ソフトウェア工学 117－1
（1997．11．21）

書き換え規則の部品化に基づくソフト自動合成系開発の容易化

工藤 智広　　山之内 徹

NEC C&C メディア研究所

{*t-kudo, yamanouchi*}*@ccm.cl.nec.co.jp*

個々の開発プロジェクトに専用のプログラムジェネレータを開発するメタツールとして、ソフト自動合成シェルSOFTEXSHELLが利用され効果を上げているが、SOFTEXSHELLにおけるツールプログラミングが、LRP処理(リスト処理、再帰処理、パタンマッチング)を多用するなど、手続き型言語に慣れた開発者に馴染みにくいことが現場での実施拡大の障壁となっている。本稿では、このLRP処理の詳細を隠蔽して部品化することで開発を容易化する手法を提案する。本手法を実用ツールのトップダウン開発で試行したところ、詳細化過程の6割をこの部品の利用によってカバーできることがわかり、本手法を適切に実施することの効果が大きいことを確認した。

Generator Development Method using Reduction Rule Components

Tomohiro KUDO　　Toru YAMANOUCHI

C&C Media Research Laboratories, NEC Corporation

{*t-kudo, yamanouchi*}*@ccm.cl.nec.co.jp*

Software synthesis shell SOFTEXSHELL has been successfully applyed to various product software developments. However, SOFTEXSHELL often requires users to deal with LRP-processing(List processing, Recursion and Pattern matching), which often are unfamiliar to many software engineers. The authors propose a method for those users to utilize SOFTEXSHELL without dealing with LRP-processing by the use of LRP-processing components. The method was evaluated with a deployed program generator and the result showed that 9 LRP-processing components covered 60% of total refinement steps.

1 はじめに

大規模ソフトの開発において品質や納期などの目標を達成するためには、適切な開発支援機構の利用が不可欠である。ただし、実際の開発プロジェクトの様相は業務領域、開発体制、ソフトアーキテクチャなどにより千差万別であり、単なる汎用CASEツールの導入だけで対応することには限界がある。そこで目標達成のために、ドメイン(業務領域)の調査・分析を行い、目標達成に必要な要件を満たす開発方式および開発プロセスを設計し、それに合わせて汎用ツールの利用方法標準化や専用ツールの設計/開発を行うことが重要となる。このように、アプリケーションソフトウェア開発に先立って、その開発方法およびプロセスや開発支援ツールを開発する活動をメソッドエンジニアリング[2]と呼び、活動の実施者をメソッドエンジニア(以降MEと略す)と呼ぶ。

著者らは、こうしたメソッドエンジニアリングを実践するツールとして、項書き換え系をベースとしたソフトウエア自動合成シェルSOFTEXSHELL[5][8]の開発と現実のプロジェクトへの適用を行っており、生産性向上などの効果を上げている[4][7][3]。

MEはSOFTEXSHELLを利用してメソッドエンジニアリングを次の手順で行う[7]。

1. メソッド(開発者/プロダクト/ツールの関係を静的に捉えたもの)の設計
2. プロセス(メソッドから作成した時系列上の実行プラン)の設計
3. ツール(開発自動化/支援機能を持つプログラム)の開発

上記手順のうち、メソッドおよびプロセスの設計時では、主に要求/ドメイン分析評価、メソッド設計のスキルがMEに必要である。これは、通常、業務領域に関する知識やシステム開発経験をベースにして培われるものであり、個人的素養の他に経験に依存する度合いが大きい。

一方、ツール開発時には主にツール開発能力が必要であり、これを満たすためには、一般的なシステム開発能力とともに、ツール開発の開発方法論、記述言語、環境などに対する習熟がMEに必要となる。

SOFTEXSHELLを利用したメソッドエンジニアリングを開発プロジェクトに適用する場合は、上記スキルを持ったMEからなる数名のグループを結成する(各MEが全てのスキルを持つ必要はなく分担してもよい)が、現実にはこれらのスキルを十分に有するME候補者は必ずしも多くない。したがって、より多くのプロジェクトでSOFTEXSHELLによるメソッドエンジニアリングを実施するためには、MEのサポートを行ってその負担や必要なスキルレベルを軽減する技術が必要である。

