードに対しては従来のテストカバレッジで評価できる。

ランザクションフローテスト、データフローテスト、ドメインテストそして、遷移テストによるテストカバレッジを用いるのは、困難であると考える。

- GUI はユーザの指示によって、その動作を終了する
  正しく設計された GUI では、ユーザの指示が無い限りは、その動作を自動的に終了させることはない。ユーザの操作が正しいときはもちろん、仕様外の操作を行ったとしても、ユーザに指示を仰がずにプログラムを終了させるべきではない。　このことは、GUI はユーザの指示が無い限りは、その動作を無限に続けることを意味する。　分岐カバレッジ、パスカバレッジでは、全ての分岐とパスをクリアしたとしても、100 回目に通過するの分岐やパスで不具合がないとは言い切れない。

### 2.2.6　テストオラクル

テストするということは、テストケースを実行するだけでなく、その実行結果が正しいのかを評価することである。結果が正しいことを評価しなければ意味をなさない。

評価を行う際には、そのテストケースを実行したことによるアプリケーションの正しい動作が、あらかじめわかっていなくてはならない。その「正しい動作」のことをテストオラクル、または単にオラクルと呼ぶ[10] 。アプリケーションを $f()$、テストケースを $T$ としたとき、$a = f(T)$ で表される $a$ の正しさを評価する $A$ がテストオラクルであり、$a = A$ となるとき、アプリケーションはそのテストを通過したことになる。

テストオラクルは、比較できるものならばどんなものでもよい。例えば、コンパイラプログラムをテストするならば、テストケースはいくつかのサンプルソースコードであり、テストオラクルは正しい動作をするバイナリコードである。

回帰テストをする場合、図 2.1 の変更前のアプリケーションの動作がテストオラクルと見ることが出来る。この場合、コード変更がない機能に関しては、オリジナルの機能とまったく同じ動作をしなくてはならない。もし、動作が異なる場合には、他の部分のコード変更による波及不具合が発生していることを示す。

GUI の場合、このテストオラクルを設定することが難しい。GUI では、「見た目」の状態遷移が大切である。「見た目」の状態遷移とは、ウィンドウ内で起きる描画の流れであり、ユーザが操作を行ってはじめて決定されるものである。ユーザ操作が行われるたびに、「見た目」が正しいのかという判断をする情報はどのように記録すべきだろうか？GUI の画面を写真として取るのか？デジタルデータとしてストレージに保存するのか？いずれにしても、「見た目」の状態遷移を確実にチェックするためには、あらゆる操作シーケンスに対応した描画毎の情報が必要となりその総量は莫大なものとなる。

[10] 本論文では、テストオラクルを用いる。

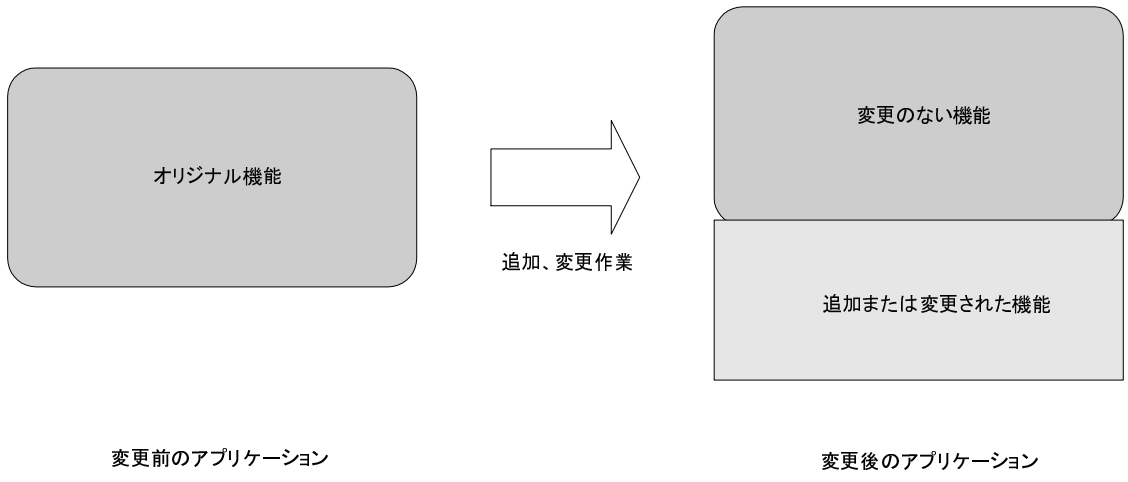


図 2.1: 回帰テスト時のテストオラクル

## 2.3 GUI テストの自動化

2.2 節で示した GUI テストの困難さにより、その作業コスト特に人件費は非常に高い。そこで、GUI テストの自動化によってそのコスト削減を目指そうという動きがある。以下では、GUI テスト自動化の現状と利点について述べる。

### 2.3.1 GUI テスト自動化の現状

GUI に関する研究分野の中で、GUI テスト自動化は現在でも研究の余地がある [4]。
それは GUI の性質により解決すべき難点がいくつも存在するためである。
それらのいくつかを以下に挙げる。

- テスト設計時：テスト項目選出法、テストケース生成法、テストオラクル生成法
- テスト実行時：実行優先度決定法、実行結果の記録法
- テスト評価時：テストオラクルとの比較法、無効なテストケース排除法

これらは人間が行うにしても経験と知識を必要とする難しい問題である。そのため、それぞれをコンピュータで自動的に行わせ、さらに人間がテストすることと同等かそれ以上の精度を持たせることは、高度な人工知能を構築することと同一であり、非常に困難な問題であると言える。

現在、GUI テスト自動化ツールと呼ばれているものは、ツールで半自動的にまたは手動で記述したテストケースを自動実行させる形を取っており、回帰テストや負荷テスト等で大きな効果をあげている。ただし、テストケースの生成コストに関しては依然として困難な問題がある。これについては 2.5 節で後述する。

### 2.3.2 GUI テスト自動化の利点

GUI テスト自動化の利点は非常に高い。GUI に限らないことも含め、自動化による利点を以下に挙げる。

- 機能的回帰テストのコスト削減
  現在の GUI テスト自動化ツールの多くは機能的なテスト項目の回帰テスト作業のコストを削減することを目的にしている。このことは、手動での回帰テストのコストが非常に高いことを示している。
  GUI の場合、機能追加や変更時に設計すべきテストケースの数が膨大になる。変更されていない機能に関する膨大なテストケースを自動的に実行できることのコスト削減率は非常に大きい。

- 負荷テストのコスト削減
  自動化により、コストを抑えたまま特定のルーチンや機能を長時間繰り返しテストさせることが可能になる。
  また、自動化ツールを仮想的なテスト担当者と見なせるため、WEB サービスにアクセスする何千人の仮想ユーザを生成させ、実際の運用に近いテストを最小限のコストで行うことができる。

- 大規模なテスト実行が可能
  回帰テスト、負荷テストやその他にテストの種類に限らず、自動化の利点を最も表していると言える。
  特に回帰テストでは、手動の場合、ビルドアップ毎に過去の全てのテストケースを実行するコストは、非常に高く、重要度の高いものだけを行うのが普通である。
  しかし、過去のテストケースが自動化されているのであれば、ビルドアップ毎にそれらを全て実行することも可能である。この利点は品質保証の面で非常に優れていると言える。

- ハイブリッドなテスト実行
  比較的定型的なテストケースに関しては自動的に行い、人間の判断が必要なクリティカルなテストケースに複数のテスト担当者を適用する。このようにすることで、重要度の高い部分に経験の豊富なテスト担当者の作業時間を割り当てることが可能になる。

- テストケース資産の効果的運用
  アプリケーションのコードを流用して開発する際には、この利点は非常に高い。特に GUI の場合、ユーザビリティを考慮しバージョンアップによる極端なレイアウトの変更は行われないことが多い。そのためコードを流用しなくとも操作仕様が同

じ部分はテストケースをそのまま資産運用できる可能性がある[11] 。

- テスト進捗状況の数値化が容易
  テスト評価が自動化されることで、その出力データからカバレッジ達成率を計算できる。

- テスト全体の 20 %を自動化することで、80 %以上の時間当たりの作業効率向上を期待できる[12]
  テスト作業のネックとなる部分、例えば時間の要するテスト項目や繰り返し頻度の高いテスト項目等は全体の 20 %程度であることが多い。それらを自動化することで、大幅な効果を期待できる。

## 2.4 GUI テスト自動化が可能なテストの種類

GUI テストの自動化による利点が高いことは、2.3.2 節で示したが、あらゆるテストが実行できるわけではない。特に物理的な作業が必要なテストについては、自動化は非常に困難である。
実際に自動化されているテストの種類を以下に挙げる。

### 2.4.1 チェックリストテスト

チェックリストは、下位レベルのコンポーネントの単純なチェックに有効なテスト法である。この方法によりテストされるテスト項目の種類を以下に示す。

- GUI は標準的な機能をカバーしているか？
  - ウィンドウのサイズ、位置、タイプ (モーダル/モードレス)
  - 標準的なボタン機能 (クローズ、最小化、最大化等)
- GUI の水準/慣例は守られているか？
  - 標準的なボタン (OK、キャンセル、コンティニュ)、外観、色、サイズ、配置
  - ボタンやウィジットの統一性のある利用
  - オブジェクトのキャプションに標準的もしくは一貫した表現の名前がついているか？

[11] 設定したテストオラクルは、必ずしも資産運用できるとは限らない。これは、開発環境や開発アプリケーションに依存する。
[12] 20-80 の法則:「結果の 80 %は、わずか 20 %の原因がもたらしている」。科学的な根拠はないが、統計学の見地では一般的な法則である。

これらは、テストケースを作成するのも、テストオラクルを作成するのも容易である。さらに、それぞれのウィンドウ単体でテスト可能であり、結合テストや機能テストを行う前にテスト可能である。

### 2.4.2 ナビゲーションテスト

ナビゲーションテストは、GUI の「見た目」の状態遷移を結合テストするブラックボックステストの 1 つである。このテスト方法は、ウィンドウの呼び出しをプログラム化しやすく、自動化に非常に親和性がある。

新規ウィンドウを作成し、それをアプリケーションに結合するには、そのウィンドウをアプリケーションから呼び出すためのメニューや他のウィンドウから呼び出されるボタンなどを作成しなければならない。

そのウィンドウに関するテストケース作成の作業は以下が挙げられる。

- ウィンドウへの (アプリケーションの動作に必要な) 正規の呼び出しを全て選出し、それぞれの呼び出しのためのテストケース作成
- ウィンドウからの他の機能への正規の呼び出しを全て選出し、それぞれの呼び出しのためのテストケース作成
- (呼び出し側である) ウィンドウへの戻り呼び出し (例：呼び出されたウィンドウを閉じる等) を全て選出し、それぞれの戻り呼び出しのためのテストケース作成

さらに、メニュー、ボタン、ショットカット等の複数の呼び出し方法がある場合、それぞれで利用可能な手順ごとに 1 つのテストケースを作成する必要もある。

### 2.4.3 同期テスト

同期テストは、2 つ以上のウィンドウや機能間の依存関係をテストするものである。

例えば、ウィンドウ A、B が開いている状態で、ウィンドウ A であるデータを変更すると、ウィンドウ B の表示も変更される場合を依存関係があるという。同様に、1 つのウィンドウ内で、機能的に別グループのウィジット間で、あるウィジットの動作が他のウィジットに変化を発生する場合も依存関係である。

このテストへのテストケース生成は、ナビゲーションテストのそれと類似している。基本的には、全ての依存関係を選出し、それぞれに有効なテストケースを生成するというものである。

依存関係が選出されていれば、あるウィジットを操作によって作用する各ウィンドウが特定されるので、プログラム化することにより自動化が可能となる。

### 2.4.4 負荷テスト

アプリケーションの性質によっては、数日間から数週間か、それ以上の長期間にわたり動作するように保証しなければならない。そのような場合、実際に長期間の統合テストを行い、可能性のある不具合を調査したり、動作保証を行うのが負荷テストである。

負荷テストは、一連の操作を長期間繰り返し行ったり、多数のユーザによる操作を想定したテストを行う。このようなテストケースは、自動化の対コストパフォーマンスが高く、テスト法のその性質からも自動化を行いやすい。

### 2.4.5 機能テスト

このテストを行う GUI テスト自動化ツールは多数存在している。

機能テストとは、ユーザの操作を自動的に記録、検証、再生することで、、アプリケーションが期待通りに動作するかどうかをテストするものである。

要件の正確な収集、テスト計画の作成、スケジューリングと実行、結果の分析、不具合の管理といったテストの主要な各段階における開発達成度、そしてテストカバレッジの達成度から、アプリケーションの完成度を測ることが可能である。

ユーザの操作を自動的に記録するだけでなく、手動で記述したテストケーススクリプトを実行できるツールが多数存在していることからも、自動化との親和性の高さを示していると言える。

## 2.5 GUI テスト自動化時におけるテストケース生成のコスト問題

GUI テスト自動化時のテストケース生成コスト問題について述べる前に、ある企業での自動化の成功例を示す。

以下は、あるシステム開発企業での成功例である。

- 適用形態：
  C++で構築した顧客サービスアプリケーションに適用
- 導入コスト：
  テストケース作成に約 30 時間 (手動に比べて 2,3 倍)
- 実行コスト：
  全てのテストケース実行に約 90 分 (手動に比べて 1/10〜1/20)
- 保守コスト：
  4 半期毎のアプリケーションのビルドアップテストに伴うテストケース保守作業

- 適用結果：
  導入から約 1 年で導入コストおよび保守コストを回収

1 年という短期間で各コストを回収することができる例は、実際にはそれほど多くない。多くの企業では、テストの自動化に投資したコストを回収できずにいる。

この例の場合、比較的小規模の開発形態であり、契約先の一定の顧客向けのアプリケーションであったことが成功に結びついた要因とできる。だが、より一般的な PC 向け GUI アプリケーションや組み込み機器の GUI 等では、想定されるユーザが想定される操作を行うとは限らず、様々な事象を想定するテストケースを作成する必要がある。

現在の GUI テスト自動化では、以下の 3 つが主なテストケースの生成法となる。

1. スクリプトを手動で記述する
   GUI テスト自動化ツールには普通テストケース記述言語処理系を搭載しており、それを用いてテストケースをプログラム化できる。この場合、テストケース自身に不具合が含まれる危険性が非常に高い。

2. 操作イベントをキャプチャして、それをスクリプトとする
   GUI テスト自動化ツールの Capture/Replay 機能を用いて、GUI 操作をその通りに記録し再生してテストケースをプログラム化できる。ただし、出力されるスクリプトは、単なるイベントの羅列であるため、再利用性は乏しい。そのため、3. のように編集を行うのが普通である。

3. キャプチャしたスクリプトを編集する
   2. で出力されたスクリプトから必要な情報のみを取り出して手動で再編集したり、テストカバレッジの高いテストケースに再編集する。

しかし、近年の GUI の大規模化、そして 2.2 節での理由により、作成しなければならないテストケースの総数は莫大に膨れ上がり、テストケース生成コストが自動化による利点を相殺してしまう。

例えば、複数の画面で構成される GUI アプリケーションのテストケースの生成を行う場合を考えてみる。3 つのイベントをハンドリングする必要がウィジットが、1 画面当たり 10 個レイアウトされているとすると、全ウィジットの押下順を並べ替えて操作するシーケンスをテストするためには、最大で ${}_{30}P_{10} = 109027350431999$ だけのテストケースが必要になる。画面間の遷移を考慮すると、問題はさらに大きくなる。

3 章では、その生成コストを効果的に削減するテストケース自動生成手法について検討する。

# 第3章　GUIのテストケース自動生成法の検討

本章では、GUI テストケースの自動生成法について検討する。

GUI のためのテストケースを自動生成法は様々なものが提案されているが、それぞれ長所短所がある。本章の初めに GUI テストケース自動生成に求められる条件を述べた後、既存の手法の紹介とそれぞれの手法についての検討をしている。

そして、どのような生成器が有効であるかを検討した結果である、テスト対象のアプリケーションを実際に操作するだけで、生成システムが自動的に可読性を高めた再利用性のあるテストケースを生成したり、高いカバレッジを達成しつつ有用なテストケースを優先的に生成できる生成器が有効であるという結論を述べる

## 3.1　GUIテストケース自動生成に求められる条件

テストケースを自動的に生成する場合、実際にテスト工程に適用する際には、そのツールやシステムがいくつかの現実的な条件を満たしている必要がある。

以下に満たすべきそれらの条件を示す。

- ツールやシステムの新規学習時間が少ないこと
- テストカバレッジが高いこと
- 特定の操作シーケンスに注目したテストケースが生成しやすいこと
  - テスト担当者は、特定の機能毎の特定の操作別にテストケースを管理しようとする。
  - 開発者は、コーディングしている個所に注目した操作シーケンスをテストしようとする。
- テストケースをスクリプトとしてエクスポートできること
  自動生成器が生成しきれないシーケンスに修正したり、ループ構造や場合分けのような簡単なコード追加をすることでより効果的なテストケースを構築可能になる。
- 手動用のテストケースをインポートできること

