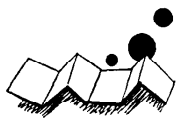
解　説



ソフトウェアにおける信頼性†

鳥居宏次†† 杉藤芳雄†† 大時和仁††

1. はじめに

ソフトウェアに関して，現場では生産性向上ということが目標である．それとともに良質のソフトウェアでなくてはならないという目標もあって，これに関しては高信頼性のものでなくてはならないといういい方も広く用いられている．

たしかに，信頼できるものを期待する気持はわかるが，高信頼性という裏には，正しく動作するソフトウェアは望むべくもないから，誤りができるだけ少ないことを期待するという意味が含まれているはずである．

単に誤りの多少だけではなく，良質のソフトウェアについて議論したいというのが，ソフトウェアの評価 (Software Evaluation) とか，測度 (Software Metrics) と呼ばれるものである．

したがって，ソフトウェアにおける信頼性というイメージは，測度の中の一つの評価基準として使われており，実感は人それぞれによって違っている可能性もある．

信頼性の議論は，誤りを意識して初めて存在する概念であるが，誤りを起こさないようにする立場と，誤りが起きてしまったらどのように処理すべきかとの立場がある．チューリング機械でどのような計算も可能であることの類推から，少し無節操な表現をとると，ソフトウェアを作りさえすれば，かなりの計算，ひいては処理も可能になるはずである．ここで，ソフトウェアが作れさえすればよいわけだが，最近のように，大規模で複雑な処理が期待される場合，上記の二つの立場はどちらも，現状では中途半端でしかないというべきである．

本稿では，上記二つの立場からソフトウェアの信頼性なる概念を眺め，整理するとともに，わずかな私見を述べるつもりである．

第2章では，信頼性の裏に存在するはずの，誤りということがソフトウェアの特殊性を考慮した時どのようなものであるかを，ハードウェアでの誤りと比較しながら整理する．

第3章では，誤りのないソフトウェア作成の立場の議論を，プログラミング作成手法，プログラムの正当性，プログラミング言語の意味記述の点から議論する．これは，fault avoidance と呼ばれる考えに基づく．

第4章では，二つの立場のうち，後者として，ある程度の誤りは覚悟して，その影響を最小限にしようとする，fail safe とか fault tolerant と呼ばれる考えをわずかな具体例を通して議論する．

第5章では，信頼性に関連すると思われるいくつかの関連事項を述べるとともに，今後についての私見を含めてまとめとする．

2. ソフトウェアにおける信頼性の概念

2.1 誤りの分類

信頼性の議論をするには，誤りとは何を指すかを明確にしておかなくてはならない．虫，バグなどソフトウェア独特の単語もあるが，fault, failure, error などもそれなりに使われてきた．MTBF (Mean Time Between Failure) とか，fault tolerant, error correcting code などは，他の言葉でのいい替えはきかないようだ．

同様に，ソフトウェアでは虫やバグの他に，エラーまたは「誤り」が普通の意味で使われている言葉である．

本稿では，誤りという言葉を積極的に使うことにするが，必要ならば上述のような，フォールト，フェイラ，エラーなど従来から使い慣れている言葉の意味で，カッコ内にそれぞれの英語を書きそえることにする．

† Reliability on Software by Koji TORII, Yoshio SUGITO and Kazuhito OMAKI (Computer Science Division, Electrotechnical Laboratory).

†† 電子技術総合研究所ソフトウェア部

ソフトウェアの誤りを議論するまえに，信頼性という概念が広く使われてきたハードウェア面での誤りと対比しておよその分類を考える．

ハードウェア面での誤り (failure) の原因の代表としては，

(1) 基本部品の故障，

(2) 回路構成上または論理上のミス，

(3) 装置が受けつけ得ない状態になる外部原因

などがあげられる．

これらに対し，ソフトウェアではどのような原因が考えられるのであろうか．まず存在するはずのないのが(1)の基本部品の故障である．アセンブリ言語の命令セットは常に正しく実行されるのである．いうまでもなく，ハードウェアの部品の故障に対する備えはされなければならず，ソフトウェアによってある程度自動的な故障部品の検出は行われているのであろうが，その議論はソフトウェアの誤りの範囲外である．

