ビジュアルプログラミングシステムのための UNDO 機能

A new undo method for visual programming systems

馬場 昭宏[†] 田中 二郎[††]
Akihiro BABA Jiro TANAKA

†筑波大学 大学院 工学研究科
Doctoral Program in Engineering, University of Tsukuba
††筑波大学 電子情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba

概 要

従来の UNDO 機能は時間順で管理されているために余計なステップを経なければならなかったり、すべてのバージョンを残していなかったために作業の効率が必ずしも良くはなかった。

本論文では、木構造を用いた UNDO モデルと、ミニチュアを用いたインタフェースを提案し、それらを用いてビジュアルプログラミングシステム PP のために作成した UNDO 機能について述べる。

1 はじめに

われわれはビジュアルプログラミングシステムのプロトタイプとして PP [1]の試作を行なってきた。

PP は並列論理型言語 GHC [2]に基づくシステムであり、インタプリティブな環境であるために試行錯誤を繰り返しながら目的のプログラムを作成することができる。

試行錯誤の過程においてユーザは個々の間違った操作に対して元に戻す方法を考えなければならない。このことは特に初心者にとっては問題となる。

そこで UNDO 機能が重要になってくる。UNDO とは一度行なった操作を取り消すことである。UNDO 機能を使用することにより間違った操作を元に戻すことができるため、有効である。

2 従来の UNDO

ここでは従来の UNDO 機能の例として Emacs [3]と Tgif [4]について述べる。

Emacs では任意の回数だけ元の状態に戻すことのできる UNDO 機能を提供している。通常、1 つのコマンドが使われると、変更取消し記録とよばれる記憶域に項目が 1 つ作られる。UNDO コマンドを 1 回適用すると、1 つ前の状態に戻る。これを全ての変更取り消し記録が取り消されるまで続けることができる。

ほかのコマンドを使用すると、UNDO の繰り返しを中断する。一度中断するとこれまでの UNDO コマンド自身が取り消し可能な普通のコマンドと見なされる。Emacs ではこのことを利用して UNDO の UNDO を行なう。

Emacs における UNDO の実行過程を図 1に示す。

図 1の例では、状態 4 から状態 0 にするまでに 6 ステップかかっている。これは UNDO 操作自身が記録に残っていることが原因である。通常は状態の数はこの例よりもはるかに多くなり、UNDO もより多く使われるため、目的の状態に戻るまでにはより多くのステップ数を必要とする。これでは同じ状態を行ったり来たりしているようでフラストレー

入力	番号	状態	画面	ヒストリ
	0	0	abcd	0
delete	1	1	abc	01
delete	2	2	ab	012
delete	3	3	a	0123
UNDO	4	2	ab	01232
UNDO	5	1	abc	012321
e	6	4	abce	0123214
UNDO	7	1	abc	01232141
UNDO	8	2	ab	012321412
UNDO	9	3	a	0123214123
UNDO	10	2	ab	01232141232
UNDO	11	1	abc	012321412321
UNDO	12	0	abcd	0123214123210

図1　Emacs での UNDO の実行過程

ションを感じる。

一方、Tgif では UNDO と REDO によりヒストリ中を移動するという方式を採用している。Tgif における UNDO の実行過程を図 2に示す。状態 0、1、2、3、4 は図 1の各状態に対応している。

番号 0 から番号 2 の間はヒストリには次々に新しい状態が付け加えられていく。番号 3 から番号 5 まではヒストリ自身には変化はない。変化するのは、その中での現在の状態へのポインタである。図 2では太字で表したものが現在のポインタの位置である。UNDO をすることでポインタが左 (古い方) に、REDO をすることでポインタが右 (新しい方) に動く。番号 6 では状態 4 が作られ、その時ヒストリからはポインタより後の状態 2 と状態 3 は削除され、状態 4 が加えられる。

番号	状態	ヒストリ
0	0	**0**
1	1	0**1**
2	2	01**2**
3	3	012**3**
4	2	01**2**3
5	1	0**1**23
6	4	01**4**
7	1	0**1**4
8	0	**0**14

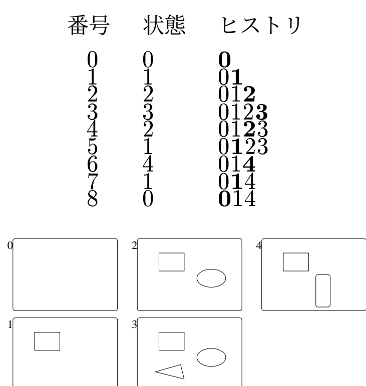

図2　Tgif での UNDO の実行過程

Tgif の UNDO 機能は Emacs と比較してより少ないステップ数で前の状態に戻ることを可能としている。しかし UNDO してから新たな状態を作ると、図 2に示したように、それまでに作った状態は消されてしまう (図 2)。これではユーザは元の状態に戻れるという前提のもとに試行錯誤を重ねることができなくなってしまう。

