属性文法型計算モデルによる版・構成管理システムの記述

今泉貴史[†] 篠田陽一[††] 片山卓也[††,†††]

属性文法は，木構造上のノードに属性を張り付け，その上での属性計算を用いてさまざまな性質を表現しようとするものである．したがって，基本構造として木構造を持つオブジェクトは，属性文法を用いてその性質を記述することが可能である．本論文では，プログラムの開発時に使用されるツールである版管理システムと構成管理システムとを木構造を扱うシステムであると見なし，その属性文法による記述に関して報告する．これらのシステムは木構造を変更しながらユーザとのインタラクションにより処理を進めるシステムであり，従来は属性文法で記述するのは困難と考えられていた．本論文では，このような問題に対しても属性文法によるアプローチが有効であることを，具体的なシステムの記述を通して示す．インタラクティブなシステムを記述するために属性文法に必要とされる機構についても考察する．

Description of Version/Configuration Control System Using Attribute Grammars

TAKASHI IMAIZUMI,[†] YOICHI SHINODA[††] and TAKUYA KATAYAMA[††,†††]

Attribute grammars are formal-models for describing the semantics of formal languages and have been applied to the description and construction of compilers. However, since their principle is to attach attributes on the syntax tree and to evaluate them in a certain way, they could be essentially useful for manipulating tree-structured objects. In this paper, we describe the design and implementation of a version/configuration control system using attribute grammars. These systems manipulate tree-structured objects, but users modify these tree-structured objects also. So it is assumed that writing these systems in attribute grammars is not easy. Through out experiment of this system, we verified that attribute grammars are useful for designing these systems.

1. はじめに

属性文法（Attribute Grammars）は，プログラミング言語の意味記述のためにD.E. Knuthにより導入された形式的体系であり，以来，特にコンパイラの記述とその自動生成を目的とした研究が続けられてきた[1)~5)]．しかし，属性文法の応用はコンパイラだけに限定されるだけでなく，CADシステム[6)]，プログラミング言語[7)]，構造エディタ生成系[8)]などにも応用されている．

言語処理においては，処理されるべき記号列を構文解析して得られる構文木の上で属性計算がおこなわれる．構文木の作成と属性計算は独立した処理であり，内部構造として木構造を持ち，その木構造の上でなんらかの演算を施す場合には，属性文法を応用することが可能であると考えられる．属性文法の特徴として記述の局所性や読解性が高いことが知られており，属性文法を応用すれば，保守性の高いソフトウェアの作成を期待できる．

我々は，木構造を操作するシステムの記述の可能性を確認するために，版管理システムと構成管理システムの機能を併せ持つ版・構成管理システムSSDS（Simple Software Development System）を属性文法を用いて開発した．版管理システムでは，それぞれのバージョンを並べたバージョン木と呼ばれる木構造を操作し，構成管理システムでは，プログラムの構成要素の依存関係を木構造としたもの（依存木）を操作する．これらのシステムが取り扱う構造は木構造であり，属性文法を用いて記述することが可能であると考えられる．

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

しかし，版管理システムでのバージョン木や構成管理システムでの依存木は，ユーザの操作によりしばしば変更される．通常の属性文法では，すでに構築された木構造の上で計算をおこなうため，木構造が変更され続けるようなシステムを記述することは困難である．この点を解決するために，高階属性文法[9),10)]やTransformational Attribute Grammar[11)]が提案されている．しかしいずれの場合も，木構造の変更は文法中に静的に記述された範囲でしかおこなえないため，ユーザのインタラクティブな操作により木構造を変更し続けるシステムには適当ではない．

属性文法にもとづく構造エディタ生成系であるSynthesizer Generator[8)]は，編集対象となる言語の意味を構文木上の属性計算により指定して，構造エディタを生成するシステムである．プログラムが編集された場合，編集作業により変更された木構造の上でインクリメンタルに属性計算がおこなわれる[12)]．このため，Synthesizer Generatorはインタラクティブなシステムの記述に用いることができる．

本論文では，Synthesizer Generatorを用いて版・構成管理システムを記述する．その際，Synthesizer Generatorの持つ機能の中で純粋な属性文法で記述できる範囲と，属性文法以外の手法を用いる部分を明確にする．また，構文木を保存しながらインタラクティブな操作により内容が変更されるようなシステムを，属性文法を用いて記述するために必要な機構についても明らかにする．

本論文では，2章で対象とする属性文法について述べる．3章では版・構成管理システムの各機能を属性文法にマッピングする．4章では版・構成管理システムの記述について述べ，5章では，2章で定義した属性文法に関して考察する．

2. 版・構成管理システム記述のための属性文法

2.1 属性文法の基本モデル

属性文法は，文脈自由文法に属性と属性間の関係を記述した意味規則を付加したものである．

属性文法 = 文脈自由文法 + 属性 + 意味規則

入力記号列をパーズした結果として構文木が構築された後に属性を構文木に張り付ける．この属性が付けられた構文木を属性付き木と呼ぶが，属性文法における属性評価（木に付けられた属性の評価）は，この属性付き木の上のすべての属性の値を決定することになる．

属性には構文木中を下向きに流れるものと上向きに流れるものの2種類がある．下向きに流れる属性を相続属性（inherited attribute），上向きに流れる属性を合成属性（synthesized attribute）と呼ぶ．一つの属性は必ずこのうちのいずれかである．一つの生成規則に付けられる意味規則においては，左辺の非終端記号の合成属性と，右辺の非終端記号の相続属性を定義する．また，この時の定義は関数的である．

2.2 基本モデルの拡張

属性文法の基本モデルをそのまま用いたのでは，版・構成管理システムのように構文木を保持しながらインタラクティブな操作により構文木や構文木上の属性を変更し続けるシステムを記述することはできない．本節では，属性文法の基本モデルに拡張をおこなう点について述べる．

