アルゴリズム 60－9
（1998．1．22）

Visual Debuggerにおける履歴情報の保存と利用

森 憲一† 金田和文* 山下英生* 山下雅史*
(† 広島大学大学院 工学研究科 * 広島大学工学部)

概要:

平田と吉田 [1] はプログラム中のデータ構造やアルゴリズムを可視化することでデバッグを支援するシステム Visual Debugger を提案した。さらに平田 [2] は、C言語で記述されたプログラム中のユーザが指定した数値データと幾何図形との対応関係を可視化するシステムを製作した。

しかし、そのシステムはデータと図形の対応づけに重点を置いており、バックトレースなどのデバッガとしての基本的な操作すら実装されてはおらず、過去の状態参照をする機能もない。

そこで本研究では、有用なデバッグ情報の一つである作業履歴の保存について検討する。また、デバッグ作業中におけるプログラムの状態変化の履歴の保存機能と履歴情報に対するいくつかの操作を Visual Debugger に実装するとともに、作業履歴の保存と利用のためのユーザインターフェースの開発も行なったので、それについても報告する。

The Storage and Usage of Trace Information Using Visual Debugger

Ken'ichi Mori†, Kazufumi Kaneda*, Hideo Yamashita*, Masafumi Yamashita*
(† Graduate School of Engeneering, Hiroshima U.
* Faculty of Engeneering, Hiroshima U.)

Abstract:

Hirata and Yoshida [1] proposed a system called Visual Debugger for supporting programmers in debugging software by visualizing the behavior of programs. Hirata [2] also built a system that can visualize the relationship between numerical data used in a program coded in C or C++ and geometrical objects specified by a programmer.

However, basic functions that display the results of past operations (e.g. back-trace) have not yet been implemented, since the system focuses on visualizing the relationship between data and objects.

In this paper we discuss how to store and use trace information when debugging programs, as well as implement the function of storing trace information and develop a user interface to the Visual Debugger.

1 はじめに

幾何図形を扱うアルゴリズムの開発やそのデバッグを行なう際、その動作過程を正確に把握することが問題となる。これは、従来のデバッグ方法が数値情報のみに依存していたため、アルゴリズムの動作を直観的に把握することが難しいからである。殊に計算幾何学の分野においては様々なアルゴリズムが開発されているが、その実装においてデバッグ作業が難航することは容易に想像できる。

アルゴリズムに起因するバグを発見するには、プログラム中のデータ構造の論理形状と実行中の各ステップにおける処理過程の可視化を行ない、それらを関連づけてグラフィカルに表示することが望ましい。従来のアルゴリズム可視化手法として、

- プログラム毎に個別のルーチンを作成する
- 既存のアルゴリズム可視化システム [3] [4] を利用する

などが考えられるが、前者は生産性の面で非常に効率が悪い。後者に関しては過去にいくつかのシステムが開発されているが、既存のアルゴリズム可視化システムは完成されたアルゴリズムの動作過程の把握を主眼としているため、アルゴリズムの開発およびそのデバッグには不向きである。

そこで平田と吉田 [1] は、グラフィックス表示機能を利用してアルゴリズムの開発およびそのデバッグをサポートするアルゴリズム可視化システム Visual Debugger を提案し、その概念を基にその有用性を実証するためのシステムを製作した。

さらに平田 [2] は、CおよびC++言語で記述されたプログラム中の数値データと幾何図形との対応関係をユーザが指定することにより、それらの可視化を行なうことができるシステムを製作した。

しかし、そのシステムはデータと幾何図形の対応づけに重点を置いており、バックトレースなどのデバッガとしての基本的な操作は実装されていない。また [2] のシステムを用いて実際にデバッグを行なう際には、状態の巻き戻しや異なった入力列との比較を行なうなどの補助機能が望まれる。

そこで本研究では、有力なデバッグ情報の一つである作業履歴の保存について検討し、デバッグ作業中におけるプログラムの状態変化の履歴の保存機能と履歴情報に対するいくつかの操作を Visual Debugger に実装するとともに、作業履歴の保存と利用のためのユーザインターフェースの開発およびその実装を行なったので、それらについて報告する。

