論文 / 著書情報
Article / Book Information

題目(和文)	オブジェクト指向属性文法によるソフトウェアリポジトリの生成システム
Title(English)	
著者(和文)	萩原威志
Author(English)	
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第3460号, 授与年月日:1997年3月26日, 学位の種別:課程博士, 審査員:
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第3460号, Conferred date:1997/3/26, Degree Type:Course doctor, Examiner:
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

オブジェクト指向属性文法による
ソフトウェアリポジトリの生成システム

指導教官　片山 卓也

提出者　　東京工業大学
理工学研究科 情報工学専攻
萩原 威志

目 次

図目次

表 目 次

第1章　はじめに

1.1　概要

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）の管理を行な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なうソフトウェアリポジトリは複雑なシステムであり、リポジトリ設計からシステム実現の作業量もかなりのものとなる。そのため、構築したリポジトリのシステムのテスト・デバッグにかかる時間も無視することが出来ない。我々は、この問題に対するひとつの解決法として、抽象的な記法からの機械的なリポジトリの構築システムを作成した。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。

1.2　ソフトウェア開発におけるリポジトリの重要性

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 [KL89, ZM90] 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なうのは難しい。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 [BGMT88] 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な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な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なうのは難しい。

1.3 リポジトリシステムの自動生成の必要性

現在、ソースコードの管理を目的としたものには、様々な開発環境やツールが存在するが、高度な関係処理を行なう実用的なソフトウェアリポジトリを有するものは存在していない。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、バージョンなどの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。言い換えると、①リポジトリに対する仕様の未熟さ、②リポジトリの設計技術の未熟さ、が問題であり、この問題に対応するためには、試験的なリポジトリのシステム構築を行ないながら、経験を得ていくしかないと思われる。このためには、高度な関係処理をすばやく記述し、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となる。

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リポジトリシステムを、オブジェクト指向データベースとして自動生成することを考える。この目的のために、我々は計算モデル OOAG [SK90a, Shi89, 権藤 95, GISK93] を採用する。OOAG は、属性文法 [Knu68] 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、そして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。これらの機能は、リポジトリの仕様記述に応用できるものであり、この OOAG の記述から属性評価器を構築することで、動作可能なリポジトリのシステムが機械的に構築できる。

1.4 関連研究

CACTIS [HK86], GRAS [KAW93] 等のシステムは、効率的な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 Synthesizer Generator[Gra92] 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言える。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み合わせで実現できる範囲内のことを、少し便利に行な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もちメンテナンスし易いという特徴を受け継いでいる。

1.5 MAGE システム

MAGE は、OOAG モデルに基づき記述されたリポジトリ仕様を、オブジェクト指向データベース管理システム上のデータベース・クライアントに変換して実行するシステムである。MAGE を利用することにより、OOAG モデルで記述されたリポジトリ仕様が、そのまま計算機上で動作するリポジトリシステムに変換できる。

我々は、MAGE をリポジトリ生成系を含むリポジトリシステムの開発環境として構築し、リポジトリシステム生成系だけでなく、仕様設計を援助する GUI や構造エディタも構築した。これにより、従来は大変大がかりであったリポジトリ構築のためのシステム開発を、手軽にそして短期間で行なえるようにした。これは、かなり重要な意味を持つ。従来は膨大な予算と時間をかけて開発していた、そしてコード管理のような通常の一般的な作業工程を持つような箇所であれば、市販のものをカスタマイズして利用していたリポジトリ構築システムを、ユーザが独自の開発環境に合うように自分で構築できるようになるからである。

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。属性文法には、処理可能な文法の大きさによって様々な属性評価の戦略が存在し、特に OOAG の特殊な属性評価のためには篠田により LCDIA アルゴリズムが得られている。しかし、その属性評価では、属性評価器の構築時ではなく、実行時に処理しなければならない作業量が多いために、実現の仕方により属性評価器の実行速度に大きな差が生じる。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、実行時の処理にかかる時間を極力減らしている。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 での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。

1.6 本論文の構成

本論文は、以下の構成をとる。

2 章では、我々が機械的に生成することを考えているソフトウェアリポジトリの性質を明確にし、OOAG の応用に関する我々の考え方を示す。現状のソフトウェア開発で利用されているリポジトリの性質、実現方法について議論し、その後 OOAG を利用してどの

ようにリポジトリを生成するのかを述べる。

3章では、OOAG の解説を行なう。主に OOAG における計算がどのように進むのか、そして OOAG モデルの計算を記述する言語である OSL 言語の言語仕様に関して説明している。

4章では、OSL 言語により記述されたリポジトリ仕様から、実行可能なデータベース・クライアントを構築する MAGE システムの説明を行なう。MAGE は、実際には OSL 言語のプログラミング環境として構築されており、構造エディタや GUI ツールによる OSL での記述の補助から、生成したシステムのデバッグまでを支援するものであるが、ここでは記述されたリポジトリの仕様を実行する、属性評価器の設計と実現での高速化のためのテクニックの説明を主としている。

5章では、MAGE により OOAG の計算モデルを応用したアプリケーション構築を行なうことに関して、いくつかの観点から考察を与えた。また、ここでは C のソースコード・リポジトリを構築する場合の手順を、実例をあげながら説明している。

最後に6章で、これまでに片山研究室で MAGE, OOAG に関して行なわれてきた研究、および MAGE, OOAG に関連の強いいくつかの研究に関して簡単にまとめ、OOAG の将来の課題について議論している。

第2章　ソフトウェアリポジトリ

ソフトウェア開発の成果物は、目的の計算機環境で動作するソフトウェアパッケージであるが、計算機環境の複雑化や計算機自身の高機能化により複雑な処理を高速に行なうことが可能になったために個々のソフトウェア自身、および目的のシステムを実現するこれらのソフトウェアによるパッケージが巨大化し、成果物を得るまでの膨大な量の情報（ソフトウェアオブジェクト）を、工業的な手法で効率的に扱う必要性が生じてきている。このソフトウェアオブジェクトの工業的な手法での効率的な管理を行なうのがソフトウェアリポジトリで、ソフトウェア開発環境にとって欠かせないものであるといえる。

ただし、一口にソフトウェアリポジトリと言っても、この言葉は様々な考え方のもとで、いろいろな使われ方をしている。たとえば、ファイルのバージョン管理ツールにより使用されるただのファイル置き場もリポジトリと呼ばれるし、高度なソフトウェア処理を行なうソフトウェア開発環境の高機能データベースシステムも、ソフトウェア開発環境のツール統合のための統一化したデータ表現機構を持たせたシステムも、ただリポジトリと呼ばれる。

この章の目的は、我々のソフトウェアリポジトリの自動生成システムにより構築されるリポジトリシステムの特徴を理解し、有効性を示すことである。まず、最初に2.1節では、現在のソフトウェア開発環境において、どのようなものを使ってリポジトリを構築しているのか、その実現方法をまとめる。次に、2.2節で、現在のソフトウェア開発環境では、どのようにリポジトリを利用しているのか、そしてそれをどのようなシステムを利用して構築しているのかをまとめている。2.3節では、ここまでの調査を元に、OOAGをどのようなリポジトリ構築に応用しようとしているのかを明確にする。2.4節では、属性文法に基づいたCASEツール生成系の研究と関連の強い構造指向プログラミング環境においてのデータ共有の技術と、OOAGをそのような環境構築に応用する場合の大きな違いについて考察している。最後に2.5節では、CASE環境のデータ交換や、データ蓄積における国際標準に関する話題について、OOAGでのリポジトリとの関連を考察している。

2.1 リポジトリの実現方法

ソフトウェア開発においてリポジトリを構築するには、おもに以下の4つの方法が存在し、様々なシステムが構築されている。

- 通常のファイルシステム上にディレクトリ構造でファイル単位で蓄える。
- データベース管理システム（DBMS）を利用してデータベースとして構築する。
- ファイルシステムとデータベース両方を利用する。
- ファイルシステムそのものをリポジトリ用に拡張する。

以下、それぞれについて簡単に説明する。

2.1.1 ファイルシステム上での実現

ファイルシステム上にディレクトリの階層構造を構築し、RCS [Tic85]、SCCS [Gla75] などを利用しチェックイン、チェックアウトによるファイルのバージョンの管理、MAKE [Fel79] によりソフトウェアのコンフィギュレーションの管理を行なう。特別なツールやシステムが必要になるわけではなく、手軽な方法であるが、ディレクトリ階層の構造、ファイルの名前つけの規則などにより、蓄えるファイルの管理を行なうしかない。また、1つのファイルに記述する情報量や内容の切り分けなどを規定することは出来ず、利用者に任せるしかないため、管理仕切れなくなる要素が多い。

この際のソフトウェアオブジェクトである、ファイルの管理作業の自動化は、すべてツール側に任される。その管理ツールのコンフィギュレーションもまた大変である。たとえば、ファイルが増えた場合や、ディレクトリ構成に変更があった場合のMakefile のメンテナンスなどである。

以上の理由から、ファイルシステムだけを利用して生産性の高いリポジトリ構築を行なうのは難しい。

2.1.2 DBMS を利用する実現

ファイルシステム上のリポジトリよりも、高度な情報管理を行なうことを目的として、ソフトウェアリポジトリがデータベースとして構築されることもある。[Not85, Ber87] などでは、ソフトウェア開発環境におけるデータベースは、管理すべきオブジェクトが実世界の抽象ではなく、それ自身データベースにより管理されるようなプログラムやドキュメント類などのメタな概念を含むために、リレーショナル型の一般のデータベース管理システムでは、実現が難しいとされていたが、オブジェクト指向型などのデータベース技術の進歩により、データベース上に構築されるリポジトリも増えている。

リポジトリをデータベースとして構築する目的は、主に「ソフトウェアオブジェクトへの型付けによるツール間のデータ共有」と「高度な索引構築と情報検索」である。PCTE や IRDS [溝口 88] などのリポジトリの標準環境では、リポジトリ内容の標準化作業も進められている。

DBMS を利用したリポジトリ構築の際の難点は、データベースにアクセスするための専用の開発ツール群が必要になることである。索引を上手に作れば情報の検索は容易になるが、データベースの設計の良否により開発環境が使い物になるかどうかが左右される。データベースアクセスの速度的問題も大きく、高速なデータベース操作が必要である。一般にデータベースの設計は難しく、リポジトリのシステム実現までの手順は費用のかかる作業である。

2.1.3 融合型の実現

ソフトウェアオブジェクトの格納にはファイルシステムを利用し、それぞれのソフトウェアオブジェクトの付加的な属性情報だけをデータベースで管理する実現方法もある。ClearCase [Atr92] などの商用コンフィギュレーション管理ツールで用いられている。ファイルシステムベースのツールをそのまま利用できる利点がある。データベースは、補助情報の管理にだけ用いるので、データベースのシステムに対する要求も大きくなく、通常のリレーショナル型のデータベース（RDB）で十分である。

2.1.4 拡張ファイルシステムの構築

ソフトウェアリポジトリのために、通常使われるファイルシステム自体を拡張する試みも行なわれている [CL93]。ファイルシステムを拡張し、ユーザからは透過的にバージョンの管理などを行なったり、他のファイルから派生するファイルの管理機能、ファイルの不変性の宣言などの機能を付加している。通常のファイルシステム上で動作する使いなれたツールがそのまま利用できるのが利点である。しかし、利用できる環境は限られる。

2.2 現在のリポジトリの使われ方

ソフトウェアリポジトリを構築するためのデータベースに、必要であると言われている主要な機能的項目について、現在のソフトウェア開発環境の実験システムなどで作成されているリポジトリについて、調査した結果が表2.1である。

表2.1のそれぞれの項目について説明する。

データモデル リポジトリに蓄えるソフトウェアオブジェクトの型を定義するためのモデル。現在では、ほとんどがオブジェクト指向型か、ER モデル拡張型である。

リンク ソフトウェアオブジェクト間の関連リンク機能。自由なリンクの作成が可能かどうか。オブジェクトに構造を持たせるだけでなく、関連するものをリンクして相互参照を可能にすることができるか。

永続性 リポジトリ中のソフトウェアオブジェクトの永続性を明確にサポートするかどうか。ツール（ソフトウェアプロセス）からツールへのデータの受け渡しのために一時的に利用する場合など、永続性を要求しない用途も存在する。必要かどうかは、使われ方による。

バージョン管理 リポジトリ中のソフトウェアオブジェクトのバージョン管理機能をリポジトリのシステム側でサポートするかどうか。

整合性チェック リポジトリ中のソフトウェアオブジェクトの意味的整合性チェック機能を含むかどうか。

オブジェクト構造化 複合オブジェクト（他のオブジェクトを含むオブジェクト）を扱える。

イベントマネジメント（トリガ） 外部からのイベントに対して処理を開始する機能を持つか。リポジトリ中のデータを、常に一貫した状態に保つための機能を実現するために、イベントを監視して不整合なデータを再計算するために必要となる。

long, nested トランザクション ロングトランザクション、ネストしたトランザクションのサポート。たとえば、ソフトウェアの開発において、ファイルをチェックアウト・編集・チェックインまでをトランザクションと考えると、トランザクションがかなり長期間になることもある。このような事態に対応する機能を有するかどうか。

2.2.1 「ソフトウェア開発環境」でのリポジトリ

ソフトウェア開発環境、およびソフトウェアプロセス記述におけるリポジトリの主な役割は、記述されるプロセスのそれぞれの作業項目の間のデータの橋渡しである。

表2.1のTRITON [Hei92] と PLEIADES [TC93] は、ともに ARCADIA [TBC+88] プロジェクトで構築されたオブジェクト管理システムである。TRITON は、ソースコードで公開されている実験的オブジェクト指向データベース管理システムである Exodus [CDG+90] を拡張して、複数の言語からの利用の仕組み、C++ から Exodus の記述言語である E への変換系などを含めたシステムとして、PLEIADES は Ada 風の記述言語を持ち、以下を特徴とするオブジェクトマネジメントシステムとして実現されている。

- 属性つきグラフに基づく型モデル
- 永続性

	EPOS	MERLIN	ALF (PCTE OMS)	ADELE	TRITON	PLEIADES	CACTIS	GRAS
データモデル	Object-Relation		拡張 E-R	拡張 E-R	C++/E 自動変換	属性つきグラフ	属性つきグラフ	属性つきグラフ
リンク			○	○	○	○	○	○
永続性	○				○	○		
バージョン管理	○			○				
整合性チェック		○				○	○	○
オブジェクト構造化	○		○	○		○	○	○
イベントマネジメント (トリガ)	○	○		○	○			○
long, nested トランザクション	○	○						○
備考		PROLOG ベース, GEMSTONE, GRAS 使用	PCTE OMS 使用		Exodus を拡張			並行・分散・マルチユーザサポート

表 2.1: リポジトリに関する調査

- 柔軟な query ベースのリポジトリアクセス機能
- オブジェクトの一貫性の管理

TRITON や PLEIADES は、統一した型モデルを用意することにより、アクセスするツールのデータ共有をめざしている。名古屋大学の阿草研究室で構築されている Sapid [吉田 95] は、既存のリポジトリが扱うことのできない細かな構成要素を管理する細粒度ソフトウェア・リポジトリに基づいた CASE ツール・プラットフォームであるが、基本的な立場は従来からのデータベースやアクセス言語の設計であり、TRITON や PLEIADES などと同じ範疇のシステムであると言える。

ALF [CBD+94] は、プロセス中心型のソフトウェア開発環境で、PCTE のリポジトリである OMS を利用してデータ管理を行なっている。

EPOS [CDG+89] は、一般的、統合化された、知識ベースをもつのプログラミング環境をめざしたもので、ツールインタフェースを提供するデータベースシステムを中心に据えている。

ADELE [BE86] は、元は Pascal のプログラミング環境におけるプログラムコードの管理に特化したデータベースとして設計されたもので、後に任意の言語で記述された大規模なプログラムを管理するための、一般的なデータベースとして設計し直されている。

MERLIN [PS92] は、ルールベースの永続的なソフトウェアプロセスの実行機能を与えるプロセス中心型の開発環境である。それぞれのルールは、オブジェクト指向データベースに永続的に格納される。したがって、MERLIN で重要視されているのは、ソフトウェアプロセスのルールを管理するリポジトリで、ルールの整合性などのチェックを行なうものである。

2.2.2 データベースの拡張

データベース上へのリポジトリ構築については、2.1.2 節でふれたが、近年のオブジェクト指向データベースの技術の進歩により、データベースのシステム側に対する問題は解決されてきている。しかし、複雑なソフトウェアオブジェクトを処理するリポジトリを、実際にどのように設計するかという問題は依然困難さを残している。

リポジトリ設計の困難さを少しでもデータベースのシステム側で吸収するために、データベースシステムの拡張も盛んに行なわれていて、実際にいくつかのデータベース管理システムが構築され、ソースコード公開されているものも存在する [KAW93]。そして、このような公開されたデータベース管理システムを実際に利用して構築されたプログラミング環境も存在している。

このようなデータベース管理システム側の拡張で重要視されるものをリストする。

- 柔軟なデータモデル記述
- 派生オブジェクトの効率的な計算

- 利用言語インタフェースの拡充

表2.1では、拡張型のデータベース管理システムとしてCACTISとGRASをとりあげている。特にGRASは、ソースコード公開されていて自由に利用できるデータベース管理システムとして、かなり高い完成度を持ったシステムである。たとえばMERLINでは、オブジェクト指向データベース管理システムであるGEMSTONE [Gem95]とともにGRASも利用しており、用途に応じてそれぞれのシステムの特徴を活かして使い分けている。またTRITONでは、Exodusを拡張して利用している。

CACTISとGRASは、派生オブジェクトを効率的に再計算するための仕組みとして、インクリメンタルに再計算を行なう機構を導入したデータベース管理システムである。データモデルとしては、グラフ構造を採用し、この構造の上に**属性**を定義し、派生関係を記述することにより、属性文法の処理系で利用される、インクリメンタル属性評価アルゴリズムを応用して、派生オブジェクトの効率的な再計算を実現している。通常のオブジェクト指向モデルやERモデルのデータベース管理システムの上で、データベースにこのインクリメンタルな再計算機構を実現するのは容易な作業ではなく、この仕組みをデータベース管理システム側に組み込んで、容易に利用できるようにしたことは、大変評価できることである。

2.3 OOAG がターゲットとする領域

ここでは、OOAGの有効性を認識するにあたり扱うリポジトリの性格を明確にする。

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なう。

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) タイプ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、動作を自由に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リポジトリを構築できる。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。これは、2.1.1節で説明したとおりである。

タイプ2のリポジトリ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で、この定義を用いてツールでの情報共有が可能だが、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）の構築が必要。
- 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。通常は、リポジトリのソースコードは提供されず、都合にあわせて改変することも不可能である。
- 汎用目的に開発されたものは、よほど良くできていない限り、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ようと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項目めを簡単に出来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。

2.3.1 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の大量の要求がある。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、ソフトウェアシステムの構成を管理する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は向上する。

- ソフトウェアオブジェクトの変更に伴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[1]
- 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、及びその構造の変更

2.3.2 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点の実現に重点を置く。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, sccs 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なうことが出来るが、これらはファイルベースの管理しか行なうことが出来ず、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な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よ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 OOAG 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。

[1] トリガで管理タスクを開始するのとの違いは、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。

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なうための優れた候補になり得る。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。
- 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、効率がよい。
- 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な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。

2.4 構造指向環境におけるデータ共有

ソフトウェアの開発環境の構築法に関する研究のひとつに 「構造指向アプローチ (structure-oriented approach)」がある。その主な目的は、ソフトウェアの開発を行なうプログラム言語の構文的構造に着目して、開発環境を構成する各種ツールの開発を統合的に行なえるようにすることである。この考え方の元で、実際に数多くの開発環境が実験的に、あるいは商用に開発されている (Gandalf, Synthesizer Generator, Mjølner, etc.) [Not85, Gra92, KLMM93]。

このような構造指向環境におけるツール開発では、扱うプログラム言語の構文的構造を表わす抽象構文木 (AST) をすべてのツールで共有して開発を行なう。言い換えると、共通の AST に基づいてツール開発を行なうわけである。これは、ツール開発における作業の共有、再利用であると同時に、この共通の AST によりプログラムを構文解析して得られた構文解析後のデータを、それぞれのツールで共有することも可能にする。これにより各ツールが、毎回構文解析を行なう作業を省き、解析済みの情報を共有できるわけである[2]。

[2]インタラクティブな用途とは関係のない LR 属性文法におけるインクリメンタル属性評価に関する研

構造指向アプローチは、ツール開発の作業を容易にするだけでなく、開発したその環境におけるプログラム開発も効率良く行なえるのである。

2.4.1 構文構造共有のための技術

ここでは、構文構造（AST）の環境内のツールでの共有に関して考察する。

従来の構造指向環境での AST 共有は、おおよそ 図2.1 に示した形式であった。共有のためにデータベースを導入しているシステムであっても、その利用方法は構文解析後の内部形式のデータを含む AST をデータベースに蓄え、ツールはこれをダンプ/リストアすることにより共有を行なうような形式であった。例えば、構造エディタの既存プログラムの読み込みがデータベースからの各種情報付きの AST のリストアに、そしてセーブがそのダンプをデータベースにコミットする作業に相当していただけであった。言うなれば、従来のファイルシステムベースの開発環境におけるワークスペース（ディレクトリ）が、データベースに置き換わっただけのイメージで、AST データのファイルを蓄えているただのディレクトリと大差なかった。この方法での利点は、ツール起動の度に行わなければならなかった、毎回の構文解析の作業が節約できることのみであった。また、ツールの内部情報も含めてかなり深いレベルの情報まで共有すると、以下のような問題点も存在する。

- 例えば新機能の追加にともない毎回ツールの（バイナリの）再構築が必要になる。
- 再構築の際に、過去の記述内容に対する変更を行なうことが多い。
- 過去の記述で生成されたデータベースの移行の問題が浮上する。

これに対して、我々が OOAG で行なおうとしている共有のイメージは、図2.2 のようになる。データベースと各ツールは、オブジェクト指向データベースを利用した永続記憶機構を利用して結ばれる。従来、各ツールで扱う情報の内部データである属性付き木[3]を、ダンプ/リストアしていた部分はデータベース管理システムの caching メカニズムに吸収される。しかし、注目すべき点はこのようなことよりも、

「共通の AST とツール毎に定義される属性を独立させることの試み」

である。同じ言語を扱う場合でも、各々のツールで AST は共通になるのに対して、内部データである属性とその評価ルールは、ツールにより独立した（disjoint な）部分が多い。そのため、それぞれのツールで定義する属性と評価ルールを、ひとつの属性付き木にまとめようとすると、

