C++ 言語のシステム向き拡張ツール：OPTEC†

杉本 明†† 小島泰三†† 阿部 茂††

大規模計算機応用システムの開発では，多数の応用プログラマがシステム基本部を構成するライブラリ群を理解し，システムアーキテクチャとして規定された利用法に従ってプログラム開発を行わなければならない．しかしシステムの高機能化，分散化に伴い，システムアーキテクチャが複雑化し，ライブラリの量も増大している．その結果，応用プログラム開発の生産性低下が問題になっている．本論文では C++ 言語のシステム向き拡張ツール OPTEC について，その概要と特徴を述べる．OPTEC の目的は，システム基本部として構成されたライブラリ群に対して，その利用が容易となるようプログラミング言語を拡張，カスタマイズすることである．本論文ではまず，C++ 言語のクラスライブラリの再利用性を高める手法として，ライブラリに対する C++ 言語の拡張，カスタマイズが有効であることを示す．OPTEC では拡張言語から C++ 言語へのプログラム変換に木書換え手法を用いている．その特徴は，意味解析と木書換えの統合化により柔軟な拡張を可能としたこと，階層的書換えと呼ぶ手法により効率良く変換が行えること，変換出力に対する柔軟なプログラミング機能などである．また本論文では試作した OPTEC の適用結果についても述べ，有効性を明らかにする．

1. はじめに

工業プラントの制御や電力系統の監視などを行う大規模分散システムの開発では，あらかじめ多数のサーバプロセスやライブラリルーチンを，システム基本部として用意する．そしてこの基本部を用いて，50人から100人といった規模のプログラマにより応用システムが開発される．全体の要求機能を協調して実現するためには，個々のプログラマはシステム基本部のアーキテクチャを理解し，用意されたサーバプロセスやライブラリルーチンを適切に用いなければならない．

ところが，近年，システム機能の高度化に伴い，システム基本部のアーキテクチャが複雑化し，応用ソフトウェア開発の生産性低下が問題になっている．例えば，プラント制御と事務処理の統合化，操作性の良いユーザインタフェースといった要求に応じるため，ライブラリルーチンや共通に利用すべきデータ構造の種類が増加している．さらにシステムの拡張性や高信頼性などを実現するためには，特定の手順に従ったライブラリルーチンの使用をプログラマに強いることも必要となる．このためプログラマの教育に時間がかかり，またプログラム誤りも発生しやすくなっている．

このような問題に対処するためには，そのシステム基本部の実現の詳細を隠ぺいした，より抽象度の高いデータモデルや処理モデルを，応用プログラマに提供することが必要である．そのためにはプログラミング言語自体のシステム向き拡張が重要となる．確かに近年提案された汎用プログラミング言語は，従来のものと比較して優れた抽象化機能を持っている．しかしある特定のシステムや問題を自然に記述できる枠組みを提供するためには，言語自体の拡張が必要となる．例えば，CLU[1] は手続き，データ，制御の3つの抽象化機能を持つ言語であるが，Argus 分散システムアーキテクチャにおけるプログラミング言語として，CLU を拡張した Argus 言語[2] が開発された．

本論文では，オブジェクト指向言語 C++[3] を対象に，これを個々のシステムアーキテクチャにおけるプログラミングに適したシステム向き言語として拡張，カスタマイズする手法について論じる．C++ をベース言語として選択した理由は，応用問題領域の自然なプログラミングを可能とするシステム向き言語を構築する上で，オブジェクト指向言語がベース言語として適していると考えられるからである．また C++ は，従来からシステム開発に用いられてきたC言語を拡張したもので，プログラマにとっても親和性が高い．

C++ についても，ある問題領域の記述に適したように，あるいは新しいパラダイムを支援するように言語を拡張することは，従来から多く行われてきた．例えば，並行処理記述用の Concurrent C++[4] や，リアルタイムシステム記述用の RTC++[5]，オブジェクト指向データベースのデータ操作言語の O++[6] などが提案されている．しかし，これらの言語拡張は，C++ 言語の処理系を直接改造することで行われて

† OPTEC: A Language Translator Generator for Extending C++ to a System-Specific Language by AKIRA SUGIMOTO, TAIZO KOJIMA and SHIGERU ABE (Central Research Laboratory, Mitsubishi Electric Corp.).

†† 三菱電機(株)中央研究所

きた.

システム向き言語の場合には，その言語処理系が容易に作成できるツールが必要となる．システム向き言語は，個々のシステムアーキテクチャや応用領域に対応したものを設計する必要がある．また言語仕様が最初から固まるわけではなく，言語仕様の実験的な拡張が可能でなければならない．さらにハードウェアやOS，ライブラリの更新など，システム基本部の変更にも対応する必要がある．その度に C++ 言語の処理系を直接改造することは困難である．

OPTEC は，C++ 言語のシステム向き拡張を容易化することを目的に作成したツールである．システム向き言語によるプログラムを C++ 言語プログラムに変換する言語処理系を，言語拡張の定義から自動的に生成する．システム向き言語では，システムアーキテクチャを実現したライブラリ関数やクラスライブラリを実行時支援ルーチンとして用いる．C++ への変換においては，処理パタンやライブラリルーチンのきめ細かい選択を行わなければならない．OPTEC ではこの選択の柔軟な指定を可能とするため，プログラムの変換を書換えルールにより定義し，構文木のパタンマッチングによりルールを適用していく，木書換え手法と呼ばれる方法を用いている．

