オブジェクト指向属性文法 OOAG による ソフトウェアリポジトリシステムの自動生成

萩原威志[†] 片山卓也[††]

本論文では，ソフトウェアリポジトリを構築するためのオブジェクト指向データベースの自動生成について述べる．ソフトウェアリポジトリにおいては，要求仕様，設計に関する様々なドキュメント，ソースコードなどのソフトウェアプロダクトが，互いに密接に関連しあっていて，それらに変更が加えられた場合にはそれらの関連に従って一貫した状態で更新やメンテナンスを行う必要がある．このようなリポジトリシステムを記述するための体系として，通常の属性文法に対して，変更を管理し，一貫性を保つための機能を組み込んだ計算モデル OOAG が文献 1)，2) において導入されている．OOAG では，ソフトウェアオブジェクトの構造を文脈自由文法形式で表現し，この構造上に属性とその関係を記述することで仕様を記述する．我々は，OOAG で記述されたリポジトリ仕様を C++の一連のクラス定義に変換して実行するシステム MAGE を設計・実装した．MAGE の実装では，OOAG の属性評価器の実行速度の向上のためにいくつかの特別なテクニックを導入している．MAGE は，OOAG のオブジェクトをオブジェクト指向データベースに格納する．MAGE で生成された C++のクラス定義を OOAG の属性評価器のライブラリとともにコンパイルすることで，ソフトウェアリポジトリシステムを得ることができる．

Generating Software Repository System by Object-oriented Attribute Grammars

TAKESHI HAGIWARA[†] and TAKUYA KATAYAMA[††]

This paper describes automatic generation of object-oriented database for the use of software repository. In software repositories, software products such as requirement specifications, design documents and source code are closely related to one another and need to be maintained and updated consistently according these relations when changes are made to them. To describe such repository system, a computational model called OOAG (Object-Oriented Attribute Grammars) has been introduced[1),2)], which incorporates functions for managing changes and maintaining consistency into traditional attribute grammars. We describe the specification of repository by representing the structure of software objects with the form similar to context free grammars and describing the semantics of this structure with attributes and relationships among them. We design and implement the system MAGE that translates repository specification written in OOAG to a set of C++ class definitions. In this implementation, we employ some techniques to improve execution speed of OOAG attribute evaluator. MAGE stores OOAG objects in object-oriented database. We can get a software repository system by compiling a generated set of C++ class definitions with the attribute evaluator library of OOAG.

1. はじめに

ソフトウェアリポジトリは，ソフトウェア開発環境における各種情報（ソフトウェアオブジェクト）の管理を行う一種のデータベースで，これをうまく作ることが開発環境構築の成功には重要な要素である．しかし，複雑な構造のソフトウェアオブジェクトの高度な処理を行うソフトウェアリポジトリは複雑なシステムとなるので，抽象的な記法からの機械的な構築システムが必要となる．我々はこれをソフトウェアリポジトリの自動生成系と呼ぶ．本論文では，これを目標とするシステムである MAGE システムと，特にソフトウェアオブジェクトの処理を実行する属性評価器の設計と実現について述べる．

† 東京工業大学情報工学科
Department of Computer Science, Tokyo Institute of Technology
†† 北陸先端科学技術大学院大学
School of Information Science, Japan Advanced Institute of Science and Technology

ソフトウェアオブジェクトとそれらの派生オブジェクトの間には互いに複雑な関係がある．たとえば，階層的に記述された設計図面とその階層に収まらない場合の関連リンク，ソースコードとその部分の仕様書・各種ドキュメント，実行可能なバイナリとそれを構成するためのソースコード群と作成手順，などがあげられる．ソフトウェアリポジトリが有効に機能するためには，このようなソフトウェアオブジェクト間の関係を高度に扱わなければならない．たとえば，変更が加えられたときの変更の影響範囲の特定とその内容の検索，変更の自動伝播，関係の衝突（一貫性の破壊）の検出などがその例である．しかし，リポジトリが扱う関係の高度な処理の記述は非常に難しい．たとえば，オブジェクト指向データベースは，一般に記述の柔軟性は十分に高いが，関係の高度な処理の内容は各オブジェクト中のメソッドプログラム中に埋もれてしまう．このため，仕様を満たすように正確に関係処理の記述を行うのは難しい．また，統一的なソフトウェア開発環境を構築するための国際規格のひとつに PCTE[3] がある．PCTE では，ソフトウェアオブジェクトを ER モデルで型付けを行うことによりソフトウェアオブジェクトの扱いを統一し，さらに開発環境構築のために，ソフトウェアオブジェクトの処理を行うツールとのインタフェースを規定している．しかし，各ツールの記述には，低レベルなプログラミング言語を用いなければならず，やはり正確に関係処理の記述を行うのは難しい．

現在，高度な関係処理を行う実用的なソフトウェアリポジトリは存在しない．既存のシステムは，ソースコードとそのドキュメントをファイルという粒度で管理しているにすぎない．その理由は，関係の記述の困難さ，高度な関係処理の技術そのものの未熟さ，にあると我々は考える．この問題に対応するためには，高度な関係処理を簡単に記述し，この記述からリポジトリを機械的に生成できるシステムが必要である．

我々は，複雑な一貫性の検査，派生変数の計算，データオブジェクトの変更の伝播などの能力を持つオブジェクト指向データベースの自動生成を考える．この目的のために，我々は計算モデル OOAG[1),2),4),5)] を採用する．OOAG は属性文法[6)] に基づく計算モデルで，形式的なオブジェクトの構造と関係に基づく意味を同時に記述できる．OOAG は，変更伝播に基づき派生オブジェクトを自動的に再計算し，オブジェクト間の関係を検査するための宣言的記法を与えている．

CACTIS[7)]，GRAS[8)] などのシステムは，派生値の自動計算のための機構として，派生値のインクリメンタルな計算機構を導入したデータベース管理システムである．派生値のインクリメンタルな計算のために，属性文法に基づく構造エディタ生成系である CSG[9)] などで使われているインクリメンタル属性評価アルゴリズムを改良して用いている．この意味では，派生値のインクリメンタル計算のために属性文法のインクリメンタル属性評価アルゴリズムを使用する OOAG と似た概念を導入しているといえる．しかし，これらは抽象的な記述システムを持たず，C 言語などへのインタフェースが存在するだけで，リポジトリ記述の問題の複雑さを解決していないと考える．言い換えると，トリガ機構などの組合せで実現できる範囲内のことを，少し便利に行えるようになっただけである．これらのシステムと比較して，OOAG は属性文法の持つ階層的かつ抽象的な記述スタイルも含めた属性文法拡張なので，OOAG の記述は高い読解性を持ちメンテナンスしやすいという特徴を受け継いでいる．

この論文では，OOAG で記述されたリポジトリ仕様からソフトウェアリポジトリシステムを生成する方法を述べる．我々は，リポジトリシステムを自動生成するためのシステムである MAGE を設計・実現した．MAGE システムは OOAG で記述したリポジトリ仕様を C++のクラス定義群に変換する．MAGE の実現では，OOAG の属性評価器の実行速度を向上させるために，いくつかのテクニックを導入している．これらのテクニックは生成されたリポジトリシステムのソフトウェアオブジェクト管理の高速化に大きく貢献している．MAGE は，OOAG のオブジェクト定義を，C++のクラス定義群に変換する．実際のデータベース操作は，オブジェクト指向データベース管理システムの機能を利用することで実現している．この C++のクラス定義群を OOAG の属性評価器ライブラリとともにコンパイルすることで，クライアント・サーバモデルのソフトウェアリポジトリシステムが生成できる．MAGE システムの実現は，OOAG によるソフトウェアリポジトリの自動生成というゴールへの大きな前進である．