2.2.1 木構造の変更

属性文法において木構造を操作する方法には，

- 属性文法の枠組を拡張して木構造を操作する方法を与える
- 属性文法の枠組の外で木構造を操作する方法を与える

が考えられる．

高階属性文法の非終端属性は，属性文法を拡張して木構造を操作する方法を与えた例である[9)]．このモデルでは，属性計算の結果として得られる木構造型の属性を，部分木として構文木の中に張りつけることができる．この方法により構築される構文木は入力列をパーズしたものとは異なるが，一旦入力列をパーズし，その構文木から得られるデータのみを用いて構文木を拡張するため，高階属性文法により得られる構文木もやはり静的なものである．つまり，入力列が決まれば構文木の形は決定されてしまい，それを変更することはできない．そのため，この方法をそのまま今回のシステムに用いることはできない．

Synthesizer Generatorの構造変更コマンドは木構造の変換方法を指定するものだが，これは属性文法の枠組を外れた部分で指定するものである．構造変更コマンドは，ある部分木に注目している時に，その部分木があらかじめ指定されているパターンと一致している時にはコマンドを起動することができる．この際，注目している部分木をあらかじめ決められている形に変形することができる．変形後の木の形を指定する場合には，パターンマッチで用いたパターン変数，パターン変数に対応するノードに付けられた属性の値，関数を用いることができるため，文脈に依存した変形も可能である．本論文では，この形式の変形を用いる．

2.2.2 属性評価アルゴリズム

木構造を変更した場合属性の値を再計算する必要が

あるが,

(1) すべての属性値を計算し直す

(2) 変更された部分木に依存する属性値のみを再計算する

の二つの方法が考えられる．(1) の方法では，おこなわれた変更が局所的なものである場合にも構文木全体の属性値を再計算しなければならないため再計算のコストが大きい．一方，(2) の方法では，変更された部分木に依存する属性を洗い出すためのコストは必要になるが，構文木が大きい場合には全体としてのコストは (1) の場合と比較して小さいものとなる．本論文では，属性計算にはインクリメンタル属性評価器[12)~14)]を用いることを仮定している．

2.2.3 ファイルシステムの扱い

属性文法は関数型の計算モデルであり,「値」を計算するためのものである．したがって，二次記憶に格納されているファイルを直接扱うことはできない．

属性文法においてファイルを扱う方法としては，

- ファイルシステムを属性付き木として表現する
- ファイルは終端記号に対応し，その合成属性がファイルの内容を表す
- ファイルシステムを表す属性を用いる
- 属性計算関数によりファイルの内容にアクセスする

などの方法が考えられる．

ファイルシステムを属性付き木として表現する場合，ファイルシステムは構文木中で一つの非終端記号として表される．この場合，この一つの属性付き木をファイルシステムにアクセスする複数の非終端記号で共有することができないため，ファイルシステムにアクセスするノードごとに別のファイルシステムを持つことになる．この場合，一つのファイルシステムに加えられた変更を，ほかのファイルシステムが認識できないという問題がある．ファイルを終端記号として表現した場合にも，同様な問題が起きる．

ファイルシステムを表す属性を設けた場合，このような問題は回避することができる．しかし，属性の流れは属性文法の中に静的に記述されるため，動的に変更されるファイルシステムを記述するのは困難である．

属性計算関数を用いてファイルの内容にアクセスする場合は，属性計算関数の関数性を損なうことになる．つまり，同じ属性計算をおこなうにも関わらず，結果として異なる値が得られることになる．しかし，属性評価器としてインクリメンタル属性評価器を仮定することによりこの問題を回避することが可能である．そこで，本論文では，この方法を用いてファイルの内容にアクセスする．したがって，ファイルの内容はファイルシステム上に存在するものをそのまま用い，属性文法内部ではファイルを表現するファイル名を用いる．

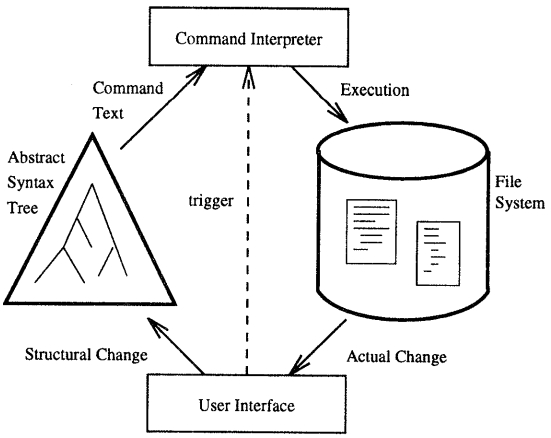


図 1 外部環境との相互作用

Fig. 1 The interaction between the system and its external environment.

ファイルを操作する場合，属性文法では，ファイルシステムを扱う処理はすべてコマンド文字列として表現し，これを外部環境に渡す．外部環境では，渡されたコマンド文字列をコマンドインタプリタにより実行することでファイルシステムを操作する（図 1）．

2.3 属性文法の記法

本システムの記述には，前節で述べた拡張機能を表現することが可能な属性文法として，構造エディタ生成系 Synthesizer Generator[8)]を使用した．ここでは以降の説明のために，Synthesizer Generator の仕様記述言語 SSL の記法について簡単に述べる．

SSL では，抽象構文によって，エディタの取り扱う抽象構文木を定義する．文脈自由文法における非終端記号をフィラ（phyla）と呼び，生成規則にラベルを付けそれをオペレータ（operator）と呼ぶ．抽象構文に付随して宣言される意味規則の部分では，言語の静的な意味が記述される．図 2 では，二つの生成規則が記述されている．

図 2 では，expr というフィラの宣言がおこなわれている．フィラ expr には Plus と Number という二つのオペレータがあり，オペレータ Plus が選択された場合はさらに二つのフィラ expr（非終端記号）に分割され，オペレータ Number の場合にはフィラ INT（終端記号）に分割されることを示している．オペレータ Plus には複数のフィラ expr が出現するが，これらは左から数えた番号を **expr$** の後ろに付けることにより識別する．また **$$** は左辺のフィラ（つまり **expr$1**

```
expr { syn INT val; };

expr : Plus(expr expr) {
          $$.val = expr$2.val + expr$3.val;
       }
     | Number(INT) {
          $$.val = INT;
       }
     ;
```

図2 抽象構文と意味規則
Fig. 2 The abstract syntax and semantic rules.

```
transform expr
    on "plus" Number(i):
          Plus(Number(i), Number(i));
```

図3 構造変更コマンド
Fig. 3 A transform command.

