特　集　「知的プログラミングの基礎」

知識処理的アプローチによるソフトウェア設計支援

Knowledge-based Assistance for Software Design

毛利 友治*
Tomoharu Mohri

＊　(株)富士通研究所ソフトウェア研究部
Software Laboratory, Fujitsu Laboratories Ltd.

1987年5月28日　受理

Keywords : software design, knowledge-base, artifitial intelligence.

1. はじめに

近年，ソフトウェア開発に知識情報処理技術を用いようとする動きが盛んである．ソフトウェア開発活動は，人間の諸活動の中でも，最も知的な活動の1つと考えられ，またそれゆえに，飛躍的な生産性向上が難しい分野でもあった．したがって，そのような状況のもとで苦しんできたソフトウェア開発者が，人工知能研究の一分野としての知識情報処理技術の発展に大きな期待を抱き，その成果をソフトウェア開発支援に援用したいと願うのはごく自然なことである．

本稿では，ソフトウェア開発活動の中でも，最も知的であると思われる設計活動に中心をおき，ソフトウェア設計支援に知識情報処理技術の応用を試みている最近の研究動向を紹介する．

2. ソフトウェア設計活動と知識利用

一般に設計問題の目標は，要求仕様を満たすべく既知の基本要素からなる新しい対象を合成することである．このような設計問題の大きな特質は，合成した対象を評価するための解析問題をその一部として含むとともに，設計対象自体が，設計が進むにつれて操作され可変となることである[1]．ソフトウェア設計において，上記の一般的特質がどのような形態で現れてくるかを，渡辺らの提唱するソフトウェア開発3元モデル[2]で見てみる．

図1にソフトウェア開発3元モデルの概念図を示す．このモデルではソフトウェア開発を，対象系，資産系，処理系の3つの系からなるととらえている．対象系とは開発対象であるソフトウェアそのもの，すなわち，仕様，設計書，中間生産物，プログラムなどで，時間とともに変化する．資産系には，各種ノウハウ，設計情報，ソフトウェア部品などが含まれる．これらの中には文書化されずに記憶に頼るものも含む．処理系は，人間系と言い換えてもいいもので，3つの階層からなると考えている．

I/O階層というのは，手作業の階層であり，設計書を書くとかコーディングをするというような作業を指す．変換階層というのは，資産の検索や演繹的推論による変換作業，変換結果の評価などを行う階層である．一

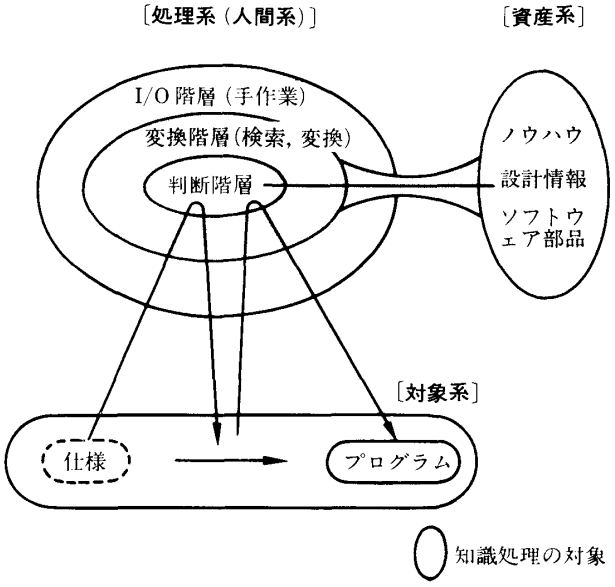


図1　ソフトウェア開発3元モデル

番奥の判断階層では，帰納推論や直観，発想といった高度な頭脳活動により，種々の設計上の判断を行う．

本モデルでは，処理系が，必要な情報を資産系から参照・利用しつつ，対象系を仕様から順次詳細化してプログラムに変換していく．図中の，処理系における判断・変換階層が設計活動における人間の頭脳活動の領域と考えられる．

現在の知識工学は，演繹的推論を主体としており，帰納的推論や直観，発想といった活動を計算機化することは困難である．したがって，ソフトウェア設計に知識工学を援用しようとする場合，2つの立場が考えられる．第1は，判断階層における活動をあまり必要としない分野・工程で，対象系の変換作業だけで，目標とするソフトウェアを自動生成しようとする立場である．第2は，自動生成をあきらめ，資産系，すなわち，ノウハウや設計情報の知識ベース化と，その再利用を中心として，人間の判断活動を支援しようとする立場である．前者の場合は，生成に用いられる知識の一貫性と完全性が大きなポイントとなる．後者の場合は，人間の頭脳活動の部分代替としてのツールによる検索・変換支援と，人間による判断活動とがうまく協調できることが，大きなポイントとなる．

以下3章で自動化を目指した研究例を，4章で再利用を目指した研究例を紹介する．

3．自動化支援と知識ベース

設計の自動化を目指した研究例として，富士通の藤本らによって開発されている SKBS (SDL Knowledge Based System)[3][4] を紹介する．SKBS は，電子交換ソフトウェアの仕様記述言語である，SDL (Functional Specification and Description Language) により記述された呼処理プログラムの機能仕様から，プログラムを自動生成することを目指しているシステムである．

