7 テスト／デバッグ技法の効果と効率

松尾谷徹
(有)デバッグ工学研究所

ソフトウェア開発においてテストとデバッグの占める割合は大きく，実践の場ではさまざまな技法が経験的に選択され使われている．しかし，それら技法の効果と効率を客観的に評価する方法が不足しているため，不適切な使われ方が多々生じている．ここでは，テストとデバッグを分離し，その効果と効率を評価する方法を示し，テスト工程別に技法の効果と効率について概説する．

実践におけるテストの課題

ソフトウェア開発ライフサイクルにおいてテスト活動の占める割合が50%から80%に達している現実がある．この問題に対するアプローチは2つで，1つは，開発時に誤りが混入しないよう工夫をする方法（fault avoidance）であり，もう1つはテスト活動を合理的に行うことである．ここでは，後者のアプローチから，テストの効果と効率について考え，結論としてテストの効果は，開発とテストの検算活動の効果と捉えることを示す．

この10年間を振りかえると，テストに関する実用書があふれ，さまざまな技法やツールが出現し，テストの進歩を感じる[1)]．一方，どんな基準で，どんなテスト技法を選択すればよいのか迷う時代に入っている．個々の技法やツールについて，How-toは知られているが，効果について述べられることは稀である．テスト技法を利用する者の中には，より少ないコストで済む安易な方法や流行を追う傾向すら見られる．

テストの効果に対するモデルや測定は，ソフトウェア工学の中でも未完成で難しい分野であり，実用化が遅れている．テストについて論ずる上で，テストの効果を評価しない限り，産業界においてテストの健全な発展はあり得ない．本稿では，テストの効果，テストとデバッグの効率に関して簡単なモデル化を行い考える．次に，実際のテストにおける手法や技法をテストの工程に従い3つの段階に分けて評価する．3つの段階とは，コンポーネントテスト，統合テスト，システムテストである．

テストとデバッグのモデル化

テストとデバッグの効果と効率を考えるために，ここではモデル化を行い，モデルの上で効果と効率について定義する．まず，テストに対する概念的な捉え方として，工業製品に対する検査の概念を払拭し，テストを検算の一種として考えることから，簿記の歴史を例に示す．次に，ソフトウェアのテストの要素として，仕様，実装，テストケースの3者を考え，検算の有効範囲とフォールトの種類から効果と効率を定義する．

［テストとデバッグの定義］

基本となる用語の定義を行う．ソフトウェアが意図したとおりに機能しないことをソフトウェア故障（softwae-failure）と呼び，その原因をフォールト（fault）と定義する．フォールトは，バグ（bug）あるいは誤り（error）と呼ばれることもあるが，たとえばコーディングを誤る（動詞）など「フォールト」が作られる原因とは区別される．

ソフトウェア故障は，正常な状態から劣化や消耗の結果として故障へ遷移するハードウェア故障とは異なるメカニズムである．ソフトウェア故障は，特定の環境や入力条件において現象として顕在化する．ソフトウェア故障の有無や程度を評価することをテストと呼び，その原因であるフォールトを除去することをデバッグと呼び両者を区別する．

［簿記の歴史に学ぶ］

簿記とは，経済活動における金銭上のやりとり（取引）を伝票や帳簿に記録し，計算することであり，15世紀頃から始まったとされている．簿記を人手による一種の情報システムと捉え，例として考える．伝票から帳簿への転記や計算が正しく行われないと帳簿にはフォールトが混入する．単位活動に含まれるフォールトの数を誤り率pとすると，帳簿上で行ったN件の活動の累積値には$N*p$件のフォールトが含まれていることになる☆1．

