ソフトウェア工学から見たプログラム合成変換技術

筑波大学 経営システム科学
玉井 哲雄

1992年12月11日

概 要

プログラム変換技術や合成技術は，ソフトウェア工学という視点から見ても重要な基礎技術であるが，実践的な応用という意味では，それほど浸透しているとはいえない．本論文では，まずソフトウェア工学と合成変換技術との関係の経緯を振り返る．そして現在のソフトウェア工学のいくつかの話題を，合成変換技術との関連という視点から取り上げ，解説する．さらに，自動プログラミングの純粋な研究から産業界への適用という道程を，長い時間をかけて歩んできている C. Green 主導のプロジェクト (PSI⇒CHI⇒Refine) を事例として取り上げながら，合成変換技術の実用化の可能性を探る．

1 はじめに

1990年代もかなり進行した現在では，プログラム合成変換という研究分野とソフトウェア工学とは，ある意味で互い独立した関係にあり，両者は異質のものとまで極論しても，あながちにおかしいとは言えない．しかし，ソフトウェア工学が誕生し活気に溢れていた1970年代では，そうではなかった．当時は，Dijkstra や Hoare などがソフトウェア工学の中心的な指導者と見られていたし，彼らもソフトウェア工学という土俵の上で発言をし，研究成果を問うていた．

それで思い出すのが，1987年3月に米国カリフォルニア州モンタレイで行なわれた第9回の ICSE (International Conference on Software Engineering) での，Harlan Mills の発言である．最初の日の基調講演として，後でも話題として触れることになる，L. Osterweil による「ソフトウェアプロセスもまたソフトウェアである」[19] という話が終り，会場からの質問を受け付けたところ，最初に立ったのが Mills だった．その時の状況を，筆者が帰国後に書いた文章を引用して紹介しよう [34].

「彼の質問は，いきなり，この会場にチューリング賞の授賞者がいたら手を挙げてほしい，という唐突な問いから始まった．該当者がいないことを確認したうえで，Mills は ICSE からチューリング賞クラスの知的指導者がなぜか排除されてきているという．ICSE の当初は，Dijkstra, Hoare, Gries 等がいたのに，なんらかの社会的なプロセスによって，反知性 (anti-intellectual) の方向に来てしまったのではないか．もう彼らから学ぶことはないのか．という強烈なパンチを，あの Mills が浴びせたのである．会場にたちまち緊張が走る．

Osterweil が多少の動揺を見せながらも，さすがにこういう議論に慣れたアメリカ人らしく弁じたてた．しかし，Mills の批判は会議の運営に対して向けられたものだから，噛み合うはずもない．もともと Osterweil が答える必要はなかったのである．Mills も，これは主催者に言っているのだというようなことを，付け加えた．とにかくこのやりとりから，米国のソフトウェアの学界にも込みいった勢力争いがありそうなことが，想像された.」

この現象には，種々の解釈が可能である．

一つには Mills が言外に匂わせたような政治的な要素，つまり研究者の派閥争いが反映したものという面があろう．しかし，これはおそらく派生的な要因である．

やはり根本には，研究分野の専門化，細分化という一般傾向があるだろう．新しい分野が生まれる時には，その分野の核となる概念が，なるべく広い範囲を取り込みうるような普遍性と，一つの旗印の下に個別の多様な課題を引き寄せるような求心性とを持つことが必要であろう．そして一旦その分野が成立し，成熟過程に入っていくと，独立性の強い部分分野が分離していく．その時にやはり，残るものと分れるものとの間に，何らかの基準による区分線が引かれる．

ソフトウェア工学の場合，この区分は開発すべきソフトウェアの規模の大小ということで，象徴的に定められてきたように思われる．ソフトウェア工学は，ソフトウェアの生産性と信頼性の向上を目指すものであり，それらが問題となるのは対象とするソフトウェアの規模が大きい場合である，という論理である．そうなると関心は，一人の人間が作るのではなく，多人数で組織的に作るソフトウェア開発をどうするかという点に移り，設計・プログラミング技術よりも管理に重点が置かれる．キーワードを並べてみれば，ソフトウェア計測 (metrics)，品質管理，プロジェクト管理，構成管理，プロトタイピング，形式的査閲 (formal inspection)，構造化分析／設計，オブジェクト指向分析／設計，コンピュータ支援協調作業 (CSCW) などとなる．