(1) 属性名の衝突の回避

究 [中井 96] が行なわれていることを考えると、構文解析済みの情報の再利用に関して構文指向環境の構築以外でも要求が強いことがうかがえる。

[3]OOAG は属性文法の拡張であるため内部データは「AST + 属性」による木で表現されている。

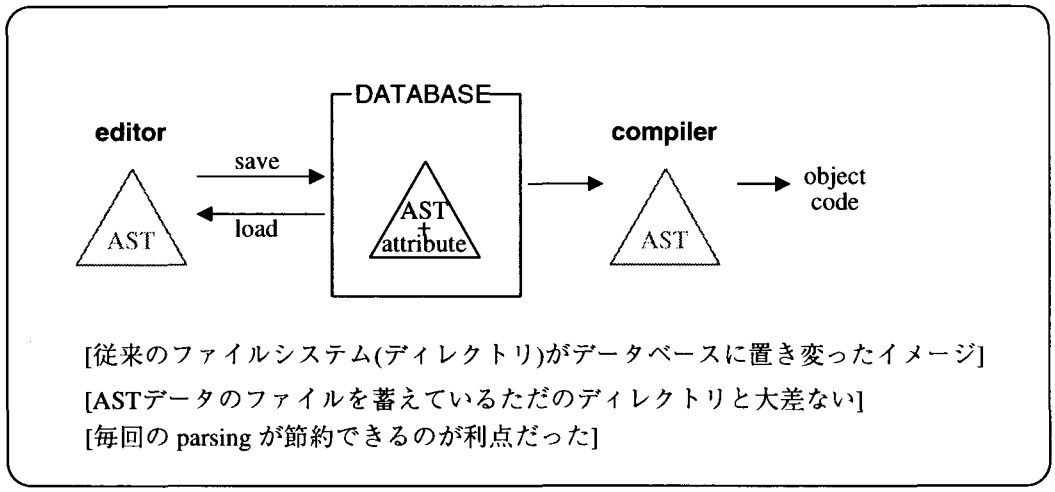


図 2.1: 従来の構造指向環境のデータベースのイメージ

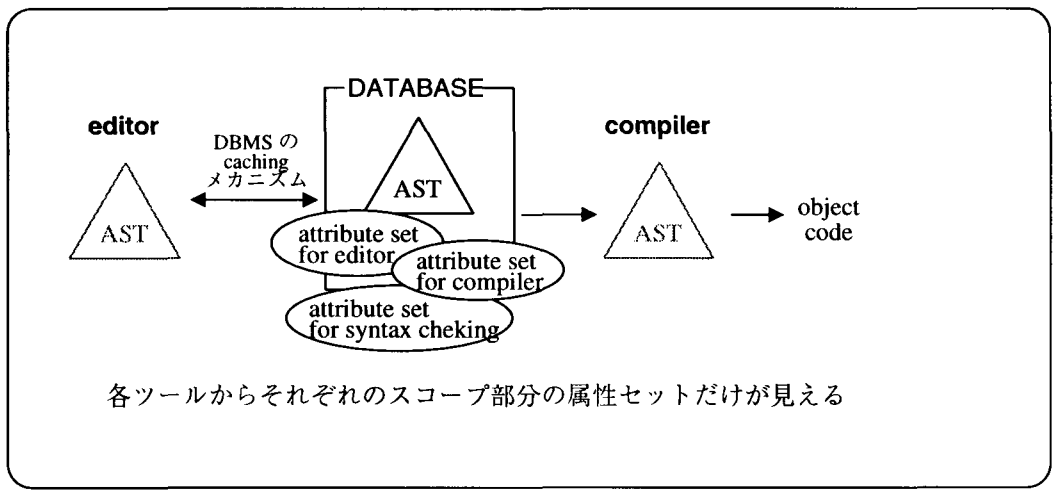


図 2.2: OOAG での構造指向環境のデータベースのイメージ

(2) 効率の良い属性評価戦略がとれなくなる可能性が高いことに対する対策

(3) 新たなツール構築にともなう属性付き木の更新

という問題を解決しなくてはならない。

(1) の問題は、名前の衝突を回避するプリプロセッサの提供により解決できる。(2) の問題は、type 3 circularity を引き起こすような ordered 属性評価器を構築しようとする場合に起こる。しかし、現在の時点で OOAG にこのような属性評価器を提供しておらず、この問題は現時点では差し支えない。(3) の問題は、従来までの構造指向環境において非常に大きな問題であった。これに対して、生成されたデータベース（AST）の自動変換などに関する研究も行なわれている [GKL94] 。OOAG の処理系である MAGE では、データベース・インタフェースにはオブジェクト指向データベース管理システムを利用して、その永続記憶管理機構を応用し、属性付き木情報と属性評価器を一体化している。OOAG の属性付き木は、木の各ノードを表すオブジェクトの集合で表現され、オブジェクト指向データベースに格納されている。このオブジェクトの型情報を含むオブジェクト指向データベース中のオブジェクトに、スキーマ進化のメカニズムを機能させることにより、各ツールで共有している属性付き木の連続的な進化を狙っている。これは、(1), (2) の問題とも関連して、共通の AST 上に disjoint な属性集合でツールを実現しようという考え方の上に成り立っている。

まとめると、MAGE では以下の機能により、効果的に構文木の共有が可能である。

(1) 新たなツール構築時の、属性名の衝突を回避する機能を用意した。ツール記述[4]の際の属性の命名規則制定に対する煩わしさを軽減し[5]、ツール毎に disjoint な属性集合の定義を行なえる。

(2) (1) により、属性・評価関数の追加方向の進化だけに絞り、OODB 上のスキーマ進化機能を利用できる範囲の問題に落した。

(3) この結果、再コンパイルや既存データベースの変換などを行なうことなく、新たなツールの追加を可能にした。

[4]ここでの「ツール」はリポジトリ内のデータのメンテナンスを行なうだけのものも含む。

[5]属性定義を global なもの（他のツールと share するもの）と local なものに分類するだけでいいだろう。

2.5 CASE データの標準化との関連

CASE データの標準化と言えるものに以下がある。

- **CDIF（Case Data Interchange Format）**
  CASE ツール間のデータ交換に用いられる標準形式。CDIF [篠木 93] でのデータ交換は、主に ASCII コードファイルの交換を前提とし、人にも読むことが可能なデータ交換用言語を中心としている。

  データ交換形式の標準化は、オブジェクトマネジメントサービスの基盤となるデータ統合化に位置づけられるが、データ統合化には「ツール間でのデータ交換」と、「統合化されたデータの蓄積・管理」の 2 つの観点があり、CDIF は前者の標準である。

  CDIF では、ツールで扱われるデータのモデルを ERA モデルで表記して、データの変換の方法記述を行なう。

- **データの蓄積・管理の標準化**
  PCTE の Common Data Schema, IRDS の Contents Module などは、リポジトリに蓄えられるデータの型（リポジトリ内容）を規定する標準化である。構築されるツールは、これらの標準のデータ型を利用する形で、データを構築することで、情報の一般化を行なうことが可能である。

我々の OOAG でのリポジトリに対するアプローチは、これらのものとは基本的に行なっていることに大きな違いがある。OOAG では、データの構造だけでなくそれに対する処理を記述する。これは、従来のリポジトリの概念では、ツールの構築において行なうべき内容であった。リポジトリのシステムは、これらのツールに対するインタフェースを用意するだけであり、そのためにデータ共有するためには、蓄える情報の標準化や、異なるツールでデータを交換する場合の標準などが必要であった。

「OOAG で生成したリポジトリの統合環境への組み込み」という観点では、CDIF は役立つ概念である可能性が残っている。属性文法は、「データの変換」に用いるには有利な点が多い。たとえば、属性文法の確立した応用分野であるコンパイラは、高級なプログラミング言語からマイクロプロセッサが解釈する 2 進数列へ、意味を変えずに変換するための変換系である。ただし、属性文法を応用するには変換対象とするデータが木（階層）構造を持つことが前提となる。CDIF では、変換対象とする CASE データのモデルを、基本的に E-R モデルで記述し、それを ASCII テキスト形式の言語表現にして使用する。図情報を表現するこのような言語は、通常のプログラミング言語などとは異なり、構文解析の結果の構文に沿って意味を記述しにくいため、属性文法の技術が有効に利用できるかはわからない。しかし、技術が応用できる可能性は高く、研究の余地がある問題である。

第3章　オブジェクト指向属性文法 OOAG

この章では、オブジェクト指向属性文法 OOAG の計算原理と、その記述言語 OSL の解説を行なう。この章の構成は以下の通りである。

（**3.1 節**）オブジェクト指向属性文法 OOAG の計算原理

（**3.2 節**）記述言語 OSL の言語仕様

（**3.3 節**）OSL 記述の例題

（**3.4 節**）OOAG による関連研究の紹介

3.1　OOAG の計算モデル

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[RT87, RT89] 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 [VSK89, TC90b] 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。OOAG の記述言語である 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 3.1）。

(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。

(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。

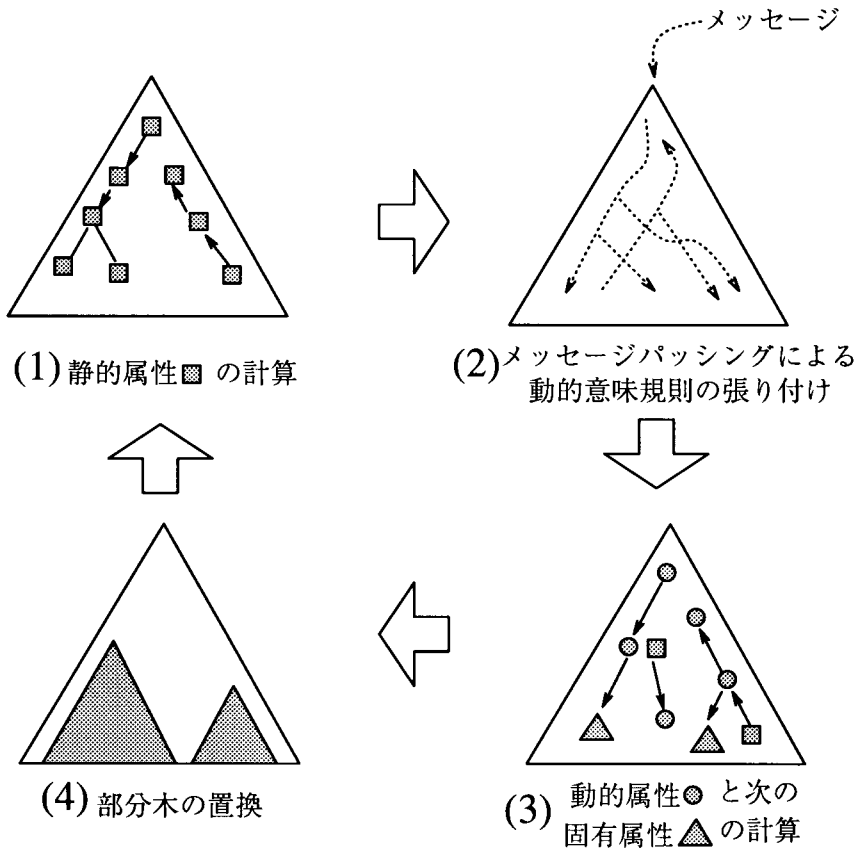


図 3.1: OOAG の評価ループの概念図

3.2 OSL 言語仕様

OSL 言語は、OOAG の計算モデルの記述言語であり、OOAG にもとづく属性文法を記述するための言語である。以下、OSL の言語仕様を解説する。

3.2.1 予約語

次の名前は予約語であり、ユーザーが識別子として使用することはできない。

class, function, local, new, self, with, case, otherwise, not, and, or, xor, imply, mod, exp, true, false, external, int, real, char, string, bool

3.2.2 識別子

識別子は予約語以外の英字で始まり英数字および "_" の列が続いたもので、クラス名、属性名などに用いられる。

識別子の中の大文字と小文字は別のものとして扱われる。

3.2.3 データ型

属性が持つデータ型には、原始データ型、複合データ型、参照データ型の3つの型がある。複合固有属性は複合データ型であり、葉固有属性は複合データ型以外のデータ型である。その他の属性はどのデータ型もとれる。

原始データ型

システムに埋め込まれている構造を持たない型である。

整数型 (int)

整数を要素とする型である。通常の四則演算、剰余、べき乗および大小比較が定義されている。

実数型 (real)

実数を要素とする型である。整数に対する演算のうち、剰余を除いたものが適用可能である。

文字型 (char)

文字を要素とする型である。演算子として一致、不一致のみが定義される。

文字列型 (string)

文字列を表す型である。演算子として一致、不一致の他に文字列の結合 (+) が定義されている。

論理型 (bool)

値として falseか trueを採る。一致、不一致の他に、論理和 (or), 論理積 (and), 論理否定 (not), 排他的論理和 (xor), 含意 (imply) の各演算子が定義されている。

複合データ型

構造を持つデータにはクラスを用いる。属性が持つオブジェクト内も属性を持ち、メッセージを受け取ることができる。静的属性がオブジェクトを持つ場合、オブジェクトの内部構造は静的に決まっているので、部分木の置換を行うメッセージは意味がない。属性が持つオブジェクトのルートノードに相続属性があってはならない。

参照データ型

オブジェクトを示す名前を保持するデータ型である。名前から直接オブジェクト内の属性を参照することはできない。オブジェクトに対するアクセスはメッセージによって行う。

3.2.4 クラス

OOAG では、属性文法での 1 つの生成規則を 1 つのクラスと考え、オブジェクトを生成する。この 1 つの親子関係を定義するクラスを RHS クラスと言い、親子関係を定義する規則を構成規則と言う。構成規則における左辺が等しい RHS クラスをまとめたクラスを LHS クラスと言う。

クラスの定義は静的仕様記述と動的仕様記述から成る。

静的仕様記述

静的仕様記述はクラスの構造を定義するものである。そのオブジェクトが持つオブジェクトとそのオブジェクトに関連づけられる属性を宣言、定義する。

あるオブジェクトが自分の内部に持つオブジェクトを固有属性と言う。固有属性には葉固有属性と複合固有属性の2種類が存在し、LHS クラスとして宣言されているオブジェクトは複合固有属性である。

```
class <LHS クラス名> ( <静的相続属性リスト> | <静的合成属性リスト> )
    → <RHS クラス名> [ <固有属性リスト> ]
    {
        <静的意味規則>
    }
```

相続属性リストと合成属性リストは、

```
<型名> <属性名>
```

の組をカンマ(",") で区切って並べる。
固有属性の記述は固有属性の種類によって異なる。

- 複合固有属性

  ```
  <LHS クラス名> ( <静的相続属性リスト> | <静的合成属性リスト> )
  ```

- 葉固有属性

  ```
  <原始データ型名>  <葉固有属性名>
  <参照データ型名>  <葉固有属性名>
  ```

固有属性リストはこれをカンマ(",") で区切って並べたものである。属性リスト中の属性名、型名、順序はプログラム全体を通して、矛盾があってはいけない。ひとつの RHS クラス宣言に同じ名前の葉固有属性が複数現れてはいけない。
静的意味規則では以下の記述を行う。

- 構成規則の左辺の静的合成属性の値を決定する式。
- 構成規則の右辺の静的相続属性の値を決定する式。
- 静的局所属性の値を決定する式。
- 静的局所属性の宣言

このうち、上の2つは必ず記述されていなければならない。
静的局所属性は予約語 local を用いて、

local <型名> <属性名> ;

と宣言する。

動的仕様記述

動的仕様記述は、メッセージの伝搬を記述する計算規則と動的属性を計算するための動的意味規則から構成される。

<入力メッセージリスト>
⇒ <出力メッセージリスト>
{
<動的意味規則>
}

この記述で、入力メッセージリストのメッセージを全て受け取ったときに出力メッセージリストのメッセージを送出し、動的意味規則に従って動的属性を評価することを意味する。出力メッセージを送出するためには入力メッセージの入力属性がすべて評価済みでなければならない。この制約は、実行速度の効率化のためである。
入力メッセージは空であることもある。これは必ず起動される計算規則であり、入力メッセージがなくても出力メッセージが送出されることを意味する。
メッセージはオブジェクトの間で交わされる。原始データ型の属性に対するメッセージの送受信はない。
メッセージリストは、

<オブジェクト名>:<メッセージ名>(<動的相続属性リスト>|<動的相続属性リスト>)

をカンマ(",")で区切って並べる。オブジェクト名とは LHS クラス名かオブジェクトを持つ属性名である。構成規則に同じ LHS クラスが複数出現する場合は左から順に "$n" (n = 1, 2, ...) をクラス名に付加する。LHS クラス名の代わりに疑似固有属性 self を使うこともできる。self は構成規則における左辺の LHS クラスを表している。self は特別な意味も持っており、self が受け取るメッセージは自分自身が出したメッセージに限る。これにより、プライベートな動的仕様を記述できる。
入力属性と出力属性を次のように定義する。

- 入力属性
  構成規則の左辺の LHS クラスが持つメッセージの動的相続属性と構成規則の右辺の LHS クラスが持つメッセージの動的合成属性

- 出力属性
  構成規則の左辺の LHS クラスが持つメッセージの動的合成属性と構成規則の右辺の LHS クラスが持つメッセージの動的相続属性

動的意味規則では、以下の記述を行う。

- 入力属性の動的合成属性の値を決定する式
- 出力属性の動的相続属性の値を決定する式
- 動的局所属性の値を決定する式
- 動的局所属性の宣言
- 固有属性の値を決定する式

このうち、上の2つは必ず記述されていなければならない。
動的局所属性は予約語 local を用いて、

```
local  <型名> <属性名> ;
```

と宣言する。
固有属性の値を定義するとき、左辺に現れる固有属性は右辺に現れる固有属性と別のものであることを強調するために、次のように予約語 new を付加する。

```
( new X ) = <式> ;
```

ここで、X は構成規則の左辺を含む任意の固有属性である。
静的属性や動的属性の値によって、メッセージを制御するためにメッセージのガードの記述を行う。メッセージのガードとは条件式であり、予約語 case を用いて以下のように記述する。

```
<入力メッセージリスト>
  case <条件 1>
    ⇒ <出力メッセージリスト 1>
    { <動的意味規則 1> }
```

```
case <条件 2>
  ⇒ <出力メッセージリスト 2>
  { <動的意味規則 2> }
 ⋮
otherwise
  ⇒ <出力メッセージリスト>
  { <動的意味規則> }
```

条件は上から順に評価され、真となる条件が現れたところでその条件に対応する出力メッセージの送出と動的意味規則の評価が行われる。予約語 otherwise は常に真になるガードであり、otherwise より上のガードが全て偽の場合に有効になる。

3.2.5 属性と式

属性の値を決定する式は次の形式で記述される。

```
<属性> = <式> ;
```

属性は全て型づけされており、属性と式は同じ型でなければならない。

属性

属性の値を定義、参照する際の記述方法は、属性の種類によって異なる。

複合固有属性	(new X$n) , X$n
葉固有属性	(new a) , a
静的相続属性	X$n.a
静的合成属性	X$n.a
静的局所属性	a
動的相続属性	X$n.a
動的合成属性	X$n.a
動的局所属性	a

ここで、X は LHS クラス名、a は属性名、n は構成規則中に同じ LHS クラスが複数ある場合の左からの番号 ($n > 0$) とする。$n = 1$ のとき "$n" は省略できる。

静的属性や動的属性が複合データ型である場合、属性が持つオブジェクトのルートノードに相続属性があってはならない。オブジェクト内の属性も評価され、ルートノードの合成属性は属性名にさらに ".attr_name" を付けて参照することができる。

静的局所属性は宣言した静的意味規則で値を定義し、その RHS クラスの静的意味規則と動的意味規則で参照できる。動的局所属性は宣言した動的意味規則でのみ定義・参照できる。

式

式は属性の値を定義するものである。式には以下のような種類がある。

- 定数
- 属性名
- 単項演算子 ⟨式⟩
- ⟨式⟩ 2 項演算子 ⟨式⟩
- (⟨式⟩)
- ⟨式⟩ ? ⟨式⟩ : ⟨式⟩
- with 式
- 関数
- コンストラクタ

定数

各原始データ型の定数を表記する方法は以下の通り。

1. 整数型
   整数型の定数は −1, 0, +10 のように整数を 10 進数と符号で表記する。
2. 実数型
   実数型の定数は 3.14 のように整数と小数点で表記する。
3. 文字型
   文字型の定数は ’ ’ で文字をくくって表記する。文字 ’ は ’\’’、文字 \ は ’\\’ で表す。表 3.1 に示す特殊文字も ’ ’ でくくって使用することができる。
4. 文字列型
   文字列定数は " " で文字列をくくって表記する。文字列中の " は \"、\ は \\ で表す。表 3.1 に示す特殊文字も " " 内に含めることができる。
5. 論理型
   true, false が論理型の定数である。

\t	タブ (TAB)
\n	改行 (LF)
\r	復帰 (CR)
\f	改頁 (FF)
\ooo	3 桁の 8 進数 ooo に相当するコードを持つ文字
\xhh	2 桁の 16 進数 hh に相当するコードを持つ文字 (a ～ f は小文字でも大文字でもよい)
\'	' 自身
\"	" 自身
\\	\ 自身

表 3.1: OSL における特殊文字

演算子

用意されている演算子を以下に示す。

- 単項演算子
  - 参照演算子 (*)
  - 名前演算子 (&)
  - 単項プラス演算子 (+)
  - 単項マイナス演算子 (-)
  - 論理 NOT 演算子 (not)
- 2 項演算子
  - 累乗 (exp)
  - 乗算 (*)
  - 除算 (/)
  - 剰余 (mod)
  - 加算 (+)
  - 減算 (-)
  - 小なり (<)
  - 大なり (>)
  - 以下 (<=)

- 以上 (>=)
- 等価 (==)
- 不等価 (!=)
- 論理演算子 (and, or, xor, imply)

with 式

with 式は任意の式に対してパターンマッチングを行い、条件分岐を行う構文である。

```
with ( <式> ) {
  <パターン 1> : <式 1> ,
  <パターン 2> : <式 2> ,
       ⋮
  <パターン n> : <式 n>
}
```

with 式の値は最初にマッチするパターンに対応する式の値である。パターンには以下の種類がある。

- 定数
- パターン変数
- * (必ずマッチするパターン)
- 固有属性リストがパターンのリストとなっているコンストラクタ

( ) 中の式がどのパターンにもマッチしないということがあってはならない。

関数

関数は属性の値を計算するために定義・使用する。

```
<型名> function <関数名> ( <引数リスト> )
{
  <式>
}
```

引数リストは

<型名> <引数名>

の組をカンマ(",") で区切って並べる。
式は型名と同じ型の式でなければならない。

コンストラクタ

動的属性評価ではオブジェクトを生成することができ、以下のように記述する。

<RHS クラス名> [<固有属性リスト>]

これは、固有属性リストで指定されたオブジェクトを固有属性として持つRHS クラスのインスタンスを生成することを意味する。この記述によって生成されたオブジェクトの型はその RHS クラスををまとめている LHS クラスである。

3.2.6 オブジェクトと名前

複合固有属性以外の属性は、あるオブジェクトを示す名前を保持することができる。オブジェクトからオブジェクトへのコピーではオブジェクトの中身が全てコピーされ、中身は同じであるがオブジェクトとしては区別される。名前を使うことによって、複数の属性が同じオブジェクトを参照できる。また、木構造上では離れた位置にあるオブジェクトに対してメッセージを送ることができる。静的評価中は名前が指すオブジェクトには全くアクセスできない。動的評価でも名前を持つ属性からその名前が示すオブジェクト内の属性を直接取り出すことはできず、属性内の属性に対するアクセスはメッセージによって行う。動的評価では名前が指すオブジェクトの木構造を参照することはでき、with 式で名前を使うことは可能である。

オブジェクトの名前を持つ属性の型名は、LHS クラス名 * 型である。名前からオブジェクトへの変換には * を属性名の前に置き、オブジェクトから名前への変換は & を属性名の前に置くことによって表記する。左辺のオブジェクトに & を付けることでオブジェクトアイデンティティを保ったままオブジェクトをコピーすることができる。例えば、

```
X  obj
X* name
```

