問題領域向けトランスレータ構築支援ツールキット

登内敏夫[†] 中島震[†]

問題領域向け言語（DSL）およびその処理系を用いると問題領域に応じた抽象度の高い仕様記述を行えるため，ソフトウェア開発の生産性と保守性を向上させることができる．DSL を用いたソフトウェアの開発経験を通して，ソフトウェアの長期保守を実現するためには DSL や DSL 処理系の保守性を向上させることが重要であることが判明した．本論文では新たな DSL 処理系開発ツールキットを提案・実現し，これにより DSL 処理系の保守が向上したことを示す．

A Toolkit for Developing DSL Translators

Toshio Tonouchi[†] and Shin Nakajima[†]

A domain-specific language (DSL) and its processors improve the productivity and maintainability of a software system in the target application domain. It is because a DSL provides programmers with abstract notations for the specifications of the system. We find, through our experience of adapting a DSL and its processor to develop a software system requiring a long-term maintenance, that the maintainability of a DSL and a DSL processor is an important issue as well as that of a target software system. This paper describes a new DSL toolkit for developing DSL translators. We show that the toolkit improves the maintainability of developed translators.

1. はじめに

ソフトウェア開発生産性，再利用性，保守性の向上を目的とする問題領域向け言語（Domain-Specific Language; DSL）が注目を集めている．DSL は問題領域ごとに設計された言語であり，問題領域に適した構文を用いて開発対象ソフトウェアの仕様を記述する．仕様から実行コードやドキュメントを生成する DSL 処理系を用いることでソフトウェア開発生産性が向上する．プログラムコードを再利用するより，DSL で記述した抽象度の高い分析・設計レベルの記述を再利用する方が，再利用性が向上することが知られている[12),15)]．

一方，保守作業では，改修したソフトウェアの検査コストや修正コストよりも，保守対象ソフトウェアの要求仕様と構造を理解するための作業が保守コストの多くを占める[4)]．DSL で記述された抽象度の高い仕様記述を参照することで，保守担当者はソフトウェアの要求仕様を容易に理解することができる．また，仕様を入力としプログラムコードを生成する DSL 処理系が提供されている場合，保守担当者はプログラムの構造を理解しなくてもプログラムコードを改版することができる．

保守は発生原因により次の 3 種類に分類される[14)]．

修正保守 欠陥を改修する保守．

適応保守 プログラム実行環境の変更にともなう保守．

完全化保守 機能拡張など，要求仕様変更に対応するための保守．

また，本論文では以下を**波及保守**と定義する．

波及保守 開発対象ソフトウェアの保守にともない，その仕様を記述した DSL の言語体系もしくは開発対象プログラムを生成した DSL 処理系に対して発生する保守．

たとえば，開発対象ソフトウェアを他 OS に移植する場合，新たな OS 上で稼働するコードを生成するように DSL 処理系を変更する必要がある．

DSL を用いた開発における開発対象ソフトウェアの保守性を向上させるには，波及保守を含めた DSL 処理系の保守を欠かすことができない．そのため，DSL 処理系自身の保守を考慮した DSL 構築支援環境や方法論が必要である．

我々は，DSL 処理系構築ツールキット **Rosetta** を実現した．Rosetta は以下の特徴を有する問題領域に対して波及保守性を向上する．

† NEC ネットワーキング研究所
Networking Research Laboratories, NEC Corporation

(1) 長期保守が重要視されている問題領域.
(2) 仕様を曖昧さなく記述可能な問題領域.
(3) 仕様記述のための DSL の言語体系に対する変更が少ない問題領域.

DSL 処理系を導入するコストと，DSL 処理系を用いて開発対象ソフトウェアを開発するコストとの和は，開発対象ソフトウェアを直接開発するコストに比べて大きい．しかし，保守コストまで含めると前者が後者のコストを下回ることが予想される．つまり，保守まで考えたコストを考える必要のある条件 1 の問題領域が DSL 処理系の適用対象となる．

計算機は曖昧な仕様を解釈できないため，上記条件 2 は DSL 処理系を導入するための必要条件である．

条件 3 は，Rosetta が波及保守のうち DSL 言語体系に対して発生する保守ではなく，DSL 処理系に対して発生する保守を主な対象としていることを述べている．

本論文では Rosetta を紹介するとともに，Rosetta をもちいて GDMO [11] 処理系を開発した事例を通して，Rosetta が波及保守性の向上に効果があることを示す．

2. 関 連 研 究

DSL および DSL 処理系は開発対象ソフトウェアの保守を容易にするが，DSL に変更が及ぶような大きな変更要求に対しては保守性が劣化することが指摘されている[2]．そこで DSL 処理系開発の生産性および保守性を向上させるため，アプリケーションジェネレータ構築ツールなどの DSL 処理系を構築するツールが重要となる[1]．

