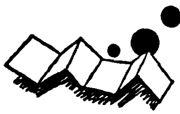
解　説



プログラミングの形式化とソフトウェア・ツールのありかた†

謝　章　文††

1. はじめに

ソフトウェア・ツールとは何か？ その背景にある思想は，いったいどのようなものなのか？ ソフトウェア工学におけるその役割りを，いかなる観点から考察し，その現状を評価し，さらに，その正しいありかた，および将来像を描くべきか？

これらの問いかけに対して真剣に取り組もうとするならば，これらと密接に関連するソフトウェア工学における，より基本的な次の問いかけに対する答を用意しなければならない．

なぜ良いソフトウェアを作ることが非常にむずかしいのか？ 逆に，どうしてプログラムは誰にでも作れ，誰にでもプログラマになれると信じられているのか？ いったいプログラミングとは何なのか？ ソフトウェア工学はその工学体系あるいは技術体系の基礎を支えるいかなる理論をもっているのか？ そこで必要な理論はどのようなものなのか？ プログラミングは科学になれたのか，またはいまだに手工芸なのか？ もし科学になれるならば，プログラミングは数学とどう違うのか？ プログラミング技術をどう習得し，どのように指導すればよいのか？

これらの問題の提示は，1960年代後半のいくつかの国際会議の席上においてなされ，また E. W. Dijkstra や国外国内のソフトウェア関係の有識者によっても幾度となく繰り返され，“ソフトウェアの危機”の認識は広く行き渡り，それによって，“ソフトウェア工学”，“プログラミング方法論”，“ソフトウェア・ツール”，新しくは，“要求定義技術”などの用語といくらかの概念や手法が考察されてきた．しかし，問いかけに対する解答は与えられたのだろうか？ 残念ながら“否”と答えざるをえない．

2. ソフトウェア・ツールとは

ソフトウェア・ツールになるような“良いプログラム”の具体的な作成の手引き書である，B. W. Kernighan & P. J. Plauger の “Software Tools” によると，ソフトウェア・ツールの条件は (1)計算機を使用すること，すなわち，プログラムであること (2)特殊な場合でなく，一般的な問題を解決するものであること (3)非常に使い易く，人々が自分のプログラムを作成する代りにそれを使うようなものであること (4)幾つかまとめて使えること，すなわち，システムを構成できることなどである．また，ソフトウェア・ツールは，“良いプログラム”であり，“良いプログラム”の条件は (1)“良い設計” 修正，保守が容易であるもの (2)人間工学的 大変便利に使えること (3)信頼性 正しい答が得られること (4)効率 十分使用に耐えること　などである．

“ソフトウェア・ツール”という用語は，以上から明らかなように (1)ソフトウェアの品質に関する側面 (2)ツールとしての側面　を持つ．

ソフトウェアの品質に関する側面は，ソフトウェア・ツールとはある品質以上のソフトウェアに対する呼称であるとする考え方である．これは，ソフトウェアという用語の意味する対象の範囲が，質的にも量的にも非常に広すぎ，一括して議論できないような場合が多々あるため，より厳密な分類を行おうという流れにそうものである．

ソフトウェア・ツールのツールとしての側面およびそこにおける基本思想は，ソフトウェア工学，すなわちその対象である大規模・高品質ソフトウェアの製造開発における本質的困難の認識から生じたと思われる．

これ以後，ソフトウェア工学における最も重要でかつ基礎的な，他のプログラムの開発を助けるプログラムであるソフトウェア・ツールに対して考察してゆく．したがって，このあと，ソフトウェア・ツールと

---

† Software Tools and Formalization of Programming by Akifumi SHA (Institute of Computer Sciences, Kyoto Sangyo University).

†† 京都産業大学・計算機科学研究所

は"ソフトウェア工学ツール"という狭義の意味で用いる.

3. メタソフトウェア工学

ソフトウェアの開発は，メタソフトウェア工学的に考察すると，大きくつぎの4つの部分からなる.

i. 現実世界の目的を把握し，仕様書という記述にする作業

ii. 仕様書という記述から，プログラムという記述への変形作業

iii. プログラムを実行し，現実世界で運用する作業

iv. 現実世界で，プログラムの運用の結果が目的（要求）を充たすかどうかという評価作業

目的から仕様書への作業は，従来，システム要求，ソフトウェア要求，基本設計および詳細設計という工程で行われている．目的と仕様書の関係は，一方は現実世界の，他方は記号の世界の存在物であり，これらを形式的理論体系内で処理することはできない．この作業における唯一の理性的方法は人間機械系におけるフィードバック・ループによる人間工学的手法である．この要求から記述への変換は，仕様言語の意味論(Semantics) により定まる仕様書（記述）の意味と要求との適切性に関する評価が伴わなければならない．仕様言語の構文論 (Syntax) および意味論を適当に形式化すれば，この適切性の評価の基礎である，仕様書の無矛盾性（仕様書を充たすモデルが存在するということ）および完全性（必要十分であること）の判定を形式的理論体系内で取り扱うことができる.

