構造指向型システムのための実行可能な仕様記述言語

今 泉 貴 史[†] 権 藤 克 彦[†] 萩 原 威 志[††]
松 塚 貴 英[†] 片 山 卓 也[†,†††]

ラピッドプロトタイピングにおいては，抽象度の高い実行可能な仕様記述言語が，大きな役割を演じる．本論文では，仕様を実際に動作させることのできる記述言語である計算モデル OOAG について述べる．OOAG は，属性文法に対してシステムの動的な側面も記述できるようにオブジェクト指向の拡張を行ったモデルである．また，実際に OOAG を用いて UNIX ファイルシステムを記述することにより，OOAG が構造指向型システムを記述する能力を有することを確認した．

An Executable Specification Language for Structure Oriented Systems

TAKASHI IMAIZUMI,[†] KATSUHIKO GONDOW,[†] TAKESHI HAGIWARA,[††]
TAKAHIDE MATSUTSUKA[†] and TAKUYA KATAYAMA[†,†††]

A specification language is important in rapid prototyping area in software development environment. In this paper, we explain a computational model OOAG. Using this model, we can execute the specification just written in OOAG based language. OOAG is a model based on attribute grammars and extended by object oriented concepts for describing system's dynamic behavior. We show the ability of OOAG for describing structure oriented systems, by describing UNIX file system using OOAG.

1. はじめに

1.1 実行可能仕様記述言語

これまで，大規模なソフトウェアに高い信頼性と保守性を与えるために，さまざまな研究がなされてきた．例えば，プログラムの自動合成や正当性検証などが挙げられる．

プログラムの自動合成では，仕様だけを与え，その仕様を満たすようなプログラムを自動的に合成する．ソフトウェアの仕様は，プログラミング言語と比較し，より抽象度の高い言語を用いて記述する．そのため，読解性の高いものとなり，ソフトウェアの保守性が上がる．

正当性の検証では，ソフトウェアの仕様や実際のプログラムを機械的に検証することにより，その「正しさ」を調べる．この正しさには，与えられた仕様が完全なものとなっているか，矛盾していないかなどが含まれる．

ソフトウェアの仕様を記述するための言語が仕様記述言語である．仕様記述言語には，あいまいでなく，読解性が高く，抽象度が高いことが求められる．また，実際に仕様を記述する段階では，どんな複雑な問題でも任意の抽象レベルで構成的に記述できることも必要となる．

われわれは与えられた仕様を実行できる（もしくは，仕様からプログラムを自動合成できる）ことが重要であると考える．これは，仕様とプログラムの不整合を回避したり，プロトタイプの作成が容易となる点からである．

記述した仕様の正当性が検証できた後，実際に動作するソフトウェアを得るために別なプログラミング言語を用いる場合，仕様からプログラムに変換する段階で誤りを加えてしまう危険性も考えられる．もし記述した仕様を動作させることができれば，このような誤りを防止することができる．

また，仕様をそのまま動作させることが可能ならば，プロトタイプシステムや実際のシステムとして利用できる．抽象度の高い仕様を記述した場合でも，システムの基本となる部分は動作可能である．これをプロト

† 東京工業大学情報理工学研究科計算工学専攻
Department of Computer Science, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
†† 東京工業大学工学部情報工学科
Department of Computer Science, Faculty of Engineering, Tokyo Institute of Technology
††† 北陸先端科学技術大学院大学情報科学研究科
Japan Advanced Institute of Science and Technology

タイプシステムとして実際に実行しながら仕様を作成する初期の段階で誤りを検出することもできる.

このような,実行可能仕様記述言語を考える場合,仕様とプログラムの間に区別はない.これは,仕様からプログラムの自動合成を行って実行可能となる場合でも同様である.ここでは仕様記述言語の中で実行可能なものを考えたが,逆に,プログラミング言語でも先に述べた仕様記述言語の性質を満たすものは実行可能仕様記述言語と呼ぶことができるだろう.

実行可能仕様記述言語がプログラミング言語と本質的に異なるのは,プログラムを動作させるための細かな指示を行わず,システムの持つ意味を記述する点である.そのため,システムの性質を記述できるような言語体系とする必要がある.

1.2 仕様記述言語としてのOOAG

われわれは,実行可能仕様記述言語であり,構造指向型システムの記述に適した計算モデルOOAGを開発している[1)~3)].計算モデルOOAGは,属性文法に対してオブジェクト指向的な拡張を行うことによりシステムの動的な側面も記述できるように拡張された計算モデルである.この計算モデルが対象とするシステムは,基本的構造として木構造を持ったシステムである(これを構造指向型システムと呼ぶ).

構造指向型システムとしては,構文木を扱うようなコンパイラや構造エディタなどが代表的である.一般的に,ソフトウェアは数多くのモジュールから構築されるが,この包含関係を親子関係と見れば,ほとんどのシステムは基本構造として木構造を持っているとみなすことができる.複数のモジュールから参照されるライブラリを表現する場合,木構造ではなくDAGとなってしまうが,ライブラリを含むモジュールを複数の場所から参照していると考えれば,木構造として扱うことができる.

計算モデルOOAGの基本モデルとなっている属性文法は,1968年にD. E. Knuthにより提案された計算モデルであり[4),5)],コンパイラの記述とその自動生成を目的とした研究が進められてきた[6)].コンパイラが対象とする言語の仕様を属性文法により記述しておき,その記述に基づいてコンパイラを生成するのである.属性文法は,すでに仕様記述言語としての成果を上げているといえる.属性文法の応用はコンパイラに限られるものではなく,構造エディタの仕様を属性文法型言語で指定しておき,この記述に基づいて構造エディタを生成するシステムも開発されている[7),8)].

属性文法を一言でいうと,

文脈自由文法+属性+意味規則

となる.つまり,文脈自由文法の非終端記号に対して属性を付加し,その属性の値を計算するための意味規則を加えたものである.属性の計算方法は,1つの生成規則の中で意味規則として記述される.そのため,属性文法の記述は局所的となる.また,計算の実行順序は属性の間の依存関係により自動的に決定されるため明示的に指定する必要はない.

基本的な属性文法が計算の対象とするものは,文脈自由文法に基づいて構築された構文木である.つまり,すでに作成された構文木の上に属性を張り付け,その属性間の関係を記述することによりシステムの性質を記述する.入力文字列をパーズし構文木を構築するプロセスは属性文法とは別に定義するものであり,属性文法にとってパーズは必須なわけではない.文脈自由文法と,その生成規則にしたがって作成された木構造があれば,その上で計算を行うことができる.したがって,適当な生成規則を与えることにより,基本的な構造が木構造となるような構造指向的システムの性質を記述することができる.