Draco [15] は DSL で記述された分析，設計，コードの再利用を行うことで，ソフトウェアの生産性，保守性を向上させる．Draco はある問題領域の DSL を異なる問題領域の DSL に詳細化変換することで設計情報の再利用を行うことができる．詳細化変換を組み合わせることで仕様から実行可能なコードを生成することができる．すなわち，Draco は DSL 処理系を組み合わせることで DSL 処理系を構築するツールと見なすことができる．しかし，Draco では新たな DSL を構築・保守するための方法とシステム支援が提供されていない[5]．そのため，DSL 処理系に対する保守である波及保守を行うことは困難である．

文献 2) では項置き換えシステム（TRS）と文脈自由文法マッチングシステムを組み合わせた ASF+SDF を使って DSL 処理系を構築することで DSL の保守性を向上させた．ASF+SDF では DSL の文法を Syntax Definition Formalism（SDF）で，DSL の意味を Algebraic Specification Formalism（ASF）で記述でき，さらには，DSL 言語処理系および関連ツールを構築できる．DSL を構築・保守する場合，一部機能の製造，検査を行うことで生産性，保守性が向上する．DSL 開発者は処理系の一部機能を稼働・検査することで，設計や実現・保守方法の妥当さを確認できるからである．しかし，TRS では全書き換え（等式）規則を与えないと正しく動作しないため，部分的な製造，検査ができない．

Stage [1] では，テンプレートにより出力コードを定義する．入力に応じて出力コードが変化する部分（可変部分）と，変化しない部分（不変部分）に分け，不変部分は出力コードをテンプレートにそのまま記述し，可変部分に関しては，入力を表す具象構文木（CST）にアクセスする処理をテンプレートに埋め込む．ところが，Stage では再帰構造を記述する仕組みがないため適用範囲が狭い．一般的な文脈自由文法を入力とする処理系では再帰的な構文木構造を操作する必要がある．たとえば，数式を与えるとその演算を行うプログラムを出力するトランスレータの場合，数式を表す入力記述は再帰的な構造を有するため，再帰的な処理を記述する必要がある．

KHEPERA [3] は抽象構文木（AST）を異なる AST に（複数回）変換し，変換後の AST をプリティプリントするという考え方に基づきトランスレータを実現するトランスレータ構築ツールキットである．トランスレータ開発者は専用言語で変換規則を記述することができる．トランスレータの保守性を向上させるため，KHEPERA は入力記述と出力コードの対応関係を示すデバッグ機能を提供している．トランスレータ開発者が入力コードを指定すると，KHEPERA は入力コードに対応する出力コードを示す．このデバッグ機構はトランスレータ利用者にとって，入力記述のデバッグに役立つ．また，トランスレータ開発者にとっては，入力記述と出力コードの対応関係を知ることで，トランスレータを保守するうえで修正すべき変換規則を見つけやすくなる．しかし，AST の複数の変換規則により複数回 AST を変換するため，変更箇所が複数の変換規則に跨る可能性があり，各々の変換規則の修正の影響すべてを見通したうえで修正を行うことが困難である．

これまで発表された研究には問題点が残されており，これらを解決するのが Rosetta であり，次の要件を満たす．

- DSL 処理系構築のための手順と支援ツールが提

供されている．
- DSL 処理系の一部機能のみを実行・検査できる．
- 入れ子構造を含むコードのような再帰的に定義される出力コードを生成できる．
- DSL 処理系プログラムを変更すべき箇所が容易に発見できる．
- DSL 処理系プログラムに手を加えることによる影響範囲を容易に推測できる．

3. DSL 構築ツールキット Rosetta

DSL ランスレータ構築ツールキット Rosetta は DSL ランスレータの開発を支援する．開発対象トランスレータを Java で記述するため，トランスレータ開発者は Java の知識を有することを仮定する．

3.1 アーキテクチャ

図 1 に Rosetta のアーキテクチャを示す．図中の大きな矩形は Rosetta を使って開発したトランスレータを示す．網掛けされた矩形は Rosetta が提供するライブラリモジュールもしくはオフラインツールである．下線を引いたモジュール（矩形），もしくはファイル（菱形）はトランスレータ開発者（もしくはトランスレータ設計者，DSL 設計者）が作成する．

Rosetta を用いて開発したトランスレータは次のように動作する．

(1) パーザは DSL で記述した入力ファイルを構文解析し，AST を生成する．AST は抽象構文木データベースに格納される．

(2) 構文解析後，意味検査器が抽象構文木データベース内の AST にアクセスし，入力が意味的に正しいかを検査する．意味的なエラーを発見したら，意味検査器は適切なメッセージを出力し，処理を継続するか停止するかを判断する．

(3) 意味検査処理が完了した後，コードジェネレータは抽象構文木データベース内の AST を出力コードに変換する．**出力テンプレート** はコードジェネレータが行う変換の仕方を記述したスクリプトである．

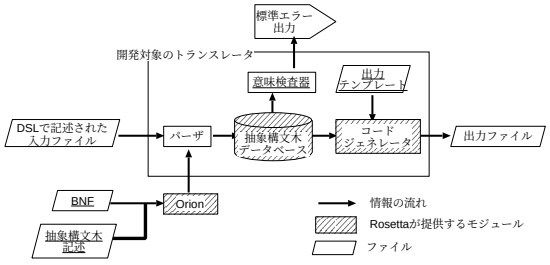

図 1 Rosetta のアーキテクチャ
Fig. 1 The architecture of Rosetta.