仕様書からプログラムへの変形作業は，従来コーディング・デバッグの工程で行われ，それを補完するものとして，テスト・保守（修理）の工程の一部が存在する．この仕様書からプログラムへの変形は，仕様書に対するプログラムの正当性の判定が伴わなければならない．仕様書とプログラムの関係は，両者とも記号の世界の存在物であり，変形および正当性の判定は，仕様言語とプログラム言語の構文論および意味論を形式的に十分なものを採用すれば，形式的理論体系内で取り扱える事柄である.

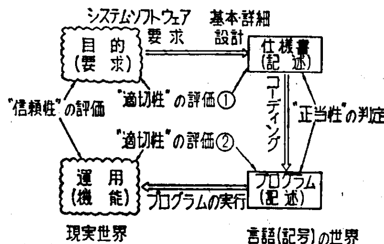


図-1　ソフトウェア開発のメタソフトウェア工学的分析

プログラムを実行し運用する作業は，従来，運用・保守というソフトウェア・ライフサイクル中最も期間の長いものである．この作業はプログラム言語の意味論により定まるプログラムの意味を計算機による実行という1つの解釈 (interpretation) で，機能として実現することであり，プログラムの意味と機能との適切性に関する評価を伴わなければならない.

プログラム言語は正当性の判定のためには，仕様言語と共通の指示的意味論 (denotational semantics) を，運用に対する適切性の評価のためには，実行解釈を規定する操作的意味論 (operational semantics) を持たなければならない．また，2つの意味論は整合していなければならない．操作的意味論はハードウェアとソフトウェアのインタフェイスの役割りを担う.

目的とソフトウェアの運用の結果は，両者とも現実世界の存在物であり，そこでの信頼性，効率および操作性に関する品質評価は，"十分間に合う"とか"満足できる"という基準概念に基づく"良い"とか"合格"とかいう類の判断である．ソフトウェアの信頼性の概念は，つぎの3つの概念と緊密な関係を持つ：

i. 目的（要求）に対する仕様書の適切性

ii. 仕様書に対するプログラムの正当性

iii. プログラム言語に対する実行解釈の適切性

効率・操作性の概念は信頼性の概念ほど簡単（基本的）なものでなく，現在，効率に関しては，プログラムに対する計算の複雑さ(Computational Complexity)の解析やフロー・アナリシスなどの理論的アプローチが進められている.

4. 論理的プログラム合成から

仕様記述言語およびプログラム言語の構文論および意味論が十分形式化されているとき，仕様書からプログラムへの変形が十分理論的に行えることを論理的プログラム合成 (Logical Program Synthesis, LPS)[2,3] の原理に基づいて例証する.

仕様記述言語は，関数記号を含む一階述語論理の言語とし，仕様書はそのスコーレム標準形，すなわち節形式 (clause form) で書かれる．論理式 (formula) から論理節 (clause) への変形アルゴリズムが存在することはよく知られている.

```
《Specification》
~Conjecture:
−10. (X Y) −R(x, y)
Axioms:
1. ( ) S(nil)
2. ( ) I(nil,nil)
3. (L M) R(l,m) v −S(m) v −I(l,m)
4. (L M) −R(l,m) v S(m)
5. (L M) −R(l,m) v I(l,m)
6. (L M) −S(m) v S(merge(l,m))
7. (L M N) −I(l,m) v I(cons(n,l), merge(n,m))
8. (L M)=(l,nil) v −R(cons(car(l), cdr(l)), m) v R(l,m)
Executable predicates ............(=)
Non executable predicates ......(S I R)
Output functions ..................MSORT
Primitive functions ...............(MERGE CONS CAR CDR)
Input variables ......................(X)
Output variables.....................(Y)
Medium variables ...................(L M N)
Constants ......... ....................(NIL)
```