木書換え手法は，従来からコンパイラのコード生成系の生成技術として研究されてきた[7),8)]．また言語拡張の手段としても研究されている[9)~13)]．これら従来の研究と比較して，OPTEC の最大の特徴は，意味解析とコード生成を統合化した木書換え手法を用いていることである．C++ 言語は適用される変数の型により演算子や関数の意味を変える多重定義機能を備えており，限定された形ではあるが，言語の意味を拡張できる．システム向きに言語を拡張する場合には，この機能を一般化することが有効である．そのためには変数宣言や型宣言を拡張して新たな属性を導入し，属性に応じて適用するルールを選択する必要がある．しかし従来の研究は3章で述べるように，属性設定を行う意味解析機能が不十分であった．OPTEC では属性文法[14)]における合成属性と相続属性を共に書換えルール内で設定でき，柔軟な書換えを可能としている．

また本論文では，意味解析を統合化した木書換えを効率良く行う階層的書換え手法についても述べる．合成属性は木を上向きに，相続属性は木を下向きに伝播する．意味解析を木書換え手法に統合化するためには，このような伝播の方向の違う属性を，書換え時に効率良く設定できなければならない．OPTEC では，ルール記述パタンを C++ 言語のシステム向き拡張に必要な範囲に制限し，階層的に木を巡回して C++ コードの出力を行うことにより，効率の良い書換えを行っている．さらに OPTEC では，相続属性を用いたプログラミングにより，従来の木書換え手法では困難であった複雑な変換が可能となっている．

以下，2章では C++ 言語のシステム向き拡張について具体例によりその性質を示す．3章では拡張された言語による記述を C++ プログラムへ変換する言語処理系の生成手法について，従来の研究について述べ，OPTEC の特徴的手法について詳細を明らかにする．そして4章では試作した OPTEC の全体構成と適用結果について述べる．

2. C++ 言語のシステム向き拡張

システム向き言語は問題向き言語の一種であり，そこでは応用領域の問題記述に適した記述法の導入が行われる．本論文で特にシステム向き言語と呼ぶ理由は，システム向き言語では，システムアーキテクチャ実現に用いたライブラリ群の適切な再利用が促進されるように，言語が拡張されるからである．すなわちシステム向き言語は，利用するライブラリ群に対してカスタマイズされたプログラミング言語である．

実際のシステム向き拡張では，目的別に用意された様々なライブラリ群を統一的に扱う必要があるが，紙数の関係から，本論文では1つのライブラリだけを例として用いる．以下では図1のクラス継承関係を持つ，ユーザインタフェース構築用クラスライブラリを例として，C++ 言語のシステム向き拡張について述べる．

2.1 クラス利用の容易化と変数宣言の拡張

ライブラリ上のクラスの再利用を容易化するためには，クラスの具体的な利用情報を知る必要性を，できるだけ小さくすることが重要である．そのためには，応用プログラマに対して提供するデータモデルや処理モデルの抽象化をはかり，そのようなモデルに基づく

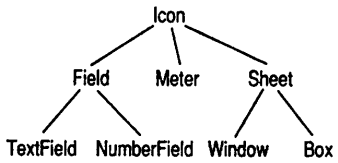

図1 例として用いるクラスの階層図
Fig. 1 Class hierarchy of example classes.

```
------------------------- case a ---------------------------------------------
visual int speed as Meter;     ==> Meter speed("speed");
int x = speed;                 ==> int x = speed.data;
speed = x + 4;                 ==> speed.put(x + 4);
------------------------- case b ---------------------------------------------
visual char* name as Field;    ==> TextField name("name");
visual int number as Field;    ==> NumberField number("number");
------------------------- case c ---------------------------------------------
with (new Window) {            ==> { Window* temp = new Window;
  show ("Car Info." +                temp->show(vbox4(new Label("Car Info."),
        ("Name:  " - name) +                        hbox(new Label("Name:  "), &name),
        ("Number:" - number) +                      hbox(new Label("Number:"), &number),
        speed);                                     &speed));
}                                  }
------------------------- case d ---------------------------------------------
sheetdef Panel {               ==> class Panel : public Sheet {
    width = 200;                   public: Icon* title;
    height = 300;                          Icon* body;
    visual title at (50, 20);              Panel(char*, Icon*, Icon*);
    visual body  at (20, 60);      };
};                                 Panel::Panel(char* name, Icon& a, Icon& b) : (name) {
                                       width = 200;
                                       height = 300;
                                       add(title = a, 50, 20);
                                       add(body  = b, 20, 60);
                                   }
-------------------------------------------------------------------------------
```

図 2 拡張言語による記述例と C++ への変換例
Fig. 2 Examples of extended descriptions and their conversions to C++.

記述を可能とすることが必要である.

例えば図1の Icon クラスのサブクラス Meter を整数変数の値の表示に利用するものとする. Meter クラスをプログラムで直接利用する場合には, 1)エンドユーザが別途アイコンの色やサイズを設定できるよう, オブジェクト生成時にはアイコン識別名を引数とする構築関数を呼び出す, 2)表示される整数値はメンバ変数 data が持つ, 3)表示を更新するにはメンバ関数 put を呼び出す, といった情報を知らなければならない.

これに対して, 以下のように変数宣言を拡張し, 視覚的変数と呼ぶ変数を導入する.

visual 型名 変数名 as アイコンタイプ;

この宣言により視覚的変数として指定すると, 型名で指定した整数変数や文字列変数のようにプログラム内では扱え, しかもその値が, アイコンタイプにより指定したオブジェクトにより常に画面上に表示されるものとする. すると, 上記のような情報を応用プログラマが知らなくても, 自然な記述によりライブラリ上のクラスを利用できるようになる.