(2)の論理的なミスについては，むしろソフトウェアで最も重要視されている原因である．とくに，ソフトウェア設計段階での論理的な誤りを避けるべきであると考えられており，不注意なミスとは本質的に区別されている．ここでの不注意なミスというのは，文字の書き誤りなど些細な誤りというものであり，人間である限り仕方のないことであるし，それに対しては，最近のソフトウェアではかなりの手当が期待できる．

(3)の受けつけ得ない入力値がかかるということも，ソフトウェアではきわめて自然な誤りで，いわゆる入力データの誤り (error) といわれるものに対比できる．ただし，入力データの誤りは，さらに2種類に分類されるべきである．すなわち，

(i) 予期していた誤り

(ii) 予期していなかった誤り

である．

たとえば，システムが次に加減乗除からなる算術式が入ってくるものと期待しているときに，$a++b$ が入ってくれば，これは明らかに構文上の誤りであって，システムはそれなりの対策を立てうるはずである．一方，同じ状態でコードセットにないコードが入ってくると，構文以前の誤りを起すことになり予期していなかった誤りに属する．

単にこのことは，入力データ誤りだけではなく，ソフトウェアのバグでもよく生ずることである．端的な例は，バグのために決まった配列の範囲を越えてまで書き込んでしまうプログラムの場合，命令部分まで壊してしまって，プログラムの暴走ということもよくあることである．

予期しうる誤りについては，正しくない (invalid) データと呼び，予期し得ない場合に対しては，悪い (bad) データと呼んで区別することもある[16]．

なお，実時間 (real time) ソフトウェア，とくに，プロセス処理などの場合，システム全体としてはハードウェアをも組み込んでいるため，ハードウェア誤りをも考慮したソフトウェアになっているはずであり，上の分類そのままにはなっていないのは当然である．

2.2 誤り対策と信頼性

誤り訂正符号に見られるように，誤りを検出した上で，訂正することまで可能な場合がある．しかし，検出や訂正は十分準備された上で，厳密な処理が可能な場合しか望めず，通常は，その誤りが致命的な影響を与えないようにすることである．それが fault tolerant とか fail safe の精神であろう．

一方，ソフトウェアにおいては，ごく限られた場合には誤り検出や，それに伴う誤り訂正が行われつつある．限られた文法での構文解析や，知識データベースを利用した，綴り字の誤り訂正などは実用的になっている．しかし，誤りを致命的にしないために，その部分だけに誤りを抑え込もうとか，その誤りを避けた別の実行径路を作ろうとする上述のような試みは，主に論理的ミスに起因することの多いソフトウェア誤りでは，限られた場合以外無理というべきである．

数少ない可能な例は，多くの独立なプロセスが走っているようなオペレーティング・システム (OS) においてである．問題になった特定のプロセスのみに機性を停めておくことであり実用的にも意味がある．しかし，ほとんどのソフトウェアのバグは簡単にはいかず，そのために，汎用のソフトウェアでは例外処理 (exception handling) の対策として，かなりの部分が費やされることになる．

冗長性の導入や，多重化により信頼性を高めようとするハードウェア中心の考えはソフトウェアにはほとんど通用しない．

したがって，現実のソフトウェアの誤り対策としては，生じた誤りの対策を立てることには，ほとほと手を焼いており，誤りを生じさせないようにすべきというのが最近の傾向である．上述の fail safe や fault tolerant に対して，fault avoidance という考えである．当然，生じた誤り対策と，誤りを事前に防ぐという二つの考えは従来から存在することだが，特にソフ

トウェアにおいては，技術が十分熟していないためか，後者を重要視する．それでもなお現実のソフトウェアでは事故対策に追われるわけであるが，プログラミング手法などを中心とした，1970年代のソフトウェア工学は，いかに誤りのないプログラムを作るかという一点に議論の対象が絞られてきた．

本稿でも，以後はこの二つの面から全体を組みあげていく．

3. Fault Avoidance

3.1 プログラミング方法論と正当性

現実的なプログラミング方法論と呼ばれるものは，およそ fault avoidance をねらったものであるといえる．ソフトウェアにはライフサイクルがあるといわれる．解くべき問題の記述，すなわち要求仕様なる記述から始まって設計，コーディング，テスト，デバッグ，文書化，保守，管理といった経緯を指している．絶対的期間は個々について異なり，10年以上も使われなくてはならないソフトウェアも存在しよう．長生きすればするほど，大きければ大きいほど保守管理の面からも誤りが少ないこと，あるいはないことを期待することになる．