なお本論文の構成は，以下のようになっている．2 章では，我々が考えるソフトウェアリポジトリシステムについて議論する．3 章では，計算モデル OOAG とその記述言語である OSL 言語について簡単に説明する．4 章では，生成系を含む環境 MAGE の実装に関して述べる．5 章はまとめである．

2. ソフトウェアリポジトリ

我々は，ソフトウェアリポジトリを，それが提供する

機能やソフトウェアオブジェクトを扱う抽象度によって，いくつかのタイプに分類して考えている．我々が，MAGE システムで生成しようと考えているリポジトリシステムの性格を明らかにするために，ここでは以下のような分類を行う．

（タイプ1） ソフトウェアオブジェクトの単純な集合体．たとえば通常のファイルシステム上に構築されたディレクトリ構造によるものなど．このタイプのものは，ファイル単位のソフトウェアオブジェクトだけを扱う．

（タイプ2） データベースシステムを利用し，ソフトウェア開発において生成されるソフトウェアオブジェクトとそれらの関係を管理するもの．ソフトウェアオブジェクトの単位は自由になり，型付きで扱える．

（タイプ3） 2の特徴に加え，個別のソフトウェア開発に合わせてカスタマイズしたり，動作を変更したりすることが可能なもの．

タイプ1の特定の目的に特化しない汎用的なものほど，大量のソフトウェアオブジェクトをまとめて扱う大規模なデータベースを構築できる．リポジトリのシステム自体として特別の機能は持たず，リポジトリの構造や格納できるオブジェクトの型なども何も規定や制限できない．タイプ2は少し高度になって，データベースシステムを利用し，リポジトリ構造，インデックス情報，ソフトウェアオブジェクト間の関係などを規定できるものを想定している．たとえば，PCTE のリポジトリである OMS では，ER モデルに基づくスキーマで記述した型付きのソフトウェアオブジェクトを扱う．細かなソフトウェアオブジェクトの型定義が可能だが，これを基にソフトウェアオブジェクトを操作するツールを構築するのは単純な作業ではない．このタイプには以下の難点がある．

- オブジェクトの操作のためにデータベースにアクセスする専用の各種ツール（make などに対応するもの）が必要．
- リポジトリシステム自体にオブジェクト操作のための機能を組み込むのは簡単ではない．
- 汎用目的に開発されたものは，細かなカスタマイズ，オブジェクト処理の制御が難しい．

我々が構築しようとしているのは，タイプ3のソフトウェアリポジトリである．ここで狙っているのは，特定の目的や環境に特化した比較的小規模ではあるが，ソフトウェアオブジェクトの間の一貫性のチェックや変更の伝播などの，複雑で先進的な管理を考慮したものである．このような用途を狙ったものとしては，GRAS などのデータベース管理システムにインクリメンタル計算機能を付加する試みがあり，派生値の自動計算などのプログラミングの負荷を軽減しているものもあるが，仕様作成からプログラミングまでの難しさを基本的に解消してはいない．我々は，タイプ2のリポジトリの難点としてあげたもののうち，1項目めは解決できないが，2項目め，3項目めを簡単にできるようにすることで，外部ツールを利用したオブジェクト管理ではなく，リポジトリシステム自体の機能として実現できるようにすることを狙っている．

2.1 ソフトウェアリポジトリに対する要求

ソフトウェアリポジトリシステムには，ソフトウェアオブジェクトの管理作業の自動化や管理作業の効率化など大量の要求がある．たとえば，バージョン管理，構成管理，派生データの自動計算などが典型的である．現在のほとんどのソフトウェア開発環境では，ファイルシステム上にリポジトリが構築され，各種のツールを用いることにより，ファイルのバージョンを管理したり[10),11)]，ソフトウェアシステムの構成を管理する[12)]という要求を部分的に満たしている．しかし，ファイルシステム上に構築されたこれらのリポジトリは，どのようにファイルを管理するのかは利用者任せであり，運用方法を間違えることでリポジトリの構造を崩してしまったり，その影響によりオブジェクト管理スクリプトの動作を妨げることになる．

これを防ぐには，リポジトリ構造をシステム側で制限できることが必要で，かつ運用方法まで構造定義と同時に指定できる必要がある．以下の機能をソフトウェアリポジトリシステムに組み込むことにより，ソフトウェア開発の生産性が向上する．

- ソフトウェアオブジェクトの変更にともなって自動的に管理のための動作を開始するための機能（トリガ機構）
- 何らかのイベントに対して作業を開始するのではなく，自動的にソフトウェアオブジェクト間の関係の一貫性を保つ機能☆
- 派生値の自動計算などの作業のインクリメンタル処理
- 複雑な構造を持つソフトウェアオブジェクトの表現，およびその構造の変更

2.2 ソフトウェアリポジトリの構築と運用のための計算モデル

我々は，前述の機能を実現するには，リポジトリ構

☆ トリガで管理タスクを開始することとの違いは，トリガを仕掛ける箇所に対する配慮がいらない点である．

築の土台であるリポジトリシステム自体にソフトウェアオブジェクトを管理するための計算メカニズムを導入するとうまくいくと考える．ここでは，

- ソフトウェアオブジェクトの操作方法をプログラム可能にすること
- ソフトウェアオブジェクトの構造とその操作（管理）方法を同時に記述できるようにすること

の2点の実現に重点をおく．このような機能を従来のファイルシステム上に構築されたリポジトリで実現するには，make や rcs，cvs などの外部ツールを用いることにより，構成管理やバージョン管理の機械化を行うことができるが，これらはファイルベースの管理しか行うことができず，ファイルへの分割方法もあらかじめ規定した形式に制限することはできず，リポジトリ上の作業者に任せることしかできなかった．

また従来は，リポジトリシステムの設計・実現は，仕様作成とは別に行わなければならなかったが，

- リポジトリの仕様から効率の良いリポジトリシステムを機械的に生成することを可能にすること

を実現することで，生産性は飛躍的に向上する．我々は，リポジトリの形式化のために属性文法型の計算モデルを採用し，リポジトリ記述に応用する．これにより，上記の要求を満たすことができると考える．

属性文法は，プログラミング言語の形式的仕様記述，およびコンパイラの自動生成のツールとして研究されてきた．属性文法の計算モデルの本質は，木構造のノードに結び付けられた属性値の関数的な計算である．その記述は，属性の計算のための意味規則を持つ，木を構成する文法規則の集合である．属性文法のこの特徴は，オブジェクトの永続化と動的な計算の機構が付加できるならば，リポジトリシステムの形式化を行うための優れた候補になりうる．OOAG は，通常の属性文法を拡張して，リポジトリシステムの仕様記述のための要求を満足することを狙っている．

リポジトリシステムの記述・生成系としての OOAG の特徴を以下に示す．

- オブジェクトの構造を has-a 関係のスキーマで記述する．各部品間の関係はスキーマごとに属性計算として記述する．離れた位置に存在するオブジェクトを参照するためのモデルは研究段階である．
- 属性計算の規則は関数的で，属性が付加される箇所の規則内の局所的なものなので，記述内容が理解しやすい．そして，属性計算は制約充足系の一種であるが，prolog のバックトラックによるものでも，局所伝播法などとも異なり，充足する方向の決まった関数の再計算だけに制限されているため，効率が良い．
- 必要なオブジェクトを得るための操作の適用順序は，オブジェクトの依存関係をもとに属性評価器により機械的に計算されるので，これをプログラミングする必要がない．
- オブジェクト操作を行うリポジトリエンジンは，属性評価器として文法記述から機械的に構成することができる．

