ソフトウェア保守・再利用の支援を目的とした
プログラム解析手法に関する研究

横森 励士

2003 年 8 月

内容梗概

ソフトウェアの品質改善，開発作業における生産性の向上を目的として，さまざまな研究活動が行われている．その中の一手法として，プログラムからその性質や振る舞いを抽出することによって開発者に有益な情報を提供する，プログラム解析と呼ばれる手法がある．プログラム解析によって抽出される性質や振る舞いの例として，データフロー，制御フロー，エイリアス関係，手続き呼び出し関係，クラス階層情報などが挙げられる．また，これらのプログラム解析によって抽出された情報を，様々な目的に応じて応用するプログラム解析技術も提案されている．代表的なプログラム解析技術として，コンパイラにおける最適化や，フォールト位置の特定などを主目的としたプログラムスライスなどが挙げられる．

本論文では，情報漏洩解析手法，影響波及解析手法，ソフトウェア部品の再利用性評価手法の 3 つのプログラム解析技術に着目し，保守や再利用における支援を目的とした以下の解析手法を提案および実現する．

1. プログラムスライシングを利用した情報漏洩解析手法
2. オブジェクト指向言語を対象とした影響波及解析手法
3. 利用関係を用いたソフトウェア部品評価手法
4. 動的情報を用いたソフトウェア部品評価手法

情報漏洩解析手法は，プログラムに入力される機密度の高い情報が，プログラムの実行を通してどのように処理および出力されるかを解析する手法である．主にプログラムの検証に利用されるが，情報フローを用いて入力値の機密度から，各出力にどの機密度を持つ値が出力されうるかを解析し示すことで，プログラムの注意すべき部分の把握や安全性の確認に利用できると考えられる．しかし，現在の手法が対象としている言語はきわめて単純な構造の言語のみで，実用性に乏しく，情報フローを用いた手法の実現例は存在しない．

1. では，プログラムスライスにおけるプログラム依存グラフを利用することで，情報漏洩解析を行う手法を提案する．プログラムスライスにおける手法を情報漏洩解析に利用することで，より一般的な言語を対象とした情報漏洩解析が可能となる．また，提案手法を既存のスライシングツールに機能追加の形で実装し，適用した事例を紹介する．適用事例では，同じようなプログラムであっても情報フローが異なるために機密度の高い出力文の数に大きな違いが出ることから，情報漏洩解析によるプログラムの安全性の確認が重要であることを確認した．また，一つの関数の返り値を隠蔽するだけで，機密度の高い出力文が大きく減少することから，関数の返り値の機密度を変更できる機能が情報隠蔽を考慮すべき部分の推定に利用でき，より現実的な情報漏洩解析が行えることを確認した．

影響波及解析手法は，プログラムに対する変更において, 変更の影響を受ける部分 (被影響部分) を識別するための手法である．回帰テストにおける，再試験が必要なテストケースの限定を主目的としているが,「プログラムの改変時に，計算結果が変わりうる場所を知りたい」という場合に保守支援に利用できると考えられる．しかし，現在の影響波及解析手法における影響の定義が「テストケースの限定」を目的としたものに特化されており，状況に応じた利用目的の変化を想定していない．また，手法の実現例が少なく，オブジェクト指向言語の特徴を考慮した影響波及解析手法の実現が必要となっている．

2. では，オブジェクト指向言語を対象とした影響波及解析手法を提案および実現する．提案手法では，オブジェクト指向言語 JAVA を対象に，クラスメンバ間の関係を表す 2 つのグラフに基づき解析を行う．影響の種類に応じて辺を定義し，影響波及解析を行う際にユーザの利用目的に応じて影響波及ルールを定義することで，目的に応じた影響波及解析を行うことができる．また，提案手法を JAVA 影響波及解析システムとして実装し，実際のソフトウェアの更新履歴に適用した．適用したところ，システムにより抽出された被影響部分は，実際の変更作業において修正が行われていた．そのため，本システムを用いることにより修正個所の特定を効果的に行うことができると考えられる．

