Vol. 38 No. 12 情報処理学会論文誌 Dec. 1997

ビットマップ型言語におけるモジュール機能

山 本 格 也[†]

パターン置換えルールにより，ビットマップのパターン変化を記述するビットマップ型言語がいくつか提案されている．本稿ではビットマップ型言語において，複数のルールをまとめたサブプログラムを，あたかも1つのルールのように扱うことを可能にするモジュール機能を提案する．モジュール機能の実現に必要な呼び出しルールや，多種の引数パターンをモジュールへ渡すことを可能にするマッチング緩和法も提案する．さらに，ビットマップ型言語3D-Visulan上にモジュール機能を実装し，モジュールを使用したアプリケーション例を構築することで，提案したモジュール機能の有効性を示す．

A Module System for Bitmap-based Languages

Kakuya Yamamoto[†]

In bitmap-based languages, pattern replacement rules express the motion of bitmap patterns. This paper proposes a module system for bitmap-based languages, in which a subprogram consisting of a group of rules can be treated as a single rule. The proposed module system includes "module-calling rules" that are necessary to realize the module system and "matching relaxation methods" that allow calls with various kinds of argument-patterns. We show that the module system is effective by presenting an implementation of the module system on the 3D-Visulan bitmap-based language and an application using module calls.

1. はじめに

プログラムの表現メディアとして，従来からの文字に加えて絵や三次元物体を用いる，数々のビジュアルプログラミング言語が提案されてきた．本稿が対象とするビットマップ型言語はビジュアルプログラミング言語の一種であり，データとなるビットマップのパターンを，パターン置換えルールによって変更する．ビットマップ型言語の具体例としては，BITPICT[1]，Visulan[2]，3D-Visulan[3]などがあげられる．

ビットマップ型言語では，プログラムが複雑になるとルール数が増加するために，複数のルールをまとめるモジュール機能が必要となる．モジュール機能は，たとえばFORTRAN言語におけるサブルーチンや関数，C言語における関数などに対応する機能である．ビットマップ型言語は，データとの二次元マッチング（あるいは多次元マッチング）により，変更するデータ部分を特定する．この特色のために，既存言語のサブプログラム機能をそのまま適応できず，これまでモジュール機能は提案されていなかった．

本稿では，ビットマップ型言語において，複数のルールをまとめて1つのモジュールとし，そのモジュールをあたかも1つのルールのように扱えるモジュール機能を提案する．モジュール機能を実現するために必要となる呼び出しルールや，マッチング緩和法も提案する．また，三次元ビットマップ型言語3D-Visulanにモジュール機能を実装し，モジュールの再帰呼び出しの利用例として三目並べ（Tic-Tac-Toe）の思考プログラムを動作させることで，提案したモジュール機能の有効性を示す．

本稿の2章では，本稿が対象とするビットマップ型言語を定義する．3章ではビットマップ型言語におけるモジュール機能を提案する．4章ではモジュール機能の実装と，アプリケーション例を示す．5章では関連研究との比較や考察を行い，最後の6章でまとめる．

2. ビットマップ型言語

本稿が対象とするビットマップ型言語を以下で定義する．**ビットマップ**は，色を属性として持つ点（pixel）の二次元集合，または三次元集合である．**パターン**は，ビットマップ上に表現された絵や物体である．ビット

† 京都大学大学院工学研究科情報工学専攻
Department of Information Science, Kyoto University

マップ型言語が扱うデータは，すべてパターンである．

プログラムは，パターンの**置換えルール**の集合である．各ルールは，**使用前パターン**と**使用後パターン**から成る．プログラムを実行すると，各ルールは使用前パターンと同じパターンをビットマップから探す（**マッチング**を行う）．マッチングに成功したルールは，マッチングした部分のパターンを使用後パターンに置き換える．この置換えの動作は，代入の概念でとらえられる．代入する場所は使用前パターンにより決定され，代入する値は使用後パターンで与えられる．

ルール間には優先関係があり，優先順位の高いルールからマッチングが試されることで，複数ルールによる置換え競合を避ける．ルールによってパターンが変更されると，再び最も優先順位の高いルールからマッチングが試されて，マッチングに成功したルールがデータを変更する．マッチングに成功するルールがある限り，データの変更は繰り返される．すべてのルールがマッチングに失敗すると，実行は停止する．

3. モジュール機能の提案

3.1 モジュールの基本構造

モジュールは，**モジュール識別子**，**内部ルール**，**初期パターン**から定義される．**モジュール識別子**は，モジュールを特定するためのパターンであり，既存言語におけるサブプログラム名に対応する．**内部ルール**は，パターン置換えルールである．内部ルールは複数個あってもよい．初期パターンについては後述する．

