NAIST-IS-DT0061206

博士論文

ソースコード解析に基づくソフトウェア品質確保

佐藤　慎一

平成 18 年 9 月 6 日

奈良先端科学技術大学院大学
情報科学研究科　情報システム学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に博士（工学）授与の要件として提出した博士論文である.

佐藤　慎一

論文審査委員：松本　健一　教授
関　浩之　教授
飯田　元　教授
門田　暁人　助教授

ソースコード解析に基づくソフトウェア品質確保*

佐藤　慎一

内容梗概

ソフトウェア開発において，限られた開発期間で品質を確保するためには，上流工程での十分な検討に加えて，下流工程においてもコーディングにおける品質の作りこみ，試験における適切なリソース配分，デバッグ作業の効率化がそれぞれ重要となる．本論文では，ソースコード解析がこれら３つの観点いずれにおいても利用可能な技術である点に着目し，実用規模のソースコードを解析した結果からコーディングにおける品質作りこみと試験におけるリソース配分の基準となる指針を得るとともに，ソースコード解析法のひとつである依存関係解析に基づくデバッグ支援ツールを提案する．具体的な成果は次のとおりである．

(1) コードクローンと保守性・信頼性との関係の定量的評価

コードクローンはソフトウェアの信頼性や保守性を低下させる一要因であると考えられているが，その定量的な評価は行われていない．本研究では，あるレガシーソフトウェアを題材としてコードクローンと保守性・信頼性との関係を定量的に分析した.分析はモジュールを単位として行い,保守性尺度として改版数を，信頼性尺度として欠陥検出密度を用い，コードクローンの含有率および最長コードクローン行数との関係を分析した．その結果，コードクローンを含むモジュールは含まないモジュールよりも信頼性が平均的に 40%高く，保守性が平均的に 40%低かった．この結果から，コーディング時にコードクローンを生成する場合に保守性と信頼性の間のトレードオフについて考慮する必要があることを示した.

(2)レガシーソフトウェアにおける fault-prone モジュール予測の有効性評価

これまで，モジュールメトリクスに基づき欠陥を含む可能性のある(fault-prone)モジュールを分類するモデルが多く提案されているが，直近のデータしか利用できないレガシーソフトウェアにおける適用事例はほとんど報告されていない．本研究では，あるレガシーソフトウェアを題材として判別分析，ニューラル

* 奈良先端科学技術大学院大学　情報科学研究科　情報システム学専攻　博士論文，NAIST-IS-DT0061206，平成 18 年 9 月 6 日

ネットワーク，分類木の 3 つの予測モデルを用いて予測精度の評価を行った．その結果，レガシーソフトウェアにおいてもこれらの予測モデルが fault-prone モジュールの絞込みに効果があり，試験の効率化が図れることを示した．

(3)プログラムスライシング・部分評価機能を組み込んだデバッグ支援ツール

デバッグ対象範囲の効率的な絞り込みを目的として，プログラムスライシングおよび部分評価によってソースコードから特定の入力または出力に関係する文のみを抽出し，実行できる機能を持ったデバッグ支援ツールを提案する．試作したツールを用いた実験の結果，誤った出力を示した変数に関する部分のみを抽出することで，デバッグ範囲を 33 行から 25 行に絞ることができ，欠陥の発見が容易になることを確認した．

キーワード

ソースコード解析，ソフトウェアメトリクス，プログラムスライス，コードクローン，ソフトウェア品質

Software Quality Improvement Based on Source Code Analysis

Shinichi Sato

Abstract

To improve software quality in software development and maintenance with limited resources, efficient debugging, appropriate resource allocation for testing and quality improvement in coding phase are important. This paper focuses on source code analysis techniques because they can be useful for those three activities. Then this paper provides guidelines for quality improvement in coding and for appropriate allocation of testing resources based on source code analysis of industry software, and proposes a debug support tool based on program dependence analysis. The details are described in the followings.

(1) Quantitative verification of relationship between code clones and maintainability and reliability

Code clones are considered as a factor to decline software reliability and maintainability. However, it is not quantitatively verified. This study quantitatively analyzes relationship between code clones and software reliability and maintainability of legacy software. In this study, the analysis focuses on each module. The number of revisions of each module is used as the maintainability measure and defect density after delivered is used as the reliability measure. Then relationship between these measures and coverage and maximum length of code clones of each module are analyzed. The results showed that the reliability of modules containing code clones was 40% higher than modules not including code clones, and that the maintainability was 40% lower. This means that a coder has to consider trade-off between reliability and maintainability when creating code clones.

(2) Verifying the effectiveness of predicting fault-prone modules in legacy software

There are many fault-prone modules prediction models based on module metrics until now, however, there are few reports for legacy software for which only recent data is generally available. This study verifies the effectiveness of the prediction models for real legacy software. As the prediction models, discriminant analysis, neural network and classification tree are adopted. The results showed that all of the prediction models were effective for legacy software to put emphasis on modules which should be tested selectively and that it was expected to increase efficiency of testing.

(3) The debug support tool with program slicing and partial evaluation function

To narrow the scope of debugging efficiently, this study proposes the debug support tool that can extract only a part of the program relative to specific inputs or outputs using program slicing and partial evaluation. The experiment showed that the prototype tool could narrow 33 lines of code to 25 lines by extracting only the part of program relative to the illegal output and that the tool made it easier to detect defects.

Keywords: source code analysis, software metrics, program slice, code clone, software quality

関連発表論文

学術論文誌

- 佐藤 慎一, 飯田 元, 井上 克郎, “プログラムの依存関係解析に基づくデバッグ支援ツールの試作”, 情報処理学会論文誌, Vol.37, No.4, pp.536-545, 1996.
- 門田 暁人, 佐藤 慎一, 神谷 年洋, 松本 健一, “コードクローンに基づくレガシーソフトウェアの品質の分析”, 情報処理学会論文誌, Vol.44, No.8, pp.2178-2188, 2003.

国際会議

- Shinichi Sato, Takeshi Hayama, Akito Monden, and Kenichi Matsumoto, “Classifying Defect-Prone Modules Based on Intra-Module Complexity Metrics from a Large-Scale System”, In Proc. 2nd World Congress for Software Quality, pp.607-612, 2000.
- Shinichi Sato, Akito Monden, and Kenichi Matsumoto, “Evaluating the Applicability of Reliability Prediction Models Between Different Software”, In Proc. International Workshop on Principles of Software Evolution, pp.87-94, 2002.
- Shuji Takabayashi, Akito Monden, Shinichi Sato, Ken-ichi Matsumoto, Katsuro Inoue, and Koji Torii, “The Detection of Fault-Prone Program Using a Neural Network”, In Proc. International Symposium on Future Software Technology’99, pp.81-86, 1999.
- Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shinichi Sato, and Ken-ichi Matsumoto, “Software Quality Analysis by Code Clones in Industrial Legacy Software”, In Proc. 8th IEEE International Software Metrics Symposium, pp.87-94, 2002.

国内会議（査読あり）

- 佐藤 慎一, 端山 毅, 門田 暁人, 松本 健一, “ソフトウェア複雑度メトリクスに基づく大規模システムの欠陥モジュール分類とその評価”, 第 19 回ソフトウェア生産における品質管理シンポジウム報文集, 1999.

研究会／全国大会

- 佐藤 慎一, 小林 孝則, 飯田 元, 井上 克郎, 鳥居 宏次, “プログラムの依存関係解析に基づくデバッグ支援システムの試作”, 電子情報通信学会技術研究報告, No.SS95-4, pp.23-30, 1995.
- 佐藤 慎一, 植田 良一, 井上 克郎, “再帰やポインタを含むプログラムの効率的な依存関係解析法の提案”, 電子情報通信学会技術研究報告, SS95-37, pp. 9-16, 1996.
- 佐藤 慎一, 飯田 元, 井上 克郎, 鳥居 宏次, “プログラムスライスの抽出・実行機能を組み込んだデバッグ支援システムの試作”, 情報処理学会第４９回(平成６年後期)全国大会論文集(5), pp.217-218, 1994.
- 中江 大海, 神谷 年洋, 門田 暁人, 加藤 裕史, 佐藤 慎一, 井上 克郎, “レガシーソフトウェアを対象とするクローンコードの定量的分析”, 電子情報通信学会技術研究報告, No.SS2000-49, pp.57-64, 2001.

目次

1. はじめに 1

2. レガシーソフトウェアにおけるコードクローンの存在とモジュール品質との関係分析 5

2.1. はじめに 5
2.2. 目的とアプローチ 6
2.3. コードクローンの定義と分類 8
2.4. 実験 12
2.5. 実験結果と考察 13
2.6. 信頼性と保守性のトレードオフ 21
2.7. まとめ 22

3. モジュールメトリクスに基づくレガシーソフトウェアの fault-prone モジュール予測の有効性評価 25

3.1. はじめに 25
3.2. Fault-prone モジュール予測方法 27
3.3. 実験概要 27
3.4. 予測結果の評価 34
3.5. 実験結果 37
3.6. 考察 41
3.7. まとめ 47

4. プログラムスライシング・部分評価を用いたデバッグ支援ツール 49

4.1. はじめに 49
4.2. プログラム依存関係解析の概要 51
4.3. 試作ツールの概要 58
4.4. 実行例 61
4.5. 考察 65
4.6. まとめ 65

5. おわりに 67

図目次

図 1-1　ソフトウェア品質確保に関わる作業.....1
図 1-2　研究の位置づけ.....4
図 2-1　コードクローンの生成.....6
図 2-2　2 種類のコードクローンペア.....10
図 2-3　4 種類のモジュール.....11
図 2-4　コードクローンメトリクス.....12
図 2-5　コードクローンカバレッジ.....14
図 2-6　モジュールの分類.....14
図 2-7　コードクローンと信頼性の関係.....15
図 2-8　モジュール種別と信頼性の関係.....16
図 2-9　MAXLEN と信頼性の関係.....17
図 2-10　COVERAGE と信頼性の関係.....17
図 2-11　コードクローンと改版数の関係.....18
図 2-12　モジュール種別と保守性の関係.....19
図 2-13　MAXLEN と保守性の関係.....20
図 2-14　MAXLEN/LOC と改版数の関係.....21
図 3-1　ニューラルネットワークモデル.....33
図 3-2　分類木の例.....34
図 3-3　Alberg Diagram.....37
図 3-4　Alberg Diagram による評価結果（言語 A）.....40
図 3-5　Alberg Diagram による評価結果（言語 B）.....40
図 3-6　分類木（上位のみ）.....45
図 4-1　PDG の元のプログラム.....53
図 4-2　PDG の例.....54
図 4-3　部分評価前のプログラム.....57
図 4-4　部分評価後のプログラム.....58
図 4-5　ツールの構成図.....60
図 4-6　欠陥を含んだプログラム.....62
図 4-7　プログラムの実行結果.....63
図 4-8　スライス選択画面.....63
図 4-9　スライス結果（網掛けの部分がスライス）.....64

表目次

表 3-1　メトリクス一覧 30
表 3-2　各メトリクスの平均と標準偏差 31
表 3-3　分類結果マトリクス 35
表 3-4　評価結果(言語 A) 38
表 3-5　評価結果(言語 B) 39
表 3-6　x%の欠陥モジュールを抽出するためのモジュール数 41
表 3-7　Fault-prone モジュールの抽出に影響が大きかったメトリクス 43
表 3-8　成長を途中でストップさせた分類木の予測精度 46

1. はじめに

ソフトウェアは企業の基幹業務から日常の生活にいたるあらゆるところで用いられており，ひとたび故障が発生すると，その社会的影響は甚大である．ソフトウェア工学の基礎知識を体系化した SWEBOK(Software Engineering Body Of Knowledge)[47]で定義される 10 個の知識領域のひとつに「ソフトウェア品質」が含まれていることからもその重要性がわかる．品質を確保するための具体的な活動として，一般的には，上流工程で仕様書や設計書への欠陥の混入を防ぐためのレビューが行われた後，図 1-1に示すようにコーディングでソースコードが生成され，試験を行って欠陥が検出されれば修正（デバッグ）が行われ，それが繰り返されることによってソフトウェアの品質が確保される．これらの活動は，実際にプログラムを動作させてその結果を検証することによって品質を確保するもので，SWEBOK では動的技法として位置づけられる．ソフトウェアは高品質であるに越したことはないが，高品質を目指せば目指すほどそれに要する時間やコストも増大するため，品質を確保しつつこれらの作業を効率的に行うことも重要である．

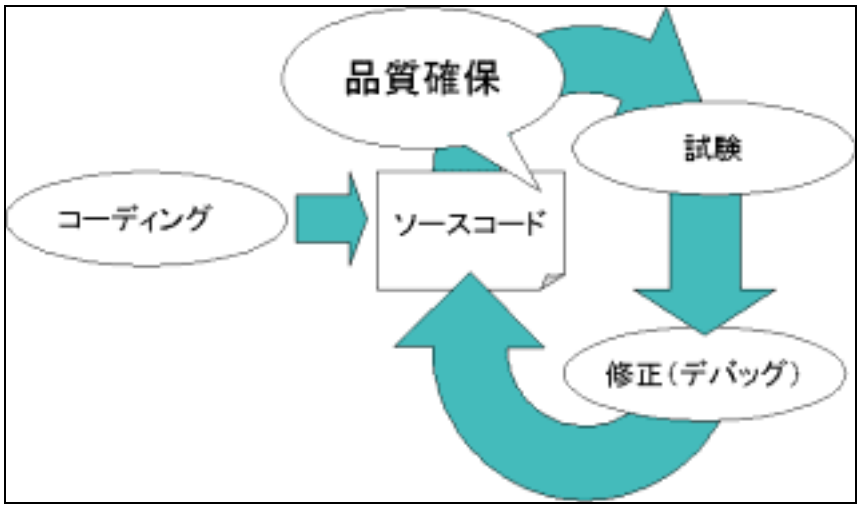


図 1-1　ソフトウェア品質確保に関わる作業

一方 SWEBOK では，動的技法に対応するもうひとつの品質確保技法として静的技法をあげている．静的技法は，プログラムを実行することなく，ソフトウェアの複雑度や制御フローなどを分析してこれらの活動を支援するものである．本論文では，図 1-1に示す品質確保活動の効率化に向けて，静的技法として位置づ

けられるソースコード解析が適用可能であることに着目し，実用規模のソースコードを解析した結果からコーディングにおける品質作りこみの指針および試験において適切なリソースを配分する指針を得るとともに，ソースコード解析技術のひとつである依存関係解析に基づくデバッグ支援ツールを提案する．

まず，実際に運用されたあるレガシーソフトウェア[1]を対象として，ソースコード解析結果とソフトウェア品質の関係を分析した 2 つのケーススタディについて述べる．1 つめのケーススタディでは，ソースコード解析技術としてコードクローン抽出に着目し，レガシーソフトウェアにおけるコードクローンの存在とソフトウェア品質との関係を定量的に分析する．コードクローンとは，ソフトウェア中に含まれる類似もしくは同一のコード片のことであり，近年提案された解析技術によって大規模ソフトウェアにおけるコードクローンを効率よく抽出できるようになった[5]．コードクローンの存在は，一般的にはソフトウェアの保守性・信頼性を低下させる要因のひとつと考えられてきたが，その存在がソフトウェアに及ぼす影響についての定量的な評価は行われていなかった．本研究ではあるレガシーソフトウェアを対象に，モジュールのコードクローン含有率と信頼性および保守性との関係を分析した．分析はモジュール単位で行い，コードクローンを単一モジュールに閉じて存在するものとモジュール間にまたがって存在するものとに分類して，それぞれのコードクローンの含有率と保守性・信頼性との関係を分析した．ソフトウェアの保守性尺度としてはモジュールの改版数を，信頼性尺度としてモジュールの欠陥検出密度を用いた．その結果，コードクローンの生成は，保守コスト増大の原因となり得るが，信頼性の低下を抑える可能性があることがわかった．ただし，200 行を超える大きなコードクローンを生成した場合には，信頼性低下の原因となり得ることもわかった．この結果は，ソフトウェア開発においてコードクローンが保守性・信頼性の与える影響に関しての指針を提供するものである．

もうひとつのケーススタディでは，ソースコード解析技術としてモジュールメトリクス計測に着目し，複数のメトリクス値に基づいてレガシーソフトウェアか

[1] レガシーソフトウェアとは，ユーザからの新たな要求による機能追加やデバッグのための修正が長年にわたって繰り返されたソフトウェアである．そのために，一般的には規模や複雑さが増大していると考えられている．

ら欠陥を含むモジュールを予測する手法の有効性を検証する．これまでにもモジュールメトリクスに基づいて fault-prone モジュールを分類もしくは予測する研究は行われてきたが，本研究では，これまでに研究実績がほとんどない長期間の運用実績がありつつ開発時の成果物やデータが十分に入手できないレガシーソフトウェアの保守工程を想定し，リリース後の限られた情報のみを用いて欠陥を含んでいる可能性の高い(fault-prone)モジュールを予測する手法を適用し有効性を検証した．具体的には，ソースコードからモジュールメトリクスを測定して，リリース後 3 年間に検出された欠陥データに基づいて予測モデルを構築し，その後欠陥が検出されたモジュールをどれだけ正しく予測できたかを評価した．予測モデルとして判別分析，ニューラルネットワーク，分類木の三種類を適用し，それぞれによる予測結果の違いも比較した．本研究の結果，単に欠陥を含むか含まないかを予測するだけでなく，欠陥が含まれる可能性が高い順にモジュールを並べることにより,少ないモジュールで欠陥を含むモジュールの大部分を予測できた．この手法を用いれば，ソフトウェアから fault-prone モジュールを事前に予測することにより重点的に試験対象とすべきモジュールを絞り込めることから，それらを重点的に試験対象とすることによって試験リソースの配分を効率よく行ったり，欠陥を検出して故障発生を未然に防ぐことにより，故障発生時の修正コストを削減できる．

最後に，ソースコード解析技術としてプログラム依存関係解析に着目し，プログラムスライシングおよび部分評価を応用したデバッグ支援ツールを提案するとともに実際に試作を行う．プログラムスライシングとは，プログラムの各文の依存関係を解析して，ある特定の変数に関係のある部分（スライス）を抽出する技術である[80]．また部分評価とは，プログラムに対する入力条件を指定することによって，プログラム中の変数の値を決定していくことにより，プログラムを簡素化する技術である[23][74]．提案するツールは，ステップ実行やトレースなど一般的なデバッグツールが持つ機能以外にプログラムスライスの抽出や部分評価を行って，デバッグの対象となる部分を小さくする機能を持つ．この機能を用いれば，デバッグ対象をプログラム中で欠陥に関係のある部分のみに絞り込むことができるため，デバッグ作業の効率化が期待される．

以上述べた研究内容を図 1-1で示した各品質確保作業と対応させて研究の位置

づけを示すと図 1-2のようになる．このように，各研究の成果は，それぞれの活動に対して指針を提供したり効率化の手法を提案するものである．

これ以降の各章で，各研究の内容について，研究動向も含めて詳しく述べる．まず2章でコードクローンとソフトウェア信頼性・保守性との関係を分析した研究について述べる．次に，3章でモジュールメトリクスに基づく fault-prone モジュール予測の有効性評価，4章でプログラムスライシング・部分評価に基づくデバッグ支援ツールについて述べ，5章で本論文のまとめについて述べる．

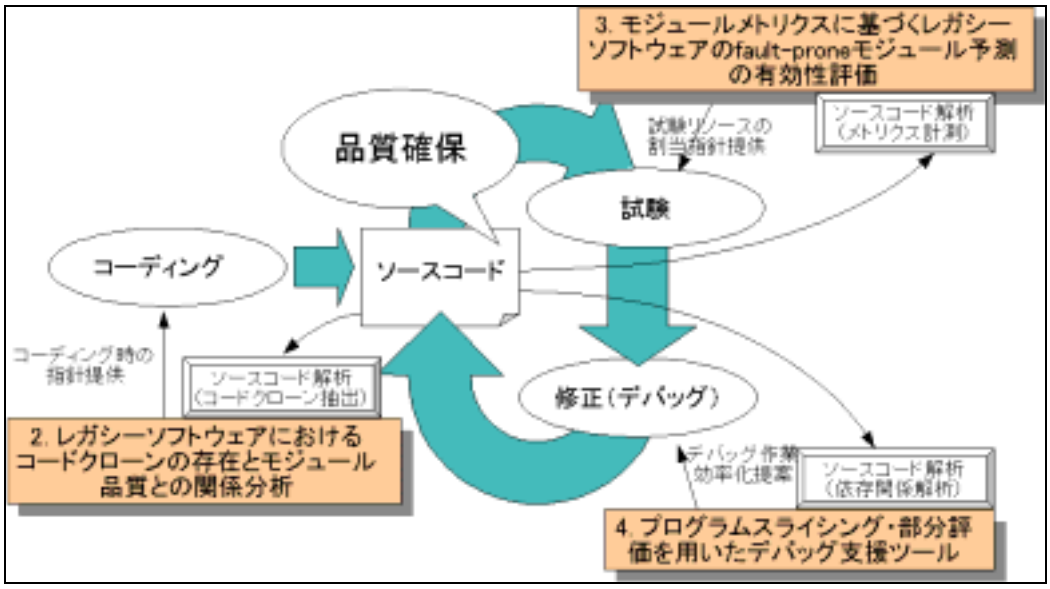


図 1-2　研究の位置づけ

2. レガシーソフトウェアにおけるコードクローンの存在とモジュール品質との関係分析

2.1. はじめに

今日，多くの企業において，レガシーソフトウェアの品質低下による保守コストの増大が問題となっている[48][52][64][68]．本研究では，レガシーソフトウェアの品質低下，及び，保守コスト増大の具体的な要因の一つとして，コードクローン(Code Clone)に着目する．コードクローンとは，ソースコード中の重複したコード列のことであり，主にソースコードのコピー＆ペーストを行うことにより生成される(図 2-1)．コードクローンは，ソフトウェアの構造を複雑にし，ソフトウェア品質を低下させる一因であると言われている．実際に大規模ソフトウェアのソースコードの 5%～60%の部分をコードクローンが占めていた事例も報告されている[10][13][51]．多数のコードクローンを含むソフトウェアでは，重複する多数のコード列の一つのコピーに変更を加える際に，他の全てのコピーにも同様の変更を行わなければならないことが多く，保守コストを増大させる原因となる[13]．さらに，変更し忘れや見逃しが生じた場合には，欠陥が混入し，信頼性を低下させる原因にもなる．しかし，コードクローンと保守性・信頼性の関係はこれまで定量的には明らかにされていない．

本研究では，20 年以上前に開発されたある大規模なレガシーソフトウェアを対象とし，コードクローンと保守性・信頼性の関係を定量的に分析する．ある種のコードクローンが保守性や信頼性に対して特に悪影響を与えることが明らかになれば，そのようなコードクローンを除去したり，できるだけ生じさせないようにフィードバックすることで，保守作業の軽減，及び，保守コストの削減に役立つと期待される．

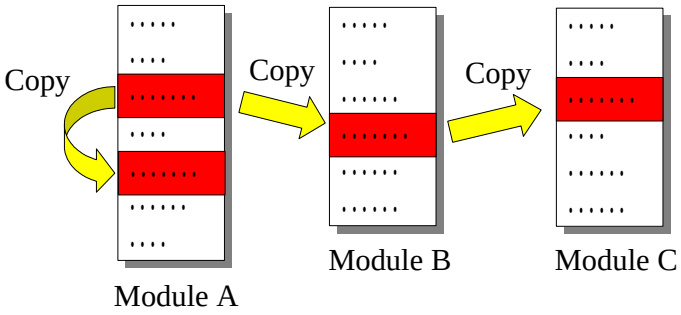


図 2-1　コードクローンの生成

近年，Baker[5]が大規模 C 言語プログラム中のクローンコードを現実的な時間内で検出する手法を提案したことがきっかけとなり，コードクローンの検出方法に関する研究が数多く行われるようになった[6][7][8][10][13][26][30][31][35][37][43]．また，オブジェクト指向プログラムを対象として，クローンコードを除去するための系統的な方法も提案されている[20]．これらの研究は，コードクローンを検出・除去する方法の開発が主目的であるのに対して，本研究では，検出したコードクローンの分析を主目的とする．

以降，2.2節では，本研究の目的とアプローチを詳細に述べる．2.3節では，本研究で採用したコードクローン検出方法，及び，コードクローンに関する定義について述べる．2.4節では，あるレガシーソフトウェアを題材とし，コードクローンと信頼性・保守性の関係を分析した実験について述べ，2.5節で実験結果と考察を述べる．

2.2. 目的とアプローチ

本研究の目的は，大規模レガシーソフトウェアを対象としたケーススタディによって,コードクローンと保守性･信頼性の関係を定量的に分析することである．コードクローンと保守性・信頼性の関係を議論するために，コードクローンをモジュール内の位置およびその大きさ（行数）によって分類するとともに，評価するためのメトリクスを定義する．そして，対象となるソフトウェアからクローンを検出・分類するとともにメトリクスの計測を行い，独立に計測されたソフトウェアの保守性・信頼性メトリクスと比較することで，コードクローンと保守性・信頼性の関係を評価する．

多くのソフトウェアシステムでは，保守プロセスデータがモジュール毎に計測

されている．たとえば，故障の発生と修正，機能追加，改版数，作成日時，更新日時などの情報は，モジュール毎に記録されていることが多い．ソースコードの行数や変数の個数を始めとするプロダクトデータも，モジュール単位での計測が可能である．そこで，本研究では，モジュール単位での分析を行うことにした．

分析に際しては，分析対象となるソフトウェアの各モジュールについて，信頼性と保守性を予め推定しておく必要がある．本研究では，信頼性を推定する一つの方法として，保守工程において過去に検出された欠陥数を用いる．過去に多くの欠陥が検出されたソフトウェアモジュールは，欠陥検出数の少ないモジュールよりも信頼性が低かったといえる．

保守性は，保守に要した労力（人月や人日）を計測することで推定できるように思われる．しかし，本研究で対象としているような複数の開発者・保守作業者が関わるソフトウェアの場合には，ある保守作業者が特定の作業に要した時間を計測したとしても，保守性を計測したことにならない．保守に要した時間を計測することは，保守性の計測だけでなく，保守作業の量の計測，作業者の性質の計測，作業の能率の計測，作業環境の計測などを含むためである[64].

