JAIST Repository
https://dspace.jaist.ac.jp/

Title	ソースプログラムの差分解析と波及解析に基づく開発状況の把握法に関する研究
Author(s)	林崎，浩典
Citation	
Issue Date	2000-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1328
Rights	
Description	Supervisor:落水 浩一郎，情報科学研究科，修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

ソースプログラムの差分解析と波及解析に基づく開発状況の把握法に関する研究

指導教官　落水 浩一郎 教授

北陸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

林崎 浩典

平成 12 年 3 月 30 日

Copyright © 2000 by Hironori Hayashizaki

目 次

1 はじめに **1**

1.1 本研究の背景 1

1.2 本研究の目的 2

1.3 本論文の構成 2

2 プログラムの変更管理・影響解析に関する研究の現状 **4**

2.1 差分抽出 4

2.2 影響解析 5

2.3 変更履歴の追跡 6

2.4 まとめ 7

3 競合解析 **8**

3.1 解析の方針と処理の流れ 8

3.2 システム依存グラフ 10

3.3 グラフ作成 15

3.4 システム依存グラフ間の差分解析 16

3.5 ブランチ間の波及解析 20

3.6 差分解析と波及解析に基づく競合の検知 21

3.6.1 競合の検知 21

3.6.2 競合解析の範囲 22

4 プロトタイプシステムの設計 **25**

4.1 設計方針 25

4.2 ユーザ・インタフェースの設計 25

4.2.1 全体構成 26

4.2.2 マーキング機能 26

4.2.3 詳細情報機能 27
4.3 解析システムの設計 28

5 プロトタイプシステムの実装 30
5.1 システムの利用法 30
5.2 システムの構成 30
5.3 ソースプログラムの解析 32
5.3.1 データの準備 32
5.3.2 クラス解析サブシステム 34
5.3.3 波及解析サブシステム 35
5.3.4 競合検知サブシステム 35
5.4 システムの機能 35

6 議論 39
6.1 未来版管理システムへの応用 39

7 おわりに 42
7.1 まとめ 42
7.2 今後の課題 42

参考文献 45

図目次

3.1 解析の流れ . . . 9
3.2 システム依存グラフの例 . . . 11
3.3 ソースプログラム . . . 12
3.4 差分の抽出 . . . 16
3.5 システム依存グラフ間の対応 . . . 18
3.6 変更作業による影響の波及 . . . 20
3.7 競合の検知 . . . 22
3.8 戻り値に影響する変更例 . . . 24

4.1 ソースプログラムへのマーキング . . . 26

5.1 システムの利用 . . . 31
5.2 システムの構成図 . . . 32
5.3 クラス解析と対応関係 . . . 34
5.4 競合検知の例 . . . 36
5.5 マーキング . . . 37
5.6 詳細情報 . . . 37
5.7 全体画面 . . . 38

第 1 章

はじめに

1.1 本研究の背景

ソフトウエア開発保守現場では、顧客、市場ニーズや管理部門などから舞い込む数多くの変更要求に対応する必要がある。また、近年の PC 関連のソフトウエアは、新規機能が拡張し続けられており、迅速な開発が求められている。すなわち、変更要求ごとに同一の成果物を元にして、開発者ごとに分担して並行に作業を進めることが求められる。

しかし、変更作業がシステムの機能拡張やデバッグといった異なる要求の下で行われていても、将来的には個々の変更を単一の製品として統合する必要があり、それぞれの開発の間で変更方針や影響を考慮しながら、マージを行う際の後戻りなどを防ぐ必要がある。

このような過酷な条件において、作業を迅速かつ円滑に進めるためには競合をいち早く察知することが重要である。開発自身に関連する競合が発生している状況を把握できれば、その効果は高いと考えられる。すなわち、競合の状況を把握し、必要ならばメンバーと調整を行い、間違いが間違いを産み出すような状況を避けなければならない。

この場合、悪循環を避けるためにメンバー間で電子メールなどを用いて連絡が交わされるが、自然言語によるコミュニケーションでは、メンバー間の認識の不一致が生じることもあり、事故や混乱を招く可能性を完全には否定できない。また、メンバー間の認識に何の保証もないので、実際にマージしてデバッグするまで本当に認識が一致していたのかはどうかは分からない。つまり、開発者は、自身の知らないところで自分に関係のある変更が行われているかもしれないという、不安定な状況下で開発を進めなければならなくなる。

本研究においては、上記のような状況を把握するための手段をソースプログラムから得られる情報に限定して、ソースプログラムを解析することで、状況を反映している部分

を取り出し、状況の確認を可能にする手法を検討する。構成管理システムは、開発過程で作り出されたソースプログラムや仕様書をリポジトリに保存し、円滑な開発を支援することをねらいとしている。構成管理システムには、過去の開発の有用な情報が蓄積されている。開発者は、過去のバージョンからの変更の経緯を知るために、複数のバージョンに渡って、成果物の差分やリポジトリから作業ログ情報を得て、手作業で解析を行い、その内容を理解し、必要な情報を見つけ出すことが可能である。しかし、このような作業はコストが高い上に手作業であるために誤りも生じやすい。

また、プログラム理解は開発で多大な労力を費やす作業でもあり、現在までに、理解を支援するための多くの研究がなされてきた。プログラム解析などの技術は、開発支援において保守やテスティングを正確に、または効率良く実現する目的で、主に利用されている。しかし、プログラムの変更部分や作業量を把握し、開発者が理解の必要な部分の見当をつけるために、変更作業の視覚化や変更履歴の追跡に関する研究を発展させる必要がある。

1.2 本研究の目的

本研究では、CVS 等のバージョン管理システム内の別のブランチ間で、将来的にプログラムのコンパイルや動作に影響を及ぼすような変更が行われたことを察知し、作業状況を把握するシステムを検討する。また、オブジェクト指向言語を対象とし、差分抽出や波及解析の技法を組み合わせた、競合を検知する手法の開発を目的とする。具体的には、各手法を組み合わせて競合の検知の手法を以下の通りに定義し、一部のプロトタイプ実装を示す。

1. Java 言語で記述されたソースプログラムをシステム依存グラフに変換する。
2. 差分として、同じブランチ上のシステム依存グラフ間の節の対応を解析する。
3. 波及として、異なるブランチ間のシステム依存グラフの対応を解析する。
4. 競合として、異なるブランチからの波及の全体として、解析する。このときに、差分の解析などの情報を用いて影響の波及を検知する。

1.3 本論文の構成

本論文の構成について説明する。

- 第 2 章では、差分解析、影響解析、変更履歴の追跡に関する研究と現状を紹介し、その成果と課題について述べる。
- 第 3 章では、並行作業における開発状況の把握法として、プログラムの解析を利用した競合の検知について述べる。
- 第 4 章では、プロトタイプシステムの設計として、ブランチ間の状況を把握するためのユーザ・インタフェースと解析システムの設計について述べる。
- 第 5 章では、第 4 章に述べた設計に従ったプロトタイプシステムの実装について説明する。
- 第 6 章では、本研究で提案した把握法を未来版管理システムの汚染マーク機構への応用について述べる。
- 第 7 章では、本研究のまとめと今後の課題について述べる。

第 2章

プログラムの変更管理・影響解析に関する研究の現状

本章ではソースプログラムに対して加えられた変更を識別する技法および、影響を解析する手法について取り上げ、その現状と有用性について述べる。関連研究として、差分抽出、影響解析、変更履歴の追跡の 3 項目に注目する。差分抽出は、ソースプログラムへの作業内容を理解するために有効である。影響解析は、作業の結果として影響を与える部分を特定することなどを知るために有効な手段である。変更履歴の追跡は、作業内容の変化を捉えるために有効な手段である。

2.1 差分抽出

差分抽出法には、テキストベースと意味的な差分の抽出がある。それらのツールは、効率の良い開発履歴の保存やプログラムの理解支援といった差分抽出の目的の違いに根付いた特徴を持つ。

テキストベースの差分抽出ツールとして、UNIX diff がよく知られている。このツールは、RCS [2] や CVS といったバージョン管理システムで用いられている。その目的は、ソースプログラムや仕様書などの種々の成果物の変更履歴を効率良く保存することにある。これらのツールはテキストの行ごとに差分を抽出するので、汎用性は高いといえる。しかし、変更されていない部分まで抽出されるため、意味的な差分ではない部分まで抽出されてしまい、開発者が変更の内容を理解するのが困難である場合がある。

ソースプログラムの意味的な差分を抽出する技術として、ソースプログラムを構文木レベルで解析し、構文木に基づいた差分抽出がある。これは、ソースプログラムに含まれる

コメント文などのプログラムの実行に影響を及ぼさない部分を抽出することがなく、開発者が理解しやすい差分を抽出できるといった利点がある。しかし、構文木の比較を行っているため、制御文の削除や追加に対してその制御文が含むすべてのステートメントが削除または追加の対象となるという問題がある [6]。

差分抽出ツールを用いて競合を見つけ出すには、変更を加えられる前後のソースプログラムから差分を抽出し、その差分から変更作業の内容を理解し、影響を与える箇所を発見するという作業が必要である。差分抽出は有用であるが、上述したように、変更の内容を理解するためにはいくつかの手順と作業内容の理解、その変更が影響を及ぼすソースプログラムとの対応関係の理解が必要であり、これらの作業を行うには、多くの労力を必要とする。

2.2 影響解析

プログラムスライシング

プログラムスライシングは Weiser によって提案された手法である。プログラムスライスは、スライシング基準を設定し、その基準に影響を及ぼす命令を制御依存関係やデータ依存関係を解析することで抽出される [1]。この抽出されたプログラムスライスは実行可能である。

プログラムスライシングなどのプログラム解析技法では、構文とデータフロー等の依存関係をグラフにしてプログラムを表現した依存関係グラフを利用する。また、オブジェクト指向のプログラムは、他のオブジェクトに定義されている操作を呼び出し、そのオブジェクトの持つ属性を変化させる手続きにより構成される。そのため、オブジェクトの操作、つまり、プロシジャ間に渡ってプログラムを解析する必要がある。本研究では、3.2 節で述べるオブジェクト指向プログラムのためのシステム依存グラフを利用して、解析を行う。

CFGCM

ソースプログラムに対する変更を構文木レベルで構成管理を行う CFGCM（Context Free Grammar based Fine-Grained Configuration Management）[7] が提案されている。従来の構成管理システムにおいては、ソースプログラムの変更を行単位で認識しているため、変更を正確に認識できず変更管理の支援が困難であった。このシステムは、変更が及

ぼす影響範囲にあたるプログラムの構文木をロックする手法を用いた解決する手段を与えて、変更管理の支援を行っている。

システムの利用方法は、利用者が初めに変更の計画をよく練り、作業するプログラムの一部分を指定することで作業範囲をシステムに知らせる。システムと対話的に影響を及ぼす範囲を決めた上で、実際の作業を行った後に結果を保存する。この作業の進め方により、作業箇所と影響範囲を前もって特定することができる。

