論文の内容の要旨

論文題目　　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
氏　　名　　丸山　一貴

1. 序論

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り、デバッグを確実に行うことは極めて重要である。

デバッグ工程での労力のほとんどは、個別のバグを修正するデバッグ作業に割かれるが、デバッグ作業における大きな問題は、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。このため、現在のデバッグ作業には 2 つの欠点がある。第一は、その場限りの作業であるため、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こと。第二は、ケースバイケースで対処するため、自動化の余地がないこと。

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。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。

デバッグパターンを定義するため、デバッグ作業の中のバグの原因を発見する手続きを説明するものとして「仮説検証ループ」を提案し、その中の「検証方法の選択」と「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。また、このループによって、デバッグの自動化に関連する他のいくつかの研究もデバッグパターンの一種として解釈できる。

以上より、本論文の目的は、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である。

2. プログラム実行点

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おり、通常、デバッグ中のプログラマはこれを意識している。そこで、本研究では実行点を行番号とタイムスタンプの組として表現することを提案する。タイムスタンプとは、制御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であり、デバッギ（デバッグ対象プログラム）の変数として追加される。

以下のようなコードでループ本体（2 行目）が 3 回実行される場合、本体の 1 回目の実行点は（test.c:2, 0）であり、2 回目は（test.c:2, 1)、3 回目は（test.c:2, 2）となる。

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

一般には、このループ構造に到達したときにタイムスタンプの値がいくつになっているかは分からないので、プログラマが直接指定することはできない。タイムスタンプの値を含む実行点の情報は、デバッグ中に行った以前の実行で取得済みでなければならない。この意味で、デバッギは再現性を持つという前提がある。デバッガを用いたデバッグでは、デバッギの再実行を繰り返しながらバグを追及していくことが基本であり、これを cyclic debugging と呼ぶ。cyclic debugging ではデバッギの再現性は通常の前提である。

3. タイムスタンプ機構の実装 － C

タイムスタンプ更新機構を C のデバッギに自動的に組み込む方法について述べる。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(タイムスタンプ更新コード)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い、実装の対象は GCC-2.95.2 とした。

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
・goto 文によるジャンプ

タイムスタンプ更新コードにより、実行速度の低下とコードサイズの増大が発生する。これらのオーバヘッドを、sed と awk をベンチマークとして計測したところ、実行速度は約 1.6 倍の速度低下、ファイルサイズはおよそ 1.1 倍から 1.4 倍の増大であった。これらは実用上、問題とはならないオーバヘッドであるといえる。

4. タイムスタンプ機構の実装 － Java

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 メンバーとした。タイムスタンプ更新の対象となるバイトコードは、以下の通りである。

・[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 バイトコード群が該当。
・[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、関係する 16 命令を対象に含める。
・[無条件ジャンプ]
goto と goto_w が対象となる。
・[その他の制御ジャンプ]
jsr と ret、jsr_w が該当。
・[例外]
catch ブロックの先頭に挿入する。

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 を用いて Java で記述した。プログラマは、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。

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、SPEC JVM98 のベンチマークプログラムを用いて計測した。実行速度は約 1.3 倍、ファイルサイズはおよそ 1.2 から 1.3 倍となった。C の場合と同様、実用上の問題はないと考えられる。

5. 実行点の応用 ― デバッグパターンの例

実行点を用いることで、デバッグ作業中にプログラマが手動で行っていた作業を自動化することができる。これは前述のデバッグパターンの自動化であり、ここでは 3 つのパターンを紹介する。これらは 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。

第一は「実行点の印づけ」である。大規模なプログラムの中をデバッガで移動していると、うっかりコマンドを間違って、あるいはバグの原因となる場所の推測を誤っていて、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、デバッギの振舞が「ここまでは正しい」というときにその実行点を記録しておく（印づけをする）ことが有効である。

第二は「逆向きウォッチポイント」である。C プログラムにおける配列のバッファオーバランや、マルチスレッドプログラムにおける共有変数への誤った代入によるバグを追及す

るには、変数への書き込みを捕捉するウォッチポイントを用いる。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。これを自動化する機構が逆向きウォッチポイントである。

第三は「タイムスタンプを用いた実行系列中の 2 分探索」である。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が、これは手動では到底できないような手続きを実行する例である。デバッギは正しく動作していないが、原因について全く手がかりがないというとき、プログラマはデバッギの実行をある程度まで進めて不正になっているかどうかを調べる。もし正しければさらに実行を進め、そこでまた調べる。正しくなければデバッギの実行を最初からやり直して少し手前で停止させ、そこでまた調べる、ということを繰り返す。これを自動的に行うパターンがこの 2 分探索である。

6. マルチスレッドプログラム

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が、再現性を持たないデバッギに対しても、適切な記録、再生機構によって再現性を持たせることで前述のパターンを適用することができる。マルチスレッドプログラムの場合には、スレッド切替のタイミングを記録して再生すれば良い。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、再生機構を実装し、オーバヘッドを計測した。

7. 結論

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、熟練のプログラマが行うデバッグ過程を自動化して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、熟練プログラマのデバッグ技術を抽出することで、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。

設計におけ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。職業プログラマは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。日曜プログラマとでも呼べる人々は、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。入門教育を受けているプログラマは、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が可能となる。