# 仕様変更プロセスの効果的な再利用 ── まね方をまねる ──

松浦 佐江子[†,☆] 本位田 真一[†,☆☆]

われわれの研究目的は仕様が変更された場合に，プログラムの作成プロセスを利用して既知のプログラムを修正し，変更要求を満たすプログラムを作成する方法を確立することである．われわれは広範囲言語 Extended ML を用いた仕様・プログラム・プロセスを統一的に扱う枠組の上で仕様変更プロセスを形式化し，系統的な再利用方法を提案してきた．仕様変更プロセスは既知のモジュールから仕様変更の要求を満たす新しいモジュールを作成する仕様の差分定義プロセスと，プログラムを作成した過程である合成プロセスをまねるプロセスから成る．しかし，仕様変更は各々が独立に行われた複数の経験によって達成されることが多いので，一つのプログラム作成のプロセスをその都度利用するだけでは再利用の効率が良くない．また，過去に行った仕様変更と同様な仕様変更を行ってプログラムを得たい場合もある．本稿ではわれわれの提案した再利用方法を拡張し，複数の仕様変更プロセスを効果的に利用した仕様変更の実現方法を提案する．このような知識の積み重ねを支援するソフトウェア開発環境を構築することによって，仕様変更に追従するプログラムの自動合成への道が開けると考える．

# Effective Reuse of Specification Change Processes — Imitating the Imitating Process —

SAEKO MATSUURA[†,☆] and SHINICHI HONIDEN[†,☆☆]

Our goal is to formalize the program development method so that it may be incorporated into automatic and intelligent programming environment. In this environment, a program can be altered to meet the specification change using a program deriving process from the specification. We have proposed a systematic method for reuse on a single unified framework where specification, process and program could be expressed without ambiguities by a wide spectrum language Extended ML. On this framework, a specification change process was formulated, so that it could be manipulated easily and concisely. It consists of a difference defining process and an imitating process on the program deriving process. Specification change is often expressed by a collection of several known experiences. It seems that it is inefficient to imitate a single process every time the specifications are changed though we had an experience of imitating it. In this paper, we extend our framework for reuse so that it can realize more effective reuse of good programming experience. On the extended framework, we propose a procedure for reusing not only one specification change process but also an effective composition of processes. This procedure leads to automatic synthesis of a program meeting the specification change.

## 1. はじめに

われわれは仕様が変更された時に仕様からプログラムが作成されたプロセスを利用して，既存のプログラムを修正する手法の確立を目指している．仕様からプログラムを作成するプロセスは，設計者のプログラム作成の意図を表す具体的な事例である．文献 9) において，このような事例を仕様変更要求に適合するように適切に修正し利用する方法を Standard ML[10] の拡張言語である広範囲言語 Extended ML[13] を用いて提案した．本手法によって，つぎの問題を解決することができた．

(1) 仕様変更要求を満たすように既知の仕様を加工する手順をモジュール関係として分類し定義することによって，プログラムを再利用できる根

† 情報処理振興事業協会（IPA）新ソフトウェア構造化モデル研究本部
Information-technology Promotion Agency (IPA)
☆ 現在，(株) 管理工学研究所より出向
Presently with Kanri Kogaku Kenkyusyo, Ltd.
☆☆ 現在，(株) 東芝 システム・ソフトウェア生産技術研究所
Presently with Systems and Software Engineering Laboratory, Toshiba Corporation

拠を明らかにした．

(2) 仕様からプログラムを作成するプロセスを**合成プロセス**と呼び，これをモジュール関係に従って系統的に再利用する手順を定義した．この結果，仕様および合成プロセスといった設計者のプログラム作成意図を保存したまま，仕様変更要求を満たすように既存のプログラムを修正することができた．

しかし，既存の仕様やプログラムを広範囲に効率良く，しかもプログラム開発の過程において無理なく再利用するためには，つぎのような問題が残されている．

- 通常の仕様変更は仕様間の差異が一つのモジュール関係で表されるのではなく，複数のモジュール関係によって定義される．また，仕様変更要求は一度期に与えられるものではない．そこで一つの合成プロセスをまねる時には，必ずプログラムを部分的に再合成しなければならないので，個々のモジュール関係ごとに順次合成プロセスをまねていたのでは効率が良くない．
- 経験を積むと，過去に経験した仕様変更と同様な仕様変更を行ってプログラムを作成したいこともある．

われわれはこの問題を解決するために，つぎのようにアプローチする．

- 本手法では，プログラムを再利用するためにプログラムを対象とした合成するという行為の手順を再利用している．これを**合成プロセスをまねる**と呼ぶ．同様に，合成の手順を再利用するために，合成手順を加工する行為の手順，すなわち合成プロセスのまね方を再利用することも考えられる．本稿では，これを**まね方をまねる**と呼び，この方法を用いて合成プロセスを効率良く再利用し，複雑な問題に対する解決方法や適用範囲の拡張方法を提案する．

本稿の構成はつぎのとおりである．2章では，本手法の利用手順を簡単な例を用いて説明し，われわれの目的を示す．3章では，われわれの提案する仕様変更におけるまねるメカニズムの定義を復習し，これを拡張する．そして再利用を効率的に行うために，合成プロセスを再利用するプロセスを再利用して，さまざまな仕様変更に対処する方法について説明する．4章では，具体的な事例を用いてまね方をまねる手順を示す．最後に，ソフトウェア開発における再利用性の観点から本手法の有効性を関連研究との比較によって議論する．さらに本手法の適用範囲に関する問題点ならびに今後の課題を述べる．

## 2. 目的の例示

本稿の目的を説明するために仕様記述の例として有名な図書貸し出しシステムの例題[12]を用いる．本稿では仕様変更について考察するので，文献12)で列挙された機能は定義されているものと仮定し，「貸し出し」機能に着目して議論する．ここで「貸し出し」機能の仕様は以下のとおりである．

□仕様

- 利用者が記入した貸し出し票に該当する図書が貸し出し可能であれば，システムは貸し出し票を受理し，貸し出しデータを更新する．
- 貸し出しが不可能であれば，予約リストに貸し出し票のデータを登録する．

われわれは本手法の一部をHyper Cardを使って実現した．**図1**は本手法を用いてシステムを開発する時の設計者の利用手順を示している．オリジナルシステムの開発およびそれに対する仕様変更時における本手法の利用手順を以下に説明する．ここで図内の番号と以下の説明の番号は対応している．

(1) 設計者はA図書館の依頼により，付録A.1に示す図書貸し出しシステムの形式仕様を作成した．複数のモジュールが定義され，各定義はカードに保存される．つぎに設計者は本システムに用意された推論規則やモジュールの定義等を選択し，プログラムを合成する．設計者の各操作列がカードの列として記録される．これが「合成プロセス」である．

同じ仕様の図書貸し出しシステム作成をB図書館からも依頼されたので，設計者はA図書館用に開発したシステムをB図書館にも提供した．

(2) A図書館では利用者へのサービスとして，ネットワークを利用して近郊の図書館にある本も貸し出すことにした．この変更を**広域サービス化**と呼ぶ．A図書館はこの変更を設計者に依頼した．そこで，設計者は本システムに用意されたモジュールを操作するコマンドを使い，変更したいモジュール内の式を選択したり，新規の式を入力して新しいモジュールを作成する（付録A.3は広域サービス化されたシステムの設計仕様の変更点である．）．各コマンドは「モジュール関係」を定義するためのものであり，各操作が仕様のまね方としてカードに記録される．これが図1における「まね方」である．つづいて新しいモジュールにおいて，元のモジュールのプログラムの合成プロセスをまねてプログラム

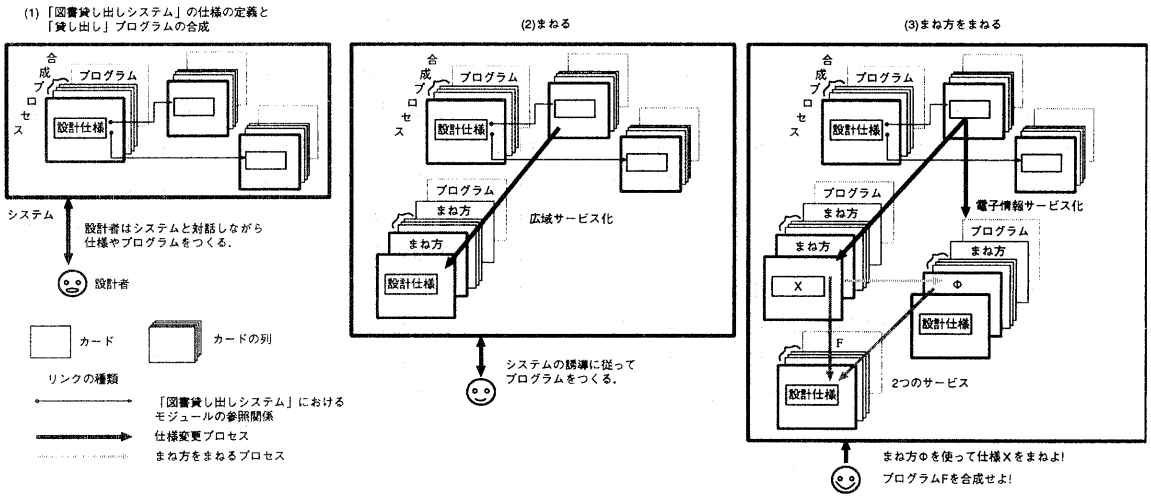



図 1 本手法の利用手順
Fig. 1 How to use our method.

