

解説

ソフトウェアの信頼性*

大 島 裕**

1. ま え が き

ソフトウェア***は、工業製品というよりむしろ芸術的産物とみられがちであったため、信頼性技術の対象として認識されるのが遅くなった。しかしながら、社会活動の多様化高度化に対応する方法として、ソフトウェアの役割りが大きくなったこと、システムの開発費にソフトウェアの費用が占める割合が多くなったこと、ソフトウェアがシステム開発のネックとなり不十分な品質のまま使用に供される危険が増えてきたことなどの理由でソフトウェアの信頼性に対する関心が高まってきている。

一般に、システムや製品が信頼できるということは、“ある使用状態において、期待される期間 期待される機能・性能を発揮する”ということである。信頼できるシステムや製品は、①故障が起り難く、②(もし故障を直して使うのであれば)故障が速かに直せるものでなければならない。故障の起り難さは信頼性ということばで、故障の直し易さは保全性ということばで表わされるが、両者を合せて信頼性と表現することもある。

信頼性(ならびに保全性)は、図-1に示すようにシステムや製品の価値を構成する1要素である。信頼性技術の目的は、所望の信頼性を有するシステムや製

品を作成するのに貢献することだけでなく、信頼性と他の要素との間のトレードオフの評価に寄与することにもある。

2. ソフトウェアの信頼性

2.1 “ソフトウェアの信頼性”の範囲

ソフトウェアシステム*がサービスシステム**の構成要素となっている場合、ソフトウェアシステムの信頼性に関する要素を次のように考えることができる。即ち、ソフトウェアシステムは、一般には、バグ***をもつモジュールから構成されており、バグの影響を少なくするための、ソフトウェアとしての対策を内蔵している。そして、その対策のために、ハードウェアシステムとオペレータのサポートを利用する。一方、ソフトウェアシステムは、ハードウェアシステムとオペレータの信頼性を高める機能(障害処理、誤操作防止)をサポートしている****。

ソフトウェアの信頼性を論ずる場合、上記の、ハードウェアシステムやオペレータの信頼性を高めるためのソフトウェアのサポートの内容にまでわたることもあるが、本稿では、そこには立入らない。

2.2 ソフトウェア障害の様相

ソフトウェア自体には経時変化がないから、障害は、専ら環境と入力の変化が誘因となって発生する。環境の変化としては、ソフトウェアシステムを新しいサービスシステムに適合するようにシステムジェネレーションを行った場合、新しいモジュールを追加した場合、あるいは、新しいコンパイラでコンパイルし直

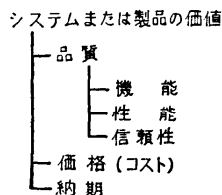


図-1 システムまたは製品の価値の構成要素

* Software Reliability by Yutaka OHSHIMA (N. T. T. Yokosuka Electrical Communication Laboratory).

** 日本電信電話公社 横須賀電気通信研究所

*** ソフトウェアは一般に、プログラムと、プログラムの内容・使い方・保守の仕方などに関するドキュメントから成っているが、プログラムのみをさしていることもある。

* ソフトウェアシステムとしてはオンラインシステムのオペレーティングシステム+業務プログラムをイメージに描いて話をすすめる。

** ソフトウェアシステム、ハードウェアシステム、オペレータを含めた全体をシステムとしてとらえている。

*** ソフトウェアの欠陥で障害(現象)の原因になるものをいう。

**** この機能を遂行するソフトウェアのモジュールもバグを持つと考えるのが一般的であるから、それらがソフトウェアシステムに加わることによってソフトウェアシステムの信頼性は若干なりとも低下するが、サービスシステムの信頼性は向上する。……向上しない場合はそのようなサポートは見合せるべきである。

した場合などがある。また、入力の変化はソフトウェアが使用されるごとにある訳であるが、新しい利用者のジョブとか従来にない大規模なジョブが投入されると障害が発生しやすいことはよく知られている。