SDL は，国際電信電話諮問委員会 (CCITT) の勧告による，電子交換機の内部状態間の遷移をフローチャート風に記述する言語であり，内部状態 (State)，信号の着信 (Input)，発信 (Output)，判断 (Decision)，処理 (Task) などの各種シンボルが用意されている．各シンボル内の記述には特に規定がなく，任意の抽象度で機能仕様を記述できる．SKBS では，概要状態遷移図 (G−STD ; General Level State Transition Diagram) と詳細状態遷移図 (D−STD ; Detailed State Transition Diagram) と呼ぶ，抽象度が異なる2種類の機能仕様記述を用いる．図2にその

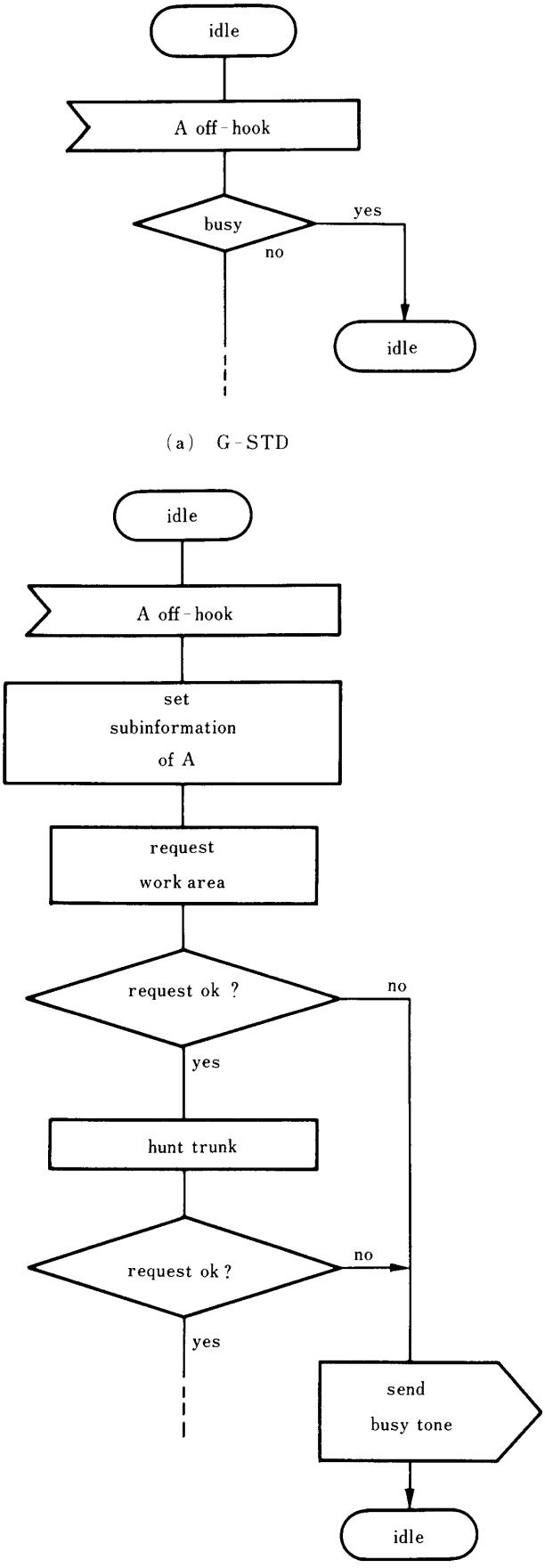


図 2 SDLによる仕様記述例

記述例を示す．

G-STD は，主に状態間の遷移関係に注目して処理の概要を表現するものであり，遷移のトリガと分岐を，比較的抽象的な名前の Input と Decision を用いて記述する．他方，D-STD は，より抽象度の低い Input, Decision, Output, および Task を用いて記述され，実際のソース・コードとほぼ対応する．

SKBS は，概要状態遷移図から詳細状態遷移図を生成するタスク生成サブシステムと，詳細状態遷移図からプログラムを生成するコード生成サブシステムから構成されている．それぞれのサブシステムは知識ベースに基づいて動作する．

タスク生成に関する知識は，いわば呼処理プログラムの設計詳細化の知識に相当し，概要状態遷移図，詳細状態遷移図のそれぞれの構成要素（それぞれ G-STD 要素，D-STD 要素と呼ぶ）について，フレーム形式で表現されている．図 2 (a)の概要状態遷移図から図 2 (b)の詳細状態遷移図を生成する際の知識の一部を図 3 に例示する．

種別	属性	記述
入力	名前	X off-hook
	マッチング条件	X は加入者名
	作用	加入者 X が受話器を上げた
判断	名前	busy
	作用	(OR (NOT 加入者用作業領域を確保できた) (NOT トランクを確保できた)

(a) G-STD 要素フレームの例

種別	属性	記述
入力	名前	X off-hook
	作用	加入者 X が受話器を上げた
	必要な後処理	X の補助情報を設定する
判断	名前	request ok?
	作用	加入者の作業領域を確保できた
	必要な前処理	加入者作業領域を要求する
	判断結果	
判断結果	名前	yes
判断結果	名前	no
	必要な前処理	busy tone を送出する