3. オブジェクト指向属性文法 OOAG

OOAG は，プログラミング言語の形式仕様記述やコンパイラの自動生成のツールとして研究されてきた属性文法（以下 AG と略す）をもとに開発された．宣言的構造，意味定義と構文定義の分離，理解しやすく高い保守性の要因となる局所的な記述，そして属性の関数的な計算に基づく明確な記述は，すべて属性文法のすぐれた特徴である．

3.1 OSL 言語の簡単な説明

我々は，OOAG の記述言語を OSL（Object Specification Language）と呼んでいる．OSL 記述は「静的仕様記述」と「動的仕様記述」という 2 つの部分に分類できる．以下では OSL 言語の各構成要素を簡単に説明する．詳細については文献13) を参照して欲しい．

3.1.1 静的仕様記述

静的仕様記述では，オブジェクトの静的な関係を記述する．

クラス

OOAG のクラス宣言は，以下の形式を持つ．

$$\textbf{class}\ X_0 \to R[X_1, \ldots, X_m] \quad \{\ldots\}$$

ここで X_0 は定義されるオブジェクト，$X_1, \ldots, X_m$ はそのオブジェクトの内部オブジェクトである．R はそのルールに対するラベルで，記述するオブジェクトのオブジェクト構成子として使用される．X_0 を **LHS** クラス，R を **RHS** クラスと呼ぶ．$\{\ldots\}$ の部分を**静的意味規則**と呼ぶ．

静的属性

静的属性の宣言は，クラス宣言の X_i $(0 \le i \le n)$ に添え付けされ，次の形式を持つ．

$$X_i(i_1, \ldots, i_p | s_1, \ldots, s_q)$$

ここで，$i_1, \ldots, i_p$ は**静的相続属性**，$s_1, \ldots, s_q$ は**静的合成属性**である．

静的相続属性と静的合成属性がオブジェクト X_i に関係付けられるのに対して，**静的局所属性**はクラスに対して関係付けられる．静的局所属性 l の宣言は，静的意味規則において次の形式で行われる．

local l

静的相続属性と静的合成属性は，オブジェクトの外部インタフェースである．固有属性は，オブジェクトの状態を保持する．我々は，**固有属性**という用語を内部オブジェクト X_i に対しても用いている．内部オブジェクトが，オブジェクトの状態を保持するからだ．固有属性は，オブジェクト指向パラダイムにおけるインスタンス変数に相当する．

静的意味規則

静的意味規則は，オブジェクトの静的合成属性の値，内部オブジェクトの静的相続属性の値，そして必要ならばクラスに関連付けられた静的局所属性の値を定義する．各静的意味規則の形式は次のとおり．

$a = f(a_1, \ldots, a_r)$

ここで，a は記述中の他の属性 $a_1, \ldots, a_r$ で定義される属性，f は $a_1, \ldots, a_r$ 上の関数である．$X.i$ という記法は，オブジェクト X の静的属性 i を表し，n という記法は，固有属性 n または静的局所属性 n を表す．生成される任意の木に対して静的属性の依存グラフは計算可能になるように非循環でなければならない．

3.1.2 動的仕様記述

メッセージパッシングを記述する動的仕様記述の集合は，オブジェクトの動的な振舞いを定義する．動的仕様記述は，以下のものから構成される．

メッセージ

メッセージパッシングの記述は次の形式を持つ．

$in_1, \ldots, in_s \Rightarrow out_1, \ldots, out_t \; \{\ldots\} \quad (s, t \geq 0)$

ここで in_i $(1 \leq i \leq s)$ は **入力メッセージ**で，out_i $(1 \leq j \leq t)$ は **出力メッセージ**である．$\{\ldots\}$ の部分は，**動的意味規則**である．

動的属性

それぞれのメッセージは，いくつかの動的属性を持つことができる．メッセージは次の形式を持つ．

$Obj : mesg_name(i_1, \ldots, i_p | s_1, \ldots, s_q)$

ここで Obj はオブジェクト，$mesg_name$ はメッセージ名，$i_1, \ldots, i_p$ は **動的相続属性**，$s_1, \ldots, s_q$ は **動的合成属性**である．

動的意味規則

動的意味規則は，静的意味規則と同じ $a = f(a_1, \ldots, a_r)$ という形式を持つ．ただし，固有属性の値が定義できることと，$a_1, \ldots, a_r$ に動的属性が現れてもよい点が異なる．固有属性の値を定義する形式は

$(\mathbf{new}\ a) = f(a_1, \ldots, a_r)$

である．**new** は固有属性の新たな次の値を以前の値と区別するための予約語である．固有属性の値は，動的属性の評価が完了した後に新しい値と置き換えられる．この記法により，動的属性の計算の結果として，木構造の変更，拡張が可能になる．

表1 静的仕様記述での用語の対応
Table 1 The correspondence of the term in the static specifications.

OOAG での用語	通常の AG での用語
オブジェクト (NODE オブジェクト)	Production instance
葉固有属性	終端記号
LHS クラス名	非終端記号
RHS クラス名	生成規則のラベル

メッセージパッシング

メッセージパッシング記述は，入力メッセージの集合と出力メッセージの集合の組を記述する．出力メッセージは，以下の条件が満たされるときに目的のオブジェクトに送信される．

- すべての入力メッセージが受信され，かつ出力メッセージのすべての出力属性の値が評価済みである．あるいは，
- 入力メッセージがない場合．

ただし，出力属性の集合には，メッセージパッシングの条件を表すガード式を付加することができる．ガード式は，出力メッセージを出力するかどうかを制御する．つまり，ガード式が偽の場合には，その動的意味規則は評価されず，出力メッセージは出力されない．

3.2 OSL 記述と通常の属性文法の記述の対応

計算モデル OOAG は，属性文法に基づいた計算モデルである．

OOAG のクラスは，通常の属性文法でいう生成規則に相当する．静的仕様記述は，通常の属性文法のモデルと等価である．OOAG の静的仕様記述で使われる用語と，通常の属性文法で使われる用語の対応を **表1** に示す．

動的仕様記述の部分が通常の属性文法の計算モデルに対して拡張された部分である．この拡張により，通常の属性文法のモデルでは不可能な，構成された属性付き木の構造の変更とその属性の再計算を可能にしている．メッセージパッシングの機構は，属性付き木の構造上の情報に作用する関数と見なすことができる．

3.3 OOAG による記述の例

図1 の OSL 記述は，プログラムオブジェクトを表現するクラス P とモジュールオブジェクトを表現するクラス M を記述した簡単な例である．この例から生成されるオブジェクト P は，2つの子オブジェクト M を持ち，オブジェクト M はそれぞれ子オブジェクト M と葉固有属性 cache，current を持ち，**図2** のような

構造を持つ．

P の静的仕様記述（意味規則）部分には，静的合成属性 **executable** に，2 つの子オブジェクト M の静的合成属性 **object_code** から，関数 **link** によりその状態の時点での実行コードが得られるよう記述してある．

M の子オブジェクトとなっている ROBJ は，ここではプログラムのソースコードのバージョン管理サービスを提供するオブジェクトであると仮定している．また，**cache** は ROBJ との入出力のためのキャッシュ，**current** は現在のオブジェクトコードを示す葉固有属性である．