また,属性文法は関数的な計算モデルである.属性は通常のプログラミング言語における変数のような役割を示すが,値を代入するのは1回のみであり,一度値が決まればそれ以降変更されることはない.プログラムの検証系を構成するために,この属性文法の関数性は良い性質であるといえる.実際,属性文法型プログラミング言語の環境の一部として,検証系も存在する[9)].

計算モデルOOAGでは,システムの動的な側面も記述するために属性文法に対してオブジェクト指向的な拡張を施している.属性文法は,でき上がった構文木の上で計算を行うため,システム内部のさまざまな値の関係を記述するには向いているが,データベースシステムのように保持するデータが刻々と変化するようなシステムを記述するには向いていない.この点を解決するために,Higher Order Attribute Grammars[10),11)]やTransformational Attribute Grammar[12)]なども研究されているが,いずれの場合にも木構造の変更は文法中に静的に記述された範囲でしか行えない.

OOAGにおけるメッセージは,一時的に計算する規則を指定すると同時に,自分自身の構造を変更する機能を持っている.そのため,OOAGを用いることにより,構造を変更させながら動作し続けるシステムを記述することができる.メッセージにより指定された計算を行う場合にも,属性文法の基本的特性である関数性や単一代入性は保持している.そのため,この拡張

により属性文法の特徴である記述の局所性や読解性には影響を与えない．

1.3 本論文の構成

本論文では，構造指向型システムを記述するための仕様記述言語 OOAG について，その計算モデルや言語について述べる．2章では，計算モデル OOAG とその記述言語である言語 OSL, OOAG を用いて仕様を記述・実行するための環境である MAGE について簡単に述べる．3章で UNIX ファイルシステムを題材として実際に記述を行い，この記述実験により OOAG の抱える問題点を洗い出し，4章で考察する．

2. 計算モデル OOAG と記述言語 OSL

この章では，計算モデル OOAG のための記述言語 OSL による例題を用いて，OOAG の計算原理の概略を説明する．詳細は文献 2), 3) を参照してほしい．また，OOAG のための統合環境 MAGE に関しても簡単に説明する．

2.1 概 要

属性文法は，文脈自由文法が生成する文の導出木上の各ノードに，属性と呼ばれる（単一代入性を持つ）変数を付随した形式的体系である．属性の値の計算方法が，(1) 文脈自由文法の各生成規則ごとに分割して記述されること，(2) 関数的であることの2点から，属性文法による記述は局所性・読解性が非常に高いといわれている．

その一方で，コンパイラ以外の応用が少ないのは，その関数的な記述だけでは書き難いシステムが多いためである．いったん計算された属性つき木（導出木とその上の属性値）は不変であり，属性つき木を入力データとした計算の記述や，属性つき木自身を更新する手段は属性文法にはない．

Synthesizer Generator[7),8)] は，ユーザによる部分木の編集作業を許すことで，属性文法の欠点を克服しようとしている．また高階属性文法は，(1) 属性つき木の部分木を入力データとする属性計算の記述，(2) 属性値に依存した部分木の計算の記述，を純粋に関数的な方法で導入している．

OOAG は，これらのアプローチをさらに進めて，属性つき木自身を計算対象とする記述法を導入して，属性文法を拡張した計算モデルである．属性文法の利点を失わないことを狙った結果，拡張部分の記述法は属性文法形式であり，かつ拡張部分は関数的に計算される計算モデルとなった．

OOAG では属性つき木（の任意の部分木）をオブジェクトと呼ぶ．OSL によるオブジェクト記述の構成を以下にまとめる．

- **静的仕様記述**：通常の属性文法の記述にほぼ相当
  - **構成規則**：オブジェクトの構造の定義
  - **静的意味規則**：**静的属性**の関数的定義
- **動的仕様記述**：OOAG 特有の記述
  - **計算規則**：メッセージ送受の定義
  - **動的意味規則**：**動的属性**と**固有属性**の関数的定義

OOAG の計算は，属性文法の関数的計算（**静的計算**と呼ぶ）と，拡張部分の関数的計算（**動的計算**と呼ぶ）を交互に繰り返すことで進む．この静的計算と動的計算では，1つの木構造に対して処理を行う．動的計算においては，次のループで用いる固有属性の値（つまり木構造自身）を計算する．ここで計算された新しい木構造が次のループで用いられる．したがって，木構造を状態を表すものと考えると，動的計算を終了し，次の静的計算を行う前に状態遷移が起こることになる．動的計算部分をより詳しく分解すると，OOAG の計算は1つのループで次の4つのフェーズを繰り返す（図1）．

（1） 静的意味規則にしたがって静的属性の値を計算する．この時点で動的属性は存在しない．
（2） 計算規則にしたがってメッセージが流れる．動的意味規則を属性つき木に張り付ける．
（3） 張り付けられた動的意味規則にしたがって，動的属性の値と次のループで有効となる固有属性の値（オブジェクト）を計算する．

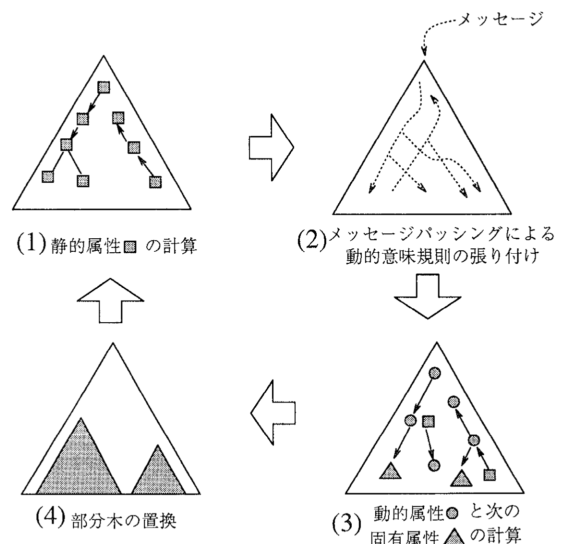


図1 OOAG の評価ループの概念図
Fig. 1 Evaluation loop in OOAG.

