自動プログラミング
—ソフトウェア工学とAIのはざまで—

玉井哲雄
筑波大学 経営システム科学

1992 年 10 月 27 日

1 はじめに

自動プログラミングという言葉は，人を惹きつける魅力をもつ一方で，錬金術的な胡散臭さを感じさせる用語でもある．

昔は Fortran のコンパイラを自動プログラミングシステムと呼んだとか，いやさらに遡ってアセンブラをそう呼んだという話がよく聞かれるように，言葉としての自動プログラミングの歴史は古い．AI 分野でいう自動プログラミングに限定してみても，かなり早くから，主流とはいわぬまでも一定の部分領域をなしてきた．実際，AI 百科事典の Handbook of Artificial Intelligence [9] でも，その第 1 版から自動プログラミングは独立した一つの章（第 X 章）を与えられている[1]．

自動プログラミングが AI の重要な対象の一つとなりうることは，プログラム作成，より総合的な言い方をすればソフトウェア開発が，高度な知的作業であることを考えれば，納得がいこう．人工的なモデルによる知能の解明という AI の大きな目標に，ソフトウェア開発という知識集約性の高い作業が，研究対象として適合するからである．しかも，AI の研究や実用化が，ソフトウェアを作ることで可能となることから，AI 研究者や技術者にとってソフトウェアが身近なものであるという事情もある．

もちろん，実用性という意味での期待も高い．ソフトウェアが産業や社会において占める役割は，ますます大きくなっているが，その生産供給体制が需要に追いついていないという状況は，しばしば指摘される．傍証としてよく挙げられるのが，ソフトウェア産業の人材難であり，社会全体におけるソフトウェア技術者不足である [4]．ソフトウェア生産が自動化され，それによって生産性が向上し，品質も確保されるとしたら，これほど歓迎されることはない．

実際，情報処理開発協会が実施している過去の AI 白書の調査でも，自動プログラミングシステムの導入状況について毎年調べられているが，調査時点での導入数は少ないものの，今後への期待が高いという結果が，常に出ている．たとえば，1988 年の調査では [1]，自動プログラミングシステムを導入していると回答した企業は，262 事業体中わずか 6 社であるが，今後導入を予定するとしたものが 130 社 (50%) ある．これをとらえて白書は,「ニーズとシーズの間に大きなギャップがあることを示している」との判断を下している．同じ調査で将来普及の見通しを聞いているが，本格使用開始時期を 3 年超，5 年以内と予測する回答がもっとも多く (22%)，さらに 5 年超，10 年以内とするものがこれに次ぐ (19%)．この見通しは，エキスパートシステムはもちろん，自動翻訳システムの本格使用開始の予測と比べても，遅いと見ていることになり，今後の導入を 50%が予定しているのと比較すると，興味深い結果である．

1990 年の調査では，自動プログラミングシステムの導入数は表 1 のように増えてはいるが，これらの多くはいわゆる CASE ツールや第 4 世代言語に属するものであり，AI でいう自動プログラミングとは多少性格の違うものであることも分かっている [2].

[1] その後，新たに出版された第 IV 巻 [10] では，知識ベースソフトウェア工学 (knowledge-based software engineering) という章が加えられている

表 1: 自動プログラミングシステムの導入状況

調査年	導入件数	回答数	導入比率 (%)
1987	6	303	2.0
1988	6	262	2.3
1990	56	405	13.8

このように自動プログラミングに対する産業からの期待の高さと，実際に実現されている技術レベルとの間には，大きな乖離がある．また研究として行なわれている試作システムと，現場で使われているツールとの間にも，その機能だけでなく，基本的な考え方や方法のレベルで，相当の違いがある．

この間を埋めるのが，ソフトウェア工学の役割であろう．以下では AI とソフトウェア工学の接点に立って，自動プログラミングの技術状況を整理する．

2 自動プログラミングの多様な視点

自動プログラミングには様々な研究開発例や実践例があり，そのアプローチも多様である．一つの試みとしてこれを方法論の観点から，形式的アプローチ，知識的アプローチ，実践的アプローチの 3 種類に分けてみる（図 1 参照）[1].

