ソフトウェア工学 97－8
（1994．3．1）

視覚的例示を用いたコーディングエディタ

竹尾 信行† 加登 基二† 平川 正人‡ 市川 忠男‡
広島大学大学院工学研究科†
広島大学工学部‡

あらまし

手続き型言語を用いたプログラムのコーディング作業とその動作確認を、視覚的情報を利用して支援するエディタを提案する。

視覚的プログラミングの研究は、具体例を動的にシステムに示すことによってプログラミングの初心者にも分かりやすくプログラムを作成させる方法と、プログラミングの熟練者に対しても、制御構造を静的に視覚表示することによって、コーディング時に構文誤りを犯すことのないようにする方法とに主眼を置いていた。前者では複雑な処理を行うことが困難である。後者では、あらたなプログラミング技法を習得する必要があり、作成したプログラムが意図したとおりの動作を行うかどうかの確認が困難である。

本システムによれば、視覚的例示とテキストコードの双方を用いることで、それぞれの利点を活かしながらコーディング作業を進めることができる。

An Editor with Visual Demonstration

Nobuyuki Takeo† Motoji Kado† Masahito Hirakawa‡ Tadao Ichikawa‡
Graduate School of Hiroshima Univ.†
Faculty of Engineering, Hiroshima Univ.‡
1-4-1 Kagamiyama Higashi-Hiroshima 724, Japan

Abstract

Studies of visual programming can be devided into two types. One is to develop the systems which help novice programmers make programs easily allowing them to demonstrate what to make by concrete examples. The other is to develop the sytsems providing programming experts with static visual representations of control structures in order to prevent them from making structural errors. The former systems are difficult to make complex programs. In the latter case, the systems oblige programmers to learn new programming rules. In addition, it is also difficult to check whether the programs are working properly when executed. The system proposed in this paper enables programmers to make programs easily allowing them to use both textual code and visual demonstration.

1 はじめに

Pascal 言語等の汎用の手続き型プログラミング言語を使ってプログラムを作成する時には、配列やリスト構造に対する処理に関しては、コーディングに先立って実際に紙に具体的な例を描いて動作を確認しながら行うことが多い。また、一度作成したプログラムでも、ソースコードを目で追いながら紙に絵を描くほうが、ただソースコードだけに頼るより動作の確認が容易に行える。

最終結果として“絵”が残らないプログラムに於いても、そのプログラムの作成過程で“絵”がイメージされる場合は、積極的に“絵”を利用することが有効である。

視覚情報を用いてプログラミングを行う方法として、例示プログラミング [3] がある。例示プログラミングにおいては、ユーザは具体的な値を使って作業を進め、システムがユーザ操作を汎化することによってプログラムが生成される。Eager [4] では、ユーザがアプリケーション上である作業を繰り返し行うと、それをプログラムとみなして残りの処理を行う。Tinker [5] では、プログラミングの初心者自身が、プログラムしたい内容を簡単な例から順にシステムに教えていくことによってプログラミング法を習得していく。Metamouse [7] では、描画ツールにおいて、ユーザが繰り返し作業の最初の数例を教える。これによって残りの作業をシステムが行う。また Pygmalion [9] では、変数を表すアイコンに具体的な値を入れてその値に対する処理を明示的に示している。

プログラムの動作を分りやすく示す方法としてプログラムの視覚化がある [1] [2] [10] 。

しかしながら、視覚言語システムの多くは、これまで視覚情報のみを利用することに固執してきた。そのため、それらのシステムでは、結果的にかえってユーザに負担を与えている面もある。プログラミング全体を視覚言語システムに頼るのではなく、視覚表現とテキストコードの双方を利用することによってプログラミングの生産性は向上するものと期待できる。

本研究では、例示の対象を配列とリスト構造に選び、例示とテキストの両方を用いて処理をコーディングするエディタを提案する。ソースコードのなかでイメージとして“絵”が表れる部分については、コメントをテキストで書くかわりに“絵”をもちいて意味する動作を残すことができる。また、テキストで表現されたプログラムの動作をアニメーションの形でユーザに見せることもでき、プログラムの作成が容易となる。