図-2 形式的仕様書の1例 (MSORT)

反証 (refutation) とは，帰謬法による証明 (proof) のことである．

Sort のための形式的仕様書は，**図-2** のようなものである．

図-2の各論理節は，その左側に，番号と出現する変数リストをもつ．論理節-10は入力変数 X，出力変数 Yを自由変数として含む入出力関係を記述している開論理節であるが，反証を行うためその否定形で与えられている．公理 (Axioms) である論理節 1〜8 は，その変数を全称量化子 (universal quantifier) で束縛した通常の節形式である．

この形式言語は，自然な解釈 (original interpretation) のもとでは，つぎのように読める．

$R\ (x,y)$: yはxをソートしたものである．

$S\ (x)$: xは昇順に並んでいる．

$I\ (x,y)$: xに含まれる要素とyに含まれる要素は等しい．

この問題における要求仕様は，つぎのようになる．

- xを与えられると $R(x,y)$ をみたすYを作る．
- $\forall_x \forall_y\ \mathrm{R}(x,y) \equiv I(x,y) \wedge S(x)$.

上の条件は，論理節-10 と入出力変数の指定として，下の条件は，論理節 3〜5 として与えられている．

設計仕様は，つぎのようなものである．

- merge を基本演算とする．すなわちアルゴリズムとして Merge Sort を選ぶ．
- データ構造はリストとする．
- 他の実行可能な基本演算として；cons, car, cdr, nil, 基本述語として；=(EQ)，すなわち，LISP プログラムの作成を意図する．

```
conditinal terms:
((Y <- NIL) if (=X NIL))
((Y <- (MERGE (CAR X) (MSORT (CDR X)))) if (-=X NIL))
```

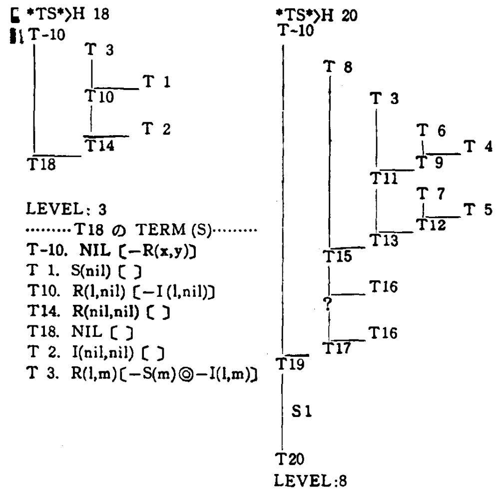


図-3 MSORT の条件項表現とその反証図

論理節1は nil と $S(x)$ の関係，論理節2は nil と $I(x,y)$ の関係，論理節6は merge と $S(x)$ の関係，論理節7は cons, merge, および $I(x,y)$ の関係，論理節8は $x=\mathrm{nil} \vee x=\mathrm{cons}\ (\mathrm{car}(x), \mathrm{cdr}(x))$ と $R(x,y)$ との関係を表わしている．

節形式の仕様書を，反証手続きを基にして構成された変形手続きにより，汎関数表現に基づく中間形式，すなわち条件項表現 (conditional terms) へ変形する．**図-3** はこの例の変形が2つの (条件付) 反証に基づいていることを示し，さらにそれらの反証図を参考資料として示している．右側の反証図の TERMS (論理節の計算機内部表現) は省略してある．

一般に与えられた論理節の集合が矛盾しているならば，レゾリューションに基づく反証法によって反証図が構成できる．論理節の集合からの反証図の構成は，セミアルゴリズムしか存在しないことが知られている．しかし，これは選択公理の適用と新しい関数 (汎関数を指示する) 記号を導入し，元の論理節の集合，すなわち，公理体系の保存型拡張 (conservative extension) を行うことにより，アルゴリズムが存在する領域を十分拡げうる．

ソフトウェア工学では，プログラムが存在する問題しか扱わないので，誤りのない仕様書に対しては，セミアルゴリズムは必ず停止するばかりでなく，実用上の大部分の仕様書はアルゴリズムが存在する上記の領域に入ると思われる．

また，一般に，レゾリューションに基づく自動証明

```
(MSORT
  [LAMBDA (X)
    (PROG (Y)
          (if (EQ X NIL)
          then (Y—NIL)
               (RETURN Y))
          (if (NOT (EQ X NIL))
          then [Y—(MERGE (CAR X)
                              (MSORT (CDR X]
               (RETURN Y])
congratulations!!
17862 conses              —MSORT( ((2 4 6) (1 3 5 7) (8]
34.664 seconds             (1 2 3 4 5 6 7 8)
249.264 seconds, real time
```

図-4 合成された MSORT プログラムとその実行例

系のセミアルゴリズムは，その効率が問題視されていたが，大幅に改善された[2].