モジュールは，**呼び出す**（call）ことで利用される．モジュールを呼び出すと，制御がモジュールへ移り，モジュールの実行が開始される．モジュールの実行が終了すると，モジュールを呼び出した側へ制御が戻される（return）．ビットマップ型言語では，マッチングに成功したルールがデータを置き換えることを繰り返す．モジュールが呼び出されると，その繰返しが一時停止して，内部ルールがデータを置き換えることを繰り返す．すべての内部ルールがマッチングに失敗すると，モジュールの実行は終了し，一時停止していた呼び出し側の実行が再開する．

各モジュールは独自のビットマップ（**内部ビットマップ**）を持っており，内部ルールはそのビットマップを操作する．内部ビットマップは，モジュールが呼び出されると生成される．初期値は，定義された**初期パターン**である．内部ビットマップは，生成された後に内部ルールにより操作され，実行が終了すると消滅する．

呼び出し側とモジュールとの間でのパターンの受渡しは，**引数**と**戻り値**により実現される．引数は，モジュールを呼び出す際に，呼び出し側のビットマップから内部ビットマップへ渡されるパターンである．戻り値は，モジュールの実行が終了する際に，内部ビットマップから呼び出し側へ返されるパターンである．

3.2 呼び出しルール

モジュールを呼び出す際に，どのように呼び出すのか，どのパターンを引数とするのか，モジュールからの戻り値をどうするのか，という問題を解決するために，モジュールの**呼び出しルール**を提案する．

呼び出しルールは，通常のルールと同様に使用前パターンと使用後パターンから定義される．マッチングも通常のルールと同様に，使用前パターンと同じパターンがビットマップに存在すれば成功し，存在しなければ失敗する．呼び出しルールは，呼び出すモジュールの識別子を使用後パターンとする．使用後パターンが，定義されているモジュールの識別子であるかどうかを調べることで，そのルールが呼び出しルールなのか，あるいは通常のルールなのかを区別できる．呼び出しルールがマッチングに成功すると，使用後パターンで示したモジュールを呼び出す．

引数は，使用前パターンがマッチングしたパターンとする．モジュールは，内部ビットマップの初期パターンの一部を引数で置き換える．置き換える部分は，たとえば左下奥のように，言語仕様で定める．その後，内部ビットマップのパターンは内部ルールによって変更され，変更後のパターンを戻り値として返す．返された戻り値は，引数として用いたパターンの新たなパターンとして用いられる．つまり，呼び出しルールにマッチングしたパターンは，モジュールから返されたパターンで置き換えられる．

以上の呼び出しルールの動作によって，モジュールの呼び出しとデータの受渡しという，必要最低限のモジュール機能が実現される．呼び出しルールは通常のルールと同様にビットマップの一部を置き換える．その置換えは，モジュールが持つ複数の内部ルールによる一連の置換えで実現される．よってプログラマは，呼び出しルールを用いることで，モジュールをあたかも1つの置換えルールのように扱える．呼び出しごとに内部ビットマップを用意することで，再帰呼び出しも可能となる．

3.3 マッチング緩和法

呼び出しルールによって，必要最低限のモジュール機能は実現できる．しかし呼び出しルールには引数に関連して，ある問題点が存在する．これを明らかにするために，引数について説明する．その後，解決策として，列挙パターンや Don'tCare 色によるマッチン

グ緩和法を提案する．

3.3.1 引 数

引数には，モジュール製作者が決めた制約が存在する．モジュール利用者は，その制約に沿った引数をモジュールへ渡さなければならないので，その制約を知っている必要がある．さらに，引数がどのようなパターンに変更されるのか，つまりどのような戻り値が返されるのかも知っている必要がある．しかし，引数がどのように変更されるのか，あるいは，戻り値へ至る過渡的なパターンを知る必要はない．

モジュールが複数種類の引数を受け取れるように定義されている場合でも，呼び出しルールは1種類の引数しかモジュールへ渡せない．たとえば，○と△の2種類の引数を受け取れるモジュールを定義しておいても，使用前パターンに○を用いた呼び出しルールは，○しか渡せず，△を用いた呼び出しルールは，△しか渡せない．引数は使用前パターンのマッチングによって決まるため，1つの呼び出しルールがモジュールへ渡せるパターンは，使用前パターンに示された1種類のパターンのみに限られる．