の時、obj と name の間で型変換を伴う記述は以下の3通りある。

`name = &obj;`	name に obj の名前を入れる。
`obj = *name;`	name が示すオブジェクトを obj にコピーする。
`(new obj) = *name;`	obj が固有属性の場合。
`(new &obj) = name;`	obj の名前を name にする。

複数の属性が同じオブジェクトを持つことはできない。つまり、上の4番めの記述によって、オブジェクトアイデンティティを保ったコピーをした場合、以前そのオブジェクトを保持していた属性は他のオブジェクトに変更されなければならない。

3.2.7 外部クラス

ファイル I/O などの OOAG では記述できない操作や、複雑なデータ構造を持ったデータを扱いたい場合、外部クラスを使うことで実現できる。

外部クラスは C++ で記述し、OOAG 側では固有属性、静的属性を持たないノードに見える。外部クラス内のメンバ変数へはメッセージによってアクセスする。このメッセージは C++ でメンバ関数として定義されていなければならない。メッセージを受けるとそれに対応する C++ のメンバ関数が呼び出される。このメッセージは必ず1つの出力属性を持っており、メンバ関数の返値がメッセージの出力属性の値となる。

外部クラスではトランジションを定義することはできず、メッセージを出力できない。

外部クラスの宣言は予約語 external で行う。

```
external <外部クラス名>
  <メッセージリスト>
```

メッセージリストでは入力メッセージのみを宣言する。

3.3 OSL による記述例

図3.2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3.3の様な構造を持つ。

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（図3.2: 17 ～ 22行目）。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（図3.2: 24 ～ 29行目）。

図3.3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る舞いを示している。

```
1  /* P の静的仕様記述 */
2  class P( | EXE_CODE executable )
3      → Program[ M( | OBJ_CODE obj_code ),
4                 M( | OBJ_CODE obj_code ) ]
5      {
6         P.executable = link( M$1.obj_code, M$2.obj_code );
7      }

8  /* M の静的仕様記述 */
9  class M( | OBJ_CODE obj_code )
10     → Module[ ROBJ( | ), CACHE cache, OBJ_CODE current ]
11     {
12        M.obj_code = current ;
13     }

14     /* M の動的仕様記述 */
15     M:retrieve( VER revision, BOOL update | OBJ_CODE req_obj )
16        case (in_cache(cache, revision))
17          ⇒
18          {
19            M.req_obj = look_up( cache, M.revision ) ;
20            (new current) =
21                 if ( M.update == true, M.req_obj, current ) ;
22          }
23        otherwise
24          ⇒ ROBJ:retrieve( VER revision | STR source )
25          {
26            M.req_obj = compile( ROBJ.source );
27            (new cache) = lru_update( cache, M.revision, M.req_obj );
28            (new current) = if ( M.update == true, M.req_obj, current ) ;
29          }
```

図 3.2: OSL 記述の例

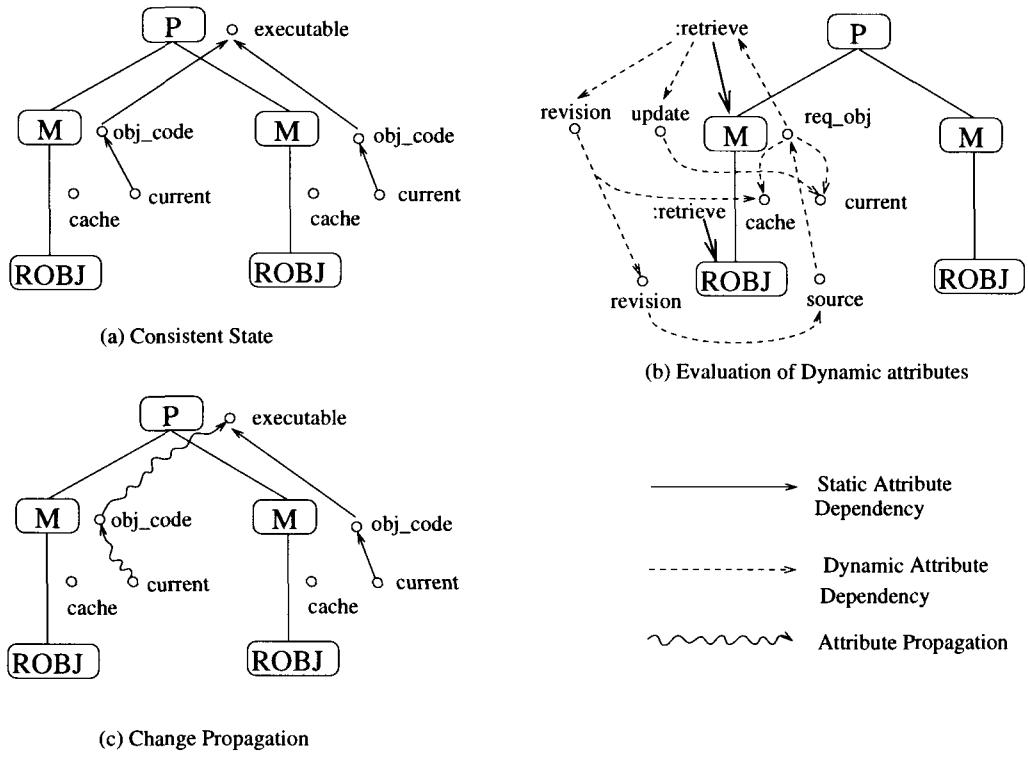


図 3.3: オブジェクト P の動作

3.4 OOAG によるリポジトリ記述関連の研究

ここでは、ここ数年片山研究室で行なわれてきたOOAG によるリポジトリ記述に関連する研究の内容を簡単にまとめる。

3.4.1 バージョン管理

本論文での OOAG は、自由なリポジトリの記述システムであり、OOAG 自体に特定の戦略を持つバージョン管理システムが組み込まれていたりはしない。リポジトリ構造の設計と、そこでのソフトウェアオブジェクトのバージョン管理メカニズムは、OOAG により明示的に記述される必要がある（ライブラリ的に使用できるようにする必要はあるだろう）。

属性文法がバージョン管理システムの記述に応用できることの確認として、属性文法によるファイルのバージョン管理システムの記述が行なわれている [ISK90, 今泉 92]。

Synthesizer Generator と OOAG の両方で実際のシステム構築が行なわれている。

3.4.2 UNIX ファイルシステムの記述

リポジトリの原型とも言うべきファイルシステム型のリポジトリ構築が、OOAG により可能であることの確認として、UNIX のファイルシステムの動作をシミュレートする OOAG 記述の実験が行なわれた [MI94, 今泉 95]。この記述実験では、構造指向システムを記述するための実行可能な仕様記述言語として、OOAG の記述言語 OSL が使用できることの確認でもあった。UNIX のファイルシステムの仕様が、ファイルシステムの構造も含めて OOAG で記述でき、動作確認が行なえたことで、OSL 記述言語の記述能力が示されたわけである。

ファイルシステム型のリポジトリが確実に OOAG で記述可能で、リポジトリとして動作させることが出来ることが確認されたことの意味は大きい。最低でもファイルシステム程度のリポジトリ構築が行なえることが判明したからである。

3.4.3 ソフトウェア開発環境

本論文では、OOAG をソフトウェアリポジトリの生成系としてとらえているが、ソフトウェアリポジトリだけではなく、ソフトウェア開発環境全体の仕様作成から環境生成のための道具として利用する研究も行なわれている [GISK93, 権藤 95]。

ソフトウェア開発環境の構築システムに関して、cut & paste モデルなどの従来の属性文法型システムとの比較を行ない、OOAG の優れている点を示している。

第4章　MAGE の設計と実現

この章では、MAGE に関して説明する。まず、4.1節で MAGE の概要を説明する。4.2節では、MAGE で実現している OOAG の制限などについて説明している。4.3節で、MAGE で生成したリポジトリの実行を行なう、属性評価器の設計と実現を述べる。ここでは、OOAG の属性評価法の簡単な説明と、これを高速に実行するために行なった特殊な手法の説明を主とする。最後に4.4節で、実現した属性評価器の性能評価を行なっている。

4.1　MAGE の概要

MAGE は、オブジェクト指向データベース管理システム（OODBMS）を利用して、オブジェクト指向属性文法 OOAG でリポジトリとなるオブジェクト指向データベースを機械的に生成するシステムである。具体的には OSL 言語によるソフトウェアリポジトリ記述を、OODBMS の C++ 言語インタフェースのコードに変換することにより、リポジトリシステムをデータベース・クライアントとして実現するものである。

まず、最初に MAGE のアーキテクチャの概要と、リポジトリ開発環境の構成について説明する。

4.1.1　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 を開発した。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、MAGE は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。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。

我々は、オブジェクト指向データベース管理システムの一つである ObjectStore [Obj93] を用いて MAGE の開発を行なった。ObjectStore のもつ 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

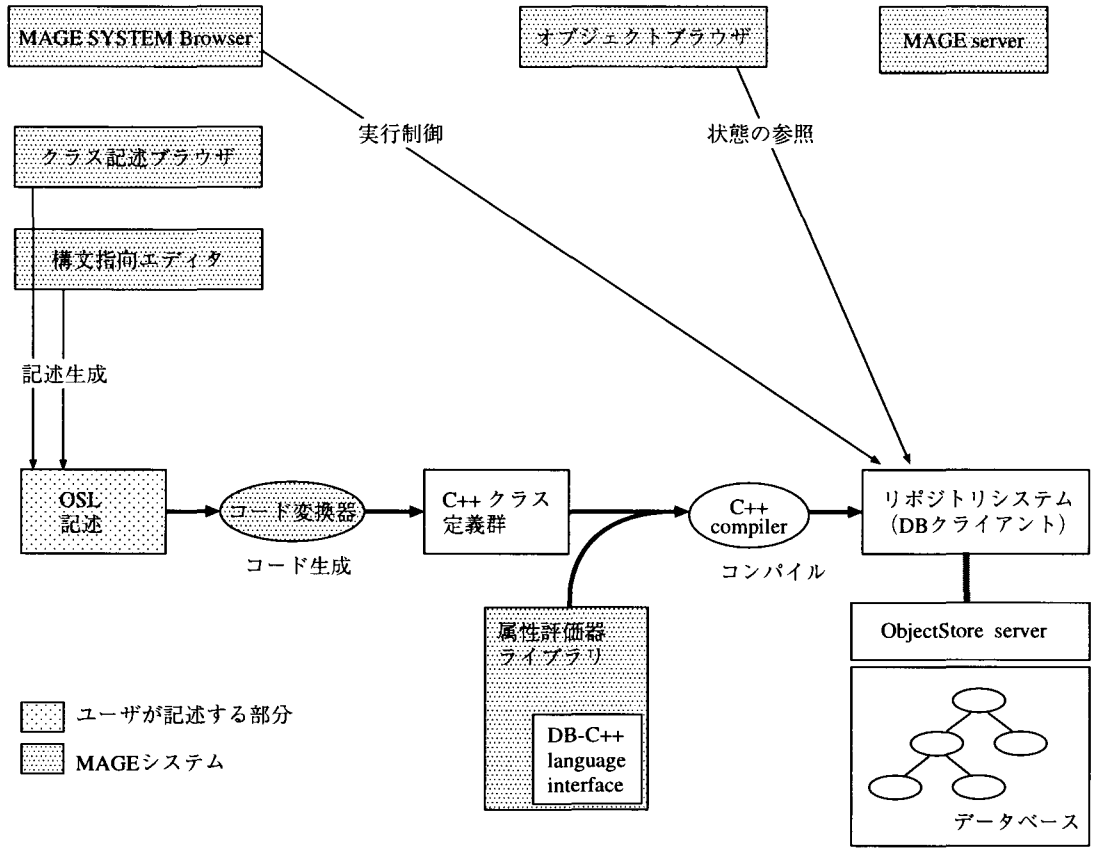


図 4.1: MAGE でのリポジトリシステムの生成方法

いる。そして、ObjectStore で使用されているこれらの技術は、我々が OOAG の効率的な実行のために行な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 4.3 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 4.1 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 による記述を行な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出来る。

4.1.2 リポジトリの開発環境の構成

前節で、簡単に MAGE の概要について説明したが、ここではリポジトリの開発を援助する開発環境の構成要素についてもう少し詳しく説明する。以下、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 のシステム構成は、図 4.1 で示した通りで、以下各ツールの説明を行なう。

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。ユーザのリポジトリ開発セッションは、このシステムブラウザの起動から始まる。

クラス記述ブラウザ / OSL エディタ

言語 OSL は、型や関数の導入など、拡張が進められるにつれ、複雑な構造を持つようになった。MAGE では、OSL の複雑さによるリポジトリ開発の効率の悪化を克服するために、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。これらは、MAGE プロトコル（後述）を用いて互いに情報交換しているため、エディタでの編集作業と同時にクラスブラウザの表示内容も更新される。

OSL エディタは、CSG 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ない、エラーを逐次表示する機能を持つ。これによりコンパイル・編集のくり返しを減らし、効率的なプログラム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、プログラム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。また、エディタと連係していて、ユーザは目的の位置をクラスブラウザで素早く指定することによりエディタでの編集に移ることができる。

クラスブラウザが提供する情報には、記述ファイルに関する情報、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が存在する。

オブジェクトブラウザ

オブジェクトブラウザは、コンパイルされた OSL プログラム（これを MAGE ではプロセッサと呼ぶ）と通信して、その実行制御や状態を表示する。属性評価のフェーズ毎にプロセッサの実行を制御でき、またこのときに属性を用いて制御の条件を与えることができるので、OSL プログラム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ので、オブジェ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、その属性値も参照できる。

オブジェクトブラウザが提供する情報には、実行中のプロセッサのオブジェクト構造、オブジェクトの静的・動的属性の値、現在の評価フェーズ、メッセージの伝達状況などが存在する。

プロセッサ/トランスレータ

プロセッサは、OSL 記述から生成されたリポジトリシステムで、オブジェクト指向データベース管理システム ObjectStore のクライアントシステムである。プロセッサには、属性評価器が組み込まれ、OSL 記述の内容にしたがってリポジトリのオブジェクト操作を行なう。

エディタで編集した OSL プログラムは、トランスレータを用いて C++ のクラス定義に変換し、これをコンパイルすることにより実行可能なプロセッサを得る。トランスレータは、OSL によるクラス定義を、属性評価のためのライブラリとして記述されている、Node クラスを継承する C++ のクラス階層に変換する。

プロセッサは単体ではユーザとのインタラクションができないので、ユーザはオブジェクトブラウザを用いてプロセッサの制御を行なう。

MAGE サーバ / MAGE プロトコル

MAGE サーバは、MAGE のツール間で交信される MAGE プロトコルの配送を主な目的とする。MAGE プロトコルは TCP/IP 上に設計されており、MAGE サーバを介してネットワーク上の異なるホストで動作するツール間の通信を可能にしている。MAGE プロトコルによる通信のためのモジュールは、ライブラリとして準備されているため、このプロトコルに従う新たなツール開発は容易になっている。

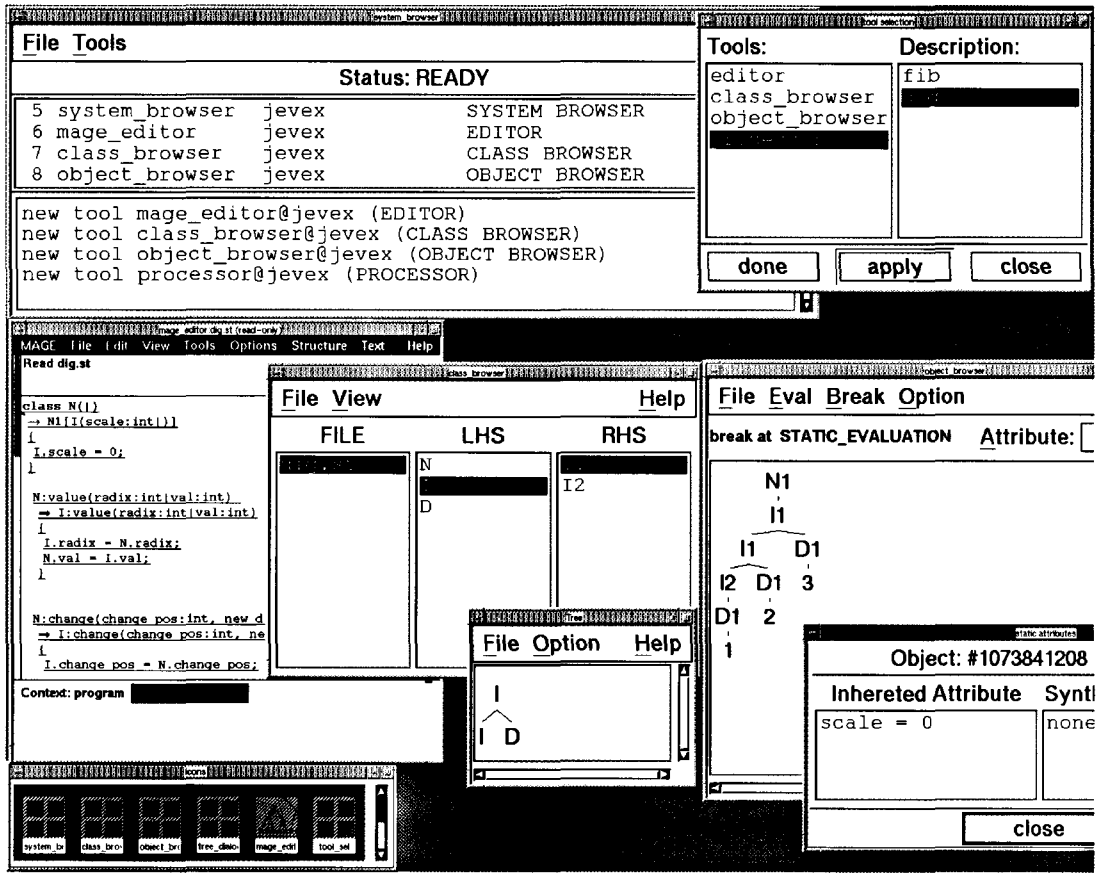


図 4.2: MAGE の実行画面のスナップショット

4.2 MAGE で実現する OOAG

計算モデル OOAG は、篠田により提案され [Shi89]、その後権藤による動的計算に関する計算モデルの拡張と、xlisp によるプロトタイプ属性評価器の構築が行なわれ [権藤 91]、そして本論文での属性評価器に至っている。それぞれ「第 1 世代」「第 2 世代」「第 3 世代」と呼ぶことにする[1]。この間、計算モデル自体の拡張、及び評価器構築の上での実現するモデルの制限などが行なわれており、それぞれの扱う OOAG は少しずつ異なっている。

ここでは、本論文での属性評価器で実現している OOAG に関して、特に重要な以下の 3 点について説明する。

- オブジェクトの名前参照に関して
- メッセージの処理に関して
- 生成後のシステムの実行形式に関して

4.2.1 オブジェクトの名前参照

OOAG では、基本的に木構造（階層）を持つ対象のみを扱い、グラフ構造を持つような対象を扱うことが出来ない。しかし、これでは現実の問題に柔軟に対応することが難しい。特に、ソフトウェアリポジトリにおいては、ソフトウェアオブジェクトの参照関係が複雑になりがちであり、この点は大きな欠点になる可能性を含んでいる。そこで、木構造の離れた場所に存在するオブジェクトに対する参照を可能にするための仕組みが「オブジェクトの名前参照」である。

OOAG では、名前参照されたオブジェクトを「名前つきオブジェクト」と呼び、権藤による拡張（第 2 世代）で実現された。この名前つきオブジェクトのセマンティクスを以下に示す。

権藤バージョンでの名前つきオブジェクトのセマンティクス:

- 参照オブジェクトにメッセージを送信可能。受信は不可。
- 参照先のオブジェクトの静的属性の値が参照可能。

しかし、この第 2 世代で導入された名前つきオブジェクトには、以下のような欠点が存在した。

第 2 世代の OOAG の名前つきオブジェクトにおける欠点:

- オブジェクトに複数メッセージの到着の可能性が生じるが、ロック機構を持たない。この場合、どのメッセージが有効になりどれが無視されるのかは未定義であり、それを制御することも出来ない。

[1] これに関しては、6.1 節でもう少し詳しく記述を行なった

- 静的属性が名前参照により循環することが起こり得る。
- 静的属性評価器も複雑になる。これによりさらに静的属性評価器の実行時処理が増大し、名前参照を使用しない場合の処理時間にも悪い影響を及ぼす。

現在得られているバージョンでは、オブジェクトの名前参照の機能の実現は見送っている。これは、上記問題に対する良い解決法が見つかっていないからである。しかし、オブジェクトの参照機能はアプリケーション記述には必要な機能であり、現在も良い導入法を模索中である。

4.2.2 メッセージの制御と動的属性の評価

第2世代の OOAG の動的計算の評価には、以下が問題であった。

動的計算の高速化に関する問題:

属性の評価と、メッセージ制御が絡み合い、動的計算の仕組み自体が複雑になる。このため、効率の良い動的属性評価器を構成することが出来ない。

本バージョン（第3世代）では、このメッセージの制御と動的属性の評価の機構を簡略化するために、第2世代の OOAG の動的計算に以下のような制限を設けた。

本評価器において設けた制限:

メッセージは、その出力属性の値がすべて決定されるまで出力されない。

この制限では、記述できる属性文法に影響を与えることはない。たとえば、第2世代のバージョンで評価できる 図4.3 の記述は、オブジェクト `X` において、メッセージ `Y:m` の属性 `Y.a` は、メッセージ `Z:m` の属性 `Z.c` の値が得られないと値が決まらない。同じく、メッセージ `Z:m` の属性 `Z.a` は、メッセージ `Y:m` の属性 `Y.d` の値が得られないと値が決まらない。したがって、`Y:m` も `Z:m` も出力することが出来ず、本バージョンでは処理できなくなる。しかし、図4.4 のようにメッセージを分割して書き変えることで、`Y:m1`、`Z:m1` がそれぞれ出力され、`Y.d`、`Z.c` の値が決定し、その結果 `Y.a`、`Z.a` の値も評価できるので、`Y:m2`、`Z:m2` も出力されるようになり、同じ属性文法が本バージョンで処理できるようになる。