ここでは，合成したプログラムを直ちに実行できるという利点から，プログラム言語として LISP を採用している．中間表現から LISP プログラムへの変形は簡単な問題であり，アルゴリズムが存在する．

図-4 では，合成された MSORT の簡単な実行例を与えている．

5. ソフトウェア工学の基礎

5.1 プログラミングの形式化

ここでは，論理的プログラム合成において行われる変形手続きの意味するものを考察し，プログラミングの本質，およびその複雑さ (complexity) について論ずる．

仕様書は，つぎのものを含む．

i．ある理論の公理体系

ii．写像または関数の暗示的 (implicit) な定義：入出力の指定と入出力関係による．

iii．基本演算および基本述語の指定（構成されるプログラムの実行可能性を保証するため）

iv．構成されるプログラムの形式（回帰形または反復形）

プログラミングとは，iii，ivの条件を満たすような，その公理体系で定まる数学的構造をもつ世界での，関数の明示的 (explicit) な定義，およびその表現を求めることである．

プログラミングの本質を理解しやすくするために，iii，ivは無視し，iiは，その関係と入出力の指定から，入力変数は全称量化子で，出力変数は存在量化子で束縛した，閉じた論理節を考え，その論理節が i の公理体系における定理であるかどうかという簡単な証明問題を考えることにすれば，プログラミングが証明手続きを含むことは自明である．さらに，公理体系が同じで，入出力関係および入出力の指定が異なる仕様書を考慮すれば，プログラミングは，その公理体系の無矛盾性の証明（その公理体系における定理のすべてを真とする解釈，モデルが存在することの証明）を行うことをも含むことが理解できる．

その上に，すべての仕様書が一階論理の言語で書けると仮定しても（大部分が一階の範囲で書けることは判っているが），公理体系は，一般に仕様書によって与えられるのだから，任意の一階理論の公理体系の無矛盾性の証明を行うことを含むことになる．

一般に公理系の無矛盾性証明は，モデルの構成を暗示的または明示的に含む．

ここで，問題を少し戻して（複雑にして），iii，ivを無視するだけにする．単なる公理系の無矛盾性の証明では，任意のモデルの構成が許されるが，与えられた公理系において暗示的に定められた関数を明示的に求める手続きは，Herbrand theory に基づくようなその言語内構築物によるモデルの明示的な構成が要求される．

さらに，プログラムの表現は有限表現でなければならないので，十分な表現能力を持つために，必要なだけ新しい記号を導入することにより，もとの公理体系からの保存型拡張を行わなければならない．

iiiの条件は，モデル構成における言語内構築物に関する制限を与える．簡単化した問題では，これは証明図に関する制限，すなわちある条件を充たす証明の構成を要求する．

ivの条件は，表現力を増すための公理体系の保存型拡張の仕方に対する制限を与える．

上記の事柄を実行する手続きは，現在幾つかの問題が存在し，一部は会話形となるが，幸いなことに，ほとんど機械的作業である．

完全な仕様書から，正しいプログラムを作成することが，十分理論的に行えるばかりでなく，ほとんど機械的作業からなるとすれば，ソフトウェア開発の最も大切な部分は，仕様書の作成である．

仕様書の作成の中心は，公理体系の作成と入出力関係の把握からなる．

5.2 ソフトウェア開発の本質

以上で明らかなように，ソフトウェアの開発とは，ひとつの理論として把握されるような，ある構造（数学的構造）をもつ現実世界（モデル）の公理化を，その要求，設計工程における仕様書作成として行うこと

Peano Arithmetic

(nonlogical) Axioms

1 $S(x) \neq 0$

2 $S(x)=S(y) \rightarrow x=y$

3 $x+0=x$

4 $x+S(y)=S(x+y)$

5 $x \cdot 0=0$

6 $x \cdot S(y)=(x \cdot y)+x$

7 $(x<0)$

8 $x<S(y) \leftrightarrow x<y \vee x=y$

Induction axiom schema

$$A[0] \wedge \forall x(A[x] \rightarrow A[S(x)]) \rightarrow \forall x A[x]$$

図-5 ペアノの算術公理系

を含み，かつ，そこにおいてある命題の証明を行うことを含む．この証明すべき命題（定理）は，システム要求により決るものであるから，ソフトウェア開発の一般的手続きは，任意の公理体系における無矛盾性の証明を行えるものでなければならない．

