関数スキーマベースを用いたソフトウェア設計自動化†

鯵坂恒夫†† 阿草清滋†† 大野 豊††

ソフトウェア設計の自動化のために形式化された設計工程において，仕様データベースを利用する技法について述べる．設計工程への入力は目的とするソフトウェアの入出力データの仕様，工程からの出力はその入出力データ間の写像を実現する関数の仕様である．このような工程に適合した設計仕様記述言語Fによれば，データおよび関数はともに結合子とよぶ作用素を用いて形式的に記述されるので，それらの間の対応規則を定式化することができる．しかし構造に限定された少数の規則をもとにデータ仕様から機械的に関数を導く方法には限界がある．そこで一般化した関数仕様である関数スキーマを設計仕様部品として関数スキーマベースに蓄え，これを設計に利用する技法を提案する．各関数スキーマには検索のための見出しとして対応するデータ変換式が付随している．これはデータ構造の要素の現れ方やデータ間関係も含めてデータ仕様をパターン化したものである．すなわち関数スキーマベースはデータと関数の対応規則をデータベース化したものといえる．関数スキーマは与えられたデータ仕様とデータ変換式との単一化操作によって検索され，それと同時に特殊化される．この単一化の規則は設計作業における設計者やプログラマの思考法を定式化したものと考えられる．このような処理内容による検索手段は部品化と再利用による設計自動化にきわめて重要である．

1. はじめに

自動制御や CAD/CAM，あるいはオフィス・オートメーションなどさまざまな分野で計算機を導入した自動化がすすんでいるのに比べて，計算機ソフトウェア自体の設計自動化はあまり進展していないように思われる．それはソフトウェアを構成する基本要素（演算機能，制御機能やデータ要素）の組合せの自由度がきわめて高く，さまざまな効果がそれぞれ多様な方法で実現されうるからである．

一般にある作業を自動化しようとするとき，そこで用いられている概念や機能を整理し，形式化し，場合によっては単純化する必要がある．本研究ではソフトウェア設計を自動化するため，次のように設計工程を規定する．すなわち，設計工程への入力は目的とするソフトウェアの入出力データの仕様，工程からの出力は与えられた入出力データ間の写像を実現する関数の仕様とする．筆者らはこのような工程による設計に適した仕様記述言語Fをすでに提案した[6]．F言語ではデータおよび関数をともに結合子（combinator）とよぶ作用素を用いて形式的に記述するので，それらの間の対応規則を定式化することができる．この対応規則に従って，与えられたデータ仕様から目的ソフトウェアの処理構造を機械的に導くことができる．

しかしこのように設計視野の限定および工程の形式化をはかっても，なお設計の過程は多岐にわたる．複雑なデータ構造変換やデータの値に関する仕様に基づく変換処理となると，少数の構成的な対応規則を反復適用して機械的な設計を行うのは限界がある．そこで本論文ではデータと関数の対応規則をデータベース化し，それを用いた設計自動化の技法を提案する．結合子によるデータおよび関数の記述はパターン化しやすく，それぞれデータスキーマ，関数スキーマとなる．データベースにはデータスキーマを見出しとしてそれに対応する関数スキーマが蓄えられる．これを関数スキーマベースとよぶ．最近，ソフトウェアの部品化と再利用に関する議論が盛んになりつつあるが，その対象を完成部品としてのプログラム・コード列とすると，プログラム構成要素の組合せの多様性のために部品化の基準設定，部品の検索ともに困難が予想される．本研究は処理の枠組を部品化しそれをソフトウェア設計に利用する方向を示すものである．

2. 仕様記述言語 F に基づくソフトウェア設計自動化の基礎

本章では設計仕様記述言語Fおよびそれに基づくソフトウェア設計自動化の基本的な考え方について，その概要を述べる．

2.1 F言語によるデータ仕様

F言語に基づく設計工程への入力となるデータ仕様を次のように定義する．

- データ仕様はデータ定義式およびソート関係式の集合である．
- 〈データ定義式〉::=〈データ型〉==〈データ結合式〉

† Software Design Automation Based on Function Schema Base by TSUNEO AJISAKA, KIYOSHI AGUSA and YUTAKA OHNO (Department of Information Science, Faculty of Engineering, Kyoto University).

†† 京都大学工学部情報工学教室

```
"domain" == ["start","stop","users"]
  "users" == ["user"...]
  "user" == ["userid","procs"]
    "procs" == ["proc"...]
    "proc" == ["pid","term"]
      "term" == ("norm","trap")
"range" == ["header","recs","trailer"]
  "recs" == ["rec"...]
  "rec" == ["userid","nproc","ntrap"]

"nproc" = cnt("proc")
"ntrap" = cnt("trap")
"header" = cnst()
"trailer" = cnst()
```

図 1 データ仕様記述例 1
Fig. 1 Example of data specification 1.