一連の開発シナリオで変更の影響範囲を特定するために、プログラムの構文で影響が伝播するための依存関係を定義している。依存関係として、関数・手続き呼び、変数フロー、関数戻り値関係、変数、定数定義、型定義、has-a 関係、引数間関係があり、それらをソースプログラムから解析し依存グラフを構築する。影響はこれらの依存関係とプログラムの制御構造にしたがって伝播する。例えば、代入文 foo = a + b があるとき、この代入文の右辺式が変更すると左辺式の変数 foo に影響を与えることになる。さらに、foo が関数の戻り値であれば、その関数を利用している部分にも影響を与えることになる。例で示したように伝播は再帰的に影響する範囲を解析を行う。従って、影響は広範囲に渡ることになる。文献 [7] では、影響が連鎖する回数を設定するなどして、ロックする領域と変更により作業に影響を受ける利用者への通知を限定する方法を述べている。

2.3 変更履歴の追跡

変更履歴の追跡は、バージョン管理システムなどによって蓄積された過去の情報から、有用な情報を抽出して、過去のバージョンからの変更経緯を知りることで、バグや再利用等の開発に役立つ情報を得るために必要な活動である。

開発履歴を視覚化し、ユーザにシステム開発の進行状況の把握を支援するシステムが開発されている。例えば、SeeSys [3] が挙げられる。特に大規模システムの開発において現在進行中の開発はどこで行われているのか、どの部分にエラーを含む傾向があるのか把握し、プロジェクトのマネジメントに役立てるときがある。

SeeSys はファイル、ディレクトリ、サブシステムに関連づけられた特徴のソフトウエアメトリクスを表示する視覚化システムである。視覚化による支援は、コメントを含まないソースプログラムの部分に着目して、特徴として修正された数を把握することができる。つまり、前述したファイル等の成果物に対して、どこに、どれだけの修正が加えられたのか知ることができる。ただし、このシステムは、複数のバージョンに渡って開発の経緯を調べたい場合に、理解が困難であるという問題をもつ。

また、複数バージョン間の変更の経緯を把握する研究がある。テキスト行ごとの比較

では困難であるプログラムの詳細な対応関係の理解を支援する手法が提案されている [4]。ソースプログラムの解析は構文木レベルで行い、2 つの木構造の最大共通部分グラフを求めるアルゴリズムを利用して、バージョン間の変更と構文木の対応関係を識別している。複数のバージョン間での対応関係を得るために、構文木間の共通する構文の頂点を結び付けて解決している。この手法を用いて、C 言語で記述されたソースプログラムを対象とした複数のバージョン間の関数の対応関係を一覧表示できるツールも開発されている。ツールは関数ごとの変更されたステートメントの数を追加・削除のレベルで表示する。また、ネストの深さごとに変更されたステートメント数を表示する機能がある。それらの機能は、特定のプログラムに集中して変更が行われていることを示す結果があるときに、プログラムが複雑になりすぎているなどの推測を立て、実際に作業ソースコードを調査するための手がかりとして利用できる。

2.4 まとめ

成果と課題

差分抽出

- 構文木レベルでの解析は、ステートメント追加・削除などの作業内容を理解するために有用であるが、修正や移動といった高度に意味的な解析は完全には行えない。また、差分抽出の限界は、バージョン間でソースプログラムを比較することの限界ともなる。

影響解析

影響解析を行う場合にはプログラムの依存グラフを定義し、定義されたいくつかの依存関係をもとにデータ依存等からプログラムのスライスを取り出す。また、プログラムの静的な解析を行う場合には、次のことが言える。

- 静的な解析では、プログラムの条件分岐などは実行時でないと次の命令ステップが分からない。このため、確実に影響するかどうか正確には理解できない。
- また、ポインタ解析で近似することは可能であるが、完全には分からない。
- 影響が連鎖すると複雑かつ広範囲になり、理解しずらくなる恐れがある。

第 3 章

競合解析

第 1 章において、競合を検知する手法の提案を目的として述べた。本章では、システム依存グラフを用いて、競合を検知するためのソースプログラムの解析手法について述べる。

3.1 解析の方針と処理の流れ

本節では、提案する解析の概要について、その処理手順を示しながら説明する。本研究では、CVS 等のバージョン管理システムを用いて開発を進めいていることを想定としており、解析システムはバージョンツリーが更新されたことをきっかけとして、ソースプログラムの解析を始めることを考えている。また、バージョン管理システムを用いた開発では、プログラムの変更による影響が大きいと思われる場合などには、バージョンツリーのブランチを作成し、メインツリーから枝分かれして作業を進める。

図 3.1 は、開発者A と開発者B がバージョン管理システム内において、同じバージョンのソースプログラムを基にしてブランチを作成し、機能の拡張や保守などの作業を並行して進めている様子を表している。開発者B の立場から作業状況の把握するときに、開発者A の作業するプログラムを利用すれば、加えられた変更によって、開発者B の利用しているプログラムと異なっていると考えられる。従って、将来的に開発者A の変更したプログラムを利用する場合を仮定すると、変更作業による影響を与えると考えられ、メインツリーへマージする以前のブランチ上で行われているときに、影響が波及する可能性を検知し、開発者B が作業の状況の把握を支援する。

関連研究では、差分抽出、影響解析が個々に異なる目的に用いられる技法であった。本研究においては、それらの手法を組み合わせた競合の検知を目指す。よって、それらの解

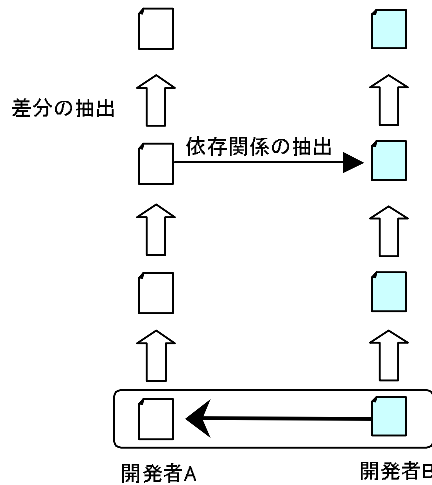


図 3.1: 解析の流れ

析を実現できるプログラムの表現を用いる必要があり、プログラムの構文木に近い表現のプログラム解析に活用される依存グラフを生成する。つまり、差分抽出や波及解析は、依存グラフに基づいて定義される手法である。以下に、解析の流れについて説明する。

1. システム依存グラフの作成

   はじめに、本研究における解析はシステム依存グラフを対象としているので、ソースプログラムを構文・意味解析し、システム依存グラフを作成する。システム依存グラフとは、制御依存関係やデータ依存関係などによりプログラムの構造を表現したグラフである。競合を検知するために、プログラミング言語仕様が規定する構文のみならず、制御依存関係やデータフローなどといったプログラムの記述の意味も解析することが必要である。

2. 差分の抽出

   差分の抽出は、あるブランチ上の（バージョンの異なる）システム依存グラフ間の対応づけを行い、対応のない部分を差分として取り出す。対応づけとは変更作業の内容を保存しておくために、2 つのグラフで同じ内容の節を探し出し、対応のあることを示す辺で結び付ける作業である。この処理により、対応のつかない部分を挿入または削除の操作により変更されたプログラムとして抽出することが可能となる。

3. 波及解析

このステップでは、異なるブランチ上のシステム依存グラフ間の対応づけを行う。例えば、図 3.1 において開発者B の作業したシステム依存グラフと、開発者A のブランチ上にあるシステム依存グラフの間にメソッド呼び出しなどの依存関係がある場合を対応があるといい、この対応を見つけ出す処理を行う。直感的には、開発者A と開発者B のプログラムを一緒にコンパイルまたは実行したときに、影響を及ぼすような変更を探し出すために用いられる。

4. 競合の解析

このステップでは、差分の抽出と波及解析の結果を利用して、競合を検知する。競合とは、あるブランチ上のバージョンに対して、他のブランチのシステム依存グラフからの波及の全体として定義する。波及解析の結果として得られた対応が、差分の抽出として得られた対応を見て変更がある場合に、変更による影響が波及するとして検知する。

3.2 システム依存グラフ

本研究では、Java 言語のソースプログラムの解析を対象としており、オブジェクト指向プログラムのためのシステム依存グラフを利用する。システム依存グラフについて、概要を説明する。

依存グラフは、プログラムの最適化、理解などの問題を解決するために重要な技術として利用されている。本研究では、オブジェクト指向言語 Java をソースプログラムの対象とており、Harrold らが提案し [5]、蜂須が Java 言語向けに拡張した [11] オブジェクト指向システム依存グラフを用いる。システム依存グラフは、プロシジャの集まり、つまり、Java 言語ではクラスやメソッドの集まりからなる main メソッドを含むシステム全体を表現することができる。この表現は、メソッド間に（呼び出し関係に）渡ってプログラムを解析するために必要とされる。これによって、パラメータや戻り値のデータ依存を正確に求めることが可能となる。

図 3.2 にシステム依存グラフの例を示す。また、図 3.3 に対応するソースプログラムも示す。以下に、図 3.2,3.3 を用いながら、用語の説明とシステム依存グラフの定義について述べる。

定義と使用

変数 x を**定義**するとは、変数 x に値を設定することを言う。また、変数 x を**使用**するとは、変数 x の値を参照することを言う。

図 3.2: システム依存グラフの例

文 S_1 における変数 x の定義が文 S_2 に**到達する**とは、S_1 に S_2 至るパスが存在して、変数 x はそのパスに含まれる文の中で、定義されていないことを言う。

データ依存関係

次の条件が満たされる場合に、文 S_1 から文 S_2 へデータ依存関係があるという。

- ある変数 x が存在して、
- 文 S_1 における変数 x の定義が文 S_2 に到達する。

例えば、図 3.3 の文 $S1$ の変数 a の定義は、文 $S4$ に到達するので、データ依存関係が存在する。図 3.2 $S1$ 節から $S4$ 節へは、*data dependence* 辺が張られている。

```
      class bar {
          public static void main( String[] args ){
S1:             int a = 1;
S2:             int b = 2;
                int c;
S3:             foo obj = new foo();
S4:             c = obj.add( a, b );
          }
      }

      class foo {
          int total;

          foo(){
S5:             total = 0;
          }
          int add( int a, int b ){
S6:             total = a + b;
S7:             return total;
          }
      }
```

図 3.3: ソースプログラム

制御依存関係

文 S_2 の実行が文 S_1 の実行結果に支配されているときに、S_1 から S_2 への制御依存があるという。例えば、図 3.2 中では、`main` から各文へ *control dependece* 辺が張られており、（番号が記されているので分かるが）左から実行順に並んでいる。