以下、2章では提案するシステムの概要を述べる。3章ではユーザの例示操作、4章ではアニメーション実現にあたっての内部表現、5章ではテキストコードの生成について述べる。

2 システム概要

2.1 プログラミング手順

本システムでは、例示を行う視覚的なアイテムを配列やリスト構造などコーディング作業時に絵として理解しやすいものに限定する。これによってユーザは容易に例示を行うことができ、アニメーション時にテキストコードとアニメーション画面上の各アイテムの対応づけも容易に理解できるようになる。

本システムを用いたプログラム開発の手順を図1に示す。ユーザは、画面上に用意した視覚的アイテムに例示を行うことによって、ボトムアップ的にシステムにテキストコードを生成させることが出来る。逆に、システムは、テキストコードから、それに対応するアニメーションをユーザに示すこともできるようになっており、プログラムが正しく抽出／記述できていなければ、アニメーション（例示）画面上でさらに例示を行うか、あるいはテキストコードに修正を加える。

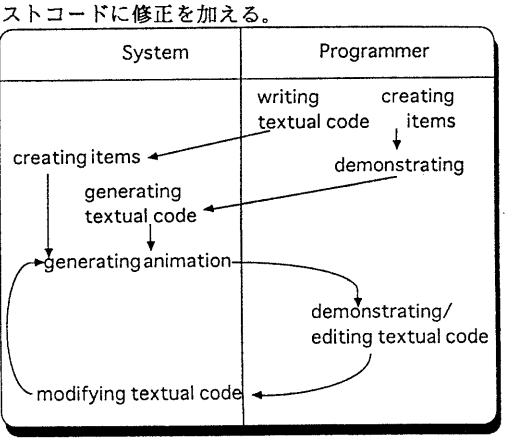


図1 プログラミング手順

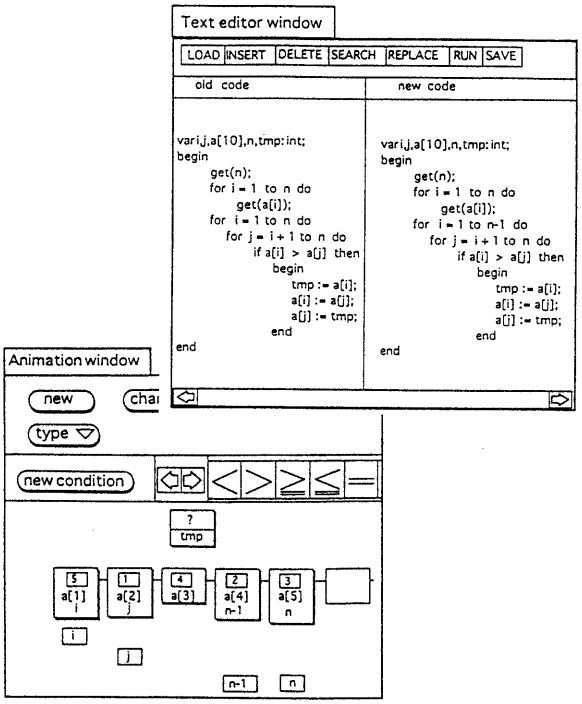


図2　インタフェース画面

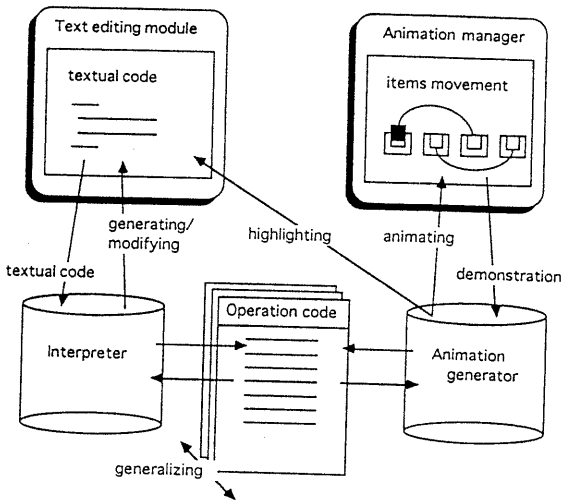


図3　システム構成

2.2　システム構成

具体的なインタフェース画面を図2に示す。アニメーション／例示用のウィンドウ (animation window) およびテキストコードを表示するウィンドウ (text editor window) から構成されている。また、ユーザからの例示によって対応するテキストプログラムの修正が行われるが、修正元のプログラムも見えるように、text editor window は2つに分割されている。