3.2 Orion

Orion は Rosetta が提供するオフラインツールであり，DSL 処理系で使用するパーザを生成する．具体的には，Orion はパーザジェネレータ JavaCUP [7] の入力ソースコードを出力し，JavaCUP がパーザの Java プログラムコードを生成する．

Orion は DSL の文法を規定する BNF ファイルと，AST の仕様を規定した**抽象構文木記述**とを入力とする．

たとえば，数式を表す以下の BNF が与えられたとする．

```
exp    ::= exp PLUS term
       |   term ;
term   ::= term MULTI factor
       |   factor  ;
factor ::= LPER exp RPER ;
       |   NUM ;
```

`exp`，`term`，`factor` は非終端記号を表す．`PLUS` は “+”，`MULTI` は “*”，`LPER` は “(”，`RPER` は “)”，`NUM` は整数値トークンを表す終端記号である．

以下の**抽象構文規則**群を，上記 BNF に対する抽象構文木記述の例としてあげる．

```
exp ::= (left) exp (op) PLUS (right) term
    |   (exp)  term.factor.exp
    |   (num)  term.factor.NUM
    |   (term) term             ;
term ::= (left) term (op) MULTI
         (right) factor
    |    (exp) factor.exp
    |    (num) factor.NUM    ;
factor ::=  (exp) exp
       |    (num) NUM  ;
```

終端記号と非終端記号は AST のノードを表す．左辺（::=の左部分）が示すノードは右辺が示すノードの上位に位置する．右辺の終端記号と非終端記号の左側には “(*name*)” により**枝名** *name* が指定される．枝名は上位ノードから下位ノードに伸びる**枝**に対して与えられた名前である．右辺に終端記号および非終端記号が “.” で連結したノード (ex. `factor.exp`) がある．これは，左辺 `term` で示す AST ノードの枝名 `exp` で指定される下位ノードは，BNF 規則 `term` から，`factor`，`exp` を順次手繰った先のノードを示す．

このような BNF ファイルと抽象構文木記述を入力とし Orion が生成したパーザに，入力 “3 * (2 + 1)” を与えると**図 2** に示す AST を生成する．

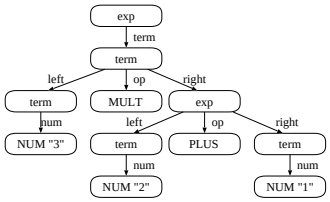

図 2 抽象構文木の例
Fig. 2 An example of an abstract syntax tree.

```
/* 数値演算結果を出力する． */
#include <$lt>stdio.h<$gt><p>

int calc() {<p>
 int ans = 0;<p>
 ans =
<INCLUDE "exp.gen">
   <POINT name=EXP, address=&(/ROOT:*:)>
</INCLUDE>;<p>
 return ans;<p>
}<p>

void main(int argc, char* argv[]) {<p>
 printf("Answer = %d\n", calc());<p>
}<p>
```

図 3 calculator.gen
Fig. 3 calculator.gen.

3.3 出力テンプレート

コードジェネレータは図 2 で示すような AST を出力コードに変換する．変換規則を出力テンプレートに記述する．

出力テンプレートの例として “calculator.gen” を**図 3** を示す．“calculator.gen” の `INCLUDE` タグから呼ばれるファイル “exp.gen” を**図 4** に示す．ファイル “term.gen” については記載を省略する．“calculator.gen” を変換規則として与えた場合，コードジェネレータは**図 5** で示すコードを生成する．

出力テンプレートは次の 2 種類の要素から構成されている．

テキスト出力部 コードジェネレータが出力テンプレートの内容をそのまま出力する部分．たとえば，“`int calc() {`” がテキスト出力部である．

タグ “<”，“>” で囲まれた部分をタグと呼ぶ．コードジェネレータがタグを解釈し，適切な動作を行う．たとえば，INCLUDE タグは指定したファイルを出力テンプレートとして「サブルーチンコール」をする．

```
<EXISTS $(&EXP/op:*:)><THEN>
  <INCLUDE "exp.gen">
   <POINT name=EXP, address $(&EXP/left:*:)>
  </INCLUDE><s>+<s>
  <INCLUE "exp.gen">
    <POINT name=EXP, address $(&EXP/right:*:)>
  </INCLUDE>
</THEN></EXISTS>
<EXISTS $(&EXP/exp:*:)><THEN>
 (<INCLUE "exp.gen">
    <POINT name=EXP, address $(&EXP/exp:*:)>
  </INCLUDE>)
</THEN></EXISTS>
<EXISTS $(&EXP/term:*:)><THEN>
  <INCLUE "term.gen">
   <POINT name=TERM, address $(&EXP/term:*:)>
  </INCLUDE>
</THEN></EXISTS>
<EXISTS $(&EXP/num:*:)><THEN>
   <$(&EXP/num:*:)>
</THEN></EXISTS>
```

図 4 exp.gen
Fig. 4 exp.gen.

```
/* 数値演算結果を出力する． */<p>
#include <stdio.h>
int calc() {
 int ans = 0;
 ans = 3 + ( 2 + 1 );
 return ans;
}
void main(int argc, char* argv[]) {
 printf("Answer = %d\n", calc());
}
```

図 5 生成プログラム
Fig. 5 A generated program file.