1970年代はプログラミング方法論やそれを支援するプログラミング言語など華やかな提案が相次いだ時代である．その辺の話については，信頼性と関係させての議論は必ずしも容易でないが，本特集の松本氏によるものがあるため，詳細はそちらに譲ることにする．

話は固くなるが，誤りのないプログラムを作ることが信頼性を高める理想的な方法であることは当然である．すなわち正しいプログラムを作る方法の議論であって，Dijkstra などは，数学の定理における証明を考えるように，プログラム証明を考え，その証明過程をプログラミング言語で書き直したものがプログラムであるとするものである．

しかし証明から出発するのは必ずしもうまくいく保証がないため，でき上ったプログラムが正しいことを証明しようとするのが次に現実的である．しかし，そこで問題になることは，プログラムが正しいということは，どういうことかであり，この点を明確にしておかなくてはならない．

人間が解きたい問題がまず存在していて，すべての場合について期待通りの解答を与えてくれるプログラムが存在すれば，それは正しいプログラムといえるであろう．そのためには，人間が考えている，解きたい問題というのは何かということが，それ以前に明確になっていなくてはなるまい．いわゆる，問題の仕様が正確に記述されていなくてはならない．そこで重要になるのが，厳密な仕様記述方式である．厳密性は形式的にも正確であることに通ずるため，形式的な（厳密な）仕様記述法がいくつか提案されている．しかし，ほとんどの手法は，あまりにも厳密性を追求しているため，現実のソフトウェアには通用しにくいものになっている[31]．

さらに一歩話を進めるならば，仕様言語による記述であれ，プログラミング言語による記述であれ，その記述がどういう意味を指しているかが明確になっていない限り，あいまいな話になる．

repeat A until B とか，while B do A とかの繰り返しを指示する構文がよくある．このとき，たとえば repeat 文において始めから B が真なら，A は実行されるのかどうかがよく問題になる．同様に，while 文で B が始めから偽であるときはどうなるのかである．通常この例ではほとんど問題がないが，一般的に現実場面では，各言語の処理系により異なることがあると考えておくのが常識になっている．これらの様子は，普通は口答で，または，マニュアルで日本語や英語で説明されていて，納得させられることになっている．しかし，マニュアルというのは，すべてを信用してはいけないと誰もが考えている．厳密さでは，欠けた部分やあいまいな表現，矛盾した内容などが混在していることになっていて，信用すべきは，自分が使っている処理系がどのような結果を出すのか自分で調べるのが最もよいとさえ考えるようになっている．

これは，いいかえれば，言語の意味が明確に表現できていないために生ずることで，さかのぼれば，プログラムの正当性の議論は，それを記述する言語の意味に大きく起因するということになる．

こういう経緯から，厳密な意味記述方式がいくつか提案されてきた．どれをとっても，現実のソフトウェアに応用できるほど簡単なものでないように思えるが，基本的には避けて通ることのできない道である．信頼性の話の中に形式的意味記述の話題が現われるのは少し唐突な感じもするが，簡単でうまく説明されている資料が記憶にないため，次節で代表例を簡単に整理しておく．

3.2 意味記述と形式的意味論

“意味記述”とは，一般的にはある対象 O がもっている「意味」をある言語 Ls で記述あるいは定義する

ことであるが，Oの種類により次の3通りに大別されると考えられる．別の表現をすると，ややあいまいな用語である．

（1） Oが概念や問題である場合

たとえば，スタックなどの「概念」を抽象データ型として定義することや，巡回行商人などの「問題」の仕様を与えることを，Ls により行うことである．いわゆるプログラミングも，概念や問題（およびその解法）をプログラミング言語 Ls により表現する作業とみなすことが可能であろう．

（2） Oが（プログラミング用や仕様記述用の）言語 Lo そのものである場合

言語はシンタックスとセマンティクス（さらにプラグマティクスを含めることもある．）とから構成されるものであるとの考え方に立脚して，当該言語 Lo のセマンティクスを Ls により記述することをいう．これは言語 Lo の仕様を定義する場合に，とくに必要とされることである．

