社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.
SS99-52 (2000-01)

バージョン間の差分を利用したデバグ手法の提案

寺口 正義† 松下 誠† 井上 克郎†‡

† 大阪大学大学院基礎工学研究科
〒 560-8531 大阪府豊中市待兼山町 1-3
06-6850-6571

‡ 奈良先端科学技術大学院大学情報科学研究科
〒 630-0101 奈良県生駒市高山町 8916-5
0743-72-5314

Email: {teraguti,matusita,inoue}@ics.es.osaka-u.ac.jp

あらまし 正常に機能することが事前にわかっているバージョンと，欠陥が含まれるバージョンとの差分に着目して自動的にテストを行うことで，欠陥の原因を特定する研究が行われている．しかし，テストの度にソースプログラムから実行可能ファイルを作成する必要があるため，非常に多くのテスト時間が必要となっていた．また，テスト作業にのみ重点が置かれており，デバグ作業まで考慮されておらず，実用的とはいえなかった．本研究では，従来よりもテスト実行時間を減らし，テストからデバグまでの一連の流れを支援することで実際のソフトウェア保守に利用可能なデバグ手法の提案を行う．本手法によって，ソフトウェア保守においてデバグ作業をより行いやすくなることが期待できる．

キーワード バージョン管理，差分，デバグ

A Proposal of Debugging Method with Changes between Versions

Masayoshi Teraguchi† Makoto Matsushita† Katsuro Inoue†‡

† Graduate School of Engineering Science,
Osaka University
1-3 Machikaneyama, Toyonaka,
Osaka, 560-8531, Japan
+81 6 6850 6571

‡ Graduate School of Information Science,
Nara Institute of Science and Technology
8916-5, Takayama, Ikoma,
Nara, 630-0101, Japan
+81 743 72 5314

Email: {teraguti,matusita,inoue}@ics.es.osaka-u.ac.jp

Abstract Researches on identification of the cause of bugs have emerged, with automatic testing which notices the delta between the version which we know in advance to operate correctly and the version that contain bugs. However, whenever testing, executable program is created by source program. It takes much time to test. And since these put emphasis on only testing, not debugging, these are not practical in software maintenance. In this paper, we propose a debugging method which applies to practical software maintenance, for reducing execution time than usual and supporting a series of transactions from testing to debugging. With this method, debugging activities can be done more easily in software maintenace.

key words Version Management, Delta, Debugging

1 まえがき

ソフトウェアの保守とは，本稼働中のソフトウェアの運用の継続を可能にするため，あるいはそれらを改善するための工程である [3]．ソフトウェアにかかる全費用のうち保守の占める割合は約 80%にものぼり [4]，ソフトウェアに携わる人が費やす総時間の約 65%が，保守やそれに関連する作業に費やされている [2]．このことから，ソフトウェア保守の重要性を認識できる．ソフトウェア保守の本質を理解するために，Swanson はソフトウェア保守が必要となる要因を 3 つの基本的な種類に分類している [14]．

(1) ソフトウェア中のエラーに起因する欠陥
(2) ソフトウェアを取り巻く環境の変化
(3) ユーザや保守担当者の要求

さらに，Swanson はこれらの基本的な要因に対応して実行される保守活動を次のように定義している．

(1) 修正保守：欠陥の識別，修正
(2) 適応保守：環境の変化に応じた修正
(3) 完全化保守：性能の改善，機能の変更や追加

一般的にソフトウェア保守活動の中で修正保守が一番多いと考えられがちであるが，1980 年代初めに Lientz と Swanson によって行われた調査によると，修正保守は保守作業の 20%にすぎず，完全化保守が 55%を占めることが報告されている [10]．

これらの保守活動においては，既存のソフトウェアに対する多くの変更が発生する．しかし，ソフトウェアに変更を加える際にエラーを発生させてしまう確率は 50%から 80%の間であることが Hetzel の研究 [7] で示されている．従って，保守活動の 75%を占める修正保守，完全化保守において，ソフトウェアに変更を加えた場合に，変更された部分の機能だけではなく変更されていない部分の機能に関しても動作確認が必要となる．