このように出力テンプレートは HTML に似た構文を有することで，以下の利点を有する．

- 多くのユーザに馴染みがあり，記述方法を理解しやすい．
- テキスト出力部と DSL 処理系の出力イメージとの対応がとりやすい．

出力テンプレートは以下の構文を有する．

- データ型として抽象構文木ノードポインタ型（以下ポインタ型），文字列型，スタック型がある．ま

た，これらのデータ型を格納する変数が提供されている．ポインタ型はASTノードを指し示す．たとえば，図4の EXP はポインタ型の変数である．スタック型は文字列型もしくはポインタ型を要素とする配列である．

- ASTノードを指し示すため，ノード検索式を使用する．たとえば，“$(&EXP/exp/term:*:)” はポインタ型 EXP が指し示すノードの枝 exp の下のノードのさらに枝 term の下のノードである．
- INCLUDEタグによりサブルーチンコールができる．上記例において “exp.gen” から “exp.gen” を INCLUDE していることから分かるように再帰呼び出しも可能である．INCLUDEタグでは，POINT タグなどでポインタ型などの実引数を値渡しする．
- EXISTSタグなどによる条件分岐が可能である．たとえば，“<EXISTS $(&EXP/op:*:)>” では，ノード EXP の op の枝先にノードが存在する場合，“<THEN> .. </THEN>” の間を実行する．
- FORタグによりループ機構が提供されている．FORタグを使うと，与えられた検索式に合致するASTノードが存在する限り処理を繰り返す．FORタグで使用する検索式（パターン検索式）の例を示す．“(&EXP, /num:*:)” はノード EXP を頂点とするサブツリーの中から枝 num の指すASTノードをすべて捜し出す．

3.4 トランスレータ開発方法

Rosettaでは，**図6**で示す手順でDSLトランスレータを開発することを想定している．図6の数字は以下に述べる手順の番号と対応している．

(1) DSL設計者は問題領域に対する経験をもとに，DSLの文法と意味を定める．文法はBNFで記述し，意味は自然言語で記述する．
4章で述べるGDMO[11]のように標準化されたDSLを利用する場合は，本ステップを省略できる．

(2) トランスレータ設計者は，トランスレータの出力仕様を定める．図5で示した出力の仕様を**図7**に示す．図中の [..] は条件分岐を表す．右肩上の数字が指す注釈で示す条件が成立する場合，[..] 内のコードが生成される．図中の < .. > はループを表す．右肩上の数字が指す注釈でループ継続条件を記述する．出力仕様はトランスレータ開発者が参照する．

(3) トランスレータ設計者はASTの仕様を抽象構文木記述で記述する．

(4) トランスレータ設計者は意味検査項目を定める．意味的に不正な場合を列挙し，その発生条件を明確にし，それぞれにエラーメッセージを定める．

(5) トランスレータ設計者はトランスレータのコマンド形式を定める．

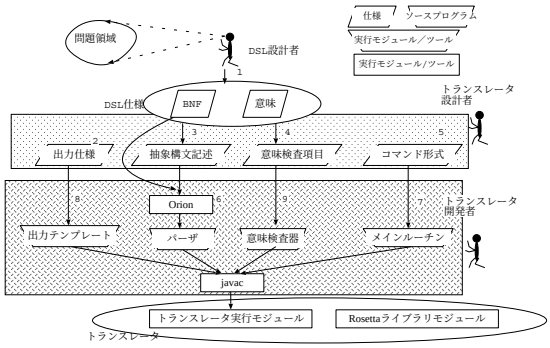

図6 Rosettaを用いたDSL開発方法
Fig. 6 Rosetta development method for domain-specific language processors.

```
/* 数値演算結果を出力する．*/
int calc() {
   int ans = 0;
   ans =
[   < exp>*5 + < exp>*5        ]*1
[
  [< term>*6 * [(< exp >*5)]*4  [ num*6]*5]*3
  [ (< exp>*5   ]*7
  [  num*6 ]*8
]*2
[    num*6     ]*3
[    exp*5                ]*4;
    return ans;
}
void main(int argc, char* argv[]) {
   printf("Answer = %d
n", calc());
}
```

1．exp の下に op の枝がある場合．
2．exp の下に term の枝がある場合．
3．exp の下に num の枝がある場合．
4．exp の下に exp の枝がある場合．
5．exp は再帰的に呼びだし．
:

図7 出力仕様
Fig. 7 The specification of a generated file.