本研究では，単純ではあるが実践的な一つの解決策として，ソフトウェアに含まれる各モジュールの改版数（リビジョン数）を用いる．一般に，機能追加，機能変更，修正等のためにモジュールに変更が加えられる（改版される）と，そのモジュールの改版数が 1 増加する．改版にはコストを要するため，改版数の大きなモジュールは小さなモジュールよりも，平均的により多くの保守コストが費やされてきたといえる．従って，改版のための保守コストを要するか否かという観点でモジュールの保守性を評価する場合には，改版数はモジュールの保守性の指標となると考えられる．Eick らは 1000 万行規模のレガシーソフトウェアを分析した結果，改版が進むにつれて Code Decay（コードの劣化）が進行することを示していることからも[18]，改版数はコードの劣化の指標にもなり得る．

保守性を推定するもう一つの方法として，ソフトウェアの複雑さを表すメトリクス(Complexity Metrics)を用いることも考えられる．これまで，複雑なソフトウェアほど保守性が低いと考えて，cyclomatic 数[48]，ソフトウェアサイエンス尺度[27]，オブジェクト指向言語における Chidamber と Kemerer の尺度[13]を始めとして，数多くのメトリクスが提案されている[12][19][26][49]．しかし，こ

れらのメトリクスは，コードクローンの有無とは本質的に独立であり，コードクローンの存在により評価できる保守性とは異なる保守性の側面を表現している．例として，あるモジュールを計測した結果，「１行あたりのサイクロマティック数」が極めて大きく，保守性が低いと予測されたとしても，このモジュール中にコードクローンが多く含まれていた，もしくは，含まれていなかった，のいずれの場合においても，コードクローンの有無とサイクロマティック数が独立であるため，コードクローンの有無と保守性の関係を明らかにすることはできない．従って，コードクローンと保守性の関係の分析においては，従来の Complexity Metrics を用いずに，モジュールの保守性を推定する必要がある．

2.3. コードクローンの定義と分類

2.3.1. コードクローン

コードクローンとは，ソースコード中の重複したコード列のことであり，ソフトウェアの開発者や保守作業者がコード列をコピー＆ペーストするなどのいくつかの原因によって作られる．コピー＆ペースト以外の原因としては，コードジェネレータによって生成されたコード，特定のコーディングスタイルによるもの，パフォーマンスを稼ぐための意図的な繰り返し，偶然の一致などがある[10]．コピー＆ペーストによって作られたコードクローンの場合には，コード列に部分的な変更を加えることも多く，そのような部分的に異なる類似のコード列の組もコードクローンと見なすのが一般的である．ただし，どのようなコード列がコードクローンと見なされるかについての正確な定義は，これまで提案された数多くのクローン検出手法やツールごとに異なる[5][6][7][8][10][17][30][31][34][35][37][43][49].

本研究では，神谷らが提案したトークンベースのクローン検出方法[34][35]を用いる．この方法は，プログラミング言語の構文規則に基づいた処理を行い，コード列中の空白やコメント，インデントが異なったり，あるいは，変数名などが書き換えられたりした場合も，コードクローンとして検出する．この手法を実装したツール CCFinder は，COBOL 等のプログラミング言語で記述されたレガシーソフトウェアへの適用が可能である．この方法によるクローン検出手順の概略を次に示す．

(1) 字句解析

入力として与えられたソフトウェアに含まれる全てのソースファイルを，プログラミング言語の字句規則に従ってトークンに分割する．ソースファイル中の空白やコメントは無視される．

(2) トークン変換

型，変数，定数に属するトークンは，同一のトークンに置き換えられる．この置き換えにより，例えば，変数名だけが異なるコード列の組をコードクローンとして検出できるようになる．

(3) マッチング，及び，フォーマッティング

変換後のトークン列に含まれる全ての部分列の集合から，同一の部分列の組を探し出してコードクローンとして検出する．検出にあたっては Suffix Tree マッチングアルゴリズム[26]を用いることで，ソフトウェアのサイズ n に対して O(n)の計算量で検出できる．最後に，検出されたトークン列の位置情報を元のソースファイル上の行番号に変換し，出力する．

2.3.2. コードクローンおよびモジュールの分類

本研究では，コードクローンとして検出された互いに等しい（もしくは類似する）二つのコード列の組を「コードクローンペア」と呼ぶ．コードクローンペアを，それらが存在するモジュール内（間）での位置によって，次の二種類に分類する（図 2-2参照）.

- In-module クローンペア

  コードクローンペアを構成する二つのコード列の両方が同一のモジュールに存在する．

- Inter-module クローンペア

  コードクローンペアを構成する二つのコード列がそれぞれ異なるモジュールに存在する．

それぞれのコードクローンペアは，ソフトウェア品質に対する影響が異なると考えられる． Inter-module クローンペアは，類似の処理を行うコード断片が二つの複数のモジュールにまたがるため，モジュール間の結合度を増大させる可能性がある一方，In-module クローンペアは，モジュール間の結合度にさほど影響を与えない可能性がある．

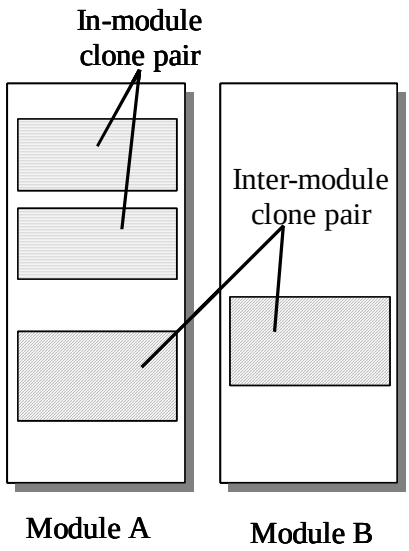


図 2-2　2 種類のコードクローンペア

次に，上記のコードクローンの分類に基づいて，ソフトウェア中のモジュールを以下の種類に分類する(図 2-3参照).

- Non-clone モジュール

  コードクローン列を全く含まないモジュールである.

- Clone-included モジュール

  一つ以上のコードクローン列を含むモジュールである．さらに次の三つの種類に分類される.

  - Closed モジュール

    コードクローン列として，In-module クローンペアに属するもののみを含むモジュールである.

  - Related モジュール

    コードクローン列として, Inter-module クローンペアに属するもののみを含むモジュールである.

  - Composite モジュール

    In-module クローンペアに属するコードクローン列と, Inter-module クローンペアに属するコードクローン列の両方を含むモジュールである.

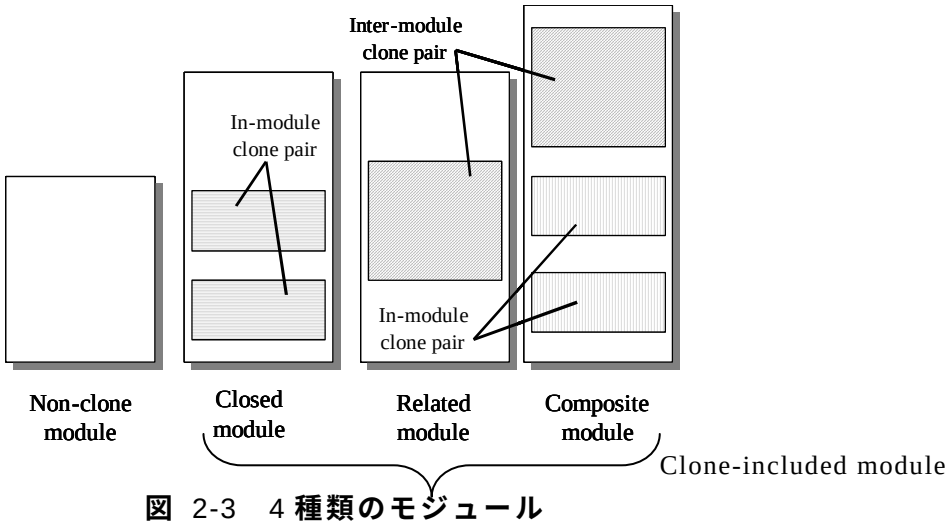


図 2-3　4 種類のモジュール

2.3.3.　コードクローンメトリクス

検出されたコードクローンに基づいて各モジュールを評価するための尺度（コードクローンメトリクス）を定義する．ここでは，二つのコードクローンメトリクスを次の通り定義する．

- MAXLEN (最大コードクローン行数：Maximum length of code clone)
  モジュールに含まれるコードクローン列のうち，最大のものの行数．長いコード列のコピー&ペーストを行うと，この値が大きくなる．
- COVERAGE (コードクローンカバレッジ：Coverage of code clone)
  モジュールに含まれる行のうち，コードクローン列に含まれる行の割合（パーセントで表す）．一つのモジュール全体をコピー&ペーストした場合には，コピー元のモジュールの COVERAGE は 100%となる．

各メトリクスの例を図 2-4に示す．図中，モジュール B に含まれる二つのコードクローン列は 20 行であるため,モジュール B の MAXLEN は 20 となる.また，モジュール D に含まれる三つのコードクローン列のうち最大のものは 40 行であるため，モジュール D の MAXLEN は 40 となる．

一方，図 2-4のモジュール B の大きさは 80 行であり，いずれかのコードクローン列に含まれる行は 40 行(＝20＋20)であるため,モジュール B の COVERAGE は 50%(＝40÷80×100)となる.同様に,モジュール D の大きさは 100 行であり，いずれかのコードクローン列に含まれる行は 80 行(＝40＋20＋20)であるため，モジュール D の COVERAGE は 80%(＝80÷100×100)となる．

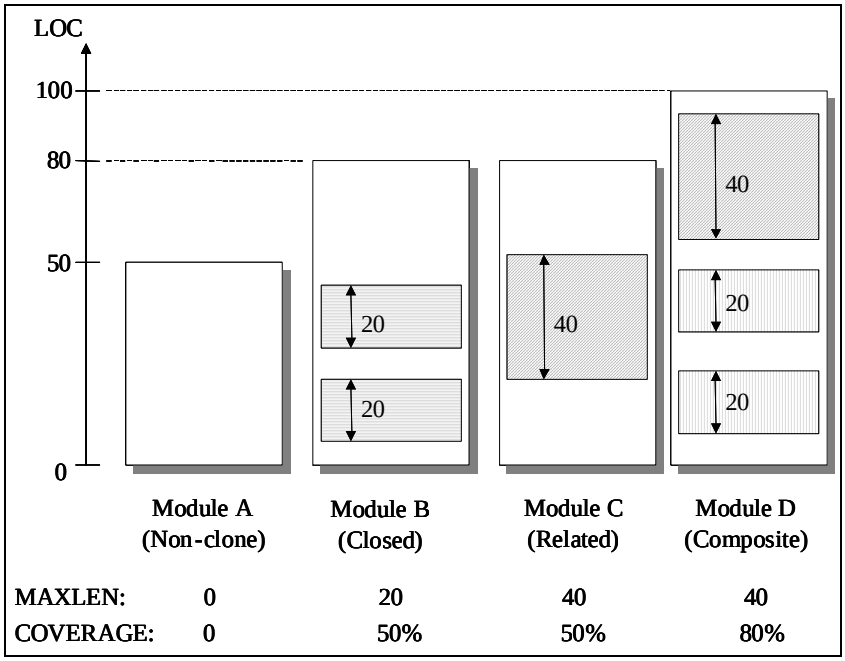


図 2-4　コードクローンメトリクス

2.4. 実験

2.4.1. 実験の目的

本実験の主目的は，ある大規模レガシーソフトウェアを題材として，次の二点について分析することである．

- モジュールの信頼性（欠陥数）とコードクローンの関係
- モジュールの保守性（改版数）とコードクローンの関係

本実験では，上記の二つの関係の分析に先立って，ソフトウェア中のコードクローンの分布についても調査した．

2.4.2. 分析対象のソフトウェア

本研究では,ある企業で実際に運用されたレガシーソフトウェアを対象とした．本ソフトウェアはある組織の業務ソフトウェアで,20 年以上前に初期バージョンがリリースされた．Capers Jones による分類[32]に従うと，MIS(Management Information System)に該当する．MIS は，企業，官公庁，銀行等において業務の基盤となるアプリケーションソフトウェアであり,例えば,給与計算システム，会計処理システム，銀行業務システム，保険業務システム，航空予約システム等が該当する．本ソフトウェアを含む多くの MIS の特徴は，大規模で，データベー

スを含み，メインフレーム上で稼働することである．総規模は約 1MSLOC で，モジュールの規模は，最小のもので 9 行，最大のもので 7103 行，平均は 631 行であった．

2.4.3. データの計測

本実験では，各モジュールの欠陥数として，分析対象ソフトウェアの対象バージョンがリリースされた後 6 年間の運用中に検出された値を用いた．モジュールの信頼性の尺度としては，モジュールの規模の要因を排除するために，欠陥数をモジュールの行数で割った値「1 行あたりの欠陥数」を用いた．単なる欠陥数を信頼性の尺度として用いると，規模の大きなモジュールほど検出欠陥数が多くなる傾向にあり，コードクローンと信頼性の関係の分析が困難になると判断した．なお，6 年間に欠陥が検出されたモジュールは全体の約 14%であった．

各モジュールの改版数は，リリース時点で計測された数値を用いた．平均値は約 50，最小値は 0，最大値は 338 であった．ソフトウェアの改版としては，単なる仕様変更以外にも，データパッチ等で逃げていたものを正式にモジュール反映するもの，扱うデータ規模（データベース規模）の拡大によるもの，外部インタフェースの変更によるもの等が含まれる．

コードクローンの検出にあたっては，偶然に一致する（コピー＆ペーストを原因としない）コード列の組がコードクローンとして検出されることをなるべく避けるために，30 行以上一致するコード列の組をコードクローンとして検出することとし，30 行未満の一致列は無視した．

2.5. 実験結果と考察

2.5.1. コードクローンの分布

COVERAGE とモジュール数の関係を図 2-5に示す．図に示されるように，約 50%のモジュールは COVERAGE＞0，すなわち，clone-included モジュールであった．類似するシステムの過去の事例では，COBOL で記述された Payroll システムが 59%のコードクローンを含んでいたという結果が報告されており，今回の対象システムでも同様の傾向が見られる[17].

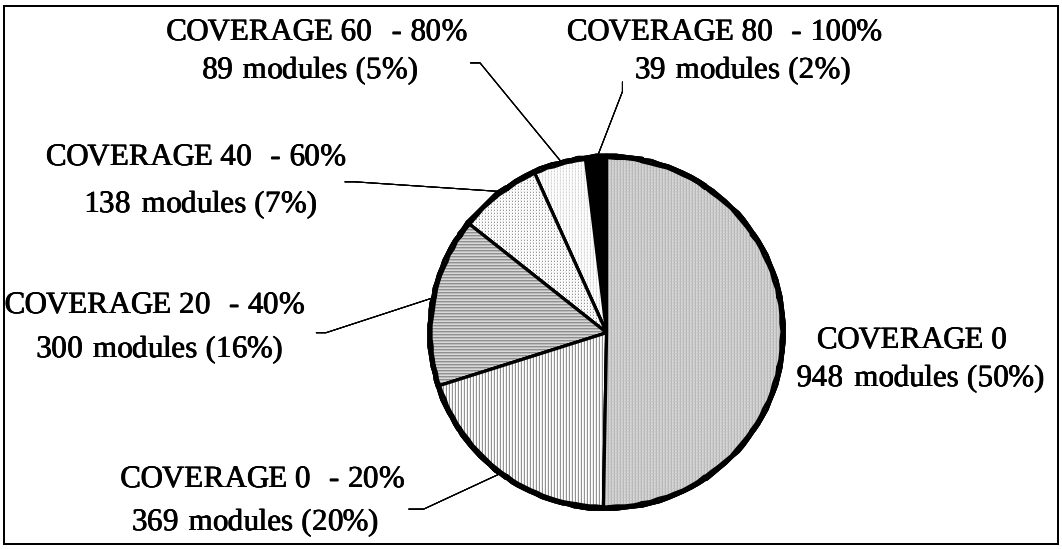


図 2-5 コードクローンカバレッジ

検出されたコードクローンに基づいて2.3.2節で述べた基準に従ってモジュールを分類した結果を図 2-6に示す．図に示されるように，コードクローンを含むモジュール(clone-included モジュール)のうち，closed モジュールの占める率は7%であったのに対して，related モジュールは 34%占めた．つまり，検出されたコードクローンの大部分は inter-module クローンであった．

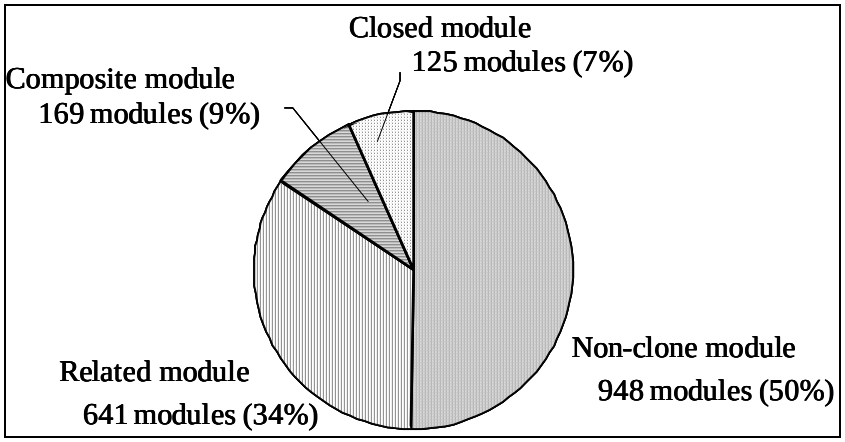


図 2-6 モジュールの分類

2.5.2. コードクローンと信頼性の関係

Non-clone モジュールと clone-included モジュールの 1 行あたりの欠陥数の平均値を図 2-7に示す．図より，clone-included モジュールは，non-clone モジュ

ールよりも信頼性が平均的に約 40%高い（有意水準 5%で有意差あり）．2.1節では，一つのコードクローン列に変更を加える際には，同一の全てのコードクローン列にも同様の変更を行わなければならないことが多く，変更し忘れや見逃しが生じると欠陥混入の原因となりうると述べたが，その逆の結果が得られたことになる．つまり，このソフトウェアにおいては，このようなコピー＆ペーストによるコード生成を行っているが，そうすることによって，一からコードを記述するよりも，変更し忘れや見逃しによる欠陥混入の可能性を減らせる可能性があることを示した．もう一つの解釈としては，このようなコードクローンは，新しい種類の機能を含まないため，未知の種類の欠陥を混入させる機会が少なかったことも考えられる．

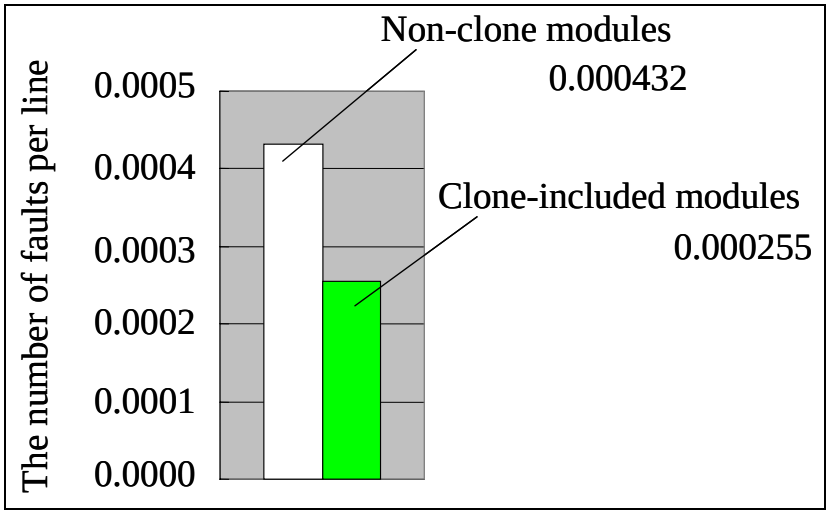


図 2-7　コードクローンと信頼性の関係

さらに，clone-included モジュールについて，各分類の 1 行あたりの欠陥数を図 2-8に示す．コードクローンを含むモジュールのうちでは，related モジュールが最も信頼性が高かったが，統計的な有意差は認められなかった．