を合成する．設計者は本システムの誘導に従って上記の「まね方」を使って (1) で蓄積されたプログラムの合成プロセスを修正し，目的のプログラムを得る．この操作も「まね方」としてカードに記録される．

(3) 一方，B 図書館では本だけでなく，電子化された文書も情報として提供するサービスを始めることにした．この変更を**電子情報サービス化**と呼ぶ．設計者はこの変更も依頼されたので，(2) の方法と同様にしてシステムを修正した（付録 A.4 は電子情報サービス化されたシステムの設計仕様の変更点である.）．

しばらくして，A 図書館も電子情報サービス化を行うことになり，設計者にこの変更を依頼してきた．これまでの経緯から本システムには合成プロセスやまね方等のさまざまな仕様変更の知識が蓄積されている．そこで，設計者が二つの要求（広域サービス化と電子情報サービス化）を満たすシステムを作りたい場合には，つぎのように本システムを操作すればよい．まず変更したいモジュール X（ここでは広域サービス化の仕様）と変更方法であるまね方 Φ（ここでは電子情報サービス化のまね方）を指定する．これによって二つの要求を示す新しいモジュールがつくられる．つぎに，生成された新しいモジュールに対し合成コマンドを起動すれば，まね方をまねて目的のプログラムが合成される．

ここで重要なことは，システムを電子情報サービス化する場合に広域サービス化された図書貸し出しシステムの合成プロセスを単に加工して再利用するのではなく，電子情報サービス化した時の手順を再利用して加工することである．ここでは，どのように加工すれば良いかが既知の情報であり，再利用はこの分効率良く行われる．

## 3. まねるメカニズム

本章では，はじめに本手法の拡張における着想を説明し，つづいて仕様変更におけるまねるメカニズムについて説明する．

### 3.1 メタな思考

人間の思考形態は，対象とそれに対する行為という観点において，**図 2** のように層状になっていると考えられる．図 2 において，各層上の四角形の上部は行為を，下部はその行為の対象を表している．矢線 '**Execute**' が示すように，上部の行為が下部の対象に対して行われる．そして，その行為の履歴が記録される（矢線 '**Record**'）．一方，矢線 '**Represent**' は，記録された履歴をもとに行為が行われることを表している．すなわち，**Layer i+1** は **Layer i** に対してメタな思考を行うところであると考える．

本システムにおいて設計仕様からプログラムを合成するプロセスをこの枠組において考えてみよう．以下の番号は図中の番号と対応する．

① 設計仕様から一般的な推論規則やプログラム作成の戦略に従ってプログラムを合成する．設計者が **Layer 1** において，システムの規定する手段を用いて合成プロセスを定義する．結果の合成プロセスが **Layer 2** に記録される．

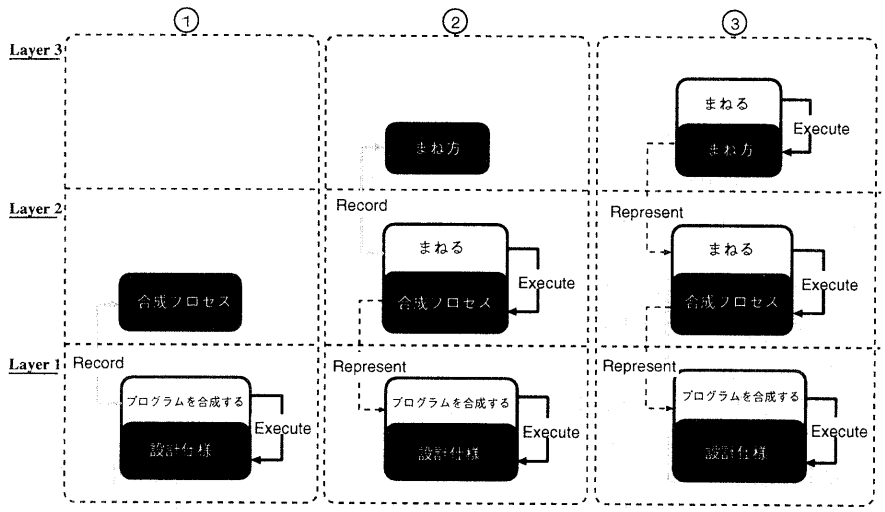



図2 プログラム合成のプロセス
Fig. 2 Program synthesis process.

② 既存の合成プロセスをまねた結果の合成プロセスを使って設計仕様からプログラムを合成する．設計者が **Layer 2** において既存の合成プロセスをまねることによって，目的の合成プロセスを半自動的に定義する．合成プロセスのまね方が **Layer 3** に記録される．

③ 合成プロセスのまね方をまねて，既存の合成プロセスをまねる．得られた合成プロセスを用いて設計仕様からプログラムが合成される．**Layer 3** においてまね方をまねることによって既存の合成プロセスが自動的に変更される．

① の場合には人間が推論規則を使って一からプログラムを合成しなければならない．② の場合には，① で入念に作られた合成プロセスというプログラム作成の事例を仕様の変更手続きに基づき修正する．これによって，仕様変更要求を満たすプログラムを半自動的に合成することができる．これが文献 9) で定義した**仕様変更プロセス**である．このような枠組を想定すると，単に対象をまねるだけでなく，③ のように何かをまねた手順をまねる対象として考えることも自然にできる．

文献 9) では上記 ① によって作成された知識を ② のメカニズムによって利用することを提案した．本稿では ③ のように，その利用履歴も新たな知識として利用することを提案する．以下において，各 ①〜③ プロセスの構成，獲得方法およびまねられる根拠について説明する．

### 3.2 設計仕様・プログラム・合成プロセス

まねるメカニズムを具体化するための基礎知識は

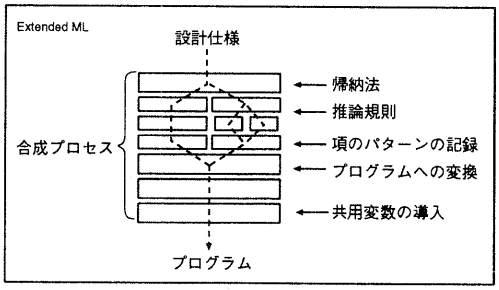



図3 設計仕様・プログラム・合成プロセス
Fig. 3 Specification · Program · Synthesis process.

3.1 節 ① のプロセスでつくられる．本稿ではこれを**図 3** のようなプログラムが設計仕様から合成される体系として Extended ML で定義する．

**設計仕様**は，Extended ML のモジュールを用いて定義される．Extended ML のモジュールはインタフェース部と実現部から構成される．インタフェース部は型・変数の宣言・それらに対する操作の仕様から構成される．ここで操作の仕様は一階述語論理における公理の集合である．一方，実現部は型の定義およびインタフェースに定義された操作を実現する関数から構成される．本稿では実現部の型の定義を**データ仕様**，各操作を**アクション**，インタフェース部の操作の仕様を**アクション仕様**と呼ぶ．付録 A.1 は Extended ML で記述された図書貸し出しシステムの設計仕様の一部である．設計仕様の構文の説明をコメント☆に記す．

Extended ML は一階述語論理に基づく仕様記述言

☆ 以下付録の記述において，(* ... *) はコメントを表す．

表1 仕様変更プロセスの形式化
Table 1 Formulation of specification change process.

| プロセス | 対象 | 根拠 | 手段 |
|---|---|---|---|
| 仕様の差分定義プロセス | 設計仕様 | モジュール関係 | モジュール操作 |
| 合成プロセスをまねるプロセス | 合成プロセス | 仕様のまね方 | プロセス操作 |