3　PP

PP(図 3) では図形表現とテキスト表現はそれぞれ因果結合しており、一方を入力すると他方へと変換される。図形表現は有向グラフとして表され、テキスト表現との対応は以下のようになっている。

引数　最も外側の円周上の円形。

ゴール、項　ともに円形。項はゴールよりも小さい。

論理変数　共有する引数、ゴール、項の間を結ぶ矢印。

ガード、ボディ　図 3において薄い色で表される円形領域がガード、その内側の円形領域がボディである。

図形の入力はマウスを用いて行なう。

入力された図形表現は必ずしも見やすいものであるとは限らないため、自動レイアウト機能によりレイアウトすることもできる。

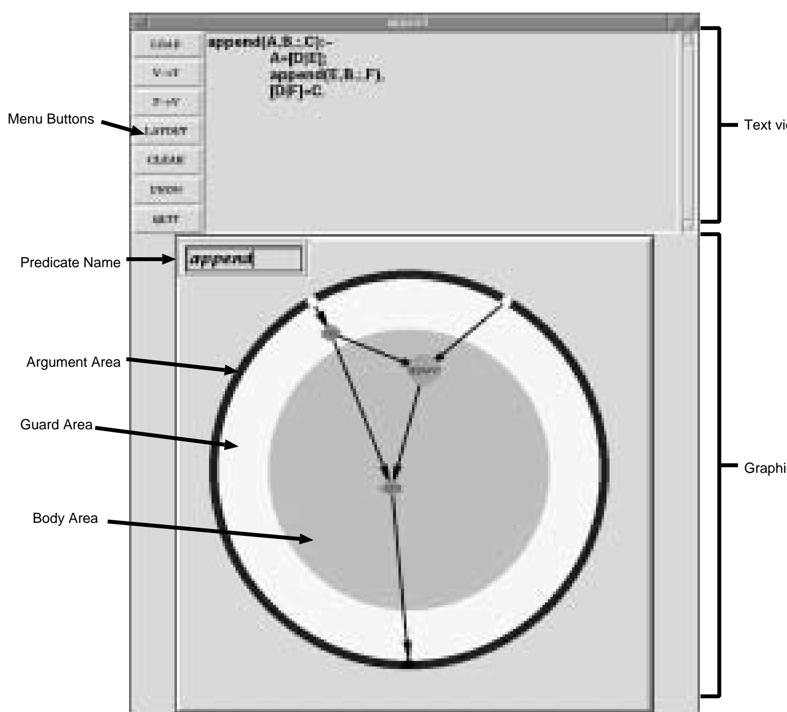

図3　PP

4 PP の UNDO 機能

Tgif の UNDO 機能において状態を削除せずに残しておくことを考えると、状態の集合は 1 つの木構造を形成する (図 4)。このようにしてできた木構造を状態の木と呼ぶことにする。

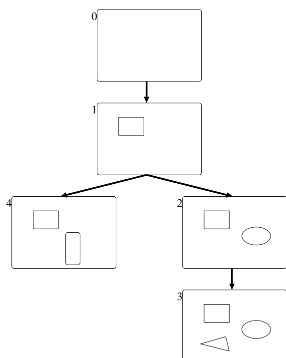

図 4 状態の木

これらの考察に基づき、図 5に示すインタフェースを試作した。新しい状態が作られると、その状態のミニチュアが前の状態の子として新たに表示される。各ミニチュアをマウスの左ボタンでクリックすることでその状態に移ることができる。このインタフェースの図において、兄弟どうしは若いものほど左に配置される。

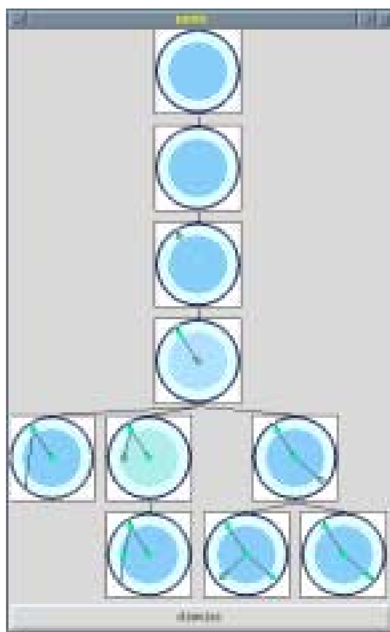

図 5 試作した UNDO インタフェース

状態の数が増えるに従って UNDO のインタフェースは大きくなるため、全体を見渡すのが困難になる。この問題を解決するための方法として、本システムでは保存される状態の数を制限する方式を採用した。

この方式では状態の数が上限に達し、さらに新たな状態が作られたときは状態を一つ削除する。削除の対象となるのは状態の木において現在の状態からの距離が最も大きいものである。ここで、状態 ab 間の距離とは、状態の木を描いたときに状態 a と状態 b とを結ぶエッジの本数の最小値である。