図2 a にその記述例と C++ への変換例を示す. 図の変換例では int 型でかつ Meter 型オブジェクトにより表示される視覚的変数 speed の宣言に対応して, 変換後では speed がクラス Meter のオブジェクトとして所定の構築関数により生成されている. そして変数の参照更新に対し, それぞれ適切な変換が行われている. 逆にシステム向き言語の処理系作成ツールとしては, 上記のような変数宣言の拡張と, 図2 a に示すような変換が定義できるものでなければならない.

2.2 適切なサブクラスの選択自動化

オブジェクト指向言語によるクラスライブラリでは, 継承により上位概念のクラスからより特殊化したサブクラスが定義されている. プログラマはその中から適切なサブクラスを選択する必要があるが, 大規模なクラスライブラリではその選択は容易ではない. この選択を支援することもシステム向き言語の役割である. 例えば図2 b のような支援を行うと, 応用プログラマはサブクラスについての詳細な知識を得る必要がなく, クラスライブラリの再利用性が高まる. 図2 b は, 各変数型に依存して用意したクラス Field のサブクラスから, 変換時にクラス選択を行っている例である.

2.3 文脈の切り替えと文脈に依存した変換

前述の例における視覚的変数は, 通常は整数や文字列型などの基本型として扱えるものとした. しかしアイコンとしての画面上の配置をプログラムにより記述したい場合もある. 図2 c 左の例では with 文を導入し, 対象とする表現がアイコン配置のためのクラス Sheet のサブクラスの型を持つとき, with 文の内部の文では視覚的変数をアイコンとして扱うものとしている. このように, 特定の構文に対しては文脈を切り替え, その文脈に依存した変換を行うことが必要である.

なお図では, アイコンAとBに対して A+B は, AとBを垂直に配置した Box オブジェクトを関数 vbox で生成することを意味するものとしている. Box クラスは InterView[15] で提案された手法に従って水

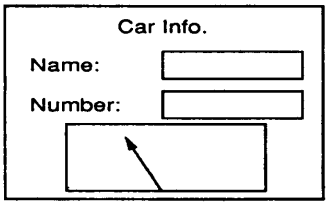

図 3 アイコン配置の例
Fig. 3 An example of Icon layout.

平垂直のアイコン配置を行う．また図では文字列定数に対しては自動的にラベルオブジェクトが生成され，結果として図3に示すアイコン配置となる．

2.4 ライブラリの利用知識を用いた最適化

C++ の演算子の多重定義は演算子の局所的パタンにより関数を決定する．例えば2項演算子なら2つのオペランドの型のみから実際に用いる関数を選択する．しかしその結果，実行効率が低下する場合がある．例えばアイコンA，B，Cに対して，A+B+Cは演算子の優先順序から (A+B)+C と等価であり，局所的な関数選択では vbox (vbox (A, B), C) と解釈される．ところが元々 Box 型が任意の個数のアイコンを垂直に配置でき，配置するアイコンを同時に指定するほうが配置の計算コストが小さくなるものとする．すると，図2 c右のようにパタンに応じて適切な関数を利用することで，実行効率が改善できる．このような最適化もシステム向き拡張言語の処理系の役割である．

2.5 クラス定義方法のカスタマイズ

C++ のようなオブジェクト指向言語では，継承に使われることを目的とした，抽象的なクラスをライブラリに用意することが多い．しかし一般にサブクラスを定義する場合には，継承するクラスの実現を良く知っている必要がある．これに対処するためには，継承を前提としたクラスに対して，そのサブクラス定義に適した表記法を導入することが有効となる．例えば前述の Sheet クラスに対して，そのサブクラス定義を容易化した表記例と変換例を図2 dに示す．

2.6 制御構文の多重定義

C++ では関数や演算子の多重定義しか許していない．システム向き拡張言語では制御構文に含まれる式の型により，処理パタンを変えられることも必要となる．例えば 2.3 節で導入した with 文は，対象とする表現式の型により異なった C++ 言語への変換を必要とする．またループ処理を行う構文，例えば forall 文を導入し，型により個々の要素の取出し法を変更すれば，応用プログラマに対して，クラスの内部構造をより効果的に隠ぺいすることが可能となる．

3. システム向き拡張言語の処理系生成手法

本章では前章で述べた C++ のシステム向き拡張言語の処理系生成手法として，パタンマッチによる言語変換手法について論じる．パタンマッチによる言語変換では，拡張言語により記述されたプログラムからパタンマッチにより拡張された構文，表現を抽出し，この部分をベース言語による表現に置き換える．新たに拡張する部分のパタンとその変換だけを指定すればよいので，拡張部分の差分的な定義が可能である．

図4に本章で考察する言語処理系の構成を示す．字句解析部ではソースをスキャンして構文解析部へトークンを渡す．構文解析部ではトークン列から構文木を構成する．パタン変換部では意味解析により型宣言や変数宣言の処理や型チェックを行い，C++ 言語のコードを生成する．もちろんシステム向き言語の処理系では，字句解析部と構文解析部についても C++ からの拡張が必要となる．2章で述べたように，システム向き言語では新しい予約語や構文を導入する必要があるからである．しかし，字句解析部と構文解析部についてはその生成手法の研究は進んでおり，ツールとしても LEX[16] や YACC[17] が広く利用できる．これらの研究を利用して，差分的に C++ からの拡張部分を定義できるツールを構築することは，比較的容易である．

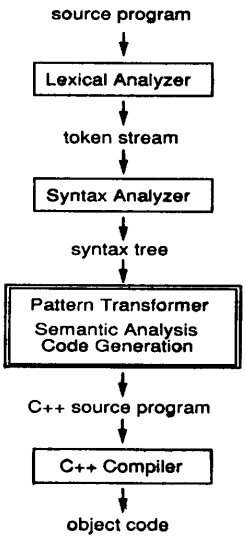

図 4 言語処理系の構成図
Fig. 4 Flow of program transformation.