M の動的仕様記述（意味規則）部分で記述されている **:retrieve** メッセージは，必要なオブジェクトコードのリビジョン番号 **revision** と取り出したオブジェクトコードを現在のオブジェクトコードにするかを

```
/* P の静的仕様記述 */
class P(|executable)
  -> Program[M(|obj_code), M(|obj_code)]
 { P.executable=link(M$1.obj_code, M$2.obj_code); }
/* M の静的仕様記述 */
class M(|obj_code)
  -> Module[ROBJ(|), cache, current]
  { M.obj_code = current ; }
/* M の動的仕様記述 */
  M:retrieve(revision, update|req_obj)
   case (in_cache(cache, revision))
    =>
    { M.req_obj = look_up(cache, M.revision) ;
      (new current) = if(M.update == true, M.req_obj, current);
    }                                                      .....(A)
   otherwise
    => ROBJ:retrieve(revision|source)
    { M.req_obj = compile(ROBJ.source);
      (new cache) = lru_update(cache, M.revision, M.req_obj);
      (new current) = if(M.update == true, M.req_obj, current);
    }                                                      .....(B)
```

図 1　OSL 言語による記述の例
Fig. 1　Example of OSL description.

決定するフラグ **update** をパラメータとして受け取り，**req_obj** に取り出したオブジェクトコードを返す．要求されたリビジョンのオブジェクトコードがキャッシュの中に入っていれば，それを **req_obj** に返し，同時に **current** を置換する（A）．キャッシュになければ，ROBJ に **retrieve** メッセージを送り，必要なリビジョンのソースコードを取り出し，それをコンパイルして **req_obj** に返す．この際キャッシュ内容を更新し，**current** を置換する（B）．

図 2 は，それぞれ (a) 定常状態のオブジェクト P，(b) **retrieve** メッセージの評価での **cache** と **update** の更新の様子，(c) 変更の伝播中のオブジェクトの振舞いを示している．

4. MAGE の設計と実装

ここでは，MAGE の概要と OOAG の属性評価器の設計と実装について述べる．また OOAG の属性評価は特殊なので，以下の順に説明を行う．

(1) NODE オブジェクトという言葉と属性評価アルゴリズムの概要（4.2.1，4.2.2 項）
(2) 高速な属性評価器の設計と実装，および OODBMS 利用に関する考察（4.2.3 項）
(3) xlisp プロトタイプと比較したベンチマークによる評価（4.4 節）

4.1 アーキテクチャの概要

我々は，OOAG で記述されたリポジトリ仕様を一連の C++のクラス記述に変換して OOAG のモデルで動作するリポジトリシステムを実行するシステムである MAGE を開発している．MAGE で変換した C++のクラス定義をコンパイルし，OOAG の属性評価器

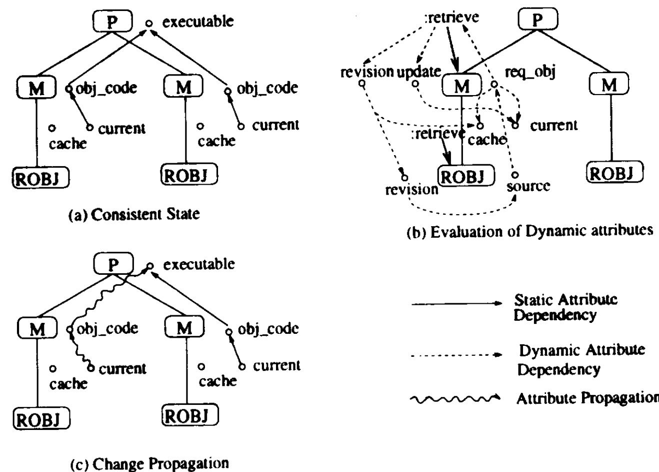


図 2　オブジェクト P の動作
Fig. 2　The behavior of P.

のライブラリとリンクすることで，OOAG のモデルの上で動作するリポジトリシステムを得ることができる．一言でいえば，データベース管理システムを利用してサーバ・クライアント型のリポジトリシステムを生成するためのシステムである．ただし，データベース管理システムを用いて OOAG のオブジェクトを永続的に扱うための部分は現在はまだ開発中である．すなわち，MAGE で生成されたリポジトリシステムは，現在はまだコンピュータのメモリ上でだけ動作している．OOAG のオブジェクトを永続的にするためには，オブジェクト指向データベース管理システムを使用する．現在，オブジェクト指向データベース管理システムのひとつである ObjectStore[14] を用いて MAGE の開発を行った．ObjectStore の持つ Virtual Memory Mapping Architecture (VMMA) アーキテクチャは，洗練されたメモリ写像，キャッシュ，クラスタリングなどの技術を使用して，データアクセスを最適化し，高いパフォーマンスを実現している．そして，これらの使用されている技術は，我々が OOAG の効率的な実行のために行ってきた実装手法に悪影響を及ぼすことはない．これについては，この後の 4.2 節で述べるが，これは VMMA では不揮発的なデータと同等の速度で永続データを扱うことが可能だからだ．さらに，ObjectStore のデータベースインタフェース記述言語を利用すれば，既存のシステムのコードの大半をそのまま移行できるという利点がある．

図 3 は，MAGE でどのようにリポジトリシステムを生成するのかを示している．MAGE の主な構成部品は，

(1) システム制御，生成したシステムの状態参照用の各ブラウザ

(2) OOAG による記述を行うための構造エディタと，記述内容の構造をグラフィカルに表示するクラス記述ブラウザ

(3) OOAG 記述から C++へのコード変換システム

(4) OOAG の属性評価器のライブラリ

(5) オブジェクト指向データベース管理システム (ObjectStore)

で，このうちシステム生成のために重要なのは (3)，(4)，(5) である．MAGE でリポジトリシステムを得るためには，まず専用の構造エディタとクラス記述ブラウザを使って，OOAG でリポジトリの構造とデータ管理のスキーマを記述することから始める．この OOAG で記述された仕様を，コード変換システムで C++のクラスに変換し，そして OOAG の属性評価器のライブラリ，ObjectStore のライブラリとともにコンパイルすると，リポジトリシステムができる．

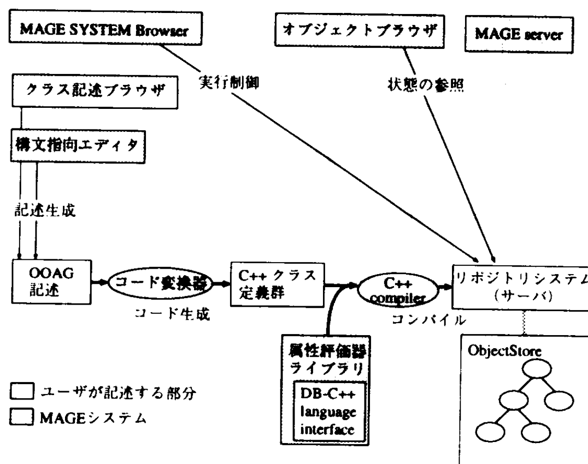


図 3 MAGE でのリポジトリシステムの生成方法
Fig. 3 How to get repository in MAGE.

