

EVA:仕様変更プロセスを用いたプログラム開発支援システム

松浦 佐江子[†] 来 間 啓 伸[†] 本位田 真一^{†,☆}

我々はシステムへの要求が変更されたときに、新たな要求を満たすようにプログラムを修正する方法を提案してきた。そして、このような作業を容易に行えるプログラム開発支援の確立を目指している。このためには、人間のシステム利用プロセスを考慮し、人間にとって有効な人間と計算機の役割分担が明確となった支援を実現する必要がある。そこで、我々は本方法に基づいてプログラム開発支援システム EVA を開発した。本稿では、プログラム開発における EVA 計算機支援の実現および有効性について議論する。EVA は人間のプログラム作成作業・プログラム変更作業を記録し、これらの作業を再利用して、人間が与えた変更要求を満たすプログラムを生成する機能を提供する。これらの機能を真に使いやすい機能として利用者に提供するためには、計算機が変更に対してプログラム全体の構文および意味的な整合性を保証すること、プログラムのどこを修正すればよいかを誘導すること、人間にとって明らかな変更を代行すること等が本質的に必要であり、このような支援を行うことによって、人間の作業負担を軽減したり、変更に対して安定したシステムを保守することができると考える。EVA は我々の提案する方法に基づいて設計され、かつこれらの3つの支援を実現したプログラム開発支援システムである。

EVA: A Flexible Programming Support System Using Specification Change Process

SAEKO MATSUURA,[†] HIRONOBU KURUMA[†] and SHINICHI HONIDENT^{†,☆}

We proposed a new method allowing a program to be modified to meet new requirements through the use of program-deriving processes. Based on our method, we aim at constructing a flexible programming support system whereby we can create a program easily and safely. We have constructed a programming support system EVA, and this paper explains its configuration and features. As programming is human intellectual work, computer systems can not carry out all of programming work for us. Thus computer systems need to satisfy the following three conditions to provide us with flexible and safety support. First, they need to be able to guarantee that the modified program has consistency in terms of syntactic and semantic rules. Second, they need to have the capability of showing a place where modification is needed in a program. Third, they need to be able to carry out well known work for us. EVA is a flexible programming support system based on our method, and provides us flexible computational support that meets the above three conditions. This is because EVA records human working processes whereby she creates programs and changes them, and gives us a procedure for generating programs that meet new requirements by reusing these processes.

1. はじめに

我々にはいったん作成されたプログラムを次々と提出される要求に応じて変更することが求められている。しかし、システムに対するあらゆる変更要求をあらかじめ予測することはできない。そこで、我々はシステムへの要求が変更されたときに、プログラムを容易に

修正することができるプログラム開発支援の確立を目指している。

プログラム修正は人間が行うことができる知的作業であり、その一般解は存在しない。すなわち、人間はプログラムの意図やプログラムの作成経験を認識しており、この知識を用いて、状況に応じた適切なプログラム修正方法を見つけることができる。しかし、計算機はプログラムだけからこのような推論を行うことはできない。

現在、プログラム開発のための方法論やビジュアル言語等を支援した様々な CASE ツールが提案されている。これらのツールでは方法論の規則、言語の規則

[†] 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、株式会社東芝

Presently with Presently with Toshiba Corporation

等, プログラム作成において利用可能な規則を規定することはできるが, 問題の多様性や利用者の思考過程の違いによって, これらの規則の利用手順を一般化することはできない. 作成の一般解がないので, 変更に際してはプログラムを解析して, 再度規則の適用を考察しなければならない. ところで, プログラム作成および変更時において人間が行う主な作業は記述された内容を要求を満たすように規則を用いて編集することであり, この編集作業が人間のプログラム作成および変更時の思考過程を表している. この過程を記録することによって, 計算機はプログラムを修正するための知識を持つことができる. そこで, 頻繁なプログラムの変更を支援し, 変更に対して安定したシステムを保守するためには, 人間の多様な思考過程を編集作業を通して人間から獲得し, 計算機はこれを再利用してプログラムを変更することが有効であると考えられる.

さて, このような方針に基づき我々の目指すプログラム開発支援を実現するためには, 次の課題を解決しなければならない.

- (1) 人間のプログラム開発作業を利用してプログラムを修正する基本メカニズムを確立する. すなわち, プログラムの性質を議論できる言語を設定し, 人間の作業の表現および獲得方法, 人間の作業の修正方法を定式化する.
- (2) 変更要求は次々と生じるので, 効率良く変更作業を行わなければならない. このために再利用可能な変更作業の事例を蓄積し, 既存の変更作業を再利用する方法を確立する.
- (3) プログラム開発支援を確立するためには, 人間のシステム利用プロセスを考慮し, 人間にとって有効な人間と計算機の役割分担が明確となった支援を実現する必要がある. 我々はこのような支援が利用者にとって「真に使いやすい」支援であると考え.

我々はプログラムの作成プロセスを用いて要求の変更を満たすようにプログラムを修正する方法を提案してきた^{6),7)}. 文献6)では, 形式的仕様記述言語を用いて人間の作業プロセスを形式化し, プログラムの作成プロセスを用いてプログラムを修正する方法を設計モデルとその上での変更の観点から論じ, 上記(1)の解を与えた. そして, 文献7)では6)の方法を拡張し, (2)の問題を解決し, 効率の良いプログラム修正について論じた. 我々の目指すプログラム開発支援を確立するためには, 本方法を「真に使いやすい」機能として利用者に提供し, それによって(3)で述べた計算機支援を行う必要がある. そこで, 我々は本方法に基づ

いてプログラム開発支援システム EVA を開発した. 本稿では, プログラム開発における EVA 計算機支援の実現および有効性について議論する.

EVA は文献6), 7)で提案したプログラムの修正方法を次の機能として実現する.

(I) 人間のプログラム作成作業・プログラム変更作業を記録する.

(II) 作業を修正して再利用する.

EVA の目的は(3)で述べた計算機支援を行うことである. そこで, 上述の2つの機能において次のような支援を実現することが課題であると考え.

(a) 形式的な記述には多くの制約があり, 変更作業により多くの影響が生じる. すべての制約をつねに念頭に置いて作業することは人間にとって負担である. そこで, 計算機は人間の作業の影響を監視し, 可能な範囲で自動的な整合を行う.

(b) 人間が作業すべき部分を特定し, 他の部分の処理は計算機に任せることができる.

(c) 一度行った作業は, 多少異なる状況においても計算機が再現してくれる.

そこで, EVA は各課題に対して次の計算機支援を実現する. そして, 上記の機能を真に使いやすい機能として利用者に提供し, 人間の作業負担を軽減したり, 既存のプログラムを利用して安定したシステムを保守することを目指す.

(I) の機能を利用して変更作業を行う責任は人間にある. そこで(a)を実現するために,

- 変更に対してプログラム全体の構文および意味的な整合性を保証する.

(II) の機能においては, 各々(b), (c)を実現するために,

- 人間と計算機の役割を分離し, プログラムのどこを修正すればよいかを誘導する.
- ある状況で行われた作業を異なる状況においても利用する場合に, これを自動的に修正し, 人間にとって明らかな変更を代行する.

さて, 本稿の構成は次のとおりである. 2章では, プログラムの変更について議論し, プロセスを用いたプログラム変更の必要性を説明する. 3章では, EVA の基本メカニズムについて説明する. 4章では, この基本メカニズムを「真に使いやすい」機能として提供するための EVA 計算機支援の実現方法について説明する. 最後に例題を用いた EVA システムの評価および今後の課題について議論する.

2. プログラムの変更例

本章では、表の操作を例として、プログラムの変更について議論する。そして、人間のプログラム変更方法から現在の計算機支援の限界を考察し、作業プロセスを用いたプログラム変更の必要性を説明する。

「表」は次のような要求を満たすものである。

- 任意の要素と鍵の組の集合である。
- 鍵を与えて対応する要素を検索できる。これを lookup とする。
- 鍵と要素を与えて、要素の更新ができる。これを update とする。

たとえば、「従業員番号」を鍵として「従業員名」を要素とする表を考え、lookup と update を定義する。付録 A.1*の「fun lookup」と「fun update」がこれらのプログラムである。ここで、lookup の第一引数が「表」を第二引数が「鍵」を表している。また、update の第一引数が「表」、第二引数が「鍵」をそして第三引数が「追加する要素」を表している。

さて、ここで「表」に対して、次の4つの変更要求の例を考えてみる。

- (1) 「従業員名」のほかに「所属」の項目を要素に追加する。
- (2) 「従業員番号」のほかに「支社番号」を鍵に追加する。
- (3) 「従業員名」のほかに「所属」の項目を要素に追加する。そして lookup を「従業員番号」を与えて「所属」を検索する lookup' に変更する。
- (4) 「従業員名」のほかに「所属」の項目を要素に追加する。そして update を「従業員番号」と新しい「所属」を与えて、所属のみを更新する update' に変更する。このとき、「従業員名」は変わらない。

各々の変更要求に対して、次のようにプログラムの変更が考えられる。

- (1) 要素のデータが「従業員名」と「所属」の組型となった。しかし、付録 A.1 のプログラムは多相型関数であるので、要素の型が変更されても、そのまま動作する。
- (2) 鍵のデータが「従業員番号」と「支社番号」の組型となった。そこで、鍵を比較する関数 ground_eq の定義を2つの要素が各々一致した場合に等しくなるように変更する。この場合、ground_eq の呼び出し形式は変わらないの