（3） Oがすでにある言語 Lo により記述されているもの（たとえばプログラムや仕様）である場合

当該Oをいかなる目的で“意味記述”するかにより，2通りの用法がある．

a）Oを検証する目的のとき

検証用の論理体系をもつ言語 Ls により，Oの意図するところ（あるいは内容）を記述することに相当し，たとえば Lo で記述されているプログラムに関する表明 (assertion) を述語論理（言語）Ls で記述することである．この場合，一般には Ls≠Lo であるが，Ls=Lo ならばプログラムを検証向きに書き直す手間が省略されることになり，論理プログラミング[19]の考え方となる．

b）Oを翻訳する目的のとき

Lo で表現されているOを，Ls≠Lo なる言語 Ls により，意味的に等しい記述に変換することに相当し，たとえばコンパイラではいわゆる意味付けルーチンにより実現される．また，プログラムのコメントもこの場合の一種と考えられよう．

以上の（1）～（3）において，Ls の表現形式は目的や用途に応じて自然言語に基づくものから形式意味論に基づくものまで多様である．

次に，形式意味論といわれることもあるが，プログラミング言語（および仕様記述言語）のセマンティクスを，形式的な体系により把握しようとする研究がある．

その主な目的は，言語の設計者，言語プロセッサ作成者，言語の利用者の三者間で共通に指針となる明白であいまいさのない言語の定義を与えること，およびプログラム（および仕様）の形式証明への基礎を与えることである．とくに後者の目的により，形式意味論は正当性，ひいては信頼性と深い関係がある．形式意味論は発展途上にあり，そのアプローチも多種多様であるため，統一的見解が成立しにくい状況にあるが，以下に代表的アプローチを記す．

（1） 構文指向形意味論 (syntax oriented semantics)

コンパイラを強く意識した意味論であり，構文の解析や変換を行うたびに意味的属性や意味付けルーチンを割り当てる方法である．たとえば，生成システム (production system)[7]，属性文法 (attribute grammar)[18]，W-文法 (W-grammar)[32] などがある．

（2） 公理的意味論 (axiomatic semantics)

公理的な論理体系（公理と推論規則）により意味記述を与えるものでありプログラムの証明に主たる関心がある．Floyd-Hoare の方法はその一例である．この方法を直接実際のプログラムの検証に適用することは困難であることが多いが，現場のテスト用に用いようとしたシステムに，McDonnell Douglas 社の PET (Program Evaluator and Tester) がある[30]．PET は Fortran のソースプログラムのテスト用に開発されているものであり，プログラムの先頭または途中に変数の変域等々に関する表明 (assertion) を与えておき，実行途中にその表明に関する情報を出力するものである．

（3） 操作的意味論 (operational semantics)

抽象機械（とくに状態遷移機械）を想定し，その動作系列を与えることにより意味記述を与えるものである．チューリング機械，Landin の SECD 機械[20]，PL/I の Vienna 定義法[3]はその例である．

（4） 表示的意味論 (denotational semantics)

言語の構成要素から，ある数学的実体（整数，集合，関数など．これらは表示される物 (denotation) として用いられる．）への写像として「意味」を定義する立場であり，数学的構造は豊かである．数学的意味論 (mathematical semantics)．不動点意味論 (fixed-point semantics)[23]，Scott-Strachey 法[29]ともいう．

（5） 代数的意味論 (algebraic semantics)

抽象プログラムおよび denotation をそれぞれある特別な代数（イニシャル Σ-代数 (initial Σ-algebra)

と呼ばれる.)と考え，抽象プログラムからdenotationへの写像として「意味」を定義するが，この写像に関して二つの代数間の準同形写像とみなす立場であり，とくに抽象データ型の形式的仕様記述（たとえばGoguen[10]）の分野で発展してきたものである．

以上の(1)～(5)のアプローチ間の境界は明瞭ではなく，あくまでも目安にすぎない．たとえば，「(4)は(5)の一種である」とも，「(5)は(4)の一種である」とも考えられよう．

さて現実に直接計算機実行が可能な仕様の記述方法があれば，仕様とプログラムの相異という問題は生じない．このような考えのもとに，仕様の形式化とその実行システムの実働化の研究が行われている．たとえば，仕様を等式の集合として記述し，書き換えシステムとして実現する，代数的仕様記述の実働化研究がある[9]．これは最近の関数型言語によるプログラミングのアプローチとも密接な関係があり，多くの研究が行われている．また，問題の仕様をモジュール化し，属性文法の概念を用いてモジュール間の階層化と入出力関係とを表わし，さらにその属性文法を実行可能な形にコンパイルしようとする試みもある[17]．これも，直接実行可能な仕様の書き方を指向していると考えられる．