4.2 属性評価器の実装の詳細

4.2.1 NODE オブジェクト

NODE オブジェクトは，属性付き木のノードをオブジェクトとして扱うための基本的な概念である．OOAG では，すべてのリポジトリのソフトウェアオブジェクトの型は，何らかの OOAG のクラスとして記述され，そして実際のオブジェクトは木構造の形に編成される．これは，OOAG のクラスが属性文法での生成規則に対応しているためである．

NODE オブジェクトは，属性値や具体的なソフトウェアのデータ，属性値を評価するための関数，OOAG のメッセージハンドラなどの，木のノードに対応したすべての情報を管理する．NODE オブジェクトでの属性値の扱いには，特別な注意が必要である．**図 4** の文法規則から生成される木の一部を例にとる．通常の属性評価アルゴリズムでは，一度の属性の出現は評価アルゴリズム中一度だけしか現れない．しかし 4.2.2 項で説明する我々のアルゴリズムでは，属性の評価のためのスケジューリンググラフを NODE オブジェクトごとに分割するため，1 度の属性の出現が上記の文法規則のテキスト表現で 2 度現れているのと同じように，

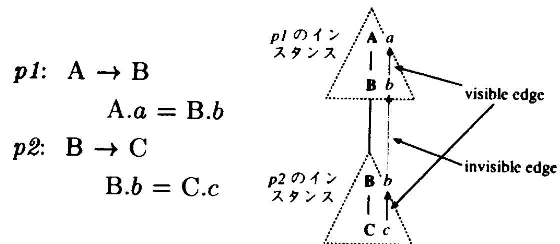


図 4 NODE オブジェクトと属性
Fig. 4 NODE object and its attribute.

それぞれの NODE オブジェクトで1回ずつ，計2回現れる．このとき，有効な値を保持するのは，その値が計算される NODE オブジェクトに格納される属性で（すなわち属性が意味規則の左辺に現れる NODE オブジェクト），他方の値の参照時にはもし有効な方の値に変化があったのならばその値をロードする必要がある．上記の例でいえば，属性 a の有効な値を保持するのは *p1* のインスタンスで，属性 b の有効な値を保持するのは *p2* のインスタンスである．*p1* のインスタンスで A.a を評価する際に B.b が参照されるが，このとき *p2* のインスタンスの B.b の値に変化があれば，*p2* のインスタンスからその値をロードしなければならない．

4.2.2 OOAG の属性評価器

計算モデル OOAG のための属性評価器では，2種類の属性評価を扱う必要がある．静的属性の評価を行う静的属性評価と，メッセージのパスに一時的に割り当てられる動的属性の評価を行う動的属性評価である．

静的属性と固有属性の評価アルゴリズムは，文献1)において *Locally Controlled Distributed Incremental Attribute Evaluation*（LCDIA）アルゴリズムが与えられている．その基本的なアイデアは，属性の依存を表す大域的なスケジューリンググラフ M をそれぞれの適切なノードに**ローカルモデル M_L** として分割して格納・管理することである．この方法では，それぞれのノードの M_L は，ノードとそれに属する枝の属性しか含まない．したがって，親子関係にあるノード間の属性インスタンスの間には，明示的には現れない見えない依存枝が存在する（4.2.1 項の図を参照）．LCDIA アルゴリズムでは，親子関係にある2つのオブジェクトの M_L 中の属性インスタンスを関連付け，M での評価すべき属性インスタンスへの依存する入力枝の数が0であることを判定するために，2つのシグナル PROPAGATE, RELAX を使用する☆．以降で LCDIA アルゴリズムの基本的な部分を説明する．

属性インスタンス b の値が変化すると，b がそのノードの合成属性ならその親オブジェクトに，子オブジェクトの相続属性なら子オブジェクトに，シグナル PROPAGATE (b) が送られる．最初の PROPAGATE シグナルは動的意味規則の記述内容の評価により送られると考える．

PROPAGATE シグナルハンドラは，以下のことを行う．

☆ システムメッセージの意味．OOAG のメッセージと区別するため「シグナル」と呼ぶ．

- ローカルモデル M_L の拡張
- M_L から独立した属性を選択
- このような属性のそれぞれについて，その属性値を計算
- 属性値が変更されたら PROPAGATE シグナルを，変更されなかったら RELAX シグナルを送信

RELAX シグナルは，2つのノードオブジェクトの間の不要になった見えない枝を除くために使用される、通常の分散型ではないアルゴリズムでの大域スケジューリンググラフ M の独立した頂点は，その頂点に結び付く属性がけっして新たな値を受け取ることがないことを意味する．局所制御アルゴリズムでは，独立した頂点が2つの意味を持つ．上記の意味に加えて，ノードが PROPAGATE シグナルハンドラを実行していない場合は，独立した頂点に結び付く属性はその値に変更が起こる候補となる．言い換えると，そのノードは PROPAGATE シグナルか RELAX シグナルがこれらの属性に対して到着するのを待っている．すなわち，ローカルモデル M_L 中の独立した頂点は，実際に独立しているわけではなく，これらの頂点には見えない枝が存在する．この見えない枝は，そのうち他の NODE オブジェクトにより送られた PROPAGATE か RELAX シグナルが到着するであろうことを表す．つまり，M_L 中の本当に独立した頂点は，PROPAGATE シグナルが到着したときに毎回判定されなければならない．PROPAGATE, RELAX シグナルのシグナルハンドラを，それぞれ **図5**，**図6** に示す．

4.2.3 高速な評価器の設計

OOAG の静的属性の評価を高速に行うためには，特に以下の点に注意する必要がある．

- ATTRIBUTE オブジェクトのアドレス変換
- ATTRIBUTE オブジェクトの依存関係の抽出の最適化
- メモリ管理方法の簡素化

ATTRIBUTE オブジェクトは，属性インスタンスに関する情報を格納するオブジェクトである．以下，それぞれの項目について我々がとった方法を説明する．

4.2.3.1 NODE オブジェクト間での ATTRIBUTE 参照のためのリンクポインタ

NODE オブジェクト間での ATTRIBUTE 参照は，属性値の評価の以下の場面で必要になる．

(1) PROPAGATE, RELAX シグナルの処理の最初で，シグナル発信元の属性値を得る操作

(2) NODE オブジェクト間の依存グラフのマージ操作で，各頂点の示す属性インスタンスをターゲットの NODE オブジェクトの情報に変換す

```
PROPAGATE(b) メッセージハンドラ:
let
  b, d = 属性インスタンス
  NewValue, OldValue = 属性値
  S = 属性インスタンスの集合
in
  if ( b が M_L 中に存在しない ) then
    M_L = M_L ∪ b.C
  fi
  S := M_L 中の b に依存するすべての属性インスタンス
  M_L から b を除く
  while S ≠ φ do
    S から d を1つ選択し, これを S から除く
    if ( M_L には d が依存する属性インスタンスが存在しない ) then
      M_L から d を除く
      OldValue := d の値
      d を評価
      NewValue := d の値
      if ( NewValue ≠ OldValue ) then
        PROPAGATE(d) を出力
      else
        RELAX(d) を出力
      fi
    else
      d を評価されなければならない属性インスタンスとしてマーク
    fi
  od
```

図5 LCDIA アルゴリズム—PROPAGATE メッセージハンドラ

Fig. 5 LCDIA algorithm—PROPAGATE message hanlder.

```
RELAX(b) メッセージハンドラ:
let
  b, d = 属性インスタンス
  NewValue, OldValue = 属性値
  S = 属性インスタンスの集合
in
  S := M_L 中の b に依存するすべての属性インスタンス
  M_L から b を除く
  while S ≠ φ do
    S から d を1つ選択し, これを S から除く
    if ( d が評価されなければならない
        属性インスタンスとしてマークされている ) then
      M_L から d を除く
      OldValue := d の値
      d を評価
      NewValue := d の値
      if ( NewValue ≠ OldValue ) then
        PROPAGATE(d) を送出
      else
        RELAX(d) を送出
      fi
    else
      RELAX(d) を送出
    fi
  od
```

図6 LCDIA アルゴリズム—RELAX メッセージハンドラ

Fig. 6 LCDIA algorithm—RELAX message handler.