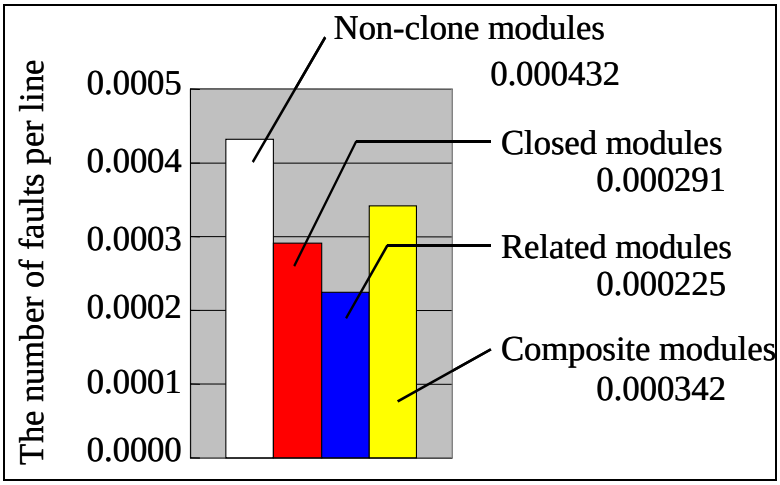


図 2-8　モジュール種別と信頼性の関係

次に，コードクローンメトリクスと信頼性の関係の分析として，MAXLEN と 1 行あたりの欠陥数の関係を図 2-9に示す．MAXLEN の値が大きくなるに従って信頼性は次第に向上していくが，200 行以上の長さのコードクローン列を持つモジュール群（MAXLEN≧200）では，逆に，最も低い信頼性を示した（有意水準 5%で有意差あり）．このことから，一定以上のサイズのコードクローンを生成することは，信頼性を低下させる原因となると予想される．なお，MAXLEN を行数で正規化した値（MAXLEN / LOC）と 1 行あたりの欠陥数の間には，明確な関係は見られなかった．

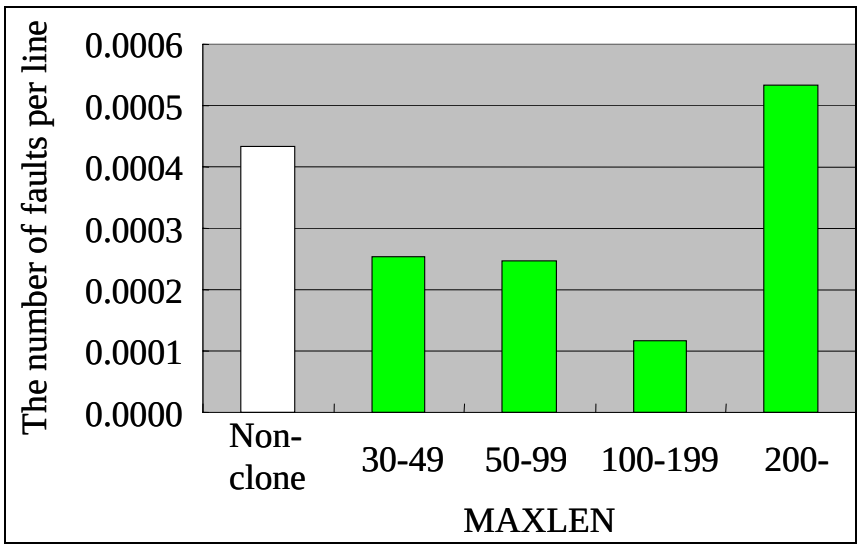


図 2-9　MAXLEN と信頼性の関係

COVERAGE と 1 行あたりの欠陥数の関係を図 2-10に示す．COVERAGE が 0 より大きく 80%以下の場合，non-clone モジュールよりも信頼性が平均的に高く（有意水準 5%で有意差あり），COVERAGE が 80%を超えると non-clone モジュールの 2 倍程度の信頼性となったが，有意差は認められなかった．

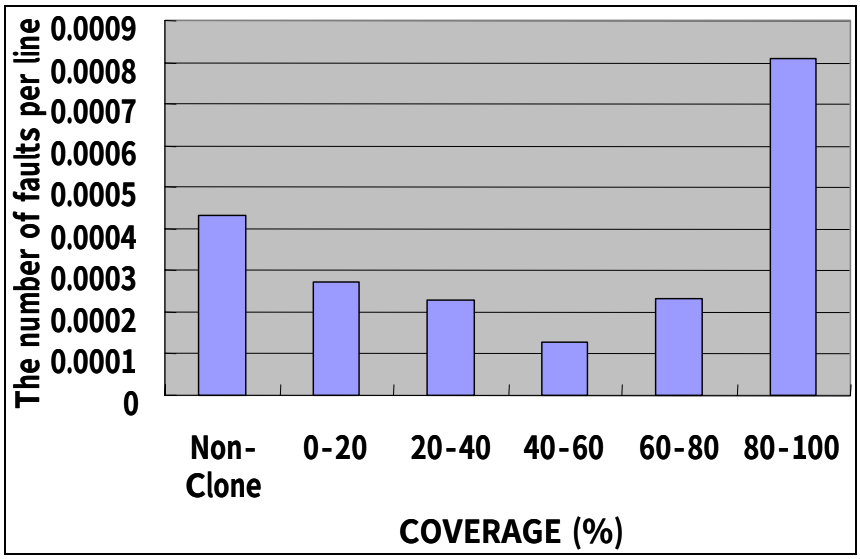


図 2-10　COVERAGE と信頼性の関係

このソフトウェアの保守関係者へのインタビューで判明したこととして，コードクローンが生成される一つのケースに，変更のリスクが高い部分のコード列に

対して機能追加を行う際，変更したい部分のコード列からコピー＆ペーストしてコードクローンを生成し，コピー元のコード列からペースト後のコード列へと処理を分岐させた上で,ペースト後のコード列に変更を加えるというものがあった．このようにコードクローンを生成してから変更を行うことは，ソフトウェアの構造を複雑にするというデメリットがある一方で，変更による欠陥混入のリスクを低減できる可能性がある．特に，長年の保守の過程で保守作業者の入れ替わりが起こり，ソフトウェア全体を熟知した人間がいない場合には，コードクローンの生成は欠陥混入のリスク回避の一つの手段となり得る．

2.5.3. コードクローンと保守性の関係

Non-clone モジュールと clone-included モジュールの改版数の平均値を図2-11に示す．図より，clone-included モジュールは，non-clone モジュールよりも改版数が平均的に約 40%大きいといえる（有意水準 5%で有意差あり）．従って，このソフトウェアが 20 年以上前に初期バージョンがリリースされて以来，clone-included は non-clone モジュールと比べて平均的により多くの改版コストを要したと推測される（clone-included，non-clone の各モジュール群における 1 回当たりの改版コストの平均に偏りがないと仮定した場合）．

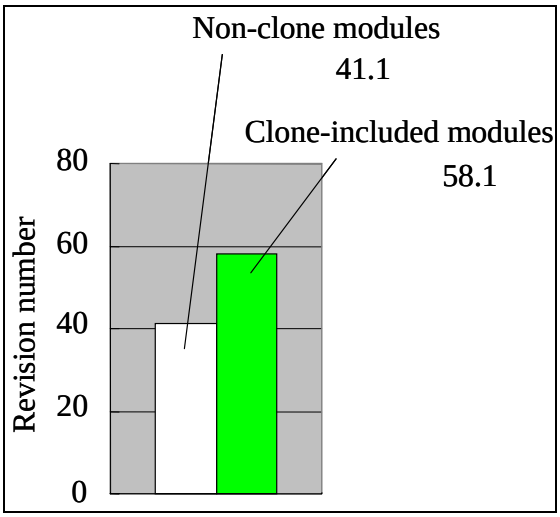


図 2-11 コードクローンと改版数の関係

この結果に対しては，二通りの解釈が考えられる．一つは，コードクローンの存在が改版数の増大の原因となったという解釈である．もう一つは，改版数の大きいことがコードクローン生成の原因となったという解釈である．現時点では，

後者の解釈には合理的な説明が与えられないが，前者については次のような説明が考えられる．2.1節で述べたように，一つのコードクローン列に変更を加える際には，同一の全てのコードクローン列にも同様の変更を行わなければならないことが多い[17]．このとき，変更が加えられた全てのモジュールにおいて改版数が増加するため，clone-included モジュールの集合全体の平均改版数を飛躍的に増大させる原因となる．従って，改版に要する保守コストの観点からモジュールの保守性を評価すると，clone-included モジュールは non-clone モジュールよりも平均的に保守性が低いといえる．

次に，各種類のモジュールの改版数を図 2-12に示す．Closed モジュール，related モジュール，及び，composite モジュールのどれもが，non-clone モジュールよりも平均的に改版数が大きいといえる．さらに，コードクローンを含むモジュールのうちでは，closed モジュールが最も改版数が大きかった（ただし，closed と related の間に有意水準 5%で有意差があるが，closed と composite，及び，related と composite の間には有意差なし）．図 2-9で示したように，closed モジュールは related モジュールと比較して，平均的により多くの欠陥を発生させたため，欠陥修正のための改版が多くなった可能性がある．

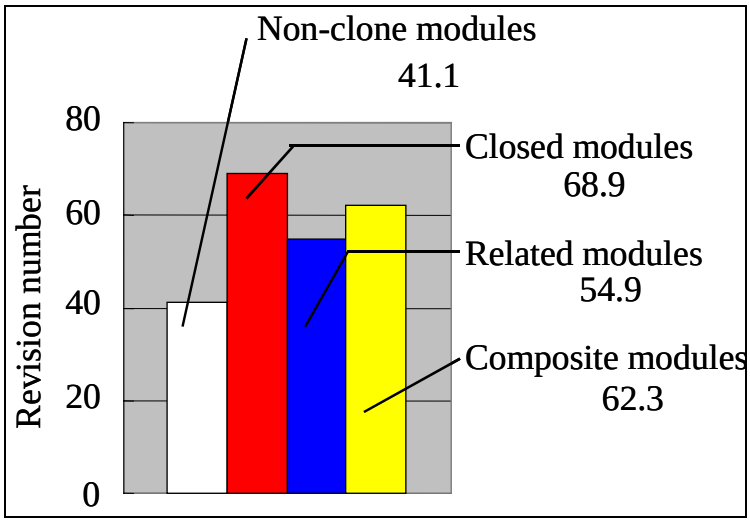


図 2-12　モジュール種別と保守性の関係

コードクローンメトリクスと保守性の関係の分析として，MAXLEN と改版数の関係を図 2-13に示す．MAXLEN の値が大きくなるに従って平均の改版数は次第に増大する傾向があった（Non-clone モジュールとその他のモジュール群との

間に有意水準 5%で有意差あり．MAXLEN が 30～49 と 50～99 のモジュール群の間には有意差なし．MAXLEN が 100～199 のモジュール群は，他のモジュール群との間に有意差あり．同様に，MAXLEN が 200～のモジュール群についても，他のモジュール群との間に有意差あり）．このことから，サイズの大きなコードクローンを生成することは改版コストを著しく増大させる可能性がある．

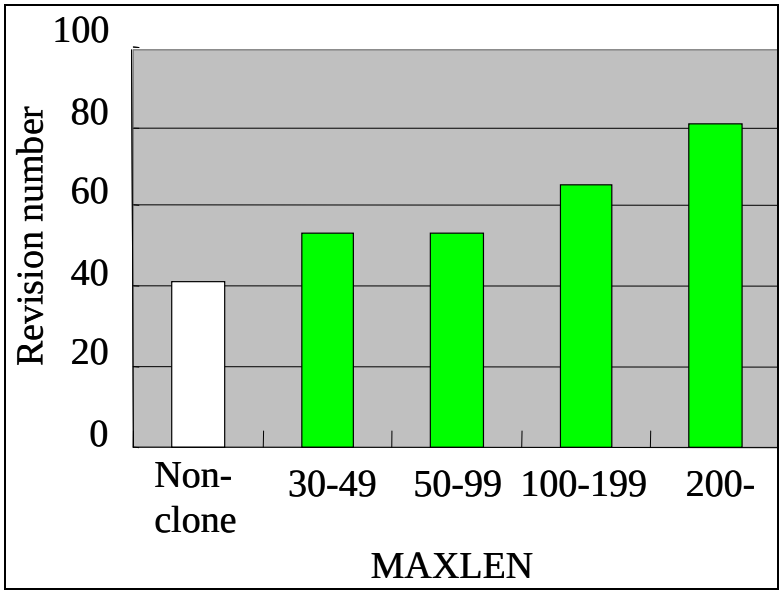


図 2-13　MAXLEN と保守性の関係

MAXLEN をモジュールの行数で正規化した値（MAXLEN / LOC）と改版数の関係を図 2-14に示す．図より，MAXLEN/LOC が 0 より大きく 0.2 以下の場合，non-clone モジュールよりも改版数が平均的に大きい（有意水準 5%で有意差あり）ことが分かった．特に，「MAXLEN/LOC が 0～0.1 の範囲にあり，MAXLEN が 100 以上」という条件を満たすモジュール群では改版数の平均値は 117 となり，non-clone モジュールにおける平均改版数 41 と比較して非常に大きな値を取った．なお，COVERAGE と改版数の間には，明確な関係は見られなかった．

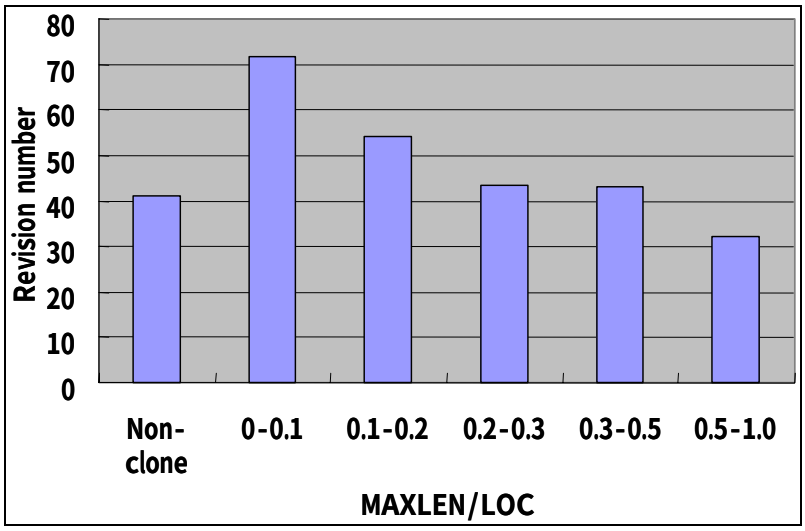


図 2-14　MAXLEN/LOC と改版数の関係

このソフトウェアの保守関係者へのインタビューでは，機能追加，機能変更，及び，修正などを行った際に，「横並びチェック」と呼ばれる作業を行っていることが分かった．この横並びチェックでは，欠陥が発生したり変更を加えた部分のコード列と類似の機能を持ったコード列を特定し，同様の変更が必要となるかどうかを判断する．例えば，同じようなマクロ呼び出しはしていないか，同じようなデータベースアクセスをしている箇所はないか，同じような臨界値処置をしていない部分はないか，同じロジックを作り込んでいないか等を調査する．このようなチェック対象となる類似の処理はコードクローンを形成していることも多く，コードクローンの存在が，横並びチェック，及び，改版のためのコストを増大させることが伺える．

2.6. 信頼性と保守性のトレードオフ

前節までの結果より，コードクローンを生成することは，信頼性の低下を抑える可能性があるが，一方では保守コスト増大の原因となり得ることが分かった．従って，コードクローンの生成においては，信頼性と保守性はトレードオフの関係にあるといえる．ただし，図 2-9と図 2-13に示されるように，200 行を超える大きなコードクローンを生成した場合には，信頼性と保守性の両方を低下させるため,そのような大きなコードクローンはできる限り生成しないことが望ましい．

以上により，信頼性の確保と保守性の確保を両立させるためには，著しく欠陥混入のリスクの高いモジュールに変更を加える場合にのみコードクローン生成に

よってリスクを回避し，それ以外の変更時にはできる限りコードクローンを生成させないことが望ましいと考えられる．

2.7. まとめ

本研究では，20 年以上前に開発されたある大規模なレガシーソフトウェアを題材とし，コードクローンと信頼性の関係，及び，コードクローンと保守性の関係を分析した．主な分析結果は次の通りである．

- 信頼性に関する分析結果
  - Clone-included モジュールは，non-clone モジュールに比べて，平均的に 40%信頼性が高い(1 行あたりの欠陥数が少ない)．
  - ただし，200 行を超える大きさのコードクローンを含むモジュール(MAXLEN≧200)は，non-clone モジュールよりも平均的に信頼性が低い．
- 保守性に関する分析結果
  - Clone-included モジュールは，non-clone モジュールに比べて，平均的に 40%改版数が大きい．従って，改版のためにより多くの保守コストが費やされてきたと考えられる．
  - MAXLEN の値が大きいほど(すなわち，モジュールに含まれるコードクローンのサイズが大きいほど)，改版数がより大きい傾向にある．
  - MAXLEN/LOC が 0 より大きく 0.2 以下の場合，non-clone モジュールよりも平均的に改版数が大きいことが分かった．

以上の結果から，コードクローンの生成は，信頼性の低下を抑える可能性があるが，一方では保守コスト増大の原因となり得るため，信頼性と保守性はトレードオフの関係にあるといえる．ただし，200 行を超える大きなコードクローンを生成した場合には，信頼性と保守性の両方の低下の原因となり得ることがわかった． そのため，この結果から推奨される今後の保守作業の指針としては，大きなコードクローンを生成しないこと，及び，著しく欠陥混入のリスクの高いモジュールに変更を加える場合にのみコードクローン生成によってリスクを回避し，それ以外の変更時にはできる限りコードクローンを生成させないことが望ましいと考えられる．

なお，本研究の結果は，今回対象としたソフトウェアに限定した結果であり，

異なる言語で記述されたり，異なる特徴を持ったソフトウェアにおいては，異なる結果が得られる可能性がある．今回対象としたソフトウェアは 20 年以上にわたって保守され，現在も変更が加えられているソフトウェアである．そのため，変更に伴って新規に欠陥が混入することもある．このようなソフトウェアの場合には，変更を行う際に，新規に作るよりも他のモジュールに存在する（長年の保守を経て品質が高まっている）コードをコピーし，それを編集してコーディングを行うことによって品質の低下を抑えられることは十分に期待できる．実際にこのソフトウェアの保守担当者にヒアリングを行ったところ，このソフトウェアは定型業務が多く開発者はテンプレートに従って開発を行うこともあるので，その部分がコードクローンとして抽出されれば，そのモジュールの品質が高くなるのは理解できるとのコメントも得られた．ただし，同じレガシーソフトウェアでも変更が少ないソフトウェアでは，新たに欠陥が混入することも少ないため，コーディング時にコードをコピーすることによる品質低下抑止の効果はあまり得られない可能性がある．また，新規に開発されるソフトウェアの場合には，開発中に類似コードをコピーしたとしても，コピー元のプログラムも新規に開発されたものであるため必ずしも品質が良いとは限らないので，こちらもコードクローンによる品質低下抑止の効果を得られず，異なる結果が得られる可能性がある．現在も COBOL もしくは拡張 COBOL で記述されたレガシーソフトウェアは数多く存在しており，また，新規に開発されるソフトウェアに COBOL 言語が用いられる場合も少なくなく[12]，本研究の結果は，それらのソフトウェアの開発者，保守作業者にとって有用であると考える．今後同様の追実験を重ねていくことで，結果の信頼性を高めていくことが必要となる．

また，本研究ではコードクローンの生成により信頼性と保守性の間にトレードオフの関係があることを示したが，本来は信頼性の低下を抑えつつ保守性も向上させることが望ましい．そのための方法として，コーディングの際，信頼性の低下を抑えるためにコピー＆ペーストしてコードを生成しておき，その後クローンとなっているコードを共通化し，ライブラリ化もしくは共通モジュール化する方法が考えられる．リファクタリング[20]では重複コードの問題について指摘するとともに，それを共通化するための手順を提示しているが，オブジェクト指向言語を想定しており，今回対象としたレガシーソフトウェアにおける重複コードの

共通化手順は提供していない．保守担当者にヒアリングしたところ，現在問題なく動作しているソースコードに手を加えて共通化することは新たな作業が発生する上，仮に実施したとしてもそれによる新たな欠陥の混入の可能性もあるため，実行するのは難しいとのことであった．また，仮に共通化した場合でも，それによってモジュール結合度が高くなって複雑さが増すことも考えられるため，一概に共通化すればよいというものではなく，様々な観点で総合的に考える必要がある．

別の方法として，コードクローンの存在を前提とし，開発プロセスとしてその存在箇所をきちんと把握しておくことによって保守性の低下を抑える方法も考えられる．2.5.3節で述べた「横並びチェック」はそのような作業の一例といえるが，このような作業を効率よく行えるように，コードクローンの箇所を必要に応じて効率よく特定するとともに利用者に理解しやすい形で提示するツール[2]を保守作業の中で用いることなどが考えられる．

本研究で分析されたコードクローンは，コピー＆ペーストで生成されたものであるが，コードクローンが生成された原因によって，信頼性や保守性に与える影響も異なると考えられる．例えば， GUI ビルダツールによって自動生成されたプログラムコードや Flex や Bison 等の字句・構文解析器生成ツールを用いた場合には，コードクローンを多く含むコードが生成される．今後，より多くの種類のソフトウェア及びコードクローンに対して分析を深めていく必要がある．

[2] このようなツールの例として CloneWarrior(http://se.naist.jp/clonewarrior/)がある．

3. モジュールメトリクスに基づくレガシーソフトウェアの fault-prone モジュール予測の有効性評価

3.1. はじめに

レガシーソフトウェアは長期間にわたって使いつづけられ，その間故障発生に伴う欠陥修正や，機能追加に伴う多くの変更が加えられる．また，何年かおきに大規模なシステム更改が行われ，ハードウェアの拡張や機能の変更などを行ったバージョンがリリースされる．ソフトウェアを構成する各モジュールはシステム更改を契機に作り直されることもあれば，そのまま再利用されることもある．このようなことを繰り返すうちにモジュールが開発された時点でのデータは消失しているか陳腐化する場合が多いため,直近のデータしか利用できないことが多い．

