論　文

プログラム差分を用いたデバッグ支援手法 DMET

松下　誠[†a)]　寺口　正義[††]　井上　克郎[†]

Program-Delta Oriented Debugging Supporting Method DMET

Makoto MATSUSHITA[†a)], Masayoshi TERAGUTI[††], and Katsuro INOUE[†]

あらまし　正常に機能することが事前に分かっているリビジョンと，欠陥が含まれるリビ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，テストのたびにソースプログラムからソフトウェア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，デバッグ作業まで考慮されておらず，実用的とはいえなかった．そこで本研究では，従来よりもテスト実行時間を減らし，テストからデバッ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ッグ手法 DMET の提案を行う．また，本手法の有効性を確認するため，DMET を用いた開発支援環境 DSUS の実装を行い，比較実験を行った．その結果，DMET を用いることにより，デバッグ時間全体を短縮できることが分かった．

キーワード　リビジョン管理，差分，デバッグ

1. まえがき

ソフトウェアの保守とは，本稼動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]．

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三つの基本的な種類に分類している [15]．

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

更に，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．

- 修正保守：エラーの識別，修正
- 適応保守：環境の変化に応じた修正
- 完全化保守：性能の改善，機能の変更や追加

一般的に，これらのソフトウェア保守活動の中では，修正保守のための活動が一番多いと考えられがちである．しかし，1980 年代はじめに 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 [11]．

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 の研究で示されている [7]．したがって，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．

開発の際に起きるこれらの動作確認のために，回帰テスト [6], [10], [13] が広く用いられている．また，開発されるソフトウェアはリビジョン管理システム [1], [16] 等のソフトウェア管理システムを用いて管理されていることが多い．この 2 点に着目して，ソフトウェア管理システムによって管理されているソフトウェアを対象として，完全化保守を支援する研究がこれまで行われてきている [12], [17]．しかし，テストを実行する際

† 大阪大学大学院情報科学研究科，豊中市
Graduate School of Engineering Science, Osaka University, 1–3 Machikaneyama-cho, Toyonaka-shi, 560–8531 Japan
†† IBM 東京基礎研究所，大和市
Tokyo Research Laboratory, IBM Japan, Ltd., 1623–14 Shimotsuruma, Yamato-shi, 242–8502 Japan
a) E-mail: matusita@ist.osaka-u.ac.jp

に利用するテストツール [9] の設定を，状況に応じて手作業で行う必要がある．また，テスト後に行うデバッグ作業に対する支援も十分とはいえない．

そこで本研究では，機能の変更や拡張を目的とした完全化保守作業を対象とする．このような保守作業では，作業対象となる機能以外の部分も影響を受けて，テストで欠陥が検出されることがある．本論文では，このような欠陥の除去を効率良く行うために，回帰テストを用いたデバッグ手法 DMET の提案を行う．

本手法では，まずプロダクトのリビジョンを順に取り出し，テストツールを利用して自動的にテストを行う．テストの結果，欠陥を出力するリビジョンと正常に出力するリビジョンを発見した場合，このリビジョン間で行われた修正にエラーがあるとする．次に，この修正内容（以下，単に差分と呼ぶ）が現在のリビジョンにおいてどの部分に相当するかを調べる．この結果を利用して，利用者はデバッグを行い，欠陥の修正を行う．欠陥を訂正した後，その変更を古いリビジョンにも反映することで，以降のテストに役立てる．

また，DMET の有効性を検証することを目的として，DMET を用いたデバッグ支援システム DSUS の試作を行い，DSUS を用いた比較実験を行った．この結果，DMET はテストからデバッグまでの一連の流れを支援することができ，実際の保守においてデバッグ作業時間を短縮できることが分かった．

2. ソフトウェア保守

2.1 保守作業

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，基本的に次の三つに分類することができる．

（1） ソフトウェア及びなされるべき変更の理解
（2） 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

このうち，（1）が正しく行われれば，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，欠陥の原因究明及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及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リビ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，様々なプロダクトの識別や制御，状態の把握等を解決する作業を指す [5]．また，リビ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，管理する作業であり，様々な管理手法のモデルが提案されている．更に，そのモデルに基づくリビジョン管理システムが実装されている [1], [16]．通常プロダクトの改訂は複数回行われ，各改訂ごとに作成されるプロダクトをリビジョンと呼ぶ．