(b) D-STD 要素フレームの例

図 3 SKBS の知識表現の例

タスク生成サブシステムは，入力である概要状態遷移図上の各 G-STD 要素を順に取り出し，対応するフレームを検索する．そこから作用を取り出し，それに該当する作用を持つ D-STD 要素のフレームを検索する．求める D-STD 要素フレームが見つかると，まず，その D-STD 要素のマッチング条件を評価し，条件が成立することを確認し，G-STD 要素を該当する D-STD 要素に置き換える．次に必要な前処理を取り出し，それが現在生成中の D-STD 上ですでに達成された処理かどうかをチェックする．達成されていない場合は，該当する作用を持つ D-STD 要素フレームを検索し，そのフレームで記述されている要素を必要な前処理として挿入する．

このようにして，必要な前処理がすべて達成されるまで，後向き推論を行う．また，作用が，OR や AND などを用いて記述されているときは分解して，それぞれの作用が達成されるように推論する．必要な後処理についても，同様の後向き推論を行って G-STD 要素を D-STD 要素の列に展開する．以上の処理を概要状態遷移図の各要素について順次行うことによって詳細状態遷移図が生成される．

この知識表現は，外見的にはフレーム形式を用いているが，実際には，後向き推論のためのルールを表現したものになっている．

コード生成サブシステムについては，詳細状態遷移図のレベルがプログラム・コードに近いものになるので，ほぼ 1 対 1 の対応となる変換ルールでプログラムを生成する．それ以外の知識としては，文脈依存の記述を解釈する知識，複雑な構造を持つデータへのアクセスパスが省略された場合，それを補充する知識，コード最適化の知識が用いられている．これらの知識を変更することによって，CHILL など複数の言語のプログラムを生成することができる．

4. 再利用支援と知識ベース

4・1 IDeA

設計の再利用支援を目指した研究の例として，まずイリノイ大学の Lubars らにより開発されている IDeA[5][6] というシステムを取り上げる．IDeA は，設計スキーマという考え方に基づいたソフトウェア開発パラダイムを支援する．

設計スキーマは，いわば抽象化された設計であり，類似の設計のクラス（設計ファミリー）を代表している．設計スキーマは，再利用可能な設計情報の集まりとして知識ベース化されており，スキーマが適用でき

る問題のクラスが大きいほど，再利用性も高い．設計スキーマに含まれる情報としては，スキーマが実現を意図する機能（領域依存のオペレーション），設計上の決定事項間の関係，特定化（specialization）と詳細化（refinement）の規則，動作可能なインスタンスを作り出すためのプロトタイピング情報などがある．

設計スキーマは，特定化と詳細化の2つの手段により，下位の設計スキーマと関係づけられる．特定化とはis−a関係に相当する．すなわち，ある抽象的な設計スキーマの制約の一部を指定することにより，そのスキーマを特定の分野に適用した場合の設計スキーマが得られる．設計ファミリーとは，実はこの特定化された設計スキーマの集合のことである．

図4に特定化によるスキーマの関係づけの例を示す．IDeAでは設計スキーマの表現手法として，構造化分析[7]に用いられるデータフロー図を採用している．この例では，Compute Dependent Revenueという設計スキーマにおいて，入力であるRevenue Dependenceがfixed rateタイプ（この場合はパーセンテージ）の場合は，Compute Fixed Rate Dependent Revenueという特定化された設計スキーマとなり，Revenue Dependenceがscheduleタイプの場合は，Compute Scheduled Dependent Revenueという特定化された設計スキーマとなることを表している．

特定化がis−a関係であるのに対して，詳細化とはpart−of関係に相当する．図6に詳細化による関係づけの例を示す．

設計ファミリーにはまた，polymorphic/overloadedという属性が成立する．polymorphicとは，設計ファミリーを構成するすべてのメンバが同一の詳細化を共有する場合であり，そうでない場合をoverloadedという．図4に示したCompute Fixed Rate Dependent RevenueとCompute Scheduled Dependent Revenueとはoverloadedの関係になる．これに対して，Compute Fixed Rate Dependent Revenueの特定化として，図5に示されたCompute Fixed Rate Income TaxとCompute Fixed Rate Sales Commissionとはpolymorphicの関係にある．すなわち，共にその詳細化された形は図6となる．

設計スキーマと単純な設計テンプレートの決定的な違いは，制約（constraint）の伝播の概念である．制約とは，いわば抽象的な設計スキーマを特定化する際に指定される，設計上の決定事項といってよい．図4の特定化の例において，Revenue Dependenceのタイプを Fixed-rate にするか，Schedule とするかがこれにあたる．ある設計上の決定が行われ，制約が具体的に定まると，特定の設計スキーマが選択されることになる．特定化された設計スキーマに，それに対応した詳細化が定義されていれば，それにより詳細化が一段進むことになる．定められた制約は，詳細化された設計スキーマにも適用される．逆に，詳細化が新たな制約を付け加えることもあり，その場合，その制約は設計の他の部分にも波及し，さらに詳細化を押し進める．