ソフトウェアに変更を加えた後で，ソフトウェアが仕様通りの性能で動作するかどうかをテストするための一手法として回帰テストがある [6, 12, 9]．回帰テストは実際のソフトウェア保守において活用されており，欠陥の早期発見に役立っている．

しかし，近年のソフトウェア開発の大規模化，複雑化に伴い，ソフトウェア保守もより大規模かつ複雑なものとなっている．そのため，ソフトウェアの品質および生産性を高めるために，ソフトウェア保守活動においても作業を効率良く行うことが必要となる．従来の保守活動において，ソフトウェアに変更を加えるために行われる作業は基本的に次の 3 つである．

(1) ソフトウェアおよびなされるべき変更の理解
(2) システムの変更を実現するためのソフトウェアの変更
(3) 変更後のソフトウェアの動作確認

ソフトウェア保守でソフトウェアおよび行われた修正を把握，理解すれば，発見された欠陥の原因を究明するために役立つ．逆に，ソフトウェアに対する理解が乏しければ，欠陥の原因究明および訂正に大幅な時間がかかる．そこで，ソフトウェアの理解性および保守性を向上させるための一手法としてソフトウェア構成管理やバージョン管理を行う方法がある．

ソフトウェア構成管理とはソフトウェア開発，保守過程で作成されるプロダクトの識別や制御，状態の把握等を解決する研究である [5]．

また，バージョン管理とは開発チームが作成したプロダクト (ソースプログラムや付随する文書) に対する様々な修正を正しく認識し，組織化し，管理することを目的とした研究であり，様々な管理手法のモデルが提案されている．さらに，そのモデルに基づくバージョン管理システムが実装されている [1, 15]．通常プロダクトの改訂は複数回行われ，各改訂毎に作成されるプロダクトをバージョンと呼ぶ．

バージョン管理システムを用いて開発されたソフトウェアの保守段階では，仕様通りの性能で動作するバージョン (以下，基準バージョンと呼ぶ) が存在する．従来のソフトウェア保守では，上述した保守作業の (2)，(3) において，基準バージョンのプログラムへの変更を行い，回帰テストを用いてソフトウェアの動作をテストする．テストで欠陥が発見されれば，その欠陥の原因となるエラーを取り除く (デバグを行う) ために，保守作業の (1)，(2) を繰り返し行う．しかし，変更していない機能に欠陥が見つかった場合，その原因となるエラーを発見することは困難である．

そこで，基準バージョンから現在のバージョンに至るまでの間に行われた修正を用いて自動的にテストを行うことで，欠陥の原因となるエラーを特定するための研究が行われている [11, 16]．しかし，テストを行う度に，プログラムの基準バージョンに変更を加えてコンパイルを行い，実行可能ファイルを作成するため，テストに多大な時間を費やしてしまう．また，欠陥を発見する度に，その欠陥に応じたテストケースを作成する必要があるた

め，テストに利用しているテストツール [8] に関する教育や訓練が必要となる．さらには，欠陥の原因を特定するテスト作業にのみ重点がおかれており，その後のデバグ作業までは考慮されていないため，実際の保守作業にそのままでは適用できない．

本研究では，保守作業において，プログラムに変更を加え，ソフトウェアの動作をテストすることで変更していない部分に欠陥が見つかった場合に，適用可能なデバグ手法を提案する．本手法では，まず実行可能ファイルのバージョンを順に取り出し，テストツール [13] を利用して自動的にテストを行う．テストによって，欠陥を出力するバージョンと正常に出力するバージョンが並ぶ部分を発見すると，発見したバージョン間で行われた修正にエラーがあると考える．次に現在のバージョン上でその時に行われた修正 (以下，差分と呼ぶ) を強調する．最後に欠陥を訂正した後で，訂正で行った変更を古いバージョンに反映することで以降のテストに役立てる．本手法によって，テストからデバグまでの一連の流れを支援することができる．また，実際の保守においてデバグ作業をより行いやすくなることが期待できる．