したがって本章ではパタン変換部に焦点をあてる．以下，パタンマッチによる木書換え手法について従来の研究について述べ，その問題点を示す．次に OPTEC で用いた意味解析と書換えの統合化手法について述べる．

3.1 パタンマッチによる木書換え手法

パタンマッチによる木書換え手法は，言語拡張の手段として従来から研究されている[9)~13)]．従来の研究で，特に C++ 拡張の差分的な定義手法として有効と考えられるのは，Cheatham らにより提案された，意味解析により得られる型をマッチング条件とした木書換え手法である[10)]．Cheatham らは抽象度の高いレベルで記述されたプログラムの具体化，詳細化過程を支援する方法として，この手法を用いている[11)]．限定的ではあるが，同様な手法は拡張可能言語 EC[12)] においても使われている．

Cheatham らの手法において，木書換えによるプログラム変換を定義するルールは，テンプレート，意味条件，書換えパタンからなる．テンプレートは構文木であり，その葉には任意の文や表現式とマッチするパラメータを含む．書換え対象であるプログラム中の部分木とテンプレートがマッチすると，パラメータとマッチした接点が意味条件で指定した型を持つかどうかがチェックされる．これが満たされると，マッチした部分木を書換えパタンにより書き換える．

構文パタンに加えて型などの属性をルールの適用条件とすれば，柔軟に変換を定義できる．Cheatham らは2章で述べた最適化や制御構文の多重定義などについては，ルールがモジュール性高く定義できることを示した．しかし Cheatham らの研究では，変数宣言や型宣言の拡張などについては考慮されていない．Cheatham らはベース言語に対する意味解析ルーチンをそのまま使用し，ルールを適用する度に，書き換えられた木に対して意味解析を再度行う方法を取っている．このため，変数宣言や型宣言の書換えでは，その宣言のスコープ範囲の意味解析を再度行う必要が生じ効率が悪い．しかも2章で例として用いた visual のような属性は，変換後のベース言語による表現からは抽出できない．したがってシステム向き拡張により新たに導入した変数属性に対応した式や文の変換が行えない．このような問題点に対処するためには，拡張可能でかつ効率よく実行できる意味解析手法が必要である．

一方，システム向き拡張言語の処理系として用いる場合，システム開発時に日常的に使われるものであることから，木の書換え処理自体の高速化も必要である．木書換え手法は，従来からコンパイラのコード生成系の生成技術としても研究されてきた[7)]．それらの研究ではより効率の良い書換えが行われている．

木書換えをコンパイラのコード生成系として用いる場合，意味解析処理を行った後生成した中間言語による表現を木で表す．そして，目的機械に依存したコード生成規則を木書換えルールとして与える．従来の言語拡張を目的とした研究とは違い，ルールは書換えパタンを持たない．テンプレートにマッチした部分木を単一節点に還元しながら，その副作用としてルールのアクション部を実行し，部分コードを生成する．実際に木を書き換える必要がないので，高速化が計れる．

さらに Aho ら[8)] は目的機械の命令セット仕様を属性文法で表し，ルールに意味条件を導入している．すなわちテンプレート中のパラメータに対して，レジスタの種類やビット幅などの属性をマッチング条件として指定できるようにした．しかも木書換えにより還元した節点に対してルールで属性を設定することで，属性文法では合成属性と呼ばれる，木を上向きに伝播する属性を書換え時に処理している．

OPTEC で用いた書換え手法は，基本的に Aho らの研究を言語拡張に発展させたものである．機械語への変換では，中間言語による表現を生成する時点で，すでに変数宣言や型宣言は処理されている．しかし言語の拡張では，それらの宣言も書換え時に扱う必要がある．また文脈に依存した変換も実現する必要がある．Cheatham らの研究では，文脈に依存した変換については，ルールの中にサブルールを記述して，書き換えられた部分木にのみ，その適用を制御する方式を取っている．しかしこの手法では文脈に依存した変換の間で共通なサブルールがあっても，個々に記述する必要がある．書換えルールの中で新たな属性を設定でき，その属性が木を下向きに伝播されるなら，文脈に従う変換も意味条件で指定できる．次の3.2節で述べるように，OPTEC では，属性文法において相続属性と呼ばれる下向きに伝播する属性も，書換えルールから設定できるようにし，柔軟な変換を可能とした．

また機械語の場合には，木の葉から上向きにルール適用によりコード出力を行えばよく，また上向きに伝播する合成属性しか扱っていないため，ルールの適用順序の決定には問題がなかった．しかし C++ 言語の記述を出力する場合は，木の根から下向きに出力を

行う必要があり，また伝播の方向が違う合成属性と相続属性を共に扱う必要もある．ルールの適用順序の決定が困難になっている．OPTEC ではこれを効率よく行うため，階層的書換え法と呼ぶ手法を導入している．3.3 節ではそのアルゴリズムを示す．