3.3 信頼性の測定尺度

（i） テスト(検査)と検定

プログラムが完全に正しいというのがかなり困難であるとすると，次にはどの程度正しいかを知りたくなる．案外その辺が信頼性という言葉の感触に似てくることではあるが．どの程度の虫(bug)がとれたかを調べるのが検査であり，プログラムのテストと称され，現状ではきわめて需要の多いものである．

プログラムのテストについては本特集の岸田氏の詳しい解説があるため，ここでは詳細は省略する．しかし，テストは上述の正しさと本質的に異なるもので，いくつかのテストデータを実行し期待通りの解答が得られるか否かを判定するだけであるため，完全なfault avoidanceではなく，どの程度避けることができたかの目安を得る程度のものと考えるべきである．

まず，コンパイラのテストについて述べる．コンパイラは，その入力であるソースプログラムの言語の規格が文書の形ではあるが比較的正確に定められているため，テストしやすいプログラムであるといえるかも知れない．アメリカ国防省はコンパイラのテストに対して熱意があるらしく，CobolやFortranのコンパイラ検定システム（それぞれCCVS[2]，FCVS[14]と名付けられている）を開発している．これらのシステムは規格を満たす数百個のテスト用のソースプログラムを用意しておき，これらを性能テストされるべきコンパイラに入力してコンパイルする．そして，これらを実行してみて所期の出力が得られるかどうかによって，そのコンパイラの検定をしようとするものである．テスト用のソースプログラムは，テスト結果の出力ルーチンを含むように作られていることが望ましいという．

一方，各種のプログラムを個別にテストするシステムも開発されている．前述したPET[30]もその一つと考えられる．しかし，PETは文献で見る限り，いわゆるトレーサに似ているため，その出力結果からプログラムのエラーをみつけるのは，それほど容易ではないと思われる．もっとも現場ではこの種のツールが有効に使われているのかも知れない．

ところで，プログラムのテストは入力データを与えることによって行われるべきであると考え，そのためのテスト用の入力データを，テストされるプログラムから如何にして作ったらよいかという研究がある．考えられるすべての入力の組み合わせを作ってテストする訳にはいかないので，プログラムの構造に基づいてテスト用のデータを作るという考え方があり[15]，日本でも各種のシステムが作られている[26]．これらは，プログラムのすべてのpathを一度は辿るように記号実行を行い，それらのpathを辿れるようなデータを作るものである．しかし，このように記号実行によってpathを調べることは時間的効率が悪いので，必要と思われるテストデータをプログラマが会話的に入力し，システムは二つの入力データに対して異なるpathをみつけてプログラマに教えたり修正したりする方が現実的であるとする提案もある[5]．

一方，DAVE[8),27]はもう少し受動的なシステムではあるが，プログラムの構造に対して区間解析(interval analysis)[1]を行い，参照される変数に前もって値が代入されているかどうかなどの診断メッセージを出す．区間解析のアルゴリズムは理論的によく研究されているが，現実のプログラムに対する処理の時間は比較的大きい．しかし，実際に動くシステムであるというのは，それだけでも立派なことである．

（ii） 確率的モデルによる計量

これまで述べてきたソフトウェアの評価の立場とちがって，確率論的に信頼性を計量しようとする試みも

数多くある[12),21),24)]．そこではソフトウェアをモデル化するためのいくつかの仮定が設けられている．たとえば，ソフトウェアの誤りは互いに独立にある確率分布に従って生起するとか，エラーが検出される確率は残りのエラーの個数に比例するとかいった仮定が設けられている．そして，ある計算センタのOSの稼動状況を計測したところ，確率的モデルが非常によい近似になるという報告がある[12)]．また，OSだけでなく，多くの応用プログラムの開発過程においても，論理エラーの収集やそれによるプログラムの評価に，確率的手法を適用したプログラム作成システムを作ろうとする試みもある[22)]．しかしながら，プログラムによってはモデル化の際の仮定が妥当でない場合もあり，適用には慎重を要する[22)]．