(6) トランスレータ開発者は Orion に BNF と抽象構文木記述を入力として与えて，パーザプログラムを生成する．
(7) コマンド形式の仕様に従って，トランスレータ開発者はメインルーチンを製造する．
(8) 出力仕様と抽象構文木記述を参照し，トランスレータ開発者は出力テンプレートを記述する．
(9) トランスレータ開発者は意味的に不正な場合を検出する意味検査器を製造する．

本手順によりトランスレータ開発において DSL 設計者，トンラスレータ設計者，トランスレータ開発者の役割と，作成する仕様を明確化している．対象問題領域に深く精通し，プログラミング技術をも有する有能な技術者が長期間，保守に携わることができるならばこのような繁雑な手順を必要とはしない．しかし，役割分担を行うことで各業務に有能な技術者を活用することができ，仕様を残すことで保守を容易にする．

3.5 Rosetta における保守性向上

Rosetta は以下の点で保守性を向上させている．

- Orion は抽象構文木記述に基づきパーザを生成するため，抽象構文木記述はパーザの振舞いと必ず一致していることが保証されている．すなわち，抽象構文木記述は信頼のおける設計文書である．保守作業において設計文書が存在しなかったり，設計文書がプログラムの挙動と一致しているか不安があるために，プログラムの構造理解のための作業が発生し保守効率が低下することを避けることができる．
- 出力テンプレートはトランスレータが生成する出力に近い形式で記述される．たとえば，生成コードの 1 行目と “calculator.gen” の 1 行目が対応していることが容易に分かる．出力コードと出力テンプレートの対応が明確なため．保守の際，出力テンプレートの変更すべき箇所を容易に見つけることができる．また，出力テンプレートを修正した際，出力に対する影響範囲を見通しやすい．
- 出力テンプレートはファイル単位でモジュールが分割されている．サブモジュールを実現するファイルを呼び出すことで，一部の機能のみを実行・検査することができる．トランスレータ開発者は，部分的に機能を実行することで実現・保守方法の妥当性を効率的に検討することができる．
- 出力テンプレートは，再帰呼び出し，条件分岐を有する強力な抽象構文木操作言語である．文脈自由文法は BNF で記述されるが，BNF では条件分岐 (|) と，再帰的構文（BNF 右辺で左辺非終端記号を参照）が記述可能である．出力テンプレートは文脈自由文法から生成した抽象構文木をたどることが可能な構文を有している．
  また，Java プログラムの呼び出し機構を有している．これにより四則演算などタグとして提供しない機能を実現できる．

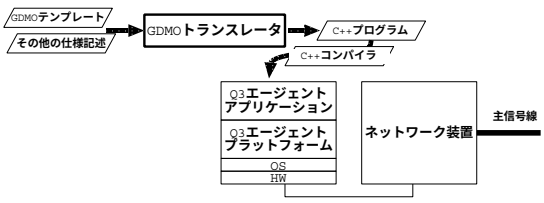

図 8 GDMO トランスレータと Q3 エージェントプラットフォーム
Fig. 8 A GDMO translator and Q3 agent platform.

4. Rosetta による GDMO トランスレータの開発

OSI ネットワーク管理 Q3 エージェント開発のために，Q3 エージェントプラットフォーム[16]と GDMO トランスレータを開発した．GDMO[11]は OSI 管理モデル記述用 DSL であり，GDMO トランスレータは Rosetta で開発した DSL 処理系である．GDMO トランスレータはネットワーク管理モデルの仕様を記述した GDMO テンプレートおよび独自に定めた仕様記述を入力とし，Q3 エージェントプラットフォーム上で稼働する C++プログラムを生成する（**図 8**）．

Q3 エージェントは公衆網の管理を実現するシステムであるため，数十年にわたる保守を要求される．GDMO[11]は ITU-T 勧告で定義されており，GDMO の言語体系に対する変更は少ない．そのため，GDMO トランスレータは Rosetta には適した適用例といえる．

ある海底ケーブル管理システムとして開発した Q3 エージェントプログラムは仕様レベルで約 100 個のクラス定義と，数万個のオブジェクトインスタンスの初期化コードを含むので，プログラム規模は膨大になる．たとえば，GDMO トランスレータの適用例として 17 kL の入力記述から 221 kL の C++プログラムコードを生成した．GDMO トランスレータはクラスに関する仕様（GDMO ほか）とインスタンスに関する独自の仕様記述から，膨大な量のプログラムコードを自動生成するため，ソフトウェア生産性が向上する．

GDMO トランスレータを使うことで，入力記述を変更し再度トランスレータを起動することで保守をすることができる．たとえば，オブジェクトインスタンスは包含木[10]と呼ぶ木構造モデルを構成しているが，

仕様記述を変更することでオブジェクトインスタンスの位置を容易に変更することができる．

このようにRosettaによる実用トランスレータの開発を行い，本トランスレータがQ3エージェント開発に有用であることを確認した．

次に，波及保守の必要性を示すために，GDMOトランスレータで実際に発生した保守作業を分析する．GDMOトランスレータはVer 1.0から23回のバージョンアップを繰り返し，80回の保守作業が発生した．その保守作業を下記に分類した．