システム依存グラフの要素

次に、グラフを構成する節と辺について、その定義を示す。

- *class entry* 節（C）はクラスを表す。

- $method\,entry$ 節（M）はメソッドを表す。
- $method\,variable$ 節（V）はメンバ変数を表す。
- $statement\,entry$ 節（S）は文を表す。
- $parameter$ 節（P）はパラメータを表し、さらに 4 種類に分けられる。
  - 形式パラメータ（$formal_in$, $formal_out$）
  - 実パラメータ（$actual_in$, $actual_out$）
  - メンバ変数（$member_in$, $member_out$）
  - 戻り値（$return$, $returned$）
- $call$ 節（$CALL$）はメソッド呼び出しを表す。
- $polymorphic\,choice$ 節（PC）は動的なメソッド呼び出しを表す。
- $class\,member$ 辺（$C\times M$,$C\times V$）は、クラスがメソッドまたはメンバ変数を持つことを表す。
- $class\,inheritance$ 辺（$C\times C$）は、クラス間の継承関連を表す。
- $control\,dependence$ 辺（$M\times S$, $S\times S$,$S\times CALL$, $S\times PC$,$CALL\times P$）は、制御依存関係を表す。
- $date\,dependence$ 辺（$S\times S$, $P\times S$,$S\times P$）は、データ依存関係を表す。
- $member\,reference$ 辺（$P\times V$）は、メンバ変数との参照関連を表す。
- $call$ 辺（$CALL\times M$, $PC\times CALL$）は、メソッド呼び出しを表す。
- $parameter$ 辺（$P\times P$）は、形式パラメータと実パラメータ、$return$ と $returned$ 間の関連を表す。
- $summary$ 辺（$P\times P$）は、$actual_in$ から $returned$ へのデータ依存関係を表す。

図 3.2 を用いてシステム依存グラフの解説をする。クラス`foo`は、メソッド`main`を持ち、メソッドと文は、含み制御依存関係で結び付けられている。つまり、メソッド`main`は、文`S1`, `S2`, `S3`, `S4`から構成されるという関係にある。また、明示的に示されていないが左から実行される順に整列されている。メソッド`main`内でクラス`foo`のメソッド`add`から呼び出されていることを示している。例は、その呼び出し先、パラメータと戻り値を対応づけている。さらに、$summary$ 辺を求めることで、パラメータと戻り値との間に推移的なデータ依存関係があることを知ることができる。また、$member_in$,$member_out$ 節か

らメンバ変数 $total$ に $member\ reference$ 辺があるため、メソッド add 内で使用、定義されることを読み取ることができる。

Harrold らは C++のプログラムを例題として用いている。$summary$ 辺は、呼び出された側で形式パラメータの in と out 間にデータ依存関係があるなら、実パラメータの in と out 間に $summary$ 辺を作成する。しかし、Java 言語において参照型のパラメータ変数のときは、参照しているインスタンスへのポインタを渡す。よって、呼び出された側で形式パラメータ変数を定義するような文があったとしても、呼び出し側には影響しない。Java の場合には、$return$ から $formal_in$ パラメータに推移的なデータ依存があるとき、例で示した $summary$ 辺が作成される。

依存グラフによる表現の特徴

オブジェクト指向言語のためのシステム依存グラフの表現の特徴について述べる。本研究で利用しているグラフは、次の 2 項目について明確ではない。

- インスタンスの表現
  クラスの記述は定義なので、インスタンス化しなければ、オブジェクトにアクセスできない。つまり、プログラムを厳密に表現するためには、ある特定のクラスインスタンスを識別し、そのインスタンスに対してアクセスする表現を必要とする。この表現方法は正確であるが、大規模なプログラムの場合は、複雑なグラフになる。また、再帰的な呼び出しは、入れ子のように連鎖してグラフが生成されるため制限した表現法を工夫する必要がある。

- インスタンス変数・メソッド
  また、本研究で利用しているグラフは、インスタンスを区別していないために、静的にアクセスできる変数・メソッドなのか否か区別して表現していない。

ここで、採用したグラフとオブジェクト指向言語の特徴を正確に表現した場合のグラフの比較を行う。正確に解析した場合には、利用関係を明確に知ることができる。そのため、例えば、あるインスタンスのメソッドを呼び出して、インスタンス変数を定義した等のことが正確に理解でき、波及解析も正確にできると考えられる。しかし、採用したグラフの場合には、ソースプログラムに対応して変更箇所が理解しやすいなどの特徴が挙げられる。

3.3 グラフ作成

システム依存グラフを作成するには、下記に述べる作業が必要である。クラス作成では、ソースプログラムから依存グラフの構成要素となる情報を解析する。Japid を利用することにより、細粒度でプログラムの情報を得ることが出来が、それに加えて、以下の情報を計算する必要がある。

- 制御依存関係

  図 3.2 の示すように、メソッドは文に構成されるという関連を持ち、左から実行順に整列されている。メソッド呼び出しがある時には、呼び出し先の文を実行することになる。しかし、例外処理を考えてみると、例外が挙げられて（`throw`）、受け取った（`catch`）文に制御が移る。単純なテキストの並びと異なる実行の流れを表現することもあり、この場合を想定して、システム依存グラフを設計する必要がある。ただし、本研究においては、そのような例外処理を解析の範囲とせずに、簡単なシステム依存グラフを用いた解析を進める。

- データ依存関係

  メソッド内のデータフローを計算し、$data\,dependence$ 辺を作成する。具体的には、下記の式に示した関係を満たす集合を求めることを行う。

  $$out[S] = gen[S] \cup (in[S] - kill[S])$$

  この式では、S を文（例えば、代入文）としたときに、S の出口で生きているデータフローの定義は、S 内で定義されたフローか、または S に入ってきた定義のうち S 内で殺されない定義となる。データフローをメソッド内について計算し、どの変数が生きているかという生存情報を利用して、定義している文と利用している文の間に、$data\,dependence$ 辺を張る。

- メソッド呼び出し

  メソッド呼び出しを解析することで、依存グラフの呼び出し辺を作成する情報となる。ただし、多相型によるメソッド呼び出しの場合には、静的な解析によって完全に解析することができない。データフローを解析し、変数に代入されているオブジェクトの型を決定したり、インスタンス化されていないクラスを呼び出しの対象からはずす Rapid Type Analysis[8] 等の手法を用いて近似する必要がある。

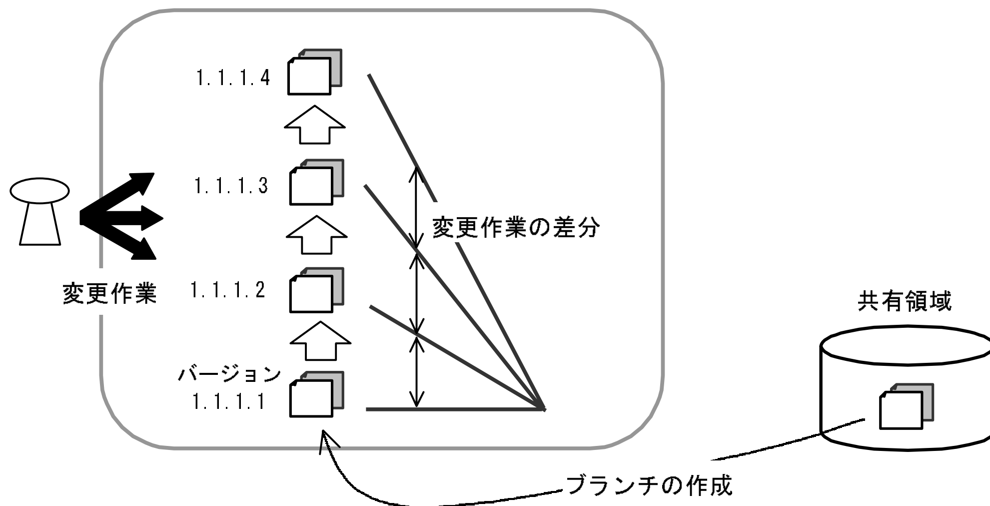


図 3.4: 差分の抽出

- フィールド参照

  プログラム中でフィールド参照を行っている部分を見つけ、参照しているフィールドが宣言されいるクラスまたはインタフェースを決定する。Java 言語の場合、インタフェースで宣言されているフィールドは、公開かつ静的な宣言である。また、クラスのメンバフィールドへの参照も、`obj.foo().var` のようにな場合にでも、戻り値型となっているクラスのメンバを参照するので、静的に決定できる。

- クラス階層情報

  システム依存グラフではクラスの $class\,inheritance$ 辺よって、クラス間の継承関係が理解できる。しかし、グラフ作成時に予めクラスで継承される有効なクラスメンバを求める計算を行う。なぜなら、Java 言語仕様に従って、クラスにおいて有効なメソッドやフィールドを逐一計算することを無くすためである。

3.4 システム依存グラフ間の差分解析

図 3.4 は、特定の作業者のシステム依存グラフの差分を抽出することによって開発状況を把握する方法を示している。最初に、開発者は共有領域にブランチを作成し、開発作業を開始する。次に、変更作業を行い新しいバージョンとしてソースプログラムを登録する。バージョンが変化しているものが、ソースプログラムから作成されたシステム依存グ

ラフを示している。図 3.4 は繰り返された変更作業を表しており、グラフの差分を抽出することにより、クラス、メソッド、文といったレベルで作業した範囲とその内容を掴み取ることが可能となる。

このような解析は、共有されている元のソースプログラムに対して変化しつつある状況を察知するために有用である。一般に、最終的に共同開発者の成果物とマージするまでは、整合性があるか否かを確認することが困難であるため、調整のコストが高くなる。例えば、クラスにあるメソッドのシグネチャが変えられているときには、以前の呼び出し方ではメソッドが見つからず、場合によっては、そのクラスの利用者は、知らずに、そのクラスのスーパークラスを呼び出している可能性もある。また、実際の開発現場では、開発の途中で要求や仕様を修正することが求められ、そのような状況下では、十分な資料を残したり、連絡が行き届かないこともある。従って、作業者が関係する変更作業を行っていることを認識することは、有用である。

