ソフトウェア工学　121－21
（1998. 11.　6）

デザインパターンに基づくドメイン依存言語向けツールキット

別府 祥之　　登内 敏夫　　中島 震

NEC C&Cメディア研究所
〒216-8555 川崎市宮前区宮崎4丁目1-1
TEL: 044-856-2259
E-mail: beppu@ccm.cl.nec.co.jp

あらまし　ドメイン依存言語(DSL)を用いたアプリケーションジェネレータを中間データ構造に注目したツールキットを用いて作成した経験について報告する。DSLは特定問題領域のプログラムに共通する特徴を表現可能にしアプリケーションの開発生産性を向上させるが、DSL自身の開発工数が大きいという問題ある。そこで、抽象構文木などの中間データ構造の簡潔な記述方法を定義するとともに、この記述から中間データ構造を表すJavaのプログラムと中間データ構造をたどりながら型チェックやコード生成を行なうためのスケルトンを自動生成するツールを開発した。このツールをFScriptというネットワーク管理を行なうDSLの開発に適用したところ作成すべきプログラムサイズを40%削減することができた。

キーワード　ドメイン依存言語、デザインパターン、再利用、中間データ構造、ネットワーク管理

A Design Pattern based Toolkit for Domain Specific Language Implementation

Yasuyuki BEPPU　　Toshio TONOUCHI　　Shin NAKAJIMA

C&C Media Research Laboratories
NEC Corporation
4-1-1,Miyazaki Miyamae-ku, Kawasaki,
Kanagawa, 〒216-8555, Japan
TEL: 81-44-856-2259
E-mail: beppu@ccm.cl.nec.co.jp

Abstract　The paper presents our experience in developing an appliation generator that is based on a Domain Specific Language (DSL). Although DSLs increase productivity and quality of programs in its application domain, constructing a new DSL and its related tools are costly. Our approach is to develop a toolkit that generates Java programs of both tree-like internal data structures and skelton codes for tree walking. The paper also reports that using the toolkit results in decreasing 40% of program codes constituting the DSL tool.

key words Domain Specific Language, Design Pattern, Reuse, Abstract Syntax Tree, Network Managenment System

1 はじめに

アプリケーションプログラムの開発効率化ならびに品質向上へのアプローチとして、特定問題領域に専用のジェネレータを用いる方法がある [3]。特に、特定問題領域のプログラムに共通する特徴を表現可能な一種の設計言語であるドメイン依存言語 (Domain Specific Language:DSL) が注目を集めている [4] [7]。DSL を用いるアプローチでは、ジェネレータは DSL による入力仕様を C++ 等の実行可能なプログラムに変換するトランスレータとして実現する [5]。

我々は、OSI ネットワーク管理 [10] を応用領域として選び、ドメイン依存言語 FScript と FScript による仕様記述から C++ プログラムを自動生成するトランスレータ FOG を開発している [8] [9] [12] [2]。一般に、FOG のようなトランスレータは抽象構文木、パーザ、構文木変換、型チェッカ、コードジェネレータなどから構成されるが、これらの開発工数が大きいことが問題である。

本稿では、FOG の開発効率を向上させる目的で作成したドメイン依存言語ツール開発のためのツールキットについて、方法・効果を報告する。型チェッカやコードジェネレータを作成する場合、ビジターデザインパターンの考え方で関連操作を一つのクラスにまとめることでプログラムの良構造化ならびに再利用性を高めることができる [6]。そこで、抽象構文木など中間データ構造の簡潔な記述から中間データ構造を表すクラスとビジターデザインパターンを用いて中間データ構造をたどるスケルトンを生成するツール Abstract Syntax Tree Generator(ASTG) を開発した [2]。ASTG を FOG 開発に適用した結果、中間データの構造を簡潔に記述できるとともに、構文木変換、型チェ
ッカ、コードジェネレータすべてをビジターデザインパターンに基づくアーキテクチャーで作成でき、記述すべきプログラムサイズを 40% 削減できた。ツールの可搬性を考慮して FOG、ASTG は Java で作成した。

