Formal Methods 批判

— SEA SIGFM 発足記念シンポジウム —

2001 年 3 月 3 日

（ミレニアムの桃の節句）

こんどう　ひでたか
金藤 栄孝

（株）日立製作所　システム開発研究所

Formal Methods の最大の功績

（他の工業分野と比べて）現在のソフトウェア開発の現状が，如何にプロフェッショナルでないか，という事をアカラサマにした事.

即ち，現在のソフトウェア産業が，如何に巨大な潜在失業者の受け皿となっており，その結果として，ソフトウェア開発の従事者の中に，如何に多くの「ソフトウェアに要求される論理的思考」に適さない人間が紛れ込んでいるか，という事を露見させた事.

ソフトウェアの正しさの保証

ソフトウェアの草創期からの課題であり，既に創始者達が関心を持っていた.

- von Neumann (with Goldstein, 1947)

  フローダイアグラム中の各ポイントに中間表明を記述し，処理を実現するコードを作る助けとする. (Dijkstra 的なプログラム導出計算の流れの源流？)

- Turing (1949)

  プログラムの正しさを「チェックする」という明確な問題意識の下での同様のアプローチ. (Floyd 的プログラム検証の流れの源流？)

より modern なアプローチは,

- Floyd (1967): 帰納的表明付け法
- Hoare (1969): Hoare 論理

これらのプログラム検証への modern なアプローチと共に，様々なプログラミング言語の形式的定義法 (*e.g.* ULD/VDL) の研究の流れが加わって，現在の Formal Methods の流れの源を構成している.

何故，Formal Methods の普及が進まないのか？

(1) 社会的要因

「顧客は要求を知らない」(Jackson)

⇓

システム全体の仕様を最初から形式的な言語で書ける詳細さと精密さで書く，という formal methods の哲学は画餅に過ぎない.

注意

日本の社会では，欧米社会と比べても，一般に**「契約」の観念に乏しい**ので，より一層，上記の問題（顧客が要求仕様を最初に明確に示さない事）が根深い問題として存在する，と考えるべき.

(2) Formal Methods 自体の要因

(a) ペダンティック

少なからぬ formal methods の教科書には，**理論的背景を知らないと，その形式仕様言語を正しく使えないかの如く書いてある．**

これが如何にトンでもない事か，という事は，例えば，プログラミング言語の場合，「正しく理解して使うには Scott 理論を知らなければならない」と要求している，という状況を考えれば明らか.

(b) ツールへの過度の依存，或いは，**ツール至上主義**

Brooks, Jr. がイミジクも述べている通り，ツールで解決できるのは**偶有的困難**だけであって，ソフトウェア開発の真の障害となっている**本質的困難**（抽象的で複雑な対象に取り組む事）は解決できない.

しかも， formal methods の多くのツールは，所謂，検証系もしくは検証支援系と呼ばれるモノであるが，これらが対象とする問題が『論理式の証明／反証』という決定不能な問題であるので，本質的にユーザの介入が必要であり，コンパイラの様な完全自動ツールとはなれない.

(c) 方法論の欠如

単に，「等式の集合で規定される代数として定義する」とか「抽象状態機械として定義する」とか「木構造間の準同型として定義する」といった指針しか与えておらず，実際の問題に現れる様々な対象を，その仕様言語が要求する「構造」として如何にモデル化すれば良いのかの手順的指針が無い．

属人性の問題を何ら解消しておらず，**出来る人だけが出来る**の域を越えていない．

ソフトウェア産業の最大の問題は，「複雑な問題を分解して要求された仕様を満たすソフトウェアを作る」という**極めて創造的な活動**を，通常の創造性しか持たない一般技術者が行なわねばならない，という点にあり，その困難を乗り越えるには，**方法論による手助け**が不可欠である．

(d) 非数学的性格（構文的思考の要求）

Formal methods は数理論理学的ではあるが，数学的ではない！（Formal methods は，丁度，数理論理学がメタレベルに立って，数学をオブジェクトレベル化＝構文化して，構文による記述の対象としてから見ているのと同様，ソフトウェア開発をメタレベルから捉えている）．

一方，現場のソフトウェア技術者にとってのソフトウェア開発とは，（数学者にとっての数学と同様に），正にメタレベルでの活動そのものなのであり，彼らにとって，ソフトウェアは意味に基づく思考の対象なのである

Thesis（Formal Methodsの発想）

プログラムは或る論理的な体系で完全に規定されるモノ（即ち，再帰的関数の形式言語による表現）である.