以降，2 章では，これまでに行われた欠陥の原因特定手法について述べる．3 章では提案するデバグ手法について述べる．4 章において，本手法を評価するための実験方針について述べ，最後に，5 章で本研究のまとめと今後の課題を述べる．

2 欠陥の原因特定に関する研究

本章では，これまでに行われてきた欠陥の原因となるエラーを特定するための手法に関する研究について取り上げ，既存の手法が抱える問題点について述べる．

2.1 Ness と Ngo の研究

Ness と Ngo はグレイ研究所において，コンパイラ開発のために *regression containment* と呼ばれる手法を利用している [11]．Ness らの手法では，まず回帰テストを自動的に行う．この回帰テストが失敗した場合，基準バージョンが正しく動作することに着目して，構成管理から取得した修正を実際の適用順に施しながらテストを繰り返し行う．テストが失敗した段階でその時に適用した修正をバグの原因となるエラーとして特定する．

しかし，Ness らの手法は特定の状況ではうまく動作するが，単一の差分だけではなく複数の差分が適用されることで初めてテストが失敗する場合や，変更の適用によりコンパイルできないといった矛盾が生じる場合にはうまく動作しない．

2.2 Zeller の研究

Zeller は Ness らの問題点である複数のエラーによる欠陥や変更の適用による矛盾にも対応可能な手法を提案している [16]．Zeller の手法では，修正を適用する場合の順序は考慮せず，修正を集合の一要素として捉える．従って，行われた修正の数が n であれば，考えられる集合の数が 2^n となる．Zeller は考えられる集合の中から欠陥が生じる要素数が最小の集合を見つけ出すアルゴリズムを利用する．修正の集合を利用することで，複数のエラーによって引き起こされる欠陥の原因を特定することに成功している．また，変更の適用による矛盾にも対応できるアルゴリズムであり，Ness らの手法では見つけることができなかったエラーも発見することができる．

しかし，アルゴリズムを複雑にすることでバグの原因となる差分を特定する精度は上がるが，修正の数に対して指数的にテストすべき集合の数が増加する．従って，テストの回数が増加し，テストに多大の時間がかかる．

2.3 問題点

Ness らと Zeller のどちらの手法も基準バージョンのソースプログラムを基本として考えており，テストを行うためには毎回そのソースコードに修正を適用して，コンパイルを行わなければならない．ソフトウェアの大規模化に伴い，コンパイルにかかる時間も増大するため，テスト時間の大半をコンパイルが占めることになる．また，欠陥の原因を探るために自分でテストケースを作成してテストを行う必要があり，各々の手法で利用しているテストツール [8] の学習や訓練を行わなければならない．さらには，欠陥の原因を特定するためのテスト作業にのみ重点がおかれており，デバグ作業までは考慮されていないため，実際の保守作業に適用することは難しい．

3 提案するデバグ手法

本章では，2 章で述べた既存の手法の持つ問題点を解決するための，既存の手法を改良したデバグ手法について述べる．

3.1 特徴

ソースプログラムだけではなく実行可能ファイルのバージョンも管理することで，既存の手法ではテスト毎に行っていたコンパイルを行わないようにして，テスト時間の削減を図る．

また，テスト開始からデバグ終了までの時間の短縮を考えた場合，テスト作業に重点をおき原因を突き止めるためのアルゴリズムを複雑化する方

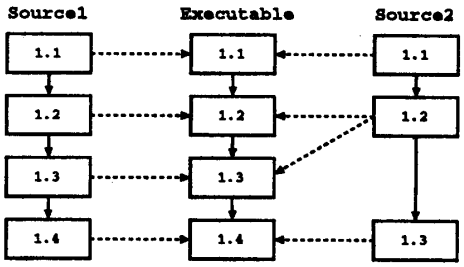

図 1: バージョン間の関連情報

がよいのか，デバグ作業に重点をおきアルゴリズムを単純化する方が良いのかは一概にわからない．本来デバグ作業は人の手によって行われるものであり，デバグまでの一連の流れを支援するという観点から考慮すれば，デバグに重点をおくことが重要であると，我々は考えている．従って，本手法ではテスト時間を短縮し，デバグに重点をおくために，既存の手法を改良した単純なテストアルゴリズムを採用する．