（4） 計算した固有属性の値を実際に置換する．動的属性は消滅する．

2.2 静的仕様記述

2.2.1 構成規則

次の記述は構成規則の例であり，「クラス exp がクラス INT から構成されるか，あるいは 2 つの exp から構成される」ことを示す．

```
class exp->Const[INT]
class exp->Sum[exp, exp]
```

ここで各構成規則を **RHS クラス**と呼び，Const や Sum を **RHS クラス名**と呼ぶ．RHS クラス名はこのクラスのインスタンスを生成する関数記号（コンストラクタ）としても使用される．同じ左辺（ここでは exp）を持つ構成規則を総称して **LHS クラス**と呼び，exp を **LHS クラス名**と呼ぶ．つまり，LHS クラス名は，一般に元となる文脈自由文法の非終端記号である．なお，INT は整数を表すプリミティブな LHS クラスである．他にもプリミティブな LHS クラスがあるが，これらは文脈自由文法の終端記号である．

－＞の右側の LHS クラスを**固有属性**と呼ぶ．例えば，Const の固有属性は INT，Sum の固有属性は exp$2 と exp$3 である．同じ LHS クラス名を区別するために左側から$1，$2，…と番号をつける．

構成規則では静的属性の宣言も行う．次の例では，同じ左辺 exp を持つ 2 つの構成規則を 1 つにまとめた上で，LHS クラス exp に，静的相続属性（begin）と静的合成属性（end）を宣言している．

```
class exp (begin | end)
  ->Const[INT]
  ->Sum[exp (begin | end),
        exp (begin | end)]
```

「相続」「合成」という分類は，直観的にいうと関数の入力と出力に相当する．

2.2.2 静的意味規則

各構成規則には静的意味規則が付随する．次の記述は，構成規則 Sum とそれに付随する静的意味規則の集合（{と}の間）の例である．

```
class exp (begin | end)
  ->Sum[exp (begin | end),
        exp (begin | end)]
    {local pos;
     pos=exp$1.begin;
     exp$1.end=exp$3.end;
     exp$2.begin=plus (exp$1.begin, 1);
     exp$3.begin=exp$2.end;            }
```

local pos は静的局所属性の宣言であり，構成規則（ここでは Sum）に関連づけられる．

静的意味規則は，*output-attribute*＝*expression* という形をしている．構成規則を $X_0 \to f[X_1, \cdots, X_n]$ としたとき，*output-attribute* は次のいずれかである．

- X_0 の静的合成属性（exp$1.end）
- X_i（$1 \leqq i \leqq n$）の静的相続属性（exp$2.begin, exp$3.begin）
- f の局所属性（pos）

expression に出現してよい項は，次のいずれかである．

- X_i（$0 \leqq i \leqq n$）の静的属性（exp$1.begin, exp$1.end, exp$2.begin, exp$2.end, exp$3.begin, exp$3.end, pos）
- 固有属性 X_i（$1 \leqq i \leqq n$）(exp$1, exp$2)

plus は加算を示す関数記号である．関数記号が RHS クラス名の場合は，カッコではなく，特にブラケット（[と]）を用いる．例えば Sum の静的意味規則中で，Sum[exp$3, exp$2] は固有属性 exp$2 と exp$3 を入れ換えたオブジェクトを示す．

2.3 動的仕様記述

2.3.1 計算規則

次の記述は構成規則 Sum に付随する計算規則の例であり，「（親から）exp$1 にメッセージ find が来たら，子どもである exp$2 と exp$3 に同じメッセージ find を先送りする」ことを示す．

```
exp$1: find=>exp$2: find, exp$3: find
```

ここで＝＞の左辺を**入力メッセージ**，右辺の各要素を**出力メッセージ**と呼ぶ．メッセージ中の exp$1, exp$2, exp$3 はメッセージの通信ポートである．構成規則を $X_0 \to f[X_1, \cdots X_n]$ としたとき，通信ポートは X_i（$0 \leqq i \leqq n$）か **self** である．**self** は自分自身のノードを示す．

入力メッセージは 0 個以上並べてよい．例えば，次の例は「両方の子ども exp$2 と exp$3 から同じメッセージ up が来たら，（親への通信ポートである）exp$1 に up を送る」ことを示す．

```
exp$2: up, exp$3: up=>exp$1: up
```

同様に，兄弟間のメッセージパッシングも記述できる．入力メッセージが 0 個の例は 2.4.2 項で示す．

構成規則で静的属性の宣言をするのに対して，計算規則では動的属性の宣言を行う．次の例では動的相続属性（var）と動的合成属性（poslist）を宣言している．

```
exp$1: find (var | poslist)
    =>exp$2: find (var | poslist),
      exp$3: find (var | poslist)
```

この例では動的属性の宣言は 1 つずつだが，一般に 0

個以上の動的属性を宣言してよい．各動的属性の評価とメッセージの伝播は独立であり，手続きにおける手続き呼び出しとその引数の評価とは異なる．

2.3.2 動的意味規則

各計算規則には動的意味規則が付随する．動的意味規則は静的意味規則と同様に *output-attribute*＝*expression* という形をしている．構成規則を $X_0 \to f[X_1, \cdots, X_n]$ としたとき，*expression*は，$X_i (0 \leqq i \leqq n)$ の動的属性が出現してよいこと以外は静的意味規則のそれと同じである．*output-attribute* は，(1) 動的属性（X_0の動的合成属性，X_i $(1 \leqq i \leqq n)$ の動的相続属性，f の動的局所属性），(2) 固有属性 $X_i (1 \leqq i \leqq n)$，のいずれかである．動的属性や静的属性とは異なり，固有属性の意味規則は必ずしも定義する必要はない．

次の記述は動的属性の定義だけからなる動的意味規則の集合（｛と｝の間）の例である．

```
exp$1: find (var | poslist)
   =>exp$2: find (var | poslist),
       exp$3: find (var | poslist)
  {  exp$2.var=exp$1.var ;
     exp$3.var=exp$1.var ;
     exp$1.poslist=append (
            exp$2.poslist, exp$3.poslist) ; }
```

次の記述は固有属性を定義する動的意味規則の例（Sum に付随する）であり，「メッセージ swap を受け取ると，左右の子どもを入れ換える」ことを示す．

```
exp$1: swap ( | )=>
   {    (new exp$2)=exp$3 ;
        (new exp$3)=exp$2 ; }
```

左辺と右辺の固有属性が別の値であることを強調するために，キーワード new を付加している．また記述の簡便さを狙い，左辺に（new X_0）の出現を許す．例えば上の例は次のように書き直せる．

```
exp$1: swap ( | )=>
   {(new exp$1)=Sum[exp$3, exp$2] ; }
```