修正保守 60件
適応保守 2件
完全化保守 12件
波及保守 6件

本事例では波及保守は1割発生した．波及保守には開発対象に対する保守だけではなくDSL処理系に対する保守も必要なため，他の保守に比べて工数を要する[9]．本事例からも分かるように，開発対象の保守性を高めるためには波及保守性を高めることが重要である．

5. 実験による評価と考察

同一の仕様を有するトランスレータをRosettaとVisitorパターン[6]でそれぞれ開発し，Rosettaの保守性を定量的に評価した．ASTノードの種類ごとに対応するVisitorメソッドが起動されるため，Visitorパターンは言語処理系の開発に適しているデザインパターンであることが知られている．開発対象であるHTMLトランスレータは，GDMOテンプレートとASN.1ファイルを入力としそれらをドキュメント化したHTMLファイルを生成する．

それぞれのトランスレータをJavaプログラミングの経験を有する2人のトランスレータ開発者が開発した．2人ともRosettaを用いた経験はない．今回の実験では出力部分の開発のみを行い，4章で述べたGDMOトランスレータで開発したパーザや意味検査器を流用した．それぞれのトランスレータ開発者は出力仕様を定め，抽象構文木定義を参照して出力テンプレートやVisitorプログラムを開発する．

開発では両者に同じ要求仕様（ver.1）を与え，開発に要した工数を計測した．Ver.1完成後，新たに機能拡張を要請しver.2を開発し，工数を計測した．すなわち，本計測工数は機能拡張にともなう完全化保守作業の工数である．Ver.1とver.2で要した工数を**表1**にまとめた．「規模」は，プログラマが記述したソースの行数（kL）である．「修正」は，ver.1とver.2のソースをUNIX diffコマンドで比較した結果を出力した行数である．

表1 完全化保守に要した工数の比較（人時）
Table 1 The cost of the perfective maintenance (person-hours).

	Ver.1		Ver.2		
	工数（人時）	規模（kL）	工数（人時）	修正（kL）	修正(L)/修正箇所
Rosetta	200	9.0	53	4.0	11
Visitor	192	8.0	87	2.3	6.3

表2 完全化保守作業の内訳（修正箇所数）
Table 2 Details of perfective maintenance (the number of bugs).

	追加	削除	変更	合計
Rosetta	211	17	123	351
Visitor	88	88	192	368

表1のVer.2における工数を比べると，Rosettaの方がVisitorの60%の工数で完全化保守を行えることが分かる．しかし，Rosettaの方がVisitorより修正行数が多い．この内訳を**表2**に示す．修正行数はRosettaが多かったが，修正箇所はわずかに少ない．つまり，Rosettaの修正箇所はVisitorに比べ，局所的に集中して存在する傾向が強い（表1の「修正(L)/修正行数欄）．修正箇所が点在していると，保守が困難になる．複数箇所に点在していると修正箇所を見つけることが困難となるためと，修正箇所間の関連を気にする必要があるためである．

Rosettaの修正内容はVisitorに比べ，「変更」と「削除」よりも「追加」が多い．たとえば，新機能を実現するため出力コードに関数定義コードを追加する場合，Rosettaならば関数定義の出力イメージを出力テンプレートの適切な場所に「追加」すればよい．追加コードは出力テンプレート中の他のコードに影響を及ぼさない．一方，Visitorならば既存のVisitorメソッドの中に関数定義コードを出力するコードを埋め込む．このとき，既存Visitorメソッドに手を加えるため，既存メソッドのコードを「変更」，「削除」することが多い．既存コードの内容を注意深く吟味しなければ，「変更」と「削除」により既存コードが動作しない可能性がある．

次に，波及保守について考察する．本実験での開発対象はDSL処理系である．波及保守とはDSL処理系に発生する保守のため，本実験における完全化保守は，波及保守と見なせる．表1より，波及保守に要する工数は，完全化保守のための工数と同様，Rosettaの方が少ないと推測できる．

表 3 障害 1 件改修に要した工数（人時）
Table 3 The cost of repairing a bug (person-hours).

	Ver.1	Ver.2	平均
Rosetta	2.39	2.11	2.45
Visitor	3.04	2.68	2.77

修正保守に要する工数を計測するため，HTML トランスレータ開発での障害票 1 件を改修するのに要した工数の平均を**表 3** に示す．Rosetta の修正保守工数の方が少ないことが分かる．

6. Rosetta で開発した DSL 処理系の課題

Rosetta で開発した DSL 処理系の処理時間がかかることが分かっている．GDMO トランスレータでは約 2 万行の入力に対して，100 MB のヒープ領域を必要とした．Sparc Station 20（OS は Solaris）上で，C++プログラムコードの生成が完了するまで 5，6 時間要した．しかし，全ファイルを出力することは頻繁にはなく，修正した部分に関するファイルのみを出力すればよいので，4 章で紹介した開発プロジェクトでは処理時間が大きな問題となることはなかった．