（∴）

ソフトウェア開発は完全に形式的な体系の枠内で進められるべきである.

Antithesis（Knuth の文芸的プログラミング）

> プログラムは単なる計算機への指令ではなく，何よりも先ず人間によって読まれるモノである.

（∴）

プログラムを作るという行為は，文学作品を執筆する事に最も近い.

Synthesis（私の立場）

プログラムは本質的に何らかの論理体系で裏付けられ，且つ同時に，少なくとも工業的に作られるプログラムの場合，他人によっても読まれねばならない（保守や改良の時など）．

（∴）

産業的なソフトウェア開発は数学の本を書く行為に最も近い．即ち，

- 論理的に厳密でなければならない．
- 人間にとって理解容易な構造（ソフトウェア固有の経験的構造）で構成されなければならない．

以上の二つの要件を同時に満たさねばならない

Formal Methods は，最初の要件しか考慮していない.

- FM の解説で必ずと言っても過言でないほど出て来るスタックとかキューとかは「構造」と呼ぶには余りに小さすぎる．寧ろ，基本的素材のレベルに近い.

Formal Methods の非数学的な諸点

(1) 極端な還元主義

数学者の数理論理学への反感の原因

(2) 形式的証明

FM の研究者自身が（論文を書く時に）やらない／やれない事を，他人＝ソフトウェア技術者に要求している.

産業的ソフトウェア開発の目的は充分な品質のソフトウェア自体であり，正しさの証明を完璧に正しく書き下す事ではない.

(3) 閉じた言語

FM 利用者の思考を syntactical な形に強く束縛し，意味からの発想を妨げる.

問題の仕様を捉え理解するの為の良い着想が湧いても，その発想を支える論理体系（例えば，様相論理）が，閉じた形式言語である仕様記述言語の提供する論理体系（例えば，等式論理）の枠内に納まらねば，その閉じた言語を使おうとする限り，着想は捨てられねばならない.

思考の手段である筈の言語が何時の間にか目的と化す

(4) 実行可能性

- （要求分析の手段としての）プロトタイピングと仕様の記述とは全く異なる.

- 仕様が実行可能であれば，プログラムとの間の差は実行速度だけになり，仕様とプログラムとの間の本質的な違いが消え失せてしまう.

- 思考を突き詰める動機の阻害となる.

  記述を動かす事は楽しい.

  一方，正しい記述を得る為に考えを突き詰める事は苦しい.

  人間は楽な方に流され勝ち.

しかし，徹底的に考え抜かねば正しい仕様は得られない.

実行が思考の代替となるなら，バッチ → TSS → W/S → PC と**よりプログラムの実行が容易となる様に開発スタイルが変化して来た中でソフトウェアの品質は上がって来た筈.**

そもそも，効率良く実行できるプログラムそれ自体が，仕様の代わりになれる筈.

最後の結論が間違っている，という事は，**実行は思考の代替となり得ない**事を示している！

実行可能である事の危険性

実行する事は考える事より楽しい

（∴）

記述が実行可能だと，ソフトウェア技術者は**記述の中身が正しいかを考えるよりも動く様にする事に関心の重点を置いてしまい勝ち.**

その結果，往々にして，間違った或いは役に立たない内容の実行可能な記述を書いてしまう.

実行可能である事の危険性の実例

(1) 或るコンパイラ開発の折，ある部分の処理が極めて複雑であった.

(2) そこで，その部分の実現を考える為に，その部分のプロトタイプを関数的な言語で作り，最終的な実現の方式を検討しようとした.

(3) プロトタイピング・チームは，**ともかくプロトタイプを動く様にする事に関心が向いてしまった.**

(4) その結果，作られたプロトタイプは，難しい点を全て避けてしまった何の役にも立たない代物となった.