で、関数 lookup および update の定義は不変である。

- (3) (1) で述べたように、与えられた「従業員番号」に対して、lookup は「従業員名」と「所属」の組を出力する。lookup' を定義するためには、次のような組型の値を与えて、その第二要素を返す関数 snd を定義し、これと lookup の関数合成として lookup' を定義すればよい。

```
fun snd (x,y) = y
```

- (4) (1) で述べたように、update の第三引数は「従業員名」と「所属」の組である。しかし、update' の第三引数は「所属」のみである。そして、出力される「表」は与えられた「従業員番号」に対して「所属」のみが更新され「従業員名」はそのままでなければならない。そこで、この場合には、上記の例のように単純に update を変更して update' を得るというわけにはいかない。

ただし、(4) の場合にも、次のような機械的な変更操作をプログラムに施せばよいことは分かる。

- 表の要素を組型に変える。以下、「 $\Rightarrow$ 」はプログラム内の変数の置換えを表す。

```
e  $\Rightarrow$  (e1,e2)
```

```
x  $\Rightarrow$  (x1,x2)
```

- update の右辺の組型の要素の中で第三引数に相当する値を左辺の変数で置き換える。

```
(x1,x2)  $\Rightarrow$  (x1,x)
```

しかし、このような変更操作を行った結果、以下に示すように未定義の変数「x1」が update のプログラムに残ったままである。

```
fun update' nil i x = [(i,(x1,x))]
  | update' ((k,(e1,e2))::rest) i x =
    if Order.ground_eq (k,i)
    then (k,(x1,x))::rest
    else (k,(e1,e2))::(update' rest i x)
```

このとき、人間は、表が空である場合には update' はエラーであり、x と置き換えられた要素の第一要素ははじめの値と同じであることを変更要求から判断でき、上記のプログラムをそれらしく変更することができ、

しかし、計算機にはこの判断はできない。それは未定義変数に対する推論がプログラムだけでは機械的に行われないからである。さらに、このプログラムの変更が本当に要求を満たしているかどうか分からない。すなわち、計算機がこのような推論を行うためには、どのように update のプログラムが作られたか、

* 付録において (* ... *) の記号はコメントを表す。

update と update' の関係は何かを認識できることが必要である。

変更要求にプログラムが追従できるかは、記述言語の抽象化能力にも依存する。すなわち、付録 A.1 のようにプログラムを記述することによって、要求によっては、上記の例 (1)~(3) のように元のプログラムとは独立に変更を行うことができる。しかし、(4) の場合には、変更を追従できるような抽象化されたプログラムをあらかじめ記述することはできない。すなわち、プログラムをいかに抽象的に記述しても、機械的な修正には限界がある。

そこで、EVA ではどのように update のプログラムが作られたか、update と update' の関係は何かということプロセスとして形式的にとらえ、これらを利用してプログラムを変更する。付録 A.2 は「表」の形式的仕様記述である。仕様はプログラムに比べて、人間の要求を自然に表現できる。そこで EVA ではこのような仕様から付録 A.1 のプログラムを作成する。そして、このプログラム作成プロセスを形式的に記録する。一方、変更の際には仕様に対する上記の変更要求を形式的な変更操作列として定義し、前述の未定義変数を推論するために、変更要求を仕様レベルで形式的にとらえる。この変更操作列を用いて update プログラムの作成プロセス内の未定義変数を機械的に決定し、update' プログラムを生成する。しかも、この操作によってプログラムが定義された変更要求を満たしていることも保証される。

3. 基本メカニズム

文献 6), 7) において、広範囲言語 Extended ML¹¹⁾★を用いた仕様・プログラム・プロセスを統一的に扱う枠組みを定義し、この上で仕様変更作業において人間が行うさまざまなプロセスを形式化した。そして、これらのプロセスを用いて仕様変更されたときに変更要求を満たすようにプログラムを修正する方法を提案した。これが EVA の基本メカニズムとなる。

本方法においては 2 つの「変更プロセス」が基本となる。第一の変更プロセスは人間が既存の仕様から変更要求を満たす新しい仕様を作成するプロセスであり、図 1 の矢線「変更」がこれを表している。このプロセスを「仕様変更プロセス」と呼ぶ。第二の変更プロセスは、図 1 の点線で表される人間が仕様からプログラムを作成した過程であり、これを「合成プロセス」と

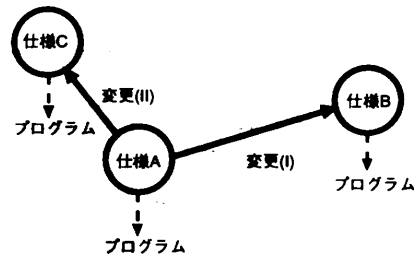


図 1 仕様変更とプログラムとの対応

Fig. 1 A diagram of specification change and programs.

表 1 変更プロセスの定式化

Table 1 Formulation of changing process.

(1) change = object $\rightarrow$ object (* 変更プロセスの型 *)
(2) Change = $\alpha_k \circ \alpha_{k-1} \circ \dots \circ \alpha_1$: change (* 変更プロセス *) α_j : change (* 人間の操作 *)
(3) m_i : string list $\rightarrow$ change (* プリミティブな操作 *) for $\forall j \in \{1, \dots, k\}$. $\exists i$ s.t. $\alpha_j = m_i$ $args_{m_i}$ $args_{m_i}$: string list (* 対象式のリスト *)

呼ぶ。EVA はこれらのプロセスを人間から獲得し、蓄積する。そして、変更 (I) の「仕様変更プロセス」を用いて、仕様 A の「合成プロセス」を修正し、変更された仕様 B を満たすプログラムを生成する。さらに、蓄積された変更 (I) を再利用して仕様 C を変更し、仕様 B および仕様 C の 2 つの変更要求を満たすプログラムを得ることができる。

すなわち、EVA では図 1 に登場するプロダクト（仕様およびプログラム）を再利用するために矢線で表されている人間が行う作業プロセスを利用している。本章では、EVA の基本メカニズムの概要を説明する。

3.1 変更プロセスの定式化

前述の 2 つの「変更プロセス」の定式化を表 1 のように行った。以下、これを説明する。説明文中の番号は表 1 の番号に対応する。

- 「変更プロセス」はある対象から対象への関数 (表 1 (1)) と見なせるので、これを人間が与える操作の列 (表 1 (2) $\alpha_k \circ \alpha_{k-1} \circ \dots \circ \alpha_1$) によって定義する。
- 仕様を記述する言語を定め、その言語によって規定されるプリミティブな操作関数 (m_i) を用いて人間の操作を定義する (表 1 (3))。

これより、「仕様変更プロセス」は仕様から仕様への関数と考えられる。Extended ML では、型および一階述語論理によって関数の仕様を記述する。そこで「仕様変更プロセス」におけるプリミティブな操作関数は、データ型の変更・論理式の変更・関数の引数の変更・関数名の変更等の「仕様」を変更するための関

★ Extended ML は関数型言語 Standard ML^{8),9)}の拡張言語である。

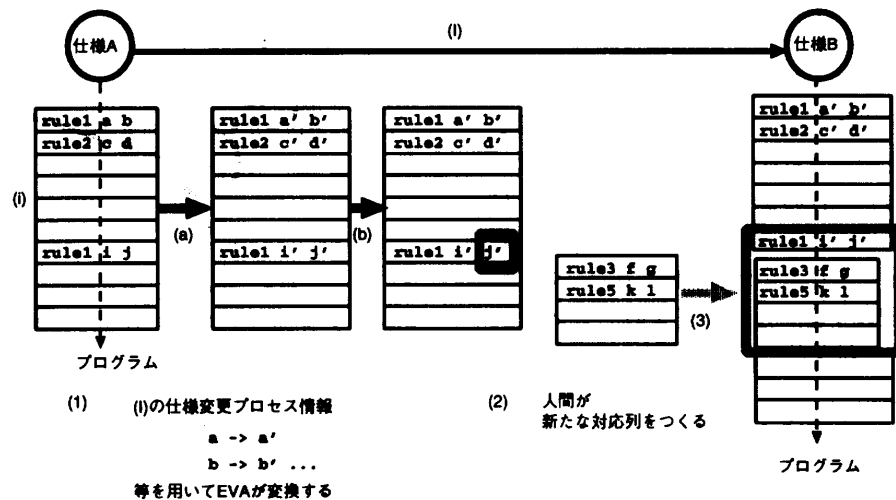


図2 プログラムの修正

Fig. 2 Program modification.