と同じ）を表す．

属性は，フィラ expr に対して INT 型☆の合成属性 val が定義されている（相続属性の場合，syn ではなく inh と記述する）．それぞれの意味規則の部分（{ と } に囲まれた部分）では，左辺のフィラの val 属性の値を右辺のフィラの属性の値などから計算している．Number オペレータの場合，右辺のフィラそのものの値が左辺のフィラの属性に代入されている．

ここには挙げていないが，局所属性を用いることも可能である．この属性は，フィラではなくオペレータに対して宣言するもので，属性計算の途中で一時的に値を保持するために用いることができる．

また SSL には構造変更コマンドが用意されており，これにより構文木の構造変更をおこなうことができる．構造変更コマンドでは，編集しているオブジェクトの木構造に関するパターンマッチを用いて変換を記述しておき，ユーザが指示した場合にこの変換を実行する．

図3 は，現在注目しているオブジェクトが Number (2) という構造の場合には，plus というコマンドにより Plus(Number(2), Number(2)) という構造に変更することを表している．構造変更コマンドで指定した i はパターン変数であり，この場合には 2 にマッチする．そのため，変更後の構文木においても i に対応する部分は 2 となっている．

☆ INT は，SSL で準備されている整数を表す型である．この例のように，これを終端記号に用いることもできる．複合型を定義する場合，抽象構文の定義と同様な方法でおこなう．この場合，オペレータが直和型のタグとなり，右辺の記号が直積型を構成する．

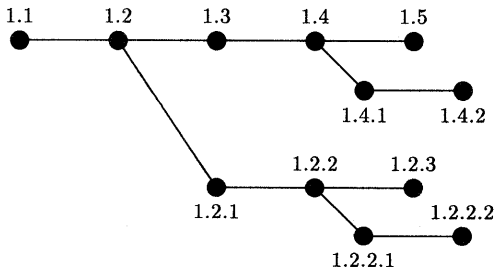


図4 バージョン木の例
Fig. 4 An example of version tree.

3. 版・構成管理システムの概略設計

本章では，本論文で取り上げる版・構成管理システム SSDS の仕様とその機能の属性文法へのマッピング方法について述べる．

3.1 版・構成管理システムの仕様

プログラム開発では，ソースコードの版管理をおこなうために RCS (Revision Control System)[15),16)]や SCCS (Source Code Control System)[17)]などの版管理システムが用いられる．また，プログラムの構成方法を管理するためには，make[18)]などの構成管理システムが用いられる．本論文で取り上げる版・構成管理システム SSDS (Simple Software Development System) は，簡単な版管理や構成管理の機構を提供するものである．

3.1.1 版管理システムの仕様

版管理システムは，次のような機能を持つ．

- 新しいバージョンを登録する
- 以前のバージョンを取り出す
- バージョンにログをつける

バージョンを登録する際には，元となるバージョンが存在するため，この関係を結ぶと一つの木構造となる．**図4** では，最初のバージョンとしてバージョン 1.1 が存在し，バージョン 1.2 はバージョン 1.1 から，バージョン 1.3 とバージョン 1.2.1 はどちらもバージョン 1.2 から作成されたことを示している．版管理システムの操作は，このバージョン木の変形やバージョン木の上での計算である．

また，バージョンの保存には，基本的には差分形式を用いる．すべてのバージョンを保持するのではなく，最初のバージョンのみを完全な形で保持し，これ以外のバージョンはすべて直前のバージョンとの差分形式で保存する．この機構により，大きなファイルの一部だけが変更された場合にも，使用するディスク領域を

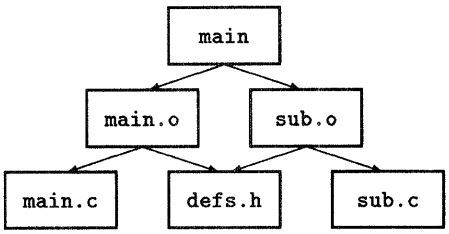


図5 依存木の例
Fig. 5 An example of dependency tree.

小さくする.

3.1.2 構成管理システムの仕様

構成管理システムは，次のような機能を持つ.

- システムの構築方法を管理する
- システム構築に必要な処理（インストラクション）のうち最小限の処理をおこなう

構成管理システムでは，最終的に作成したいシステムとそのシステムを構成するモジュールの関係，各モジュールとそのモジュールを構成するソースプログラムの関係といった依存関係をむすぶことにより，一つの木を得ることができる．これを依存木と呼ぶ（厳密には，一つのソースコードが複数のモジュールで参照されることもあり，木構造ではなくDAGとなる．DAGであることを明記する場合には，依存グラフと呼ぶ）．構成管理システムの操作は，依存木の構築とその上での計算である（図5）.

3.1.3 版管理システムと構成管理システムの連動

SSDSでは，構成管理システムの機能を用いてシステムを構築する際，版管理システムに登録されているファイルを用いることができる．この際，バージョンの指定方法としては，

- 最新のものを用いる
- バージョン番号を指定する
- 日時を指定する
- ログの条件を用いる

などが可能である．この指定をおこなうことにより，ソースファイルの指定したバージョンを用いてシステムを構築することができる.

3.1.4 ユーザの入力

バージョンに付けるログ，バージョン番号の指定，日時の指定，ログの条件，作成したいシステムの名前などは，ユーザの入力用に設けられたデータ入力部ですべてを入力する．ここで入力されたデータは，システムのほかの部分に渡され，必要に応じて参照される．これは，システムの実行時に扱われる値である.

版管理システムでは，新たにファイルを管理する場合にファイル名を入力する必要がある．この処理は，SSDSの版管理システム部でファイル名の入力をおこなう.

表1 版管理システムと属性文法のマッピング
Table 1 Mapping between version control systems and attribute grammars.

版管理システム	属性文法1	属性文法2
バージョン木	構文木	構文木
バージョン情報	属性	終端記号
バージョン間の差分	属性	属性
バージョンの回復	属性計算	属性計算＋副作用
バージョンの登録	構造変更	構造変更

構成管理システムにおける依存関係の指定では，ユーザがすべてのファイルに対する依存関係を一つの木構造になるように指定するのは困難である．そこで，ユーザの入力は

- 一つのファイルがほかのどのファイルに依存しているか
- ファイルを作成するために実行するコマンド列（インストラクション）

を入力し，この情報を元にシステム側で一つの依存木を構築する.