差分解析の手順

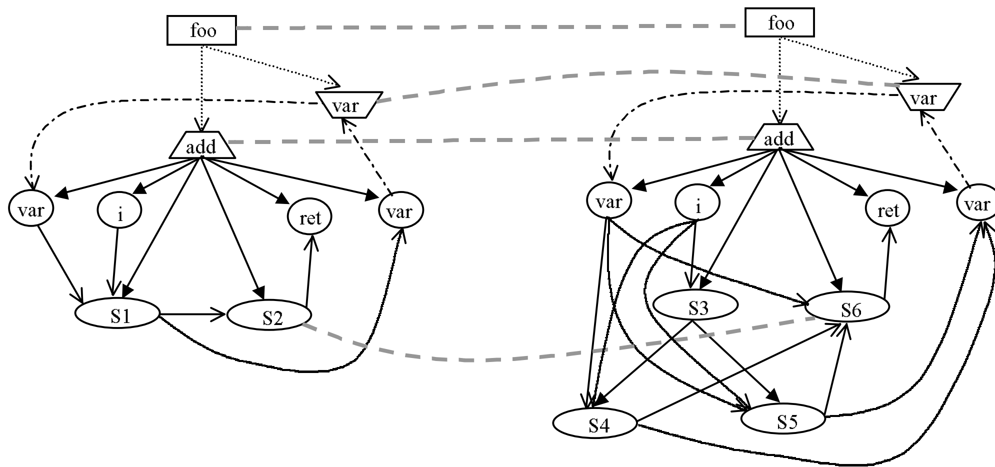
図 3.5 に示す例を用いて、システム依存グラフ間の差分抽出の手順を説明する。テキストベースの差分抽出方法は、テキスト行指向のパーティショニングによって、テキストの共通部分と非共通部分（差）に分割することを行う。一方、グラフの差分抽出は、双方のバージョンに共通な節を求める方法で実現する。その実現法として、以下の 3 つのステップの手順から処理を構成する。

1. クラスの名前

   はじめに、クラスはクラス名を比較して、名前が同じ場合に対応づけをする。図 3.5 は、クラス`foo`の差分解析の結果を表し、クラス`foo`の間を結び付ける辺が、対応のとれている箇所を示している。この例では、双方のクラス名が同じであることから、同一のクラスが存続しているとされ、対応づけられる。

2. メンバ変数の名前、メソッドのシグネチャ

   2 つの目として、クラスを構成するメンバ変数とメソッドの対応づけを行う。メンバ変数は、前述のクラスのときと同様に、単純に名前のみの比較を行う。結果として、図 3.5 におけるメンバ変数`var`の例のように対応づけられる。次に、メソッドの場合では、図のように変更前後の関係を対応づけるために、変更された内容を調べ、同じメソッドなのか否かを判定する必要がある。例えば、メソッドの名前を変えられた場合では、他の変更も加えられている可能性を考慮して、2 つのメソッド



変更前

```
    class foo {
      protected int var;
      public int add( int i ){
S1:     var = var + i;
S2:     return var;
      }
    }
```

変更後

```
    class foo {
      protected int var;
      public int add( int i ){
S3:     if( i >= 0 ){
S4:       var = var + i;
        }
        else {
S5:       var = var - i;
        }
S6:     return var;
      }
    }
```

図 3.5: システム依存グラフ間の対応

が同じか否か確実な判断は困難である。メソッドのときには、シグネチャが同じ場合に対応があるとする。

3. メソッド内の文

メソッド内の文の差分抽出について述べる。クラス、クラスメンバなどは、名前の一致で挿入と削除を判定している。メソッド内部の文の場合、$method\,entry$ を根とし、そこから木の葉に向かって同じ頂点かどうか調べて行く方法が考えられる。図 3.5 の

例を利用して説明する。実際の変更作業は、プログラムに変更前後で、条件分岐文 $S3$ の中へ代入文 $S4$ が含まれるように変更されている。根からの一致では、最初に文 $S1$ と文 $S3$ を比較する。$S3$ は`if`文なので $S1$ と一致せず、それ以降にも一致する文もない。よって、挿入された文であると判断され、さらに $S3$ から *control dependence* 辺が張ってある節も同様に追加された文とみなされる。文 $S2$ と文 $S6$ は同じ内容なので、一致する。

文の対応づけに関して、ある文のまとまりを条件分岐文で囲った場合などは、削除と条件分岐ブロックの追加となってしまい誤差が大きいとも言える。例えば、2 つの木の間の距離を計算する SSPM(Strongly Structured Preserving Mapping) を用いて、木の節の最大共通部分を求めるアルゴリズムを利用するなどの方法が望ましい。このように求められた対応する節の間に、図 3.5 の灰色の点線で示される対応関係を作成する。以上の処理ステップを踏むことにより、対応づけの無い節は、次のような意味をもつことになる。

- 変更前では、作業により削除されたグラフの節を表す。
- 変更後では、作業により挿入されたグラフの節を表す。

そして、以上の作業から次の構成要素を差分として、抽出することが可能となる。

- 節（クラス、メソッド、メンバ変数、文）

  節の差分からは、ソースプログラムの差分と同様に追加、削除といった直接的な作業内容を知ることができる。また、開発者はプログラムのどの部分を変更したのか、という位置的な情報を調べることができる。その作業を行った開発者が、権限や役割を違反した変更をしていないのかなどの確認が可能となる。

- 辺（メソッド呼び出し、メンバ変数の参照）

  辺の差分は、メソッド呼び出しとメンバ変数の参照に着目する。差分は、挿入または削除された *statement* 節から見つけることができる。この辺の変化を知ることで、クラス間の結び付きや、メンバ変数へ影響を与える作業の変化を知ることができる。

ここでは、本研究の差分抽出法の特徴を関連研究において紹介した差分抽出の技法と対比することで説明する。差分抽出の技法として、プログラムテキストに基づく方法と構文木レベルでの意味的な解析を行う方法の 2 種類を挙げた。それらは、テキストデータとして存在するソースプログラムへの作業内容を理解するために利用されていた。また、構文木自体の差分としても、少々粗粒度になるがシステム依存グラフにおける節の差分に相当する。

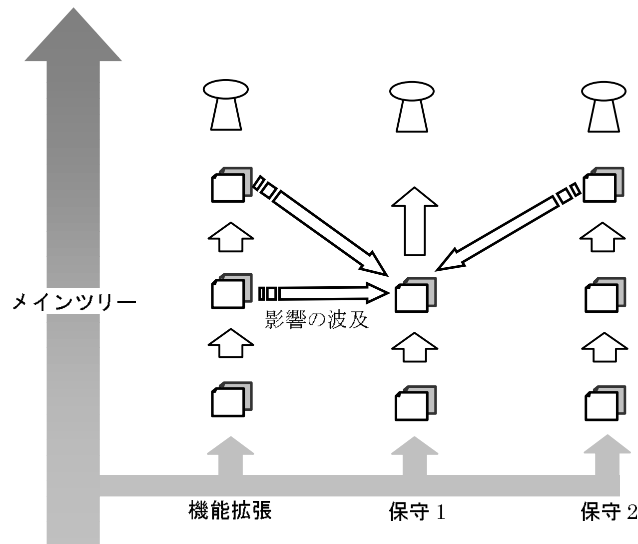


図 3.6: 変更作業による影響の波及

さらに、本研究は開発の状況把握という視点から、開発者の注目している部分を知ることができれば望ましい。そこで、すでにプログラムの依存関係を解析しているので、辺に基づく差分の抽出が行える。辺の差分抽出により、依存関係の変化が捉えることが可能となる。差分抽出として有用な関係は、$call$、$member\,refrence$ 辺というプリミティブなものから、その情報を利用して、クラス間の結び付きの変化としても捉える事ができる。また、その結び付きとして、利用、継承等の関連に分類することができる。

3.5 ブランチ間の波及解析

図 3.6 は、本流の開発（メインツリー）から分岐（ブランチを作成）した、複数のブランチ（機能拡張、保守1、保守2）をもつ開発を示している。各ブランチにおいて、縦方向に変化しているものは、図 3.4 と同様にソースプログラムを解析して生成されたシステム依存グラフである。図 3.6 中の保守1 に焦点をあてた場合、機能拡張、保守2 のブランチから影響受ける可能性を示している。つまり、保守1 以外のブランチでは、一貫性のある変更作業であっても、保守1 の立場にとって、その変更作業は影響を及ぼす可能性のある矛盾した作業とみなせることになる。

ブランチ間の影響は個々に作業者やグループが異なるために、十分に連絡が取られていない可能性がある。最終的に双方のブランチで開発されたソースプログラムをバージョン

管理システムのメインツリーへ順にマージしていったときに、他のブランチのマージが影響を及ぼし、双方の開発作業について後戻りして修正を行う必要が出てくる恐れがある。これは、円滑な開発に混乱を引き起こし、大きな不利益となると思われる。

変更による影響が波及する範囲を特定するためには、変更箇所が及ぼす広範囲に渡る影響の波及を全て計算することを行わずに、自分の開発しているプログラムが他人の影響を受けるような依存関係を持つ部分を探し出すことで、影響の波及を解析する。以下に、抽出対象の影響を波及させる依存関係を挙げた。

- メソッド呼び出し
  利用または継承関連にあるクラスのメソッドとの間に張られている $call$ 辺。

- メンバ変数の参照
  この場合のメンバ変数の参照はクラス内部のメンバ変数と異なり、宣言されている外部のメンバ変数の参照とする。

- 継承関連
  全てのクラス内部の $class\,inheritance$ を抽出する。

3.6 差分解析と波及解析に基づく競合の検知

本節では、3.4 節の差分抽出と 3.5 節の波及解析からの処理結果を利用して、競合を検知する方法について検討する。

3.6.1 競合の検知

先に、競合をあるブランチ上のバージョンに対して、他のブランチのシステム依存グラフからの波及の全体として定義した。図 3.7 を例にすると、開発者A と開発者C のシステム依存グラフは、変更が加えられている。そして、開発者B のプログラムへ波及効果 1,2,3 の影響を及ぼす結果となり、その全体が競合となる。従って、競合の検知は、波及効果の全体を探し出すこととなる。例えば、検知された情報にから、相手の開発者が実際に行った作業内容と予定されている作業内容との食い違いを見つけることが可能となる。例えば、あるクラスで宣言されているメンバ変数の名前を変えられたときには、それを参照しているプログラムの全ての部分を修正する必要がある。

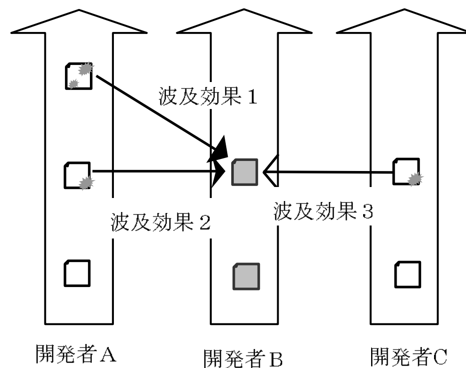


図 3.7: 競合の検知

競合解析の複雑さの問題　競合の検知を行うシステムを実現する場合に、解析の複雑さが問題になると考えられる。図 3.7 は、開発者B のある特定のシステム依存グラフに波及する開発者A、開発者C の変更作業ある。従って、同じ開発者B のシステム依存グラフにおいても、異なるバージョンごとに競合検知の処理を行う必要がある。この複雑さを軽減する方法として、前のバージョンから変更されていない箇所の解析結果を引き継ぐなどして、以前の解析結果を再利用する工夫をする必要がある。

