ソフトウェア工学 112－9
（1996．11．15）

# 項書換えシステムのソフトウェア自動合成への適用

友部 実、佐藤明良、山之内徹
tomobe@swl.cl.nec.co.jp, a-sato@swl.cl.nec.co.jp, yamano@swl.cl.nec.co.jp
NEC C&C 研究所
〒216 川崎市宮前区宮崎 4-1-1

我々は項書換えシステムに基づくソフトウェア自動合成シェル SOFTEXSHELL を開発し、特定の領域固有のプログラムジェネレータを生成することによりソフトウェア開発支援環境を構築している。本稿ではソフトウェア開発プロセスへ自動合成システムを適用する際の問題点を整理し、従来行ってきたジェネレータの設計プロセスについて述べる。この設計プロセスで得られたジェネレータのモデルが、項書き換え系の一つであるコンストラクタシステムで表現できることを示し、コンストラクタシステムに基づくソフトウェア自動合成シェル SOFTEXSHELL-2 の実現を行った。SOFTEXSHELL-2 を用いることにより、変換プロセスの再利用やプロトタイピングにより、ジェネレータの開発を効率良く行えることが可能となった。

# Application of Term Rewriting System to Software Synthesis System

Minoru Tomobe,Akiyoshi Sato,Toru Yamanouchi
NEC C&C Research Labs.
4-1-1, Miyazaki,Miyamae-ku,Kawasaki,Kanagawa, 216 Japan

This paper describes the design and implementation of software synthesis shell SOFTEXSHELL-2. We have developed some domain-specific software development environment with software sysnthesis shell SOFTEXSHELL. The SOFTEXSHELL-2 is being developed to improve generator development productivity. SOFTEXSHELL-2 uses a Constructor System (a kind of Term Rewriting System) for its transformation mechanism. We describe the characteristic of Constructor System and how it works for improvement of generator development productivity.

## 1 はじめに

ソフトウェア開発の効率化を目的として、ソフトウェア自動合成システムやプログラムジェネレータが数多く適用され、幾つかのシステムが実績を上げている[3][4]。ソフトウェア開発プロセスにソフトウェア自動合成システムを適用する際には、対象となるプログラムを生成することのみならず、自動合成システムを含めたソフトウェア開発プロセスを再設計し、仕様やテストデータといったソフトウェア開発に必要な中間生成物も生成対象とすることにより、ソフトウェア開発全体の生産性を向上することができる[1]。

しかしこれを実現するためには、開発プロセス毎に専用の自動合成システムを数多く開発する必要があるため、自動合成システム自体の構築を効率良く行わなければツール開発の工数を含めた開発プロセス全体の生産性の向上を図ることはできない。このような目的を実現するため、我々は特定の領域固有のプログラムジェネレータを生成するソフトウェア自動合成シェルSOFTEXSHELL[2]を開発し、幾つかの分野で適用を行って来た。

本稿では、ソフトウェア開発プロセスへ自動合成システムを適用する際の問題点について述べ、この問題点を解決するために我々の用いたジェネレータ設計プロセスについて説明する。次に、この設計プロセスで得られたジェネレータの仕様からジェネレータを容易に実現する機構としてコンストラクタシステムを用いたソフトウェア自動合成シェルSOFTEXSHELL-2の機能について簡単に説明する。最後にSOFTEXSHELL-2を用いてジェネレータを開発することにより、ジェネレータの機能を実現する変換プロセスの再利用や、変換プロセス毎の独立したプロトタイピングが可能になることを示す。

## 2 ソフトウェア開発プロセスへの自動合成システムの適用

従来提案されているソフトウェア自動合成システムの多くは、特定の領域で設計やコーディングといった単独のフェーズに対して適用されたものであり、ソフトウェア開発プロセス全体への適用は考慮されていない。

我々は特定の問題領域毎に開発プロセスを分析することによって、プロセスの中で人手で行う作業とプログラムジェネレータといったツールを用いて行う作業を抽出し、ツールを含めた開発プロセス全体を再構成することによってソフトウェア開発全体の生産性の向上を図る手法を提案した[1]。