以下、2 章で NMS のアーキテクチャーと FOG の構成を述べ、3 章で ASTG の機能と利用方法を説明し、4 章で FOG の開発に ASTG を利用した結果を述べ、5 章で ASTG の効果、今後の課題について議論する。

2 FOG: NMS-AP 生成ツール

2.1 3 層 NMS アーキテクチャー

ターゲットとする OSI ネットワーク管理 [10] を行なう NMS プラットフォームは図 1で示すような 3 層アーキテクチャーからなる。

GUI GUI はネットワーク管理者にインターフェースを提供する部分であり、ユーザインターフェースビルダなどで作成できる。

機能オブジェクト (FuO) FuO は MO の操作を行なうアプリケーションロジックを実装する部分である。GUI とは CORBA を用いて IDL のデータ型を、MO とは CMIS コマンドに似たプラットフォームの API を用いて ASN.1 のデータ型を受け渡す。FuO で IDL と ASN.1 のデータ型の変換も行なう。

管理オブジェクト (MO) MO は管理対象ネットワーク装置 (NE) のソフトウェアモデルであり、GDMO テンプレートと ASN.1 により定義される。MO は永続化のためのデータベース (DB) とのインターフェースと NE とのインターフェースを持つ。

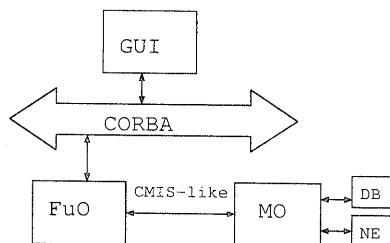

図 1: NMS プラットフォーム

2.2 FScript

FuO は、アプリケーションロジックを記述する部分であり MO 操作の処理、IDL と ASN.1 のデータ型の変換や例外処理などが混在し複雑である。FuO を記述するために FScript というスクリプト言語を開発した [8] [12]。FScript は ASN.1、GDMO、IDL を統一的に扱うことができるとともに、CMIS[1]相当のプラットフォーム API を仮想化したコマンドを用いることができる。

[1]CMIS は OSI 管理のためのコマンドである。[10]

2.3 FOGの構成

図2にFOGの構成を示す。
FOGの各構成要素の役割は以下の通りである。

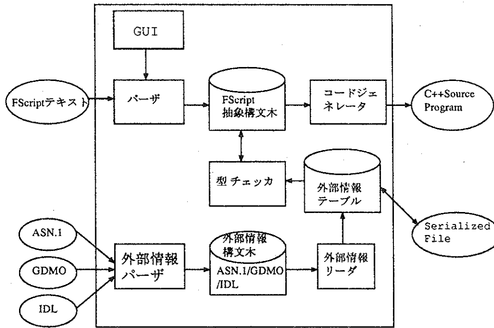

図 2: FOGの構成

GUI FScript記述するための構文エディタである。管理オブジェクト(MO)を操作する記述の支援として専用の入力形式を提供する。

パーザ FScriptテキストまたはGUIからの入力情報から、構文解析を行ない、FScriptの抽象構文木を作成する。パーザ作成にはJavaCC [11] を用いた。

抽象構文木 FScriptの中間データ形式であり、型チェッカ、コードジェネレータなどから参照される。

型チェッカ FScript抽象構文木の変数を表すノードに型付けを行なったり、外部情報テーブルを用いてASN.1/GDMO/IDLとの整合性を静的に検査する。

外部情報パーザ ASN.1、GDMO、IDLの3つのパーザからなり、それぞれについて具象構文に沿った構文木を作成する。

外部情報リーダ 外部情報の構文木を入力としFOGで用いる外部情報テーブルを作成する。

外部情報テーブル FOGの型チェックに必要なASN.1、GDMO、IDLに関する情報を保持する。

コードジェネレータ FScriptの抽象構文木からNMSプラットフォームで作動するC++プログラムを生成する。

2.4 再利用

FScriptのようなドメイン依存言語を用いた場合の再利用方法を説明する。図3で示すようにプラットフォームが移行した場合、アプリケーションジェネレータであるFOGを修正することによりFScriptによるAP設計情報とASN.1などのAPライブラリを再利用できる。さらに、FOGようなアプリケーションジェネレータを作成するためのツールキットは、プラットフォームの移行の際にも利用できる。

