プログラム実行点の概念に基づく デバッグパターンの抽出と自動化

丸山 一貴

目 次

第1章　背景

1.1　バグの影響と原因

現状では、製品としてのソフトウェアの中で、バグを完全に追放したものはほとんど存在しない。現代社会では生活のすみずみにまで計算機が普及しているため、バグは社会的に大きな影響を及ぼすこととなる。

例えば、携帯電話端末に搭載された Java VM のバグが発覚して端末の無償交換を行うことになれば、これに要する直接的な費用だけでなく、企業イメージをも傷つけて大きな損失を招く。また、デーモン・プロセスのバッファがオーバランするようなバグは結果として OS のセキュリティホールとなり、これを悪用するワームの蔓延という事態を引き起こす。

家電のネットワークへの接続も進められており、今後もソフトウェアの利用範囲の拡大は避けられない。また、ソフトウェアそのものの複雑化も一層進行すると考えられる。これらより、バグはより増加し、その影響は一層深刻になる。

そもそもバグが混入する（エンバグと呼ぶ）のは、いつであろうか。ソフトウェアの開発工程は概ね 5 つの工程に分けられる。要求定義、設計、コーディング、テスト、デバッグである。この中でエンバグが起こるのは要求定義、設計、コーディングの各工程である[1]。

ここで、バグを駆逐するために 2 つの方針が考えられる。1 つはエンバグを防止すること、もう 1 つはデバッグを確実にすることである。エンバグの防止を徹底すればデバッグは不要になると主張する向きもある [McC93] が、現実的とは言いがたい。従って、ソフトウェア開発ではこれら 2 つを共に実施することが重要となる。以下、第 1.2 節と第 1.3 節でそれぞれについて述べる。

1.2　エンバグの防止

本節では、デバッグ工程よりも上流の工程でバグの混入を防ぐための、代表的な方法について述べる。

Formal methods[Inc92] はプログラムの仕様を記述する段階から自然言語を排し、形式的言語で記述して曖昧さをなくすという手法である。この方法はパリの地下鉄

[1] 実際には、デバッグ工程におけるコーディング作業でもエンバグは発生するが、ここでは触れないこととする。

で採用されているSACEMや、IBMとOxfordが開発したCICSのように一定の成果を挙げている[GCR97]。しかし既に市場で流通し、バージョンアップ等の保守段階に達している多くのプログラムに適用することは難しい。

デザインパターン[GHJV95]は、オブジェクト指向プログラミングにおけるクラス階層の典型的な設計例を、パターンとして列挙している。プログラマは自分のソフトウェアの各コンポーネントを該当するパターンに当てはめることで、良い設計ための指針を得ることができる。良い設計の結果として、プログラマはコーディング工程で必要以上に複雑なコードを書くことがなくなり、間接的にコーディング工程でのエンバグを抑止する効果があると考えられる。

オブジェクト指向言語はデータのカプセル化によって、オブジェクト内部の変数を外部からアクセスできないようにできる。また、Eiffelのpre-conditionやpost-conditionのようにassertionを用いてデータの妥当性をチェックする機能を持った言語もある。こうした機能を使えばバグの原因となった場所を特定するのに有効であるが、assertionを書く労力はプログラマが担わなくてはならないし、assertionはメソッド呼び出しのたびに評価されるのでオーバヘッドは大きいと考えられる。

Extreme Programming（XP）[Bec99]はソフトウェアを細かいサイクルで段階的に開発することで、工期の短縮とともに品質の向上を目指しており、そのために開発に関わる者が取り組むべき手順や姿勢について論じている。XPでは開発計画や設計から具体的なコーディングの段階まで様々な提言を行っているが、デバッグに関するものとして重視すべきは2つある。1つはテストであり、プログラマが書くテストは単体テストと呼ばれる。プログラマはメソッドの機能を確認するためのテストや見つかった問題を独立させるためのテストなど、単体テストを書いて自分のコードの断片が正しいことを確認する。コーディングに先だってテストを書くこともある。もう1つはペアプログラミングである。1人がコーディングを担当し、もう1人はそれを見ながらコーディングの検証などを行って助言をする。これはコーディングと同時にコードレビューを行っていると考えることができる。XPの方法論はソフトウェア開発の多くの現場で効果を発揮し、バグを減らすことに貢献する。

これらの手法はいずれも一定の成果を挙げるが、1つの手法だけでバグを駆逐することはできず、組み合わせて用いることが重要である。それでも網をかいくぐるバグはあり、それらはデバッグ工程で取り除かなくてはならない。

1.3 テストとデバッグ

バグ駆逐の2つの方法のうち、前節ではエンバグの防止について述べた。本節では確実なデバッグについて述べる。

要求定義、設計、コーディングの各工程での努力にも関わらずエンバグが行われた場合、テスト工程でバグを発見し、デバッグ工程で取り除かなければならない。テストとデバッグの重要性は古くから認識されており、テスト・デバッグ工程はソ

フトウェア開発において投入すべきリソース（マンパワー、時間、コストなど）の45%を占めていると言われる [Ish86]。

コーディング工程の成果物である未テストのソフトウェアは、テスト工程で仕様を満たすかどうかの検査を受ける。テストで発覚したバグはデバッグ工程へ送られ、駆逐される。しかし一般には、テストで全てのバグを発見するのは困難である。結果として、出荷製品にはバグが含まれたままとなり、ユーザからの報告によって発覚することとなる。

一方で、テストで発見されたが修正されなかったバグというのも存在する。出荷品質を高く保つため、本来はテストで発見されたバグは全て修正した上で出荷されるべきだが、経営的判断によってデバッグが終了する前に出荷される場合がこれにあたる。

市場に流通しているソフトウェアの多くは、このような既知のバグ（存在することは分かっているが、修正が完了していないバグ）を抱えている。例えば Microsoft の Internet Explorer にはユーザから指摘されているバグで修正されていないものが存在する [Tho, Liu]。Internet Explorer はこれまでのセキュリティホールとユーザの多さとから多数の情報が集められているが、他の製品版ソフトウェアも同様の状況にあると考えられる。無料で入手できる Mozilla や Sun JDK もバグのデータベースを公開 [Moz, Suna] しており、発見されていてもなお修正されていないバグは多数存在する。これらの事実はテスト工程におけるバグ発見の困難とは別に、デバッグ工程におけるバグ修正も困難であることを示している。

デバッグ工程の時間を短縮してソフトウェア開発の効率を高めるために、また、ソフトウェアの信頼性をより高めるために、デバッグ工程のプログラマの作業を支援することが不可欠である。次節ではデバッグ工程改善の概要について述べる。

1.4 デバッグ工程の改善

デバッグ工程全体は、個別のバグを修正するデバッグ作業と、複数のデバッグ作業全体を管理する工程管理との、2つに分かれる。

一方のデバッグ作業をモデル化すると、1つの入力から2段階の出力を生成するものと考えることができる。入力は、バグの結果として現れるエラーの症状（bug manifestation）である。これにはエラーを発現させるための条件も含まれる。第1段階の出力はそのエラーが発現する経緯であり、第2段階の出力はエラーの根源を除去したソフトウェアである。

もう一方の工程管理については、前述のバグデータベース [Moz, Suna] のような追跡システムが利用されている。火星探査機 Mars Pathfinder は火星に着陸後、1日に1回、コンピュータがリセットしてしまうという現象を起こした [KP99]。このバグは発射前のテストでも発生していたにも関わらず、開発陣は見落としてしまったという。こうした問題は追跡システムを適切に利用することで解決できる。

しかし、個別のバグを解消するデバッグ作業には大きな問題が残されている。それは、この作業がその場限りの、1回だけのものだということである。デバッグ作業はバグ固有の現象に基づいて行うので、特定の状況下でしか役に立たないと考えられがちであり、デバッグ作業は熟練が必要な職人芸であると思われてきた。このため、現在のデバッグ作業には以下のような欠点がある。

- その場限りの作業であるため、デバッグ作業の戦略と戦術（デバッグのノウハウ）を体系立てて整理し、他人に伝えることができない。
- ケースバイケースで対処するため、自動化の余地がない。

一方で、プログラマはデバッグ作業の中にある種のパターンが存在することを、経験的に知っている。例えば、デバッグ中のプログラマは以下のようなことを呟くことがある。「このようなエラーの症状は、あのようなバグに起因していることが多い。となれば、以前も行ったあの方法でそれを確認し、修正すべきコードを見つけ出せるはずだ。」

このプログラマの独り言の中で、エラーの症状から原因となるバグを推測している部分（前半の一文）はデバッグ作業の第1段階の出力（バグ発現の経緯）を生み出しており、人間の知性が必要となる。しかし、その経緯を確かめて修正対象のコードを見つけ出す部分（後半の一文）は典型的な作業と考えられ、再利用と自動化の余地があるといえる。このような典型的な作業はデバッグにおける戦術であり、デバッグパターンと呼ぶことができる。

1.5 目的

デバッグにおけるプログラマの労力を軽減し、デバッグ工程にかかる時間を短縮するために、デバッグ作業中にプログラマが行う典型的な作業を自動化するための基礎技術として、プログラム実行点の概念を用い、いくつかのデバッグパターンの提案と自動化を行うことを本研究の目的とする。

ここでいうプログラム実行点とは、プログラマが持っている、プログラムの実行がどこまで進んだかという実行状態のイメージを指している。詳細は第4章で述べる。

1.6 本論文の構成

第2章では、デバッグ作業においてバグの原因を発見するための手続きを、「仮説検証ループ」として説明できることを述べる。デバッグのいくつかのケーススタディを通じて、プログラマが行っている思考プロセスとの対応を示し、デバッグパターンの正確な定義を示す。

第 3 章では、本研究の予備研究として位置付けられる、関数単位疑似逆実行に関する研究について述べる。

第 4 章では、本研究で扱う自動化のための基礎技術である、実行点（position）という概念について述べる。また、その表現方式を比較し、本研究で採用するタイムスタンプ方式について説明する。

第 5 章では、実行点のために必要となるタイムスタンプ機構を C と Java のプログラムに自動的に組み込む手法について述べる。また、実行速度とファイルサイズのオーバヘッドを示す。

第 6 章では、3 種類のデバッグパターンについて紹介し、実行点を用いてどのように実現するかを述べる。

第 7 章では、再現性のないデバッギとして Java のマルチスレッドプログラムを取り上げ、デバッグパターンを適用するために必要なスレッド切替の記録、再生の機構について述べ、オーバヘッドを示す。

第 8 章では、実行点のいくつかの表現形式の比較と、関連研究との比較を行う。

第 9 章では、本研究の結論を述べ、意義と展望について述べる。

第2章　デバッグパターン

本章では、デバッグ作業の中のバグの原因を発見する手続きを「仮説検証ループ」として説明できることを述べる。デバッグのいくつかのケーススタディを通じて、プログラマが行っている思考プロセスを明らかにし、デバッグパターンの定義を与える。

2.1　デバッグ作業の分析

バグは、端的にはコーディングの誤りである。その誤りが変数の値の誤りとして伝搬してゆき、結果として、どこかの段階でエラーの症状を発現する。デバッグ作業はバグの結果から原因を見つけ出すという時間に逆行する行為であり、本質的に困難である。

2.1.1　デバッグ作業の概要

バグのほとんどはプログラマが論理的に正しくないコードを書いてしまったことに起因しているので、自分の書いた全てのコードを精読して論理的に正しいかどうかを検証すれば良い。実際、あるバグを取り除く最後の段階では、該当箇所のコードを見て検証することになる。しかしながら大規模なプログラムでは、たとえそれが自分で書いたものであっても、最初からこの方法を採ることは難しい。従って、見るべき範囲を絞り込むことが必要になる。

範囲の絞り込みのため、プログラマは現れたバグの結果を見てその原因を推測する。例えば、表示されるはずの文字が 1 行分足りないといったような、原因を推測しやすい結果が現れるバグでは、経験豊富なプログラマにとって推測は比較的容易である。そうでないバグでは、コード中に印字命令を挿入したりデバッガを用いたりして、実行時のデバッグ対象プログラム（以下、デバッギと呼ぶ）の内部状態を検査することで、推測のための情報を手に入れる。

入手した情報を元にコードの該当箇所を調べると、エラー症状の直接的な原因を発見することができる。例えば、表示が 1 行足りないのは、ループ継続の条件部で呼び出される関数の返り値がおかしいからだ、といった具合である（図 2.1）。次のステップとして、プログラマはその関数のコードを調べる。このように、結果から原因を推測することの繰り返しによって、真のバグを見つけ出すことができる。

```
    :
while( f(i, j) ){
  printf("...");
}
    :
```

図 2.1: エラー症状の直接的な原因の例

原因の推測では、プログラマはエラー発現の経緯に関して仮説を設定している。そしてコードの該当箇所を見るなどにより、その仮説を検証する。よって、真のバグを見つけ出すための繰り返しを、「仮説検証ループ」と呼ぶことにする。

2.1.2　仮説検証ループ

McConnell はデバッギングを科学的手法と対比することで、以下のような 5 段階の手順が望ましいとしている [McC93]。

1. エラーを安定させる（確実に再現させる）。
2. エラーの原因を見つける。
3. エラーを修正する。
4. 修正を検査する。
5. 同じようなエラーを探す。

本研究では第 2 の「エラー原因の発見」に注目しており、前述の仮説検証ループはここに含まれている。本論文の残りの部分では、この第 2 を指してデバッグ作業ということにする。

遠藤は、この第 2 の段階におけるプログラマの行動を、5 段階の仮説検証ループとして示している [End03]。

1. 実験を繰り返してデータを採る。
2. エラー発現の経緯を推測し、仮説とする。
3. 仮説の検証方法を考案。
4. 検証を行う。

5. 検証の結果、仮説が棄却された場合は 2 段階目に戻る。新たな仮説の設定が困難な場合は 1 段階目に戻る。

これはデバッグ作業の分析として非常に有力である。しかし、後述するケーススタディで述べるように、現実のデバッグ作業では仮説検証ループの中に、別の、さらに細かいループが存在する。

よって、本論文ではデバッグ作業を以下の 5 つの段階に分類し、以降はこれを仮説検証ループと呼ぶ。

第 1 段階 仮説設定のための情報収集を行う。ここには、エラー発現の条件を特定する作業（入力データの変化など）や、プログラムのコードの観察も含まれる。

第 2 段階 収集した情報に基づいてエラー発現の経緯を推測し、仮説を設定する。

第 3 段階 検証方法を考案する。

第 4 段階 検証を実施する。検証が困難な場合は第 3 段階へ戻り、別の検証方法を考案する。

第 5 段階 修正すべきコードを確認した場合は終了。そうでない場合は新たな仮説設定のため、第 1 段階に戻る。ここには、仮説が棄却された場合だけでなく、仮説が立証された場合の新たな仮説設定も含まれる。

この分類では、仮説検証ループは 2 つのループを持っている。1 つは第 1〜5 段階のループであり、仮説を設定して検証し、次の仮説を設定するというループである。もう 1 つは第 3〜4 段階のループであり、ある仮説の検証方法を決定して検証を実施したものの、検証に失敗した場合には別の検証方法に切替えて、同一の仮説の検証を続行する、という場合に相当する。

遠藤の分類との相違点は以下の通りである。

- 遠藤の分類では、仮説が棄却された場合（エラー発現の経緯を正しく見抜けなかった場合）にループを繰り返すことになっている。本論文の分類では、仮説が立証された場合にも第 1〜5 段階のループを繰り返す。これは McConnell[McC93] の言う、「仮説の洗練」にあたる。これは以下のような考え方のことである。

  > 最初に立てる仮説は、エラー症状の全てを説明できなくてもよい。一部の症状を説明することができたなら、その他の症状をも説明できるように、仮説を段階的に洗練させていけばよい。

  この考え方を採り入れることで、本論文の分類はデバッグ作業をより包括的に説明することに成功した。

- 本論文の分類では、仮説は変えずに検証だけをやり直す、第 3〜4 段階のループが存在する。実際のデバッグ作業では、検証のために選択した方法が不十分であることに気づいたり、あるいは余りにも時間的コストが高いことに気づいたりすることで、検証方法の変更を余儀なくされることがある。この小さなループはこれを反映するためのものである。

本章の残りの部分では、この仮説検証ループの妥当性を示すために実際のデバッグ作業のケーススタディを見ていき、最終的にデバッグパターンとその自動化の定義を与える。しかしその前に、第 3 段階と第 4 段階で必要となる、デバッギの内部状態の取得方法について述べておく。

2.2　デバッギの内部状態の取得

実行時のデバッギの内部状態を取得する方法は、コード中に印字命令を挿入する方法（以下、printf デバッグと呼ぶ）とデバッガを用いる方法とがある。

2.2.1　printf デバッグ

printf デバッグでは、デバッギ中の必要な箇所に C における printf のような、変数の値を表示する命令を追加しておく。これを再コンパイルして実行すれば、デバッギがどこから期待通りに振舞わなくなっているかを調べることができる。

デバッグの経験が豊富なプログラマにとってはこれで十分な情報を取得できる場合も多い。しかし一般には、かなり単純で小規模なプログラムの、ごく簡単なバグを取る時にしか適用できない。大規模なプログラムや複雑なバグに対してこの方法を行うと、大量の printf 文を挿入して様々な変数の値を表示する必要が起こる上に、膨大な量の表示を解析しなくてはならない。

2.2.2　デバッガの機能

デバッギの内部状態をより効率良く取得するため、デバッグを補助する専用のツール、デバッガが用いられる。デバッガは OS とハードウェアのサポートを利用して非常に豊富な機能を提供するが [Ros97]、主に利用されるのは以下のようなものである。括弧内に、代表的なデバッガである GDB[SPS+00] での対応するコマンド名を示す。

- デバッギの状態を検査するコマンド（状態検査コマンド）
  - 変数の値を表示（print）
  - 変数の値を変更（set）

- デバッギの実行を制御するコマンド（実行制御コマンド）
  - ステップ実行：実行をコード行単位で進める（`step`、`next`）
  - ブレークポイント：特定のコード行に制御が到達したらデバッギの実行を停止する（`break`）
  - ウォッチポイント：特定の変数（メモリ領域）にアクセスがあったらデバッギの実行を停止する（`watch`）

デバッガを利用するプログラマは、これら状態検査コマンドと実行制御コマンドを相補的に利用してデバッグを行う。

デバッガによるデバッグ作業では、デバッギの再実行を繰り返して仮説検証ループを進めていく。これは cyclic debugging と呼ばれる [RBdK00]。

2.3 ケーススタディ

本節では、第 2.1.2 節で示した仮説検証ループの妥当性を示すため、3 つの具体的なデバッグ作業を例に取り、仮説検証ループにどのように当てはめられるかを述べる。

2.3.1 ケース 1：出力データの異常

エラーの症状 デバッギはテキストデータを出力するプログラムであるが、出力データが常に 1 行足りないというエラー症状を示していた。

仮説検証ループ（1 回目） プログラマは 1 行足りないという実行結果から、出力に関するループが 1 回足りないのではないかという疑いを持った。そこで、プログラマは `printf` デバッグを実施して、その表示内容から出力ループの最後の 1 回が足りないことを確認した。

ここで、プログラマは検証方法として `printf` デバッグを選択したが、他にもデバッガを用いて検証する方法がある（図 2.2）。例えば、デバッガはブレークポイントのヒット回数を数えるので、その回数を利用する方法がある。また、デバッガの実行制御コマンドを利用して出力ループ終了直後に制御を移動し、状態検査コマンドを利用して出力ループの制御変数の値を調べることで、出力ループの回数を確かめられるかもしれない。設定した仮説を検証する方法は複数存在し、プログラマはその中で、そのケースに適していると考える方法を選択する。

仮説検証ループ（2 回目） 次に調べなければならないのは、なぜ出力ループの最後の 1 回が足りないか、である。そこでプログラマは出力ループの継続条件部を調べ、以下のような記述を見つけた。

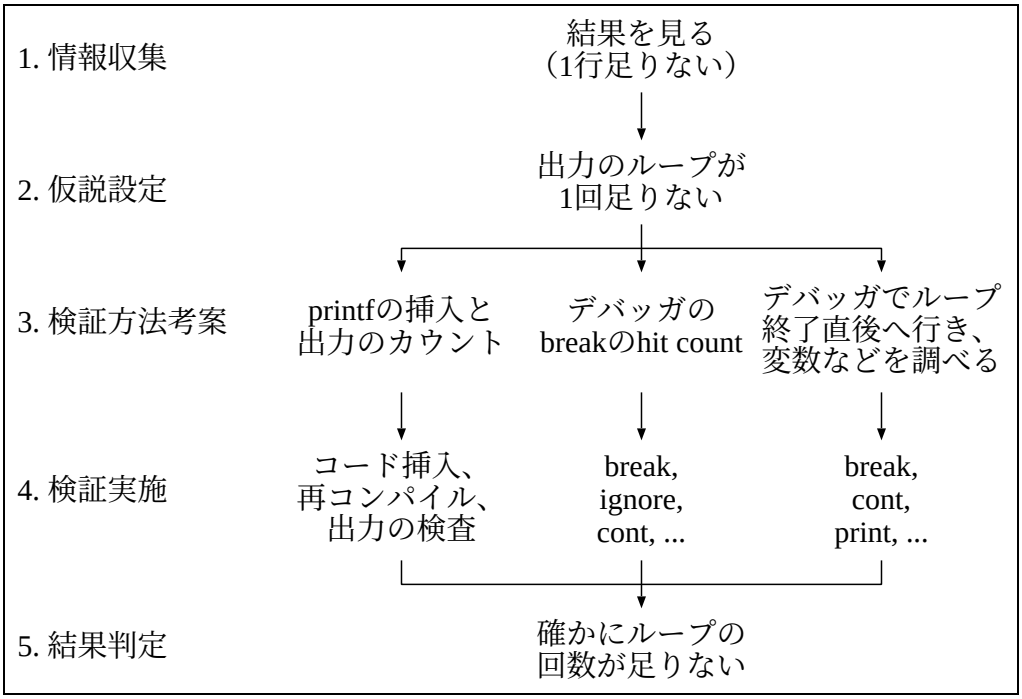

図 2.2: ケース 1 の仮説検証ループ（1 回目）

```
for(i = 1; i < n; i++)
```

これは本来 <= であるべきところを < と書いてしまっているのではないかと考えた（図 2.3 の左側）。思考実験による検証を行った結果、エラー発現の経緯を合理的に説明できるので、これがバグであると確信し、コードを修正して正しい出力となることを確かめ、デバッグを完了した。