著者らは、SOFTEXSHELLにおけるツールプログラミングで多用されるLRP処理(リスト処理、再帰処理、パターンマッチング)が多くの開発者にとって馴染みにくく、このことがMEに必要なツール開発スキルを上げている現実に着目した。本稿では、こうしたLRP処理をブラックボックス部品化して、その詳細な実現方法を知らなくてもツール開発を行えるようにすることで、MEの負担を減らすとともにME候補者を増やすことを目的とした手法を提案する。

2 SOFTEXSHELLによるツール開発

2.1 特徴

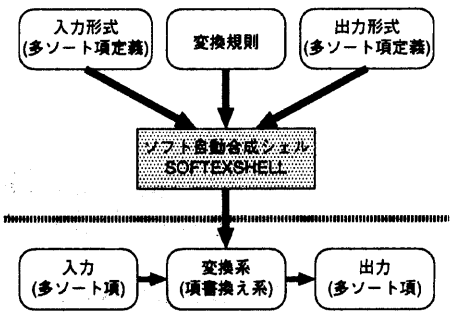

図1: SOFTEXSHELLの概念図

ソフトウエア自動合成シェルSOFTEXSHELLは、多ソート項書換えシステムに基づき、分野専用自動合成システム開発を容易化するメタツールである。SOFTEXSHELLによるツール開発では、入力形式と出力形式を表す多ソート項定義とその間の変換過程である変換規則をSOFTEXSHELLに与えることで、入力形式のインスタンスをそれに対応する出力形式のインスタンスに変換する変換系(ジェネレータ)を作成する(図1)。

SOFTEXSHELLによる変換系開発には次の特徴がある。

- 入力形式/出力形式あるいはその間の中間形式を多ソート項により定義するため、各形式

の構文を明確化することができ、自動的なチェックが可能となる。このことは他のツールやジェネレータと連携させるときの整合性保持に役立つ。

- 項書換え系では副作用がなく、変換処理が特定のコンテキストによって影響を受けることがないため、処理記述を安全に再利用しやすい。

- 項書換え系では処理の途中経過も項形式で表されるため、処理の確認が行いやすくプロトタイピングなどの段階的開発が容易に行える。

2.2 LRP 処理

SOFTEXSHELL による変換規則の記述例を以下にしめす。ME はツール開発時にこうした記述を行う[1]。

```
datatype Employee is employee of string *
  string * int;
 (* 「社員」は 名前(string)、所属名(string)、
                          TEL(int) からなる *)
op Select : list * string -> list;

rule Select([], office) ==> [];
                        (* 変換規則1 *)
rule Select([ employee(name, office, tel) ::
  T ], office)
==> [ employee(name, office, tel) ::
  Select(T, office) ];  (* 変換規則2 *)

rule Select([ employee(name, elsewhere,
  tel) :: T ], office)
 ==> Select(T, office); (* 変換規則3 *)
```

この例は「社員の集合から特定部署に所属する社員を抜き出した集合を作成する」という処理の例であるが、SOFTEXSHELL によるプログラミングでは、開発現場で多く利用されている C などの手続き型言語と比べて、この例に現れている以下の3種類の処理を多用する、という特徴がある[2]。

リスト処理 社員の集合など、要素が可変のデータ構造を表すのにリストを利用している。リストに対する基本操作は car、cdr などである ([H::T] と記述すると H が car、T が cdr を表す)。

再帰処理 「社員集合 (リスト) を先頭の要素から要素があるかぎり調べて行く」という繰り返し処理を実現するために、変換規則 1 ～ 3 からなる再帰を行っている。一般に SOFTEXSHELL は明示的な繰り返し構造を持たないため、再帰によって繰り返しを実現する。

パタンマッチング 「社員がある指定した部署にしているか」という条件判断を実現するために、変換規則 2 と 3 を用意し、どちらにパタンマッチするかにより変換の方法を切り換えている。一般に SOFTEXSHELL は明示的な分岐構造を持たないため、パタンマッチングによって分岐を実現する。

これらを、本稿では LRP 処理 (List processing, Recursion, Pattern matching) と呼ぶ。LRP 処理は、C などの手続き型言語を利用してシステム開発を行ってきた多くの開発者に取っては馴染みが薄いもので、プログラミングの習得に時間を要するものである。