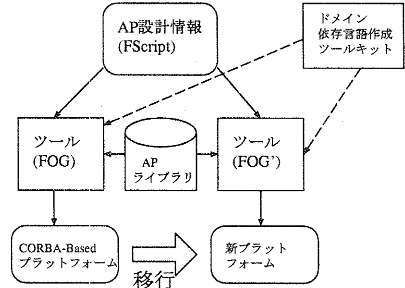

図 3: ドメイン依存言語を用いた再利用

3 ASTG

FOGのような、トランスレータの開発では言語処理についての開発工数が大きいことが問題である。そこでトランスレータの構成要素のうち抽象構文木などの中間データ構造に着目したツールキットAbstract Syntax Tree Generator(ASTG)を開発した。

3.1 ASTGの機能

ASTGは抽象構文木、パーザ、型チェッカ、コードジェネレータ、などから構成されるトランスレータをJavaで作成することを目的とする。ASTGは抽象構文木のデータ構造をAbstract Syntax Description Language for Java(ASDLJ)により簡潔に記述し、この記述から中間データを表すJavaのクラス、中間データの内部データをダンプするプログラム、中間データをたどるためのスケルトンを生成する。このスケルトンを利用することで、型チェッカやコードジェネレータを作成できる(図4参照)。

3.2 抽象構文木記述言語: ASDLJ

抽象構文木記述言語(ASDLJ)は、Javaの抽象構文木などのデータ構造を記述するための言語であ

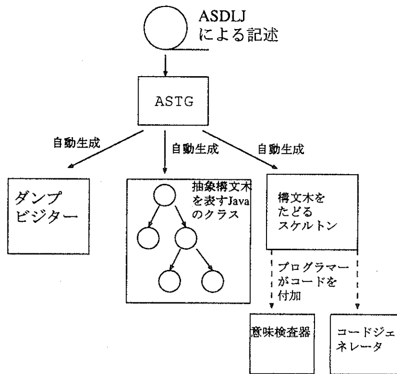

図 4: ASTG

り図 5に記述例を示す。

```
Exp ::= Binary(BOperator op,Exp l,Exp r)
      | Unary(UOperator op,Exp exp)
      | Int(String val)
      | Bool(Bconst val)
      | Literal(String val)
      | Bra(Exp exp)
      %attributes(Type? type)
BOperator = PLUS | MINUS | MUL | DIV
          | AND | OR
UOperator = NEGATIVE | NEGATE
Bconst    = TRUE | FALSE
Type = INT | BOOLEAN | STRING
```

図 5: ASDLJ による抽象構文木の記述例

1. 一般のクラス
継承関係に関係ない一般のクラスを定義する場合

`<クラス名> = ( <型名> <メンバ名>, ... , <型名> <メンバ名> )`

の形式で記述する。"="の後に"("が来るのが特徴であり、型名としてはint型などのJavaのプリミティブ型やjava.util.Hashtableなどのクラスライブラリの型を指定することができる。

```
token = (String image, int Line, int Column)
```

の部分はString型のメンバimage、int型のメンバLine、int型のメンバColumnを持つクラスtokenを定義する。クラス名、型名、メンバ名にはアルファベットの大文字または子文字で始まる英数文字列を用いることができる。

2. 継承関係を表すクラス
継承関係を用いたクラス表す場合、"::="を用いて記述する。"::="の左辺にスーパークラス、右辺にサブクラス名、を記述する。また左辺のクラスのメンバは`%attribures( )`の中に記述する。例えば、図5のExpクラスはBinaryからBraまでの6つをサブクラスとして持ち、またType型のメンバtypeを持つ。