数である。本定義によって、仕様変更プロセスはこれらの操作関数とその対象式の組の列として表現される。

一方、「合成プロセス」はこのような仕様レベルにおける「変更」の情報をプログラムに取り込むための基礎知識となる。「合成プロセス」におけるプリミティブな操作関数は、自然演繹の推論規則・プログラムの制御構造の導入規則等の仕様をプログラムに対応付けるための関数である。本定義によって、合成プロセスはこれらの規則とその対象式の組の列として表現される。

3.2 仕様変更プロセスを用いたプログラムの修正

3.1 節の定式化に基づき、「仕様変更プロセス」を利用してプログラムを修正する方法を次のように定義した。図2において、矢線 (I) は仕様変更プロセスを、四角形の列 (i) は仕様とプログラムの対応列である合成プロセスを表している。以下の説明における番号は図2の番号に対応している。

- (1) EVA が仕様変更プロセスを合成プロセスに適用する*ことによって、
 - (a) 合成プロセス内の対象式を仕様変更プロセスの情報によって変更後の仕様内の対象式に置き換える。
 - (b) 合成プロセス内で、プログラムに対する論理式の変更点 (図2の太線で囲まれた部分) を見つける。
- (2) 人間が変更された論理式に対して前述の推論規則を適用して、新たな合成プロセスを作る。
- (3) EVA が、(2) で定義された合成プロセスを (1)

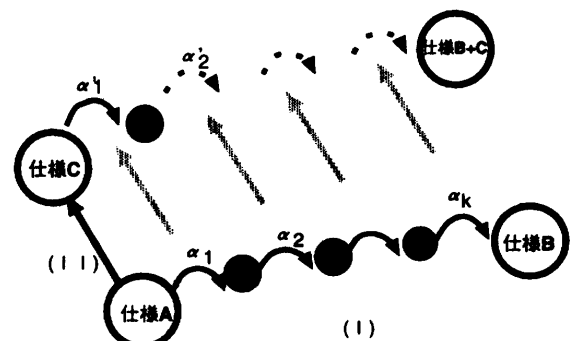


図3 仕様変更プロセスの組合せ

Fig. 3 Reusing a specification change process.

の結果に埋め込むことによって、「変更された仕様」から「目的のプログラム」への対応である合成プロセスを導出する。

本稿では (1)~(3) の手順で合成プロセスを修正して、人間が与えた仕様変更要求を満たすプログラムを得ることを「まねる」と呼ぶ。

3.3 仕様変更プロセスの再利用

EVA ではある条件のもとで他の人間が蓄積した「仕様変更プロセス」を取り入れて目的のプログラムを修正することができる。さて、図3のように1つの仕様Aから各々異なる変更 (I), (II) により仕様Bと仕様Cが生成されたとする。このとき、仕様Bと仕様Cをとともに満たすプログラムを作りたいとする。2つの仕様が共通の仕様からの変更で得られた場合に、これら2つの変更を取り入れた仕様B+Cを満たすプログラムを生成する手順を図3を用いて説明する。

3.1 節で述べたように、「仕様変更プロセス」は人間の操作の列 (図3 (I) $\alpha_1, \alpha_2, \dots, \alpha_k$) であり、人間の

* ここで、「適用する」とは変更プロセスを関数と考え、データである合成プロセスに関数適用することである。

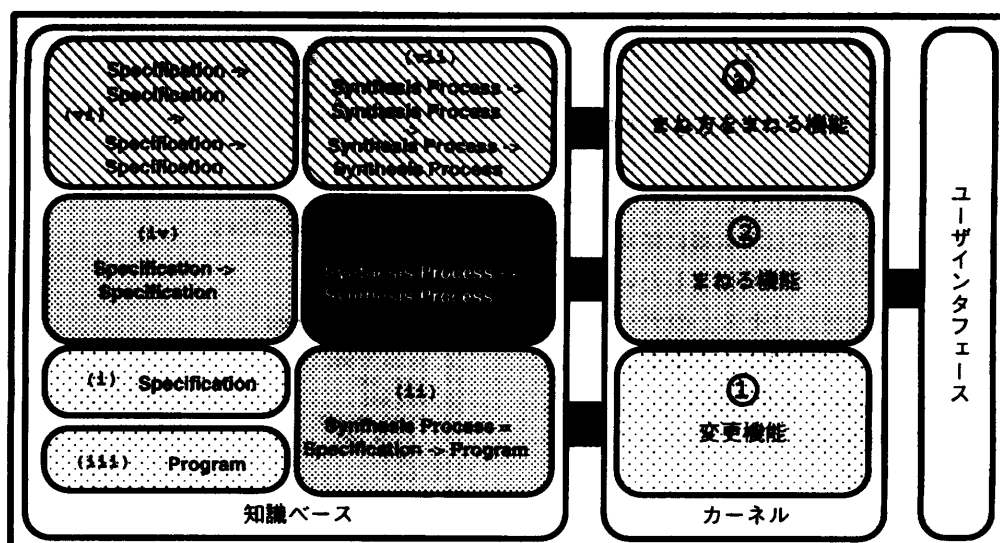


図4 システム構成

Fig. 4 The structure of EVA.

操作は前述のプリミティブな操作関数とその対象式によって定義されている。仕様 A から仕様 B への「仕様変更プロセス」を別の対象 (図 3 仕様 C) に適用しようとする、対象式が異なるので、仕様 C 内の対象式に対する操作の不適合が生じる。そこで、対象式を他方の「仕様変更プロセス」(図 3 (II)) を用いて修正することによって各操作が修正され、これらが仕様 C に対して適合可能になる。修正された各操作 (図 3 $\alpha'_1, \alpha'_2, \dots, \alpha'_k$) をその順序で仕様 C に適用することによって 2 つの「仕様変更プロセス」を取り入れた仕様 B + C が生成される。このような組合せを用いてプログラムを修正することを本稿では「まね方をまねる」と呼ぶ。

4. EVA システム

4.1 特徴

EVA は CML (Concurrent ML) および eXene (CML 上で動作する X ウィンドウインタフェース) を用いて開発された。CML は関数型言語 Standard ML を並列処理が行えるように拡張した言語である。

EVA は 3 章で説明した基本メカニズムに基づいて設計され、以下に示す 3 つの機能を持つ。

- ① 変更機能 (3.1 節参照)
- ② まねる機能 (3.2 節参照)
- ③ まね方をまねる機能 (3.3 節参照)

EVA は次の 3 つのサブシステムから構成される (図 4 参照)。

(1) カーネル

上述の 3 つの機能から構成され、基本メカニ

ズムの処理を行う。

(2) 知識ベース

カーネルの機能において入出力となるデータから構成され、これらを蓄積・管理する。データには、仕様 (図 4 (i))・合成プロセス (図 4 (ii))・プログラム (図 4 (iii))・仕様変更プロセス (図 4 (iv))・合成プロセスをまねるプロセス (図 4 (v))・仕様変更プロセスをまねるプロセス (図 4 (vi))・合成プロセスをまねるプロセス (図 4 (vii)) の 7 種類がある。図 4 の知識ベースの四角形内の記号は格納されているデータの型を表している^{*}。たとえば、② まねる機能は (iv) の仕様変更プロセスと (ii) の合成プロセスを入力として、合成プロセスを出力する。このプロセスが知識ベースの (v) 合成プロセスをまねるプロセス、すなわち合成プロセスから合成プロセスを生成する関数である。

(3) ユーザインタフェース

次に、EVA の特徴を説明する。EVA の利用者は上記の 3 つの機能を用いて仕様変更要求を満たすようにプログラムを修正する作業を行う。そこで、これらの機能を「真に使いやすい」機能として利用者に提供することが EVA の特徴である。使いやすさを提供するために、EVA ではこれら 3 つの機能において次の処理を行う。

- ① EVA を利用して変更を行うのは人間である。そこで、1 つの操作後にプログラム全体の構文およ

^{*} 「A → B」の記号は A から B への関数を表している。

び意味的な整合性が保たれることは人間の作業負担の軽減およびプログラム品質の安定の観点から重要である。そこで、EVA は仕様変更プロセスの各プリミティブ操作に対する変更波及処理を行う。

- ② プログラムをまねるためには、人間はプログラムの修正すべき部分のみを知りたい。そして、仕様変更要求から明らかな修正は計算機に任せ、なるべく少ない労力でプログラムを修正したい。そこで、EVA は「仕様変更プロセス」のデータを用いた合成プロセスの自動変換および変更点の抽出を行い、人間のまねる作業を支援する。ここで、変更点とは仕様変更要求から生じたプログラムの修正箇所のことである。

- ③ EVA に蓄積されたまねるプロセスは人間による変更作業の 1 つのお手本である。そこで、EVA はすでにある変更を自動的に修正して取り込むことで、人間にとって明らかな変更の代行を行う。以下、各項目について説明する。

4.2 整合性の保証

本節では「変更機能」における処理の特徴を説明する。

EVA の「仕様」は複数のモジュールの集合である。そして、モジュールは型と述語論理で記述される。このとき、仕様と対応付けられるプログラムは関数の集合であり、この関数の満たすべき論理式を公理と呼ぶ。各モジュールは次のような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・それらに関する公理から構成される。一方、実現部は型の定義および型定義に関連して詳細化された公理から構成される。付録 A.2 はモジュールの一部であり、たとえば「axiom...」が公理の記述を表している。

さて、3.1 節で述べた定義により仕様変更プロセスを定義するプリミティブな操作関数は、モジュールからモジュールへの関数である。これを EVA では「モジュール操作関数」と呼ぶ。現在 EVA では利用者に対して 23 個のモジュール操作関数を提供している。前述のモジュールの構成要素に基づいて、モジュール操作関数を次のように分類する。

- (1) 識別子の変更：
仕様において使われている関数・型・モジュール名等の識別子を変更する操作である。
- (2) 関数の型の変更：
関数の型の構造を変更する操作である。
- (3) データ型の変更：
レコードのフィールドを増やしたり、データを再帰的に拡張する操作である。

```

- ManipCommand.Specification_Change "Library" "WAS_Library";
Input Specification_Change Name :
WIDE AREA SERVICE
Select Module Name :
1. Library
2. User
3. Set
4
Input module name (new) :
Library=> Was_Library
Select a command :
1. valname_trans
2. valtype_add
3. typename_trans
4. datatype_name_trans
5. datatype_trans
6. axiom_trans
7. type_add
8. val_add
9. datatype_add
10. axiom_add
11. Save
12. Exit
5
Select a module relation name :
1. parallel
2. series
3. sum
4. dataext
5. strext
6
Input datatype name :
1. library
2. card
3. landingrecord
4. book
5. reply
6. thing
7. reserve_rec
8. reservetag
9. landingtag

```

図 5 仕様変更手順

Fig. 5 The procedure for changing a specification.