簿記システムの故障に当たるのは，決算において帳簿

☆1 pは0.01から0.005程度である．

7 テスト／デバッグ技法の効果と効率

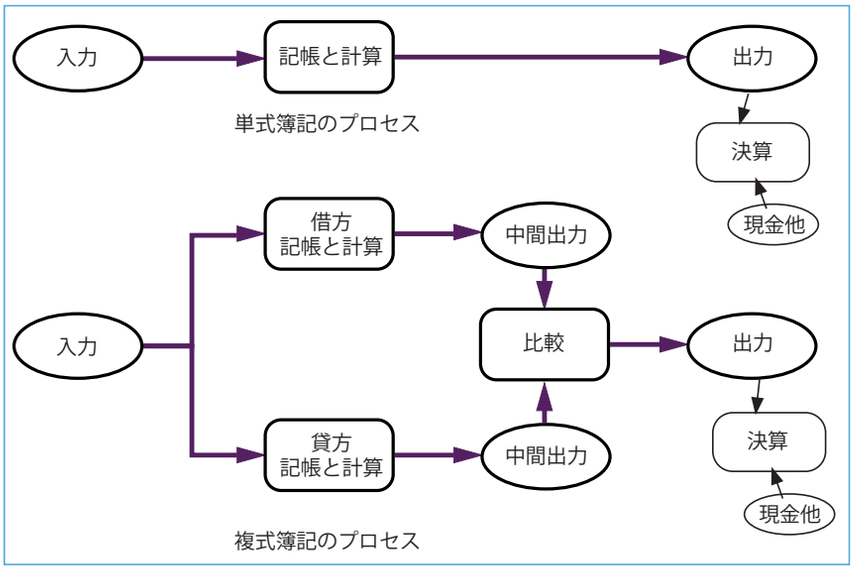

図 -1　単式簿記と複式簿記

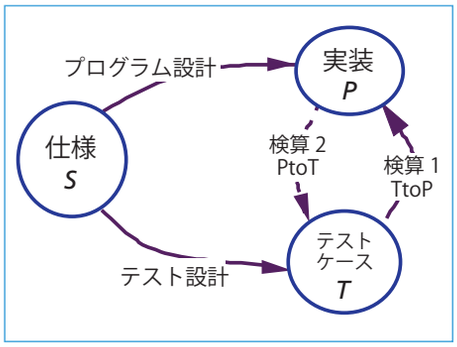

図 -2　ソフトウェアの検算

と品物や現金の実数とが一致しない現象と考えることができる．つまり簿記システムにおける故障は，決算というテストによって評価できる．故障が見つかってもフォールトを除去するデバッグは容易ではない．決算の期間に行われた N 件の活動すべてを見直す必要があり，再計算を行っても再計算にフォールトが混入するので $N*p=0$ にするのは不可能に近い．初期の簿記システムにおける決算期間は，現在からは想像できないほど長く，20 ～ 30 年であり，決算では残差を求め損金処理など経理処置が行われた．

企業の活動規模が大きくなると，帳簿の残差も大きな金額となり問題であった．この問題を解決するため，ヴェネチア式簿記と呼ばれる複式簿記が 15 世紀に発明されている．複式簿記とは，取引の記録や計算を借方と貸方（debtor and creditor）の両方で行い，部分的に検算を行う方式である．この方式は，信用取引など現金出納が発生しなくても，また，決算を待たなくても，内部で借方と貸方の累積値から検算することができる．

図 -1 に単式簿記と複式簿記の違いについて示す．複式簿記の検算は，日次単位，月次単位，勘定単位などで細分化し，検算する期間内の N の値を小さくすることができる．借方の誤り率を p_d，貸方の誤り率を p_c とすると，両者が同時に誤る率は $p_d * p_c$ なので，複式簿記の品質は単式簿記と比較し飛躍的に改善された．

複式簿記の導入コストは，記載や計算の手間が単式簿記と比べて 2 倍以上になるが，総合的な観点で簿記システムの品質を考えるなら，後戻りコストが低減し合理的である．複式簿記が合理的であることは明白であるが，実践の場で受け入れられるには，歴史的に見て非常に長い期間を要した☆2．

☆2 一橋大学附属図書館が詳しい情報を提供している．

［ソフトウェアテストのモデル化］

簿記システムにおける検算機構の観点から，ソフトウェアのテストについて考える．ソフトウェアのテストは，次の 3 つの要素について比較を行う．3 つとは，仕様 S，実装されたプログラム P，テストケース T であり，その要素の関係を**図 -2** に示す．実装されたプログラムにテストデータを与え，動作させた結果とテストの予測正解値を比較することが具体的な検算である．しかし複式簿記と比べると，次の 2 つの特性で大きな差がある．

