4C－3　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手法

工藤 智広　　佐藤 明良　　山之内 徹
NEC C&C研究所

1　はじめに

ソフトウエア自動合成シェルSOFTEXSHELL [1]により開発した専用プログラムジェネレータが現実の開発プロジェクトで効果をあげている [2][3]。こうしたジェネレータの効果的な適用を行うには、ジェネレータの適用箇所と機能をいかに決定するかが鍵となる。

しかし、プロジェクトの厳しいスケジュールや、先端的プロジェクトゆえの開発プロセスの未定式化などにより、あらかじめ適用箇所や機能に関する充分な分析を行えない状況がありうる。

この状況においては、ジェネレータ仕様を段階的に決定していき、各時点でテスト実行と動作確認を行うことにより、たとえジェネレータの実装方式や機能がすべて明らかになっていなくてもジェネレータの開発を進めることができる、というプロトタイピングによる開発スタイルが有効である。

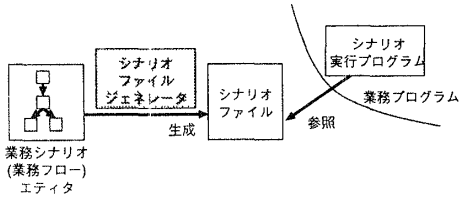

図 1: 適用事例

本稿では、SOFTEXSHELLを利用してジェネレータのプロトタイピングを行う手法について述べる。この手法は、実際開発事例(図1)であるシナリオファイルジェネレータ(GUIエディタで記述した業務フロー(シナリオ)データから、シナリオ実行プログラムが参照する設定ファイルを出力する)におけるプロトタイピングの事例から得られた内容をまとめたものである。

2　SOFTEXSHELLでのジェネレータ開発

SOFTEXSHELLによるジェネレータ開発では、図2に示すように、ジェネレータに対する入出力形式の型と、その間の変換処理を記述した関数の定義を行う(図3および図4に前述事例からの例を示す)。

Prototyping Method for Program Generators using Software Synthesis Shell SOFTEXSHELL
Tomohiro Kudo, Akiyoshi Sato and Toru Yamanouchi
NEC Corporation.

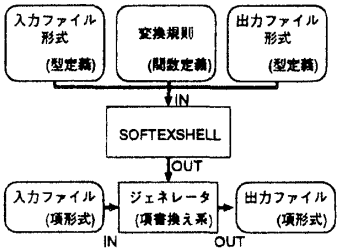

図 2: SOFTEXSHELLによるジェネレータ開発

SOFTEXSHELLのジェネレータは項書換え系(TRS)である。TRSでは、他の動作原理を持つ言語処理系と違い、関数が処理を行う定義域が全域でなくても実行が行え、エラーとなることがない(未定義の入力に対しては関数がリダクションされずにそのまま残るだけ)。この性質を利用すると、関数を部分的に定義して実行を行い動作を確認できるため、より小さい修正単位でのプロトタイピングが可能となる。

```
datatype NODE = node of int * string;
  (* ノードはIDとラベルからなる *)
datatype ARC = arc of int * int * int * string;
  (* アークはIDと遷移元/先のノードIDとラベルからなる *)
datatype SCENARIO = scenario of list * list;
  (* シナリオはノードのリストとアークのリストからなる *)
```

図 3: 型定義の例

```
(* ノードのラベルを取り出す関数の定義 *)
op GetNodeLabel : NODE -> string;
rule GetNodeLabel(node(id, name)) ==> name;
```

図 4: 関数定義の例

3　プロトタイピング手法

3.1　プロトタイピングのサイクル

ジェネレータのプロトタイピングは以下の各ステップを繰り返すことで行う(図5)。

型・関数定義追加・変更：型と関数定義を詳細化していく作業。ただし、開発の初期段階では、ジェネレータの外部インタフェース決定のため、まず、入出力形式の型と、これらの変換を表すメイン関数を定義する。それ以降は、定義の追加や変更を段階的に行ってジェネレータを完成させる。