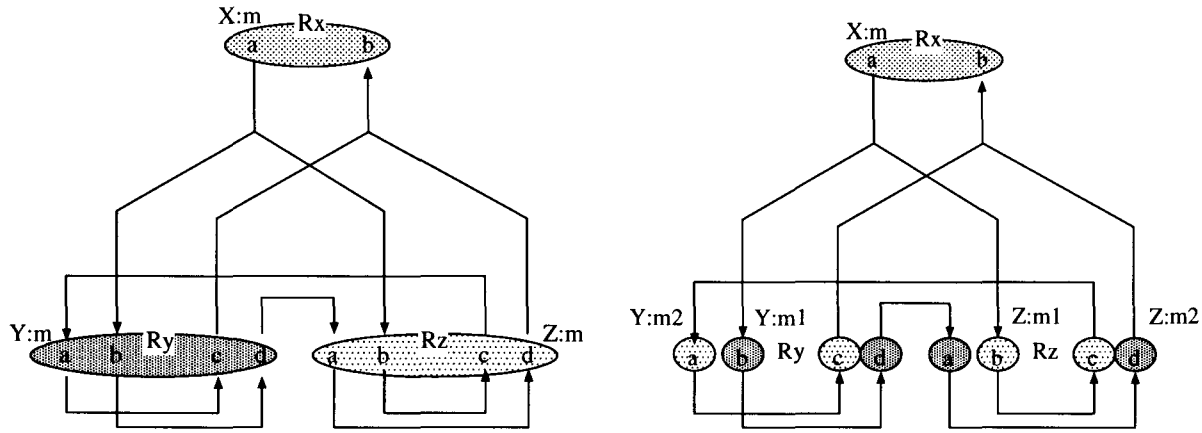
この制限により、記述者によりいっそう動的評価の戦略を指定させることになる。これは、属性文法記述の評価戦略とは関係なく属性どうしの関数的関係だけを記述するという利点をある意味で損ねることになるが、これはメッセージのパスを指定している段階で崩れているので、割り切り方の問題である。

```
1  class X(|) → Rx[ Y(|), Z(|) ]
2          { }
3    X:m(a|b) ⇒ Y:m(a,b|c,d), Z:m(a,b|c,d)
4          {
5                  Y.a = Z.c ;  Y.b = X.a ;
6                  Z.a = Y.d ;  Z.b = X.a ;
7                  X.b = f1(Y.c, Z.d) ;
8          }

9  class Y(|) → Ry[ ]
10         { }
11   Y:m(a,b|c,d) ⇒
12         {
13                 Y.c = Y.a ;  Y.d = Y.b ;
14         }

15 class Z(|) → Rz[ ]
16         { }
17   Z:m(a,b|c,d) ⇒
18         {
19                 Z.c = Z.b ;  Z.d = Z.a ;
20         }
```

図 4.3: 新処理系では評価できない動的記述例

```
1  class X(|) → Rx[ Y(|), Z(|) ]
2         { }
3    X:m(a|b) ⇒
4          Y:m1(b|d), Z:m1(b|c), Y:m2(a|c), Z:m2(a|d)
5          {
6                  Y.a = Z.c ;  Y.b = X.a ;
7                  Z.a = Y.d ;  Z.b = X.a ;
8                  X.b = f1(Y.c, Z.d) ;
9          }
```

図 4.4: 図 4.3 と同等の評価可能な動的記述例



(a) 分割前の状態　　(b) 分割後の状態

図 4.5: メッセージの分割問題

ここで、この問題についてもう少し詳しく考えて見る。図 4.5(a) が前者の方の記述を図示したもの、図 4.5(b) が後者の書き変えた記述を図示したものである。メッセージの処理単位ごとに色を塗り分けてある。この図を参照しながら考えると、このメッセージの分割は、ordered 属性文法 [Kas80] の文法判定におけるパーティション分割の問題と似ていることに気づく（図中では同じ色の属性がメッセージの単位)。パーティションは、属性評価のための訪問列の作成において、同一の訪問時に評価することが出来る属性の組で、ordered 属性文法の評価器構築では、この組を用いて順序づけを行なう。OOAG の動的計算では、メッセージの送信により訪問の候補が決まるので、「メッセージ ≈ 訪問」と考えると、出力メッセージの出力属性値が決定するもので組分けする作業とパーティション分割は同じ作業であるという予測がつく。したがって、まだ「メッセージの分割の機械化」「動的属性評価への ordered 属性文法評価器の構築」という可能性を残している。

4.2.3 生成したシステムの実行方式

第 2 世代の属性評価器は xlisp インタプリタで動作し、生成した評価器への命令も xlisp コマンドにより行なっていた。

本論文での属性評価器は、リポジトリにアクセスするデータベースクライアントのひとつとして生成される。具体的には、オブジェクト指向データベース管理システムを利用し、永続記憶管理機構を用いることにより、OOAG の属性つき木をオブジェクト指向データベース上に生成している。このため、リポジトリに対するコマンドの発行形式、およびリポジトリのシステム形態も全く異なったものとなっている。図 4.6(a)、図 4.6(b) が、それぞれのシステムの形態の模式図である。

両者のもっとも大きな違いは、属性評価器が組み込まれる位置である。権藤の属性評価器の場合は、リポジトリに対するコマンドは xlisp インタプリタで処理され、これにより属性評価器が動作してリポジトリを操作するととらえることが出来る。つまり、属性評価器はリポジトリに付属するものとして位置する（図 4.6(a)）。

これに対し、現在の属性評価器は、リポジトリのデータベースを操作する DBMS サーバのクライアントとして構築される。実際のリポジトリは、DBMS サーバの方で管理され、属性評価器はそのクライアントとしてリポジトリ・データベースを処理する。つまり、属性評価器はリポジトリとは離れて、それぞれのクライアントの方に内蔵される形式となる（図 4.6(b)）。ひとつひとつのクライアントごとに属性評価器が組み込まれ、それぞれが DBMS サーバのリポジトリにアクセスできるので、ここでトランザクションとロッキングの問題が生じる。

最終的には図 4.6(c) のように、DBMS サーバのリポジトリにアクセスするクライアントである属性評価器を 1 つだけにし、これにリポジトリに対するコマンドを処理するインタプリタを内蔵し、これがリポジトリを管理するサーバとして動作するようにする必要があると考えている。

4.3 属性評価器の設計と実現

この節では、OSL 言語によるリポジトリ仕様から生成したリポジトリの、オブジェクト操作に関する計算を実行する属性評価器の設計と実現、および OSL 言語から C++ クラス階層への変換に関して説明する。属性評価器の設計は、生成したリポジトリシステムのオブジェクト操作の実行速度に影響を及ぼす、非常に重要な要素であり、少しでも効率の良い実現方法を探ることが、OOAG によるリポジトリの自動生成システムの実用性に関するキーポイントである。

また、OOAG の属性評価は特殊であるので、以下の順に説明を行なう。

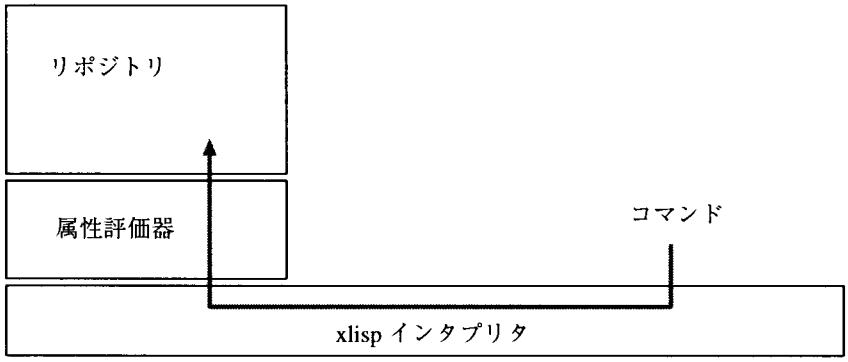
（**4.3.1 節**）NODE オブジェクトという用語

（**4.3.2 節**）属性評価アルゴリズムの概要

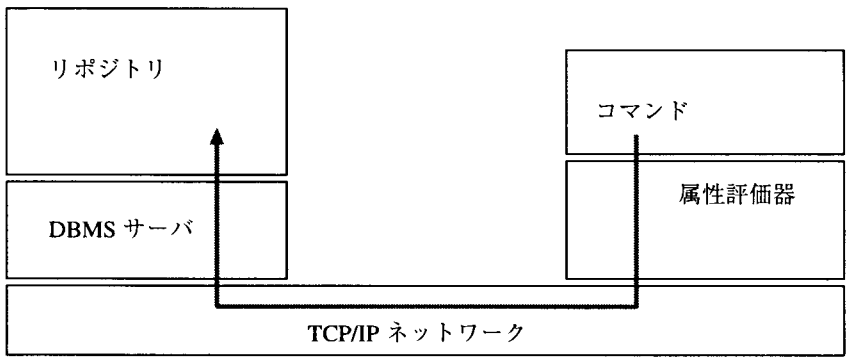
（**4.3.3 節**）動的評価器の設計の概略

（**4.3.4 節**）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

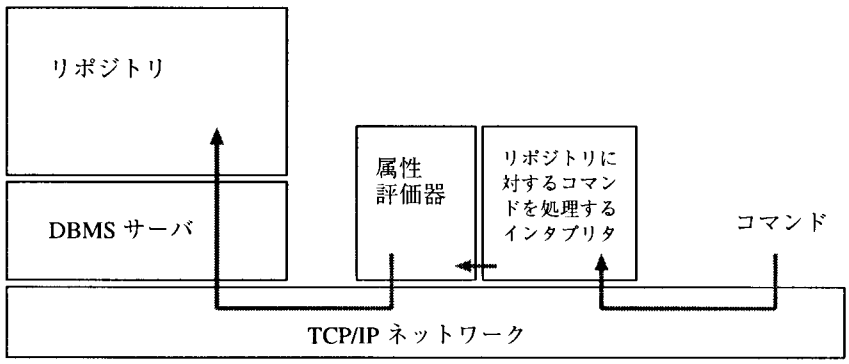
（**4.3.5 節**）OSL 記述から C++ クラス階層への変換システム



(a) 権藤の属性評価器の実行形態



(b) 現在の属性評価器の実行形態



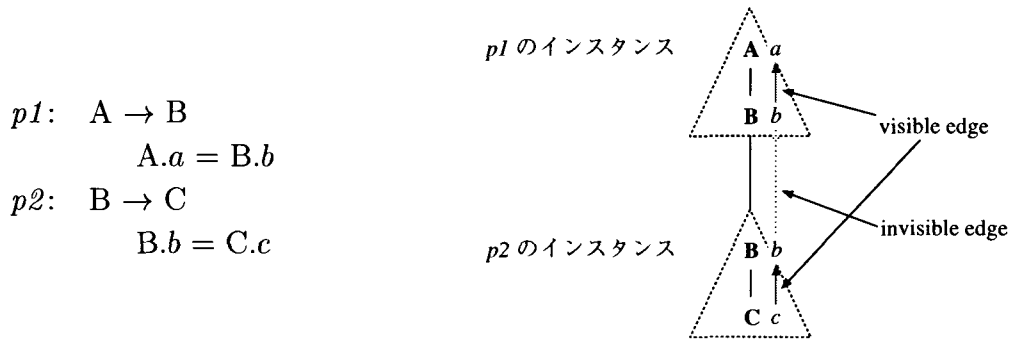
(c) 最終的な実行形態

図 4.6: 生成したシステムの実行の形態

4.3.1 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 4.7(a) の文法規則から生成される木の一部（図 4.7(b)）を例に取る。



(a) 文法の例　　(b) 生成される木のサンプル

図 4.7: 木のノードの分割の例

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われない。しかし 4.3.2 節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 オブジェクト毎に分割するため、一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、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、属性文法の意味計算の上で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われる 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。

4.3.2 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なう静的属性評価と、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な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 [SK90a] において *Locally Controlled Distributed Incremental Attribute Evaluation* アルゴリズム（LCDIA アルゴリズム）が与えられている。動的属性の評価に関しては、権藤のプロトタイプとは異なり、4.2 節で述べたとおり、記述に制限を設けたサブセットで実現している。以降、それぞれの属性評価器について説明する。

静的属性評価（LCDIA アルゴリズム）

LCDIA アルゴリズムの基本的なアイデアは、実行時の属性の評価に用いる属性依存を表わ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、明示的には現われない見えない依存枝が存在することになる（4.3.1 節の図を参照）。

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づけ、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[2]。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 の値が変化したとすると、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動的属性評価器から送られると考える。

PROPAGATE シグナルハンドラは、以下のことを行なう。

- 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

[2] 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。

```
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 をひとつ選択し、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 4.8: LCDIA アルゴリズム – PROPAGATE メッセージハンドラ

```
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 をひとつ選択し、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 4.9: LCDIA アルゴリズム – RELAX メッセージハンドラ

M の独立した頂点は、その頂点に結びつく属性が決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つ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わす。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4.8、図4.9に示す。

メッセージ制御と動的属性評価器

OOAG の計算モデルでは、メッセージの処理と動的属性の評価は別のフェーズである(図3.1参照)。しかし、権藤のプロトタイプ処理系では、動的属性評価器は「ペトリネットモデルのメッセージ制御」と「動的属性の評価」を同時に進行させる複雑なものとなっていた。これは、メッセージのガード（3.2.4節参照）のために、式中で使用されている属性値が必要なためである。

4.2.2節ですでに述べたが、今回の実現でこの「メッセージ処理」と「動的属性の評価」の同時進行による動的属性評価器の複雑化の対策として、権藤の OOAG の動的計算モデルに対してメッセージの送信に対する制限を設けた。効率的なアルゴリズムを提案することはまだ出来ていないが、これにより記述できる内容に関する影響を低く抑えたまま、動的属性評価器を簡単化するところまでは成功した。

4.3.3 動的評価器の構築

OOAG の静的計算を実現する評価器の C++ 言語化は、文献 [萩原 93, 萩原 94] においてすでに大まかな設計と実現が得られているため、ここでは設計の説明は省略する。これは、データベース生成を意識したものではないが、ほぼそのまま利用可能である。ここでは、そこにおいて行なわれていない動的計算を実現する、動的評価器の C++ 言語による設計を簡単に説明する。

静的計算、動的計算を含めた高速処理のためのテクニックについては、4.3.4節で説明されている。

動的評価器構築のための準備

```
1  class  lhs_name ( sinh0 , ... | ssyn0 ,...)
2    → rhs_name [ obj0( | ), ... ]
3    {
4      ...
5    }
6      in_message0 ( in0 , in1 , ... | out0 , out1 , ...), ...
7        ⇒ out_message0 ( in0 , in1 , ... | out0 , out1 , ...), ...
8      {
9        eq0 ;
10       ...
11     }
```

トランジション: 入力メッセージ列と出力メッセージ列の組である。
上記の例では、$\text{in\_message}_0(...|...), ... \Rightarrow \text{out\_message}_0(...|...), ...$ の部分に対応する。動的属性評価器内部では、この組に順に番号付けし識別する。case 文が使用された場合は、それぞれの条件ごとに別のトランジションとして数える。ただし、入力メッセージ領域は case に属するすべてのトランジションで共有する。

トランジションの識別: 動的評価器内では、トランジションは Transition オブジェクトの配列で管理される。各 RHS クラスで定義された順に 0、1、2 ... と番号付けされ、これを配列のインデックスに使用する。

メッセージの識別: トランジションと同様に、メッセージは Message オブジェクトの配列で管理される。入力メッセージも出力メッセージも、それぞれ最初のものから 0、1、2... と番号付けされ、これをインデックスとして識別される。特定のメッセージを示すには、どのトランジションか、入力メッセージ/出力メッセージの種別も示す必要がある。

動的属性の識別: 動的属性は、Message オブジェクト内で管理される。入力属性も出力属性もそれぞれ最初のものから 0、1、2 ... と番号付けされる。特定の動的属性を示すには、どのトランジションか、どのメッセージかも示す必要がある。

動的処理関連のクラス

動的評価器関連の構成部品を以下に示し、簡単な説明を与える。

Transition クラス: 入力メッセージリスト、出力メッセージリスト、トランジション内

の動的属性の依存グラフ、処理済メッセージ集合を保持する。以下、それぞれについて説明する。

入力メッセージリスト、出力メッセージリスト: このトランジションの Message オブジェクトの管理を行なう。それぞれ OSL 記述での出現順に番号付けされ、識別される。

トランジション内の動的属性の依存グラフ: このトランジションでの動的属性の依存関係を保持する。トランジション・オブジェクトの初期化時に、グラフ作成を行なう。この作成のためのコードは、OSL 記述からトランスレータで出力する。

処理済メッセージ集合: 出力メッセージの出力属性値がすべて評価されると、そのメッセージが出力されるが、出力先のオブジェクトでの処理が終わって、そのメッセージの入力属性値もすべて評価されいる状態のメッセージを、処理済みメッセージと呼ぶ。この処理済みのメッセージをマークするための集合。

eval 関数: 動的評価器のスケジューラからディスパッチされる関数で、動的属性評価の戦略をインプリメントしている。動的属性評価を行なう本体である。現在の動的属性評価は、メッセージと動的属性の状態を参照しながら、メッセージに基づき木を訪問して属性評価を進める単純なもので、トランスレート時の静的解析など特別な処理は行なっていない。

Message クラス: 入力属性リスト、出力属性リスト、送信側オブジェクト、受信側オブジェクト、送信側オブジェクトでのトランジション番号、受信側オブジェクトでのトランジション番号を保持する。

Node クラス: Node クラスでは、動的評価器に関連して以下のものを管理している。

トランジション配列 この Node オブジェクトで宣言されたトランジション・オブジェクトの配列

受信メッセージ集合 受信された入力メッセージ・オブジェクトを保持する。

ガード関数: case 文の場合、条件式に適合する場合の現在のトランジションを採用し、適合しない場合トランジション番号をインクリメントして fireScheduler に登録する。この関数は、Transition オブジェクトのグラフにダミーノードとして登録される。RHS クラスのフレンド関数として、トランスレータでコード生成する。

属性評価関数: 出力属性の値を計算する関数。OSL の動的仕様記述中の属性関係式ごとに生成される。静的仕様記述の実装と同様、属性値の評価後に評価済みフラグをセットする。静的属性の場合と異なり、以前の値と同じかどうかのチェックは必要ない。この関数は、ターゲットの Attribute オブジェクトに登録される。RHS クラスのフレンド関数として、トランスレータでコード生成する。

```
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
        ⋮                          // 動的属性の属性依存グラフの作成
}
```

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

トランスレータでコード生成

図 4.10: ATTRIBUTE クラスとその初期化のコードの断片

4.3.4 高速な属性評価器の設計

OOAG の属性の評価を高速に行なうためには、特に以下の点に注意する必要があることが、第2世代の MAGE のプロトタイプ属性評価器を分析することにより判明した。

- 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。特に、LCDIA アルゴリズムでは、おなじ属性インスタンスを表わす ATTRIBUTE オブジェクトが、親子2つのオブジェクトに生成されるので、処理が複雑化する。

以下、それぞれの項目について我々が取った方法を説明する。

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なうことが出来る (図4.10: `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なうことが出来るのだが、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 4.10: `setup_attribute_backpointer()` メソッド)。

依存グラフの枝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

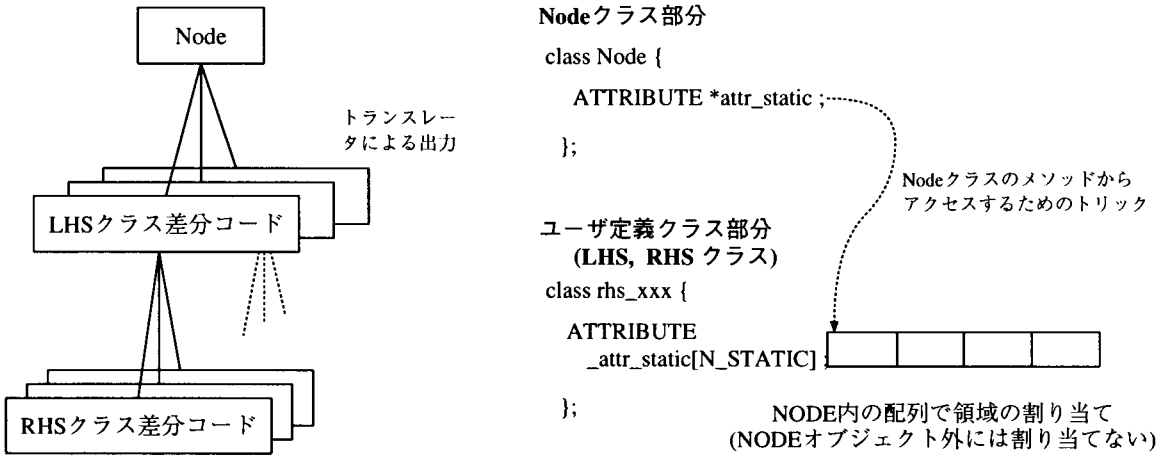
(2) 指定した頂点だけからなる到達可能性を示す部分グラフの抽出

(3) グラフの合成

これらの操作を高速に行な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) の操作を高速に行なえることが重要で、これが出来れば (2) の操作も高速に行なえるようになる。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。これにより、特定の頂点に依存する頂点はすぐにわかるため、到達可能性を示す部分グラフ抽出が高速に得られる。

ただし、このままでは逆の操作 (ある頂点に入力する頂点を求める操作) には、グラフの全探索が必要である。ただし、属性評価器では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。

4.3.1 節での説明の通り、近隣の NODE オブジェクトでは、同じ属性を表わすためのフィールドを分散して保有している。このため、同じ属性を表わす頂点を 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上で操作を行なう必要がある。たとえば、オブジェクト X において、オブジェクト X 内のグラフ G_X



(a) クラス階層　　(b) 属性レコードの参照

図 4.11: NODE クラス群の構造

とオブジェクト Y 内のグラフ G_Y の操作を行なうには、オブジェクト Y から G_Y を入手して、G_Y の各頂点をオブジェクト X 内の頂点に変換した上で操作を行なう必要がある。このドメインを合わせる操作については「**NODE オブジェクト間での ATTRIBUTE 参照のためのリンクポインタ**」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「**ATTRIBUTE オブジェクトの管理方法と依存グラフの構造**」で説明するように、グラフ頂点が直接 ATTRIBUTE オブジェクトを参照するような設計を行なっているからである。

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 4.11(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り当てを行なう。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り当てを行なうことにより、領域管理処理のオー

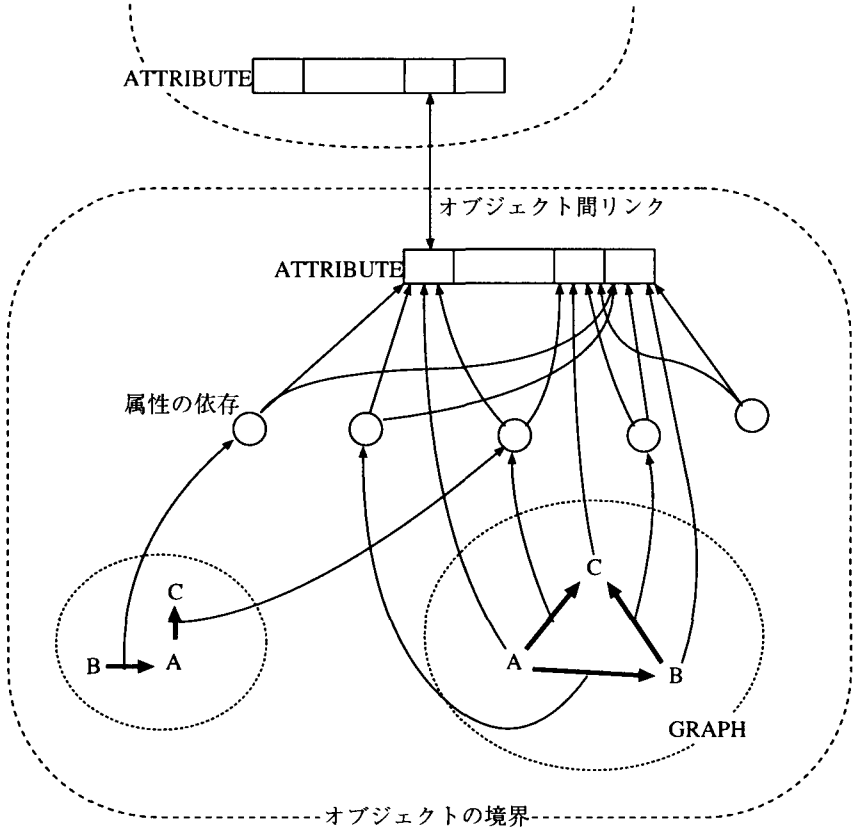


図 4.12: グラフ構造の模式図

バーヘッドを減らすためである。ただし、ここでひと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出来ない。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 4.11(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。