● データ型はその名前を二重引用符で括って表記する.

● $\delta, \delta_1, \cdots, \delta_n$ をデータ型とするとき,

δ, $[\delta_1, \cdots, \delta_n]$, $[\delta \ldots]$, $\{\delta_1, \cdots, \delta_n\}$

はデータ結合式である.

ただし, [,], [...], { , } はそれぞれ連接, 反復, 選択のデータ結合子[6] である.

● データ定義式の右辺が単独のデータ型であるとき, 左右両辺のデータ型を同一とみなす.

● データ定義式の左辺に現れないデータ型をとくにソートと呼ぶ.

● 〈ソート関係式〉::=〈ソート〉=〈ソート生成式〉

〈ソート生成式〉

::=〈ソート生成子〉

(〈データ型〉|〈ソート生成式〉, ……)

● データ仕様には "domain", "range" を左辺にもつデータ定義式が一つずつある.

データ結合子がデータ構造を構成するのに対して, ソート生成子はデータ間関係やデータの意味 (データ値に関する情報) を表す. ソート生成子は本稿の例には cnst, cnt, sum がでてくるが, その集合は固定しておらず, 場合に応じて問題向きに定義しなければならないものもあろう. データ結合子に比べてはるかに形式化の困難な概念要素である.

図 1 に F 言語によるデータ仕様の記述例を示す. 例題は次のとおりである. ある計算機システムのログ情報を一部要約したファイルは, ログの開始, 終了時刻とともに各ユーザの起動したプロセスの ID とその終了状態 (正常終了またはトラップされて異常終了) を記録している. これを入力して, ユーザごとに起動プロセス数およびトラップされたプロセス数を集計する. ソート生成子 cnt は反復構造の要素であるそのオペランドの個数を表すソートを与え, cnst は定数的に生成されるソートであることを表す.

2.2 F 言語による関数仕様

設計工程から出力されるのは以下のように定義される F 言語による関数仕様である.

● 関数仕様は関数定義式の集合である.

● 〈関数定義式〉::=〈関数〉:〈定義域〉=>〈値域〉

==〈関数結合式〉

● 定義域, 値域はデータ型またはデータ結合式である.

● $\phi_1, \cdots, \phi_n$ を関数, $\psi_1, \cdots, \psi_{n-1}$ を命題関数, Φ を関数結合子とするとき,

$[\phi_1, \cdots, \phi_n]$, $\{\psi_1 -> \phi_1; \cdots; \psi_{n-1} -> \phi_{n-1}; \phi_n\}$,

$\phi_1 . \cdots . \phi_n$ および $\Phi(\phi_1, \cdots, \phi_n)$

は関数結合式である*.

ただし, [,], { -> ; }, . はそれぞれ連接, 選択, 合成の関数結合子である. $\Phi(\phi_1, \cdots, \phi_n)$ の形で使われる関数結合子のうち基本的なものは縮約結合子 R および分配結合子 A で, これらはそれぞれ Backus の FP の関数形式 insert, apply-to-all[2] と同様のものである[6].

● 関数仕様には main を左辺にもつ関数定義式が一つあり, その定義域, 値域は "domain", "range" である.

関数結合子には定義域, 値域に現れるデータ結合子に対応して基本的な構造変換を行うもの (2.3 節参照) の他に, ソート生成子に対応して一定の処理を行うものや, あるきまった定義域, 値域のパターンに対応してより複雑な写像を実現するもの (3 章参照) もある. 後者は本稿の例には CNT, SUM, SEAR, COLAT などがでてくるが, その集合は固定していない. ただし, それらは基本的な関数結合子と特定の原始関数の

```
main: "domain" => "range"
  == [header,p_recs.users,trailer]
p_recs: "users" => "recs"
  == A(p_rec)
p_rec: "user" => "rec"
  == [userid,p_nproc.procs,p_ntrap.procs]
p_nproc: "procs" => "nproc"
  == CNT()
p_ntrap: "procs" => "ntrap"
  == CNT().A(p_ntrap_1)
p_ntrap_1: "proc" => "proc"
  == (is_trap.term -> id; nil )
```

図 2 関数仕様記述例 1
Fig. 2 Example of function specification 1.

* 関数結合式のネスト, すなわち結合子のオペランドが結合式であるものも便宜的に用いる.