3.2 版・構成管理システムの属性文法へのマッピング

SSDSを属性文法を用いて記述するにあたり，SSDSの各機能を属性文法にマッピングする必要がある．ここでは，そのマッピングについて述べる.

3.2.1 版管理システムの属性文法へのマッピング

版管理システムは，バージョン木上で計算をおこなうことによりその機能を提供する．版管理システムの各機能の属性文法へのマッピングとしては**表1**の属性文法1のようなものが考えられる．バージョン木を構文木に対応付けることにより，各バージョンに付けられる情報や親バージョンとの差分は，それぞれのノードが持つ属性となる．バージョンの回復は，差分を適当な方法で計算することにより得ることができる．バージョンを登録する場合にはバージョン木の構造を変更するため，通常の属性文法では対応することができない．これには構造変更を用いる.

表1の属性文法1では単に「属性」としているが，これらは二つの種類に分けられる．一つは，構文木上のノードの位置により機械的に決定できる値を持つ属性であり，もう一つはノードの持つ終端記号から値を決定する属性である．これら二つの区別を明確にする．機械的に決定できる場合には「属性」，終端記号の情報として保持する場合には「終端記号」と表す.

表1の属性文法1では，バージョン間の差分を属性としているが，これには問題がある．差分を属性とし

て保持すると，属性計算がおこなわれた段階ですべてのバージョンの内容が属性値として得られてしまう．各バージョンの差分は差分ファイルとしてファイルシステム上に保持し，属性としてはそのファイル名を保持する．

差分をファイル名として保持すると，属性計算だけでファイルの回復をおこなうことはできなくなる．これは，システムの外部環境に対して実行すべきコマンド列を渡すことにより実現する．

以上の点を考慮すると，表1の属性文法2のようなマッピングとしなければならない．

3.2.2 構成管理システムの属性文法へのマッピング

構成管理システムにおいては，ファイル間の依存関係に基づいて依存木が構築される（図5）．しかし，一つのヘッダファイルが複数のソースファイルで用いられたり，システム中のいくつかのモジュールが同じオブジェクトファイルを用いる場合もある．この場合，依存関係は正確には木にはならない（図5のようにDAGを構成する）．本システムでは，属性文法を用いてこの依存関係を扱うために，依存関係のグラフを木構造に変換した依存木を扱う（**図6**）．そのため，元の依存関係がグラフ構造となっていた場合には，一つの依存木の中に同じ名前を持つノードが複数存在することになる（図6の`defs.h`参照）．

構成管理システムにおける依存関係の指定部分は，「一つのシステムやモジュール（ターゲットと呼ぶ）が複数のモジュールやソースファイル（依存しているファイルと呼ぶ）から構築される」ことを示している．この関係は，作成するターゲットを左辺，ターゲットが依存しているファイルを右辺とみなすと，生成規則として見ることが可能である．この考えを基にすると，構成管理システムと属性文法のマッピングとしては**表2**の属性文法1のようなものが考えられる．

このマッピングでは，依存関係をそのままSSDSが用いる生成規則に対応付けているため，ファイルの処理はそのまま意味規則部分に記述することができ，システムを簡単に構築することができる．しかし，生成規則はSSDSの作成時に指定するため，作成されたSSDSでは一つのシステムの作成しかおこなえない．

この問題を解決するために，本システムでは，ユーザにより与えられた依存関係の情報を元に，システムが構造変更を用いて依存木を構築する方法を選択する．この方法では，ユーザが依存関係を入力する部分と実際に依存木を構築する部分を分けることが可能であり，ユーザは直接依存するものの間の関係を任意の順序で記述することができる．インストラクションは版管理システムのバージョン情報と同様に終端記号として保持するとしていたが，依存関係の記述をおこなう部分と依存木を分けたため，これは依存関係を記述した部分の終端記号の値から属性としてテーブルを作成し，それを依存木に属性として流すことにより実現する．

以上の点を考慮すると，表2の属性文法2のようなマッピングとしなければならない．

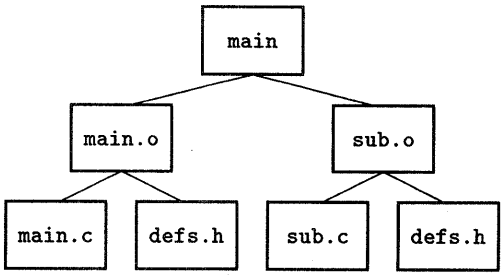


図6 変換した依存木の例
Fig. 6 An example of converted dependency tree.

表2 構成管理システムと属性文法のマッピング
Table 2 Mapping between configuration control systems and attribute grammars.

構成管理システム	属性文法1	属性文法2
ファイルの依存木	構文木	構文木
依存木の作成	構文解析	構造変更
ファイルの依存関係	生成規則	属性
編集するファイル	終端記号	終端記号
中間ファイル	非終端記号	終端記号
作成するファイル	開始記号	終端記号
インストラクション	終端記号	属性
ターゲットの作成	属性計算	属性計算+副作用

4. 版・構成管理システムの記述

本節では，版・構成管理システムSSDSの記述について述べる．前節で述べた版・構成管理システムの機能と属性文法とのマッピングに基づき，詳細設計をおこないながらSSLで記述する．

ここでは，版管理システムの記述，構成管理システムの記述，版管理システムと構成管理システムの連動機能，副作用の制御方法などについて述べる．

4.1 版管理システムの記述

本節では，版管理システムでの基本構造となるバージョン木の構造とその上でのバージョン名の計算方法，版管理システムの基本操作であるバージョンの回復について述べる．

4.1.1 バージョン木の構造

属性文法を用いてバージョン木を構成する場合，バージョンの表現方法やバージョン木の形などの違いにより，考えられる定義方法として次の3つの場合がある．

```
revision : RevisionNull()
         | Revision(log revision revision)
         ;
```

図7 バージョン木の定義
Fig. 7 Definition of version tree.