(4) 公理の変更：

新たな論理和・論理積の追加等の操作である。

仕様変更を行うとき、EVA の利用者はシステムのモジュール構成から対象システムの概要を読み取ることから始める。利用者の次の主たる関心事は特定されたモジュールにおけるデータの変更とそのデータを対象とする関数の公理の変更である^{*}。そこで、主たる関心事に合わせて 23 個のモジュール操作関数を階層化する。すなわち、(3) および (4) に属する操作は「データ型の変更」および「公理の変更」という項目の下位に位置付ける。これによって利用者はデータ型を変更する作業や公理を変更する作業を他の作業と切り分けを行うことができると考える。

ここで、EVA における仕様変更手順について図 5 を用いて説明する。たとえば Library という仕様を WAS_Library という仕様に変更したいとする。この

^{*} 変更要求を上記のモジュール操作の列として定義するためには、要求を仕様のどの部分の変更ととらえるかについて分析しなければならない。分析の 1 つの指針については文献 6) を参照のこと。

とき、EVA の利用者は図 5 の第 1 行目のようにコマンドを入力する。すると EVA の変更機能が呼び出され、この仕様変更プロセスの名前を利用者に問い合わせてくる。ここでは WIDE_AREA_SERVICE と入力する。これ以降、この名前で仕様変更プロセスにアクセスすることができる。続いて、仕様 Library を構成するモジュールのリスト（ここでは 3 つのモジュール）が表示される。そこで、利用者はこの中から変更対象モジュールを選択し、これに新しい名前を付ける。すると、モジュール操作の一覧^{*}が表示されるので、1 つのモジュール操作を選択する。ここでは「5」を選択し、続いて「データ型の変更」に属するモジュール操作を決定する。次に決定したモジュール操作の入力値を EVA の誘導に従って定義する。定義方法は、EVA が提示した候補から選択する・テキストを切り貼りする・キー入力するのいずれかである。たとえば、図 5 では EVA が候補のデータ型のリストを表示している。このような操作を繰り返し行うことによって、利用者は目的の仕様変更要求を定義することができる。

このようにして仕様変更は仕様の構造に基づいたモジュール操作の列として形式的に定義することができる。しかし、仕様変更は複数のモジュールの変更であるから、1 つの変更操作を行ったときにモジュール間における構文および意味の整合性が保たれる必要がある。そこで、整合性を保つために現在計算機で可能な処理を次の 2 つの場合に分類して考える。

- ① 言語の構文規則および意味規則から自動的に修正できる。
- ② 言語の構文規則および意味規則上は正しいが、1 つの変更操作から生じた変更箇所に対して、仕様変更要求を反映させるように利用者が新たな変更を定義しなければならない。ただし、言語の構文規則および意味規則から修正箇所を特定することができる。

各場合について EVA で行っている処理の例を説明する。

- ① 関数名・データ型名・モジュール名等を変更する場合を考える。このモジュール操作を仮に「`*name_trans : string → string → module → module`」と記す。このとき、利用者は操作（`*name_trans`）を指定し、EVA の誘導に従ってモジュール内の変更対象名とその新たな名前を文字列として与える。ここで変更対象を old、新た

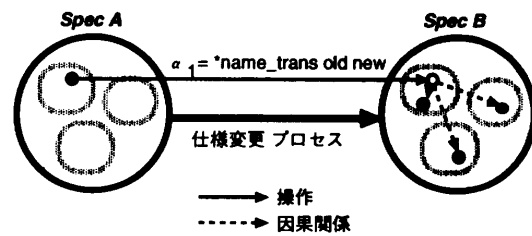


図 6 整合性の保証

Fig. 6 Consistency of the specifications.

な名前を new とする。図 6 は Spec A から Spec B への仕様変更を表している。円が仕様であり、円内の四角形が仕様を構成するモジュールである。ここで、 α_1 が上記の操作であり、点線は変更された名前が他のモジュールで使用されている様子を示している。

そこで、Spec A への α_1 の適用後、Spec B における名前の整合性をとるために、各名前の構文および意味規則を考慮して、全モジュールにモジュール操作「`*trans old new`」を適用する。ここで `*trans` が他の部分で使用されている old を new に変更する操作である。この操作は利用者がはじめの操作で与えた情報だけで決定されるので、`*name_trans` の操作を行った結果において、モジュール間の整合性は自動的に保証される。

- ② 関数の引数を増やす場合を考える。このモジュール操作を「`valtype_add : string list → int → module → module`」と記す。ここで関数 `valtype_add` の引数 `string list` は変更対象関数名・追加する型から成るリストを、そして `int` は追加する型の位置の情報を表している。

この操作の後では、指定された関数の定義およびこれを参照している関数定義における呼び出しボタンが変更されなければならない。そこで全モジュールに対して「`funpattern_trans : string list → int → string → module → module`」を適用する。ここで、`funpattern_trans` の第一および第二引数は `valtype_add` の引数と同じであるが、第三引数は定義および呼び出しボタンにおいて使用される新たな変数名である。この変数は追加された型を持つ。

このとき、EVA は構文および意味規則に基づいて正しい変数を生成し、各公理の定義において呼び出しボタンを変更する。しかし、関数の引数を増やすという操作のみでは、その関数の公理や呼び出されている公理において、新しい引数があるような役割を果たせばよいかを計算機が自動的

^{*} ここで表示されているのがトップレベルの操作名リストである。

^{**} ここで「:」の左側が関数名を、右側がその型を表している。

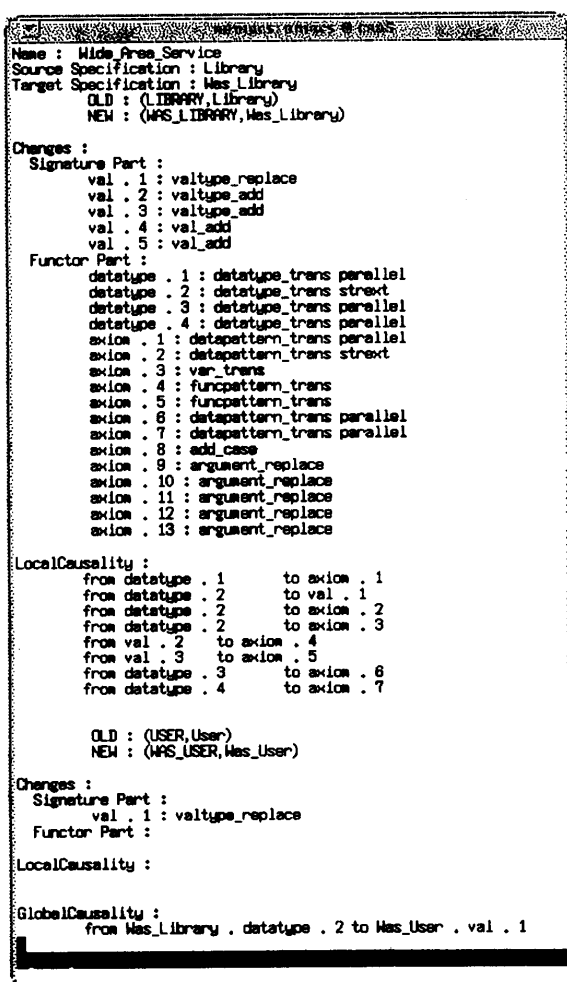


図7 仕様変更プロセスデータ

Fig. 7 Data for changing specification.