表 1 定義域，値域と関数との基本的な対応規則 (ECR)
Table 1 Elementary correspondence rules between domain/range and function (ECR).

定義域	値域	関数
"x"*	"y" == ["p", "q"] 〈連接〉	[f, g] 〈連接〉 f: "x" =〉 "p" g: "x" =〉 "q"
"x" == ["a"...] 〈反復〉	"y" 〈ソート〉	R(f, g) 〈縮約〉 f: ["a", "y"] =〉 "y"***
"x" == ["a"...] 〈反復〉	"y" == ["p"...] 〈反復〉	A(f) 〈分配〉 f: "a" =〉 "p"
"x" == {"a", "b"} 〈選択〉	"y"**	{is_a-〉 f; g} 〈選択〉 f: "a" =〉 "y" g: "b" =〉 "y"
"x"=={"a", "b"} 〈選択〉	"y" == {"p", "q"} 〈選択〉	{is_a-〉 f; g} 〈選択〉 f: "x" =〉 "p" g: "b"=〉"q"

* 右辺はどのデータ結合式でもよい.
** 右辺は選択を除くどのデータ結合式でもよい.
*** q は "y" の単位元的なオブジェクト（縮約作用の初期値）を与える定数関数である.

組合せで作られる.

図1のデータ仕様に対応する関数仕様を**図2**に示す. userid や term のように定義域に現れるデータ型と同名の関数は，連接構造の要素をとりだすセレクタ関数[2]の便宜的な記法である. header や trailer のように値域に現れるデータ型と同名の関数は定数関数であることを表す. CNT は計数処理を行う関数結合子である.

2.3 データと関数の対応

F言語に基づくソフトウェア設計の基本的な考え方は，データと関数の間の定式化された対応規則を設計工程に入力される入出力データ仕様に繰り返し適用し，その間の写像を実現する関数仕様を機械的に導くものである[1]. 定義域，値域を与えるデータ定義式と関数結合式（およびそのオペランドとなる関数の定義域，値域）との基本的な対応規則 Elementary Correspondence Rules, ECR) を**表1**に示す.

図1のデータ仕様にこの ECR を適用し図2の関数仕様を導いてみると，main や p_rec は1番目，p_recs は3番目の規則に従って得られることがわかる. CNT は2番目の規則に現れる縮約結合子 R の特殊化されたものである. p_ntrap_1 の選択結合は，連接と選択のデータ結合子に関する分配法則を "procs" に適用した後，ECR の4番目の規則によって導かれる.

この例題は ECR に比較的よく従う例であるが，p_recs の定義域が連接構造の "domain" 全体ではなくその要素の "users" であること（p_nproc, p_ntrap も同様）や，p_rec の第1オペランドがセレクタ関数であることは ECR からは導けない. 関数結合子 CNT の使用はソート関係式に基づいており，また p_ntrap における関数合成も ECR の範囲を越えている.

図3にそのデータ仕様を示す例題[5]は関数合成が本質的である例である. 文献の題目や著者などのデータを含むファイルと文献の要約を集めたファイルをもつシステムに検索要求を入力し，文献の要約を出力する. 検索が二重になっており，それが関数合成によって実現される. 2回の検索処理を仲介するデータとして

"ixs"== ["indx"...]

を用いて，以下のように設計されるべきである.

main : "domain" =〉 "range"
  == g. f
f : "domain" =〉 ["absfile", "ixs"]
  == [absfile, f 2 (docfile, request)]*
f 2 : ["docfile", "request"] =〉 "ixs"
g : ["absfile", "ixs"] =〉 "range"

ECR の限界は関数仕様を導くための情報を一つのデータ結合式に表された純粋にシンタクティックなデータ構造だけに求めているところからくる. 設計情報はデータ構造の要素の現れ方にも多く含まれている. たとえば最初の例題で p_recs の定義域が "users" であることは，p_recs の値域 "recs" の反復要素の中に "userid" があり，同じ "userid" が "users" の反復要素に含まれていることから推定できる. また図3

```
"domain" == ["docfile","absfile","request"]
  "docfile" == ["document"...]
    "document" == ["title","docdata","keyws","indx"]
      "keyws" == ["keyword"...]
  "absfile" == ["absrec"...]
    "absrec" == ["indx","abstract"]
  "request" == "keyword"

"range" == ["abstract"...]
```

図 3 データ仕様記述例2
Fig. 3 Example of data specification 2.

* f 2 (docfile, request) は f 2. [docfile, request] の便宜的記法である.