この制限により，複数種類のパターンをモジュールに渡したい場合は，その種類の数だけ呼び出しルールを用意する必要が生じる．ところが，実際にモジュールへ渡したいパターンの種類は組合せによって膨大な数になる場合が多く，その種類ごとに呼び出しルールを記述することは現実的ではない．たとえば，あるゲームの盤面を1手進めるモジュールがあったとする．そのモジュールを呼び出すために，すべての盤面パターンの数だけ呼び出しルールを記述することは実際上不可能である．

上記の問題点を解決する方法として，列挙パターンやDon'tCare色によるマッチング緩和法を以下で提案する．マッチング緩和法により，1つの呼び出しルールで多様なパターンをモジュールへ渡せるようになり，多種の引数を扱うモジュールの利用が現実的となる．

3.3.2 列挙パターン

1つの呼び出しルールで，異なる種類の引数をモジュールへ渡すには，使用前パターンが複数種類のパターンにマッチングすればよい．**列挙パターン**（enumeration）は，複数種類のパターンにマッチングするパターンである．列挙パターンの定義は，1つの列挙パターンと呼ばれるパターンと，複数（1つ以上）の**列挙子**（enumerator）と呼ばれるパターンから成る．列挙パターンを呼び出しルールの使用前パターン内で用いると，列挙パターンの部分がいずれかの列挙子となったパターンにマッチングする．

たとえば，○形と△形の2つの列挙子によって，◇形の列挙パターンが定義されているとする．呼び出しルールの使用前パターンに◇を用いれば，○がビットマップに存在すれば引数として○を，△がビットマップに存在すれば引数として△をモジュールへ渡せる．

使用前パターン内では，複数の列挙パターンが使用できる．同じ列挙パターンを複数回用いても，異なる列挙パターンを用いてもよい．また，列挙パターンの定義においては，列挙子として，他で定義された列挙パターンを用いることもできる．

3.3.3 Don'tCare 色

Don'tCare色として定義された色を，使用前パターン内で用いると，すべての色とマッチングする．どの色をDon'tCare色とするかは，ユーザが定義する．ユーザは呼び出しルールの使用前パターン内の一部分をDon'tCare色とすることで，その部分に対応するビットマップのパターンがどのようなパターンであっても，マッチングさせることができる．その結果，多様な引数パターンをモジュールへ渡せる．

たとえば，使用前パターンが○のパターンで，○の内部がDon'tCare色の場合，ビットマップに○のパターンがあれば，その内部のパターンにかかわらずマッチングに成功する．その結果，呼び出しルールは，その○を中身ごとモジュールへ渡せる．

3.4 組込みモジュール

通常のモジュールはユーザが定義するが，あらかじめ定義されているモジュール（**組込みモジュール**）も考えられる．たとえば，マウスのボタン入力や乱数入力などのシステム入出力を，組込みモジュールとして用意しておけば，ビットマップ型言語でシステム入出力を扱うことが可能となる．

たとえば，マウスのボタンの状態を返すモジュールがあれば，マウスのボタン入力を扱うプログラムを作成できる．マウスの形をしたパターンを引数として組込みモジュールへ渡すと，そのときマウスのボタンが押されていれば，ボタンが押されているマウスの形のパターンを返し，ボタンが押されていなければ，ボタンが押されていないマウスの形のパターンを返す．

モジュール利用者は，そのモジュールが組込みモジュールなのか，ユーザによって定義されたモジュールなのかの違いを意識せずに利用できる．

4. モジュール機能の実現

4.1 3D-Visulanへの実装

提案したモジュール機能を，三次元ビットマップ型言語3D-Visulanに実装した．

まず，3D-Visulanにおいてデータやルールがどのように表現されているかを示すために，簡単なアプリケーション例を**図1**に示す．図1の左側の台が**データステージ**と呼ばれる台であり，台上の空間がプログラムが操作するビットマップ（**データ部**）である．図1の右側には置換えルールが1つ存在する．置換えルールの左半分が使用前パターン，右半分が使用後パターンを表している．データステージやルールの大きさは自由であるが，形と色は言語仕様により定まっている．図1の例では，データ部に2人の太った人がいる．使用前パターンは太った人を表し，使用後パターンはやせた人を表しているので，このルールは太った人をやせさせることを意味する．実行すると，データ部の2人はやせる．

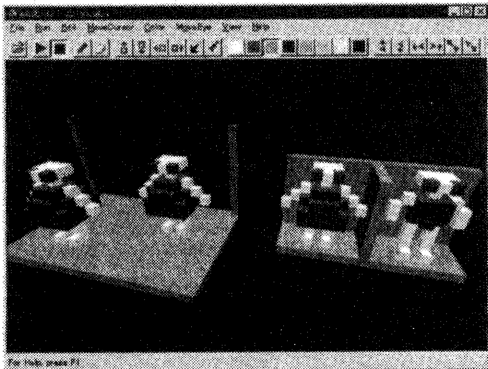


図1 3D-Visulanにおけるデータとルール
Fig. 1 Data and a rule in 3D-Visulan.