3 LRP 処理の部品化

著者らは、前章までの内容を踏まえ、SOFTEXSHELL によるメソッドエンジニアリングにおける ME のサポート手段の一つとして、多くの開発者にとって馴染みにくい LRP 処理を部品化してジェネレータ開発を行う手法を提案する。

3.1 LRP 処理部品

LRP 処理部品とは、SOFTEXSHELL プログラミングでよく現れる LRP 処理のパタンを、パラメタ化した複数の変換規則の形で部品化したもので、具体的には次の要素からなる。

LRP 処理内容記述 その部品によりどのような処理を実現できるかを自然言語で記述したもの。

LRP 処理テンプレート 処理内容記述を実現する際に記述する変換規則群をパラメタ化したもの。テンプレート中のパラメタは「${ パラメタ名 }」として表す。

現在、図 2に示す 9 種類の LRP 部品を発見している。このうち、2 番の部品の内容を以下に示す

[1] なお、本稿では SOFTEXSHELL2[8] の記法を利用している (項定義は型として、変換規則は関数として定義する)。

[2] これらの処理は、Lisp などの関数型言語や Prolog などの論理型言語のように宣言的記述が主体となる言語においては基本的な概念であり、これらの言語に慣れた ME にとっては、SOFTEXSHELL プログラミングの習得は容易である。

No.	内容記述
1	ある変数の値による分岐
2	リストの各要素に対して処理を行い その結果をリスト化する
3	リストからある条件を満たすもの だけを選び出して残す
4	リストからある条件を満たすもの を1個選び出す
5	リストの各要素に対して処理を行い その結果のリストを結合
6	リストを2個作成し、それを結合
7	リストの先頭要素に対する処理と それ以外に対する処理の結果を結合
8	リストがなくなるまである処理を繰り返し
9	リスト中にある条件を満たすものが あるかどうかチェック

図 2: LRP 処理部品一覧

内容記述：「(2) リストの各要素に対して処理を行いその結果をリスト化する」

テンプレート:

```
op ${関数} : list * ${入力型列} -> list;
op ${処理関数} : ${入力リスト要素型} * ${入力型列}
  -> ${出力リスト要素型};
rule ${関数}([], ${入力変数列}) ==> [];
rule ${関数}([ h :: t ], ${入力変数列})
 ==> [ ${処理関数}(h, ${入力変数列})
 :: ${関数}(t, ${入力変数列}) ];
```

LRP 処理部品の利用は次の手順で行う。

1. 実装すべき機能を実現するのに合う「LRP 処理内容記述」を持った LRP 処理部品を選択する。

2. 選択した部品の「LRP 処理テンプレート」のパラメタを具体化し、変換規則として生成する。

LRP 処理テンプレートのパラメタの具体化では、例えば入力型のように仕様からほぼ自動的に決まるもの、変数名など衝突がなければ自動的に決定可能なものがある。したがってユーザは部品の選択と最低限の項目 (関数名などわかりやすい名前の方がよいもの) の決定だけ行い、後の作業はできるだけ自動化する方が、記述間違いの防止や詳細の情報隠蔽の点から望ましい。

3.2 LRP 処理部品の利用例

SOFTEXSHELL による実際のジェネレータ開発において、LRP 処理部品を利用する方法として、ここではジェネレータのトップダウン開発における利用方法を示す。

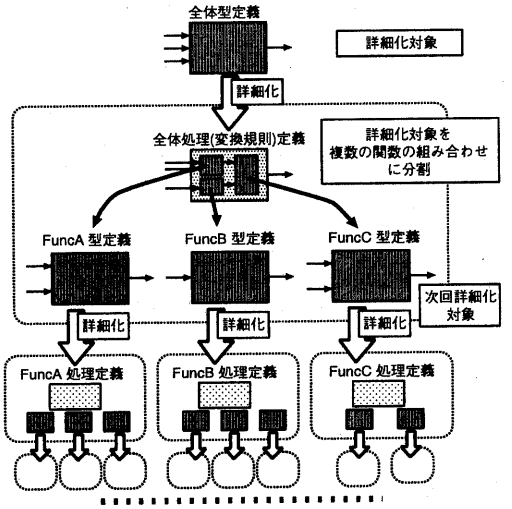

図 3: トップダウン開発のイメージ

SOFTEXSHELL におけるトップダウン開発とは、まず全体の変換過程を一つの関数として捉えて入出力の型を定義し、これを出発点に詳細化過程を繰り返すことで図3のようにジェネレータを完成させていくものである。詳細化とは、ある未定義関数の関数の処理を分析し、他の関数を組み合わせることでこの処理を実現していく作業である。もし組み合わせに利用した関数が未定義なものであれば、この関数が以降での詳細化対象候補となる。