に判断することはできない。すなわち、仕様の定義が構文および意味的に正しくても、仕様変更要求に適合しているかは定義する人間でなければ分からないからである。このような場合には計算機は変更される可能性のある箇所を特定することはできるが、本来の変更自身は利用者の責任において行わなければならない。

このように1つのモジュール操作とその操作から派生したモジュール操作との関係を「因果関係」と呼ぶ。各モジュール操作・因果関係およびそれらの発生順序が図4(iv)の「仕様変更プロセスデータ」としてEVAに記録される。図7はEVAによる仕様変更プロセスデータの表示である。仕様変更プロセスは仕様を構成するモジュールごとに記録される。図7の仕様変更プロセスには2つのモジュールに対する変更操作が記録されている(図7 OLD: NEW:)。モジュールLibraryには22個のモジュール操作(図7 Changes:)が、モジュールUserには1個のモジュール操作が記

録されている。因果関係には同一のモジュール間の関係(図7 LocalCausality:)と、異なるモジュール間の関係(図7 GlobalCausality:)の2種類があり、図7では合わせて9個の因果関係が記録されている。すなわち、図7において利用者が自主的に行った操作はval 2~5, datatype 1~4, axiom 8~13の14個であり、残りの9個はEVAが誘導した操作である。

仕様変更操作はあくまでも利用者の責任で要求を定義するものであるが、このように因果関係を処理できるモジュール操作を提供することによって、構文および意味規則によるモジュール間の整合性を自動的に保つことができる場合がある。そうでない場合においても、EVAは利用者に整合性に関するアドバイスを与えることができる。さらにExtended MLは強く型付けされた言語であることから、型推論機能により変更後の式の型の整合性をつねに検査することができる。

もう1つの変更プロセスである合成プロセスは、自然演繹の規則を用いた変換の列によって定義される。この場合の変換は等価な変換であるので、他の定義への影響を考慮する必要はない。

4.3 修正の誘導

本節では、「まねる機能」における修正の誘導を説明する。

EVAの特徴は、仕様とプログラムの対応列である合成プロセスを仕様レベルの変更を定義するモジュール操作関数を使って修正し、変更された仕様を満たすプログラムを得ることである。そこでEVAはまねる機能を実現するために次の2つの誘導機能を持つ。第一の機能は合成プロセスにモジュール操作を適用する自動変換機能である。第二の機能は公理の変更操作から、合成プロセスにおける公理の変更点を見つける機能である。そこで人間は変更された部分の論理式にのみ着目して新たな対応列を作ればよいことになる。以下、合成プロセスの作成手順について簡単に説明した後、各機能について説明する。

4.3.1 合成プロセスの作成手順

まずEVAの利用者はプログラムを導出したい関数名を選択する。するとEVAが選択された関数の入力データボタンを型の定義から導出するので、利用者はこのボタンごとに目的のプログラムを作成する。EVAは利用者の作業プロセスを記録し、図8左側のように構造化して合成プロセスデータを蓄積する。図8において各四角形は合成プロセスのサブプロセスを表している。また、太線の四角形がこのボタンごとの導出単位を表している。利用者は各ボタンごとに適当な仮定を与えて一階述語論理で記述された公理をプログラム

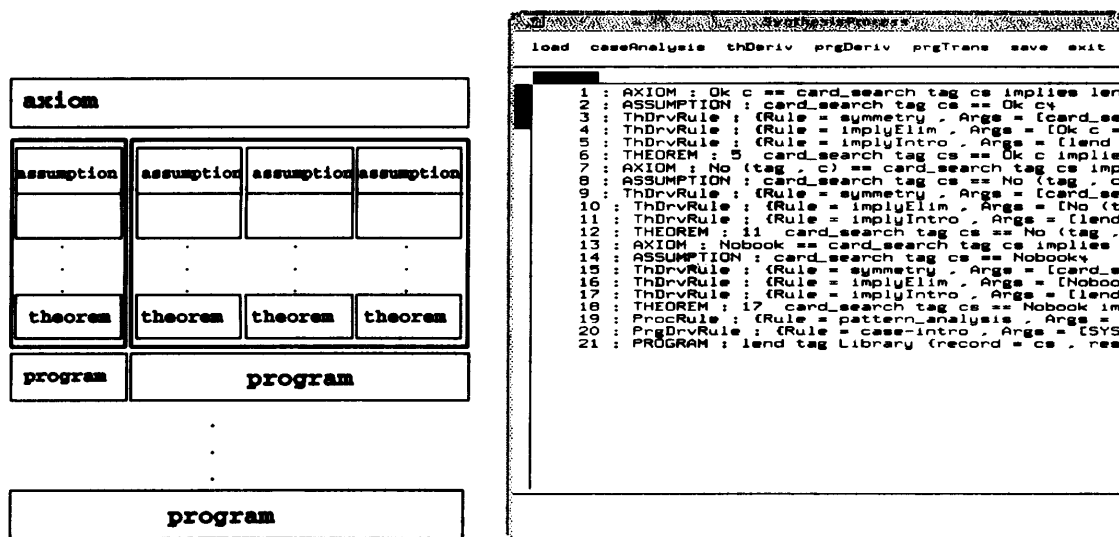


図8 合成プロセス

Fig.8 Synthesis process.

の構造が入る形式に整理する。このように自然演繹の推論規則を用いて整理された論理式を theorem と呼ぶ。そして、利用者はこれらの theorem をまとめて適宜 ML のプログラムに変換する。

図8右側のウインドウは、利用者が1つのボタンに対してプログラムを導出しているところを表している。各行は、利用者が選択した規則とその引数から構成される。これがサブプロセスの実体であり、行番号は利用者の操作順に付けられる。利用者はメニューから規則名を1つ選択し、EVAの誘導に従って規則の入力値を定義する。定義方法はサブプロセスの行番号を指定する・テキストを切り貼りする・キー入力するのいずれかである。ここで、第1行から第6行・第7行から第12行・第13行から第18行が3つの theorem の導出過程を表している。たとえば、第1行から第6行を生成するために利用者は次の操作を行う。まず、公理が表示されているウインドウから1つの公理を選択する(第1行)。次に第6行の theorem を導くために、仮定を入力する(第2行)。自然演繹の規則メニューからの規則の選択と行番号による引数の指定を行い、第6行の theorem を導く(第3~6行)。また、第19・20行はMLのプログラムへの変換を、第21行は結果のプログラムを表している。

4.3.2 自動変換機能

合成プロセスの修正は3.2節で述べた手順で行われる。図2(1)(a)が自動変換機能である。

モジュール操作関数は仕様に記述された「式」を別の「式」に変換する関数である。これらの関数を図2のように「 $a \rightarrow a'$, $b \rightarrow b'$...」と単純化して記す。

ここで「a」等が式を表している。

一方、合成プロセスは公理を表す式の部分式と3.1節で述べた変換規則の組の列であり、図2のように上述の式「a」「b」等を含んでいる。そこで、合成プロセス内の式にモジュール操作関数「 $a \rightarrow a'$, $b \rightarrow b'$...」を適用することで変更された式を得ることができる。この変更された式は、変更された仕様に記述された式そのものである。

4.3.3 変更点の抽出機能

次に図2(2)の処理について説明する。

前述の自動変換機能によって、公理を変更するモジュール操作が合成プロセスに適用された。EVAではこの操作結果から公理の変更点を抽出する。そこで抽出手順を説明する前に、公理の変更操作の種類について説明する。

公理の変更操作は次のように分類できる。表2は各変更操作関数の概要と変更ボタンを示している。表2の分類番号は以下の説明の番号に対応する。

- (1) 論理式に新たな論理式の論理和・論理積・前提となる論理式を追加する。
- (2) 新たな公理を定義する。
- (3) 論理式を構成する各式の定義を変更する。

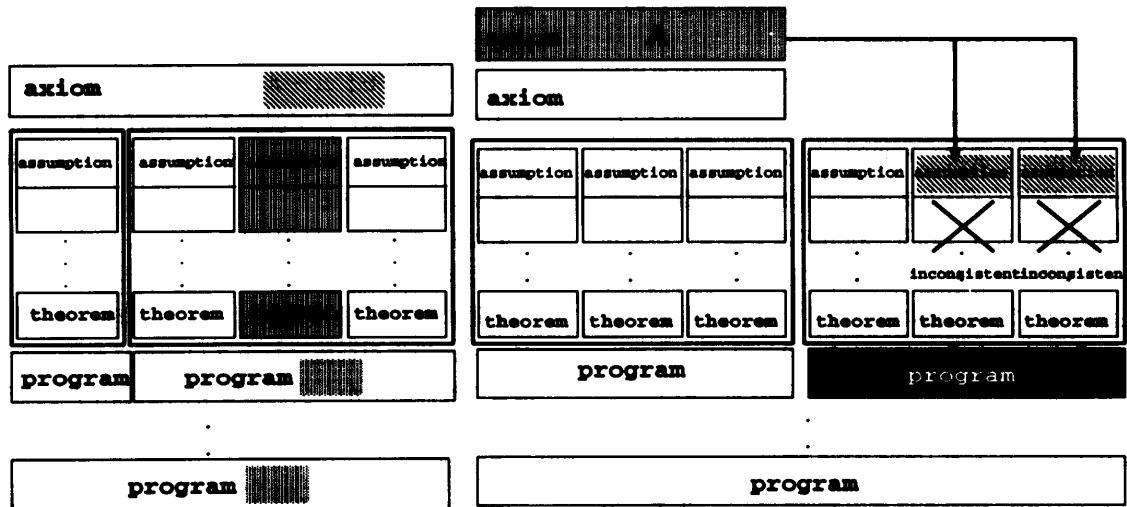
続いて、図9を用いて変更点抽出の処理について説明する。公理の変更点の抽出は公理の論理的関係の変更を用いて行う。そこで、表2の分類(1)および(2)が抽出処理に用いられる。

さて、(1)の操作による変更点抽出の処理は次のように行われる。(1)では論理式の構造が変更されたので、その部分に対応するプログラムを導出し直さな

表 2 公理の変更操作

Table 2 Manipulations for the axiom.