ソフトウェアの障害の様相は、データ・ベースを破壊するもの、システムの処理機能が全部停止するもの（システムダウン）、ユーザに誤まりと分らないような結果を出力するもの、ユーザに容易に分るような誤りを出力するもの、などいろいろである。通常は、はじめの二つ（データ・ベースの破壊とシステムダウン）が重視される。3番目のものは、はじめの二つより、一見、軽度であるように思われるが、例えば、超高層ビルの耐震設計を行うプログラムの出力が誤っていたとなると、その影響はシステムダウンの比ではないかもしれない。そこで、ソフトウェア障害を重要度により重みづけして取り扱う必要があるのであるが、これは、サービスの内容に高度に依存する問題である。

2.3 ソフトウェア障害時の処理

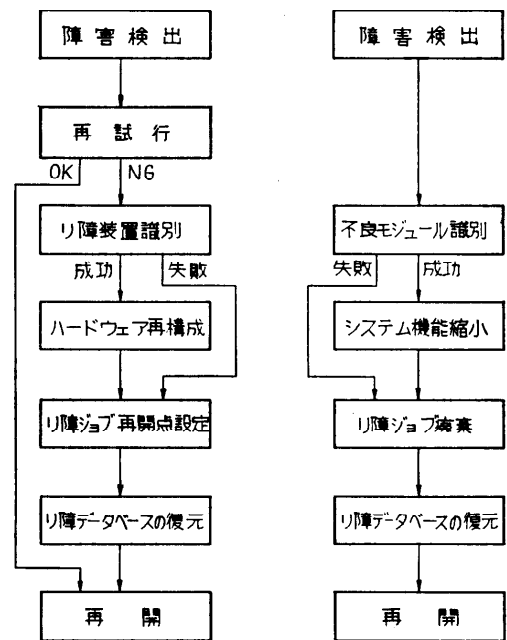
ハードウェア障害時のソフトウェアシステムの処理とソフトウェア障害時のソフトウェアシステムの処理を図-2に対比して示す。

ハードウェア障害の場合は、まず再試行を行い、不成功であれば予備装置を用いてシステムを再構成するのが一般的である。そのあと、直接り障したジョブならびにそのり障したジョブの影響をうけるジョブの再開点にさかのぼり、り障したデータ・ベースをその再開点に見合う状態に戻してから、処理を再開する。

ソフトウェア障害の場合、再試行が意味がないこと、不良モジュールの識別が困難なこと*、および、予備モジュールが通常は存在しないこと**がハードウェア障害の場合と異なっている。

まず、ソフトウェア障害の原因となった不良モジュールが判明した場合、そのモジュールがもつ機能がシステムにとって必須でなければ、その機能に関連するモジュールを閉塞して、システムの機能を縮小（フォールバック）した状態で再開することができる。もちろん、り障したジョブは廃棄し、り障したデータ・ベースは復元せねばならない***。

次に、障害の原因となった不良モジュールが判明し



(a) ハードウェア障害の場合 (b) ソフトウェア障害の場合

図-2 障害時のソフトウェアシステムの処理

ない場合、あるいは、判明してもシステムとして必要度の高いモジュールである場合は、閉塞せずに処理を再開せざるを得ない。この場合も、り障したジョブは廃棄し、り障したデータ・ベースは復元せねばならない。この場合、再び同様な入力があれば障害が発生するのはやむをえない。

3. ソフトウェアの信頼性技術の概観

表-1 はソフトウェアの信頼性に関連する課題を列挙したものである。ここで、これらの事項の意義について概観し、次節以降に個々の内容について述べる。

3.1 信頼性の定量的な扱い

信頼性ならびにその測度（信頼度、保全度など）については、JIS における定義が基本的にはソフトウェアにも適用できるが、障害が入力に依存するというソフトウェアの特質に基づいた測度の定義が必要と思われる。システムを構成するモジュールの信頼度からシ

表-1 ソフトウェア信頼性に関連する課題

- (1) 信頼性の定量的な扱い
- (2) ソフトウェアシステムの信頼性対策（バグの害を抑える）
- (3) ソフトウェアモジュールの高信頼化（バグを作らない）
- (4) ソフトウェアのテスト（バグを取除く）
- (5) 信頼性の評価