グラフ操作のために枝の追加・削除が頻繁に行なわれる。このため、この周りの領域管理を効率よく行なうことが重要。今回の設計は以下の通り。

- グラフの頂点として ATTRIBUTE オブジェクトへのポインタを採用。前述の通り、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得る。また、どのグラフに属さないこともあり得る。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なう。

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。

動的属性評価器の関連事項

動的属性評価器の設計の要点はメモリ管理の問題で、動的計算時にだけ必要となる管理領域をどのように管理するかである。OOAG の計算モデル的には、メッセージの伝達とともに一時的に意味規則（したがって属性も）割り当てられることになっているが、今回の実現では、一時的な割り当てのためのメモリ管理のオーバーヘッドを嫌ったため、以下の戦略を採った。

- メッセージのインスタンスを、あらかじめ作成する。
- これに伴い、動的属性のインスタンスもあらかじめ作成する。

これらのメッセージや動的属性のインスタンスの作成は、NODE オブジェクトの作成時に同時に行なう。しかし、部分木の置換操作により、メッセージの NODE オブジェクト間のリンク接続が切れてしまうので、このために初期化コードは initialize_transitions()、initialize_transitions2() の二つのメソッドに分割される。これらのコードは、それぞれの RHS クラスにより異なるため、トランスレータで生成する必要がある。

initialize_transitions(): カレントオブジェクトのトランジション初期化、メッセージインスタンス、動的属性インスタンスの生成を行なう。

initialize_transitions2(): NODE::graft() からコールされる。NODE 間のメッセージオブジェクトのポインタ類のリンクの接続、動的属性の依存グラフ生成を行なう。

OODBMS 利用に関する問題

本節のここまで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なうこと

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。これは、データの安全性を保証するデータベース管理をプログラミングすることは、本研究の本質とは関係なく、十分な機能を持ったデータベース管理システムが市販されているからである。

だが、このデータベースに格納される NODE オブジェクトには、属性評価を行な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なうものではなく、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なった。

- 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++ で出来ること
- 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（VMMA）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オブジェクト指向データベース管理システムである ObjectStore である。ObjectStore の特徴について簡単に解説する。

ObjectStore の特徴

ObjectStore は C 及び C++ アプリケーションから利用することを前提に開発されたオブジェクト指向型のデータベース管理システム（OODBMS）で、従来のデータベースのデータ問い合わせ機能及びデータ管理能力と、オブジェクト指向言語 C++ の柔軟性と能力を兼ね備えたシステムである。

特に OOAG で使用するために必要となる以下の特徴をすべて備える数少ない OODBMS のひとつである。

- C ポインタ互換オブジェクト識別子
- VMMA に裏付けされる高速性
- 柔軟な永続データの扱い
  - 型によらないインスタンスごとの永続性の制御

- 永続オブジェクト内に揮発データを保有（参照）できること

- C、C++ プログラムからの移行のための柔軟な言語インタフェース

以下に、これらに関して簡単に説明する。

オブジェクト識別子

OODBMS にとって重要なのがオブジェクト識別子の扱いである。ObjectStore では、データベース上の実際のデータ位置を指し示す値をオブジェクト識別子として使用している。これは、インタフェース言語上からは、C ポインタ互換になっており、通常のポインタ操作がそのまま行なえるようになっている。

VMMA アーキテクチャ

ObjectStore の高速性は、仮想記憶マッピングアーキテクチャ(VMMA) という独自のアーキテクチャにより実現されている。またこれと同時に、巧妙なメモリマッピング、キャッシュ、クラスタリングなどによるデータアクセスの最適化とネットワークのクライアント/サーバ方式の計算能力を最大限に引き出すことにより最高のパフォーマンスを実現している。

VMMA の重要な特徴を以下にあげる。

- 永続的なデータは、C や C++ のプログラム中で一時的（非永続的）なデータとまったく同様に扱うことが可能。このことを、型と独立な永続性と呼んでいる。

- 大量のデータの検索や操作に伴うオーバーヘッドを最小限にとどめることが可能。ObjectStore は、この機能をオペレーティング・システムの仮想記憶管理機構を使用して実現している。この技術を、仮想記憶マッピングと呼んでいる。

- バージョン管理されているデータを、バージョン管理されていないデータと同じ速度でアクセス可能。メモリ内では、バージョン管理されているデータも、バージョン管理されていないデータと同じようにアクセス可能である。この技術を、型と独立なバージョン管理と呼んでいる。

VMMA では、プログラム用記憶領域やデータベース用記憶領域といった分割された記憶管理ではなく、単一の記憶管理機能を提供している。このため、C や C++ のプログラムから、直接データベースをアクセスすることが可能である。

VMMA はオペレーティング・システムの仮想記憶管理を利用していて、アプリケーションのデータ参照時にメモリにそのデータがなければページフォルトが発生し、ObjectStore が自動的にデータベースから参照されたデータをページングするようになっている。この動作は、通常のオペレーティング・システムがディスク上のスワップ領域との間で行なっ

ている仮想記憶管理と同一で、違いはスワップ領域がデータベースになっただけとみなすことが出来る。ObjectStore の高速性はこの機能に裏付けされていて、またこの機能のためデータベース操作をアプリケーション記述者から隠蔽することが可能になっている。

永続データの取り扱い

ObjectStore が他の OODBMS に比較して特徴的なのが、永続データの柔軟な取り扱いである。ObjectStore では、永続性とデータ型の概念が独立で、インスタンスごとに永続性の制御が可能である。

永続性とデータ型が独立していない場合、通常次のような問題が起こる。

- 一時的なデータ、永続的なデータに関して別々に関数を記述しなければならない。
- 永続的なデータへのアクセスのために加えなければならない余分な命令のために、他言語で書かれたライブラリとの互換性が失われる。
- C では、オブジェクト/クラスの概念がないため、C のデータ型をクラス定義に再パッケージしなければならない。これは、ソースコードレベルでの重大な変更であり、テスト・デバッグに多くの時間を費やすことになる。

言語インタフェース

ObjectStore では C、C++ への言語インタフェース、ObjectStore 独自の C++ 言語拡張である DDL/DML の 3 種類が使用でき、そのときの用途に応じてもっとも適したものを利用すればよい。データベースを利用するように作られていない既存の C または C++ アプリケーションを ObjectStore を利用するように移行する作業は、以下の手順で行なうことが出来る。

(1) トランザクションの境界を識別するためのプログラムの書き換えを行なう。

(2) メモリ領域割り当てを行なう malloc および new を ObjectStore 版に置き換え、データを永続的に記憶割り当てする。

(3) ファイル I/O に関する記述を ObjectStore のコマンドに置き換える。

(4) アプリケーションの再コンパイル。

4.3.5 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、4.3.4 節で述べたように、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 OSL から C++ へのコード変換の作業が属性文法に基づくシステムで行なわれていることは、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、属性文法の属性つ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。

コード変換で行なっている主な作業を以下にまとめる。また、具体的なコード生成結果は、付録 A に示した。

- 属性領域の割り当て
- 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 属性評価器の性能評価

ここでは、アルゴリズム確認の目的でxlisp [Bet85] インタプリタ上に構築されたプロトタイプシステムと、本論文でとりあげている C++ に変換して実行するシステムの処理速度の比較を行ない、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上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やりとりの方法 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、参照の度に毎回動的束縛を行なっている。C++ 版ではトランスレート時にオブジェクトの属性つき木への接続時に、属性インスタンスの束縛を行な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なえばよい。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、4.3.4 節の通りである。2 番目の項目については、木構造の変更の度に起こる特徴グラフのグラフ演算の高速化のために取った手法である。

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なった結果が図 4.1 である。図 4.1 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、OSL 記述を図 4.13 に載せる。テストでは、ルートノードとして FibNull[] オブジェクトの相続属性に正数を与えて、処理を行なわせ、合成属性が得られて状態が安定するまでの時間を計測した。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伴う静的属性計算の繰り返しから、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）のオーバーヘッドに対する評価が行なえる。

xlisp は、ほぼ全関数が 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

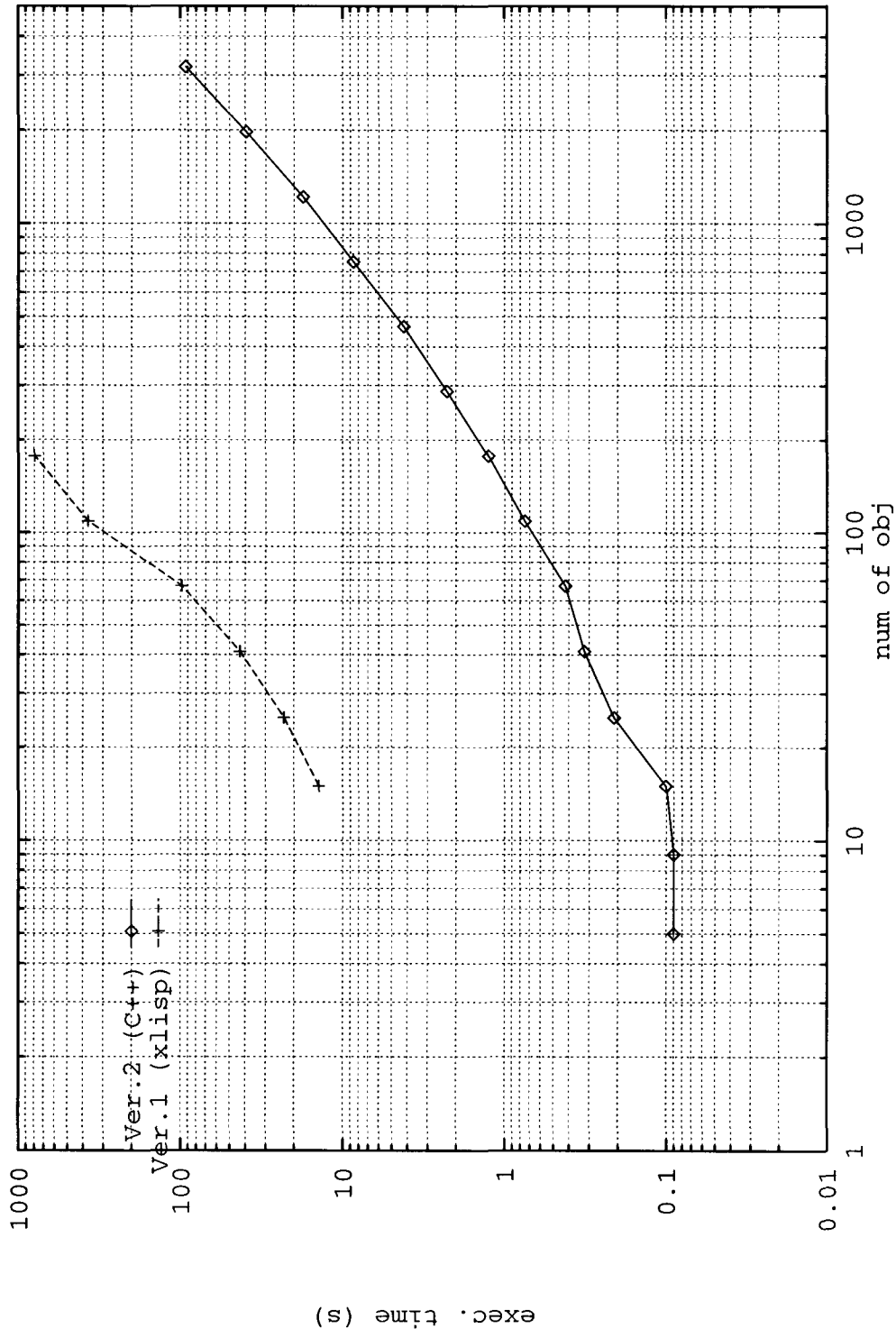


表 4.1: ベンチマーク結果

```
1  class fib(int in|int out)
2    →  FibNull[]
3    {
4      fib.out = 0 ;
5    }
6       /* no input messages */
7        case (fib.in < 2)
8          ⇒    /* no output messages */
9          {
10           (new fib) = FibOne[] ;
11         }
12       otherwise
13         ⇒    /* no output messages */
14         {
15           (new fib) = FibPair[FibNull[], FibNull[]] ;
16         }

17   →  FibOne[]
18   {
19     fib.out = 1 ;
20   }

21   →  FibPair[fib(int in|int out), fib(int in|int out)]
22   {
23     fib$1.out = fib$2.out + fib$3.out ;
24     fib$2.in  = fib$1.in - 1 ;
25     fib$3.in  = fib$1.in - 2 ;
26   }
```

図 4.13: テストプログラム

実験ではないので一概には結論出来ないが、それでも200倍以上の高速化出来たのは、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度に束縛を行なうのに比べれば、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出来る。

第5章　MAGE による OOAG のアプリケーション構築

この章では、前章までに説明したリポジトリ構築システム MAGE をどのように利用して、アプリケーションを構築できるかについて考察する。

5.1　構造指向プログラミング環境

5.1.1　情報共有の形

文献 [KLMM93] では、以下のような環境アーキテクチャに分類し説明している。

(1) disjoint な情報表現を持つ別々のツールの集合
強調し合わない別々のツールからなる環境。ツール間の情報共有は全く行なわない。

(2) パイプライン・アーキテクチャ
パイプラインの両端のツールの間でだけ情報表現を合わせれば良い。ツール間の情報共有はパイプラインの間のフォーマットのみ。

(3) 常駐型（residential）アーキテクチャ
環境のすべてのツールで同じアドレススペースを持ち、作業のサスペンド時にはその空間そのものを保存し、リスタート時にはそのまま再現できるようなシステム (Interlisp, Smalltalk-80)。

(4) バスアーキテクチャ
他のツールと通信するプロトコルを環境で統一して設定する (PCTE)。ツールはこれにしたがって情報交換する。

(5) クライアントサーバアーキテクチャ
バスアーキテクチャの発展形。共有情報を管理するサーバを配置する方法。

(6) 共通の情報表現を持つアーキテクチャ
情報表現の定義のしかたに焦点を当てた情報表現の共有アーキテクチャ。

後者のものほど、環境のツール間の情報共有度が大きくなる。Mjølner[KLMM93] は、(6) タイプの環境構築ツールで、中央の情報共有表現に抽象構文木（AST）を使っている。PCTE のオブジェクト管理サービスにおいても、E-R モデルによる共通表現を提供しており、(6) タイプの環境に近付いてきている。

MAGE で構築する環境を前述の枠組に当てはめるとすると、Mjølner と同様 (6) タイプのものになる。MAGE では、さらにただの AST ではなく「属性付き木」を共有のベースとしている（ツールの間で属性も共有できる）。これにより、例えばシンタックス・チェッカとコンパイラを構築する場合、両者でシンボルテーブルなどが共有できる。

5.1.2 属性付き木の共有

通常、構造指向環境では、各ツールで使用するための AST を共有する（図 5.1(a)）。

MAGE で生成する環境の各ツールでは、属性付き木を共有する（図 5.1(b)）。同じ言語を処理するシンタックス・チェッカやコンパイラなどの言語処理ツールでは、シンボルテーブルを計算する属性などが、共有のための特別な記述を必要とすることなく共有できるため、AST の共有だけを考えるよりも更にツールの開発効率が高くなる。シンタックス・チェッカの記述で使用したシンボルテーブルを計算する属性を、コンパイラの記述においてそのまま利用できるからである。

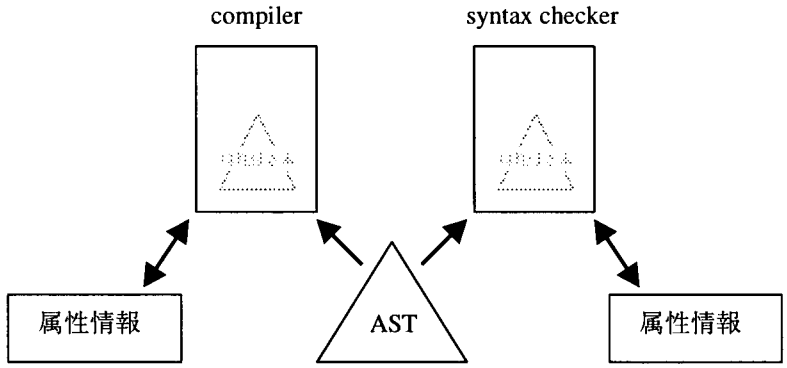
ひとつ考慮しなくてはならない点が、MAGE で構築するツールは通常のツールのイメージとは異なり、データ操作部（アルゴリズム）とデータ（構造）に分かれない点である。通常のツールは、ツール固有のデータ操作を記述する部分とデータをロード・セーブする部分が分かれる。これは Mjølner 環境構築システムでも同じである。すなわち、「共有」はデータ部分のみの共有であり、ツールの本体であるデータ操作部はツール毎にまったく独立である（図 5.2）。AST を共有する構造指向環境でも、それぞれのツールは独立にデータを持つのが通常である。

これに対して、MAGE はデータを表現する構造である属性付き木と、データ操作部分である属性評価アルゴリズムとそれぞれの属性の関係式を、オブジェクト指向技術により一体化しており、これらを明確に分離することは不可能な形式でツールを生成する。

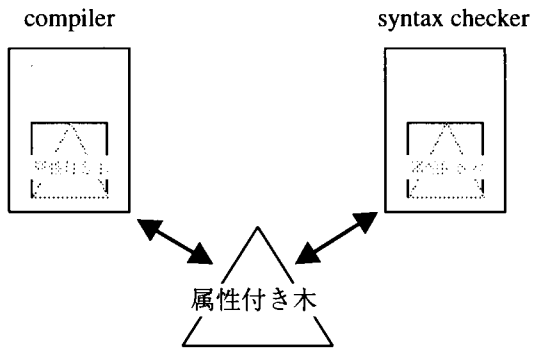
木の構造を定義する AST により共通の骨格が定義され、ツールごとに異なる属性とその関係式は肉付けを定義するものと言える。これのイメージを図示してみたものが 図 5.3 である。従来と大きく異なる点は、ツール構築時に使用するデータの骨格定義である AST と、これに附属するツール動作時のデータである属性を、まとめて共有してしまうことである。

5.1.3 ツール間通信のメカニズム

属性付き木の共有の話とは別に、環境内のツール間の通信のメカニズムに関しては、一考の余地がある。ツール間の通信を



(a) 通常の構造指向環境の場合



(b) MAGE の場合

図 5.1: 通常の構造指向環境での AST 共有と MAGE の違い

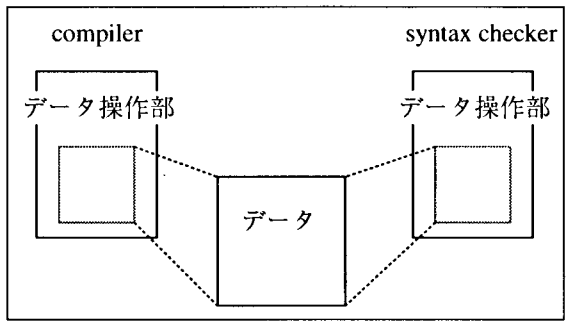


図 5.2: 従来のツールのデータ共有

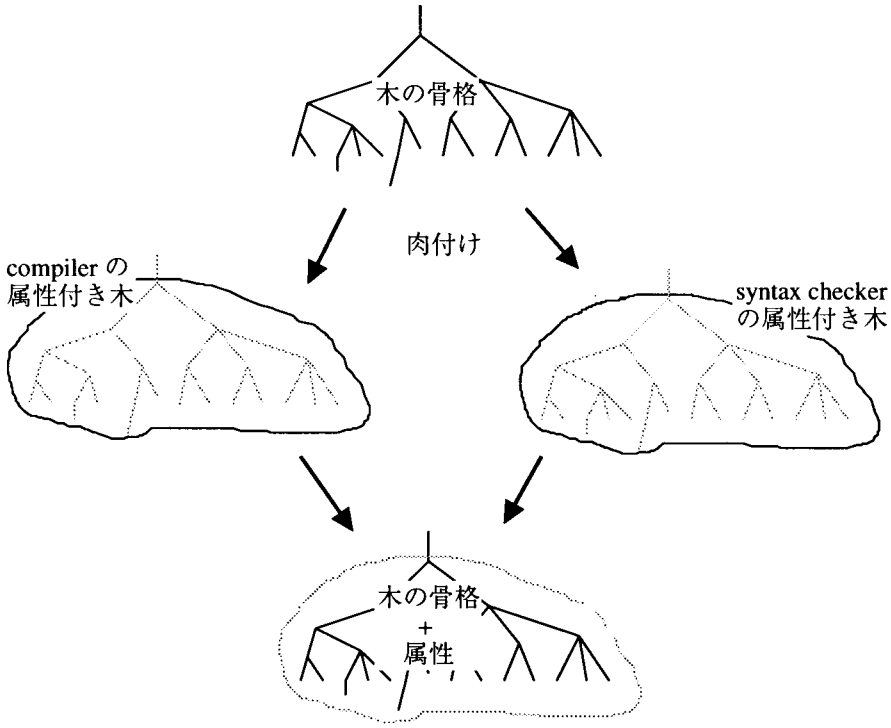


図 5.3: MAGE での共有のイメージ

(1) 静的属性値を用いて行なうのか、

(2) OOAG のメッセージを用いて行なうのか

である。(1) は、共有メモリによるツール間通信のイメージに相当する。ツール間で共通の属性を参照することで、情報の交換を行なう。(2) は、ツールを表現する属性付き木の間でメッセージを交換することにより情報交換する。

これは環境を1つの属性付き木で表現するのか、複数の属性付き木で表現するのかという問題にも影響する。現在の OOAG モデルでは、メッセージの交換には、木の上での親子関係が必要なので、

(1) 環境を構成するツールを、1つの木にまとめあげる文法（ルール）を設計するか、

(2) 個別の木をまたぐメッセージ（図 5.4）を導入するか

が必要になるからである。

では MAGE で構築する環境の属性付き木は、

(1) データを表現するものか

(2) ツールを表現するものか

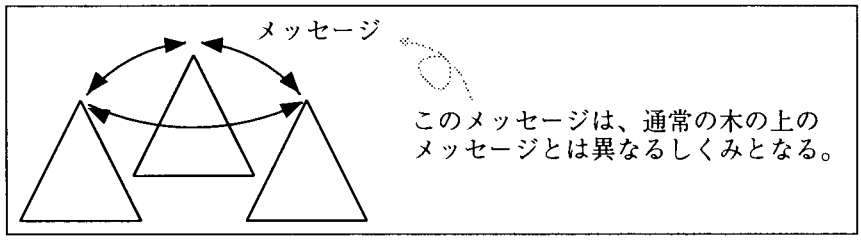


図 5.4: 複数の属性付き木を利用可能にする場合

どちらであろうか。属性文法は、

- データ構造主体の計算を記述する体系であり、
- アルゴリズムを主体に記述する体系ではない。

ということは、自然な記述を行なえば、属性付き木は「データの構造」を表現するものとなる。例えば、通常言語処理系では扱うデータが言語だから AST を表現すると考えられる。したがって、MAGE では扱うべきデータを AST 形式で表現し、その共通の AST 構造を基にする（個別の）属性付き木の上での静的属性の共有メモリ的な使用により、情報交換が行なわれる環境を構築するのが通常の形態となる。