このようにして，1つの制約，すなわち設計上の決

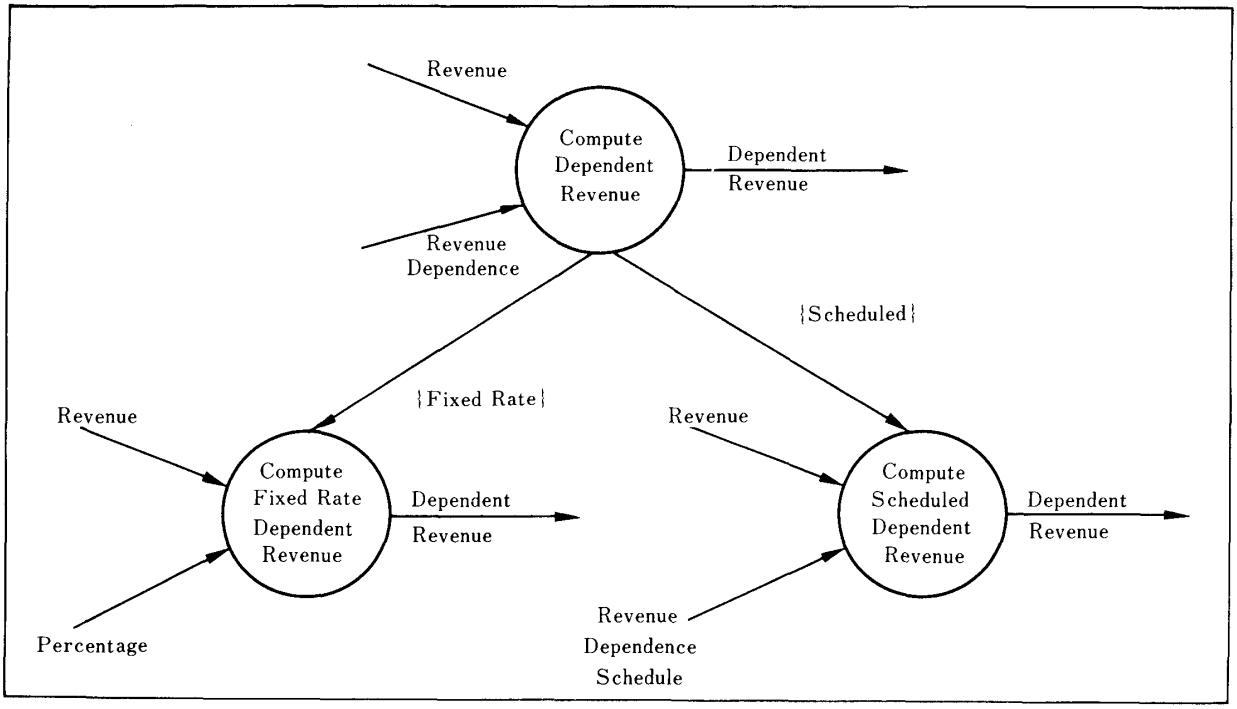


図4 Compute Dependent Revenue 設計ファミリーの特定化

定事項が設計の詳細化の連鎖を引き起こす．このことは，現実の設計上の決定事項の依存関係をよくモデル化している．

IDeA は，上記のような性質を持つ設計スキーマにより，構成された知識ベースを用いて，ユーザの設計スキーマの選択，特定化を支援し，詳細化と制約の伝播を自動化している．設計の完全性は，データフロー図においてすべての入出力が指定されているか否かによって判断し，無矛盾性は，束縛の伝播を自動化することにより保証している．

IDeA は，現実の設計活動を比較的よくモデル化するとともに，設計の再利用に関する多くの示唆を与えている．設計スキーマはかなり限定された領域で構築されることになろうが，その上で，最小限のコストで特定のユーザ向けアプリケーションの設計が得られるよう工夫されている．