形式的アプローチ 論理や数学的な形式性を備えた方法に基づき，自動的にプログラムを生成しようとするもの．ある意味で AI の本流に属し，比較的歴史も長い．その特徴は，次のようである．

- 形式的仕様を前提とする．仕様記述方法の研究自身も，含まれる．
- 理論的に明確な基盤を持ち，小規模プログラムに対しては自動合成能力が実証されている．
- 研究的な色彩が強く，実用的なレベルにはまだ達していない．

主な方法としては，論理に基づく演繹的な手法，プログラム変換法，および帰納的な手法がある．

知識的アプローチ ソフトウェア設計や開発上の知識を知識ベース化して，ソフトウェア開発用のエキスパートシステムを作ろうとするもの．知識工学の隆盛とともに，研究開発例が増えてきた．その特徴は，次のようである．

- 他分野で成功例のある知識ベースシステムの手法を，ソフトウェアの分析・設計および実現のフェーズに適用するものである．
- 形式的アプローチに比べると，プログラム生成よりも分析・設計作業の支援を目指すものが多い．またそれに関連し，完全な自動化よりも人間の補助的役割をねらっている．
- ソフトウェア開発過程における生成物（仕様，プログラム，テストケース，データ，メモなど）をオブジェクトベースとしてしまい，それらに対するプロセスをルールベースやその他の知識表現言語で記述し利用するという，知的ソフトウェア開発環境という概念を中核とするものが，典型的である．
- まだ概念的な段階の研究例が多く，その概念を確かめるためのプロトタイプが作られているというのが，実態である．

実践的アプローチ 主として事務処理分野で，限定した問題領域におけるプログラムの自動生成や部品合成によるソフトウェア作成という方法をとるもの．必ずしも AI 的なアプローチとはいえないが，第 4 世代言語や CASE などの形で実際に使われているものがかなりある．その特徴は，

- 開発プロセスを支援するタイプのものと，プログラムの自動生成に焦点をおくものとがある．
- 問題領域としては事務処理分野を対象にするものが多いが，なかにはシステムソフトウェアやリアルタイムソフトウェアを対象にするものもある．

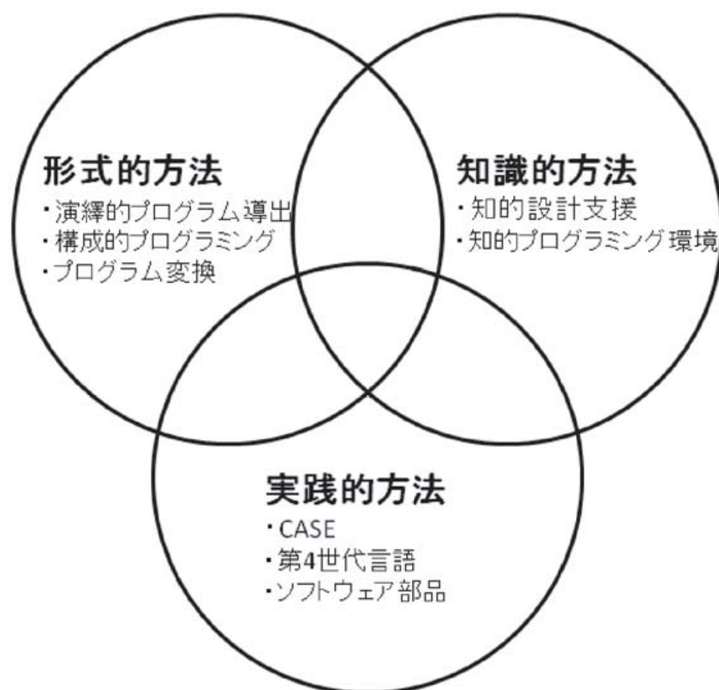

図 1: 方法論による自動プログラミングの類別

以下では多少視点を変えて，次の 4 つの目標に分けて自動プログラミングを考えてみることにする．

1. 仕様や例からのプログラムの自動生成
2. 形式的 (formal) なソフトウェア開発技法
3. ソフトウェア開発プロセスのモデル化と自動化
4. ソフトウェア設計・プログラミングの省力化

このうち (1) が，通常使われる狭い意味での自動プログラミングであり，上記の形式的方法のすべてと知識的方法の一部や実践的方法のごく一部を含む．(2) 以降はソフトウェア工学的な見地から，より実践的な方向を目指して「自動化」の考え方に変化を与えたものと見ることができる．

たとえば (2) は，(1) が前提とする形式的な仕様や形式的な生成技法を，人間が実際的なソフトウェアを開発する際に使える技術として体系化し利用しようとするものである．また (3) は，開発プロセス自身をプログラム化可能な対象として捉えようとする主張を含むが，それはメタレベルの自動化という意味を持ちながら，ソフトウェア開発プロセスのモデル化を通した分析評価という点で実際的な意義も高い．方法として，知識的アプローチに分類できるものが比較的多い．(4) は自動化やコンピュータによる支援がしやすい部分に限定して，実践的なツールや言語を作り利用しようとする立場であり，上にあげた実践的アプローチにほぼ該当する．