この文脈では，“合成・変換” のような技術で扱えるプログラムは “玩具プログラム” であるとして，実用化にはほど遠いものという判断が下された．つまり “合成・変換” という技術分野（というのはかなり曖昧な定義だが，この研究会の参加者にはおそらくほぼ共通する認識があるだろうと仮定して）が，ソフトウェア工学から分離独立したという現象は，残った側からすれば役に立たないとして切り捨てたのだという解釈になるかも知れない．

ところが大規模ソフトウェアの開発技術を目指したはずのソフトウェア工学も，それに成功しているとはいい難い．むしろ目標に対してきわめて遅々とした歩みを歩んでいるに過ぎないというべきであろう．実際，ソフトウェア工学の成果に対してきわめて悲観的な評価を示した論文が，ソフトウェア工学の指導的な立場にあると目される人達によって書かれ，大きな衝撃を与えたことは記憶に新しい．一つは，D. Parnas による SDI (Strategic Defense Initiative, いわゆるスターウォーズ計画) 用のソフトウェア開発が技術上不可能であることを論じたもの [20]，もう一つは F. Brooks の狼男，つまりソフトウェア開発という怪物，を倒す “銀の弾丸” は見つからないというものである [4]．

このような悲観的な論調は現在でも続いていて，たとえば最近の Scientific American 誌に “ソフトウェアのリスク” と題する解説があるが [17]，ソフトウェアの信頼性が不確実であるという事態に対し，可能な対処法として挙げているのが次のようなものである．

1. ソフトウェアに対しては要求仕様を定義するのを避ける．たとえば米国連邦航空管理顧問会議の出している回状では，航空機の種々の構成要素の重大故障発生確率を飛行時間当たり 10^{-9}とすることを取り決めているが，この対象からソフトウェアは定量的なエラー評価が不可能だとして明示的に除外されているという．
2. システム内でソフトウェアの果たす役割をあまり重大なものでないようにする．
3. 現状のソフトウェアの信頼性の限界を認めて，システム全体の安全性のレベルが多少甘くなるのを許容する．

何とも寂しい話である．

改めて，ソフトウェア工学が小規模プログラムに役立つ技術を切り離してしまったことが，不幸だったのではないかと思う．もっとも Parnas にしても Brooks にしても，自動プログラミングや形式的な手法が問題解決の武器にならないかと検討し，結局は本当の救いにならないと判断している．確かにその通りであろう．しかし一方で，これらの技術から得られた知見が，もう少し “現場” に移植されてもいい．

大体ソフトウェア工学では，“大規模複雑な” ソフトウェアを開発しなければならないという大前提にはまったく疑問をさしはさまないで，常に話を展開してきた．しかし，そろそろ考え直すべき時が来たようだ．原始プログラムの行数にして数百万行とか数千万行というシステムが今や珍しくないといっても，このあたりが人間の扱いうる規模の限界であることは，ほぼ誰しも認めるところだろう．そこで，次のような点がもっと考えられていい．

1. あまり必要でないソフトウェアを作らないこと
2. とくに既存のシステムの作り直しに際して，冗長な機能，使われていない機能，を切り捨てるという “不要求” 獲得を意識的に行なうこと．

3. システムの “統合化” が本当に有効か事前によく検討すること．

それに，ソフトウェア開発のすべての問題が，規模の大きさにのみ起因するわけではない．例えば，小規模多品種のプログラムをどう開発するかも別の難しい問題である（筆者はこの課題の存在に，学習ソフトウェアの開発というテーマによって気づかされた）．

ダウンサイジングというキーワードと，ソフトウェア産業の史上初めての不況という経済状況は，ソフトウェアの大規模化という前提自体への反省を促す契機として，象徴的なように思われる．