また，長期間の運用中に，多くの変更が加えられているため，その構造が複雑になるとともに欠陥（フォールト）が混入し，検出されないまま潜んでいる可能性もある．レガシーソフトウェアは，現在でも多くの企業や組織で使用され，重要な業務を担っていることが多いため，これらの欠陥が顕在化すればその被害は甚大なものとなる．そこで，全モジュールの中から欠陥を潜在的に含む可能性の高い(fault-prone)モジュールを予測できれば，それらに注力して試験を行うことによって欠陥を効率よく発見でき，故障を未然に防ぐことができるため非常に有用である．

これまでにモジュールの様々なメトリクスを計測し，それらの値に基づいてソフトウェアから fault-prone モジュールを分類する研究が行われてきた．その一般的な手順は，従来から提案されているモジュールメトリクス（行数，ソフトウェアサイエンス尺度[27]，cyclomatic 数[48]など）と各モジュールから検出された欠陥数を計測し，それらの値に基づいて fault-prone モジュールを分類するモデルを構築し，分類精度を評価するものである．分類モデルとしては分類木や各種の多変量解析手法およびニューラルネットなど様々な手法が用いられてきた．Porter らはエントロピーを用いた分類木生成手法を提案し，NASA の 16 プロジェクトのソフトウェアを対象に行った分類実験で 79%の正答率を示した[60]．高橋らは，AIC[3]を用いて木のノード数が少なくなるように Porter らの分類木生成手法を改良し，85KSLOC のソフトウェアを対象に行った分類実験の結果，大きく精度を落とすことなく fault-prone モジュールを分類できることを示した

[70]．Pighin らは，判別分析を用いて 350KSLOC のソフトウェアを対象に行った分類実験の結果，最良のモデルで 96%の正答率を示した[58]．高橋らは，同じく判別分析だが変数選択の際に AIC を用いて最良のモデルを選択する手法を提案し，78 モジュールのソフトウェアを対象に実施した予測実験では 96%の正答率を示した[72]. Khoshgoftaar らはニューラルネットを用いて分類モデルを構築し，大規模ソフトウェアを対象とした分類実験の結果 74%の正答率を示した[39]．彼らはさらにロジスティック回帰分析による分類実験も行い，1643 モジュールのソフトウェアを対象にした分類実験の結果 73%の正答率を示した[40]. このように，これまでの研究で用いられた分類モデルも対象となったソフトウェアの規模は異なるが，いずれの研究結果においても高い精度で fault-prone モジュールを分類しており，モジュールメトリクスに基づく fault-prone モジュール分類手法の有効性を示している．しかし，従来の研究では開発時のデータが揃ったソフトウェアを対象としていることが多く，直近のデータしか使えないレガシーソフトウェアにおける検証は十分ではない．また，従来研究の多くは対象ソフトウェアのモジュールをモデル構築用と評価用に分割して分類精度を評価しているが，実際にこのようなモデルを用いる際には，ある時点までに収集されたデータを用いてモデルを構築し，今後欠陥が検出される可能性のある fault-prone モジュールを予測する方が実用的である．このような実際の開発・保守作業を想定したモデルの予測精度を評価した事例も少ない．

本研究では，実際のレガシーソフトウェアを対象として fault-prone モジュールを予測するケーススタディを行い，その有効性を検証した．その際，保守工程での適用を想定し，モデル構築の際には直近のデータのみを用いた．また，モデル構築の際は，3 種類の構築手法（判別分析，分類木，ニューラルネット）を適用し，手法による予測結果の違いについても評価した．さらに，モジュールが fault-prone か fault-prone でないかを単純に予測した結果だけでなく，実際の保守作業での有用性を想定して，予測結果のモジュールを fault-prone である可能性が高い順に並べたときの fault-prone モジュールの検出力についても評価した．

以下，本稿では，3.2節でレガシーソフトウェアの特徴とそれを踏まえた fault-prone モジュール予測手順について述べ，3.3節でその手順に従って実施した実験内容について述べる．3.4節で結果の評価手法について述べた後，3.5節で

実験結果について述べ，さらに3.6節でそれらの結果を考察し，最後にまとめと今後の課題について述べる．

3.2. Fault-prone モジュール予測方法

本研究で前提とする予測方法は，基本的には従来の研究と同様，ソフトウェアの各モジュールについてモジュールメトリクスと過去に検出された欠陥数を測定し，それらに基づいて予測モデルを構築し，そのモデルに予測対象モジュールのメトリクスデータを入力することによって,fault-proneness を予測するものである．

ただし，本研究ではレガシーソフトウェアの保守工程を想定している点が特徴である．レガシーソフトウェアは最初のバージョンが開発されてから非常に長い年月が経過しており，その後数多くの修正や機能追加が加えられる．一定の期間（数年おき）でシステムの大規模な更改が行われ，プラットフォームの移行や大規模な機能追加などが行われる．このとき，ソフトウェアの各モジュールは再利用されて使いつづけられるものもあれば，新たに追加されたり作り直されるものも存在する．

このことから，レガシーソフトウェアの各モジュールにおける開発・試験時のドキュメントや情報はすでに消失しているか,すでに陳腐化していることが多い．しかしその一方で，長期間の使用実績があるため，従来の研究で使用されていた製造・試験時のモジュールよりも，モジュールが使用されてきた期間や変更数といった履歴データが有効であり，またそれらがモジュールの複雑さにとって大きな意味を持つと考えられる．そこで，モジュールメトリクスとして，ソースコードメトリクスだけでなく，履歴データも用いる．

3.3. 実験概要

本章では,3.2節の手順に基づいて実際のレガシーソフトウェアを対象として実施した fault-prone モジュール予測実験の概要について述べる．

3.3.1. 実験の目的

本研究は，以下の各項目について実際のシステムのデータを用いて検証することを目的に実施した．

1. レガシーシステムの保守段階において有効なデータのみを用いて，fault-prone モジュールを予測する方法の有効性を検証する．

2．fault-prone モジュール予測に効果の高いメトリクスを分析する．

3．予測モデルによる予測精度の違いを評価する．

3.3.2. 対象ソフトウェア

本研究の対象ソフトウェアは，2.4.2節で述べたものと同じソフトウェアである．初期バージョンのリリース後 20 年以上にわたって新たな要求による機能追加や変更，及び欠陥修正のための修正が多く加えられてきた．さらにプラットフォームの変更に伴う大規模な更改が数回行われてきた．今回対象としたソフトウェアは，初期バージョンから何度かの更改を経たバージョンである．

このソフトウェアの各モジュールは，対象となったメインフレーム特有の二種類の言語（以降，言語 A,B と呼ぶ）で記述されている．言語 A はシステムの最初のバージョンがリリースされた当初に使われていた古い言語である．構造化プログラミングを意識した言語となっているが，複雑な構文も多い．言語 B は同じプラットフォームで動作するが，言語 A の後に使用されるようになった言語である．COBOL の構文を拡張した文法で，コーディングが容易になっている．近年の開発では言語 B が使われることが多く，中には，最初は言語 A で記述されていたが，更改のタイミングで言語 B に書き換えられたモジュールも存在する．つまり，言語 A で記述されたモジュールは古いが長期に渡って安定して稼動しているものが多く，一方，言語 B で記述されたモジュールは比較的新しいが変更量も多いモジュールが多いと考えられる．

本研究では，直近のデータしか利用できない状況を想定し，欠陥数を，対象としたソフトウェアのバージョンがリリースされた後に欠陥修正のためにモジュールに加えられた変更の数とした．この中には，リリース前にソフトウェアに残存していた欠陥の修正だけでなく，リリース後の機能追加や変更に伴って新たに混入した欠陥の修正も含まれる．リリース後のデータのみを対象としたために欠陥が検出されたモジュールの割合が小さく，また欠陥が検出されたモジュールにおいても欠陥数はほとんどが 1 件と非常に少なかったので，欠陥数そのものを予測するのではなく，各モジュールの欠陥の有無を予測することとした．

対象ソフトウェアのモジュール数は，言語 A の学習用と予測用それぞれで 1919 と 1933，言語 B で 1807 と 1870 であった．学習用と予測用のモジュール数に違いがあるのは，リリース後に新規に追加されたモジュールが存在するため

である．それらの中で欠陥が検出されたモジュールの数は，言語 A の学習用と予測用で 103 および 40，言語 B で 210 および 71 であった．また，モジュールの総行数は言語 A で記述されたものが約 700KSLOC，言語 B で記述されたものが約 1MSLOC であった．このように，記述された言語によってモジュールの特性が大きく異なることから，言語別にモデルを構築して評価した．

3.3.3. モジュールメトリクス

本研究を行うにあたり，ソースコードから各種モジュールメトリクスを計測するツールを自作し，言語に依存しない規模，機能複雑度，モジュール結合度などの複数の観点から，従来から提案されているメトリクス[22][27][46][48][62]を測定した．その一覧を表 3-1に示す．自作したツールではこれら以外にも多くのメトリクスを計測することができるが，メトリクスの多くは行数との相関が高かったため，そのようなメトリクスについては行数で割って正規化するとともに，相関が高いものを除くことによって分析に用いるメトリクスを選定した．

また，ソースコードメトリクスだけでなく，レガシーソフトウェアの特徴である長期間使用されてきたモジュールの特性を評価するために，履歴メトリクスも用いた．履歴メトリクス REV は，モジュールの改版数であり，モジュールが生成されて以降に加えられた変更数である．また，AGE はモジュールが生成されてからメトリクス計測日までの日数を意味し，この数値が大きいほど古いモジュールであることを意味する．モジュールの中には同じ変更量でもモジュール生成時に多く変更され，その後はほとんど変更されずに安定しているモジュールと，最近でも多くの変更が加えられているモジュールがあり，これらの複雑さ特性は異なると考えられる．MODDAYS はこれを反映するためのメトリクスで，この数値が大きいほど，モジュールが長年にわたって変更されつづけていて安定していないことを意味する．

表 3-1　メトリクス一覧

分類	名称	説明
サイズメトリクス	SLOC	モジュールの行数
	N_COMM	コメント行数/SLOC
	N_INPROC	モジュール内手続き数/SLOC
ソフトウェアサイエンスメトリクス	n1	オペレータの種類数
	n2	オペランドの種類数
	N_N1	オペレータ総出現数/SLOC
	N_N2	オペランド総出現数/SLOC
	VOLUME	Halstead ソフトウェアサイエンス尺度の Volume 以下の式で定義される V = (N1+N2)log2(n1+n2)
	DIFFICULTY	Halstead ソフトウェアサイエンス尺度の Difficulty 以下の式で定義される D = (n1/2)*(N2/n2)
フロー複雑度メトリクス	MAXNEST	最大ネストレベル
	N_CYCLOMATIC	平均 Cyclomatic 数
	N_NEST	平均ネストレベル
	N_JUMP	平均ジャンプ文数(GOTO 文など)
モジュール結合度メトリクス	N_VEXUSE	平均外部変数参照数
	N_EXCSUM	平均外部モジュール呼出数
	N_INCSUM	平均内部手続き呼出数
履歴メトリクス	REV	改訂数
	MODDAYS	モジュールが生成されてから最後に修正されるまでの日数
	AGE	モジュールが生成されてから現在までの日数

予測モデル構築用のモジュールから計測した各メトリクスの平均値と標準偏差を表 3-2に示す．この表から，メトリクス値は言語 B で記述されたモジュールの方が言語 A で記述されたモジュールよりも全体的に大きいことがわかる．特に AGE は言語 A の方が大きい(古い)にもかかわらず，言語 B の方が REV が多い．この結果からも，言語 A で記述されたモジュールは古くて安定しており，言語 B で記述されたモジュールは新しいが変更量も多い傾向を示すことがわかる．

表 3-2　各メトリクスの平均と標準偏差

名称	言語 A		言語 B	
	平均	標準偏差	平均	標準偏差
SLOC	372.0	12.27	631.8	13.39
N_COMM	0.3966	0.0045	0.5050	0.0037
N_INPROC	0.0168	0.0003	0.0180	0.0003
n1	18.52	0.1740	22.30	0.1523
n2	133.0	3.496	232.8	3.447
N_N1	1.588	0.0098	0.8489	0.0049
N_N2	1.107	0.0114	1.041	0.0054
VOLUME	8841.5	359.8	10528.0	268.7
DIFFICULTY	27.93	0.5069	30.94	0.5063
MAXNEST	6.562	0.0812	5.140	0.0445
N_CYCLOMATIC	0.1625	0.0024	0.1990	0.0018
N_NEST	5.209	0.0580	2.723	0.0245
N_JUMP	0.0808	0.0020	0.0544	0.0012
N_VEXUSE	0.1328	0.0034	0.4072	0.0064
N_EXCSUM	0.0110	0.0005	0.0184	0.0004
N_INCSUM	0.0164	0.0006	0.0141	0.0005
REV	34.02	1.017	50.14	1.031
MODDAYS	3420.0	50.47	2497.0	37.62
AGE	2624.7	52.02	2156.7	37.39

3.3.4.　モデルの構築

各モジュールのメトリクス値と検出された欠陥数に基づいて予測モデルを構築する手法として，線形判別分析，ニューラルネット，分類木の三種類を用いた．これらの手法を選択した理由としては，従来の研究でもよく用いられているということと，判別分析は線形に集合を分ける，ニューラルネットは非線形に集合を分ける，分類木は非連続に集合を分けるというように，各手法がそれぞれ異なる特性を持つためである．

以下，各手法の概要について述べる．

1．判別分析

本研究の判別分析モデルは，各メトリクス値を説明変数にとり，目的変数を fault-prone か fault-prone でないかに判別する．通常，モデルに使用される変数が，目的変数に本当に影響を与えているのか不明確な場合には，変数増加法，変数減少法，ステップワイズ法などの変数選択法がよく用いられる．これらの方法は一般的に変数選択／削除を F 統計量によって決定するが，この方法ではメトリ

クスが採用されるか否かが p 値に大きく依存する．本研究では，赤池の情報量基準 AIC[3]に基づいて変数選択を行うことにより，そのような曖昧さを除外して変数を選択することによりモデルを構築した．以下に，この変数選択手順を示す．

a. 説明変数となるメトリクスが 1 つだけのモデルを構築する．説明変数の候補数と同数のモデルが構築される．
b. 選択されている変数で構成されるモデルについて AIC を計算し，その値が最も低いモデルを選択する．判別分析モデル M の情報量基準 AIC は以下の式で表され，この値が小さいほど良いモデルである．

   AIC(M) = (サンプル数)×(残差平方和) + 2 ×（パラメータ数）

c. ステップ b で選択されたモデルに対して，残りの説明変数の候補に着目し，さらに 1 つの説明変数を追加したモデルを構築する．残っていた説明変数の候補と同数のモデルが構築される．
d. 追加すべき説明変数の候補がなくなるまで，あるいはすべての変数を選択しても AIC の値が小さくなくなるまで，ステップ b，c を繰り返す．

2. ニューラルネットワーク

ニューラルネットワークは，線形判別分析とは異なり，入力変数と出力変数間の非線形な関係を表現できるため，線形判別分析よりもより正確かつ予測精度の高いモデルが構築できることが期待される．

本研究では，図 3-1のような 3 層(入力層，中間層，出力層)からなる階層型ニューラルネットワークを用いた．図に示すように，中間層のユニットは 3 つに固定する．入力層から説明変数の値が入力されると，出力層から目的変数に対する判別値が出力される．

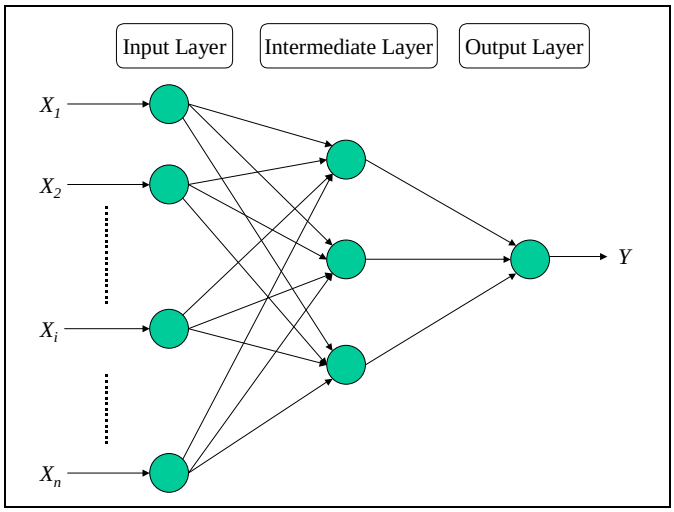


図 3-1　ニューラルネットワークモデル

パラメータ推定の方法としては，誤差逆伝搬アルゴリズム(Back Propagation)を用いた．その際，教師信号として以下を与えた．

- 学習用モジュールが故障の検出されたモジュールのとき

$$\frac{Faultを含むモジュール数}{全モジュール数}$$

- 学習用モジュールが故障の検出されていないモジュールのとき

$$-\frac{Faultを含まないモジュール数}{全モジュール数}$$

こうすることによって，出力結果の正負で fault-prone か fault-prone でないかを判別できるだけでなく，その数値の大小によってモデルによる fault-prone の優先順位をつけることができる．

また，線形判別分析の場合と同様，ニューラルネットワークにおいても AIC による変数選択を行い，有効な変数のみからなるモデル構築を行った．

なお，ニューラルネットワークでは学習回数によりモデル内のパラメータが変化する．その最適な学習回数の決定は難しいため，本研究では学習回数を 10000, 30000, 50000, 100000 としたときそれぞれについて予測モデルを構築し，学習回数の違いによる予測精度およびモデルの違いを比較した．

3. 分類木

分類木はこれまでに多くの研究で用いられている．分類木はその木構造において複数の分類パターンを持つため，判別分析やニューラルネットとは異なる非連続な関係を表現できる．そのため，分類木によっても非常に予測精度の高いモデルが構築できることが期待される．

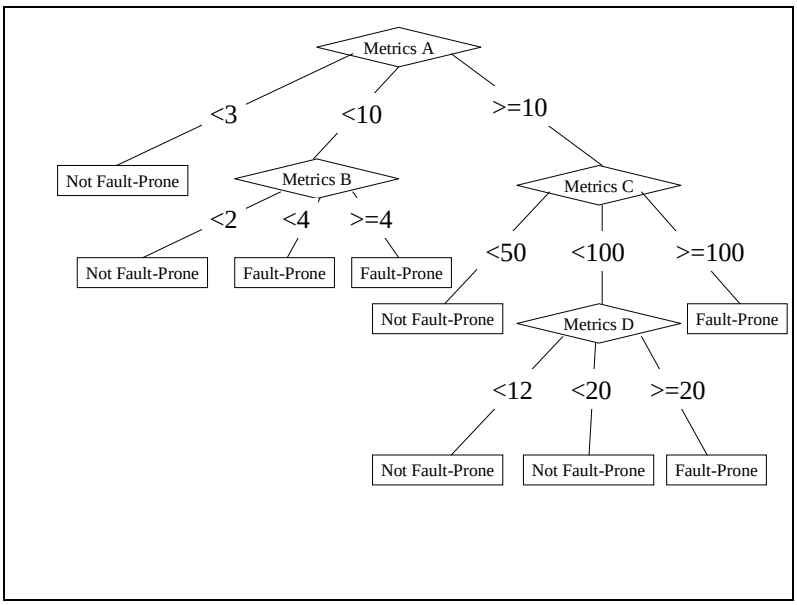


図 3-2　分類木の例

本研究では，分類木の構築方法として，Porter らが提案したエントロピーに基づいてノードのメトリクスを選択する方法[65]を元に，木の各ノードを AIC によって絞り込む方法[70]を用いた．

3.4. 予測結果の評価

3.3節で述べた手法に基づいて学習用データから予測モデルを構築した後，評価用データを適用してその中に含まれる fault-prone モジュールを予測するわけだが，この予測結果は，表 3-3に示すマトリクスで表現できる．ここで，表中の N_{ij} は，各セルに該当するモジュールの数を意味する．

表 3-3　分類結果マトリクス

		分類結果	
		Fault-free	Fault-prone
実際の結果	欠陥を含まないモジュール	N_{11}	N_{12}
	1 個以上の欠陥を含むモジュール	N_{21}	N_{22}

本研究では，表 3-3のマトリクスに基づいて定義される以下の尺度を評価尺度として用いた．これらは，過去の研究でも用いられた評価尺度である[46].

- Accuracy

  欠陥を含んでいたモジュールおよび含んでいなかったモジュールを正しく分類できた割合を示す．$\left(\frac{N_{11}+N_{22}}{N_{11}+N_{12}+N_{21}+N_{22}}\right)\times 100$で定義される．

- Correctness

  「欠陥を含んでいる」と分類されたモジュールのうち，実際に欠陥を含んでいたモジュールの割合を示す．$\left(\frac{N_{22}}{N_{12}+N_{22}}\right)\times 100$で定義される．

- Completeness

  欠陥を含んでいたモジュールのうち，「欠陥を含んでいる」と分類されたモジュールの割合を示す．これは，実際に欠陥を含んでいたモジュールをどれだけモデルが抽出できたかを意味する．この値が低いほど，モデルは欠陥を含んでいたモジュールを「欠陥を含んでいない」と分類していることになる．

  $\left(\frac{N_{22}}{N_{21}+N_{22}}\right)\times 100$で定義される．

- J coefficient

Accuracy，Correctness および Completeness は互いにトレードオフの関係にあるため，これらを総合的に評価するための尺度である．以下の式で定義

され，-1 から 1 の値をとるが，値が 0 より大きい場合には，その予測結果は偶然に得られる結果よりもよいと評価される．

$$J = \frac{n_{22}}{n_{21}+n_{22}} + \frac{n_{11}}{n_{11}+n_{12}} - 1$$

今回のような目的の場合には，欠陥を含んでいなかったモジュールと欠陥を含んでいたモジュール両方を正しく予測するよりも，実際に欠陥を含んでいたモジュールがどれだけ抽出されたかを予測する方が実用上は重要である．したがって，欠陥を含んでいないモジュールの予測精度まであわせて評価する Accuracy や J coefficient よりも Correctness および Completeness を重視したいが，それらにはトレードオフがあるため，Accuracy や J coefficient の値も評価時の参考とする．

実際の保守作業においてこのようなモデルを用いる際には，欠陥を含むモジュールを完全に予測できなくても，fault-prone である可能性が高いモジュールを順にリストアップし，その時点で使えるリソースに応じて重点的に調査すべきモジュールを絞り込めるだけでも非常に有用である．

そこで，さらに別の評価指標として，予測結果から fault-prone である可能性の高い順(rank-order)にモジュールを抽出した際に実際に欠陥を含んでいたモジュールがどれだけ早期に抽出されたかの割合も用いることとした．この結果は，Alberg Diagram[55]によって表現できる．Alberg Diagram の例を図 3-3に示す．Alberg Diagram は，モデルによって fault-proneness が高いと予測された順にモジュールを横軸に並べ，そのうち実際に欠陥を含んでいたモジュールの累積数をプロットしていくことによって描かれるグラフである．グラフの曲線が左上に凸であるほど予測力が高く，右下に凸であるほど予測力が低いといえる[3]．

[3]文献[55]では，回帰分析の結果を対象としているため，Alberg Diagram は横軸にモデルの予測結果（含まれる欠陥数そのもの）の降順にモジュールを並べ，縦軸には実際に検出された欠陥数の割合をプロットしたグラフとなっている．本研究では，モジュールに含まれる欠陥数そのものではなく欠陥の有無を予測しているため，若干定義が異なるが意味は同じである．

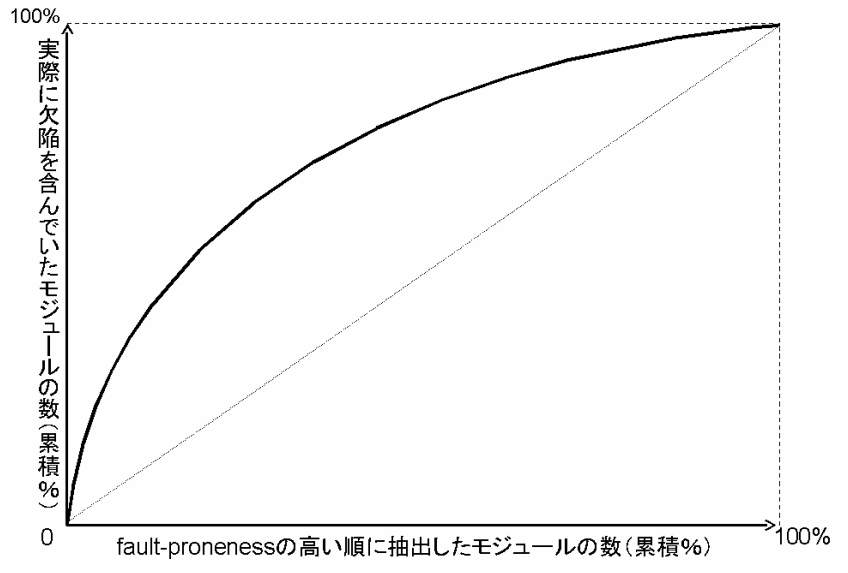


図 3-3　Alberg Diagram

各分析手法においてモジュールの fault-proneness を評価する方法は以下のとおりである.