このように、検証方法には思考実験のような、具体的な手作業が伴わないものも含める。

ここでは比較的単純なミスであったが、もし出力ループの条件部が以下のようであったらどうであろうか。

```
while( f(i) )
```

この場合、関数 f() の返り値が正しいかどうかが問題となるので、2 回目の仮説検証ループの仮説は「関数の返り値が不正」ということになる（図 2.3 の右側）。検証方法として、1 回目の仮説検証ループでも使用した printf デバッグが利用可能である。しかし、不幸にして関数 f() の return 文が複雑な操作を行っているような場合には、printf の出力を見ても仮説が検証できない可能性がある。その場合は printf デバッグによる仮説検証は断念し、デバッガを用いた仮説検証方法を採用することになろう。このとき、第 2 段階で設定した仮説には何らの変更は生じていないが、検証方法の変更によって第 3 段階と第 4 段階の間で小さいループが起こったと考えられる。

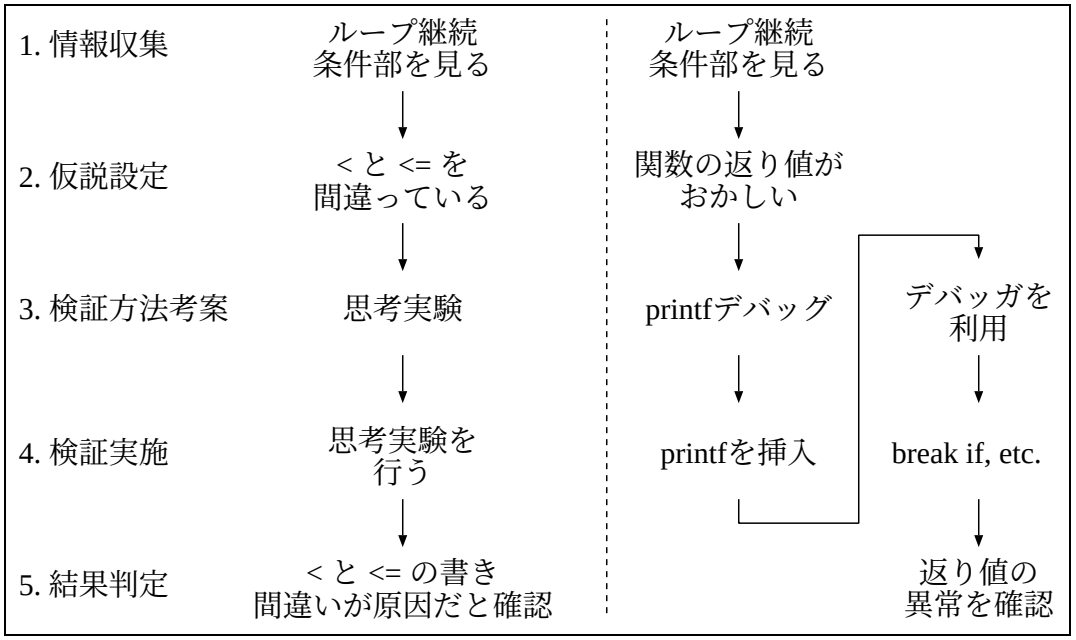

図 2.3: ケース 1 の仮説検証ループ（2 回目）

考察　ケース 1 を通じて以下の 4 点を述べた。第 1 は、デバッグの基本である仮説検証ループの繰り返しについて。第 2 は、1 つの仮説に対して複数の検証方法が考えられること。第 3 は、プログラマはそれらの中から 1 つを選んでいること。第 4 は、選んだ検証方法が失敗した場合には別の検証方法を選択して、仮説の検証を継続できること。

ここでは仮説検証ループの例示のため、かなり細かく（まわりくどく）説明した。実際にプログラマが行う手順では、仮説検証ループの 1 回目と 2 回目は明確には分かれていないかもしれない。また、誤りが自明な場合には仮説の検証を明確に行わないこともありうる。しかし、その思考プロセスを詳細に分解すれば、ここに示したような手続きを行っている。経験豊富なプログラマは自身の経験によって、途中を省略し、より効率良くデバッグ作業を進めることができる。

2.3.2 ケース 2：配列のオーバラン

エラーの症状　デバッギは C で書かれており、最終的に 2 桁の自然数を出力するプログラムとなるはずであった。しかし、デバッギは信じられないほど大きな数を出力した。

仮説検証ループ　1 回目の仮説検証ループでは、計算のアルゴリズムが間違っているという仮説に基づいて思考実験による検証を行ったが、誤りは発見できなかった。

次に、アルゴリズムをコードに移すに際して間違ったのではないかと疑った。そこ

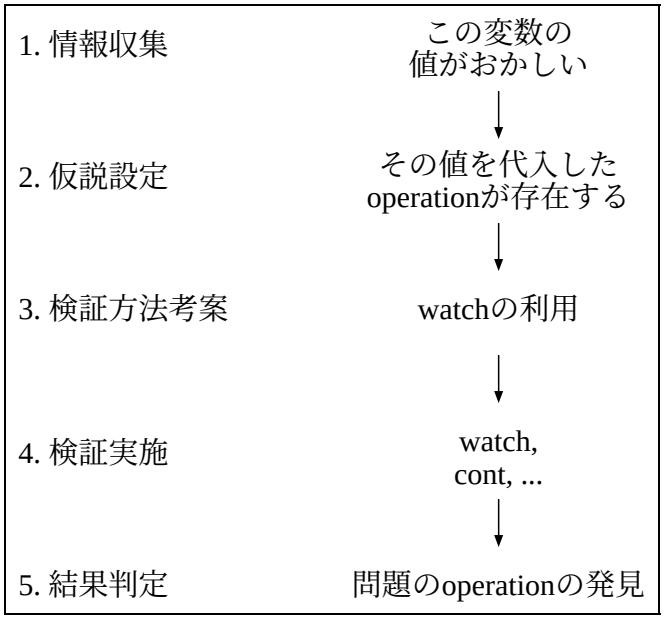

図 2.4: ケース 2 の仮説検証ループ（3 回目：1〜2 回目は省略）

で 2 回目の仮説検証ループでは、変数へ代入している箇所での右辺値がおかしい、という仮説を設定して `printf` デバッグを試みたが、正常な値しか見つけられなかった。

そこで、3 回目の仮説検証ループでは、他のオペレーションが意図しない代入を行っており、その結果として変数の値が上書きされるという仮説を設定した（図 2.4）。検証方法として、デバッガのウォッチポイント（GDB コマンドは `watch`）を利用する方法を考案し、デバッガのコマンドを駆使して異常な値になる瞬間を見つけることができた。デバッギが文字列のために用意したバッファサイズを超える、長い文字列を書き込んでしまったため、そのバッファの隣にあった変数が不幸にも上書きされてしまったことが分かった。バッファサイズを超える場合の例外処理を記述して動作を確認し、デバッグを完了した。

考察 ケース 2 では、C プログラミングで頻出のバグを取り上げた。このような「意図しない代入」は、C のような実行時のチェックが不十分な言語以外でも起こる。

例えば、Java でマルチスレッドプログラムを作成したとする。複数のスレッドから代入される変数は、スレッド切替のタイミングによっては、想定していない順序で代入されて正しくない値になってしまうことがある。あるいは、大規模なプログラムでは代入のコードが他のコードに埋もれてしまい、そこで誤った値が代入されていることに気づかないこともありうる。

これらのいずれの場合でも、図 2.4 に示した仮説検証ループは有効である。すなわち、異なるデバッグ局面でも仮説検証ループは再利用可能である。

```
    public void actionPerformed(ActionEvent ae){
        :
(1)   connection.send(field.getText());
(2)   field.setEditable(false);
        :
    }

    public void promptAgain(){
        :
(3)   field.setEditable(true);
        :
    }
```

図 2.5: ケース 3 のデバッギのコード（一部）

2.3.3 ケース 3：マルチスレッドプログラム

エラーの症状 デバッギは Java で書かれており、サーバと通信を行う、GUI を備えたクライアントプログラムである。そのコードの一部を図 2.5 に簡略化して示す。ユーザから GUI へのテキスト入力を受けると入力データをサーバに送信する（図中の (1)）と同時に、その入力フィールドを一旦、編集禁止に設定する（図中の (2)）。その後、サーバから再入力を指示する応答が送られてきたため、入力フィールドを再び編集可能に設定（図中の (3)）してユーザに再入力を促す。

サーバが再入力を指示したとき、このデバッギは 10 回に 1 回程度は正しく動作したが、残りの 9 回は入力フィールドが編集禁止のままになってしまう、というエラー症状を発現した。

仮説検証ループ 当初はコードに本質的な誤りがあるとは考えず、Java の Window Toolkit である Swing の使用方法の誤りだと思い込んでしまった。

そこで、1 回目の仮説検証ループでは、Swing 使用方法の誤りによる、という仮説を設定した。リファレンスマニュアルを確認するという検証方法を実施したが、使用法に誤りは認められなかった。次に、デバッギのコードをマニュアルのサンプルコードに近付けるよう変更する、という検証方法を行ったが、これも成功しなかった。そこで、デバッギのコードは元に戻し、2 箇所の `setEditable()` の直後でその結果を表示するという、`printf` デバッグによる検証方法を実施したが、`setEditable()` は正常に機能していた。以上より、使用方法を誤ったという仮説は棄却し、新たな仮説を模索することにした。

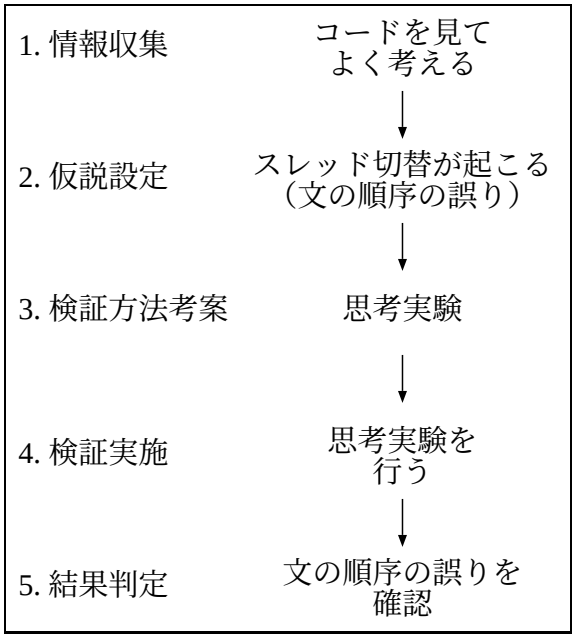

図 2.6: ケース 3 の仮説検証ループ（3 回目：1〜2 回目は省略）

2 回目の仮説検証ループでは、図 2.5 の (2) と (3) 以外にも `setEditable()` している箇所がある、という仮説を立てた。ソースコード 全体に対して `grep`（テキスト検索）を行う、という検証方法を実施したが、上記以外の箇所は見つけられなかった。よって第 2 の仮説も棄却した。

3 回目の仮説検証ループでは、直ちに仮説を立てることができなかったので、第 1 段階の情報収集として、(2) と (3) を中心にデバッギのコードを見てよく考えることとした。正常に動作する場合もある、ということから、スレッド切替のタイミングに依存したバグの可能性が高まった。デバッギは GUI を持ち、サーバとの通信には専用のスレッドが割り当てられていることを思い出し、(1) と (2) の間でスレッド切替が発生しているのではないかと考えるに至った。そこで、(1) と (2) の順序が逆である、という仮説を設定した（図 2.6）。思考実験による検証を実施して合理的に説明されることを確かめ、仮説を立証した。コードを書き換えて正常に動作することを確認し、デバッグを完了した[1]。

考察 ケース 3 に関連して、仮説検証ループにおける 2 つの重要な知見を示す。

1. 仮説検証ループの繰り返しに適切なループの実行を追加することによって、仮説の設定が容易になる。この追加されるループを補助ループと呼ぶことにする。

[1] このバグの場合、1 回目の仮説検証ループにおいて 2 箇所の `setEditable()` の直後で `printf` デバッグを行った際に `true`、`false` の順序で表示されれば、バグに気づくであろう。しかしながら、`setEditable()` と `printf` の間でスレッド切替が発生する可能性があるので、結果的にヒントを与えることにはなっても、厳密な検証とはならない。

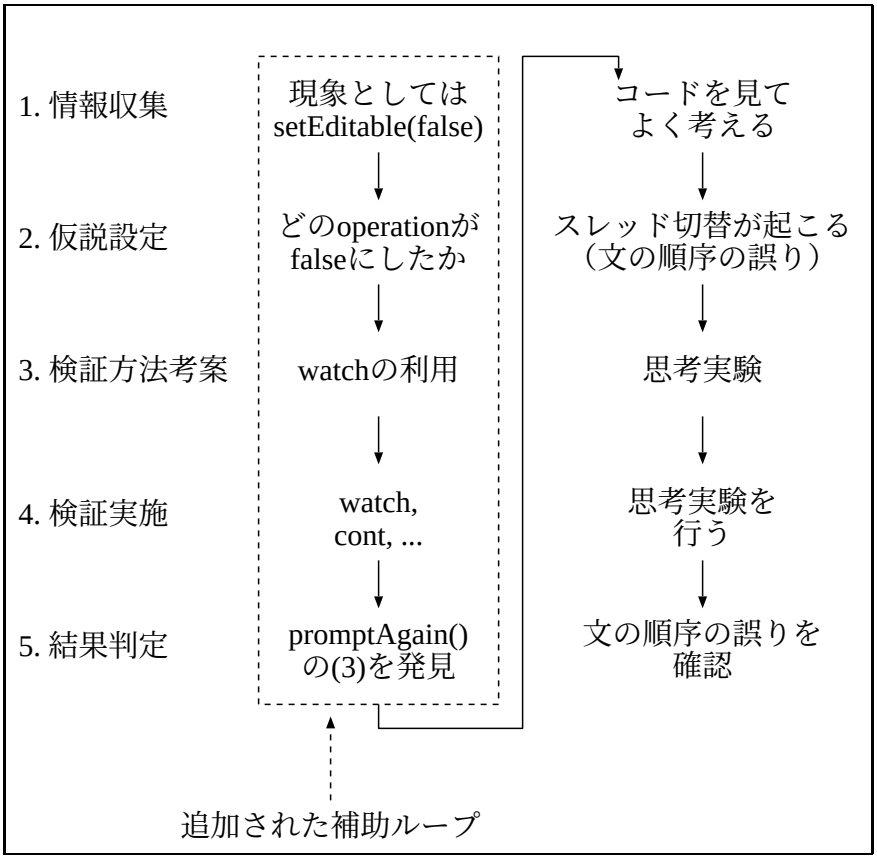

図 2.7: ケース 3 のための補助ループ

そもそも、スレッド切替のタイミングに依存するバグはエラー発現の条件を固定することができないため、一般にデバッグは困難である。多くの場合は、ここで述べた 2 回目の仮説検証ループなどを行ってエラーに関連するコード箇所をある程度絞った上で、コードを見て考えることになる。遠藤 [End03] も指摘するように、仮説検証ループの中では第 2 段階の仮説設定が非常な難関である。従って第 1 段階での情報収集が重要となるが、スレッド切替に関する情報は一般には得られないため、仮説設定はなお一層困難になる。

そこで、DejaVu[CS98] のような、スレッド切替を記録して再生する Java VM を利用すれば、前述の 3 回目の仮説検証ループの前に図 2.7 のような補助ループを実施でき、「スレッド切替が起こる」という仮説設定が比較的容易になる。`watch` によって、`setEditable(true)` が行われたあとで `setEditable(false)` が行われることが確認されるからである[2]。なお、この戦略は前述のケース 2 と同じであり、仮説検証ループの再利用に相当している。

[2] この例では `setEditable()` の利用箇所が 2 箇所しかないので、`watch` の代わりに `break` を用いる検証方法も有効である。その場合は 2 箇所の `setEditable()` 両方ともに `break` を設定し、どちらのブレークポイントが先にヒットするかを調べれば良い。

2. プログラマは仮説検証ループの検証方法に要するコストを見積もって、仮説検証ループを選択するということである。

   この watch を用いて問題の代入箇所を発見するという戦略には問題がある。ケース3では setEditable() が2回しか実行されないため、watch で停止する回数も2回だけである。しかし一般には、watch の停止回数は多い。作業そのものは単純であるが、それゆえに人間が行うにはややコストが高いものである。このため、スレッド切替の記録、再生機構があったとしてもプログラマが高コストを嫌って watch の戦略を避けることもあろう。その場合には、直ちに「コードを見て考える」という戦略を採ることになるが、仮説を設定できなければ watch の戦略に頼らざるを得ず、結果として時間を浪費することになる。

2.3.4 まとめ

3つのケーススタディでの考察をまとめると、以下のようになる。

- デバッグ作業は仮説検証ループを繰り返すものと定義できる。
- 1つの仮説に対して複数の検証方法が存在する。プログラマは自身の能力に照らして、総合的なコストが最も低い方法を選択する。選択の結果、検証に失敗した場合は別の検証方法に切替えて仮説の検証を続行する。
- 1つの仮説検証ループは、異なるデバッグ局面であっても共通して利用可能である（再利用可能である）。
- 仮説の設定を容易にするため、補助的な仮説検証ループを追加することができる。補助ループを用いるかどうかは2つの点に依存する。1つは、プログラマがその補助ループに着眼できるか否か。いま1つは、その補助ループの検証方法に要するコストが高いか否か。検証方法の選択と同様、プログラマは自身の能力に応じて補助ループの採用を決定する。

2.4 デバッグパターン

仮説検証ループにおいて、第2段階の仮説設定と第3〜4段階の仮説検証が最も重要な作業であり、デバッグの本質であると言える。そのうち、第2段階ではプログラマの知識と経験、発想力が不可欠であり、計算機による支援は難しい。これに対して、第4段階には（検証方法が思考実験である場合を除けば）支援の余地がある。特にデバッガを用いた検証方法を選択した場合には、2点の余地が挙げられる。1つは、デバッガの実行制御がデバッグの本質とかけ離れた作業であること。もう1つは、その作業の意味に明らかなパターンが存在することである。

2.4.1 実行制御の困難

デバッガは強力なツールであるが、その機能はOSやハードウェアのサポートを直接的に実現したもので、デバッグ中にプログラマが行いたいと考える高水準な作業内容との間には隔たりがある。従って、プログラマはデバッガコマンドをどう組み合わせて希望の検証を実施するかをも考えなければならない。これはデバッグの本質とはかけ離れており、既存のデバッガの制限によるものである。

デバッガを利用した検証作業においては、特に実行制御に要するコストが高い。デバッガによる実行制御は第2.2.2節で述べた3種類のデバッガコマンドの組み合わせとなる。ステップ実行はデバッガで制御可能な最小単位の実行を実現するので、デバッギの実行は着実に1行ずつ進行し、コマンドの組み合わせは必要ないと考えることもできる。しかしステップ実行の繰り返しは非常に低速であり、また、デバッギの中で注目する必要がないと分かっている部分にまでも注意を割く事になる。ステップ実行だけで実行制御を組み立てることは現実的ではなく、ブレークポイント（前述の `break`）やウォッチポイント（`watch`）のような、特定のイベント（行への到達や変数への代入）で駆動する機構と組み合わせることになる。プログラマは要所で変数の値を表示し、その結果によって今後の実行制御の方針を決め、コマンドを組み立てる。

この組み立てはプログラマが常に直接指示しなくてはならない。コマンドを間違った場合にはデバッギを先頭から再実行して同じ作業を繰り返す必要がある。また、検証に失敗して別の検証方法を試みる場合や、第2段階で設定した仮説が棄却された場合には、新たな検証実施のために制御を再び別の点へ移動させることになる。これは特に苦痛を伴う作業であり、実行制御はデバッグに要する時間の多くを占め、プログラマの拘束時間も長い。

さらに、そのときに行ったデバッガ操作はコマンドの羅列であり、その場限りのもので、再利用ができない。しかしその羅列の中には、デバッガを用いてどのような作業を行ったかという、コマンド列よりも抽象化された明確な意味が存在する。

2.4.2 パターンの存在

仮説検証ループに意味のパターンがありループ全体を再利用可能であったように、その一部である検証実施段階（第4段階）もまた意味のパターンを持ち、再利用可能であることをプログラマは経験的に知っている。例えば、以下のようなものが挙げられる。

- コマンドを間違って目標とする実行状態をうっかり通り過ぎてしまったので、以前の状態まで戻りたい。
- ある変数が予期しない値になっているときに、その値を代入したオペレーションを知りたい（ケース2で述べた、デバッガの `watch` を用いる検証方法）。

- 何が原因か見当がつかないときに、プログラムがどの段階まで正しく動いていたかを知りたい。

こうした典型的な作業は現在は完全に手動で行われているが、近年の計算機の強力な処理能力を利用して自動化することができれば、プログラマはデバッガを制御する苦痛から解放されて、仮説の設定と検証というデバッグ作業の本質に集中できるようになる。