さらに，既存の手法では手作業で行っていたテストケースの作成も，対象とするソフトウェアをある程度限定することで，自動生成できるようにする．これにより，従来の保守作業を少しでも効率化できると考える．

3.2 前提条件

ソフトウェア保守において我々の考えるデバグ手法を利用する際に，ソフトウェアおよびその開発と保守，ソフトウェア保守担当者に必要となる前提条件について述べる．

- バージョン管理システムを利用する．

  バージョン管理システムによってプロダクトに対する様々な修正を管理できるならば，管理されている修正に関する情報を欠陥の原因を特定するために利用できる．バージョン管理の対象としては，ソフトウェアのソースプログラム，実行可能ファイルとする．実行可能ファイルをバージョンとして登録する際には，付随情報として実行可能ファイルのバージョンとソースプログラムのバージョンの関連も記録する (図1参照)．デバグ時にバージョン間の差分を特定するためにこの情報を利用する．

- ソフトウェアに基準バージョンが存在する．

  通常ソフトウェア開発が一通り完了したら，仕様通りに実行される製品としてソフトウェアがリリースされて，ソフトウェアの保守段階に入る．従って，保守段階では，ソフトウェアには必ず基準バージョンが存在すると考えられる．

- ソフトウェアに対する入力は変更されない．

  ソフトウェア保守活動において，ソフトウェアに対する入力が変更される (例えば，入力パラメータが増減する) ことは十分に考えられるが，本手法では考慮しない．

- ソフトウェアは入力に対して何らかの出力を出す．

  本手法においても，既存の原因特定手法と同様にテストツールを利用する．テストツールを利用する場合には，入力に対して何らかの出力がなければ，その出力が正しいかどうかを判定することは困難である．従って，テストを自動的に行うために，ソフトウェアが入力に対して何らかの出力を出すことを前提とする．

- 利用するハードウェア，運用環境は変更されない．

  テストツールを利用してテスト結果を比較するためには，一つのソフトウェアに同じ入力を与えれば，常に同じ出力を出すことが最低条件となる．従って，欠陥が常に発生するような環境を維持するためには，少なくとも，ハードウェア，運用環境が常に同じ状況になければならない．

- ソフトウェア保守担当者があらかじめソフトウェアを理解している．

  本手法においてはバージョン管理システムを利用していることを前提としており，保守担当者は管理されているソースプログラム，それに付随する文書を理解することで，ソフトウェアを理解できると考えている．

3.3 デバグ

デバグとは，診断と修復の作業であり，ソフトウェアに欠陥が発見されると開始され，ソフトウェアの修正とテストが成功すると終了する．デバグ作業は5つの段階，すなわち，精通化，安定化，局所化，修正，妥当性確認より構成される．本手法はこの5つの段階に基づいて支援を行う．