近年，大量の大規模ソフトウェアが開発されており，開発現場ではソフトウェアの再利用がよく用いられているが，現在のプログラム開発環境では個々の開発者の知識に頼っており，知識の共有が満足になされていない．知識の共有を目的としたソフトウェア部品検索システムにおいては，部品を選別する基準として，目的にあった部品を選出できることと同様に，いろいろなソフトウェアで使われ完成度が高い部品を選出できることが重要となる．そのため，部品の利用実績を定量的に評価した指標が必要となるが，従来の再利用性評価手法は部品単体から抽出可能な特性のみを用いて評価したもので，利用実績を考慮したものではない。

3. では，ソフトウェア部品の検索において部品の選別に利用するための手法として，部品間の利用関係に基づいて各部品を評価し，順位付けする手法を提案する．この手法では，複数のシステム内の部品間に存在する利用関係から構築されたグラフを用いて，繰り返し計算により各部品の重みを求める．求められた重みは，開発者が利用関係に沿って参照を行うと仮定した場合の各部品の参照されやすさを表しており，よく利用される部品や，重要な部品から利用される部品の順位が高くなる．提案手法に基づき，JAVA で開発されたソフトウェア群から部品順位を計測するシステムを開発し，適用実験を行う．実験結果では，利用される回数の多いクラスや重要な機能を持つクラスが上位を占め，本手法で測定した部品の順位が利用実績を示すものであることを確認できた．この手法を部品検索において利用することで，利用実績のある部品を効果的に取得できることが期待できる．

さらに 4. では，動的情報を用いて部品を評価する手法を提案する．提案手法では，ソフトウェア実行時に得られる動的な呼び出し関係を用いて解析を行うことで，ある実行時に中心的な役割を果たしている部品を容易に特定することができる．この提案手法をもとに，Java アプリケーションを対象として，部品の貢献度を評価値として求めるシステムを実現し，適用実験を行った．適用実験では，プログラムの実行時に中心的な役割を果たしている部品が利用回数にかかわらず上位を占め，本手法が対象システムの振る舞いの理解に有効であることを確認した．

論文一覧

主要論文

[1-1] Reishi Yokomori, Fumiaki Ohata, Yoshiaki Takata, Hiroyuki Seki, Katsuro Inoue: “Analysis and Implementation Method of Program to Detect Inappropriate Information Leak ”, Proceedings of the Second Asia-Pacific Conference on Quality Software(APAQS 2001), pp.5-12, HongKong, China, December, 2001.

[1-2] Reishi Yokomori, Fumiaki Ohata, Yoshiaki Takata, Hiroyuki Seki, Katsuro Inoue: “An information-leak analysis system based on program slicing”, Information and Software Technology, published by Elsevier, Vol.44, No.15, pp.903–910, 2002.

[1-3] 横森 励士, 近藤 和弘, 大畑 文明, 井上 克郎: “オブジェクト指向プログラムの変更作業を支援する影響波及解析システム ”, 電子情報通信学会論文誌 D-I, Vol.J86-D-I, No.3, pp.150-158, 2003.

[1-4] Reishi Yokomori, Takashi Ishio, Tetsuo Yamamoto, Makoto Matsushita, Shinji Kusumoto, Katsuro Inoue: “Java Program Analysis Projects in Osaka University: Aspect-Based Slicing System ADAS and Ranked-Component Search System SPARS-J”, Proceedings of the 25th International Conference on Software Engineering (ICSE2003), pp.828–829, Research Demonstration, Portland, Oregon, USA, May, 2003.

[1-5] 横森 励士, 藤原 晃, 山本 哲男, 松下 誠, 楠本 真二, 井上 克郎: “利用実績に基づくソフトウェア部品重要度評価システム”, 電子情報通信学会論文誌 D-I(採録決定).

[1-6] 藤井 将人, 横森 励士, 山本 哲男, 井上 克郎: “動的情報を利用したソフトウェア部品評価手法”, 電子情報通信学会論文誌 D-I(採録決定).