例えばあるGUIアプリケーションプログラムの開発プロジェクトにおいて、再構成を行う以前は図1に示す開発プロセスで作業を行っていた。このプロジェクトではアプリケーションの実現環境が特殊だった為、全ての作業を手作業で行っており、また各開発工程も全て逐次的に行われていた。

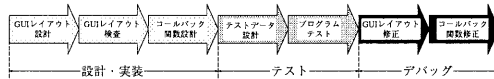


図 1: 再構成前の開発プロセス

この開発プロセスを自動合成ツールを考慮して再構成した結果、図2に示すように、各種ジェネレータを各工程で用いることにより、並行作業が可能なプロセスとなった。この開発プロセスでは図3に示すような開発支援環境により、開発プロセス全体の支援が可能となり、大幅に開発期間を短縮することに成功した。

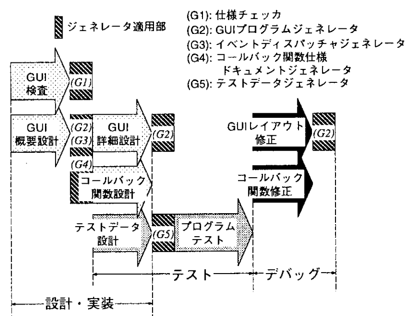


図 2: 再構成後の開発プロセス

図3に示すように、再構成後の開発プロセスではジェネレータは対象となる領域のプログラムを生成するだけでなく、開発を支援する為の帳票の出力など、開発プロセス内で必要な中間生成物の自動生成も行い開発プロセス全体を支援する。しかしこれを実現する為には各々の開発プロセス毎に複数のジェネレータを開発する必要があり、個々のプログラムジェネレータを効率良く開発することが、ツールの開発工数を含めた開発プロセス全体の生産性を向上する為のポイントとなる。

実際の開発プロジェクトにおいて、プログラムジェネレータといった開発支援環境の構築はアプリケーションプラットフォームの開発と並行して行われることが多い。この為、アプリケーション開発者にプラッ

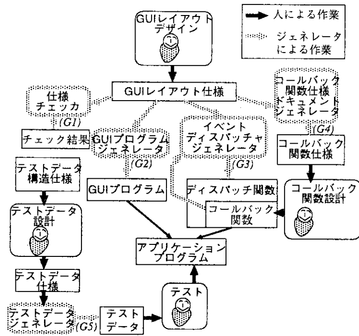


図 3: 再構成後のツール構成

トフォームの完成と同時に開発支援環境を提供する為には、プラットフォームの仕様が決定する以前に開発支援環境の設計を進めなければならない。この為、プログラムジェネレータといったプラットフォームの実装に依存するツールはプロトタイピングにより開発を進める必要がある。また前述の手法では、多くのジェネレータを同時に開発する為、各ジェネレータに共通する機能を共有し個々のジェネレータの開発工数を削減するとともに、複数のジェネレータ開発者が並行して開発作業を進める必要がある。

我々は入力仕様から出力を得る過程を変換プロセスとしてモデル化し、以下のような特長を持つジェネレータ設計プロセスにより、プログラムジェネレータの開発を行ってきた。

- 各ジェネレータの機能を複数の変換プロセスに分割し、共有可能な変換プロセスは共有する。
- 各変換プロセスを独立してプロトタイピングにより実装し、ジェネレータの開発を並行して進める。

次節ではこのジェネレータ設計プロセスについて説明する。

## 3 ジェネレータの設計プロセス

### 3.1 ジェネレータのモデル化

入力仕様から出力を得る過程は図 4に示すモデルで表現される。パーザは与えられた入力仕様 $S$ を入力仕様の構文を規定する文脈自由文法 $G$ を用いて構文解析を行い、抽象構文木 $S_t$ を得る。次に、これを変換プロセス $f$ によって、ジェネレータの出力に対応する抽象構文木 $O_t$ に変換する。プリンタは抽象構文木 $O_t$ から出力 $O$ の構文を規定する文脈自由文法 $G'$ を用いて出力 $O$ を出力する。