2.4.3 デバッグパターンの定義

以上より、仮説検証ループの第3段階（検証方法の選択）と第4段階（検証の実施）に内在する本質的な手続きをデバッグパターンと呼ぶことにする。具体的には第2.4.2節で示した3つの例がこれにあたる。本論文ではデバッグパターンを自動化することによって、デバッグ作業中のプログラマを支援する方法について述べる。

デバッグパターンの自動化により、直接的にはデバッグ作業に要する時間の減少という利点がある。それだけでなく、仮説の検証における労力が低減されるため、プログラマにとっては仮説設定の心理的バリヤーが低下することになる。これにより、様々な仮説を積極的に検証することを可能にし、確実なデバッグを促進する。

デバッグパターンは、オブジェクト指向プログラミングの設計におけるデザインパターン [GHJV95] と対比することができる。デザインパターンは典型的なオブジェクトの階層構造をパターンとしてまとめることで、設計のノウハウを再利用することを可能にした。デバッグパターンも同様に、デバッグ作業における本質的な手続きをパターンとしてまとめ、デバッグのノウハウを再利用できるようにする。

2.5 パターンとしての関連研究

関連研究の中には、デバッグパターンの1つとして位置付けられるものがある。

2.5.1 逆実行

実行を遡りたいというのは自然な欲求であり、逆実行には長い研究の歴史がある。これらは状態保存による逆実行と、再実行を用いた疑似逆実行とに大別できる。いずれの方式にせよ、デバッガを用いた検証方法を選択した場合には、逆実行はプログラマの手作業を自動化する有効なパターンと言える。

```
fget(func=legal and
            ((type=for and port=enter) or
             (type=if  and port=exit))),
  ( if   current(type=for)
    then display_board,
         current_var(lineNb, val=Lin),
         current_var(colNb, val=Col)
    else current_var(i, val=I),
         current_var(board(I), val=BdI),
         current_var(ok, val=Ok)).
```

図 2.8: Coca[Duc99] における実行制御コマンド

2.5.2 イベント駆動型実行制御

GDB や JDB のような一般的なソースレベルデバッガは、実行制御をソースコードのステートメントを単位として行う。これに対して、代入や関数呼び出し、ループなどをイベントとして扱い、イベントを単位として実行制御を行う方法もある。

Ducassé[Duc99] は、条件付ブレークポイントの条件をプログラマが Prolog に似た書式で記述できる仕組みを提案している。例えば、図 2.8 のようなコマンドをデバッガに与えると、「関数 `legal` の中で `for` 文に入るか `if` 文から出るようなときにだけ実行を停止せよ。`for` のときであれば盤面を表示し、変数 `lineNb` と `colNb` も表示せよ。それ以外のときは変数 `i` と `board(I)` と `ok` を表示せよ。」という指示をしたことになる。

この仕組みも新たなデバッグパターンの 1 つとして位置付けられる。従来のソースコード型実行制御よりもイベント駆動型実行制御の方が仮説の検証が行いやすい、という場合はこのパターンを用いてより容易に検証を実施できるからである。

2.5.3 Delta Debugging

Zeller ら [HZ00, Zel02] は Delta Debugging を提唱している。あるデバッギが正常終了するときと異常終了するときの入力データを 2 分探索的に比較して、入力データの中から異常の原因となったデータを自動的に抽出するという手法である。また、入力や引数の違いによって生じるデバッギ内の変数の値を比較して、その伝播の関係を抽出することも行っている。これは非常に有用な手法であり、現代の計算機の強力な計算能力によってプログラマの労力を軽減するという観点で、本研究と近しいものと言える。

この手法もまた、パターンの1つとして位置付けることができる。このようなパターンの自動化が提供されることで、これまでは検証方法が高コストであるために敬遠されてきた仮説も検証されやすくなる。この仮説検証ループを前述の補助ループのように用いれば、後続の仮説の設定の敷居を下げる効果も期待できる。

第3章　関数単位疑似逆実行

デバッグ作業における典型的なオペレーションとして代表的なものに逆実行がある。通常のデバッガはデバッギをその実行順どおりにしか動かすことができないので、現在位置の直前の情報を得るためにさえ、プログラマはデバッギの実行を停止させて先頭から再実行させなくてはならない。逆実行とは、デバッガやインタプリタのサポートによって実行状態を過去の状態に戻す機能を言う。

本章では逆実行の技術の中から再実行による疑似逆実行を紹介し、プログラマにとって有用な自動化である関数単位疑似逆実行とその高速化について述べる。一般的な逆実行の手法と本研究との比較については第 8.2 節で述べる。

3.1　疑似逆実行

プログラムの実行は変数への代入、すなわちメモリの書き換えの繰り返しで進行するので、逆実行のためにはメモリ内容の変遷を記録しなくてはならない。しかしデバッギに再現性があれば、再実行して直前で停止することで疑似的に逆実行を実現することができる。デバッグ作業中のプログラマはこのプロセスを手動で行っており、疑似逆実行はこれを自動化していると考えられる。例えば行単位の疑似逆実行を行う場合、再実行の際に停止すべき行は現在行の直前ということになる（図 3.1 中の target）。

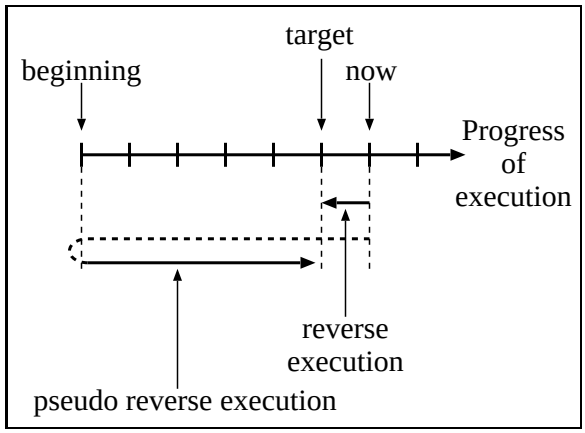

図 3.1: 疑似逆実行

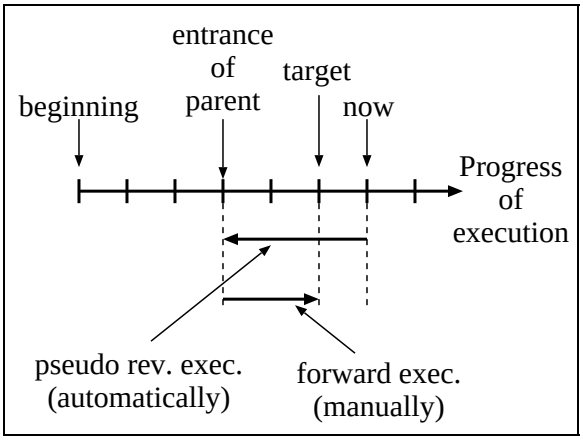

図 3.2: 関数単位疑似逆実行

より粒度の低い単位として関数単位の疑似逆実行がある [Ter00]。これは逆実行の単位を関数呼び出しに定め、再実行の際に停止すべき行を現在実行中の関数の入口とするものである。この方式では現在の関数の入口までは自動的に戻ることができるので、プログラマはそこから順方向に実行を進めて目的の行まで移動することになる（図 3.2）。デバッガには `finish` というコマンドがあり、現在の関数を終了するところまで実行を進めて親関数（現在実行中の関数を呼び出した関数）のフレームへ戻ることができる。関数単位疑似逆実行はこの `finish` を逆向きにしたものと考えることができ、人間が行うデバッグ作業において自然な場所と言え、十分にプログラマの労力を軽減する機能である。

3.2　従来手法

本節では、寺田 [Ter00] が実装した C での関数単位疑似逆実行の手法（以下、従来手法と呼ぶ）とその問題点について述べる。

従来手法では、再実行の際の移動先となる現在の関数の入口を識別するため、特別なカウンタ（以下、タイムスタンプと呼ぶ）を用意して関数呼び出しが起こった際にタイムスタンプをインクリメントする。この機構をプロファイル用関数[1]の再定義で実現により、容易かつポータブルに実装することに成功している。

図 3.3 に例を示す。現在、関数 `me` の now の部分を実行中とすると、その入口は図の通りである。図中の網かけの丸印はタイムスタンプ更新場所であり、左の数字がそのときのタイムスタンプ値を示す。この例では自分の呼び出しは 125 である。

125 という値が分かればデバッガを用いてそこまで移動することができるが、移動目標のタイムスタンプ値（ここでは 125）をどのようにして得るか、ということが

[1]プログラムの性能測定のために、関数呼び出しの入口で呼び出される関数。この呼び出しはコンパイラによって自動的に挿入される。GCC[Sta99] の場合は `-pg` オプションを指定することで、`mcount` という関数の呼び出しが挿入される。

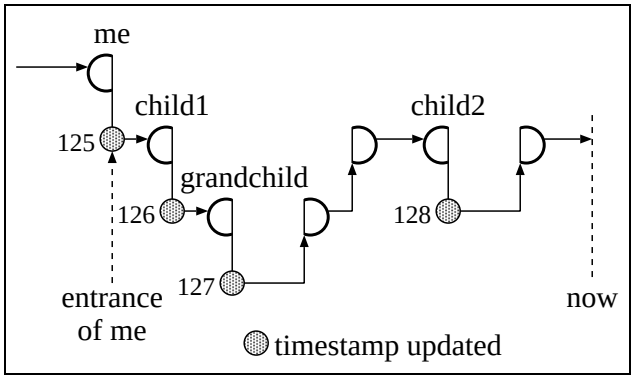

図 3.3: 現在の関数の入口

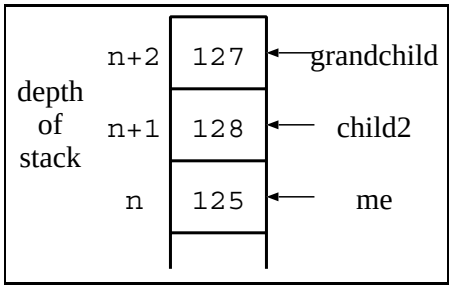

図 3.4: シミュレートするスタック

問題である。現在点でのタイムスタンプ値は child2 の入口で更新した値であり、目標点から現在点までの間に何回更新されたか、すなわち何回の関数呼び出しがあったかは一般には知ることができない。

関数呼び出しには必ず 1 つのタイムスタンプ値が対応するので、関数呼び出しのスタックをシミュレートして呼び出し深さごとの最新タイムスタンプを保持すれば良い。関数 me の呼び出しが深さ n とすると、シミュレートするスタックの深さ n に相当する場所にはタイムスタンプ値として 125が保持される（図3.4)。深さ $n+1$ には child2 のタイムスタンプ値である 128 が、深さ $n+2$ には grandchild が呼び出されたときの 127が残っている。me の呼び出し時点でのタイムスタンプ値を知りたいときには、自分の呼び出し深さである n を見ると、目標とする関数呼び出しのタイムスタンプ値が手に入る。

しかし、関数呼び出しの出口をトラップできないことから現在点の呼び出し深さが正確に分からないため、正しい深さを取得するための scan pass を導入することで解決している（図3.5)。すなわち、現在の関数の入口に戻るためには scan pass と re-exec. pass の 2 パスが必要になる。

scan pass では、デバッガの持つスタック検査コマンド（GDB では where）を用いてスタックの深さを正確に知る。従って、scan pass では通常のデバッギの実行時間に加えて、デバッガによるスタック検査の時間がかかってしまう。

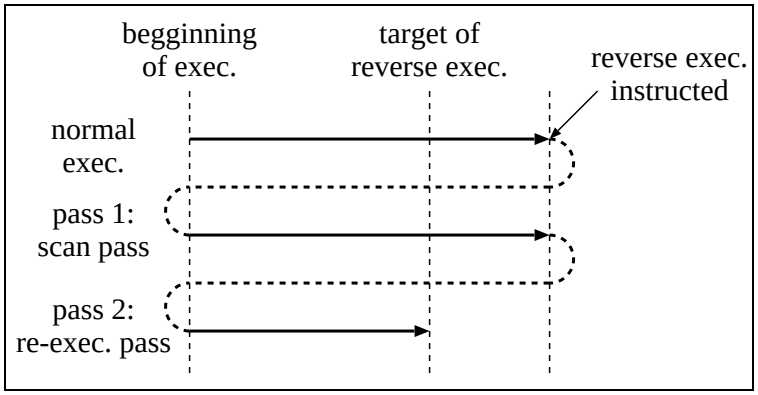

図 3.5: 従来手法 [Ter00]

3.3　新手法

従来手法はポータブルで、処理系を変更しないで済む実装方法を取った代償として、2パスを要するというパフォーマンス上の問題点を持っていた。本節では、移植性を失う代わりに、処理系を変更しないままで 2パス問題を解決する新手法 [MT00b] について述べる。

3.3.1　フック関数呼び出しの挿入

2パスの問題は関数呼び出しの出口を捕捉して、図 3.5 の通常実行中においても呼び出し深さを正確に保持することができれば解決する。そこで、関数呼び出しの出口（関数定義の末尾）にフック関数 `mcount2` を呼び出すコードを追加する。関数呼び出しの入口部分については、従来手法同様にプロファイル用関数の機構を流用し、フック関数 `mcount1` の呼び出しを挿入する。これらの挿入は、実装の容易さを考慮してアセンブリコードレベルで実装した。フック関数の定義を図 3.6 に示す。

この実装では関数呼び出しの出口の処理が `return` 文の場合も兼ねているので、返り値を上書きしてしまわないよう、`mcount2` が使用するレジスタを全て保存した上で `mcount2` を呼び出し、再びレジスタの内容を復元する、という手順で行っている（図 3.7）。以上により、1パスで現在の関数の入口へ戻ることに成功した。

3.3.2　ユーザインタフェース

ユーザが利用する手段として、GDB のフロントエンドである `mygdb`[Ter99] に機能付加する形で実装した。利用例を図 3.8 に示す。図の状況は、関数 `read_pattern_space` 内をデバッガで実行しているところであり、下 2/3 程度に実行中のソースコードが表示されている。図中の下部で `now` というコメントで示してある網かけの部分が、デバッガがこれから実行しようとする行を示している。ここで関数単位疑似逆実行

```
#define MAX_DEPTH 256
int stack_ts[MAX_DEPTH];
int depth = -1;
int timestamp = -1;
int ref = -1;

void mcount1(void){
  depth++;
  timestamp++;
  stack_ts[depth] = timestamp;
}

void mcount2(void){
  depth--;
}
```

図 3.6: フック関数の実装

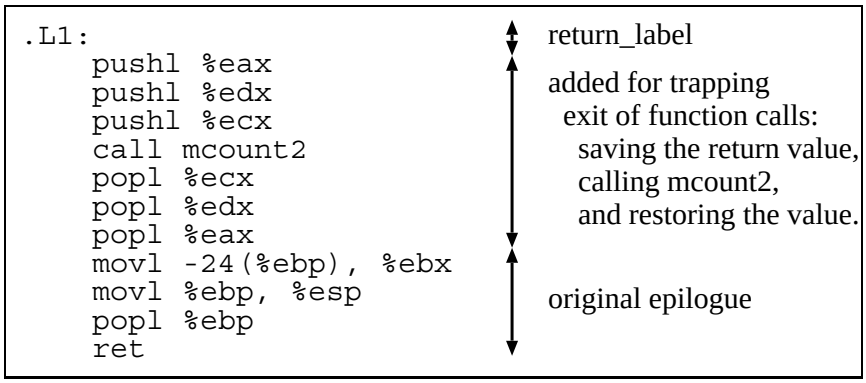

図 3.7: 関数呼び出しの出口のトラップ

を指示すると、コメント `target point` で示した場所、すなわち現在実行中の関数である `read_pattern_space` の先頭に制御が移動する。

実際にデバッガの一機能として利用される場合、ユーザが設定したブレークポイントなどにより、単純に再実行を行ってもデバッギが期待通りの場所で停止するとは限らない。また、頻繁にブレークポイントにヒットしてしまうと再実行の際の実行速度が極端に低下する。

そこで、再実行を行う前にユーザが設定したブレークポイントを一旦全て無効

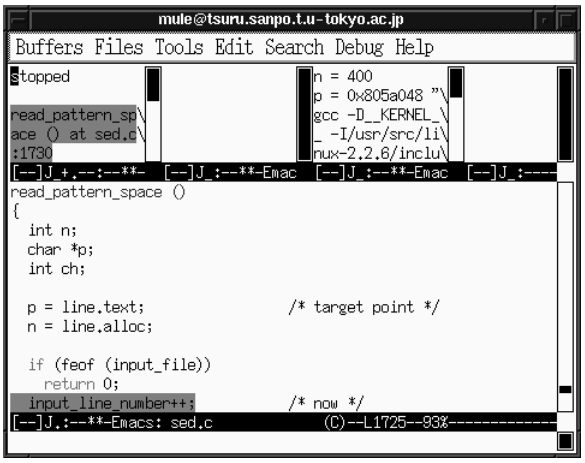

図 3.8: mygdb での利用例

（disable）にし、逆実行が終了した後に再び有効（enable）にする、という方法を採っている。

3.3.3　処理時間計測

新手法の処理時間を、以下の条件で計測した。デバッギとして sed-1.18 を用い、10816 行（1379312 バイト）のテキストファイル[2]に対して s/make/MAKE/g を行って /dev/null へ出力した場合について、126471 回目の関数呼び出しで停止させ、その呼び出しの入口への逆実行に要する時間を計測した[3]。計測に用いた計算機は i486DX2 50MHz、メモリ 24MB、Linux-2.0.35 という構成で、時間の計測には GNU time-1.7 での elapsed time を用いた。以下の 2 種類の処理時間を、従来手法と新手法について調べた。

通常実行 -g オプションと-pg オプションでコンパイルしたデバッギを通常実行[4]。この場合だけ、特に参考のために、-O3 オプションでコンパイルしたデバッギの実行時間も計測した。

directrun 上記のデバッギを、逆実行を実装した Perl スクリプト（以下、ドライバプログラムと呼ぶ）で 1 パスだけ実行。このスクリプトは GDB を起動し、これを介してデバッギを制御するので、この時間には Perl と GDB の起動時間も含まれる。

[2]Linux カーネルを make したときのログファイルを連結したもの。
[3]逆実行の開始点と目標点が同一となるが、処理のオーバヘッドを知るには十分である。
[4]プロファイル用関数の仕組みを流用するので、-pg が必要になる。新手法用のデバッギは、これらオプションを指定して得られたアセンブリコードをさらに変換して、実行コードを得る。

表 3.1: 新手法と従来手法による sed の処理時間 [MT00b]

	old method	new method	optimized
normal exec.	4.816	4.854	3.088
directrun	14.126	14.114	—
rev-myself	**28.574**	**21.136**	—
without scan pass	21.194	—	—

単位は秒

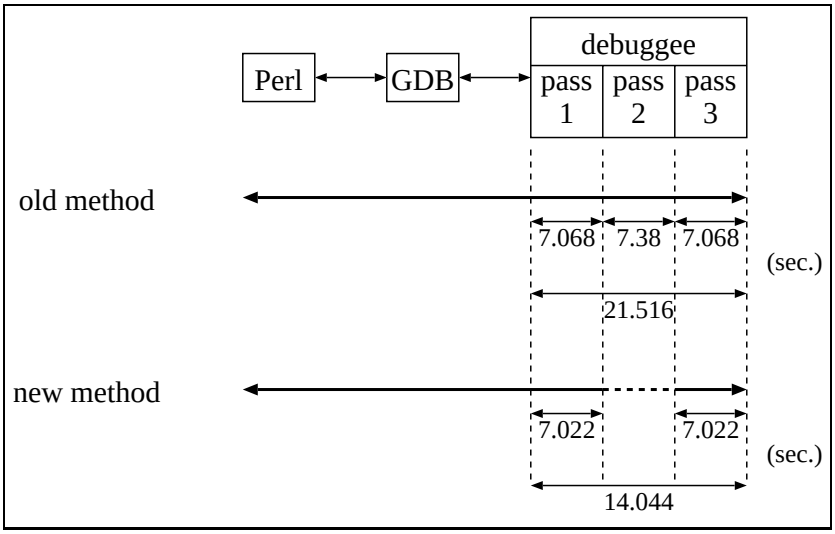

図 3.9: 各手法の処理時間の内訳

rev-myself 上記のデバッギで、ドライバプログラムによって自分自身の関数呼び出しの入口へ逆実行した場合の実行時間。

without scan pass 従来手法に関してのみ、scan pass を行わなかった場合の処理時間も計測した[5]。

計測の結果を表 3.1 に示す。関数呼び出しの出口を捕捉するようになったことで、通常実行時のデバッギの実行速度は従来手法よりも低下しているとは言え、微々たるものである。

一方、表 3.1 より得られた実行時間の内訳は図 3.9 のようである。Perl と GDB の起動に要した時間を除いた実行時間はおよそ 0.65 倍になっており、逆実行速度は 3 パスから 2 パスになった分、減少し、新手法の効果を示している。

前述の通り、関数単位疑似逆実行はデバッグ作業中のプログラマがよく行う作業のひとつを代行するものと言える。デバッグ作業では他にも実行制御のパターンが

[5] この場合に限っては最初から目標点のタイムスタンプが分かっているので、scan pass を不要にしても目標点に制御を移動できる。

存在するが、関数呼び出しのみでタイムスタンプを更新する手法では実行制御の移動先が限定されてしまう。実行制御の移動先として任意の場所を指示するためにはより一般的な表現が必要となる。

第4章　プログラム実行点

一般的な実行制御を自動化するために、プログラム実行点という概念を用いる。これはプログラムの実行状態をその進捗度として表すものである。本章では実行点の必要性とその表現方式について述べ、単純な応用である動的ブレークポイントを導入する。