の例題において，“request” すなわち “keyword” が “docfile” の反復要素にも現れ，さらにそこに含まれる “indx” が “absfile” の反復要素として現れていることを一般化すれば，二重の検索処理のパターンが得られることが見込まれる．

次章に述べる関数スキーマベースは，このような設計作業における着眼点に関する観察をもとにデータ仕様をソート関係式も含めてパターン化し，それとともに対応するパターン化された関数仕様をデータベース化したものである．

3. 関数スキーマベースと自動設計

ソフトウェアの部品化と再利用はソフトウェア開発工程の自動化のための有力な手段として最近とくに注目されている．部品化と再利用における課題は次の二つに大別される．

（Ⅰ） 部品の作成法

（1） 汎用性，標準性，再利用性，大きさなどの観点から何を部品として採用するかの基準．

（2） 既存のソフトウェアから部品を抽出する方法．

（Ⅱ） 部品の適用法

（1） 部品を検索する方法．

（2） 汎用部品を特殊化する方法．

（3） 部品間の結合法．

部品化をその対象によって分類すると，ソフトウェア・ライフサイクルの軸に沿って要求仕様，設計仕様，プログラムの部品化が考えられる．しかし，プログラムはその構成要素の概念レベルが低く，したがって要素の組合せ方がきわめて多様になるために標準部品の設定がむずかしい．また検索についても部品名による検索ではなく処理の内容をもとにした検索を行おうとすると，構成要素の組合せ方の一般化が容易でなくコンパクトな検索キーが抽出できない．一方，要求仕様には問題ごとに異なる概念要素が用いられることが多く，部品化のためには思いきった整理と体系化が必要であると考えられるがこれについては本論文では議論しない．

本章では設計仕様の部品化と再利用による設計工程の自動化について述べる．標準部品となるのは前章で述べた関数結合式を一般化した関数スキーマである．部品の検索キーにはデータ結合式およびソート関係式を一般化したデータスキーマによって定義域，値域を定めるデータ変換式を用いる．部品はデータ仕様とデータ変換式との単一化操作によって検索され，汎用部品である関数スキーマは検索と同時に特殊化される．

3.1 データ変換式

関数スキーマの検索のための見出しとなるデータ変換式を次のように定義する．

●〈データ変換式〉
　::=〈データスキーマ〉 =〉〈データスキーマ〉
　　{〈スキーマ定義式〉}
　　{〈ソート関係図式〉}

●データ型変数（$a, b, \cdots$ で表す），ソート変数（$s, t, \cdots$ で表す）はデータスキーマである．

●$\sigma_1, \cdots, \sigma_n$ をデータスキーマ，Σ をデータスキーマ結合子とするとき，$\Sigma(\sigma_1, \cdots, \sigma_n)$ はデータスキーマである．

●データスキーマ結合子には C, I, S, C', I^*, X があって，はじめの三つはそれぞれ連接，反復，選択のデータ結合子に対応している．あとの三つについては 3.3 節で述べる．

●データスキーマ $C(\sigma_1,\cdots,\sigma_n)$ は少なくとも $\sigma_1, \cdots, \sigma_n$ を要素として含む連接構造を表す．すなわち連接構造の要素として $\sigma_1, \cdots, \sigma_n$ 以外のものを含んでいてもよく，また $\sigma_1, \cdots, \sigma_n$ の出現順序も任意である．S についても同様である．

●データスキーマ $I(\sigma)$は σ を要素とする反復構造を表す．I についてはオペランドはたかだか一つで反復要素を陽に示す必要がないときは省略してもよい．

●〈スキーマ定義式〉
　::=〈データ型変数〉 ==〈データスキーマ〉

●〈ソート関係図式〉
　::=〈ソート変数〉=〈ソートスキーマ〉
　〈ソートスキーマ〉
　::=〈ソート生成子〉
　　(〈データ型変数〉|〈ソートスキーマ〉, …)

例として図1のデータ仕様をデータ変換式の形に書くと次のようになる．

$C(s, t, IC(u, I(a))) \Rightarrow C(p, IC(u, y, z), q)^{\star}$

$a == C(v, S(w, x))$

$y = \mathrm{cnt}(a)$　$z = \mathrm{cnt}(x)$

$p = \mathrm{cnst}(\,)$　$q = \mathrm{cnst}(\,)$

データ仕様のパターン化を困難にしている最大の要因は連接や選択のデータ結合子が順不同，不定個のオ

---

* $I(C(a))$ の代りに $IC(a)$ のように，あいまいにならない限り括弧を省略する．

ペランドをとることである．データスキーマはたんに変数を導入してデータ仕様をパラメータ化しただけでなく，データスキーマ結合子の導入によって関数の設計に本質的な影響を与えるデータ構造の要素を抽出することにより，データ仕様のより幅広い融通性のあるパターンを記述することができる．これは3.3節に述べる単一化規則に反映され，関数スキーマベースのコンパクトな検索キーとなる．