語であり，自然演繹法[5]の推論規則を用いてサブセットである実行可能なMLのプログラムを導出することができる．ここで一階述語論理における公理への推論規則の適用等によってアクション仕様からそのプログラムを導出する過程がプログラムの**合成**であり，その履歴が**合成プロセス**である．合成プロセスはつぎに示す操作をアクション仕様に適用し，書き換えられた式を返すサブプロセスから構成される．設計者が以下の操作および操作対象を指定し，さらに必要な項目を与えて合成プロセスを定義する．そこで合成プロセスはこれらの操作をいかに組み合わせてプログラムを導出したかという設計者のプログラム作成意図を表している．図3の各四角形がサブプロセスを表している．そして点線の手順でプログラムに到達する．

- 帰納的定義の仮定を導入する等の合成の戦略を決定する．
- and の導入・除去，限定子の導入・除去等，自然演繹法における推論規則を公理に適用する．付録A.2のコメント (* 1... *) を参照のこと．
- 論理式をλ式に対応付ける．設計者は推論規則を用いて公理を書き換え，制御構造をもつプログラムを導出する．公理をプログラムの変換規則が適用できる段階まで書き換えた時に成立する項のパターンを記録する．これを**パターン環境**と呼ぶ．付録A.2のコメント (* 2... *) を参照のこと．
- if や while 文の導入等，プログラムへの変換規則を式に適用する．
- 共用変数を導入する☆．

設計者はこれらのサブプロセスを履歴として記録しながら，MLのプログラムが得られるまでこれを繰り返す．付録A.1における `fun` で定義された `lend` は合成の結果得られたML記述のプログラムである．

### 3.3 まねるプロセス

#### 3.3.1 仕様変更プロセスの形式化

仕様変更プロセスは，既知の仕様からモジュール関係を使って新たな要求を満たす設計仕様を定義するプロセスと，既知のプログラムの合成プロセスを加工して変更された設計仕様を満たすプログラムを合成するプロセスから成る．前者を**仕様の差分定義プロセス**，後者の合成プロセスを加工するプロセスを合成プロセスを**まねるプロセス**と呼ぶ．3.2節の基礎知識に基づき，仕様変更プロセスを図4のように定義する．

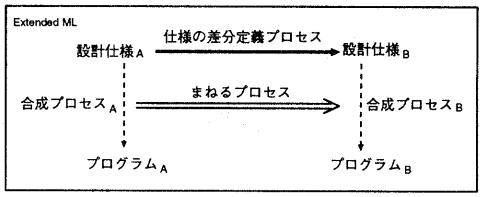



図4 仕様変更プロセス
Fig. 4 Specification change process.

まねるということは，つぎのように形式的に定義される．

まねるとは，まねる対象を，ある根拠を利用して，ある手段によって加工することである．

そこで，図4の枠組において，二つの仕様変更のプロセスは**表1**のように形式化される．すなわち，各プロセスは各々の根拠を利用して，モジュール操作・プロセス操作といった手段を使って対象を加工する手順を定めるものである．

**モジュール操作**とはモジュール関係により設計仕様を変更する手段であり，データ変換・公理の追加や削除・公理前提部への論理式の追加・公理前提部の論理式における部分式の置き換え・公理帰結部の主機能定義式における右辺の式の置き換え等のML記述のモジュール操作関数の組合せで定義される．

一方，**プロセス操作**とは3.3.3項で述べる仕様のまね方により合成プロセスを変更する手段であり，モジュール操作関数の適用・推論規則等の適用・パターン環境の変更・サブプロセスの操作といったML記述のプロセス操作関数の組合せで定義される．各操作関数の入力は，対象を各々の形式で構造化して表現したMLのデータである．

#### 3.3.2 まねる根拠

まねるための根拠はまねる対象間の類似性と考えられる．類推の研究[1),2),11)]においても指摘されているように，類似性を定義する場合には対象としている問題（本稿においてはプログラムの再利用）において何に着目して「似ている」という価値を定義するかを明確にしなければならない．本稿においては設計仕様間の

☆「貸し出し（lend）」の設計仕様において変数 library は lend の入力変数であったが，これを付録A.1のプログラムのように共用変数とした．

類似性および合成プロセス間の類似性をつぎのように考える．

本稿では仕様変更を既知の設計仕様からの変化として定義したので，不変の部分が二つの設計仕様の共通部分である．モジュールのデータ仕様およびアクション仕様の変更を定義するモジュール間の関係を**モジュール関係**と呼ぶ．モジュール関係には，継承関係（データの直積直列・直積並列・直和関係）・データ拡張・構造拡張の5種類がある☆．これが本稿における二つの設計仕様間の類似性を定めている．仕様変更を一般的に分類することは難しいが，仕様変更の意味は一般・個体，上位・下位，単純・複合等の対象の概念関係としても考えることができ，モジュール関係はこのような意味における分類名称の一つである．例えば図書貸し出しシステムの広域サービス化を考える．これは直観的にはアクションの対象領域が拡張される変更である．まずネットワーク上の個々のシステムを識別するために識別子データを既存の図書館データと直積関係に追加する．つぎにアクション「貸し出し」の対象を一つの図書館データから，複数の同種の図書館データの集合に拡張する．このようなデータ仕様の変更に伴い，アクション「貸し出し」は貸し出し時の条件が他の図書館での貸し出し状況に依存するように変更される．この変更は「貸し出し」の公理の前提部に他の図書館での貸し出し状況を定義する新たな論理式を論理記号の操作によって導入することによって行われる．すなわち，本稿における設計仕様の類似性は仕様変更の意味を形式仕様上での意味（型の関係および論理式の関係）によって表現した類似性であり，モジュール操作関数による具体的な手続きとして表現される．

つぎに合成プロセス間の類似性について考える．設計仕様および合成プロセスは同じ記号から生成される共通の項をもつ．合成プロセスは設計仕様に現れる項以外には合成の過程において導入される自然演繹の推論規則，MLの予約語，プロセス操作の予約語，新たに定義された項およびそれによって置換された項のみを含む．推論規則等の変換が項の関係を定義している．一方，モジュール操作関数は変更対象と変更点および新しい項を入力として変更された設計仕様を出力する関数として定義されている．これは設計仕様内の項の変更（データ仕様の変更）および項の関係の変更（アクション仕様の変更）を定義している．変更点は設計仕様内の項およびモジュール関係を表す定数などである．そこで設計仕様の変更に伴い，合成プロセス内の項や項の関係が変更される．例えば上記の「貸し出し」の変更に対して，合成プロセス上では1ステップのサブプロセスの入力が論理式の積に変わり，さらに追加された論理式を推論する必要が生じる．その他のサブプロセスは不変であり，「貸し出し」公理の推論過程における部分問題の推論過程，帰納的定義の過程，制御構造の導入，共用変数の導入の仕方等設計者がはじめに「貸し出し」を定義した意図は保存されている．

### 3.3.3 ま ね 方

まねるの形式的定義から，モジュール操作やプロセス操作によるまねる動作は変更点の値（項や定数等）とまねる対象を入力とし，変更された対象を出力する各操作関数の組合せで定義される．そこで仕様変更プロセスにおけるまね方はつぎのように定義される．

**[定義 1]** 【まね方】

$$\left\{\begin{array}{c}\text{設計仕様}\\ \text{合成プロセス}\end{array}\right\}\ \text{のまね方} \equiv$$

$$(\text{apply}\ \left\{\begin{array}{c}\text{モジュール操作関数}\\ \text{プロセス操作関数}\end{array}\right\}\ \text{変更点の値})$$

の形式の項から生成されるML記述の項

ただし，例外処理式は除くものとする．ここで「apply」は関数を値に適用することを表す． □

3.3.2 項で述べたとおり，設計仕様のまね方が合成プロセスを修正する根拠となっている．

### 3.4 まね方をまねるプロセス

### 3.4.1 仕様変更プロセスの拡張

3.1 節における考察より，3.3 節で述べた仕様変更プロセスの枠組は**図5**のように拡張される．すなわち，これは図4が表す知識を再利用する枠組である．ここで問題となるのは複雑化した知識をどのような条件下で利用できるかということであり，この条件がソフトウェア開発において妥当であるかを考えなければ

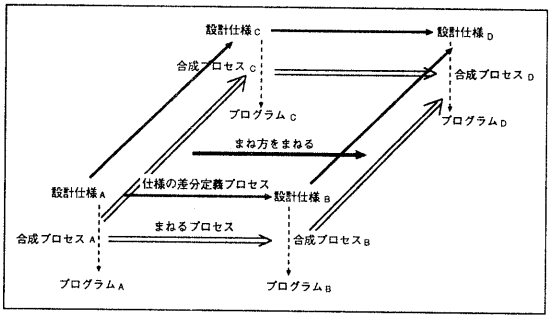



**図5** 仕様変更プロセスの拡張
Fig. 5 Extension of specification change process.

☆ 付録A.3およびA.4のコメントを参照のこと．

ならない．定義1のまね方は設計仕様（合成プロセス）を入力として設計仕様（合成プロセス）を出力する関数である．すなわち，変更点の値はまね方が定義された入力の設計仕様に依存する．このような具体的なまね方を別の設計仕様に対して利用できるようにすることがわれわれの目的であり，これがまね方をまねるということである．さらに，図5のまね方をまねるプロセスが表1の観点でどのように形式化できるかを考察する．