3.6.2　競合解析の範囲

メソッド・メンバ変数へのアクセス

クラス・インタフェースの宣言を変更された場合は、影響を受ける可能性がある。

- 挿入または削除

  クラス、メソッド、メンバ変数を参照していた場合に、挿入または削除により、影響を受ける可能性が考えられる。メソッド呼び出しやメンバ変数の参照により他のクラスに依存関係のあるプログラムを開発していた場合に、依存関係にあるクラス宣言が変更されるならば、そのクラスのメソッドやメンバ変数を使用できない可能性がある。例えば、メソッドのシグネチャを変えられたようなときが考えられる。

  また、メソッドやメンバ変数は、宣言されたクラスによっても影響を及ぼすと考えられる。これは、実際にプログラムは動作するが、他の開発者の何らかの意図によって、変えられてしまっているため、保守的な見方として、この場合も影響の可能性があるものとする。例えば、メソッド呼び出し先のクラス中で、スーパクラスのメ

ソッドを上書きするように修正されていたとする。この場合、実際に呼び出すメソッドが異なり、従来の働きと違ってくると考えられる。このような変更が頻繁に起きるようならば、注意しなければならないなどの推測が立てられる。

- クラスメンバの型

  メソッドの戻り値の型やメンバ変数の型が修正されていたときには、影響を受ける可能性が考えられる。例えば、プリミティブ型のメンバ変数を参照型へ修正を考えてみる。その変数を参照しているプログラム文は、型が異なるので、そのメンバ変数への整合性が保てない。修正の検出は、クラスメンバの差分解析が名前（やシグネチャ）の一致により判定していることから、変更前後において対応するが、型が異なるためわかる。修正によりクラスメンバを利用している部分との整合性を失う可能性がある。

- スコープの変更

  クラス、メソッド、メンバ変数のスコープを修正された場合に、影響があると考えられる。例えば、スコープが`public`から`protected`に変更されたときなどがアクセス不可能な場合として挙げられる。

継承・実装関連の変更

クラスの継承またはインタフェースの実装関連が変更された場合を言う。利用関連または実際に継承してるオブジェクトの継承・実装関連が変更されたら、例えば、型の互換性を失い、呼び出されるメソッドは、意図するものと異なる恐れがある。この関連の変更を検出する方法は、クラス階層について解析していた情報を利用する。

戻り値に影響を与える変更

メソッドの戻り値に影響を与えるような変更が行われた場合、戻り値（例えば、オブジェクトの参照など）は、利用者の期待と異なっているかもしれない。また、そのメソッドに影響が波及するような変更が加えられる予定との確認が可能となる。

解析の方法としては、返り値の型が`void`型を除いて、$return$パラメータからデータ依存関係にある文を全て取り出し、その中から挿入または削除された文を探し出せばよい。図3.8は、図3.5のプログラム例を使って戻り値に影響する変更を表してある。以下に図の例を用いて解析手順を示す。

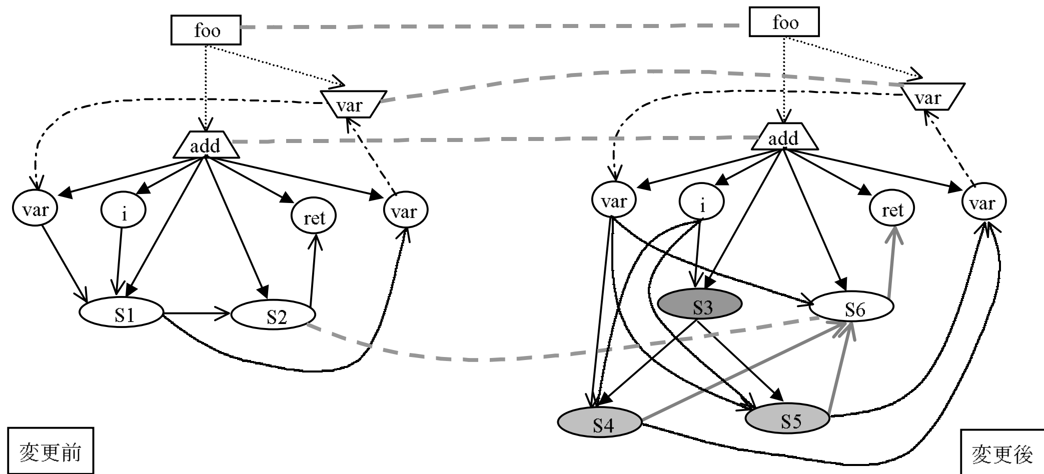


図 3.8: 戻り値に影響する変更例

1. 図 3.8 の変更後のグラフについて、$return$ パラメータから $data\,dependence$ 辺を辿っていき、データ依存関係にある文を選びだす（$S6, S4, S5$）。
2. その中で変更前のグラフと対応関係のない文をマークする（$S4, S5$）。
3. 次に、それらマークされた文へ $control\,dependence$ 辺のある文を探す（$S3$）。なければ、終了。
4. 変更前のグラフと対応が無いとき、その文もマークする（$S3$）。前のステップに戻る。

以上の手順で**挿入**操作による影響が分かる。また、**削除**操作の場合は上記の手続きを変更前のグラフに、変更前から変更後のグラフへの対応関係について適用すれば求められる。ちなみに、図 3.8 では、$S1$ に相当する。

第 4 章

プロトタイプシステムの設計

本章では、プロトタイプシステムにおける解析法の範囲と競合の理解支援を設計方針として述べ、ユーザ・インタフェースと解析システムの設計について示す。第 3 章において述べた解析方法を利用し、プロトタイプシステムとして、その一部分を実現することを目指す。それに伴い、変化しつつある状況を支援する手段についても検討する。

4.1 設計方針

プロトタイプシステムにおいて適用する競合検知の範囲は、クラス宣言の変更を対象とする。競合の検知（3.6 節）により、個々のプログラム間の影響が分かる。プロトタイプシステムでは、特定のブランチ上での変化しつつある状況の把握を変更履歴の追跡を可能にするユーザ・インタフェースを実現することで支援する。さらに、ソースプログラム中にはいくつもの変更が加えられて、波及する影響が数は多くなる場合を考慮する必要がある。さらに、複数バージョンと比較することも考慮して、プロトタイプシステムの利用者が、複雑な解析結果を容易に理解するための支援法を検討する。

4.2 ユーザ・インタフェースの設計

利用者に対して直接的なサービスの窓口を提供するユーザ・インタフェースの設計を示す。はじめに、ユーザ・インタフェースの設計方針として全体構成を示し、そして個々の機能について図 4.1 を用いながら解説する。

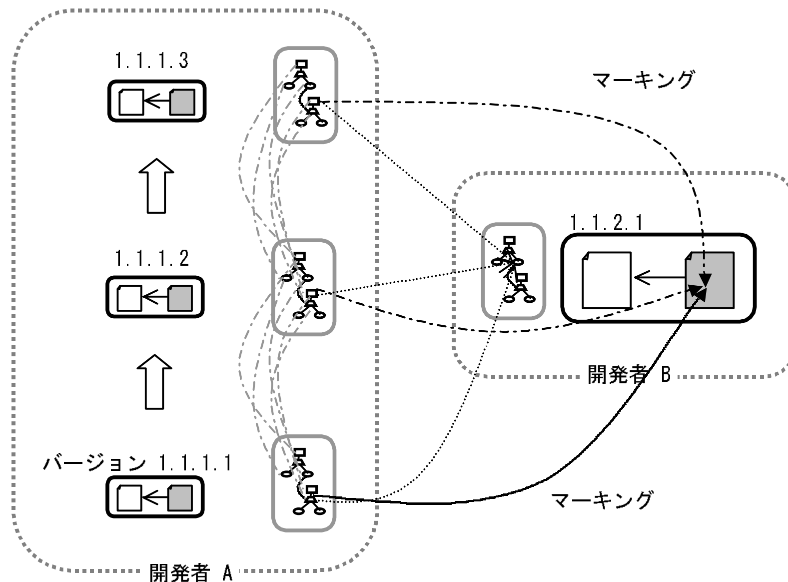


図 4.1: ソースプログラムへのマーキング

4.2.1　全体構成

システムの利用者には、数多くある競合の種類を簡単に見分けられ、直感的に把握できる仕組みが必要である。直感的な把握を促すために利用者とのインタフェースは、ソースプログラム上に影響があることを直接的に示すように設計する。

ユーザ・インタフェースは、ソースプログラム上で影響を受ける部分にマークすること（マーキング機能）で、利用者へ明示的に通知する。また、変更された側と影響を受ける側を対応させて表示させる。これは、利用者が両方のソースプログラムを見比べることで、実際の変更部分と影響部分の対応づけを把握できるからである。また、利用者はマークにイベントを送ることで、要求に応じて付加的な情報を手に入れることができるようにする（詳細情報機能）。

4.2.2　マーキング機能

利用者は、波及してくる影響の数と種類を容易に知ることができるように、ソースプログラムへのマーキングを利用する。図 4.1 はソースプログラムへのマーキングを示した図である。図 4.1 で説明すると、開発者A のブランチにある 3 つのバージョン（1.1.1.1, 1.1.1.2, 1.1.1.3）のシステム依存グラフから、開発者B のソースプログラムへ伸ている矢印がマーキングを示している。このマークにより利用者へ競合を通知する。マーキ

ングによりソースプログラムが反映している開発の状況を把握できるものと考えられる。また、マーキングされた理由とマーキン関係上に表示できると把握しやすいものと考えられる。

4.2.3 詳細情報機能

詳細情報機能は、プログラムがマークされた理由を示す情報を提供する。この機能は、マーキングの役割を補うことを目的としている。単にマークされたという事実だけでは、正確に状況を把握することが難しいと思われるので、波及してくる影響の詳細について情報を提供する。

図 4.1 では、マーキングを表す矢印は、2 種類ある。これは、マーキングされる部分は同じ所としても、マークされる原因が違っていることを表している。例えば、次のような場合が考えられる。開発者B のプログラムと開発者A のプログラムの間に、メソッド呼び出しの依存関係があるとする。開発者A が作業するバージョン`1.1.1.1` では、そのメソッドの戻り値の型が変えられており、バージョン`1.1.1.2`, `1.1.1.3` においては、シグネチャが変えられているため削除されたとみなされている。ソースプログラムの同じ箇所にマーキングされるが、各バージョンで起きた影響は異なっている。このように、単にマーキングだけでは得られない状況が発生している場合も考えられる。解析により得られる情報から、詳細な情報として以下に 3 項目を洗い出した。

- 競合箇所
  影響を与える変更作業が加えられたプログラムの部分を示す。図 4.1 のように、実際には複数のバージョンの解析では、一つのマークに対して複数の競合箇所が存在するときもある。

- バージョン名
  競合箇所のあるバージョンの名前を指す。一つのバージョン名には、いくつかの競合箇所を含む場合があり、このときには、マークも複数付けられている。