前置きが長くなった．以下ではまず第 2 章でプログラムの合成変換技術と何らかの関連を持つような最近のソフトウェア工学における話題を取り上げ紹介する． 続いて第 3 章で，自動プログラミングの純粋な研究から産業界への適用という道程を，長い時間をかけて歩んできている C. Green 主導のプロジェクト (PSI⇒CHI⇒Refine) を事例として取り上げながら，合成変換技術の実用化について議論する．結局，第 2 章と 3 章との共通項を取り出してくると，リエンジニアリングとか再開発とかいう話題になる．最後に第 4 章で，合成変換技術を実用面から見た場合の展望を，身近な範囲で行なってみることにする．

2　合成変換技術との関連で見たソフトウェア工学の最近の話題

合成変換技術と何らかの関連を持つと思われるソフトウェア工学の最近の動きの中から，形式的なソフトウェア開発技法，ソフトウェア・プロセス・モデル，リエンジニアリングの 3 つを取り上げる．

2.1　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．これらは当然，合成変換技術との関連が深いが，とくに産業的な適用を図る立場からは，厳密な仕様記述に重点を置き，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，必ずしも主眼とされていないところに特徴がある [9].

なぜヨーロッパでこのような活動が活発なのかという理由については，もちろん数学や論理学の伝統の強みということもあるだろうが，やはり Esprit プロジェクトの影響が大きいのではないかと思われる．ICOT プロジェクトに刺激された Esprit では，その対象分野を ICOT より広くとり，LSI の開発技術やソフトウェア工学を含めた．また，開発の体制は EC 内の国を跨ることはもちろんだが，産業界と学界との連係を意図している．その結果，もともと大学から生まれた Z などを，産業界に適用する試みが推進されたようである．

VDM[13] 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[26] 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 が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話や [22]，浮動小数点演算の形式的仕様を与え，実際の演算素子開発に適用した例 [2] などが，よく引き合いに出される．前者は 2000 ページにわたる仕様を Z で書き，それに基づいて 50,000 行程度のプログラムを開発したという規模の大きさと，Z を用いた場

合と用いなかった場合との比較を通じて，信頼性は数倍向上し，コストは9%改善されたというようなデータを示したことで知られる．後者は，その成果を英国女王が表彰したことで有名になった．

RAISE(Rigorous Approach to Industrial Software Engineering)[3] 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. Bjϕrner 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，形式的な開発技法が効果をあげる可能性が強く，期待が高い．

2.2 ソフトウェアの開発プロセスのモデル化

エンジニアリングの分野でプロセスが重要であることは当然であり，実際化学工業を初めとする多くの生産分野では，プロセスの設計がエンジニアリングの中核を占めてきた．それに比べると，ソフトウェアの製造については，従来プロセスよりもプロダクトの方に研究開発の重点がおかれていた傾向がある．

1985 年の第 8 回 ICSE (International Conference on Software Engineering) で，委員長の M. Lehman がプロセスの重要性を強調し，全体テーマとして掲げたことが，ソフトウェア工学の歴史の中ではあえて特筆すべきことであるというのが，実態である．

1987 年に L. Osterweil[19] 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．

この動きを合成変換技術との比較で摸式的にいえば，プログラミングを自動化するプロセスを考える代わりに，プログラム開発のプロセスを自動化するということになろうか．つまり自動化の対象となるプロセスのメタレベルが，一つ上がっているという解釈である．もっともプロセスをプログラムとして記述できるのは，開発プロセス全体からいうとごく限られた部分でしかないという議論もある．また，プロセスを実行する主体の一つに人間があるが，人間の不確定な行動をなるべく確定的なものとして捉えていこうとする一部の機械主義的な立場には批判もある．

この Osterweil によるプロセスプログラミングの提唱以降，ソフトウェアプロセスの研究が活発化した．その研究の舞台の主なものとして，1984 年以来続いている国際ソフトウェアプロセスワークショップ (ISPW) がある．1993 年 3 月にはドイツで第 8 回の ISPW が開かれる．また，1991 年からは，ソフトウェアプロセス国際会議 (ICSP) も始まった．日本でも，国内のソフトウェアプロセスワークショップが毎年開かれ，すでに 4 回を数える．