関連論文

[2-1] 大畑 文明, 横森 励士, 西松 顯, 井上 克郎: “スライス計算効率化のためのプログラム依存グラフの節点集約法”, 電子情報通信学会論文誌 D-I, Vol.J84-D-I, No.7, pp.1021-1029, 2001.

[2-2] Katsuro Inoue, Reishi Yokomori, Hikaru Fujiwara, Tetsuo Yamamoto, Makoto Matsushita, Shinji Kusumoto: “Component Rank: Relative Significance Rank for Software Component Search”, Proceedings of the 25th International Conference on Software Engineering (ICSE2003), pp.14–24, Portland, Oregon, USA, May, 2003.

[2-3] 山中 祐介, 横森 励士, 井上 克郎: “エイリアス関係を考慮した Java スライシングツール”, 電子情報通信学会論文誌 D-I(採録決定).

謝辞

本研究の全般に関し，常日頃より適切なご指導を賜わりました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授に，心から深く感謝申し上げます.

本論文を執筆するにあたり，適切なご助言とご指導を頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 谷口 健一 教授，同マルチメディア工学専攻 藤原 融 教授に心から感謝致します.

大阪大学大学院基礎工学研究科情報数理系専攻在籍中に，適切なご助言とご指導を頂きました，大阪大学大学院情報科学研究科 宮原 秀夫 教授，柏原 敏伸 教授，菊野 亨 教授，萩原 兼一 教授，今井 正治 教授，東野 輝夫 教授，今瀬 真 教授，村田 正幸 教授，増澤 利光 教授，松田 秀雄 教授に感謝致します.

本論文を執筆するにあたり，直接ご指導および適切なご助言を頂きました，奈良先端科学技術大学院大学情報科学研究科 関 浩之 教授，高田 喜朗 助手に心より感謝致します.

本論文を執筆するにあたり，直接具体的なご指導を頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 楠本 真二 助教授，松下 誠 助手に心より感謝致します.

本研究を行うにあたり，ご助言やご指導を頂きました，奈良先端科学技術大学院大学情報科学研究科 國信 茂太 氏に感謝致します.

本研究を行うにあたり，ご助言やご指導を頂きました，株式会社東芝 大畑 文明 氏，ソニー・エリクソン・モバイルコミュニケーションズ株式会社 近藤 和弘 氏，ソニー株式会社 藤井 将人 氏，松下電器産業株式会社 藤原 晃 氏，科学技術振興事業団 山本 哲男 氏に感謝致します.

最後に，井上研究室の皆様のご助言, ご協力に御礼申し上げます.

目 次

図 目 次

表 目 次

第1章　まえがき

1.1　プログラム解析

近年，様々な場所で多様な目的でソフトウェアが頻繁に利用されるようになり，ソフトウェアシステムが担う社会的役割もますます重要となっている．これらの状況の下，コンピュータ上で動作するソフトウェアは大規模化の一途をたどっており，ソフトウェア開発はますます複雑なものとなってきている．ソフトウェアの大規模化に伴い，プログラム開発段階におけるデバッグのコストが開発コストの 50〜80 %を占めるようになったといわれている [35]．それに伴い，保守段階においてプログラムを理解し，どの場所に変更を加える必要があるかを把握することも極めて難しくなっている．

そのため高品質な複雑なソフトウェアを効率よく開発することは，ソフトウェアに関する研究において重要なテーマとなっている [31]．開発作業における生産性向上を目指し，さまざまな研究活動が行われているが，その中の一つとして，ソフトウェア開発工程において重要な要素の一つであるプログラム（または，そのソースコード）からその性質や振る舞いを抽出し、それを開発者に提供することでソフトウェア開発を支援する，**プログラム解析**（*Program Analysis*）がある [20].