(1) 精通化：ソフトウェアを理解する．

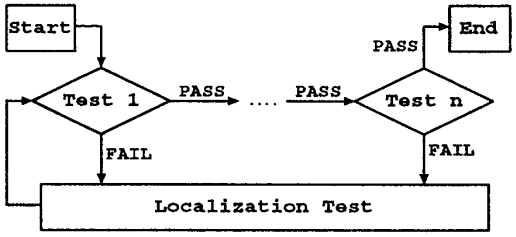

図 2: 回帰テスト

本手法では保守担当者があらかじめ行うものとしており，支援しない．

(2) 安定化：テストの再現性を保つ．
出力を安定させるために，一つの欠陥の原因となるエラーを探すテストにおいては同一の入力を与える．

(3) 局所化：欠陥の原因となるエラーを発見する．
自動的にテストを行って突き止められたバージョン間の差分に欠陥の原因となるエラーが含まれていると考える．

(4) 修正：エラーを取り除く．
局所化の段階で発見されたバージョン間の差分を現在のバージョン上で強調表示することで，デバグの支援を行う．

(5) 妥当性確認：ソフトウェアの動作を確認する．
修正の段階でのソフトウェアの変更の確認，および新たな欠陥の存在有無の確認をするために回帰テストを利用する．

本デバグ手法では局所化で発見されたバージョン間の差分を必ず現在のバージョン上で強調表示することで修正を行う．妥当性確認およびその後のデバグのため，現在バージョンに対してデバグで行った変更は，古いバージョンにも反映させる必要がある．

従って，デバグ手法実現のための 3 つの柱として，安定化の段階および局所化の段階を支援するためのテスト手法，修正の段階を支援するための表示手法，および妥当性確認の段階を支援するための反映手法を考える．

3.4 テスト手法

ソフトウェアに対して何らかの変更を行う度に，用意されたテストを全て実行する形式の回帰テストを自動的に行う (図 2 参照)．

実行されるテストの総数を n とし，実行可能ファイルの基準バージョンを b，現在のバージョンを m とし，バージョン m のソフトウェアを V(m) とする．テストケース $i(1 \leq i \leq n)$ のテストを行う際には，

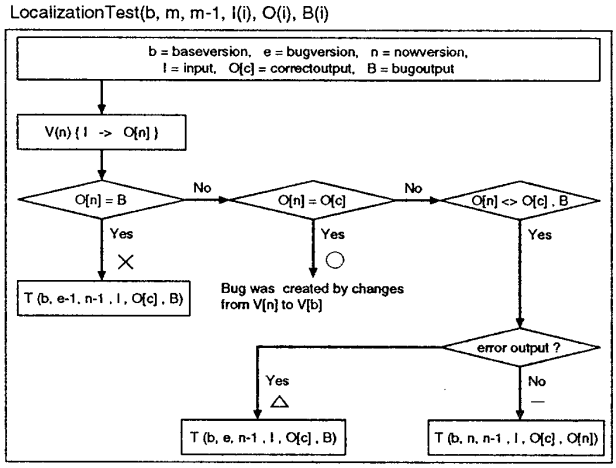

図 3: 局所化テストアルゴリズム

入力 (I(i) とする) と期待する正常出力 (O(i) とする) を利用する．回帰テストにおいてテストケース i で欠陥が発見されれば，その欠陥出力 (B(i) とする) を記憶し，欠陥の原因を特定するテスト (以下，局所化テストとする) を行う．このとき，局所化テストのアルゴリズムは次のようになる (図 3 参照)．

(1) バージョン m-1 の実行可能ファイルを呼ぶ．
(2) 常に I(i) を与えてテストを行う．
(3) テストによって得られた出力を，O(i) および B(i) と比較する．
(4) B(i) と一致 (表記:×)
同一欠陥を出力するバージョンにおいては，欠陥が同一原因によって引き起こされると考え，バージョン m と m-1 の間の差分には欠陥の原因となるエラーが含まれていないと考える．図 4 では (1) にあたり，この場合 V9 と V10 の間の差分には原因となるエラーは含まれない．
(5) O(i) と一致 (表記:○)
基準バージョンではテストケース i が必ず成功すること，および現在バージョンではテストケース i が必ず失敗することがあらかじめわかっている．従って，テストを続けると必ずテストケース i が成功するバージョンが見つかることは保証される．この場合，バージョン m と m-1 の間の差分に欠陥の原因となるエラーが含まれていると考え，テストを終了する．図 4 では (2) にあたり，この場合 V6 と V7 の間の差分には原因となるエラーが含まれる．
(6) 結果が出力されない (表記:△)
テスト実行中に，セグメントエラー等の欠陥