これらの研究を分類すると，たとえば次のようになろう．

1. 実際のプロセス，とくに人間の行動が主要な要素となる開発過程を観察し，分析し，改善方法を探る [7, 28].
2. 規範とすべきプロセスモデルを提示する [11, 8].
   この範疇に入るものとしては，ソフトウェアプロセスの円熟度を評価し，プロセスの改善につなげるという CMU/SEI の W. Humphrey 等のモデル [11]（現在は Capability Maturity Model という名前になっている）が，とくに米国のソフトウェア産業界には大きなインパクトを与えている．
3. プロセスの形式的記述に適した形式システム，言語の研究および実際の記述実験を行なう．
   研究例には，階層的関数型言語を用いた HFSP（片山等，東工大 [15]），代数的仕様記述

を用いたPDL（井上等，阪大[12]），状態遷移モデルを拡張した市販のツールStatemateを用いた例（M. Kellner, SEI[16]）など，多くのものがある．

4. プロセスという概念を核とするソフトウェア開発環境を開発する．
   Osterweil等はArcadiaというプロジェクトを推進し[30]，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試している．
   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[14]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．
   他に，落水等によるVela [32]，Ambriora等によるOikos [1]，Huff等によるGrapple [10]，Schäfer等によるMerlin [21]，などがある．

なおソフトウェアプロセスモデルの最近のサーベイとして，Curtis等の論文[6]はよくまとまっている．

2.3 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．用語としては，再(re)エンジニアリングとか逆(reverse)エンジニアリングなどをよく耳にするようになった．この分野の用語整理と解説を兼ねたものに文献[5]がある．

このような関心が生じてきた原因としては，やはり保守の問題がますます重大になってきたことがあるだろう．CASEツールに対する期待が一時非常に高まったが，その唱い文句の一つは保守に効果を発揮するということであった．しかしそれは，新たに行なうシステム開発にCASEを用いると，そこで作られるリポジトリ（要求仕様，設計仕様，データモデル，プログラム，テストケースなど，開発で生じる生成物を，それらの構造や関係についての情報と共にしまっておく倉庫）が，保守で役に立ちますということで，すでに存在し運用されていて，原始プログラム以外には確実な情報がなくなっているソフトウェアに対しては，直接適用できるものではなかった．

リエンジニアリング用のツールとして世の中に出ているものの基本的な機能や，それを実現するための方法は，10年前とさほど変わっていないように見える．結局ほとんどのものは原始プログラムを対象として，その構造や要素間の種々の関係を分析し，場合によっては構造を改善するなどのプログラムの変換を行なう．そこで使われる手法の中心は，制御フローやデータフローの解析技法である．ただし，10年前と大きく違うのは，結果の見せ方である．グラフィックスを駆使し，複数の窓を使った表示を連動させ，マウスなどによる利用者の指示に反応する．

しかし，もう少し進んだ技術を背景とする例もある．それがもともと自動プログラミングという枠組でプログラム変換技術を産業界に適用しようとする努力を続けた後の，いわば方向転換の結果として出てきたところが面白い．具体的には，Refineというツールであるが，その経緯は次章で述べる．ただ，この種のツールは，上に述べた保守ためのCASEの問題点を，設計時に必要な情報を入力しなくても，既存の原始プログラムを解析して，保守や再構築に役立つ必要情報を提供したり，ある種の自動変換を可能にする，という形で対処するものであるという位置づけも可能なことを指摘しておく．

われわれも最近，ソフトウェアの保守，あるいは進化プロセスに関心を持っている．とくにリエンジニアリングよりある意味で大胆な戦略である作り直し，すなわち古いソフトウェアを捨てて新しいものを作り直すというプロセスを，実証的に分析している[27, 33]．第1章の終りに述べた“不要求”分析やシステム統合化の再検討という知見も，これらの分析結果から得たものである．

3　自動プログラミングの実用化—一つの事例

自動プログラミングの研究としてよく知られているプロジェクトには，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 [23] や，プログラム作成時に知的な支援をするエディタ KBEmacs [31] がある．

以下では，Programmer's Apprentice よりさらに息の長い，C. Green による一連のプロジェクトを事例研究の対象として取り上げよう．これを取り上げる理由は，次のようである．