2 ビジュアルデバッガ

[1] での提案を要約すると、まずデータ構造の論理形状と、その中に蓄えられたデータを描画するとともにアルゴリズムの動作過程を可視化する。次に、データ構造に蓄えられたデータと幾何図形との対応関係を可視化することにより、ユーザはアルゴリズムの動作を直観的に理解することができる。この提案システムは、コンパイラや OS のエラーメッセージなどには現れない、純粋にアルゴリズムのみに起因するバグの発見を支援するデバッグシステムとしての利用が可能である。

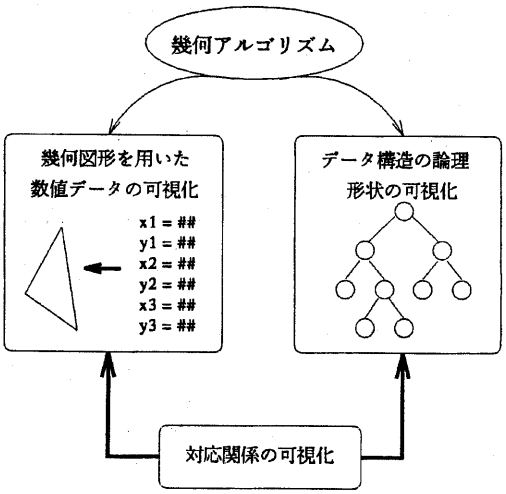

図 1: Visual Debugger の概念図.

この概念を基に [2] で製作されたシステムは、以下のように動作する。CおよびC++言語でコーディングされたデバッグ対象であるプログラム (アプリケーションプログラム) をシステムの子プロセスとして起動し、ソケットを用いたプロセス間通信を用いてデータの授受が可能な状態とする。

ユーザは、マウスを用いてシステムのメインウィンドウから幾何図形を入力する。次に、システム

はアプリケーションプログラム側に入力幾何図形を記述する数値データを送信する。アプリケーションプログラムは入力を受取り、それに対してアルゴリズムを動作させ、その中で起こったデータ構造の変化などをシステム側に送り、それらがシステムのウィンドウ上で視覚的に確認できる。ユーザは、データ構造と幾何図形との対応関係を以下のように定めることができる。アプリケーションプログラムに特定の関数を埋め込むことで、指定した変数の値をシステムに転送し、可視化システムに送られてきたデータに対して幾何図形を対応づける。

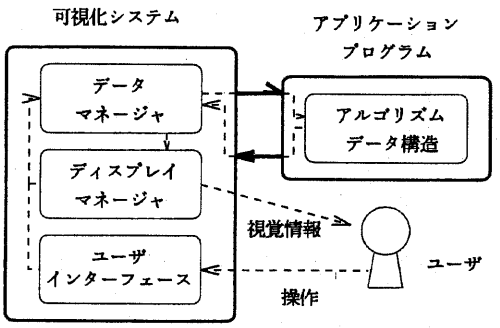

図 2: 平田 [2] のシステムの概念図.

しかし、実際にこのシステムを用いてプログラムのデバッグを行なう際、以下のような問題点がある。

- バグと思しきプログラムの不審な振舞いを視覚的に確認できるが、その機会は一度きりであるかもしれない。つまり、見逃す、またはバグかどうかの判断が難しいイベントに対して、再度同じ状況で確認することが困難である。
- バグが不審な振舞いそのものではなく、過去のイベントに依存する可能性がある場合、過去の作業履歴を参照できない。
- いくつかの入力列を与えて対応する動作の比較を行ないたい場合、それらの作業を別々に行なわなければならない。しかしその場合、視覚的に比較することができない。

そこで本研究では、過去の作業履歴の保存とその利用を検討する。

3 履歴情報の保存と利用

本稿では、以後、過去のプログラムの状態に関する情報を単に履歴情報と呼ぶことにする。入力データ列、その入力によるユーザプログラムの振舞い、システム内のデータ構造の変化、ウィンドウ情報の変化などがこれに含まれる。