また，（3）を行うにあたり，ソフトウェアに変更を加えた後で，ソフトウェアが仕様どお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], [10], [13]．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，修正が正しく行われているかを検査する際に役立てられている．

2.2 既存手法の問題点

リビ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，仕様どおりの性能で動作するリビジョン（以下，基準リビジョンと呼ぶ）が存在している．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，基準リビジョンのプログラムを変更し，回帰テストを用いてソフトウェアの動作を確認している．

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（デバッグを行う）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一般的に困難である．

そこで，基準リビジョンから現在のリビ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 [12], [17]．しかし，テストを行うたびに，基準リビジョンに変更を加えてコンパイルを行い，テストを適用するソフトウェアを作成するため，テストに多大な時間が必要となる．また，欠陥を発見するたびに，その欠陥に応じてテストケースを逐一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 [9] に関する教育や訓練も必要となる．更には，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，その後のデバッ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といった問題がある．

3. デバッグ手法 DMET

本章では，我々の提案する，リビジョン管理システムが保持しているプログラムの差分情報を用いた，回帰テストを利用するデバッグ手法 DMET について述

べる．

3.1 概　　要

DMET（Debugging METhod）は，基準リビジョンでは全機能が正常に働いており，保守作業の際に機能に欠陥を作りこんだ場合のデバッグ作業を効率良く行うことを支援するための手法である．DMET はテスト手法，表示手法，反映手法という三つの手法を繰り返し適用することにより，欠陥の除去を行う．

テスト手法では，テストツールを利用して自動的にテストを行い，欠陥の原因を含むリビジョン間を特定する．DMET では回帰テストを用いることにより，各リビジョンが欠陥を含んでいるかを確認する．回帰テストにおいて欠陥が発見された場合には，その欠陥がどのリビジョンによって作りこまれたかを確認するためのテスト（以降「局所化テスト」と呼ぶ）を行う．

表示手法では，テスト手法で特定された実行可能プログラムのリビジョン間の差分を，ソースプログラムの最新リビジョン上で表示する．一般的にソースプログラムの修正作業は最新リビジョン上で行われるため，特定されたリビジョンから最新リビジョンまでに行われた変更を，本手法では自動的に求める．

反映手法では，テスト手法で特定されたリビジョンまでリビジョンをさかのぼりながら，最新リビジョン上で修正作業時に行った変更を反映する．本論文では以降，最新リビジョン上での修正変更を反映する必要があるテストで欠陥を生じたリビジョンをまとめて反映必須リビジョンと呼ぶことにする．反映必須リビジョンに対して修正内容を適用することで，それ以降のテスト作業を効率良く続けることができる．

以下，本手法が前提とするソフトウェア開発環境について述べた後，テスト手法，表示手法，及び反映手法の具体的手順について説明する．

3.2 DMET を適用する際の前提環境

ソフトウェア保守において我々の考えるデバッグ手法を利用する際に，ソフトウェア及びその開発と保守，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．ある程度成熟したオープンソースソフトウェアの開発事例など，この種の前提は広く普及している．よって我々は，これらの前提条件が近年のソフトウェア開発環境で問題とはならないと考えている．

- リビジョン管理システムを用いた開発を行う

開発中の作業履歴を比較的詳細な単位で把握するために，リビジョン管理システムを用いた開発作業を前提とする．リビ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，コンパイル済みファイルとする．コンパイル済みファイルをリビジョンとして登録する際には，付随情報として当該ファイルとソースプログラムのリビジョンの関連も記録する．

- 基準リビジョンが存在する

本手法はソフトウェアの保守作業を対象としている．このため，保守作業が開始される時点でのソフトウェアを基準リビジョンとすることにより，修正前の全機能が必ず動作することが保証されているリビジョンが必ず存在することとする．

- 入力に対して何らかの出力を行う

本手法では，テストツールを用いてテストデータの入力に対して出力が正しいものであるかを判定しているため，出力が行われるソフトウェアを対象とする．ここで出力とは，例えば端末への文字列の出力等，一般的に目に見える結果として出されるものを仮定する．