3 仕様や例からのプログラムの自動生成

形式的な仕様記述を前提とし，その仕様からプログラムを自動生成するものと，プログラムの入出力例や実行例からそれに合う動作をするプログラムを生成するものとがある．前者を演繹的な自動プログラミング，後者を帰納的な自動プログラミングと呼ぶこともできる．

仕様から演繹的にプログラムを生成するには，なんらかの論理体系の上で，プログラムの入力条件を満たす入力に対し，出力条件を満たす出力が存在するという存在定理を演繹的に証明し，その証明からプログラムを導出するという，かなり古くから試みられている方法が，もっとも一般性の高いものとして認められている．この旧来の考え方が，近年改めて注目されている．これは，そのような目的に向いた構成的論理の研究が進み，論理的な命題をプログラムにおける型と見なし，その証明を型の要素と見なすという対応が自然につけられることが分かってきたこと，などによる．とくにわが国では，このようなアプローチに構成的プログラミングの命名がなされ，活発な研究の動きがある [8, 3]．

一方，仕様とプログラムをまったく別個のものと考えず，より抽象度の高いものが仕様であり，それをより具体的で，実行が可能なもの，あるいはより実行効率の良いものに変換するという技法が，プログラム変換の考え方である．この方法では，多くの場合，仕様とプログラムは同じ記法で書かれる．純粋に演繹的な方法よりもプログラム変換は実際的な面があり，試作システムも多い．

その一例として，米国の Kestrel Institute で開発されている半自動プログラム開発システム KIDS を取り上げよう．Kestrel Institute は，長年にわたり自動プログラミングの研究開発に精力を傾けてきた C. Green が設立した研究所である．

Green が手掛けた一連の研究開発の中で，プログラムを導出するための手法としては，与えられた仕様に対してアルゴリズムやデータ構造を設計するための規則を，知識ベースとして整備するという方法論が，一貫してとられてきた．そのもっとも最近の成果が，D. R. Smith により開発された KIDS(Kestrel Interactive Development System) である [16]．KIDS は Reasoning Systems が提供する商用の知識ベースプログラミング開発環境 Refine の上で開発された研究システムで，探索型のアルゴリズムを用いる効率的なプログラムの生成などに成功している．

その他のよく知られている研究プロジェクトには，ドイツのミュンヘン工科大学における CIP，米国マサチューセッツ工科大学における Programmer's Apprentice, 米国シュルンベルジェ研究所の D. Barstow による PHiNix などがある．このうちとくに息の長いのが Programmer's Apprentice で，比較的最近の成果としては，要求獲得を支援するためのツール Requirements Apprentice [15] や，プログラム作成時に知的な支援をするエディタ KBEmacs [20] がある．

なお，とくに日本におけるこれらの動向をサーベイしたものに [18] などがある．

4 形式的なソフトウェア開発技法

この数年，ヨーロッパを中心に，VDM, Z, RAISE などの形式的な仕様言語および開発技法の提案と，産業レベルのソフトウェア開発への適用努力が，めざましく続けられている．これらは厳密な仕様記述に重点がおかれており，開発されたソフトウェアの正当性の証明や自動的なプログラム生成は，主眼とされていないが，そこで用いられている開発技法は，自動プログラミングの技術をある意味でベースとしている．

VDM[12] は，その名が Vienna Development Method に由来するものであることすら，いまやほとんど明示されなくなったが，もともとはプログラミング言語の操作的意味論を与えるウィーン定義言語 (Vienna definition language) の流れを汲むものである．現在は操作的意味論という枠組からは完全に離れ，むしろ表示的意味論の影響を受けながら，述語論理表現に基づく仕様記述記法と，その上でのプログラム開発技法を総称したものとなっている．C. Jones を中心とするイギリスの流派と，D. Bjϕrner を中心とするデンマークの流派とがあり，両者で記法も多少異なっている．

Z[17] はイギリスのオックスフォード大学を中心に開発された，形式的な仕様記述言語である．VDM の影響を受け，それと類似する点が多いが，記法としてより簡潔になるような工夫をしている．実用化の例と

して，IBM の提供するデータ通信管理システム CICS の全体の仕様をこれで記述し，それにより保守性を高め，新しい版のエラー削減に画期的な成果を挙げたことなどが，よく引用される．