さらに 3.4 節では，ルールのアクション部での相続属性を用いたプログラミングにより，従来の手法では困難であった複雑な変換が可能となることを示す．

3.2 木書換えと意味解析の統合化

本節では OPTEC のルール記述例をもとに，木書換えと意味解析の統合化について述べる．

（a） 式の変換と合成属性の設定

OPTEC のルール記述は以下の構文で行う．

rule ルール名 テンプレート
where 意味条件；
reduce 合成属性の設定；
{アクション}

図 5 に記述例を示す．テンプレートは，式を表す場合は（ ）で囲み，その他は［ ］で囲む．図の先頭のルール r1 は 2 章で例として用いたアイコン配置の関数選択のルール記述例である．

r1 のテンプレートの中で $expr::a は，構文クラスが式（expression）の節点とマッチするパラメータである．r1 の意味条件で a->type<=$type (Icon*) は，パラメータ a とマッチした節点が，Icon かそのサブクラスへのポインタを型として持つことをチェックする．$type は引数となる文字列をあらかじめ解析して，その型を表すオブジェクトを返す特殊マクロである．この意味条件が満たされると，r1 は reduce の中でルールとマッチした節点の型を合成属性として設定し，C++ で記述されたアクション部を実行する．アクションにおいて emit はコードを出力する関数である．

（b） 記号表の操作

属性文法では，変数の宣言はそのスコープ範囲の木の葉に対する相続属性として伝播される．しかし，OPTEC では変換処理の実行効率を考慮し，プログラム中に現れる記号に対する属性については，書換えルール内から記号表を直接操作する手法を用いた．

図 5 の r2 は図 2 a の視覚的変数の宣言の変換を指示するルールである．アクション部の declareVariable (var) は var で示される識別子を記号表に登録する．その後，この時生成されたシンボルに対して type などの属性を設定している．OPTEC では記号表上のシンボルに，新たな属性を設定できる．visualType はそのような例であり，この属性が式の変換の条件として用いられる．例えば図 5 の r3 と r4 は意味条件として visualType をチェックしており，図 2 a の下 2 行に対応する変換を指示する．r4 がマッチした代入の左辺の節点は r3 ともマッチするが，より広い範囲のパタンとマッチする r4 が優先される．

```
rule r1 ($expr::a + $expr::b + $expr::c + $expr::d)
where a->type <= $type(Icon*),
      b->type <= $type(Icon*),
      c->type <= $type(Icon*),
      d->type <= $type(Icon*);
reduce type = $type(Box*);
{   emit("vbox4(%n, %n, %n, %n)", a, b, c, d);   }

rule r2 [visual int $id::var as $typename::icon_class;]
{
    Symbol* item = declareVariable(var);
    if(icon_class == $type(Field))
       icon_class = $type(NumberField);
    item->type = icon_class;
    item->visualType = $type(int);
    emit("%n %n(\"%n\");", icon_class, var, var);
}

rule r3 ($id::var)
where var->visualType == $type(int);
reduce type = $type(int);
{   emit("%n.data", var);   }

rule r4 ($id::var = $expr::exp)
where var->visualType == $type(int);
reduce type = $type(int);
{   emit("%n.put(%n)", var, exp);   }
```

図 5 OPTEC の書換えルール記述例
Fig. 5 Examples of OPTEC rewrite rules.

```
rule r5 [with ($expr::target) $stmt::stmt]
where target->type <= $type(Sheet*);
{
    $$table  := target->type->GetTable();
    $$mode   := WITH_SHEET_INSTANCE;
    $$object := gensym("temp")
    emit("{%n %n = %n; %n}",target->type,
                            $$object,
                            target, stmt);
}

rule r6 ($id::var)
where $$mode == WITH_SHEET_INSTANCE,
      var->visualType;
reduce type = $type(Icon*);
{   emit("&%n", var);   }

rule r7 ($id::var)
where $$mode == WITH_SHEET_INSTANCE,
      var->type == $type(char*);
reduce type = $type(Icon*);
{   emit("new Label(%n)", var);   }

rule r8 ($id::func ($arg_list::args))
where $$mode == WITH_SHEET_INSTANCE,
      func->category == MEMBER_FUNC;
reduce type = rusult_type(func, args);
{   emit("%n->%n(%n)", $$object, func, args);}

rule r9 ($expr::a + $expr::b + $expr::c)
where a->type <= $type(Icon*),
      b->type <= $type(Icon*),
      c->type <= $type(Icon*);
reduce type = $type(Box*);
{   emit("vbox3(%n, %n, %n)", a, b, c);}
```

図 6 書換えルール例（図 2 c）
Fig. 6 Rules for rewriting Fig. 2 case c.

（c） 相続属性による文脈依存の変換

記号表に登録する属性以外の相続属性は，OPTECでは環境変数と呼ぶもので表す．これを利用して文脈に依存したルール選択が実現できる．**図6**のｒ5とｒ6は図２ｃの変換を指示するルールの一部である．環境変数は $$ で始まる名前を持つ．ルールｒ5のアクションにおいては，環境変数 $$mode の値を設定している．この値は相続属性としてｒ5のマッチした節点の下側の節点で有効となる．ルールｒ6ではこの値を意味条件でチェックし，文脈に依存した変換を行う．なお，ルールｒ5のアクションでは，記号表も相続属性として設定している．環境変数 $$table に設定された記号表は，ｒ5のマッチする節点を根とする部分木の葉で，記号に対する属性の検索に用いられる．

3.3 階層的木書換え手法

本節では木書換えのアルゴリズムについて述べる．テンプレートだけを考慮すると，構文木の各節点には多数のルールがマッチする．ルールは意味条件を持つので，ルールの選択にあたっては各節点の属性をチェックする必要がある．ところが木のある節点の属性は，その下側と上側の節点にマッチするルール選択に依存する．下側の節点で選択されたルールにより合成属性が，上側の節点で選択されたルールにより相続属性が設定されていなければ，その節点のルールの意味条件をチェックできない．したがって単純に，構文木の根から下向きに，あるいは，葉から上向きにルールを順次選択していくことはできない．またルールは，より広い範囲のパターンにマッチするものを優先する必要がある．