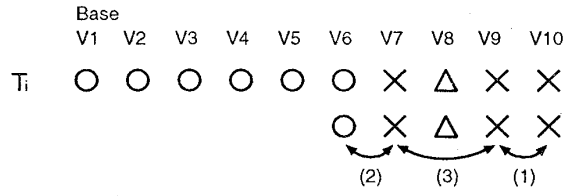

図 4: 局所化テストの出力結果

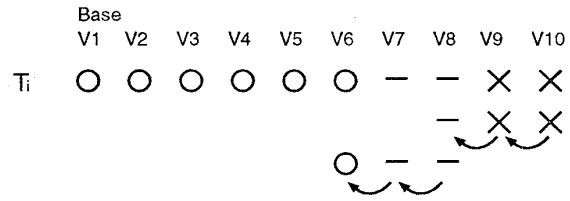

図 5: 新しい欠陥の発見

によってそれ以上テストを続けられない状況はしばしば起こり得る．この場合テストツールを用いて出力結果を比較することは難しい．従って，テスト結果として△を出力したバージョンは局所化テストにおいて考慮しない．図4では(3)にあたり，この場合V8の出力が△なので，V8を考慮せず，続いてV7の出力を比較する．

(7) 上記のどれにも属さない(表記:－)
新たな欠陥が見つかったと考える．新たに見つかった欠陥を取り除かなければ，最初に見つかった欠陥の原因を含む差分を特定することは困難である．そのため，新たに見つかった欠陥の原因を先に特定する．図5では，V8において新たに見つかった欠陥に対して局所化テストを実行してV6とV7の間の差分に原因となるエラーが含まれると特定する．しかし，ここで見つかった差分は新しい欠陥に対するものであり，元の欠陥に対する差分を見つけるためには再度局所化テストを実行する必要がある．

ソフトウェア更新時には，どのような系列でテスト結果が移行していくかわからない．従って，同じ出力×が続き，初めて出力○が得られるバージョン間の差分に欠陥の原因となるエラーが含まれると考える．既存の手法では二分探索を利用しているが，これではあるバージョン $i(b \leq i \leq m)$ においてテスト結果が出力×であっても，$j(i \leq j \leq m)$

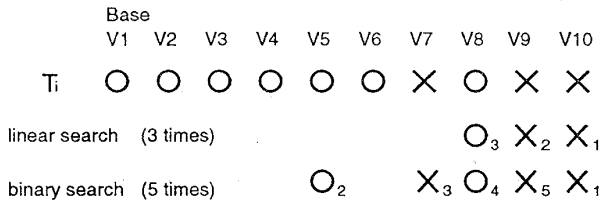

図 6: 線形探索と二分探索

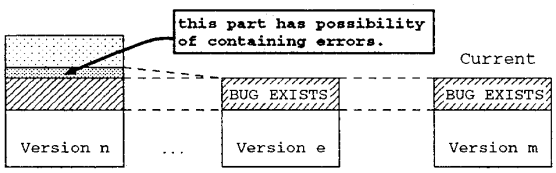

図 7: バージョン間の差分でコードを削除した場合

を満たす全てのバージョン j においてテスト結果が出力○にならないことを示す必要がある(図6参照)．この場合，二分探索よりも線形探索の方がテスト回数が少なくて済む．そこで差分を特定するために線形探索を利用する．

3.5 表示手法

テスト手法で発見したバージョン(テスト出力が○であるバージョンをnとし，テスト出力が×であるバージョンをeとする)間の差分を現在のバージョンで表示するためには，まず実行可能ファイルとソースプログラムの関連情報を利用して，どのソースプログラムのどのバージョン間の差分を強調表示すればいいかを調査する．その調査内容を利用して，差分表示を次のようにして行う．

削除 現在バージョンのソースプログラム上に現れず，変更前のソースプログラムを強調表示できない(図7参照)．新たに削除されたソースを見るための機構が必要となる．

挿入 現在バージョンのソースプログラム上に現れる部分を強調表示する(図8参照)．局所化テストで発見したバージョン以降，引き起こされる欠陥は変化しない．挿入後にも何らかの変更が行われるが，欠陥の致命的な原因となる部分は変更していないと考えられる．