2.4 その他

本論文中で使用する OSL のその他の 4 つの機能：(1) with 式，(2) メッセージのガード，(3) オブジェクトの名前参照，(4) 外部オブジェクトをこの節で説明する．

2.4.1 with 式

オブジェクトの構成要素を選択するために with 式を用いる．with 式は Synthesizer Generator の with 式と全く同じである．with 式の形式は次のとおり．

with (*expression*$_0$) {
  pattern$_1$: *expression*$_1$,
  pattern$_2$: *expression*$_2$,
  …
  pattern$_n$: *expression*$_n$
  }

この with 式の値は，*expression*$_0$にマッチした最初の *pattern*$_i$に対応する *expression*$_i$の値である．

次の記述は with 式を用いた積和の式を和積の式に変更する動的仕様記述である．

```
exp$1: distribute ( | )=>
   {(new exp$1)=with (exp$1)
     {Prod[Sum[a, b], c] : Sum[Prod[a, c],
       Prod[b, c]],
       Prod[a, Sum[b, c]] : Sum[Prod[a, b],
       Prod[a, c]],
       a: exp$1} ; }
```

ここで a, b, c はパターン変数であり，任意の部分木にマッチする．

2.4.2 メッセージのガード

メッセージのガードの形式は以下のとおり．

input-messages
  case *condition*$_1$
  ⇒*output-messages*$_1$
   {*dynamic-equations*$_1$}
  …
  case *condition*$_n$
  ⇒*output-message*$_n$
   {*dynamic-equations*$_n$}

ガードがある場合，記述順で最初に真になった条件 *condition*$_i$に付随する出力メッセージ *output-messages*$_i$と動的意味規則 *dynamic-equations*$_i$が有効になる．条件中の式に出現する動的属性は，入力メッセージの動的入力属性だけが許される．

入力メッセージが空の場合にメッセージのガードを使えば，オブジェクトの構造や静的属性の値に依存して起動される計算規則を記述することができる．

次の記述は，除数が 0 の場合にメッセージ error を親に送信する（構成規則は class exp->Div[exp, exp]である）．

```
case (equal (exp$3.val, 0))
  =>exp$1: error ( | message)
   {
    exp$1.message= "divided by zero" ;
   }
```

2.4.3 オブジェクトの名前参照

オブジェクトの名前参照は，必ずしも親子関係のないノードを子ノードとして扱うことを可能にする．

o がオブジェクトを示す項であるとき，&o は o の名前である(この名前はシステム内でユニークである).逆に,n がオブジェクトの名前を示す項である時，＊ n は n が示すオブジェクトを示す.

構成規則中で，$X_i(1 \leq i \leq n)$ に＊マークがある場合，名前による別の木の参照を示す.

class X_0->f[…, ＊ X_i, …]

名前参照の場合，子ノードの静的相続属性の値を定義することはできない．この値は，実際の親が定義しているためである．この他にも，名前参照の使用にはいくつかの制限があるが，ここでは割愛する.

次の記述例では，メッセージ link が送られると，exp$1.name の値はノード req_pos のオブジェクト名となり，＊ exp はその名前のオブジェクトを参照する.

```
class top (|)
  ->Root[exp (begin|end), * exp (|end)]

  top: link (req_pos|)
    =>exp$1: get_name (req_pos|name)
     { exp$1.req_pos=top.req_pos;
       (new exp$2)=exp$1.name; }
```

メッセージ get_name はガードを使って要求されたノードに伝えられるとする。このノードにおける exp.name にオブジェクト名を束縛する定義は例えば次のようになる.

```
exp: get_name (req_pos|name)
  case (equal (exp.req_pos, pos))
    =>
     { exp.name=&exp; }
```

2.4.4 外部クラス

OOAG を用いて記述することが困難なオブジェクトは，外部クラスとして実現する．外部クラスは OSL 以外の言語で記述し，OSL の記述から呼び出す．OSL の記述中で，

```
external class tree_view (|);
```

と宣言して，LHS クラス tree_view が外部クラスとして実現されていることを示す.

外部クラスは，システムの概要設計を行っている段階で詳細部分を隠して実行する場合にも用いる．この場合の外部クラスは，人間が相続属性の値から合成属性の値を計算したり,メッセージに答える実現となる.

2.5 開発環境 MAGE

MAGE は OOAG による仕様記述のための統合環境で,OSL による記述を実行するための属性評価器を中心に，構造エディタ，記述の構造や実行状態を視覚的に表示するブラウザを用意し，記述の能率の向上を目指した構成になっている．現在のシステムは，オブジェクト指向 lisp 処理系である xlisp[13] により GUI 環境の構築が可能な WINTERP システム[14] を用いて，そのほとんどの部分を実現している．構造エディタ部分は，Synthesizer Generator を用いている.

2.5.1 MAGE の構成

ここでは，MAGE の構成要素であるツールの機能・特徴について述べる.MAGE のシステム構成を図 2 に示す.

システムブラウザ　システムブラウザは,MAGE の中心的ツールで，MAGE システム全体を制御する．主な役割は，MAGE のツール起動・終了，MAGE システムの状態表示である．ユーザのセッションは，このシステムブラウザの起動から始まる.

クラスブラウザと OSL エディタ　OSL 記述量の増大につれ，その記述構造の把握が困難になるため，MAGE では，クラスブラウザと専用の OSL エディタという 2 つのツールを用意している．OSL エディタは，Synthesizer Generator を用いて作成された構造エディタで，プログラムの編集と同時に静的な意味検査を行いエラーを逐次表示する機能を持つ．これにより lisp へのトランスレート・編集のくり返しを減らし，効率的な開発が可能になっている．クラスブラウザは，複雑な構造を持った OSL の構文をグラフィカルに表示することにより，その構造を素早く認識できるようにすることを目的としている．クラスブラウザは，クラス名に関する情報，宣言された属性・メッセージに関する情報，クラスの構造に関する情報などを提供し，OSL 記述の構造を簡潔に示す.