- 競合理由
  影響が波及すると判断された理由である。

図 4.1 のマーキングを表す矢印の種類から、種類の異なる競合が発生している図であることが分かる。バージョンの変化を追うと、バージョン`1.1.1.2` で新たな変化していることが調べられる。このように、バージョン間での競合の変化を見つけやすくする必要もあると考えられる。

4.3 解析システムの設計

本節では、先に述べた競合解析の範囲から要求される事項から、解析システムとしての機能を設計する。クラスを検査するには、少なくとも粗粒度の依存グラフを生成する機能やシグネチャやスコープ等の整合性の検査を実施する機能も必要とする。また、複数バージョンのグラフを管理し、他の解析モジュールからグラフへの適切なアクセスを提供する機能も実現する必要がある。

グラフ生成

クラス、メソッド、メンバ変数といったグラフの構成要素の生成を行う。また、部分的に波及解析を実現するために、メソッド呼び出しやメンバ変数の参照をしている文を見つけ、その依存関係に関する情報を保持しておく機能も必要とする。

クラス階層

システム依存グラフではクラスの $class\,inheritance$ 辺によって、クラス間の継承関係が理解できる。実際は、単にクラス・インタフェース間の関連のみならず、その関連に基づき多重継承されるメソッドや隠蔽されるフィールドを言語仕様に従って、探索しなければならない。そこで、あるクラスにおいて有効なメソッドやフィールドを探索毎に計算せずに、予めクラス毎に継承される有効なクラスメンバを求める。

複数バージョンの管理の役割

波及解析や競合解析を行うためには、複数バージョンのシステム依存グラフを管理し、必要な情報へのアクセスを提供する必要がある。例えば、バージョン 1.0 のクラス `foo` へのアクセスを提供するために、その参照を返すといったアクセスライブラリの役割をする。現段階では、クラス階層ライブラリに管理機能を付け加えて、プロトタイプシステムを実装している。

宣言の検査

クラスメンバの整合性を調べる場合に、戻り値の型、修飾子やシグネチャの検査が必要となる。戻り値の型や修飾子は、異なるバージョン間の検査に利用できる。

あるクラスにメソッドがあるかないかは、メソッドのシグネチャによって判断することができ、そのためのシグネチャの整合性を検査することもできる。また、バージョンが異なると継承しているメソッドが、上書きされるように変更されるときには、シグネチャが等しさだけでは、判断できない。そこで、メソッドが実際に宣言されたクラス名も同一かどうか調べ、同じメソッドが有効か否か検査する。

検査手順

以下、検査の順番を示す。

1. 宣言された場所
   異なるメソッドやメンバ変数を利用しているのならば、影響が波及する可能性があるとみなす。クラスメンバの修飾子の検査では違いが分からないので、はじめに検査を行う。

2. スコープ
   公開されて（`public`）アクセス可能だったメンバが、`private` または`protected` に変更されてアクセスを許可されないような場合を確かめる。

3. 型
   期待される型とは異なるので、プログラムが動作しないなど考えられる。

第 5 章

プロトタイプシステムの実装

本章では、前述した設計を基に実装したプロトタイプのシステムの構成、およびその機能について述べる。

5.1　システムの利用法

プロトタイプシステムの利用法の概略を述べる（図 5.1）。はじめに、利用者はシステムの入力情報となるソースプログラムを準備する。このソースプログラムには 2 つの分類があり、他のブランチで変更作業が行われた複数のバージョンのものと、利用者のブランチそのものである。そして、出力情報を利用者は、解析された結果を利用者のソースプログラム上にマーキングされることで、現在の開発状況を覗き見ることが可能となる。

5.2　システムの構成

図 5.2 はプロトタイプシステムの構成図を示している。プロトタイプシステムは、CASE ツールの開発を支援する Japid [11] を利用して実装した。以下、本章では Japid を利用して実装された部分のシステムをプロトタイプシステムと呼ぶ。Japid については、以下に述べる説明にとどめる。

Japid

Japid は J-model という Java の構文を 15 個のクラスと 51 個の関連で表現した実体関連モデルに基づいてソースプログラムを解析し、細粒度でソフトウエアを扱うことができる。図 5.2 に示したように Japid のシステム構成は、解析器、ソフトウエアデータベース（以下、SDB）およびアクセスライブラリからなる。

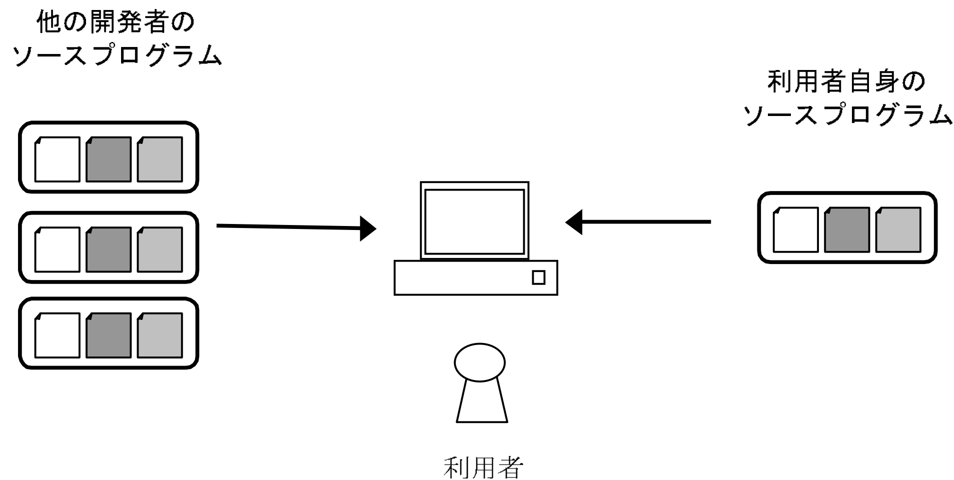


図 5.1: システムの利用

ソースプログラムは解析器により、解析情報は J-model として SDB へ保存される。SDB は一種のオブジェクト指向データベースと見なすことができる。アクセスライブラリは、SDB 中のオブジェクトにアクセスするための Java 言語で記述されたクラスライブラリである。プロトタイプシステムはこのクラスライブラリを利用している。

プロトタイプシステムは、3 つのサブシステムから構成されている。

1. クラス解析サブシステム
   部分的なシステム依存グラフおよびクラス階層情報を作成する。

2. 波及解析サブシステム
   入力として与えられた利用者と他の開発者のソースプログラム間に依存関係が成り立つことを見つけ、その依存関係を抽出する。

3. 競合検知サブシステム
   抽出された依存関係を用いて競合を探し出し、マーキング処理を実施する。

プロトタイプシステムは、処理結果の出力を HTML 形式で生成する。これにより、利用者は Web ブラウザ等のアプリケーションから結果を容易に閲覧することが可能となり、ソースプログラム間の競合を容易に確認することができる。

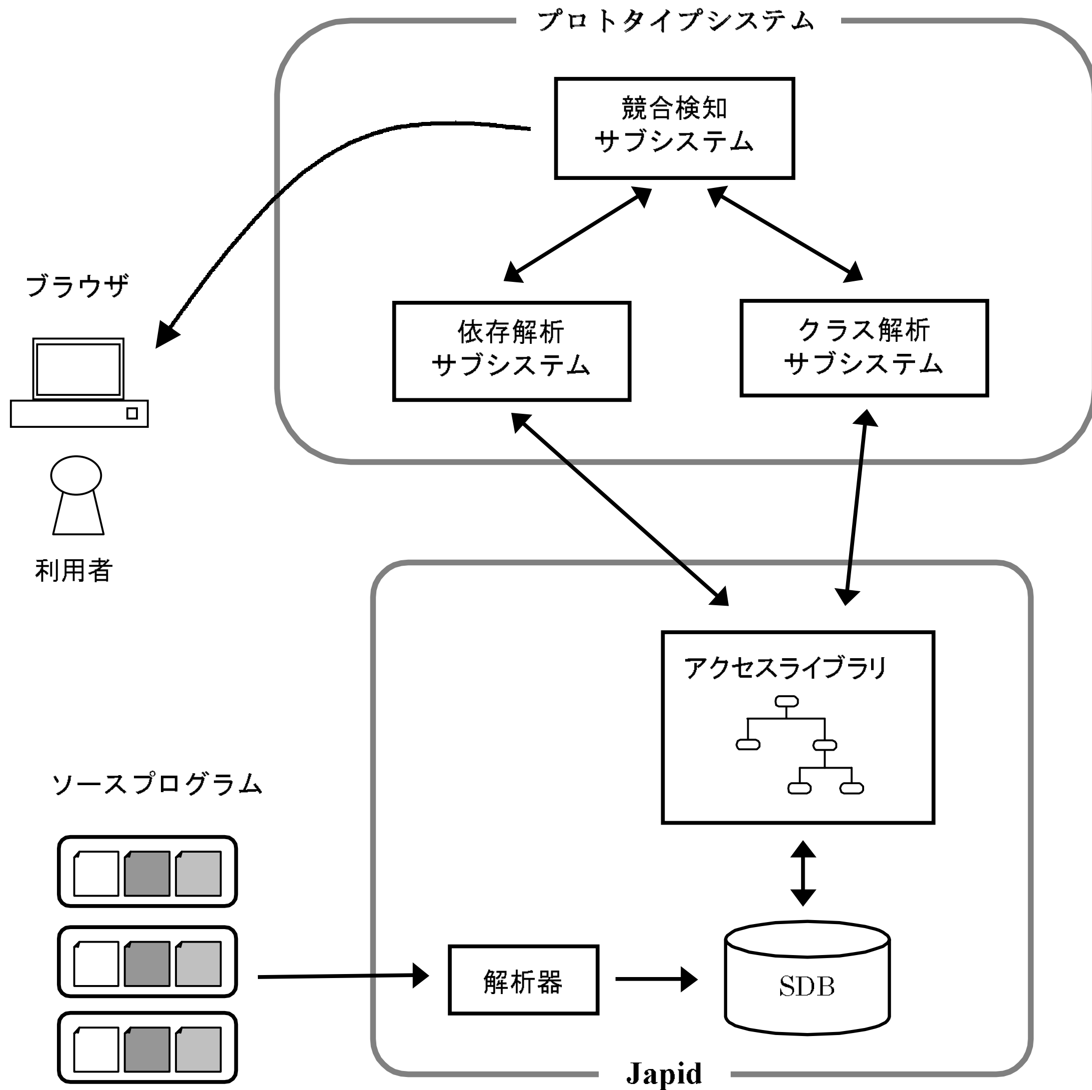


図 5.2: システムの構成図

5.3 ソースプログラムの解析

5.3.1 データの準備

ソースプログラムの解析工程で、システムの利用者は解析の対象となる複数バージョンのソースプログラムをチェックアウトして、Japid の SDB を作成する必要がある。この部