4.1　実行点の概念

デバッグを行っているとき、特にデバッガを用いているときに、多くのプログラマはデバッギの進捗度（デバッギの実行がどこまで進んだか）を漠然とイメージしている。それは例えば、「この関数の呼び出しから戻ってきた、この場所」といったようにソースコードに直接対応する形のこともあれば、「木構造のトラバースで、このノードに到達したとき」などと、データ構造に対応する形のこともある。このようなデバッギの進捗度は、プログラムの実行過程における1点と考えられることから、これをプログラム実行点と呼ぶことにする。

ここで、手続き型言語のプログラムが実行される場合について考えると、ソースコードの各行を制御構造に従って順に実行していくので、プログラムの実行の完全なトレースは実行された行の時系列で表すことができる。これをプログラムの実行系列と呼ぶ。プログラムが入力データなどによって振舞を変える場合は、同一のプログラムであっても実行系列は実行ごとに異なる。一般には、プログラムを実行するごとに特定の実行系列が生まれる。プログラム実行点は、このような特定の実行系列における絶対座標を表すものと考えることができる。

4.2　実行点の表現形式

概念としてのプログラム実行点を具体的に表現する形式を考えると、以下の4通りの表現が考えられる。

ステップ実行の回数 ステップ実行はデバッガで制御可能な最小単位の実行を実現する。従ってこれを繰り返せば完全な実行系列を得ることができる。

デバッガコマンド履歴 デバッガコマンドの履歴を再生すれば、行った実行制御を再現して同一の実行点に移動できる。

```
test.c

        :
(1) while(i < a){
(2)   i += b;
(3) }
        :
```

図 4.1: ループ構造を持つデバッギ

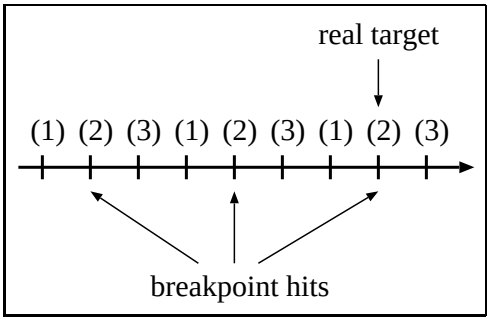

図 4.2: ループ構造を持つデバッギの実行系列

行番号とブレークポイントのヒット回数の組 デバッガはブレークポイントのヒット回数をカウントしているので、行番号とこの回数の組。

行番号とタイムスタンプの組 制御点が上向きにジャンプするときにインクリメントされるカウンタ（タイムスタンプ）と、行番号の組。

最初の 3 つには、実行効率などの問題があるので採用しない。詳細は第 8.1 節で述べることとし、以下では第 4 の選択肢に関する説明を行う。

実行点は実行系列における座標であるという考えに基づけば、ソースコードの行番号（正確には、ファイル名と行番号の組）が利用できるように思われる。実際、デバッガにおけるブレークポイントは行番号を利用する。しかし、一般にブレークポイントはデバッギの実行中に何度もヒットする。このことから、行番号は実行系列における座標としては不十分である。

図 4.1 のようなループ構造を持つデバッギ test.c を例に採って考える。このループの本体が 3 回実行されるとすると、(2) で示した行 test.c:2 は実行系列中に 3 回登場することになる（図 4.2）。

ファイル名と行番号が静的な情報に基づいているために、実行系列中の複数の test.c:2 を区別することができない。このような (file:line) で表される位置を静的な

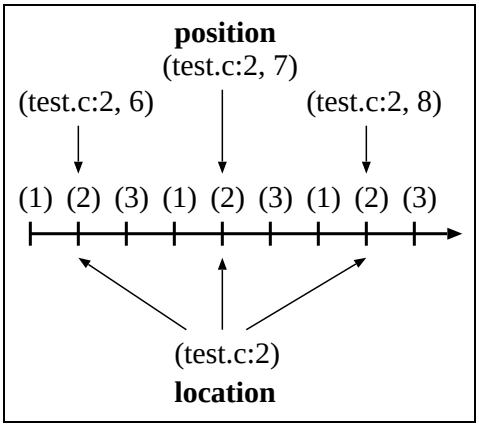

図 4.3: location と position

位置と考え、**location** と呼ぶ。これは静的に unique な情報で、ソースコード中の一点を表すことができる。

これら複数の test.c を区別するため、location に動的な情報を付加して動的な位置を表現する。実行系列中に同じ location が登場する際に必ず更新されるカウンタを用いれば、location とカウンタの組は動的に unique となり、実行系列中の 1 点を識別できるようになる。第 3 章での命名にならい、このカウンタをタイムスタンプと呼ぶ。そして、この動的な位置こそが実行点の概念を表したものである。**location** に対して **position** と呼ぶことにし、(file:line, timestamp) のように表すこととする。この表現方法は Mellor-Crummey らも Software Instruction Counter[MCL89] を用いた表現方法として挙げているが、彼らの実装は実行点の表現としては不十分である。これについては第 8 章で述べる。

図 4.1 の例で、実行制御の移動目標となる position での `timestamp` が 8 であったとすると、その position は (test.c:2, 8) と表せる。一方、それ以前に登場する 2 回の (test.c:2) はそれぞれ (test.c:2, 6) と (test.c:2, 7) となる（図 4.3）。

なお、ここではタイムスタンプの値が既知であることを前提にしている。position はデバッグ作業中に利用するため、タイムスタンプの値を含む position の情報は cyclic debugging における以前の実行で取得済みでなければならない。また、現時点ではデバッギは再現性を持つという前提である。再現性を持たないデバッギは第 7 章で扱う。

タイムスタンプの値は、プログラマがその値を推定して直接指定できるような性質のものではない。例えば「あのループの 325 回目に興味がある」と思っても、そのときのタイムスタンプ値を推定することは事実上不可能であり、デバッガを用いてその position まで到達して変数 `timestamp` の値を取得するしか方法はない。タイムスタンプ値は、第 6 章で述べるような cyclic debugging を基礎とするアプリケーションから利用することを想定しており、プログラマが直接利用することは想定されていない。

position を他の形式で表現することもできる。例えば、実行制御には常にステップ実行を用いると決めると、position はステップ実行の回数として表される。また、デバッガのコマンド履歴を完全に再現すれば同一の実行状態に到達できるので、これも position の表現たりうる。あるいは、実行を開始する前にソースコードの全ての行にブレークポイントを設定しておけば、該当行に設定したブレークポイントのヒット回数を position として用いることもできる。これらの表現形式と、本研究で採用したタイムスタンプを用いた形式との比較は第 8 章で行う。

4.3　動的ブレークポイント

本節では position に対して設定できるブレークポイントを考え、これを動的ブレークポイントと呼ぶ。一方、通常のブレークポイントは location に対して設定するので、これを静的ブレークポイントと呼ぶ。

動的ブレークポイントは GDB の条件付ブレークポイントを用いれば簡単に実現できる。(test.c:2, 8) に対して設定する場合は以下のような GDB コマンドを用いればよい。

```
break test.c:2 if timestamp = 8
```

しかし条件付ブレークポイントの実装は、ブレークポイントにヒットするたびに条件をデバッガが評価して成否を決定するため、一般には多くのプロセススイッチが発生して実行速度が犠牲になる。効率の良い実装については、第 5 章で C と Java への実装の詳細とともに述べる。

動的ブレークポイントは position の単純な応用であり、第 2.4.2 節で述べたデバッグパターンを実現するための基礎となるものである。各パターンの詳細については第 6 章で述べる。

4.4　タイムスタンプのインクリメント

タイムスタンプはループの実行のように、同一の location が繰り返し実行され得る場合にインクリメントされなければならない。従って、C のソースコードの場合を例とすると、以下のような場所に制御が到達したときにインクリメントが起こる。

- 関数呼び出しの入口と出口（関数定義の先頭と末端）
- `return` 文
- ループの繰り返し
- 上にあるラベルへの `goto` 文によるジャンプ

関数呼び出しの入口でのインクリメントは、2 回以上呼び出される関数の呼び出しを区別するために必要となる。再帰的に定義された関数では再帰呼び出しからリターンした後で同一の location を通過する可能性があるので、関数呼び出しの出口でもインクリメントする。`return` 文は関数呼び出しの出口と同様であるが、返り値を持つ場合があり、実装上は別のものとして扱うことになる。

第5章　CおよびJavaへの実装

本章では、前章で述べたタイムスタンプ更新機構をC及びJavaのデバッギに自動的に組み込む方法[MT03b]について述べる。

5.1　Cへの実装

第3章の次のステップとして、まずはCに対する実装を行った[MT00a]。前章のタイムスタンプはデバッギ中に大域変数`timestamp`として導入し、これをインクリメントするコードをデバッギ中の制御ジャンプ命令の直前に追加する。コードの追加はコンパイラの中間コードレベルで行った。

5.1.1　更新機構挿入のレベル – C

デバッギにタイムスタンプ機構を挿入するには、中間コードレベルを含めて以下の3つのレベルが考えられる。本節ではそれぞれ方法の比較を述べる。

1. ソースコードレベル
2. 中間コードレベル
3. アセンブリコードレベル

第1のソースコードレベルとは、ソースコード変換を用いて行う方法である。この方法はソースコードそのものを対象とするので、処理系やOS、プラットフォームを選ばないという利点があるが、デバッグ中にプログラマから見えるコードは変換後のコードになってしまうという欠点がある。

第2の中間コードレベルでは、コンパイラが内部的に持っている中間コードを変換してタイムスタンプ機構を挿入する。コンパイラはソースコードを入力として受け取ると一旦この中間コードに変換し、最適化などを施した後で対象プラットフォームのアセンブリコードとして出力する（図5.1）。例えば、GNU C Compiler[Sta99]（GCC）ではRegister Transfer Language（RTL）と呼ばれる中間コードを利用している。中間コードレベルで挿入すればポータビリティが失われることはないが、特定の処理系に依存することになる。

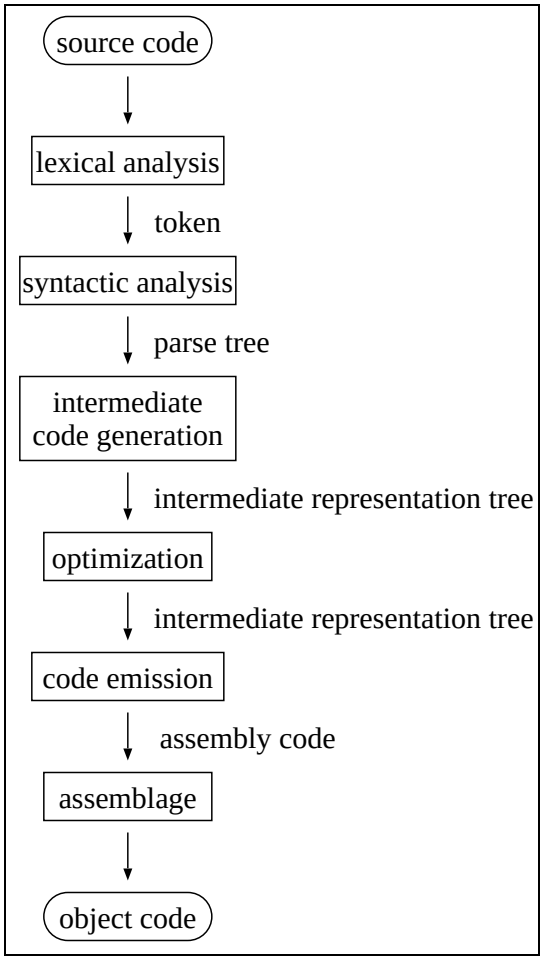

図 5.1: コンパイラの構成

第3のアセンブリコードレベルでは、図5.1の assembly code を変換する。これは実装が最も容易な方法であり、第3.3節で述べた関数単位疑似逆実行における新手法の実装で採用していたが、各環境毎に実装する必要があるためポータビリティに欠ける。

以上より、第2の中間コードレベルで実装することとし、対象とする処理系は様々なプラットフォームで広く利用されていることと GDB との連係を考慮して、GCC を選んだ。今回は実装時点での最新版である GCC-2.95.2 を実装の対象とした。

5.1.2 タイムスタンプ更新コード

タイムスタンプ更新場所に到達したときに図5.2のタイムスタンプ更新マクロ `INC_TS` を実行するように、デバッギが変換される。デバッギからデバッガへのトラップが、特定のタイムスタンプ値に到達したときにだけ起こるように工夫してある。

デバッギはコンパイラによって、全てのタイムスタンプ更新場所にマクロ `INC_TS` を

```
int timestamp = -1;
int ref = -1;

void brake(void){}

#define INC_TS if(++timestamp == ref) brake();
```

図 5.2: タイムスタンプ更新コード

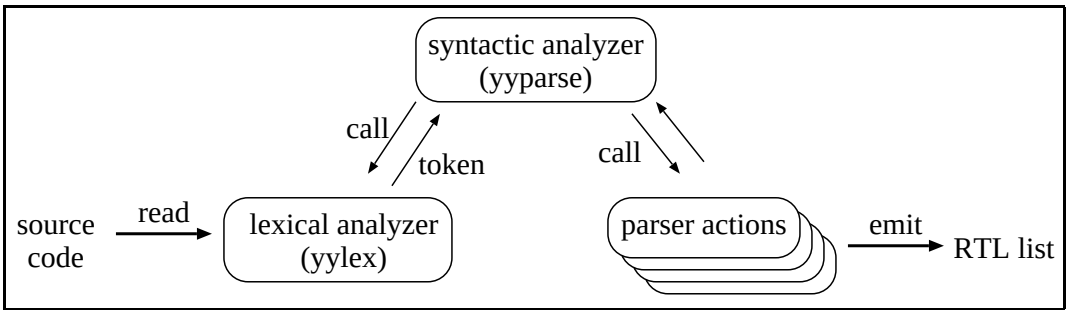

図 5.3: 構文解析器の振舞

展開したコードの RTL を自動的に挿入される。さらに、このマクロのために、デバッギの各ファイルの先頭部分には変数 `timestamp` と `ref` の `extern` 宣言、関数 `brake` のプロトタイプ宣言が RTL として挿入される。これらの実体は別途ライブラリとして用意し、コンパイラが自動的にリンクする。

5.1.3 GCC の変更点

図 5.1 ではコンパイラの処理のステージを列挙したが、実際のコンパイラの処理は明確には分かれていない。GCC は構文解析器である関数 `yyparse` が字句解析器である関数 `yylex` を呼び出し、読み込んだトークンを構文解析して対応するアクションを呼び出し、そのアクションが随時 RTL を出力するようになっている（図 5.3）。RTL は双方向リスト構造をしているので、ここでいう出力とはそのリスト構造へ新しい RTL 式を追加することを意味する。

タイムスタンプ更新コードの追加は構文解析器のアクションで処理するため、追加の場所はソースコードの構造と対応づけて考える。前章で述べた通り、以下の 4 種類の場所で追加する。

- 関数呼び出しの入口・出口

- return 文
- ループの繰り返し
- 上にあるラベルへの goto 文によるジャンプ

なお、この方法では、longjmp() の呼び出しは一般のライブラリ関数の呼び出しと同等となり、タイムスタンプをインクリメントしない。従って、現在は longjmp() を含むコード（longjmp() の呼び出しを含む実行系列）では、position の妥当性は保たれない。

構文解析器への変更

構文解析器である関数 yyparse はコンパイラが新しいソースファイルを読み込むたびに呼び出されるので、その処理の最初に変数 timestamp と ref の extern 宣言、関数 brake のプロトタイプ宣言を RTL リストへ追加する。

アクションへの変更

前章で挙げた、タイムスタンプを更新すべき 4 つの場合のそれぞれについて、どのアクションのどの部分で INC_TS の RTL を挿入すれば良いかを述べる。

ループの繰り返し ループが繰り返される場合は 2 通りある。1 つはループ本体の末尾に到達した場合で、もう 1 つは continue 文が実行された場合である。これらを考慮して、GCC はループを図 5.4 のような RTL リストに展開する。start_label はループ本体の末尾に到達したときのジャンプ先ラベルであり、continue_label は continue 文が実行されたときのジャンプ先ラベルである。ここでは for 文を取り上げたが、do-while 文でもほぼ同様になる。while 文では特に、start_label と continue_label が一致する。

タイムスタンプ更新場所であるループの繰り返しは本来ループ本体（図中の 4）の末尾に相当するが、continue 文による繰り返しも一緒に扱うようにした方が実装が容易になるので、start_label の直後（図中の☆）で更新することにする。ループに入る最初の 1 回だけ余分に更新されることになるが、全体の計算量に比べれば無視できる量である。

while 文のときは、構文解析器はアクションである関数 expand_start_loop を呼び出す。for と do-while のときは、expand_start_loop_continue_elsewhere が呼び出され、これが expand_start_loop を呼び出す。start_label の出力は expand_start_loop 内の関数 emit_label の呼び出しで行われるので、その直後で INC_TS を展開した RTL を RTL リストの末尾に追加した。

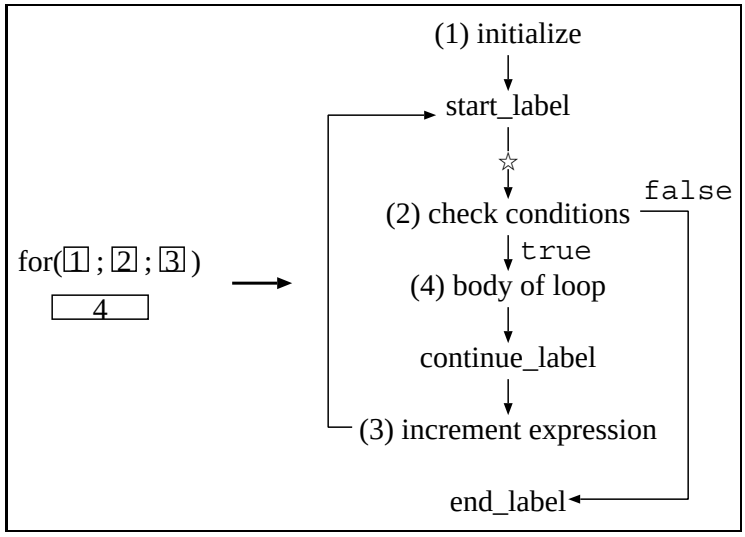

図 5.4: ループ構造の RTL への展開（☆は更新コード挿入位置）

goto 文　ANSI C に基づく goto 以外にも、GCC 独自の拡張として nonlocal goto と computed goto がある。nonlocal goto は nested function において外側の関数内部のラベルへジャンプするものであり、computed goto はジャンプ先のラベルを変数の値を用いて決定する。いずれも使用頻度は低いと考えられるので、今回の実装では考慮しなかった。また、本来はジャンプ先のラベルが下方にある場合はタイムスタンプ更新は不要であるが、実装を容易にするためジャンプ先ラベルの位置の判定を行わないことにした[1]。

通常の goto 文と nonlocal goto の場合、構文解析器はアクションである関数 expand_goto を呼び出す。expand_goto は nonlocal goto かどうかを判定して、通常の goto の場合は直ちに expand_goto_internal を呼び出すので、この呼び出し直前で INC_TS を追加するように変更した。

return 文　return 文のときは構文解析器がアクション c_expand_return を呼び出すが、返り値がある場合とない場合とでその後の動作が異なる。返り値がない場合は expand_null_return が呼ばれてそこから expand_null_return_1 が呼ばれるが、返り値がある場合は expand_return 経由で expand_null_return_1 が呼ばれ、いずれの場合も最終的にはそこで関数定義のエピローグへ向かって expand_goto_internal によってジャンプすることになる。

INC_TS の追加は 2 つの場合が合流する関数 expand_null_return_1 で行うこともできるが、返り値がある場合、その段階では既にレジスタ内に計算済みの返り値が設定されており、そこでタイムスタンプ更新マクロを実行すると返り値を壊してし

[1] タイムスタンプの更新は、実行系列中に同一の location が再び登場するまでの間に少なくとも 1 回登場すれば良いので、更新し過ぎてしまっても position の妥当性は失われない。ただし、余分な更新を行うことで処理速度が多少犠牲になる。

まうことになる。返り値が保存されている可能性のあるレジスタをスタックに退避してからタイムスタンプを更新するという方法もあるが、ここでは return 文の処理を開始する前である、c_expand_return の最初で追加することにする。

関数呼び出しの入口 関数定義の先頭ではまず、関数名と型の情報などを登録する start_function が呼ばれ、次に引数の名前と型を登録する store_parm_decls が呼ばれ、定義本体である compound statement の処理に移る。

関数呼び出しの入口で、かつ、関数定義のときにのみ INC_TS が実行されるためには、compound statement の処理に入る前に追加されることが望ましい[2]。そこで、store_parm_decls の末尾で追加する。

関数呼び出しの出口 関数のエピローグ部分を処理するアクションは finish_function である。このアクションは、return 文が実行されたときのジャンプの目標となる return_label をエピローグ部分に設定する。return_label よりも下に INC_TS を追加してしまうと、return 文のときに余分なタイムスタンプ更新が発生するだけでなく、レジスタに保存された返り値を破壊する可能性があるので、return_label より前に追加されなくてはならない。従って、関数 finish_function の最初で追加する。

コンパイラド ライバへの変更

必要なときだけタイムスタンプ機構が組み込まれるように、GCC に新しいオプション -pg2 を追加する。新しいオプションを設定するには、構造体 default_compilers を修正して、コンパイラ本体である cc1 のオプション設定に追加する。次に cc1 の main 関数のオプション解析ループの中に追加して、-pg2 が指定された時だけ特別なフラグを立てるようにする。前述の INC_TS 追加用のコードはこのフラグが立っている時だけ動作するようにしておく。