- 既存のテストケース資産運用
- 手動テストから自動テストへのスムーズな移行

これらは、そのツールやシステムを導入および運用する際に、コストへの影響が大きい要素である。開発形態やテスト形態に応じて、他の条件も必要な場合もある。

## 3.2 既存のGUIテストケース自動生成法

GUIテストケースの自動生成法は、3つのカテゴリに大別できる。本論文ではそれぞれCapture/Replay手法、組み合わせ生成手法、Planning手法と呼ぶこととする。

以下にそれらの性質と長所短所を示す。

### 3.2.1 Capture/Replay手法 [13, 15, 23]

市販されている多くのGUIテスト自動化ツールに搭載されている機能の1つに数えられる。

この手法は、テスト対象のGUIの操作履歴を記録し、テストケーススクリプトを生成する。ただし、GUIイベントシーケンスがそのままテストケーススクリプトのステートメント列としてテキストファイル等に出力される。テスト実行は、それらのテストケーススクリプトを再生することで行われる。

長所は、使用法が簡単であること、特定の操作イベントシーケンスを再現可能なため利用方法次第で様々なテストケースが生成できることである。

短所としては、生成されるスクリプトは単純にイベントシーケンスの羅列であるため可読性が低く再利用性が低いことと、高いカバレッジ達成をするためには多くのイベントシーケンスを記録する必要があることである。

### 3.2.2 組み合わせ生成手法

いくつかの手法があるが、基本的には、発生の可能性のあるイベント群から網羅的なテストを行うためのテストケースを生成する。そのためカバレッジ達成率は高い。生成とテスト実行は同時に行われ、基本的にスクリプト生成[1] のアプローチは取らない。

以下に代表的な手法を2つ挙げる。

- テストケースジェネレータプログラミング法 [19]
  重要な操作イベントポイントを記述したテキストから、それらを組み合わせることで、テストケースを生成する。

[1] ここでいうスクリプトとは、スクリプト自身がテストケースを表現しているものである。テストケースを生成するためのメタスクリプトではない。

長所は、人間の (そのポイントが重要であるという) 意図をある程度システムに伝えることが可能なことである。

短所としては、テキストを記述する際に、タイプミスやイベントポイントの書き忘れが生じやすいことである。

- 有限ステートマシン利用法 [17, 21, 22]

  初期画面から可能な操作シーケンスを有限ステートマシンで組み合わせ的に生成する。

  長所は、網羅的なテストケースをするためテストカバレッジが高いことと、生成作業がほぼ全自動化されていること等である。

  短所としては、生成ルールを後から制御できないため特定のイベントシーケンス向けのテストケースが生成できないことと、生成されるテストケース数が莫大になりすぎることである。

### 3.2.3 Planning 手法 [3, 4, 7]

人間が指示した生成プラン (ルール) に基づいて生成する。生成とテスト実行は同時に行われ、スクリプト出力は行わない。プランはあらかじめ人間がプランニングツールを用いて、詳細に指示する必要がある。次の操作作用に遷移するための前提条件と操作作用 (イベント) のペアを列挙したものを記述したものがプランである。そして、あるシナリオの初期状態と最終状態を表す 2 画面を入力することでテストケースを生成する。有限ステートマシン利用法と似ているが、この手法では同じ 2 画面を入力してもプラン毎に異なるテストケースを生成する点で異なる。

長所は、生成プランによって、具体的な生成制御を行えることである。

短所としては、プランニングは一種のプログラミングであるため、プランニングツールの使用法やそのプランニングルールに対する学習時間が必要なことである。

## 3.3 既存手法の比較と新しい手法の検討

既存の各手法を 3.1 節の条件において比較したのが表 3.1 である。

| | 学習時間 | カバレッジ | 特定シーケンス生成 | スクリプト化 | インポート |
|---|---|---|---|---|---|
| Capture/Replay | ○ | × | ○ | △ | ○ |
| 組み合わせ | ○ | ○ | × | × | × |
| Planning | × | △ | △ | × | △ |

表 3.1: 既存の自動生成法の比較

各手法の特性から次の特性を持つ自動生成器が有効であると考えることができる。

- テスト対象のアプリケーションの GUI 操作だけでテストケースを生成できる
  テストするアプリケーションを操作することがテストケース作成作業になるため新規学習時間が抑えられる。

- テストケースは、スクリプトとして出力できる
  スクリプトとしてテキストファイルに出力することで、テストケースの内容をエディタで閲覧、編集することが可能になる。これはテストケースのポータビリティ性向上にもつながる。

- 可読性が高く再利用性のあるテストケースを生成できる
  スクリプトに出力できたとしても、単なるイベントシーケンスの羅列のような場合、その可読性は低く、どの部分を再利用できるのかを解析しなくてはならない。生成システムにより、自動的に可読性を高め、再利用可能な部分を分離できる必要がある。

- テストパターン数の爆発化を抑えつつテストカバレッジの高いテストケースが生成できる
  テストカバレッジ達成は、テストを行うことの本質であるため、テストカバレッジの高いテストケース生成は必然である。ただし、網羅的に生成するのではなく、有用なテストケースを生成する必要がある。

本研究では、このような特性を持つ自動生成器を提案し、その実現に例示によるプログラミング手法を用いている。

# 第4章　例示プログラミング(PBD)の概要

本研究では、GUI テストケースの自動生成に例示プログラミング手法を応用している。例示プログラミングは、プログラムの振る舞いや効果の例を与えることによってプログラムを生成するアプローチの 1 つに含まれる。

以下に例示プログラミングの概要について述べる。

## 4.1　例示プログラミングとは

今日では様々な場面で計算機を使う機会が増えており、以前に増して計算機の操作性について言及されるようになってきている。

もちろん、操作の複雑さは計算機機能の向上に伴うものではあるが、比較的単純と思われる作業にも多くの手間が必要となる場面が多い。

この原因の 1 つとして、計算機にインタラクティブ性が追加されたことがある。計算機に行わせたい仕事をまとめて指示するバッチ処理と異なり、インタラクティブなシステムでは人間と計算機が対話することで作業を進めていくという概念を含んでいる。

インタラクティブシステムにおいてもスクリプトやマクロ機能によってバッチ処理的な指示をすることが可能なシステムもある。しかし、これらの機能を使うためには、ユーザにプログラミング作業が必要となり、初級ユーザのみならず上級ユーザでもその手間を削減したいと考えている。そのような場合、ユーザの実際の操作をお手本として、その操作に続く操作をシステムが自動的に行ったり、次回同じような操作をするときに利用可能なスクリプトやマクロを出力してやれば、ユーザビリティが向上すると考えるのが例示によるプログラミングという手法である。

例によってプログラミングを自動的に生成させる手法は、PBE(Programming By Example) [1] 、PBD(Programming By Demonstration) などと呼ばれる。

PBE は、ユーザがある処理の前後のデータの例を示すと、システムはその入力を出力へと変換する方法を見つけプログラム化する手法である。PBD は、ユーザがシステム上で例となる操作を行うと、システムはそれらの操作を記録しできる限り汎化する手法である。

[1] 自動プログラミングと呼ばれている時期もあった。

いずれの手法でも、システムはなんらかの推論機構を用いて、ユーザからの例を再利用できる形へ変換することを行う。変換されたものの利便性が高いほど、ユーザ操作の総量を大きく減らすことが可能になる手法として現在注目されている。

本研究では、特に例示プログラミング (PBD) 手法について注目し、テストケース自動生成に適用することを目指している。

## 4.2 例示プログラミングにおける推論機構

例示プログラミングでは、例となるユーザの操作が示されると、それらの汎化結果を出力する。汎化の処理を行う際には、各種の推論手法を用いることができる [32]。

- 過去の操作履歴を利用する

  繰り返し操作に対して自動的に次の操作を実行する例示プログラミングシステムでは、ある操作が行われた際にユーザの過去の操作履歴を参照し、該当しそうな次の操作のいくつかを示すことで、その後の操作量を減らすことが可能である。

- 汎用的な例示のデータを利用する

  あらかじめ多くの例示データをシステムに明示的に入力しておくと、機能操作の領域がある程度決まっている機能への操作については、次回行われる可能性が高いものをいくつか示すことができる。

  また、上級ユーザの知識とも言える例示データを利用して、初級ユーザも上級ユーザと同じ精度や効率のある操作を行うことが可能でもある。

- 機械学習や帰納的推論等を利用する

  ユーザからの例示データからの推論の精度を高めることで、次の操作の単純な予測だけでなく、ユーザの意図を汲み取ったり、好みや癖を抽出して適応的な動作をさせることが可能になる。このような推論機構を用いたシステムは、人工知能的な要素を持つことになり、さらなるユーザビリティを実現することが可能である。

  また、過去の偉人たちにより提唱、証明された方法論を用いることでそのシステムの推論機構の有効性を証明できる。

## 4.3 例示プログラミングの困難さ

例示プログラミングシステムは、非常に有用な手法と考えられている一方、その普及率はそれほど高くない。それは、この手法がある一面では非常に便利であっても、別の面で大きな問題点を有しているためである。

以下に、例示プログラミング全般の問題点について示す [32]。

1. プログラムを作成できない

   過去の履歴から推論を行う単純な例示プログラミングシステムでは、プログラム

を作成できず、次のユーザの操作の単純な推論しかできない。そのため、パラメタを含む操作のマクロ定義や条件判断/繰り返しを含む複雑な操作の自動化に使用できない。

2. プログラム生成のための指示が面倒
   単純な例示プログラミングシステムでは、システムに対してユーザが明示的にプログラムの作成をする必要は無いが、高度な推論を行うシステムの多くでは、例示を与える際にプログラムを作成していることを多かれ少なかれユーザが認識していることが必要である。そのため、特別な動作が必要になる。
   例えば、キーボードマクロのような単純な例示システムの部類に入るシステムにおいても、操作の開始と終了を明示的にシステムに指示する必要がある。このようなシステム上では、繰り返し操作を何度か実行してしまった後で繰り返しに気づくことも多いはずで、そのような場合にはあらためて繰り返し作業を行い、プログラム化させるというわずらわしい事態が発生する。

3. プログラム実行のための指示が面倒
   例示により適切なプログラムを作ることができた場合でも、これを適切な状況で実行するのが難しい場合がある。例えばキーボードマクロの場合、実行する状況が定義した状況と違っていた場合は全く異なる実行結果が返ってきてしまう。
   つまり、定義するコンテキストと実行時のコンテキストの違いの認識をユーザ側で行うか、システム側で行うかで大きな違いが生まれてしまう。

4. 正しいプログラムの作成が困難
   例示プログラミングシステムは帰納的推論によりプログラムを作成する自動プログラミングシステムの一種と考えることができるが、インタラクションにおける例示データは量が少ないことが多く、誤りや無関係なデータが含まれていることも多いためプログラムを生成することは難しい。そして、ユーザの意図と一致したプログラムの生成はさらに難しい。各種の例示プログラミングシステムのうち推論を行うものでは、システムが間違った推論を行わないようにするためユーザが頻繁に推論の間違いをチェックして修正する工夫がなされていたり、間違った推論結果を容易に修正できるようになっていることが多い。

5. 実行に関するリスクが大きい
   プログラムの実行に対して undo のような機構を用意しておけば間違った処理からの回復が可能であるが、間違った実行にユーザが気づかずにそのまま操作を進めてしまったり、物理的に回復が不可能な状況に陥ってしまうということもある。完全にシステムに実行をまかせることのリスクをどのような対処するかは、例示システムにおける重要な要点の 1 つである。

6. 例示プログラミング機構を使用しない方が楽である可能性がある
   推論を行う例示プログラミングシステムでは、システムは常に間違った推論を行

う可能性があるため、それを訂正しながら正しいプログラムを生成するための手間を考えると例示プログラミングシステムを使用しない方が得になる可能性が常につきまとうし、推論を行わないシステムについても同様のことがある。

7. 推論機構が操作の妨げになる
   いくつかの例示プログラミングシステムでは推論された結果が常にユーザに提示されるが、そのような機能がわずらわしくなる場合もある。また、常に推論機構を動作させることによるシステムの動作速度の低下という問題もある。

8. 大量のデータが必要
   多くの例示プログラミングシステムではヒューリスティックにより少ない例から汎化を行う工夫がされているが、高度な推論を行おうとするシステムは本質的に多量のデータを必要とするものが多く、応用が限られてしまっている。

9. ヒューリスティックが多用されている
   多くの例示プログラミングシステムでは、少ない例からでも一般化プログラムが生成できるようにするために各種のヒューリスティックが用いられている。しかしこれらのヒューリスティックは特定のアプリケーション/特定のインターフェイス手法/特定のユーザを仮定していることが普通であるため、広い範囲のシステムはユーザに受け入れられない可能性がある。また推論方式やヒューリスティックについてよく知らないユーザにとってひどく使いにくくなる可能性がある。

このような要因を持つ例示プログラミングシステムは、なにかしらの致命的問題を抱えたシステムとなる可能性がある。逆に言えば、これらの問題点は例示プログラミングシステムを評価するガイドラインと言える。

## 4.4 例示プログラミングシステムに求められる条件

4.3 節で示した問題点から、例示プログラミングシステム構築のガイドラインを求めることができる。以下にガイドラインとなる項目を示す。

- プログラムを出力する
  単純な例示プログラミングは有用ではあるものの適用範囲が狭い。複雑な処理を実行させるため、例示プログラミングシステムにはスクリプトやマクロの定義/繰り返し/条件判断などのプログラムを出力できるようにする。
  これは問題点 1 を解決するための条件となる。
  また、プログラムが出力されれば、資産と出来るし再編集も可能になる。

- ユーザの暗黙的意図を自動的に汲み取る
  例示データを明示的に与える操作はユーザにとって負担が大きいので、操作履歴

情報のような暗黙的例示データを活用したり、システムに例を自動生成させたりすることにより、ユーザが特に意識しなくてもプログラムを自動的に生成できるようにする。
　これは問題点2を解決するための条件となる。

- 推論機構の動作開始をユーザが指示できる
　システムが自動的に推論を行った結果をユーザに提示することはせず、ユーザの指示により推論を行う。もしくは、推論開始のタイミングをただ1つとする。例えば、アプリケーションの終了時に次回起動時に再利用可能な推論結果を示すようにする。
　これは問題点7を解決するための条件となる。

- システム利用の新規学習時間を少なくする
　推論結果の実行をする方法が複雑だったり、通常の操作法と大幅に異なるようであれば、ユーザは混乱し、推論機構を利用しなくなる。なるべく多くの事柄を学習しなくても良い、実行開始法を提供する必要がある。
　これは問題点3を解決するための条件となる。

- undoのような推論の実行を取り消す機構を用意する
　誤った推論の実行のリスクを回避できるようにする
　これは問題点3を解決するための条件となる。

- ヒューリスティックな推論は最低限に抑える
　多くのヒューリスティックに基づいた複雑な推論は、その処理時間が多く必要なことがある。シンプルで汎用的な推論手法を使用し、アプリケーション固有のヒューリスティックの利用比率を抑えることが必要である。
　これは問題点4. 6. 8. 9. を解決するための条件となる。
　もちろん、ヒューリスティックな推論は非常に効果的な推論を導くことも多いため、その利用比率の検討は重要である。

このガイドラインに基づいた例示プログラミングシステムを一言で表すと、「少量の例示データから汎用の単純な手法によりユーザの暗黙的な意図をプログラムとして自動的に抽出し、簡単な操作でその後の操作に適用することを可能にするシステム[32]」ということになる。

# 第5章　例示プログラミング手法を用いたテストケースの自動生成の提案と実現

本章では、本研究のアプローチについて述べる。

本研究では、GUI テストケース自動生成器システムの実現に例示プログラミング手法を用いる。例示プログラミング手法では、ユーザの操作シーケンスから汎化推論等を行い、再利用可能なプログラムを生成する手法である。

以下では、テストケース作成者の GUI 操作シーケンスを基に汎化推論とゆらぎテンプレート適用によって、一般化テストケーススクリプトおよびゆらぎテストケーススクリプトを生成する GUI テストケース自動生成器の提案とその実現について述べる。

## 5.1　生成対象とするGUIの条件

本研究では、ボタンのみで構成される GUI、つまりボタンウィジットやハードウェアキーを押下することで操作可能な GUI をテストケース生成の対象としている。それは次の理由による。