オブジェクトブラウザ　オブジェクトブラウザは，OSL 記述の実行制御や状態を表示するツールである．属性評価のフェーズごとにプロセッサの実行を制御でき，その時点での内部状態をチェックできるので，OSL 記述のデバッガとしての機能も有している．OOAG のオブジェクトは木構造を形成する．オブジェ

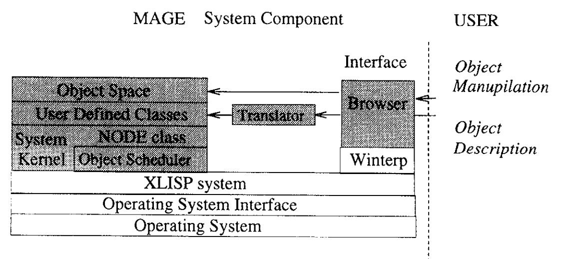


図 2 MAGE の構成
Fig. 2 Structure of MAGE.

クトブラウザはその実行中のオブジェクト構造を木で表現し，この木の上でオブジェクトを指定することにより，属性値を参照できる．

トランスレータ OSL エディタで編集した OSL 記述は，OSL エディタの機能の一部として実現されているトランスレータを用いて，xlisp プログラムに変換して MAGE システムに登録する．登録された OSL 記述は，クラスブラウザでその構造を確認したり，オブジェクトブラウザを用いて実行状態を参照しながら動作させることができる．

3．OOAG による仕様記述例

本章では，仕様記述言語としての OOAG の有用性を調べるため，OSL を用いて実際に仕様を記述した結果に関して報告する[15]．記述対象としては UNIX ファイルシステム[16),17)]を選んだ．これは，ファイルシステムが木構造としてデータ（ファイル）を管理しているためである．

3.1 記述対象

本実験では，ファイルシステムのみを記述の対象とするが，カーネル内のデータ構造は互いに作用しあっており，完全にファイルシステムだけを独立して設計することは困難である．そのため，ファイルシステムの記述に必要な他のデータ構造（プロセスなど）に関しても記述を行う．

UNIX ファイルシステム上には，通常ファイル，ディレクトリファイル，デバイスファイルなどがあるが，今回は通常ファイルとディレクトリファイルのみを扱う．実際にはファイルに格納されたデータが重要となるが，本実験ではファイルシステムを i ノードのデータベースとしてとらえる．つまり，ファイルの中身は扱わず，i ノードの処理に関してのみ記述を行う．また，単一のファイルシステムのみを対象とする．

3.2 データ構造の記述

ファイルシステムに関連してカーネルが持つデータ構造には，カーネル内に 1 つ存在するファイルテーブルと i ノードテーブル，プロセスごとに存在するファイルディスクリプタテーブルがある．

プロセスがファイルを処理する場合，ファイルディスクリプタを用いてファイルを指定する．この値は，

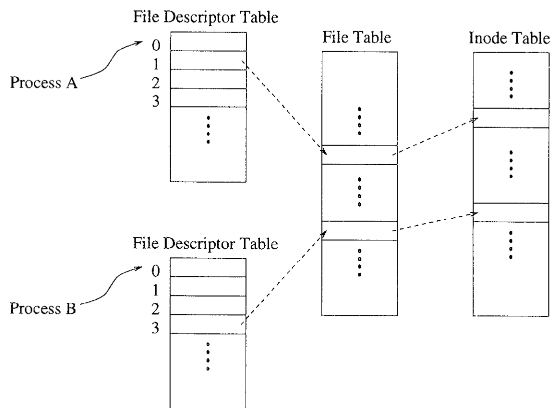


図3 カーネル内データ構造
Fig. 3 Data structures in the UNIX kernel.

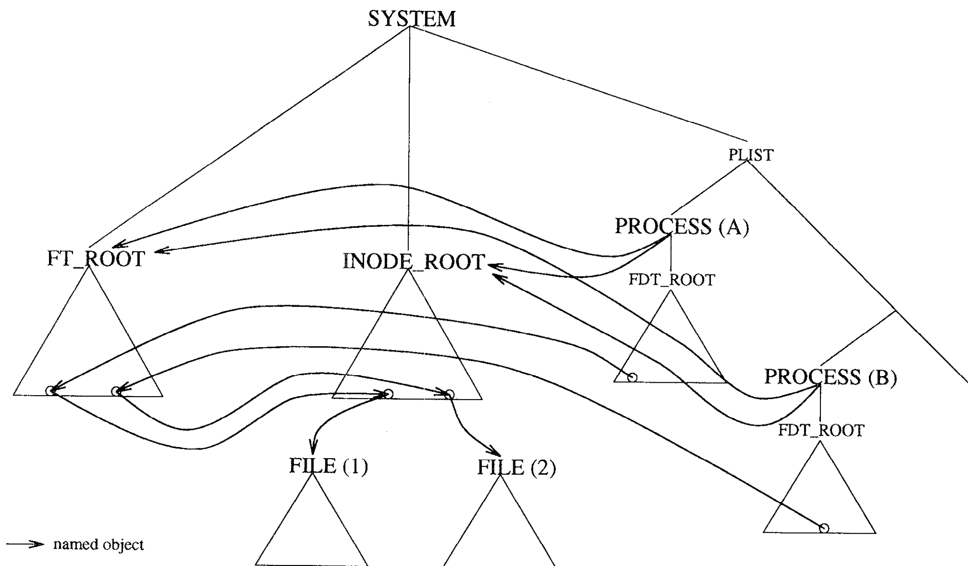


図4 各クラス間の関係
Fig. 4 Relation between classes.

プロセスのためにカーネル内に割り当てられたファイルディスクリプタテーブルの要素を指定するものである．ファイルディスクリプタテーブルの要素は，ファイルテーブルの要素への参照である．ファイルテーブルの要素は，そのファイルの参照数，ファイルのバイト変移やオープン許可モードなどのデータに加えて，iノードテーブルの要素への参照を持つ（図3）．