4. ソフトウェアにおける fault torelance

一般には fault torelable なソフトを作ろうとするとき，ハードとはちがって同じソフトのコピーを作っておいて，それに制御を渡せばよいというものではない．しかし，ロケットの弾道計算や飛行機の航路計算などのような実時間処理の分野においては，異なった方法で同じ計算をする二つのプログラムを用意しておいてバックアップさせる方式がとられる[13)]．たとえば，通常は一つのセンサからのデータを入力して計算するプログラムが走るが，データに異常が生じたときは別のセンサのデータを入力して同じ計算をするプログラムが走る．データが異常であることの判定は，たとえば温度や速度などが短時間に急激には変化しないので，急激な変化をしているデータがセンサから入力されたら異常データと判定したりする．また，同じ種類の計算機で全く同じ計算を行い，その多数決をとるという方法も採られている．このように，実時間処理の分野では，ソフトによるバックアップということが重要な要素となる．

この例は，センサというハードの異常に対してバックアップのプログラムを用いようとするものである．これに対して，プログラムには論理エラーがつきものであり，それを免れる良い方法が今のところないので，論理エラーが生じたとき同じ計算を行うバックアップ用のプログラムが起動するように，プログラム全体を作成しようという考え方がある[28)]．これはたとえば，sortのプログラムを作ろうとするとき，実行前にデータのテストを行い，ある条件（そのプログラムを実行する直前で期待される条件）を満たしていたらクイックソートを行い，万一その条件が成立しない場合を想定してバックアップ用にバブルソートのプログラムが起動するようにしておく，といった作成方法である．

一方，OSにおいては一つの failure が他の資源やプロセスにとって致命的なものにならないような配慮が特に必要である．そのためのソフト及びハードの構成方法が研究されている[6)]．命令を，スーパバイザコールのモードと，そうでないモードとに分けて，ハード的にOSの核を保護するということは古くから行われている．

5. おわりに

システムが信頼しうるという時には，たとえば，誰からも大切なものが盗まれたり，邪魔されたりすることもないという場合があるかもしれない．最近ニュース記事になるコンピュータ犯罪も，利用者から見れば信頼できないと考えることになる．しかし，この議論は安全性（Security）の問題として，古くはOSやファイルでのアクセス権や，データベースでのセキュリティ，符号理論から発展しつつある暗号化などとして取り扱われており，本稿ではソフトウェアの信頼性の議論外だと考えている．

本稿で触れなかった中に，ソフトウェアの品質評価や測度に関して，ソフトウェア科学（Software Science）[11)]と名付けられた，一種独特の考え方がある．プログラム中の変数の個数が行数に対して多いプログラムは質が良くないのではないか，など定性的に直感で感じている事実を，定量的に取り扱おうとする試みであって，ニュートン力学の法則の類似をとるなど，ユニークな研究である．その是非はまだいい切れないとは思うがそのユニークさには大いに興味を憶えるものであって，定量化の執念が感じられる．全面的ではないにせよ，心に停めるべきであって何らかの役に立つと思えるものである．

本稿では，fault が起こらないようにする avoid の立場と，起こったあとの処理の態度としての torelant とか safe とか呼ばれる立場とから眺めてきた．英語の方では Reliable Software という言葉が70年代の中頃から広く使われ始めているが，さほど厳密な意味もなく使われつつあるようであり，今後もケース・バイ・ケースの意味あいで使われると想定されるが，本稿が何らかの位置付けをしようとする時の参考になれば幸いである．