(1) 一つのフィラ revision でバージョン木を表現し，バージョンが存在しないことを示すオペレータを準備する．
(2) 一つのフィラ revision でバージョン木を表現し，派生したバージョンの数によりオペレータを使い分ける．
(3) フィラ revision がバージョンを示すのは2の方法と同じだが，リスト構造を用いRCSと同様に柔軟なバージョン木を構成する．

本システムでは1の方法を用いて実現をおこなった．これは，オペレータの数が少ないので意味規則の記述量が減るためである．この方法を用いたため，バージョン木は2分木となり，SCCSやRCSとは異なる構造となる（図4）．

フィラ revision （以降，SSLでバージョンを表すフィラをたんに revision と呼ぶ）の宣言（図7）において，一つの revision は二つの revision を持つ．前に書かれたものは，通常の開発の流れに沿ってできる兄弟の revision であり，後ろに書かれたものは枝別れのもので，子供の revision と呼ぶ．例えば，図4の“1.2.2”を考えると，兄弟の revision は“1.2.3”であり，子供の revision は“1.2.2.1”である．また，構文規則中の複数の revision を区別するために revision$n という表記を用いる．ここで n は構文規則中の何番目の生起かを示す．この記法を用いると兄弟の revision は revision$2 であり，子供の revision は revision$3 となる．

4.1.2 バージョン名の決定方法

バージョン名は，そのバージョンがバージョン木のどこに位置するかで一意に決定できる．それぞれの revision でこの計算をおこなうために，バージョン木を流れる属性は次の二つに分けられている．

rev_bo （相続属性） バージョン名の下1桁を除いた文字列
rev_no （相続属性） バージョン名の下1桁（INT型）

また，差分ファイル名などに用いるために，各 revision のバージョン名を示す次の属性も準備されている．

revision_no （局所属性） 各 revision のバージョン名

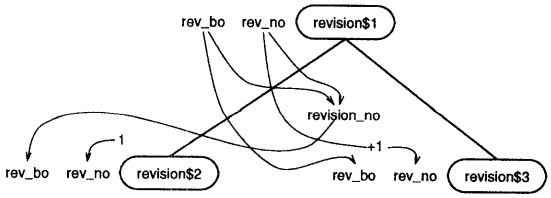


図8 バージョン名の計算
Fig. 8 Calculation of version names.

```
revision : Revision {
             revision_no =
                       $$.rev_bo #
                       INTtoSTR($$.rev_no);

             revision$2.rev_bo =
                       revision_no # ".";
             revision$2.rev_no = 1;

             revision$3.rev_bo = $$.rev_bo;
             revision$3.rev_no =
                       $$.rev_no + 1;
             }
         ;
```

図9 バージョン名の計算
Fig. 9 Calculation of version names.

この時，属性の流れは図8のようになる．バージョン1.2.2を例に取ると，rev_bo は“1.2.”で rev_no が2である．

この処理を記述したものが図9である．図9において，# という記号は文字列の連結を示し，関数INTtoSTRはINT型から文字列への変換をおこなう関数である．

4.1.3 バージョンの回復方法

バージョンの回復は，そのままの形で保存されているバージョン1.1を，回復したいバージョンまでのパス上の各バージョンの差分ファイルの内容で編集することによりおこなう．このために用意されている属性としては次の二つがある．

pre_files （相続属性） 親のバージョンを回復するための差分ファイル名の並び
delta_files （合成属性） 自分のバージョンを回復するための差分ファイル名の並び

この段階では，自分自身を回復するために必要な差分ファイル名の並びだけが計算されている（図10）．

実際の回復は，ユーザによる構造変更コマンドで起

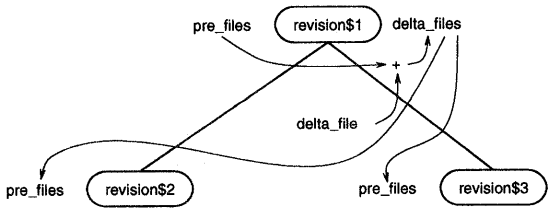


図10 差分ファイル名の収集
Fig. 10 Collection of delta file names.

動される．実行される構造変更コマンドは関数を呼び出すが，構造としては何も変更しないものである．ただ，引数として渡された差分ファイル名の並びを用いて実際にバージョンの回復作業をおこなう．

delta_files は，親により使用されることはないため局所属性とすべき値であるが，構造変更コマンドの中で関数への引数とするために合成属性として定義されている．

4.2 構成管理システムの記述

本節では，構成管理システムでの基本構造となる依存木の構造とその構築方法，構成管理システムの基本操作であるターゲットの構築方法，DAG を木構造に変換したために必要となった属性について述べる．

4.2.1 依存木の構造

依存木に対する抽象構文として，次の場合が考えられる．

(1) 依存木のノードをいくつかの子供を持つものであると捕え，いくつかのケースを想定し複数のオペレータを作成する．
(2) 依存木のノードを不定個の子供を持つものであると捕え，不定個の子供はリストとし，そのリスト自体を子供として持つ構造にする．

本システムでは，(2) の方法を用いた（**図11**）．これは，ターゲットがいくつのファイルに依存するのかは実行時にならなければ分からないためである．

4.2.2 依存木の構築方法

ユーザが入力する依存関係は，**図12** のようなものである．つまり，ファイル間の依存関係は，直接依存するものだけが指定されている．この情報と作成したいターゲット名から，図6 のような依存木を構築する．

この操作は，構文木の構造を変更するために構造変更コマンドとして実現されるが，依存木の構築自身は SSL の上では関数として実現されている．依存関係を記述した部分木とターゲット名を引数とし，依存木を構築する．この結果得られた依存木を構文木中に張り付け，この上で属性計算をおこなうことにより，ターゲットの作成をおこなう．

```
file_node       : FileNode(_file_name
                           file_node_list)
                ;

file_node_list  : FileNodeListNil()
                | FileNodeList(file_node
                           file_node_list)
                ;
```

図11 依存木の定義
Fig. 11 Definition of dependency trees.

```
main    : main.o sub.o
        cc -o main main.o sub.o

main.o  : main.c defs.h
        cc -c main.c

sub.o   : sub.c defs.h
        cc -c sub.c
```

図12 ユーザが入力する依存関係
Fig. 12 Dependencies supplied by user.