RAISE(Rigorous Approach to Industrial Software Engineering)[11] はヨーロッパの Esprit プロジェクトの 1 つとして開発が進められているもので，ヨーロッパで VDM を指導してきた D. Bjϕorner を中心にしており，やはり数学的な形式性を重視している．仕様記述言語として RSL を定義し，それを用いたソフトウェアの開発方法論を展開している．RSL は，宣言的な記述だけでなく，命令的な記述や並列動作の記述のための基本要素を備えているところが，VDM などと異なる．

その他，とくに通信プロトコルなどの並列動作システムの仕様記述用の言語に Lotos や Estelle があり，これらの仕様からプログラムを生成するシステムも作られている．一般に並列システムの仕様記述やシステム開発には誤りが入りやすいため，形式的な開発技法や前節にあげたような導出技術が効果をあげる可能性が強く，期待が高い．

5　ソフトウェアの開発プロセスのモデル化・自動化

1987 年に L. Osterweil[14] がプロセスプログラミングを提唱して以来，ソフトウェアの開発プロセスを形式的な記述体系に基づいて記述し，モデル化するという試みが多様に行なわれている．その目的も，プロセスの実証的な分析から，評価，改善，標準プロセスの規定など多岐にわたるが，プロセスプログラミングを提唱する Osterweil 等は，記述したプロセスの実行という面を強調し，最終的には開発プロセスの自動化を目指している．

Osterweil 等は Arcadia というプロジェクトを推進し [19]，プロセス記述用の言語とその記述および実行支援システムの開発を行なっている．また，要求分析，設計，テストなど様々なフェーズにおけるプロセスの記述も実際に行なっている．

プロセスという概念を核とするソフトウェア開発環境の開発の試みは，他にも多くの例がある．たとえばコロンビア大学の G. Kaiser を中心に研究開発が進められている Marvel というシステム [13] では，ルールベースによるプロセス記述手段とオブジェクトベースによるソフトウェアの生成物管理とを組み合わせた開発環境を提供している．

6　ソフトウェアの設計・プログラミングの省力化

CASE(Computer Aided Software Engineering) やいわゆる第 4 世代言語と呼ばれるツールなどが目指すものは，プログラム開発の部分的な省力化や，実際的な支援である．その中で知識ベースの方法を利用しようとする例や，プログラム合成技術を部分的に応用しようとする例もあるが，総じて AI の直接的な影響は少ない．

とくに事務処理の分野の自動プログラミング動向をサーベイしたものに，少し古いが [6, 5] がある．なお，これらのサーベイを載せた [7] には，他に制御分野，通信分野に特化した自動プログラミングの動向解説などもあり，現在でも参照する意味がある．

7　リエンジニアリングへの応用

最近の傾向として，ソフトウェアの再構成や再利用技術に実務的な関心が高まっている．この分野で，自動プログラミングから派生した技術を産業界に適用している事例として，KIDS を紹介したところで挙げた Refine を利用しているものがいくつかある．たとえば，Fortran, Ada, C のプログラムのコード分析ツール，種々のプログラミング言語間の変換系，などが Refine の上で作られている．この Refine が，当初トップダウンにソフトウェアを開発する，まさに自動プログラミングの正統的なツールとして使われることを目指しながら，リエンジニアリングに実用的な利用法を見い出すに至った経緯を見るのは，興味深い．

歴史的に見ると，Reasoning Systems 社の Refine が生み出される前身となったプロジェクトに Kestrel Institute で行なわれた CHI プロジェクトがあり，さらにその前身として PSI プロジェクトがあった．そのいずれも指揮したのは Cordell Green である．

PSI

C. Green が Standford 大学にいた時代，1970 年代の半ばに行なわれたプロジェクト．PSI は自然言語による仕様記述からプログラムを自動的に合成するという，壮大な目標をもったプロジェクトで，いくつかのサブシステムが試作された．このプロジェクトから，David Barstow(現 Schlumberger-Doll 研究所), Elaine Kant(現 Carnegie-Mellon 大学) などが育った．

CHI

C. Green は 1978 年に Systems Control Technology という組織を作り，1981 年にそれを改組して Kestrel Institute とした．この両研究機関を拠点として 1978-1984 に行なわれたプロジェクトが CHI である．なお，Kestrel Institute は現在でも存続している．

CHI の成果の中心は，DKB という開発対象プログラムに関する知識ベースの仕組みと，広範囲言語 (wide-spectrum language)V である．V は仕様記述にも使えるし，手続き的な記述も書ける．