分類	関数	概要
(1)	insert_exp_and	公理の論理式をそれと新たな論理式の論理積で置き換える. $A : \text{exp}$ $B \text{ implies } C : \text{exp} ==> (A \text{ andalso } B) \text{ implies } C$
	insert_exp_or	公理の論理式をそれと新たな論理式の論理和で置き換える. $A : \text{exp}$ $B \text{ implies } C : \text{exp} ==> (A \text{ orelse } B) \text{ implies } C$
	add_condition	公理の帰結部に新たな論理式を前提として追加する. $A : \text{exp}$ $B \text{ implies } C : \text{exp} ==> B \text{ implies } (A \text{ implies } C)$
	add_case	公理の帰結部に対し, 新たな論理式を前提とした複数の論理式を生成し, それらの論理和で公理の帰結部を置き換える. $A1, A2, \dots, An : \text{exp} \quad B \text{ implies } C : \text{exp} ==>$ $B \text{ implies } (A1 \text{ implies } C \text{ orelse}$ $A2 \text{ implies } C \text{ orelse } \dots$ $\text{orelse } An \text{ implies } C)$
(2)	axiom_add	新たな公理を追加する.
(3)	argument_replace	公理で用いられている関数の入力値を変更する. $\text{function } \text{arg1 } \text{arg2} ==> \text{function } \text{newarg1 } \text{arg2}$
	applyexp_replace	公理で用いられている関数の参照形式を変更する. $\text{function1 } \text{args1} ==> \text{function2 } \text{args2}$



Case 1

Case 2

図 9 変更点の抽出とプログラムの修正

Fig. 9 Finding a changing point and program modification.

ればならない。すなわち、この場合には変更対象の合成プロセス内で導入されたプログラムの構造の部分構造を作成することになる。ところで、合成プロセスのデータは対象式と規則の組であり、4.3.1 項で述べたようにサブプロセス番号で識別および構造化されている。そこで、公理の変更操作を合成プロセスに適用した結果、図 9 の Case 1 のように公理の変更点に対応するサブプロセス構造を特定することができる。斜線の施された部分が、この対応を表している。そこで、

公理の変更操作を適用したときに変更された合成プロセス内のサブプロセス番号を特定し、合成プロセス内のサブプロセスの部分的な入れ換えを行うことによってプログラム構造の変更を行う。

次に (2) の場合を考える。たとえば、利用者があるデータの場合分けを定義し、その追加されたデータに対して新たな公理 A を追加したとする。このとき、追加されたデータに対して、新たな公理 A をも満たすようにプログラムを作り直さなければならない。ここ

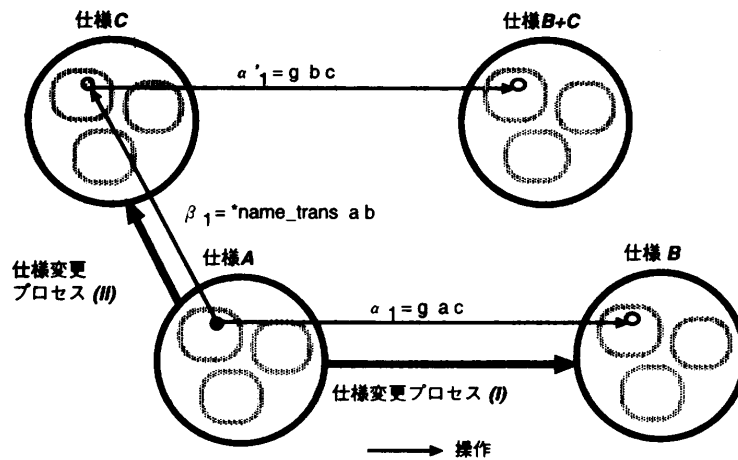


図 10 静的処理

Fig. 10 Static transaction.

で公理 A が既定義の公理内の論理式と矛盾するとする。このような場合、EVA は次のような誘導を行う。図 9 Case 2 の左側の theorem のブロックは、はじめのデータに対する公理から得られたものである。新たなデータに対する公理自体は変わっていないので、そのデータに対するプログラムははじめのプログラムと同様に導出できる。そこで、EVA はこのブロックをコピーする。右側のブロックがこのコピーである。次に EVA は追加された公理 A を各 theorem 導出の仮定と照合し、矛盾している場合にはこのブロック自体を取り除く（図 9 Case 2 の×印がこの行為を表す）。このようにして、EVA は新たなデータに対するプログラム（図 9 Case 2 の斜線部分）を導き出すことができる。ただし、EVA が自動的に見つけられる矛盾は、同じ述語が異なる値をとる場合である。

4.4 代行の実現

EVA のようにプログラム開発のプロセスを蓄積しておく、既知の変更を用いてプログラムを変更することができる。これをまね方をまねると呼んだ。ただし、まね方をまねるには既存の変更と変更したいプログラムの間にある条件が成り立つことが必要である[☆]。ここでは簡単に両者が図 3 のような関係を満たすこととする。しかし、両者の関係が定義されているとはいえ、一般には既存の変更が定義された状況と、利用者が現在変更したいプログラムが作成された状況は異なる。ゆえに、既存の変更を異なる状況において用いることは一般には難しいと思われる。

EVA は 3.3 節で述べた仕様変更プロセスの修正等

により、この問題を解決する。すなわち、EVA は次のようなモジュール操作の解釈によって柔軟な操作の活用を行い、利用者に対して既存の変更を代行することができる。

● 静的処理：

- － モジュール操作を仕様に適用する前に、一方のモジュール操作で他方のモジュール操作の引数を修正する。
- － モジュール操作にはいくつかの種類がある。一方の変更で使用されているモジュール操作の性質から、他方の変更のモジュール操作の部分的な再適用を行う。

● 動的処理：

モジュール操作の適用時に操作の入力値を新たな状況において柔軟に解釈することによって、異なる状況においても変更を実行する。

以下、これらの処理の要点を例を用いて説明する。

4.4.1 静的処理

静的処理には前述のとおり 2 つの処理がある。図 3 の (I) の操作を仕様 C に適用する場合を考える。

第一の処理は操作の適用前に行うものである。図 10 を用いて、これを説明する。今、仕様変更プロセス (II) には名前変更のモジュール操作「*name_trans a b」があるとする。このとき、a を引数とするモジュール操作が (I) になければ、(I) のモジュール操作はそのまま仕様 C に適用することができる^{☆☆}。一方、a を引数とするモジュール操作が (I) にある場合、これを「g a c」とする。このとき、*name_trans を g の引数に適

[☆] 蓄積される変更の単位およびまね方をまねる条件については、文献 7) を参照のこと。

^{☆☆} ただし、ここでは簡単のため b に関する名前の衝突はないものとする。

用して、モジュール操作を「g b c」に変更する。変更されたモジュール操作を仕様 C に適用する。

第二の処理は操作の適用後に行うもので、次のとおりである。詳細は省略したが、4.2 節で述べたように、モジュール操作にはデータ型の変更を行うものがある。本操作はデータ型の構造に応じて異なる変更処理を行う操作に細分されるが、その中に既定義のデータ型を再帰的に用いて新たなデータ型を定義する操作がある。たとえば、(I) においてこのような再帰的なデータ型の変更が行われ、そのデータを入力とする関数の公理が変更された場合を考える。このとき、このモジュール操作を仕様 C に適用することによって生成された仕様内に (II) のモジュール操作の操作対象となるべき式が生成されている場合がある。そこで、(II) の操作を再適用することによって、新たに生成された式に関しても要求された変更を行うことができる。

4.4.2 動的処理

静的処理で修正されたモジュール操作 (I) を仕様 C に適用する。このときの公理の変更操作の解釈において以下に示す動的処理が行われる。

公理の変更操作は表 2 に示したとおりであるが、各モジュール操作の入力値には表 2 に示した式定義のほかに、論理式内の部分式を特定するための「置き換える根拠」となる式がある。ここで「置き換える根拠」について説明する。EVA は利用者が定義した仕様変更プロセスの情報のみから、既存の変更を代行しなければならない。そこで、EVA が複数の式の中から変更操作を施したい式を特定するために、特定を可能にする部分式を利用者が公理の式の中から操作時に選択しておく。これは、利用者が指定したモジュール操作をどのような状況で行いたいを示す宣言であり、各操作による変更の有無を決定する条件である。現在、根拠として選択できる式は公理に含まれる等式である。

EVA は「置き換える根拠」を用いて、モジュール操作の入力値を柔軟に解釈する方法を提供する。仕様 C 内の公理が「置き換える根拠」の式そのものを含んでいれば、その公理は変更対象となる。しかし、いつでも等しい式があるとは限らないので、EVA はパターンマッチにより変更対象公理を特定する。そして、特定された公理に対して次の 2 つの方法でモジュール操作の入力値を修正し、変更を実行する。

第一の方法はパターンマッチの結果を用いた修正である。たとえば、公理の変更操作「add_case」(表 2 参照) を関数「lend」の公理に対して行う場合を考える。利用者が「lend」の公理の「Ok c == card_search tag cs」という等式を根拠として選択し、2 つの場合分け