GUI の代表的な構成要素は、ウィンドウ、ウィジット、そしてポインティングデバイスである。ユーザはポインティングデバイスを通して、ウィンドウやウィジットを操作することで、対応するイベントを GUI に通知する。

低レベルのイベントには多数の種類があるが、ユーザの GUI 操作という高レベルでは、その数はそれほど多くない。一般のユーザが認識できているイベントは、ボタンクリックとポインタの移動、ドラッグ＆ドロップの 3 つと言える。また、ウィジットの多くもユーザ操作の上ではボタンウィジットのサブクラスと見ることができる。チェックボックス、リストボックス、メニュー、テキストフィールド、ラベル等はウィジットを押下したり、キーボードを使用することで操作が可能である。

マウスデバイス等の操作が必須であるドローソフトなどは、対象としていない。例示プログラミング手法では、ユーザ操作からその操作意図を認識するが、ポインタの軌跡からユーザの意図を認識することは非常に困難なためである。

例えば、ユーザがマウスを S 字状にアイコンをドラッグ＆ドロップしたとする。ユーザが単にドラッグ＆ドロップ機能を使用しているだけならば、S 字状の軌跡には意味は無い。

しかし、そのアプリケーションがドローソフトの場合、その軌跡には重要な意味が含まれている可能性がある。このような場合、アプリケーションに依存した例示プログラミング手法が必要になり、必ずしもポインタの軌跡情報を必要としない多くのアプリケーションとは異なるアプローチが必要となってしまう。

必ずしもポインタの軌跡情報を必要ではない、ボタンのみで構成される GUI には、ワードプロセッサや表計算ソフト、WEB アプリケーションや組み込み機器の GUI など、多くのアプリケーションがある。これらのアプリケーションでも、2.2 節で述べた GUI テストの困難さは存在する。

これらのことから、本研究ではボタンのみで構成される GUI、もしくはボタンと等価なウィジットのみで構成される GUI を対象としている。

## 5.2 例示プログラミング手法を用いたアプローチ

本研究では、GUI のテストケース生成コストを削減するための GUI テストケース自動生成器システムの実現に例示プログラミング手法を用いる。

例示プログラミングはプログラムの振る舞いや効果の例を与えることによってプログラムを生成する手法であり、テストケース作成者の GUI 操作シーケンスから推論やゆらぎテンプレート適用によって目的に応じた各種のテストケーススクリプトを自動生成するという本研究のアプローチに適していると考えることができる。

まず、本システムが対象とする GUI とユーザは以下の通り。

- ボタンのみで構成される GUI アプリケーション
- システムのユーザはテスト担当者と開発者

次に 3.1 節で提案した GUI テストケース自動生成器の要件を以下に示す。

1. テスト対象のアプリケーションの GUI 操作だけで生成できる
2. テストケースは、スクリプトとして出力できる
3. 可読性が高く再利用性のあるテストケースを生成できる
4. テストパターン数の爆発化を抑えつつテストカバレッジの高いテストケースが生成できる

本研究では、ユーザの GUI 操作を基に汎化推論とゆらぎテンプレート適用という 2 つのテストケース生成処理を行う例示プログラミングシステム (図 5.1) を提案する。

このシステムによる GUI テストケース生成の処理の流れは以下の通りである。

1. ユーザの GUI 操作シーケンスのキャプチャリング

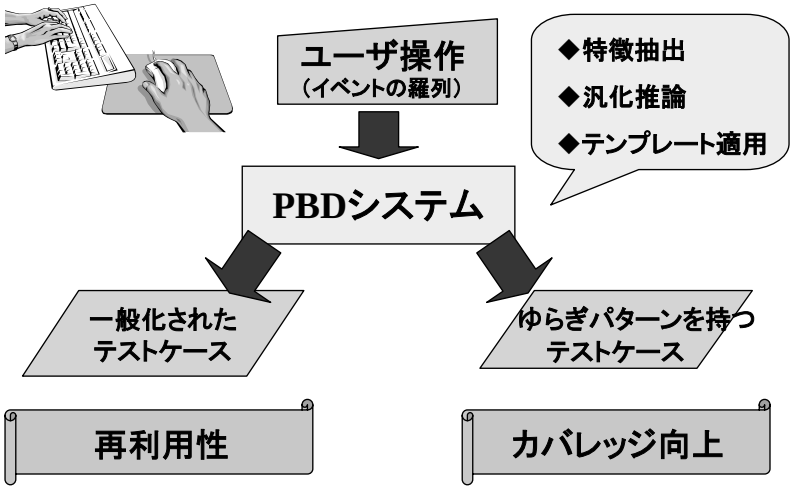


図 5.1: 例示プログラミング手法を用いた GUI テストケース自動生成器

2. 特徴抽出

3. 汎化推論およびゆらぎテンプレート適用

4. テストケーススクリプト生成

生成されるテストケースは要件 3. と 4. を目的する 2 種類のスクリプトとなる。

**一般化テストケーススクリプト**: 可読性や再利用性の向上を目的としたテストケーススクリプト。GUI 操作の羅列から構造特徴を抽出し、汎化推論を行うことで生成される。スクリプトには制御構造化やサブルーチン化そしてコメント等が追加され、人間によるテストケースの編集を助ける。

**ゆらぎテストケーススクリプト**: テストカバレッジ達成率の向上を目的としたテストケーススクリプト。ある機能操作のユーザの GUI 操作シーケンスを代表的な操作パターンと見なし、そのパターンに対してゆらぎテンプレートを適用することで生成される。テンプレートにはテストケース作成者経験が反映されており、適度な量のテストパターン数に抑えつつテストカバレッジの向上を助ける。

## 5.3 GUI の構造特徴

一般的な GUI の構造的なデザインに注目すると、GUI は機能別にレベル分けされた階層的なウィンドウ構成を持っていることがわかる。そのため、一般ユーザは、この階層上で GUI を操作することになる。そして、テストケース作成者も、機能テストを行う際に

は、一般ユーザの立場に合わせて、この機能的階層を意識してテストケースを作成している。よって、テストケースは機能的階層を基に作成されていることがわかる。

ある操作シーケンスのイベントフローグラフを図5.2に示す。このグラフは、右回り先行順のグラフである。C で表されるのが、ボタン押下イベントでその番号はボタンウィジットのIDである。

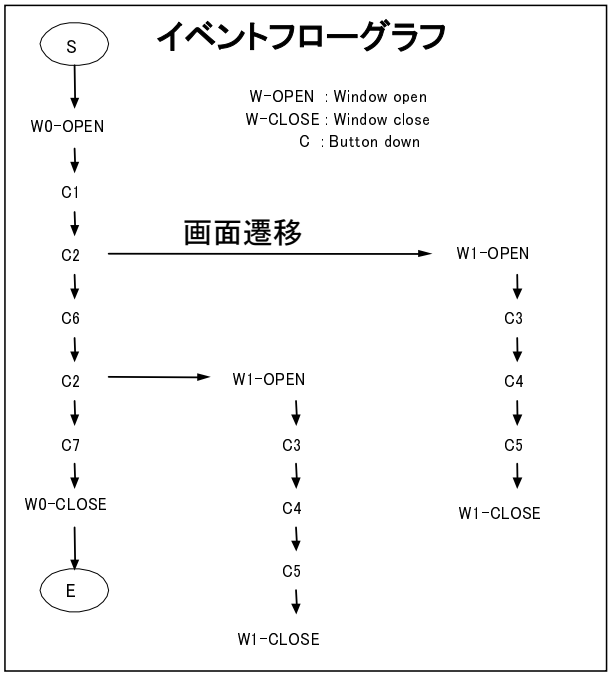


図 5.2: イベントフローグラフ

ボタン押下イベントの動作を見ると、ユーザ操作によるボタンの挙動は２つのグループがあることがわかる。

- 選択や設定等を行うボタン押下
- 画面遷移を行うボタン押下

GUIが機能別にレベル分けされたデザインをしていることから、連続している「選択や設定等を行うボタン押下」は機能操作上関連の深いものであると判断できる。そして、「画面遷移を行うボタン押下」からウィンドウが閉じられるまでのボタン押下は、機能的階層の区切りを表していると判断できる。

よって、この2つのグループを分別することで、GUIの機能的階層の認識ができると考えることができる。

これら2つのグループをそれぞれ、Linear-Command Pattern(以下、Lパターン)、Transit-Layer Pattern(以下、Tパターン) と呼称する。

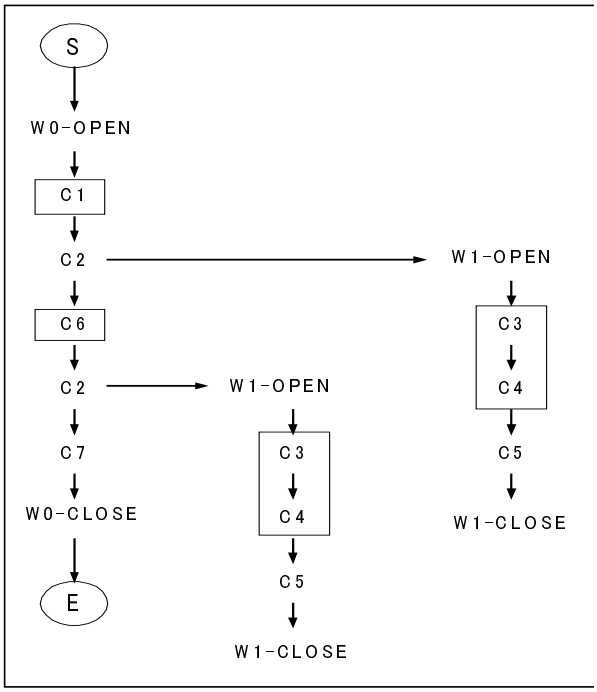


図 5.3: Linear-Command Pattern

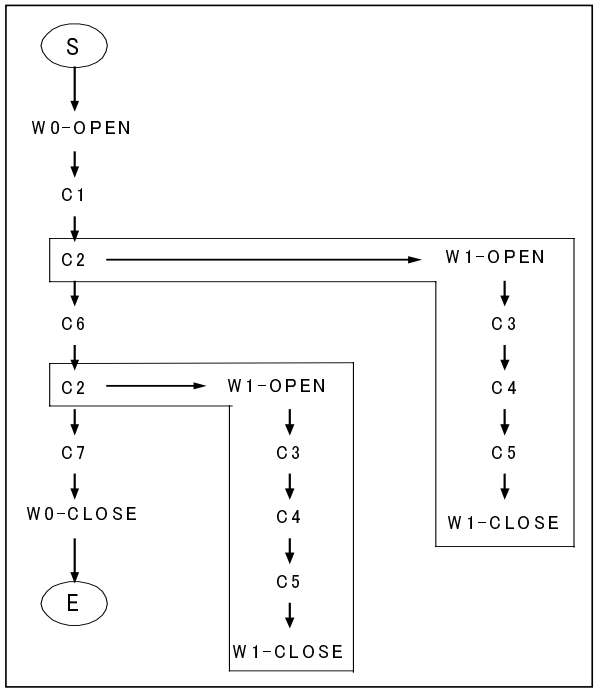


図 5.4: Transit-Layer Pattern

### 5.3.1 Linear-Command Pattern

L パターン内 (図 5.3) の各要素は、「選択や設定等を行うボタン押下」である。機能的階層から見ると、L パターンはその階層内での操作シーケンスを表現している。よってパターン内の各要素は機能操作上の関連が深いボタン押下イベントを表す。

L パターンで表現されるものを以下にまとめる。

- 操作表現：機能的階層内での「選択や設定等を行うボタン」の操作シーケンス
- イベント表現：ウィンドウの生成・消滅に関係の無いボタン押下イベント

### 5.3.2 Transit-layer Pattern

T パターン (図 5.4) は、「画面遷移を行うボタン押下」によって分けられた機能的階層そのものである。T パターンから、操作シーケンスから機能的階層を識別することができる。

T パターンで表現されるものを以下にまとめる。

- 操作表現：操作シーケンスの機能的階層の 1 単位
- イベント表現：ウィンドウの生成・消滅イベントとその間のボタン押下イベント

- パターン表現：機能的サブ階層 (他の L,T パターン) を含むネスト表現

## 5.4 構造的特徴の抽出法

GUI の構造的特徴である機能的階層に注目したパターン、L パターンと T パターンを自動抽出する方法を以下に示す。

- L パターン抽出
  - C ノードのみを子に持つ C ノードの連続列を、L ノードとして付け替える
- T パターン抽出
  - W-OPEN ノードを子に持つC ノードから、W-CLOSE ノードまでの一連のノード列を T ノードとして付け替える
  - T ノードはネストし、下の階層のノードを全て含む

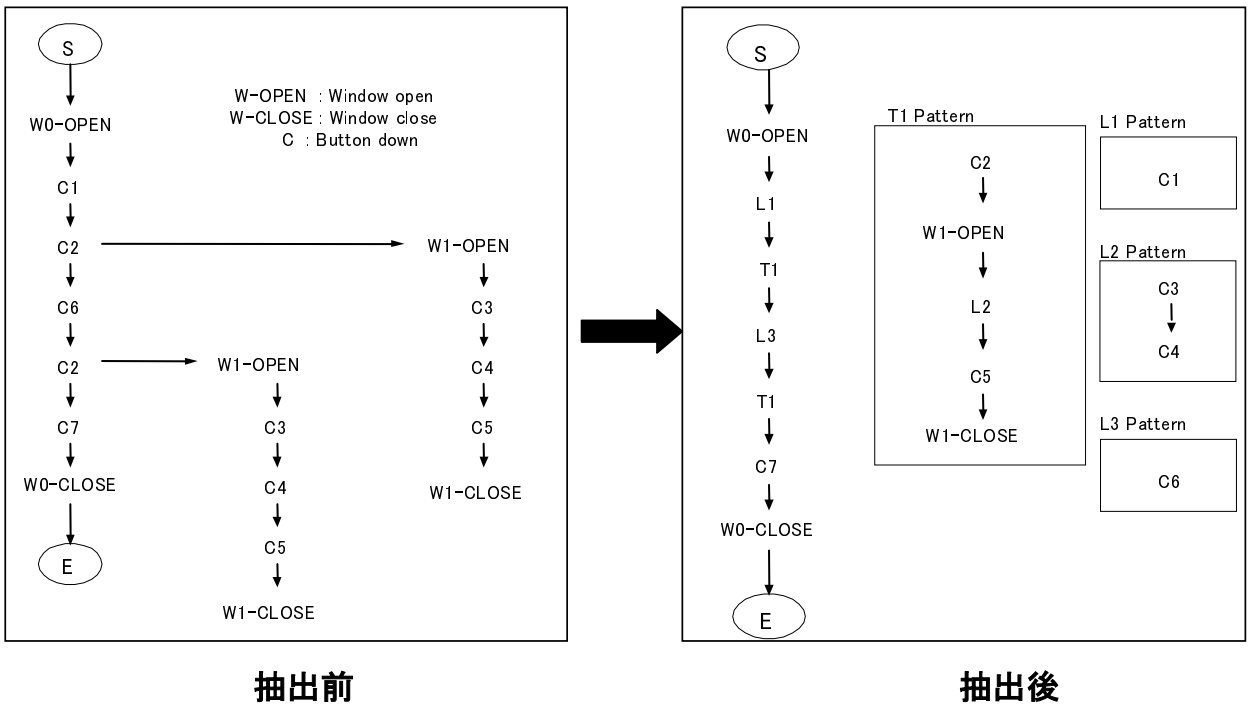


図 5.5: パターン抽出例

図 5.5 にパターン抽出例を示す。同一のパターンには、同一のパターン名がつけられていることがわかる。各パターンには、重複回数がカウントされており、重複出現パターンを特定することができる。例では、T1 と L2 の 2 パターンが重複出現するパターンとして、認識されていることが分かる。

## 5.5 汎化推論とゆらぎテンプレート適用によるテストケーススクリプト生成

本研究で自動生成するテストケーススクリプトの種類は、「一般化テストケーススクリプト」と「ゆらぎテストケーススクリプト」の 2 つである。それぞれ、汎化推論とゆらぎテンプレート適用を行うことで生成を行う。

出力されるスクリプトは TCS 言語で記述されたものとなる。TCS 言語は本研究で設計したテストケース記述言語[1] である。

### 5.5.1 一般化テストケーススクリプト生成

キャプチャリングしたユーザ操作シーケンスは、そのままの形でもその操作シーケンスを再現するテストケースとして使用可能である。実際に、Capture/Replay 手法 [13, 15, 23] で用いられている。だが、通常はそのままでは用いずに、出力されたイベントシーケンスを基に手動で構造化やコメント追加をしたスクリプトに再編集しなおして使用されることが多い。それはテストケースの改良や保守作業の効率化を測るためである。この再編集作業はコストのかかる作業である。それは、編集前のスクリプトが操作シーケンスのイベントの単なる羅列であるため、可読性が低く再利用性を考慮していないものだからである。