この方法を用いた場合，循環の存在する依存関係を木構造に変更することはできない．本システムでは，依存木の構築中にこれまで作成したノードのファイル名を保存しておき，すでに構築したノードと同じ名前のノードが依存するファイル名のリストに存在する場合には，循環が存在する旨のメッセージを出力し，そのノードの作成はおこなわない．

4.2.3 ターゲットの作成

依存木が構築された段階で，属性計算によりターゲットの作成をおこなう．これには，依存関係のあるすべてのファイルの最終更新時刻を自分自身の最終更新時刻と比較し，依存するファイルのほうが新しい場合には依存関係の記述に付けられたインストラクションを実行する．

このために準備されている属性には，

mod_time（**合成属性**） 部分木に存在するノードに対応するファイルの最終更新時刻のうち，最新のもの
instruction（**合成属性**） 実行しなければならないインストラクションの列
e_type（**相続属性**） ターゲットを実際に作成するかどうかを示す
mod_time0（**局所属性**） 注目しているノードに対応するファイルの処理を始める前の最終更新時刻

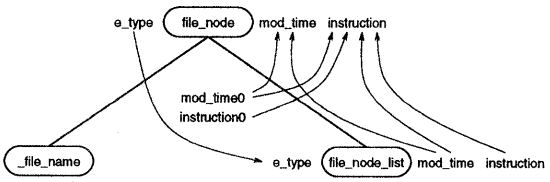


図13 更新時刻の計算
Fig. 13 Calculation of modification times.

instruction0（局所属性） 注目しているノードに対応するファイルの作成のために実行しなければならないインストラクション

がある（図13）.

4.2.4 共有ノードの表現

依存グラフを依存木に変更したことにより，本来なら共有されるべきノードが二つのノードとして出現する．この時，複数のノードで同じインストラクションの実行をしないようにする必要がある．

簡単のために，二つのノード（ノードAとノードBと呼ぶ）が存在し，インストラクションを実行しなければならない場合を考える．ターゲットの作成を実際におこなうのであれば，**mod_time0** の値がそれぞれのノードで異なるため，前節で述べた方法でうまくいくように思われるが，実際には両方のノードで **mod_time0** を計算するタイミングは決められないため，この方法ではうまくいかない．また，実際にターゲットの作成をおこなわない場合（つまり **e_type** が偽の場合），ファイルシステムから情報を得る **mod_time0** の値は両方のノードで等しくなってしまう．

そのため，新たに本システムが管理するファイルの最終更新時刻を格納するテーブルを準備し，これをインストラクションの計算と同時に流す．もし，ノードAでインストラクションの実行がおこなわれるのであれば，このテーブルの中の時刻は現在の時刻に修正する．ノードBでは，ファイルシステムから得られる実際のファイルの最終更新時刻だけでなく，このテーブルから得られる時刻も用いて **mod_time0** の値を計算する．

4.3 版管理システムと構成管理システムの連動

本システムにおいては，版管理システムと構成管理システムは，それぞれが独自に動作しているだけでなく，協調して動作している．本論文で説明したSSDSでは，構成管理システムがターゲットを作成する時に，版管理システムが管理しているソースファイルのバージョンを回復して用いることができる．

ターゲットを作成する際に版管理されたファイルを用いる場合には，データ入力部でどのようなバージョンの選択をおこなうのかを指定する．この時，

- どのような形式でバージョンの選択をするのか
- どんな値で選択をおこなうか

というデータが属性として版管理システムに渡される．

版管理システムでは，渡された属性に基づき，必要となった時に回復すべきバージョンを計算する．最適なバージョンを選択した後，そのバージョンを回復するために必要な方法を計算し，それをファイル名から回復するためのコマンド文字列を返すような属性として計算する．すべてのファイルに対してこの属性を集めたものは，属性として構成管理システムに渡される．

構成管理システムでは，版管理システムにより渡された属性を依存関係記述部から得られるインストラクションのテーブルとマージする．通常，この属性の中のソースファイルに対するエントリは空となっている．これはそのファイルを作成するために何の処理も必要無いためである．ソースファイルの処理をおこなう場合，このテーブルの中に空でないエントリが存在するときには，そこに格納されたインストラクションを実行する．

4.4 副作用の制御

構成管理システムにおけるインストラクションの実行は，システムに与えられた情報から作成される依存木の上の属性計算としておこなわれる．実際には依存木を作成することがインストラクションの実行準備であり，ここで作成された木の上での属性計算の段階でインストラクションの実行がおこなわれる．インストラクションの実行は関数的な属性計算ではなく，ファイルシステム上のファイルが変更されるという副作用を持つ．このような副作用を持つ計算は，不要な再計算がおこなわれないように特に注意しなければならない．

本システムではインクリメンタル属性評価器を仮定しているため，構文木の一部を変更した時でもインストラクションを示す属性が変更されない場合には実際の処理はおこなわれない．しかし，依存木が構築された後の属性の計算では，入力部，版管理システム，構成管理システムから得られる値を用いて計算をおこなっているため，これらの値が変更される場合には注意する必要がある．

インストラクションの処理が終わり属性の値が決定した後でも，ユーザがデータ入力部や版管理システム部の編集作業をするなどにより合成属性が変化した場合には，副作用（インストラクションの実行）を持つ属性の再計算がおこなわれてしまう（図14，二重丸は属性値が変化したことを示す）．

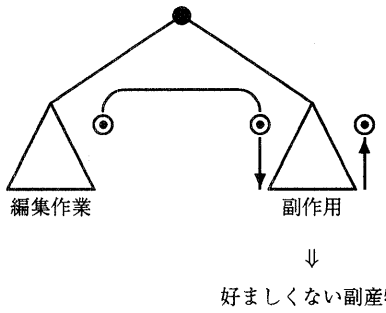


図14 好ましくない副作用
Fig. 14 Unpreferable side effects.