を表す式「book c == Book {title=t,author=a}」と「book c == Elt {title=t,author=a}」を入力したとする。これらの値がモジュール操作「add_case」の引数データとして「仕様変更プロセス」に格納される。すなわち、本モジュール操作は、この根拠を表す式が公理内に存在するときに、「add_case」の変更を行うことを意味している。さて、この操作を別の「仕様 C」に適用することを考える。図 11 の上段が仕様 C であり、「lend」に関する公理は 3 つの「axiom」宣言で成り立っている。今、「仕様 C」の「lend」の公理には「Ok c == card_search tag cs」および「card_search tag (rec l) == Ok c1」という等式がある (図 11 の四角形で囲まれた式)。前述の「置き換える根拠」の式とのパターンマッチにより、これらの等式を含む 2 つの「axiom」が変更対象公理として特定される。このとき、EVA はパターンマッチの結果である対応関係「c ⇔ c1, cs ⇔ rec l」により、場合分けの 2 つの式を「book c1 == Book {title=t,author=a}」と「book c1 == Elt {title=t,author=a}」として解釈し、本モジュール操作をこれらの公理に施す。図 11 の下段における四角形で囲まれた式が、変更された公理である。

第二の方法は型による解釈を用いた修正である。たとえば、公理の変更操作「applyexp_replace」(表 2 参照) を関数「lend」の公理に対して行う場合を考える。このモジュール操作は公理内の関数の参照パターンを新しいパターンで置き換えるものであり、変更対象式定義・置き換える式定義・置き換える根拠を引数とする。今、変更対象式定義を「Lend c」、置き換える式定義を「Copy c」として、公理内の「Lend c」を「Copy c」に置き換えることを考える。さて、「仕様 C」において「置き換える根拠」の式によって特定された公理内には「Lend c」という式がなかったとする。すなわち、どの部分式も「Lend c」とパターンがマッチしなかったので、EVA が指定された操作を「仕様 C」に適用しようとしても、「仕様 C」は利用者の意図に反して変更されない。そこで、EVA では対象式の型を推論して、「仕様 C」内で変更対象式を特定する。すなわち、「Lend c」の型を thing とすると「仕様 C」の「置き換える根拠」で特定された公理内において、型 thing を持つ式を特定して、これを「Copy c」と置き換える。

5. 評価

5.1 他の研究との比較

コンポーネントウェアの研究ではコードレス開発を目指して、柔軟に組合せが可能な部品とそれらを利用するアーキテクチャーを提案している¹⁾。しかし、い

```

axiom Nobook == card_search tag cs
implies lend tag (Network (host = Library (record = cs, reserve = t, id = i), loc = ls)) == \
(Network (host = Library (record = cs, reserve = t, id = i), loc = ls), Notice "Nothing")

axiom Ok c == card_search tag cs
implies lend tag (Network (host = Library (record = cs, reserve = t, id = i), loc = ls)) == \
(Network (host = Library (record = cards_update c tag cs, reserve = t, id = i), loc = ls), Lend c)

axiom No (tag, c) == card_search tag cs
implies exists l =>
member l ls == true
andalso card_search tag (getRec l) == Ok c1
implies lend tag (Network (host = Library (record = cs, reserve = t, id = i), loc = ls)) == \
(Network (host = Library (record = cs, reserve = rtable_add (getId l) tag t, id = i), loc = ls), Reserve \
(rtag_issue (getId l) tag c (second (order ls i (getId l) tag (Ok c1))))))
or else forall i =>
member l ls == true
implies forall c1 =>
card_search tag (getRec l) /= Ok c1
implies lend tag (Network (host = Library (record = cs, reserve = t, id = i), loc = ls)) == \
(Network (host = Library (record = cs, reserve = rtable_add i tag t, id = i), loc = ls), Reserve (rt\
ag_issue i tag c (plan c)))

axiom Nobook == card_search tag cs
implies lend tag (Network (host = Library (record = cs, reserve = t, id = i), loc = ls)) == \
(Network (host = Library (record = cs, reserve = t, id = i), loc = ls), Notice "Nothing")

axiom Ok c == card_search tag cs
implies book c == Book (title = t, author = a)
implies lend tag (Network (host = Library (record = cs, reserve = t, id = i), loc = ls)) == \
(Network (host = Library (record = cards_update c tag cs, reserve = t, id = i), loc = ls), Lend \
c)
or else book c == Eit (title = t, author = a)
implies lend tag (Network (host = Library (record = cs, reserve = t, id = i), loc = ls)) == \
(Network (host = Library (record = cards_update c tag cs, reserve = t, id = i), loc = ls), Copy \
c)

axiom No (tag, c) == card_search tag cs
implies exists l =>
member l ls == true
andalso card_search tag (getRec l) == Ok c1
implies book c1 == Book (title = t, author = a)
implies lend tag (Network (host = Library (record = cs, reserve = t, id = i), loc = ls)) == \
(Network (host = Library (record = cs, reserve = rtable_add (getId l) tag t, id = i), loc = ls), \
Reserve (rtag_issue (getId l) tag c (second (order ls i (getId l) tag (Ok c1))))))
or else book c1 == Eit (title = t, author = a)
implies lend tag (Network (host = Library (record = cs, reserve = t, id = i), loc = ls)) == \
(Network (host = Library (record = cs, reserve = rtable_add (getId l) tag t, id = i), loc = ls), \
Copy c1)
or else forall i =>
member l ls == true
implies forall c1 =>
card_search tag (getRec l) /= Ok c1
implies lend tag (Network (host = Library (record = cs, reserve = t, id = i), loc = ls)) == \
(Network (host = Library (record = cs, reserve = rtable_add i tag t, id = i), loc = ls), Reserve \
(rtag_issue i tag c (plan c)))

```

図 11 バタンマッチ

Fig. 11 Pattern match.

ろいろな人間が利用することから考えても、変更せずに利用できる部品をあらかじめ構築することは難しい問題である。

ソフトウェア開発の変換アプローチ^{3)~5),12)}や Design Rationale 技術^{2),10)}に見られるように、成果物だけでなくソフトウェアの開発プロセスを蓄積し利用することの必要性が認識されている。しかし、仕様と等価なプログラムを得るための変換アプローチでは、変換のルールといった一般的知識を再利用して仕様からプログラムを作成している。そこで、仕様の変更の種類によっては、既存の一連の変換手続きが新たな仕様によく適用できない場合もある。

一方、EVA は仕様やプログラムといったプロダクトだけでなく、仕様からプログラムを作成するプロセスや仕様を変更するプロセスを一体化した部品およびそれらを利用する枠組みを提供している。これらのプ

ロセスは構築するシステムごとの具体的なプロセスであるが、EVA の柔軟な解釈で異なる状況においてもこれを利用することができる。すなわち、既存の部品が利用できない場合でも、プロセスを利用した解釈によって、さまざまな利用者の予期せぬ要求に部品を適合させることができると考える。

5.2 例題によるシステムの評価

本稿で述べた EVA の 3 つの支援は人間の作業負担を軽減することとプログラムの品質を保証することを目的としている。そこで、文献 7) で示した「図書館貸し出しシステムの仕様変更」の EVA 上での実験結果を基に EVA の支援の有効性について評価する。

さて、EVA による「図書館貸し出しシステムの仕様変更」の実験過程はおよそ次のとおりであった。筆者が EVA システムを利用して図書館貸し出しシステム「Lib.」を開発する。そして、本システムを仮想的な 3

つの図書館「中央図書館」「港図書館」「足立図書館」に供給した。まず、「中央図書館」から書籍だけでなくソフトウェアの配布も行えるようにしたいとの要求を受けた。そこで、筆者は「中央図書館」からの要求をモジュール操作を用いて仕様変更プロセスとして定義し、まねる機能によって合成プロセスの部分的な修正を行い、システムの機能拡張を行った。このとき、モジュール操作数は6個であった。一方、「港図書館」と「足立図書館」からは自己の図書館に書籍がない場合、互いに書籍を融通し合えるようにしたいとの要求を受け、同様に「Lib.」を変更した。前述の図7がこのときの仕様変更プロセスであり、モジュール操作数は23個である。さて、しばらくして「中央図書館」からも「港図書館」と「足立図書館」の仲間入りをして書籍とソフトウェアの融通を行えるように機能拡張したいという要求が提出された。このとき、筆者は書籍もソフトウェアも他の図書館と融通し合える仕様を満たすシステムをEVAに対して新しい仕様の名前を与えるだけで構築することができ、EVAの変更の代行を実証することができた。

なお、図12が図7の仕様変更プロセスを「中央図書館」の仕様に適用した結果である。これは4.4.1項で述べた処理により、EVAが自動的に生成したものである。再帰的なデータ型の拡張操作により、「中央図書館」の仕様変更で用いた2つの操作(図12 axiom.14, axiom.15)が再適用されている。

EVAの計算機支援の有効性を実験結果から、次のように評価を行った。

- 利用者の作業負担

EVA上での実験は変更要求を分析し、変更項目を洗い出してから行った。洗い出した項目に従って変更操作を行うことができ、洗い出し項目に含まれない操作については「整合性の保証」により変更を行うことができた。ただし、試行錯誤する場面では、操作の取消しが必要である。