削除の候補が複数存在する場合、最初に作られたものを削除する。

なお、ユーザが不必要な状態を意図的に削除することもできる。

5 作成したインタフェースの評価

4節で述べたインタフェース (allTree) の他に、 2 つのインタフェースを作成し、それらの比較を行なった。

比較のために作成したインタフェースは次のものである。

linearGraph(図 6) Emacs の UNDO に似たインタフェースである。図 6において、中央の current と書かれた円の左側にあるミニチュアをクリックすると記録された作業履歴の一つ前に UNDO することができる。
Emacs では UNDO 以外のコマンドを使うことで UNDO の UNDO を行なうが、 linearGraph では中央の右側にあるミニチュアが Tgif の REDO に相当するような機能を持ち、それによって一つ先の状態に移ることができる。

allGraph(図 7) ミニチュアを作成された順に正方形の領域中に並べたものである。ミニチュアをクリックすることでその状態に移る。

図 6 linearGraph

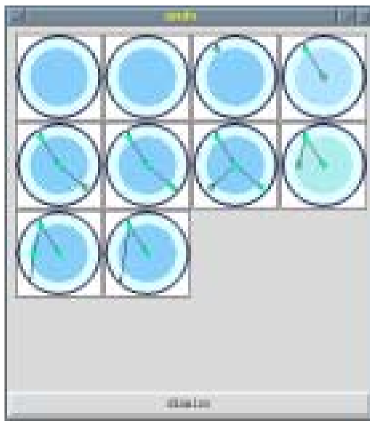

図 7 allGraph

行なった実験は次のようなものである。

このシステムをはじめて使う 9 人の被験者に

1. 図 8に示す 4 つの状態を UNDO を用いながら a 〜 d の順に作る。
2. 状態 d から状態 a に UNDO する。

という作業を上述の 3 つのインタフェースそれぞれについて行なってもらった。

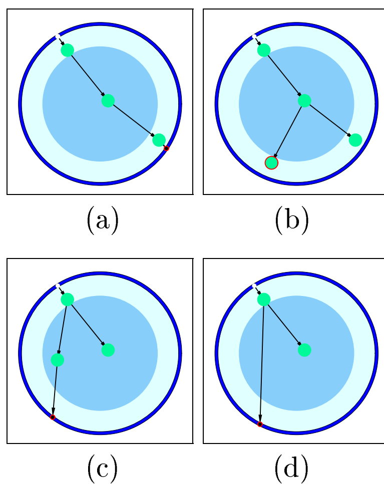

図 8 作る状態

このうち、 2をするのにかかった

- 時間 (t_{sum})
- ステップ数 (s_{act})

を測定し、それらから 1 ステップ当たりの時間 (t_{one}) を次式により求めた。

$$t_{one} = \frac{t_{sum}}{s_{act}}$$

測定を行い、平均を求めた結果は表 1のようになった。

	$t_{sum}[sec]$	$s_{act}[step]$	$t_{one}[sec/step]$
allTree	19.1	1	19.1
linearGraph	26.4	7	3.8
allGraph	19.3	1	19.3

表 1 各インタフェースの測定の結果

この結果を見ると合計時間は allTree、 allGraph とも 19 秒程度、 linearGraph で 26 秒程度であり、allTree は linearGraph よりも少ない時間で目的の状態に UNDO することが可能であることがわかる。

ステップ数においても、 allTree、 allGraph は 1 ステップ、 linearGraph は 7 ステップであり、木構造を用いたインタフェースが有効であることがわかる。 allTree は、その性質から必ず 1 ステップで目的の状態に移れるので、状態の数が増えると更にその有効性が出てくると思われる。

1 ステップあたりの時間においては linearGraph が 4 秒程度であり、 allTree、 allGraph の 19 秒程度と比較するとはるかに短いが、ステップ数が多いことが linearGraph の不利な点である。例えば一つの状態を描画するのに時間がかかるようなシステムにおいては 1 ステップあたりの時間は増えてしまい、操作感が低下する可能性がある。

また、今回の実験では allTree と allGraph の違いはほとんど出なかったが、それは作られる状態の数が少ないためであろう。状態の数が多くなると単に作成順に並べるよりも作成した構造を反映した木構造の方が目的の状態を探すのに有利であると考えられる。

参考文献

[1] 田中二郎, 後藤和貴, 馬場昭宏. ビジュアルプログラミングシステムにおける入力法の効率化. 日本ソフトウェア科学会第 12 回大会論文集, pp. 165–168, 1995.

[2] 淵一博監修, 古川康一, 溝口文雄共編. 並列論理型言語 GHC とその応用. 知識情報処理シリーズ 6. 共立出版, 1987.

[3] RichardStallman 著, 竹内郁雄, 天海良治監訳. GNU Emacs マニュアル. 共立出版, 1988.

[4] William Chia-Wei Cheng. Url:http://bourbon.cs.ucla.edu:8001/tgif/