### 3.4.2 まね方をまねる根拠

システムに対する要求は一時にすべてが提出されるわけではない．むしろ，要求は五月雨式に設計者に与えられる．すなわち2章の例で示したように，仕様変更要求のシステムへの反映は線形ではなく並行に行われると考えるのが自然である．例えば，オリジナルの図書貸し出しシステムに対する仕様変更要求は2章の例の他にも，「単行本以外に定期刊行物も貸し出したい」・「1人の利用者に対する貸し出し数は制限したい」といったことが考えられる．これらの要求はすべてアクション「貸し出し」に対する要求であるが互いに独立して解決することができる．そこで本手法を利用するために，つぎのようなソフトウェアの開発形態を前提とする．

まずシステムは基本的な要求を満たす核となるシステムからつくりはじめ，並行開発の可能な要求に対してはそれぞれ独自に変更を行う．ここで二つの異なる開発が並行であるとは，一方のまね方における変更点の論理式が他方のまね方により変更されたアクションの公理の集合全体に対して矛盾を生じないということである☆．このようなまね方を独立なまね方と呼ぶ．

複雑な問題でも核になる部分を抽出したり，要求を並行開発できるように分離することは可能である．その上で二つの開発要素間の関係を分析し，要求の重なり部分を抽出すればよい．十分理解できる要求間の関係を分析することは設計者にとって理解しやすい作業であり，このような前提は容易なソフトウェア開発にとって妥当であると考える．

### 3.4.3 まね方をまねる条件

3.2節で定義された本手法における知識の基本単位を拡大モジュールと呼び，以下 $Emod_{label}$☆☆と記す．すなわち $Emod_{label}$ は設計仕様・合成プロセス・プログラムで構成され，これらを各々 $Spec_{label}$・$Proc_{label}$・$Prog_{label}$ と記す．

☆ 部分的な矛盾は公理の削除を意味しているので，本手法で扱うことができる．

☆☆ label は識別子である．

3.3節の仕様変更プロセスで獲得されたまね方は基底となる拡大モジュール **B1**，モジュール操作・プロセス操作およびその結果得られる変更された拡大モジュール **B2** の集合であり，**B1**，**B2** の間に成り立つモジュール関係でラベル付けされている．これを $Emod_{B1} \xrightarrow{\{mod\_mp_{12},proc\_mp_{12}\}} Emod_{B2}$ と記す．ここで $mod\_mp_{12}$ と $proc\_mp_{12}$ は，各々仕様のまね方と合成プロセスのまね方を表し，添字は関係が成り立つ拡大モジュールの添字から付記される．

3.4.2項で述べた開発条件を前提とすると，まね方を使って仕様変更を実現するための条件はつぎのように定義できる．

**[定義 2]** 【まね方をまねる条件】

設計仕様 $Spec_{C1}$ に対して，$Spec_{C2} \xrightarrow{\{mod\_mp_{23}\}} Spec_{C3}$ で表される仕様変更を行いたいとする．目的の拡大モジュールを $Emod_{C4}$ とする．さらに，$Spec_{C1}$ は拡大モジュール $Emod_{C1}$ の，$Spec_{C2} \xrightarrow{\{mod\_mp_{23}\}} Spec_{C3}$ はまね方 $Emod_{C2} \xrightarrow{\{mod\_mp_{23},proc\_mp_{23}\}} Emod_{C3}$ の構成要素とする．ここで，$Spec_i$ が $Emod_i$ の構成要素であるとは，$Spec_i$ を満たすプログラムを作成した経験である $Emod_i$ が存在することである．また，$Spec_i \xrightarrow{\{mod\_mp_{ij}\}} Spec_j$ が $Emod_i \xrightarrow{\{mod\_mp_{ij},proc\_mp_{ij}\}} Emod_j$ の構成要素であるとは，ある仕様変更 $Spec_i \xrightarrow{\{mod\_mp_{ij}\}} Spec_j$ に対して合成プロセスをまねてプログラムを合成した経験である $Emod_i \xrightarrow{\{mod\_mp_{ij},proc\_mp_{ij}\}} Emod_j$ が存在することである．この時，つぎの(1)または(2)が成立する時にまね方をまねることができる．

(1) $Spec_{C1} = Spec_{C2}$．この時，目的の拡大モジュールは $Emod_{C3}$ である．ここで＝は同一の記述を意味する．

(2) $Spec_{C1}$ が $Emod_{C1}$ の構成要素であるまね方 $Emod_{C0} \xrightarrow{\{mod\_mp_{01},proc\_mp_{01}\}} Emod_{C1}$ が存在し，$Spec_{C0} = Spec_{C2}$ かつ $mod\_mp_{01}$ と $mod\_mp_{23}$ は独立なまね方である．この二つのまね方を互いに**類似のまね方**と呼ぶ． □

(2)の条件は同じ核をもつ並行な開発が存在することである．この時，$Emod_{C2} \xrightarrow{\{mod\_mp_{23},proc\_mp_{23}\}} Emod_{C3}$ をまねたまね方を $Emod_{C1}$ に適用し，目的の $Emod_{C4}$ を合成する手順を4章で示す．同じ核をもち並行に開発されたモジュールは共通の項と独立したまね方を持っている．そこで，一方の設計仕様あるいは合成プロセスに対し，共通の項をもつように前者のまね方を使って他方のまね方を修正し，修正された

表2 拡張された仕様変更プロセスの形式化
Table 2 Formulation of extended specification change process.

| プロセス | 対象 | 根拠 | 手段 |
|---|---|---|---|
| 仕様のまね方をまねるプロセス | 仕様のまね方 | 類似の仕様のまね方 | モジュール操作 |
| 合成プロセスのまね方をまねるプロセス | 合成プロセスのまね方 | 類似の合成プロセスのまね方 | プロセス操作 |

まね方を $Emod_{C1}$ に適用することができる．以上の考察により，3.3 節で述べた形式化に従うと，拡張された仕様変更プロセスの枠組におけるまね方をまねるプロセスは**表 2** のようになる．

## 4. まね方をまねる

本章では 3 章で述べたまねるメカニズムを Extended ML を用いて具体化する．まず，まね方を定義するための二つのプロセスを示し，つぎにこれらをまねる手順を説明する．説明には 2 章で述べた例を用いる．

### 4.1 仕様の差分定義プロセス

仕様の差分定義プロセスは前述のモジュール操作関数のみを使って，変更対象モジュールを修正する手続きである．本プロセスは ML の関数であるモジュール操作関数を組み合わせた ML の関数として定義される．付録 A.4 の電子情報サービス化の設計仕様は，下記に示す仕様の差分定義プロセスを経て定義される．以下の説明における各小ステップは各々の仕様を定義する際に用いたモジュール操作を説明している．付録 A.3 のコメントにおいて [ ] で囲まれた文も同様であり，`datapattern_trans` 等がモジュール操作関数である．

(1) オリジナルのシステムにおける図書のデータ（`'book'`）は書名（`'title'`）と著者（`'author'`）のフィールドをもつ物理的な本を表すデータである．継承（直和）関係により，一般化された図書のデータ`'book'`に対し，付録 A.4 の `book` のように図書のデータの場合わけである電子情報データを新たな種類のデータとして追加する．つぎのモジュール操作を行う．

(a) オリジナルのデータ仕様（`functor Library`）に対して，データ型の変換を行い `datatype` の定義を変更し，新しいデータ仕様（`functor EIS_Library`）とする．

(2) つぎのモジュール操作で，「貸し出し（`lend`）」の公理を電子情報データにも対処できる公理に変更する．

(a) オリジナルのアクション仕様（`signature LIBRARY`）が項 "`Card {book=Book {title=t, author=a}, id=n, record=rs}`" を含むならば，つぎのデータパターン変換を行う．

(A) データパターン変換

`[datapattern_trans D_series`☆`'Card {book=Book {title=t, author=a}, id=n, record=rs}' 'Card {book=Book {obj=b}, id=n, record=rs}']`

および `[datapattern_trans D_series b 'Obj {title=t,author=a}']`

(B) データパターン変換

`[datapattern_trans D_sum 'Card {book=Book {obj=b},id=n,record=rs}' 'Card {book=Elt {obj=e},id=n,record=rs}']`