次にモジュール機能の使用例を示す．**図2**は，データ部に作られた「囲い」の扉を閉じるアプリケーション例である．扉を閉じる動作を1つのモジュールが担当している．図2の手前左側にはデータ部が，手前中央には呼び出しルールが存在する．データ部には高さの異なる直方体の箱が2つと，それらの囲いとがある（**図3**左下）．囲いの扉は空いている．呼び出しルールの使用前パターンは，データ部と同一のパターンである．使用後パターンは，扉の閉じた囲いのパターンであり，後述するモジュールの識別子と同じパターンである．図2の奥にはモジュールの定義があり，3つの台が連なった形をしている．1番左の台上のパターンは，これら一連の台がモジュールを定義していることを表すパターンであり，言語仕様で決まっている．2番目の台上のパターンは，モジュール識別子である．このモジュールの場合は，扉の閉じた囲いのパターン

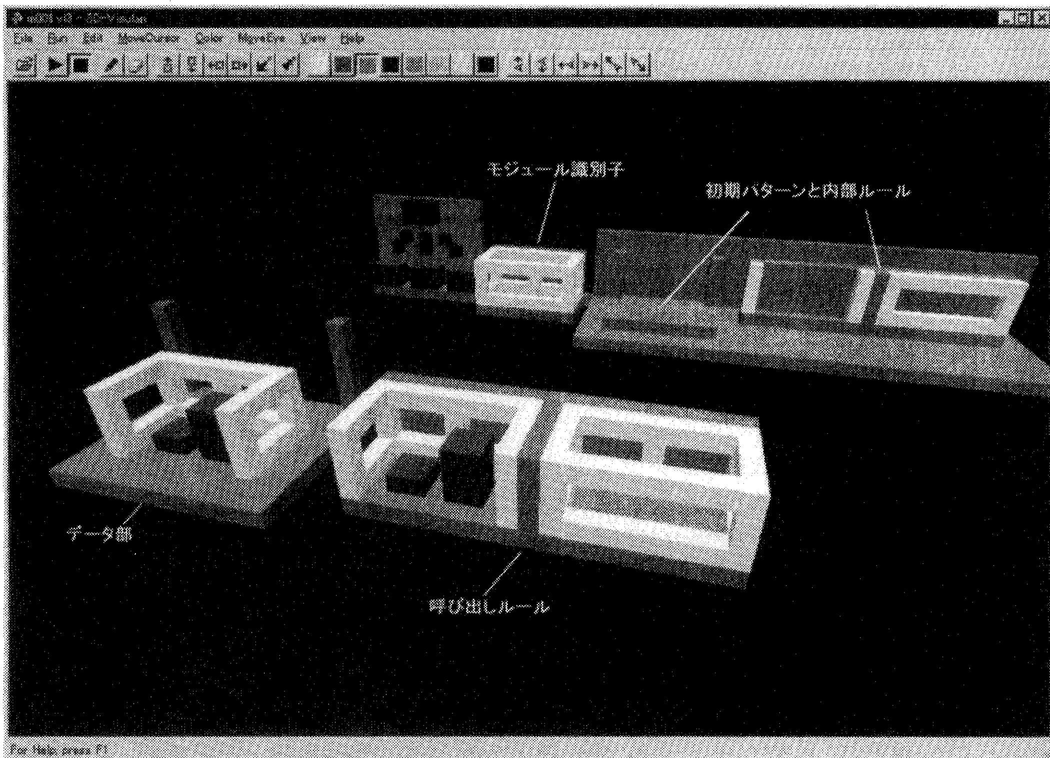


図2 モジュール使用例
Fig. 2 Example of a module call.