しかし、構造指向環境では、環境内のすべてのツールが同じ AST を共有するわけではない。当然、独立したツールも存在するのが普通である。これは、MAGE により環境を構築する場合にも言える話であり、この場合の異なるグループのツール群の間の通信には、木にまたがるメッセージを利用する（図 5.5）。ただし、現在 OOAG のモデル内では、このような機能を規定しておらず、これはツール構築時のプログラマの実現に任されることになる。

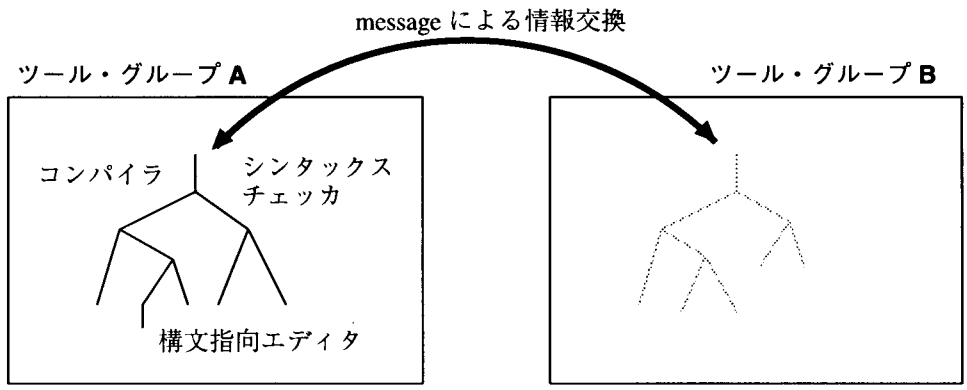


図 5.5: AST を共有するグループ間のメッセージ

ファイルシステム形式のデータベースのエミュレーション

ファイルシステムは、「ファイル名」をインデックスとする「ファイルの中身」を管理するデータベースであると考えることができる。同じファイルシステム形式のデータベースを構築する場合でも、いろいろな方法が存在する。例えば、UNIX での UFS のインプリメンテーションを真似て、内部データの管理形式などをそのままの形で実現することも可能であるし、通常ユーザ側から見えている階層構造の形をシミュレートする形式で実現することも可能である。

文献 [MI94, 今泉 95] では、この両者の形式によるファイルシステムの記述を行ない、OOAG の利点・欠点を議論している。

5.2 データベース的観点からの問題点

ここでは、OOAG のモデルの上でデータベースを構築する場合に、特別に考慮しなくてはならない点について考察する。

5.2.1 「属性付き木の存在空間」の問題

4.4 節で性能評価のために利用した OSL 記述 `fib.ooag` を例に取って考える。

この記述で生成される fib(n) オブジェクトは、定常状態にあるときその内部オブジェクト（部分木）として、fib($n-1$) と fib($n-2$) オブジェクトを含む。同様にして、この fib(n) をルートとするオブジェクト空間には、オブジェクト fib(k) ($1 \leq k \leq n$) が存在する。

では、この木から fib(k) を retrieve するには、どうしたらよいだろうか。この fib の場合なら、fib(n) の部分木に fib($n-1$) が含まれることが分かっているので、retrieve メッセージ 1 つで処理できる。しかし、より一般的な場合を想定すると、ルートに外部に export するオブジェクトのインデックス・テーブルを持つ合成属性を計算しなくてはいけない。

しかし、これは

(1) データベース・プログラマへの負担が増大

(2) 余分な記述が増える分、記述の可読性が低下（重要な部分が分かりにくくなる）

(3) 1 つの大きな木にしなくてはならない

などの問題を生じさせる。

以上の問題は、そのままソフトウェアデータベース（リポジトリ）記述に応用する場合にも当てはまる。上記の問題の fib オブジェクトを関数オブジェクト、木全体をプログラムの構文解析木であると考えてみると良い。

5.2.2 我々はどこまで「木」で扱うのか?

前節の問題に対応する方法はいくつかある。

- 1つの大きな木で扱うことにしてしまう。例えば、複数のプログラムの木を扱う場合であれば、プログラムの木のリストとして1つの木にまとめる。
  しかし、これは別の問題 — いろんなもののインタフェース方法をどうする? — を引き起す。
  - 記述者がインデックス作成のルールを記述する。
  - 記述者は exportable であることを宣言するだけでよい。システム側でインデックス作成のルールを自動的に付加する。
- 複数の木が存在可能。例えば、プログラム A の木、プログラム B の木などが独立して利用可能とする。
  - オブジェクトのクラスに export フラグを追加する。export フラグのあるオブジェクトは生成時にシステムのオブジェクト・テーブルに登録され、(OOAG とは別の枠組で) 外部に export される。(フラットなネーム・スペース)

しかし、どちらにしろ、複雑なインデックスを作ろうとすると、記述者が自分で作るしかない。したがって、現状では1つの木が存在する世界を想定し、インデックス作成はすべて記述者に任せることにする。

5.3 例題: C ソースコード・リポジトリの記述

ここでは、プログラミング言語の構造が、ほぼそのままデータベーススキーマになることを示すための例題として、MAGE で C プログラムのソースコード・リポジトリを設計する。問題を複雑にしないために、プリプロセッサに関する事項は、考慮しないこととする。ソースコード・リポジトリの設計は、以下のステップで進行する。

step 1. OSL クラス構造の設計

step 2. 静的属性と静的意味規則の定義

step 3. 動的属性と動的意味規則の定義

以下それぞれのステップについて、サンプルの OSL 記述の断片をあげて示していく。

```
1  class translation_unit(|)
2      → TranslationUnit[external_declaration(|), translation_unit(|)]
3  class external_declaration(|)
4      → ExternalDeclarationNull[]
5      → ExternalDeclaration1[function_definition(|)]
6      → ExternalDeclaration2[declaration(|)]
7  class function_definition(|)
8      → FunctionDefinition[string declaration_specifiers , string declarator ,
9                           declaration_list(|), compound_statement(|)]
10 class ...
```

図 5.6: OSL クラス構造の定義

5.3.1 OSL クラス構造の設計

まず、目的の言語の一般的な EBNF 記述をもとにして OSL クラス構造を設計する。これは具体的には、扱うオブジェクトの構造を定義することである。通常の構造指向環境生成系で言えば AST 定義のステップに相当し、1 つの OSL クラスは、プログラミング言語の定義の 1 構成規則に相当するので、このステップはほとんどそのまま進めることが可能である。このステップでは、どの詳細レベルまで言語構造を保って扱うのかを決定しなくてはならない。OOAG の概念で言い換えると、葉固有属性で何を表現するのかを決定しなくてはならない。これは、我々のリポジトリに対する要求により決定される。例えば、関数の内部情報まで扱うようなリポジトリが必要であれば、関数内部まで言語構造を保った表現しておかなくてはならない。図 5.6 に示す記述例では、ノード *function_definition* の子ノードの *declaration_specifiers* と *declarator* は、クラス *FunctionDefinition* において文字列型の値を持つ葉固有属性として宣言されている (8 行目)。

5.3.2 静的属性と静的意味規則の定義

OSL クラス構造定義の次は、定義したそれぞれの RHS クラスに対して、静的属性と静的意味規則の定義を行なう。以下に、静的仕様記述において記述する主な事項を列挙する。

- 管理しているオブジェクトの変更に応じて、インクリメンタルに値の計算を行なう必要のあるもの / 行わせたいもの。
  例: シンボルテーブル、情報検索用の索引、ほか

```
1  class F_INFO (|)
2    --> FunctionInformation[string type, string name, arg_info_alist arg_info,
3                            compound_statement *body]
4  class translation_unit(|f_info_alist f_table)
5    --> TranslationUnit[external_declaration(|F_INFO f_info),
6                        translation_unit(|f_info_alist f_table) ]
7    {
8      translation_unit$1.f_table = add_elem(external_declaration.f_info,
9                                            translation_unit$2.f_table);
10   }

11 class external_declaration(|F_INFO f_info)
12   --> ExternalDeclaration1[function_definition(|F_INFO f_info)]
13   {
14     external_declaration.f_info = function_definition.f_info;
15   }

16 class function_definition(|F_INFO f_info)
17   --> FunctionDefinition[string declaration_specifiers, string declarator,
18                          declaration_list(|), compound_statement(|)]
19   {
20     function_definition.f_info =
21       FunctionInformation[declaration_specifiers, declarator,
22                           declaration_list.arg_info, &compound_statement];
23   }

24 class ...
```

図 5.7: 静的属性と静的意味規則の付加

- 必要なデータの計算のために必要となる中間情報の計算

図 5.7 の記述では、リポジトリ中の関数定義に関する情報を集めたテーブルを逐次メンテナンスするために、静的合成属性としてそれを計算している。この例で、属性の型 *arg_info_alist*, *f_info_alist* は、それぞれ引数リスト情報、関数本体に関する情報を保持する連想リストと仮定している。クラス *F_INFO* は、情報テーブルの各要素の型として使用している。ルートノードの属性 *translation_unit.f_table* は、木の構造が変化した場合にはいつでも属性評価器により逐次的に再計算される。図 5.8 は、この属性評価の様子を模式的に表現したものである。

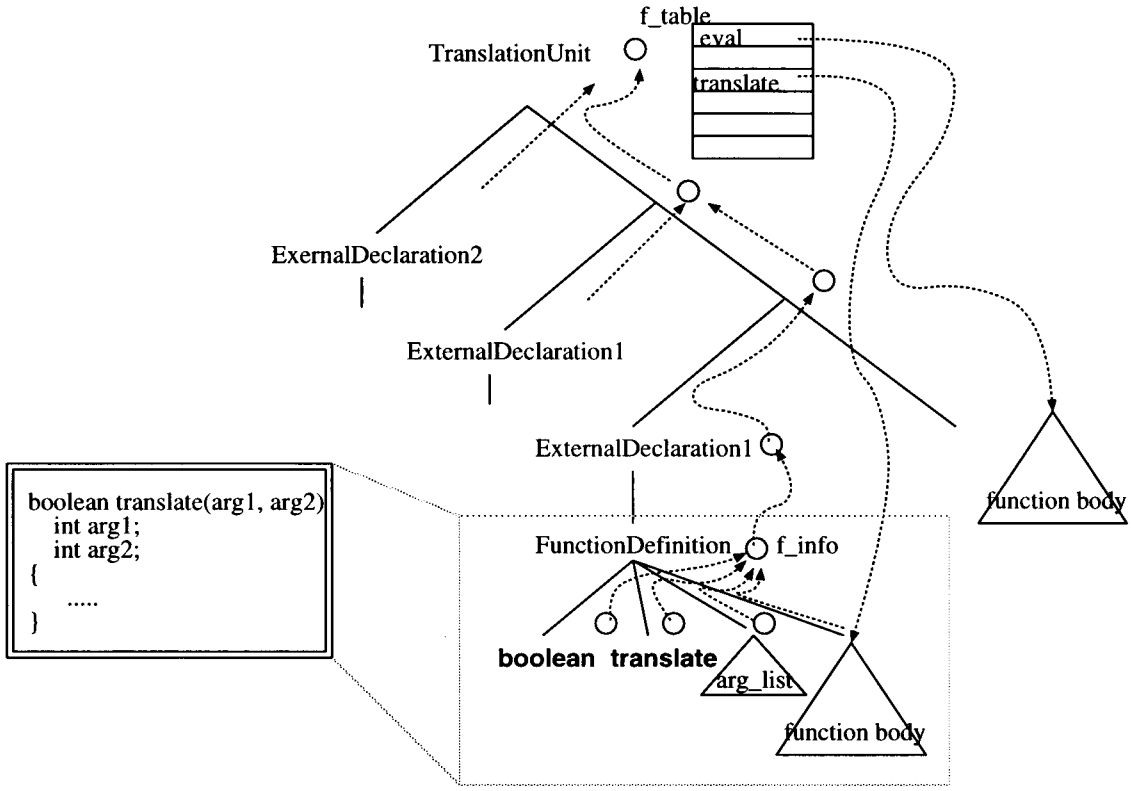


図 5.8: 関数定義に関する情報の作成

5.3.3 動的属性と動的意味規則の定義

最後に、リポジトリ中からの情報の検索や更新、外部イベントにより起動される処理（トリガ）などの動的なジョブを、動的仕様記述（メッセージ、動的属性、動的意味規則）により定義する。以下に、動的仕様記述において記述する主な事項を列挙する。

- 木の構造を変化させるもの
  例: データ更新操作
- ユーザとのインタラクションを行なうもの
  例: Query 操作、retrieval 操作
- 構造変更の度に計算する必要はないもの、
  あるいは計算すると著しく効率が悪いものの計算
  例: コンパイルコードの生成

図 5.9 の記述例では、構文的には間違いではないが、おそらくタイピングのミスであると思われる「**if** ($exp_1 = exp_2$)」という構造を探すための一連のメッセージ *if_check* を定義している（図 5.10 参照)。この例では、このような構造が存在するかどうかをチェックしているだけであるが、27〜33 行目のようにして部分木を「**if** ($exp_1 == exp_2$)」という構造に update するためのメッセージ *correct* を定義することも可能である。

次に retrieval の例として、蓄積されている C 言語の AST から関数のソースコードを取り出す例題を考える。この際、ANSI スタイルと traditional スタイルのどちらの形式で取り出すかを指定するため、図 5.11 の記述例では *get_by_ansi_style*, *get_by_traditional_style* の 2 つのメッセージを定義している。

5.3.4 この節のまとめ

この節を通して例示してきたように、OSL では（意味的に）強く結びついた小さな部品間の関係で、必要なオブジェクト（データ）を構築する。このため、オブジェクト操作の記述を非常に見通しよく行なうことが可能である。

OOAG では、従来のソフトウェア開発環境における「ファイル」に相当する物体は、AST として表現される。したがって、プログラミング言語構造に基づく問題などは容易に記述することが出来るが、扱うことが困難な問題も存在する。例えば、ファイルシステムにおけるディレクトリの木構造のようにインスタンス・レベルでは木のように見えても、スキーマ・レベルではうまく木で表現できないオブジェクトも存在する。この問題の例として、ソフトウェアの構築には欠かせないツールである make を取り上げる。

make は、目的のプログラムの実行形式を得るためのファイルの構成関係を Makefile に記述し、最短のパスで目的物を得られるようにプログラムするものだ。この Makefile の記述は、ファイルシステムのシステムレベルの構造を示すものではなく、特定のプログラム開発における実際のファイルやディレクトリの位置を示す、ファイルシステムのインスタンスレベルの記述である。これは、単純にファイルの依存を示す木構造になるが、OOAG でこのようなインスタンスレベルのスキーマを記述するのは現実的ではない。再利用性が乏しいからだ。

このような問題に対する対応は、今後の考慮事項である。

```
1  class selection_statement(|)
2     --> SelectionStatement1[expression(|), statement(|)]                    /* if */
3         { ... }
4         selection_statement::if_check(|bool result)
5           => expression::if_check(|bool result)
6            { selection_statement.result = expression.result ; }
7     --> SelectionStatement2[expression(|), statement(|), statement(|)]  /* if else */
8     --> SelectionStatement3[expression(|), statement(|)]                 /* switch */
9  class expression(|)
10    --> Expression1[assignment_expression(|)]
11        { ... }
12     expression::if_check(|bool result)
13       => assignment_expression::if_check(|bool result)
14        { expression.result = assignment_expression.result ; }
15     expression::correct(|)
16       => assignment_expression::correct(|)
17        { }
18    --> Expression2[expression(|), assignment_expression(|)]
19 class assignment_expression(|)
20    --> AssignmentExpression1[conditional_expression(|)]
21    --> AssignmentExpression2[unary_expression(|), assignment_operator(|),
22                                                  assignment_expression(|)]
23        { ... }
24    assignment_expression::if_check(|bool result)
25      => /* no output messages */
26       { assignment_expression.result = false ; }                        /* error? */
27    assignment_expression::correct(|)
28      => /* no output messages */
29       { (new assignment_expression) = with (assignment_expression) {
30            AssignmentExpression2[X,"=",Y]:
31             AssignmentExpression1[ConditionalExpression1[
32                                        ...EqualityExpression[X,"==",Y]...]]
33            }; }
34 class conditional_expression(|)
35    --> ConditionalExpression1[logical_OR_expression(|)]
36    --> ConditionalExpression2[logical_OR_expression(|), expression(|),
37                               conditional_expression(|)]                    /* ? : */
```

図 5.9: 動的属性と動的意味規則の付加 (1)

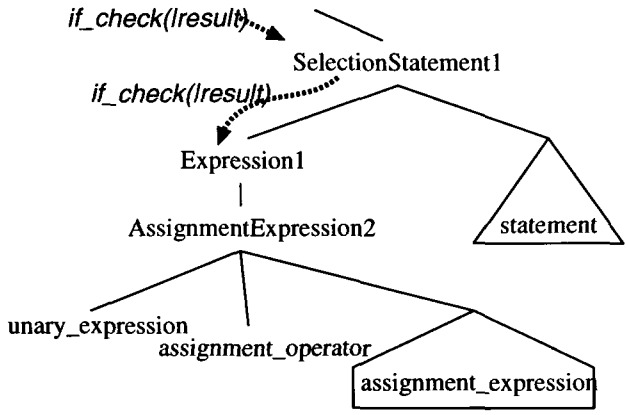


図 5.10: **if** 文のチェックの例

```
1  class function_definition(|)
2    → FunctionDefinition[...]
3      function_definition::get_by_ansi_style(|string code)
4        ⇒ declaration_list::get_by_ansi_style(|string code)
5          { function_definition.code = declaration_specifier + declarator +
6                                       declaration_list.code +
7                                       compound_statement.code ; }
8                                                    /* ANSI 形式のコードを形成 */
9      function_definition::get_by_traditional_style(|string code)
10       ⇒ declaration_list::get_by_traditional_style(|string code)
11         { function_definition.code = ... }        /* traditional 形式のコードを形成 */
```

図 5.11: 動的属性と動的意味規則の付加 (2) 関数のソースコードの取り出し

5.4 MAGE のコード生成の実際

表5.1に、実際に簡単な OSL 記述を MAGE により C++ 記述に変換し、どのぐらいの量のコードになるのかを示した。また、付録 A において、実際に生成された C++ コードを載せるとともに、それらの解説を行なった。これから分かるように、C++ コード量は、属性数、動的記述の量（メッセージ数、トランジション数）には単純に比例して増大する。

表5.1では、RHS クラス数の観点で変換後のコード量の概算を行なった。変換後のコード量のオーバーヘッドが大きいのは、RHS クラス定義に関する部分のみであるので、よって RHS クラス数から変換後のコード量を見積もることが可能である。

OSL 記述名	解説
dig.ooag	1桁の数字を表す文字を終端記号とする数字の列を表現する木を構成し、メッセージの出力属性で指定した整数により、異なる解釈で結果を計算する例。例えば、図5.12(b)の木にメッセージ `N::value(10|result)` を送ると、終端記号列 123 を 10 進数として解釈し `N.result = 123`。`N::value(8|result)` を送ると、8進数として解釈し `N.result = 83` となるような計算を行なう例。図5.12(a)にソースコードを載せた。
fib.ooag	4.4節を参照

(a) サンプル記述の解説

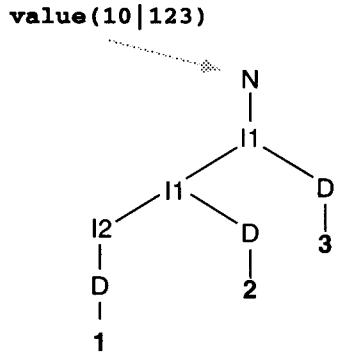
OSL 記述名	OSL 記述の量	変換後の C++ 記述の量
dig.ooag	3 LHS, 4 RHS class 76 line	→ 800 line
fib.ooag	1 LHS, 3 RHS class 30 line	→ 500 line

(b) コード量の比較

表 5.1: OSL 記述変換後の C++ コード量

```
class N(|)
  → N1[I(int scale|)]
  {
    I.scale = 0;
  }

    N:value(int radix|int val)
      ⇒ I:value(int radix|int val)
      {
        I.radix = N.radix;
        N.val = I.val;
      }


    N:change(int change_pos, D new_digit|)
      ⇒ I:change(int change_pos, D new_digit|)
      {
        I.change_pos = N.change_pos;
        I.new_digit = N.new_digit;
      }

class I(int scale|)
  → I1[I(int scale|), D(int scale|)]
  {
    I$2.scale = I$1.scale + 1;
    D.scale = I$1.scale;
  }

    I$1:value(int radix|int val)
      ⇒ I$2:value(int radix|int val),
        D:value(int radix|int val)
      {
        I$2.radix = I$1.radix;
        D.radix = I$1.radix;
        I$1.val = I$2.val + D.val;
      }


    I$1:change(int change_pos, D new_digit|)
      case(I$1.change_pos == I$1.scale)
        ⇒
        {
          (new D) = I$1.new_digit;
        }

      otherwise
        ⇒ I$2:change(int change_pos, D new_digit|)
        {
          I$2.change_pos = I$1.change_pos;
          I$2.new_digit = I$1.new_digit;
        }

  → I2[D(int scale|)]
  {
    D.scale = I.scale;
  }

    I:value(int radix|int val)
      ⇒ D:value(int radix|int val)
      {
        D.radix = I.radix;
        I.val = D.val;
      }


    I:change(int change_pos, D new_digit|)
      case(I.change_pos == I.scale)
        ⇒
        {
          (new D) = I.new_digit;
        }

class D(int scale|)
  → D1[int digit]
   {}
    D:value(int radix|int val) ⇒
      {
        D.val = digit * D.radix exp D.scale;
      }
```

(a) dig.ooag のソースコード



(b) dig.ooag の木の例

図 5.12: dig.ooag

第6章　おわりに

6.1　MAGE と OOAG の変遷

ここでは、これまで片山研究室で行われてきた OOAG, MAGE に関する研究を簡単にまとめる。MAGE プロジェクトは、現在までに以下の3つの段階を経て進行してきている。

- **第1世代** ([SK90a, Shi89])
  OOAG の基礎的な機構

  (1) メッセージ・パッシングによる同期部分木置換と動的属性評価

  (2) オブジェクト指向的な側面 — 非終端記号・生成規則をクラス、属性・部分木をインスタンス変数とした。

  が示された。

  OOAG に対応した属性評価アルゴリズム（LCDIA アルゴリズム – 4.3.2 節を参照）の提案が行なわれた。

- **第2世代** ([GISK93])
  文献 [GISK93] において OSL 言語の構文の拡張が行なわれ、メッセージのガード、raise 機構（条件式によりトリガされるメッセージ・パッシング）、遠隔オブジェクト参照を含む動的なオブジェクトの動作仕様が、より正確に定義された。小さなシステムをいくつか記述することで、OOAG をソフトウェア開発環境に応用する利点も議論された。これに使用された MAGE プロトタイプシステムは、xlisp システム上に構築されていた。

- **第3世代** (現在のもの)
  Unix ファイルシステムや、コンフィギュレーション管理システムを OOAG を使って記述する実験の結果、我々の関心はデータベース的観点、特に永続的な機構に移ってきた。第2章で述べたように、OOAG を利用する我々のアプローチは、PCTE やオブジェクト指向データベースを直接利用するやり方と比較して、優れた性質を持っている。

  第2世代の実験用に構築された MAGE システムは、C++ と SSL（Synthesizer Generator の言語）により再構築された。このシステムでの新しい特徴を、以下に列挙する。