3. 列挙型
列挙型(Enum型)に対応するクラスは次の形式で記述する。

`<クラス名> = <要素 1> | <要素 2> | ... | <要素 n>`

図3のBOperatorクラスはPLUS,MINUSなど6つの要素を持つクラスとなる。

4. List型のクラス
メンバの型として`<型名>*`のように"*"を用いることによりList型のクラスを指定することができる。

```
Block = (Stmt* stmts)
```

と記述すれば、BlockクラスはStmtのリストに対応するクラスをメンバとしてもつ。またリスト特有の、n番目の要素をとるなどのメソッドも自動的に生成される。

5. オプショナルなメンバ
型名の後に"?"をつけることによりそのメンバがオプショナルであることを示す。図5Expクラスのメンバtypeはオプショナルである。

3.3 ビジターアーキテクチャー

ドメイン依存言語からC++などの汎用言語へのトランスレータの構成要素として、型チェッカやコードジェネレータがある。これらは、各抽象構文木のノードにそれぞれ型チェックとコードジェネレートメソッドを実装するという方法でも実現できるが、型チェックという一つの機能を実現するためのメソッドが複数のクラスに分離し保守性が悪い。関連ある操作を一つのクラスにまとめた型チェックビジター、コードジェネレートビジターを作成するというアーキテクチャーで作成すれば、プログラムの構造がわかり易くなるとともに、図3のプラットフォーム移行時にコードジェネレートビジターのみの変更で対応できる。ASTGはビジターアーキテクチャーに基づき構文木をたどるスケルトンを自動生成する。

3.4 スケルトン生成とその利用例

ASTGは構文木に対して、ビジターパターン[6]に似た形の構文木をたどるスケルトンを生成する。スケルトンは、構文木の各クラスをを引数とし(Javaのプリミティブ型以外の)メンバを再帰的にたどる、visit<クラス名>というメソッドがからなる。図5の抽象構文木のUnaryクラスに対応するスケルトンを図6に、このスケルトンを用いて型チェックを行なう例を図7で示す。ここでは、if文を埋め込むことにより型チェックを行ない、たどる必要のないvisit_UOperator(s.op); の部分はコメントアウトしている。

```
public void visit_Unary (Unary s){
    //inheritType
    //instance variables
    //UOperator op
    //Exp exp

    visit_UOperator(s.op);
    visit_Exp(s.exp);
}
```

図 6: 構文木をたどるスケルトン

```
public void visit_Unary (Unary s){
    //inheritType
    //instance variables
    //UOperator op
    //Exp exp

    //visit_UOperator(s.op);
    visit_Exp(s.exp);
    if(s.exp.type.number == Type.BOOLEAN) {
        s.type = new Type(Type.BOOLEAN);
    } else {
        System.out.println("TypeCheckError");
    }
}
```

図 7: 型チェックの例 (Unaryクラス)

3.5 ダンプビジター

ASTGは抽象構文木のデータ構造をたどりながら、木構造をテキスト表現にして出力するダンプビジターも自動生成する。ダンプビジターは抽象構文木が図8の状態に対して、図9のようなテキスト表現を出力する。抽象構文木をたどりながらスケルトンに埋め込むコードを、木構造に対応するインデントの制御と型名、メンバ名、値の出力のコードを決定したことにより、ダンプビジターの自動生成が可能となった。

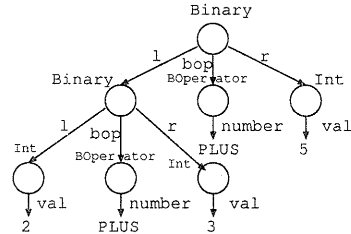

図 8: 抽象構文木の状態

```
[Binary]
    op [BOperator]
        number [int] : PLUS(1)
    l [Binary]
        op [BOperator]
            number [int] : PLUS(1)
        l [Int]
            val [String] : 2
        r [Int]
            val [String] : 3
    r [Int]
        val [String] : 5
```

図 9: ダンプビジターの出力例

4 ASTGのFOGへの適用

FOG(図2参照)の開発にあたり、外部情報パーザ、外部情報構文木(他のプロジェクトと共通で利用し既に作成済みであった)とGUI(抽象構文木と関係がない)以外の部分に対してASTGの生成したコード、スケルトンなどを適用した。各モジュールについての適用結果を報告する。