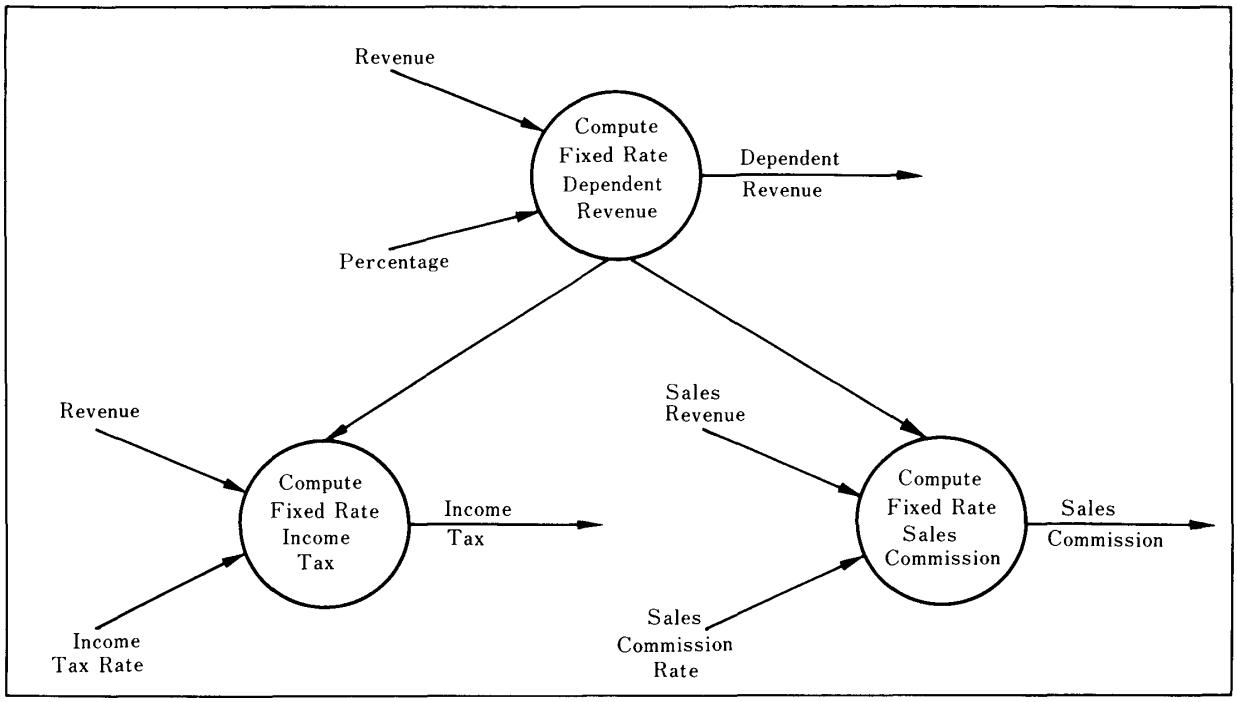


図 5　Compute Fixed Rate Dependent Revenue 設計サブファミリーの特定化と polymorphic

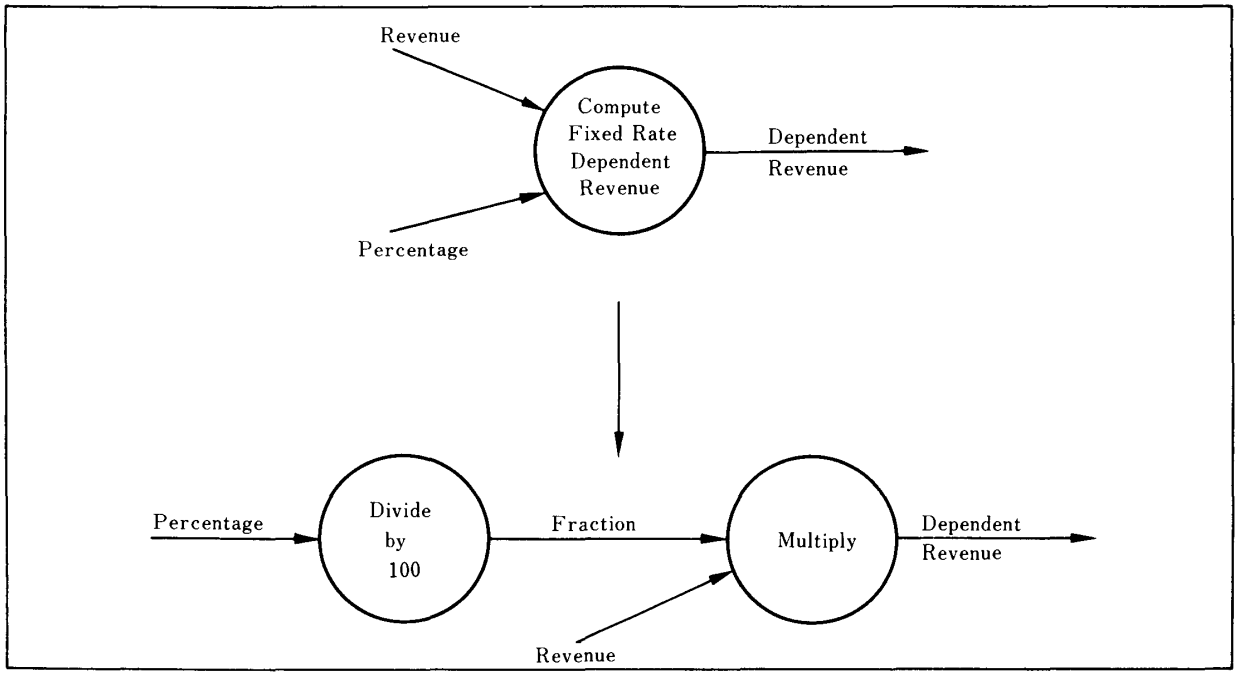


図 6　Compute Fixed Rate Dependent Revenue 設計サブファミリーの詳細化

```
OBJECT TYPE
   type-name : generic-object
   child-of  : ()
   parent-of : unknown
   components: (identifier : <string> ;
                type       : <string> ;
                because-of : <set of objects>)
   operators : (define, remove)

OBJECT TYPE
   type-name : dataflow
   child-of  : generic-object
   parent-of : unknown
   components: (part-of    : dataflow
                medium     : <string>
                from, to   : process)o
   operators : (redirect, nostart, noend)

OBJECT TYPE
   type-name : transform
   child-of  : generic-object
   parent-of : (process, external-entity, datastore)
   components: (inputs, outputs : <set of dataflows>)
   operators : ()
```