次に、brake などの定義が含まれるライブラリを -pg2 指定時に自動的にコンパイラがリンクするよう、各プラットフォーム用のヘッダファイルに含まれる LIB_SPEC マクロを修正する。例えば i386-linux であれば gcc/config/linux.h に含まれている。

追加されるコード

GCC に対する変更は全部で 70 行程度であった。このような変更を施した GCC の動作を以下のプラットフォームで確認した。別のプラットフォームへ移植するため

[2]構文解析器による compound statement の処理は、関数定義に限らず if 文などで登場する場合も同様に行われるので、compound statement のアクションで追加を行うと不要なタイムスタンプ更新が増え、実行性能に悪影響を与えてしまう。

```
incl timestamp
movl timestamp,%eax
cmpl ref,%eax
jne .L6
call brake
```

```
sethi   %hi(_timestamp), %o1
ld      [%o1+%lo(_timestamp)], %o0
add     %o0, 1, %o0
st      %o0, [%o1+%lo(_timestamp)]
sethi   %hi(_ref), %o1
ld      [%o1+%lo(_ref)], %o1
cmp     %o0, %o1
bne     L21
mov     0, %l2
call    _brake, 0
```

図 5.5: 追加されるアセンブリコード（i386-linux（左）と sparc-sunos4（右））

に必要なことは、プラットフォームごとのヘッダファイルに含まれるマクロを新たに 1 行書き直すだけである。

- i386-linux
- sparc-sunos4
- alpha-linux
- sparc-sun-solaris2.8

実際に追加された RTL によるアセンブリコードを、i386-linux と sparc-sunos4 のそれぞれについて図 5.5 に示す。これらは、オプションとしてデバッグ用とタイムスタンプ機構導入用、最適化用である、`-g -O -pg2` をつけて生成したコードである。

5.1.4 オーバヘッド

タイムスタンプ機構によるオーバヘッドを計測した。表 5.1 は gawk-2.15.6 をデバッギとした場合の実行速度のオーバヘッドを示している。その他のベンチマークについては表 5.2 に示す。empty loop ベンチマークは空ループのみを持つデバッギで、実行時間のうちの大部分をタイムスタンプのインクリメントが占めており、タイムスタンプ機構のオーバヘッドの最大値を示している。sed ベンチマークは sed-1.18 を用いた。また、表中のプラットフォーム名については詳細を表 5.3 に示してある[3]。

[3]SPARC-1 プラットフォームは既に失われており、残念ながら詳細な仕様は不明である。

表 5.1: C におけるオーバヘッド – gawk

GCC options	Intel-1	Alpha	SPARC-1
-g -O	2.57	4.64	17.44
	(1.00)	(1.00)	(1.00)
-g -O -pg2	3.78	6.34	24.40
	(1.47)	(1.37)	(1.40)

単位は秒（括弧内は比）

表 5.2: C におけるオーバヘッド – その他のベンチマーク

App.	GCC options	Intel-2	SPARC-2
empty loop	-g -O	2.762	1.012
		(1.00)	(1.00)
	-g -O -pg2	9.632	10.112
		(3.49)	(9.99)
sed	-g -O	9.182	3.182
		(1.00)	(1.00)
	-g -O -pg2	10.02	4.916
		(1.09)	(1.54)
gawk	-g -O	2.87	2.842
		(1.00)	(1.00)
	-g -O -pg2	4.466	4.598
		(1.56)	(1.62)

単位は秒（括弧内は比）

表 5.3: C におけるオーバヘッド – 各プラットフォームの概要

Platform	Architecture		OS
Alpha	Alpha	500MHz	Linux (glibc2)
Intel-1	Celeron	400MHz	Linux (glibc1)
Intel-2	Celeron	366MHz	Linux (glibc2)
SPARC-1	SPARC	N/A	SunOS-4
SPARC-2	UltraSPARC	500MHz	Solaris-8

表 5.4: C におけるファイルサイズ増分

App.	GCC options	Intel-2	SPARC-2
empty loop	-g -O	1144	2160
		(1.00)	(1.00)
	-g -O -pg2	1248	2416
		(1.09)	(1.12)
sed	-g -O	41030	55263
		(1.00)	(1.00)
	-g -O -pg2	52430	77215
		(1.28)	(1.40)
gawk	-g -O	140046	169310
		(1.00)	(1.00)
	-g -O -pg2	169414	223078
		(1.21)	(1.32)

単位はバイト（括弧内は比）

タイムスタンプの更新回数は、empty loop で 500000002 回、sed で 85888810 回、gawk で 96817515 回となっている。1 回のインクリメントにかかる平均時間はそれぞれおよそ 13.7 ナノ秒、9.76 ナノ秒、16.5 ナノ秒であり、実用に耐える。

表 5.4 はそれぞれのベンチマークのファイルサイズの増分を示しており、これらも実用に耐える数値であり。

5.1.5 動的ブレークポイントの実装

第 4.3 節では、デバッガの条件付ブレークポイントによる動的ブレークポイントの実装を紹介したが、ここでは最低限のオーバヘッドでの実装を示す。

まず、タイムスタンプ値が目標点のそれと等しくなる更新位置まで実行する（step 1）。すると、そこから目標の position までの間に、目標点と同じ location が別の position として存在し得ないことが保証されているので、通常のブレークポイントを用いて、さらに真の目標点まで実行すれば良い（step 2）。

図 4.1 におけるタイムスタンプ更新のタイミングを図 5.6 に示す。この例では目標点と同一の location は実行系列中に 3 回登場する（図中の△印）。しかしタイムスタンプが更新されるタイミングは、隣り合う location の間に必ず 1 つ存在する。すなわち、タイムスタンプが目標値（目標 position でのタイムスタンプ値）に更新されてから、目標 position までの間には目標 location は存在しない[4]。従って、前述の

[4] この場合の例では、タイムスタンプ更新点の間に必ず目標点と同一の location が出現しているが、出現しない場合ももちろん存在する。タイムスタンプは、その値が次に更新されるまでの間に、目標点と同一の location が 0 回または 1 回だけ出現することを保証する。

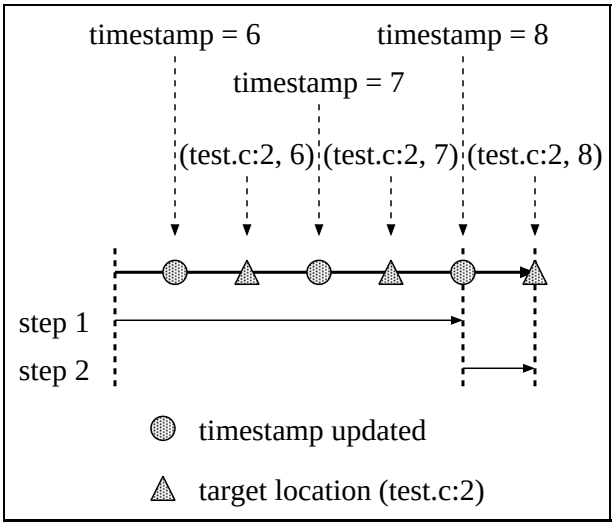

図 5.6: 動的ブレークポイントの実行手順

step 1 と step 2 は図のようになる。この方法は、`timestamp` と `ref` の比較をデバッギのコンテクストで実行し、デバッガへのプロセススイッチを最小にしたものと言える。

5.2　Java への実装

C に実装した経験に基づいて、Java プログラムに対してはコンパイラが生成したクラスファイル（バイトコード）を変換することでコードの追加を実現した。独立した `Timestamp` クラスを用意し、タイムスタンプ機構に必要な要素はそのクラスの static member とした。

5.2.1　更新機構挿入のレベル – Java

Java へ実装する場合の選択肢として以下の 4 つのケースを検討した。

1. コンパイラ（ソースコードレベル）
2. クラスファイル（バイトコードレベル）
3. VM（バイトコード実行レベル）
4. JPDA（デバッグインタフェースレベル）

第 1 と第 3 は C における検討と同様の理由で候補から除外される。第 4 の JPDA（Java Platform Debugger Architecture）は移植性に優れるが、デバッガに組み込ん

で利用する形となり、デバッギとの間のプロセススイッチによる実行速度の大幅な低下が懸念されるためこれも除外される[5]。

バイトコードは C におけるアセンブリコードと同質のものであるが、その形式は仕様 [LY99] によって定められており、ターゲットアーキテクチャに依存することがない。

5.2.2 タイムスタンプ更新対象のバイトコード

タイムスタンプ更新の対象となるバイトコードを検討する。

メソッド呼び出しの入口と出口 入口に対応するバイトコードはないが、メソッド本体の先頭がこれにあたる。出口には `return` バイトコード（命令コード：172–177）が該当する。
メソッド本体の先頭ではなく、`invokevirtual` のような、メソッドを起動するバイトコードを対象とする方法もある。しかしこの方法には追加するコード量がより大きくなるという欠点がある。また、変換していないクラスファイルのメソッド呼び出し（例えばシステムライブラリの呼び出しなど）までがタイムスタンプ更新の対象となり、オーバヘッドを増加させてしまうという欠点もある。

条件分岐 バイトコードの仕様として上向きのジャンプが起こりうるので、以下の 16 命令を対象に含める。`ifeq`（命令コード：153）から `if_acmpne`（命令コード：166）までと、`ifnull`（命令コード：198）、`ifnonnull`（命令コード：199）が該当。

無条件ジャンプ `goto`（命令コード：167）と `goto_w`（命令コード：200）。

その他の制御ジャンプ `jsr`（命令コード：168）と `ret`（命令コード：169）、`jsr_w`（命令コード：201）が該当。

例外 例外に関するバイトコードには `athrow`（命令コード：191）があるが、これだけでは NullPointerException のような、ユーザのコードが明示的に `throw` しない例外に対処できない。よって、例外を発生（throw）する場所ではなく、例外を捕捉（catch）する場所を対象としなければならない。タイムスタンプ更新コードは `catch` ブロックの先頭に挿入する。

[5] JPDA にはデバッギに組み込んで同一プロセス内で動作する JVMDI と、デバッガとして外部プロセスで動作する JDI が定義されており、JVMDI で実装した場合はデバッガとのプロセススイッチは発生しない。しかしながら、現在の Sun による Java の実装では JVMDI と JDI を同時に利用することができず、JVMDI で実装したタイムスタンプ機構と通常利用されるデバッガ `jdb` を同時に起動することができなくなる。

```
public final class Timestamp{
  private static long ts, ref;

  static{
    ts = 0;
    ref = 0;
  }

  public static void brake(){}
}
```

図 5.7: Timestamp クラス

```
getstatic Timestamp.ts
lconst 1
ladd
dup2
putstatic Timestamp.ts
getstatic Timesatmp.ref
lcmp
ifne XX
invokestatic Timestamp.brake()
```

図 5.8: 追加されるバイトコード

これらのうち、条件分岐と無条件ジャンプ、その他の制御ジャンプでは、各バイトコードはジャンプ先の番地をオフセットとして引数に保持している。これが負である場合（上向きジャンプの場合）に限って更新コードを挿入する。

5.2.3 タイムスタンプ更新コード

Timestamp クラスは図 5.7 のように定義されている。また、追加されるタイムスタンプ更新コードは図 5.8 のようである。図中の XX は制御ジャンプを起こすバイトコード（invokestatic の直後のバイトコード）の位置を示す。

表 5.5: Java におけるオーバヘッド

Benchmark	with JIT		without JIT	
	original	transformed	original	transformed
empty loop	12.932	41.448	–	–
	(1.00)	(3.21)	–	–
_201_compress	1.3364	2.1986	–	–
	(1.00)	(1.65)	–	–
_202_jess	0.1962	0.2162	–	–
	(1.00)	(1.10)	–	–
_209_db	0.4882	0.4664	2.1442	2.4814
	(1.00)	(0.96)	(1.00)	(1.16)
_222_mpegaudio	0.1894	0.2444	–	–
	(1.00)	(1.29)	–	–
_228_jack	0.5288	0.6536	–	–
	(1.00)	(1.24)	–	–

単位は秒（括弧内は比）

5.2.4 クラスファイル変換プログラム

クラスファイルを変換するプログラムはバイトコード解析ツール BCEL[Dah99] を用いて Java で記述し、コード量は 200 行程度である。このプログラムはタイムスタンプ更新コードの挿入に加え、関連するフィールド名などをコンスタントプールに追加する。

ユーザは、タイムスタンプによる実行制御の対象としたいコードを含むクラスファイルを、事前にこの変換プログラムで変換しておく。実行制御の対象としない場合は変換しなくてよく、その分、実行時のオーバヘッドを減らすことができる。

5.2.5 オーバヘッド

タイムスタンプ機構を組み込んだ Java プログラムのオーバヘッドを計測する。対象のプログラムには、オーバヘッドが最大になる空ループのみを持つプログラム（empty loop）と SPEC JVM98[Sta] のベンチマークプログラムを用いる。また、計測を行った環境は Pentium-III 733MHz、640MB メモリ、Linux-2.4.7、JDK-1.4.2 である。

実行時間の計測は各ベンチマークを 7 回連続して実行し、最短と最長の結果を除外した 5 回分の平均を求めた。結果を表 5.5 に示す。empty loop は最悪の場合を示しており、この実装方式での速度低下はたかだか 3.2 倍程度であると言える。しかし概ね 1.5 倍から 2 倍程度であり実用に耐える。

表 5.6: Java におけるファイルサイズ増分

Benchmark	original	transformed
empty loop	276	507
	(1.00)	(1.83)
_201_compress	17821	21944
	(1.00)	(1.23)
_202_jess	396536	467309
	(1.00)	(1.18)
_209_db	10156	13097
	(1.00)	(1.29)
_222_mpegaudio	120182	143235
	(1.00)	(1.19)
_228_jack	132516	166959
	(1.00)	(1.26)

単位はバイト（括弧内は比）

クラスファイルの大きさの合計を比べたものを表 5.6 に示す。元のファイルが極端に小さい空ループの場合に増分がやや顕著になるものの、多くの場合においてほとんど変化がなく実用上の問題はないと考える。

第6章　実行点の応用：デバッグパターンの実装

position を用いて、実行系列中の任意の 1 点を特定できるようになった。これを用いた応用は 2 種類に大別できる。1 つは人間が手動で行っていた手続きを自動化するような応用であり、もう 1 つは、それをさらに進めて、手動では到底できないような手続きを行うものである。これらはひどく乱暴な手法に見えるかも知れないが、近年の高速な計算機を利用することで、人間が手動で行うよりもずっと効率良くデバッグを行うための有用な手法である。

6.1　実行点の印づけ

通常のデバッグにおいてバグの原因である場所が曖昧にしか分かっていないとき、その場所の手前に（静的）ブレークポイントを設定して制御を移動し、ステップ実行を繰り返し行う、という方法で制御点を動かす。大規模なプログラムの中をこの方法で移動していると、うっかりコマンドを間違って問題の場所を通り過ぎてしまうことがある。このようなときには、そこへ至る経路を覚えておくか書き留めるかしておき、デバッギを先頭から再実行してその経路を手動で追いかけなくてはならない。

こうした面倒を避けるためには、実行点、すなわち position に印づけをすることが有効である。デバッギの振舞が「ここまでは正しい」というときに動的ブレークポイントを用いて印をつけ、後になってやり直したいようなことがあったらその position まで簡単に戻ってくることができる。実行系列の中で、いわば「足場」のようなものを確保していると言える。

さらに、印をつけたいくつもの position を区別するのに、単なる ID 番号ではなく、自分でコメント文を書くことによって識別をより簡単にすることもできる。コメントとしては、例えばこんなものがあるだろう。「今、右ブレースを読み込んだので、複合文の処理をするパーザアクションに制御が移るに違いない。」

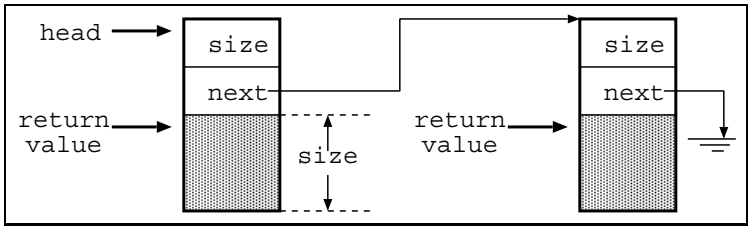

図 6.1: simple_alloc による領域管理

6.2 逆向きウォッチポイント

メモリを動的に確保するようなプログラムを考える。malloc のようなアロケータ関数でメモリを確保するものの、プログラマは誤って、確保した領域よりも大きなデータを書き込んでしまい、malloc や free が使うような管理領域の情報を破壊してしまう。しかしその破壊操作は直ちにはエラーを引き起こさず、バグの影響はずっと後になって顕在化する。配列に、その確保した領域を越えてデータを書き込んでしまうときも同様で、プログラマは後になって代入した覚えのない変数の値がおかしくなっていることに気づく。

このような不正な書き込みを捕捉するために、デバッガにはデータアクセスブレークポイントという仕組みが備わっており、GDB ではウォッチポイントと呼ばれている。おかしな値になってしまった変数に対してウォッチポイントを設定すると、その変数に対応するメモリ領域へ書き込みが発生する度にトラップが起こり、その内容が変更されたことをプログラマに通知する。

一般には変数は何度も書き換えられるものであり、その度にトラップが発生して、プログラマはそれぞれの書き込みで何が起こったかを検査しなくてはならない。しかしながら、多くの場合、ある問題を引き起こした書き込みというのはそのバグの影響が現れた時点から時間的に一番近いものが原因になっていて、実際に我々はそれを探すことを手作業で行っている。従って、ウォッチポイントによる直前のトラップまで自動的に制御を移す仕組みがあれば、プログラマの作業がかなり軽減されることになる。この機構を「逆向きウォッチポイント」[MT03a] と呼ぶことにする。

この手続きは動的スライシング [AH90] を手動で生成することに相当する。この手法の利点は、プログラムの制御点を、ある変数が代入された position に実際に移すことができるという点にある。この際、C における “unconstrained pointers” [BG96] の問題は考慮する必要がない。スライシングの技術はソースコード解析によっているが、デバッガのウォッチポイントハードウェアの支援を受けて代入に関する完全に正確な情報を手に入れることができる。

```
    int main(void){
      int i;

(1)   init_heap();
(2)   area1 = simple_alloc(4);
(3)   for(i = 0; i < 253; i++){
(4)     area2 = simple_alloc(4);
(5)     simple_free(area2);
      }
(6)   area2 = simple_alloc(4);
(7)   string_copy(area1, "abcdefghijk");
(8)   area3 = simple_alloc(4);
(9)   return(0);
    }
```

図 6.2: メモリ確保プログラム

6.2.1 利用例

C における malloc のような、メモリ確保をするプログラムを考える。関数 simple_alloc は引数で指定された大きさの領域を確保し、確保した領域のアドレスを返す。simple_alloc は領域を線形リストで管理し、新しい割り当ては first-fit に基づいて行う。管理のために、割り当てられた領域にはヘッダ情報として、(1) その領域の大きさと (2) 次に割り当てられた領域のヘッダへのポインタが格納されている (図 6.1)。関数 simple_free は、simple_alloc で確保した領域が不要になったときに、再利用できるようにヒープに返すのに用いる。このプログラムの main 関数を図 6.2 に示す。

このプログラムは、最初に行 (2) で 4 バイトの領域を area1 に確保する。次に、別の 4 バイトを area2 に確保し (4)、それを解放するという操作を 253 回繰り返した後で、再度 area2 に確保をする (6)。そして関数 string_copy を用いて area1 に文字列をコピー (7) するときに、area2 の管理領域を破壊する。最後に area3 に領域を確保して終了する。area3 が確保された後のヒープの状態は、図 6.3 のようになっているはずであるが、このプログラムは segmentation fault を引き起こす。早速、これをデバッグしてみる。

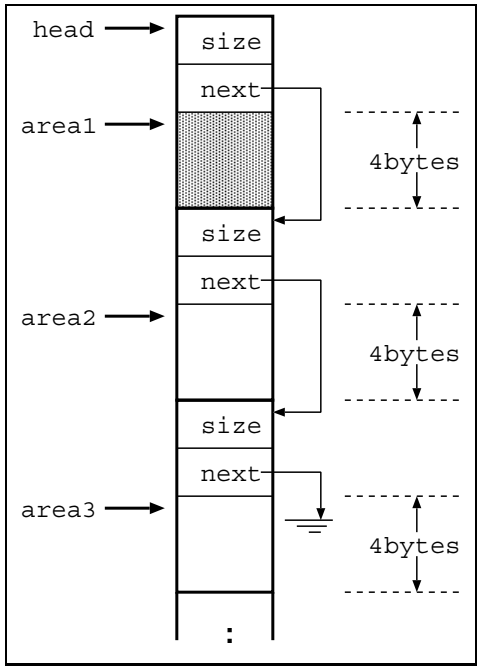

図 6.3: 本来のヒープの構造

バグ発覚場所の特定

このプログラムをデバッガを用いて動かすと、バグは図6.2の行（8）の simple_alloc を呼び出したところでエラーとして発覚していることが分かる。これは area3 に領域を確保しようとしているところである。simple_alloc は first-fit なので、area3 は area2 のすぐ後ろに割り当てられるべきである。simple_alloc は空き領域を探すために先頭（head）から next をたどっていって、area2 の next にアクセスしたはずである。そこで area2 の管理領域をデバッガで調べてみると以下のように表示された。

```
(gdb) print head->next->size
$2 = 1751606885
(gdb) print head->next->next
$3 = (struct cell *) 0x6b6a69
```

area2 は 4 バイトのはずなのにかなり大きなものになっている。しかも線形リストの末尾のはずが、その次のセルがひどく遠いところにあることになっている（図6.4）。ここに至って、何らかの操作によって area2 の管理領域が破壊されたと推測できる。