本システムでは，この点を解決するためにデータを保持するための構文木を導入している．依存木を構築する際に，

- 処理を実際におこなうかどうか
- どのような形式でバージョンの選択をおこなうか
- インストラクションのテーブル

などの計算に必要な情報を編集作業用の構文木や版管理システムから取り出し，新たな情報保持用の木として保持する．以降の計算は，この情報保持用の木の合成属性の値を用いる．つまり，データ入力部などの合成属性を直接参照するのではなく，普通は変更されることのない情報保持用の木から得られる情報を用いて計算をおこなうのである．

この方法を用いることにより，依存木が構築された後でユーザがデータ入力部を編集したり版管理システムを変更したりした場合にも，不要なインストラクションが実行されない．

4.5 システムの評価

本システムの作成においては，2章で定義した属性文法の拡張を用いて記述をおこなった．属性文法の基本モデルに対して拡張が必要だったのは，

- 構文木の構造をユーザからの指示により変更したい
- ファイルシステムにアクセスしたい

という二つの要望からである．版・構成管理システムに関して言えば，この拡張のみで十分記述することができた．

本システムを定量的にRCSおよびmakeと比較してみる☆．ただし，本システムはプロトタイプシステムとして作成したため，RCSやmakeに比べれば持つ機能は少ない．RCSはいくつかのサブプログラムから構成されているため，ここではバージョンの登録をおこなうciとバージョンの回復をおこなうcoだけを抽出し，さらに管理ファイルを読み込む部分，バージョン番号を生成する部分，時間を処理する部分，ciとcoのメイン部分だけを比較の対象とする．makeに関しては，メイン部分，ファイル処理部分，Makefile読み込み部分，ターゲットの再作成部分に関する部分を抽出し，比較の対象とする．

☆ ここで用いるRCSのソースはFree Software Foundationが配布しているRCS Version 5.6である．makeのソースもFree Software Foundationが配布しているGNU Make Version 3.68である．

表3 バージョン処理に必要な時間
Table 3 Time needed for manipulation of versions.

	RCS	SSDS
バージョンの登録	22.36秒	54.93秒
バージョンの回復	28.05秒	48.89秒

単純にソースファイルの行数で比較をおこなうと，SSDSはSSLで約900行とC言語で約170行，RCSで約3500行，makeで約4000行である．この記述の大きさの違いは，属性文法が版管理システムや構成管理システムを記述するのに適していることを示しているといえる．また，本システムを作成するために用いた非終端記号の数は14個，属性の数は48個であり，関数もSSLで20個，C言語で15個である．非終端記号を関数とみなして単純に合計すると49個となる．これに比較して，RCSでは118個，makeでは43個，合計で161個，3倍以上の関数を使用している．

速度面での比較を，**表3**に挙げる．これは，400行程度のファイルのバージョン27個を処理するのに要した時間である☆☆．SSDSでは，ユーザとのインタラクティブな操作に基づいて処理がおこなわれるため，測定には処理をおこなっている時間をシステムの反応を見ながら実際に測定している．この結果を見るとSSDSはRCSに比較してかなり劣っているが，SSDSにおけるファイル処理は属性文法では副作用となる外部関数として扱っているため，この部分を改良することで改善できる．

5. 版・構成管理システム記述のための属性文法に関する考察

ここでは，関連研究との比較，および，2章で拡張をおこなった属性文法に対する考察を述べる．

5.1 関連研究

高階属性文法[9),10)]では，構文木を拡張する方法として非終端属性が導入されている．この方法では，構

☆☆ 測定は，SONY社製のワークステーションNWS-3410（CPU: R3000，クロック：20 MHz）上でおこなった．

文規則の中に構文木を拡張する部分をあらかじめ指定しておく．また，実際の構文木の拡張は，非終端属性の値が計算された時点でおこなわれる．この非終端属性値の計算においては，それまでの文脈から得られる情報が用いられる．マルチパスコンパイラにおいて中間表現を木構造として表現しながらそれぞれのパスでの処理を属性文法で記述する場合や，定型文書のあるカラムに入力された値に基づいて後続部分の処理を変える場合などには向いている．しかし，今回作成した版・構成管理システムのように，ユーザのアクションに基づき既に構築された構文木を変更するようなシステムには用いることができない．

2章でおこなった拡張は，オブジェクト指向属性文法OOAG[19)~21)]でも実際におこなわれている．OOAGでは，属性文法をオブジェクト（OOAGでは属性付き木をオブジェクトと呼ぶ）の静的な性質の表現に用い，動的な側面は拡張部分であるメッセージ記述を用いておこなう．メッセージの起動は，規則に書かれた場合のみならず，ユーザがシステムの外側から起動することも可能である．また，次節で述べる「ユーザの指示により評価する文法を分離する機能」も，メッセージ記述の一部として実現されている．本システムをOOAGを用いて記述することにより，今回の実験結果を再確認することができると考える．

5.2 純粋な属性文法からの拡張部分に関する考察

本システムを構築するために，純粋な属性文法だけでは対応することが不可能だった点に関して考察する．

一つは構文木中のノードを変更する部分である．このためには，属性値から構文木を作成する必要があるが，その処理はユーザの指示により起動される．本システムにおいては，Synthesizer Generatorの構造変更コマンドという機構を用いてさまざまな機能の実現をおこなっている．属性文法に対して構文木自身を変形することができる機構を設けることで，本システムのような木構造を持ったオブジェクトを操作するシステムを構築する能力が得られることが判明した．

この構文木を変更する機能は，次のフェーズで用いる構文木の構造を属性値として計算し，それを次のアクションが必要になった時点（バージョンの登録など）で再び属性評価器に与えることで，通常の属性文法評価器でも実現することが可能である．この場合，版・構成管理システムのユーザ入力部が開始記号の相続属性に対応する．また，構文木中のノードを指定するための属性も必要になる．ユーザによる指示は，開始記号の相続属性を定義した上での属性評価の開始となる．

バージョンを回復する場合に，差分でなく差分ファイル名を保持した理由は，属性計算によりバージョンの内容自身が属性として存在することになるためであった．すべての属性を常に無矛盾な状態にするのではなく，通常は必要な属性のみを計算しておき，属性値が必要になった場合には計算していなかった属性値を計算するような機構を用いることで，この問題を解決することもできる．これは，常に計算しておく属性文法と，普段は計算をおこなわない属性文法を分けておき，ユーザの指示により後者の評価をおこなうことにより，実現することが可能である．