(a) データフロー図に関する知識表現の例

```
{identifier : London
 type       : external-entity
 because-of : ()
 inputs     : ()
 outputs    : (London-direct-sales-invoices,
               London-assigned-sales-invoices,
               London-statistical-sales-invoices)}

{identifier : London-direct-sales-invoices,
 type       : dataflow
 because-of : (London)
 part-of    : ()
 medium     : magnetic-tape
 from       : London
 to         : auto-load-and-edit }
```

(b) データフロー図のインスタンス定義例

図 7 データフロー図の知識表現とそのインスタンス定義例

4・2 REMAP

REMAP (REpresentation and MAintenance of Process knowledge)[8] は，New York 大学で研究されているシステムであり，設計過程に関する知識を知識ベース化し，要求仕様の変更や類似システムの開発時に活用することを目指している．

REMAP の設計過程に関する知識の基本的な考え方は，以下のようである．

① 設計過程は，相互に依存した設計上の決定事項 (design decisions) の連鎖からなる．

② 決定事項間の関係は，アプリケーション固有の規則に基づいている．

③ システムがプロトタイピングにより漸次に開発される場合は，開発者は，あるサブシステムから得られた経験を他のサブシステムの類似のコンポーネントの開発時に，類推により利用している．

REMAP は，上記の考察に基づき，①設計上の決定事項間の依存関係の表現と，決定事項の変更時の影響のトレース，②依存関係の一般的基礎となるアプリケーション固有の規則の抽出，③サブシステム中の類似性の検出，といった機能を有する．

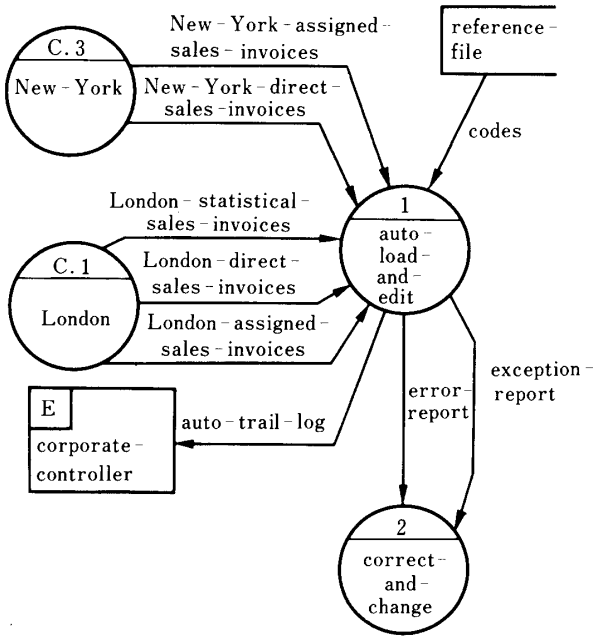


図 8 データフロー図の例

設計の表現には IDeA と同様に，構造化分析に用いられるデータフロー図を用いている．データフロー図に関する知識は，フレームを用いてシステム内にモデル化されており，設計者はそのインスタンスを定義することにより，設計を定義する．図 7 にデータフロー図に関する知識表現と，そのインスタンスの定義例を示す．また，対応するデータフロー図を図 8 に示す．

インスタンス上の各フレームは，1 つの設計上の単位 (“設計オブジェクト” と呼んでいる) に対応する．REMAP は，1 つの設計オブジェクトの出現により，1 つの設計上の決定がなされたと解釈する．したがって，設計上の決定事項間の依存関係は，設計オブジェクト間の依存関係と解釈され，この依存関係を表現するために，設計オブジェクトには特別のスロット (“because-of” スロット) が設けられている．図 7 の London-direct-sales-invoices のフレームの because-of スロットは，設計オブジェクト London を値として持っているが，これは「London という設計オブジェクトが存在しなくなれば，London-direct-sales-invoices という設計オブジェクトはその存在根拠を失う」と解釈される．したがって，設計変更により London がなくなれば，London-direct-sales-invoices も同時に消去されることになる．REMAP は，このスロットによる設計オブジェクト間のネットワークを把握しておくことにより，設計変更時の影響

範囲を把握することができる．

REMAP は，次にこのような依存関係の例から，アプリケーション固有の一般化された規則を導くことを目指している．たとえば，図8の例における auto-load-and-edit は New-York からの2つのデータフロー（New-York-assigned-sales-invoices, New-York-direct-sales-invoices）により，その存在根拠を保証されているが，実は，それらのデータフローのすべての性質が必要なのではなく，それらのデータフローが“計算機化されている”場合にのみその存在根拠がある．このことを一般化されたアプリケーション固有の規則として書くと以下のようになる．

{dataflow medium : computerized}
→auto-load-and-edit
{dataflow medium : paper}
→manual-add-and-edit