本システムスは、図3に示すようにテキストエディタモジュールとアニメーションマネジャ、インタプリタ、アニメーションジェネレータから構成される。

テキストエディタモジュール: テキストエディタモジュールでは、ユーザはマウスポインタとキーボードを使用して通常のコーディング作業を行う。テキストエディタモジュールは2画面からなる。旧テキストコード画面と新テキストコード画面である。システムは、例示を通して新しく生成したテキストコードを新テキストコード画面に表示し、それまでのテキストコードを旧テキストコード画面に移動させる。

アニメーションマネジャ: アニメーションマネジャでは、animation window 上でのユーザの操作と視覚アイテムの表示の管理を司どる。

アニメーションジェネレータ: アニメーションジェネレータは、ユーザの例示記述を内部的な操作列（オペレーションコード）に変換し、インタプリタに渡す。また、逆に操作列からアニメーション用のコードを生成する。

インタプリタ: インタプリタは、テキストコードとアニメーション操作列の相互変換を行う。

3　例示

3.1　例示の基本部品

ユーザのデモンストレーション操作や、システムのアニメーションで用いられる基本部品を以下に示す。

data_item: 例示操作の対象となる値 (数値、文字)。

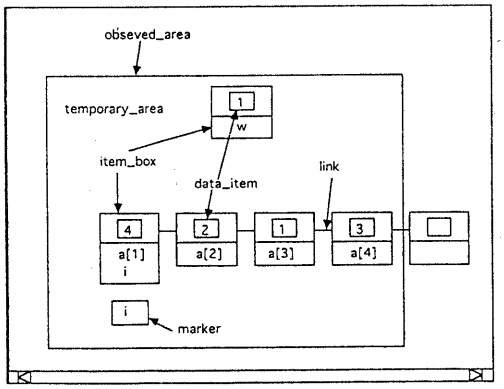


図4　基本部品とその配置

item_box: **data_item** を格納する矩形領域であり、参照可能な変数名を保持する。変数用、配列用、リスト構造用、構造体用がある。配列用では **item_box** は不可動で、それぞれの位置が配列の順番に相当する。リスト構造用では **item_box** は可動であり、それらの位置関係は矢印でしめされる。構造体用 **item_box** を使用するためにはユーザが構造体を定義する必要がある。これはソースコードで指定することもできるし、定義画面上で定義することも出来る。表示の際はあらかじめ注目している項目を指定し、通常はその項目のみを表示する。

marker: 注目している **item_box** の位置を保持する。配列ではその番号、リストではポインタを表す。ユーザの操作によってシステムが自動的に生成する場合とユーザによって明示的に生成される場合とがある。

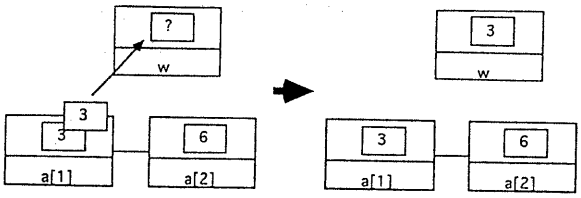
link: ボックス間の位置関係を示す。リスト構造では可動でポインタのリンク、配列では不可動で配列の順序を決める。

さらに基本部品ではないが、例示処理用に次のものを用意している。

temporary_area: **data_item** を一時的に退避させておく領域。item_box の外側はすべてこの領域である。

observed_area: 例示を行う時に注目しているデータの範囲を指定する領域。

一次元配列の場合の画面構成例を図4に示す。なお、構成手順については 3.2 で説明する。



(move_d, a[1], w)(w = a[1])