ある世界を十分に公理化するという問題（公理系の作成）は，数学においても非常に難しいものであり，自然数の算術の公理系の場合は，その主要な貢献者の名にちなんで，Peano の算術公理系と呼ばれる．

また，ある公理系の無矛盾性の証明を行うことは，数学基礎論においても非常に難しいものであり，自然数の算術の公理系の無矛盾性証明は，Gentzen, Gödel, Kleene 等の超一流の学者によってなされた．しかも，ソフトウェア開発における無矛盾性の証明は，その証明から手続き（汎関数表現）が構成されるような，Gödel 流の証明でなければならず，与えられる公理系が定まっていないため，Gödel 流の証明よりもっと一般的な手法でなければならない．また，証明から構成される手続きは，人間が理解すればよいものでなく，計算機で実行可能なものでなければならないという，より厳しい制約さえある．

上記の事柄を考慮するならば，ソフトウェア開発に貢献するような理論が，より専門的であり，素人向きでないものであることは自明である．

6. ソフトウェア・ツールのありかた

計算機械の出現の当初から，"機械にできる仕事は機械にまかせ，人間は人間しかできない仕事をするべきである．" という Pascal の思想が，計算機の利用の基本思想の1つであった．さらに，現在のような高速高性能な電子計算機の出現によって，共同作業システムとしての人間機械系 (man-machine system) というより高度な形態は，機械と人間の相異なる特性に従う，機械のすべき仕事と人間のすべき仕事の区別に基礎をおく．

機械のすべき仕事とは，(1)人間にはできない仕事：高速処理，大量処理，高信頼性処理，高頻度割込み処理，多重並行処理，多種類の処理，大規模管理など，(2)人間と機械との両方にできる仕事：このうち機械の方がより得意な仕事相手の処理（思考）速度にあわせなければならない処理などである．

機械処理の特徴は，形式的体系の考察から明らかなように，データ（異なる表現）は増加することがあっても，情報（新しい意味）は増加しないということである．したがって，人間のなすべき仕事は必要があるかぎり情報を増加させる処理である．これは機械にはできない仕事である．

以上のような観点から，ソフトウェア・ツールの役割りはつぎの2つに大別される．(1)完全に理論的（形式的）に解明された作業のような機械のなすべき仕事を人間にさせないこと (2)人間のすべき仕事に対する機械的補助作業を行うこと，すなわち，人間の機械系への情報を増加させる作業の補助，および人間自身の知識の増加と概念の整頓に必要な作業の補助，これには効率を高めるようなアクティブな補助と人間の誤りを検出するようなパッシブな補助とが考えられる．

ソフトウェア・ツールの適切なありかたは，(1)その適用業務の本質 (2)そこにおける理論化，形式化（機械化）の範囲（適用限界） (3)形式化不能な部分とそこに適用される人間工学的考察 などに基づき (1)ソフトウェア・ツールの種類 (2)各ツールの役割り (3)その役割りに適した理論的または人間工学的方法の採用が定められていなければならず，かつそれ自身高品質でなければならない．

ここで，ソフトウェア開発のためのソフトウェア・ツールについて，プログラミングの形式化に基づいて，その現状の評価および将来像について簡単に述べる．考察の基礎になる開発工程の変化をつぎに示す．参考のために検証系 (verification system) に基づく開発工程も共に示す．(**図-6**)

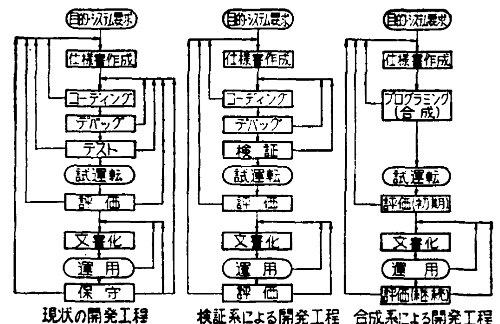


図-6 プログラミングの形式化と開発工程の変化

図-6 における仕様作成工程とは，ソフトウェア要求，基本設計および詳細設計の工程を意味する．合成系によるプログラミング工程と検証系による検証工程以外の各工程が，新しいエラーの発生源である．