- 判別分析

  判別分析の定義より，結果数値は判別境界からの距離を意味するので，その値が大きいほど判別境界からの距離が遠く，欠陥を含んでいる可能性が高いことになる．そこで，予測結果の判別値の大小で順位をつける.

- ニューラルネット

  本研究におけるモデルの定義より，判別分析と同様，予測結果の数値が大きいものほど境界からの距離が遠くなるため，欠陥を含む可能性が高いと判断する.

分類木については, 各リーフに分類されたモジュールに順位をつけることができないため，rank-order の評価対象からは除外する.

3.5. 実験結果

本章では3.3節で述べた手順に基づいて行った予測実験の結果を示す.

3.5.1. 誤り率による評価結果

各モデルを用いて予測した結果およびその評価結果を表 3-4および表 3-5に示す.ニューラルネットについてはモデル構築時の学習回数を変化させているので, 各学習回数における結果を示す.

各予測モデルの結果についてカイ 2 乗検定により有意差があるかどうかを検定したところ，言語 A,B いずれにおいても，判別分析，ニューラルネット，分類木のように手法を変更した場合には有意水準 0.05 で有意差が認められたが,ニューラルネットで学習回数を変更した場合の結果については，有意差が検出されないことがあった．つまり，ニューラルネットにおいては，学習回数を変更することによって若干精度が改善されることもあるが，その差は非常にわずかなものであったことになる．

表 3-4，表 3-5より，言語 A，言語 B のいずれにおいても，判別分析は Completeness が他の手法よりも高く，実際に欠陥を含むモジュールを高い割合で抽出できたことがわかる．しかし，その一方で Correctness が低く，分類結果に誤りが多いことがわかる．ニューラルネットは Completeness が判別分析より若干悪いものの，Correctness は改善されている．分類木は Accuracy が非常に良いが，Completeness の値が非常に悪く，実際に欠陥を含むモジュールをほとんど抽出できなかった．

J coefficient については，いずれのモデルについても正の値を示し，一応の効果があったことがわかる．言語 A についてはニューラルネット(30000 回)がもっとも高い数値を示し，言語 B については判別分析がもっとも高い数値を示した．一方，言語 B の分類木は 0.09 と非常に低い数値を示した．

表 3-4　評価結果(言語 A)

	判別分析	ニューラルネット				分類木
		10000	30000	50000	100000	
N_{11}	1229	1658	1671	1705	1774	1802
N_{12}	664	235	222	188	119	91
N_{21}	4	14	11	14	18	26
N_{22}	36	26	29	26	22	14
Accuracy	65.44%	87.12%	87.95%	89.55%	92.91%	93.95%
Correctness	5.14%	9.96%	11.55%	12.15%	15.60%	13.33%
Completeness	90.00%	65.00%	72.50%	65.00%	55.00%	35.00%
J coefficient	0.549	0.526	0.608	0.551	0.487	0.302

表 3-5　評価結果(言語 B)

	判別分析	ニューラルネット				分類木
		10000	30000	50000	100000	
N_{11}	1294	1452	1317	1552	1562	1613
N_{12}	504	346	481	246	236	185
N_{21}	30	40	37	41	41	57
N_{22}	41	31	34	30	30	14
Accuracy	71.43%	79.35%	72.28%	84.64%	85.18%	87.05%
Correctness	7.52%	8.22%	6.60%	10.87%	11.28%	7.04%
Completeness	57.75%	43.66%	47.89%	42.25%	42.25%	19.72%
J coefficient	0.297	0.244	0.211	0.286	0.291	0.094

3.5.2.　Rank-order による評価結果

次に,モデルに fault-prone の可能性が高い順にモジュールを抽出したときに,実際に検出された欠陥を含むモジュールの割合をプロットし，Alberg Diagram[55]を作成した. 言語 A での結果を 図 3-4に, 言語 B での結果を図 3-5 に示す.

これらの図を比較した結果，全般的に，言語 A で書かれたモジュールは言語 B で書かれたモジュールよりも高い精度を示した．また，ニューラルネットでは，いずれも学習回数が高くなるほど予測精度が高くなる傾向を示したが，言語 A では 30000 回で最良の精度を示し，ほぼ収束したように見えるのに対し，言語 B では 100000 回が最良の精度を示した.

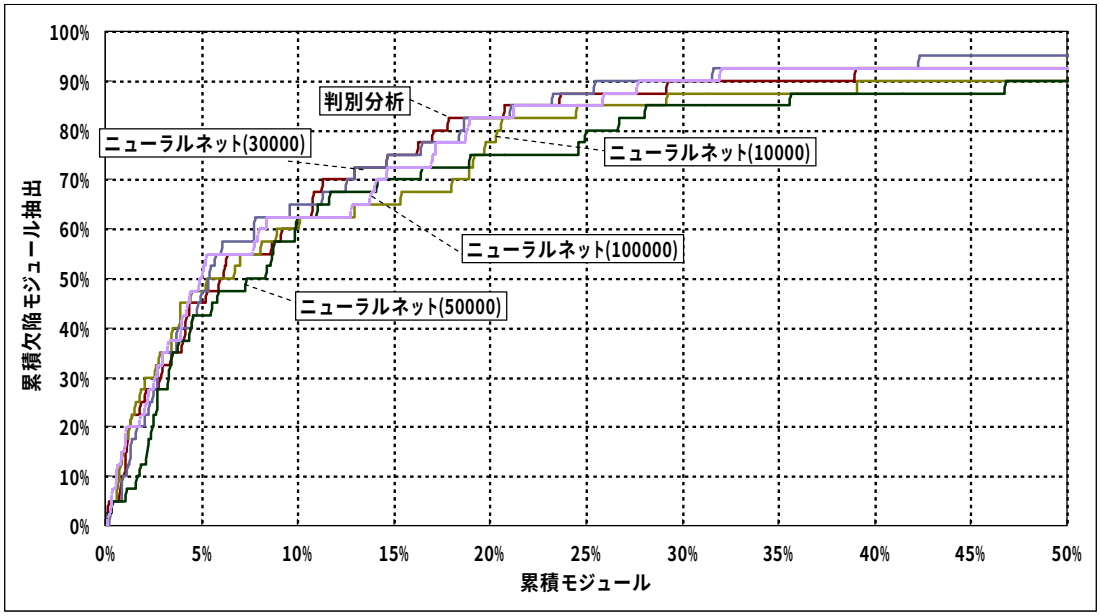


図 3-4　Alberg Diagram による評価結果（言語 A）

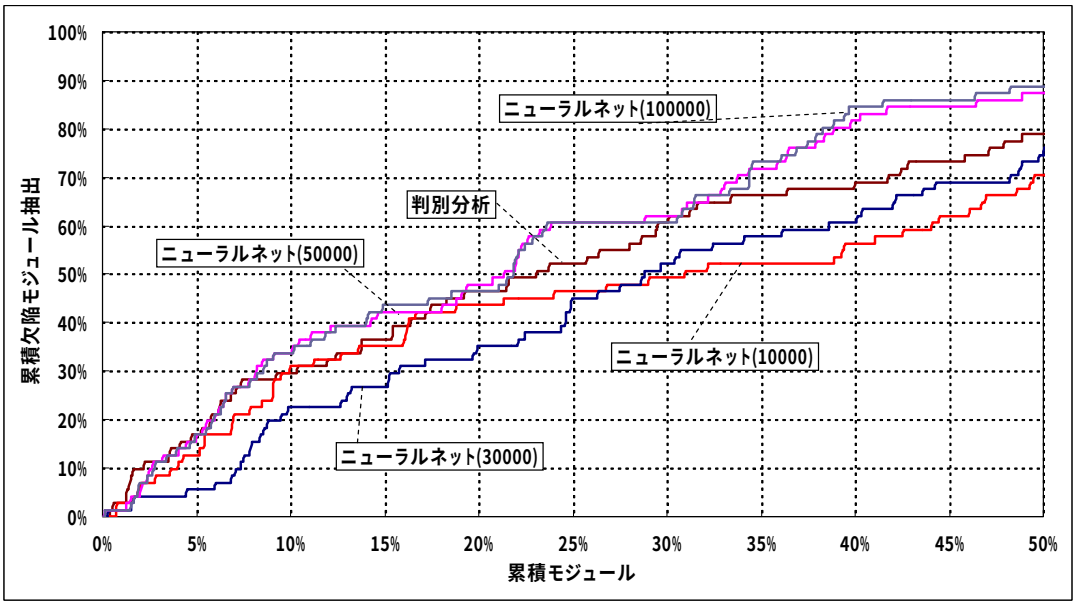


図 3-5　Alberg Diagram による評価結果（言語 B）

さらに，各モデルにおいて，全体の x%(x=80,90,95)の欠陥を含んだモジュールを抽出するために，何%のモジュールがリストアップされたかを表 3-6に示す．たとえば，言語 A の判別分析では，欠陥を含んだモジュールの 80%を抽出するために，全体の 17%のモジュールを fault-prone モジュールとして抽出すればよか

ったことがわかる．

表 3-6　x%の欠陥モジュールを抽出するためのモジュール数

(a)言語 A

欠陥モジュール%	判別分析	ニューラルネット							
		10000		30000		50000		100000	
95%	1042	(53.91%)	1651	(85.41%)	818	(42.32%)	1921	(99.38%)	1010	(52.25%)
90%	565	(29.23%)	756	(39.11%)	492	(25.45%)	905	(46.82%)	536	(27.73%)
80%	330	(17.07%)	394	(20.38%)	357	(18.47%)	483	(24.99%)	363	(18.78%)

(b)言語 B

欠陥モジュール%	判別分析	ニューラルネット							
		10000		30000		50000		100000	
95%	1520	(81.33%)	1466	(78.44%)	1669	(89.30%)	1603	(85.77%)	1584	(84.75%)
90%	1215	(65.01%)	1267	(67.79%)	1060	(56.71%)	1143	(61.16%)	1156	(61.85%)
80%	939	(50.24%)	973	(52.06%)	962	(51.47%)	726	(38.84%)	716	(38.31%)

3.6. 考察

以下，3.3.1節で示した目的と照らし合わせて，3.5節の結果を考察する．

3.6.1. レガシーソフトウェアにおける fault-prone モジュール予測の有効性

表 3-4より，言語 A の線形判別分析では Completeness が 90%という非常に高い数値を示した．つまり，実際に欠陥を含んでいたモジュールの 90%を予測できたことを意味する．ただし，Correctness が 5.14%と非常に低く，実際には欠陥を含んでいなかったモジュールの 30%が fault-prone であると予測された．言語 B については，全般的に Correctness, Completeness ともに言語 A の結果よりも悪い結果となった．実際にこの予測結果を開発や保守作業で活用する場合，fault-prone と予測されたモジュールを担当者に提示し,重点的に試験対象とすることになる．しかし，予測結果の中で実際に欠陥を含んでいるモジュールの割合が低かったり，また fault-prone ではないと予測されたモジュールにも（割合は少なくても）実際に欠陥を含むモジュールが存在することになると，有益な情報をもたらしたとはいえない．

一方，図 3-4および表 3-6の結果では，言語 A では予測結果において欠陥を含

む可能性が高い上位約 20%のモジュールの中で，実際に欠陥を含むモジュールの 80%を抽出することができた．言語 B においても，言語 A ほどではないが，上位 38～52%のモジュールに，実際に欠陥を含むモジュールの 80%が含まれていた．この結果は，リソースの限られている開発現場で，モデルによって出力されるモジュールの fault-proneness の高いものに注力して重点的に試験を行うことによって欠陥を含むモジュールを効率よく抽出できることを示している．つまり，直近のデータが乏しいレガシーソフトウェアにおいても，モジュールメトリクスおよび履歴メトリクスを用いて欠陥を含むモジュールを予測する手法は有効であるといえる．

3.6.2. 各モデルで効果の大きかったメトリクス

各モデルにおいて欠陥モジュール予測に影響の大きかったメトリクスを調べた．

本研究では，判別分析法，ニューラルネットでは変数選択に AIC を用いているため，早期に選択されたメトリクスほど fault-prone モジュールの予測に効果的である．これを利用して，各モデルにおいて影響の大きかったメトリクスを抽出した．判別分析およびニューラルネットにおいて選択されたメトリクスを表 3-7 に示す．表の上位にあるメトリクスほどモデル構築プロセスの早期に選択されており，効果が大きいことを意味する．

表 3-7　Fault-prone モジュールの抽出に影響が大きかったメトリクス

(a) 言語 A

判別分析	ニューラルネット			
	10000	30000	50000	100000
REV	REV	REV	REV	REV
SLOC	MAXNEST	MAXNEST	MAXNEST	MAXNEST
N_VEXUSE	N_VEXUSE	N_VEXUSE	N_VEXUSE	N_VEXUSE
N_INCSUM	SLOC	SLOC	SLOC	SLOC
N_N1	N_N1	N_N1	N_N1	MODDAYS
MAXNEST	N_CYCLOMATIC	N_CYCLOMATIC	N_INCSUM	N_N1
N_NEST	n1	n1	N_CYCLOMATIC	N_NEST
n1	N_JUMP	N_JUMP	n1	n1
N_JUMP	n2	N_INCSUM	N_JUMP	N_COMM
VOLUME	MODDAYS	VOLUME	n2	N_CYCLOMATIC
	DIFFICULTY	DIFFICULTY	DIFFICULTY	n2
	VOLUME	n2	N_N2	DIFFICULTY
	N_COMM	N_N2	VOLUME	AGE
	N_N2	N_COMM	MODDAYS	VOLUME
	N_NEST	N_NEST	N_NEST	N_JUMP
	N_INCSUM	N_EXCSUM	N_COMM	N_N2
	N_EXCSUM	MODDAYS	AGE	N_EXCSUM
	AGE	AGE	N_INPROC	N_INPROC
	N_INPROC	N_INPROC	N_EXCSUM	N_INCSUM

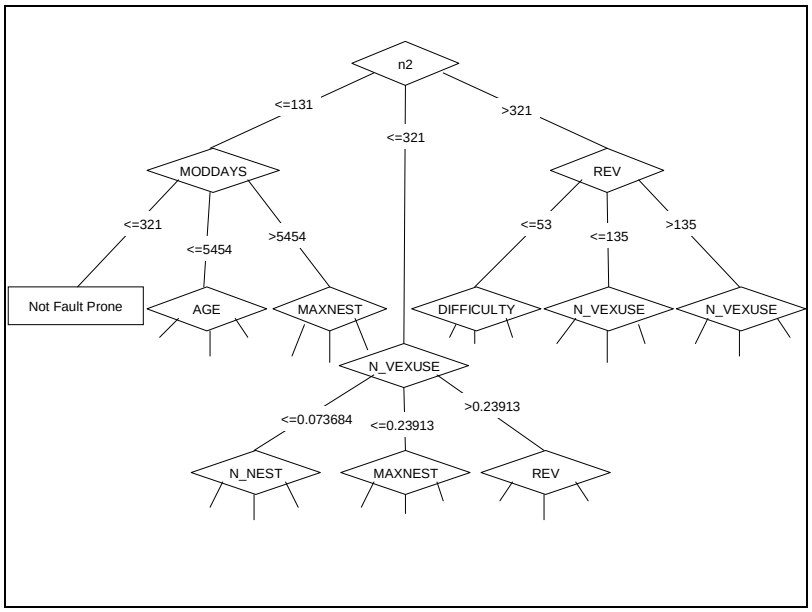
(b) 言語 B

判別分析	ニューラルネット			
	10000	30000	50000	100000
REV	MODDAYS	MODDAYS	MODDAYS	MODDAYS
AGE	REV	REV	AGE	AGE
N_NEST	N_NEST	N_NEST	REV	REV
N_VEXUSE	N_VEXUSE	AGE	n2	N_NEST
MODDAYS	AGE	n1	N_NEST	n2
N_CYCLOMATIC	DIFFICULTY	N_VEXUSE	N_VEXUSE	MAXNEST
N_INPROC	N_CYCLOMATIC	N_N1	MAXNEST	N_VEXUSE
MAXNEST	N_N1	MAXNEST	N_N2	N_N2
n2	MAXNEST	N_COMM	N_CYCLOMATIC	N_CYCLOMATIC
n1	SLOC	SLOC	N_N1	N_N1
N_INCSUM	n2	VOLUME	SLOC	SLOC
	N_COMM	n2	VOLUME	VOLUME
	N_EXCSUM	N_CYCLOMATIC	DIFFICULTY	DIFFICULTY
	N_N2	N_N2	N_COMM	N_COMM
	VOLUME	DIFFICULTY	N_JUMP	N_JUMP
	N_INCSUM	N_JUMP	N_EXCSUM	N_EXCSUM
		N_INPROC	N_INPROC	N_INCSUM
		N_EXCSUM	N_INCSUM	N_INPROC
		N_INCSUM	n1	n1

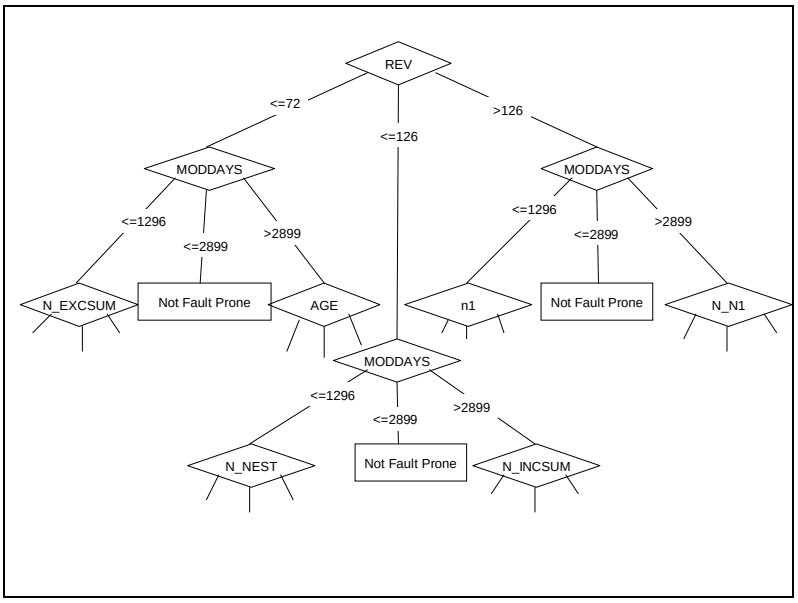
分類木についても，今回の生成方法では，メトリクス選定時にエントロピーが小さい（分類結果に曖昧さが少ない）メトリクスを選択しているため，木のルートに近いものほど効果が大きいことを意味する．そこで，各言語にて構築された分類木の一部（ルートノードから三段階下のレベルまで）を図 3-6に示す．言語 A ではルートノードには n2 が選択されているが，その下のノードでは REV, MODDAYS，N_VEXUSE が選択されている．言語 B については，ルートノードで REV が，その下のノードでは，MODDAYS が選択されている．これらの結果から，いずれの言語および分析手法においても REV が非常に効果的であったことがわかる．

REV 以外のメトリクスでは, 言語 A では MAXNEST, SLOC, N_VEXUSE といったソースコードメトリクスの影響が大きかったのに対し，言語 B では AGE や MODDAYS といった履歴メトリクスの影響が依然として大きかった. 言語 B では履歴メトリクス以外では，n2，N_NEST，N_VEXUSE などが共通して選択されていた．

以上の結果と，各言語で記述されたモジュールの特性（言語 A は古くて安定している，言語 B は比較的新しく，変更量が多い）から，言語 A で記述されたモジュールでは仕様や設計上の欠陥は長期にわたって保守された結果ほとんど残っておらず，特にネストの深さ，モジュールの大きさ，外部変数の使用量といったモジュールの特性が，欠陥の見落としの原因になったと推測される．一方，言語 B で記述されたモジュールは機能追加等が頻繁に加えられていることから，新たに作りこまれる欠陥も多く，モジュール構造の複雑さが欠陥に及ぼす影響は履歴メトリクスに比べても比較的少なかったと考えられる．



(a) 言語 A



(b) 言語 B

図 3-6　分類木（上位のみ）

3.6.3.　モデル構築手法による予測精度の違い

いずれの言語においても，各モデル構築手法によって評価結果には大きな差が出た．ニューラルネットは非線形，分類木は非連続という性質を持つため，線形である判別分析よりもより柔軟なモデル構築が可能なため予測精度が向上するこ

とを期待していたが，実際には Correctness は改善されたものの，Completeness は悪化し，必ずしも精度が向上したとはいえなかった．

また，図 3-4，図 3-5からは，言語 A においては各手法間で大きな差はなかったが，言語 B においてはニューラルネットが判別分析よりも若干良い結果を示し，分類木がもっとも悪い結果を示した．分類木の場合，各メトリクスの閾値の決め方や，分類の終了条件（どこまで木を生成するのか，リーフノードにおいて何%欠陥を含むモジュールが残っていたら Fault-prone と判断するのか）によって結果が大きく異なる．たとえば，木の各ノードを生成するときに，その時点で欠陥を含むモジュールの割合が 5%(lowerbound)以下もしくは 20%(upperbound)以上になったら木の成長をストップさせて，リーフノードとするようにして木を生成して同様の予測を行ったところ，表 3-8の結果が得られた．表 3-4の結果に比べると， Completeness が大幅に改善され，J coefficient は他のモデルに比べても最良の値を示した．一方，Correctness が悪化しており，やはりトレードオフの関係にあることがわかる．このように，分類木を生成できるところまで生成すると，木の構造が非常に細かくなってしまい，結果として学習データに特化しすぎたモデルが出来上がってしまう．一方，木の生成を途中で停止させて元データに特化しないモデルを構築すると，学習データに対しての分類精度が低下するが，予測時の精度は他のモデルと同等にまで向上すると考えられる．

表 3-8　成長を途中でストップさせた分類木の予測精度

	言語 A	言語 B
Accuracy	87.79%	71.11%
Correctness	11.72%	7.44%
Completeness	75.00%	57.75%
J coefficient	0.631	0.294

ニューラルネットは判別分析よりも若干良い精度を示したが，判別分析に比べてモデルの構築に非常に時間がかかる．特に今回我々が用いたモデル構築手法では AIC に基づいて変数選択を行っているため，最終的なモデルを得るために n(n+1)/2 個のモデル(ここで，n はメトリクス候補の数)を構築することになるため，判別分析より時間がかかる．参考までに，今回の実験で 100000 回学習させ

るモデルを構築したところ，SGI の Origin2000(MIPS R10000 プロセッサ搭載)上で実行させても 5 日もの時間を要した.判別分析モデルの構築に要する時間は，それに比べると無視できるくらい小さい.