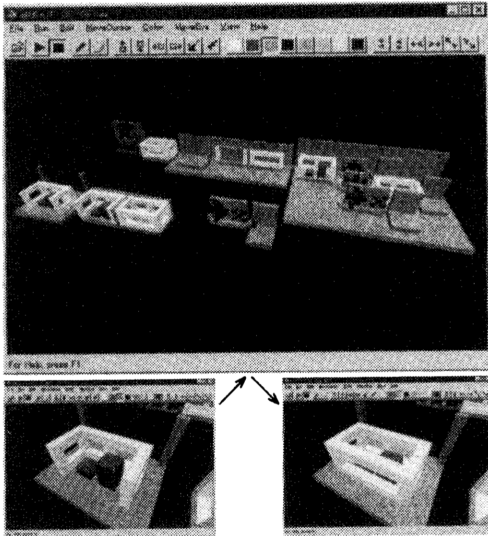


図3 モジュール使用例の実行経過
Fig. 3 Examples of a module call in execution.

である．3番目の台上には，1つのデータ部とルールがあり，それぞれ初期パターンと内部ルールの定義となる．このモジュールの場合，初期パターンには何の物体も存在せず，扉を閉じる内部ルールが1つ存在する．

図2を実行すると，まず，呼び出しルールの使用前パターンがデータ部とマッチングし，モジュールを呼び出す．モジュールが呼び出されると，モジュール定義の3番目の台の右隣に，4番目の台が動的に生成される．台上には内部ビットマップが生成され，定義された初期パターンがコピーされる．その後，コピーされた初期パターンの左下奥部分が引数で置き換えられる．左下奥部分の置換えは，内部ビットマップの初期パターンの左下奥の角と，引数の左下奥の角が重なるようになされる．この例では，初期パターンには何も存在せず，引数は，2つの箱とそれらの囲いのパターンである．この時点の途中経過図が図3上である．その後，モジュールの実行が開始され，内部ルールによって内部ビットマップにある囲いの扉は閉じられる．囲いの扉が閉じると，マッチングしなくなる．よってモジュールの実行は停止し，内部ビットマップは消滅する．扉が閉じた囲いのパターンは戻り値として呼び出し側へ返され，図2手前左側のデータ部の囲いの扉も閉じた形のパターンとなる（図3右下）．

なお3D-Visulanでは，すべての情報を1つのビットマップで表現している．図3上の経過図で，モジュール定義の4番目の台上には，内部ビットマップに加えて，その台の識別子，呼び出し元の識別子，その台の実行状態が三次元物体として表現されている．呼び出しルールの右隣にも，呼び出し元の識別子が生成されている．

図4 Don'tCare色によるモジュール呼び出し例
Fig. 4 Example of a call with the Don'tCare color.

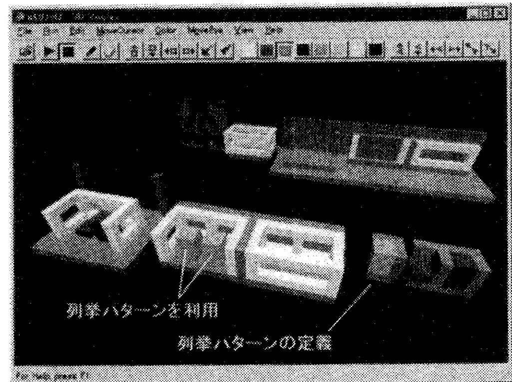
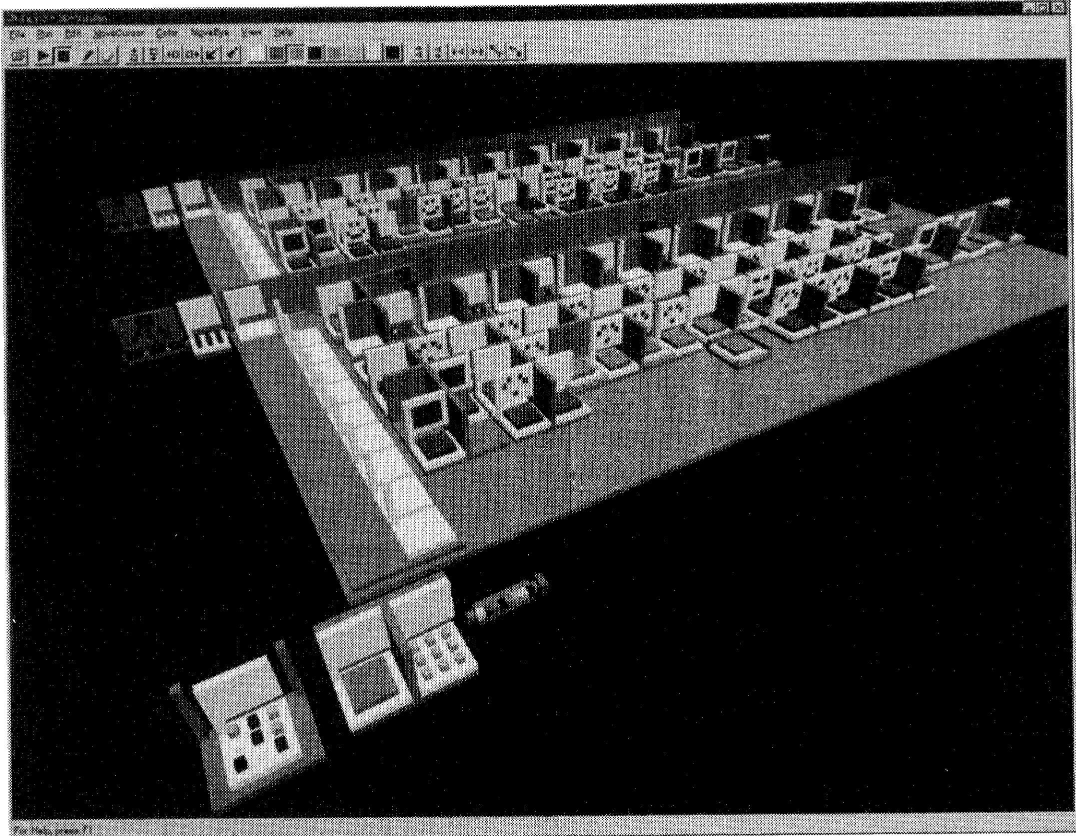