これらのデータ構造の，OOAGでの表現方法を決める．データ構造は構成規則として記述する．ファイルテーブル，ファイルディスクリプタテーブル，iノードテーブルは，いずれも最大要素数が決められている．これらのデータ構造は，2分木で表現する．プロセスはファイルディスクリプタテーブルを持つが，プロセス数は制限がない．したがって，プロセスは2分木ではなくリスト構造として扱う．

```
class SYSTEM(|)
  -> System1[FT_ROOT(|),INODE_ROOT(|),PLIST(|)]
class PLIST(|)
  -> PList1[PROCESS(|pid),PLIST(|)]
  -> PListNil[]
class PROCESS(|pid)
  -> Proc[FDT_ROOT(pid|),pid]
class FDT_ROOT(pid|)
  -> FDT_Root1[FDT_NODE(pid,num|nout),*INODE_ROOT(|),
               *FT_ROOT(|),cur,free,flag,mode,syscall]
class FDT_NODE(pid,num|nout)
  -> FDT_Node1[FDT_NODE(pid,num|nout),
               FDT_NODE(pid,num|nout)]
  -> FDT_Node2[FDT_LEAF(pid,num|nout),
               FDT_LEAF(pid,num|nout)]
class FDT_LEAF(pid,num|nout)
  -> FDT_Leaf1[*FT_LEAF(|nout)]
class FT_ROOT(|)
  -> FT_Root1[FT_NODE(num|nout)]
class FT_NODE(num|nout)
  -> FT_Node1[FT_NODE(num|nout),FT_NODE(num|nout)]
  -> FT_Node2[FT_LEAF(num|nout),FT_LEAF(num|nout)]
class FT_LEAF(num|nout)
  -> FT_Leaf1[refer,flag,offset,
              *INODE_LEAF(|nout,mode,ilocked)]
  -> FT_LeafNil[]
class INODE_ROOT(|)
  -> Inode_Root1[INODE_NODE(num|nout),queue,in1,in2]
class INODE_NODE(num|nout)
  -> Inode_Node1[INODE_NODE(num|nout),
                 INODE_NODE(num|nout)]
  -> Inode_Node2[INODE_LEAF(num|nout),
                 INODE_LEAF(num|nout)]
class INODE_LEAF(num|nout)
  -> Inode_Leaf1[link,size,ilocked,*FILE(|)]
  -> Inode_LeafNil[]
class FILE(|)
  -> FNormal[*L_file_normal(|)]
  -> FDir[DIR_ROOT(|)]
  -> FNil[]
class DIR_ROOT(|)
  -> Dir_Root1[DIR_NODE(|)]
class DIR_NODE(|)
  -> Dir_Node1[DIR_NODE(|),DIR_LEAF(|filename,inodenum)]
  -> Dir_Node2[DIR_LEAF(|filename,inodenum)]
class DIR_LEAF(|filename,inodenum)
  -> Dir_Leaf1[filename,inodenum]
external class file_normal[];
```

図5　ファイルシステムの構成規則
Fig. 5　Production rules for file system.

テーブルの要素が他のテーブルの要素を指している場合，OOAGではオブジェクトの名前参照を用いて表現する．オブジェクトの名前参照では，指す先に向かってメッセージを送出することができる．また，合成属性の値を知ることもできるため，C言語でのポインタとほぼ同様なセンスで用いることができる．

各クラスの関係を図にすると，**図4**となる．この例では，AとBという2つのプロセスが存在し，それぞれ通常ファイル，ディレクトリファイルをオープンしている状態を示す．

これらのクラスを記述したものが，**図5**である．PLISTがプロセステーブルを，FDT_ROOTがファイルディスクリプタテーブルを，FT_ROOTがファイルテーブルを，INODE_ROOTがiノードテーブルをそれぞれ示している．

3.3　手続きの記述

システムコールやアルゴリズムの手続きの呼び出しは，OOAGでは計算規則として実装する．これは，なんらかの条件が整った時点でこの処理が行われるためである．

利用者は，プロセスからシステムコールに対応するメッセージを出力して処理を行う．メッセージは属性とともに木構造を伝播し，処理が必要なデータ構造に到達する．ここで処理を行った後，属性が木構造を逆に伝播して結果を返す．

ifreeアルゴリズムの動作は次のようになる．

（1）ifreeメッセージがINODE_ROOTに到着する
（2）メッセージをINODE_NODEに渡す
（3）INODE_NODEにメッセージが到着したとき
　Inode_Node1（子供もノード）の場合
　（a）iノード番号に従いINODE_NODE$2かINODE_NODE$3にメッセージを渡す
　（b）（3）に戻る
　Inode_Node2（子供はリーフ）の場合
　（a）iノード番号が子供のものと一致していればメッセージを子供に渡す
　（b）一致していなければエラー
（4）INODE_LEAFにメッセージが到着した場合
　（a）他のプロセスによりロックされていなければ開放する
　（b）ロックされていればエラーを返す

この処理を記述したものが，**図6**である．このプログラムの中に動的意味規則を指定していないものがいくつかあるが，これは親から受けとった属性をそのまま

```
class INODE_ROOT(|)
  -> Inode_Root1[INODE_NODE(num|nout),queue,in1,in2]
    INODE_ROOT:ifree(pid,inum|out)
      => INODE_NODE:ifree(pid,inum|out)
      {
        INODE_NODE.inum = INODE_ROOT.inum;
        INODE_NODE.pid = INODE_ROOT.pid;
        INODE_ROOT.out = INODE_NODE.out;
      }

class INODE_NODE(num|nout)
  -> Inode_Node1[INODE_NODE(num|nout),
                 INODE_NODE(num|nout)]
    INODE_NODE$1:ifree(pid,inum|out)
      case (less (INODE_NODE$1.inum,INODE_NODE$3.num))
        => INODE_NODE$2:ifree(pid,inum|out)
      otherwise
        => INODE_NODE$3:ifree(pid,inum|out)
  -> Inode_Node2[INODE_LEAF(num|nout),
                 INODE_LEAF(num|nout)]
    INODE_NODE:ifree(pid,inum|out)
      case (equal (INODE_NODE.inum,INODE_LEAF$1.num))
        => INODE_LEAF$1:ifree(pid|out)
      case (equal (INODE_NODE.inum,INODE_LEAF$2.num))
        => INODE_LEAF$2:ifree(pid|out)
      otherwise
        =>  /* error! */
        {
          INODE_NODE.out = 'error;
        }

class INODE_LEAF(num|nout)
  -> Inode_Leaf1[link,size,ilocked,*FILE(|)]
    INODE_LEAF:ifree(pid|out)
      case (or(null(ilocked) ,
               equal(ilocked,INODE_LEAF.pid)))
        =>
        {
          (new FILE) = FNil[];
          (new link) = 0;
          (new ilocked) = 'nil;
          INODE_LEAF.out = 'nil;
        }
      otherwise
        => /* inode is locked. */
          {
            INODE_LEAF.out = 'notallowed;
          }
```

図6 ifree アルゴリズムの記述
Fig. 6 Description of ifree algorithm.