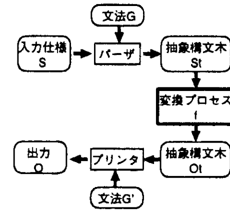


図 4: ジェネレータのモデル化

ジェネレータの設計プロセスは

1. 入出力を規定する文法 $G, G'$ の設計
2. 変換プロセス $f$ の設計と詳細化

によって構成される。

### 3.2 ジェネレータの設計プロセス

我々はジェネレータの設計プロセスを以下の 4 つのフェーズに分けて定義した。ここでは各フェーズで行う作業と、その結果得られる設計帳票の記述について説明する。

**分析フェーズ**

2節の例で示したように、開発プロセスの中でジェネレータが適用可能な部分の抽出を行う。次にジェネレータの入出力、並びに機能の分析を行い、その結果をデータフローダイアグラムによって表現する。データフローダイアグラムは変換プロセス、データフロー、データストアから構成される。データストアは入力仕様や出力プログラム、出力データ等を表し、変換プロセスはジェネレータの機能を実現する関数を表す。データフローは変換プロセス間、プロセス・データストア間の入出力を接続する。例えば図 3に示すプログラムジェネレータのデータフローダイアグラムは図 5に示すようなものとなる。

**変換プロセス設計フェーズ**

このフェーズでは、分析フェーズで得られたデータフローダイアグラムからダイアグラム中に現れる変換プロセスの詳細化と、入出力となるデータストアの定義、並びにデータフローの型の設計を行う。

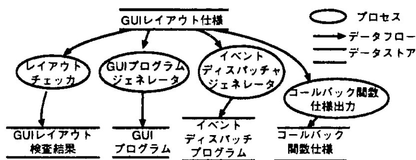

図 5: 変換プロセスのデータフロー

**変換プロセスの詳細化** 分析フェーズで定義した変換プロセスの機能を詳細化し、変換プロセスを複数の変換プロセスに分割する。

**入出力の定義** 分析フェーズで定義したデータストアの詳細な定義として、3.1節に示したモデル化に基づき、入力仕様、出力プログラムの文法 $G$ を定義する。

**データフローの型定義** 変換プロセスの詳細化の結果や、入出力の文法定義を元に、ダイアグラム中のデータフロー上を流れるデータのデータ型の設計を行う。

このフェーズでは、例えば図5中のイベントディスパッチャジェネレータは図6に示すような複数の変換プロセスに分割され、データストア、データフローそれぞれの型が定義される。

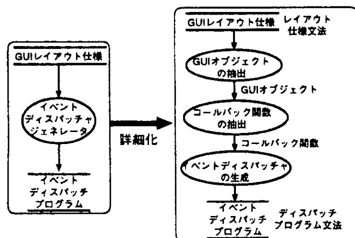

図 6: 変換プロセスの詳細化

**変換プロセス再構成フェーズ**
このフェーズでは、変換プロセス設計フェーズで得られた詳細化されたデータフローダイアグラムで、変換プロセス間のデータフローや、データストアの型の詳細化を行いながら、データの共有が可能な部分を抽出する。データの共有が可能な部分は変換プロセスを共有しプロセスの再構成を行う。図7に、図5を詳細化し、変換プロセスの再構成を行ったデータフローダイアグラムを示す。この例では分析の結果、GUIレイアウト仕様からGUIオブジェクトを抽出する変換プロセスと、コールバック関数の抽出を行う変換プロセスが共有された。