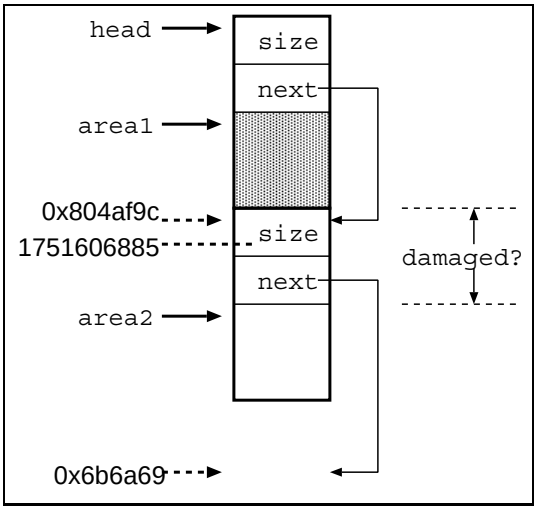

図 6.4: 破壊されたヒープ

逆向きウォッチポイントによるデバッグ

そこで、注目する領域に最後に書き込んだ操作を知るために逆向きウォッチポイントを用いる。area2 の管理領域（0x804af9c）に対して逆向きウォッチポイントを行うと、以下のような出力が得られた。

```
Old value = 6776421
New value = 1751606885
string_copy (loc=0x804af98 "abcdefgh",
       string=0x8049758 "abcdefghijk")
    at allocator.c:145
145         i++;
```

これで、最後の操作が行（7）で呼び出された関数 string_copy 内であり、4 バイトしかない領域に大き過ぎるデータを書き込んだのが原因だと分かった。

通常のデバッグ

もし逆向きウォッチポイントがなかったら、このデバッグはやっかいなものになる。実行の最初に通常のウォッチポイントを設定してからデバッグを始めると、対象領域はループを回る度に繰り返し書き換えられるので、デバッギは何度も停止してしまう。そして、止まる度に表示された値が正当なものであるかを調べ、デバッガに継続を指示しなくてはならない。一般にはこの停止が何回起こるかは分からないので、プログラマは大量の時間と労力を浪費することになる。

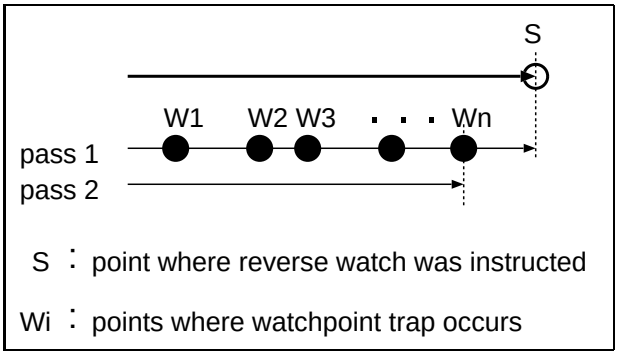

図 6.5: 逆向きウォッチポイントのアルゴリズム

6.2.2　逆向きウォッチポイントの実装

逆向きウォッチポイントは前章で述べた動的ブレークポイントと既存のデバッガを用いれば容易に実装することができる。図 6.5 にそのアルゴリズムを示す。

1. 逆向きウォッチポイントが指示された position（図中の **S**）に動的ブレークポイントを設定する。

2. pass 1： デバッギを再実行し、ウォッチポイントによるトラップで停止したときは、その position に印づけをするために、動的ブレークポイントを設定するために必要な情報である (file:line, timestamp) を取得する。図中の **W1** から **Wn** がこれにあたる。それから実行を続行する。

3. 制御点が逆向きウォッチポイントを指定した position に到達したら（動的ブレークポイントを設定したのでこれを知ることができる）、

4. pass 2： もう一度デバッギを再実行し、ウォッチポイントによる最後のトラップが発生した場所（図中の **Wn**）に、新たに動的ブレークポイントを設定してから実行を続行する。

5. 目標の position である **Wn** で停止する。

逆向きウォッチポイントはデバッガとのインタラクションが必要になるので、新しいデバッガは Expect モジュールを用いた Perl スクリプトで記述し、GDB のラッパーとして実装した。GDB は GDB-5.0 を用いた。

6.2.3　逆向きウォッチポイントの処理時間

逆向きウォッチポイントの処理にかかる時間を計測した。デバッギには図 6.2 に示したものを用いた。この処理時間には以下のものが含まれる。

表 6.1: 逆向きウォッチポイントの処理時間

trap of watchpoint (times)	elapsed time (sec.)
511	6.56
1011	11.94
1511	17.58
2011	23.46

- Perl と GDB の起動時間。
- pass 0： segmentation fault で停止する、通常実行パス。
- pass 1： 逆向きウォッチポイントの pass 1。デバッギを先頭から再実行し、ウォッチポイントで停止する度に、デバッガを用いて現在の position 情報である (file:line, timestamp) を取得、保存する。
- pass 2： 逆向きウォッチポイントの pass 2。pass 1 で取得したリストの最後の position まで、動的ブレークポイントで用いて先頭から再び再実行する。

逆向きウォッチポイントの処理時間は pass 1 におけるウォッチポイントの停止回数に依存するので、これを 511 回から 2011 回まで変えて計測を行った。計測に用いた計算機は Celeron 400MHz、Linux-2.0.36 であり、GNU time-1.7 で elapsed time を計った。結果を表 6.1 に示す。

デバッギそのものの実行時間は `time` コマンドで計測できないほど微々たるものであったので、表 6.1 の結果は逆向きウォッチポイントの処理時間そのものであるとみなしても差し支えない。処理時間はウォッチポイントの停止回数にほぼ比例しており、停止回数はデバッギの総実行時間にほぼ比例すると考えられるので、実用上問題のない実行時間であると考える。

逆向きウォッチポイントは、特定の条件を満たす position のリストを取得して、そのうちの最後の position に制御を移動している、とみなすこともできる。逆向きウォッチポイントでは、その条件として対象の変数へのウォッチポイントを用いたが、条件付きブレークポイントのような他の条件も利用できるし、制御の移動先として最後の点ではなく中間点を使うこともできる。次節ではその例を挙げる。

6.3 タイムスタンプを用いた実行系列中の 2 分探索

ここまでは、これまでプログラマが手動で行ってきた手続きを自動化するというものであった。ここでは、手動では到底できないような手続きを実行する例について述べる。

逆向きウォッチポイントは特定の変数に対して設定したので、その変数を監視してバグの原因を自動的に探すことができた。これに対して本節では、ある条件が成立しなくなるときの position を自動的に探してバグ特定を補助する仕組みとして、タイムスタンプを用いた実行系列中の 2 分探索（以下、単に 2 分法と呼ぶ）を提案する。

タイムスタンプというのはステップ実行カウンタをもっと疎らにしたもので、その値はプログラム実行系列中における大まかな位置を表していると考えられる。例えば、タイムスタンプ値が 1 で始まり 10000 で終わるようなプログラムを考えると、その値が 5000 になる場所は実行系列において概ね半分の位置とみなせるということである。そこで、タイムスタンプ値をキーとする 2 分法を用いて実行系列中を移動することで、ある条件が不成立になるときの position を見つけられる。

この手法については既に Tolmach ら [TA95] によって提案されているが、本研究ではこれを関数型プログラミングの領域から手続き型プログラミングの領域に持ち込むことに成功した。

条件の例として、双方向リストを構成するプログラムがあるとすると、そのリストの一貫性を調べるためのある種のアサーションをコード中に用意しておけば、それが最初に偽になるときの position を知ることができる。例えば、リストの先頭から順方向リンクをたどって末尾まで行き、逆方向リンクをたどって先頭まで戻ってくることができる、といったアサーションが考えられるであろう。条件の満たすべき要件については第 6.3.3 節で述べる。

双方向リストのデータ構造の一貫性を調べるアサーションであれば、デバッギ実行中のその真偽は、通常、真から偽へ 1 回変化するだけであり、偽（異常な状態）になってから復旧することはまずあり得ない。しかし、アサーションやデバッギによっては、デバッギ実行中に真になったり偽になったりを繰り返す場合もあり得る。そのようなときでも、この手法を用いれば、いくつかある「真から偽への変化」のうちの 1 つは見つけることができる。

6.3.1　2 分法のアルゴリズム

2 分法を実行するプログラムは、以下のようにしてデバッギを制御すれば良い。

1. タイムスタンプの値が左端と右端の値の中間値になる position へ、制御を移動する。最初は、左端の点は実行系列の先頭に等しく、右端の点は系列の末尾に等しい。

2. 条件を評価し、

   - 真であれば、現在点と右端の中間点へ移動し、左端の点を新しい現在点に変更して対象領域を半分にする。

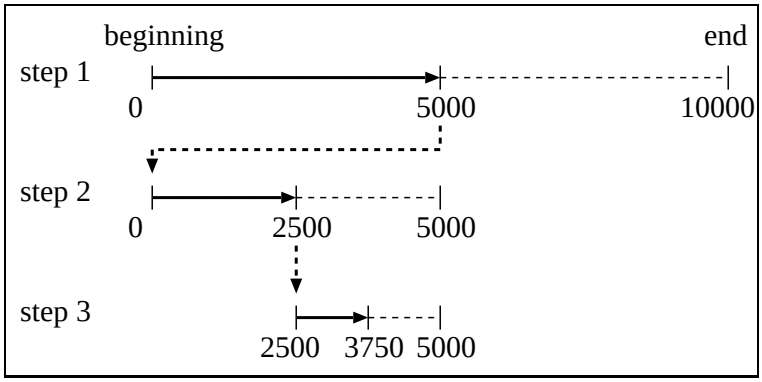

図 6.6: 2 分法の例

- 偽であれば、左端と現在点の中間点へ逆実行を用いて戻り、右端の点を新しい現在点に変更して対象領域を半分にする。

3. 以上を繰り返す。

この例を図 6.6 に示す。タイムスタンプ値が 10000 まで増えるプログラムを扱うとする。step 1 では、1 と 10000 の中間値である 5000 まで進む。条件を評価して偽であったとすると、step 2 に移り、デバッギを再実行して左端（タイムスタンプ値が 1）と現在点（5000）の中間点まで戻る。タイムスタンプ値が 2500 の場所で条件を評価して真になったとすると、step 3 で現在点の右端の中間点（タイムスタンプ値が 3750）へと進む。この手続きを繰り返すと、指定の条件が最初に偽になる position が見つけられる。

6.3.2 2 分法と `assert` との比較

条件が不成立になる状況とは、プログラムが `assert` ライブラリ関数で異常終了する場合と同等であると考えられるので、2 つの利点を除いては、2 分法は `assert` を用いて実行を停止させることに等しい。2 分法が優れる点のうちの 1 つは、`assert` は必要になるかも知れない全ての場所（location）に手動で挿入しなければならない、という点である。2 分法では手動で挿入する必要はない。もう 1 点は、挿入された `assert` はその呼び出しが起こる度に条件を評価してしまう、という点である。2 分法を用いれば、タイムスタンプの最大値（実行系列末尾でのタイムスタンプ値）を n とすると評価の最大回数は $\log n$ となるので、条件が複雑なものであれば、条件の評価に要する時間は 2 分法の方が `assert` よりも短くなる。

2 分法の処理に要する時間の大半は、条件の評価とデバッギ自身の実行時間である。条件評価は高々 $\log n$ 回であり、デバッギ自身の実行時間の合計も 1 回の実行時間の高々 $\log n$ 倍で済む。タイムスタンプとして 32 ビットの符号なし整数を用いれば、条件評価は 32 回以内、実行時間は 32 倍以内と見積もることができ、実現可

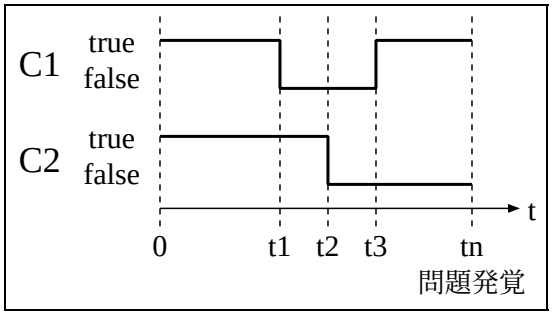

図 6.7: 単調性を持たない条件の例

能な範囲にあるものと考える。

6.3.3　2 分法の条件の要件

2 分法は条件が成立から不成立へと変化する瞬間（パルス信号における立ち下がりのような瞬間）を発見する手法である。実行の過程で立ち下がりが複数存在するような条件を選択した場合でも、2 分法はその中の 1 つを見つけ出すことができる。しかしそのためには、デバッギの実行の最後で不成立となるような条件を選ばなければならない。従って 2 分法の条件は、立ち下がりは何回存在しても良いが、実行の最後で不成立となる、という要件を満たさなくてはならない。

本当のバグに関連する条件がこの要件を満たさない場合、顕在化する場所を発見するには 2 分法を何段階かに分けて適用することになる。図 6.7 のようなシナリオが考えられる。本当のバグは条件 C1 におけるある変数を一時的に不正な値にする（図中の t_1）。その不正な値は別の条件 C2 の別の変数に伝播し（図中の t_2）、C1 は正しい状態に戻り（図中の t_3）、C2 は不成立となる。C1 を使って 2 分法でデバッギを調べても、C1 が要件を満たさないためにその立ち下がり（t_1）を捉えることはできない。しかし条件として C2 を用いれば、C2 の立ち下がり（t_2）を見つけることはできる。その position では C1 はまだ不成立となっているから、2 分法の開始時の右端をその position（t_2）にし、C1 を用いて 2 分法を行えば C1 の立ち下がり（t_1）を捕捉できることになる。

前述の双方向リストの一貫性を検査する条件を選択した場合、リストのセルのつなぎ変えを行っている瞬間は条件が不成立になる。しかしこの場合の不成立は過渡的な状態であるから 2 分法の条件評価の対象外としなければならない。従って 2 分法の実現には、ある関数の実行中は条件評価を行わない、というような仕組みが必要である。

第7章　マルチスレッドプログラムへの応用

これまでは再現性を持つデバッギに限って議論してきた。本章では、再現性を持たない（非決定的な）デバッギに対して、前章で述べた応用を適用するための方法について述べる。

7.1　非決定性の除去

プログラムの非決定性の要因には、外的要因と内的要因が存在する。入力を伴うプログラムではそのタイミングが決定的であっても（同期式入力）、入力データが変化することによってプログラムは再現性を失う。ネットワークプログラムや GUI（Graphical User Interface）のように非同期式入力を行う場合は、入力データだけでなく、入力が発生したタイミングも非決定性の要因となる。これらの要因は全て外的なものである。一方で、マルチスレッドプログラムではスレッド切替が非決定性の要因となっており、これは内的な要因であると言える。

こうしたプログラムをデバッグする際には cyclic debugging のやり方は通用しなくなってしまうが、可能な限り非決定性を排除し、バグの再現性を高めてデバッグするのが定石である。外的要因による非決定性がある場合には、外界の環境を決定的にすれば良い。例えば同期式入力を伴うデバッギであれば、確実にバグが発生する入力データを用意して入力を固定することで再現性を持たせ、cyclic debugging を行う。あるいは GUI を持つプログラムでは、Xlab[Ver98] のような X Window System のイベントを記録して再生するツールを利用することで、再現性を持たせることができる。非決定性が内的要因によっている、マルチスレッドプログラムのような場合には、スレッド切替のタイミングを記録して再生すれば良い [CS98]。

本章の残りの部分では、ユニプロセッサ上での Java のマルチスレッドプログラムを対象として、JPDA を用いたスレッド切替の記録・再生の実装方法を述べ、オーバヘッドを示す。

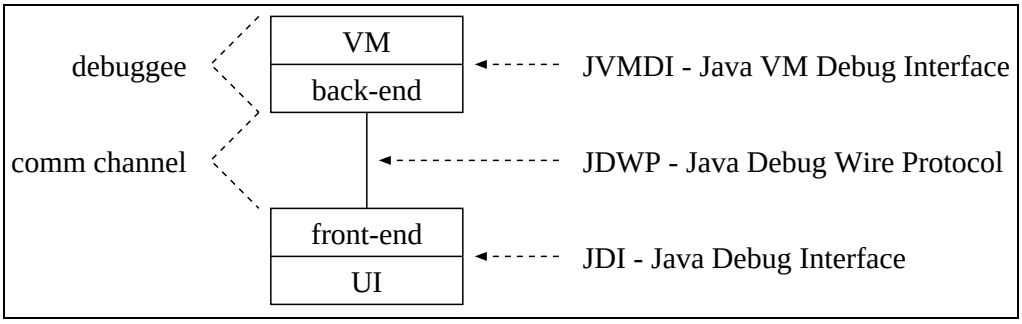

図 7.1: JPDA の構成 [Sunb]

7.2　スレッド 切替の記録

JPDA にはデバッグ用のインタフェースとして、JDI（Java Debug Interface）と JVMDI（Java Virtual Machine Debug Interface）の 2 つが定義されている（図 7.1）。

一方の JDI は通常のデバッガが利用するインタフェースであり、Sun が提供する Java SDK に付属のデバッガである `jdb` はこの JDI を利用して書かれた Java アプリケーションである。従ってデバッギとは異なる VM で動作し、OS の上でも別プロセスとなる。

他方の JVMDI は、デバッガのコードをデバッギを動作させる VM に組み込んで動作させるためのインタフェースである。こちらは C のような native の言語で書く必要があるが、デバッギとは同一のプロセスで動作する。

スレッド 切替を記録する際の時間的なオーバヘッドをなるべく小さくするという立場から、JVMDI を用いて記録することを選択した。

JVMDI はイベント駆動型で利用しなければならない。また、スレッド 切替を通知させるような仕組みも存在しない。そこで、基本的な記録のアルゴリズムは図 7.2 のようになる。

7.2.1　スレッドの識別

図 7.2 の (1) では、スレッドが同一かどうかを識別している。JPDA を利用して取得できるスレッドの識別子には、Java オブジェクトとしてのハッシュ値とスレッド名とがある。

ハッシュ値は 1 回の実行の間では unique であるが、もう一度実行した場合には同じスレッドに異なるハッシュ値が割り当てられることもあり、cyclic debugging の手法を利用するという観点からは不適当である。

スレッド 名は、プログラマが指定しない場合はクラス `Thread` のコンストラクタが呼ばれた順に数字が割り振られるので、スレッド 切替を保存すれば繰り返し実行しても変化しない。しかしスレッド 名はプログラマが自由な文字列を指定可能であり、同じ名前のスレッドが発生することもある。

```
    バイトコード単位のステップ実行イベントの通知を設定;
    while(イベントの通知を待つ){
      if(ステップ実行イベント){
        ステップ回数++;
(1)     if(前回のスレッド != 今回のスレッド){
(2)       ログに出力;
          前回のスレッド ← 今回のスレッド;
        }
      }
    }
```

図 7.2: スレッド切替を JVMDI で記録するアルゴリズム

ここでは cyclic debugging に基づく再実行を利用したデバッグパターンを考えるので、スレッドの識別子にはスレッド名を用いることとする。従って、プログラマはスレッドごとに異なる名前がつくようにデバッギを作成しなければならない。

7.2.2 ログの形式

ログにはスレッド名とステップ実行のカウントを記録する。ログの例を図 7.3 に示す。ログは 2 つのカラムからなり、第 1 カラムがスレッド名、第 2 カラムがステップ数である。ログは最初の行を除いて 2 行の組になっている。例えば 2 行目と 3 行目が組であり、スレッド `s1` がステップ数 10 から 15 まで実行されたことを意味する。ログの 1 行目の前には本来 `main  0` という行が存在しなくてはならないが、実装の都合上、出力されない。

図 7.2 の (2) に示したログへの出力では、毎回 2 行が出力される。1 行目は図 7.3 の○印であり、2 行目は図 7.3 の△印である。○印の行は図 7.2 の「前回のスレッド」、すなわち「実行を中断するスレッド」のステップ実行情報にあたる。△印の行は「今回のスレッド」、すなわち「実行を再開するスレッド」のステップ実行情報である。

7.2.3 タイムスタンプ更新の排他制御

Java の実装では、図 5.8 に示したタイムスタンプ更新コードは critical section であり、マルチスレッドプログラムでは排他制御が必要になる。排他制御の実現には、クラスファイル変換による方法と実行時に行う方法とがある。

Java のソースレベルであれば、図 7.4 に示したように `synchronized` を用いれば

```
main                    9
s1                     10
s1                     15          ← ○
main                   16          ← △
main                   16
s1                     17
s1                     25
main                   26
main                   30
          :
```

図 7.3: スレッド切替ログの例（先頭部分のみ）

```
synchronized(Timestamp.lock){
  if(++Timestamp.ts == Timestamp.ref) Timestamp.brake();
}
```

図 7.4: タイムスタンプ更新コードの排他制御（ソースレベル）

よく、クラスファイル変換プログラムは図のソースコードに相当するバイトコード列を挿入すればよい。その代わり、マルチスレッドプログラム用の変換プログラムを別途用意することになり、プログラマは 2 種類の変換プログラムの使い分けが必要となる。あるいは、デバッギがマルチスレッドであるか否かに関わらず、いずれの場合でも `synchronized` 方式の変換プログラムを用いるという方針もあり得るが、シングルスレッドのデバッギでは多大なオーバヘッドを発生するため、やはり好ましくない。

ここでは、デバッギがマルチスレッドか否かに関わらず同一のクラスファイル変換プログラムが利用できるように、排他制御を実行時、すなわちスレッド切替の記録時に行うこととした。よって、スレッド切替記録プログラム（以下、レコーダと呼ぶ）は図 7.5 のような手続きを行う。