* 障害を起した（例えばアドレス不正を起した）モジュールはハードウェアの割り込み等で判っても、真の原因は別のモジュールにあるかもしれない。

** 一般的ではないが 5.2 節で述べるような予備モジュールを考えたこともできる。

*** り障したデータ・ベースに関係のないジョブは、データ・ベース復元の前に再開することができる。

システム全体の信頼度を計算したり、逆に全体の信頼度をモジュールに配分したりする方法については、ハードウェアにおける考え方がソフトウェアにも基本的に適用できると思われるが、ソフトウェアシステムではモジュール数が非常に多く、また、モジュールが動作するかどうかシステムへの入力に依存するため、実用的な取り扱いには困難と思われる。

3.2 ソフトウェアシステムの信頼性対策

大規模なソフトウェアシステムは多くのモジュールから構成されるので、モジュールのバグを直ちにシステムの障害にするような方式では、システムの信頼性が不足するケースが多い。これを補うためのシステムの対策としてモジュールにバグがあった場合にリ障するジョブの範囲を局限する方法、システムの立ち直りを早くする方法、システムの機能をフォールバックして障害の誘因となる入力によるじょう乱を防止する方法等の工夫が望まれる。

システムの信頼度を高めるには、このような方式上の工夫とともに、使い込まれて信頼度が高くなったモジュールを使用するのが有効である。そのためには、モジュール間のインターフェイスを標準化し、システム間で流用できるようにする必要がある。

3.3 ソフトウェアモジュールの高信頼化

モジュールの設計製造において、バグを作らないのがソフトウェアシステム高信頼化の基本であり、そのためには、設計文書、プログラミング方式、プログラミング言語等の面での配慮が必要である。

3.4 ソフトウェアのテスト

ソフトウェアの信頼性は設計によって作り込まれるのが理想であるが、実際には、デバギングの過程が必要であり、その方法、そのためのツールの工夫等が必要である。

3.5 信頼性の評価

ソフトウェアのモジュールの信頼度、保全度、さらにソフトウェアシステム全体の信頼度、保全度を測定および予測することができてはじめて、信頼性目標達成のための有効な活動が実施できる。

4. 信頼性の定義^{1), 2)}

4.1 信頼性

JIS Z 8115 によれば、信頼性とは、“系、機器、部品などの機能の時間的安定性を表わす度合、または性

質”であり、信頼度は“系、機器、部品などが規定の条件の下で、意図する期間中、規定の機能*を遂行する確率”と定義される。これによれば、信頼性は時間の関数であり、次のような測度が用いられている。

信頼度 $R(t)$

故障密度 $f(t) = -\frac{dR(t)}{dt}$

故障率 $\lambda(t) = \frac{f(t)}{R(t)}$

平均故障間隔 MTBF = $\int_0^{\infty} t f(t) dt = \int_0^{\infty} R(t) dt$

もし、 $\lambda(t) = \lambda = \text{一定ならば}$,

$$MTBF = \frac{1}{\lambda}$$

上記の定義はソフトウェアにもあてはまるが、ソフトウェアの場合は、経時変化がないので、“規定の機能を遂行する”かどうかは専ら入力の値、入力の時系列で決まるのが特徴である。つまり単なる時間の経過は問題でなく、あくまで、入力の問題である。そこでソフトウェアの信頼度を次のように定義することができよう³⁾。

(A) ソフトウェアの入力として考える全ゆる入力から成る空間（入力空間 S_I ）を考え、その空間内の各点（ソフトウェアへの入力）に対してソフトウェアの応答は一意に決まるとする。このとき“ソフトウェアの（その入力空間上での）信頼度 R_I ”は次式で与えられる。

$$R_I = \sum_i^N P_i e_i$$

ここで、

N = 入力空間の点の数

P_i = i 番目の点が発生する確率

$e_i = 1$ (i 番目の点に対するソフトウェアの応答が正しい場合)

$= 0$ (そうでない場合)