以上をまとめると，今回用いた予測モデルの中では，非線形および非連続な予測モデルであるニューラルネットや分類木は，線形モデルである判別分析よりも分類精度が劇的には改善できず，モデル構築の手間を考慮すると，実用上は判別分析で十分であった.

3.7. まとめ

レガシーソフトウェアの保守工程を想定して，リリース後に有効な最近のデータのみを用いて，三種類の方法で fault-prone モジュールを予測する実験を行った．その結果をまとめると以下のようになる.

- 単に予測モデルによって fault-prone モジュールかどうかを予測するだけでは, fault-prone と予測されたモジュールの中で実際に欠陥を含んでいるモジュールの割合が低かったり,また fault-prone ではないと予測されたモジュールにも（割合は少なくても）実際に欠陥を含むモジュールが存在するため, 実際の開発や保守作業に有益な情報をもたらすとはいえなかった. しかし, fault-prone の可能性が高い順にモジュールを優先順位付けすることによって，その上位に選択された小数のモジュールの中で実際に欠陥を含むモジュールの大部分を抽出することができた．この結果は，モジュールの fault-proneness の高いものに注力して重点的に試験を行うことによって欠陥を含むモジュールを効率よく抽出できることを示しており，直近のデータが乏しいレガシーソフトウェアにおいても，モジュールメトリクスおよび履歴メトリクスを用いた fault-prone モジュール予測手法の有効性を示した.
- 同じレガシーソフトウェアでも，比較的古くて安定しているモジュールよりも最近も頻繁に変更が加えられているモジュールでは，モジュールメトリクスよりも履歴メトリクスの方が, fault-proneness に及ぼす影響は大きい.
- 分類木ではモデルが学習データに特化しすぎ逆に予測時の精度は悪化する. しかし，モデル構築時にある点で枝の成長を止めるようにすると，他のモ

デルと同等の予測精度が得られた．精度にそれほどの差がないことと，モデル構築に要する時間を考えると，実用上は線形判別分析で十分であった．

今後の課題としては，言語 B での結果を踏まえて欠陥の種類別にモデルを構築してその結果を明らかにすることと，他のソフトウェアにも同様の予測を行い，結果の妥当性を検証することが挙げられる．

4. プログラムスライシング・部分評価を用いたデバッグ支援ツール

4.1. はじめに

ソフトウェアの多機能化・大規模にともない，プログラムのデバッグも困難になっている．一般に，プログラムをデバッグする際には，対象となる範囲のソースコード全体を参照することになるが，規模が大きくなると全体の把握や欠陥に関する部分の特定に多くの時間がかかってしまう．実際には，欠陥に関係する部分は一部分であるので，その部分だけをソースコードから抽出するような機能があれば，開発者は欠陥に関係ない部分を参照する必要がなくなり欠陥位置の特定が容易になるためデバッグ効率の向上が期待される．ただし，その際に開発者が欠陥に関連する部分の抽出を手動で行うのは手間がかかるうえ，ときとして関連のある部分を見過ごす危険があるので，機械的に効率良く抽出することができれば非常に有効である．

そこで，本研究では，プログラムスライシング及び部分評価手法を用いて，プログラム全体から注目する文や変数に関係のある部分のみを抽出し，それに対して処理を行うようなデバッグ支援ツールを提案し,実際にツールの試作を行った．

プログラムスライシング[80]とは，プログラム中のある文 s での変数の値に影響を与える文の集合，もしくは s での変数の定義が影響を与える文の集合である（こうして抽出された文の集合をスライスと呼ぶ).プログラムスライシングにより，ある出力変数に関係のある文のみをプログラムから抽出することができる．スライスには，動的スライス（プログラムの実行系列を解析して得られるスライス）と静的スライス（プログラムの依存関係を解析し，それに基づいて求められるスライス）の二種類があるが，本研究では，抽出したスライスを１つのプログラムとして扱ってデバッグを行えるようにするため，静的スライスを対象としている．

また，部分評価とは，プログラムに対する入力変数の値をある定数に固定し，プログラム中の文や式の書換えを行い,目的とする部分の抽出を行うことをいう．

プログラムスライシングおよび部分評価の手法は，それぞれ独立して詳しく研究されている[23][29][57][74][76]．本研究では，このようなツールの有効性を評価することを目的としているため，試作ツールにおけるスライスの抽出アルゴリ

ズムは比較的実現が容易でかつ実行効率が現実的な，再帰を含む関数間の依存解析手法[76]に基づいている．他にもポインタ変数の解析[14][57]や，それに伴う変数のエイリアス解析[61]なども提案されているが，それらを効率よく扱えるアルゴリズムがまだ存在しなかったことから，本ツールでは実装しなかった．部分評価方法については，値の代入による式の評価と不要式の削除を行う方法に基づいている[36][74]．プログラムの記号実行やループの展開[15]，ブレスブルガ式の評価[53]などの技法も提案されていたが，実現の困難さや非効率さ，さらに，デバッガとして用いるには元のプログラムとの対応関係を保存する必要性があることなどから本ツールではより簡単な方法を用いた．

スライスを利用したデバッグ支援ツールとしては，Spider[1]，Chase[66]などが提案されているが，これらはいずれも動的スライスに基づくもので，実行させるたびにスライスを計算しなければならないうえ，実行系列を保存しておく必要があるために解析結果が膨大な量になる．本研究で試作したツールは静的スライスを対象としているため，一度依存関係を解析すれば，その結果からスライスを容易に計算できる．

静的スライスを用いたデバッグ手法としては，ダイシング[45]やアルゴリズミック・デバッギング[21]が提案されている．これらの手法では，スライスを使って直接欠陥の位置を特定しようとしているが，本研究では，スライスや部分評価を行うことにより,プログラムのうち,一部の機能のみを実行する部分を抽出し，その抽出部分をデバッグの対象とすることによりデバッグ時の参照範囲を縮小させることを目的としている点が異なる． また，スライスによって有効な抽出が行いにくい場合でも，本ツールでは，部分評価の機能を持つため，対象を小さくできる場合がある．たとえば入力のパラメータで１つの機能の実行が指定/除外されるようなプログラムに対しては，入力パラメータを特定の値として部分評価することによって,容易にそのサブシステムを抽出したり除いたりすることができる．このような静的スライスと部分評価を組み合わせたデバッグ支援ツールは今までに存在しない．さらに，本ツールは，抽出した部分を１つのプログラムとして扱うことができるとともに，必要ならば抽出部分にさらにスライシングや部分評価を適用することもできる．

本稿では，これ以降，4.2節でプログラムの依存関係解析の概要を，4.3節で作

成したツールの概要を，4.4節でツールの実行例およびその評価について述べる．

4.2. プログラム依存関係解析の概要

本稿で述べるデバッグ支援ツールは，プログラムの依存関係解析の結果得られるプログラム依存グラフ(Program Dependence Graph, 略して PDG)を利用し，スライシングや部分評価を行うことによってプログラムの参照範囲を小さくすることによりデバッグの支援を行う．

4.2.1. プログラム依存グラフ（PDG）

PDG はプログラム内の文の依存関係を表すグラフである[29]．PDG の節はプログラム中の各文および if 文や while 文の条件判定部分を表し，辺は変数の影響を伝えるデータ依存（Data Dependence，略して DD）関係および条件文や繰り返し文の制御の影響を伝える制御依存（Control Dependence，略して CD）関係を表す．

DD は，各頂点の到達定義集合（Reaching Definitions，略して RD）を求めることによって得られる．PDG 上でのある頂点 t の RD とは，変数 v と頂点 s との組(v,s)の集合である．これは，

- プログラム中の文 s で変数 v を定義している．
- プログラム中の 2 つの文 s から t へのすべての実行パスの中で, v を定義しないパスが少なくとも 1 つ存在する．

ことを示している．t の RD に(v,s)が含まれ，かつ t が v を参照するとき，s から t への DD 関係があるという．

また，ある条件判定部分 s の結果により文 t の実行の有無が決定されるとき，s と t との間に CD が存在するものとする．すなわち CD は if 文や while 文の条件判定部分からそれらの内部ブロックに属する文への影響である．これはプログラムを解析すれば容易に求められる．

一般にプログラムには複数の手続きや関数が定義されており，各手続き・関数間には引数や大域変数を通じて DD 関係が生じる．これらの DD 関係を表すために，PDG にプログラム中の文とは直接対応しない節点（中継節点と呼ぶ）を用意する．

PDG は, ソースコードを解析し, プログラムの各文を PDG の節点に切りわけ，プログラム中の各文における RD を求め，それをもとにして PDG の各依存関係

辺を生成することによって作成される．

例として図 4-1のプログラムに対応する PDG を図 4-2に示す．図の PDG の中で角の丸い四角がプログラム中の各文に対応する節で，楕円が中継節点である．

また，有向辺のうち，実線で名前がついているものが DD 関係の辺で，破線のものが CD 関係の辺である．実線についている名前は，その DD 関係の辺が影響を伝える変数の名前である．また，破線で囲まれている部分はプログラム上の 1 つの手続きまたは関数を表す．

```
1     program euclid(input,output);
2     var x,y,g,l:integer;
3     function gcd(m,n:integer):integer;forward;
4     procedure swap(var a,b:integer);
5       var       temp:integer;
6       begin
7         temp:=a;
8         a:=b;
9         b:=temp;
10      end;
11    function lcm(a,b:integer):integer;
12      var       c:integer;
13      begin
14        c:=gcd(a,b);
15        lcm:=(a div c)*(b div c)*c
16      end;
17    function gcd;
18      var       w:integer;
19      begin
20        if m＜n then begin
21          swap(m,n);
22        end;
23        while n＜＞0 do begin
24          w:=m mod n;
25          m:=n;
26          n:=w;
27        end;
28        gcd:=m;
29      end;
30    begin!
31      writeln('Input x and y');
32      readln(x,y);
33      writeln('x=',x,' y=',y);
34      g:=gcd(x,y);
35      l:=lcm(x,y);
36      writeln('gcd=',g);
37      writeln('lcm=',l);
38    end.
```

図 4-1　PDG の元のプログラム

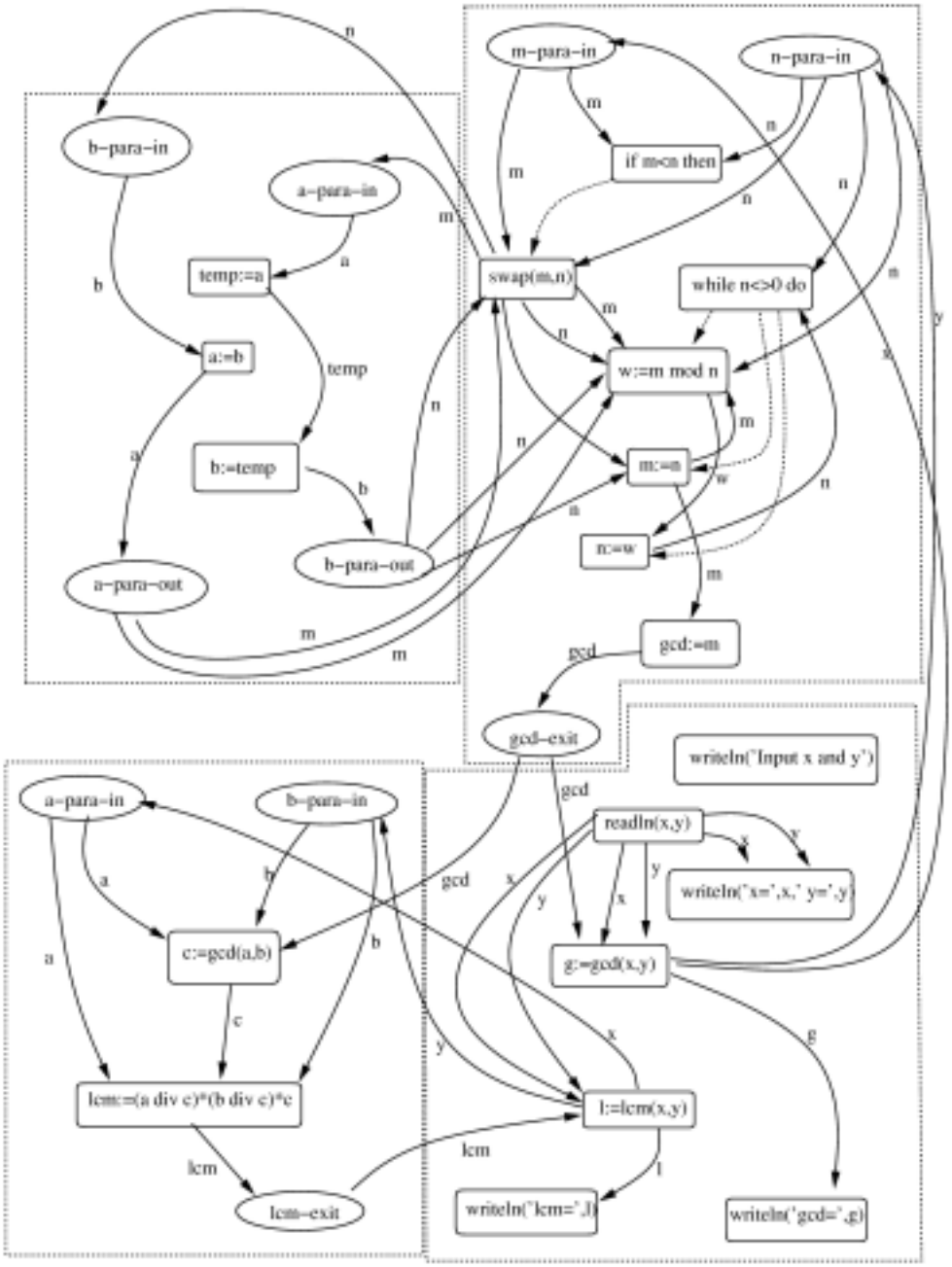


図 4-2　PDG の例

4.2.2.　スライス

4.1節で述べたように，本ツールで扱うスライスは，静的スライスである．文 s における変数 v に関するスライスとは，PDG 上において，CD 関係の辺または

DD 関係の辺を辿って文 s の変数 v に到達できる節点集合に対応する文の集合である．

PDG を与えられた頂点から辺の順方向に探索していくものを forward slice と呼び，逆方向に探索していくものを backward slice と呼ぶ．

スライスの例を図 4-1に示す．このプログラムの 36 行目で参照されている変数 g に関する backward slice が，図中の下線部分に相当する．この場合，変数 g に影響を及ぼさない部分，すなわち関数 lcm はスライスに含まれていない．

4.2.3. 部分評価

本研究での部分評価とは，プログラムに対する入力変数のとる値を固定し，それを参照する式や変数の値を順次書き換えたり，実行されない不要な文や手続きを削除することをいう．部分評価を適用すれば，入力変数もしくはそれに相当する大域変数の値によってどの機能を使うか，もしくは使わないかを決定しているようなプログラムに対しては，使用しない機能に関する部分をプログラムから削除することができる．

本研究の部分評価手法は，高谷が提案した手法[36][74]に基づく．この手法におけるプログラム *P* の部分評価手法の概略を以下に示す．

1. *P* の大域変数の中から固定の対象とする入力変数 x(複数個でもよい)を選ぶ．
2. 目的とする機能を選び出すのに応じて x の値（特定の値 a）を定め，変数とその値の組 $\langle x,a \rangle$ を作る．その他の大域変数と局所変数 y については不定値(⊥)との組 $\langle y,\perp \rangle$ を作る（これらの組の集合 $\{\langle x_1,a_1 \rangle \cdots \langle x_n,a_n \rangle\}$ を「状態」と呼ぶ）．
3. 主プログラムの本体 *M* および初期状態 *State* を引数として関数 *simple*（*M*, *State*）を実行し，その返り値を主プログラムの部分評価結果として出力する．
4. 部分評価の結果，呼び出しが削除されなかった各手続きおよび関数についてそれらが呼び出された時のすべてのプログラム状態を演算○を使って併合し，その手続きの初期プログラム状態 F_{iState} を作成する（演算○の意味は本稿末の付録に示す）．
5. 呼び出しが削除されなかった各手続きおよび関数について，手続きの本体

F_i およびその手続きの初期プログラム状態 F_{iState} を引数として関数 *simple*(F_i,F_{iState})を実行し，その返り値を各手続きおよび関数の部分評価結果として出力する．

関数 *simple* (*S*,*State*)は与えられた状態 *State* の下で，文 *S* を部分評価した結果の文を返す．*simple* は文の種類ごとに定義されており，（右辺の値が決まる場合の）代入文の実行，（条件式の値が決まる場合の）条件分岐文の書き換え，（同じく条件式の値が決まる場合の）繰り返し文の書き換えを行う．関数 *simple* による詳細な各文の部分評価手順は本稿末の付録に示す．

この手順でプログラムが部分評価されると，元のプログラムにあった手続きおよび関数の呼び出しや代入文および入力文で定義された変数の参照がなくなることが起こりうる．それらについては上記の手順で部分評価された結果のプログラムを再度解析することによって検出できる．一度も呼び出されない手続きおよび関数は，依存関係解析において関数の呼び出し関係を解析する際に生成される CFG(Call Flow Graph)から，どこからも呼ばれない関数を検出することによって得られる．また，代入文もしくは入力文が不要となるのは，その定義を参照する文が存在しない，もしくは自分自身の参照にしか用いられない場合である．このような代入文もしくは入力文は，PDG を解析することによって検出可能である．こうして検出された不要な手続きおよび関数ならびに代入文，入力文を削除することによってさらにプログラムを簡素化できる．

部分評価の例として，図 4-3に示すプログラムの入力変数のうち，変数 b の値を'n'に，変数 c の値を'y'に固定して部分評価を行った結果のプログラムを図 4-4 に示す．入力変数 b の値を'n'に固定した結果，入力文が削除され，また，その値が'y'のときに実行されるはずであった 27 行目から 37 行目の部分が削除されている．また，入力変数 c の値を'y'に固定した結果，24 行目の if 節が削除されている．

```
1    program wordcount(in,out);
2    var a,b,c:char;
3        in:array[0..1000] of char;
4        i:integer;
5        letter,word,line:integer;
6        isinword:boolean;
7
8
9    begin
10       writeln('Count the letter?(y/n)');
11       readln(a);
12       writeln('Count the word?(y/n)');
13       readln(b);
14       writeln('Count the line?(y/n)');
15       readln(c);
16       i := 0;
17       letter := 0;
18       word := 0;
19       line := 0;
20       isinword := false;
21       while in[i] <> EOF do begin
22         if a = 'y' then
23           letter := letter + 1;
24         if c = 'y' then
25          if in[i] = EOL then
26            line := line + 1;
27       if b = 'y' then
28         if (in[i]<>' ')and(in[i]<>EOL) then
29           begin
30             if isinword = false then
31              begin
32                isinword := true;
33                word := word +1
34              end
35           end
36          else
37           isinword := false;
38       i := i +1
39     end;
40     if a = 'y' then
41       writeln('letter =',letter);
42     if b = 'y' then
43       writeln('word   =',word);
44     if c = 'y' then
45       writeln('line   =',line)
46   end.
```

図 4-3　部分評価前のプログラム

```
program wordcount(in,out);
  var a,b,c:char;
      in:array[0..1000] of char;
      i:integer;
      letter,word,line:integer;
      isinword:boolean;

  begin
     writeln('Count the letter?(y/n)');
     readln(a);
     writeln('Count the word?(y/n)');
     writeln('Count the line?(y/n)');
     i:=0;
     letter:=0;
     line:=0;
     while in[i] <> EOF do begin
        if a='y' then

           letter:=letter+1;
        if in[i]=EOLN then

           line:=line+1;
        i:=i+1
     end;
     if a='y' then

        writeln('letter =',letter);
     writeln('line =',line)
  end.
```

図 4-4　部分評価後のプログラム

4.3. 試作ツールの概要

本節では，本研究で試作したデバッグ支援ツールの概要について述べる．本ツールの実装には，解析や実行部には C 言語を，インタフェースには Tcl/Tk を用いた．

4.3.1.　言語仕様

本ツールが対象とする言語は,以下のような仕様を持つ Pascal のサブセットである．

- 文として，代入文，条件文，繰り返し文，手続き呼出文，begin-end で囲まれる複合文を扱う．
- 手続きは再帰呼び出しも扱う．ただし，部分評価ではこれは扱えない．手続きの引数の渡し方については，値渡しと変数渡しの 2 種類がある．

- 変数のデータ型はスカラー型のみでポインタ型は扱わない．具体的には整数型，文字型，論理型およびそれらを要素に持つ配列型とした．

このように，言語仕様は単純化されてはいるものの，プログラミング言語の基本的な文の構造はすべて含んでおり，拡張は容易に行える．

4.3.2. 機能

本ツールは以下のような特長を持つ．

- 通常のデバッガと同様にプログラムのステップ実行，変数の参照，ブレークポイントの設定などが行える．
- プログラムスライスとして backward slice，forward slice のどちらでも抽出でき，また，そのときに，PDG のうち制御依存とデータ依存の片方だけを探索したり，希望の段階までを探索することもできる．
- 入力変数もしくは入力に相当する変数の値を定数に固定することにより，部分評価を行って，プログラムの一部の機能のみを実現する部分のみを抽出することができる．
- 抽出したプログラム部分に対してもデバッグ作業を行える．
- プログラムの抽出部分を一つのプログラムとして構文的に正しくなるように修正し，保存できる．