- 入力や利用環境等は変更されない

一般的に，ソフトウェア保守活動においては規模の大きな仕様変更等は行われないため，本手法では，入力パラメータ自体が増減するなどといった，ソフトウェアに対する入力が変更されることは想定しない．また，同様の理由から，ハードウェアや動作環境といった，ソフトウェアの利用環境についても変更は行われないことを本手法は前提とする．

このようなソフトウェア開発環境においては，開発者は小さな作業単位でその結果をリビジョン管理システムに登録することが一般的である．一人の開発者が小規模の開発を行っている場合，リビジョン登録を行うたびに回帰テストを実行することも可能であろうが，一般的には複数の開発者が協調しながらそれぞれの作業を行っており，また一人の作業者も同時に複数の作業を並行して行うことも珍しくはない．また，あるまとまった作業の途中経過を記録するために，修正が不完全なものをリビジョンとして登録することもよく行われる．このような状況においては，開発者がリビジョン登録を行うごとに回帰テストを実行しても意味がないため，ある程度まとまった作業が終了した時点で，修正が正しく行われたかを回帰テストなどによって確認することとなる．

3.3 テスト手法

テスト手法では，ソフトウェアに対して何らかの変更を行うたびに，用意されたテストをすべて実行する回帰テストを自動的に行う．

テスト対象となるソフトウェアに対して用意された，

完全化保守を行うために用いるテストの集合を T_1〜T_n とする．これらのテストは，保守作業が開始される際にあらかじめ用意する．また，あるファイルに対する i 番目のリビジョンを V_i としたとき，基準リビジョンは V_B，最新リビジョンは $V_L(B \leq L)$ として表すものとする．いま，ある時点のソフトウェアに対してテスト $T_i(1 \leq i \leq n)$ を適用する場合を考える．まずあらかじめ，各ファイルについて正しく動作することがあらかじめ分かっている V_B に対して，T_i の入力である I_i を与えた場合の出力結果 $O_{B,i}$ をこのソフトウェアの正常出力とする．

テスト手法ではまず，V_L に対してすべての $T_i(1 \leq i \leq n)$ を適用し，その出力 $O_{L,i}$ が $O_{B,i}$ と一致するかを確認する．もし，あるテスト T_i を適用した際に欠陥が発見された場合，局所化テスト $LclT(i, L-1, L)$ を行う．

なお，局所化テスト $LclT(i, j, k)$ とは，$V_j(j \leq k)$ に対して T_i を適用した結果を判定し，かつ欠陥の原因がどのリビジョンとどのリビジョン間に存在しているかを返すテストである（図 1）．ただし，ここでは V_k に対して T_i を用いてテストを行った際，その出力 $O_{k,i}$ が $O_{L,i}$ と同一であることが分かっている．局所化テスト $LclT(i, j, k)$ のアルゴリズムを図 2 に示す．

（1） j が基準リビジョン B と一致しているならば，「欠陥は V_B と V_k 間に存在する」と判断して終了する．

（2） V_j に対し，テスト T_i を適用する．具体的には，I_i を V_j に与えて，出力 $O_{j,i}$ を得る．

（3） $O_{j,i}$ と $O_{k,i}$ が一致すれば，リビジョン j にはまだ欠陥が存在していることになるため，テストの実行結果は「×」（同一の欠陥が含まれる）と判定する．更に再帰的に $LclT(i, j-1, j)$ を実行し，その結果を求める結果とする．

（4） もし $O_{j,i}$ と $O_{B,i}$ が一致した場合には，リビジョン j の時点では欠陥が含まれていなかったことになるため，テストの実行結果は「○」（欠陥が含まれない）と判定する．また，「欠陥は V_j と V_k 間に存在する」と判断して終了する．

（5） それ以外の $O_{j,i}$ 自体が得られた場合には，テストの実行結果は「−」（新たな欠陥が発見された）と判定する．また，実行結果がソフトウェアの異常終了のため得られなかった場合には「△」（結果が得られない）であったとする．どちらの場合においても，更に再帰的に $LclT(i, j-1, k)$ を実行し，その結果を求める結果とする．