このような一般化された規則を，知識として持つことは，設計変更時や類似システムの開発時の設計支援に大きな力を発揮することが予想される．しかしながら，現在のところ REMAP は，依存関係の具体例から，このような一般化された規則を自動抽出する方法は見い出せておらず，抽出は人手に頼っている．

前述の一般化された規則は，データフローをさらに分類し図9のようなオブジェクト階層を構成することにより，

{computerized-invoices}
→auto-load-and-edit
{paper-invoices}
→manual-add-and-edit

と表現することもできる．このように，REMAP は，設計オブジェクトに関する知識を，図9に示したような一般化の階層構造と，その階層構造中のノードに対応づけたルールとして記憶している．類推による設計知識の適用は，まず，新しい設計オブジェクトが，一般化の階層構造のどの抽象レベルに位置するかを決定することから始め，次に，そのレベルで適用可能なルールを新しい設計オブジェクトに適用する．その後，一般化の階層構造をさらに上位にたどりつつ，適用可能なルールがあれば順次適用していく．

このようにして，REMAP は，設計過程の知識を動的に獲得し，システムの保守や開発に再利用することを目指している．

現在のところ，REMAP は，①依存関係の一般化規則を導く理論，手法が示されていない，②新しい設計オブジェクトの抽象レベルを決定する具体的方法が示されていない，などの問題はあるが，新しい設計知識を比較的容易に追加し，再利用できる枠組みを提供している．

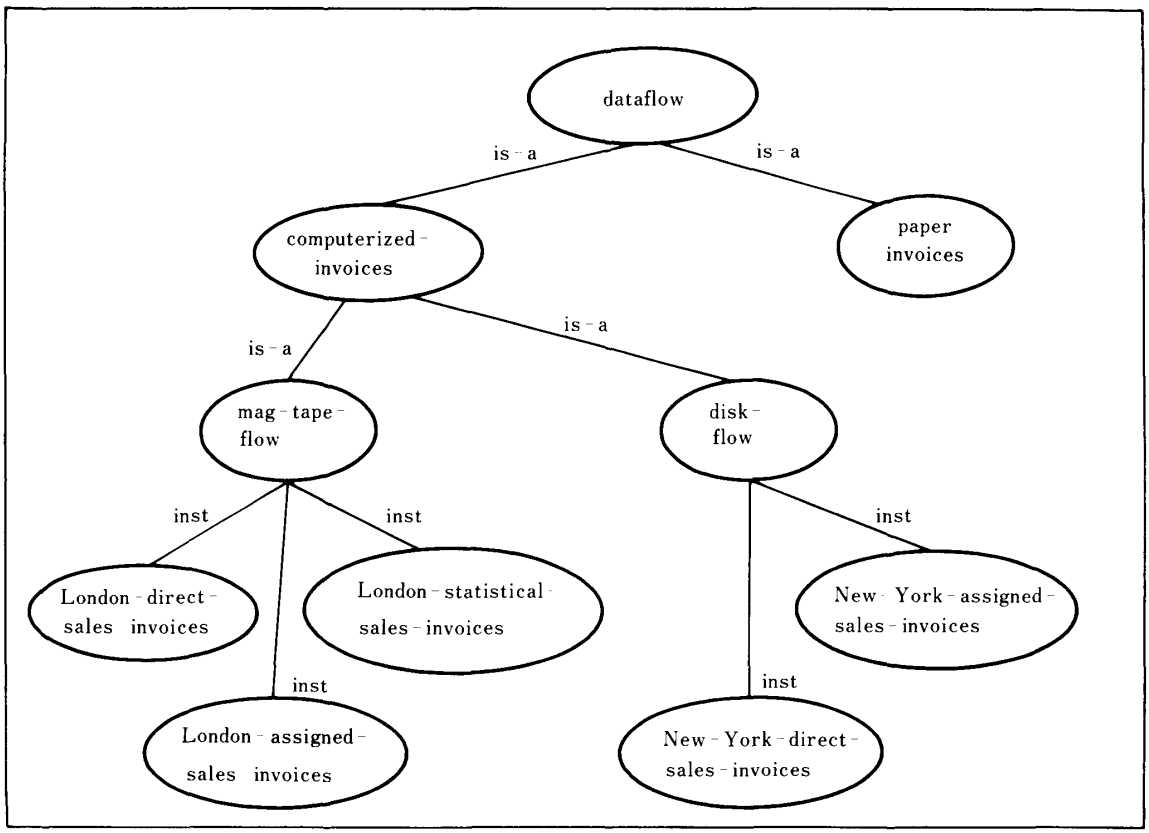


図 9　一般化の階層構造

5. 知的ソフトウェア設計支援の周辺

自動化支援ではいうまでもなく，自動化に用いられる知識の質，量が大きなポイントとなる．知識ベース中の設計知識の完全性・一貫性の確認は，現在のところ人手に頼るしかない．ごく限定された分野ではそれも可能であろうが，少し大きなシステムになるとそれも不可能となる．したがって，自動化支援が成功するためには以下のような条件が必要となろう．