る操作

このうち1に関しては, シグナル発信時に属性値も引数として渡す方法もあるが, シグナル処理の queue に渡す情報が増えるので採用しない. また, 他の局面でも NODE オブジェクト間の ATTRIBUTE 参照が必要なので, この参照操作の高速化は不可欠である.

この ATTRIBUTE 参照の高速化のために, NODE オブジェクト間の対応する ATTRIBUTE オブジェクトの間に双方向リンクを作成する. このリンクの初期化は, NODE オブジェクトを作成して木構造に接続する時点で行うことができる (**図7**, `NODE::graft()` メソッド). `graft` メソッドは, 引数 `target` をルートとする部分木を, `newobj` をルートとする部分木に置換する. OOAG では, ある NODE オブジェクトの固有属性として接続される NODE オブジェクトは, OSL 記述からあらかじめ判明している. このため, この親オブジェクトの方で, ATTRIBUTE の双方向リンクを行うことができるのだが, これはそれぞれの NODE オブジェクトのクラスにより異なる作業で, したがってこれも OSL 記述のトランスレート時にコード生成しなければならない (図7, `setup_attribute_backpointer()` メソッド).

4.2.3.2 依存グラフの始点頂点によるハッシュ化したグラフ構造

OOAG で扱うグラフ構造は, 属性インスタンスの依存関係を表現する目的のもので, 適用される操作は以下のものに限られている.

(1) 特定頂点に依存する頂点の獲得

(2) 指定した頂点だけからなる到達可能性を示す部分グラフの抽出

(3) グラフの合成

これらの操作を高速に行うためのグラフ構造を設計しなければならない. この目的のためには1番目の操作を高速に行えることが重要で, これができれば2番目の操作も高速に行えるようになる. そこで我々は, グラフ中の始点頂点から出る有向枝の終点頂点をすぐに得られるように, 始点頂点でハッシュ化したグラフ構造を設計した.

なお, 巨大な構造のグラフを扱う必要はない. これは, LCDIA アルゴリズムでは, 大域的な属性依存は扱わず, それぞれの NODE オブジェクトの局所的な属性依存だけを表現できればよく, かつ各 NODE オブジェクトの属性インスタンスの個数は定数でおさえられると仮定しているからだ. 4.2.1 項での説明のとおり, 近隣の NODE オブジェクトでは, 同じ属性を表すためのフィールドを分散して保有している. このため, 同じ属性を表す頂点を2つの NODE オブジェクト内のグラフの間で対応付け, グラフ操作のドメインを合わせたうえで操作を行う必要がある. このドメインを合わせる操作については, 上記項目で説明した NODE オブジェクト間での ATTRIBUTE 参照の方法で高速な処理が可能である. これは, 順序は異なるが, 下記項目で説明するようにグラフ頂点が直接 AT-

```
class ATTRIBUTE
{
        :
   BaseType      *value1;   // 属性値
   ATTRIBUTE *value2;   // 関連情報へのポインタ
        :
};

NODE::graft(Node &target, Node &newobj)
{
     :
  target = newobj ;
  setup_attribute_backpointer();   // 属性リンクの接続
  initialize_transitions2();        // メッセージリンクの接続
     :
}

some_rhs_class::setup_attribute_backpointer()
{
        :
  attr[x].value2 = & some_new_node->some_att
  attr[x].value2->value2 = & attr[x].value2;
        :
}
```

トランスレータでコード生成

図7 ATTRIBUTE クラスとその初期化のコードの断片
Fig. 7 Fragment of ATTRIBUTE class and it's initialization code.

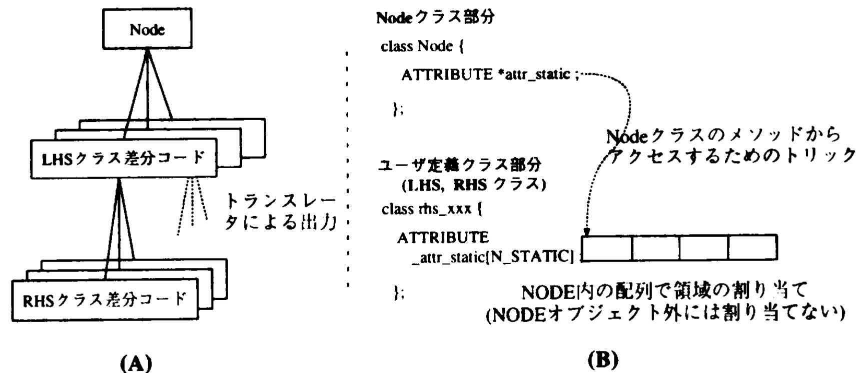


図8 NODE クラス群の構造
Fig. 8 NODE class hierarchy.

TRIBUTE オブジェクトを参照するような設計を行っているからである．

4.2.3.3 ATTRIBUTE オブジェクトの管理方法と依存グラフの構造

ATTRIBUTE オブジェクトの管理方法について説明する前に，まず NODE クラス群のクラス構造について説明する．NODE クラス群は，Node 抽象クラスを親とする3層のクラス階層構造を持つ（図8(A)）．Node の直接の子となるのは，OSL 記述の LHS クラスに相当する部分の差分情報を定義する抽象クラスで，このそれぞれの抽象クラスのサブクラスとして RHS クラスに相当する差分情報を定義するクラスである．これらの OSL 記述から生成されるサブクラス群は，OSL 記述トランスレータによりコード生成される．

属性に関する情報（ATTRIBUTE オブジェクトなど）の大きさは，それぞれの RHS クラスにより異なるため，RHS クラスから生成されるサブクラス中で割当てを行う．このとき NODE オブジェクト内の領域として ATTRIBUTE オブジェクトを割り当て，ATTRIBUTE の領域管理を NODE オブジェクトと一元化している．これは，NODE オブジェクトに関する情報領域の一括割当てを行うことにより，領域管理処理のオーバヘッドを減らすためである．ただし，ここで1つ注意しなければならない点がある．実際の NODE オブジェクトは，RHS クラスから生成されたクラスから実体化されるが，属性評価器は Node クラスのメソッド群として設計されている．属性評価器から属性データにアクセスする必要があるのだが，このままではアクセスすることができない．このため，属性評価器から属性データにアクセスするために，Node クラス部分に実際の属性データを参照するためのポインタを用意している（図8(B)）．これにより，Node クラスのメソッド群である属性評価器からのアクセスを可能にしている．

次に属性の依存グラフの構造について説明する（図9参照）．

グラフ操作のために枝の追加・削除が頻繁に行われる．このため，この周りの領域管理を効率良く行うことが重要である．今回の設計は以下のとおり．

- グラフの頂点として ATTRIBUTE オブジェ

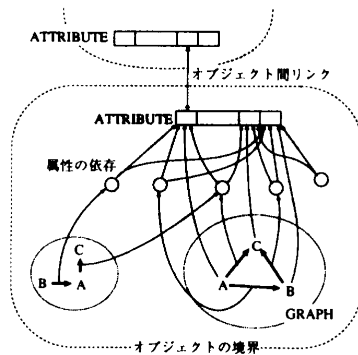


図9 グラフ構造の模式図
Fig. 9 The structure of graph data.