分の工程は、将来的には、CVS 等のバージョン管理システムから、自動的にチェックアウトしてくるように工夫することが望まれる。

解析システム実装の説明は下記に示したプログラムを例題を用いながら、述べていくことにする。最初に、例題の概要を説明する。ソースプログラムから理解できることは、ReturnMessage とVersesPlayer 間は、ReturnMessage のdecode メソッド内から、VersesPlayer のaccept を呼び出すという依存関係が成り立っていることがわかる。このプログラム断片は、ゲームプログラムのプレイヤークラスとプレイヤー間の通信をとりもつメッセージパッシング機構のメッセージのクラスである。

また、これらのソースプログラムは、開発者A とB が別々の変更スレッドで作業した結果である。VersesPlayer は、開発者 A のブランチで開発を進めており、バージョン1.0 と2.0 の間で、accept メソッドの戻り値の型がboolean からint へ変更されている。開発者 B は、ReturnMessage を開発している。このままでは、メソッド呼び出しの整合性を保てなくなる。

```
public class VersesPlayer  // version 1.0
  public boolean accept( String word ){
      //省略
  }
}
public class VersesPlayer{  // version 2.0
  public int accept( String word ){
      //省略
  }
}

public class ReturnMessage extends VersesMessage { // version 1.0
  public boolean decode() {
    if( !player.accept( word ) ){
      //省略
    }
  }
}
```

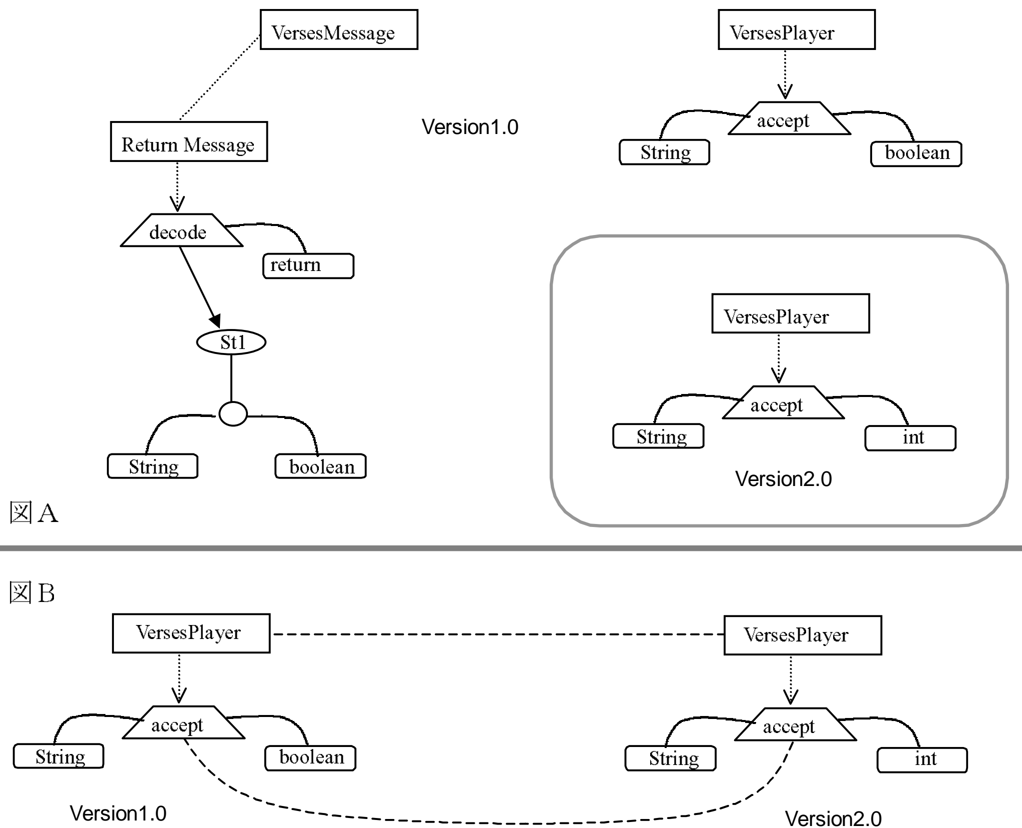


図 5.3: クラス解析と対応関係

5.3.2 クラス解析サブシステム

クラス解析サブシステムは、Japid のアクセスライブラリが複数のバージョンに対するアクセスを提供していないことを補う。また、部分的にシステム依存グラフを生成を実現し、さらにクラス階層解析を実施することで、競合検知サブシステムに情報を提供することも行う。プロトタイプで必要とする情報は、クラスやインタフェースのメソッドおよびフィールドに関する情報である。

図 5.3 のA は、このサブシステムによって解析された例題プログラムの部分的なシステム依存グラフを示している。図の例に従って順に解析手順を説明して行く。

1. クラス解析

   全てのクラスについて、クラスの持つメソッド・フィールドおよび、それらのパラメータ、戻り値や型などの情報を持つ、グラフを生成する。

2. 継承・実装関連の解析

システムの入力となったクラスの継承・実装関連を抽出し、前のステップで作成したグラフ同士の間に継承・実装関連を意味する辺を作成する。

3. 対応関係

   図 5.3 の B は、クラス・メソッドレベルでの対応関係の作成を表している。バージョン 1.0 と 2.0 とでは、accept メソッドの戻り値の型が異なっている。しかし、クラスはクラス名、メソッドはシグネチャを検査して、同じか否か見分けている。

5.3.3 波及解析サブシステム

波及解析サブシステムは、利用者のプログラム中に含まれる依存関係を抽出し、競合検知サブシステムに情報を提供する部分である。依存関係として、メソッド呼び出し、フィールド参照に注目した。例えば、図 5.3 の A の ReturnMessage と VersesPlayer の間にある呼び出し関係を抽出する処理を行う。

5.3.4 競合検知サブシステム

競合検知サブシステムは、クラス解析の結果と依存解析の結果を用いて異なるバージョンのソースプログラムとインタフェースが一致しない場合に、ソースプログラムに HTML のハイパーリンクとしてマークをつける。図 5.4 の例では、抽出されたブランチ間の依存関係を検査して、その依存関係から影響が伝播するかどうか確かめる。

5.4 システムの機能

競合関係の表示

図 5.5 に示すように、ソースコードに付けられたハイパーリンクから、競合のあること理解できる。マークの部分をクリックすると、次に説明する詳細情報がブラウザに読み込まれる仕組みになっている。

詳細情報

図 5.6 は、詳細情報の画面である。マークには、フィールド参照とメソッド呼び出しの場合があり、それぞれに対応した情報が表示される。以下に表示される内容を列挙する。

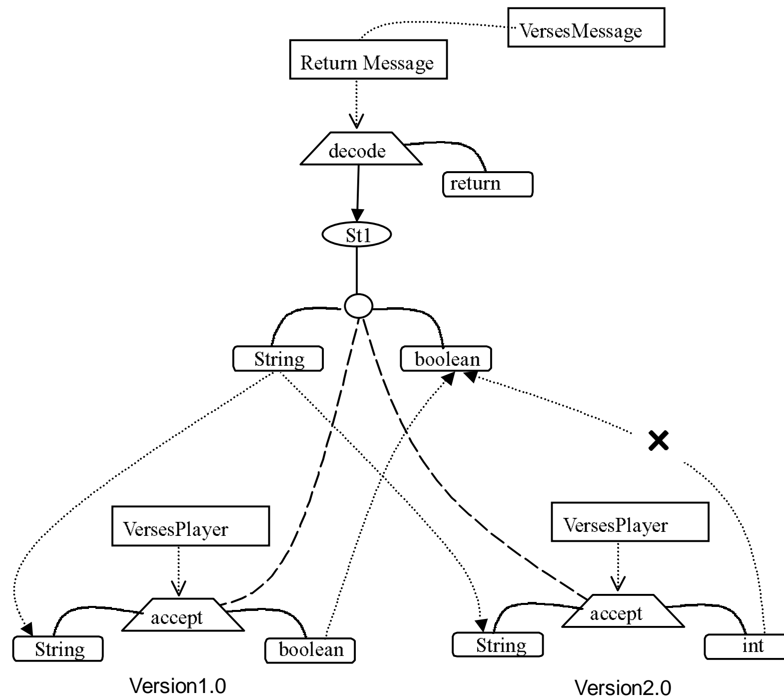


図 5.4: 競合検知の例

- 競合のあるフィールドまたはメソッド名。また、この名前の部分は前述のマーキングと同様にハイパーリンク化されており、クリックした時点で、対応するバージョンのソースプログラムが読み込まれる。
- クラス名またはインタフェース名
- バージョン名
- 競合の理由

さらに、利用者は同じ詳細情報の画面には、異なるバージョンでの解析結果を表示されるので、競合の発生したバージョンを調査することが可能となる。

全体画面

図 5.7 は全体画面を示している。解析結果は、HTML のフレーム機能を利用して、左のフレームに変更を加えられたソースプログラムを配置し、右のフレームに依存関係のあるソースプログラムを配置することで、双方を見比べることができる。また、左下のフレームは詳細情報を表示する部分である。

```
String word = (String)entry.elementAt(0);
try {
if (!player.accept(word)) {
if (player.isWrong(word)) {
player.conceded();
```