本研究では、そのような作業のコストを削減するため、可読性と再利用性の高いスクリプトの生成を行う。パターンの種別や出現頻度からヒューリスティックな汎化推論によって制御構造化やサブルーチン化そしてコメント追加を行うという方法を取る。

以下に汎化推論処理について述べる。

- サブルーチン化

  T ノードは、機能的な意味において分離可能と見なし、再利用しやすいようにサブルーチンとする。

- ループ化

  同じ機能を繰り返しテストする場合、L-T ノードが連続し、かつ T ノードが同一名称をもつシーケンスが発生しやすい。そこで、2 回以上出現している T ノード (以下、重複 T ノードと呼ぶ) を子に持つ L ノードは、ループ化の利点があると見なし、foreach や repeat 等のループブロックに変換する。

  ループ化の利点とは、ループ回数のパラメータを変更したり、ループブロック内に条件判断文を追加するといった手動での再編集が行いやすくなるだろうというヒューリスティックな推論に基づく。

- コメント追加

[1] 6.3 節にて解説する。

- W-OPEN、W-CLOSE ノードは、親ノードのコメント文として出力する。
  ただし、親ノードが START ノードの場合は、main ルーチンのコメント文とする。
- サブルーチンの定義部には、サブルーチン名をコメント文として出力する

- イベント発生
  イベント発生は、TCS 言語の組み込み関数 push() を用いる。
  L パターンの要素となっている C ノードは、リスト型[2] の引数とする。それ以外の C ノードは、直接 push() 関数の引数とする。L パターンの要素を 1 つの集合のままで扱うという推論は、抽象度を高めるとともに、その集合に処理を行うことで、新たなテストケースの作成作業が行いやすくなるだろうという判断に基づく。

図 5.5 で抽出された L,T パターンを基に汎化推論を行った例を図 5.6 に示す。

```
1: // Sample Testcase
2: func main( ) {          // W0-OPEN
3:     var L1 = [ c1 ] ;
4:     var L3 = [ c6 ] ;
5:     foreach L ( L1, L3 ) {
6:         push( L ) ;
7:         T1( ) ;          // Call T1
8:     }
9:     push( c7 ) ;       // W0-CLOSE
10: }
11: func T1( ) {           // T1
12:     var L2 = [ c3, c4 ] ;
13:     push( c2 ) ;       // W1-OPEN
14:     push( L2 ) ;
15:     push( c5 ) ;       // W1-CLOSE
16: }
```

図 5.6: 一般化テストケース出力例

行番号 3,4,12 にて、それぞれ L1～L3 パターンをリスト型オブジェクトとして初期化宣言している。重複 T ノードである T1 は、行番号 11～16 にサブルーチン出力されている。また、L1 と L3 は重複ノード T1 を子に持っていたことから、foreach ループブロックとし

[2] TCS 言語では、データ型に整数型、浮動少数型、文字列型のほか、リスト型も使用できる

てまとめられている。そしてL1〜L3は、その集合を保ったままpush()関数の引数として使用されている。

本研究では、上記のヒューリスティックな汎化推論を行い、一般化テストケーススクリプトの生成を行う。

### 5.5.2 ゆらぎテストケーススクリプト生成

3.2節で述べたように、Capture/Replay手法[13, 15, 23]の短所は、入力した操作シーケンス以上のテストパターンを生成できない点であった。

テストケース作成時における問題点は、2.5節でのように十分な自動化を行うためのテストケースを用意するためのコストが高いことである。そのことを考慮すると、入力した操作シーケンス以上のテストパターンを生成できる必要がある。しかし、テストカバレッジを100%に近づけようと、有限ステートマシン利用法[17, 21, 22]のように盲目的にテストパターンを生成すれば、数の爆発化による弊害が生じてしまう。

通常テストケース作成者は、テスト作業を行う際に、一定のテスト操作プランを適用して作業を進めている。そのテストパターンは、過去の経験則やアプリケーションの特性から生み出されているものである。これらの知識は数の爆発化を抑えつつ有用なテストケースを生成するのに非常に有効である。

本研究では、それらの知識をゆらぎテンプレートという形で利用する。テンプレートには、テストケース作成者がよく行うテスト操作プランをL,Tパターンに適用する形で記述されている。このアプローチを取ることで、Planning手法[3, 4, 7]のように手動で生成プランを入力することをしなくとも、有効度の高いテストケースを優先的に生成しつつも、カバレッジ達成率を向上することができる。

各テンプレートは、アプリケーションに依存するため、ユーザが定義することが基本となる。ただし、一般的なテスト操作テクニックが記述されたテンプレートはシステム側で用意する。今回は、2つのテンプレートを用意し、評価実験で用いている。(7.3節参照)

また、ゆらぎテストケーススクリプトは、各テンプレートの適用毎にTCS言語によって記述されたものとして生成される。

## 5.6 自動生成器の実装

### 5.6.1 実装環境

設計した自動生成器(以下、本システム)の実装環境は以下の通りである。

**OS** :Windows 2000 SP3

**開発ツール** :Visual C++

本システムは、イベントキャプチャ機能を含めて、クラス数 14、約 4000 行で構成される。行数には、コメント行、空行を含んでいる。ただし、開発ツールによる自動生成コードは含んでいない。

### 5.6.2 実装仕様

本システムの実装仕様の概要を以下に示す。いくつかの仕様については、実装環境に依存している。

#### イベントキャプチャリング機構

本システムは、ユーザの操作イベントシーケンスを取得するために、イベントキャプチャリング機構を搭載している。ここでは、Windows OS 上での実装方法について述べる。

テスト対象のアプリケーションは、スレッドとして動作している。本システムでは、そのイベントの監視をスレッドのイベントをフック[3] することで行っている。

本システムでは、特徴抽出に必要なイベントのみをフィルタリングすることでイベントを取得している。図 5.7 は、その動作概念図である。フックしているイベントの種類とその用途を以下に示する。

- WH_CBT:ウィンドウのオープンクローズイベントの取得
- WH_CALLWNDPROC:ソフトウェアキー (ウィジット) 押下イベントの取得
- WH_KEYBOARD:ハードウェアキー (キーボート) 押下イベントの取得

ユーザ操作によって、フックをかけたイベントが発生すると、各イベントのコールバック関数が呼ばれる。イベントは、必ずしも特徴抽出に必要なものとは限らないため、各コールバック関数内で、フィルタリングする[4] 。その後、自動生成器内のイベントログクラスにそのイベント情報を送る。

本システムのイベントキャプチャ機構では、以上の方法を用いて、特徴抽出に必要なイベントのみを取得している。

[3]Windows OS では、イベントフックの方法を 2 つ提供している。ひとつは、スレッドローカルなフックでローカルフックと呼ばれている。もうひとつは、システムフックで、OS 全体のイベントに対してフックをかけることができる。本システムでは、ローカルフックを用いているが、内部的にはシステムフックへ移行できるようになっている。

[4]特にウィンドウのオープン/クローズイベントに関しては、システム内部に作成したローカルウィンドウスタックで管理することで、対象スレッドのイベントのみを取得することを保証している。

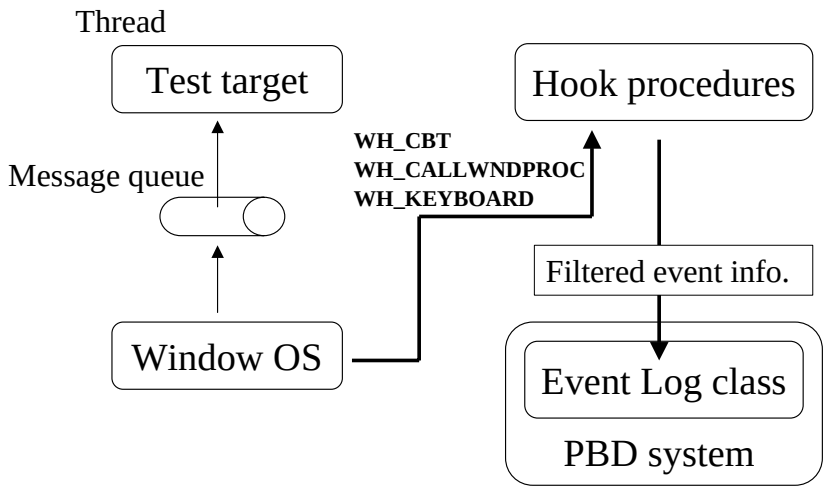


図 5.7: イベントのキャプチャリング

### 5.6.3 パターン管理

L パターンと T パターンと特徴抽出する際に、各パターンに関する情報を管理し、スクリプト生成時に利用できるようにする必要がある。ここでは、その管理データ構造について述べる。

各パターンの情報は、パターン管理クラスの 2 つの管理マップで管理されている。これらの管理マップは STL の map クラスを用いて実装されている。パターン検索キーは、パターン内のイベントシーケンス自体のハッシュ値である。この値は、パターン管理クラスのメソッドを用いて求めることができる。

- パターン名管理マップ：
  パターン名に対応するパターン検索キーとパターン出現数を管理する
  - キー：パターン名
  - 登録データ：パターン検索キー、パターン出現数
- パターン管理マップ：
  パターン検索キーに対応するイベントシーケンスとパターン名を管理する
  - キー：パターン検索キー
  - 管理データ：パターン内のイベントシーケンス、パターン名

お互いの管理マップは、もう一方へのハッシュキーを含んでいるため、パターン名かパターン検索キーを用いて、スクリプト生成に必要な情報を管理することができる。

### 5.6.4 動作例

図5.8は、本システムの動作例である。実際にテスト対象のGUIアプリケーションを操作したイベント列が、汎化処理によって構造的なスクリプトに変換されている様子が確認できる。

```
=== PBD Event Log Start ==========
Window : Name = 飛行機チケット予約システム, Status = (
Button Click : ID = 1017
Window : Name = テンキー, Status = OPEN
Button Click : ID = 1047
Button Click : ID = 1049
Button Click : ID = 1044
Button Click : ID = 1045
Button Click : ID = 1
Window : Name = テンキー, Status = CLOSE
Button Click : ID = 1027
Window : Name = 搭乗便選択, Status = OPEN
Button Click : ID = 1038
Window : Name = 搭乗便選択, Status = CLOSE
Button Click : ID = 1013
Button Click : ID = 1027
Window : Name = 搭乗便選択, Status = OPEN
Button Click : ID = 1038
Window : Name = 搭乗便選択, Status = CLOSE
Button Click : ID = 1014
Button Click : ID = 1014
Button Click : ID = 1027
Window : Name = 搭乗便選択, Status = OPEN
Button Click : ID = 1038
Window : Name = 搭乗便選択, Status = CLOSE
Button Click : ID = 1005
Window : Name = チケット予約, Status = OPEN
Button Click : ID = 1
Window : Name = チケット予約, Status = CLOSE
Button Click : ID = 1
Window : Name = 飛行機チケット予約システム, Status = (
=== PBD Event Log End ============
```

```
// Generalized Script

func main(){            //WINDOW-OPEN 飛行機チケット予約システム
        var L1 = [ 1013 ];
        var L2 = [ 1014, 1014 ];

        T0();
        T1();
        var i, L ;      // Temporary for loop
        L = [ L1, L2 ];
        for( i = 0 ; i <  length( L ) ; i ++ ) {
                push( L[i] );
                T1();
        }

        T2();
        push( 1 );              //WINDOW-CLOSE 飛行機チケット予約システ
}

func T0(){                      // T0
        var L0 = [ 1047, 1049, 1044, 1045 ];

        push( 1017 );           //WINDOW-OPEN テンキー
        push( L0 );
        push( 1 );              //WINDOW-CLOSE テンキー
}

func T1(){                      // T1
        push( 1027 );           //WINDOW-OPEN 搭乗便選択
        push( 1038 );           //WINDOW-CLOSE 搭乗便選択
}

func T2(){                      // T2
        push( 1005 );           //WINDOW-OPEN チケット予約
        push( 1 );              //WINDOW-CLOSE チケット予約
}
```

図 5.8: 自動生成動作例

# 第6章　テストケース実行環境の設計と実装

本章では、自動生成したテストケースを実行するための環境の設計とその実装について述べる。

本研究のテストケース実行環境は、テストケーススクリプトの処理系、およびGUIの「見た目」の状態遷移をテストするために必要なテストオラクルの取得と比較の機能を含んでいる。

## 6.1　テストケース実行環境に求められる要件

テストケース実行環境を設計する際に満たすべき要件について考察する。

テストケース実行を自動化もしくは半自動化することは、回帰テストの高いコストへの解決策の1つとなり得る上に、負荷テストのような単調作業を効果的にこなすことを可能にする。

以下に、GUIの不具合検出を行うためのテストケース実行環境要件を列挙する。

- GUI操作シーケンスを再現できる
  現在はGUIの実装にイベントドリブン方式が主流となっている。よって、GUIの操作シーケンスはイベントシーケンスとしてアプリケーション内で処理されている。このため、テストケース上に表現されている、そのイベントシーケンスをテスト実行時に再現できることが必要となる。つまり、必要十分な種類のイベントをテスト対象のアプリケーション上で逐次発行できなければならない。

- テスト作業をコンピュータが解釈できる
  実際の手動用テストケースでは、テスト時の条件や一連の操作繰り返しを自然言語で表現している。そのようなテスト作業中の条件分岐やループ構造をコンピュータが解釈しテストシーケンスを進められるためには、テスト作業を表現できるようなプログラム言語が必要であり、テストケース実行環境にはそのプログラム言語の解釈機能が必要となる。

- テスト作業状況/結果をテストログとして記録できる
  アプリケーションをテストする作業は、不具合発生個所を特定するだけでなく、

そのアプリケーションの品質を確認し、品質保証を行うためでもある。特に回帰テストによってアプリケーションのバグ曲線の様子やコード修正による波及の範囲を把握することは、品質管理において重要事項である。よって、テストログを記録保存することで、それらの作業を支援する機能は必須とも言える。

- GUI 操作のどの時点で不具合が発生したかを特定できる
  GUI のテストを行う際には、その下位レイヤであるビジネスロジックやアプリケーションロジックの不具合発生を検出できるとともに、GUI 自体の「見た目」の不具合をも検出できなければならない。そのためには、イベント毎に「見ため」の状態遷移を記録し、その正しい状態遷移との比較ができなければならない。

一般的な GUI テスト自動化ツールは、これらの各要件を基本機能として装備している。そのことからも上記の各要件を GUI テスト実行環境設計のガイドラインと見なすことは妥当であることがわかる。

## 6.2 テストケース実行環境の設計

6.1 節の要件から、次の機能を持つテスト実行環境を設計した。

- テストケーススクリプトインタプリタ
  本研究で設計した TCS 言語 (Test-Case Script) を用いて記述されたテストケーススクリプトを解釈しイベント発生リクエストを行う。テストマネージャと連動することで、GUI 操作シーケンスを再現する。

- テストマネージャ
  テストケースインタプリタからのイベント発生リクエストを受け取り、テスト対象のアプリケーションに対して、実際のイベントを発生させる。各イベントに応じて変化する GUI のスクリーンショットおよび各種の内部ステータスを記録することで、どのイベント発生時に不具合が生じるかを判断するテストログを生成する。

- テストログデータベース
  テストマネージャによって記録されたテストログを格納する。このテストログは保持され、不具合発生個所の特定や回帰テスト時の品質判断に利用される。