この定義は入力空間があまりに巨大なため実際的でない。実際に測定の対象とし得るのは、テストにおける入力から成る S_I の部分空間、および、ユーザが使用する場合の入力から成る S_I の部分空間である。これらの部分空間の上での信頼度を上と同様に定義することができる。しかし、実測を指向するのであれば次の定義が、より実際的である。

(B) ソフトウェアに対するテストケースの集合を考える。テストケースとしては、開発段階でのテストまたは利用者に提供したあとの利用者の

* ここでいう機能には1章で述べた性能（機能の実現の量的な側面）も含むものと考えられる。

使い方を, システムの利用法にマッチした適切な基準で数えあげるものとする. このときソフトウェアの信頼度 R_T は次式で与えられる.

$$R_T = \frac{1}{N} \sum_{i=1}^N E_i(n_i) \cdot W_i$$

ここで,

N = テストケースの数

n_i = i 番目のテストで見つかったエラーの数

E_i = i 番目のテストケースでのエラーパフォーマンス (エラーがなければ 1, エラーの数が増えると 0 に近づく)

W_i = i 番目のテストケースの重要度を表す

係数 (ただし $\sum_i W_i = N$)

4.2 保 全 性

JIS によれば, 保全とは“修理可能な系, 機器, 部品などの信頼性を維持するために行う処理”であり, 保全性とは“修理可能な系, 機器, 部品などに備わる保全の容易さを表す度合または性質”である. 保全度は“修理可能な系, 機器, 部品などが, 規定の条件において保全が実施されるとき, 規定の時間内に保全を終了する確率”と定義される. 保全性の測度には次のものが用いられる.

保全度 $M(\tau)$

保全密度 $m(\tau) = \frac{dM(\tau)}{d\tau}$

平均修理時間 $MTTR = \int_0^{\infty} \tau m(\tau) d\tau$
 $= \int_0^{\infty} (1 - M(\tau)) d\tau$

ソフトウェアの場合も上記の定義があてはまる. 保全の時間としては, システムが処理能力を回復するまでの時間, データ・ベースが復元されるまでの時間, および, バグが修正されるまでの時間に注目する必要がある.

4.3 アベイラビリティ

アベイラビリティは“修理可能な系, 機器, 部品などが, ある特定の瞬間に機能を維持している確率”であり,

$$A(t, \tau) = R(t) + (1 - R(t))M(\tau)$$

で表わされる. アベイラビリティは次の式で求めることが多い.

$$A = \frac{MTBF}{MTBF + MTTR}$$

5. ソフトウェアシステムの信頼性対策

ハードウェアシステムの信頼性を高めるためのシステムの対策として, 次の 3 つの方法がとられる.

- 分割処理
- 冗長構成
- フォールバック

ソフトウェアシステムの信頼性対策もこの 3 つに分けて考えることができる.

5.1 分割処理

障害の影響をシステム全体に及ぼさないためには, 処理を分割して行うのがよい. 多重処理のためのソフトウェアシステムの制御構造は自ずとこの形になっている. さらに, セグメント方式とリング機構などのハードウェアによるメモリ保護機能を活用することにより, ソフトウェアバグの影響範囲をかなり局限できる. すなわち, メモリの汚染範囲を, 障害の誘因となった入力を処理しているタスク (多重処理における個々の処理の実行主体) が書き込みうる領域のみに局限できる. またデータ・ベースについても, タスクが書き込みうる範囲をタスクごとにある程度限定すれば, 汚染範囲を限定し, 復元の作業量を軽減することができる⁴⁾.

5.2 冗 長 構 成

2.3 節で述べたように, ソフトウェアにおいては予備モジュールは通常存在しないが, 次のような方法も考えられる.

(1) 性能向上の改造を行ったモジュールに対して, 改造前のモジュール (性能は劣るが機能は同等で, より信頼できる) を予備モジュールとして用い, 障害時にはとりかえる. この方法は比較的実行し易い.

(2) 機能を改良したモジュールに対して, 改良前のモジュールを予備にする. この方法の実行は比較的困難と思われる.