1. とにかく長い歴史を誇る．その間，いろいろな要素が取り込まれ，流れの変化もあり，多様な側面を持つ．
2. 純粋の研究として始まり実際的な応用に至るという過程を，一連の連続したプロジェクトとして歩んできている．
3. すでに述べたリエンジニアリングとの関連という点からだけでも，面白いツールを出している．

この他，筆者の個人的な事情をいえば，プログラムの検証や合成技法の研究にわれわれなりに取り組んでいた 1970 年代後半から，ある程度文献などで Green のプロジェクトに馴染みを持っており，そこから生まれた Reasoning Systems という会社には設立時の 1984 年と今年 1992 年と 2 回訪れていること，Reasoning Systems の製品である Refine を入手し，ゼミなどで試し始めていること，が挙げられる．

歴史的には，次のような順序でプロジェクトが起ち上げられ進められてきた．

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．

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．

なお，Kestrel Institute は現在でも存続している．そのもっとも最近の成果の一つは，D. R. Smith により開発された KIDS(Kestrel Interactive Development System) である [24, 25]. 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．

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たようである．結局ツールとしての Refine も，当初は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，ボトムアップのリ

エンジニアリングという分野に実用的な利用法を見い出すに至ったわけである．それを可能にしたのが，Dialect という構文解析系生成系である．これを用いて原始プログラムから分析操作の対象となるオブジェクトベースを作り，それに操作を施すことで，自由な“リエンジニアリング”が可能となる．その際，とくに言語としての Refine がもつプログラム変換機能が有効に働く．

Refine は次のようなものから構成されている．

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．言語としての Refine は CHI プロジェクトで作られた V の直接の後継言語で，論理式，集合や写像を直接記述でき，しかもある程度の効率で実行ができる．さらにオブジェクトベースへの操作を変換規則として記述でき，とくに対象オブジェクトが Refine プログラムの構文解析木であれば，プログラム変換が実現できる．

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．

4　おわりに

これまで AI という言葉は使わないできたが，ここで AI との関係についてちょっと触れておきたい．合成・変換といった技術は AI の一部とみなされる場合もあるが，ものの捉え方に違う面もあるのではないかと思う．

第一に，AI はやはり人間の知能の働きをモデル化しようとする意識が強いのに対して，合成や変換を研究する人々は抽象化され形式化された計算システムに関心があり，人間が頭の中でどうやっているかは二の次であるというのが普通であろう．これがソフトウェア開発への適用という実際的な場面でどう現れてくるかというと，合成・変換を基とするシステムは，たとえ玩具プログラムであれある範囲の対象に対しては一般性のある自動生成の手法を提供するのに対して，いわゆる AI をベースとするシステム（たとえば Programmer's Apprentice）は，かなり複雑な例題でも適当な発見的知識を与えればうまく動作するように見えるが，それが他の例題でどの程度通用するのかよく分らないという感じがある．

第二に，AI はソフトウェアの問題領域の記述，モデル化という点には馴染みがいい反面，問題の解法・アルゴリズムという点では，合成変換に一歩譲るだろう．これもそれぞれの分野の研究者の関心の違いに帰着することかも知れない．AI のモデル化能力という点は，オブジェクト指向分析がソフトウェア工学の世界でももてはやされていることと通じるところがあり，今後さらに実用価値が高まる可能性はある．しかし一方で，解法アルゴリズムの必要性は，やはり軽視できない．

プログラム変換の概念をもっともうまく取り入れているソフトウェア工学の実践に近い手法は M. Jackson の JSD 法であろうが，これに続くものがあまりないのが実態といえるかも知れない．しかし，上に述べた AI との対比は，合成変換にもう少し実践的な活用法を期待してもいいかも知れないという意味をこめたものである．

参考文献

[1] Ambriola, V., Ciancarini, P. and Montangero, C.: Software Process Enactment in Oikos, *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, ACM, 1990, pp. 183–192.

[2] Barrett, G.: Formal Methods Applied to a Floating-Point Number System, *IEEE Trans. Softw. Eng.*, Vol. 15, No. 5 (1989), pp. 611–621.