① 分野固有のオペレーションがよく整理され，完備していること

② オペレーションの起動順序（プログラムの制御構造）がパターン化されており，その適用基準が明確であること

③ 分野における設計手法と，設計を表現する手段（仕様記述言語）が確立していること

再利用支援では，人間との作業の協調性と制約の伝播（constraint propagation）の自動化が重要となる．人間との協調作業という観点からは，IDeA も，REMAP も，人間の判断（設計上の決定事項）をいかに支援中に取り込み，記録し，再利用に活かすかに腐心している．制約の伝播という観点からは，REMAP の“一般化された規則”は，伝播の経路を陽に記述したものといってもよい．再利用による設計支援のポイントは，このような「設計過程」の知識をいかに記録し，活用するかにありそうである．

ここで紹介したシステムは，いずれも設計仕様の表現に図形を有効に用いている点が興味深い．設計作業支援における対話用インタフェースとしての長所を最大限に活かすためであろう．今後もこのような傾向が進むとすれば形式言語による仕様記述の研究とともに，図形による仕様記述のセマンティクスを，内部表現としてフォーマルに表現する研究が重要となってこよう．その点からは，ビジュアル・プログラミングに関する一連の動き[9]-[11]にも注目したい．

ソフトウェア・ライフサイクル全般にわたって，知識工学的手法の援用を試みている例としては，Kestrel Institute の C. Green らにより行われている，KBSA (Knowledge-Based Software Assistant) の研究[12]がある．このグループは，PSI[13]，CHI[14]といった自動プログラミングシステムを開発したことで著名であるが，KBSA プロジェクトでは，知識ベースに基づいた新しいソフトウェア開発パラダイムの確立を目指しており，今後その動向が注目される．

6. おわりに

設計問題は，知識工学応用の分野の中でも難しい部類に入る．ましてや，方法論が未成熟なソフトウェア設計の分野では多くの困難が予想される．したがって，研究例もまだまだ少なく，実のある成果を生みだすにはかなりの時間がかかるであろう．

しかしながら，知識工学を中心とした人工知能研究は，今後さらにその内容を充実させていくものと思われ，高次の頭脳活動としてのソフトウェア開発と人工知能研究のかかわり合いも，今後ますます深くなっていくものと思われる．

◇ 参 考 文 献 ◇

(1) 川戸，斉藤：論理装置の CAD における知識工学の応用，情報処理学会誌，Vol. 25, No. 10, pp. 1161-1168 (1984).

(2) 渡辺，小野，堂山：ソフトウェア開発モデルと知識処理適用法，昭和 61 年度電子通信学会全国大会，pp. 8-211-8-212 (1986).

(3) Fujimoto, H., *et al.* : Telecommunications software design support system based on specification languages, Proc. of Eurocon '84, pp. 271-275 (1984).

(4) 加藤，吉田，広瀬，鈴木：知識ベースを用いた SDL 支援システム，情報処理学会知識工学と人工知能研究会資料，34-5 (1984).

(5) Lubars, M. D. and Harandi, M. T. : Knowledge-based software design using design schemas, Proc. of 9th International Conf. on Softw. Eng., pp. 253-262 (1987).

(6) Lubars, M. D. and Harandi, M. T. : Intelligent support for software specification and design, IEEE Expert, Vol. 1, No. 4, pp. 33-41 (1986).

(7) DeMarco, T. : Structured Analysys and System Specification, YOURDON Inc., New York (1978).

(8) Dhar, V. and Jarke, M. : Using teleological design knowledge for large systems development and maintenance, Proc. of the International Workshop on Expert Systems&Their Applications, pp. 359-388 (1986).

(9) Moriconi, M. and Hare, D. F. : Visualizing program design through PegaSys, IEEE Comp., Vol. 18,No. 8, pp. 72-85 (1985).

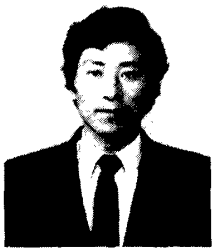
(10) Brown, M. H. and Sedgewick, R. : Techniques for algorithm animation, IEEE Software, Vol. 2, No. 1, pp. 28-39 (1985).

(11) Glinert, P. E. and Tanimoto, S. I. : Pict ; An interactive graphical programming environment, IEEE Comp., Vol. 17, No. 11, pp. 7-25 (1984).

(12) Green, C., *et al.* : Report on a Knowledge-Based Soft-

ware Assitant, Artifitial Intelligence and Software Engineering, pp. 377-428, Morgan Kaufmann Publishers Inc., Los Altos California, (1986).
(13) Green, C., *et al.* : Results in knowledge-based program synthesis, Proc. of IJCAI-6, pp. 342-344 (1979).
(14) Smith, D. R., *et al.* : Research on knowledge-based software environments at Kestrel Institute, IEEE Trans. on Softw. Eng., Vol. SE-11, No. 11, pp. 1278-1295 (1985).

著 者 紹 介



毛利 友治

昭和48年九州大学工学部卒業. 昭和50年同大学院修士課程修了. 同年(株)富士通研究所入社. 現在, ソフトウェア研究部主任研究員. ソフトウェア工学, 人工知能の研究に従事. 情報処理学会, IEEE会員.