クトへのポインタを採用．前述のとおり，ATTRIBUTE オブジェクトは NODE オブジェクトとともに管理されているので，グラフ操作から頂点オブジェクトの領域管理問題を取り除いた．

- グラフの有向枝は，始点と終点の頂点を指すポインタの組．頂点である ATTRIBUTE オブジェクトでは，その頂点への入力枝・出力枝を管理する．
- 頂点・枝オブジェクトが特定のグラフに属するかどうかは，グラフとは別の概念．それぞれの頂点・枝オブジェクトは複数のグラフに属することもありうる．また，どのグラフに属さないこともありうる．このため，枝オブジェクトの領域解放は，頂点である ATTRIBUTE オブジェクトの領域解放時に行う．

このグラフ構造設計の要点は，領域の開放は NODE オブジェクトの領域管理問題にまとめて，グラフ操作から領域管理問題を取り除いたことで，これにより処理を高速にしたことである．

4.2.3.4 OODBMS 利用に関する問題

以上の高速化技術の要点は，以下のものである．

- OSL からのコード生成時に判明する静的情報を展開し，実行時の作業量を軽減すること
- 領域管理に関する問題を NODE オブジェクトにまとめ，効率の良い領域管理を行うこと

我々の方針では，データベース生成に際して NODE オブジェクト自体をデータベースに格納することにしている．そして NODE オブジェクトには，属性評価を行うための機構が組み込まれている．つまり，この属性評価アルゴリズムは，木構造のデータを1つ1つたどって計算を行うものではなく，従来のデータベースアプリケーションとは異なる性質を持つ．これを実現するためには，オブジェクト指向データベース管理システムを用いて，永続的オブジェクトとして NODE オブジェクトを実現するのが良い方法である．我々は，この観点から現在利用可能なオブジェクト指向データベース管理システムの調査を行った．

- NODE オブジェクトを永続化
- 永続オブジェクトの扱いの高速さ
- データベース操作が C++でできること
- C++のポインタがオブジェクト ID になることが望ましい
- 高速であること

有力な候補は，メモリマップ型アーキテクチャを持ち，今回の技術がほぼそのまま有効で，かつライブラリ集も豊富な ObjectStore で，現在これを利用してデータベース化する作業を進めている．

4.3 OSL コードトランスレーションシステム

我々は，OOAG から C++への OSL コードトランスレーションシステムを Synthesizer Generator (CSG) を利用して作成した．OOAG のクラスは，それぞれ対応する C++のクラスに変換される．LCDIA 属性評価器は，これらの変換されたクラスのスーパークラスにおいて，一連のメソッド群で実装されている．このため，コードトランスレーションシステムは，それぞれのクラスで異なっている部分しか生成しない．このようなコードには，主に属性インスタンス，それぞれの属性評価関数，ノードや属性インスタンスの初期化のために様々なコードなどがある．

コードトランスレーションシステムは，属性インスタンスの値へのアクセスを最適化するために，4.2.3 項で述べたように，複雑なデータのマッピングを展開する．これには，OOAG のシンボルを C++のコード情報へ写像するたくさんのマッピングテーブルを扱う必要がある．この点では，属性文法が非常に役に立った．

我々のコード変換の作業が属性文法に基づくシステムで行われていることは，とても興味深い点のひとつだ．属性文法は，今回の OSL トランスレータのような言語処理系の構築には相性が良く，短期間での開発が可能である．OOAG は，属性文法の属性付き木に相当するデータをデータベース化する．これには，一貫性のとれている状態の属性値も含んでいる．したがって，構文解析の問題を別にすれば，OOAG ならソースコードの更新に応じて，トランスレーション後のコードを自動的に生成するようなシステムを比較的容易に構築できる．現在 CSG の属性文法で記述しているコード生成規則を OOAG に置き換えればいい．この話を一般化すると，リポジトリの内容の変化に応じて自動的にコンパイルの結果を生成するようなシステムが得られるであろうことを示している．

現在の OOAG の記述言語 (OSL) では，まだクラ

ス定義の集合しか生成できない．トランザクションの記述は，C++のコードを直接記述する必要がある．

コード変換で行っている主な作業を以下にまとめる．

- 属性領域の割当て
- 局所属性依存グラフの生成
- 属性値の評価関数の生成
- トランジションの処理コードの生成
- 各種領域の初期化コードの生成
- オブジェクトの接続用コード（属性インスタンス，メッセージ，その他）の生成．
  属性名から物理ポインタへの変換も含む．

4.4 性能評価

ここでは，アルゴリズム確認の目的で xlisp[15] インタプリタ上に構築されたプロトタイプシステムと，本論文で取りあげている C++に変換して実行するシステムの処理速度の比較を行い，C++バージョンの設計において導入した属性評価アルゴリズム実装のうえでの技術の有効性を確認する．

まず最初に，xlisp 版から C++版での大きな変更点を以下にあげる．

属性インスタンス値の交換方法 xlisp 版では属性インスタンス名による値取得．他オブジェクトの値取得には，参照のたびに毎回動的束縛を行っている．C++版ではトランスレート時にオブジェクトの属性付き木への接続時に，属性インスタンスの束縛を行うコードを生成する．したがって，属性インスタンス束縛はオブジェクトの接続時にだけ行えばよい．参照は束縛後のポインタによる直接参照．

特徴グラフのグラフ構造 xlisp 版では始点ノードと終点ノードの組によるフラット構造．すべてのグラフ操作に対してリスト構造の順探索が必要．C++版では始点ノードによるハッシュ化構造．

最初の項目については，4.2.3 項のとおりである．2 番目の項目については，木構造の変更のたびに起こる特徴グラフのグラフ演算の高速化のためにとった手法である．

これによる効果を確認するために，オブジェクトを増殖させながら静的属性計算を繰り返すベンチマークテストを行った結果が **図 10** である．図 10 では，横軸にオブジェクト数（=木構造の大きさ），縦軸に計算時間をとっている．ベンチマークプログラムの処理内容は，「ルートオブジェクトの相続属性に正数を与えて合成属性に相続属性値のフィボナッチ数を計算する．相続属性値 $in \leq 1$ なら，合成属性として 1 を伝える．相続属性値 $in > 1$ なら，それぞれ $in-1, in-2$ を

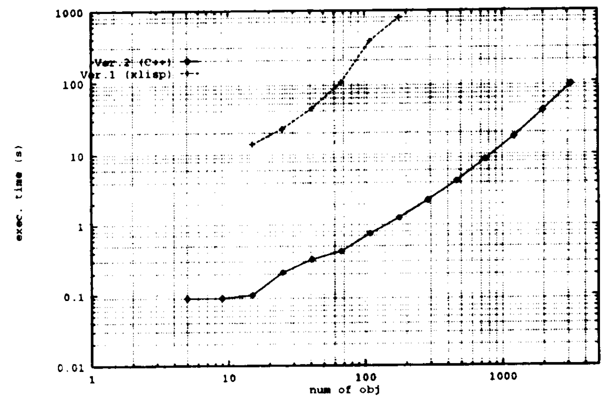


図 10 ベンチマーク結果
Fig. 10 The result of benchmark test.