Reasoning Systems

C. Green はさらに 1984 年に，Reasoning Systems という会社を作った．ねらいは，CHI で開発された技術を，産業界に適用しようというものである．そのためのツールとして Refine が作られた．

Refine の当初の目的は，仕様記述とプログラム開発の自動化にあった．ただ，ツールとして販売するというより，顧客企業の生産性向上などを目標としたコンサルティング業務を請け負い，その際に道具として Refine を利用するという形態だったようだ．

しかし，やはりそのようなアプローチは産業界になかなか受け入れられにくく，この数年，リエンジニアリングへの適用に方向転換して，少しずつ成功し始めているようである．

Refine の構成

Refine　同じ名前の言語 Refine で記述された仕様やプログラムをコンパイルする．また，それによって作られるオブジェクトベースに種々の操作を行なうことができる．

Dialect　言語の構文を記述すると，それをもとに構文解析系 (parser) が生成できる．C, Cobol, Ada などは，既にできあいのものが提供される．

Intervista　会話的なユーザインタフェースをつくるための道具を提供する．

この 3 つをまとめて，Refinery と呼ぶ．

またこの上に作られた各種言語用の CASE ツールがある．Refine/C, Refine/Cobol, Refine/Ada, Refine/Fortran など．これらはそれぞれの言語で書かれた原始プログラムを解析したり，またカスタマイズすることにより必要な形式に変換することを支援するツールで，まさにリエンジニアリング用の機能を持つものといえる．

参考文献

[1] ICOT-JIPDEC AI センター編： 人工知能の技術と利用— AI 白書，日本情報処理開発協会，1989.

[2] ICOT-JIPDEC AI センター編： 人工知能の技術と市場の動向に関する調査研究報告書，日本情報処理開発協会，1991.

[3] 小林聡： 構成的証明からのプログラムの抽出，コンピュータソフトウェア，Vol. 7, No. 4 (1990), pp. 3–18.

[4] 通商産業省編： 2000 年のソフトウェア人材，コンピュータ・エージ，1987.

[5] 古宮誠一，原田実： 部品合成による自動プログラミング，[7] pp. 1329–1345.

[6] 原田実： 事務処理分野における自動プログラミング，[7] pp. 1378–1397.

[7] 原田実，福永光一 編： 大特集：自動プログラミング，情報処理，Vol. 28, No. 10, 1987.

[8] 林晋，小林聡： 構成的プログラミングの基礎，遊星社，1991.

[9] Barr, A. and Feigenbaum, E. A. ed: *The Handbook of Artificial Intelligence II*, William Kaufmann, 1982. 邦訳，田中幸吉，淵一博 監訳： 人工知能ハンドブック II，共立出版，1983.

[10] Barr, A. and Feigenbaum, E. A. ed: *The Handbook of Artificial Intelligence IV*, Addison-Wesley, 1989.

[11] Brock, S. and George, C.: *RAISE Method Manual*, RAISE/CRI/DOC/3/V1, Computer Resources International, 1990.

[12] Jones, C. B.: *Systematic Software Development using VDM, 2nd ed.*, Prentice Hall, 1990.

[13] Kaiser, G. E., Feiler, P. H. and Popovich, S. S.: Intelligent Assistance for Software Development and Maintenace, *IEEE Software*, Vol. 5, No. 3 (1988), pp. 40–49.

[14] Osterweil, L.: Software Processes Are Software too, *Proc. 9th International Conference on Software Engineering*, IEEE, March 1987, pp. 2–13.

[15] Reubenstein, H. B. and Waters, R. C.: The Requirements Apprentice: Automatic Assistance for Requirements Acquisition, *IEEE Trans. Software Engineering*, Vol. 17, No. 3 (1991), pp. 226–240.

[16] Smith, R. S.: KIDS: A Semiautomatic Program Development System, *IEEE Trans. Software Engineering*, Vol. 16, No. 9 (1990), pp. 1024–1043.

[17] Spivey, J. M.: *The Z Notation — A Reference Manual*, Prentice Hall, 1989.

[18] Tamai, T.: Applying the Knowledge Engineering Approach to Software Engineering, in Matsumoto, Y. and Ohno, Y. ed.: *Japanese Perspectives in Software Engineering*, Addison-Wesley, London, 1989, pp. 207–227.

[19] Taylor, R. N., Belz, F. C., Clarke, L. A., Osterweil, L. J., Selby, R. W., Wileden, J. C., Wolf, A. and Young, M.: Foundations for the Arcadia Environment Architecture, *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM, November 1988, pp. 1–13.

[20] Waters, R. C.: The Programmer's Apprentice: A Session with KBEmacs, *IEEE Trans. Software Engineering*, Vol. 11, No. 11 (1985), pp. 1296–1320.