ここで、次の処理を詳細化することを考える

「ある整数と整数のリストが与えられたときに、リストの各要素に整数の値をそれぞれ加える (関数名 MapAdd)」

例：$MapAdd(4,[2,1,5,4])->[6,5,9,8]$

この詳細化では、前掲の「(2) あるリストの各要素に対して処理を行い、その結果をリスト化する」LRP 処理部品を利用できる。ここで、各パラメタは次のように決定すればよい。

関数 MapAdd

入力型列 int

入力リスト要素型 int

出力リスト要素型 int

入力変数列 x

処理関数 Add

実際に具体化を行うと次のようになる。

```
op MapAdd : list * int -> list;
op Add : int * int -> int;
rule MapAdd([], x) ==> [];
rule MapAdd([ h :: t ], x)
 ==> [ Add(h, x) :: MapAdd(t, x) ];
```

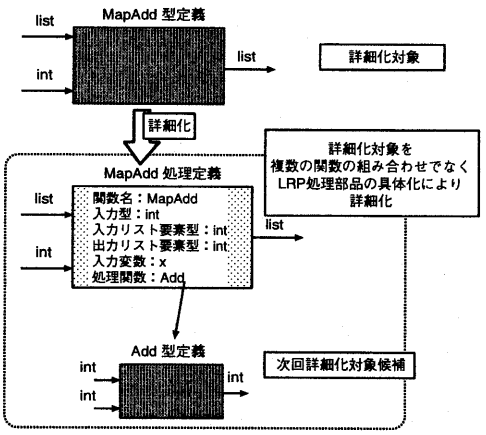

図 4: LRP 処理部品を利用した詳細化

この過程は、通常の詳細化方法 (図 3) と異なり、内容を隠蔽したブラックボックスを利用しそのパラメタを決めることで実質的に詳細化を行ったことに相当する (図 4)。ちなみに次の詳細化候補にパラメタとして指定した処理関数 Add が入ることになる。

このように LRP 処理によって実装する必要がある部分に対して、LRP 処理部品を利用しパラメタだけを与える、ということにより、内容の詳細に踏み込まなくてもトップダウンな詳細化を行える。

4 本手法の適用

本手法の実現可能性と有効性を確認するため、実際の開発プロジェクト向けに開発したジェネレータを題材にし、同じ機能を持つものを LRP 処理部品を利用して開発した。

4.1 適用プロジェクト事例

対象となったのは流通業の店舗系システムの開発プロジェクトである。このシステムのうち画面系

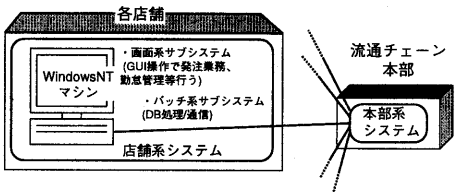

図 5: 流通業店舗システム

サブシステムでは、店舗にいるユーザが GUI 操作を行い、商品発注や在庫管理、勤怠管理などの店舗業務を行う。(図 5)[6]。

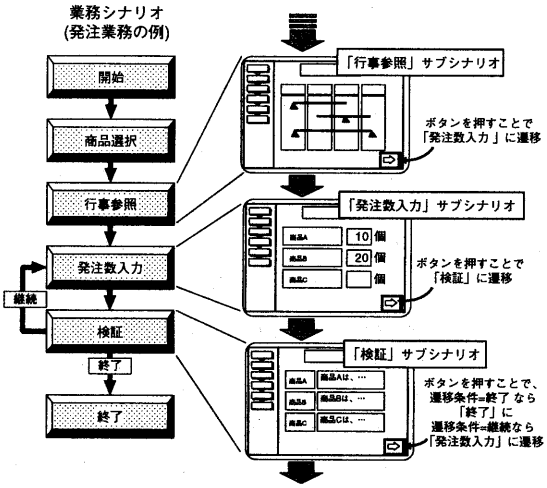

図 6: 業務シナリオの例

画面系サブシステムの開発では、各業務ごとに複数の GUI 画面間の流れ (業務シナリオ) の設計 / 製造を行い、それを元に、各 GUI 画面に対応するプログラム (サブシナリオ) の設計 / 製造を行う。