3.2 関数スキーマ

関数スキーマの定義を以下に述べる．

- 関数および関数変数は関数スキーマである．対応するデータ変換式の定義域側のデータ型変数と同名の関数変数は，連接構造の要素をとりだす関数であることを表す．値域側のデータ型変数と同名の関数変数は定数関数であることを表す．
- 関数スキーマをオペランドとする関数結合式は関数スキーマである．
- $\lambda_1, \cdots, \lambda_n$ を関数スキーマ，Λを関数スキーマ結合子とするとき，$\Lambda(\lambda_1, \cdots, \lambda_n)$ は関数スキーマである．
- 関数スキーマ結合子は C, S のいずれかであり，それぞれ連接，選択の関数結合子に対応している．
- 関数スキーマ $C(\lambda_1, \cdots, \lambda_n)$ は少なくとも $\lambda_1, \cdots, \lambda_n$ を要素として含む連接結合の関数を表す．Sについても同様であるが，最後を除く奇数番目のオペランドは命題関数変数であり次のオペランドと対になっている．

要約すると，関数スキーマは関数変数を含むネストされた関数結合式である．ただし，データスキーマ結合子に対応して，やはり不定個のオペランドをとる連接と選択の関数結合子についてはスキーマ化したものも用いる．

3.3 データ仕様とデータ変換式の単一化

関数スキーマベースを用いる設計過程は，ECR の代りに関数スキーマベースを用いることを除いて 2.3 節に述べたものと同じである．関数スキーマは未設計の関数の定義域，値域を与えるデータ仕様と，関数スキーマベースに見出しとして含まれるデータ変換式との単一化操作によって検索される．この単一化操作は設計作業において設計者やプログラマが着眼する点，すなわちデータ仕様のなかで設計に本質的な影響を与える部分を抽出する作業を形式化，自動化したものである．

単一化の規則を以下に挙げる．

- ソート変数にはソートが，データ型変数にはデータ型（ソートも含む）が代入可能である．同じ変数には同じソートまたはデータ型が代入されなければならない．
- データスキーマ結合子 C, I, S には対応するデータ結合子がマッチする．
- I^* には連続任意個の（0を含む）反復データ結合子がマッチする．
- 定義域に現れる連続する連接データ結合子は単一化にあたっては展開する．これは定義域の多重連接構造は関数の構造設計には寄与せず，たとえば

  [["a", "b"], "c"] と ["a", "b", "c"]

  を同一視してよいということを反映している．
- C' には連接データ結合子がマッチするが無視してもよい．
- Xにはいかなるデータ結合子の組合せおよび（オペランドをもたないときは）単独のデータ型もマッチする．
- 代入を行った後データスキーマ結合子のオペランドとして現れないデータ結合子のオペランドは無視して単一化される．また，データスキーマ結合子のオペランドとデータ結合子のオペランドの出現順序も無視して単一化される．
- ソート関係図式とソート関係式の単一化においてはソート生成子がリテラルとして一致することが必要である．

代入やオペランドの対応に関する規則からもわかるように，ここでいう単一化はデータ仕様をデータ変換式に単一化するもので，節形式述語論理式の単一化とは異なり方向性がある．

データ変換式の変数への代入と同時に，対応する関数スキーマの変数のうちデータ型変数と同名のものに対しても代入が行われる．すなわち，検索のための単一化操作と同時に関数スキーマが特殊化される．

単一化の例として再び図1，図2に示した例題を用いる．main の2番目のオペランドとなる関数の定義域，値域はそれぞれ "domain", "recs" であるが，これは次のデータ変換式と単一化可能である．

$C(IC(S)) \Rightarrow IC(S)$

これを確かめるには定義域，値域をデータ変換式の形に書いてみるとよい．ただし変数は用いずデータ型をすべてそのままデータスキーマ結合子のオペランドとする．この例の場合，

C("start", "stop". IC("userid", "procs"))*

* データスキーマ $C(IC(s))$ との単一化に成功した時点で "procs" の展開は抑制される．

$\Rightarrow$ IC("userid", "nproc", "ntrap")

となる．ここで s に "userid" を代入し，上述の規則に従って他のオペランドを無視すると上のデータ変換式と単一化される．この変換式に対応する関数スキーマは