ソフトウェア・ツール全般について

ソフトウェア・ツールの開発，評価および将来像は，それが属する個々の工程の考察によってのみならず，開発工程全体の考察によって考えられねばならない．例えば，現状の開発工程では保守工程に要する費用が非常に大きな割合をしめている．また，大規模ソフトウェアを開発するほど，仕様書作成とテストの割合が大きくなる．したがって，現状ではテスト用ツール，保守用ツールは非常に重要である．しかし，合成系による開発工程では全く事情が異なる．

既存のソフトウェア・ツールは現状の開発工程における評価のみならず，ソフトウェア開発の本質にもとづく新しい開発工程にどの程度適応・貢献できるかによっても評価されなければならない．

ソフトウェア・ツールはその役割りによって理論的ツールおよび人間工学的ツールに区別され，プログラミングや文書化の一部の作成工程では，記号論理学や数学基礎論に基づく検証系や合成系のような信頼性を保証する理論的ツールが増えると思われる．

個々のソフトウェア・ツールについて

要求定義支援ツールや設計用ツールなどの仕様書作成ツールはますます重要性を増す．現実世界の把握やアルゴリズムの理解を助ける自動シミュレーション・ツールのような人間工学的アクティブ・ツールと仕様記述言語のツールが必要である．要求・設計の矛盾や不完全さを分析・チェックする自動要求定義ツールや自動設計ツールのようなパッシブ・ツールの機能は合成系が行う．

仕様記述言語の設計基準は，(1)プログラミングの形式化が行えることという理論的基準 (2)使い易く，理解しやすいことなどの人間工学的評価基準からなる．理論的基準を充たす言語を核言語とし，人間工学的評価基準を充たすように順次改良してゆくのが最も合理的である．逆方向の改良は大変難しい．

製作用ツールは合成系のような理論的ツールに移り変わる．デバッグ・エイド，テスト用ツールおよび保守用修理ツールは，仕様書に対するプログラムの製造エラーがなくなり，仕様書と独立にプログラムを変更することが許されなくなるので不要になる．合成系ではプログラム言語の指定ができるのでアルゴリズムの複雑さを解析するための人間向きのアルゴリズム記述言語と実行用の計算機向き言語に分かれる．

解析と評価のツールでは性能（効率），操作性に関するツールの重要性が再認識される．

保守用更新作業ツールは再合成過程の効率化として合成系に吸収される．

7. おわりに

“間に合う”，“役に立つ”という評価基準から“正しさ”という判定基準へ移ったとき，必然的にソフトウェアの開発は，対象の公理化という“仕様技術”と公理系の無矛盾性の証明を内包する“プログラミング”という非常に高度で難しい問題になる．それと同時に，工芸から科学の領域へ変質していく．工芸としてのソフトウェア開発は，高品質大規模ソフトウェアへの道は閉ざされていたが，全くの素人にも取り付きやすく，かつそれなりに非常に高度に発達しており，名工・達人の存在を許す，ある意味で楽しい世界であった．工学としてのソフトウェア開発は，今やっとその入口にたどり着いたばかりであり，工芸家達の高度な技芸に比すべき何ものをも持たない．この非常に類似した，しかし本質的には全く異なる2つのソフトウェア開発の存在が，数々のソフトウェア開発に関する全く相反する主張を許し，いたずらに混乱をまねいた原因と思われる．

ソフトウェア・ツールをはじめ，ソフトウェア工学の各分野は，現状では工学的ソフトウェア開発が実用化の域に達していないので，工芸的ソフトウェア開発の工程に対症療法的に寄与しなければならないが，科学的考察に基づいて，その本来あるべき姿，進むべき方向を決定すべきであると思う．

参考文献

1) 國井利泰監修：ソフトウェア工学要求仕様技術，bit, 8（増刊），(1978).
2) 謝 章文：論理的プログラム合成と構造的反証原理，情報処理，ソフトウェア工学 9-7, (1979).
3) 謝 章文：Foundations of Logical Program Synthesis, 情報処理，ソフトウェア工学 4-1, (1977).
4) Dijkstra, E. W.: The Humble Programmer, CACM, Vol. 15, No. 10 (1972).
5) Kernighan, B. W. and Plauger, P. J.: Software Tools, Addison-Wesley (1976).
6) Boehm, B. W.: Software Engineering, IEEE Tran. On Computers, (1976).

（昭和54年3月22日受付）