- テストログ比較ツール
  指定された 2 つのテストログを比較し、その比較結果を HTML 文書として出力する。

図 6.1 は本研究のテストケース実行環境の動作を示したものである。テストケーススクリプトを入力した後の動作の流れを以下に示す。

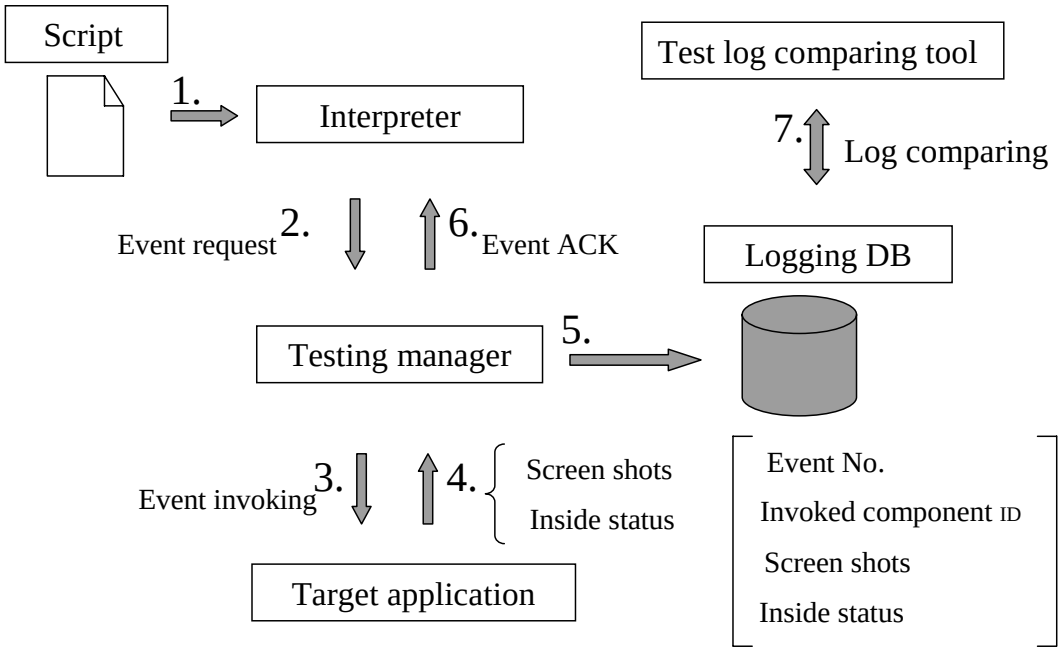


図 6.1: テストケース実行環境動作シーケンス

1. テストケースを記述したスクリプトファイルを取り込み、インタプリタへ入力する。

2. インタプリタがスクリプトを解釈しテストマネージャにイベント発生のリクエストをする。

3. テストマネージャは、リクエストに応じたイベントをテスト対象のアプリケーション内で発生させる。

4. アプリケーションは一連の処理を行った後、GUI にその処理結果を表示する。その時のスクリーンショットと内部ステータスはテストマネージャによって記録される。内部ステータスは、GUI 上に表れないが GUI の操作シーケンスに依存するような不具合を検出するのに用いられる。

5. その後それらのログは、イベントの発生番号とイベントを受け取ったウィジットコンポーネントの ID と共にテストログとしてデータベースに格納される。

6. イベントに対するロギングが終了したことをインタプリタへ通知する。インタプリタはスクリプトの処理が終わるまで、順次イベント発生のリクエストを行う。

7. 記録されたデータの内、正しい動作をしているもの[1] をテストオラクルとし、テストログ比較ツールを用いて回帰テスト結果の評価を行う。

[1] この確認は人間が実際に目で確認して行う。

## 6.3 テストケーススクリプト処理系の設計

本研究では、テストケース実行環境を設計するにあたり、新たに TCS 言語と呼ぶテストケース記述言語を設計した。この TCS 言語は、リスト表現が可能であり、かつコンパクトな仕様となっている。

### 6.3.1 TCS 言語 : Test Case Script language

本研究では、GUI 操作イベントを機能的階層別にグループ化し、その構造的特徴を L パターンおよび T パターンとしてに分別している。TCS 言語では、グループ分けされた集合を効率よく扱うために言語仕様レベルでリスト構造を含めている。

また、C 言語に似たコンパクト文法を持たせることで、C 言語や JAVA 等の既存の言語プログラマがテストケースを記述する際の新規学習時間の短縮を狙っていると共に、言語処理系のサイズを小さくすることも考慮している。

TCS 言語の特徴を以下に示す。(言語仕様に関しては、付録を参照のこと)

- 柔軟なデータ構造を構築できる型無し言語[2]
- 変数の明示的な宣言 (型指定は不要) をさせることで、単純なプログラムミスのリスク回避をしている
- スカラ型 (整数、浮動小数、文字列、NULL) およびリスト型の 2 種類のデータ型を用意
- C 言語に似た構造化記述が可能
- JAVA、Perl の特徴をいくつか取り入れている。
  例 : GOTO 文の廃止、ループラベルの導入、リストを用いた関数の可変長引数と可変長返り値
- コンパクトな言語仕様

これらの特徴により GUI テストケースの記述性と記述容易性を目指した言語仕様となっている。

本研究のテストケース実行環境は、テスト対象のアプリケーションに直接組み込むような用途も考慮しており、コンパクトな言語仕様であることが最大の利点であると考えている。実装上の工夫としては、組み込み関数を導入しやすいようにし、対象アプリケーションに応じた拡張をできるようにしている。

[2] 変数に型はない。データにのみ型がある。

### 6.3.2 処理系のシステム構成

テストケーススクリプトの処理系の構成を図 6.2 に示す。

本処理系は、セミコンパイル型インタプリタの形をとっており、TCS スクリプトを TCS-P コードと呼ぶバイトコードにセミコンパイルしたものを実行する。TCS-P コードの実行エンジンはスタックマシンとして実装されている。

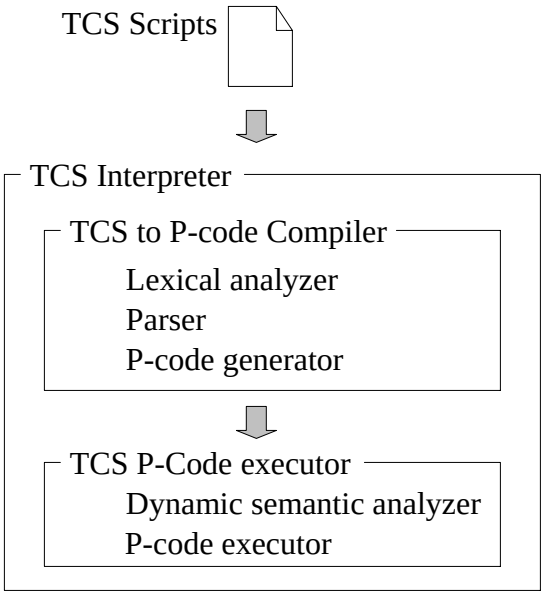


図 6.2: テストケーススクリプト処理系の構成

## 6.4 テストロギング

2.2.6 節で述べたように、GUI テストでは「見た目」の状態遷移に対するテストオラクルが重要である。本研究ではイベント毎にウィンドウのスクリーンショットを取り、その動作が正しいことを一度人間の目で確認したものをテストオラクルとして使用する。そして、回帰テスト毎にそれらのスクリーンショットを比較することで、どの操作によって画面に不具合が現れたのかを特定できる実行環境を実現した。

以下では、ウィンドウのスクリーンショットを GUI のテストオラクルとする事の利点と欠点、そして欠点への解決案について述べる。その後ロギングを行うタイミングの問題について述べる。

### 6.4.1 GUI テストオラクルとスクリーンショット

本研究では、GUI テストオラクルにウィンドウのスクリーンショットを用いることにした。その理由として GUI の「見た目」の不具合を検出するという要件を満たすためである。GUI では、ユーザの操作に応じて「見た目」の状態が遷移していく。この状態遷移は、下位レイヤのビジネスロジックやアプリケーションロジックの処理およびウィンドウシステムによる処理によって行われ、その処理結果が最終的に画面に出力される。つまり、ユーザの操作と画面を一意に関連付けてテストオラクルに設定することで「見た目」の状態遷移をテストできることになる。

以下にイベント毎のスクリーンショットを GUI テストオラクルとすることの利点を列挙する。なお、画面情報以外の特殊な情報を扱う GUI アプリケーションについては考慮していない。

1. GUI の不具合発生タイミングが確実に特定できる
   ユーザが目にする唯一の情報であるスクリーンショットを操作イベント毎に記録することで、不具合発生の手順を再現できる。これにより、どの時点で画面に不具合が現れたかを特定できる。

2. グラフィックスデータの変更を検出できる
   ユーザ操作に応じて画面上にグラフィカルデータを表示するアプリケーションでは、対応するデータが正しく表示されているかを確認する必要がある。そのようなアプリケーションでは、グラフィカルデータの内部管理を ID やファイル名などで行っていることが多い。もしそれらの ID やファイル名の一致だけをテストするならば、データ自体に変更が加えられた場合を検出できない。これは、実際の表示画面のスクリーンショットを取ることの妥当性を訴える大きな要因の 1 つと言える。

3. アプリケーションのローカライズミスを検出できる。
   2. と同様であるが、ローカライズデータはアプリケーション内部で、文字列データへの ID を用いて間接的に管理されていることが普通である。あるメニュー表示文

字列の ID は、日本語版でも英語版でも同じになる。この場合も実際の表示画面を記録しなければデータ変更を検出できない。

これらの利点は、人間のテスト担当者がテスト実行時に目視によって確認していることをコンピュータで実現できていることと同義である。

しかし、欠点として、ウィンドウのスクリーンショットを扱うための処理時間が大きいという問題がある。ディメンション $(sx, sy)$ のスクリーンショットをビットマップデータのまま扱うとそのデータ量は $O(sx \times sy)$ になる。もし 24bit カラーのビットマップデータを扱うとすれば、そのデータ量はかなりの大きさになることがわかる。通常、データ処理に必要な CPU 時間は、そのデータ量に比例したものになる。

ここで、データを圧縮形式で扱うことについて検討する。テストケース実行中は、イベント毎にテスト対象のアプリケーションのスクリーンショットを取りかつファイルに出力する。このときの処理時間は、ウィンドウハンドルを基にビットマップデータを参照し各種のログデータとともにファイルに格納するという処理に費やされる。このときに処理時間の多くはファイル I/O への書き込みアクセスである[3] 。圧縮形式を用いる場合は、これにビットマップデータの圧縮処理時間が追加されるが、圧縮後のデータ量は元の 1/2〜1/20 になるため、ファイル I/O へのアクセス時間が削減される。よって、トータルな処理時間は、実用上は問題のない処理時間内に抑えられると期待できる。本研究の環境 (Windows2000, Celeron500MHz, ディスクキャッシュオン) において、$400 \times 350$, 24bitColor のウィンドウに対してロギングに必要な処理時間は 50[ms] 以下という結果が得られている。

以上のことを踏まえ、スクリーンショットの保存形式に圧縮形式を利用することの利点を列挙する。

- ロギング処理時間の短縮
  扱うデータ量を小さくできるため、コンピュータシステムのボトルネックであるファイル I/O の負荷が抑えられることによるトータルな処理時間の短縮が期待できる。

- テストログデータ量のコンパクト化
  莫大な数となるテストログに圧縮形式を利用することで、そのデータ総量をコンパクトな状態で保存できる。

本研究のテストケース実行環境では、スクリーンショットの保存形式に保存品質 100% [4] のベースライン JPEG 形式を用いている。

[3] ディスクキャッシュについては考慮していない。

[4] JPEG 形式では、保存品質 100% においても不可逆データになる。スクリーンショットに高画質を求める場合、JPEG2000 形式のようなロスレス圧縮をサポートしているものを利用すると良い。

### 6.4.2 ロギングタイミングの問題

イベント毎にスクリーンショットを取る際に、どのタイミングでロギング処理を行うべきかという問題が生じる。もしイベントを発生した直後の描画イベントに反応してロギングを行うと図 6.3 のようにログを取りこぼす可能性がある。

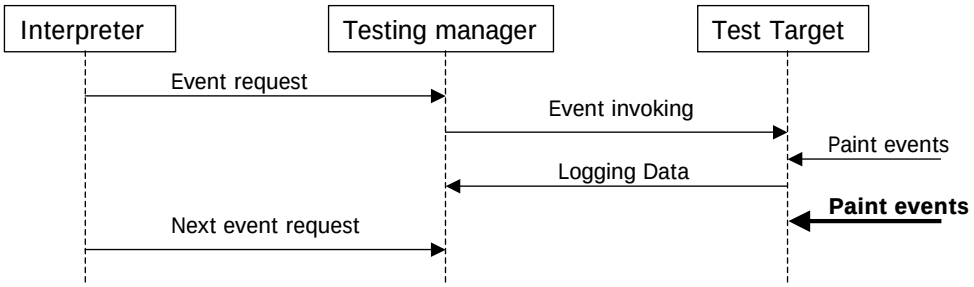


図 6.3: ロギングタイミングの問題

複雑な GUI アプリケーションでは、1 つの操作イベントに対して複数の描画イベントが発生することが普通であるウィンドウシステムによっては複数の描画イベントをできる限り 1 つの描画イベントにまとめようとするが、必ずしも 1 つになるとは限らない。。特に複数のウィジット同士が同期している場合は、あるウィジットの描画イベントが他のウィジットの描画イベントを発生させてしまう。

ログを取りこぼさないようにするには、ウィンドウシステムから全描画イベントの処理通知を受けた後にロギングすればよい。しかし、現在のイベントドリブン式の GUI 設計では、ウィンドウシステム自身も全ての描画イベントの発生が完了したかを知り得ない。

本研究では、各操作イベント発生間隔にインターバル時間を設けることで対応した (図 6.4)。

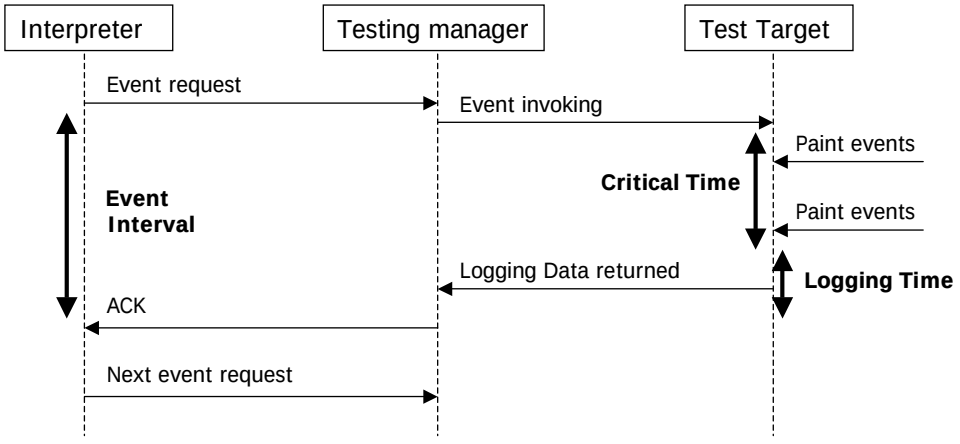


図 6.4: ロギングタイミングインターバル値

インターバル時間は、テスト対象のアプリケーションやテストを行う環境に依存した値であり、以下の式で表される。このインターバル時間は、ロギングの失敗が全く無いことは保証しないが、適切な値を設定することで、実用上の問題はない。

$$\begin{aligned} \text{Event Interval Time} &= \text{Critical Time(of event transaction)} \\ &+ \text{Logging Time(for capturing a screen and string the data)} \\ &+ \text{Margin Time}( = \text{a few msec} \ll \text{Critical Time}) \end{aligned}$$

**Critical Time** :
テスト対象の GUI アプリケーションやシステムの要求仕様で規定された値

- アプリケーション自身の処理時間
- 他システムとのトランザクション時間
- ウィンドウシステムの処理時間

**Logging Time** :
テスト実行時の環境の表示部解像度や I/O に依存した値

## 6.5 テストケース実行環境の実装

設計したテストケース実行環境の実装環境は以下の通りである。

**OS** :Windows 2000 SP3

**開発ツール** :Visual C++, Visual Basic SP6, GNU Bison 1.28, flex 2.5.4

行数には、コメント行、空行を含んでいる。ただし、開発ツールによる自動生成コードは含んでいない。

テストケーススクリプトインタプリタ (Visual C++, Bison, flex) クラス数 19 全行数 約 7000

テストマネージャ(Visual C++) クラス数 1 全行数 約 700

テストログ比較ツール (Visual Basic) クラス数 0 全行数 約 500

図 6.5 は、テストケース実行環境のテスト実行中の動作である。テストケーススクリプトを実行しながら、各イベント毎のスクリーンショットをテストログとして記録している。テスト対象のアプリケーションは、7 章の評価実験で用いているものと同一である。

テストログを比較するツールの動作例を、図 6.6 に示す。これは上記のテストケーススクリプト実行によって得ることができたテストログの比較解析中の動作を示している。比較結果は、HTML 文書として保存されるため、WEB ブラウザ等で閲覧可能である。