プログラム解析とはプログラム内の関係や性質を

- グラフ化
- 数値化
- 記号化
- …

することによりプログラムを抽象化し，抽象化した情報を利用して解析を行うことを指す．利用目的に応じてプログラムを抽象化することで，開発者にとって有用な「プログラムの特徴」のみを抽出することが容易となり，ソフトウェア開発を支援することができる．

プログラム解析におけるグラフ化は，解析対象全体において個々の関係を抽出し，抽象化し表現することを目的とした場合が多い．グラフ化の例として，

- プログラムのソースコードを木構造で表現した抽象構文木
- 手続き，メソッドなどの呼び出し関係をグラフ化した CFG(Call Flow Graph)
- クラスの継承関係をグラフ化したクラス階層構造
- プログラム文間のデータの流れや制御構造をグラフ化したプログラム依存グラフ

などが挙げられる．グラフ化によって個々の関係が明確になるため，構文木からプログラム依存グラフを作成する場合のように，グラフ化された情報を用いてより高度なプログラム解析が行われることも多い．例としては，エイリアス関係（同一メモリ空間を指す可能性のある式間の同値関係）にある変数の対の情報をもとに，より正確なデータフロー関係の解析を行うなどが挙げられる．一般的に複数の解析を組み合わせた場合，解析コストが上がるが，得られる情報の精度が向上することが知られている．

解析方法の例としては，

- グラフ上の辺の探索による，到達可能な節点の計算
- グラフの比較による，同一部分の検出
- クラスの階層構造の深さ，手続き（メソッド）の数などの数値化
- 利用関係などの関係の行列化

などが挙げられる．

一方で，プログラム解析における数値化（記号化）は，解析対象における個々を抽象化し個々の性質を取り出すことを目的としている．解析情報の数値化の例として

- クラス数，メソッド数，LOC などのメトリクス
- トークンの抽象化を目的とした記号化
- プログラムの品質や再利用性の評価値
- ソフトウェア間の類似度

などが挙げられる．これらの数値化された情報は個々の性質をある観点から観測したもので，これらの情報を複数組み合わせることで，より多面的な観点から個々の解析対象を観測することができる．そのため，これらの数値を組み合わせることで，部品を評価するための新たな評価基準を生み出すができることも多い．数値化された情報の多くは，統計的手法を用いた評価に利用されることが多い．また，記号化された情報は個々の性質をある観点から観測したものであるが，配列化や行列化を行うことで，解析対象全体の特徴を示すことができる．そのため，統計的手法を用いた評価が行われることもあるが，単に比較するために利用されることも多い．

これらの情報の抽出には大別して，静的解析，動的解析の 2 種類の方法が利用される．静的解析とは，可能性のあるすべての実行において，一度でも起こる事象に関する情報をすべて収集する手法で，ソースコードなどプログラム実行前の情報のみを用いて行われる．その一方で，動的解析は，特定の入力が与えられたある一つの実行において，そこで起こる事象に関する情報のみを収集する手法で，実行履歴などプログラム実行後の情報を用いて解析が行われる．