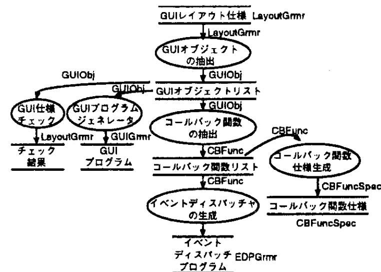

図 7: 再構成された変換プロセス

**実装フェーズ**
データフローダイアグラム中の各変換プロセスの実装を行う。

本節で述べたジェネレータ設計プロセスは次のような特長を持つ。

- 変換プロセスを共有することにより、ジェネレータを構築する為に必要な変換プロセスの開発量を削減する。
- 各変換プロセスの入出力の文法を厳密に規定することにより、各モジュール毎に独立した開発を可能にする。

以上の特長を実現するため、我々は従来のSOFTEXSHELLにコンストラクタシステムを取り入れたSOFTEXSHELL-2を開発した。次節では、このコンストラクタシステムについて説明し、SOFTEXSHELL-2を用いてジェネレータを実装することにより、上記特長が実現できることを示す。

## 4 コンストラクタシステムに基づくソフトウェア自動合成シェルSOFTEXSHELL-2

本節ではコンストラクタシステムについて説明し、SOFTEXSHELL-2におけるコンストラクタシステムの記述形式について述べる。次にSOFTEXSHELL-2の特長とリダクションの実行例を示す。

### 4.1 コンストラクタシステム

ここでは、コンストラクタシステム[5]の定義について述べる。

**定義 4.1(多ソート指標)** 多ソート指標 $\Sigma$ とはソート記号集合 $S$ と関数記号集合 $F$ の組 $\Sigma \stackrel{\mathrm{def}}{=} < S, F >$ である。ただし、$\forall f \in F; f_{s_1 \dots s_n s}; n \geq 1; s_1, \dots s_n, s \in S$ である。ここで、ソート列 $s_1 \dots s_n$ は関数記号 $f$ の定義域、$s$ は $f$ の値域を示す。□

**定義 4.2(変数)** 変数集合 $V$ は、ソート記号 $s \in S$ を持つ変数 $v_s$ の集合である。□

**定義 4.3(多ソート項)** 多ソート指標を $\Sigma =< S, F >$、変数集合を $V$ とするとき、ソート $s$ の多ソート項集合 $T_s(\Sigma, V)$ は以下のように定義される。:

1. $v_s \in V, s \in S$ ならば $v_s \in T_s(\Sigma, V)$
2. $f_{s_1 \dots s_n s} \in F;\ t_1 \in T_{s_1}(\Sigma, V), \dots, t_n \in T_{s_n}(\Sigma, V)$ ならば $f(t_1, \dots, t_n) \in T_s(\Sigma, V)$

多ソート項集合 $T(\Sigma, V)$ とは $T(\Sigma, V) \stackrel{\mathrm{def}}{=} \bigcup_{s \in S} T_s(\Sigma, V)$ である。□

**定義 4.4(多ソート項書換え規則)** 多ソート項集合を $T(\Sigma, V)$ とし $l, r \in T(\Sigma, V)$ かつ $r$ に現れる変数は $l$ に現れるとするとき、$l \rightarrow r$ を多ソート書換え規則という。□

**定義 4.5(コンストラクタシステム)** 多ソート項集合 $T(\Sigma, V), \Sigma =< F, S >$ と、この多ソート項上の書換え規則 $R = \{l \rightarrow r \mid l, r \in T(\Sigma, V)\}$ で定義された項書換え系 $< \Sigma, R >$ のうち、多ソート項集合 $T(\Sigma, V)$ の関数記号 $F$ が、書換え規則 $l \rightarrow r \in R$ の左辺 $l = f(t_1, \dots, t_n)$ の最外部に現れる関数記号 $f$ の集合 $D$ と、それ以外の関数記号の集合 $C$ に分割される時、$< \Sigma, R >$ をコンストラクタシステムという。この時 $D$ をオペレータ、$C$ をコンストラクタと呼ぶ。□