$A(C(s))$. ＊1

である．＊1 は連接構造のデータスキーマからその要素をとりだすスキーマ化されたセレクタ関数である．関数変数への同時代入とともに一部整形を施すと

A([userid, p_nproc, p_ntrap]). users*

のように特殊化される．

3.4 関数スキーマベースの構成と検索

関数スキーマベースは以下の規則に従って構成，検索される．

● データ変換式の値域側データスキーマに現れる最左結合子の種類によって四つの1次ブロックが構成される．したがって未設計の関数の値域の最も外側にあるデータ結合子を調べて1次ブロックのいずれかを選択し検索する．

● ワイルドオペレータである I^*, X を含むデータ変換式に対しては2次ブロックが構成される．そのような変換式とマッチした場合は2次ブロックのほうに移って検索を続ける．

● 各ブロック内は順次検索である．したがってデータ変換式は先のものが後のものを含まないように配列される．

三つめの規則に使われているデータ変換式の包含関係については次のとおりである．前節に述べた単一化

* 図2は設計過程が一通り終了した後にさらに整形を行ったもので，この p_nproc, p_ntrap と図2のそれらとは若干異なっている．

```
  /***** Primary Block 1  range C *****/
1 C(a,II*C(s)) => C(a,II*C(s))  a == IC(s)      /* GOTO Secondary Block UPDATE */
2 C(S(a,c),S(b,d)) => C(a,b)                    S(is_a.*1,S(is_b.*2,C(a,b),nil),nil)
3 X => C(a)                                     C(p_a)
  /***** Primary Block 2  range I *****/
4 C(a,II*C(s)) => a  a == IC(s)                 /* GOTO Secondary Block UPDATE */
5 C(IC(s),II*C(s)) => II*C(s)                   /* GOTO Secondary Block COLLATING */
6 C(IC(I*C'(s),a),I*C'(s)) => I(a)              /* GOTO Secondary Block SEARCH */
7 C(IC(I*C'(s),I*C'(t)),IC(I*C'(t),a),I*C'(s)) => I(a)
                                                /* GOTO Secondary Block SEARCH */
8 C(IC(s)) => IC(s)                             A(C(s)).*1
9 C(IIC(s)) => IC(s)                            A(C(s.head)).*1
10 I => I                                       A(f)
  /***** Primary Block 3  range S *****/
11 C(S(a,s),S(b,s)) => S(c,d)  s = lost()       S(is_a.*1,S(is_b.*2,p_c,p_d),p_d)
12 S(a,b) => S(c,d)                             S(is_a,p_c,p_d)
  /***** Primary Block 4  range sort *****/
13 X(a) => s  s = cnt(a)                        /* GOTO Secondary Block COUNT */
14 X(a) => s  s = sum(s)                        /* GOTO Secondary Block TOTAL */
15 IC(s) => s                                   s.head
16 I => s                                       R(f,g)
17 S(a,b) => s                                  S(is_a,f,g)
18 C'(s) => s                                   s
19 X => s  s = cnst()                           s
  /***** Secondary Block COUNT *****/
20 C'I(a) => s  s = cnt(a)                      CNT().*1
21 C'IC'I(a) => s  s = cnt(a)                   SUM().A(CNT().*1).*1
22 C'IC'S(a) => s  s = cnt(a)                   CNT().A(S(is_a.*1,id,nil)).*1
23 C'IC'S(a) => s  a == I(b)  s = cnt(b)        SUM().A(CNT()).A(S(is_a.*1,id,nil)).*1
  /***** Secondary Block TOTAL *****/
24 C'IC'(s) => t  t = sum(s)                    SUM(s).*1
25 C'IC'IC'(s) => t  t = sum(s)                 SUM().A(SUM(s).*1).*1
  /***** Secondary Block SEARCH *****/
26 C(IC(s,a),s) => I(a)                         SEAR(eq)
27 C(IC(I(s),a),s) => I(a)                      SEAR(member)
28 C(IC(s,a),I(s)) => I(a)                      debl.A(SEAR(eq)).D(*2)
29 C(IC(s,t),IC(t,a),s) => I(a)                 debl.A(SEAR(eq)).D(*2).[*2.SEAR(eq).[*1,*3]]
  /***** Secondary Block COLLATING *****/
30 C(IC(s),IC(s)) => IC(s,S(a,b,c))             COLAT(f,g,h)
31 C(IC(s),IC(s)) => IC(s)                      COLAT(f,nil,nil)
32 C(IC(s),IIC(s)) => IC(s,S(a,b,c))            COLAT2(f,g,h)
33 C(IC(s),IIC(s)) => IC(s,I)                   COLAT2(f,nil,nil)
34 C(IC(s),IIC(s)) => IC(s)                     COLAT2(f,nil,nil)
35 C(IC(s),IC(s,I)) => IC(s)                    COLAT(f,nil,nil)
36 C(IC(s),IIIC(s)) => IC(s)                    COLAT3(f,nil,nil)
  /***** Secondary Block UPDATE *****/
37 C(a,IC(s)) => (a,IC(s))  a == IC(s)          UPDTE(f,g)
38 C(a,IC(s)) => a  a == IC(s)                  UPDTA(f,g)
39 C(a,IIC(s)) => (a,IC(s))  a == IC(s)         UPDTE2(f,g)
40 C(a,IIC(s)) => a  a == IC(s)                 UPDTA2(f,g)
```