1.1.1 プログラム解析情報の利用例

これらの抽出された情報をもとに，利用目的に応じて様々な解析が行われている．

- グラフ化した情報を用いたプログラム解析の例

  最適化コードの生成: コンパイル時に必要のない命令を削除

  テストデータの自動生成: 「テストを行いたい実行経路」を通るような入力データを実行履歴から生成

  プログラムの結合: 似た部分を結合することで，ただ単に結合した場合よりも高速化を計る

  デバッグ支援: プログラムスライス [38] を用いることで，デバッグ対象を限定

  影響波及解析: 再テストすべきテストケースを限定することで，テスト工程を効率化

  モデルチェック: プログラムの正当性や安全性の検証

  情報漏洩解析: プログラムの中で，セキュリティポリシーを満たさない文を検出

  プログラム理解支援: 解析結果情報を提示することで，保守およびデバッグ作業を支援

- 数値化（記号化）した情報を用いたプログラム解析の例

  ソフトウェア部品の評価: 数値化された部品の性質から再利用性や品質を評価

  コードクローンの把握: コピーされたソースコードの検出

  コピー部品の把握: 数値化（記号化）された情報を配列化し，解析効率を上げる

  ソフトウェア（部品）のクラスタリング: 単語の出現頻度やソフトウェア間の類似度などを比較し，分類

  理解支援: 解析結果情報を選別の基準とすることで，大量の部品からの選別作業を支援

なお，ここで挙げる例は一部で，抽出されるプログラム解析情報および利用目的はこれら以外にも多く存在する．また，いくつかのプログラム解析情報を組み合わせることで，新たな利用目的も考案されている．

本論文では，プログラム解析手法のうち，グラフ化した情報を用いたプログラム解析の例として

- 情報漏洩解析
- 影響波及解析

に，数値化した情報を求めるプログラム解析の例として

- ソフトウェア部品の再利用性評価

に着目する．以降，情報漏洩解析，影響波及解析，再利用性評価手法についてそれぞれ説明し，デバッグや保守工程におけるプログラム理解の支援や，再利用における部品の理解支援へ利用する方法について考察する．

1.2 情報漏洩解析

1.2.1 セキュリティクラスと情報フロー

セキュリティクラス（*Security Class,SC*）とは各データの機密度を表す値で，データ d の SC を $SC(d)$ とする．同様に，**クリアランス**（*Clearance*）はユーザ（プロセス）がどの程度のデータまでアクセスできるかを表し，ユーザ u のクリアランスを $clear(u)$ とする．

システムにおける各 SC の機密度の強弱を表したものを**セキュリティモデル**（*Security Model*）といい，各元が SC と対応した束を用いて表される．セキュリティモデルでは任意の 2 元の最小上界，最小下界が定義されており，一意な最大元（*high*），最小元（*low*）が存在する．プログラム中の定数の SC は low である．セキュリティモデルの例を図 1.1-(a) に示す.

あるオブジェクト v からオブジェクト u へ，何らかの形で情報が推移していくとき，v から u への**情報フロー**（*Information Flow*）が存在すると定義する．情報フローにはその推移の仕方によって，**明示的フロー**（*Explicit Flow*），**暗示的フロー**（*Implicit Flow*）の 2 種類が存在する．プログラム中の文 s_1 で変数 v を定義しており，文 s_2 でその変数 v を参照して変数 u が定義されているとき，u の値は v によって決まるため，v から u に明示的フローが存在すると定義する．また文 s_1 が変数 v を参照する条件文であり，その条件文のブロック内に変数 u を定義する文 s_2 が存在するとき，v の値によって文 s_2 が実行されるかどうかが決まる，そのため v から u に暗示的フローが存在すると定義する．

図 1.1-(a) のようなセキュリティモデルを持つプログラム上で，$SC(x) = 1$，$SC(y) = 2$, $SC(z) = 0$ のとき，x と y の和を z に代入するときを考える（z := x + y）．このとき x と y の値が z にフローし，z は x と y 両方のデータに関する情報を持つことになる．このとき z を参照できるのは，x と y 両方のデータを読むことができるクリアランスを持つユーザであるべきである．よって，代入後の $SC(z)$ は，$SC(x)$ と $SC(y)$ の最小上界となり $SC(z) = 3$ となる．このように，複数のデータからの情報フローを受ける変数は，入力となるデータの最小上界である SC を持つことになる．

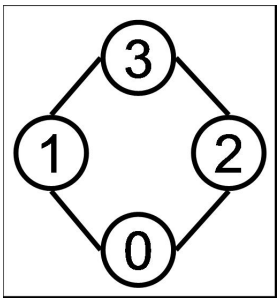
図 1.1-(b) に，図 1.1-(a) のようなセキュリティモデルを持つプログラム上での，情報漏洩解析を示す．$SC(x)$ と $SC(y)$ の最小上界を取る演算を SC の和演算といい $SC(x) \vee SC(y)$ と示す.