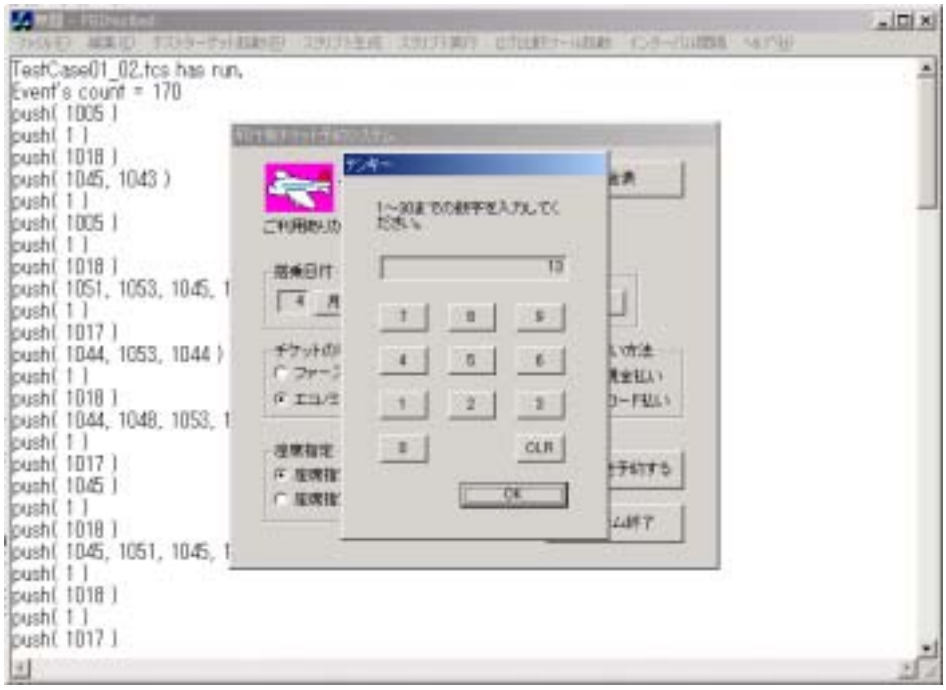


図 6.5: テストケース実行中動作例

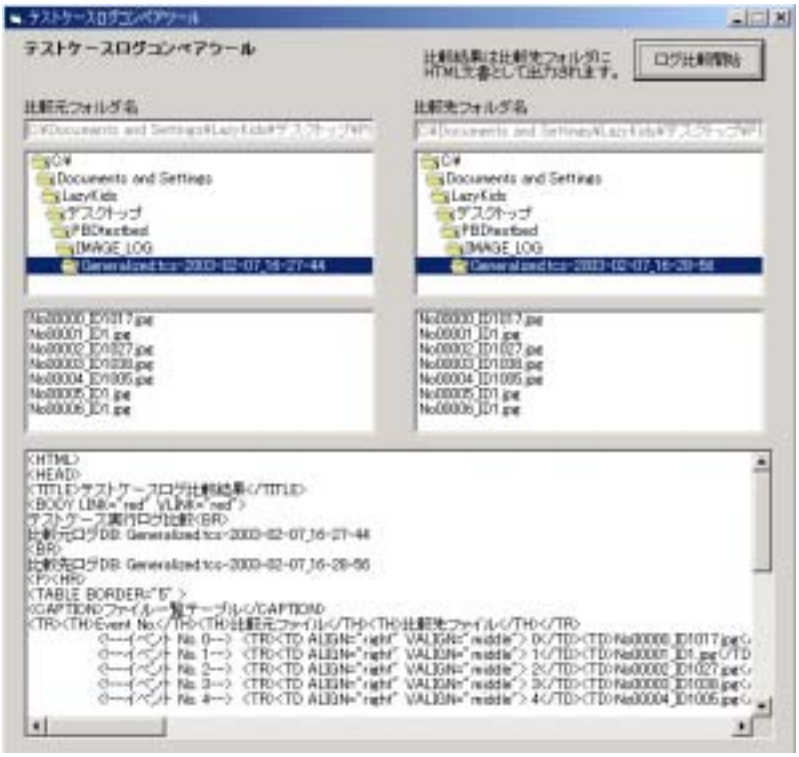


図 6.6: テストログ比較ツール動作例

# 第7章　評価

## 7.1　評価実験用アプリケーションとテストケースの概要

本研究のシステムを評価するにあたり、航空旅客券を予約するための GUI アプリケーションを作成し、テスト対象とした。以下にその GUI 仕様概要を示す。

- 設定可能項目：搭乗日付、搭乗便、チケットの種類、予約人数、座席指定の有無と位置、支払方法
- 押下可能ウィジット総数：40
- ウィンドウ総数：6
- ウィンドウ関連数：6

各ウィンドウのイメージとウィンドウ間の関連については図 7.1 に示す。

このアプリケーションに対するテストケースを 7 つ設定 (表 7.2) し、イベントのキャプチャリングと特徴抽出を行ったものが表 7.1 である。

表 7.1: テストケースのイベント数および特徴抽出数

| テストケース名 | キャプチャイベント [個] (ウィンドウ開閉、ボタン押下) | ボタン押下 [個] | L パターン [個] | T パターン [個] |
|---|---|---|---|---|
| TestCase 1 | 54 | 36 | 6 | 7 |
| TestCase 2 | 256 | 170 | 21 | 24 |
| TestCase 3 | 39 | 19 | 0 | 5 |
| TestCase 4 | 108 | 76 | 5 | 1 |
| TestCase 5 | 18 | 12 | 2 | 1 |
| TestCase 6 | 76 | 42 | 4 | 5 |
| TestCase 7 | 60 | 38 | 5 | 3 |

## 7.2　一般化テストケーススクリプト生成実験

7 つのテストケースそれぞれに汎化推論を行い、一般化テストケーススクリプトを生成させた。それぞれの出力結果から汎化による可読性と再利用性の向上効果について以下に述べる。

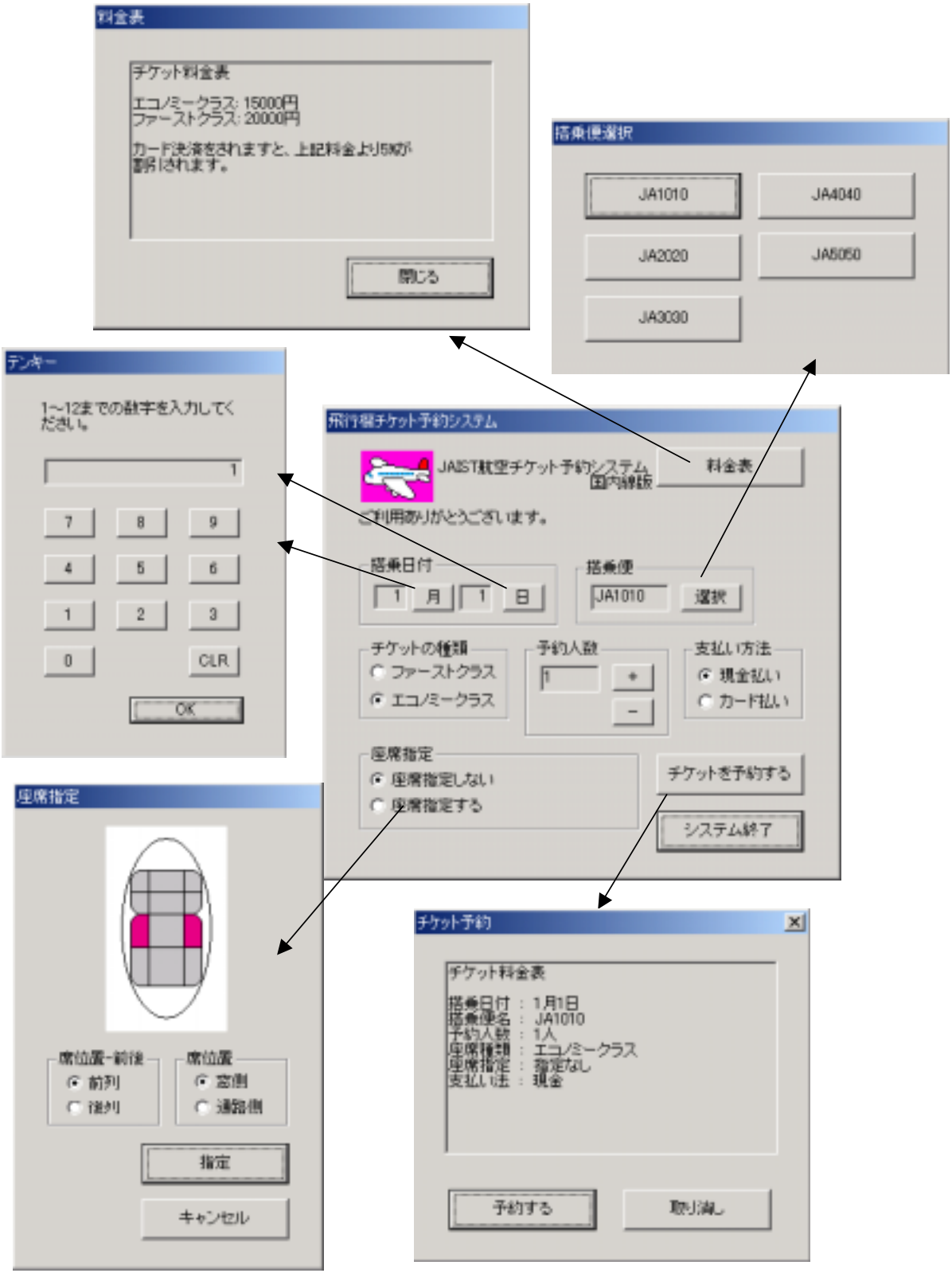


図 7.1: テスト対象のウィンドウ関連

表 7.2: テストケース内容

| テストケース名 | テストケース内容 |
|---|---|
| TestCase 1 | チケット予約に関して一通りの機能動作確認 |
| TestCase 2 | 日付設定機能チェック<br>・搭乗予約日が正しく設定できるか？<br>・存在しない日付設定を拒否できるか？ |
| TestCase 3 | 搭乗便設定機能チェック<br>・搭乗便 (全 5 便) の設定が行えるか？ |
| TestCase 4 | 人数設定機能チェック<br>・設定範囲内 (1〜10) での予約ができるか？<br>・範囲外の設定を拒否できるか？ |
| TestCase 5 | チケットの種類と座席指定のイラスト表示が<br>連動しているか？(全 8 通り)<br>・ファーストクラス、エコノミークラス<br>・前列、後列<br>・窓際、通路側 |
| TestCase 6 | 座席指定設定が予約画面に反映されるか？<br>・座席指定する設定（全 8 通り） |
| TestCase 7 | 全てのラジオボタン設定が予約画面に反映されるか？<br>・チケットの種類（2 通り）<br>・支払い方法（2 通り）<br>・座席指定の有無 (2 通り)<br>・全ての組み合わせ (8 通り) |

### 7.2.1 サブルーチン化による可読性と再利用性の向上

各サブルーチンは、各 T パターンに一致する。T パターンは機能操作上分離可能なものとして抽出されたパターンである。つまり、あるウィンドウをオープンし、なんらかの操作後にそのウィンドウをクローズするという一連の操作シーケンスは、それ自体閉じた操作パターンであることを意味する。そのため、T パターンをそのままサブルーチン化することは妥当である。

テストケース No.5 の汎化前のイベント列と一般化テストケーススクリプト出力結果を図 7.2 に示す。

```
Window : Name = 飛行機チケット予約システム, Status = OPEN
Button Click : ID = 1004
Window : Name = 座席指定, Status = OPEN
Button Click : ID = 1005
Button Click : ID = 1002
Button Click : ID = 1003
Button Click : ID = 1
Window : Name = 座席指定, Status = CLOSE
Button Click : ID = 1006
Button Click : ID = 1004
Window : Name = 座席指定, Status = OPEN
Button Click : ID = 1005
Button Click : ID = 1002
Button Click : ID = 1003
Button Click : ID = 1
Window : Name = 座席指定, Status = CLOSE
Button Click : ID = 1
Window : Name = 飛行機チケット予約システム, Status = CLOSE
```

```
// Generalized Script

func main(){          //WINDOW-OPEN 飛行機チケット予約システム
  var L1 = [ 1006 ];

  T0();
  push( L1 );
  T0();
  push( 1 );          //WINDOW-CLOSE 飛行機チケット予約システム
}

func T0(){            // T0
  var L0 = [ 1005, 1002, 1003 ];

  push( 1004 );       //WINDOW-OPEN 座席指定
  push( L0 );
  push( 1 );          //WINDOW-CLOSE 座席指定
}
```

図 7.2: テストケース No.5 のイベント列と一般化テストケーススクリプト出力結果

このスクリプトでは、T0 パターンがサブルーチン化され、L0 パターンがリストデータとして初期化宣言[1] されている。そのため、L0 のスコープが明確でこの操作列データに修正を加えた場合の影響範囲が T0() 内のみであることが容易に判断できる。main() と L1 パターンの関係でも同様である。そのため、手動でテストケースを再編集する場合の作業が行いやすい。これは再利用性が向上した効果と言える。

再利用性の効果は、サブルーチンが複数生成されるテストケースの場合はより顕著に表れる。サブルーチン化された操作パターンは呼び出し元との関連が明確で呼び出しの整合性が取れている。そのため、それらの呼び出しを自由に変更して、異なるテストケースを手動で容易に作成できる。これと同様のことを汎化前のイベント列に対して行おうとすれば、呼び出し側との整合性確認が必要となり、非常に手間のかかるものとなる。

[1] スクリプト中に現れる即値は、ボタンウィジットの ID を表す。

また、main() を見ると、T0() が 2 回呼び出されている。この部分は今回の汎化推論のループ化対象外の繰り返しであるが、この部分がループ化できる繰り返しであることが一瞥して認識できる。そして、必要ならば手作業でループ化して、2 回以上ループ操作するテストケースに作り直すことができる。これは可読性向上の効果であり、手動による再編集作業の効率向上につながると考えることができる。

### 7.2.2 ループ化による可読性と再利用性

今回の汎化処理でのループ化推論時に認識するパターンは、特徴抽出したイベントフローツリーにおいて、重複 T ノードを子に持つ L ノードが連続するパターンである。ループ構文には for 文を用いた。

テストケース No.7 は、メインウィンドウの各設定項目が予約確認画面に反映しているかを、繰り返しテストする操作である。これはテスト工程においてよく行われる操作パターンの 1 つである。テストケース No.7 の一般化テストケーススクリプト出力結果からループ化

```
func main(){     //WINDOW-OPEN 飛行機チケット予約システム
  var L0 = [ 1006 ];
  var L1 = [ 1010 ];
  var L2 = [ 1024 ];

  var i, L ;        // Temporary for loop
  L = [ L0, L1, L2 ];
  for( i = 0 ; i < length( L ) ; i ++ ) {
                push( L[i] );
                T0();
  }

  push( 1 );       //WINDOW-CLOSE 飛行機チケット予約システム
}
```

図 7.3: テストケース No.7 の一般化テストケーススクリプトでのループ化部分

変数 L 内の各要素 L0,L1,L2 が、T0() の呼び出しとの関連が深いと判断されたため、ループ化が行われている。これは実際にテストケースをテスト対象上で操作したときの意図とも一致する。もし、この呼び出し順以外のテストケースに再編集したいならば、L の要素の記述順を変更したり、L=[L, L]; を for ループの直前に挿入して、2 回連続ループに変更

することもできる。このようにT0()との関連性の深いL内の要素に対して重点的なテストを行うテストケースに再編集が容易である。また、T0()との関連性が低いLパターンを追加して、T()の呼び出しに不具合が生じないかの動作検証を行うテストケースに変更することも容易である。ループ内部に条件文を追加して複雑な呼び出しを行う場合も、L内の要素とT0()の関連性が明確なため作業が行いやすい。どの場合も、ループ操作が行われている個所が容易に識別できることによって、再利用性が向上していることを示している。

### 7.2.3 Lパターンを集合としての扱うことによる再利用性

いままでの例でもLパターンの集合としての扱いが可読性と再利用性に貢献していることがわかるが、ここではLパターン自体に編集を行うことによる再利用性について述べる。

Lパターンを定義している各変数は、サブルーチンスコープのみを持つローカル変数としてスクリプト出力を行っている。そして、各サブルーチンは1つのウィンドウにマップされているため、各変数に定義されている値はそのサブルーチンにマップされたウィンドウ上で押下可能なウィジットIDを指している。つまり、Lパターンを定義した変数とそのの各要素をスコープ内で使用している限りは、ウィンドウとウィジットIDの関連の整合性が保証されていることを示す。そのため、関数内の任意の位置で、push()関数を用いて呼び出したり、各変数の順番を変更したり、変数同士を結合したりといった編集を行うことができる。

これと同様のことを汎化前のイベント列に対して行おうとすれば、整合確認が必要となる。

## 7.3 ゆらぎテストケーススクリプト生成実験

7つのテストケースそれぞれにゆらぎテンプレート適用を行い、ゆらぎテストケーススクリプトを生成させた。それぞれの出力結果から各発生イベント数を求め、イベントゆらぎ率を評価した。