図5　drag操作と操作列

3.2　基本部品の配置

システムはあらかじめ一次元配列、線形リスト、木構造などの基本配置関係を提供しており、ユーザはそれを用いて基本部品を配置することができる。また、以下の様にして、新たに配置関係を登録することもできる。

Step1 : 配列またはリスト構造を表す item_box を選択する。定義されていなければ構造体定義画面で構造体名と各フィールドを定義する。

Step2 : item_box を2個以上配置する。システムは、その位置関係から反復的に item_box を配置する。

Step3 : 名前を付けて登録する。

marker は対応するの item_box に隣接する位置に置かれる。

3.3　基本操作

配列やリスト構造の値の操作は data_item の動きによって表現し、位置情報の変化は marker の動きによって表現する。

ユーザが例示を行う際の基本操作には以下に示すものがある。

drag: ユーザはマウスカーソルを **data_item**, **marker**,**link** 上に移動し、マウスボタンを押下げながらアイテムの移動を行う。**item_box - item_box** 間、**item_box - temporary_area** 間で **drag** が起る度にシステムは代入式を生

成する。また、必要に応じて **marker** を生成する。配列用 *item_box* $a[1]$ から整数型変数ボックス w に data_item を **drag** した時に、システムが生成する操作列を図5に示す。

set_condition: 条件コマンドの選択、およびそれが適用される **data_item** の指定。ユーザは条件コマンドメニューから条件を選び、それの引数となるアイテムを順にマウスカーソルでクリックしていく。詳細については、次節で説明する。

set_area: **observed_area** の指定。繰り返しや手続きに渡すデータの範囲指定に用いる。

change_phase: 処理の内容の変化をシステムに明示する。

return: 処理の終了および返値の設定。

new: **item_box** の生成。配列では、生成した box を基準線上に配置すれば例示のデータの増加とみなし、**temporary_area** 上に配置すれば変数の生成とみなす。

3.4 条件コマンドの定義

ユーザは画面上のアイテムを動かすことによってシステムにプログラムの例示を行うが、それぞれの動作の原因をシステムに示すことによって条件分岐を指定する。このシステムでは、ユーザはメニュー形式で条件コマンドを選択し、その後注目しているアイテムをクリックすることによって条件コマンドに渡す引数を示す。メニューに所望のコマンドがない場合には、コマンド定義ウインドウで条件コマンドの定義を行う。

条件コマンドは次の形式をしている。

条件コマンド = (**演算子**, **引数1** [, **引数2**, ..])

ここで、**演算子**は、関係演算子 (<, >, ≤, ≥, ==, ! =) または論理演算子 (&&, ||, !) である。**引数**は、int 型、char 型の変数, それらの算術演算式, 他の条件コマンドである。ユーザは定義するコマンド名を入力して演算子を選ぶ。その演算子に渡す引数が定数であればそれを入力し、変数であれば例示画面上でアイテムを選ぶ。この時、画面上のアイテムのうちどれを選んでもよいが、複数個選ぶ時にはアイテム間の相対位置が意味を持つかどうかを指定する。

4 オペレーションコード

システム内部で用いられるオペレーションコードは次の形式をとる。

$Operation\_code = (operation, arg1, [, arg2, \cdots])(additional\_information)$

各基本操作から生成されるオペレーションコードを以下に示す。

drag: $(move, from, to)(substitution)$

move : drag 操作の対象によって、move_d(data_item に対する drag)、move_m(marker に対する drag)、move_l(link に対する drag) に分けられる。

from : drag 操作が始った item_box。

to : drag 操作が終了した item_box。

substitution: drag 操作と等価な代入式であり、move_d の場合は両辺を変数、move_m,move_l の場合は右辺を item_box の位置情報で表す。