1.2.2 不適切な情報フローの検出を目的とした情報漏洩解析

クレジットカード番号等，第三者に知られてはならない情報を扱うプログラムや，不特定多数の人間が利用するシステムにおいては不適切な情報漏洩を防ぐことは重要な課題である．このようなシステムにおいて情報漏洩を防ぐ手段として，*Mandatory Access Control* と呼ばれる次のようなアクセス制御法がよく用いられる [32]:

(1) 各データに対してその機密度を表す**セキュリティクラス**（*Security Class*，以下，SC と省略する）を割り当てる．以下，データ d の SC を $SC(d)$ と表す.

(2) 同様に，ユーザ（プロセス）に対し，どの程度のデータまでアクセスできるかを表す**クリアランス**（*Clearance*）を割り当てる．以下ユーザ u，プロセス p のクリアラン



$low = 0, high = 3$
(a) セキュリティモデル

```
{ 各変数のSCの初期値
 SC(a)=1, SC(b)=0, SC(c)=2, SC(d)=0 }

 1:  b := a + 1;
       { SC(b)=SC(a)∨0 }
 2:  if (c > 0) then begin
 3:     d := a + 1
       { SC(d)=SC(c)∨SC(a)∨0 }
 4:  end;

{ 終了後の各変数のSC
 SC(a)=1, SC(b)=1, SC(c)=2, SC(d)=3 }
```

(b) 情報漏洩解析の例

図 1.1: セキュリティモデルと情報漏洩解析の例

スをそれぞれ $clear(u)$，$clear(p)$ と表す．

(3) $clear(u) \geq SC(d)$ のときかつそのときのみ，ユーザ u はデータ d を読むことができる．同様に，$clear(p) \geq SC(d)$ のときかつそのときのみ，プロセス p はデータ d を読むことができる．

しかし，このアクセス制御法の場合，次のような方法でプログラムにより不適切な情報漏洩が起こる．

(a) クリアランスが $SC(d)$ 以上のプロセス p がデータ d を読み込む．

(b) プロセス p がデータ d を，故意にまたは過失によって，$SC(s) < SC(d)$ であるような記憶域 s に書き出してしまう．

(c) 直接データ d を読むことができないユーザ U（$SC(s) \leq CL(U) < SC(d)$）であっても U は s に書かれている情報は読める．そのため，s にアクセスすることで，U は d に関する情報を間接的に取得でき，情報が漏洩する．

Denning らは，このような情報漏洩を防ぐためにプログラムを静的に解析する手法を提案した [10, 11]．これらのプログラムの出力の安全性を確認する手法を**情報漏洩解析**と呼ぶ．この手法では，まず，プログラムの入力となる値や変数に対して SC を，ファイルなどの出力域に対してクリアランスを設定する．つぎに，プログラム中で利用される文内の変数間のデータ授受関係を表す**情報フロー**（*Information Flow*）に基づき，不適切な情報漏洩の検出を行う．不適切な情報漏洩を引き起こす情報フローとしては，

- 低い SC を持つ変数への代入文において，高い SC を持つ変数が参照される
- 低いクリアランスを持つ出力文において，高い SC を持つ変数が参照される

がある．また [2] では，Banâtre らは [11] の手法を理論的に再検討し，解析手法の一般化を試みた．

しかし，これらの手法では再帰手続きや大域変数を考慮していないこともあり，解析対象となるプログラムが単純な構造のものに限られていた．また，関数呼び出し文に対する

解析の際，戻り値の判定は実引数全体に対してのみで，戻り値の計算に実際にどの引数の値が利用されているかを考慮していないなど，不正確な面もあった．

1.2.3 文間の情報フローを利用した情報漏洩解析

情報漏洩解析において文間の情報フローを考慮することで，変数が実行の各時点で持ちうる値の SC に基づいた解析ができるため，再帰手続きや大域変数を考慮することができる．[29] では，再起呼び出しを考慮した手法として，Palsberg は”Trust-analysis” と呼ばれる手法を提案している．この手法は，プログラム中で利用される値それぞれに対して，その値が信頼できるかそうでないかを表す信頼度（reliability）を付加し，各文で参照される値が信頼できる値かそうでないかを判定する．この手法の場合，情報の漏洩を検出するだけでなく，信頼できない値が解析対象プログラムの重要な部分で使われうるかどうかも判定することができるが，解析対象言語を拡張する必要があり，現実的であるとはいえない．