3.1 履歴情報の保存

履歴情報はデバッグに有用であるが、デバッグ作業開始時から全ての履歴情報を保存しておく必要はなく、かといってわずかな情報では過去の状況が正確に把握できない可能性もある。履歴情報を保存する手段として、

1. 状態が更新されるたびに可視化に必要な全データの複製を作る
2. 入力列を保存しておき、あらゆる状態を初期状態から再構築する
3. 状態の差分情報を定義、保存する

などが考えられる。明らかに 1. は記憶領域の浪費であるし、2. は時間の浪費である。[5] では二分木構造の更新の差分情報をデータ構造内に蓄えることで記憶領域と時間とを均等に節約できる更新操作を紹介している。しかし、本システムではリスト構造や二分木構造などに加え、それらを階層的に組み合わせた複雑なデータ構造を扱うことができるため、差分情報の定義が困難である。

そこで本研究では 1 と 2 の中間を採用し、以下の要領で履歴情報を保存する。まず、1 つの入力データに対してプログラムが一連の動作を終了するまでを 1 ステップとする。ユーザは 1 ステップ毎に任意にその時点の状態を保存することができる。このとき保存される情報は過去の状態との差分ではなく、その状態を再現するために必要となる全データである。同時に、各ステップにおける入力データを保存する。

以上の操作を行なうことで、ユーザは任意の状態から (1) 過去に保存した任意の状態の参照とその状態への復帰、(2) 過去のあらゆる状態の再現が可能になる。またこれらの応用として、(3) 異なる入力列を与えて比較を行なうことができる。

3.2 Visual Debugger への実装

Visual Debugger は、可視化システム部とアプリケーションプログラム部に分けられる。可視化システム部では、現在の状態の可視化に必要な全てのデータを特定の構造体にすべて収まるような仕様に変更することにより、状態の保存という操作を構造体の複製をつくることで行なうことができる。

アプリケーションプログラムには状態履歴を保存する機能はない。そこで、現在の状態を記憶するためにプログラム (プロセス) の状態を保存する。実際には fork システムコールを用いてプロセスのをコピーし、生成された子プロセスに以後の処理を任せることにより、自分はユーザが保存しておきたい状態を維持できる。図 3 に模式的な図を示す。

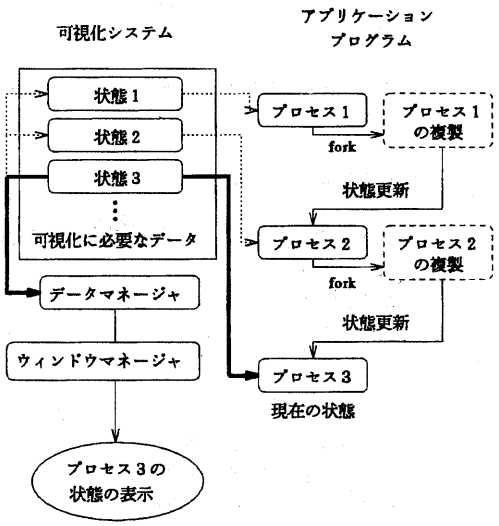

図 3: 提案機能概要

図中の太線は現在入力データのやりとりを行なうことができるコネクションを表す。これを active なコネクションであると定義する。システム内にはそれぞれのプロセスに対応した画面表示を行なうことができるのに十分なデータを保持している。

以下は実装の詳細である。

3.2.1 状態の保存

状態の保存は、以下のアルゴリズムで行なわれる。

1. 可視化システム (以下 VS) はユーザから状態の保存要求を受けとると active なユーザプログラム (以下 UP) のプロセスにシグナルを送る。このとき UP は入力待ち状態にある。