子供に渡したり，子供から返された属性をそのまま親に渡すだけの意味規則が記述されている．スペースの関係でここでは省略した．

他のシステムコールやアルゴリズムに関しても同様に計算規則として実現した．この際，計算の過程で木構造に変更を加えながら処理を行う場合には，ifree アルゴリズムのように1回のメッセージの流れでは最終的な答を返すことができない．この場合，中間的な値を木構造の中に保持しておき，この値を用いて入力メッセージが空であるメッセージのガードを記述して次のループで処理を続けることにより記述した．

表1 記述量の比較
Table 1 Comparison between C and OOAG.

	lines	bytes
C	約10,000	約270,000
OOAG	1,653	44,261

4. 考　察

4.1 評　価

3章で記述したファイルシステムが，処理系であるMAGEシステムで完全に動作することを確認した(図7)．これにより，実行可能仕様言語としてのOOAGの能力が確認できた．またこの時の記述は**表1**にあるように1,700行弱である．実現した機能の違いもあり，これをC言語で記述されたシステムと比較することはあまり適当ではないが，C言語での記述では約10,000行である．

OOAGを用いたことにより，仕様は高い可読性，抽象性を維持したまま記述することが可能であった．まず，オブジェクト間の複雑な関係や処理，アクセス方法を，それぞれのクラスに分けて記述することにより単純化することができる．オブジェクト間の関係は完全に階層化されているので，あるオブジェクトの動作について設計者が注目するのは，そのオブジェクトと1つの親オブジェクト，数個の子オブジェクトだけに絞られる．記述の局所性により，構造の変更への対応は容易である．

4.2 OOAGの改良点

実際にファイルシステムを記述した際，さまざまな問題点が明らかになった．ここでは主な問題とその解決法について述べる．

4.2.1 複数の評価ループにわたるメッセージ

あるクラスにおいて，メッセージの処理が1回の評価ループで終わらない場合，その状態を固有属性に保持しておき，次の評価ループ以降で入力メッセージが空のメッセージとして処理を行う．

この場合，返したい値を計算した時点ではすでに最初に送られたメッセージは終了しているため，そのメッセージの合成属性として値を返すことはできない．そのため，値を返すために別なメッセージを記述する必要がある．

しかし，名前参照されているオブジェクトでは，親に対してメッセージを送ると本当の親に対して送られてしまい，名前参照をしているオブジェクトには送られない．そのため，名前参照されているクラスでは，複数の評価ループを用いる処理が記述できないことに

図7 実行時画面
Fig. 7 Display.

なる．

4.2.2 排他制御

現在の OOAG は，利用者が複数存在する状況は考慮されていないため，排他制御に関する機構が準備されていない．複数の利用者が関係するシステムを記述する場合，排他制御の機構は必須となる．

排他制御は，1回のループの中では暗黙のうちに行われている．複数の同じ名前のメッセージが同時に到着した場合，到着したメッセージのうち，いずれか1つが選択され実行される．複数のメッセージのうち1つだけが実行されるのは，暗黙のうちの排他制御と見ることができる．問題となるのは，複数の評価ループにまたがる処理を行う場合の排他制御機構である．

4.2.3 改良方法

これらの問題点を解決するためには，複数の評価ループを必要とするような計算を仮想的に1回のループで終了したように見せかける機構を準備する方法が考えられる．つまり，親にとっては1回の評価ループの

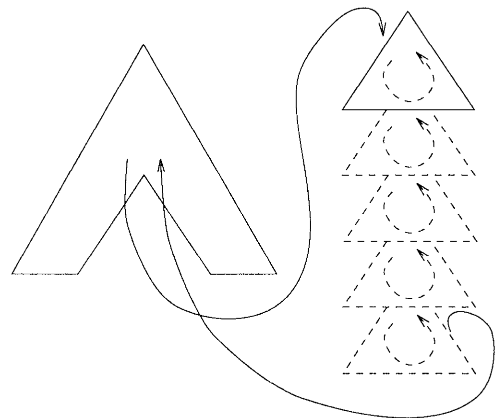


図8 複数の評価ループ
Fig. 8 Multiple evaluation loops.

間に，子供では複数の評価ループを回るような機構を準備すれば良い（図8）.

名前参照を行っている場合，子供がすぐに答を返せない時には，この機構を用いて子供だけが複数のループを回るように記述する．この機能を用いれば，処理を開始するメッセージと処理の結果を返すメッセージを別なものにする必要はなくなるため，名前参照を介したメッセージの送出の場合にも問題はなくなる．

排他制御機構を考える場合，1つのフェーズでは特に問題とならないことは既に述べた.問題となるのは，ループを複数回る間，排他制御を行わなければならない場合である．この場合にも，排他制御を行いたいオブジェクトに対して送出するメッセージを，この機能を用いるメッセージとすることで対処できる．

メッセージを受けとったオブジェクトは，他からのメッセージを受け付けずにループを繰り返す.つまり，その間は誰も処理を行うことができない，ロックされた状態となる．送られたメッセージに関する処理が終った時，別なメッセージを受け付けることができ，ロックが解除された状態となる．

4.2.4 MAGEシステムの問題点

また，以下に挙げるような問題も分かった．これらは計算モデルに対するものではなく,MAGEシステムに関するものである．

ツール独立性の確保 すべてのツールをおなじlispインタプリタ上で動作させているため，1つのツールが動作しているときには他の物は止まってしまう．

外部記憶へのインタフェース 外部記憶に情報を保持することができない．このため，実行を終了するとすべてのデータは失われてしまう．

OSLライブラリ MAGEシステムには，OSL言語のためのクラスライブラリが存在しない．

実行性能 現在のMAGEシステムはプロトタイプシステムとして作成したものであり,実行速度は遅く，メモリ使用効率も悪い．そのため，今回記述したような少し大きな記述になると，メモリの制限から実行できなくなったり，1回のループを処理するために1分程度かかることもある．

現在，インタプリタベースではなく，コンパイラベースのMAGEシステムを作成している．様々な最適化を施した結果，実行速度として100倍程度，メモリ効率として10倍程度の向上が得られている.この新しいバージョンでは，新たにネットワーク上のプロトコルを定義することにより，全てのツールを別プロセスとしている．また，外部記憶へのインタフェースや，クラスライブラリの作成も行われている．このシステムにより，これらの問題点は解決できる.