本手法では，各リビジョンを新しいものから順に線形探索を行うことにより，欠陥を含む個所の特定を行っている．仮に結果が「○」と「×」しかなく，かつ，○から×への遷移がたかだか 1 回しか起きない場合には，二分探索を用いることによりより効率の良い探索が可能である．しかし，前提としてその他の出力が含まれる場合や，何らかの理由により○から×，あるいはその逆への遷移が複数考えられるため，ここでは二分探索を用いることはできない．

3.4 表示手法

テスト手法で発見したリビジョン間の差分（ここでは，リビジョン C とリビジョン E の間とする

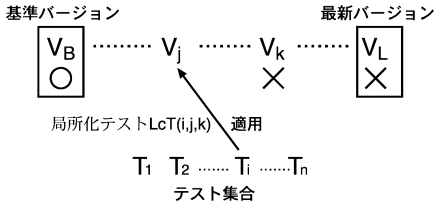


図 1 局所化テスト
Fig. 1 Localization test.

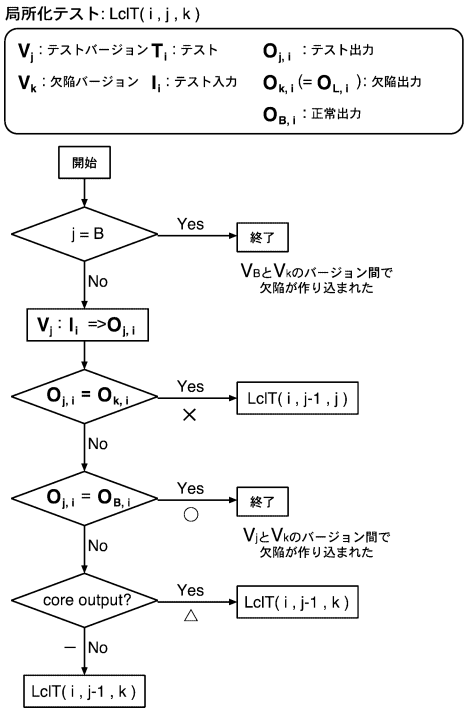


図 2 局所化テストアルゴリズム
Fig. 2 Localization test algorithm.

($C < E \leq L$)) を最新のリビジョン L で表示するために，まずソフトウェア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リビジョン間の差分を強調表示するか調べる．両リビジョン間で削除された場合，あるいは挿入された場合に応じて，以下のような表示を行う．

[**削除**] 現在リビ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自身は強調表示できない（図 3 参照）．このため，最新リビジョン上では，削除されたソースコードの場所と内容を示す印によって強調表示を行う（図 2 V_L における点線表示）．

[**挿入**] 現在リビ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 4 参照）．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されていないと考えられる．

3.5 反映手法

局所化テストで新たな欠陥が発見された場合，まずその欠陥の除去を最新リビジョン L に対して行う（以下，これをした差分を修正 ΔL と呼ぶ）．次に，局所化テストを続行するために，ΔL を過去のリビジョンに反映させる．まず，patch プログラム等を用いて，ΔL を過去のリビジョンへ適用し，新たなソフトウェアを作成すべくコンパイルを行う．もしコンパイルが成功すれば，それを新たなテスト対象プログラムとして利用する．もし ΔL の patch が失敗するか，成功してもコンパイルに失敗した場合には，そのリビジョンは以降の回帰テストの対象外とする．

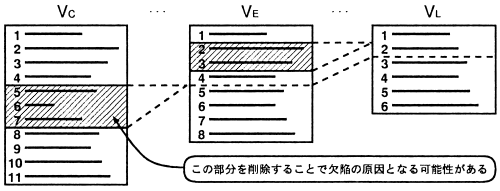


図 3 リビジョン間の差分でコードを削除した場合
Fig. 3 Code deleted between revisions.

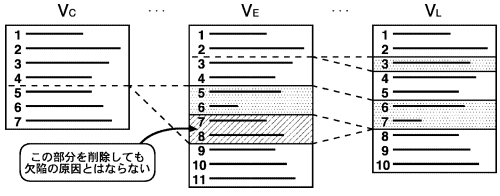


図 4 リビジョン間の差分でコードを挿入した場合
Fig. 4 Code inserted between revisions.