4.1 FScript パーザ

FScript パーザは図 10の線で囲まれた部分に対応し、FScript のテキストを入力とし FScript 抽象構文木を次の順序で作成する。図 10で太線で囲まれたモジュールは ASTG により自動生成された部分である。

FScript のパーザは、FScript テキストの文法に即した構文木 (具象構文木) を作成し、その後、型チェックやコード生成に適した抽象構文木に変換するという 2 段階に分けた。意図した構文木が作成できているかどうか確認するためにダンプビジターとプリティプリンタを用いた。

パーザ パーザは文法に従って構文解析し FScript 構文木を生成する部分であり JavaCC [11] というパーザジェネレータを用いて作成した。パーザのデバッグには、ダンプビジター (構文木) を用いた。

FScript 構文木 FScript 構文木は、84 個の Java のクラスからなり ASDLJ により 210 行で記述できた。

構文木変換 構文木をたどるスケルトンに FScript 抽象構文木のデータ構造を作成する部分のコードを埋め込むことにより人手で作成した。

FScript 抽象構文木 FScript 抽象構文木は、92 個の Java のクラスからなり (コメントを含んで) 240 行の ASDLJ により記述できた。型チェッカやコードジェネレータなど多数のモジュールから参照される部分であり、詳細設計段階で ASDLJ 記述に対してレビューを行なった。

プリティプリンタ FScript 抽象構文木をたどるスケルトンにコードを埋め込むことにより人手で作成した。

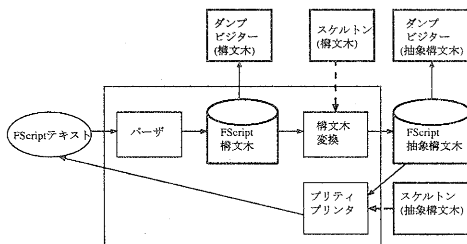

図 10: FOG(パーザ)

4.2 外部情報リーダ、外部情報テーブル

外部情報テーブルの作成 (図 2参照) には、ASN.1、GDMO、IDL という外部情報それぞれについてパーザ、構文木、リーダ、テーブルが必要である。パーザと構文木の部分については、他のプロジェクトと共通のものであり、特に構文木は具象構文の文法に沿って作成された。外部情報リーダは、ASN.1、GDMO、IDL の構文木から外部情報テーブル作成する部分である。テーブルは FOG の意味検査とコード生成に必要な情報のみをテーブルにしたもので ASTG により自動生成した。

FOG では、外部情報テーブルをコンパクトに設計すると同時に Java オブジェクトのシリアライズ化の機能を用いることで永続化した。ASN.1 や GDMO の定義ファイルには数万行というものもある。これらのファイルのパーズには分単位の時間がかかり無視できない。FOG において、永続化した外部情報テーブルを読み込むことにかかる時間は、外部情報ファイルをパーズし構文木からテーブル作成にかかる時間の約 15 分の 1 である。

4.3 型チェッカ、コードジェネレータ

型チェッカ、コードジェネレータはどちらも図 7と同じように構文木をたどるスケルトンに対して必要な操作を行なうコードを埋め込むことで実現した。両方とも、たどる対象の構文木は FScript 抽象構文木である。型チェッカの作成では外部情報テーブルを参照しながら型付けを行なうコードや代入文の両辺の型が互いに代入可能かどうか調べるコードを埋め込んだ。コードジェネレータの作成では型情報による場合分けを行なった上で C++ のプログラムを出力するコードを埋め込んだ。

5 議論

5.1 ASTG の適用効果

ASDLJ により FScript の抽象構文木のデータ構造を記述したものを設計文書とし ASTG により生成するという方法をとった。FScript 抽象構文木には数回にわたり変更がおこなわれたが最新のプログラムと設計文書の一致が最後までとれた点が最大の適用効果と思われる。

FOG 開発において、ASTG が関連する部分 (外部情報パーザと GUI を除いた部分) に対してプログラマが記述したコード量は 33KL であった。仮に ASTG が自動生成したコードもプログラマが記述する場合のコード量は 54KL となる。これはプログ