(1) 並列性　複式簿記の検算は，金額の累積値を求め，借方と貸方で比較するのでテストケースの数は少なくて済む．一方，ソフトウェアの場合，論理と機能の数だけ並列にテストケースを作りテストを行う必要がある．

(2) 非対称性　複式簿記の検算は，借方と貸方が対称であり，借方から貸方を確認することも，その逆もできる．一方，ソフトウェアの場合は図 -2 の検算 2 を行うことが難しい☆3．

［テストの効果］

テストの効果は，テストによってソフトウェア故障をどれだけ検出できるかを表すことである．ここでは，テストの効果を検算機構として検出できない場合から考える．

仕様 S を何らかの要素からなる集合と考え S とする．同様に，実装されたプログラム P とテストケース T も集合 P，T とする．それぞれの関係を**図 -3** に示す☆4．検算を要素間の比較と定義すると，検算されない原因として次の 2 項目が考えられる．

☆3 パス網羅の計測などに限られる．
☆4 関係を写像と考えるのが妥当であるが，ここでは省略する．

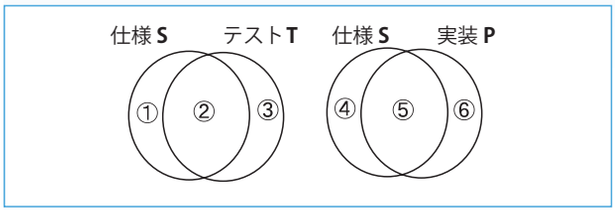

図 -3　仕様，実装，テストの集合と関係

(1) テスト設計の不完全性　図 -2 の仕様 S からテストケース T への変換をテスト設計と呼び，その不完全性によって検算されない要素が生ずる場合である．テストケースの集合と仕様の集合の関係は $T \neq S$ であり，図 -3 の①の部分 (S-T) が検算されない．③の部分は冗長でありテストケースのデバッグによりテストの改修が必要となるため，デバッグの効率を低下させるが効果には影響しない．

(2) 検算できないフォールト　テストケース設計が完全な場合，すなわち $T=S$ の場合であっても，図 -3 の⑥の部分 (P-S) が検算されない．⑥の部分は，仕様には定義されていない冗長な処理であり，実装時仕様と呼ばれフォールトとは限らないが，検算されないことが問題である．

［有則，無則，禁則］

テストを検算の視点から評価し，検算が漏れる 2 つの場合を示した．後者の発見できないフォールト，つまり冗長な処理について考える．考え方として，仕様が表していることを次の 3 つに分類する．

(1) 有則　仕様に定義されていることを意味する．通常のテストは有則に対して行われる．たとえば「入力の条件」と「処理の結果」を仕様から読み取りテストケースを作成する．

(2) 無則　仕様に定義されていることの補集合を意味する．通常のテストで考慮されることは稀である．たとえば，仕様が求めた機能が動作するのは「入力の条件」にのみ限られ，「入力の条件」以外の条件では絶対にその機能は動作しないことを意味する．動作しないことをテストするには，動作する「入力の条件」の補集合をテスト入力としてテストケースを作成する．

(3) 禁則　2 つの意味で使われる．1 つは制限事項として，「ある条件」において仕様が成立しないことを意味する．もう 1 つは，特定の条件以外で「ある機能」が絶対に動作しないことを要求する仕様を意味する．後者は安全関連分野においてインターロックと呼ばれている．ここでは，後者の意味で使う．仕様が要求ではなく，インターロックの機構や機能を詳細化すれば，禁則は有則の一部と考えることもできる．

［機能網羅と論理網羅］

テストを詳細化し，仕様に対するテストの漏れについて考える．テストは，テストケースと呼ぶ「テスト入力」と「結果の予測正解値」のペアを使って行われる．被テスト対象であるプログラムに，「テスト入力」を与えプログラムから「テスト結果」を得る．この「テスト結果」と「結果の予測正解値」を比較し，一致すれば「合格」，不一致であれば「不合格」とする．「不合格」の場合，デバッグにより「テストの故障」か「プログラムの故障」かを切り分け，フォールト部分を改修する．