図 5.5: マーキング

verses.VersesPlayer
- Version 2.0で、メソッド accept の戻り値の型が一致しません

図 5.6: 詳細情報

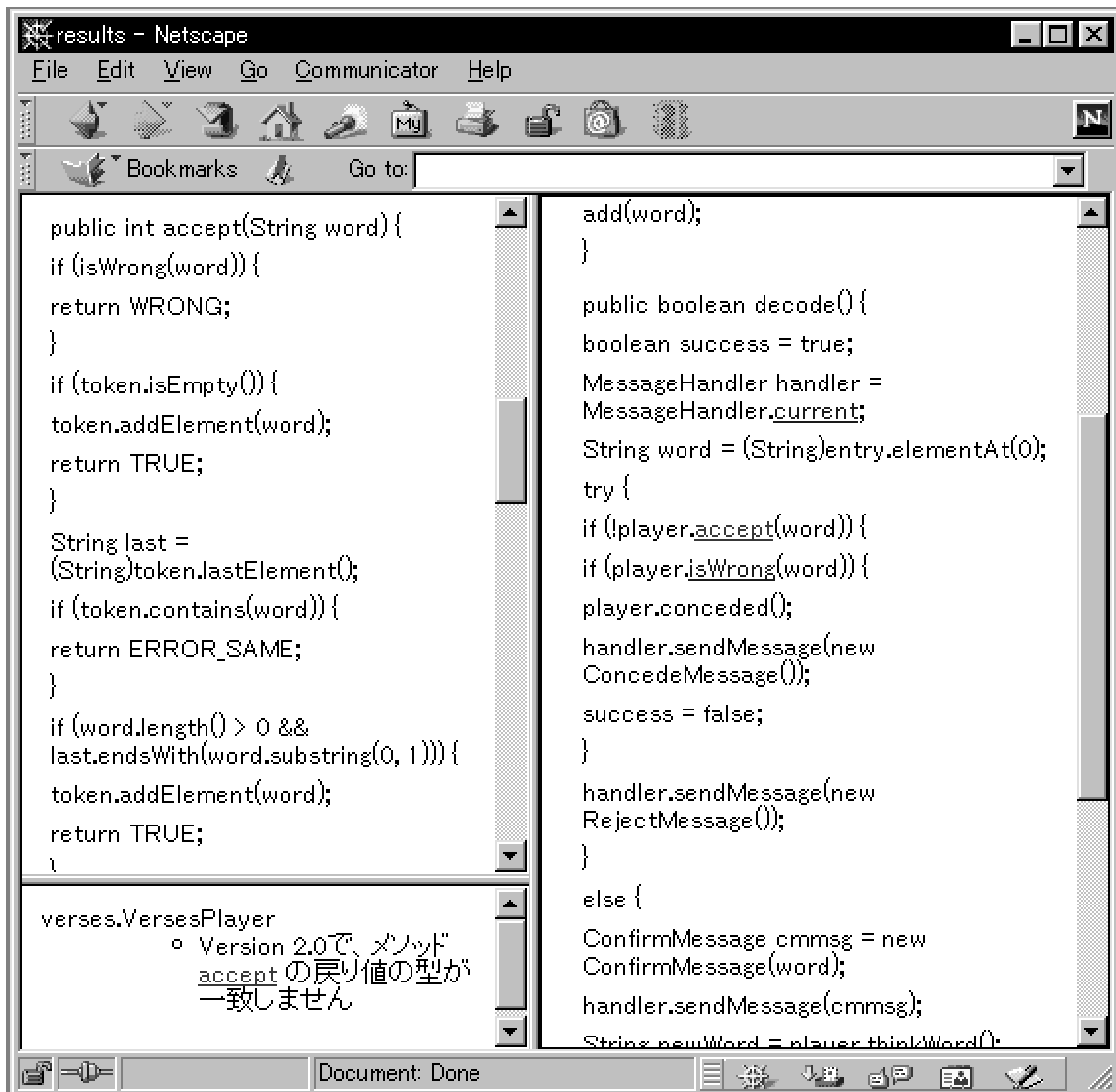


図 5.7: 全体画面

第 6 章

議論

第 3 章で差分解析・波及解析を用いて、開発状況の把握のための競合検知について述べた。本章では、本研究の解析法の一つの応用として、未来版管理システムの汚染マーキングの細粒度化について議論する。

6.1 未来版管理システムへの応用

未来版管理システム [10][9] は、ソフトウエア分散開発においては、将来の変更に備えて可能な部分の作業を前もって進めておくといった作業が必要とされ、その作業プロセスを支援する版管理システムである。未来版管理システムには、共有領域と個人の作業領域に属する中間成果物の状態間の矛盾を検出することを目的とした汚染マーク機構がある。本節においては、汚染マーク機構に着目し、本研究の解析方法の適用について述べる。

汚染マーク機構とは、依存関係のある中間成果物の状態間に発生する矛盾の発生可能性を検出することで、他の開発者の作業の影響を通知することを目的とした機構である。汚染マークは、変更要求の承認を受けることなくチェックアウトして作業を開始した中間成果物に対してつけられる。変更要求の承認を受けて、この汚染マークを解除しなければ、共有領域へチェックインすることはできない。汚染マーク機構の管理対象は、ソースプログラム以外にも仕様書なども対象としており、本研究において提案した方法は、このうちのソースプログラムのマーキングに応用することが可能である。また、現在の機構の管理は、ファイル単位の粗粒度であり、細粒度で汚染マーキングをすることが必要である。

汚染マーク機構の管理する write 競合と read 競合を対象として、競合の細粒度化を実現する方法について述べる。

- write 競合

write 競合とは、共有領域でチェックイン対象のファイルが更新されていた場合に発生する。この競合は、現在の汚染マーク機構においても自動的な検知ができる。差分解析の結果を利用して、例えば、クラスやメソッド等の変更箇所の情報を提供することにより、競合の解消を支援することができると考えられる。

- read 競合 read 競合とは、依存関係にあるプログラムが共有領域にチェックインされたときに、依存関係による影響が波及する可能性を汚染マークを伝播させることで、他の開発者と作業結果の影響があることを通知し、状況把握の支援をする。現状の汚染マーク機構では、競合を検知するために、依存関係のあるモジュールを手作業で入力する必要がある。プログラム解析を利用し、入力の作業コストを削減することが可能である。システムにより解析される依存関係は、本研究において述べたメソッド呼び出しなどの依存関係である。

  現在のファイル単位の管理は、プログラムテキストの一部分が変更されたときに、影響を及ぼす可能性があるとして競合を検知している。本研究の検知法により、以下の 2 項目について解消支援を行える。

  - クラス単位
  - 依存関係のあるシステム依存グラフの節
    例えば、メソッド呼び出しを含むようなステートメント。

  クラス単位は、現在の汚染マークのように粗粒度の情報を提供するが、実際には、細粒度の汚染マーキングの集約として取り扱われている。そして必要ならば、他の開発者が作業したクラスのどの部分（メソッド、メンバ変数）に依存していて、自分が作業しているクラスのどのステートメントに影響を波及させるか、状況の把握に役立てることができると考えられる。これにより、変更箇所を正確に素早く探し出すことができ、変更箇所と影響を受ける部分を見比べながら、修正作業を行える。

  また、プログラムの実行に影響を与えないような変更（例えば、コメント文）であったとしても、変更作業が行われたという情報は貴重である。本研究の解析方法では、コメント文等の変更は検出に対応していない。従って、そのような実行に関わりのない部分の変更については、現在の競合の検知方法と合わせて実現することが考えられる。

最後に、細粒度汚染マーク機構を実現するための課題についてまとめる。

- マーキングされる数は膨大になる可能性がある。閾値などを設けて、敏感になりすぎないシステムにする必要もある。

- 単なるマーキングでは、影響が波及した原因に関する情報が失われてしまう恐れがある。プロトタイプシステムにおけるユーザ・インタフェースの設計で述べたように、付加的な情報を利用者に提供することが望まれる。

第 7 章

おわりに

7.1 まとめ

本研究では、Java 言語を対象として、ソースプログラムから変換されてたシステム依存グラフの解析による開発状況の把握方法について提案した。状況把握のためにシステム依存グラフ間での差分抽出、波及解析、競合の検知の方法を定義し、ブランチ上の差分による状況把握法とシステム依存グラフ間の競合検知による状況把握法について述べた。また、競合の検知による状況把握に適し、変化しつつある状況を段階的に捉えることを可能にするユーザ・インタフェースを一部分の解析法を適用し、プロトタイプシステムとして実現した。まとめると、ソースプログラムからの競合検知の基礎的な方法を提案し、その方法を応用する一例として、未来版管理システムの汚染マーク機構における細粒度化ついて議論した。

7.2 今後の課題

解析システムの開発　現在の実装では、解析方法を部分的に実現しているにすぎない。提案した解析を実現するためにも、システム依存グラフの作成、差分解析等のシステムの機能を実現する必要がある。システム依存グラフにおける $polymorphic\, choice$ 節を作成する場合には、参照型の変数が指すインスタンスのクラスを近似し、解析の精度を向上させることも行う必要がある。

ユーザ・インタフェースの改善　マーキングと異なる新しいユーザ・インタフェースの開発が必要である。ソースプログラムを解析した結果は、グラフとして保存されており、こ

れを利用してグラフ構造のまま視覚化することで、競合箇所や影響の波及する様子をさらに容易に理解できると考えられる。グラフを用いた場合には、グラフにする粒度を変化させるなどの機能も考えられ、そのような機能により、段階的な開発状況の把握を支援できるものと考えられる。

評価実験　本研究では、実際の開発の中でシステムを利用して評価実験を行うことができなかった。評価実験を行う環境を整えるためにも、上記の課題の実現が必要とされ、バージョン管理システムからソースプログラムを自動的に取得する機能も必要と考えられる。

謝辞

最後に、研究を進めるにあたり、落水浩一郎教授からは始終変らぬ御指導を頂き、心より御礼申し上げます。論文審査にあたり有益なご意見を賜りました篠田陽一助教授、権藤克彦助教授に深く感謝申し上げます。また、Japid を公開していただいた名古屋大学の阿草清滋教授、愛知県立大学の山本晋一郎助教授に厚く御礼申し上げます。

そして、インテック・システム研究所の堀雅和氏、落水研究室の村越広享助手、藤枝和宏氏、猪俣敦夫氏からは、多大なる助言を頂きました。感謝しております。最後に、落水研究室、篠田研究室の皆様方に深く感謝致します。

参考文献

[1] M. Weiser: Program slicing, IEEE Transaction of Software Engineering, 10(4), pp.352–357, 1984.

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

[3] M. J. Baker and S. G. Eick: Visualizing Software Systems, Proc. 16th Int. Conf. Software Engineering, pp.59–67, 1994.

[4] 荻原 剛志, 會沢 実, 練 林, 鳥居 宏次: 構文木の相互比較による複数バージョン比較分析方法の提案, 日本ソフトウェア科学会, ソフトウェア工学の基礎II, 1995.

[5] L. D. Larsen and M. J. Harrold: Slicing Object-Oriented Software, Proc. 18th Int. Conf. Software Engineering, pp.495–505, 1996.

[6] 吉田 敦, 山本 晋一郎, 阿草 清滋: 意味を考慮した差分抽出ツール, 情報処理学会論文誌, Vol.38, No.6, pp.1163–1171, 1997.

[7] 小林 隆志, 権藤 克彦: 構文木に基づく細粒度ソフトウェアコンフィグレーションマネージメントでの影響解析, 日本ソフトウェア科学会, ソフトウェア工学の基礎IV, pp119–126, 1997.

[8] David Francis Bacon: Fast and Effective Optimization of Statically Typed Object-Oriented Languages, Ph.D. Thesis, Computer Science Divison, University of California, Berkeley, UCB/CSD-98-1017, 1997.

[9] 堀雅和, 篠田陽一, 落水浩一郎: 汚染マーキングによる未来版空間の管理, ソフトウエア・シンポジウム'98, pp161–170, 1998.

[10] 堀雅和: ソフトウエア分散開発に適した中間成果物の管理法に関する研究, 北陸先端科学技術大学院大学 博士論文, 1998.

[11] Yoshinari Hachisu: A CASE Tool Platform for Object-Oriented Program - Japid: Its Design and Implementation, School of Engineering, Nagoya University, Ph.D Dissertation, 1999.