イベントゆらぎ率は、テンプレートによって発生順序が変化したイベントの個数を元のイベント数で割ったもので、以下の式で与えられる。なお、発生順序が変化したイベント以降に続くイベント列も発生順序が変化しているものと見なしている。

$$\text{イベントゆらぎ率} = \frac{\text{発生順の変化したイベント数}}{\text{元のイベント数}}$$

今回、適用したゆらぎテンプレートは以下の2つである。

**テンプレート** No.1:同一ボタン連続押下テンプレート

– 同じボタンを複数回連続押下しても、不具合が生じないかをテストする
– 連続押下回数は、スクリプト内の変数で定義 (今回は 3 回に設定)
– L パターン内の各要素に適用

**テンプレート** No.2:ボタン押下順逆転テンプレート

– ボタンの押下順を逆転して、不具合が生じないかをテストする
– L パターン内の各要素に適用

これらのテンプレートを適用し、元のテストケースにゆらぎを与えて生成した各スクリプトのイベント合計数から、イベントゆらぎ率を算出したのが、図 7.4 である。

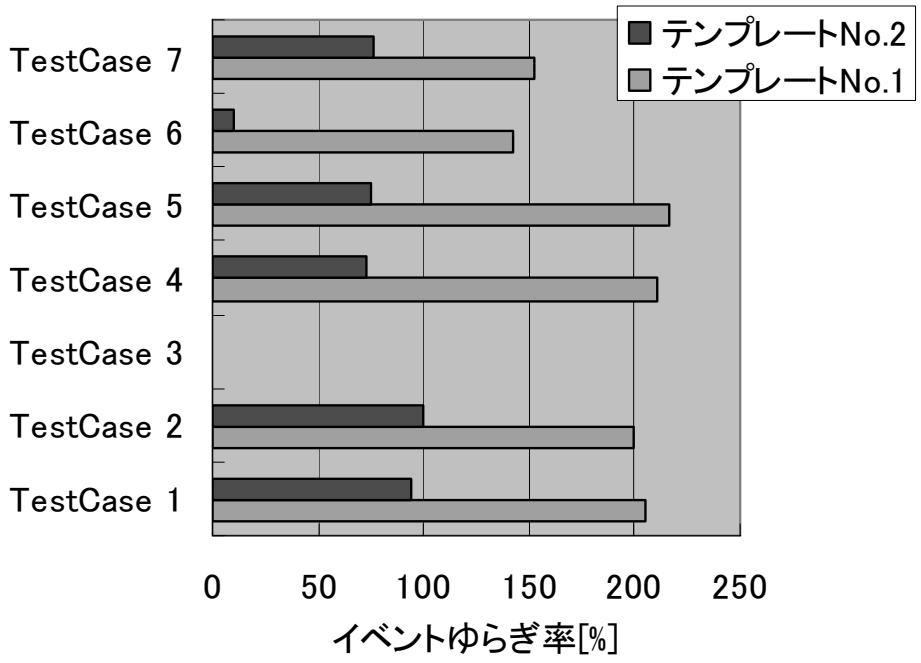


図 7.4: ゆらぎテンプレートによるイベント発生数とイベントゆらぎ率

図を見ると、テストケース No.3 以外ののテストケースに対しては 150%〜300%のゆらぎ率が得られていることがわかる。テストケース No.3 へのゆらぎ率が 0%であるのは、テストケース No.3 には L パターンが存在せず、適用した 2 つのテンプレートの適用先が L パターンであるためである。よって、T パターンに適用するタイプのテンプレートの必要性があることがわかる。

テンプレート No.2 の適用によるゆらぎ率の最高値は 100%であるが、これは元のイベントの個数を増加するタイプのテンプレートではないためである。

今回は 2 つの単純なテンプレートを適用しただけであるが、より多くのテンプレートの適用や有用性の高いテンプレートを適用してゆらぎ率を向上することで、テストカバレッジの向上を期待できる。

# 第8章　議論

## 8.1　汎化推論の改良点

テストケースから一般化スクリプトを自動生成させる際に、有用性が認められると同時に現在の汎化推論では対応できない事例がいくつか確認された。以下にそれらの事例を示す。

### 8.1.1　類似したサブルーチンの統合

本研究のヒューリスティック推論によるサブルーチン化では、その粒度が細かく生成されがちになってしまう。それは、同一の機能階層へ遷移する場合でも、その階層内での操作シーケンスが異なれば、別々のサブルーチンとして認識しているためである。しかし、いくつかの場合では、1 つのサブルーチンへ統合した方が再利用性が向上する時があると考えることができる。

図 8.1 は、テストケース No.6 の一般化テストケース出力結果の一部である。T3() と T4() は、異なるサブルーチンとして生成されているが、同一の機能階層への遷移する操作シーケンスとして統合する方が再利用性が高いと考えることができる。

TCS 言語では、サブルーチンに引数を渡すことができるため、その引数の値を用いることで、図 8.2 のように統合できる。

例のような単純な類似性を示すサブルーチン同士であれば、現状の汎化推論に統合推論機能を組み込むことは容易だが、判断の困難な類似性をもつ場合は、その類似性部分を特定するのは難しい。もし、類似性部分を特定しその結合度を推論できるのであれば、条件分岐によってサブルーチン統合ができると考えることができる。

### 8.1.2　T パターンへのループ化推論

7.2.1 節でも述べたが、図 7.2 の T0() の呼び出しに関係する部分はループ化が可能である。また、図 8.3 のような連続したサブルーチン呼び出し個所は、T1() の呼び出しとそれに関連が深いと判断できるため、ループ化の利点がある。

現在はこの部分へのループ化はできないが、これらは T パターンに関するループ化推論ルールを追加することにより実現が可能である。

```
func T3(){                  // T3
  var L1 = [ 1002 ];

  push( 1004 );             //WINDOW-OPEN 座席指定
  push( L1 );
  push( 1 );                //WINDOW-CLOSE 座席指定
}

func T4(){                  // T4
  var L2 = [ 1002, 1005 ];

  push( 1004 );             //WINDOW-OPEN 座席指定
  push( L2 );
  push( 1 );                //WINDOW-CLOSE 座席指定
}
```

図 8.1: サブルーチンの統合可能個所

```
func T3'( way ){          // T4 was integrated with T3
  var L1 = [ 1002 ];
  var L2 = [ 1002, 1005 ];
  var L = [ L1, L2 ] ;

  push( 1004 );           //WINDOW-OPEN 座席指定
  push( L[ way[ 0 ] ] );
  push( 1 );              //WINDOW-CLOSE 座席指定
}
```

図 8.2: サブルーチンの統合例

```
func main(){   //WINDOW-OPEN 飛行機チケット予約システム
  T1();
  T2();
  T1();
  T3();
  T1();
  T4();
  push( 1 );    //WINDOW-CLOSE 飛行機チケット予約システム
}
```

図 8.3: ループ化可能サブルーチン呼び出し個所

### 8.1.3 可読性の向上

汎化推論を行う目的の 1 つに、単純な時系列である操作シーケンスの可読性向上がある。生成実験から可読性が向上した結果を得ることができたが、さらなる向上の余地がある。それらを以下に列挙する。

- ウィジット ID の名前付けによる抽象化
  生成されるスクリプトでは、ウィジット ID を即値のまま出力しているため、その ID がどのようなウィジットであるかを理解できない。 通常、アプリケーションのソースコード内では、ウィジット ID を即値ではなくマクロ名等で表現している。それらの対応付けの情報を利用すれば、可読性向上につながると考える。

- サブルーチン呼び出し個所へのコメント追加
  サブルーチン名は、特徴抽出 T パターンの名前を用いて付ている。可読性向上のためには、サブルーチンの内容に適した名前付けが必要である。

## 8.2 ハイブリッドテスト実行へのゆらぎテンプレートの有用性

テストケース作成者の経験則やアプリケーションの特性に基づいたテスト操作プランを記述したゆらぎテンプレートを適用することは、見落としがちなテストケースを補完生成することに効果がある。

それだけでなく、ゆらぎテンプレートは、自動テストと手動テストのハイブリッドなテスト実行をスムーズに行うことにも有用性がある。図 8.4 にゆらぎテンプレートとテストカバレッジ達成度の関係を表したものを示す。

1 つのゆらぎテンプレートには、どのようにゆらぎを与えるかというテスト操作プランが 1 つだけ記述されている。そのため、ゆらぎテンプレート適用によって生成されたテストケースが、どのようなテストカバレッジを達成するかを判断する目安とすることができる。そして、その目安から不足している部分の手動テストを行う。この作業は、一般化テストケーススクリプト生成を行うのと同時に、ゆらぎテンプレート適用元のテストケースを増やしていることを意味する。この作業を交互に行うことで、有用なテストケースを優先してテストカバレッジを達成していくことができる。これは、他の自動生成手法 (3.2 節) にはない有用性である。

しかし、ゆらぎ率と、テストカバレッジが等価な評価方法ではないため、そのテンプレートの適用によってテストカバレッジの達成度がどの程度向上したかを検証できないことも研究の過程で判明した。それは、一般に知られた GUI 向けの標準的なテストカバレッジが存在しないため、テンプレートの有用性を定量的に検証することが困難なためである。本研究では、標準的な指針となるべき GUI テストカバレッジを示すことはできなかったが、今後の課題として検討を行っていきたい。

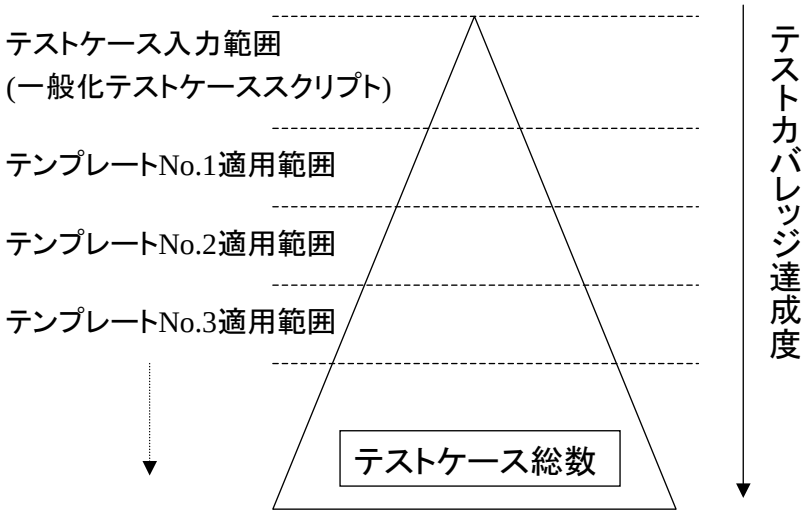


図 8.4: ゆらぎテンプレートとテストカバレッジ達成度

## 8.3 GUI テストケーススクリプト自動生成における例示プログラミング手法の意義

7 章の生成実験から、効果がある事例といくつかの問題点があることがわかった。特にその問題点を以下に示す。

- 多くのサブルーチン生成される
  粒度の細かいサブルーチンが生成されやすく、類似したサブルーチンでも異なるものとして認識されてしまう。

- 羅列されたサブルーチン呼び出しコードが発生する
  T パターンに関するループが行われないため、単なるサブルーチン呼び出しとして出力されてしまう。

- ゆらぎテンプレートの有効度が十分に検証できない
  ゆらぎ率とテストカバレッジは等価な評価法でないため、操作シーケンスに関するテストカバレッジの評価が困難で、定量的な検証ができない。

この理由として、例示プログラミングでは、ユーザの意図を完全に認識することが困難であることがあげられる。このことは、4.3 節でも述べられている。

しかし、テストケース生成のアルゴリズムに条件が合致した場合は、効果がある結果が得られている。そのため、実際の GUI アプリケーションを操作するだけで、テストケーススクリプトを自動生成することがテストケース生成を支援できる可能性をもっていると考える。今回の実験は小規模であったが、大規模な生成実験を行い、テストケースに現れ

るシーケンスパターンや再利用性やゆらぎの与え方等の統計データを調査し、多くの操作シーケンスに効果がある条件を導くことができれば、その自動生成器の有用性は高いと考える。

特に、GUI アプリケーションの開発時に利用することの有用性は高い。通常、開発者は、まとまったコード記述を行うごとに、GUI アプリケーションを実際に操作してコード記述に不具合がないかを確認する。そして、問題がなければ、次のコード記述を行う。

この作業に自動生成器を導入すれば、開発者が通常どおりに開発を進めていくだけで、テストケースとテストオラクルが生成されることになる。これらは、開発時に波及不具合がないかをテストすることに用いることができるし、テスト工程に流用することもできる。そして、ゆらぎテンプレートを適用したテストケースによって、見落としがちなテストケースを補完生成して、テストカバレッジ達成度を高めることができると考える。

## 8.4 特徴抽出法の発展性

今回の研究で、ユーザの操作シーケンスから L パターンと T パターンという、GUI の構造特徴に注目した特徴を抽出することで、テストケースの自動生成に効果のあることがわかった。

この特徴抽出法を発展させることにより、汎化推論の機能や精度を高めることが可能になると考える。また、今回の汎化推論で行うことができなかった、条件分岐推論の導入を実現できる可能性がある。

### 8.4.1 発展の方向性

本研究ではユーザ操作上の意図を認識するために、L パターンと T パターンという GUI の機能的階層特徴に注目した特徴の抽出を行っている。

今回の研究では、この構造特徴の利用効果があるという結果を得ることができたが、さらに T パターン間や L パターン間の関連度を算出することで、操作シーケンスにおけるユーザの意図を詳細に得ることができると考える。各関連度は推移確率として表現できるため、様々な数学上の手法を特徴抽出処理に応用できる。例えば、各関連度に関する推移確率行列の固有値は、その操作シーケンス上の注目すべき機能やイベントを求めるために利用できる。

以下に、特徴抽出を発展させるためのアイディアとして、以下の 3 つを示す。ただし、それらの有用性の検証は今後の課題とする。

### 8.4.2 T パターン推移確率

ユーザが機能間を移動する際には、その意図に応じた移動法則があると考えることができる。ある機能階層に遷移する確率は、その操作シーケンスにおいて、その直前に操作された機能階層によって決まっている。この特徴は、GUI の機能的階層特徴によってもたらされるものだが、各確率は操作シーケンス上のみに表現されるため、ユーザの操作意図が含まれていると判断できる。このような機能階層間の遷移関連度を T パターン推移確率として表現できる。

推移確率の高い T パターンを含むループ化対象部分を汎化の効果が高いと判断し、より深い推論を行うといったことができるのではないかと我々は予測している。また、後述する LT パターン推移確率を併用することで、効果的なループ化対象部分を検出できる可能性もある。

### 8.4.3 LT パターン推移確率

L パターンと T パターン (以下、LT パターン) の関連度を示す値として、LT パターン推移確率を提案する。現在の汎化推論では、LT パターンの関連性が認識できないため、、類似した LT パターンに対してのループ化やサブルーチン化が行えない。実現するためには、異なる部分を条件実行するための分岐推論をする必要がある。その分岐推論を行う際に、LT パターン推移確率が利用できると考える。

### 8.4.4 L パターン内イベント推移確率

L パターンは、イベント間に含まれるユーザの操作意図を認識するために、操作シーケンスから分割した部分要素である。しかし、ユーザの意図を詳細に得るためは、より細かい粒度に分割した方が有用である。その分割点を求めるために利用できそうなのが、L パターン内イベント推移確率である。

L パターン内イベント推移確率は、各 L パターン内のイベント間の関連度を表している。そのため、各 L パターンの推移確率を他の L パターンのイベント推移現確率と比較することで、関連の深いイベントシーケンスを細粒度の L パターンへ分割する情報をして利用できると考える。

また、L パターン内イベント出現確率を、各 L パターンの類似度の算出に利用することで、同じ機能階層に遷移するにもかかわらず異なる T パターンとして認識されてしまった、複数の T パターンを 1 つの T パターンとして統合することが可能になると考えることができる。

## 8.5 テストケース実行環境のGUIの「見た目」の不具合検出への効果

本研究では、GUIの「見た目」不具合を検出できるテストケース実行環境が有効であると考え、それを実現するために、操作イベント毎にウィンドウのスクリーンショットを取るテストケース実行環境を実装した。

このテストケース実行環境で出力されるテストログは、GUIの「見た目」の状態遷移を表現しており、テスト対象のアプリケーションに不具合があった場合には、どのイベント発生時にその不具合が生じたかを特定できる。

テストケースNo.1〜No.7のそれぞれに関係するウィジット10個を任意に選び、そのイベントハンドラをコメントアウトして、故意に混入させた不具合を検出できるかを試した。評価実験によって生成された一般化テストケーススクリプトを用いて、テストオラクルと比較したところ、全ての不具合が検出され、どのイベント発生時に不具合が生じたかを特定できた。(動作例：図8.5)