業務シナリオによってシステムを構成することで、1 個の画面だけで対応しきれない大量の情報をユーザにわかりやすく複数画面に分割して提示するとともに、業務の流れ (発注であれば、行事を確認してから実際の発注操作に入る、など) をスムーズに実現し、コンピュータシステムに不慣れなユーザであっても複雑な業務を要領よく実行できるようにしている (図 6)。

業務シナリオの実現機構は次の 3 つの要素からなっている (図 7)。

シナリオマネージャ シナリオ設定ファイルを参照

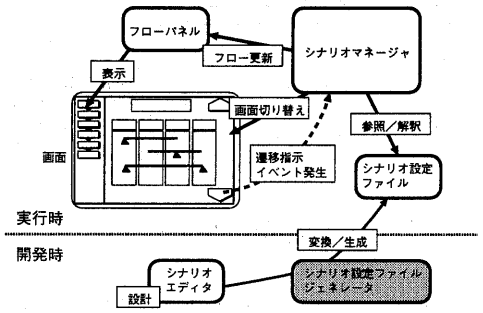

図 7: 業務シナリオ実現機構

し、そこに記述された情報を元に、ユーザが遷移を指定した場合 (通常次の画面に進むことを指示するボタンが押された場合)、その遷移先を決定し、画面を切り換えるとともに、後述するフローパネルの内容を更新する。

シナリオ設定ファイル 業務シナリオの遷移やフローの表示に関する情報をシナリオマネージャが簡単に読めるようにテキスト形式で記述したファイル (ini 形式を採用)。

フローパネル 業務シナリオの流れの一部をフロー図で表示するモジュール。表示するパスは、通過サブシナリオ、現在位置、実施予定サブシナリオを表しており、遷移状況によって内容が変更される。また、フローに分岐があり複数のパスがあっても現在位置やグラフ形状に基づく一定の規則により 1 つのパスがフローとして表示される。

このようにシナリオマネージャがシナリオ設定ファイルをその都度読み込んで解釈実行するという、インタプリタ方式を選択したのは、業務数が多く個別にコード埋め込みをすると保守の点で問題があったため、および、後での変更を容易にするためである。

なお、シナリオ自体は単純なグラフ構造であるが、実行時にこれより表示やパスに関する情報を取り出す必要がある。ところが、インタプリタ方式にした都合上、実行時の複雑な処理は速度低下を招く。したがって、シナリオに対しできるだけ前処理を行い、実行時の処理速度向上のために最適化しておく必要がある。このためシナリオ設定ファイルの内容は複雑なものとなり、直接手書きすることは、記述の手間と保守しやすさを考えると難しいと判断された。

そこで、グラフ構造を記述する専用エディタを用意し、ユーザはこれによって業務シナリオの設計 / 管理を行い、設計結果からシナリオ設定ファイルへの変換はシナリオ設定ファイルジェネレータが行うことにした。

このジェネレータは、具体的にはグラフ構造から次の情報を出力する。

サブシナリオ情報 各サブシナリオの名前や属性の一覧。

優先指定情報 通常、分岐ではいずれかの枝がデフォルト指定されるが、どの枝がデフォルト指定されているかの一覧。

フロー候補列情報 フローパネルに表示するフローの全候補の一覧 (業務シナリオ実行時には、遷移状況に応じてこの中のどれか 1 つのみが選択され、遷移が発生すると選択がやり直される)。