2. UP は 1. のシグナルを受けとると割り込みルーチン SaveState に制御が移る。

3. SaveState は fork() システムコールを呼び自分のコピーを作り、その子プロセスのプロセス ID を VS に送り、pause() する。その間、作成された子プロセスは pause() している。

4. VS は UP からその子プロセスの ID を受けとり、そのプロセスに対するソケットを用意し、その子プロセスにシグナルを送る。

5. UP の子プロセスはシグナルを受けとり、接続要求を送る。

6. 要求が accept されると処理完了となる。

7. VS は状態を再現するために必要な事項をパックし、active ポインタをその構造体へセットする。

3.2.2 過去の状態への復帰

過去の状態への復帰は active ポインタを戻りたい状態に対応する構造体にセットすることで達成される。ただし、このままで入力を行なうと復帰した状態が壊れてしまうため、前出の手順で状態を保存し、新たに作成した状態 (復帰した状態と同じ) に対して以後の操作 (データ入力等) を行なう。

3.2.3 入力列の保存

現在のシステムでは、メイン画面に点、直線などの幾何図形を直接入力し、マウスポインタの座標からユーザプログラムに送る数値データを抽出し、特定の可変長構造体にパックする。そこで、ユーザプログラムへデータを送る直前にこの構造体の複製をつくることで入力を保存する。また、この構造体をリスト構造にすることにより、ある状態からある状態へ移行した際の入力列を保存できる。

3.2.4 過去の入力列の再現 (トレース)

ユーザの入力の履歴に従ってアプリケーションをもう一度走らせることで過去の状況を再現する。ユーザは保存された任意の状態二つを指定し、古い方の状態を記録した構造体のメンバである入力データ列のリストを順方向に辿ることで過去の入力を再現できる。このとき古い方の状態を変更しないために、3.2.1 に述べた方法で状態のバックアップをとっておく。トレース中のある状態を新たにセーブすることも可能である。また、その時点からトレースを再開するか入力モードに戻るかを選択できる。トレース中のある状態でロードすると、ユーザはトレース中の状態をセーブするかどうかを選択する。そうでない場合はトレース終了時には初めにユーザが選択した状態の内、新しい方とまったく同じ状態にある。この状態の情報は、単に過去の履歴の参照だけが目的であった場合は無用なデータである。よって、特に指定のない場合は自動的に削除される。

4 ユーザインターフェース開発

先に実装した機能を最大限に活用するためのユーザインターフェースについて述べる。

4.1 設計方針

現在ではシステムウィンドウがスクリーン全体に占める割合がかなり高いため、履歴情報管理用のウィンドウサイズはできるだけ小さいものにとどめる必要がある。したがって、ユーザインターフェースはユーザに対して履歴情報をコンパクトかつ容易に理解できる形で提示し、その情報に対する操作を単純なやりとりで可能にすることが望まれる。

4.2 履歴情報の可視化

3 章で述べた機能を利用してデバッグ作業を行なっていく状況は、作業を開始した初期状態を根とし、ユーザが保存した状態をノード、入力列をエッジとする木に例えられる。この木を Context Tree と呼ぶこととする。ユーザインターフェース設計の際、作業履歴の直観的な表現である Context Tree をいかにわかりやすい形でユーザに提供できるかが問題となる。つまり、ユーザがこのツリーの示す情報から状態の履を直観的に把握できなければならない。

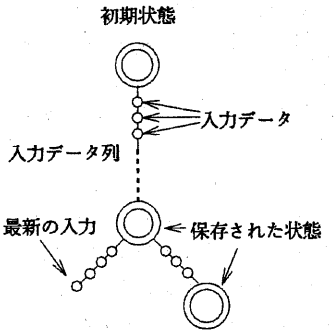

図 4: Context Tree

4.3 レイアウト

これまでの議論を踏まえて、ユーザインターフェースのレイアウトを以下の様に決定する。

4.3.1 基本的な枠組

既に存在するデバッガ本体の画面占有率を考慮し、ウィンドウの初期サイズは小さめにとる。ウィンドウの左側 150 ピクセル程度を Context Tree において現在選択中のノードの詳細表示に用い、残りを Context Tree の表示に用いる。ただし、ユーザが木の表示範囲を広げる事の出来るように Context Tree 部分のウィンドウサイズは可変とする。ウィンドウ上部にはツールバーを配置し、各機能へのアクセスを提供する。また、ウィンドウ下部にはステータスバーを用意し、提案機能を利用する際にシステムがこのエリアに簡単なメッセージ (復帰する状態に対応するノードを選んで下さい等) を表示させ、ユーザに指示を与える。