ラマの記述量を約40%削減するとともに、自動生成した部分についてはバグが混入しないので、デバッグの際にバグ原因を特定しやすかった。例えば、パーザが構文木を作成した場合、正しい構文木が作成されているかどうか確かめるために、プログラマが構文木をテキスト情報にプリントするメソッドを記述してそれを用いて確認する。間違った結果が得られた場合、パーザのバグなのか対応する構文木のプリントメソッドのバグなのか判定できないが、ダンプビジターのダンプ結果が間違っている場合はパーザのバグとすぐにわかる。

ASTGはJavaプログラムを生成するので、プロジェクトに新しく入る人にも抽象構文木の仕様がわかりやすい。FOG開発プロジェクトでは、設計段階は4人であったがコーディング段階では11人となっている。プロジェクトに途中から参加する人(一般にプログラムの専門家)が、設計文書だけから仕様を理解するのは難しいが、ASTGが自動生成する構文木を表すJavaのプログラムと構文木をたどるスケルトンのプログラムを一緒に提示するとプログラムのコーディングイメージが湧きやすくなることで仕様の理解が得られた。このことにより、設計者からプログラマへのスムーズな業務の移行が可能となった。

構文木を表すクラスに、いろんなモジュールから使われるメソッドを追加プログラマが追加する場合がある。同時に、ドメイン依存言語の実装においてユーザからの要求などにより、抽象構文木のクラスやメンバを増やすなどの拡張が必要となる場合がある。この場合ASDLJによる記述を変更し再びクラス定義を生成することになる。ASTGが生成するクラスがすでに存在する場合、再生成したクラスにプログラマが追加したメソッドなどがそのまま残るようにした。一度、クラス定義を作成したら終りではなく、後で修正, 拡張が必要になったとき簡単に移行できることが重要である。

図6の構文木をたどるスケルトンは[6]に記述されているビジターデザインパターンは厳密に比較すると少し異なる。関連する操作を1つにまとめている点で両者は同じであるが、スケルトンはビジターの抽象クラスのサブクラスになっておらずスケルトン内でメソッドを連鎖的に呼びツリーをたどっていく点で、[6]に記述されているビジターデザインパターンと異なる。Javaでビジターの抽象クラスのサブクラスにすると、引数や戻り値(一般にvoid)を抽象クラスと一致させなければならないという制約からスケルトンに埋め込めるコードに制約が生じるが、メソッドを連鎖的に呼び出す場合はメソッドの戻り値で情報を渡すこともできる。またビジターデザインパターンに比べメソッドの連鎖の方が一般に構文木の変更に強い。

5.2 関連研究との比較

Zepherの Abstract Syntax Defintion Language (ASDL) [14]は、言語に依存しないコンパイラの中間表現である抽象構文木を記述することを目指し、ツールによりC、C++、Java、MLなどの抽象構文木を表すクラスを生成する。Zepherでは様々なプログラミング言語に対応するため、記述できる型が制限されている。一方、ASDLJではJavaで抽象構文木のデータ構造を記述する場合に必要な型は記述できるようにするという考え方のもとに設計されている点で異なる。

KHEPHA [5]では、DSLの実装はパーザ、構文木変換、プリティプリンタによるコード生成により実現できると主張し、Fortran 90のサブ言語を構文木変換ルールの記述言語として用い、この変換ルールの記述から構文木変換を行なうコードを生成する。KHEPHAでは型チェックも構文木変換により行なう。

5.3 今後の課題

FOG開発では、型チェックやC++プログラム生成の部分は先に述べたように、ASTGが自動生成したスケルトンコードに、人手で型チェック処理などを埋め込む方法を採用した。今後は、型チェックやプログラム生成を行なう処理についても、簡潔な表現からの自動生成を行なうツールキットを開発したいと考えている。

文献[13]で出力テンプレートと呼ぶ形式で生成するC++プログラムのパターンを記述する方法を提案した。出力テンプレートの記述は、生成C++プログラムのイメージと簡潔な抽象構文木へのアクセスからなり、プログラム生成に必要な情報の良い表現となっている。しかし、出力テンプレート方式では構文木へのアクセス方法を汎用化しているので、作成されたコードジェネレータの動作速度が遅いという問題点がある。本稿の方法との使い分けが重要である。