4.3.3. ツール構成およびその動作

本ツールの構成図を図 4-5に示す．本ツールは，ソースコードの構文解析を行いつつプログラム中の各文を PDG の各節点と対応付けるパーザ部，その結果に基づいて PDG を作成するアナライザ部，PDG を探索してスライスを得るスライサ部，PDG を利用してプログラムの部分評価を行う部分評価部，プログラムの実行を行うインタープリタ部に分けられる．

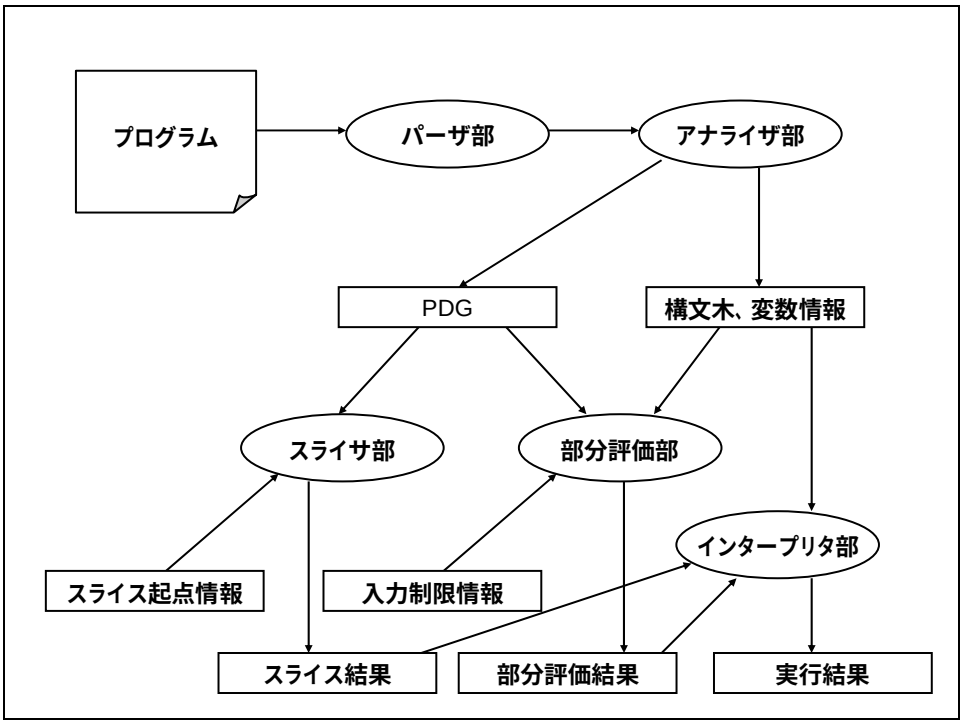


図 4-5　ツールの構成図

ユーザがデバッグの対象となるプログラムを入力すると，ツールは，パーザ部で構文解析を行ってプログラムの各文を切り分けて PDG の各節点に対応付けるとともに各文で定義および参照される変数や構文木の情報を抽出した後，アナライザ部で各節点間の依存関係の解析を行う．それらの解析が終了すると，PDG およびプログラムの実行や部分評価に必要な構文木や変数の情報などが生成される．

ユーザはこの段階でツールにどの動作をさせるかを指定する．ユーザがプログラムを実行させた場合には，ツール内では，インタープリタ部にアナライザ部が生成した構文木や変数の情報などが渡され，プログラムが実行される．その際，ユーザは，通常の実行だけでなく，ステップ実行やブレークポイントの設定，変数値の参照などのデバッグに必要な作業も行うことができる．

部分評価を行なう場合，ユーザは，まず，どの入力変数をどの値に固定するかの情報（入力制限情報）を入力する．その後，ツール内では，その入力制限情報と PDG および構文木などの情報が部分評価部に渡される．また，それらをもとにして部分評価部で部分評価が行われ，結果として，プログラムの一部の機能のみを実現するプログラムが生成される．

スライスを抽出する場合には，ユーザは，まず，スライサ部にどの変数に関するスライスを抽出するのかといった情報を入力する．その後，ツール内では，そのスライス起点情報と PDG がスライサ部に渡され，そこでスライスが計算される．そのスライス結果は，元のプログラムの部分プログラムとして出力される．

また，部分評価部やスライサ部から出力された部分プログラムもインタープリタ部で実行するとともに，デバッグ作業も行うことができる．このようにして，デバッグの対象を限定することにより，効率良くデバッグ作業を進めることができる．

4.4. 実行例

本ツールの実行例として，図 4-6のプログラムをデバッグする例を考える．このプログラムは，入力された 5 つの数の合計，最大，最小，平均を計算し，出力するプログラムであるが，一箇所に欠陥が含まれている．

```
1   program coverage(input,output);
2     var     Sum, Max, Min, Mean: integer;
3             A: array[1..5] of integer;
4             n: integer;

5   procedure calc(var sum, max, min, mean: integer);
6     var     i: integer;
7     begin
8       i := 1 + 1;
9       while i <= n do
10          begin
11            sum := sum + A[i];
12            if A[i] > max then
13              max := A[1]
14            else if A[i] < min then
15              min := A[i];
16            i := i + 1
17          end;
18        mean := sum div n;
19      end;

20  begin
21    n := 5;
22    writeln('Please Input ',n,' values');
23    readln(A[1],A[2],A[3],A[4],A[5]);
24    Sum := A[1];
25    Max := A[1];
26    Min := A[1];
27    Mean := A[1];
28    calc(Sum, Max, Min, Mean);
29    writeln('SUM         : ', Sum);
30    writeln('MAX         : ', Max);
31    writeln('MIN         : ', Min);
32    writeln('MEAN        : ', Mean)
33  end.
```

図 4-6　欠陥を含んだプログラム

1. プログラムを解析し，インタープリタ部で実行させる

この実行結果が図 4-7である．入力した 5 つの数字に対して，合計，最大値，最小値，平均値を出力しているが，このうち最大値を示す MAX の値が誤っている．

図 4-7　プログラムの実行結果

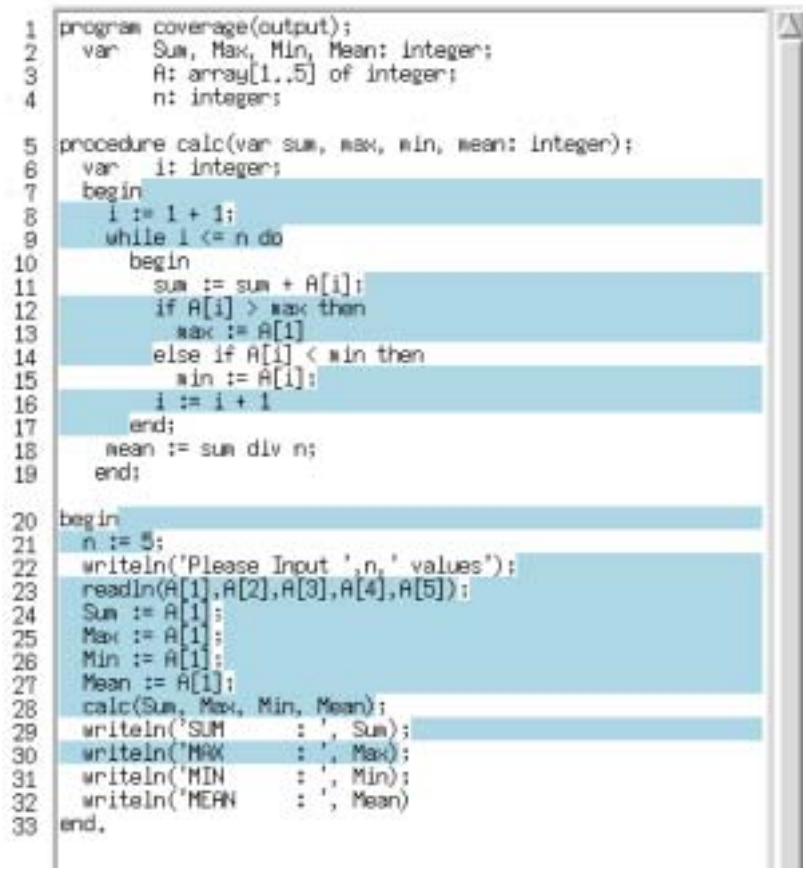
2. この値を出力した変数（図 4-6における 30 行目の変数 Max）に関するスライス（backward slice）を抽出する.

すると，図 4-8のようなウィンドウが現れる．この画面で，スライスを抽出したい変数名や, CD および DD のどちらの依存関係について PDG を探索するか，PDG を探索するパスの長さなどの情報を入力する．ここでは，変数 Max に関して，CD，DD の両方の依存関係について PDG をできる限り探索して backward slice を求めることにする. こうして得られるスライス結果は図 4-9中の網掛け部分として参照される.

図 4-8　スライス選択画面

ここで，スライスの起点となった変数 Max には直接関係のない文（24,26,27 行目の文）がスライスに含まれているが，これらは，28 行目の手続き呼出し文で参照している変数の値に影響を与える文であり，この手続き呼出文がスライスに含まれるため，これらの文もスライスに含まれる.

30 行目の Max に直接代入を行っている文は，13 行目の文である．これはスライスを計算する際に，レベルの設定を 1 などの適当な値に設定することにより得られる.



```
1  program coverage(output);
2    var   Sum, Max, Min, Mean: integer;
3          A: array[1..5] of integer;
4          n: integer;

5  procedure calc(var sum, max, min, mean: integer);
6    var   i: integer;
7    begin
8      i := 1 + 1;
9      while i <= n do
10       begin
11         sum := sum + A[i];
12         if A[i] > max then
13           max := A[1]
14         else if A[i] < min then
15           min := A[i];
16         i := i + 1
17       end;
18     mean := sum div n;
19   end;

20 begin
21   n := 5;
22   writeln('Please Input ',n,' values');
23   readln(A[1],A[2],A[3],A[4],A[5]);
24   Sum := A[1];
25   Max := A[1];
26   Min := A[1];
27   Mean := A[1];
28   calc(Sum, Max, Min, Mean);
29   writeln('SUM      : ', Sum);
30   writeln('MAX      : ', Max);
31   writeln('MIN      : ', Min);
32   writeln('MEAN     : ', Mean)
33 end.
```

図 4-9　スライス結果（網掛けの部分がスライス）

3. この文にブレークポイントを設定し，スライス部分のみを実行させる．

　すると，この文自体は何度も実行されているにもかかわらず，Max の値が変化していないことがわかる．これは，この文で参照される変数が A[1]となっているためで，それを A[i]に修正することにより正しい結果が得られる．

なお，スライスの抽出は，4.2.2節で述べたように，依存関係解析の結果生成された PDG を探索することによって行われるが，一旦 PDG が構築されれば，それを辿るだけでスライスは抽出されるので，スライシングに要する時間は非常に短い．

また，スライシングおよび部分評価を行なって，プログラムの参照部分を小さくできる割合は，入力プログラムおよび変数の指定方法などに依存する．この実行例の場合には，スライシングによって 33 行のプログラムが 25 行になった（プログラム，手続き，変数宣言文含む）．また，図 4-1のプログラムは，スライシン

グの結果，38 行のプログラムが 28 行に，図 4-3のプログラムは，部分評価の結果，46 行のプログラムが 26 行になった．

4.5. 考察

デバッグ時に，ソースコード中から欠陥に関係のある部分のみを抽出してその部分のみをデバッグの対象とすることにより，開発者の参照範囲を小さくするという当初の目的は，本ツールを用いることにより実現できる．

さらに，その依存関係情報は，１度解析すれば内部に保存できるため，同じプログラムに対しては，解析しなおすことなく別のスライス計算などの作業を行うことができる．

ただし，一般に元のプログラムが大きいときには，スライスも大きくなる[1]．本ツールでは，スライス全体を参照するだけでなく，PDG を探索するレベルを任意に設定することができるので，注目している文に直接影響する文のみを抽出したり，データ依存もしくは制御依存の一方だけを用いた PDG の探索もできるため，参照部分をさらに小さくすることができる．今後，プログラム全体ではなく，手続き単位でスライスを計算したり，実行したりできる機構があればさらに便利になると考えられる．

また，部分評価を行うと，式中の変数が定数に置き換えられる場合が起こるが，その場合でも式の意味は変わらない．ただし，部分評価により削除される部分に欠陥が含まれていた場合，当然その欠陥は部分評価結果には含まれない．しかし，本来，部分評価は入力変数もしくはそれに相当する変数の値を固定することにより，プログラムの機能を限定することなので，部分評価によって削除される部分に欠陥が存在していたとしても，それはその時点で注目している部分ではない．したがって，その時点での欠陥の原因究明には影響を与えない．

4.6. まとめ

プログラムから依存関係に基づいてスライシングや部分評価を行い，欠陥に関係のある部分のみを抽出するデバッグ支援ツールを作成した．本ツールを用いることにより，プログラム全体を対象としていた従来のデバッガを用いるよりも効率の良いデバッグを行うことができると期待される．実際に，学生のプログラミング演習において本ツールを用いてデバッグを行った結果，通常のデバッグよりも効率よく欠陥を検出できた研究事例も報告されている[54]．

また，本ツールでは，プログラムの依存関係解析だけでなく，部分評価に必要なその他の情報を得るための解析も行い，保存している．それらは必要に応じて自由に利用できるので，デバッグ以外にも，それらの情報を利用するような作業（たとえば，保守[24]など）にも役立つと考えられる．

今後の課題としては，実際の開発において用いることが出来るように言語仕様を拡大すべく，ポインタ変数や構造体も扱えるような依存解析アルゴリズム[14][44][63]を適用し，対象言語を拡張することなどがあげられる．

5. おわりに

本研究では，ソースコード解析の結果に基づきソフトウェア品質確保に関わる試験，デバッグ作業を効率よく行う手法の提案およびケーススタディの結果に基づく指針の提供を行った．

まず，ソースコード解析手法としてコードクローン抽出に着目し，モジュールに含まれるコードクローンとレガシーソフトウェアの信頼性・保守性との関係を定量的に分析した．分析はモジュールを単位とし，信頼性尺度として各モジュールでリリース後に検出された欠陥密度を，保守性尺度として各モジュールの改版数を用い，モジュールが持つコードクローンの含有率および最大長との関係を評価した．その結果，コードクローンの生成は，保守コスト増大の原因となり得るが，信頼性の低下を抑える可能性があり，信頼性と保守性はトレードオフの関係にある場合があることを定量的に示した．このように，一般的にはコードクローンの存在は，ソフトウェアの信頼性・保守性を低下させると考えられてきたが，本研究の結果，一概にそうとはいえず，コードクローンの活用によって信頼性の低下を抑えられる可能性があるという指針を示した．

次に，モジュールメトリクスの値に基づいて，開発時のデータが乏しいレガシーソフトウェアにおける fault-prone モジュール予測の有効性を検証した．実際に企業で運用されたレガシーソフトウェアを対象に，このソフトウェアの対象バージョンがリリースされた後３年間に検出された欠陥数から，判別分析，ニューラルネットワーク，分類木の三種類の予測モデルを構築し，その後に故障を発生する可能性のあるモジュールを予測した．その結果，単に fault-prone かそうでないかを予測するだけではトレードオフの関係があり，実際の開発や保守作業における有効性があるとはいえなかったが，欠陥が含まれる可能性が高い順にモジュールを並べることにより，少ないモジュールで欠陥を含むモジュールの大部分を予測できることがわかった．この結果から，モジュールメトリクスに基づいて事前に品質の悪いモジュールを洗い出すことによって重点的に試験すべきモジュール候補を絞り込むことができるため，試験効率の向上および欠陥修正対応工数の削減が期待される．

最後に，プログラムスライシングおよび部分評価手法を取り入れたデバッグ支援ツールを提案するとともに，実際にツールを試作した．このツールは，誤った

結果を出力した変数を起点にしたプログラムスライシングもしくはある特定の入力によって欠陥に関係のある機能だけを絞り込める場合に部分評価を行って欠陥に関係のある部分のみを抽出し，その部分のみをデバッグ対象範囲に絞り込むことができる．この機能により，プログラム全体を対象としていた従来のデバッガを用いるよりも効率の良いデバッグを行うことが期待される．

本論文で述べた 3 つの研究は，いずれもソースコードを解析した結果をソフトウェア品質確保活動の効率化に役立たせることを目的としている点で共通している．ここで，ソースコード解析に着目した理由は，ソースコードが仕様書や設計書などのドキュメントに比べて，ソフトウェアにおいてもっとも信頼でき，確実に存在する資産であるためである．特に2章と3章で対象としたレガシーソフトウェアにおいては，開発時の成果物やデータが消失していたり，残っていても信頼性に問題がある場合が多いため，唯一の信頼できる情報源となる場合もある．そのため，ソースコードを解析して得られる情報をソフトウェア開発作業の効率化に活用していくことは非常に実用性が高い．

本論文では 3 つのソースコード解析技術を個別に適用した結果に基づく研究成果について述べたが，今後の課題としては，これらの手法を組み合わせて分析を行うことにより，ソフトウェア品質確保の効率化に関してさらなる研究を進めることがあげられる．たとえば，コードクローンを特徴づけて分析するためのメトリクス[28]も提案されているが，これらのメトリクスを3章で述べた fault-prone モジュール予測に用いることによって予測精度が向上する可能性もある．

また，ソースコード解析結果から得られる成果物データだけでなくソフトウェア開発作業時に得られるプロセスデータと組み合わせることによって，さらに詳細な分析を行うことも今後の課題としてあげられる．たとえば，EPM(Empirical Project Monitor)[56]ではソフトウェア開発時のプロセスデータを自動的に収集する仕組みを構築し，その機能のひとつとして構成管理ツール CVS で管理されたソースコードの規模推移やチェックイン・チェックアウト履歴をリアルタイムで収集することができるが，ここで蓄積されているソースコードのモジュールメトリクスを測定し，その推移を把握するとともに他の収集データとの関係を分析することによって新たな知見を得られる可能性がある．

謝辞

本研究を進めるにあたっては，多くの方々にご助力をいただきました．そのすべての方々に深く感謝いたします．

本論文の審査を快くお引き受けいただくとともに様々な観点から貴重なご助言を賜りました奈良先端科学技術大学院大学情報科学研究科　松本　健一教授，関浩之教授，飯田　元教授，門田　暁人助教授に深く感謝申し上げます．門田助教授には,本研究を進める上で数多くの有意義な議論をさせていただいたとともに,論文の作成にあたっては詳細な部分にまでコメントいただき未熟であった私に対して熱心かつ懇切丁寧にご指導いただきました．心から感謝いたします．奈良先端科学技術大学院大学　鳥居　宏次特認教授，大阪大学大学院情報科学研究科井上　克郎教授，楠本　真二教授には，私が学部生として研究室に配属になったころから現在に至るまで多大なるご指導をいただくとともに本論文を作成するにあたって激励のお言葉を賜り後押しをしていただきました．心より感謝いたします．大阪大学大学院情報科学研究科　片桐　良実氏には，私が学部生の頃から現在に至るまで,研究を進める上で必要な様々な事務的なご支援をいただきました．心より感謝いたします．大阪大学大学院情報科学研究科　松下　誠助教授をはじめとする井上研究室の方々には，私が研究室に所属している間，公私ともに数多くのご指導をいただきました．心より感謝いたします．卒業生を含む奈良先端科学技術大学院大学情報科学研究科ソフトウェア工学講座の皆様には，あまり研究室には顔を出すことのできなかった私に対して，数多くのご助力をいただきました．心から感謝いたします．

会社業務の中で本研究を進める機会と援助を賜りました株式会社エヌ・ティ・ティ・データ　青木　利春相談役, 中村　直司顧問, 浜口　友一代表取締役社長，現東京工科大学コンピュータサイエンス学部　中村　太一教授に深く感謝いたします．また博士課程進学を勧めていただくとともに研究を進める上で数多くのご助言をいただきました同社技術開発本部企画部　箱守　聰部長，同社ビジネスソリューション事業本部情報セキュリティ推進室　山岡　正輝室長，現新日本監査法人　松田　栄之氏に深く感謝いたします．同社 SI コンピテンシー本部設計積算推進部　端山　毅部長には，入社時に直属の上司としてソフトウェア工学全般についてご指導いただくとともに現在に至っても数多くの有益なご助言をいただ

きました.心から感謝いたします.同社公共ビジネス推進部技術戦略部　赤坂　幸彦部長，渡辺　清孝課長，同社第四公共システム事業本部　友貞　雅裕シニアスペシャリストには，私が事業部に異動となった後も本研究を続けることをご了承いただくとともに数多くのご助言，ご支援をいただきました．心より感謝いたします．現北海道大学高等教育機能開発総合センター　池田　文人准教授には，エヌ・ティ・ティ・データに同期で入社した同僚として，また同じ社会人学生として，数多くの議論を通じて有益なご助言をいただきました．心より感謝いたします．また，ここには書ききれなかった株式会社エヌ・ティ・ティ・データの先輩・同僚・後輩の皆様には本研究を進める上で様々なご支援，ご助言を頂きました．深く感謝いたします．

最後に，本論文を作成するにあたって自宅に居る時間の大部分を論文作成や研究内容の検討に費やし，夫・父親としての役目を十分に果たせなかった私を影ながら支えてくれた妻　幸江と娘　徳香に心から感謝いたします．

参考文献

[1] H. Agrawal, R. A. Demillo, and E. H. Spafford, “Debugging with Dynamic Slicing and Backtracking”, Software-Practice and Experience, Vol.23, No.6, pp.589-616, 1993.

[2] H. Agrawal, “On Slicing Programs with Jump Statements”, In Proc. ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, pp.302-312, 1994.

[3] H. Akaike, “A New Look at the Statistical Model Identification”, IEEE Transactions on Automatic Control, Vol. 19, No. 6, pp.716-723, 1974.

[4] M. A. de Almeida, H. Lounis, and W. L. Melo, “An Investigation on the Use of Machine Learned Models for Estimating Correction Costs”, In Proc. 20th International Conference on Software Engineering, pp.473-476, 1998.

[5] B. S. Baker, “A Program for Identifying Duplicated Code”, In Proc. 24th Symposium on the Interface: Computing Science and Statistics, pp. 49-57, 1992.

[6] B. S. Baker, “On Finding Duplication and Near-Duplication in Large Software System”, In Proc. 2nd IEEE Working Conference on Reverse Engineering (WCRE’95), pp. 86-95, 1995.

[7] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K.A. Kontogiannis, “Measuring Clone Based Reengineering Opportunities”, In Proc. 6th IEEE International Symposium on Software Metrics (METRICS ’99), pp. 292-303, 1999.