set_condition: $(opr, item\_1, item\_2)(return\_value)$

opr: 条件コマンド

item: 条件コマンドが適用されたアイテム。

set_area: $(set\_area, item1[, item2 \cdots])$

item: set_area が適用されたアイテム。

new: $(new, item)$

item: 新たに生成されたアイテム。

例として、図6 (a) に示すデモンストレーションが与えられた時に生成される操作列を図6 (b) に示す。

テキストコードからアニメーションのためにオペレーションコードを生成する際には、上述のオペレーションコードにテキストコードに関する情報を付加する。

$animation\_code = (text\_info)(operation\_code)$

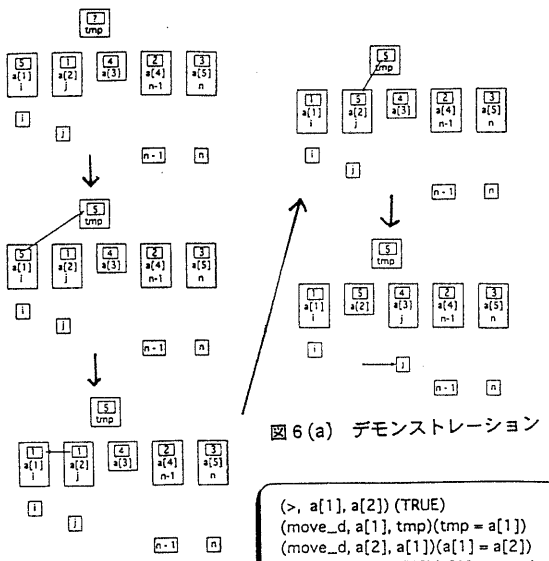
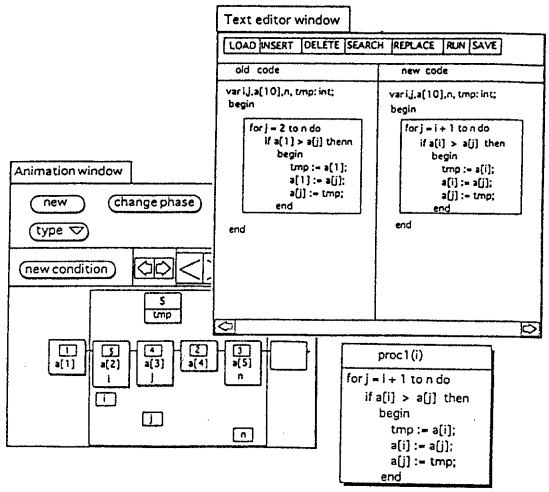


図6(a) デモンストレーション

```
(>, a[1], a[2]) (TRUE)
(move_d, a[1], tmp)(tmp = a[1])
(move_d, a[2], a[1])(a[1] = a[2])
(move_d, tmp, a[2])(a[2] = tmp1)
(move_m, a[1], a[2])( j = 2)
```

図6(b) 操作列




```
          (move_m, a[1], a[2])(i = 2)
3: proc1: (move_m, a[2], a[3])(j = 3)
4: proc1: (>, a[2], a[3])
6: proc1: (move_d, a[2], tmp)
7: proc1: (move_d, a[3], a[2])
8: proc1: (move_d, tmp, a[3])
          (move_m, a[2], a[3])(i = 3)
3: proc1: (move_m, a[3], a[4])(j = 4)
```

図7 アニメーションコードの生成

text_info は、テキストコードの行番号とそのオペレーションがどこから呼出されたかの情報である。図7にすでに記述されたプログラムからのアニメーションコードの生成例を示す。アニメーション中にユーザのデモンストレーションによって修正操作が行われた場合には、無駄なアニメーションの繰り返しをしないために、アイテムの動きで矛盾が生じないところについては再度アニメーションをしない。

5 テキストコードの生成

ユーザのデモンストレーション操作列からのテキストコードの生成に関して、システムはユーザによって最初に行われた例示で制御構造の骨組は正しく与えられているとしてテキストコードを生成し、次に行われた例示から制御の条件部の修正を行っていく。