(3) 機能を改良したソフトウェアシステムに対して, 改良前のソフトウェアシステムを予備システムとする. 電子交換機 DEX においては, バージョンアップ後一定期間はこの方法を実施している⁵⁾.

5.3 フォールバック

オンラインシステムでは, ソフトウェア障害の誘因になったデータを投入した利用者が同じデータを繰り返し投入する可能性は高いとみななければならない. 障害が頻発すると, データ・ベース復元等の処理が負担となってシステムの処理能力が大巾に低下する場合が

ある。このような場合は、ソフトウェア障害のため処理できないデータをシステムの入口で拒絶または廃棄できることが望ましい(機能のフォールバック)。この場合処理能力の低下と機能の低下の何れを選ぶかについて、オペレータの判断が介入できるようにするのが適当である。

6. ソフトウェアモジュールの高信頼化

信頼性、保全性の高いソフトウェアを高効率で開発するためには、文書化、モジュール(分割の標準)化、高級言語の使用、プログラミングの標準化などの方策が有効であるといわれている(表-2)。

6.1 文書化

設計内容の文書化は、設計誤りを防ぐ点で生産性への貢献が大きい。文書化において留意すべき点は、文書(設計文書、マニュアル)の読者*をはっきりさせること、適切な時期に作成すること、正規のものにする時期と手続き、維持の方法についてルールを明確にしておくことである。文書化の作業自体が膨大になるので、生産性を阻害する要因になりかねない。機械化、自動化が要望される。

6.2 モジュール(分割の標準)化

モジュール化は、プログラムの外部条件および内部処理の理解し易さを助け、テストを容易にする。その結果バグが減少し⁶⁾、バグが修正し易くなる。モジュール化に当っては、モジュールを構成する原則をたて、外部条件が若干変ってもモジュールそのものが分解したり、他に吸収されたりすることのないよう配慮せねばならない。またモジュール間の階層関係を明確にする必要がある。

6.3 高級言語

高級言語の採用により表-2 に挙げたメリットの他、

表-2 ソフトウェアモジュール高信頼化の方策

利 点		文 書 化	モ ジ ュ ー ル の 標 準 化 (分 割)	高 級 言 語	プ ロ グ ラ ム の 標 準 化 (モ ジ ュ ー ル 化)
信頼性	設計ミス減少	○	○		
	コーディングミス減少			○	○
保全性	リーダビリティ向上	○	○	○	○
	修正ミス減少	○	○	○	○
生産性	仕様の理解容易	○	○		
	プログラマ間のコミュニケーション	○	○		
	テストが容易	○	○		○

* プログラム設計者、テスト担当者、プログラム保守者、サービスシステム設計者、センタオペレータ、一般利用者などの別がある。

機械独立であるため多くの利用者に使用され信頼性が確認されるという利点も出てくる。

最近では性能への影響が大きいモジュールを除いて、オペレーティングシステムも大部分は高級言語で記述されるようになっている。コーディングの誤りは言語の種類によらず、文当りほぼ一定の割合で発生するといわれ、高級言語で記述すれば、その拡大率の分だけ生産性が向上する⁷⁾。さらに、高級言語による場合、6.4 節で述べる構造化と類似の制約(アセンブリ言語による場合に比べて)が自動的に導入され、誤りの発生率が減少する効果がある。システム記述用の高級言語としては、PL/I のサブセットが採用されることが多い。

6.4 プログラミングの標準化(構造化)

プログラミングにおける誤りを防ぐ方法として、標準化したマクロ命令を広範囲に使用する方法と、汎用の手続き言語を、プログラミングの構造に制約を設けて使用する方法がある。前者の方法は、処理内容が定形化できる場合には有効である(一種の問題向言語)。後者の方法として制御の流れの構造を標準化したストラクチャード・プログラミング(SP)が提唱されている⁸⁾。SP の効果としては、管理技術の工夫と併用した場合にバグが、1キロステップ当たり0.3件であったという驚異的な数字の報告がある⁹⁾。SP 単独の効果、副作用(性能低下)、適用領域等については、今後の評価にまつところである。