4.3.2 Context Tree の表示

インターフェースは、Context Tree をグラフィカルに表示する。ツリーディスプレイエリアにおける木のノードのラベルは、保存された順番と重

要度を表示する。ここで、重要度はユーザーが決める値で、状態保存時に3段階で設定する。(途中での変更も可能である。) 木の表示においては、ノードの色の濃淡を変えて重要度が一目で分かるようにする。

Context Tree の次数はその性質上制約できないので、一般的な表示が困難である。そこで実際の表示部分には木の一部をハイライト表示し、残りの部分はスクロールバーを用いて表示部分の変更を行なう。

また、ユーザは木のノードを直接クリックすることでそれぞれに対応した状態を参照できる。クリックされたノードは太枠で示され、その状態にカレントのフォーカスがあることを意味する。

インターフェースはノードの情報として

1. シリアルナンバー
2. 保存日時
3. その時点のステップ数 (入力データの個数)
4. 重要度
5. その状態に関するメモ
6. メインウィンドウの縮小表示

を持つことができる。ユーザはこれの中から表示させたい情報を選択し、ノードがクリックされた時にその情報を表示させることができる。

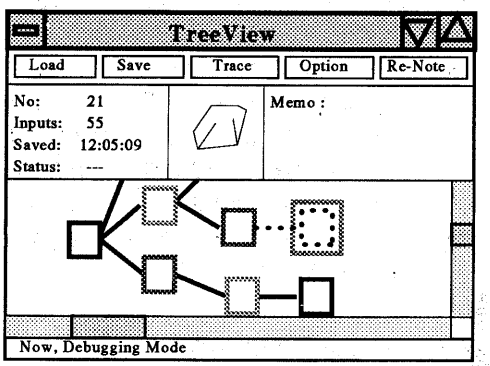

図 5: ユーザインターフェースの概観.

4.4 履歴情報の利用

提供する機能は以下の通りとする。

1. 現状態の保存 (Save)
2. 保存状態の呼出 (Load)
3. 二状態間のトレース (Trace)
4. オプションメニュー (Option)
5. 重要度、メモの書き換え (Re-Note)

ユーザはメニューバーから対応するボタンを選択することで機能を実行できる。

4.4.1 状態のセーブ

Context Tree の根のノードに関しては、システムが初期状態を自動でセーブする。以後システムから入力が与えられると、Context Tree のカレントノードに淡色表示の子ノードが表示される。これは現在入力を受け付けることができる状態を指し、保存された状態と区別される。この状態に Context Tree のカレントフォーカスがある場合にセーブボタンが押されると、既存のノードと重複しないかどうかのチェックが行なわれ、ユーザーに対して重要度とメモ書きを促すダイアログを表示する。チェックの方法としては、入力データ数の比較を行ない、一致したものについては入力列の比較を行なう。状態のセーブ及び付加情報変更時の入力は以下のようなダイアログをポップアップして行う。

4.4.2 状態のロード

目的ノードをクリックして詳細表示を出してもらい、その上でロード機能を呼び出すことにより、その状態へとジャンプする。

ロードの際には実行確認のダイアログを表示し、また、トレース実行中の状態や現在進行中の保存されていない状態にある場合は、その作業を中断する (現在の状態を保存することなく過去の状態に戻る) かどうかの確認ダイアログを表示する。

4.4.3 トレース

トレースを呼び出すと、まず始点となるノードを選択させ、終点として選択不可能なノードの色

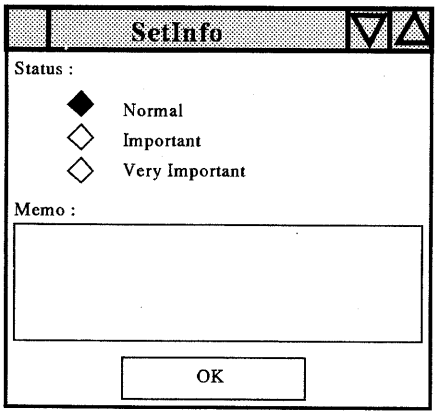

図 6: セーブ／変更ダイアログ.