図 4　関数スキーマベースの内容の抜粋
Fig. 4　Part of function schema base.

```
main: "domain" => "range"
  == f2.f1
f1: "domain" => ["absfile","ixs"]
  == [absfile,SEAR(eq)(docfile,request)]
f2: ["absfile","ixs"] => "range"
  == debl.A(SEAR(eq)).D(ixs)
/* "ixs" == ["indx"...] */
```

図 5 関数仕様記述例 2
Fig. 5 Example of function specification 2.

は一方がデータ仕様，他方がデータ変換式であって方向性がある．これをデータ変換式相互に用いて，変換式 A の変数に変換式 B の変数を代入し，A に現れない B のオペランドを無視して単一化を行う．もし単一化されれば，A は B を含むと定義する．この関係は半順序であり互いに他を含まないものが多い．

この関数スキーマベースの構成および検索手順も設計作業における人間の思考過程を反映したものである．

関数スキーマベースの内容の抜粋を**図 4** に示す．関数スキーマベースには検索キーとしてのデータ変換式，関数スキーマ，および図には示していないが関数スキーマに現れる関数変数の定義域，値域を定める情報も蓄えられる．

以下関数スキーマベースを用いた設計の例をいくつか述べる．まず図 2 の関数 p-rec の第 3 オペランドとなる関数をとりあげる．その値域 "ntrap" はソートであるから 4 番目の 1 次ブロックが検索される．定義域，値域は次のように書ける．

C("userid", IC("pid", S("norm", "trap")))
=〉 "ntrap"
"ntrap" = cnt ("trap")

したがって

s←"ntrap", a←"trap"

の代入により図 4 のデータ変換式 13 と単一化され 2 次ブロック COUNT に検索が移る．さらに検索を続けると変換式 22 と単一化され，対応する関数スキーマがとりだされる．特殊化すると

CNT( ). A({is-trap. term -〉 id ; nil}). procs*

となり，整形すれば図 2 のようになる．p-rec の第 2 オペランドとなる関数についても同様で，変換式 20 に対応する関数スキーマが用いられる．

次に図 3 のデータ仕様を変換式の形に書くと

C(IC("title", "docdata", I("keyword"), "indx"),
IC("indx", "abstract"), I("keyword"))
=〉 I("abstract")

であり，次の代入

s←"keyword", t←"indx", a←"abstract"

により，図 4 の 7 を経て 29 のデータ変換式と単一化される．得られた関数スキーマを特殊化し，整形すると**図 5** に示す関数仕様となる*.

図 6 にデータ仕様を示す例題[4] は顧客マスタファイルと取引項目ファイルを参照し，取引きに関する検査ファイルとともに送り状を作成するものである．**図 7** に設計済みの関数仕様を示す．図 4 の 3 によって main が導かれた後 s に "cnumber" を代入すると，

```
"domain" == ["cusfile","itemfile"]
  "cusfile" == ["customer"...]
    "customer" == ["cnumber","cname","cadrs"]
  "itemfile" == ["cusgroup"...]
    "cusgroup" == ["item"...]
      "item" == ["cnumber","date","inumber","bill"]
"range" == ["invoices","diags"]
  "invoices" == ["invoice"...]
  "invoice" == ["cname","cadrs","cnumber","lines"]
    "lines" == ["line"...]
    "line" == ["date","bill"]
  "diags" == ["diag"...]
  "diag" == ["cnumber","note"]
    "note" == {"howmany","none","error"}
```