5.1 汎化操作列の生成

ユーザの例示によって与えられた具体的な操作列から、汎化した操作列を生成する。操作列中の式のうち、marker のある item_box に対して行われた操作によって生成された式において、item_box の項の数字を marker が表す変数で書き換える。marker の移動によって生成された式は、(1.marker 自身の変化量)、(2. 他の marker との差分) という優先度で書き換える。図8に図6 (b) の汎化操作列例を示す。

```
(>, a[ i ], a[ j ]) (TRUE)
(move_d, a[ i ], tmp)(tmp = a[ i ])
(move_d, a[ j ], a[ i ])(a[ i ] = a[ j ])
(move_d, tmp, a[ j ])(a[ j ] = tmp)
(move_m, a[ j ], a[ j + 1 ])( j = j + 1)
```

図8 汎化操作列

5.2 条件部を繰り返しの条件とする繰り返しの検出

ここでは、ユーザの行った操作の中で、set_condition によって生成された式に注目した繰り返しの検出について説明する。

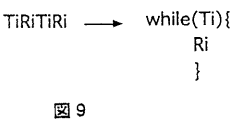


図９

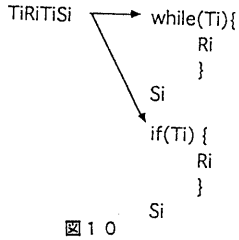
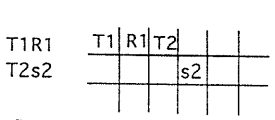
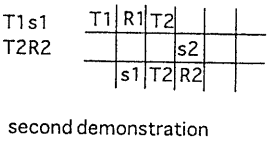


図１０





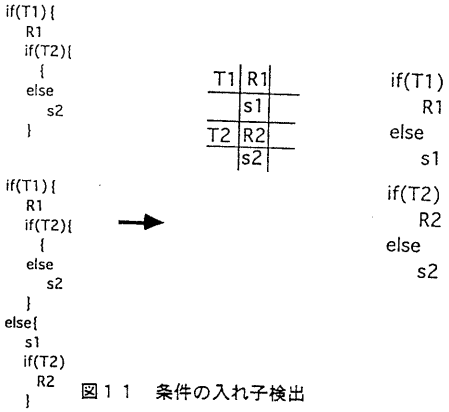


図１１　条件の入れ子検出

ユーザの操作列を条件部(T と表す)と実行部(T が真のとき R、偽のとき S と表す)に分けて考える。まず、ユーザのデモンストレーションによって得られた操作列のうち、set_condition コマンドによって生成された操作列を先頭から順に T_1, T_2, .. とラベル付けを行う。ただし、同じ操作列が表れた場合は前のラベルを使う。T で挟まれた操作列を同様に R_1(S_1), R_2(S_2), ... とラベル付けを行う。こうして得られた操作列に対して同じラベル付けをされた T_i が２つ以上現れた場合は、それが繰り返しの条件となる可能性があるかどうかを以下の様に判断する。繰り返しの可能性があれば、いったん繰り返しとみなしてテキストコードを生成し、その後ユーザに問い合せを行う。

$T_iR_iT_iR_i$: 二度同じ条件部と実行部が検出できるとそれを繰り返しとみなす（図９）。

$T_iR_iT_iS_i$: 条件部だけ検出された場合は繰り返しと条件分岐の両方の可能性がある（図１０）。いったん、繰り返しとみなしてユーザに問い合せをする。

$T_iR_iT_iR_j$: 条件部が二度とも真であるにもかかわらず実行部が異なる場合は逐次処理とみなす。

T_iS_i: はじめの条件部を偽で通過した場合は繰り返しになりえないので、二度目の条件部を監視する必要はない。

5.3 条件部を繰り返しの条件としない繰り返しの検出

ここでは、ユーザが明示的に繰り返しの条件を set_condition によって例示しない場合の繰り返しの検出について述べる。この場合は、ある変数をカウンタとする線形的な繰り返しとみなす。検出手順は以下の通りである。

step1: ユーザの操作列から item_box の位置の情報の部分、つまり数字を取り除いたコード列を生成する。

step2: step1 で生成されたコード列に対して、上に述べた手順で $TR-TR$ または $TR-TS$ が検出されると、オリジナルの操作列 T_iR_i と T_jR_j 間で差分を取る。

step3: 差分が一定値であった時、差分が０となればそれは定数である。そうでないものは T_i の数字の最小値をある変数 t とし、他の数字を (t + 最小値との差分) で置き換える。この様にして生成されたコードに対して上と同様に繰り返しが検出されると t をカウンタとする繰り返しとする。