Rosetta では，文法的に関連する AST ノードを枝で接続した AST を生成する．しかし，コード生成では意味的に関連する AST ノードをパターン検索他で検索するする必要がある．パターン検索に処理時間がかかるため，トランスレータの処理効率を悪化させている．パターン検索の効率を向上させるには，意味的に関連のある AST ノードにリンクを張り，意味グラフを作る方式が考えられる．AST の代わりに意味グラフを使えばトランスレータの処理効率は向上する．しかし，意味グラフを形式に記述する一般的な方法が存在しないので，意味グラフ方式を Rosetta では採用しなかった．意味グラフ仕様記述と，それを入力とし意味グラフを出力とするパーザ生成ツールを開発できれば，抽象構文木記述と同様意味グラフ仕様記述が信用のおける設計文書となり，開発対象のトランスレータの保守性を高めると期待できる．

Rosetta で開発したトランスレータが膨大なヒープ領域を必要とし，処理速度が遅いのは入力記述すべてを表す AST を生成するためである．膨大な入力記述を表す巨大な AST を構築し，巨大な AST を検索するため，処理速度が低下する．AST をすべて保持する代わりに，入力から順次 AST を構築し不要になった AST を廃棄する方式では処理効率は向上する．しかし，出力テンプレートでは AST のいかなる部分も参照可能なため，AST を廃棄することができない．

AST 検索処理の高速化は今後の課題である．

7. ま と め

GDMO トランスレータの開発経験を通し，開発対象ソフトウェアの保守を行うには，DSL およびその処理系の保守が不可欠であることを示した．DSL 処理系の保守性を向上することを目的とし，DSL トランスレータ構築ツールキット Rosetta を構築した．Rosetta では Orion などの支援ツールが提供されており，これらのツールを利用した開発手順が想定されている．抽象構文記述木のように，開発手順に従い作成した入力ファイルが設計文書となる．また，出力テンプレートは手続き呼び出しができ，手続き単体でも検査することができる．これにより検査効率が向上する．手続き呼び出しは自身を呼び出すことも可能なので，入れ子構造のような再帰的に定義されるコードも生成することができる．出力テンプレートは HTML と同様，タグとテキスト出力から構成される．そのため，出力コードと出力テンプレートの対応が容易に分かり，DSL 処理系保守の際，出力テンプレートの変更する部分が容易に発見でき，また，出力テンプレート修正による出力コードへの影響範囲を容易に推測することができる．

Rosetta 自体は設計文書などのバージョン管理機能を有しないが，cvs などの他ツールと組み合わせることで DSL 処理系の保守性を向上させることができる．

同一仕様のトランスレータを Rosetta と汎用言語を用いて Visitor パターンでそれぞれ開発し，その工数を比較することで，Rosetta の保守性がよいことを定量的に確認した．Rosetta は特定の問題領域の DSL 処理系に特化したものではなく，DSL 言語体系が安定した領域ならば有用性がある．さまざまな領域の多数の事例に適用することで各開発事例に対する Rosetta を開発する工数の影響は小さくなる．

謝辞 Rosetta 研究開発の動機となった GDMO トランスレータの開発および実適用に協力して下さった NEC 海洋光ネットワーク事業部の洲脇マネージャをはじめとする皆様に感謝します．

参 考 文 献

1) Cleaveland, J.C.: Building Application Generators, *IEEE Software*, pp.25–33 (July 1988).
2) van Deursen, A. and Klint, P.: Little Languages: Little Maintenance?, *Proc. First ACM SIGPLAN Workshop on Domain-Specific Languages*, *DSL '97*, pp.109–127 (1997).
3) Faith, R.E., Nyland, L.S. and Prins, J.F.: KHEPERA: A System for Rapid Implementation of Domain Specific Languages, *Proc. Con-*

ference on Domain-Specific Languages, pp.243–255 (Oct. 1997).
4) Fjeldstad, R.K. and Hamlen, W.T.: Application Program Maintenance Study-Report to Our Respondents, IBM Corporation, DB Marketing Group (1979).
5) Freeman, P.: A Conceptual Analysis of the Draco Approach to Constructing Software Systems, *IEEE Trans. Softw. Eng.*, Vol.13, No.7, pp.830–844 (Jul. 1987).
6) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Pattern: Element Reusable Object-oriented Software*, Addison-Wesley (1994).
7) Hudson, S.: CUP Parser Generator for Java, http://www.cs.princeton.edu/~appel/modern/java/CUP
8) ITU-T Recommendation X.720, Management Information Model (1993).
9) 登内敏夫，中島 震：言語処理系ツールキット Rosetta による GDMO トランスレータの実現，TM99-10 (1999).
10) ITU-T Recommendation X.720, Management Information Model (1993).
11) ITU-T Recommendation X.722, Guidelines for the Definition of Managed Object (1992).
12) Kieburtz, R.B., et al.: A Software Engineering Experiment in Software Component Generation, *Proc. ICSE*, pp.542–552 (1996).
13) Megginson Technologies, SAX 2.0: The Simple API for XML, http://www.megginson.com/SAX/index.html (Jun. 2000).
14) McClure, C.: ソフトウェア開発と保守の戦略，共立出版 (1993).
15) Neighbors, J.M.: The Draco Approach to Constructing Software from Reusable Components, *IEEE Trans. Softw. Eng.*, Vol.10, No.5, pp.564–574 (Sep. 1984).
16) 登内敏夫，中島 震：装置組み込み用高速 Q3 エージェントプラットフォームの実現，情報処理学会論文誌，第 41 巻，第 4 号，pp.1226–1233 (Apr. 2000).
17) World Wide Web Consortium: Document Object Model (DOM) Level 1 Specification Version 1.0, http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/ (Oct. 1998).