テストケースを作成する場合，仕様の何に注目して漏れなく作成するのかによって，次の 2 つのアプローチが考えられる．

(1) 機能網羅　仕様で定義された機能に着目して，テストケースを作成するアプローチである．テストケースの設計は，仕様から機能を抽出し，その機能が正しく動作する入力を考えテストケースとする．テスト設計の方法としては初歩的な方法である．

(2) 論理網羅　ある機能に注目すると，その機能が動作する条件は 1 つ以上存在する．この条件のことを論理と呼ぶ．論理には 2 種類あり「組合せ論理」と状態遷移を含む「順序論理」である．論理を抽出する方法は 2 つあり，有則から求める方法と，入力の取り得るすべての組合せから求める方法である．後者の具体的な方法は，入力の組合せを，たとえばマトリックスやデシジョンテーブルで表現してテストケースとする．

テストケースの漏れには，人為的なミスとしての漏れもあるが，ここでは，テスト技法や検算の方式による漏れをテーマとしている．機能網羅のテスト技法は，論理網羅の漏れが生ずるのは当然であり，論理網羅をカバーする技法でも，状態遷移が存在すると順序論理を取り扱う技法を用いないと漏れが生ずる．つまり，技法の選択がテストの効果を決める上で重要である．

［テストの効率］

テストの効率は，テストの効果と共に考える必要があり，単独で定義することは意味がない．テストの効果をマクロに捉えると，検算の方式とテスト技法の選択で決まることから，テストの効率は，その制約の中で合理的にテストケースの数を選択することとほぼ等価になる．たとえば，機能網羅は論理網羅よりテストケース数は少なく，一見効率的であるが，効果が異なるためテストケース数で比較できない．

ミクロな効率は，オペレーションの課題である．具体

7 テスト／デバッグ技法の効果と効率

的に考えると，ランダムテストではテストケースの重複が生じ，テストケースを識別し消し込み管理を行うオペレーションに比べ，テストの効率は低下する．テスト中に改造を繰り返すと，再テストを何度も行うことから，定期的なリリース管理を行うオペレーションと比べ，テストケースの実行回数が増加しやはり効率は悪くなる，などである．

［デバッグの効果と効率］

デバッグの効果は，検出したソフトウェア故障からフォールトをどのくらい見つけ出すことができるかの指標であり，効率はそのときの平均コストである．一般的に，テストケースの粒度が小さいほどデバッグの効果も効率も高くなる．デバッグは，ソフトウェアをブラックボックスとして扱うと困難である．一般化すると，故障に関する情報量と対象範囲の比率が重要な指標になっている．

実際のデバッグは，まず最初に行うことは，テストケース側の故障か，プログラム実装上の故障か，仕様上の問題かについて切り分ける．組込み系は，テストケースをスクリプト言語を使ってプログラミングすることから，テストのデバッグにプログラムのデバッグと同程度の手間を要する傾向がある．

テストスクリプトのデバッグであれ，プログラムのデバッグであれ，情報を収集する仕組みが重要である．特に順序論理を含むリアルタイム系は，タイムスタンプ付きのログ情報など時系列で情報を収集できないとデバッグは困難であり，テスト環境やソフトウェアそのものに試験機としての機能が求められる．

工程別のテスト／デバッグ技法

テスト技法の効果と効率について，テストの工程別に考察を行う．

［コンポーネントテスト］

コンポーネントテストとは，ユニットテストとも呼ばれ，システムの要素を分離してテスト可能な単位に分けて行うテストである．コンポーネントテストの対象は，コンパイル可能な最少単位（数十ステップの関数）から，エンタープライズ系であれば独立した業務単位までさまざまな大きさに対して行われている．しかし，技法の効果から考えると，コンポーネントテストの規模には制約があり，規模が大きくなると極端に効果が低下する．