図5 列挙パターンによるモジュール呼び出し例
Fig. 5 Example of a call with enumerations.

次に，Don'tCare色を使用した例を**図4**に示す．Don'tCare色を使用前パターン内で用いることで，囲いの中に何が存在しても，それを引数としてモジュールを呼び出すルールを作成できる．図4で呼び出しルールの右隣にあるのは，Don'tCare色の定義である．

さらに，列挙パターンを使用した例を**図5**に示す．Don'tCare色を用いた例では，囲いの中に何が入っていても囲いの扉が閉じられてしまう．列挙パターンを使用することで，囲いの中に入っている箱の高さによって扉の開閉を制御できる．箱の種類を高さによって分け，高さが1の箱，2の箱，3の箱があるとする．箱形の列挙パターンを定義し，列挙子として高さが1の箱と3の箱を用いる．呼び出しルールの使用前パターンで，定義した列挙パターンを用いる．その結果，囲いの中に入っている箱の高さが1の場合か3の場合にのみモジュールが呼び出され，囲いの扉が閉じるよう



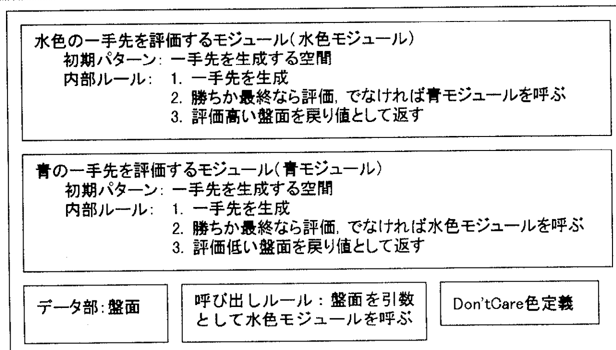


図6 三目並べの思考プログラムとその図解
Fig. 6 Tic-Tac-Toe thinking program and its diagram.

になる．図5では列挙パターンを2つ用いたルールとなっているので，2つの箱の高さがいずれも2でない場合に扉を閉じる．

4.2 アプリケーション例

モジュールを用いた3D-Visulanアプリケーション例として，再帰呼び出しを行う三目並べの思考プログラムを作成した（図**6**）．

三目並べは，縦3マス横3マスの計9マスの盤に，2人の競技者が交互に自分の記号（通常○か×）を，空いているマス目に書き入れる．縦か横か斜めに自分の記号が三目並ぶと勝ちとなるゲームである．

作成した思考プログラムは，三目並べの最善手を求めるプログラムである．○と×の代わりに青（図6で濃い色）と水色（図6で薄い色）で区別した．プログラムは，データ部，呼び出しルール，Don'tCare色の定義，水色の1手先を評価するモジュール（水色モジュール），青の1手先を評価するモジュール（青モジュール）から成る．データ部には，三目並べのゲーム途中の盤面が存在する．呼び出しルールの使用前パターンではDon'tCare色が利用され，使用後パター