(b) (a) で無変換の場合にはアクション仕様の公理に対しつぎの二つの変換を行う．

(A) アクション仕様の公理の前提部で型 `card` の変数 `c` の項 "`Ok c`" を含む論理式が "not 論理式" の形式でなければ，論理式 `#book c=Book {obj=b}`☆☆ を and の関係で追加する．

(B) アクション仕様の公理の前提部で型 `card` の変数 `c` の項 "`Ok c`" を含む論理式が "not 論理式" の形式でなければ，論理式 `#book c=Elt {obj=e}` を and の関係で追加する．

(c) 公理の前提部が論理式 `#book c=Elt {obj=e}` を含み，かつ公理帰結部が項 "`lend tag (Library {rec=cs, reserve=t})`" を含むならば，その論理式右辺におけるタプルの第 2 項を項 "`Copy c charge_recieve`" と置き換える．

上記のプロセスから得られる仕様のまね方はつぎのとおりである．例えば，付録 A.3 のデータパターンの変換：

```
[datapattern_trans Recursive
'Library {id=i,rec=cs,reserve=t}'
'Network {host=Library {id=i,rec=cs,
          reserve=t}, local=ls}' axiom ☆☆☆]
```

は，広域サービス化におけるモジュール操作の一つである．

この時，定義 1 から

```
[datapattern_trans Recursive
```

☆ D_series，D_sum 等はモジュール関係におけるデータ仕様の変更の種類を表す定数である．また，モジュール操作関数内の '...' は設計仕様の項を ML のデータ化したものである．

☆☆ $\#\alpha$ はレコードのフィールド $\alpha$ の値を取り出す関数である．

☆☆☆ axiom は変換対象の公理を表すメタ記号である．

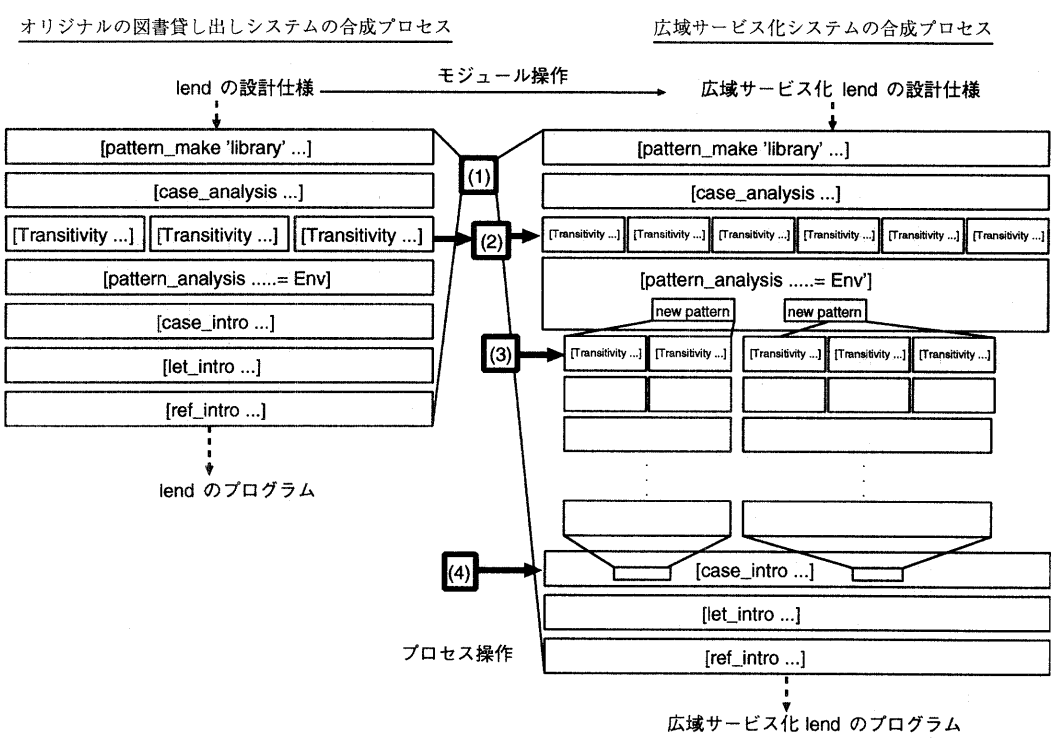



図 6 合成プロセスをまねる
Fig. 6 Imitate a synthesis process.

```
'Library {id=i,rec=cs,reserve=t}'
'Network {host=Library {id=i,rec=cs,
        reserve=t}, local=ls}']
```

がまね方である．この関数をオリジナルの公理に適用すると広域サービス化されたシステムのデータに変換された公理を返す．

上述の仕様の差分定義プロセス (2)-(c) のモジュール操作を

```
[f1 axiom1 =
if exp1 then replace exp2 axiom1 else axiom1]
```

と書くとすると，

[f1] がまね方である．既知の公理に適用すると目的の新しい公理が得られる関数である．ここで，exp1 は条件の式，exp2 は置き換えられる式，axiom1 は対象の公理を表す．replace はここで用いた複数のモジュール操作関数を省略して記述した関数である．モジュール操作が条件を含んでいるので，まね方は単なるデータ変換ではなく対象公理を抽象化した関数になっている．

### 4.2 合成プロセスをまねるプロセス

例として，図書貸し出しシステムのアクション「貸し出し（lend）」の合成プロセス Φ をまねて，広域サービス化されたシステムにおける「貸し出し（lend）」を合成する手順の一部を説明する．4.1 節で説明した仕様のまね方を使って，合成プロセス Φ を修正する．以下の説明で [ ] で囲まれた式がプロセス操作を表す．**図 6** 内の番号は以下の説明に対応する．図 6 における四角形はそれぞれサブプロセスを，[ ] は変更の操作名を表している．

(1) アクション「貸し出し (lend)」のまね方であるデータパターン変換式 datapattern_trans

```
Recursive 'Library {rec=cs,reserve=t}'
'Network {host=Library {id=i,rec=cs,
reserve=t},local=ls}'
```

を Φ に適用する．適用結果の合成プロセスを Φ′ とする．

```
[apply_transexp (datapattern_trans
Recursive 'Library {rec=cs,reserve=t}'
'Network {host=Library {id=i,rec=cs,
        reserve=t}, local=ls}') Φ].
```

(2) オリジナルの公理の前提部が変更されている場合には，これらのまね方をパターン環境に順次適用する．例えば Φ′ における第二公理に対し，付録 A.3 の φ21=[naxiom21=if ...] を適用する．

```
[apply_module_pattern φ21 Φ']
```

(3) 合成プロセスは付録 A.2 に示したようなパターン環境を含んでいるが，上記の変換で新し

いパターン環境が得られる．パターンが変化した公理を再合成する．この場合には，パターン 'A=p2 ⇒ F=E2' がパターン 'A=p2 ⇒ B1 ⇒ F=E2' orelse A=p2 ⇒ B2 ⇒ F=E2''' に変化している．付録A.3における二つの式 "naxiom21" と "naxiom22" がこれに相当する．

```
[synthesis '(A=p2 ⇒ B1 ⇒ F=E2' orelse
A=p2 ⇒ B2 ⇒ F=E2'')'=result]
```

(4) (2) で見つけた部分を再合成した結果のサブプロセスの列 (result) が $\Phi''$ におけるオリジナルのサブプロセスの列と置き換えられる．

```
[change_apply_to_pattern result Φ']
```

### 4.3 まね方をまねる手順

つぎに，定義2 (2) の場合に目的のプログラムを得るための手順を示す．本手順は仕様のまね方をまねて目的の設計仕様を合成するステップと合成プロセスのまね方をまねて目的の合成プロセスおよびプログラムを得るステップから成る．

#### 4.3.1 仕様のまね方をまねる手順

4.1節で定義した電子情報サービス化された図書貸し出しシステムの設計仕様の差分定義プロセスの各ステップは，モジュール操作関数によって定義されている．以下は仕様のまね方である各関数を広域サービス化された図書貸し出しシステムの設計仕様に適用して，両サービスを実現した図書貸し出しシステムの設計仕様を得る手続きである．

(1) データ型変換式 (4.1節 (1)-(a)) を広域サービス化された設計仕様のデータ仕様 (**datatype** 宣言) に適用する．

(2) 付録A.3の「貸し出し (**lend**)」の公理に4.1節 (2)-(a) のデータパターン変換式 (A) および (B) を順次適用する．この場合は条件のパターンを公理が含まないので無変換となる．