［ホワイトボックステスト］

作られたプログラムの実行可能なコードをすべて網羅するようにテストケースを設計する．具体的な基準として命令網羅や分岐網羅が用いられている．検算は2つの成果物の比較を行うことであり，このテスト技法単独では検算にならない．ホワイトボックステストが有効なのは，図-2の検算2として，ブラックボックステストで作成されたテストケースの漏れを，プログラム側から検算する場合である．

［機能網羅］

機能網羅は結果網羅とも呼ばれ，機能が動作した結果（出力）から機能の正当性を確認する．具体的な方法は，仕様から動詞「～を行う」「出力する」などを調べ出し，さらにその機能が正しく動作する入力条件を読み取り，テストケースを作成する方法である．関数レベルの簡単な仕様以外は，機能が動作する入力の条件を持っているので，この方法で得られるテストの効果には限界がある．

［論理網羅］

論理網羅は入力網羅とも呼ばれ，プログラムの制御に影響する入力の組合せを網羅するようにテストケースを設計する．コンポーネントテストは，一般的に状態変数をコンポーネント外に持つので組合せ論理のテストで十分である．

具体的な方法は，コンポーネントの入力を後述の同値分割を用いて分類し，組合せをマトリックスで表現する．マトリックスには，機能が動作する有則のテストケースだけでなく，動作しない無則や禁則についても記述することができる．有則のみ抽出すると機能網羅と同じになる．

論理網羅は，確実なコンポーネントテストの方法であるが，コンポーネントの規模が大きくなると，入力数が増え，テストケース数は入力数の積で増加するため膨大な数になりテストの効率が低下する．対策は，設計において機能の独立性を高め，コンポーネントに含まれる組合せ論理の数を少なくする適切なコンポーネント分割である．

［論理網羅の効率化技法］

いかなるテスト技法を用いる場合でも，テストケースを削減する共通した技法は，同値分割と境界値の技法である．

(1) 同値分割　仕様に記述された入力値や出力値に注目し，その要素 $e_1, e_2, \ldots, e_i, \ldots, e_n$ を集合 **E** とする．集合 **E** を要素単位ですべてテストするとその数は n 個となる．しかし，多くの場合，仕様は範囲を定義

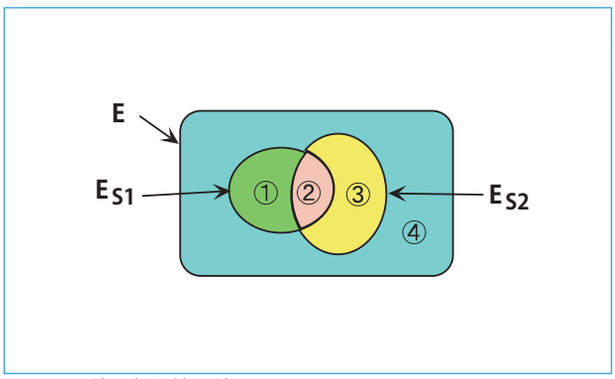

図 -4　同値分割と境界値

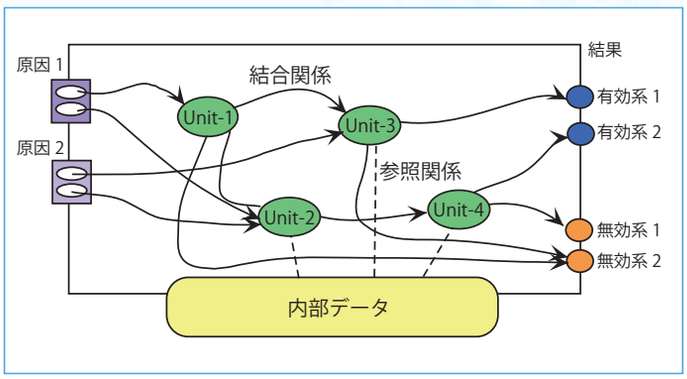

図 -5　統合テストの領域

しており，たとえば**図 -4** に示すように，部分集合 $E_{S1} = \{X_1 \leq e_i \leq Y_1\}$ と $E_{S2} = \{X_2 \leq e_i \leq Y_2\}$ が存在したとする．もし，$E_{S1} \cap E_{S2} \neq \varnothing$ ならば，① $E_{S1} - E_{S2}$，② $E_{S1} \cap E_{S2}$，③ $E_{S2} - E_{S1}$，④ $E - (E_{S1} \cup E_{S2})$ の 4 つの部分集合に分け，4 個のテストケースを設定する．②の部分が $E_{S1} \cap E_{S2} = \varnothing$ なら 3 個でよい．