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

[4] Brooks, F. P. Jr.: No Silver Bullet: Essence and Accidents of Software Engineering, *IEEE Software*, April 1987, pp. 10–19.

[5] Chikofsky, E. and Cross, J. H.: Reverse Engineering and Design Recovery: A Taxonomy, *IEEE Software*, January 1990, pp. 13–17.

[6] Curtis, B., Kellner, M. I. and Over, J.: Process Modeling, *CACM*, Vol. 35, No. 9 (1992), pp. 75–90.

[7] Curtis, B., Krasner, H., and Iscoe, N.: A Field Study of the Software Development Process for Large Systems, *CACM*, Vol. 31, No. 11 (1988), pp. 1268–1287.

[8] Frailey, D. J.: Defining a Corporate-wide Software Process, *Proceedings of the 1st International Conference on the Software Process*, IEEE, 1991, pp. 113–121.

[9] Hall, A.: Seven Myths of Formal Methods, *IEEE Software*, September 1990, pp. 11–19.

[10] Huff, K. E. and Lesser, V. R.: A Plan-Based Intelligent Assistant that Supports the Process of Programming, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM, November 1988, pp. 97–106.

[11] Humphrey, W. S.: *Managing the Software Process*, Addison-Wesley, 1989.

[12] Inoue, K., Ogihara, T., Kikuno, T. and Torii, K.: A Formal Method for Process Descriptions, *Proc. 11th International Conference on Software Engineering*, May 1989, pp. 145–153.

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

[14] 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.

[15] Katayama, T.: A Hierarchical and Functional Software Process Description and its Enaction, *Proc. 11th International Conference on Software Engineering*, IEEE, May 1989, pp. 343–352.

[16] Kellner, M. I.: Software Process Modeling Support for Management Planning and Control, *Proc. 1st Int. Conf. Softw. Process*, IEEE, October 1991, pp. 8–28.

[17] Littlewood, B. and Strigini, L.: The Risks of Software, *Scientific American*, November 1992, pp. 38–43.

[18] Lowry, M. R. and McCartney, R. D. eds.: *Automating Software Design*, AAAI Press/MIT Press, 1991.

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

[20] Parnas, D. L.: Software Aspects of Strategic Defense Systems, *CACM*, Vol. 28, No. 12 (1985), pp. 1326–1335.

[21] Peuschel, B. and Schäfer, W.: Concepts and Implementation of a Rule-based Process Engine, *Proceedings of the 14th International Conference on Software Engineering*, IEEE, 1991, pp. 262–279.

[22] Phillips, M.: CICS/ESA 3.1 Experiences, in Nicholls, J. E. ed., *Z User Workshop: Proceedings of the Fourth Annual Z User Meeting, Oxford 1989*, Springer-Verlag, 1990, pp. 179–185.

[23] 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.

[24] Smith, R. S.: KIDS: A Knowledge-Based Software Development System, in [18], pp. 483–514.

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

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

[27] Tamai, T. and Torimitsu, Y.: Software Lifetime and its Evolution Process over Generations, *Proc. Conference on Software Maintenance – 1992*, Orlando, Florida, November 1992, pp. 63–69.

[28] Tamai, T. and Itou, A.: Requirements and Design Change in Large-Scale Software Development: Analysis from the Viewpoint of Process Backtrack, Research Report No.92-06, Graduate School of Systems Management, the University of Tsukuba, Tokyo, 1992.

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

[30] 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.

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

[32] 落水浩一郎：ソフトウェアプロセスモデルに基づくソフトウェア開発支援環境 Vela，日本ソフトウェア科学会第 7 回大会論文集，1990, pp. 205–208.

[33] 玉井哲雄，鳥光陽介：世代をまたがるソフトウェア進化プロセス — ソフトウェアの寿命と作り直しに関する考察 —，日本ソフトウェア科学会第 9 回大会論文集，1992，pp. 53–56.

[34] 玉井哲雄：第 9 回 ICSE に参加して，*SEAMAIL*, Vol. 2, No. 7 (1987), pp. 22–25.