## 4.2 SOFTEXSHELL-2 の構文

SOFTEXSHELL-2 ではコンストラクタシステム $< \Sigma, R >$(但し $\Sigma =< C \cup D, S >$) を次のように表記する。

**指標 $\Sigma$ の宣言:** datatype 文では新しいソート $s_1$ を導入し、ソート $s_1$ を値域、ソート $s_2$ 定義域に持つ、コンストラクタ $c$ を宣言する。op 文ではソート $s_2$ を値域、ソート $s_1$ を定義域に持つオペレータ $d$ を宣言する。

datatype $s_1 = c$ of $s_2$;
op $d : s_1 -> s_2$;

**書き換え規則の宣言:** オペレータ $d$ に対応する書き換え規則 $lhs \rightarrow rhs$ を rule 文で定義する。

rule $lhs$ ==> $rhs$;

**モジュールの宣言:** $< \Sigma, R >$ をモジュール *name* として module 文で宣言する。ここで、*signature* は指標 $\Sigma$ の宣言を表し、*rule* は書き換え規則の宣言を表す。但し $\Sigma$ で宣言されたオペレータ $d \in D$ に対して、$d(t_1, \dots, t_n) \rightarrow r \mid t_i \in T(\Sigma, V); i = 1 \dots n$ となるような書き換え規則が無い場合、$d \in C$ と見なされる。また、指標 $\Sigma$ のみ ($< \Sigma, \phi >$) を記述、もしくは書き換え規則のみ ($< \phi, R >$) を記述したモジュールを定義することもできる。

module *name* is *signature rule* ... *rule* end;

**モジュールの参照:** モジュール *name* $=< \Sigma', R' >$ を、open 文の宣言されたモジュール $M =< \Sigma, R >$ で参照する。open 文の定義により、$M =< \Sigma \cup \Sigma', R \cup R' >$ となる。

open *name*;

**リダクションの実行:** モジュール *name* $=< \Sigma, R >$ をコンストラクタシステムとして、項 $t \in T(\Sigma, \phi)$ のリダクションを行う。

reduction $t$ with *name*;

## 4.3 SOFTEXSHELL-2 の記述例

図 8にリストの結合を行うコンストラクションシステムを SOFTEXSHELL-2 で記述した例を示す。この例ではモジュール List とモジュール Append が定義されている。モジュール List ではリスト構造を表すコンストラクタと、リスト上の操作を表すオペレータが宣言されている。モジュール Append では、open 文によってモジュール List からリストを表現するコンストラクタ cons,nil やリストの結合を行うオペレータ append の宣言が参照され、オペレータ append に対応する書換え規則を rule 文によって定義している。

```
module List is
datatype 'a list = nil
                 | cons of 'a list * 'a list;
op append : 'a list * 'a list -> 'a list;
end;
module Append is
 open List;
 rule append(nil,X) ==> X;
 rule append(cons(H,T),X) ==> cons(H,append(T,X));
end;
```

図 8: リスト結合の規則の例

リダクションの実行はこのモジュールを単位として行われる。SOFTEXSHELL-2 はリダクションの

制御や、リダクションの結果を大域変数に代入などを行う実行時制御スクリプト言語を持つ。例えば図9は、リストを表す項をモジュール Append でリダクションを実行した例である。ここではモジュール Append で定義されたオペレータ append の書き換えが実行され、リストの結合が行われている。

```
reduce append(cons(1,cons(2,nil)),
              cons(3,nil)) with Append;
--> cons(1,cons(2,cons(3,nil)))
```

図 9: リダクションの実行

図 10の例ではモジュール Foo ではリスト構造を宣言するモジュール List を参照しているが、モジュール Foo ではオペレータ append の規則が定義されていない為、リダクションの実行時には append もコンストラクタとして扱われる。このため、SOFTEXSHELL-2 では全てのオペレータに対して書き換え規則の定義を行う前に部分的な変換の実行を行うことが可能である。

```
module Foo is
open List;
op f : int list * int list -> int list;
rule f(x,y) ==> append(x,y);
end;
let x = reduce append(f(cons(1,nil),cons(2,nil)),
                      cons(3,nil)) with Foo;
--> append(append(cons(1,nil),cons(2,nil)),
          cons(3,nil))
reduce x with Append;
--> cons(1,cons(2,cons(3,nil)))
```