相続属性として持つ部分木を作成・置換する」というものである．最終的には，ルートオブジェクトの相続属性値のフィボナッチ数の個数の葉オブジェクトを持つ 2 分木が作成され，ルートオブジェクトの合成属性にフィボナッチ数が計算される．大量のオブジェクト生成とこれにともなう静的属性計算の繰返しから，静的計算の効率化のための設計変更と，これにより生じるオブジェクト生成（接続）のオーバヘッドに対する評価が行える．

xlisp は，全関数が C によるビルトイン関数で実装されている，小さく高速な lisp インタプリタである．最近の Common Lisp 処理系では，コンパイルすればかなりの実行速度の向上がなされるが，多くがマクロや lisp ライブラリで準備される Common Lisp の場合と異なり，xlisp の場合には実行速度向上の割合には疑問が残る．同じ xlisp 上での実験ではないので一概には結論できないが，それでも 200 倍以上の高速化ができたのは，設計変更の成果であると考えられる．つまり，他オブジェクトの属性インスタンス参照のたびに束縛を行うのに比べれば，オブジェクト接続時の付加的作業量も問題にならないし，グラフ構造の設計も含めてうまくいったと結論できる．

5. まとめ

OOAG は，属性文法に基づく計算モデルで，オブジェクト指向の概念を用いてそれを拡張した計算モデルである．我々は OOAG 記述からリポジトリシステムを生成し，それを実行するシステム MAGE を設計・実装した．我々は OOAG の評価を行うアルゴリズム LCDIA をすでに得ている．LCDIA は，生成規則に対応するそれぞれのクラスに実装されたメソッド集合を操作する分散型のアルゴリズムである．これに対して，通常の属性文法の評価器は，属性付き木にいくつかの

プロセスを割り当てる形をとる．我々は，LCDIA の効率的な実行のためにいくつかの手法を導入し，オブジェクトの永続化の機能を省いたバージョンの実験では，良い結果を得ている．MAGE は，OOAG のオブジェクトをオブジェクト指向データベース中の永続オブジェクトに写像することで OOAG の永続化を扱う．メモリマッピングアーキテクチャを持つ ObjectStore のようなオブジェクト指向データベース管理システムを利用すれば，我々のとった手法は永続化のための拡張を行った後でも効果がなくなることはない．

MAGE では，ソフトウェアリポジトリの構造と操作を同時に，しかも形式的に記述できる．この形式的な記述は，そのまま実行可能な形式に変換することが可能である．

リポジトリシステムをオブジェクト指向データベース上に直接構築する方法と比較して，我々の方法では，設計と実装の区別がないために開発コストを最小に抑えることができる．さらに，仕様から自動生成することによりシステムに単純な欠陥が入り込む機会を減らすことができる．一方，今までのファイルシステムを使用する方法と比較すると，我々の方法では，現在一般に使われているような開発ツールを使用するには，ツールの入出力をカプセル化するような，特別なプログラムが必要になる．しかし，今までのファイルシステム上のツールでは実現できなかったソフトウェアオブジェクトの柔軟かつ自動的な操作をプログラムできるようになった．

我々の方法では，リポジトリの設計が比較的容易なため，開発環境に応じたリポジトリの設計が短期間で行える．加えて，OOAG 記述は属性文法の形式を持ち，それゆえ新しい規則の付加などが容易に行える．なぜなら，CFG 上に分散した関数的記述は，記述の変更に対する影響範囲を特定しやすく，かつ多くの場合少量の記述の組合せなので既存のコードの解析も容易であり，また既存のコードと関係しない記述は従来の記述に影響を与えることなく付加できるからだ．この特徴は，ソフトウェアリポジトリシステムの開発にはとても役に立つものであると考える．なぜなら，ソフトウェアリポジトリの技術はいまだ未熟なので，汎用的なものを構築するにしても特定の目的のものを構築するにしても，数多くの記述実験を通して機能の追加や変更を行っていく必要あるからだ．

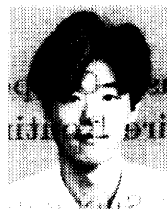
現在 OOAG に関しては，版管理システムの構築[16]，UNIX ファイルシステムを題材にした仕様記述実験[17]，ソフトウェア開発環境記述への応用[5]などの研究が進められている．しかし，記述方法に対する明確な指針がまとめあげられておらず，今後の課題として残っている．

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. International Workshop on Attribute Grammars and their Applications*, LNCS, Vol.461, pp.177–191, Springer-Verlag, (1990).
2) Gondow, K., Imaizumi, T., Shinoda, Y. and Katayama, T.: Change Management and Consistency Maintenance in Software Development Environments Using Object Oriented Attribute Grammars, *Object Technologies for Advanced Software*, LNCS, Vol.742, pp.77–94, Springer-Verlag (1993).
3) Boudier, G., Gallo, F., Minot, R. and Thomas, I.: An OVerview of PCTE and PCTE+, *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Henderson, P. (Ed.), pp.248–257 (1988).
4) Shinoda, Y.: On Application of Attribute Grammars to Software Development, PhD Thesis, Tokyo Institute of Technology (1989).
5) 権藤克彦，今泉貴史，萩原威志，片山卓也：オブジェクト指向属性文法 OOAG のソフトウェア開発環境への応用，電子情報通信学会論文誌 D-I, Vol.J78-D-I, No.5, pp.478–491 (1995).
6) Knuth, D.: Semantics of Context-free Languages, *Mathematical Systems Theory*, Vol.2, No.2, pp.127–145 (1968).
7) Hudson, S.E. and King, R.: CACTIS: A Database System for Specifying Functionally-Defined Data, *Proc. International Workshop on Object-Oriented Database Systems* (1986).
8) Kiesel, N., Schürr, A. and Westfechtel, B.: GRAS, A Graph-Oriented Database System for Engineering Applications., *CASE '93 6th Int. Conf. on Computer-Aided Software Engineering*, Lee, Reid, J. (Ed.), pp.272–286, IEEE Computer Society Press (1993).
9) GrammaTech, Inc.: The Synthesizer Generator Reference Manual, Fourth edition, One Hopkins Place, Ithaca, NY (1992).
10) Tichy, W.F.: RCS – A System for Version Control, *Software – Practice and Experiences*, Vol.15, No.7, pp.637–654 (1985).
11) Glasser, A.L.: The Source Code Control System, *IEEE Trans. Softw. Eng.*, pp.364–370 (1975).

12) Feldman, S.I.: Make – A Program for Maintaining Computer Programs, *Software – Practice and Experience*, Vol.9, pp.255–265 (1979).
13) Iida, Y.: OSL Language Specification Ver.2.0 (in Japanese), Technical Report 96TR-0012, Department of Computer Science, Tokyo Institute of Technology (1996).
14) Object Design, Inc.: ObjectStore User Guide: Library Interface, 25 Burlington Mall Road Burlington, MA 01803 (1993).
15) Betz, D.: *XLISP: An Experimental Object Oriented Language* (1985).
16) Imaizumi, T., Shinoda, Y. and Katayama, T.: Description and Implementation of File Management System Using Attribute Grammars, *Proc. International Conference Organized by the IPSJ to Commemorate the 30th Anniversary – InfoJapan'90 Information Technology Harmonizing with Society*, pp.143–150, Information Processiong Society of Japan (1990).
17) 今泉貴史，権藤克彦，萩原威志，松塚貴英，片山卓也：構造指向型システムのための実行可能な仕様記述言語，情報処理学会論文誌，Vol.36, No.5, pp.1126–1137 (1995).

(平成 8 年 5 月 27 日受付)
(平成 8 年 9 月 12 日採録)



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 年同大学大学院理工学研究科修士課程修了．現在，同大学大学院博士後期課程に在籍．属性文法型言語の処理系およびその開発環境の構築方法に興味を持つ．日本ソフトウェア科学会会員．

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，ソフトウェア開発環境，ソフトウェア方法論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．