図の (1) により、デバッギのあるスレッドが critical section に入る際の通知を起こさせる。そのときは制御が (3) に移り、JVMDI が提供するモニタである `RawMonitor` を取得して（図の (4)）、他のスレッドが critical section に入ることを抑制する。図の (5) で「残りステップ数」に設定する 8 という数字は、critical section であるタイム

```
(1) Timestamp.ts に FieldAccess イベント（読み込みアクセスの watch）を設定;
(2) while(イベントの通知を待つ){
(3)   if(FieldAccess イベント){
(4)     RawMonitorEnter();
        モニタ取得スレッド ← 今回のスレッド;
(5)     残りステップ数 ← 8;
      }else if(ステップ実行イベント){
        if(モニタ取得スレッド == 今回のスレッド){
(6)       残りステップ数--;
(7)       if(残りステップ数 == 0){
            モニタ取得スレッド ← 空文字列;
(8)         RawMonitorExit();
          }
        }
      }
    }
```

図 7.5: レコーダの排他制御アルゴリズム

スタンプ更新コード（図 5.8）を終了するまでのステップ数である[1]。その後、モニタ取得スレッドがステップ実行を行うと制御が図の (6) に至り、critical section の実行が終了した場合（図の (7)）には取得していた RawMonitor を解放する（図の (8)）。

レコーダはデバッギの振舞を大きく変えてしまうように思えるかもしれない。タイムスタンプ保護のために排他制御は導入せざるを得ず、前述の通り、クラスファイル変換プログラムによるか（第一の方法）、レコーダによるか（第二の方法）の 2 つの選択肢がある。図 7.5 に示した第二の方法による排他制御の動作は、図 7.4 に示した第一の方法によるそれと同じであり、本質的な差異はないといえる。

JVMDI のイベント待ちループは単一なので、実際には図 7.2 と図 7.5 とを組み合わせたものがレコーダの動作となる。

7.3 スレッド切替の再生

スレッド切替再生プログラム（以下、プレーヤ）の振舞は簡潔である。常に 1 つのスレッドだけを実行可能とし、スレッド切替のタイミングで実行中のスレッドを

[1] スレッド切替の記録時は Timestamp.ref は使用されないため、図 5.8 の invokestatic に到達することはない。

停止させ、次に動作すべきスレッドを再開させれば良い。図 7.3 の例では、9 ステップ目が終了したところで main スレッドを停止させ、s1 スレッドを再開させる。

プレーヤに関する本質的な部分は以上であるが、実際には JPDA の制約によっていくつかの問題が発生する。

7.3.1　JPDA による制約

第 7.2.1 節で、cyclic debugging におけるスレッドの識別にスレッド名を使わざるを得ないという制約については述べた。これはレコーダ実装時に判明した制約であったが、プレーヤ実装時に判明したその他の制約について述べる。

JDI と JVMDI の排他利用　レコーダは JVMDI で実装したので、プレーヤも同様に実装すればスレッド切替が自動的に再生されるデバッギ・プロセスが実現できる。このプロセスを、JDI を利用するデバッガによって制御するのがスマートな実装である。しかしながら、JDI と JVMDI を同時に利用できないという現在の JPDA の制約によって、この方法は選択できない。そこで、プレーヤは JDI で記述した。

前述の通り、JDI ではデバッギとデバッガが別プロセスとなるため、イベントの通知は OS におけるプロセス切替の発生を意味する。バイトコード単位のステップ実行ごとにプロセス切替を行うのは実行効率を著しく低下させると考えられる。そこで、JDI における StepRequest の addCountFilter() を用いて、イベントの発生が指定した回数に達するまでは通知しないようにしている。図 7.3 の 9 ステップ目が終了した時点を例に採ると、addCountFilter(6) とすれば良い。

新規スレッド開始の不定性　スレッド生成元によって Thread.start() が呼び出されると新規スレッドの実行が実際に開始されるが、そのタイミングには再現性がない。従って、プレーヤは新規スレッドが実行可能状態になるまで、他のスレッドを動作させることなく待たなくてはならない。

図 7.3 のログを例にとって考える。レコーダがログを記録した際には、2 行目の直前に s1 スレッドの開始のイベント（ThreadStartEvent）が発生していた。しかしながら、再生の際にプレーヤが main スレッドを 9 ステップ目まで動作させたとしても、s1 スレッドの ThreadStartEvent が発生するとは限らない。

そこで、デバッギとは無関係な IdleThread スレッド（図 7.6）を作成し、このスレッドをステップ実行することで新規スレッドの開始を待ち合わせる機構をプレーヤに採り入れた。IdleThread に関する問題点を以下に示す。

- 図の (1) は本来、空文で十分である。しかし空文のループをコンパイルすると、自分自身に goto するというバイトコード命令が 1 命令だけ出力さ

```
    public class IdleThread extends Thread{
      public IdleThread(){
        super("IdleThread");
      }

      public void run(){
        int x;
        while(true){
(1)       x = 0;
        }
      }
    }
```

図 7.6: IdleThread の実装

れ、この命令はステップ実行することができない。このため、ここでは無意味な代入を行っている。

- プレーヤがこのスレッドを動作させるために、現在の実装ではレコーダを動作させる前にデバッギを手動で書き換え、以下の 1 行を挿入している。

  `IdleThread it = new IdleThread(); it.start();`

  このため、レコーダ実行時にもこのスレッドは動作し、オーバヘッドの要因となる。

- `IdleThread` スレッドは終了することがないため、デバッギは自ら実行を終えることができない。現在は、実行の終了時に画面に文字を出力するようなデバッギを利用し、手動でレコーダを終了させている。

- この解決法には重大な欠点がある。それは、他のスレッドより先に `IdleThread` が開始するとは限らないという点である。実際には、先に `Thread.start()` を行った方が先に `ThreadStartEvent` を発生しているが、保証されているわけではない。

なお、`IdleThread` クラスはデバッグ時の実行制御の対象とならないので、バイトコード変換によるタイムスタンプ機構の挿入は不要である。

`wait()` の使用 スレッド同士が `wait()` を用いて同期を取る場合、`wait()` を実行するスレッドはステップ実行の途中（native method としての `wait()` の実行の途中）で停止してしまい、スレッド切替の契機となる `StepEvent` を発生しない。

この問題を回避するには、wait() を実行して待機状態に入ったスレッドはそのまま放置して次のスレッドをログの通りに実行し、待機状態のスレッドの wait() が終了したら（そのスレッドの StepEvent が発生したら）一旦実行を止めておく、という措置が必要となる。現段階ではこの機能は実装していないため、wait() を利用するデバッギは扱うことができない。同様に、Thread.sleep() も正常に動作しない。

main **メソッドのブレークポイント停止位置** プレーヤは main メソッドにブレークポイントを設定し、停止するのを待ってから、最初のスレッド切替までのステップ実行を設定するという実装を行っている。この際、ブレークポイントで停止する場所は、必ず main メソッドの 19 バイト目の直前（図 7.7 の (1) の直前）である。これはメソッドの入口おけるタイムスタンプ更新を終了した場所であり、8 ステップ分が既に終了したところと考えられるので、プレーヤは常に、最初のスレッド切替までのステップ数から 8 を引かなければならない。しかし実際には、この 8 という数値を固定させることができない。デバッギによってはソースコードには存在しないステップ実行が現れる場合があり、デバッギに固有の数値を手作業で見つけ出す必要がある。

この現象は、ブレークポイントではなく MethodEntryEvent を利用することで回避できる可能性もあるが、現時点では未検証である。

DestroyJavaVM **スレッドの開始** レコーダで取得したログでは、main スレッドが終了すると ThreadEndEvent が発生する。その直後に無名のスレッドが一時的に実行され、DestroyJavaVM に名前を変えて、その ThreadStartEvent が発生する。しかしプレーヤでログ通りに実行しても、main スレッドの ThreadEndEvent は発生せず、無名スレッドも DestroyJavaVM スレッドも開始されない。このため前述した IdleThread のステップ実行による待機が繰り返され、デバッギの実行が進まなくなってしまう。

現段階の実装ではこの問題を回避するため、main スレッドの最後（main メソッドの最後）で無意味な無限ループを行い、スレッドが終了しないようにしている。

本章で実装を試みたレコーダとプレーヤは、JPDA が本来意図していない利用法をしていると言える。これらの制約はこうした利用法に由来していると考えられる。マルチスレッドプログラムを決定的に動作させてデバッグしたい、という要求は明らかに存在しており、次世代の JPDA ではこれら機能の実装が強く期待される。

7.3.2　デバッギ

前述したような制約を回避できるデバッギの例として BankTest.java（図 7.8）を用意した。これは 2 つの子スレッド、bt1 と bt2 を生成し（図の (4) と (5)）、共有

```
    public static void main(java.lang.String[]);
      Code:
       0:   getstatic       #72; //Field Timestamp.ts:J
       3:   lconst_1
       4:   ladd
       5:   dup2
       6:   putstatic       #72; //Field Timestamp.ts:J
       9:   getstatic       #75; //Field Timestamp.ref:J
       12:  lcmp
       13:  ifne    19
       16:  invokestatic    #78; //Method Timestamp.brake:()V
(1)    19:  new     #5; //class IdleThread
       22:  dup
       23:  invokespecial   #6; //Method IdleThread."<init>":()V
                :
```

図 7.7: main メソッドの先頭部分のバイトコード（javap による出力）

変数 balance にアクセスする。bt1 は balance に 1 ずつ 100 回足していき、bt2 は balance から 1 ずつ 100 回引いていく（図の (3) と (4)）。両方のスレッドが終了したときに balance は 0 となるはずだが、balance にアクセスするとき（図の (1)）に排他制御を意図的に行わないようにしてあるので、途中でスレッド切替が起こると balance は 0 にならない。

main スレッドは子スレッドの終了を待ってから balance を表示する（図の (7)）。このような待ち合わせには Thread.join() を利用するのが普通だが、これは wait() を呼び出してしまうため利用できない。そこで、子スレッドの自身の実行が終わったという記録を別の共有変数 done に残し（図の (2)）、main スレッドはそれを監視する（図の (6)）ことで子スレッドの終了を知る。

このデバッギは、通常実行時には幸運にも balance は常に 0 となるが、レコーダで動作させると概ね 1/2 程度の確率で 0 とはならない。0 とならない場合のログを用いてプレーヤを動作させ、プレーヤによる明示的なスレッド切替の記録をログと対比して、レコーダとプレーヤが正常に動作することを確かめた。

```
    class BankTest extends Thread{
      static int balance;
      static int done;

      final int unit;

      BankTest(String name, int unit){
        super(name);
        this.unit = unit;
      }

      private synchronized static void done(){
        done++;
      }

      public void run(){
        for(int i = 0; i < 100; i++){
(1)       balance += unit;
        }
(2)     done();
      }

      public static void main(String[] av){
(3)     IdleThread it = new IdleThread(); it.start();
(4)     BankTest bt1 = new BankTest("bt1", 1);
(5)     BankTest bt2 = new BankTest("bt2", -1);
        bt1.start();
        bt2.start();
        while(true){
(6)       if(done == 2) break;
        }
(7)     System.out.println(balance);
      }
    }
```

図 7.8: BankTest.java

表 7.1: レコーダとプレーヤのオーバヘッド（BankTest.java）

通常実行（JIT 有効）	1.2×10^{-3}
通常実行（JIT 無効）	1.2×10^{-3}
レコーダ（JIT 無効）	1.8×10^{0}
プレーヤ（JIT 無効）	1.7×10^{1}

単位は秒

表 7.2: スレッド切替再生時のデバッギ（BankTest.java）とログの諸元

実行終了時のタイムスタンプ値	466
ログのサイズ	84924 バイト
ログの行数	1267 行
プレーヤのスレッド切替の回数	200 回
プレーヤの実行バイトコード数	38411 命令

7.4 オーバヘッド

レコーダとプレーヤのオーバヘッドを計測した。Pentium-III 733MHz、640MB メモリ、Linux-2.4.7、JDK-1.4.2 上で動作させ、ストップウォッチによる手動計測を行った。計測の結果を表 7.1 に示す。

2 種類の通常実行は実行時間が極めて短いため、同一の処理を 10000 回繰り返し、測定結果を 10000 で割ることによって表の数値を得た。従って、VM の起動時間（0.5 秒程度）はこれらの結果にはほとんど含まれていない。レコーダはログを`/dev/null`に出力する設定で、1 回分の実行時間を測定した。プレーヤは、レコーダで取得した特定のログを用いてスレッド切替の再生を行った。その際のデバッギとログに関する情報を表 7.2 に示す。

2 種類の通常実行とレコーダの動作時には`IdleThread`にたびたび制御が移るが、プレーヤの動作時には`IdleThread`は新規スレッドの開始を待つ場合にしか動作しない。実際には、`IdleThread`による待ち合わせはほとんど発生しない。

レコーダやプレーヤが動作する時には JIT を動作させることができない。比較のため、通常実行では JIT を無効にした場合についても計測したが、差は現れなかった。これには 2 つの理由が考えられる。第一は、`BankTest.run()`におけるループの回数が少なく、JIT の利点があまり活かされないということ。第二は、スレッドの生成が合計で 20000 回行われており、Linux では 1 スレッドごとにプロセスが生成されるため、このオーバヘッドが十分に大きいということである。

7.5　まとめ

本章では、ポータブルなスレッド 切替の記録と再生に、制限つきながら成功した。現状では、JPDA だけを用いて完全なスレッド 切替を実現することはできず、DejaVu[CS98] のように VM を変更することが必要になる。しかしながら、第 7.3.1 節で述べた制約を受け入れられるデバッギであれば、表 7.1 に示したオーバヘッドで、VM の変更を要しない、ポータブルなスレッド 切替の記録と再生が可能である。特に JVMDI と JDI が併用できるようになれば、プレーヤのオーバヘッドが低減し、実装も簡潔になることが期待される。

デバッグパターンはポータブルに実装することを目指してきた。マルチスレッドプログラムに対するパターンの適用はポータブルに行うことができないが、本章で提示した機能が JPDA に追加されれば、デバッグパターンの応用範囲を拡大していくことが可能である。

第8章　各種の議論

本章では最初に、第 4.2 節で先送りにした実行点の表現形式について議論し、次に関連研究との比較を行う。

8.1　実行点の表現形式

第 4.2 節で、position には他の表現形式も可能であると述べた。本節では他の表現形式と、本研究で採用したタイムスタンプ方式との比較について述べる。

ステップ実行の回数　ステップ実行はデバッガで制御可能な最小単位の実行を実現する。従ってこれを繰り返せば完全な実行系列を得ることができる。第 6 章の 3 つの応用は、この表現形式を用いても実装可能である。Boothe[Boo00] のデバッガはこの方法でステップ実行単位の疑似逆実行を実現している。しかしながら速度低下はおよそ 1.95 倍となっており、これを補うためにチェックポイントを併用している。

デバッガコマンド履歴　デバッガコマンドの履歴を再生すれば、行った実行制御を再現して同一の position に移動できる。この表現形式では position に順序関係がないため、以前の実行で訪問した position に対して現在の position が（実行系列中で）前であるか後ろであるかが分からない。「あの実行状態」に対する時間的な前後関係を把握することは、デバッグ作業では重要なことである。この順序性の欠如のため、3 つの応用のうち、2 分法を実現することができない。

また、1 つの position に対して複数の表現形式が存在することになる。例えば図 8.1 のようなデバッギを考える。`next; next; step` と `break f; cont` は共に (1) の行に到達したときの position を表現する。よって以前の実行で訪問した position と現在の position とが同一であるかどうかを知ることができない。

実行速度に関しては、この表現形式の方が優れる場合も存在する。図 8.2 のようなデバッギを考える。回数が非常に多いループがあるが関数 `f()` の行 (1) に注目したいという場合、コマンド履歴では `break f; cont;` などとなり、興味のないループに関しては通常速度で実行することができる。一方、タイムスタンプ形式では興味ないループであってもタイムスタンプのインクリメントによって実行速度は低下する。コマンド履歴形式のオーバヘッドを見積もることはできないが、長い履歴ほど高いオーバヘッドを発生させることは間違いない。

```
    int i, j, k;

    f(){
(1)   k = 0;
    }

    main(){
      i = 0;
      j = 1;
      f();
    }
```

図 8.1: コマンド履歴形式における同一 position の複数表現の例

```
    int i, j, k;

    f(){
(1)   k = 0;
    }

    main(){
      for(i = 0; i < 100000000; i++){
        j++;
      }
      f();
    }
```

図 8.2: コマンド履歴表現形式のデバッギの例

ブレークポイントのヒット回数 タイムスタンプの本質は重複して実行される location を区別することにある。デバッガはブレークポイントのヒット回数をカウントしており、これを利用すれば特定の location に関してだけは全順序を持った position が定義できる。しかしブレークポイントの設定を実行の途中で行った場合には、location の登場回数を正しくカウントしたことにならないので、再実行時に最初からブレークポイントを設定してもヒット回数が食い違ってしまう。このため、対象とする location には実行に先だってブレークポイントを設定しなくてはならないが、どの location を対象とするかを事前に決めることは難しい。従って、ソースコードの全ての行に予めブレークポイントを設定することで position を表現することになる[1]。これは明らかに現実的な方法ではないし、ステップ実行と同等のオーバヘッドが発生する。また、この方法でも position に順序性がないため 2 分法は実現できない。

以上をまとめると、タイムスタンプによる表現形式には以下のような利点があると言える。

全順序性 全順序性を持ち、任意の position は他の position と時間順（実行系列における登場順）で比較することができる。この性質を利用して 2 分法が実現できる。

uniqueness 1 つの position の表し方は 1 つしか存在しない。position と実行点の印づけは 1 対 1 対応となる。

パフォーマンス 多くの場合、最も低いオーバヘッドで position を実現する。

8.2 逆実行

本節では多数の逆実行の研究の中で、本研究と関連が深いと思われるものとの比較を行う。

8.2.1 状態保存方式

Feldman ら [FB88] や Moher[Moh88]、Wilson ら [WM89] はメモリの完全な履歴を保存することで、プログラムの任意の実行状態にアクセスできるようにしている。これらのシステムは巨大なログを扱わなければならない。本研究の方式ではデバッギの再現性を仮定すれば、行番号とタイムスタンプ値の組を保存するだけで同等の機能を提供できる。また、この仕組みは本研究との併用が可能であり、タイムスタンプ方式のオーバヘッドを抑制することもできる。

[1] 実際には空行やコメント行、変数の宣言など、デバッガでステップ実行の対象とならない行にはブレークポイントは不要である。

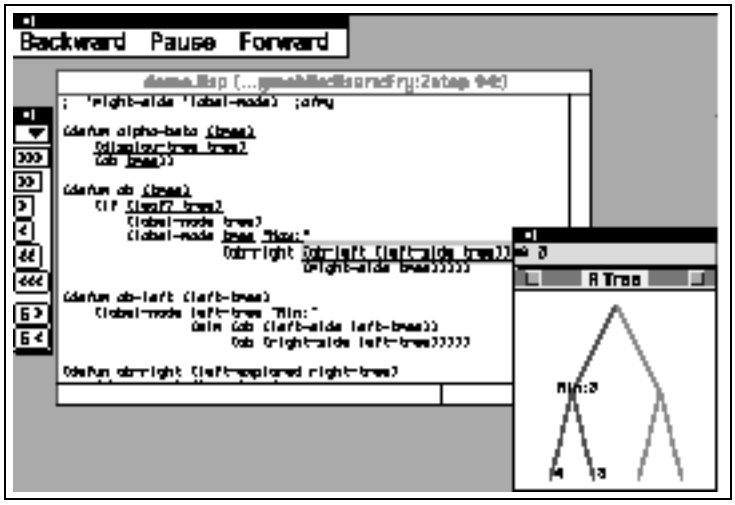

図 8.3: ZStep95[LF97]

表 8.1: inverse statement の例 [BM99]

Original Statement	Inverse Statement
`a += b;`	`a -= b;`
`i++;`	`--i;`
`b += c++ + 1;`	`b -= --c - 1;`

Lieberman ら [LF97, Lie87] は ZStep95 を開発した。これは Lisp 用の、逆実行可能なアニメーション機能つきのソースコード ステッパーである。そのインタプリタは完全な逆実行のために、Lisp プログラムの S 式の評価順序を保存している。また、あるグラフィック出力をクリックすると、それを出力した position に制御を移動する機構も持っている（図 8.3）。本研究のシステムも、グラフィック出力が行われる際にそのオブジェクトと position を結びつけて保存しておくことで同様の機能を提供することができる。タイムスタンプ機構は複数の言語に対応でき、小さいオーバヘッドで実行できるが、ZStep95 のインタプリタは Lisp のサブセットしかサポートしておらず、動作も非常に遅いとされている [LF97]。

Biswas ら [BM99] は、逆演算を行う inverse statement を用いることで静的に推測できる情報は保存を行わないようにしている。inverse statement の例を表 8.1 に示す。inverse statement では、`a = b;` のようなステートメントでは元の値を復元する逆演算を定義できないので、変数の値を保存せざるを得ない。ソースコードがないライブラリ関数の呼び出しが起こる時は、どの変数の値が変更されるかが分からないので、全ての変数を保存するとしている。また、条件分岐やループなどで制御が不連続にジャンプするときには、行番号を履歴として保存する。文献中には保存される履歴の量に関する記述はないが、実用的なほどに小さくなるとは考えられない。