4. 評価実験

本章では，DMET に基づくデバッグ支援システム DSUS を用いた，DMET の評価実験について述べる．

4.1 試作システム DSUS

DMET に基づくデバッグ支援システムの構築にあたり，我々は以下の点に留意してシステムの作成を行った．

- プログラミング言語から独立

一般的に，プログラミング言語に依存したシステムを構築することによって，より細かいデバッグ支援を行うことが可能になる．しかし，現在のプログラミング開発は多くのプログラミング言語によって行われており，DMET 自身は言語に依存しない手法であることから，DSUS には特定のプログラミング言語に依存した機能をもたせないようにした．

- テストとエラー修正の両方を支援

DSUS はテストの自動実行だけでなく，テストによって発見したエラーの修正を行うための環境を提供する．これによって，開発作業を DSUS 内で完結させることができる．

- ツールの自動実行

テストの実行をはじめ，DMET で述べられている各作業を順次正しく実行することは，開発者に対して更なる負担を強いることになる可能性がある．DSUS では，DMET の手順やそれに付随するテストを自動的に行うことによって，DMET の導入にかかる負担を減らすことを目指す．

DSUS は，DSUS$_{main}$，RCS [16]，DejaGnu [14] と，ユーザに対する GUI から構成される（図 5）．GUI を用いることにより，ソースコードの編集やエラーを含むと考えられる差分の表示などを行う．

RCS は，多くの開発環境で用いられている，リビジョン管理システムである．DSUS は，ユーザが RCS に対する操作のインタフェースとして動作することにより，誤った操作や危険な操作を行わないように作られている．また，開発者がソースコードを編集する際には，定期的にその内容を RCS へ登録することによって，RCS に登録される差分が小さくなるようにする．DejaGnu はオープンソースで開発が進められているテスト実行フレームワークであり，DSUS では回帰テストを実行するために用いられる．

DSUS$_{main}$ は DSUS 全体の中核をなす部分である．GUI の管理，RCS に対する操作，DejaGnu 実行環境

の設定やテストの管理はこの部分で行われる．DSUS の GUI（図 6）は，大きくエディタ用ウィンドウ（画面左）と状態ウィンドウ（画面右）から構成される．開発者は，状態ウィンドウから選択することにより，任意のリビジョンのソースコードを取り出して編集することができる．また，画面上部のボタンを利用して，テストの実行等を指示することができる．テストの結果，エラーが検出された場合には，エラーが含まれると考えられるリビジョンを色付きで表示する．

DSUS は C 言語によって実装されており，規模は約 20000 行である．また，GUI の実装には GTK+ を用いた．

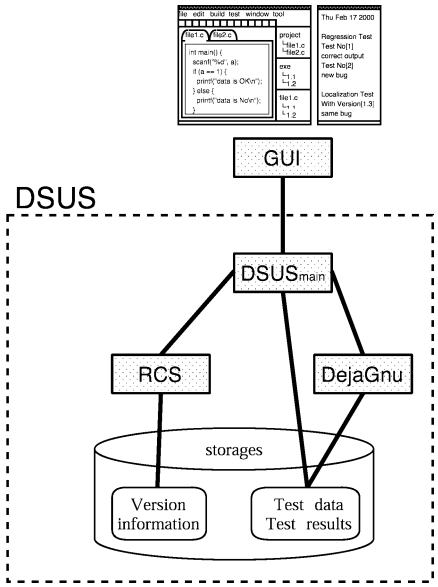


図 5　DSUS の構成
Fig. 5　The DSUS structure.

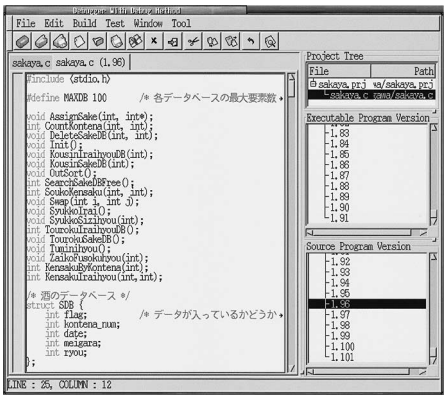


図 6　DSUS の GUI 画面
Fig. 6　Screenshot of GUI.