OPTECではルール選択を効率良く行うため，書換え対象となる構文木を，文に対応する節点を区切りとして分割している．図２ｃ左の記述に対する構文木を**図7**に示す．図7で点線で範囲を示した部分木が，文に対応する節点を区切りとした分割結果である．

OPTECの書換えは，このような階層的分割を次のように下向きに巡回して処理する．

1) 対象とする構文木の根から深さ優先で木を下降する．
2) 木の葉か文に対応する節点に達したら，上向きに木を登りながらルール選択を行い，合成属性を設定していく．個々の節点への具体的操作は次のとおり．
   2-1) その節点にテンプレートがマッチし，意味条件を満たすルールの中で，テンプレートのカバーする範囲や意味条件の個数，型のマッチの程度により適用ルールを選択する．
   2-2) もしマッチするルールが存在しなければ，あらかじめ用意されたデフォルトルールを選択する．
   2-3) 節点に対し選択したルール名をラベル付けし，ルールの reduce 実行により，その節点の合成属性を設定する．
3) 根にラベル付けされたルールのアクションを実行する．アクション実行時に emit 関数は第1引数の文字列を出力していく．この時，%n に対しては，その関数の第2引数以後のパラメータが表す節点に対して以下の操作を行う．
   3-1) もしその節点にルール名がラベル付けされていれば，そのアクションを再帰的に実行する．
   3-2) そうでなければその節点は文である．その節点を根とする構文木に対して1)からの操作を再帰的に繰り返す．

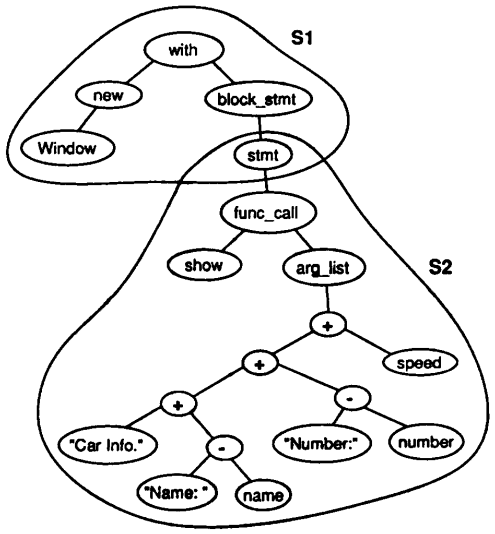

図7 構文木例（図２ｃ左）
Fig. 7 Syntax tree for Fig. 2 case c.

図7に対する変換を例として説明する．まず1)，2)の操作により図のＳ1で示す部分木が処理され，根に対して図６ｒ5のルールが選択される．このルールのアクション実行部では，環境変数の属性設定を行った後出力を行っている．この emit 関数による出力において，図7の block_stmt 節点にラベル付けされたデフォルトルールのアクションを実行し，その出力時点

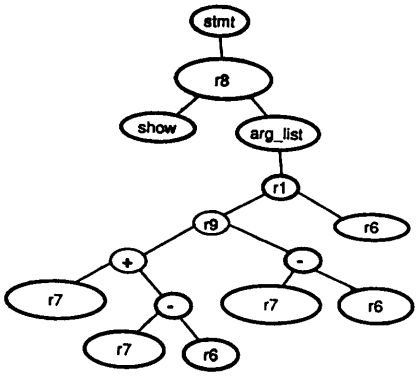

図 8 ルールがラベル付けられた部分木
Fig. 8 Subtree after labeling rules.

で図7の stmt 節点を根とする変換が開始される.

図8に図7部分木S2に対して上記1), 2)のステップが終了した時点のラベル付けを示す. 図中, 選択されたルール名を示していない節点に対しては, デフォルトルールが選択されている. stmt 節点にラベル付けされたデフォルトルールのアクションにより, 図6 r8 のアクションが実行され, 以下木を下向きにアクションが実行されていく. その結果, 最終的に図2 c 右の変換例が出力される. 図8で, アクションが実行された節点は太線で表している. 本手法ではより上位の節点にマッチしたルールが優先されるので, +接点にラベル付けされた図6 r9 のルールは, 結果としてアクションが実行されない.

以上のように, 本手法では分割された部分木を下向きに辿って処理する. そして個々の部分木に対しては, ルール選択とアクションの実行を分離し, 最初に上向き処理を行うことでルール選択と合成属性の設定を同時に行い, その後下向きに選択されたルールのアクションを実行して, コードを出力する. なお, 上記2)のルール選択については, Hoffmann ら[18]の手法を用いている. Hoffmann らの手法を用いると, ルール全体のテンプレートをあらかじめ前処理することにより, 木を上向きに一回辿るだけで, テンプレートのマッチするルール候補を各節点に設定できる. したがって上記の手法では, 全体として2回の木の巡回で, 属性設定やコード出力を含めた変換処理が行える.

本手法では, ルール記述に以下の制限を付ける必要がある. a) 文に対応する節点の合成属性は意味条件としてチェックできない. b) 文を区切りとする階層的分割に対して, 複数の部分木にまたがるテンプレートは記述できない. c) 相続属性の設定は, それが設定された部分木の下側の部分木には有効であるが, 同一部分木内では, ルールの意味条件とすることはできない.

```
rule r10 [sheetdef $id::name {$stmt_list::def};]
{
    $$table   := declareClass(name, $type(Sheet));
    $$mode    := SHEET_SUBCLASS_DEF;
    $$arg_buf := new Buffer;
    $$body_buf := new Buffer;
    emit("class %n : public Sheet {public: %n ",
          name, def);
    emit("%n(char* name %b);};", name, $$arg_buf);
    emit("%n::%n(char* name %b) : (name) { %b }",
         name, name, $$arg_buf, $$body_buf);
}
rule r11 [$id::var = $expr::exp;]
where $$mode == SHEET_SUBCLASS_DEF,
      var->table == $type(Sheet)->GetTable();
{   $$body_buf->emit("%n = %n;", var, exp);   }
```

図 9 書換えルール例 (図2 d)
Fig. 9 Rules for rewriting Fig. 2 case d.