を変更して選択しやすく表示を更新した上で、終点を選択させる。

二点が選択されたら始点と終点双方の詳細情報を同時表示兼、実行の確認ダイアログを表示する。

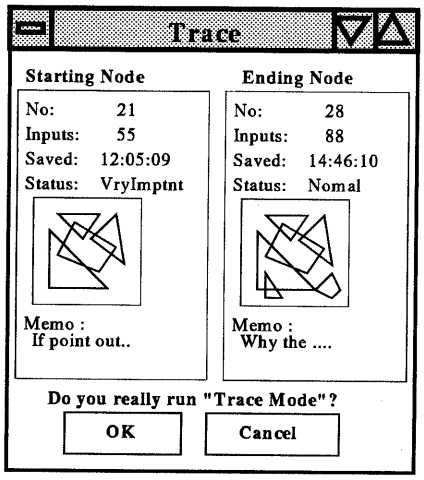

図 7: トレース実行確認ダイアログ.

実行されると、自動か手動かを制御するコントローラを表示する。自動は過去の入力列を自動で与えていき、手動はユーザが入力を一つずつ与える方法である。これとは別にシステムのステップモードがある。これは、アプリケーションプログラムから可視化情報が送られてくるたびにユーザがボタンを押して次に進むマニュアルモードと、10段階に表示速度を変更できるオートモードのどちらかを選んでデバッグ作業を行なうことができる機能である。その機能と併用することで、ユーザは自由にトレースを行なうことができる。

4.5 オプションメニュー

オプションメニューはプルダウンメニューになっており、

- 4.3.2で挙げたノードの情報の表示／非表示の選択
- トレースモードの自動、手動の選択

をラジオボタンを用いて行なう。

4.6 重要度、メモの書き換え (Re-Note)

このボタンを押すと、図6のウィンドウがポップアップされ、状態の保存時と同様の操作で重要度、メモの変更を行なうことが可能である。

5 おわりに

本稿では、Visual Debuggerにおける履歴情報の有用性を指摘し、

1. プログラムのある状態の保存
2. 過去に保存した状態への復帰
3. 過去のあらゆる状況の再現

を行なう機能の導入を提案した。また、Visual Debuggerに対し、提案機能およびユーザインターフェースの実装を行なった。

今後の課題としては、まず使用できる幾何図形プリミティブやデータ構造の拡張、さらには数値データと幾何図形との対応づけにおける概念の拡張が挙げられる。

また、可視化プロセスにおけるユーザの負担をなるべく軽減するためのソースコードエディタや、ユーザインターフェースの改良がなど望まれる。

参考文献

[1] 平田真章, 吉田英一, "凸包アルゴリズム可視化システムの製作- ビジュアルデバッガの開発に向けて - ," 平成 *6* 年度広島大学工学部卒業論文, March, 1995.

[2] 平田真章, "データ構造と幾何図形との対応関係を考慮したアルゴリズム可視化システムの開発," 平成 *8* 年度広島大学大学院工学研究科修士論文, March, 1997.

[3] Mark H. Brown, John Hershberger, "Color and Sound in Algorithm Animation," *IEEE Computer Vol. 25, No. 5*, pp. 52-63, December 1992.

[4] Gruia Catalin Roman, Kenneth C. Cox et.al., "Pavane : A System for Declarative Visualization of Concurrent Computations," *Jaurnal of Visual Language and Computing Vol. 3, No. 2*, pp. 161-193, June 1992.

[5] James. R. Driscoll, Neil Sarnak, Daniel D. Sleator, Robert E. Tarjan, "Making Data Structure Persistent," *Journal of Computer and System Sciences 38*, pp. 86-124, 1989.