図 10: リダクションの部分実行

# 5 変換プロセスの SOFTEXSHELL-2 を用いた実装

## 5.1 データフローダイアグラムとの対応

データフローダイアグラムの各要素は SOFTEXSHELL-2 では次のように表現される。

**データストア** データストア毎にモジュールとして記述する。このモジュールでは、入出力の文法規則 $G$ から定義されるコンストラクタを datatype 文で定義する。

**変換プロセス** 変換プロセス毎にモジュールとして記述する。このモジュールでは

- 変換プロセスの入出力となるコンストラクタを定義するモジュールの参照
- 変換プロセスを定義するオペレータの宣言
- 上記オペレータに対応する書き換え規則

が定義される。

**データフロー** 変換プロセスを定義するオペレータの書き換え規則の右辺にオペレータの呼出し関係として記述される。

以下、これらの定義方法について詳細に述べる。

### 5.1.1 データストアの定義

3.1節で示したように、プログラムジェネレータの入出力を表すデータストアは文脈自由文法 $G$ により定義される。SOFTEXSHELL-2 では文脈自由言語 $L(G)$ を多ソート項 $T(\Sigma,\phi)$ にマッピングし、この多ソート項により入出力を表現する。文脈自由言語 $L(G)$ と多ソート項 $T(\Sigma,\phi)$ のマッピングの定義を以下に示す。

**定義 5.1**(文脈自由言語 $L(G)$ と多ソート項のマッピング)[6]

1. 指標 $\Sigma_G = < S, F >$ を文脈自由文法 $G = (N,T,S_t,P)$ より、つぎのように定義する。ここで、$N$ は非終端記号、$T$ は終端記号、$S_t$ は開始記号、$P$ は $N\cup \rightarrow (N\cup T)+$ であるような生成規則を表す。
   - $s\in S$ iff $s\in (N\cup T)$,
   - $s : s_1 \times ... \times s_n \rightarrow s \in F$ iff $s \in (N\cup T)$ かつ $(s\rightarrow s_1...s_n)\in P$
2. 多ソート項 $t \in T(\Sigma_G,\phi)$ は文脈自由文法 $G$ により得られる文脈自由言語 $L(G)$ の構文解析木の一つである。

SOFTEXSHELL-2 ではこの多ソート項 $T(\Sigma_G,\phi)$ をコンストラクタシステムのコンストラクタ $C$ で構成される項 $T(< S,C >,\phi)$ として定義する。例えば図 7の GUI プログラムの文法が、図 11に示す文脈自由文法 $GUIGrmr$ で定義される場合、文法 $GUIGrmr$ の非終端記号 $GUIGrmr$,*statement* に対応したソートと、終端記号 $if$ に対応したコンストラクタが datatype 文を用いて定義され、このデータ構造を用いて入力仕様の抽象構文木を表現する。

### 5.1.2 変換プロセスの実装

変換プロセスは変換を行う書き換え規則を含むモジュールとして実装される。例えば図 12に示すイベントディスパッチプログラムジェネレータのデータフローダイアグラムは、4つの入出力を表すデータスト

```
○GUI プログラムを表す文法 GUIGrmr
GUIGrmr     = <func_decl>
            | <func_decl> <GUIGrmr>
func_decl   = <identifier> ( <arg> )
              { <statement> }
statement   = if <expr> then <statement>
                         else <statement>
...
○datatype 文による文法 GUIGrmr の実現
module GUIGrmr_mod is
 datatype GUIGrmr =
     func_decl of IDENTIFIER * ARG * STATEMENT
   | seq of GUIGrmr * GUIGrmr
 and STATEMENT =
    if of EXPR * STATEMENT * STATEMENT
...
end;
```