変更 削除と挿入の組合せで実現可能

3.6 反映手法

局所化テストで新たな欠陥が発見された(この時の出力を B_{new} とする)場合に，まず新たな欠陥

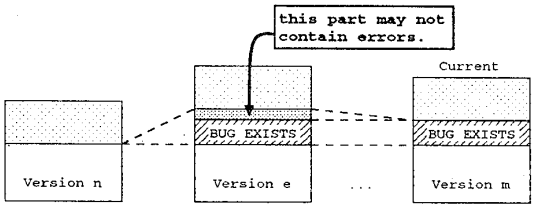

図 8: バージョン間の差分でコードを挿入した場合

を除くが，最初の欠陥は訂正されずに残っている．従って，2度目の局所化テストで最初の欠陥の原因となるエラーを探す．しかし，現在のバージョンに行った変更(これを C とする)を古いバージョンにも反映させなければ，同じテストケースを用いる限り古いバージョンはテスト出力として常に B_{new} を出力することになり，デバグが行えない．

そこで，Cを古いバージョンに反映させることが重要となる．この場合，古いバージョンのソースプログラムにCを反映させ，新たに実行可能ファイルを作成すべくコンパイルする．ただし，Cを古いバージョンに必ずしも反映できるとは限らず，また，反映できたとしてもコンパイルが成功するとは限らない．従って，実行可能ファイルを生成できる場合には，それを従来利用していた実行可能ファイルの代わりに局所化テストで利用する．何らかの理由で実行可能ファイルが生成できない場合には，そのバージョンの実行可能ファイルは局所化テストで利用しない．図9は，局所化テストによって実行可能ファイルのバージョン1.1と1.2の間の差分にエラーが含まれることがわかり，そのエラーに対する修正を行った場合の古いバージョンへの反映例である．ソース1，ソース2ともに古いバージョンへの変更の反映は成功したが，ソース1のバージョン1.3'とソース2のバージョン1.2'から実行可能ファイルのバージョン1.3'の作成に失敗したため，実行可能ファイルの系列から1.3'を取り除き，これ以降の局所化テストでは利用しない．

4 実験方針

現在，本手法の有効性および実用性を評価する実験をおこなうために本デバグ手法に基づくシステムの試作を行っている．システムをC言語で，インタフェース部分はGTK+ライブラリを利用して開発している．バージョン管理システムとしてRCS[15]を採用し，テストツールとしてDejaGnu[13]を利用する．本システムは簡易エディタとしての機能も備えており，テストからデバグまでの一連の流

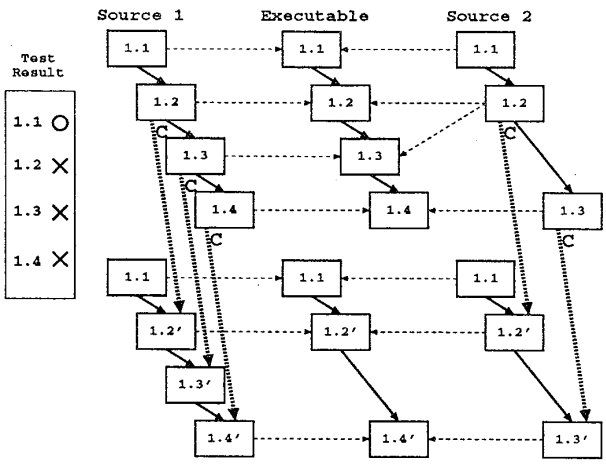

図 9: 過去のバージョンへの反映例

れを支援できる．

4.1 実験の目的

実験を行うことで，以下の評価を行うことを考えている．

(1) 普通にデバグを行う場合と本デバグ手法を用いてデバグする場合の時間にどの程度の差が見られるのかを調べ，本手法の実用性を示す．
(2) 実験全体で発見されたエラーのうち，本デバグ手法で特定されたバージョン間の差分にエラーが含まれる割合がどの程度なのかを調べる．デバグ時間との兼ね合いもあるが，これによって本手法の有効性を示す．
(3) 特定されたバージョン間の差分が小さいほど，デバグの負担が小さくなる．しかし，バージョン数が増加によるテスト回数の増大は避けられない．そこで，特定されたバージョン間の差分量が本デバグ手法に与える影響力を示す．