付　　録

A.1 出力テンプレートのタグ一覧

タ グ 一 覧

タグ	end tag	説明
FILE *v*	無	出力ファイルの指定
DATABASE *v*	無	抽象構文木データベースの指定
INCLUDE *v*	有	出力テンプレートの呼び出し．(i.e. 副手続き呼び出し)
GENELET *v*	有	Java プログラムの呼び出し．
SWITCH *v*	有	多値条件分岐．*v* と同じ値を有する CASE タグを実行する．
MATCHV *v* IN [*stack*]	有	2 値条件分岐．スタック内に *v* が存在するか，否かで条件分岐をする．
EXISTS *pattern*	有	条件分岐．抽象構文木中に *pattern* が存在するか，否かで条件分岐をする．
FOR *pattern*	有	ループ．パターン *pattern* で指定したノードの数だけ繰り返し処理を行う．
BREAK	無	ループからの脱出．必ず FOR タグ内で使用する．
ASSERT *v*, *str*	無	*v* と *str* が一致した場合，ループから脱出する．
BREAKCATCH	有	BREAK タグもしくは ASSERT タグによってループを抜け出したときに実行する処理．
P [*n*]	無	改行を（*n* 個）出力．
S [*n*]	無	空白を（*n* 個）出力．
T [*n*]	無	タブを（*n* 個）出力．
CFGFILE	無	現在実行中の出力テンプレート名を出力．デバッグ用．
PRE	有	記述内容をそのまま出力．
INSERT *v*	無	ファイル *v* の内容をそのまま出力．
ERROR	無	強制的に処理を停止する．デバッグ用．
RETURN	無	本出力テンプレートを呼び出した INCLUDE タグの直後に処理を戻す．(副手続きからの復帰)

SET *var* = *v*	無	変数 *var* に値 *v* を設定する.
PUSH *stack*, *v*	無	*v* をスタックに格納する.
CREATE *stack-name*	無	新しいスタックを作成する.
CLEAR *stack-name*	無	スタックを消去する.
$*var*	無	変数の値.
$[*stackname*, *n*]	無	スタックの *n* 番目の要素.

パラメータタグ一覧

タグ	end tag	説明
INCLUDE タグのパラメータ		
VAL NAME=*var*, VALUE = *v*	無	仮パラメータ *var* に実パラメータ *v* を代入する.
POINT NAME = *var*, ADDRESS = *addr*	無	仮パラメータ *var* にノード検索式 *addr* で指定される抽象構文木ノードを代入する.
REF NAME=*var*, REFERENCE = *stack*	無	仮パラメータ *var* にスタックを代入する.
GENELET タグのパラメータ		
PARAM NAME = *var*, VALUE = *v*	無	仮パラメータ *var* に実パラメータ *v* を代入する.
RESULT NAME = *stack*	無	呼び出した Java プログラムの処理結果を格納するスタックを指定する.
SWTICH タグのパラメータ		
CASE *v*	有	SWITCH タグの引数の値と *v* の値が一致したとき処理を実行する.
DEFAULT	有	SWITCH タグの引数の値とすべての CASE タグの値が一致しないとき, 実行する.
MATCHV/EXISTS タグのパラメータ		
THEN	有	MATCHV タグもしくは EXISTS タグの条件が成立するとき実行する.
ELSE	有	MATCHV タグもしくは EXISTS タグの条件が成立しないとき実行する.

(平成 12 年 9 月 23 日受付)
(平成 13 年 11 月 14 日採録)

登内 敏夫（正会員）

平成 4 年東京大学大学院理学系研究科情報科学専攻修士課程修了. 同年 NEC 入社. 平成 12 年〜13 年英国 Imperial College of Science, Technology, and Medicine 訪問研究員. 現在 NEC ネットワーキング研究所勤務. ネットワーク管理を対象領域としたソフトウェア基盤やソフトウェア開発用言語処理系の研究開発に従事.

中島 震（正会員）

昭和 54 年東京大学理学部物理学科卒業. 昭和 56 年同大学大学院理学系研究科修士課程修了. 同年 NEC 入社. 現在, 同社ネットワーキング研究所勤務. 分散ソフトウェア工学の研究に従事. 昭和 63 年〜平成元年米国オレゴン大学訪問研究員, 平成 4 年〜12 年東京都立大学工学部非常勤講師, 平成 13 年より法政大学情報科学部非常勤講師, 平成 13 年より科学技術振興事業団さきがけ 21「機能と構成」領域研究員を兼務. 学術博士（東京大学）. 平成 13 年度山下記念研究賞授賞. 本学会ソフトウェア工学研究会幹事, 日本ソフトウェア科学会理事・編集委員.