4.2　実験の概要

デバッグ作業を DMET に従った場合と従来手法で行った場合でかかる時間にどの程度の差が見られるのかを，DSUS を用いて調べ，本手法の実用性を示す．

本実験では，C プログラミングの経験をある程度もつ大学 4 年生及び大学院生 10 人を被験者とした．全体を二つのグループ（G_1 と G_2）に分け，グループ G_1 は DSUS を，グループ G_2 は，DSUS の機能のうち，DMET に基づくリビジョンの特定機能のみを利用できないようにした DSUS を用いて，デバッグ作業を行うこととする．なお，デバッグ対象として用いるソフトウェアの問題としては，酒屋問題 [8] を用いた．

実験は，ステップ 1 とステップ 2 に分け，ステップ 1 で，ステップ 2 の際に用いるデバッグ対象となるソフトウェアの開発と収集を行った．収集されたソフトウェアを用いて，本実験でデバッグ作業を行った．

4.3　ステップ 1

酒屋問題のうち「空コンテナマーク」部分以外を正しく実装したソフトウェアを事前に用意しておき，これに対して，空コンテナマークの機能を追加する．なお，当該ソフトウェアは C 言語で記述されており，およそ 500 行程度の規模である．得られたソフトウェアのうち，この拡張により他の機能に対して欠陥を生じるようになった二つのソフトウェア（以下，ソフトウェア A，B）をステップ 2 でデバッグ対象とするソフトウェアとした．表 1 に収集したソフトウェアの一覧とその開発履歴を示す．

例えばソフトウェア A は，最初に用意したソフトウェアを初版として合計 28 のリビジョンをもつ．ソフトウェア A は，DMET によって 25 回の局所化テストが実行され，その結果 DMET が検出したリビジョン間の差分に，欠陥が含まれていることが分かった．

4.4　ステップ 2

グループ G_1 及び G_2 に対して，ステップ 1 で得られた三つの欠陥を含むソフトウェアを与え，DSUS を用いてデバッグ作業を行ってもらった．デバッグの際に利用するテストデータについては，あらかじめ準備

表 1　本実験で用いたソフトウェアの開発履歴
Table 1　Development software and its history.

ソフトウェア名	リビジョン数	テスト回数
A	28	25
B	91	10

表 2　G_1 のデバッグ時間（DMET 利用）
Table 2　Debugging time of G_1. (with DMET)

被験者	平均（分）
T1	75
T2	62
T3	65
T4	79
T5	54
G1 全体	67.0

表 3　G_2 のデバッグ時間（DMET 非利用）
Table 3　Debugging time of G_2. (without DMET)

被験者	平均（分）
T6	237
T7	107
T8	237
T9	69
T10	165
G2 全体	163.0

したものを用いてもらい，与えたテストデータに対する処理が正しく行えるのを確認できた時点でデバッグ終了とした．デバッグを開始してから終了するまでの，各被験者が費やした時間の計測を行った．表 2 と表 3 が，実験によって得られた各被験者（T1〜T10）のデバッグ時間である（単位は分）．

4.5 考　　察

ソフトウェア A，B のデバッグ時間合計に対して，5%有意水準による Welch の検定を用いたところ，G_1 と G_2 には有意な差が見られることが確認できた．本実験により，DMET を用いることによってデバッグ時間を短縮することができることが分かった．

5. 関 連 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．

5.1 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 [12]．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，基準リビ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とき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．

5.2 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 [17]．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とらえる．したがって，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．したがって，テストの回数が増加し，テストに多大の時間がかかる．

5.3 問　題　点

Ness らと Zeller のどちらの手法も基準リビ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．また，どのソースプログラムをどのように修正したかを正確に管理した上でコンパイルを行う必要があるため，単純にコンパイルを行うだけでも繁雑な作業となる．

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の学習や訓練を行わなければならない．更には，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，デバッ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．

DMET では，どのソースプログラムをどの時点でコンパイルすることになるか，を DMET 自体の枠組みで管理することになり，DSUS のような支援環境を用いることによって容易にコンパイル作業を行うことが可能となっている．また，DMET においては最初にテストケースを作成しておき，それを用いたデバッ