また、式への型付与や型検査あるいは型推論といった処理を簡便に行なおうとする研究があり、BANE [1]等が知られている。BANEを用いると、型に関する規則から型検査を行なうプログラムを自動生成することができる。しかし、FScriptの場合、扱うデータ型は、ASN.1、IDL、GDMOのオブ

ジェクト定義、等のように非常に多い。例えば代入式の左辺と右辺の型整合性を検査するためには、非常に多くの組み合わせを考えなければならない。すなわち、型に関する規則をコンパクトに表現することと同時に、組み合わせの網羅性をチェックできるような表現形式が重要となる。

6 まとめ

ドメイン依存言語とトランスレータによるアプリケーションジェネレータは、特定領域アプリケーションの生産性を向上させるが言語処理が難しいことと開発工数が大きいことが問題である。そこで、抽象構文木などの中間データ構造の簡潔な記述方法を提案し、その記述から Java のクラス定義と構文木をたどるスケルトンを自動生成するツール ASTG を作成した。 FScript というドメイン依存言語から C++ のアプリケーションを生成するトランスレータである FOG の開発に適用したところ、 92 個の Java クラスからなる FScript の抽象構文木定義を 240 行で記述でき、プログラマが記述するコード量を 40% 削減することができた。

FOG 開発メンバである NEC C&C メディア研究所 友部主任、第一伝送事業部 三木主任、 NEC 情報システムズ 加藤主任に感謝します。

参考文献

[1] Alexander Aiken, Manuel Fähndrich, Jeffrey S.Foster, Zhendong Su: A Toolkit for Constructing Type- and Constraint-Based Progam Analysys, http://www.cs.berkeley.edu/Research/Aiken/bane.html

[2] 別府, 登内, 中島: 問題向け設計言語開発ツールキットの開発と適用, 情報処理学会 第 56 回全国大会論文集, pp.277-278(1998)

[3] J.Craig Cleavel: Building Application Generators, *IEEE Software*, pp.25-33(July 1988)

[4] Arie van Deursen and Paul Klint: Little Language: Little Maintenance?, *Proceeding of DSL '97*,pp.109-127,(1997)

[5] Rickard E.Faith, Lars S.Nyland, Jan F.Prins: KHEPHA: A System for Rapid Implementation of Domain Specific Languages, *Proceedings of Conference on Domain-Specific Languages*, pp.243-255, USENIX(1997)

[6] Gamma, E., Helm, R., Johnson, R. amd Vlissides, J.: *Design Pattrns: Element of Resunable Object-Oriented Software* , Addison-Wesley(1994)

[7] Richard B.Kieburtz and et al: A Software Engineering Experiment in Software Component Generator, *Proceedings of ICES-18*, pp.542-555, IEEE(1996)

[8] M.Miki, M.Tanaka, M. Tomobe, S.Nakajima : A Scripting Language for Network Management Applications and its Related Tool, *Proceeding of GLOBCOM'97*, pp.1714-1718, IEEE(1997)

[9] 中島, 友部, 三木: 問題向け設計言語ベースのＡＰ構築支援ツール, オブジェクト指向最前線, 情処・ＯＯ'97 シンポジウム, 朝倉書店, pp.11-14(1997)

[10] 大鐘久夫: TCP/IP と OSI ネットワーク管理, ソフトリサーチセンター (1993)

[11] Sun MicroSystems: JavaCC Documentation, http::/www/suntest.com/JavaCC/DOC

[12] 友部, 中島, 三木: 問題向け設計言語によるアプリケーション構築支援環境の構築, 日本ソフトウェア科学会 第 14 回大会論文集,pp.585-588(1997)

[13] 登内, 中島: 出力コードを容易に変更できる GDMO トランスレータ, 日本ソフトウェア科学会 第 14 回大会論文集,pp.1-4(1997)

[14] Daniel C.Wang, Andrew W.Appel, Jeff L.Korn, Christopher S.Serra: The Zephyr Abstract Syntax Description Language, *Proceedings of Conference on Domain-Specific Languages*, pp.213-227, USENIX(1997)