このことから、6.1節で示したテストケース実行環境構築の要件がガイドラインとして機能することがわかった。

図 8.5: ログ比較結果出力例

# 第9章　おわりに

## 9.1　まとめ

提案したGUIテストケース自動生成器を実装し、評価実験用アプリケーションにおいてテストケース生成実験を行った。実験結果より以下に示す効果を得ることができた。

- 一般化テストケーススクリプト生成実験
  サブルーチン化によるLパターンやTパターンの影響範囲の明確化やループ化による繰り返し部分の明確化といった可読性と再利用性が向上した。

- ゆらぎテストケーススクリプト生成実験
  元の操作シーケンスからゆらぎテンプレートのテスト操作プランに則ったスクリプトが生成され、元の操作シーケンスの1.5から3倍のイベントが発生された。

このことから、テスト対象のアプリケーションを実際に操作するだけで、生成システムが自動的に可読性を高めた再利用性のあるテストケースを生成したり、テストカバレッジの高いテストケースを生成する自動生成器が、テストケース生成を支援できる可能性があることがわかった。

また、操作イベント毎のウィンドウのスクリーンショットをテストログ出力できるテストケース実行環境を設計実装した。いくつかのウィジットのハンドラをコメントアウトした不具合を故意に混入させて、それらのウィジットが動作しない不具合を検出できることを確認できた。

しかし、評価実験によって、いくつかの問題点があることも判明した。以下にそれらを示す。

- 粒度の細かいサブルーチンが生成されやすく、多くのサブルーチン呼び出しが発生する。

- Tパターンに関するループ化が行われないため、サブルーチン呼び出しの羅列発生により可読性が下がるときがある。

- テストカバレッジの評価が困難で、ゆらぎテンプレートの有効度が十分に検証できない。

## 9.2 結論

本研究では、GUI テスト自動化ツールがテスト自動実行によりテスト実行コスト削減に効果を上げているにもかかわらず、その実行に必要なテストケーススクリプトの自動生成に関しては有効な方法を実現していないことに注目し、例示プログラミング手法を用いたテストケーススクリプト自動生成法を提案した。この生成法に基づく自動生成器の実装を行った。

結果として次の 3 点をを得ることができた。

- 推論の条件に合致した部分は、可読性と再利用性の向上効果が見られた。
- 条件に合致しない部分に関しては、逆効果になる場合がある。
- ゆらぎテンプレートの有効度を十分に検証することが困難である。

部分的な効果しか得られなかったが、テスト対象のアプリケーションを実際に操作するだけで、生成システムが自動的に可読性を高めた再利用性のあるテストケースを生成したり、高いカバレッジを達成しつつ有用なテストケースを優先的に生成できる生成器がコスト削減を支援できると考える。

しかし、実用性を確認するためには、大規模な GUI アプリケーションを用いた継続的な実験が必要なこと、多くのテストケースからスクリプトを生成すること、経験豊富なテスト作成者の意見を反映したゆらぎテンプレートを作成することなどの現実の開発環境に近い条件で実験する必要がある。

また、GUI の構造的特徴である機能的階層に注目したパターン、L パターンと T パターンが操作シーケンス上のユーザの意図を認識するために利用できることがわかったと同時にいくつかの検討点も確認した。その主なものは次の通りである。

- 類似したパターンの統合の必要性
- 条件分岐推論の必要性
- 可読性の向上の余地

以上のことを踏まえ、本研究では、いくつかの問題点や検討点があるものの、例示プログラミング手法が GUI テストケース自動生成器に部分的ではあるが有用性であるという結論に達した。

## 9.3 今後の課題

すでに議論したことも含め、今後の課題を以下に示す。

- 特徴抽出法に確率論を導入することによる発展的改良

- アプリケーションに依存した特徴への対応
- 有用なゆらぎテンプレートライブラリの構築
- 各種プラットフォームでの例示プログラミングシステム実装技術の構築
- インターバル時間に代わるロギングタイミング問題の解決法
- GUI テスト自動化ツールとしての完成度向上

# 謝辞

本論文を執筆するにあたり御指導を賜った権藤克彦助教授に深く感謝いたします。研究を進めるにあたり有益な助言を下さった、片山卓也教授、青木利晃助手に感謝いたします。

また、片山・権藤研究室の諸兄には、ゼミでの討議はもちろん、学生生活でもお世話になりましたことを感謝いたします。

その他の学生生活を豊かにしていただいた多くの方に感謝いたします。

# 参考文献


[1] Allen Cypher Eager : Programming repetitive tasks by example, ACM 1991

[2] Allen Cypher, Watch What I Do: Programming by Demonstration, http://www.acypher.com/wwid/

[3] Atif M. Memon, Martha E. Pollack, Mary Lou Soffa, Plan Generation for GUI Testing, American Association for Artificial Intelligence 1999

[4] Atif M. Memon, Martha E. Pollack, Mary Lou Soffa, Using a Goal-driven Approach to Generate Test Cases for GUIs, ACM 1999

[5] Atif M. Memon, Martha E. Pollack, Mary Lou Soffa, Automated Test Oracles for GUIs, ACM 2000

[6] Atif M. Memon, Martha E. Pollack, Mary Lou Soffa, Coverage Criteria for GUI Testing, ACM 2001

[7] Atif M. Memon, Martha E. Pollack, Mary Lou Soffa, Hierarchical GUI Test Case Generation Using Automated Planning, 2001

[8] Boris Beizer, Software Testing Techniques Second Edition ソフトウェアテスト技法, 日経BP 出版センター1994

[9] Cameron Laird, Kathryn Soraiz, Testing GUI applications, http://www.itworld.com/AppDev/1262/UIR010316testinggui1/ Unix Insider 2001

[10] Cem Kaner, Jack Falk, Hung Quoc Nguyen, テスト技術者交流会訳, Testing Computer Software 基礎から学ぶソフトウェア, 日経BP 社2001

[11] Craig N. Mills, Maria T. Potenza, John J. Fremer, William C. Ward, Computer-Based Testing Lawrence Erlbaum Associates, Inc., 2002

[12] David L.Maulsby, Ian H. Witten INDUCING PROGRAMS IN A DIRECT-MANIPULATION ENVIRONMENT, ACM 1989

[13] Elfriede Dustin, Jeff Rashka, John Paul, 向井 清訳, Automated Software Testing: Introduction, Management, and Performance 自動ソフトウェアテスト 導入から、管理・実践まで, ピアソン・エデュケーション2002

[14] Gregg Rothermel, Mary Jean Harrold, A Safe, Efficient Regression Test Selection Technique, ACM 1997

[15] Guanglun Yu, Record and Playback for Java Software Watermarking, 2001

[16] Kent Beck, Test-Driven Development:By Example, Addison-Wesley Pub Co. 2002

[17] Lee White and Husain Almezen, Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences, ISSRE 2000

[18] Myers B.A., Demonstrational Interfaces: A Step Beyond Direct Manipulation, IEEE Computer vol.25 No8 pp.61-73 1992

[19] L. R. Kepple, The black art of GUI testing, Dr. Dobb's Journal of Software Tools 19(2);40, February 1994

[20] Paul Gerrard, Testing GUI Applications, EuroSTAR '97, 24-28, 1997

[21] R.K. Shehady and D.P. Siewiorek, A Method to Automate User Interface Testing Using Variable Finite State Machines, 1997

[22] K. C. Tai, Y. C. Young, Synchronizable Test Sequences of Finite State Machines, 1995

[23] Thomas Arnold, Visual Test 4.0 技術資料, St Labs Inc.

[24] Thomas J. Ostrand, Aaron Anodide, Herbert Foster, Tarak Goradia: A Visual Test Development Environment for GUI Systems, ISSTA98 1998

[25] 服部 隆志, 半順序優先を用いた例と制約による編集 インタラクティブシステムとソフトウェア I: 日本ソフトウェア科学会 WISS'93, pp. 233-240, 近代科学社, 1994

[26] 増井 俊之, 予測/例示インタフェース,
http://www.csl.sony.co.jp/person/masui/

[27] 増井俊之, 太和田誠, 操作の繰返しによるマクロの自動生成, 情報処理学会ヒューマンインタフェース研究会研究報告 93-HI-48, Vol.93, pp. 65-72, May 1993

[28] 増井俊之, 適応/予測型テキスト編集システム, インタラクティブシステムとソフトウェア II: 日本ソフトウェア科学会 WISS'94, pp. 145-154, 近代科学社, 1994

[29] 増井俊之, 中山健, 操作の繰返しを用いた予測インタフェースの統合, コンピュータソフトウェア, Vol.11, No.6, pp. 484-492, November 1994.

[30] 増井俊之, ペンを用いた高速文章入力手法, インタラクティブシステムとソフトウェアIV: 日本ソフトウェア科学会 WISS'96, pp. 51-60, 近代科学社, December 1996

[31] 増井俊之, 予測/例示インタフェースの研究動向, コンピュータソフトウェア, Vol.14, No.1, pp. 1-16, May 1997

[32] 増井 俊之 東京大学, 予測/例示インタフェースシステムの研究, November 1997

[33] 増井俊之, 動的パタンマッチを用いた高速文章入力手法, インタラクティブシステムとソフトウェアV: 日本ソフトウェア科学会 WISS'97, pp.81-86, 近代科学社, December 1997

[34] 宮本健司 法政大学工学部, LIMIT:極限コンパスを実装した作図エディタ

[35] 宮下 健, 松岡 聡, 高橋 伸, 米澤 明憲, 複数の視覚的例による直接インターフェイスの対話的実現, インタラクティブシステムとソフトウェアI: 日本ソフトウェア科学会 WISS'93, pp. 241-248, 近代科学社, 1994

[36] 中山 健, 宮本 健司, 川合 慧, コマンド履歴からの動的スクリプト生成, インタラクティブシステムとソフトウェアII: 日本ソフトウェア科学会 WISS'94, pp. 155-164, 近代科学社, 1994

[37] 瀬下 順一, 加登 基二, 平川 正人, 市川 忠男, ユーザ操作系列からのプログラム・コンポーネントの抽出インタラクティブシステムとソフトウェアI: 日本ソフトウェア科学会 WISS'93, pp. 249-256, 近代科学社, 1994

[38] 杉浦 淳, 古関 義幸, 例示プログラミングにおけるマクロ定義の簡略化, インタラクティブシステムとソフトウェアIV: 日本ソフトウェア科学会 WISS'96, pp. 101-110, 近代科学社, 1996

[39] 銀座企画, 最新 Perl/CGI ハンドブック 秀和システム 2001

[40] ハーバート・シルト, 標準講座 C++, 翔泳社 1999

[41] ハーバート・シルト, 標準講座 STL, 翔泳社 1999

[42] John R. Levine, Tony Mason, Doug Brown, lex & yacc, A NUTSHELL HANDBOOK 1994

[43] A. V. エイホ, R. セシィ, J. D. ウルマン, コンパイラ I & II, サイエンス社 1990

[44] J. ケネス・シュルティス, LaTeX 実用ハンドブック, トッパン 1995

# 付 録A　テストケース・スクリプト言語 (TCS：TestCase Script language)仕様概要書

## A.1　変数型とデータ型

TCS では、変数には型は存在せず、データ自身に型を持つ。
ただし、変数は明示的に宣言しないと使用できない。

■データ型の種類
・スカラ型
　　NULL 型
　　整数型（-2147483648〜2147483647）
　　倍精度浮動小数型（1.7x10E-308〜1.7x10E308）
　　文字列型
・リスト型
　　スカラ型の値を順序連結した集合
　　内部的には、２分木として表現され、
　　リストの入れ子表現も可能。
　　例：[ "A", [ 1, 2, 3 ], "B", "C" ]

例:

```
123                          // 整数型
-456
3.1415923                    // 倍精度浮動小数型
6.0221367*10e23
"Hello, TCS"                 // 文字列型
[ 123, 3.14, "Hello" ]       // リスト型
```

■変数の宣言
var ステートメントで宣言する。初期化付き宣言も可能。

例:

```
var a ;
var i, j, k ;
var x = 3, y = "abs", z = [ 1, 2, "abc" ] ;
```

■変数のスコープ
ブロック（{}で囲まれた範囲）内で宣言された変数は、そのブロック内のスコープを持つ。
ブロックで囲まずに宣言された変数は、トップレベルスコープを持ち、トップレベルスコープに属する変数は、他にTCSのシステム変数も含まれる。

注)TCSでは、厳密な意味でのグローバル変数はないが、トップレベルスコープ変数をあたかもグローバル変数であるかのようにしてプログラミングを進めることが可能である。

## A.2 演算子

・基本的には、同種ペアオブジェクト同士のみが比較可能。
・整数と浮動小数で演算を行うと、浮動小数に型変換されて演算される。
・比較演算子のうち、==と!=はNULLオブジェクト同士、またはNULLオブジェクト以外との比較が可能。

■算術演算子
加法　：+
減法　：-
乗法　：*
除法　：/
剰余　：%
べき乗：**

注）+と-に関しては単項演算子の性質も持つ

■ビット演算子、シフト演算子
ビット論理積（AND）　：&
ビット論理和（OR）　：|

ビット排他的論理和（XOR）：^
ビット否定（NOT）　　：~
左ビットシフト　　：<<
右ビットシフト　　：>>

■代入演算子

=，+=，-=，*=，/=，%=，**=，&=，|=，^=，<<=，>>=

■オートインクリメント演算子、オートデクリメント演算子

++，--

■比較演算子

==，!=，<，>，<=，>=

■論理演算子

&&，||，!

■カンマ演算子

，

■リスト配列参照演算子

[，]

■演算子の優先順位

優先度高い
↑
左結合：（ ），[ ]
---
右結合：++，--，**，~，!，単項演算子（+，-）
---
左結合：*，/，%
---
左結合：+，-
---
左結合：<<，>>
---
左結合：<，>，<=，>=
---

左結合：==, !=
---
左結合：&
---
左結合：^
---
左結合：|
---
左結合：&&
---
左結合：||
---
右結合：=, +=, -=, *=, /=, %=, **=, &=, |=, ^=, <<=, >>=
---
左結合：カンマ,
↓
優先度低い

## A.3 制御構造

■分岐制御
if（条件式）文 ;
if（条件式）文１; else 文２;

■条件ループ制御
while（条件式）文 ;
for（条件式１; 条件式 2; 条件式 3）文;

■代入ループ制御
foreach 変数名（リスト）文;

指定変数に、リストで指定した要素値を順番に代入し、
要素数分だけ、指定文を繰り返す。

■その他のステートメント
break ;　　最も内側のループブロックを無条件に抜ける。
continue ;　　最も内側のループブロックの先頭に戻る。

redo ;　　最も内側のループブロックの先頭にに戻る。
　　ただし、continue と異なり、for 文などの条件式を再評価しない

■ループブロックラベル
ループブロックには、ラベルをつけることができる。
ラベルを用いることによって、処理途中で指定したループブロックを
強制的に抜けることができる。

記述例　label: ループブロック{}

ラベル: while（条件式）文 ;
ラベル: for（条件式１; 条件式 2; 条件式 3）文;
break ラベル;
continue ラベル;
redo ラベル;

■プログラム終了
exit( 式 );
返り値：なし

## A.4　ユーザ定義関数

■ユーザ定義関数の定義
func ユーザ定義関数名( 引数名 ) { 文; }

■引数
ユーザ定義関数の引数は、呼び出し時にリスト型のデータとしてエンコードされる。
引数が１つしかない場合も同様

■返り値の指定
return 返り値 ;

返り値の型は、データの型に依存する
return 文を記述しない場合は、自動的に null が返される。

## A.5 組み込み関数

■イベント発生関数
テストターゲットとしてのGUIは、ボタン押下のみで操作可能なものを想定している。よって、現状のTCSではボタンシングルクリックイベント発生関数のみ。イベントの発生は、ターゲットシステム内に組み込まれたTesting Managerによって処理される。各イベントごとに生じるであろう、スクリーンショットの変化、内部ステータスの変化も、Testing Managerによってロギングされる。

■ボタンシングルクリックイベント発生関数
push(変数(もしくは整数値));
返り値：発生したイベントの個数

push関数は、引数となるボタンコンポーネントのID(整数値)に対して、ボタン押下イベントを発生させる。

■リスト型データ操作関数

| | |
|---|---|
| reverse( リストデータ ) ; | 指定したリストの要素を逆転する |
| multiple( リストデータ, 整数 ) ; | リストの各要素を指定回数分だけ重複させる |
| null() ; | 指定したデータがNULL型なら真 |
| lnump() ; | 同様に、整数型 |
| dnump() ; | 同様に、倍精度浮動小数型 |
| strp() ; | 同様に、文字列型 |
| listp() ; | 同様に、リスト型 |

■その他組み込み関数

■標準出力関数
println
返り値：出力した要素の数

例:

```
println( 1 ) ;
println( 1, 2, 3, 4, 5 , nl ) ;
println( (1, 2, 3), (4, 5 , 6) ) ;
println( "Hello, TCS world" ) ;
```