同値分割とは，図 -4 の 2 つの部分集合 E_{S1} と E_{S2} が同値類ではない場合，論理積の部位を別の部分集合と見なして分割して，テストケースを作成することを意味する．($E_{S1} \cup E_{S2}$) の補集合④は，直接機能を動作させる入力条件ではないが，無則に対応するテストケースである．

(2) 境界値　同値分割を用いて部分集合単位でテストを行う場合，具体的なテストデータとして，部分集合間の境界値をテストの入力値とする．

［デバッグの効率］

コンポーネントテストは，テスト側もプログラム側も共にデバッグは容易である．テスト範囲の大きさに比べて解析のための情報量が多いからである．具体的には，開発者はデバッガなどを用いて故障を再現させ，プログラムの内部情報からフォールトを特定することができる．

［統合テスト］

統合テストとは，テスト済みのコンポーネント（ユニット）を含むある範囲に対して行う．テスト対象には，たとえば**図 -5** に示すような，ユニットの結合順序や内部データの参照関係など，意図された構造を持った領域に対して行う．領域には，内部データや OS やハードウェアとのインタフェースが含まれることがある．

ここでは統合テストの前提として内部構造について詳細な情報と，テストのために内部データへのアクセスが可能であることを想定している．その理由は，テストの結果としてデバッグが必須であること，および，内部データに状態遷移を含む場合，状態変数に対してアクセス可能なテスト環境を構築しないと，順序論理網羅ができないことによる．GUI の操作性など非機能テストの場合は，内部構造に関する詳細な情報を必要としない．

［統合テストの機能網羅］

統合テストにおける機能網羅の方法は，機能を抽出するだけでなら統合状態におけるコンポーネントテストの再現にしかならない．統合テストは機能の組合せについてテストを行う必要があり，組合せを漏れなく確実に行うには，何らかの技法を用いてマトリックスを作成する．

現実問題として，機能の組合せをどのように抽出するかによって，テストの効率に大きな差が生ずる．経験的な方法として，統合範囲に含まれる機能あるいは処理を「処理のバリエーション」として横軸に展開し，縦軸には業務によって変化する「データのバリエーション」を抽出し，マトリックスを作成する技法がある．この方法の利点は，「実務で生じる組合せ」を想定し，その網羅性を確保するので，仕様上のすべての組合せより数が少なく合理的である．たとえば，既存の業務のシステム化を行うエンタープライズ系や，改造をベースとする組込み系において有効である．

もう 1 つの方法は，直交表や all-pair 法を使ってコマンドやパラメータの組合せを作成するやり方である [2]．コマンドのバリエーションが多くなると直交表では難しくなるので，分割するか all-pair 法が使われる．この方法の利点は，業務ノウハウがなくても組合せの網羅ができることである．しかし，機械的に抽出した組合せには禁則の組合せが生ずるので，組合せに禁則が入らない処置を行う必要がある．

［統合テストの論理網羅］

論理網羅には，組合せ論理網羅と順序論理網羅がある．まず組合せ論理から考える．統合段階における入力の組合せは膨大になるため，マトリックスなどですべての組合せをテストすることは不可能である．よって機能が動作する論理，すなわち有則の論理を抽出してテストを行

うことになる．仕様から組合せ論理を抽出し，テストケースを作るには，デシジョンテーブルが一般的である．

形式仕様のように論理的に記述された仕様であれば，仕様からデシジョンテーブルを容易に作成できるが，実践の場で形式仕様が使われることは例外的である．効果的なデシジョンテーブルを作成することが重要であり，技法としては原因結果グラフが知られている[3]．原因結果グラフは習得が困難であり，実践の場で用いられることは例外的である．