(3) つづいて4.1節 (2)-(b) のまね方 (A) および (B) を順次適用する．つぎのような通常の 'book' データに関する公理が生成される．同様に，電子情報 'Elt {obj=e}' データに関する公理も生成される．

```
axiom Ok c=card_search tag cs
      andalso #book c=Book {obj=b}
 ⇒ lend tag (Network {host=Library {id=i,
              rec=cs,reserve=t},local=ls})
   =(Network {host=Library {id=i,
         rec=cards_update cs c tag,
     reserve=t},local=ls}, Lend c)
```

(4) 上記の手続きで変更された「貸し出し (**lend**)」の公理に対し，4.1節の電子情報サービス化のプロセスにおけるまね方 (2)-(c) をまねて適用する．

ここで，まね方をまねるとは，電子情報サービス化された図書貸し出しシステムにおけるまね方を広域サービス化されたシステムの表現と同一視することである．この手続きを**項の整合化**と呼ぶ．つぎの手続きで行われる．

4.1節 (2)-(c) のモジュール操作関数を広域サービス化された図書貸し出しシステム上の関数に変換する．

広域サービス化の二つの仕様のまね方 `datapattern_trans D_parallel...` と `datapattern_trans Recursive ...` を逐次4.1節 (2)-(c) のモジュール操作より得られるまね方に適用する．変換の結果，つぎのまね方が得られる．

[公理の前提部が論理式 `#book c=Elt {obj=e}` を含み，かつ公理帰結部が項 ``lend tag (Network {host=Library {id=i, rec=cs, reserve=t}, local=ls}'' を含むならば，その論理式右辺におけるタプルの第2項を項 ``Copy c charge_recieve'' と置き換える．] 変換されたまね方を (3) で変更された公理に適用する．適用した結果，つぎの公理が得られる．このようにまね方を適用する前に一方の仕様のまね方を他方の仕様のまね方によって修正して利用することが，仕様のまね方をまねるということである☆．

```
axiom Ok c=card_search tag cs
      andalso #book c=Elt {obj=b}
 ⇒ lend tag (Network {host=Library {id=i,
              rec=cs,reserve=t},local=ls})=
(Network {host=Library {id=i,rec=
    cards_update cs c tag, reserve=t},local=ls},
    Copy c charge_recieve)
```

#### 4.3.2 合成プロセスのまね方をまねる手順

4.3.1項の手順で，二つのサービスを行うシステムの設計仕様が作られた．一方，合成プロセスをまねた結果，各々のサービスを行う図書貸し出しシステムにおける「貸し出し (**lend**)」が定義されている．

以下，合成プロセスのまね方をまねる手順を説明する．3.3.3項の定義から，合成プロセスのまね方は仕様のまね方と同様にプロセス操作関数と仕様の差分の具体値から定義されるので，モジュール関係の種類には依存しない．**図7** 内の番号は以下の番号と対応する．

(1) 電子情報サービス化の合成プロセスのまね方に広域サービス化の仕様のまね方を適用して，項の整合化を行う．

(2) 広域サービス化の合成プロセスに電子情報サー

☆ まね方の適用評価時に項をパターンにより抽象化して処理する方法もあるが，紙面の都合上詳細は省略する．

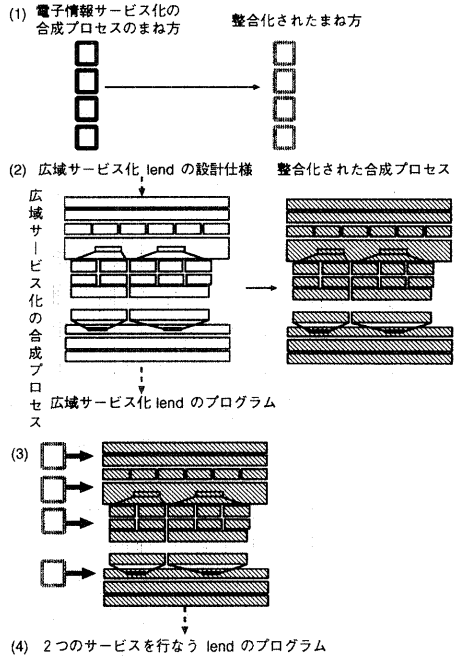



図7 まね方をまねる
Fig. 7 Imitating the imitating plan.

ビス化の仕様のまね方を適用して，項の整合化を行う．

(3) (2) の結果得られた合成プロセスに (1) で得られた合成プロセスのまね方を適用する．

(4) (3) の結果得られた合成プロセスの最後のサブプロセスの出力が，二つのサービス要求を満たす目的のプログラムである．

## 5. 考　　察

### 5.1 再利用性の評価

ソフトウェア開発において再利用する対象はさまざまである[7]．そこで，まずはじめにプログラムの直接の再利用技術の限界と，他のプロセスを利用した再利用の研究について議論する．

- プログラムを直接再利用する技術は，いわゆる高水準言語やソフトウェア部品を生み出してきたが，つぎのような問題がある．

パラメタライズ[3]は，プログラムを再利用できるようにあらかじめ抽象化して作成する技術である．本稿で述べたモジュールを作成する際には，このような抽象化技術が必要である．しかし，あらかじめ想定できないさまざまな仕様変更要求に対応するために，プログラムを十分にパラメータ化することは難しい．

オブジェクト指向[15]における継承もプログラムの直接の再利用を可能とする．しかし，これはあくまでも全く同じ機能として利用できるということである．部分的に異なるメソッドは新たにかつ独自に定義されるので，オブジェクト指向におけるメソッドの再定義は系統的な再利用ではない．

- 一方プログラムの作成において過去の作成経験を再利用することは，人工知能やソフトウェア工学の分野で，数多く提案されている[1),2),4),8),11)]．文献 9) で述べたように，何を目的とした場合に再利用が可能であるかといった再利用の根拠が希薄であり，再利用の方法が系統的ではない．また，これらの研究では一つの対象プロセスを再利用しており，本稿のようにプロセスのプロセスを利用してまね方をまねることは考えていない．

本稿の目的は，文献 9) で提案した再利用手法によって，プログラムの効果的な再利用を行うことである．効果的な再利用とは再利用手法の導入によって解ける問題が増える，結果のプログラムの品質が保証される，ソフトウェア開発において人間の介在が縮小される等の結果が期待されるものである．上記の考察に基づき，このような効果の観点においてわれわれの手法の再利用性を評価する．

まず第一に，3 章で示したようにプログラム開発における経験は豊富にあり，要求に応じて再利用できるレベルが異なる．仕様変更要求においては，「既知の仕様を～のように変える」というだけではなく，例えば，「システムを～化する」というように，変更の方法が要求されることもあるだろう．第一の効果を得るためにも，仕様やプログラム自身の再利用性を高めることと同時に，その基礎の上に立って，それらをつくるプロセス，さらに再利用するプロセスといった開発経験を系統的に定義し利用することが重要であり，本稿で述べた仕様変更プロセスの枠組はその基盤となりうると考える．

第二点は，人間の柔軟な思考を活かすためには，つぎのように抽象レベルの推論と具体レベルの推論を融合して行うことが自然であり，第二・第三の効果が期待される．われわれの提案するプログラム開発の枠組は一階述語論理における自然演繹という演繹方法をもつ．それならば，仕様が変更された時にも毎回演繹によってプログラムを得ればよさそうである．しかし，われわれはつぎの三つの理由から，演繹システムを基礎として 3.1 節 ① の手順で得られた経験を ②，③ の手順でまねる仕組みを推論システムとして定義する．

- 演繹は結果のプログラムの品質を保証することができるが，自動化することは難しい．
- 仕様変更時に類似のプログラムを得るためには，

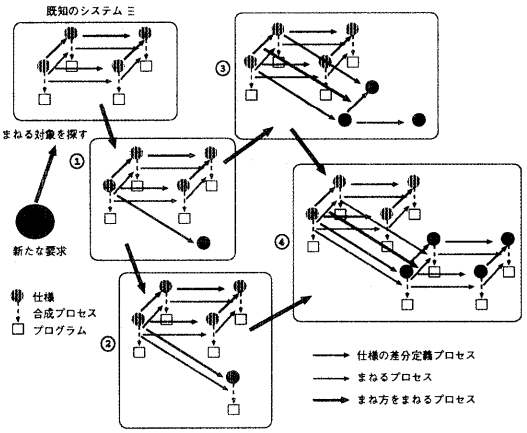



図8 類似のシステム機能の獲得
Fig. 8 Imitating system functions.

仕様をプログラムとしてどのように実現したかという設計者の意図が一貫していることが結果のプログラムの品質や信頼性を保証する上で重要である.

- 人間にとって理解しやすい仕様のレベルにおけるデータ間の類似性・アクションの類似性といったさまざまな関連に着目して,まねることによって人間の介在を減らした容易な開発が可能になる.