7. ソフトウェアのテスト

7.1 モジュールのテスト

プログラムが正しく作られているかどうかを、いわゆる“プログラムの正当性の証明”の問題として形式的な手法で扱うことは実用上は不可能であることが知られている¹⁰⁾。

実用上は、テストケースの設計、テストデータの発生、診断情報の収集などを効率よく行う工夫が必要で、そのためのツールが作成されている。

テストケース設定法としては、モジュール内の枝を余すところなく通すための、パスの最小集合を見つけるというような構造面に着目したものがまず考えられる。他にコンパイラのシンタックスをBNFで定義すれば、シンタックスチェック機能をテストするデータを自動的に発生するというような、機能面に着目したものもある¹¹⁾。また、プログラムを実行させないで検

出できる誤り（設定されていないデータを参照するなど）を指摘するツールも工夫されている^{12)~14)}。

7.2 ソフトウェアシステムの統合テスト

モジュールをソフトウェアシステムに統合する際のテスト法には、大きく分けて次の3種がある。

(1) ボトムアップテスト法

モジュールの階層構造の一番底にあるモジュール（システムの中核側にあるモジュール）をまずテストし、機能確認がすめば、段々上位（外側＝ユーザ側）に移って行く方式である。着目しているモジュールより上位のモジュールは使用しないのが特徴である。この方法はオペレーティングシステムの制御プログラムの統合時に適した方法で、時間はかかるが着実な方法である。この方法では、モジュールテストを統合段階のテストで兼ねることもできる。

(2) 一斉テスト法

全てのモジュールを統合し、システムの外側からデータを投入してテストする方法である。モジュール個々の品質がよければ効率のよい方法である。システムの一部のモジュールを改造して入れかえた場合のテストはこの方法の一種と考えられる。

(3) トップダウン法

トップダウンプログラミングと組合せて行われる方法である。システムの最上位のモジュールにデータを入力する点では(2)と同じであるが、着目している階層より下のモジュールはまだ作成されていず、適当にダミーが入っている点異なる。この方法の利点は、モジュールの単体テストが省略できることであり、その分だけ正しい本物の環境によるテストがていねいに行えることである。

7.3 ソフトウェアシステムの総合テスト

ソフトウェアシステムの総合テストは、サービスシステムのテストとして、オペレータの特性、ハードウェアシステムの実際の信頼性を含んだかたちでシステム内のハードウェア、ソフトウェア各種資源をフルに使うようにして行う必要がある。

テストの効果を示す例として、TSS用OSのシステム統合テスト（ボトムアップ方式）からサービス開始後9ヶ月までの期間におけるバグの残存状況を図-3に示す¹⁵⁾（図-3ではそれ以後に潜在しているバグは無視してある。）。これによると統合以後のテスト（主に機能面に着目）では、プログラミング（狭義）やコーディングの段階で混入したバグは機能設計時に混入したバグより抽出されにくいことが判る。この例は、

SP、高級言語、モジュールテスト用ツールの有用性を示したものといえる。

8. 信頼性の評価^{1), 2)}

8.1 バグ数の予測

ソフトウェアのバグは設計、製造、統合、検査などの過程で次第に抽出されてゆく。この状況を示すのが生長曲線であり、この形が予想できれば信頼性保証上きわめて有用である。バグ抽出生長曲線のモデルの主なものをその使用条件によって分類列举したのが表-3（次頁参照）である^{1), 2), 16)~18)}。なおこれらの生長曲線の横軸は時間であるが、ソフトウェアについては4.1節で述べたように入力が問題であるから、横軸を入力数のテストケースの数に比例するようにした方が曲線をあてはめやすい（横軸が時間の場合、バグの累積曲線はなまった階段状になる）。次に、一定のプログラミング方式、品質管理方式をとる場合、バグ発生数はほぼ同じ割合となるので、バグの総数の推定が可能である。推定のベースはソースプログラムの文の数（命令の種類を問わない）をとるのが最も簡単であるが、分岐命令の文の数をとると精度がよくなるといわれている¹⁹⁾（表-4 p 894 参照）。