そこで使われるのが，原因流れ図（CFD：Cause Flow Diagram）である．CFD は，たとえば図 -5 のユニット間の結合構造に沿って，ユニットの出力を次のユニットの入力に結合し，取り得る範囲を同値分割し，結合の順にデシジョンテーブルを作成する．この技法のメリットは，結合していないコンポーネント間の組合せを排除できることと，結合の順序を反映しているので，テストケース数が少なくなり効率が高くなることである．

［統合テストの順序網羅］

状態を持つ場合，それらの組合せを含めて論理を外部入力から網羅するのは不可能である．唯一の方法は，テスト環境においてソフトウェア内部の状態変数を書き換えたり読み出したりする，テストのための機能を付加し，順序論理を組合せ論理に縮退させる方法である．CFD 技法は，縮退させることを前提として，順序論理をデシジョンテーブルに展開する方法を含んでいる[4]．順序論理を組合せ論理に縮退させる方法は，デバッグの効率を高める上でも有効である．

［システムテスト］

システムテストは，実運用におけるシステムの挙動が与えるリスクを明らかにするために行われる．そのため，仕様書通りに作られていることをテストするのでなく，環境要因が与える影響を分析するためのテストが中心となる．検算システムとして考えると，検算すべき相手が仕様ではなく想定される環境要因なので，まず，想定される環境要因のさまざまな変化を再現する方法を構築することである．

具体的には，システムの動作環境をシミュレーションすることであり，シミュレーションの基礎となる環境要因のモデル化と数量化が必要になる．環境要因には，呼量の分布や，操作上の誤操作や，ハードウェアの故障や異常など，さまざまなシステムのドメインに特化した要因が含まれる．

システムテストは，テストの結果として単純な合格／不合格といった判定ではなく，リスクの兆候を読み取る必要がある．そのためには，システムの動的な挙動を把握できる計測機構が必要となる．故障からフォールトを同定するだけでなく，さまざまな検出器を埋め込んだデバッグ環境の上でテストを行い，わずかな兆候でも捉えることが求められる．

検証指向設計に向けて

テストを一種の検算と捉え，その効果や効率についてテスト技法の側面から述べた．検算の効果は，比較すべき両者が漏れなく比較できることであり，検算の効率は，所望の効果を得るために実行したテストケースの数によって決まる．具体的に問題になるのは，論理の網羅であり，その中でも無則の部分に対するフォールトを対象にするか否かが大きく影響している．近年，システムの複雑化とシステムが担うリスクの大きさから，メモリリークなど無則の部分を含めたテストがより強く求められている．

テストの非対称性などの課題を含め，テストやデバッグ側から自らの効果や効率を高め，いかなるソフトウェアのいかなるフォールトをも見つけ出すアプローチには限界がある．ソフトウェアの品質をさらに高めるには，検算の効果や効率を高めることを目的とし，新たな開発方法論が必要であると考える．1 つの方法は，検証の容易性を検算の容易性と捉え，検証指向設計と呼ばれるアプローチである．

いずれにしても，テストの効果や効率を定量的に計測し，比較することができない限り，実証的な研究開発は困難である．客観的な方法で，テストの効果や効率を計測し，実証主義に徹したテストのプロセス改善が普及することを期待する．

参考文献

1）松尾谷徹：ソフトウェアテストの進歩と課題，ソフトウェア工学研究会 100 回記予稿集，情報処理学会 (1994).
2）吉澤正孝，秋山浩一：ソフトウェアテスト HAYST 法入門，日科技連出版社 (2007).
3）Myers, G. J. : The Art of Software Testing, John Wiley & Sons (1979)．翻訳：長尾真一，松尾正信：ソフトウェアのテスト技法，近代科学社 (1980).
4）松尾谷徹：テストケース抽出の一方式：順序回路を含む場合，情報処理学会第 39 回全国大会，情報処理学会 (1989).

（平成 20 年 1 月 15 日受付）

松尾谷徹（正会員）
matsuodani@debugeng.com
1948 年生．（有）デバッグ工学研究所代表．PS 研究会代表．法政大学工学兼任講師．博士（システムズ・マネジメント）．テストの実践活動と人材育成／チーム育成に興味を持ちフィールド活動に従事している．