### 5.2 適用範囲の問題点

まね方をまねることの適用性について考察する.ある新たな要求に対して,既知のシステムにおける機能拡張と同様の拡張をつぎのように行うことができる.図8中の番号は以下の番号と対応する.

① 新たな要求であるモジュールと既知のシステム Ξ におけるモジュール間の関係をモジュール関係で定義する.図8における黒丸への矢線がこの関係を表す.

② 関係が定義できたら,黒丸に相当する仕様を満たすプログラムをシステム Ξ の既知のモジュールの合成プロセスをまねて合成する.

③ 一方,新たなモジュールがシステム Ξ における機能をもつように仕様のまね方をまねて,各機能に相当する仕様を定義する.システム Ξ における仕様のまね方はすべて利用できる.

④ まね方をまねる条件が成立しているところでは,合成プロセスのまね方をまねてプログラムを合成することができる.

新たな要求が「図書貸し出しシステム」のバージョンアップのような既知のシステムに対する要求であれば現在の枠組で扱うことができるが,他のレンタルシステム等に関する要求に対しても過去の仕様変更経験を利用するためには,つぎの問題を解決しなければならない.

第一に,仕様変更要求に対し既存の設計仕様の中から再利用対象を適材適所に選択しなければならない.これは,事例ベース推論[6]等の具体的な事例を再利用する際の難しい問題点である.仕様変更要求を十分に理解し,分析して,適切な既存の設計仕様と結び付けることが必要である.われわれの仕様変更プロセスの枠組は強く型付けされた言語である Extended ML によって定義されている.そして,モジュールは適宜パラメタライズされて,直観的に理解できる程度に抽象化することができ,抽象レベルにおける共通性をもつことができる.フォーマルメソッドを用いた再利用の利点は記述の意味が明確であるために知識の表現が曖昧でなくなることと,その形式性により再利用に必要な解析性をもつことである[14].このような形式性が新たな要求と既存のモジュールとの関係を分析し,モジュールの類似度を評価するデータを提供しなければならないと考える.また,仕様変更は一般・個体,上位・下位,肯定・否定等さまざまな概念関係をもつと考えられる.そこで,モジュールの抽象レベルにおける共通性を扱えるようにモジュール関係を拡張し,定義2のまね方をまねる条件を拡張することが必要である.

第二の問題は仕様変更要求に対して何を再利用して対処するのが適切であるかということである.ソフトウェア開発において再利用対象をプログラム自身とするか否かは,部品の定義や要求の種類に依存する.仕様を簡潔にまとめるためには部品化が必要であり,柔軟な対応を行うためには,変更のプロセスを細かい単位で利用することが必要である.例えば直接データ変換でプログラムが再利用できる場面で,プロセスを使うのは妥当とは思えないので,どんな場面で何を再利用することが適切であるかを検討しなければ効果的な再利用は望めない.ソフトウェアの再利用において何か一つの方法が絶対ということはなく,両方の方法が適宜用いられる環境が望ましいと考える.

## 6. おわりに

本稿ではプログラムを再利用するために,プログラムの作成プロセスおよび,そのプロセスをまねるプロセスを再利用する再利用手法を提案した.合成プロセスのまね方をまねることによって,仕様変更要求に効率良く対処できることがわかった.このようなメタな思考を計算機上で実現することによって,より柔軟で系統的な再利用を可能とするソフトウェア開発環境が

構築できると考える．

**謝辞** 本研究は，産業科学技術研究開発制度「新ソフトウェア構造化モデルの研究開発」の一環として情報処理振興事業協会（IPA）が新エネルギー・産業技術総合開発機構から委託をうけて実施したものである．

## 参考文献

1) Carbonell, J.G.: Derivational Analogy, A Theory of Reconstructive Problem Solving and Expertise Acquition, Michalski, R.S. et al. (eds.), *Machine Learning II: An Artificial Intelligence Approach*, pp.371–392, Morgan Kaufmann (1986).
2) Dershowitz, N.: Program by Analogy, Michalski, R.S. et al. (eds.), *Machine Learning II: An Artificial Intelligence Approach*, pp.393–421, Morgan Kaufmann (1986).
3) Goguen, J.A.: Principles of Parameterized Programming, Biggerstaff, T.J. et al. (eds.), *Software Reusability, Volume I, Concepts and Models*, pp.159–225, Addison Wesley (1989).
4) Goldberg, A.: Reusing Software Developments, *Proc. of the Fourth ACM SIGSOFT Symposium on Software Development Environment*, pp.107–119 (1990).
5) Gries, D.: *The Science of Programming, Texts and Monographs in Computer Science*, Springer-Verlag (1981).
6) Kolodner, J. and Riesbeck, C.: Case-Based Reasoning, IJCAI-89 Tutorial-MA2 (1989).
7) Krueger, C.W.: Software Reuse, *ACM Comput. Surv.*, Vol.24, No.2, pp.131–183 (1992).
8) Lue, J. and Xu, J.: Analogical Program Derivation Based on Type Theory, *Theoretical Computer Science*, 113, pp.259–272 (1993).
9) 松浦，本位田：仕様変更のプログラムへの写像 — 仕様変更プロセスを利用したプログラム合成 —，情報処理学会論文誌，Vol.36, No.5, pp.1211–1227 (1995).
10) Milner, R., Tofte, M. and Harper, R.: *The Definition of Standard ML*, The MIT Press (1990).
11) Mostow, J.: Design by Derivational Analogy: Issues in the Automated Replay of Design Plans, *Machine Learning*, pp.119–184, MIT Press (1990).
12) Problem Set for the 4th International Workshop on Software Specification and Designs, *Proc. of 4th International Workshop on Software Specification and Design* (1987).
13) Sannella, D.T. and Tarlecki, A.: Program Specification and Development in Standard ML, *Proc. 12th Ann. ACM Symp. on Principles of Programming Languages*, pp.67–77 (1985).
14) Schafer, W., Prieto-Diaz, R. and Matsumoto, M. (eds.): *Software Reusability*, Ellis Horwood (1994).
15) Shriver, B. and Wegner, P. (eds.): *Research Directions in Object-Oriented Programming*, The MIT Press (1987).

## 付　録

### A.1 設計仕様の例 — 図書貸し出しシステム

```
(* signature から end までが図書貸し出しシステムのモジュールインタフェースの定義であり，val はアクションの型を表す． *)
signature LIBRARY= sig
 val card_search:lendingtag -> card set -> reply
 val lend:lendingtag -> library -> library*thing
(* axiom は，アクションの公理の定義である．
Extended ML の公理は，一階述語論理における論理式であり，結合子 not,andalso,orelse, ⇒と限定記号 exists,forall を用いて構成される． *)
(* アクション「貸し出し (lend)」のアクション仕様の定義 *)
axiom Ok c = card_search tag cs
 => lend tag (Library {rec=cs,reserve=t})
 =(Library {rec=cards_update c tag cs,
            reserve=t}, Lend c)
axiom No tag c= card_search tag cs
 => lend tag (Library {rec=cs,reserve=t})
 =(Library {rec=cs,
      reserve=Reservation.rtable_add tag c t},
 Reserve(Reservation.rtag_issue tag c (plan c)))
axiom None = card_search tag cs
 =>lend tag (Library {rec=cs,reserve=t})
=(Library {rec=cs,reserve=t}, Notice "Nothing")
end
(* functor から end までが，モジュール本体を定義している．datatype は，Extended ML のデータであり，右辺で定義される構造をもつデータが定義される．functor はパラメータをもつモジュール本体を，structure はパラメータをもたないモジュール本体を表す．ここでは Library は functor であり，Reservation という structure をパラメータとしてもつ． *)
functor Library(structure Reservation):LIBRARY=
struct
 local open Reservation
 in
(* 貸し出し票のデータの定義．以下，Lending および Library のような先頭が大文字の単語は，データを構成する値構成子である． *)
datatype lendingtag
=Lending of {title:title, author:author,
             borrower:user,ymd:date}
(* 右辺の値構成子 Library で関係づけられた各型構成子 card set, table の関係は型の直積関係である．
ここで，of の後ろの{field:type,...}は，レコード型のフィールド名とその型を表す． *)
datatype library
=Library of {rec:card set, reserve:table}
(* 貸し出しカードのデータの定義 *)
datatype card
```

```
=Card of
  {book:book,id:num,record:lendingrecordset}
datatype lendingrecord
=Rec of {borrower:user,ldate:date, rdate:date}
datatype book
=Book of {title:title, author:author}
(* 右辺の | は，型の直和関係を表す． *)
datatype reply=Ok of card
              |No of lendingtag * card |None
datatype thing=Lend of card
        | Reserve of rtag | Notice of message
 end
(* 合成されたプログラム (lend)．ここで library は参照
型の変数であり，!が値の参照を：= が値の代入を表す． *)
val lend : lendingtag -> thing
fun lend tag =
    let val Library {rec=cs,reserve=t}=!library
        val result = card_search tag cs
    in case result of
        Ok c =>(library:=
     Library {rec=cards_update c tag cs,
              reserve=t}; Lend c)
      | No tag c => (library:=
    Library {rec=cs,
        reserve=Reservation.rtable_add tag c t};
 Reserve(Reservation.rtag_issue tag c (plan c)))
      | None => Notice "Nothing"
    end
```