図 6 データ仕様記述例 3
Fig. 6 Example of data specification 3.

```
main: "domain" => "range"
  == [p_invoices,p_diags]
p_invoices: "domain" => "invoices"
  == COLAT2(f1,nil,nil)
f1: ["customer","cusgroup"] => "invoices"
  == [cname.customer,cadrs.customer,
      cnumber.customer,p_lines.cusgroup]
p_lines: "cusgroup" => lines"
  == A(p_line)
p_line: "item" => "line"
  == [date,bill]
p_diags: "domain" => "diags"
  == COLAT2(f2,f3,f4)
f2: ["customer","cusgroup"] => "diag"
  == [cnumber.customer,p_howmany]
f3: "customer" => "diag"
  == [cnumber,p_none]
f4: "cusgroup" => "diag"
  == [cnumber.head,p_error]
```

図 7 関数仕様記述例 3
Fig. 7 Example of function specification 3.

---

* 関数スキーマの変数の接頭辞 p_ や is_ および id, nil などの原始関数は，特殊化の際リテラルとして扱われる．

* SEAR(f) の定義は
{null. ＊1 -〉 nil;
f(＊1. head. ＊1, ＊2)
-〉 cons(＊2. head. ＊1, SEAR(f).
[tail. ＊1, ＊2]);
SEAR (f). [tail. ＊1, ＊2]}
また D(＊n) は連接構造の n 番目の反復構造を展開する関数，debl は二重反復構造を 1 本の列にする関数である．

5を経由し34，32のデータ変換式との単一化が可能となってそれぞれ p-invoices, p-diags が得られる．COLAT 2は1対多の照合処理を行う関数結合子で，三つのオペランド関数は両方のファイルで合致するレコードおよびいずれか一方にしかないレコードに対してそれぞれ起動される．それらのオペランド関数は関数スキーマベースに含まれる情報をもとにその定義域，値域が指定され，未設計の関数として新たに設計される．このことは3や10についても同様である．f1 から f4 はいずれもまず図4の 3により設計を経た後，f1の第1，第2，第3オペランドおよび f2，f3 の第1オペランドは18によって導かれる．また f1 の第4オペランドは8，f4の第1オペランドは15によって導かれる．

4. おわりに

ソフトウェア設計の自動化のために形式化，単純化された設計工程において，データとそれに対する処理(関数) の対応規則を定式化およびデータベース化する方法について述べた．関数スキーマベースは標準部品としての関数スキーマを蓄えるものであるが，それを与えられたデータ仕様に従って検索する整った規則をもつことがとくに重要な点である．またその規則は設計作業における設計者やプログラマの思考法を定式化したものであり，ソフトウェア CAD システムへの応用が期待できる．データおよび関数をともに結合子という構成作用素 (forming operator)[3] を用いて形式的に記述することによってこれらの定式化を可能なものにしている．

最後に，関数スキーマベースの構成と利用について解決の残されている検討課題を以下に列挙する．

- 3章のはじめに述べた部品化と再利用に関する課題のうち，(I)の(2)については本論文では触れなかった．既存の設計仕様からスキーマを自動的に抽出できればさらに強力な CAD システムが構成できる．
- データ仕様とデータ変換式との単一化において複数の代入が可能である場合の規則の定式化を，単一化作用素間の関係として形式化すること．
- どの変換式とも完全には単一化できなかった場合の類似部品の選択に関して，データ仕様とデータ変換式との最大マッチングを定義すること．

これらの課題の解決にはデータ変換式の数学的性質を明らかにすることが必要である．それとともに，実際の問題の出現傾向がデータ変換式の集合のなかでどのような位置を占めるのかが興味深い問題である．

参考文献

1) Ajisaka, T., Agusa, K. and Ohno, Y.: Integral Software Development through a Functional Language, Proc. of International Symposium on Current Issues of Requirements Engineering Environments, pp. 33-38, Ohm/North-Holland, Tokyo/Amsterdam (1982).
2) Backus, J.: Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, *Comm. ACM*, Vol. 21, No. 8, pp. 613-641 (1978).
3) Backus, J.: Function Level Programs as Mathematical Objects, Proc. of the 1981 Conference on Functional Programming Languages and Computer Architecture, ACM, pp. 1-10 (1981).
4) Jackson, M. A.: *Principles of Program Design*, Academic Press, London (1975).
5) Myers, G. J.: *Composite/Structured Design*, Van Nostrand Reinhold, New York (1978).
6) Ajisaka, T., Agusa, K. and Ohno, Y.: A Combinatory Language for Computer Aided Software Design, Workshop Notes of International Workshop on Models and Languages for Software Specification and Design, pp. 132-134, Université Laval, Québec (1984).

(昭和59年6月18日受付)
(昭和59年9月20日採録)