推定したバグ総数と、上述の生長曲線の到達点が合致しないときは、テストケースの選定が不適当でないか、あるいは、他に特殊な事情はないか調べた方がよい。なお、バグ修正の際のミスは、通常、修正件数の数パーセント程度あるといわれているが、この割合を充分把握し、予測、作業方法の改善に反映させることも必要である。

計画されたテスト工程においてどのくらいのバグが抽出されるか推定することは、工程の完了時期を予想

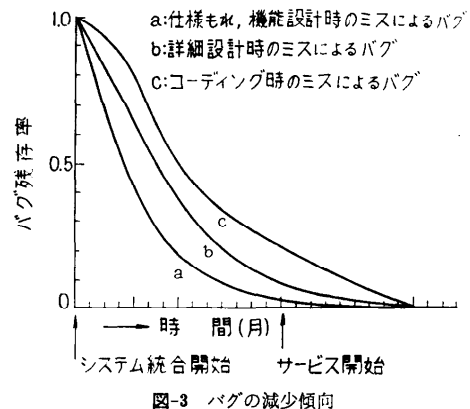


図-3 バグの減少傾向

表-3 バグ検出の生長曲線

使用パラメータ	モデル	モデル式
n : その時点までに発見した累計バグ数	ロジスティック曲線	$n = \frac{n_{\infty}}{1 + Ce^{-at}}$
n_{∞} : 総バグ数	ゴンベルツ曲線	$dn/dt = ane^{-bt}$
T : 総試験時間	Duan のべき形モデル	$n = \frac{n_{\infty}}{T} = KT^{-a}$
x : 投入努力量	指数速度モデル	$dn/dx = \lambda_0 e^{-rx}$
β : バグの存在比率 i : ステップ (工程) p : 各ステップごとにバグとして検出される確率 α : 修正される確率 n : バグ総数 n_i : i ステップでのバグ数	Lloyd-Lipow モデル ⁸⁾	$n_i = n\beta p(1 - p\alpha)^{i-1}$
M : 総モジュール数 m : 弱モジュール数 λ_0 : 良モジュールのバグ発生率 λ : 弱モジュールのバグ発生率 μ_0 : モジュールのバグ修正がうまく行く確率 $1 - \mu_0$: デグレードする確立 $N_i(t)$: t でのバグ数の期待値	デバギングモデル ⁹⁾	$N_i(t) = n_a + (i - n_a)e^{-[\lambda_0(1 - \mu_0) + \mu_0\lambda]t}$ ここで $n_a = \frac{M\lambda_0(1 - \mu_0)}{\lambda_0(1 - \mu_0) + \mu_0\lambda}$
t : 時間 I : 総命令数 n : バグ総数 $p(t)$: バグ検出率 $e(t)$: t での残存バグ数	エラーモデル ⁴⁾	$e(t) = \frac{n}{I} - \int_0^t p(t)dt$

するために必要である。テストケースのサンプルによる予備テストから、相当正確にこれを推定できる (採針法)^{20), 21)}。

8.2 保全性の評価

保全作業の時間は次の事項に影響される。

- (1) 保全の対象となるものの特性、障害の性質
- (2) 保全を行う人の技量
- (3) 保全の環境・道具立て

オンラインシステムにおいては、4.2 節で述べたように処理能力の回復がまず目標となるが、このための手順と道具立てはシステムに予め組み込んであり、オペレータが行う処理は少ないのが普通である。次に、データ・ベースの復元については、オペレータの働きが処理の成否と時間に相当影響する。以上の2種の作業の手順と所要時間については、システムテスト期間中に、オペレータの訓練と運転マニュアルの検査を兼ねて十分にテストし評価せねばならない。最後に、バグを実際に修理する作業には、設計文書の質、プログラムのリーダビリティ、システムの診断情報の量と質、プログラムライブラリの更新の容易さなどが影響するが、最も大切な要素は人である。バグ修正に要する時間は、システム開発時には、試験完了時点をきめる大きな要因となる。

8.3 使用信頼性

システム開発側で想定確認した信頼性 (固有信頼性) と、実際にユーザが使用する状態での信頼性 (使用信頼性) とは異なるのが普通である。その差を定量的に把握し、使用信頼性に見合った固有信頼性の目標を予め設定できるようにするのが望ましい。