4.2 適用の実施方法

前節の自動合成系の開発への LRP 処理部品を適用は、次の要領で実施した。

- 既に開発済みの同機能の自動合成系 (以下、リリース版) の仕様だけを見て、全く最初から開発し直した。
- 図 3のようなトップダウンアプローチを実施し、適宜途中結果を確認しながらプロトタイピング的に開発を進めた。
- トップダウン詳細化において、LRP 処理部品化できそうな部分があれば部品化し、以降の開発においてこれらの部品が利用可能であれば積極的に利用した。
- 部品とは別に共通化可能な基本的処理 (サブシナリオを表す項からサブシナリオ名を取り出す処理など) はライブラリとしてとりまとめ、積極的に再利用を行った。

4.3 適用結果

本手法の適用結果を図 8に示す。これから、以下のことがわかる。

	適用結果	リリース版実績
詳細化実施回数	85	—
うちパターン利用回数	51	—
変換規則数	208	184
共通化変換規則数	63	97

図 8: 試行結果

- 詳細化回数 85 回のうち実に 6 割の 51 回に LRP 処理部品が利用されており、この事例に対して本手法が有効であることが確認できた。
- 試行結果の変換規則数がリリース版実績よりも多くなっている一方、ライブラリとして共通化した変換規則数は逆に少なくなっている。このことは、従来の開発方法では、トップダウンだけでなく、あらかじめ共通化できる部分を見越したボトムアップな開発を併用していたのに対し、試行結果ではトップダウンのみによる開発を行ったため、あまり粒度の大きな処理の共通化が行えなかったことに起因すると思われる。

No.	出現回数	全詳細化中割合	利用機会中割合
1	22	0.26	0.43
2	9	0.11	0.18
3	8	0.09	0.16
4	5	0.06	0.10
5	3	0.04	0.06
6	1	0.01	0.02
7	1	0.01	0.02
8	1	0.01	0.02
9	1	0.01	0.02

図 9: LRP 処理部品の利用状況

LRP 処理部品の利用状況を図 9に示す。このように本試行では 9 種類の部品が使用されているが、最も利用したのが、分岐を表す以下の部品 (1 番) で、これは、パタンマッチングによって処理の分岐を行うものである。

内容記述: 「(1) ある変数の値による分岐」
テンプレート:

```
op ${関数} : ${判定値型} * ${入力型列} -> ${出力型};
op ${処理関数 1} : ${入力型列} -> ${出力型};
op ${処理関数 2} : ${入力型列} -> ${出力型};
rule ${関数}(${判定値 1}, ${入力変数列})
  ==> ${処理関数 1}(${入力変数列});
rule ${関数}(${判定値 2}, ${入力変数列})
  ==> ${処理関数 2}(${入力変数列});
  ...
```

この部品の出現数が多いのは、本事例において、場合分け処理が本質的に多く、かつ、SOFTEXSHELL での場合分け処理はこの部品のように記述する必要があったためである。

その他、開発時に気づいた点を以下に挙げる。

(1) 開発の当初、LRP 処理部品があまり揃っていない状況では、まず部品を意識せず 1 ステップの詳細化を行い、結果として部品化できそうなところをみつけて部品化する、という形で作業を進めて行ったが、ある程度、部品が揃うとともに部品の利用方法になれてきた時点では、詳細化を行う処理の内容からどの部品でこれを実現できるかをまず考え、部品化できない残りを通常の詳細化で補う、という形に変化していった。

このような LRP 処理部品主導の詳細化をさらに検討することにより、恣意的でアドホックな詳細化に代わり、より手順の明確な詳細化方法を考えることができるものと思われ、ME のツール開発を積極的にガイドする支援方式 / ツールの開発に役立つと予想できる。

(2) 例えば、業務シナリオのグラフ構造からある条件を持つパスの集合を作成する場合、次のような探索処理を行うことになる (本事例開発中には、計 3 回出現した)。

1. 初期設定 (探索位置の初期値決定)。
2. パスを伸ばして行く候補となる枝のリスト作成。[3 番の部品利用]
3. 現在のパスが条件を満たすかどうか判断し、満たしていれば、探索結果リストに現在のパスを追加する。[1 番の部品利用]
4. 探索終了条件が成立しているか判断し、満たしていれば、上のレベルに戻る。[1 番の部品利用]
5. 候補リストの先頭からを取り出し、位置とパスの更新を行って、1 の処理を呼び出す。候補リストが空ならば上のレベルに戻る。[5 番の部品利用]