5. おわりに

本論文では，構造指向型システムのための仕様記述言語として計算モデルOOAGを紹介した.OOAGは，属性文法に対してシステムの動的な側面も記述できるようにオブジェクト指向の拡張を行ったモデルである．このOOAGを用いて仕様を記述･実行するためのMAGEシステムについて触れ,その上で実際にファイルシステムを記述した実験についても述べた．この実験では，仕様記述言語としてのOOAGの基本的な能力を確認することができた．しかし，同時にいくつかの問題点も明らかになったが，これらの問題点に対するいくつかの解決方法についても考察した．

計算モデルOOAGは，仕様を直接実行できる実行可能仕様記述言語である．しかし，もう1つの側面である正当性検証を行うための検証系を準備してゆく必要もあるだろう.OOAGの基本となる属性文法に関しては性質も明らかでありシステムを容易に構築できると考える.OOAGで新たに加えた動的な仕様に関する部分は，現在その性質を明確に定義している段階である．この作業が終れば，検証系の作成も行えるであろう．

参 考 文 献

1) Shinoda, Y. and Katayama, T.: Object Oriented Extension of Attribute Grammars and Its Implementation Using Distributed Attribute Evaluation Algorithm, *Proc. Inter. Workshop on Attribute Grammars and Their Applications,* Lecture Notes in Computer Science 461, pp. 177-191, Springer-Verlag (1990).

2) 権藤克彦,片山卓也：オブジェクト指向属性文法計算モデルOOAG, 第10回日本ソフトウェア科学会大会論文集, A 3-1, pp. 25-28 (1993).

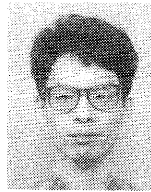
3) Gondow, K., Imaizumi, T., Shinoda, Y. and Katayama, T.: Change Management and Consistency Maintenance in Software Development Environments Using Object Oriented Attribute Grammars, Nishio, S. and Yonezawa, A. eds., *Object Technologies for Advanced Software* (*Proc. JSSST 1st Inter. Symposium*), Lecture Notes in Computer Science 742, pp. 77-94, Springer-Verlag (1993).

4) Knuth, D. E.: Semantics of Context-Free Languages, *Mathematical Systems Theory*, Vol. 2, No. 2, pp. 127-145 (1968).

5) Knuth, D. E.: Semantics of Context-Free Languages: Correction, *Mathematical Systems The-*

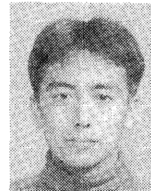
ory, Vol. 5, No. 1, pp. 95–96 (1971).
6) Deransart, P., Jourdan, M. and Lorho, B.: *Attribute Grammars: Definitions, Systems, and Bibliography*, Lecture Notes in Computer Science, Vol. 323, Springer-Verlag (1988).
7) GrammaTech, Inc., One Hopkins Place, Ithaca, New York, USA: The Synthesizer Generator Reference Manual, Release 4.1 (1993).
8) Reps, T. W. and Teitelbaum, T.: *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag (1989).
9) 片山卓也, 篠田陽一, 菊池 豊, 鈴木正人, 今泉貴史, 石原博史, 高田 勝：自分自身のためのプログラム言語の作り方, Computer Today 別冊, サイエンス社 (1988).
10) Vogt, H. H., Swierstra, S. D. and Kuiper, M. F.: Higher Order Attribute Grammars, *Proc. SIGPLAN'89 Conference on Programming Language Design and Implementation*, pp. 131–145, ACM, Portland, Oregon (1989).
11) Teitelbaum, T. and Chapman, R.: Higher-Order Attribute Grammars and Editing Environments, *Proc. ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pp. 197–208, ACM, White Plains, New York (1990).
12) Pittman, T. and Peters, J.: *The Art of Compiler Design—Theory and Practice*, Prentice Hall, Inc. (1992).
13) Betz, D. M.: *XLISP: An Object-oriented Lisp*, Version 2.1 (WINTERP) (1989).
14) Niels, P.: *The WINTERP MANUAL (Version 1.0)*, Hewlett-Packard Laboratories, Human-Computer Interaction Department (1989).
15) 松塚貴英, 今泉貴史：オブジェクト指向属性文法によるファイルシステムの形式的記述, jus UNIX シンポジウム論文集, 日本UNIXユーザ会 (1994).
16) Bach, M. J.: UNIXカーネルの設計, 共立出版 (1990), 坂本 文, 多田好克, 村井 純 訳.
17) Leffler, S. J., McKusick, M. K., Karels, M. J. and Quarterman, J. S.: UNIX 4.3 BSDの設計と実装, 丸善 (1991), 中村 明, 相田 仁, 計 宇生, 小池汎平 訳.

(平成6年 8 月5日受付)
(平成6年12月5日採録)



今泉 貴史（正会員）

1965年生. 1987年東京工業大学工学部情報工学科卒業. 1992年同大学大学院博士課程修了. 博士（工学）. 現在, 同大学工学部電気・電子工学科講師. 属性文法に基づく言語のプログラミング環境の開発に携わる. 日本ソフトウェア科学会, ACM各会員.

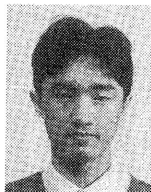


権藤 克彦

1966年生. 1989年東京工業大学工学部電気電子工学科業. 1994年同大学大学院博士課程修了. 博士(工学). 現在, 同大学大学院情報理工学研究科計算工学専攻助手. 属性文法型言語の基礎モデル・評価法・応用の研究に従事. 日本ソフトウェア科学会, ACM各会員.

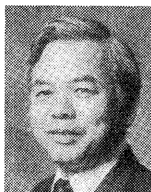
萩原 威志

1969年生. 1991年東京工業大学工学部情報工学科卒業. 1993年同大学大学院修士課程修了. 現在, 同大学院理工学研究科博士後期課程在学中. 属性文法型言語の処理系, およびそのソフトウェア工学への応用に興味を持つ. ソフトウェア科学会会員.



松塚 貴英

1971年生. 1994年東京工業大学工学部電気電子工学科卒業. 現在, 同大学大学院情報理工学研究科修士課程在学中. 属性文法に基づく言語と並列・分散処理に興味を持つ.



片山 卓也（正会員）

1939年生. 1962年東京工業大学工学部電気工学科卒業. 1964年同大学大学院修士課程修了. 工学博士. 日本アイ・ビー・エム(株), 東京工業大学工学部助手, 同助教授, 同教授を経て, 1991年より北陸先端科学技術大学院大学教授. この間, オートマトン理論, プログラミング言語, 属性文法, 関数型プログラミング, ソフトウェア工学などに関する研究を行う. 日本ソフトウェア科学会, ACM, IEEE各会員.