しかしながら, 上記のような制限は, OPTEC の対象としているシステム向き言語拡張では重要な問題とはならない. 例えば, 内部節点の型は合成属性として設定されるが, 文の型は C++ では void であり, これを変更する拡張の必要性は少ない. また b) の制限はルールの分割により対応できる. c) の制限は, 例えば単文内部で設定された相続属性に依存したルール選択を不可能とする. しかしアクションに対しては, 単文内部でも相続属性が有効であり, アクション内で相続属性をチェックすることで, 対応は可能である.

3.4 環境変数を用いたプログラミング

OPTEC のルール記述においてはアクションを C++ 言語で記述するため, 柔軟に変換をプログラミングできる. すでにルール r2 の記述では2章で述べた適切なサブクラスの選択をルール内で行っている例を示した. さらに相続属性を表す環境変数を用いることで, 木の上位と下位に適用されるルールのアクション間で情報の交換が行える.

図9に, 2章で述べたサブクラス定義のカスタマイズに対するルール例を示す. $$arg_buf および $$body_buf は自動的に生成する構築関数の引数および定義を出力するためのバッファである. これは図の r11 のルールに示すように, 下側の部分木でマッチするルールのアクション内で出力に利用され, 図の r10 のアクションで最終的な出力の組合せに用いられている.

4. OPTEC の試作と評価

言語拡張ツール OPTEC の試作は第3版まで行われている. 第1版はC言語の拡張を目的として LISP 言語でプロトタイプを試作した[19]. 第2版では第1版

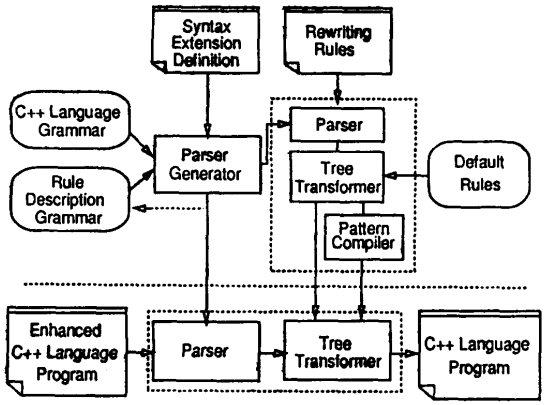

図 10 OPTEC のソフトウェア構成
Fig. 10 Organization of OPTEC system.

をC言語と構文解析用ツール YACC を用いて高速化した[20]，第3版から C++ 言語の拡張を目的として，C++ 言語と YACC により構築している．現在は C++ 言語から多重継承などを除いたサブセットのみを対象としている．

図 10 にソフトウェア構成を示す．言語拡張の仕様は2つのソースファイルで与えられる．1つは構文解析の拡張を記述した，構文拡張定義ファイルである．またもう1つはルールを記述したファイルである．システム向き言語の変換系は次のような手順で生成される．まずパーザジェネレータが構文拡張定義と，あらかじめ用意した C++ の構文解析用 YACC テンプレートから，YACC ソースファイルを生成する．これが YACC で処理されて変換系の構文解析部となる．またパーザジェネレータはルール記述処理用の構文解析部も生成する．構文解析されたルール記述は，用意されたデフォルトルールと共に処理されて，変換系のパタン変換部に相当する C++ プログラムが生成される．その際，ルールのテンプレートと意味条件は，3.3 節で述べたルール選択の高速化のため，パタンコンパイラで前処理が行われる．

現在，本システムにより電力系統制御システムのオペレータマンマシンシステム構築用言語を実装し，応用システムの開発に適用中である．そのルール数 37 個（うち 10 個はエラー処理用）で，言語拡張前と比較して約 2/3 に必要なソースプログラム量が減少した．これは主に定型的なプログラムパタンをルール化したことによる効果である．現在は小規模な適用にとどまっており，言語の学習時間やプログラミング時のエラー率まで含めた，総合的なソフトウェア生産性向上に対する効果の評価は今後の課題となる．しかし応用プログラマからは記述性が向上したとの評価を得ている．また OPTEC で生成した変換処理系は，例えば 1,000 ステップの拡張言語による記述を1秒以内で変換しており，十分実用に耐える速度を達成している．

拡張言語により記述したプログラムのデバッグは，既存のデバッグツールにより行っている．拡張言語のソースプログラム上で動きを追跡できるようにするため，変換処理系では，C++ 言語のプリプロセッサに対するファイル指定やライン指定を細かく指定している．またソースプログラム中の記号が，C++ に変換した記述でもそのまま残されているため，既存のデバッグツールによる対話的な変数内容のチェックも可能である．ただし，アクセスされる変数の型は変換後のものなので，変換後の実現を意識しなければならないという問題が残されている．

またルール記述についてもデバッグの問題が残されている．ルールの記述上の誤りやルール漏れについては，変換実験を行って検査するしかないのが現状である．ルール選択時点の構文木を表示するような，ルールデバッグツールの開発が必要となっている．