ンは水色モジュールの識別子である．よって，どのような盤面でも，水色モジュールを呼び出すことができる．各モジュールは，渡された盤面の空いているマス目それぞれについて，自分の色を書き込んだ盤面を内部ビットマップに生成し，さらに相手の手を指すモジュールを呼び出す．2つのモジュールは，最終局面を読み切るまで，互いに呼び出し合う．その結果内部ビットマップが次々と生成され，最後まで読み切ると，順々に消滅していく．盤面の評価値は，勝てるか，負けるか，引き分けるかの顔の表情パターンで表される．最終的には，モジュールのすべての内部ビットマップは消滅し，データ部にある盤面は最善手を指した盤面に置き換えられ，実行は停止する．

5. 考　　察

既存言語の例としてC言語とProlog言語をあげて，モジュール機能を考察する．C言語で，

```
void M() {
  R1();
  R2();
}
```

という関数Mがあった場合，MはR1とR2をまとめたものと考えられる．またProlog言語で，

```
R1 :-（省略）.
R2 :-（省略）.
M :- R1, R2.
```

という節Mがあった場合，MはR1とR2をまとめたものと考えられる．ビットマップ型言語のルールは単なるパターン置換えルールなので，CやPrologと同様のまとめ方はできない．本稿で提案したモジュール機能は，ビットマップ型言語独自のルールのまとめ方である．

呼び出しルールがモジュールへ渡せる引数の個数は1つである．複数の情報をモジュールへ渡したい場合は，それらを1つのパターンとしてまとめ，そのパターンを渡すこととなる．既存の言語の場合，変数をいくつも用意することができ，そのうちのいくつかをモジュールへの引数とできる．しかしビットマップ型言語の場合は，データとなるビットマップはあくまでも1つであり，1つのビットマップに複数の情報が表現される．表現されたパターンをどのように切り分けるかは，マッチングにより個別に決定される．よって，引数の個数の概念は既存の言語とは異なる．

マッチング緩和法は，呼び出しルールにおいて多種の引数をモジュールへ渡す際に不可欠となる技術であり，具体的には列挙パターンと，Don'tCare色で実現できる．すべての色の点をそれぞれ列挙子として，ある色の点を列挙パターンと定義することでDon'tCare色は実現できる．よって，マッチング緩和法は列挙パターンのみで実現できるともとらえられる．本稿で提案したマッチング緩和法は，繰返しのない正規表現と見なせる．文字列で繰返しを用いた正規表現は，異なる長さの文字列を表現できる．しかし，繰返しの考え方自体が一次元の発想であり，ビットマップにはそのまま適応できない．そのため現状では，同じ大きさのビットマップどうししかマッチングしない．たとえば，あるパターンとあるパターンとの間に存在するパターンをモジュールに渡すような利用は，提案したマッチング緩和法では実現できず，今後の課題となる．具体的には，大きさの異なるパターン間のマッチング解決法，特にデータ部に複数パターンが存在するためにマッチングが一意に定まらない場合の解決法，さらには，モジュールの初期パターンより大きい引数が渡された場合の対処法などが課題として残されている．

モジュールが独自の内部ビットマップを持つことで，呼び出し側のビットマップに影響を与えることなく複雑なパターン変更を行える点や，呼び出し側に作業用の余分な空間がない場合でもモジュールを呼び出して作業を遂行できる点が，ビットマップ型言語独自の利点としてあげられる．たとえば三目並べのように，目の前にある盤面を変えることなく次の手を考えるには，内部ビットマップは必須である．

組込みモジュールは，ビットマップ型言語において，ビットマップ以外のデータやシステム入出力を扱う枠組みを提供する．周辺機器，他のシステム，既存のプログラミング言語などとビットマップの仲介役として，組込みモジュールは利用できる．

3D-Visulanはモジュール機能を実現したビットマップ型言語の一例であり，他の実現方法も考えられる．たとえばプログラムとなるルール群とデータ部とを，3D-Visulanでは同一のビットマップ上で表現しているが，別々に表現する実現方法も可能である．3D-Visulanではすべてのモジュールが同一のビットマップで表現されているが，モジュールごとに別々のビットマップを用意する方法もある．データ部に同じパターンが複数あり，それらをモジュールへ渡す場合に，3D-Visulanでは同時に渡して複数の内部ビットマップを並列に動作させるが，逐次的に渡すシステムも考えられる．三目並べの思考プログラムでは，ビットマップ型言語でもモジュール機能を実装することで，ある程度複雑なアプリケーションも構築できることを示せた．ビットマップ型言語のみならず，現存するビジュアルプログラ