「中央図書館」のモジュール操作数は6個(ただし、因果関係による操作は0個)であり、「港図書館」のモジュール操作は23個(ただし、因果関係による操作は9個)であった。後者の方が変更項目が多く、まねる操作において人間が行わなければならない操作ステップ数も多かったが、どこを変えればよいかは明確であり、仕様変更プロセスで宣言した変換は自動的に行われ、人間と計算機の役割分担はうまく行われた。2章の例で見たように、仕様の書き方や変更の種類によってプログラムの修正方法は異なる。そこで、これらの前

```

Name : MID-Wide_Area_Service
Source Specification : EIS_Library
Target Specification : Net2_Library
OLD : (EIS_Library, EIS_Library)
NEW : (NET2_Library, Net2_Library)

Changes :
Signature Part :
val . 1 : valtype_replace
val . 2 : valtype_add
val . 3 : valtype_add
val . 4 : val_add
val . 5 : val_add
Function Part :
datatype . 1 : datatype_trans parallel
datatype . 2 : datatype_trans strnext
datatype . 3 : datatype_trans parallel
datatype . 4 : datatype_trans parallel
axiom . 1 : datapattern_trans parallel
axiom . 2 : datapattern_trans strnext
axiom . 3 : var_trans
axiom . 4 : funcpattern_trans
axiom . 5 : funcpattern_trans
axiom . 6 : datapattern_trans parallel
axiom . 7 : datapattern_trans parallel
axiom . 8 : add_case
axiom . 9 : argument_replace
axiom . 10 : argument_replace
axiom . 11 : argument_replace
axiom . 12 : argument_replace
axiom . 13 : argument_replace
axiom . 14 : add_case
axiom . 15 : applyexp_replace

LocalCausality :
from datatype . 1 to axiom . 1
from datatype . 2 to val . 1
from datatype . 2 to axiom . 2
from datatype . 2 to axiom . 3
from val . 2 to axiom . 4
from val . 3 to axiom . 5
from datatype . 3 to axiom . 6
from datatype . 4 to axiom . 7

OLD : (USER, User)
NEW : (USER, User)

Changes :
Signature Part :
val . 1 : valtype_replace
Function Part :

GlobalCausality :
from Net2_Library . datatype . 2 to User . val . 1

```

図12 代行された変更

Fig. 12 The change carried out by EVA.

提条件について十分検討し、多くの例を実験することによって、EVAの計算機支援の定量的な有効性の評価を行う必要がある。

本例の2つの変更の場合は前述のとおり「変更の代行」が実証できた。本稿では1章で述べたように、「人間にとって明らかな変更」を「EVAに蓄積された既存の変更」と考えた。しかし、既存の変更でなくても人間にとって明らかなことはある。このような変更を洗い出す必要がある。

- 品質

EVAでは合成プロセスは厳密に定義される。そこで仕様が不十分であったり、矛盾がある場合には合成の過程においてこれを発見することができた。このような作業によって、たとえ初期開発における人間の作業負担が多くても、正しい仕様の上にシステムを積み重ねていくことができるという利点がある。

5.3 今後の課題

前述の評価をふまえて、今後の課題について議論する。

仕様変更の局面を一般的に分類することは難しい。そこで、プリミティブな操作がどのような仕様変更の局面で有効であるかを例で評価しながら、不足しているプリミティブな操作について検討する。たとえば、モジュールを分割・統合する操作も必要であると考えるので、本稿で示した方針でこれらを追加する予定である。

プログラム開発支援環境が実用的になるためには、利用者の立場での利用形態を考察しなければならない。1つの方法は、自動化機能の充実である。まねることを自動的に行うためには、仕様変更プロセスの定型から合成プロセスの変更手続きを導出することが必要である。たとえば基本データ構造に基づく仕様変更プロセスの定型をライブラリとして蓄積し、これを利用することが考えられる。このような機能によって、利用者は単純でわずらわしい作業からさらに解放されることが期待される。もう1つの方法はインタフェースの改良である。GUIを用いて、利用者の邪魔にならない見たいものを見られるインタフェースを構築したいと考える。また、システムの全体構造をマクロにとらえるインタフェースも重要である。

6. おわりに

プログラム作成プロセスおよび仕様変更プロセスを用いてプログラムを修正する方法に基づいたプログラム開発支援システム EVA の実現について述べた。本稿では、さまざまな要求の変化に適合するために EVA が行う「真に使いやすい」プログラム開発支援の実現方式を示した。

これからのソフトウェア開発においては、良質なソフトウェアコンポーネントを簡単に安全に利用することが必要である。本稿で示したプログラム開発支援はこのような開発を可能にするための1つの基礎となりうると考える。

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受けて実施したものである。

参 考 文 献

- 1) 青山幹雄: コンポーネントウェア: 部品組み立て型ソフトウェア開発技術, 情報処理, Vol.37, No.1, pp.71-79 (1996).

- 2) Conklin, J. and Begeman, M.L.: gIBS: A Hypertext Tool For Exploratory Policy Discussion, *Proc. CSCW*, pp.140-152 (1988).
- 3) Feather, M.S.: Constructing Specifications by Combining Parallel Elaborations, *IEEE Trans. Software Engineering*, Vol.15, No.2, pp.198-208 (1989).
- 4) Goldberg, A.: Reusing Software Developments, *Proc. Forth ACM SIGSOFT Symposium on Software Development Environment*, 15, pp.107-119 (1990).
- 5) Johnson, W.L. and Feather, M.S.: Building an Evolution Transformation Library, *Proc. 12th International Conference of Software Engineering*, pp.238-248 (1990).
- 6) 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).
- 7) 松浦, 本位田: 仕様変更プロセスの効果的な再利用—まね方をまねる, 情報処理学会論文誌, Vol.36, No.11, pp.2666-2680 (1995).
- 8) Milner, R., Tofte, M. and Harper, R.: *The Definition of Standard ML*, The MIT Press (1990).
- 9) Paulson, L.C.: *ML for the Working Programmer*, Cambridge University Press (1991).
- 10) Potts, C.: A Generic Model for Representing Design Methods, *Proc. 11th International Conference of Software Engineering*, pp.217-226 (1989).
- 11) Sannella, D.T. and Tarlecki, A.: Program Specification and Development Standard ML, *Proc. 12th Ann. ACM Symp. on Principles of Programming Languages*, pp.67-77 (1985).
- 12) Wile, D.S.: Program Developments: Formal Explanation of Implementations, *Comm. ACM*, Vol.26, No.11, pp.902-911 (1983).

付 録

A.1 表のプログラム

(* lookup の ML によるプログラム. *)

```
fun lookup nil j = raise Lookup
| lookup ((k,e)::rest) j =
  if Order.ground_eq (k,j)
  then e
  else lookup rest j
```

(* update の ML によるプログラム. *)

```
fun update nil i x = [(i,x)]
| update ((k,e)::rest) i x =
  if Order.ground_eq (k,i)
  then (k,x)::rest
  else (k,e) :: (update rest i x)
```

A.2 表の仕様

(* 以下は, Extended ML による「表」の形式仕様である.

"signature" と "end" で囲まれた記述がモジュールのインタフェースの定義である。*)

signature TABLE =

sig

type key

type 'a table

val empty : 'a table

val lookup : 'a table -> key -> 'a

val update : 'a table -> key -> 'a -> 'a table

axiom forall ls => ls <> nil implies

forall k => List.length (lookup ls k) = 1

end

(* ここで "functor" と "end" で囲まれた記述がモジュール本体の定義である。"functor" はパラメータを持つモジュールであり、以下の functor は2つのパラメータモジュールを持つ。ここでは「表」を下記の「連想リスト」と「順序」の概念を用いて定義している。*)

functor TableFunc

(functor AssocListFunc (Order : ORDER)

: ASSOCLIST;

structure Order : ORDER) : TABLE =

struct

structure AssocList = AssocListFunc (Order)

type key = AssocList.key

type 'a table = 'a AssocList.assoclist

axiom lookup t k == AssocList.lookup t k

axiom update t k e == AssocList.update t k e

end

(* 「順序」の仕様。*)

signature ORDER =

sig

type ground

val ground_eq : ground * ground -> bool

val ground_leq : ground * ground -> bool

end

(* 「連想リスト」の仕様。*)

signature ASSOCLIST =

sig

type key

type 'a assoclist

val lookup : 'a assoclist -> key -> 'a

val length : 'a assoclist -> int

val update : 'a assoclist -> key -> 'a

-> 'a assoclist

end

functor AssocListFunc (structure Order : ORDER)

: ASSOCLIST=

struct

type key = Order.ground

type 'a assoclist = (key * 'a) list

axiom (exists (k,e) =>

List.contains (k,e) ls == true)

implies lookup ls k == e

axiom (forall (k,e) =>

List.contains (k,e) ls == false)

implies lookup ls k == raise Lookup

axiom Order.ground_eq (i,j) == true

implies lookup (update a i x) j == x

axiom Order.ground_eq (i,j) /= true

implies

lookup (update a i x) j == lookup a j

end

(平成8年6月12日受付)

(平成8年10月1日採録)



松浦佐江子 (正会員)

1955年生。1979年津田塾大学文学部数学科卒業。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。



来間 啓伸 (正会員)

1958年生。1981年広島大学理学部物理学科卒業。1984年同大学院理学研究科物理学専攻博士課程後期中退。同年(株)日立製作所入社。現在、同社システム開発研究所に所属。

主として、プログラミング言語、ソフトウェア工学の研究に従事。1994年より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。形式的手法に基づく仕様記述言語および仕様記述法に興味を持つ。日本ソフトウェア科学会、ACM、IEEE各会員。



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。1996年度大阪大学大学院および筑波大学非常勤講師兼任。主としてエージェント指向技術、オブジェクト指向技術の研究に従事。1986年度情報処理学会論文賞受賞。日本ソフトウェア科学会理事。著書「オブジェクト指向システム開発」(日経BP出版センター)など10点。