強く型付けされた言語、構造データ型、データとしてのオブジェクト、遠隔オブジェクト参照の改訂、外部クラス（C++ ライブラリとのインタフェース）、属性値に対する属性付け、効率的な実行のためのメッセージ・パッシングの制限、built-in infix オペレータ

6.2 関連研究のまとめ

属性文法型の計算モデル OOAG と、これをソフトウェア開発環境のリポジトリ構築に応用するための処理系である MAGE システムには、これらが目的としているアプリケーション領域に対して多くの関連する研究分野が存在している。ここでは、そのような関連研究をまとめる。

6.2.1 構造指向プログラム開発環境

開発環境構築系

Gandalf アクションルーチンの記述による構造指向エディタ生成系を中心とする開発環境構築系である。開発環境内のデータベースを RDB で実現することを試みたが、CASE データは一般にオブジェクトの大きさがまちまちであり、RDB の範疇でうまく扱うことには難があると結論している。

Synthesizer Generator 属性文法に基づく言語 SSL により記述されたエディタ仕様から、構文指向エディタを生成するエディタ生成系である。Ordered 属性文法のインクリメンタル属性評価器を備え、X-window 上で動作する高機能エディタを生成する。テキストの編集と同時に属性評価を進めることが可能で、文法エラーの逐次的発見のほか様々なことを行なうことが可能である。MAGE 環境のエディタやコード変換器の構築にも利用している、

Mjølner 構文指向のオブジェクト指向ソフトウェア開発環境を構築しようというプロジェクトで、Mjølner Orm system, Mjølner BETA system, OSDL の 3 つのシステムから構成されている。この中の Mjølner Orm system が、インタラクティブな、構文指向の、そしてインクリメンタル・コンパイルを可能にするシステムを構築するためのものである。Door AGs（6.2.3 節参照）などの研究が、このプロジェクトで行なわれている。これに関する詳細は、文献 [KLMM93] にある。

データベース変換に関して

TransformGen[GKL94] 構文指向型のシステムにおいて永続的データを扱う場合に深刻となる、文法（抽象構文木）の進化にともなう問題を、データベース変換系の自動

生成と言う手法で解決しようとしている。従来までは、アドホックな手法でデータベース変換するしかなかった。

構文指向環境は、形式的な記述や文法から生成され、ユーザが環境から抜けても次回のセッションですぐに再開できるようにするために、属性やその他の情報が付加された抽象構文木表現を永続データとする。しかし、文法が少しでも変更されると、昔の抽象構文木を表現したデータは、新しい環境では全く参照できない。

TransformGen では、Gandalf の ALOE 文法で定義された「変更前の文法」、「新しい文法への差分」、「規則の差分」を入力とし、「新しい文法」と「データの変換ルール」を出力する。このデータの変換ルールを変換器に入力することで、古くなったデータベースを新しい環境で使えるものに変換する。

6.2.2 Attributed Graph Grammars

CACTIS, GRAS, PROGRESS は、木だけでなくグラフ構造も扱うことのできるシステムである。これらのアプローチは、主要な木構造を持たず、木構造の抽出が難しいシステムのモデリングを自然に行なえる。多重継承を許す継承関係の階層のモデリングなどがこの例になる。また、path 式を使った deriving link は、強力な仕組みである。しかし、2つの短所が存在する。

- **記述の分散**
  すべての属性式は、ノードに付けなくてはならず、枝には付けられない。このため、記述の維持・管理が若干困難になる。たとえば、生成規則 "exp → if(exp, exp, exp)" の属性式は、生成規則ではなくノード "exp" に付けなくてはならない。"exp → sum(exp, exp)" や "exp → let(id, exp, exp)" などの同じ左辺を持つ生成規則に対する別の属性式もまた同じ "exp" に付く。このため、それぞれのノードが枝の数だけの大量の等式を持ってしまう。

- **実行速度の不足**
  属性付きグラフ文法によるアプローチでは、通常 lazy two phase incremental attribute evaluation algorithm を使用している。これは AG アプローチのアルゴリズムと比較して効率が悪い。これに関しての詳細は文献 [Rep82] にある。

6.2.3 遠隔オブジェクト参照

DR-thread[A.V90] と Door AGs[Hed94] では、OOAG と類似した遠隔オブジェクト参照の機構を導入している。どちらも（遠隔）属性を評価する効率の良いアルゴリズムに焦点を当てているが、これらのアルゴリズムは、遠隔オブジェクトの属性が常に参照可能であるという意味で静的なものである。

我々の経験では、これらのシステムでは、相互参照のモデル化において属性の依存関係に循環を引き起こしがちである。通常の属性評価アルゴリズムは、属性の依存関係に循環のないことが必要であり、相互参照はソフトウェア開発環境では頻繁に出現する。この問題に対応するために、OOAG ではメッセージ・パッシングによる動的なアクセスを許す遠隔オブジェクト参照を導入した。

6.2.4 同期・非同期の複数部分木置換システム

文献 [Rep82] により与えられた時間最適インクリメンタルアルゴリズムは、cut & paste モデルと呼ばれる同期的な単一部分木置換を仮定している。ここでの「同期」という用語は、属性評価中には部分木は置換されないことを意味している。

非同期部分木置換は、分散システム、特にソフトウェア開発環境のモデリング時に必要になることが多い。文献 [MK93, KK93] で与えられているアルゴリズムは、非同期複数部分木置換を許している。しかし、高階化、遠隔オブジェクト参照などは考慮していない。MAGE 環境では、現在同期複数部分木置換をサポートしている。非同期への拡張は、今後の拡張事項のひとつだ。

6.2.5 高階属性文法

高階属性文法 [HSM89] は AG の拡張モデルで、H.H. Vogt により提案された。この拡張により次の 2 点が可能になっている。

(1) 属性値が導出木の一部により定義できる。

(2) 導出木の一部が属性値により定義できる。

高階属性文法では、属性値により定義される非終端記号は、非終端属性と呼ばれる。すべての非終端属性は、初期段階には空の生成規則 $X \to \perp$ として扱われる。非終端属性の値が属性評価の間に決定されると、その値 — 木構造の一部 — もまた属性付けされる。

このモデルは、静的なシステムである。一旦決定された非終端属性の値を変更する手段は存在しない。同じことが、Synthesizer Generator を高階属性文法に拡張する文献 [TC90a] のアプローチにも当てはまる。そこでの非終端属性の部分木置換は強く制限されている。これらとは異なり、OOAG は動的なシステムである。OOAG では、別の木による繰返し置換が記述できる。例えば、次の記述は 2 つの部分木の左側と右側を交換する。

```
class exp(|) → Sum[exp(|), exp(|)]
  exp$1:commute(|) ⇒
    { (new exp$2) = exp$3;
      (new exp$3) = exp$2; }
```

6.2.6 時間属性付き超グラフ文法 TAHG

TAHG（Timed Attribute Hypergraph Grammar）[TW93] は、会話的図形システム生成系のためのモデルであり、超グラフ文法に時間の概念を持つ属性を取り入れている。属性は、超グラフの導出木上で計算され、時間の経過にしたがって属性の値が変化していく（TAHG では、線形離散時間を仮定している）。TAHG での時間の概念は、OOAG での評価ループと非常に類似しており、線形な離散時間上で次の時刻の属性の意味規則を記述することができる。すなわち時系列 $t_0, t_1 \ldots$ に対して、次のような属性意味規則

$$\alpha_{t_{i+1}} = f(\ldots \beta_{t_i} \ldots)$$

を記述することができる。ここで $\alpha_{t_{i+1}}$ は、時刻 t_{i+1} における属性 α の属性インスタンス、β_{t_i} は、時刻 t_i における属性 β の属性インスタンスを示す。

OOAG も線形な離散時間上での属性計算とみなせるが、次の時刻の属性だけでなく次の時刻の木構造を定義できる点が大きく異なる。

6.2.7 HFSP

HFSP (Hierarchical Functional Software Process) [SK90b] は、ソフトウェアプロセスプログラミングを AG を基づき階層的かつ関数的に扱うためのモデルである。OOAG では導出木がオブジェクトとして考えられるのに対して、HFSP ではアクティビティが導出木でモデル化される。言い換えると、ソフトウェア開発におけるいろいろなプロセス（アクティビティ）は、サブタスクで構成される階層的構造として捕えられ、開発で生成されたソフトウェアオブジェクトは属性値として捕えられる。アクティビティが失敗した場合には、そのアクティビティを表現する部分木が他のプロセスにより置換される。この操作は、木の特別な操作としてモデル化されている（メタオペレーションと呼ばれている）。

メッセージ・パッシングの概念は HFSP には含まれないが、木構造に対する操作という点で OOAG と非常に似ている。特に、オペレーション/オブジェクト と 属性/非終端記号 の対応は、2つのモデルで反転している。これは、何らかの双対関係があることを示唆している。この2つのモデルの関係をより細かく調べることで、より一般的なモデルの構築が可能であると思われる。

6.2.8 CASE 環境全般 / データベース管理システム

これについては、2章で議論した。

6.3 MAGE によるリポジトリ構築に対する評価

様々な機能を有するオブジェクト指向データベース管理システムが製品化され、このようなものをソフトウェア開発環境におけるリポジトリ構築に利用すれば、データベースの機能的には十分になってきている。問題は、データベース機能ではなく、どのようなデータベースを作成すれば役立つリポジトリになるのかに移っていて、こちらはいろいろな実験システムが作られているが、まだ決定的な解法が得られていない。

実験システムの構築であったとしても、リポジトリ・データベースの設計とシステム構築は、費用と労力のかかる問題である。これは、リポジトリ・データベースに行なわせようとする処理内容が、複雑になればなるほど大きな問題となり、また構築したシステムのテスト・デバッグにもかなりの時間を割くことになる。OOAG をリポジトリ構築に応用したもっとも大きな利点は、

(1) 階層構造を持つデータの優れた表現力

(2) その構造に沿う属性の関係記述による理解しやすいデータ処理内容の記述

(3) およびこの記述からリポジトリを生成するシステムが自動生成できること

で、リポジトリ・データベースのシステム設計だけでなく、システム構築までを少ない労力で行なえるようにしたことである。全体的に見て、OOAG をデータベース・クライアントを作成するためのツールとして割り切ったのが、良い結果を生んだ。

6.4 現実のソフトウェア開発環境で使用するための考察

本論文では、OOAG をリポジトリ構築のための道具としてとらえて、属性評価器構築とデータベース・インタフェースを与えて、オブジェクト指向データベース上に OOAG の計算モデルで動作するリポジトリを構築するためのシステムを作成した。しかし、2 章でも議論したことだが、データベース上にリポジトリを構築すると言うことは、一連の専用のアクセスツールを含めた開発環境として構築されなければ、実用にならない。

我々は、MAGE を OOAG でのリポジトリの自動生成システムとしてだけでなく、リポジトリ開発環境として実現した。この環境内に、OOAG で記述した OSL コード管理のリポジトリを組み込むのもおもしろい実験であり、OOAG で生成したリポジトリシステムを、現実のソフトウェア開発環境で使用するときのヒントが得られるだろう。また、OOAG の実用性を判断する良い材料になる。

ただし、ここには課題も残っている。OOAG によるリポジトリ開発において、利用できるライブラリ集が無いことである。現在のままでは、たとえば「バージョン管理」という要求に対して、毎回そのためのアルゴリズムを記述しなければならない。利用頻度の高い、一般的な機能に関しては、簡単に利用できるライブラリ集などにまとめあげる作業が必要である。

6.5 属性評価器の課題

6.5.1 属性文法のクラスの制限

現在の静的属性評価を行なう属性評価器は、非循環属性文法を扱うものであり、属性文法はリポジトリなどのデータ処理への応用にも耐えるものが記述可能であるが、実行時の特徴グラフ再計算などの処理がどうしても必要になるため、その分処理速度をあげることが出来ない。

ソフトウェア工学分野での要求をほぼ満たすことが可能で、静的解析により実行時のグラフ処理などの作業の必要のない効率的な属性評価器を構成できるクラスの属性文法に、Kastens の ordered 属性文法 [Kas80] がある。この ordered 属性文法のインクリメンタルな属性評価器は、Synthesizer Generator などの商用ツールで実際に使用されており、実用にも耐えるという意味で、十分な実行速度も得られることがわかっている。このような背景から、OOAG の扱う属性文法を ordered 属性文法に制限し、実装上のテクニックだけでなく、本質的に効率の良い属性評価器を得る要求が存在する。

権藤は文献 [権藤 91] において、この odered 属性文法に対する LCDIA 属性評価器の議論を与えている。これにおいて得られている結論は、

- LCDIA アルゴリズム自体は大きな変更なく使用できる。
- OOAG の属性評価器に仮定していた「必要のない属性の評価を避ける」ことは出来ない。

というものである。「必要のない属性の評価を避ける」ことが出来ない場合、属性値を得るための関数記述が、外部ツールを起動してその結果を得るようなものであった場合、無駄なツール起動を防げない状況が起こり得ることになる。

6.5.2 type 3 循環に関する対策

我々の行なっているソフトウェア工学分野に、ordered 属性文法を採用する場合の大きな問題は、ordered 属性文法の持つ type 3 循環に対する対策である。

文法記述から静的に属性の評価順序が決定し、高速な属性評価器が構築できるもっとも大きなクラスの属性文法に l-ordered 属性文法 [EF82] があるが、これは属性評価器の構築に NP 完全な問題を含む。そのため、l-ordered 属性文法ではなく、ordered 属性文法に対して属性評価器構築が行なわれるわけであるが、ordered 属性文法より大きなクラスで、属性文法の変換を行なうことにより ordered 属性文法になるようなものが存在する。ordered 属性文法の判定において「type 3 循環」が存在すると判定される場合である。

ordered 属性文法は、かなり大きなクラスで通常記述する意味のある属性文法は、ほとんど ordered 属性文法の範囲に納めることが可能である。効率の良い属性評価器が構築できることからも、これは非常によい性質である。問題は type 3 循環の検出される属性文

法で、Synthesizer Generator などでは循環の原因となった属性を、プログラマに示すぐらいのことを行なっているだけである。この情報を元に、属性文法を ordered 属性文法に変換する作業は、大変な労力を要する問題である。全体の属性依存をほぼ熟知しているはずのプログラマであっても、type 3 循環を回避するための処置にはかなりの時間を要する。

属性文法の利点のひとつに、抽象構文木を決定すれば、その後の属性と意味規則割り付けは、処理させたい内容ごとに独立した記述が行なえることがある。これは、複数人数での並行開発や、すでに存在する記述に新しい機能を付加するような場合には、特に有利な特徴である。他人や過去の記述内容に関係なく、個人の記述が行なえるからである。

我々の危惧するのは、type 3 循環の問題により、この利点が失われることである。type 3 循環は、属性の数が増えるほど、独立した属性の依存関係が増えるほど、検出されやすくなるからである。このため、規模の大きな、複数人での作業が必要となることが予想される、リポジトリ記述などの問題に ordered 属性文法を適用するには、type 3 循環問題に対する何らかの対策が必要である。

6.5.3 動的計算に関するアルゴリズムの改良

本論文では、ある程度動的計算のアルゴリズム改良についてもふれたが、基本的に静的計算を実現する LCDIA アルゴリズムの効率的な実現に関する内容が主である。4.2 節での議論の通り、動的計算に関しては、まだ改良の余地が大いに残っており、この効率的な実現も今後の課題として残っている。

6.6 計算モデル上の問題

OOAG によるリポジトリを本格的なソフトウェア開発環境に適用するには、OOAG の計算モデル的問題も多少残っている。そのうちのもっとも大きな問題は、多人数での開発環境に適用した場合の排他制御の問題（ロック機構の有無）である。この件に関しては、文献 [松塚 96] において OOAG の計算モデルを拡張して、複数人数でのソフトウェア開発環境記述に対応するための研究が行なわれている。

謝辞

本研究を行なうにあたり終始変わらぬ御指導を賜わりました片山卓也教授には心から深く感謝申し上げます。また、今泉貴史先生には、OSL 言語の仕様の変更、TCP/IP ベースの OSL プログラミング環境の構築など多くの作業を手伝っていただきました。OOAG のプロトタイプ処理系の設計者である権藤克彦助手には、プロトタイプ処理系の属性評価器の設計や属性評価アルゴリズムに関して多くの助言をいただきました。MAGE プロジェクトのメンバーに感謝いたします。この研究はメンバーの協力なしにはできなかったでしょう。特に、片山研究室の飯田君には、型システムの導入、トランスレータの機能拡張、数々の評価器のデバッグなど多くの作業を手伝ってもらいました。どうもありがとう。

最後になりましたが、片山研究室、米崎研究室、そして徳田研究室の皆さんには、いつも貴重な助言、意見、議論をいただきました。

付 録A　現実のコード生成に関する解説

ここまでの、高速化技術実現した実際のコードで見てみる。Node 基底クラスに関連する事項は、ライブラリとして実現されている。最初に、Node クラスの基本的な構造の概略を示す（A.1節）。その後サンプルの OSL 記述を示し（A.2節）、これから変換されるコード例とそれについての要点を解説する（A.3節、 A.4節）。

A.1　Node 基底クラスのデータ定義部

グラフ演算の途中経過を保存する一時的属性依存グラフ以外は、すべて Node レコード内に領域を取っている。これらは、永続記憶に一括して割り当てられるデータである。

途中経過を保存する領域は、必要な場合に永続記憶ではない一時的領域に必要に応じて割り当てを行なう（Node クラスのコンストラクタ（36 ～ 43行目 `Node::Node(...)`）とデストラクタ（44 ～ 51行目 `Node::~Node()`）。

この「永続的なデータ」と「一時的データ」の区別は、非常に重要である。ここでは、データ定義部だけを示した。メソッド宣言部は省略している。

```
 1 class Node : public BaseType {
 2   public:
 3     String              lhs_name;
 4     String              rhs_name;
 5     AttrGraph           *low_c;
 6     AttrGraph           *up_c;
 7     AttrGraph           *attr_expanded_c;
 8     AttrGraph           *attr_unexpanded_c;
 9     AttrGraph           *dep;

10     Set_of_p<Attribute> syn_vertex;
11     Set_of_p<Attribute> inh_vertex;
12     AttrGraph           local_model;
13     Set_of_p<Attribute> mark_list;
14     Set_of_p<Attribute> in_set;
15     Set_of_p<Attribute> out_set;
16     Set_of_p<Attribute> local_set;
17     Set<int>            raise_set;
18     Set_of_p<RaiseSetElem> all_raise_set;
```

```
19      Block<Transition> transition;
20      Set_of_p<Message> arrived_message;
21                                  //  When a message is received, store its pointer.

22      Attribute   *attr_self;
23      Attribute   *attr_native;
24      Attribute   *attr_static;
25      Attribute   *attr_local;
26      Attribute   *new_attr_native;

27      int         n_native;       // 子 Node の数
28      int         n_static;       // 静的属性の数（相続，合成含めて）
29      int         n_local;        // 局所属性の数

30      _PARENT_NODE parent[NUM_OBJ];

31      Node(int num_transitions);
32      virtual ~Node();
33      void fetch_attr(Attribute *);
34      void graft(Attribute &, Attribute &);
35  };

36  inline Node::Node(int num_transitions) : transition(num_transitions)
37  {
38      low_c = new AttrGraph();
39      up_c = new AttrGraph();
40      attr_expanded_c = new AttrGraph();
41      attr_unexpanded_c = new AttrGraph();
42      dep = new AttrGraph();
43  }

44  inline Node::~Node()
45  {
46      delete low_c;
47      delete up_c;
48      delete attr_expanded_c;
49      delete attr_unexpanded_c;
50      delete dep;
51  }
```

A.2 サンプルの OSL 記述

以降のトランスレート例の元になる OSL 記述を示す。

```
1  class I(int scale|)
2    → I1[I(int scale|), D(int scale|)]
3    {
4      I$2.scale = I$1.scale + 1;
5      D.scale = I$1.scale;
6    }

7      I$1:value(int radix|int val)
8        ⇒ I$2:value(int radix|int val), D:value(int radix|int val)
9        {
10         I$2.radix = I$1.radix;
11         D.radix = I$1.radix;
12         I$1.val = I$2.val + D.val;
13       }

14     I$1:change(int change_pos, D new_digit|)
15       case (I$1.change_pos == I$1.scale)
16         ⇒
17         {
18           (new D) = I$1.new_digit;
19         }

20       otherwise
21         ⇒ I$2:change(int change_pos, D new_digit|)
22         {
23           I$2.change_pos = I$1.change_pos;
24           I$2.new_digit = I$1.new_digit;
25         }
```

A.3 生成された RHS クラスの例

ここで注目しなければならないのは「属性配列の割り付け（12 ～ 14 行目）」と「属性評価関数の宣言（21 ～ 31 行目）」である。16 ～ 20 行目のオブジェクトの初期化・消滅に関する関数は、付録 A.4 で説明している。

この段階で、割り当てるべき領域の大きさが、トランスレート時の解析で決定していることが重要である。

オブジェクトの初期化時に、子オブジェクトの ID が要求されていることにも注意する。複合固有属性は、必ず子孫側から実体化される。

```
 1 #include "lhs_I.h"
 2 #undef N_NATIVE
 3 #undef N_STATIC
 4 #undef N_TRANSITIONS
 5 #define NULL 0
 6 #define N_NATIVE 2
 7 #define N_STATIC 3
 8 #define N_TRANSITIONS 3
 9 class I1 : public I
10 {
11     private:
12         Attribute attr[1+N_NATIVE+N_STATIC];
13         Attribute new_native[N_NATIVE + 1];
14         Attribute dummy[N_TRANSITIONS];
15     public:
16         I1(Node *I, Node *D);
17         ~I1();
18         void setup_attribute_backpointer();
19         void initialize_transitions();
20         void initialize_transitions2();
21         friend int eval_I$2_scale_I1(Node *);
22         friend int eval_D$1_scale_I1(Node *);
23         friend int transition0_I1(Node *);
24         friend int eval_I$2_radix_I1(Node *);
25         friend int eval_D$1_radix_I1(Node *);
26         friend int eval_I$1_val_I1(Node *);
27         friend int transition1_I1(Node *);
28         friend int eval__new_D$1_tr1_I1(Node *);
29         friend int transition2_I1(Node *);
30         friend int eval_I$2_change_pos_I1(Node *);
31         friend int eval_I$2_new_digit_I1(Node *);
32 };
```

A.4 RHS オブジェクトの初期化例

ここでは、以下の点に注意する。

- オブジェクトの初期化（3 ～ 38行目 I1::I1(...)）における属性インスタンスフィールドその他の初期化、その依存グラフ構築など（4.3.4節参照）

- setup_attribute_backpointer()（40 ～ 50行目）

  - **子オブジェクトの attr フィールドを設定している点。**
    これは、属性インスタンスの双方向リンク構築のためのものである。
  - **オブジェクトの置換用の new_native 配列の使い方。**
    new_attr_native = new_native[1] である。new_native[0] は、自身の置換用、new_native[1] 以降は、子オブジェクトの置換用の一時的フィールドである。

- initialize_transitions()（51 ～ 69行目）
  存在するメッセージの個数にあわせて、領域割り当てを行なっている。動的属性のフィールドも割り当てている。

- initialize_transitions2()（70 ～ 131行目）
  まず、すべての新しいオブジェクト new_native[*n*].value1 に対して initialize_transitions() を行ない、子オブジェクトのメッセージ、及び動的属性フィールドの初期化を行なう。