このルーチンで検出できない場合は *loop* というステートメンをプログラムテキスト中に生成しユーザに問い合せる。

5.4 条件の入れ子の検出

例示によるプログラミングを行う場合は、すべての操作は逐次的に行われるため入れ子構造などは明示されない。ここでは、ユーザの操作列から入れ子構造を検出する手順について述べる。

条件の入れ子を検出するために次に示す表記法を用いる。T_i に関して入れ子構造は横の並び、逐次処理は縦の並びで表す。T_i が真の時の処理 R_i はその行の T_i の一つ右の列に書き、T_i が偽の時の処

理 S_iは T_iの右下、つまり R_iと同一の列に書く。このとき、S_iは T_iの行の一つ以上下の行であればどこでもよい。表は Tに関して左から右、上から下へと読む。(図１１)

Step1 : ユーザの一連の操作列はすべて入れ子構造だと仮定して、上に示した表記法で書く。R_iまたは S_iが空の時は、システムで自動的にダミーの R_i, S_iを書込んでおく。

Step2 : 新たな操作列を上と同様に書込む。

Step3 : ある i に関して R_iと S_iにつづく部分に不整合がないならばそれを T_iの列の空いている行に移動する。

図11 の例では、１回目の操作列で T_1 が真の場合の入れ子とみなした T_2 の部分が、２回目の操作列で T_1 が偽の場合にも行われたので、システムは T_1 と T_2 を逐次処理とみなしている。

6 まとめ

本研究では、視覚的部品の例示とテキストコーディングの双方を用いることで、プログラムのコーディング作業時のユーザの負担を軽減するプログラムエディタについて述べた。本システムの特徴は、視覚化を行う対象を抽象的な制御構造ではなく具体的なデータとし、テキストコードによって得られる抽象的な情報と視覚的なアイテムで表される具体的な数値の双方を用いて、プログラムの修正がテキストと例示の双方で行えることである。

現在、本システムは $X-Window$上で C 言語と XView を用いて構築中である。今後の検討課題としては、

- ユーザに効果的な例示を与えてもらうための問い合せの方法
- 例示によるプログラムの記述能力についての実験的検証

がある。

参考文献

[1] M.H. Brown "Zeus: A System for Algorithm Animation and Multi View Editing", Proc. Workshop on Visual Language, Oct 1991, p.4.

[2] K.C. Kox and G.C. Roman "Visualizing Concurrent Computations", Proc. Workshop on Visual Language, Oct 1991, p.18.

[3] A.Cypher (ed.), "Watch What I Do: Programming by Demonstration", The MIT Press, 1993.

[4] A.Cypher, "Eager :Programing repetitive tasks by example", Proc.SIGCHI, Human Factors in Computing Systems, ACM, 1991, pp.33-39.

[5] H.Lieberman, "Tinker : Example-Based Programming for Artificial Intelligence ", Proceedings of the Seventh International Joint Conference on Artificial Intelligence, IJCAI, Vancouver, August 1981, p.1060.

[6] B.A.Myers, "Demonstrational Interfaces: A Step Beyond Direct Manipulation", Computer, Vol.25, No.8, Aug.1992, pp.61-73.

[7] D.Maulsby, I.Witten, and K.Kittlitz, "Metamouse: specifying graphical procedures by example." Proc.SIGGRAPH '89, Vol.23, 1989, pp.127-136.

[8] J.Nielsen, "Noncommand User Interfaces", communications of the ACM,(April 1993), pp.83-99.

[9] D.Smith, "Pygmalion: A Creative Programming Environment ", Report No. STAN-CS-75-499, Department of Computer Science, Stanford Univ. ,1975.

[10] J.T. Stasko, "Using Direct Manipulation to Build Algorithm Animations by Demonstration ", J.ACM , Vol.24 ,NO.1 , Jan. 1977, pp. 161 -175.