参 考 文 献

1) Aho, A. V. and Ullman, J. D.: The Theory of Parsing, Translation, and Compiling, Vol. II, Prentice-Hall Inc. (1973).
2) Baird, G. N.: The DOD COBOL Compiler Validation System, Proc. FJCC, pp. 819-827 (1972).
3) Bandat, K.: On the Formal Definition of PL/I, Proc. FJCC, pp. 363-373 (1968).
4) Dahl, O.-J., Dijkstra, E. W. and Hoare, C. A. R.: Structured Programming, Academic Press (1972).
5) DeMillo, R. A., Lipton, R. J. and Sayward, F. G.: Hints on Test Data Selection: Help for the Practicing Programmer, IEEE Computer, pp. 34-41 (1978).
6) Denning, P. J.: Fault-Torelant Operating Systems, Comput. Surv., Vol. 8, No. 4, pp. 359-389 (1976).
7) Floyd, R. W.: A Discriptive Language for Symbol Manipulation, J. ACM, Vol. 8, No. 4, pp. 579-584 (1961).
8) Fosdick, L. D. and Osterweil, L. J.: Data Flow Analysis in Software Reliability Comput. Surv., Vol. 8, No. 3, pp. 305-330 (1976).
9) Futatsugi, K. and Okada, K.: Specification Writing as Construction of Hierarchically Structured Clusters of Operators, Information Processing 80, North-Holland, pp. 287-292 (1980).
10) Goguen, J. A., Thatcher, J. W. and Wagner, E. G.: An Initial Algebraic Approach to the Specification and Implementation of Abstract Data Types, Current Trends in Programming Methodology, Vol. IV, Data Structuring (Yeh, R. T., ed.), Prentice-Hall (1978).
11) Halstead, M. H.: Elements of Software Science, Elsevier North-Holland Inc. (1977).
12) Hamilton, P. A. and Musa, J. D.: Measuring Reliability of Computer Center Software, Proc. 3rd International Conference on Software Engineering, pp. 29-36 (1978).
13) Hecht, H.: Fault-Tolerant Software for Real-Time Applications, Comput. Surv., Vol. 8, No. 4, pp. 391-407 (1976).
14) Hoyt, P. M.: The Navy Fortran Validation System, Proc. NCC, pp. 529-537 (1977).
15) Huang, J. C.: An Approach to Program Testing, Comput. Surv., Vol. 7, No. 3, pp. 113-128 (1975).
16) Jackson, M. A.: Principles of Program Design, Academic Press (1975).
(邦訳 鳥居宏次: 構造的プログラム設計の原理, 科学技術出版社 (1980).)
17) Katayama, T.: HFP: A Hierarchical and Functional Programming Based on Attribute Grammar, Proc. 5th International Conference on Software Engineering, pp. 343-352 (1981) など.
18) Knuth, D. E.: Semantics of Context-Free Languages, Mathematical Systems Theory, Vol. 2, No. 2, pp. 127-145 (1968).
19) Kowalski, R.: Predicate Logic as Programming Language, Information Processing 74, North-Holland, pp. 569-574 (1974).
20) Landin, P. J.: The Mechanical Evaluation of Expressions, Comput. J., Vol. 6, No. 4, p. 308 (1964).
21) Littlewood, B.: A Reliability Model for Markov Structured Software, Proc. International Conference on Reliable Software, pp. 204-207 (1975).
22) Littlewood, B.: How to Measure Software Reliability, and How Not to..., Proc. 3rd International Conference on Software Engineering, pp. 37-45 (1978).
23) Manna, Z.: Mathematical Theory of Computation, McGraw-Hill (1974).
24) Miyamoto, I.: Software Reliability in Online Real Time Environment, Proc. International Conference on Reliable Software, pp. 59-71 (1975).
25) Miyamoto, I.: Toward an Effective Software Reliability Evaluation, Proc. 3rd International Conference on Software Engineering, pp. 46-55 (1978).
26) 中島, 上村, 斎野: ソフトウェアに品質評価の時代が来る, 日経コンピュータ, pp. 38-46 (1981. 10. 19).
27) Osterweil, L. J. and Fosdick, L. D.: Some Experience with DAVE-A Fortran Program Analyzer, Proc. NCC, pp. 909-915 (1976).
28) Randell, B.: System Structure for Software Fault Tolerance, IEEE Trans. Softw. Eng., Vol. SE-1, No. 2, pp. 220-232 (1975).
29) Scott, D. and Strachey, C.: Towards Mathematical Semantics for Computing Languages, PRG-6, Oxford Univ. Press (1971).
30) Stucki, L. G. and Foshee, G. L.: New Assertion Concepts for Self-metric Software Validation, Proc. International Conference on Reliable Software, pp. 59-71 (1975).
31) 鳥居, 二木, 真野: プログラミング方法論の展望, 情報処理, Vol. 20, No. 1, pp. 22-43 (1979).
32) van Wijngaarrden et al.: Report on the Algorithmic Language ALGOL 68, Numer. Math., Vol. 14, No. 2, p. 84 (1969).

(昭和 56 年 12 月 18 日受付)