### A.2 合成プロセスのサブプロセス

```
(* 1 Some-elimination 規則が二つの論理式◆に適用され
る．結果の論理式が◇である．この (入力，規則，出力) が
サブプロセスを表す． *)
◆exists c.member c (match t (insert b bs))=true
         andalso vacant c=true
        => card_search t (insert b bs) = Ok c
◆equal t b => member b (match t (insert b bs))
                      =true
◇vacant b=true => card_search t (insert b bs)
                     =Ok b
(* 2 アクション‘‘lend’’ の公理を書き換えた結果，関
数 pattern_analysis によりつぎのような「パターン環
境」が得られる．ここで入力が書き換えられた公理の集合で
あり，公理のパターンと変数の値が返される． *)
Env {binding=[('A','card_search tag cs'),
   ('p1','Ok c'),
 ('F','lend tag (Library {rec=cs,reserve=t})'),
 ('E1','(Library {rec=cards_update c tag cs,
                  reserve=t}, Lend c)'),
 ('p2','No tag c'), ('E2','Library {rec=cs,
   reserve=Reservation.rtable_update tag c t}'),
 ('p3','None'),
 ('E3','(Library [rec=cs,reserve=t},
                      Notice "Nothing" )')],
 pattern=['A=p1=>F=E1', 'A=p2=>F=E2',
                        'A=p3=>F=E3']}
```

### A.3 仕様変更 — 広域サービス化

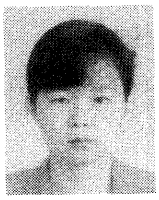
```
signature WAS_LIBRARY= sig
(* relation はモジュール関係を表す予約語である．
difference 以下が設計仕様の差分である．
これは，WAS_LIBRARY が継承と構造拡張関係によって
LIBRARY から定義されることを表している．*)
  relation [(LIBRARY,
             Inherit.Structureextension)]
  difference
(* ネットワーク上の個々のシステムを識別するために，継承
(直積並列) 関係によって，つぎのデータ変換が行われる．*)
(* データパターンの変換 :
[datapattern_trans D_parallel
 '(Library {rec=cs,reserve=t})'
 '(Library {id=i,rec=cs,reserve=t})'] *)
(* データパターンの変換 :
[datapattern_trans Recursive
 '(Library {id=i,rec=cs,reserve=t})'
 '(Network {host=Library {id=i,rec=cs,
            reserve=t},local=ls})'] *)
(* 構造拡張関係によって，アクション'lend' の対象は一つ
の図書館情報から複数の同種の図書館情報に変更される．*)
(* 利用者が要求する図書が自己の図書館で貸し出し可能な
らば，オリジナルの貸し出し処理 'lend' と同じ処理を行
う． *)
  axiom Ok c = card_search tag cs
⇒lend tag (Network {host=Library {id=i,rec=cs,
                                  reserve=t},local=ls})
 =(Network {host=Library {id=i,
    rec=cards_update cs c tag,reserve=t},
            local=ls}, Lend c)
(* [naxiom21=
  if change? oaxiom2=true
  then
   let val t1=and_add
  '(exists l.(card_search tag (#rec l))=Ok c1
    andalso (member l ls)=true)' Last oaxiom2;
     val t2=argument_replace 'rtag_issue' 4
         '2nd(order (Network {host=
       Library {id=i,rec=cs,reserve=t},
      local=ls}) i (#id l) tag (Ok c1))' t1;
     val t3=argument_replace 'rtable_add' 1
                   'id:num' '(#id l)' t2;
   in  t3
   end  else oaxiom2 ]
ここで‘‘oaxiom2’’ はオリジナルの設計仕様におけるアク
ション‘‘lend’’ の 2 番目の公理に上記のデータパターン変
換を適用した公理を表している．
一方‘‘naxiom21’’ は変更されたつぎの公理である． *)
axiom No tag c = card_search tag cs andalso
(exists l.(card_search tag (#rec l))=Ok c1
  andalso (member l ls = true))
 => lend tag
  (Network {host=Library {id=i, rec=cs,
            reserve=t},local=ls})
  =(Network {host=Library {id=i,rec=cs,
  reserve=rtable_add (#id l) tag c t},
             local=ls},
  Reserve(rtag_issue (#id l) tag c1
    2nd(order (Network {host=
    Library {id=i,rec=cs,reserve=t},
    local=ls}) i (#id l) tag (Ok c1)))))
(* つぎの公理も‘‘oaxiom2’’ を別のモジュール操作によっ
て変更した公理である．本文では‘‘naxiom22’’ と書かれて
いる． *)
 axiom No tag c = card_search tag cs andalso
 not(exists l.card_search tag (#rec l))=Ok c1
          andalso (member l ls)=true))
```

```
  => lend tag
(Network {host=Library {id=i,rec=cs,reserve=t},
                local=ls})=
(Network {host=
   Library {id=i,rec=cs,
            reserve=rtable_update i tag c t},
   local=ls}, Reserve(rtag_issue tag (plan c)))
end
functor WAS_Library(structure Reservation)
                                :WAS_LIBRARY=
struct
 local open Reservation
 in
 datatype library
 =Library of {id:num, rec:card set,
              reserve:table}
 datatype was_library
 =Network of {host:library,
              local:library set}
 end
```

### A.4 仕様変更 — 電子情報サービス化

```
signature EIS_LIBRARY=
sig   relation [(LIBRARY,Inherit)]
      difference
axiom Ok c = card_search table cs =>
 ((#book c=Book {obj=b} =>
   lend table (Library {rec=cs,reserve=t})=
(Library {rec=cards_update c tag cs,
          reserve=t}, Lend c))
      orelse
  (#book c=Elt {obj=e} =>
   lend table (Library {rec=cs,reserve=t})=
(Library {rec=cards_update c tag cs,reserve=t},
     Copy c charge_receive)))
axiom No tag c= card_search tag cs
   => lend tag (Library {rec=cs,reserve=t})=
(Library {rec=cs,
     reserve=Reservation.rtable_update tag t},
     Reserve(Reservation.rtag_issue tag c))
axiom None = card_search tag cs
  => lend tag (Library {rec=cs,reserve=t})=
(Library {rec=cs,reserve=t},Notice "Nothing")
end
functor EIS_Library(structure Reservation)
                    : EIS_LIBRARY=
struct
 local open Reservation
 in
(* 継承（直和）関係により，一般化された図書のデータ book に対し，電子情報データを新たな図書データの場合わけとして追加する．*)
datatype book=Book of {obj:object}
              | Elt of {obj:object}
datatype object=
 Obj of {title:title, author:author}
 end
```

(平成6年12月27日受付)
(平成7年 9月 6日採録)



**松浦佐江子**（正会員）

1955年生．1979年津田塾大学学芸学部数学科卒業．1982年同大学院理学研究科数学専攻修士課程修了．1985年同大学院理学研究科数学専攻博士課程単位取得退学．同年4月(株)管理工学研究所入社，研究員．以来，ソフトウェア開発環境に関する研究開発に従事．ソフトウェア開発環境および設計方法論におけるフォーマルなアプローチ，関数型言語に興味を持つ．1993年，情報処理学会研究賞受賞．日本ソフトウェア科学会，人工知能学会各会員．現在，情報処理振興事業協会・新ソフトウェア構造化モデル研究本部に出向中．

**本位田真一**（正会員）

1953年生．1976年早稲田大学理工学部電気工学科卒業．1978年同大学院理工学研究科電気工学専攻修士課程修了．工学博士．同年（株）東芝入社．現在，同社研究開発センターシステム・ソフトウェア生産技術研究所に所属．1989年より早稲田大学非常勤講師を兼任．1991年東京工業大学大学院非常勤講師．主として，ソフトウェア工学，人工知能の研究に従事．ソフトウェアの基礎理論に興味を持つ．1986年，情報処理学会論文賞受賞．著訳書「ソフトウェア開発のためのプロトタイピング・ツール」(共著),「KE養成講座(2) エキスパートシステム基礎技術」(共著),「オブジェクト指向システム分析」(共訳）など．日本ソフトウェア科学会理事．日本ソフトウェア科学会，人工知能学会，IEEE，AAAI各会員．