図 11: データストアの実現

アのモジュールと、3つの変換プロセスのモジュールとして表現される。

図 12: イベントディスパッチプログラムジェネレータのデータフロー

図13にこれらの変換プロセスの実装を示す。変換プロセスを実装するモジュールでは、変換プロセスの入出力を定義するデータストアを宣言するモジュールの参照を行ない、変換プロセスを実装するオペレータが宣言される。例えば、GUIレイアウトからGUIオブジェクトの抽出を行う変換プロセスはモジュール *GUIObjSelect_mod* として実装される。モジュールの宣言では、この変換プロセスの入出力を表すデータストア open 文により参照されている。次にオペレータ *GUIObjSelect* が宣言され、このオペレータがGUI仕様の定義 *GUIGrmr* を、GUIオブジェクト *GUIObj* に変換する関数であることが宣言されている。

ジェネレータの開発者は変換プロセスに対応した各モジュールでオペレータの書き換え規則を定義していくことで、変換プロセスの実装を行う。

次に、変換プロセス間を接続するデータフローにより各変換プロセスを接続し、ジェネレータを実装する。データフローは書き換え規則の右辺に各変換プロセスのオペレータを記述することにより実装される。図14に図13の各変換プロセスを用いて構築したイベントディスパッチプログラムジェネレータを実装する

```
module GUIObjSelect_mod is
 open GUILayoutspec_mod;(* GUI 仕様の定義 *)
 open GUIObj_mod;(* GUI オブジェクトリストの定義 *)
 op GUIObjSelect : GUILayoutspec -> GUIObj;
end;
module EvtDispatchGen_mod is
 open CBFunc_mod;(* コールバック関数の定義 *)
 open EDPGrmr_mod;(* ディスパッチプログラム *)
 op EvtDispatchGen : CBFunc -> EDPGrmr;
end;
```

図 13: 変換プロセスのモジュール

変換プロセスの定義を示す。このモジュールではイベントディスパッチプログラムの生成を行うオペレータ *EVProgGen* が定義され、入力仕様は変換プロセスのオペレータ *GUIObjeSelect*,*CBFuncSelect*, *EVDispatchGen* によって順次変換され、最終的にイベントディスパッチプログラムを表すデータ構造 *EDPGrmr* に変換される。

```
module EVProgGen_mod is
 open GUILayoutspec_mod;(* GUI 仕様の定義 *)
 open GUIObj_mod;(* GUI オブジェクトリストの定義 *)
 open CBFunc_mod;(* コールバック関数の定義 *)
 open EDPGrmr_mod;(* ディスパッチプログラム *)
 op EVProgGen : GUILayoutspec -> EDPGrmr;
 rule EVProgGen(x) ==>
       EVDispatchGen(CBFuncSelect(GUIObjSelect(x)));
end;
```

図 14: 変換プロセスの結合

## 5.2 SOFTEXSHELL-2 による実装の効果

前節で述べたように、ジェネレータ設計プロセスで得られたデータフローダイアグラムはそのままSOFTEXSHELL-2のモジュールとして実装することができる。ここではSOFTEXSHELL-2による実装の効果を

- 変換プロセスの再利用
- 変換プロセスの並行開発
- 変換プロセスのプロトタイピング

という観点から考察する。

**変換プロセスの再利用**

変換プロセスはSOFTEXSHELL-2ではオペレータを含むモジュールとして実現される。変換の実行は次のように行われる。入力仕様の文法 $G$ から構成したコンストラクタによって表される入力仕様の抽象構文木をオペレータの引き数と

して渡し、このオペレータをモジュールの書き換え規則によりリダクションすることによって、変換結果を得る。このリダクションの結果はオペレータの引き数として渡される抽象構文木とオペレータを定義する書き換え規則にのみ依存し、変換実行時の環境に依存したり、実行による副作用などを持たない。この為、ある変換プロセスを再利用する場合、再利用したいモジュールのオペレータの書き換え規則を参照し、変換の実行の際に参照したモジュールで同時にリダクションを行うだけで、これを実現することができる。