5. おわりに

本論文では C++ 言語のシステム向き拡張ツール OPTEC について，その概要と，拡張言語から C++ 言語への変換方式の特徴について述べた．まず目的とするシステム向き拡張言語について，その重要性について述べ，必要な拡張の性質を論じた．その中で，クラスライブラリに対してカスタマイズされた言語により，ライブラリの再利用性が向上することを示した．次に，OPTEC の木書換え手法の特徴である，意味解析の統合化，階層的書換え手法，環境変数によるプログラミングなどについて述べ，拡張部分が柔軟に定義でき，効率良くベース言語に変換できることを示した．また適用結果から有効性を明らかにした．

システム開発はプログラミングだけに留まらない．データ作成やマンマシン画面の対話型編集など，多くのツールを必要とする．プログラム内のデータ定義からこれらのツールに必要な情報を抽出したり，また逆にツールにより対話的に設定したデータから，プログラムの一部を生成することも重要である．このような言語を中心に統合化された，特定システム向きにカスタマイズできるソフトウェア開発環境について，今後考えていきたい．また OPTEC の言語変換手法は，

既存ソフトウェアの新しい計算機，OS，ライブラリへの移植ツールとしても有望と考えられる．今後この点についても検討を進めたい．

参考文献

1) Liskov, B. et al.: Abstraction Mechanisms in CLU, *Comm. ACM*, Vol. 20, No. 8, pp. 564-576 (1977).
2) Liskov, B.: The Argus Language and System, Alford, M. W. et al. (eds.), *Distributed Systems, Lecture Notes No. 190*, Springer-Verlag (1984).
3) Elis, M. A. and Stroustrup, B.: The Annotated C++ Reference Manual, Addison-Wesley (1990).
4) Gehani, N. H. and Roome, W. D.: *The Concurrent C Programming Language*, Silicon Press (1989).
5) Ishikawa, Y., Tokuda, H. and Mercer, C. W.: Real-Time Object-Oriented Language Design: Constructors for Timing Constraints, Tech. Report CMU-CS-90-111, CMU (1990).
6) Agrawal, R. and Gehani, N. H.: ODE (Object Database and Environment): The Language and the Data Model, Gupta, R. and Horowitz, E. (eds.), *Object-oriented Databeses with Applications to CASE, Networks, and VLSI CAD*, Prentice-Hall (1991).
7) Aho, A. V., Setchi, R. and Ullman, J. D.: *Compilers, Principles, Techniques, and Tools*, Addison-Wesley (1986).
8) Aho, A. V.: Efficient Tree Pattern Matching: an Aid to Code Generation, *Proc. of 12th ACM Symposium on Principles of Programming Languages*, pp. 334-340 (1984).
9) Layzell, P. J.: The History of Macro Processors in Programming Language Extensibility, *Comput. J.*, Vol. 28, No. 1, pp. 29-33 (1985).
10) Cheatham, T. E. et al.: Program Refinement by Transformation, *Proc. of 5th IEEE Int. Conf. on Software Engineering*, pp. 430-437 (1981).
11) Cheatham, T. E.: Reusability through Program Transformations, *IEEE Trans. Softw. Eng.*, Vol. SE-10, No. 5, pp. 589-594 (1984).
12) Katzenelson, J.: Introduction to Enhanced C (EC), *Softw. Pract. Exper.*, Vol. 13, pp. 551-576 (1983).
13) Cordy, J. R. and Promislow, E.: Specification and Automatic Prototype Implementation of Polymorphic Objects in TURING Using the TXL Dialect Processor, *Proc. of IEEE 1990 Int. Conf. on Comput. Languages*, pp. 280-285 (1990).
14) Reps, T. W.: *Generating Language-Based Environments*, p. 138, The MIT Press (1984).
15) Linton, M. A. et al.: Composing User Interfaces with InterViews, *Computer*, Vol. 22, No. 2, pp. 9-22 (1989).
16) Lesk, M. E.: Lex—A Lexical Analyzer Generator, UNIX Programmer's Manual 2, Section 20, AT&T (1979).
17) Johnson, S. C.: Yacc—Yet Another Compiler-Compiler, UNIX Programmer's Manual 2, Section 19, AT&T (1979).
18) Hoffmann, C. M. and O'Donnell, M. J.: Pattern Matching in Trees, *J. ACM*, Vol. 29, No. 1, pp. 68-95 (1982).
19) 小島，杉本ほか：C言語拡張システム：OPTEC，第39回情報処理学会全国大会論文集，pp. 1321-1322 (1981).
20) 小島，杉本ほか：C言語の問題向き拡張システム：OPTEC，情報処理学会記号処理研究会，58-1 (1991).

(平成3年7月1日受付)
(平成4年2月14日採録)

杉本　明（正会員）

昭和52年京都大学理学部卒業．昭和54年同大学院工学研究科（数理工学）修士課程修了．同年三菱電機(株)入社．以来同社中央研究所にて，設計支援システム，オブジェクト指向言語，分散システムなどの研究に従事．工学博士．電子情報通信学会，ACM各会員．

小島　泰三（正会員）

昭和57年東京大学工学部舶用機械卒業．同年4月三菱電機(株)入社．中央研究所にて，精密機械の計測と制御，設計支援システム，オブジェクト指向言語などの研究に従事．IEEE会員．

阿部　茂（正会員）

昭和46年東京大学工学部電子工学科卒業．昭和51年同大学院工学系研究科(電気工学)博士課程修了．工学博士．同年4月三菱電機(株)入社．中央研究所にて，電力系統システム，オブジェクト指向システム，幾何情報システムの研究に従事．昭和60年電気学会論文賞受賞．IEEE，電気学会，電子情報通信学会各会員．