システムの外部に存在するファイルなどを処理する方法も問題点として挙げられる．この問題点は，属性文法の記述の中でファイルシステムをどのように扱うかにより解決方法が異なる．本システムでおこなったように属性計算の副作用として実現する場合には，属性の評価をおこなう際に余計な属性評価が実行されないように注意して設計する必要がある．本論文では，新たな部分木を導入する方法を用いて副作用を管理する方法を提案した．この方法を用いることにより，これまで属性文法には向かないと考えられてきたシステムも属性文法を用いて記述することが可能となった．

属性文法に対して

- 構文木をユーザの指示により変更する機構
- ユーザの指示により評価する文法を分離する機構
- システムの外部とのインタフェースを操作する機構
- 副作用の実行を制御する機構

などを準備することにより，版・構成管理システムのようなユーザとのインタラクティブな会話に基づき処理を進めるシステムを記述することが可能である．

6. おわりに

本論文では，属性文法の新しいアプリケーションとして版・構成管理システムを記述した実験について述べた．属性文法は構造的なデータの明解な記述に適していると考えられている．本論文で述べた版・構成管理システムは，版管理システムや構成管理システムを統合したシステムである．版・構成管理システムが扱うデータ構造は基本的に木構造であり，属性文法を用いてこのようなシステムを記述するのは容易であるといえる．

ただし，既存のシステムと連動させるには，注意が必要であった．既存のシステム（ファイルシステム）は手続き的であり，属性文法は関数的である点が問題となった．しかし，本論文で述べたような方法を用いることにより，属性文法と既存のシステムを接続して

使用することが可能になった.

これまで,属性文法は本システムのようなインタラクティブなシステムを記述することには向かないと考えられていたが,構文木の変更方法,システムと外部環境とのインタラクションの方法などを明確に定義することにより,これらのシステムも記述できることが確認された.

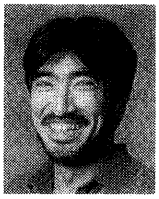
参考文献

1) Knuth, D.E.: Semantics of Context-free Languages, *Mathematical Systems Theory*, Vol.2, No.2, pp.127–145 (1968).
2) Knuth, D.E.: Semantics of Context-free Languages: Correction, *Mathematical Systems Theory*, Vol.5, No.1, pp.95–96 (1971).
3) Deransart, P., Jourdan, M. and Lorho, B.: *Attribute Grammars: Definitions, Systems, and Bibliography*, Lecture Notes in Computer Science, Vol.323, Springer-Verlag (1988).
4) 佐々政孝:属性文法,コンピュータソフトウェア, Vol.3, No.4, pp.377–395 (1986).
5) 片山卓也:属性文法 — 構造指向的かつ関数的計算モデル —, 情報処理, Vol.29, No.2, pp.146–154 (1988).
6) ICAD System, *Computer Aided Design Report*, Vol.5, No.12 (1985).
7) Katayama, T.: HFP, A Hierarchical and Functional Programming, *Proceedings of the 5th International Conference on Software Engineering*, pp.343–353 (1981).
8) GrammaTech, Inc.: *The Synthesizer Generator Reference Manual*, Ithaca, NY 14850, USA (1993). Release 4.1.
9) Vogt, H.H., Swierstra, S.D. and Kuiper, M.F.: Higher Order Attribute Grammars, *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp.131–145, ACM, Portland, Oregon (1989).
10) Teitelbaum, T. and Chapman, R.: Higher-Order Attribute Grammars and Editing Environments, *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp.197–208, ACM, White Plains, New York (1990).
11) Pittman, T. and Peters, J.: *The Art of Compiler Design — Theory and Practice*, Prentice Hall, Inc. (1992).
12) Reps, T.: Optimal-Time Incremental Semantic Analysis for Syntax-Directed Editors, *Conference Record of the 9th ACM Symposium on Principles of Programming Languages*, pp.169–176 (1982).
13) Kaplan, S.M. and Kaiser, G.E.: Incremental Attribute Evaluation in Distributed Language-Based Environments, *Proceedings of the 5th Symposium on Principles of Distributed Computing*, pp.121–130 (1986).
14) Reps, T.W. and Teitelbaum, T.: *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag (1989).
15) Tichy, W.F.: Design, Implementation, and Evaluation of a Revision Control System, *Proceedings of the 6th International Conference on Software Engineering*, pp.58–67 (1982).
16) Tichy, W.F.: RCS — A System for Version Control, *SOFTWARE — Practice and Experience*, Vol.15, No.7, pp.637–654 (1985).
17) Rochkind, M.J.: The Source Code Control System, *IEEE Trans. Softw. Eng.*, Vol.SE-1, No.4, pp.364–370 (1975).
18) Feldman, S.I.: Make — A Program for Maintaining Computer Programs, *SOFTWARE — Practice and Experience*, Vol.9, pp.255–265 (1979).
19) Shinoda, Y. and Katayama, T.: Object Oriented Extension of Attribute Grammars and Its Implementation Using Distributed Attribute Evaluation Algorithm, *Proceedings of the International Workshop on Attribute Grammars and their Applications*, Springer-Verlag, Lecture Notes in Computer Science 461, pp.177–191 (1990).
20) 権藤克彦,片山卓也:オブジェクト指向属性文法計算モデル OOAG, 日本ソフトウェア科学会第10回大会論文集, 日本ソフトウェア科学会, A3-1, pp.25–28 (1993).
21) Gondow, K., Imaizumi, T., Shinoda, Y. and Katayama, T.: Change Management and Consistency Maintenance in Software Development Environments Using Object Oriented Attribute Grammars, *Object Technologies for Advanced Software* (*Proceedings of the First JSSST International Symposium*) (Nishio, S. and Yonezawa, A.(eds.)), Lecture Notes in Computer Science 742, pp.77–94, Springer-Verlag (1993).

(平成6年12月8日受付)
(平成7年 9 月6日採録)

今泉 貴史（正会員）

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



篠田 陽一

1959年生．1983年東京工業大学工学部情報工学科卒業．1985年同大学院理工学研究科修士課程修了．1988年同博士後期課程単位取得退学．同年同工学部情報工学科助手．1991年北陸先端科学技術大学院大学助教授．現在に至る．工学博士．この間，情報環境，ソフトウェア工学，ソフトウェア開発環境などの研究に従事．

片山 卓也（正会員）

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