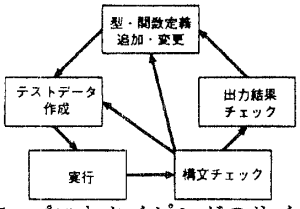

図 5: プロトタイピングのサイクル

なお、TRS の特徴により、関数は定義を行わなくてもエラーとならないため、関数の詳細化により新たな関数が出現しても、これをただちに定義せず、本質的に必要な時期まで定義する作業を遅らせることができる。

テストデータ作成：実行のために必要な入力データの例を作る。このテストデータはテストの対象としたい関数 F に対して、$F(\bar{A})$ の形式で与える。ここで $\bar{A}$ はパラメータの並びを表している。

実行：テストデータおよび現在までに記述した変換規則を合わせて SOFTEXSHELL により実行する。

構文チェック：SOFTEXSHELL のパーザが構文チェックを行い、括弧の対応ミスなどの単純な文法ミスや、テストデータや変換規則における型の不整合を調べ、エラーが検知されたならユーザに知らせる。これを見てテストデータか変換規則の修正ステップに戻る。

出力結果チェック：実行によって出力された結果をチェックし、想定する動作が行われているかどうかを調べる。動作が正しくない場合にはこれまでに修正・追加した箇所を中心に各定義を調べ、修正作業に入る。正しいことが確認された場合にはまだ詳細化されていない定義の詳細化を続ける。

3.2 定義の追加・修正作業

ここでは、図 5における各ステップのうち、定義の追加・修正について見る。このステップで行う作業は以下に述べる作業の単体または組み合わせからなる。

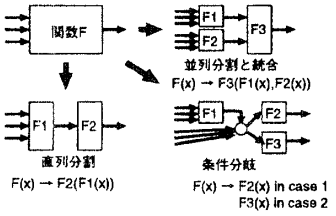

図 6: 関数の分割

入出力形式の型定義：入出力形式について初期定義を行い、これを詳細化する。詳細化は主に既存の型へのパラメータの追加と追加したパラメータの型定義によって行う。

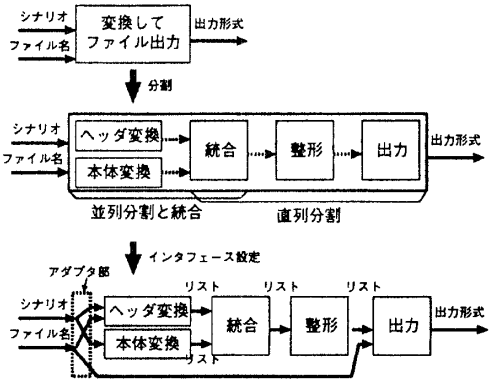

図 7: 関数の詳細化の例

関数の定義：既存の関数を詳細化していく。詳細化は、次のように行う。

関数の分割：詳細化対象関数の機能を分析して、それを実現するための関数分割方針を決める。分割のパターンとしては、主に、図 6に示すものがある。

関数インタフェースの設定：分割後の各関数の型を定義するとともに、インタフェースミス (パラメータの型や数の違い) がある場合、それを解決するためのアダプタ部を追加する。必要があれば新しい型を定義する。

この詳細化を実際に行った例を図 7に示す。これを繰り返すことによって詳細化を進めて行く。

4 おわりに

本稿では、SOFTEXSHELL を使ったジェネレータのプロトタイピング手法について、図 5のサイクルを示し、そのうちの定義の追加・修正作業の詳細化を行った。本手法の有効性を見るには、それ以外のステップでの手順を明確化するとともに、複数の開発プロジェクトへの適用が必要であり、これを行って手法の評価を行うことが今後の課題である。

参考文献

[1]Yamanouchi, Sato, Tomobe, Takeuchi, Takamura and Watanabe: "Software Syntehsis Shell SOFTEX/S", KBSE'92, 1992.

[2]Sato, Tomobe, Yamanouchi, Watanabe, and Hijikata : "Domain-Oriented Software Process Re-engineering with Software Syntehsis Shell SOFTEX/S", KBSE'95, 1995.

[3]Sato, Miki, Yamanouchi, and Watanabe : "Software Synthesis for Trade-off Design", KBSE'96, 1996.