グを行うため，欠陥に応じてテストケースを自作する必要性がなく，またテストケース作成自体にかかる時間も軽減される．DMET は欠陥の特定だけではなく，デバッグ作業まで含めた作業を対象としているため，実際の保守作業においても容易に適用することが可能であり，今回実験によってそれを示している．

6. むすび

本研究では，保守作業を対象としたデバッグ手法 DMET の提案を行った．また，試作システム DSUS を用いた実験を通じ，DMET の有効性を示した．本手法を用いることで，実際の保守においてデバッグ作業をより行いやすくなることが期待できる．

今後の課題としてまず，差分検出アルゴリズムの改良によって，欠陥の含まれる差分の検出をより確実に行うことが挙げられる．また，DMET が検出した差分が比較的大きくなった場合，DMET によって得られた結果を用いても作業時間の短縮に貢献しないことが考えられる．このような場合，DSUS があらかじめそれを判断し，単に結果を提示するだけではなく，開発者に注意するための方法などについて検討を行いたい．

文　献

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

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

[3] M. Carma（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，1993.

[4] P.M. Cashman and A.W. Holt, “A communication-oriented approach to structuring the software maintenance environment,” Software Engineering Notes, vol.5, no.1, pp.4–17, Jan. 1980.

[5] R. Conradi and B. Westfechtel, “Version models for software configuration management,” ACM Computing Surveys, vol.30, no.2, pp.232–280, 1998.

[6] T. Dogsa and I. Rozman, “CAMOTE — Computer aided module testing and design environment,” Proc. Conference on Software Maintenance - 88, pp.404–408, Phoenix, AZ, 1988.

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

[8] H. Kudo, Y. Sugiyama, M. Fujii, and K. Torii, “Quantifying a design process based on experiments,” Proc. 21th International Conference on System Sciences, pp.285–292, Hawaii, 1988.

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

[10] H.K.N. Leung and L. White, “Insights into regression testing,” Proc. Conference on Software Maintenance - 89, pp.60–69, Miami, FL, Oct. 1989.

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

[12] B. Ness and V. Ngo, “Regression containment through source code isolation,” Proc. 21st Annual Internatial Computer & Applications Conference (COMPSAC ’97), pp.616–621, IEEE Computer Society Press, 1997.

[13] B. Raither and I. Osterweil, “TRICS: A testing tool for C,” Proc. First European Software Engineering Conference, pp.254–262, Strasbourg, France, 1987.

[14] R. Savoye, Test DejaGnu testing framework for DejaGnu version 1.3, Free Software Foundation, Jan. 1996.

[15] E. Swanson, “The dimensions of maintenance,” Second International Conference on Software Engineering Proceedings, San Francisco, pp.492–497, Oct. 1976.

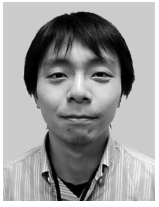
[16] W.F. Tichy, “RCS — A system for version control,” Software-Practice and Experience, vol.15, no.7, pp.637–654, 1985.

[17] A. Zeller, “Yesterday, my program worked. Today, it does not. Why?,” Proc. 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE ’99), pp.253–267, Toulouse, France, Sept. 1999.

（平成 15 年 6 月 26 日受付，16 年 1 月 6 日再受付）

松下　誠

平 5 阪大・基礎工・情報卒．平 10 同大大学院博士後期課程退学．同年同大・基礎工学研究科・助手．平 14 同大・情報科学研究科・助手．博士（工学）．ソフトウェア開発環境，ソフトウェア開発プロセス，オープンソース開発の研究に従事．



寺口　正義

平 5 阪大・基礎工・情報卒．平 12 同大大学院博士前期課程了．同年 IBM 基礎研究所．修士（工学）．在学時，デバッグ支援環境の研究に従事．

井上　克郎　（正員）

昭 54 阪大・基礎工・情報卒．昭 59 同大大学院博士課程了．同年同大・基礎工・情報・助手．昭 59〜昭 61 ハワイ大マノア校・情報工学科・助教授．平元阪大・基礎工・情報・講師．平 3 同学科・助教授．平 7 同学科・教授．平 14 同大・情報科学研究科・教授．工博．ソフトウェア工学の研究に従事．