ミング言語の範囲においても，テキスト表現を用いることなく，三目並べの思考プログラム程度の複雑なアプリケーションを構築できるシステムは，3D-Visulan が初めてである．

関連研究との関係を述べる．現存するシステムの中で，ビットマップ型言語に相当するものとしては BITPICT，Visulan，3D-Visulan があげられる．KidSim[4]は，扱うデータがビットマップではないので，本稿におけるビットマップ型言語の定義からは外れる．しかし，ビットマップの代わりに，ゲームボード（game board）と呼ばれるチェス盤や将棋盤のような四角形の二次元集合をデータ表現として用いており，それぞれのマス目には魚や岩といった絵（simulation object）が描かれる．そのために，本稿で用いた「点」という言葉を「simulation object」に置き換えれば，提案した呼び出しルールや，マッチング緩和法をそのまま適応し，モジュール機能を実現することが可能である．

置換えルールを用いるビジュアルプログラミング言語の中には，扱うデータが単純なビットマップではないシステムも提案されている．たとえば ChemTrains[5]，Mondrian[6]，Agentsheets[7]などがあげられる．これらのシステムにおいても呼び出しルールの実現は可能である．一方マッチング緩和法に対しては，ビットマップ型言語と，これらのシステムとでは哲学が異なるため，そのまま適用はできない．ビットマップ型言語では，列挙パターンや Don'tCare 色などのように，異なるパターン間のマッチングを目指す．一方，単純なビットマップ以外のデータ構造を持つ言語では，データ構造を複雑にして，柔軟性をデータ側に持たせることで，異なるデータ表現間のマッチングを目指す．よってこれらのシステムの場合，マッチング緩和に関しては，独自の方式が必要である．単純なビットマップを扱わない，置換えルールを用いるビジュアルプログラミング言語において，モジュール機能の提案はなされていない．

6. おわりに

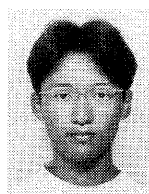
本稿では，ビットマップ型言語において，複数のルールをまとめて1つのモジュールとし，そのモジュールをあたかも1つのルールのように扱えるモジュール機能を提案した．呼び出しルールによって，モジュールの呼び出しやモジュールとのデータの受渡しが実現される．さらにマッチング緩和法により，多種の引数を一度に扱える呼び出しルールも可能となる．また，ビットマップ型言語 3D-Visulan 上にモジュール機能を実装し，モジュールの再帰呼び出しの利用例として，三目並べの思考プログラムを実際に動作させ，提案手法の有効性を示した．

謝辞　本研究を行う機会を与えてくださると同時に，丁寧なご指導を賜った湯淺太一教授に厚くお礼申し上げます．また，研究を進めるにあたり大変貴重なご助言をいただきました湯淺研究室の皆様に，深く感謝いたします．

参 考 文 献

1) Furnas, G.: New Graphical Reasoning Models for Understanding Graphical Interfaces, *Proc. CHI*, pp.71–78, ACM Press, New York (1991).
2) Yamamoto, K.: Visulan: A Visual Programming Language for Self-Changing Bitmap, *First International Conference on Visual Information Systems*, pp.88–96 (1996).
3) Yamamoto, K.: 3D-Visulan: A 3D Programming Language for 3D Applications, *Pacific Workshop on Distributed Multimedia Systems*, pp.199–206 (1996). http://www.yuasa.kuis.kyoto-u.ac.jp/ylab/yamakaku.
4) Cypher, A. and Smith, D.C.: KidSim: End User Programming of Simulations, *Proc. CHI*, ACM Press, New York (1995).
5) Bell, B. and Lewis, C.: ChemTrains: A Language for Creating Behaving Pictures, *IEEE Workshop on Visual Languages*, pp.188–195 (1993).
6) Lieberman, H.: Graphical Annotation as a Visual Language for Specifying Generalization Relations, *IEEE Workshop on Visual Languages*, pp.19–24 (1993).
7) Repenning, A.: Bending the Rules: Steps Toward Semantically Enriched Graphical Rewrite Rules, *IEEE Workshop on Visual Languages*, pp.226–233 (1995).

(平成9年 3 月6日受付)
(平成9年10月1日採録)



山本　格也（学生会員）

1971年生．1995年京都大学工学部情報工学科卒業．1996年同大学大学院工学研究科情報工学専攻修士課程修了．現在，同大学大学院工学研究科情報工学専攻博士後期課程に在学中．ビジュアルプログラミング言語に関する研究に従事．