  次に、各出力メッセージを新しい子オブジェクトの入力メッセージにリンクする。この操作には、子オブジェクトの型を得る必要があり、ここでは型名の文字列比較により型を得ている。これは、実行時にならないと現実の子オブジェクトの型は判明しないためである。

  最後は、トランジションごとの動的属性の依存関係グラフを構築している。

- eval_....(Node *)（132 ～ 210行目）
  それぞれの静的・動的意味規則に対応する属性値の評価関数である。

- transition*n*_*RHSname*(Node *)（211 ～ 237行目）
  case 文によるトランジションの条件分岐を処理するガード関数である。

```
1 #include "rhs_I1.h"
2 #include "Scheduler.h"
3 I1::I1(Node *I__$1, Node *D__$2):I(N_TRANSITIONS)
4 {
5     rhs_name = "I1";
```

```
 6     I__$1->parent[0].p = this; I__$1->parent[0].name = rhs_name;
 7     new_native[1].value1 = I__$1;
 8     D__$2->parent[0].p = this; D__$2->parent[0].name = rhs_name;
 9     new_native[2].value1 = D__$2;
10     attr_self = &attr[0];
11     attr_native = &attr[1];     n_native = N_NATIVE;
12     attr_static = &attr[N_NATIVE+1];   n_static = N_STATIC;
13     attr_local = NULL; n_local = 0;
14     new_attr_native = new_native + 1;
15     attr_self->value1 = this;
16     attr[0].init(this,"I",1,ATTR_N_TREE,NULL);
17     new_native[0].init(this,"I",1,ATTR_N_TREE,NULL);
18  /*native init*/
19     new_native[1].init(this,"I__$1",0,I__$1->attr_self->type, NULL);
20     new_native[2].init(this,"D__$2",0,D__$2->attr_self->type, NULL);
21  /*static attr init*/
22     attr[N_NATIVE+1].init(this,"I$1.scale",1,ATTR_S_INH,NULL);
23     attr[N_NATIVE+1].value1 = new type_int;
24     attr[N_NATIVE+2].init(this,"I$2.scale",0,ATTR_S_INH,eval_I$2_scale_I1);
25     attr[N_NATIVE+2].value1 = new type_int;
26     attr[N_NATIVE+3].init(this,"D$1.scale",0,ATTR_S_INH,eval_D$1_scale_I1);
27     attr[N_NATIVE+3].value1 = new type_int;
28     setup_attribute_backpointer();
29  /* initialize variables. */
30     dep->insert(new Dependency(&attr[N_NATIVE+1], &attr[N_NATIVE+2], direct));
31     dep->insert(new Dependency(&attr[N_NATIVE+1], &attr[N_NATIVE+3], direct));
32     inh_vertex.insert(&attr[N_NATIVE+1]);
33     in_set.insert(&attr[N_NATIVE+1]);
34     out_set.insert(&attr[N_NATIVE+2]);
35     out_set.insert(&attr[N_NATIVE+3]);
36  /* raise_set setup */
37     init_attr();
38  }

39  I1::~I1(){}

40  inline void I1::setup_attribute_backpointer()
41  {
42     attr[N_NATIVE+2].value2 = &((Node*)new_native[1].value1)->attr_static[1-1];
43     attr[N_NATIVE+2].value2->value2 = &attr[N_NATIVE+2];
44     attr[N_NATIVE+3].value2 = &((Node*)new_native[2].value1)->attr_static[1-1];
45     attr[N_NATIVE+3].value2->value2 = &attr[N_NATIVE+3];
46     new_native[1].value2 = &((Node*)new_native[1].value1)->new_attr_native[-1];
47     new_native[1].value2->value2 = &new_native[1];
48     new_native[2].value2 = &((Node*)new_native[2].value1)->new_attr_native[-1];
49     new_native[2].value2->value2 = &new_native[2];
```

```
50 }

51 void I1::initialize_transitions()
52 {
53     transition[0].initialize(1,2);
54     Message *m00 = new Message(1,1,NULL,this,-1,0,"value");
55     m00->in_attr[0].init(NULL,"I$1.radix",0,ATTR_D_INH,NULL);
56     m00->in_attr[0].value1 = new type_int;
57     m00->out_attr[0].init(NULL,"I$1.val",0,ATTR_D_SYN,eval_I$1_val_I1);
58     m00->out_attr[0].value1 = new type_int;
59     transition[0].set_imsgs(m00);
60     Message *m10 = new Message(2,0,NULL,this,-1,1,"change");
61     m10->in_attr[0].init(NULL,"I$1.change_pos",0,ATTR_D_INH,NULL);
62     m10->in_attr[0].value1 = new type_int;
63     m10->in_attr[1].init(NULL,"I$1.new_digit",0,ATTR_D_INH,NULL);
64     transition[1].initialize(1,0);
65     transition[1].set_imsgs(m10);
66     transition[2].initialize(1,1);
67     transition[2].set_imsgs(m10);
68     initialize_transitions2();
69 }

70 void I1::initialize_transitions2()
71 {
72     int i, j;
73     for (i=1; i<N_NATIVE+1; i++)
74         ((Node*)new_native[i].value1)->initialize_transitions();
75                                 /*Never reach here*/
76     Message *m00 = (((Node*)new_native[1].value1)->rhs_name == "I1") ?
77         ((Node*)new_native[1].value1)->transition[0].in_messages[0] :
78         ((((Node*)new_native[1].value1)->rhs_name == "I2") ?
79          ((Node*)new_native[1].value1)->transition[0].in_messages[0] :
80          ((Message*)-1));
81     m00->in_attr[0].eval = eval_I$2_radix_I1;
82     Message *m01 = (((Node*)new_native[2].value1)->rhs_name == "D1") ?
83         ((Node*)new_native[2].value1)->transition[0].in_messages[0] :
84         ((Message*)-1);
85     m01->in_attr[0].eval = eval_D$1_radix_I1;
86     transition[0].set_omsgs(m00,m01);
87     transition[1].set_omsgs();
88     Message *m20 = (((Node*)new_native[1].value1)->rhs_name == "I1") ?
89         ((Node*)new_native[1].value1)->transition[1].in_messages[0] :
90         ((((Node*)new_native[1].value1)->rhs_name == "I2") ?
91          ((Node*)new_native[1].value1)->transition[1].in_messages[0] :
92          ((Message*)-1));
93     m20->in_attr[0].eval = eval_I$2_change_pos_I1;
```

```
94      m20->in_attr[1].eval = eval_I$2_new_digit_I1;
95      transition[2].set_omsgs(m20);
96      for (i=0; i<N_TRANSITIONS; i++)
97      {
98          for (j=0; j<transition[i].out_messages.size(); j++)
99          {
100             transition[i].out_messages[j]->sender = this;
101             transition[i].out_messages[j]->sender_transition = i;
102         }
103     }
104     dummy[0].init(this, "", 0, ATTR_N_NEW,transition0_I1);
105     transition[0].dependency.remove_all();
106     transition[0].dependency.insert(&dummy[0]);
107     transition[0].dependency.insert(
108         new Dependency(&transition[0].in_messages[0]->in_attr[0],
109                        &transition[0].out_messages[0]->in_attr[0], direct));
110     transition[0].dependency.insert(
111         new Dependency(&transition[0].in_messages[0]->in_attr[0],
112                        &transition[0].out_messages[1]->in_attr[0], direct));
113     transition[0].dependency.insert(
114         new Dependency(&transition[0].out_messages[0]->out_attr[0],
115                        &transition[0].in_messages[0]->out_attr[0], direct));
116     transition[0].dependency.insert(
117         new Dependency(&transition[0].out_messages[1]->out_attr[0],
118                        &transition[0].in_messages[0]->out_attr[0], direct));
119     dummy[1].init(this, "", 0, ATTR_N_NEW,transition1_I1);
120     transition[1].dependency.remove_all();
121     transition[1].dependency.insert(&dummy[1]);
122     dummy[2].init(this, "", 0, ATTR_N_NEW,transition2_I1);
123     transition[2].dependency.remove_all();
124     transition[2].dependency.insert(&dummy[2]);
125     transition[2].dependency.insert(
126         new Dependency(&transition[1].in_messages[0]->in_attr[0],
127                        &transition[2].out_messages[0]->in_attr[0], direct));
128     transition[2].dependency.insert(
129         new Dependency(&transition[1].in_messages[0]->in_attr[1],
130                        &transition[2].out_messages[0]->in_attr[1], direct));
131 }

132 int eval_I$2_scale_I1(Node *obj)
133 {
134     I1 *target = (I1 *)obj;
135     type_int oldvalue = *(type_int*)target->attr[N_NATIVE+2].value1 ;
136     type_int *newvalue = (type_int*)target->attr[N_NATIVE+2].value1 ;
137     newvalue->value = (*(type_int*)target->attr[N_NATIVE+1].value1).value + 1;
138     target->attr[N_NATIVE+2].set_evalflag(1);
```

```
139     if (oldvalue == *newvalue) return 0;
140     else
141         return 1;
142 }

143 int eval_D$1_scale_I1(Node *obj)
144 {
145     I1 *target = (I1 *)obj;
146     type_int oldvalue = *(type_int*)target->attr[N_NATIVE+3].value1 ;
147     type_int *newvalue = (type_int*)target->attr[N_NATIVE+3].value1 ;
148     newvalue->value = (*(type_int*)target->attr[N_NATIVE+1].value1).value;
149     target->attr[N_NATIVE+3].set_evalflag(1);
150     if (oldvalue == *newvalue) return 0;
151     else
152         return 1;
153 }

154 int eval_I$2_radix_I1(Node *obj)
155 {
156     I1 *target = (I1 *)obj;
157     Transition *tr = & obj->transition[0];
158     (*(type_int*)tr->out_messages[0]->in_attr[0].value1).value =
159       (*(type_int*)target->transition[0].in_messages[0]->in_attr[0].value1).value;
160     tr->out_messages[0]->in_attr[0].set_evalflag(1);
161     return 1;
162 }

163 int eval_D$1_radix_I1(Node *obj)
164 {
165     I1 *target = (I1 *)obj;
166     Transition *tr = & obj->transition[0];
167     (*(type_int*)tr->out_messages[1]->in_attr[0].value1).value =
168       (*(type_int*)target->transition[0].in_messages[0]->in_attr[0].value1).value;
169     tr->out_messages[1]->in_attr[0].set_evalflag(1);
170     return 1;
171 }

172 int eval_I$1_val_I1(Node *obj)
173 {
174     I1 *target = (I1 *)obj;
175     Transition *tr = & obj->transition[0];
176     (*(type_int*)tr->in_messages[0]->out_attr[0].value1).value =
177         (*(type_int*)target->
178          transition[0].out_messages[0]->out_attr[0].value1).value +
179         (*(type_int*)target->
180          transition[0].out_messages[1]->out_attr[0].value1).value;
```

```
181     tr->in_messages[0]->out_attr[0].set_evalflag(1);
182     return 1;
183 }

184 int eval_new_D$1_tr1_I1(Node *obj)
185 {
186     I1 *target = (I1 *)obj;
187     Transition *tr = & obj->transition[1];
188     target->new_native[2].value1 =
189         (Node *)target->transition[1].in_messages[0]->in_attr[1].value1;
190     target->new_native[2].value1->value2 = target->new_native[2].value1;
191     return 1;
192 }

193 int eval_I$2_change_pos_I1(Node *obj)
194 {
195     I1 *target = (I1 *)obj;
196     Transition *tr = & obj->transition[2];
197     (*(type_int*)tr->out_messages[0]->in_attr[0].value1).value =
198       (*(type_int*)target->transition[1].in_messages[0]->in_attr[0].value1).value;
199     tr->out_messages[0]->in_attr[0].set_evalflag(1);
200     return 1;
201 }

202 int eval_I$2_new_digit_I1(Node *obj)
203 {
204     I1 *target = (I1 *)obj;
205     Transition *tr = & obj->transition[2];
206     tr->out_messages[0]->in_attr[1].value1 =
207         (Node *)target->transition[1].in_messages[0]->in_attr[1].value1;
208     tr->out_messages[0]->in_attr[1].set_evalflag(1);
209     return 1;
210 }

211 int transition0_I1(Node *)
212 {
213     return 0;                   /* nothing to do */
214 }

215 int transition1_I1(Node *obj)
216 {
217     I1 *target = (I1*)obj;
218     if ((*(type_int*)target->transition[1].in_messages[0]->
219          in_attr[0].value1).value ==
220         (*(type_int*)target->attr[N_NATIVE+1].value1).value)
221     {
```

```
222         eval_new_D$1_tr1_I1(obj);
223     }
224     else
225     {
226         extern Scheduler fireScheduler;
227         fireScheduler.put_signal(target,_fire,2);
228     }
229     return 0;
230 }

231 int transition2_I1(Node *obj)
232 {
233     I1 *target = (I1*)obj;
234     eval_I$2_change_pos_I1(obj);
235     eval_I$2_new_digit_I1(obj);
236     return 0;
237 }
```

関連図書

[Atr92] Atria Software, Inc. *ClearCase Concepts Manual*, 1992.

[A.V90] Scott A.Vorthmann. Coordinated incremental attribute evaluation on a dr-threaded tree. In *Proc. Int. Conf. WAGA (Paris, France)*, Vol. 461 of *Lec. Notes in Comp. Sci.*, pp. 207–221. Springer-Verlag, 1990.

[BE86] N. Belkhatir and J. Estublier. Experience with a data base of programs. In *SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments*, pp. 84–91, Palo Alto, CA, December 1986. Association for Computing Machinery, SIGPlan.

[Ber87] Philip A. Bernstein. Database system support for software engineering. In *Proceedings of the 9th International Conference on Software Engineering*, March 1987.

[Bet85] David Betz. *XLISP: An Experimental Object Oriented Language*, January 1985.

[BGMT88] Gerard Boudier, Ferdinando Gallo, Régis Minot, and Ian Thomas. An Overview of PCTE and PCTE+. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 248–257, November 1988.

[CBD+94] Gerome Canals, Nacer Boudjlida, Jean-Claude Derniame, Claude Godart, and Jacques Lonchamp. ALF: A Framework for Building Process-Centerd Software Engineering Environments. In *Software Process Modelling and Technology*. RESEARCH STUDIES PRESS LTD., 1994.

[CDG+89] Reidar Conradi, Tor Martin Didriksen, Bjørn Gulla, Even-André Karlsson, Anund Lie, Per Harald Westby, Svein Olav Hallsteinsen, Per Holager, Ole Solberg, and Asbjørn Thomassen. Design of the kernel EPOS software engineering environment. In *Proceedings of the International Conference on System Development Environments and Factories*, Berlin, Germany, May 1989.

[CDG+90] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems.* Morgan-Kaufman, 1990.

[CL93] Sheng-Yang Chiu and Roy Levin. The Vesta Repository: A File System Extension for Software Development. Technical report, Digital Equipment Corporation, June 1993.

[EF82] Joost Engelfriet and Gilberto Filé. Simple Multi-Visit Attribute Grammars. *Journal of Computer and System Sciences*, Vol. 24, No. 3, pp. 283–314, June 1982.

[Fel79] Stuart I. Feldman. Make - a program for maintaining computer programs. In *SOFTWARE - PRACTICE AND EXPERIENCE*, Vol. 9, pp. 255–265, Bell Laboratories, Murray Hill, New Jersey 07974. U.S.A., April 1979.

[Gem95] GemStone Systems, Inc. *GemStone Programming Guide (Version 4.1)*, 1995.

[GISK93] Katsuhiko Gondow, Takashi Imaizumi, Youichi Shinoda, and Takuya Katayama. Change Management and Consistency Maintenance in Software Development Environments Using Object Oriented Attribute Grammars. In *Object Technologies for Advanced Software*, pp. 77–94. Springer-Verlag, 1993. LNCS 742.

[GKL94] David Garlan, Charles W. Krueger, and Barbara Staudt Lerner. Transformgen: Automating the maintenance of structure-oriented environments. *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 3, pp. 727–774, May 1994.

[Gla75] Alan L. Glasser. The source code control system. In *IEEE Transactions on Ssoftware Engineering*, pp. 364–370, December 1975.

[Gra92] GrammaTech, Inc., One Hopkins Place, Ithaca, NY 14850, U.S.A. *The Synthesizer Generator Reference Manual*, fourth edition, 1992.

[Hed94] Görel Hedin. An overview of door attribute grammars. In Peter A. Fritzson, editor, *Compiler Construction*, Vol. 786 of *Lec. Notes in Comp. Sci.*, pp. 31–51. Springer-Verlag, 4 1994.

[Hei92] Dennis Heimbigner. Experiences with an object manager for a process-centerd environment. Technical report, Computer Science Dept. University of Colorado, 1992.

[HK86] Scott E. Hudson and Roger King. CACTIS: A Database System for Specifying Functionally-Defined Data. In *Proceedings International Workshop on Object-Oriented Database Systems*, 1986.

[HSM89] H.H.Vogt, S.D.Swieerstra, and M.F.Kuiper. Higher order attribute grammars. In *ACM SIGPLAN '89 Conf. on Progr. Languages Design and Implementation*, Vol. 24, pp. 131–145, Portland, Oregon, July 1989.

[ISK90] Takashi Imaizumi, Youichi Shinoda, and Takuya Katayama. Description and implementation of file management system using attribute grammars. In *Proceedings of an 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, October 1990.

[Kas80] Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, Vol. 13, No. 3, pp. 229–256, 1980.

[KAW93] N. Kiesel, A. Schürr, and B. Westfechtel. GRAS, A Graph-Oriented Database System for Engineering Applications. In Jarzabek Lee, Reid, editor, *CASE '93 6th Int. Conf. on Computer-Aided Software Engineering*, pp. 272–286. IEEE Computer Society Press, 1993.

[KK93] Gail E. Kaiser and Simon M. Kaplan. Parallel and distributed incremental attribute evaluation algorithms for multiuser software development environments. *ACM Trans. on Softw. Eng. and Methodology*, Vol. 2, No. 1, pp. 47–92, January 1993.

[KL89] Won Kim and Frederick H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications.* acm press, 1989.

[KLMM93] J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, and B. Magnusson. *OBJECT-ORIENTED ENVIRONMENTS: THE MJØLNER APPROACH.* PRENTICE HALL, 1993.

[Knu68] D.E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, Vol. 2, No. 2, pp. 127–145, 1968.

[MI94] 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, July 1994.

[MK93] Josephine Micallef and Gail E. Kaiser. Support algorithms for incremental attribute evaluation of asynchronous subtree replacement. *IEEE Trans. on Softw. Eng.*, Vol. 19, No. 3, pp. 231–252, March 1993.

[Not85] David Notkin. The GANDALF project. *The Journal of Systems and Software*, Vol. 5, No. 2, May 1985.

[Obj93] Object Design, Inc., 25 Burlington Mall Road Burlington, MA 01803. *ObjectStore User Guide: Library Interface*, December 1993.

[PS92] B. Peuschel and W. Schafer. Concepts and implementation of a rule-based process engine. In *Proceedings of the* 14^{th} *International Conference on Software Engineering*, pp. 262–279, May 1992.

[Rep82] Thomas Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Conference Record of the 9th ACM Symposium on Principles of Programming Languages*, pp. 169–176. Albuquerque, NM, January 1982.

[RT87] T. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual.* Depertment of Computer Science, Cornell University, Ithaca, NY., second edition, July 1987.

[RT89] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, 1989.

[Shi89] Yoichi Shinoda. *On Application of Attribute Grammars to Software Development.* PhD thesis, Tokyo Institute of Technology, 3 1989.

[SK90a] Yoichi Shinoda and Takuya Katayama. Object Oriented Extension of Attribute Grammars and Its Implementation Using Distributed Attribute Evaluation Algorithm. In *Proceedings of the International Workshop on Attribute Grammars and their Applications,* Lecture Note in Computer Science *Vol. 461*, pp. 177–191. Springer-Verlag, 1990.

[SK90b] Masato Suzuki and Takuya Katayama. Redoing: A mechanism for dynamics and flexibility of software process. In *Proceedings of an International Conference organized by the IPSJ to Commemorate the 30th Anniversary*, pp. 151–160, 1990.

[TBC+88] R.N. Taylor, F.C. Belz, L.A. Clarke, L.J. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf, and M. Young. Foundations for the Arcadia Environment

Architecture. In *Proceedings SIGSOFT '88: Third Symposium on Software Development Environments*, pp. 1–13, November 1988.

[TC90a] Tim Teitelbaum and R. Chapman. Higher-order attribute grammars and editing environments. In *Proc. ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation, Vol. 25, No. 6*, pp. 197–208, White Plains, NY, 1990.

[TC90b] Tim Teitelbaum and Richard Chapman. Higher-order attribute grammars and editing environments. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pp. 197–208. ACM, June 1990. White Plains, New York.

[TC93] Peri Tarr and Lori A. Clarke. PLEIADES: An object management system for software engineering environments. Technical report, Software Development Laboratory, Department of Computer Science, University of Massachusetts, Amherst, MA 01003, July 1993.

[Tic85] Walter F. Tichy. RCS – A system for version control. *Software – Practice and Experiences*, Vol. 15, No. 7, pp. 637–654, July 1985.

[TW93] Takehiro Tokuda and Yoshimichi Watanabe. An attribute grammar modelling of interactive figures. In *Information Modelling and Knowledge Bases V (The Third European-Japanese Seminar on Information Modelling and Knowledge Bases*, pp. 214–228, 1993.

[VSK89] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pp. 131–145. ACM, June 1989. Portland, Oregon.

[ZM90] S. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems.* Morgan-Kaufman, 1990.

[吉田 95] 吉田敦, 山本晋一郎, 阿草清滋. CASE ツール開発のためのソフトウェア操作言語. 情報処理学会論文誌, Vol. 36, No. 10, pp. 2433–2441, 1995.

[権藤 91] 権藤克彦. オブジェクト指向属性文法 OOAG とその実現法の研究. Master's thesis, 東京工業大学 情報工学科, March 1991.

[権藤 95] 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, May 1995.

[溝口 88] 溝口敏夫. 情報資源辞書システム (IRDS). 情報処理, Vol. 29, No. 3, 1988.

[今泉 92] 今泉貴史. 属性文法に基づくソフトウェア コンフィギュレーション システム. PhD thesis, 東京工業大学, 目黒区大岡山 2-12-1, January 1992.

[今泉 95] 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, May 1995.

[篠木 93] 篠木祐二, 西尾高典, 吉川彰弘. CDIF — CASE データ交換形式. コンピュータソフトウェア, Vol. 10, No. 2, pp. 13 – 25, March 1993.

[松塚 96] 松塚貴英. 複数ユーザによるソフトウェア開発に対応する計算モデル OOAG/M の研究. Master's thesis, 東京工業大学大学院情報理工学研究科 計算工学専攻, March 1996.

[中井 96] 中井央, 佐々政孝, 山下義行, 中田育男. LR 属性文法に基づいたインクリメンタルな属性評価. 情報処理学会論文誌, Vol. 37, No. 12, pp. 2254–2265, Dec. 1996.

[萩原 93] 萩原威志. オブジェクト指向属性文法 OOAG の実行系の高速化の研究. Master's thesis, 東京工業大学 情報工学科, 1993.

[萩原 94] 萩原威志, 片山卓也. オブジェクト指向属性文法 OOAG の高速な処理系の設計と実現. 電子情報通信学会技術研究報告, Vol. 94, No. 135, pp. 17–24, 1994.

[萩原 96] 萩原威志, 片山卓也. 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. 情報処理学会論文誌, Vol. 37, No. 12, pp. 2362–2375, December 1996.