参考文献

- 1) 塩見弘, 信頼性工学入門, (丸善)
- 2) 塩見弘, コンピュータリライアビリティ, 昭晃堂, (1974)
- 3) W. H. MacWilliams: Reliability of Large Real-Time Control Software Systems, 1973, IEEE Symposium on computer software reliability, April 30 through May. 2, 1973
- 4) 山本, 佐々木: DEMOS-E 用オペレーティングシステム, 電気通信研究所研究実用化報告, 第24巻, 第1号, 1975
- 5) 伊吹公夫: 電子交換機プログラム, 情報処理学会誌, Vol. 15, No. 12, 1974
- 6) 飯村, 岡, 春日, 宮沢: DIPS-1 プログラム製造支援システム, 電気通信研究所研究実用化報告, 第20巻, 第11号, 1971
- 7) 寺島, 細谷, 高橋: システム製造用言語 SYSL, 電気通信研究所研究実用化報告, 第24巻, 第1号, 1975

表-4 バグ総数の予測

使用パラメータ	相 関 関 係 の 例	備 考
プログラムの規模		同一プロジェクト等で類似のプログラムを作成する場合、従来の性格を左図のようにプロットしておくことにより、次のプログラムのバグ予測の概略を把握することができる ⁸⁾¹⁹⁾ 。
プログラムの分枝数とサブルーチンコールの数		プログラムの規模より、そのプログラムのもつ性格（この場合は分枝数とサブルーチンコールの数）による方がバグとの相関が強いという例がある ¹⁹⁾ 。 (go to less のすすめ)
プログラムのモジュール分割度合		プログラムをモジュール化することによりバグ数が少なくなるという例がある ⁹⁾ 。 (モジュール化のすすめ)

- 8) O-J. Dahl, E. W. Dijkstra, C. A. R. Hoare: Structured Programming, 1972, ACADEMIC PRESS INC (LONDON) LTD
- 9) Baker, F. T.: System Quality Through Structured Programming, Proceeding, FJCC, 1972
- 10) Barry W. Boehm: Software and Its Impact: A Quantitative Assessment, Datamation, May, 1973
- 11) K. V. Hanford: Automatic generation of test cases, IBM SYST. J., No. 4, 1970
- 12) 宮沢, 宮本, 杉村: ソフトウェア論理矛盾検出に関する SP 的アプローチ, 昭 50, 電子通信学会全国大会, 1311
- 13) 市川, 小関: フローチャート作成用プログラムの概要, 施設, Vol. 27, No. 1, 1975
- 14) James C. King: Proving Programs to be Correct, IEEE, Vol. C-20, No. 11, Nov. 1971
- 15) 大島, 益井, 宮沢, 長峯: ソフトウェアの品質保証, 電気通信研究所研究実用化報告, 第 24 巻, 第 1 号, 1975
- 16) 川崎義人: 修理系の確率模型, 系のデバッキング過程, 信頼性と品質管理研究会資料, 電気通信学会, 1963. 7
- 17) 塩見弘: 校正モデルと欠陥数予測について, 昭 47, 電子通信学会全国大会, 20
- 18) Martin L. Shooman: Operational testing and software reliability estimation during program development, 1973 IEEE Symposium on computer software reliability. April 30, through May 2, 1973.
- 19) Fumio Akiyama: An Example of software debugging, IFIP, Congress, 1971.
- 20) 宮沢正幸: ソフトウェア完成度予測の一方法, 昭 47, 電子通信学会全国大会, 1192
- 21) 坂田一志: ソフトウェアの生産管理における予測技術の定式化一動的な予測: 先取り評価手法一, 電子通信学会論文誌, 1974, 5, Vol. 57 -D No. 5
- 22) Record of 1973 IEEE Symposium on Computer Software Reliability, April 30 through May 2, 1973
- 23) Proceedings of International Conference on Reliable Software, 21-23 April 1975
(昭和 50 年 7 月 17 日受付)