表 8.2: Boothe の方法での順方向コマンドのオーバヘッド [Boo00]

command	slowdown
step	1.95
continue	1.95
next	2.48
finish	2.33
until	2.34

数値は比

8.2.2 疑似逆実行方式

Boothe[Boo00] はステップ実行のカウンタと再実行を用いて、疑似逆実行する機能を持った C デバッガを開発した。Boothe は完全な reversible debugger の実現を目指したので、直前のステートメントに容易に「逆向きステップ」するためにステップ実行のカウンタを必要としたが、position の目的はプログラム実行系列中の 1 点を特定することなので、ステップ実行カウンタはオーバスペックである。また、ステップ実行カウンタはデバッギの完全な実行系列をカウンタの形で保持していると考えられ、動的な情報だけで構成されている。本研究で提案するところの position は動的な情報であるカウンタの値だけを用いるのではなく、静的な情報である location、すなわち (file:line) と組み合わせることにより、低い更新頻度を実現したと言える。実際、Boothe の方法では、最もオーバヘッドの少ない順方向のステップ実行でも 2 倍近くの時間を要している（表 8.2)。こうした高いオーバヘッドを抑えるため、状態保存方式を併用している。

8.3 イベント駆動型実行制御

実行制御をソースコードのステートメントを単位として行うのではなく、代入や関数呼び出し、ループなどをイベントとして扱い、イベントを単位として実行制御を行う方法がある。この実行制御を利用したデバッガである Coca[Duc99] については第 2.5.2 節で述べたのでここでは繰り返さない。

イベント駆動型実行制御を行う際には、代入やループの実行などを識別するために本来のコードに追加コードを挿入するのが普通である。このような目的で行うコードの追加を instrumentation という。本研究では第 5 章で述べたように、C ではコンパイラ中間コードで、Java ではバイトコード（クラスファイル）で instrumentation code を挿入した。ここではその他の挿入方法を採用する研究について述べる。

Templer ら [TJ98, JZTB98] はイベント型の instrumentation tool である CCI を開発した。これはデバッグやプログラム理解、パフォーマンスの向上のために用いられる実行モニタリングシステムで、C のソースコードを変換することで実現する。

```
Before Instrumentation:
a = b + c;

After Instrumentation:
(EV(E_Seti,_Tcci_1=EV(E_Refi,&a)),
 EV(E_Assigni,*_Tcci_1=((_Tcci_0=(((E_Refi,&b),(b))+
                                  (EV(E_Refi,&c),(c))),
                          EV(E_Addi,_Tcci_0),_Tcci_0))),
 *_Tcci_2);
```

図 8.4: CCI における instrumentation code の例 [TJ98]

変換によってコードは図 8.4 のようになる。E_Assigni や E_Addi は加算や代入のイベントを表し、計算の経過をこれらのイベントに渡すため、一時変数_Tcci_に代入している。ソースコードを対象とするため変換されたコードはプラットフォーム非依存であるが、実行速度は laplace.c の場合で 2.09 倍、life.c の場合で 5.85 倍となる [JZTB98]。図中の例のように代入や加算も独立したイベントとして扱うためで、position システムを実現するのであれば制御フローに関するイベントだけを生成するように CCI の設定を変更すればよい。本研究では既存のソースレベルデバッガを用いてきたプログラマが容易に移行できるように、プログラマが目にするソースコードは元のままであることを重視したため、ソースコードを変換の対象としなかった。

Larus ら [LS95] はデバッギを解析、変更するツールのためのライブラリである EEL を作成した。EEL を用いれば、タイムスタンプ機構を実行コードレベルで挿入することもできる。しかしながら、この方法では特定のプラットフォームに依存することになるので、本研究では中間コードレベルを選択して GCC を変更した。

8.4 Software Instruction Counter

Mellor-Crummey ら [MCL89] は backward branch と function call で更新されるカウンタ（SIC）を提案し、プロセッサのプログラムカウンタを用いて (PC, SIC) として、本研究の実行点と同等のものを表せるとしている。また、hardware support がない環境でのウォッチポイントと疑似逆実行の実現方法について述べている。しかし、SIC は実行系列中にまばらな目盛りをつけるためだけに使われており、関数呼び出しの出口で更新されないため実行点の表現として利用することはできない。

8.5 Delta Debugging

プログラマが行う作業を自動化するものとして、Delta Debugging[HZ00, Zel02]がある。これについては第 2.5.3 節で述べたので、ここでは繰り返さない。

8.6 Occurrences

Bertot[Ber91] は lazy *lambda*-calculus の中に “occurrences” を導入した。lazy *lambda*-calculus では reduction の過程で部分木のコピーを生成していく。例えば、ある式 e が lambda 関数 $\lambda x.x + x$ に適用されるとすると、e のコピーが 2 つ作られて、$+$ の 2 つの被演算子に使われる。

このことは、プログラムの実行時に単一の location に対応する複数の position が（実行系列の中に）生成されることと同じと考えられる。Bertot は、ある式にブレークポイントを設定したとき、その式のコピーが評価された場合にも実行が停止するような機能を開発した。

手続き型言語では、このような location と position の識別は極めて容易である。なぜならば、ある命令が配置されたメモリ上の番地がブレークポイントとして利用されるからである。Bertot の目的は複数の position を（元となる）単一の location に帰着させることである。これに対して、本研究で提示したタイムスタンプ方式のプログラム実行点の目的は、複数の position に分散してしまう単一の location を区別することである。

8.7 アスペクト指向プログラミング

オブジェクト指向プログラミングでは、あるデータ型のオブジェクトにアクセスするコードを、そのデータ型の定義と共に書く。これによってアクセスするコードの分散を防ぎ、保守性を向上させている。

アスペクト指向プログラミング [KLM+97] では、本質的に分散せざるを得ないコードをアスペクトとして定義することにより、コードを 1ヶ所に集約することができる。例えば、メソッドの出入りに関するログを出力しようとすると、そのためのコードはあらゆるメソッドに分散してしまう。そこで、そのログ出力のためのアスペクトを定義し、ログ出力コードはアスペクト定義部にだけ記述して、各メソッドには記述しない。そしてアスペクトの処理系を用いて、各メソッドの出入口に自動的にログ出力コードを挿入する。Java のためのアスペクト処理系の 1 つ、AspectJ[KHH+01] では、`ajc` というコンパイラで Java ソースとアスペクトの定義をコンパイルする。

第 5 章で述べたタイムスタンプ機構の挿入は、プログラム実行点に関するアスペクトを定義して、C と Java に対して異なる方法で実現したと考えることができる。AspectJ ではループの際の制御ジャンプを指定することができないので、現段階で

は Java への実装に利用できない。

第9章　結論

本研究では、ソフトウェア開発における最終的な誤り修正工程であるデバッグ工程の中で、プログラマが行っているデバッグ作業の分析を行い、プログラマの行動が「仮説検証ループ」によって説明できることを示した。このループにある検証方法とその実施には典型的なパターンがあり、これをデバッグパターンと名付けて自動化することを目指した。

一般的なデバッグ方法である cyclic debugging においては、プログラマは往々にして少し前の場所に戻りたいと考えるものであり、逆実行と呼ばれる研究領域がある。これもデバッグパターンの一種として位置付けられる。そこで研究の初期段階として、既に提案されていた関数単位疑似逆実行をより高速化した。

これ自身は有用であるが、より多くのデバッグパターンの自動化に応用できる、汎用の基盤技術をポータブルかつ高速に実装することが必要となった。その基盤としては、プログラマがデバッグ中に持っているプログラム実行点（position）という概念がふさわしい。この表現形式としてタイムスタンプ型を選択し、C ではコンパイラ中間コードでのプログラム変換を、Java ではバイトコードでのプログラム変換を用いて実装した。position を利用して自動化されるデバッグパターンとして、「実行点の印づけ」と「逆向きウォッチポイント」、「実行系列中の 2 分探索」について述べた。

以上の議論はデバッギの再現性を仮定したものであった。本質的に再現性を持たないデバッギである Java のマルチスレッドプログラムを対象として前述の自動化を実現するため、スレッド切替を記録、再生することで再現性を持たせる方法を検討して実装し、小さなアプリケーションでその再現性を確認した。

以下では、本研究の意義と展望について述べる。

9.1　本研究の意義

第 2 章では、デバッグは仮説を立ててそれを検証するという仮説・検証サイクルによって成り立っていると述べた。そこで、デバッグ作業を、第一段階としての仮説設定フェーズと、第二段階としての仮説検証フェーズに分けて考える。効率的なデバッグとは定められた時間の中でこの二段階サイクルの回転数を上げることであると言える。

仮説設定フェーズはそのバグに固有の情報を検討する必要があり、パターンは存

在するものの、それを発見して適用する過程では人間の知性が不可欠である。一方、仮説検証フェーズにはよりはっきりしたパターンが存在する。例えば逆向きウォッチポイントのように、最後に代入したオペレーションを見つけ出す、といったものである。これまでこの2つが分離して考えられて来なかったのは、デバッグの結果（バグの原因、すなわちデバッグ時の仮説）のみが示され、その過程については注目されてこなかったからである。しかしながら、その過程にはプログラマの知識と経験が豊富に含まれている。

本研究は誤りを修正するというデバッグ作業の過程に注目した。これにより、熟練のプログラマが行うデバッグ過程を自動化して作業時間を短縮することで仮説・検証サイクルの回転数を上げ、デバッグ効率を向上させるという直接的な効果を得た。また、今までのデバッグ作業はその場限りのものであったが、デバッグ作業の再利用も可能となった。のみならず、熟練プログラマの技術を自動化という形で浮き彫りにすることで、それを他人、特に経験の浅いプログラマに供することができるようになり、知識や経験を共有する基盤を提供することに成功したと言える。

オブジェクト指向言語の設計では、クラスの用途に応じてどのような階層化と分離を行うべきかというノウハウがプログラマの中に蓄積されていた。これを体系づけて形にしたものがデザインパターンである。本研究で抽出したデバッグ作業の過程もプログラマの中に蓄積されていたものであり、逆向きウォッチポイントなどはデバッグパターンと呼ぶことができる。デザインパターンが今日広くプログラマに利用されているように、デバッグパターンも幅広いプログラマが利用可能なものである。ソフトウェア開発の現場にいる職業プログラマは前述の熟練プログラマであり、彼らにとっては労力の削減という恩恵だけでなく、作業時間の短縮によって次々と頭に浮かぶ様々な仮説を検証する機会を得ることができる。趣味としてフリーウェアを開発するような日曜プログラマとでも呼べる人々にとっては、作業時間の短縮だけでなく、より上級者が使う高度なデバッグパターンを習得できる。教養課程の学生のような入門教育を受けているプログラマにあっては、デバッガの繁雑な操作に振り回されることなく、デバッグ作業の本質を学びとることを可能にする。

9.2 展望

デザインパターンでは23のパターンが示されている[GHJV95]。デバッグパターンを有用な技術として確立するためには、何よりも多数のパターンを収集することが重要である。デバッグ作業をしているプログラマが自らの作業記録を残していくのが最善だが、多くの場合は時間に追われており、そのような余裕はないと考えられる。そこで、パターン収集のために、XPにおけるペアプログラミングをデバッグにも応用してペアデバッギングを行うのが有効だと思われる。すなわち、デバッグ作業を主に行うプログラマと、その作業の本質を確認して記録に残していくプログラマの2人をペアにする。ペアプログラミングの有効性と同じ理由で、デバッグ作

業そのものを改善することも期待できる。

デバッグパターンの収集と再利用を通じてデバッグ作業を分析するという動きが広がれば、作業の第一段階である仮説の立案も研究の対象となることが期待される。デバッグパターンのような自動化をすることはできないと思われるが、場当たり的なものとして考えられがちなデバッグ工程を分析することを通じて、従来のソフトウェア工学のアプローチとは違う側面から、できる限りバグを回避するコーディングが促進されることを期待する。

参考文献

[AH90] Hiralal Agrawal and Joseph R. Horgan. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 246–256, 1990.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 1999. (長瀬訳：XP エクストリーム・プログラミング入門, ピアソン・エデュケーション (2000)).

[Ber91] Yves Bertot. Occurences in Debugger Specifications. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 327–337, 1991.

[BG96] David W. Binkley and Keith Brian Gallagher. Program Slicing. *Advances in Computers*, Vol. 43, pp. 1–50, 1996.

[BM99] Bitan Biswas and R. Mall. Reverse Execution of Programs. *SIGPLAN Notices*, Vol. 34, No. 4, pp. 61–69, 1999.

[Boo00] Bob Boothe. Efficient Algorithms for Bidirectional Debugging. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pp. 299–310, 2000.

[CS98] Jong-Deok Choi and Harini Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pp. 48–59, 1998.

[Dah99] Markus Dahm. Byte Code Engineering. In *Java-Information-Tage 1999(JIT'99)*, 1999.
http://bcel.sourceforge.net/.

[Duc99] Mireille Ducassé. Coca: An Automated Debugger for C. In *Proceedings of the 1999 International Conference on Software Engineering*, pp. 504–513, 1999.

[End03] 遠藤知宏. 動くべくして動く. *UNIX MAGAZINE*, Vol. 18, No. 10, pp. 106–108, 2003. (連載：プログラマーの理想と現実 第3回).

[FB88] Stuart I. Feldman and Channing B. Brown. IGOR: A System for Program Debugging via Reversible Execution. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and distributed debugging*, pp. 112–123, 1988.

[GCR97] Susan Gerhart, Dan Craigen, and Ted Ralston. Observations on Industrial Practice Using Formal Methods. In *Proceedings of the 15th International Conference on Software Engineering*, pp. 24–33, 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1995. (本位田・吉田監訳：オブジェクト指向における再利用のためのデザインパターン, ソフトバンク (1995)).

[HZ00] Ralf Hildebrandt and Andreas Zeller. Simplifying Failure-Inducing Input. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA2000)*, pp. 135–145, 2000.

[Inc92] D. C. Ince. *An Introduction to Discrete Mathematics, Formal System Specification, and Z, Second Edition*. Oxford University Press Inc., 1992.

[Ish86] 石井康雄（編）. ソフトウェアの製造, 日科技連ソフトウェア品質管理シリーズ, 第3巻. 日科技連, 1986.

[JZTB98] Clinton L. Jeffery, Wenyi Zhou, Kevin Templer, and Michael Brazell. A Lightweight Architecture for Program Execution Monitoring. In *Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering*, pp. 67–74, 1998.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. *Lecture Notes in Computer Science*, Vol. 2072, pp. 327–355, 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pp. 220–242, 1997.

[KP99] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison Wesley Longman, Inc., 1999. (福崎訳：プログラミング作法, アスキー出版局, (2000)).

[LF97] Henry Lieberman and Christopher Fry. ZStep 95: A Reversible, Animated Source Code Stepper. In Blaine Price John Stasko, John Domingue and Marc Brown, editors, *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1997.

[Lie87] Henry Lieberman. Reversible Object-Oriented Interpreters. *First European Conference on Object-Oriented Programming*, 1987.

[Liu] Liu Die Yu. *Unpatched Internet Explorer Bugs*. http://www.safecenter.net/UMBRELLAWEBV4/ie_unpatched/index.html.

[LS95] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 291–300, 1995.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification (2nd Edition)*. Addison-Wesley Pub. Co., 1999.

[McC93] Steve McConnell. *Code Complete*. Microsoft Press, 1993. (石川訳：コードコンプリート–完全なプログラミングを目指して–, アスキー出版局, (1994)).

[MCL89] J. M. Mellor-Crummey and T. J. LeBlanc. A Software Instruction Counter. In *Proceedings of the third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 78–86, 1989.

[Moh88] Thomas G. Moher. PROVIDE: A Process Visualization and Debugging Environment. *IEEE Transactions on Software Engineering*, Vol. 14, No. 6, pp. 849–857, 1988.

[Moz] The Mozilla Organization. *Bugzilla Project*. http://www.bugzilla.org/.

[MT00a] 丸山一貴, 寺田実. タイムスタンプを用いたプログラム実行点のポジショニング. 情報処理学会第 61 回全国大会講演論文集, 2000.

[MT00b] 丸山一貴, 寺田実. 関数単位疑似逆実行の高速化. 情報処理学会論文誌 プログラミング, Vol. 41, No. SIG9(PRO8), pp. 1–7, 2000.

[MT03a] Kazutaka Maruyama and Minoru Terada. Debugging with Reverse Watchpoint. In *Proceedings of the Third International Conference on Quality Software (QSIC2003)*, pp. 116–123, 2003.

[MT03b] Kazutaka Maruyama and Minoru Terada. Timestamp Based Execution Control for C and Java Programs. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG2003)*, pp. 87–102, 2003.

[RBdK00] Michiel Ronsse, Koen De Bosschere, and Jacques Chassin de Kergommeaux. Execution replay and debugging. In *Proceedings of the International Workshop on Automated Debugging (AADEBUG2000)*, pp. 5–18, 2000.

[Ros97] Jonathan B. Rosenberg. *How Debuggers Work*. John Wiley & Sons, Inc., 1997. (吉川訳：デバッガの理論と実装, アスキー出版局, (1998)).

[SPS+00] Richard Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with GDB – The GNU Source-Level Debugger – Eighth Edition*, 2000. (included in GDB source tree).

[Sta] The Standard Performance Evaluation Corporation. *SPEC JVM98 Benchmarks*.
http://www.spec.org/osg/jvm98/.

[Sta99] Richard M. Stallman. *Using and Porting the GNU Compiler Collection*. Free Software Foundation, 1999. (included in GCC source tree).

[Suna] Sun Microsystems, Inc. *Bug Parade*.
http://developer.java.sun.com/developer/bugParade/index.jshtml.

[Sunb] Sun Microsystems, Inc. *Java Platform Debugger Architecture*.
http://java.sun.com/products/jpda/.

[TA95] Andrew Tolmach and Andrew W. Appel. A Debugger for Standard ML. *Journal of Functional Programming*, Vol. 5, No. 2, pp. 155–200, 1995.

[Ter99] 寺田実. 気くばりデバッガ. 中島秀之（編）, 秋のプログラミングシンポジウム「日本のプログラミング」報告集, pp. 45–52. 情報処理学会, 1999.

[Ter00] 寺田実. デバッガのためのプログラム疑似逆実行方式. 情報処理学会論文誌, Vol. 41, No. 7, pp. 1956–1963, 2000.

[Tho] Thor Larholm. *Unpatched IE security holes*.
http://www.pivx.com/larholm/unpatched/
（現在は閉鎖）.

[TJ98] Kevin S. Templer and Clinton L. Jeffery. A Configurable Automatic Instrumentation Tool for ANSI C. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, pp. 249–257, 1998.

[Ver98] Marc Vertes. Xlab – a tool to automate graphical user interfaces. In *Linux Weekly News*, May 1998.
http://mvertes.free.fr/.

[WM89] Paul R. Wilson and Thomas G. Moher. Demonic Memory for Process Histories. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 330–343, 1989.

[Zel02] Andreas Zeller. Isolating Cause-Effect Chains from Computer Programs. In *Proceedings of the SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE-10)*, pp. 1–10, 2002.

謝辞

指導教官の廣瀬通孝教授には、研究全体を見通す立場から重要なご助言を賜わりました。心よりお礼申し上げます。

電気通信大学の寺田実助教授には、学部 4 年から博士 2 年までの間、指導教官としてご指導いただき、本論文に関する様々なアイデアやご指摘をいただきました。深く感謝致します。

University of Texas at El Paso の Nigel Ward 助教授には、寺田助教授とは異なる視点から貴重なご指摘、ご批評をいただき、大変有難く、感謝致します。

大津・國吉研究室の永井おりが技官には、研究における事務手続き等にご尽力いただき、有難うございました。

また、東京大学の近山隆教授、稲葉雅幸教授、田中哲朗助教授、広田光一助教授には、予備審査において有益なご指摘を多数賜わり、論文の改良に役立てることができました。

東京大学 工学部 機械情報工学科 寺田研 卒業生の東京大学インテリジェント・モデリング・ラボラトリーの立山義祐氏、電気通信大学の塚原渉氏、株式会社日立製作所の小林広和氏には、研究室輪講や勉強会などでご指導、ご意見をいただきました。

同卒業生の首藤達生君、樋口直志君、須山哲宏君、副田俊介君、西本圭介君、林芳樹君には、研究を進めていく上で率直なご意見と惜しみないご協力をいただきました。

電気通信大学 電気通信学部 情報通信工学科 寺田研 学部生の五十嵐知久君には、ソフトウェア開発の現状についての知見を教えていただき、研究に役立てることができました。同学部生の高木一人君には、貴重なバグの例を供出していただきました。

以上の皆様のおかげで、本研究を進めることができました。ここに謝意を表したいと思います。

最後に、ここにお名前を挙げきれなかった研究室の皆様と、情報工学輪講や情報処理学会プログラミング研究会など、研究発表の場でご助言を下さった皆様に、心よりお礼申し上げます。

発表文献一覧

論文誌（査読あり）

- 丸山 一貴, 寺田 実, “関数単位疑似逆実行の高速化”, 情報処理学会論文誌 プログラミング, Vol.41, No.SIG9(PRO8), pp.1–7, 2000.
- 丸山 一貴, 寺田 実, “クラスファイル変換による Java プログラムの実行制御”, 情報処理学会論文誌 プログラミング, Vol.44, No.SIG13(PRO18), pp.38–46, 2003.

国際会議（登壇発表・査読あり）

- Kazutaka Maruyama, Minoru Terada, “Timestamp Based Execution Control for C and Java Programs”, Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG2003), pp.87–102, Ghent, Belgium, September 8–10, 2003.
- Kazutaka Maruyama, Minoru Terada, “Debugging with Reverse Watchpoint”, Proceedings of the Third International Conference on Quality Software (QSIC2003), pp.116–123, Dallas, Texas, USA, November 6–7, 2003.

国際会議（ポスター発表・査読あり）

- Kazutaka Maruyama, Minoru Terada, “Position Specification in Program Traces and Potential Applications in Semi-automated Debugging”, Student Research Forum, the ACM SIGPLAN ’01 Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20–22, 2001.

国内口頭発表（査読なし）

- 丸山 一貴, 寺田 実, “タイムスタンプを用いたプログラム実行点のポジショニング”, 情報処理学会 第 61 回全国大会, 愛媛大学, 2000 年 10 月 3–5 日.