こうした探索処理は、各ステップを切り出してそれぞれを徐々に詳細化することも原理的には可能であるが、探索処理として全体像を把握していないと正しい詳細化は不可能である[3]。

また、一般に大規模なジェネレータを開発する場合には、全体としての詳細化の階層が非常に深くなるため、単体の部品レベルだけでは、詳細化を進めることや開発結果を理解することが難しい。

このようなことから、LRP処理部品単体だけでなく、それらを組み合わせたより粒度の大きい部品を用意することでさらに高いレベルでジェネレータ開発を支援できることがわかった。

5 関連研究

本手法のLRP処理部品同様、小規模から中規模のまとまった処理やデータ構造を部品化する考え方として、設計パタン[1]がよく知られている。本項のLRP処理部品は、言語依存であるため、厳密にはパタンでなくイディオムと呼ぶべきものである。ただし、実際にある程度の経験を積んだプログラマが問題解決に用いているものを明確化し再利用しようとする発想から見れば、方向的には同じと言える。提案手法では、これをジェネレータのトップダウン詳細化という限定した状況設定で利用しているところが特徴となっている。

6 結論

本稿では、SOFTEXSHELLによるメソッドエンジニアリングを実践する際に、多くの開発者にとって不慣れと思われるLRP処理(リスト、再帰、パタンマッチング)に着目し、これを部品として用意しておき、開発に利用する手法を提案した。

このLRP処理部品を、ジェネレータのトップダウン開発における詳細化過程で利用することで、LRP処理の実現内容を隠蔽しながら開発が行えることを示し、LRP処理に未習熟なMEにとってのジェネレータ開発を容易化できることを示した。

本手法を実用レベルのジェネレータに対して試験的に適用した結果、実際に9種類のLRP処理部品を利用して開発が行え、トップダウン詳細化回数のうち6割にLRP処理部品が利用できたことから、同事例において、本手法を適切に実施することで大きな効果が期待できることがわかった。

また、本手法により開発を行った結果から、LRP処理部品主導による詳細化方法や、探索処理などの定型的な処理に対する複合的な部品の導入など、本手法をさらに効果的に実施するための方向性を見いだすことができた。

今後は、これらの方向性の検討をさらに進め、より多様かつ大規模なジェネレータ開発に適用して評価を行っていく予定である。

[3] 事実、本事例開発中において探索処理を実装する際には、かなり先までのトップダウン詳細化過程を見極めて対応した。

参考文献

[1] GAMMA, E., HELM, R., JOHNSON, R. and VLISSIDES, J. *Design Patterns : Element of Reusable Object-Oriented Software*, Addison Wesley (1995).

[2] NUSEIBAH, B., FINKELSTEIN, A. and KRAMER, J. Method Engineering for Multi-Perspective Software Development, *Information and Software Technology Journal*, **38**, 4 (1996), 267–274.

[3] SATO, A., MIKI, M., YAMANOUCHI, T. and WATANABE, M. Software Synthesis for Trade-off Design, 11th KBSE (1996).

[4] SATO, A., TOMOBE, M., YAMANOUCHI, T., WATANABE, M. and HIJIKATA, M. Domain-Oriented Software Process Re-engineering with Software Synthesis Shell SOFTEX/S, 10th KBSE (1995).

[5] YAMANOUCHI, T., SATO, A., TOMOBE, M., TAKEUCHI, M., TAKAMURA, H. and WATANABE, M. Software Synthesis Shell SOFTEX/S, 7th KBSE (1992).

[6] 工藤, 佐藤, 山之内 ソフトウエア自動合成シェルSOFTEXSHELLを利用したプログラムジェネレータのプロトタイピング方式, 情報処理学会第54回全国大会 4C-3 (1997).

[7] 佐藤, 友部, 山之内, 渡辺 ソフトウェア自動合成シェルSOFTEXSHELLを用いたドメイン指向ソフトウェア開発プロセス設計, 情報処理学会ソフトウェア工学研究会, **96**, 108 (1996), 9–16.

[8] 友部, 佐藤, 山之内 項書換えシステムのソフトウェア自動合成への適用, 情報処理学会ソフトウェア工学研究会, **96**, 112 (1996), 65–72.