**変換プロセスの並行開発**

変換プロセスのインタフェースは、それを実現するモジュールのオペレータの引き数と、そのリダクション結果のデータ構造のみによって規定される。また前述したように、変換プロセスは副作用を持たないという特長を持つ。そのため入出力のデータ構造が決定した時点で、独立にモジュールの書き換え規則の実装を行うことが可能である。また各変換プロセスの全ての入出力はコンストラクタシステムの項という非常に単純な形式で表現される為、変換プロセス間のインタフェースが取り易い。

**変換プロセスのプロトタイピング**

前節で述べたように、SOFTEXSHELL-2では書き換え規則の定義されていないオペレータはコンストラクタと見なされる。その為、ある変換プロセスで書き換え規則が定義されていない場合でも、そのオペレータの宣言があれば、変換を部分的に実行することができる。これを利用することにより、オペレータの書き換え規則が実装されていない場合でも、オペレータの宣言だけ参照し、部分的に変換を実行しながら変換プロセスをプロトタイピングすることができる。

## 6 現状と今後の課題

現在、コンストラクタシステム用いたSOFTEXSHELL-2のプロトタイプ版の開発が終了した。小規模なジェネレータ開発では、変換プロセスをモジュールとして実現することによる変換プロセスの再利用性の向上やプロトタイピングの効果が確認された。今後、大規模プロジェクトでの適用をはかり、ジェネレータの開発工数や、再利用されたモジュール数などから本方式を評価するとともに、仕様を表現する為のデータ構造や、集合やグラフといった汎用なデータ構造とその操作を行うモジュールを開発し、変換プロセスのライブラリ化を行っていきたい。

## 7 さいごに

ソフトウェア開発プロセス全体を支援する開発支援環境を構築する為には、各工程を支援する個々のジェネレータの構築を効率良く行うことが、ツールを含めた開発プロセス全体の生産性向上のポイントとなる。効率良くジェネレータを構築する方法として、ジェネレータの変換プロセスの分析から変換プロセスをデータフローダイアグラムで記述する手法について述べた。本稿では、このジェネレータの変換プロセスをコンストラクタシステムを用いて実装することを提案し、コンストラクタシステムに基づくソフトウェア自動合成シェルSOFTEXSHELL-2を開発した。これにより変換プロセスの再利用性の向上や、プロトタイピングを容易に行うことができ、ジェネレータの生産性を向上できることが判った。

## 謝辞

SOFTEXSHELL-2の言語仕様はNEC C&C研究所中島 震氏との共同検討の結果得られたものである。この場をかりて感謝致します。

## 参考文献

[1] Sato,A., Tomobe,M., Yamanouchi,T., Watanabe,M.,and Hijikata,M. : "Domain-Oriented Software Process Re-engineering with Software Synthesis Shell SOFTEX/ S",*10thKBSEConf.*, 1995.

[2] Yamanouchi,T.,Sato, A.,Tomobe M.,Takeuchi, H.,Takamura, J. and Watanabe, M. : "Software Syntehsis Shell SOFTEX/S",*7thKBSEConf.*, pp.28-37, 1992.

[3] Barstow, D.: "Domain-Specific Automatic Programming", *IEEE Trans. Software Engineering*, Vol.SE-11,No.11,1985.

[4] Lowry,M., Baalen,J.: "META-AMPHION :Synthesis of Efficient Domain-Specific Program Synthesis System",*10thKBSEConf.*,pp.2-10, 1995.

[5] Middeldorp A.,Toyama Y.:"Compileteness of Combinations of Constructor Systems", *J.SymbolicComputation*,pp.1-18,Vol.11,1993.

[6] Goguen,J.,Thatcher,J., Wagner,E., and Wright,J. : "Initial Algebra Semantics and Continuos Algebras" ,*J. of ACM*,Vol.24,No.1,pp.68-95,1977.