4.2 実験の手順

実験では，被験者が保守担当者となってソフトウェアの更新，テストおよびデバグを行う．実験の手順であるが，まず被験者を2つのグループにわける．1グループは回帰テストが実行でき，かつバージョン情報を参照できる環境を用意して実験を行う．もう1つのグループは本デバグ手法に基づくシステムを利用して実験を行う．ただし，本デバグ手法の性質上，従来のエラーを埋め込む手法を採用できない．そこで，実際にこちらで用意したソフトウェアに対して何らかの機能追加を行うことを考えている．

5 まとめ

本研究では，保守作業において，プログラムに変更を加え，テストを行うことで変更していない部分に欠陥が見つかった場合に，適用可能なデバグ手法を行った．本手法を用いることで，実際の保守においてデバグ作業をより行いやすくなることが期待できる．

今後，4章で述べた実験方針に基づいて実験を行い，本手法の有効性および実用性に関する評価を行う．

参考文献

[1] Babich, W. A., "Software Configuration Management," Addison-Wesley, Reading, Massachusetts, 1986.

[2] CASE 1988-89, Sentry Market Research, Westborough, MA, pp.13-14, 1989.

[3] Carma. M 著, ベスト CASE研究グループ 訳, "ソフトウェア開発と保守の戦略," 共立出版社, 1993.

[4] Cashman, P. M. and Holt, A. W. , "A Communication-Oriented Approach to Structuring the Software Maintenance Environment," Software Engineering Notes, 5, No.1, pp.4-17, January 1980.

[5] Conradi, R. and Westfechtel, B., "Version Models for Software Configuration Management," ACM Computing Surveys, Vol.30, No.2, pp.232-280, 1998.

[6] Dogsa, T. and Rozman, I., "CAMOTE - Computer Aided Module Testing and Design Environment," Proceedings of the Conference on Software Maintenance - 88, Phoenix, Ariz., pp.404-408, 1988.

[7] Hetzel, W., "The Complete Guide to Software Testing," QED Informaion Sciences, Wellsley, Mass., 1984.

[8] IEEE., "Test Methods for Mesureing Conformance to POSIX," ANSI/IEEE Standard 1003.3-1991, ISO/IEC Standard 13210-1994.

[9] Leung, H. K. N. and White, L., "Insights into Regression Testing," Proceedings of the Conference on Software Maintenance - 89, Miami, Fla., pp.60-69, October 1989.

[10] Lientz, B. and Swanson, E., "Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations," Addison-Wesley, Reading, MA, pp.151-157, 1980.

[11] Ness, B., and Ngo, V., "Regression containment through source code isolation.", In Proceedings of the 21st Annual Internatial Computer & Applications Conference (COMPSAC '97), IEEE Computer Society Press, pp.616-621, 1997.

[12] Raither, B. and Osterweil, I., "TRICS : a Testing Tool for C," Proceedings of the First European Software Engineering Conference, Strasbourg, France, pp.254-262, 1987.

[13] Savoye, R., "Test DejaGnu Testing Framework for DejaGnu Version 1.3," Free Software Foundation, Inc., January, 1996.

[14] Swanson, E., "The Dimensions of Maintenance," Second International Conference on Software Engineering Proceedings, San Francisco, pp.492-497, October 1976.

[15] Tichy, W. F., "RCS - A System for Version Control, Software-Practice and Experience, Vol.15, No.7, pp.637-654, 1985.

[16] Zeller, A., "Yesterday, my program worked. Today, it does not. Why?", Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE '99), Toulouse, France, September 1999.