[8] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. A. Kontogiannis, “Partial Redesign of Java Software Systems Based on Clone Analysis”, In Proc. 6th IEEE Working Conference on Reverse Engineering (WCRE ’99), pp. 326-336, 1999.

[9] T. Ball, and S. Horwitz, “Slicing Programs with Arbitrary Control-Flow”, In Proc. 1st International Workshop on Automated and Algorithmic Debugging (Lecture Notes in Computer Science 749), pp.206-222, 1993.

[10] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees", In Proc. IEEE International Conference on Software Maintenance (ICSM'98), pp. 368-377, 1998.

[11] D. R. Chase, M. Wegman, and F. K. Zadeck, "Analysis of Pointers and Structures", In Proc. the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, pp.296-310, 1990.

[12] 千葉 滋, "プログラミング, これからの１０年", オブジェクト指向最前線 2002, 近代科学社, pp. 213-218, 2002.

[13] S. R. Chidamber, and C.F. Kemerer, "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, Vol. 20, No. 6, pp. 476-493, 1994.

[14] J. Choi, M. Burke, and P. Carini, "Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects", Conference Record of 20th ACM Symposium on Principles of Programming Languages, pp.232-245, 1993.

[15] A. Coen-Porisini, F. D. Paoli, C. Ghezzi, and D. Mandrioli, "Software Specialization Via Symbolic Execution", IEEE Transactions on Software Engineering, Vol.17, No.9, pp.884-899, 1991.

[16] A. Deutsch, "Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting", In Proc. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation, pp.230-241, 1994.

[17] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code", In Proc. IEEE International Conference on Software Maintenance (ICSM'99), pp. 109-118, 1999.

[18] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data", IEEE Transactions on Software Engineering, Vol. 27, No. 1, pp. 1-12, 2001.

[19] N. E. Fenton, "Software Metrics: A Rigorous Approach", Chapman & Hall, London, 1991.

[20] M. Fowler 著, 児玉 公信, 友野 晶夫, 平澤 章, 梅澤 真史訳, “リファクタリング：プログラミングの体質改善テクニック”, ピアソン・エデュケーション, 2000.

[21] P. Fritzson, T. Gyimothy, M. Kamkar, and N. Shahmehri, “Generalized Algorithmic Debugging and Testing”, In Proc. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Vol.26, No.6, pp.317-326, 1991.

[22] 富士 仁, 古山 恒夫, 菅野 文友, “ソフトウェアの欠陥密度に影響を与える複雑さの特質とその尺度の分析”, 品質(日本品質学会), Vol.26, No.3, pp.91-101, 1996.

[23] 二村 良彦, 淵 一博, 古川 康一, 溝口 文雄, “プログラム変換”, 共立出版, 1987.

[24] K. B. Gallagher, and J. R. Lyle, “Using Program Slicing in Software Maintenance”, IEEE Transactions on Software Engineering, Vol.17, No.8, pp.751-761, 1991.

[25] A. R. Gray, and S. G. MacDonell, “A Comperison of Techniques for Developing Predictive Models of Software Metrics”, Information and Software Technology Vol. 39, pp. 425-437, 1997.

[26] D. Gusfield, “Algorithms on Strings, Trees, and Sequences”, pp. 89-180, Cambridge University Press, 1997.

[27] M. H. Halstead, “Elements of Software Science”, Elsevier Computer Science Library, 1977.

[28] 肥後 芳樹, 楠本 真二, 井上 克郎, “コードクローン分析ツール Gemini を用いたコードクローン分析手法”, 電子情報通信学会技術研究報告, SS2005-30, Vol.105, No.228, pp.37-42, 2005.

[29] S. Horwitz, and T. Reps, “The Use of Program Dependence Graphs in Software Engineering”, In Proc. 14th International Conference on Software Engineering, pp.392-411, 1992.

[30] J. H. Johnson, “Identifying Redundancy in Source Code Using Fingerprints”, In Proc. IBM Centre for Advanced Studies Conference

(CAS CON'93), pp. 171-183, 1993.

[31] J. H. Johnson, "Substring Matching for Clone Detection and Change Tracking", In Proc. IEEE International Conference on Software Maintenance (ICSM'94), pp. 120-126, 1994.

[32] C. Jones, "Applied Software Measurement Second Edition", McGraw-Hill, United States, 1997.

[33] E. Kamsities, and C. M. Lott, "An Empirical Evaluation of Three Defect-Detection Techniques", ISERN Annual Meeting, 1995.

[34] T. Kamiya, F. Ohata, K. Kondou, S. Kusumoto, and K. Inoue: "Maintenance Support Tools for Java Programs: CCFinder and JAAT", In Proc. 23rd International Conference on Software Engineering (ICSE2001), pp. 837-838, 2001.

[35] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a Multi-Linguistic Token-Based Code Clone Detection System for Large Scale Source Code", IEEE Transactions on Software Engineering, Vol.28, No.7, pp. 654-670, 2002.

[36] 小林 孝規, "入出力変数の制限情報を利用したプログラム簡素化ツールの試作", 大阪大学基礎工学部情報工学科特別研究報告, 1995.

[37] K. A. Kontogiannis, R. De Mori, E. Merlo, M. Galler, and M. Bernstein, "Pattern Matching Techniques for Clone Detection and Concept Detection", Journal of Automated Software Engineering, Kluwer Academic Publishers, vol. 3, pp.770-108, 1996.

[38] T. M. Khoshgoftaar, and D. L. Lanning, "A Neural Network Approach for Early Detection of Program Modules Having High Risk in the Maintenance Phase", Journal of Systems Software, Vol. 29, pp.85-91, 1995

[39] T. M. Khoshgoftaar, E. B. Alen, J. P. Hudepohl, and S. J. Aud, "Application of Neural Networks to Software Quality Modeling of a Very Large Telecommunication System", IEEE Transactions on Neural Networks, Vol.8, No.4, pp.902-909, 1997

[40] T. M. Khoshgoftaar, and Edward B. Allen, “Logistic Regression Modeling of Software Quality”, International Journal of Reliability, Quality and Safety Engineering, Vol. 6, No. 4, pp.303-317, 1999.

[41] T. M. Khoshgoftaar, and Edward B. Allen, “A Comparative Study of Ordering and Classification of Fault-Prone Software Modules”, Emprical Software Engineering, Vol. 4, pp.159-186, 1999.

[42] B. Korel, ”PELAS-Program Fault-Locating Assistant System”, IEEE Transactions on Software Engineering, Vol.14, No.9, pp.1253-1260, 1988.

[43] B. Laguë, E. M. Merlo, J. Mayrand, and J. Hudepohl, “Assessing the Benefits of Incorporating Function Clone Detection in A Development Process”, In Proc. IEEE International Conference on Software Maintenance (ICSM’97), pp.314-321, 1997.

[44] W. Landi, and B. G. Ryder, “A Safe Approximate Algorithm for Interprocedural Pointer Aliasing”, In Proc. the ACM SIGPLAN '92 Conference on Programming Languages Design and Implementation, pp.235-248, 1992.

[45] J. R. Lyle, and M. Weiser, “Automatic Program Bug Location by Program Slicing”, In Proc. 2nd International Conference on Computers and Applications, pp.877-883, 1987.

[46] J. Marciniak, “Encyclopedia of Software Engineering”, Wiley Interscience, 1994.

[47] 松本 吉弘 監訳, “ソフトウェアエンジニアリング基礎知識体系-SWEBOK-“, オーム社, 2003.

[48] T. McCabe, ”A Complexity Measure”, IEEE Transactions on Software Engineering, Vol. SE-2, pp.308-320, 1976.

[49] J. Mayland, C. Leblanc, and E. M. Merlo. “Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics”, In Proc. IEEE International Conference on Software Maintenance (ICSM’96), pp. 244-253, 1996.

[50] A. Monden, S. Sato, K. Matsumoto, and K. Inoue, “Modeling and

Analysis of Software Aging Process", In Proc. International Conference on Product Focused Software Process Improvement (Profes2000), Lecture Notes in Computer Science, Vol. 1840, pp. 140-153, 2000.

[51] A. Monden, S. Sato, and K. Matsumoto, "Capturing Industrial Experiences of Software Maintenance Using Product Metrics", In Proc. 5th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI2001), Vol. 2, pp. 394-399, 2001.

[52] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software Quality Analysis by Code Clones in Industrial Legacy Software", In Proc. 8th IEEE International Software Metrics Symposium (METRICS2002), pp. 87-94, 2002.

[53] K. Naoi, and N. Takahashi, "Detection of Infeasible Paths with a Path Dependence Flow Graph", Transactions of the Institute of Electronics, Information and Communication Engineers, Vol.J76D-I, No.8, pp.429-439, 1993.

[54] 西松 顕, 西江 圭介, 楠本 真二, 井上 克郎, "フォールト位置特定におけるプログラムスライスの実験的評価", 電子情報通信学会論文誌 D-I, Vol. J82-D-I, No.11, pp.1336-1344, 1999.

[55] N. Ohlsson, and H. Alberg, "Predicting Fault-prone Software Modules in Telephone Switches", IEEE Transactions on Software Engineering, Vol. 22, No. 12, pp. 886-894, 1996.

[56] 大平 雅雄, 横森 励士, 阪井 誠, 岩村 聡, 小野 英治, 新海 平, 横川 智教, "ソフトウェア開発プロジェクトのリアルタイム管理を目的とした支援システム", 電子情報通信学会論文誌 D-I, Vol.J88-D-I, No.2, pp.228-239, 2005.

[57] H. D. Pande, W. A. Landi, and B. G. Ryder, "Interprocedural Def-Use Associations for C Systems with Single Level Pointers", IEEE Transactions on Software Engineering, Vol.20, No.5, pp.385-402, 1994.

[58] M. Pighin, and R. Zamolo, "A Predictive Metric Based on Discriminant Statistical Analysis", In Proc. 19th International Conference on Software Engineering, pp.262-270, 1997.

[59] A. A. Porter, L. G. Votta Jr., and V. R. Basili, “Comparing Detection Methods for Software Requirements Inspections: a Replicated Experiment”, IEEE Transactions on Software Engineering, Vol.6, No.6, pp.563-575, Jun, 1995.

[60] A. A. Porter, and R. W. Selby, “Empirically Guided Software Development Using Metrics-Based Classification Trees”, IEEE Software, pp. 46-54, 1990.

[61] G. Ramalingam, “The Undecidability of Aliasing”, ACM Transactions on Programming Languages and Systems, Vol.16, No.5, pp.1467-1471, 1994.

[62] P. Rook 著, 菅野 文友, 大森 晃共訳, “ソフトウェア信頼性ハンドブック”, 日科技連, 1995.

[63] 佐藤 慎一, 植田 良一 井上 克郎 “再帰やポインタを含むプログラムの効率的な依存関係解析法の提案”, 信学技報, SS95-37, Jan 1996.

[64] N. F. Schneidewind, and C. Ebert, “Preserve of Redesign Legacy Systems?”, IEEE Software, Vol. 15, No.4, pp.14-17, 1998.

[65] R. W. Selby, and A. A. Porter, “Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis”, IEEE Transactions on Software Engineering, Vol.14, No.12, pp.1743-1757, 1988.

[66] T. Shimomura, “Bug Localization Based on Fault-Cause-Chasing Methods”, Transactions of Information Processing Society of Japan, Vol.34, No.3, pp.489-500, 1993.

[67] 下村 隆夫, “プログラムスライシング技術と応用”, 共立出版, 1995.

[68] H. M. Sneed, “Economics of Software Re-Engineering”, Journal of Software Maintenance: Research and Practice, Vol.3, No.3, pp.163-182, 1991.

[69] H. M. Sneed, “Planning the Reengineering of Legacy Systems”, IEEE Software, Vol.12, No.1, pp.24-34, 1995.

[70] 高橋　良英, 村岡　洋一, 中村　行宏, “ソフトウェア品質分類木の生成・評価方法”, 電子情報通信学会論文誌 D-I, Vol. J81-D-I, No. 4, pp.393-404,

1998.

[71] 高橋　良英, “モジュール相互間の関係複雑度に基づくソフトウェア品質評価モデル-ネストによる例題”, 電子情報通信学会論文誌 D-I, Vol.J79, No.5, 1996.

[72] 高橋　良英, 若山　博文, “判別効率によるソフトウェア品質判別モデルの評価方法”, 電子情報通信学会論文誌 D-I, Vol.J77-D-I, No.10, pp.663-673, 1994.

[73] 高橋　良英, 中村　行宏, “流用率と流用部・改造部間のインタフェースの複雑さによるソフトウェア保守品質評価方法”, 電子情報通信学会論文誌 D-I, Vol.J80-D-I, No.5, pp.441-449, 1997.

[74] 高谷 暢之, “入出力の制限情報を利用したプログラム簡素化手法の提案”, 大阪大学大学院基礎工学研究科修士論文, 1994.

[75] J. Tian, and J. Troster, “Comparison of Measurement and Defect Characteristics of New and Legacy Software Systems”, Journal of Systems and Software, Vol. 44, 135-146, 1998.

[76] 植田 良一, 練 林, 井上 克郎, 鳥居 宏次, “再帰を含むプログラムのスライス計算法”, 電子情報通信学会論文誌 D-I, Vol.J78-D-I, No.1, pp.11-22, 1995.

[77] J. D. Valett, “The Practical Use of Empirical Studies for Maintenance Process Improvement”, Empirical Software Engineering, Vol.2, No.2, pp. 133-142, 1997.

[78] G. A. Venkatesh, “The Semantic Approach to Program Slicing”, In Proc. the ACM SIGPLAN '91 Conference on Programming Languages Design and Implementation, pp.107-119, 1991.

[79] G. A. Venkatesh, “Experimental Results from Dynamic Slicing of C Programs”, ACM Transactions on Programming Languages and Systems, Vol.17, No.2, pp.197-216, 1995.

[80] M. Weiser, “Program Slicing”, In Proc. 5th International Conference on Software Engineering, pp.439-449, 1981.

付録

A. 部分評価手順の詳細

4.2.3節で述べた部分評価における各文の詳細な部分評価手順を示す．なお，この内容は，文献[36]および[74]の該当箇所を引用するとともに必要に応じて変更を加えたものである．

まず，簡単のために対象プログラムは以下の文およびそれらを含む主プログラムとそこから呼び出される関数および手続きから構成されるものとする．なお，*expr* は式を表す．

(1) 空文：φ

(2) 複合文：begin $S_1;S_2;...;S_n$ end

(3) 代入文：x := *expr*

(4) 入力文：readln(x)

(5) 出力文：writeln(x)

(6) 分岐文：if *expr* then S_1 else S_2

(7) 繰り返し文：while *expr* do *S*

(8) 手続き呼出文：procedure_call(*expr*)

これらの文は，与えられたプログラム状態 *State* のもとで文 *S* を部分評価した結果の文を返す関数 *simple*(*S*,*State*)を実行することによって部分評価される．関数 *simple* は与えられたプログラム状態 *State* で文 *S* を部分評価した結果の新しいプログラム状態を返す関数 *exec*(*S*,*State*) とともに定義される．

以下，各文における関数 *simple* および *exec* の出力結果について述べる．

A.1 空文

空文φは，これ以上部分評価されない．したがって，*simple*,*exec* は以下のように定義される．

simple(φ,*State*) = φ

exec(φ,*State*) = *State*

A.2 複合文

複合文(begin $S_1;S_2;...;S_n$ end)は文 S_1 から順番に部分評価していくが，文 S_i で与えられるプログラム状態は文 S_{i-1} までを部分評価した後の状態となる．よって，複合文の *simple* 関数と *exec* 関数は次のように再帰的に定義される．

simple(begin $S_1;S_2;...;S_n$ end,*State*)

= *simple*(S_1,*State*);*simple*(begin $S_2;...;S_n$ end, *exec*(S_1,*State*))

exec(begin $S_1;S_2;...;S_n$ end,*State*)

= *exec*(begin $S_2;...;S_n$ end, *exec*(S_1,*State*))

A.3 代入文

代入文では右辺の式 *expr* の中で参照される変数のうち，プログラム状態 *State* の中で値が確定しているものはその値に置き換えられる．

simple(x:=*expr*,*State*) = x:=*eval*(*expr*,*State*)

exec(x:= *expr*,*State*) = *State'*

ここで，関数 *eval*(*expr*,*State*)は，プログラム状態 *State* のもとで式 *expr* を部分評価した式を返す．この結果は，*State* において値が確定している変数をすべてその値で置き換えるとともに，計算可能となった場合にはその値を返す．また，式の中に関数 F_i の呼び出しを含んでいる場合には，手続き呼び出し文の *simple* および *exec* 関数を実行して，適宜その結果を反映する．

また，*State'*は，*eval*(*expr*,*State*)を実行した結果，式 *expr* が定数値 n に確定した場合は，状態 *State* に含まれる変数 x に関するものの値を n に，そうでない場合には⊥となるように変更した状態である．

A.4 入力文

部分評価のために入力変数の値が定数に固定されている場合は，この入力文を対応する代入文に変換する．

$$simple(\text{readln(x)},State) = \begin{cases} \text{x:=n} & (eval(\text{x},State)\text{が定数値 n を返すとき}) \\ \text{readln(x)} & (\text{それ以外のとき}) \end{cases}$$

入力文は値を入力変数に割り当てるため，状態にも変更が生じるが，この作業は初期状態を設定した時点で行われている．よって，入力文の *exec* 関数は以下のようになる．

exec(readln(x),*State*) = *State*

A.5 出力文

出力文では引数で指定された変数の値が決定している場合に置き換えればよい．

simple(writeln(x),*State*) = writeln(*eval*(x,*State*))

出力文自体は変数の持つ値を変更することはない．よって出力文の *exec* 関数は

次のようになる.

$exec$(writeln(x),$State$) = $State$

A.6 分岐文

プログラム状態 $State$ により条件式 $expr$ の値が true に決定される場合は S_1 のみ，false に決定される場合は S_2 のみが実行される．しかし，一意に定められない場合には両方の文が部分評価される.

$$simple(\text{if } expr \text{ then } S_1 \text{ else } S_2, State) = \begin{cases} simple(S_1, State) & (eval(expr, State)=\text{true のとき}) \\ simple(S_2, State) & (eval(expr, State)=\text{false のとき}) \\ \text{if } eval(expr, State) \text{ then } simple(S_1, State) \text{ else } simple(S_2, State) & (\text{それ以外のとき}) \end{cases}$$

then 節と else 節の両方が実行されたとき，その後のプログラム状態 $State$ は，それらの結果を併合したものとなる.

$$exec(\text{if } expr \text{ then } S_1 \text{ else } S_2, State) = \begin{cases} exec(S_1, State) & (eval(expr, State)=\text{true のとき}) \\ exec(S_2, State) & (eval(expr, State)=\text{false のとき}) \\ exec(S_1, State) \circ exec(S_2, State) & (\text{それ以外のとき}) \end{cases}$$

ここで，演算∘は，2 つのプログラム状態を併合する演算で，以下のように定義される.

$$State_1 = \left\{ \left\langle x_1, v^1_{x_1} \right\rangle, \ldots, \left\langle x_n, v^1_{x_n} \right\rangle \right\}, \quad State_2 = \left\{ \left\langle x_1, v^2_{x_1} \right\rangle, \ldots, \left\langle x_n, v^2_{x_n} \right\rangle \right\}$$

で表現される 2 つのプログラム状態 $State_1$ と $State_2$ があったとき，これらに演算∘を適用した時の結果は次のようになる.

$$State_1 \circ State_2 = \left\{ \left\langle x_1, v^3_{x_1} \right\rangle, \ldots, \left\langle x_n, v^3_{x_n} \right\rangle \right\} \quad \text{ただし，} \quad v^3_{x_i} = \begin{cases} v^1_{x_i} & (v^1_{x_i} = v^2_{x_i} \text{のとき}) \\ \perp & (v^1_{x_i} \neq v^2_{x_i} \text{のとき}) \end{cases}$$

この意味は，変数 x_i が両方のプログラム状態 $State_1$ と $State_2$ で同じ値を持つならば新しいプログラム状態においてもその値が保存され，そうでなければ値は⊥となるということである.

A.7 繰り返し文

繰り返し文では，分岐文と同様に部分評価結果はプログラム状態 *State* で条件式 *expr* がどのような値をとるかに依存する．ただし，繰り返し条件によっては各文が複数回実行されることがあるため,繰り返し文を最初に実行する際の状態と，2 回目以降に実行する際の状態を併合する必要が生じる場合がある．

simple(while *expr* do *S*,*State*)

$$=\begin{cases} simple(S,State);simple(\text{while } expr \text{ do } S,exec(S,State)) & \\ \qquad (eval(expr,State)=\text{true のとき}) & \\ \phi \qquad (eval(expr,State)=\text{false のとき}) & \\ \text{while } expr \text{ do } simple(S,State \circ exec(S,State)) & (\text{それ以外のとき}) \end{cases}$$

exec 関数は次のようになる．

exec(while *expr* do *S*,*State*)

$$=\begin{cases} exec(\text{while } expr \text{ do } S,exec(S,State)) & (eval(expr,State)=\text{true のとき}) \\ State & (eval(expr,State)=\text{false のとき}) \\ State \circ exec(S,State) & (\text{それ以外のとき}) \end{cases}$$

なお，部分評価はプログラムの実行速度の効率の向上も目的としているため，上記の手順では条件式 *expr* が true と評価できた場合には文 *S* を通常の文としてループの外に抽出することとしている．しかし，試作ツール実装上はその抽出回数に上限を設定し，その数が一定数を超える場合には，繰り返し文のままで部分評価することとしている（「それ以外のとき」として扱う）．

A.8 手続き呼び出し文

手続き procedure に関する手続き呼び出し文(procedure_call と表記する)では，値渡しの引数のうち入力制限によって値のわかっているものだけをその値に変換しておく．また，手続き自体の部分評価を行なうために，呼び出した時のプログラム状態 *State* をプログラム状態集合 F_{iState} に保存しておく．

simple(procedure_call(*expr*),*State*)

$$=\begin{cases} \text{procedure\_call}(eval(expr,State)) & (expr \text{ が値渡しである時}) \\ \text{procedure\_call}(expr) & (\text{それ以外のとき}) \end{cases}$$

手続き呼び出し文の部分評価を行う際，この時点では呼び出される手続き自体は部分評価されないため，手続き内で大域変数のプログラム状態が変更されるよ

うな場合にも手続き呼び出し文以降の文にその影響が反映されないため，その影響を考慮する必要がある．また，引数として変数渡しされる変数についても同様である．

exec(procedure_call(*expr*),*State*) = *State'*

ここで，*State'*は *State* に含まれる変数の状態のうち procedure を実行した際に定義される可能性のある大域変数すべてについて⊥とし，さらに引数 *expr* が変数渡しで，かつその変数が procedure 内で定義される可能性がある場合には，*expr* 中の該当する変数の値を⊥としたものである．これらの大域変数および引数が procedure を実行した際に定義される可能性があるかどうかは，依存関係解析の結果得られる各手続きもしくは関数において定義される変数の情報から求めることができる．