国信は [50, 51] において，再帰を含む手続き型プログラムに対する情報フロー解析アルゴリズム提案した．この手法は，文内および文間の情報フローを用いてプログラムの入力値の SC から各出力が保持しうる SC を求めることで，プログラムに入力される機密度の高い情報が，プログラムの実行を通してどのように処理および出力されるかを解析する．

この方法では，解析対象プログラム中すべての実行文について，その文の実行前後の各変数の SC 間で成り立つべき再帰的な関係式を定義する．この関係式に基づき，プログラムの各手続きの実行結果の SC を解析する．プログラムの main 関数の仮引数が $x_1, \ldots, x_i$，入力ファイルが $infile_1, \ldots, infile_j$，main 関数の戻り値が y_1，出力ファイルが $ofile_1, \ldots, ofile_k$ であるとき，実引数と入力ファイルに対応する i + j 個の SC の組が与えられると，それらをもとに上記の関係式を同時に満たす最小解が繰り返し計算により求められる．そして，この解が戻り値と出力ファイルに対応する 1 + k 個の SC となる．

この手法の場合，結果をソースコード上に反映させることで，検証だけでなく，保守作業やデバッグ作業における安全性の確認に利用できる．また,「機密性の高い情報が，どの部分に出力されるのかを把握したい」というような，プログラムの振る舞いの理解支援にも利用でき，保守作業において注意を払うべき部分の把握に利用できると考えられる．

1.2.4 実際に採用されている情報漏洩解析の例

プログラムの実行時において，実行中に変数の操作と共に機密度のチェックを行うにより，不適切な情報のやり取りを制限する手法が，実際のプログラムミング言語において実現されている．

一つ目は，JavaScript 1.1 において採用された**データ汚染セキュリティモデル**, *data tainting* と呼ばれる手法である．データ汚染セキュリティモデルは JavaScript 1.1 で採用されたが，JavaScript 1.2 ではセキュリティーモデルの変更によりその機構は削除された．

データ汚染セキュリティモデルは Same Origin Policy（ドメイン，プロトコル，ポート番号が同じサイトのドキュメントなら，そのプロパティにアクセス可能にするポリシー）に基づいてアクセスを制限したうえで，ページ上のコンポーネントに対して安全なアクセスを保証するための機構である．

データ汚染セキュリティモデルが有効な場合，ウインドウ中の JavaScript は他のウインドウの状態を監視する．もしあるスクリプトが他のサーバーから何らかの情報を得た場合，データ汚染セキュリティモデルはそのデータに「汚染された（”tainted”）」という情報を付加する．汚染された状態で他のページにアクセスした場合，データ汚染セキュリティモデルは安全でないアクセスが起こりうることを警告することで，情報の漏洩を防ぐ．

このデータが「汚染された」という情報はデータフロー関係に基づいて伝播する．つまり，JavaScript 内の文中で汚染されたデータを参照して値が定義された場合，その値を持つ変数にも「汚染された」という情報がつけられ，汚染された情報のロンダリング（汚染されていない状態に戻すこと）を不可能としている．

二つ目は，Perl において実現されている**データ汚染モデル**, *data-tainting model* である．Perl におけるデータ汚染モデルはプログラムの外から入ってくるデータの利用を制限するための機構である．

Perl が汚染モードで実行された場合，全てのコマンドライン引数，環境変数，ファイルから入力されるデータなどに対して「汚染された」という情報が付加される．JavaScript の場合と同様に,「汚染された」という情報はデータフロー関係や制御フロー関係に基づいて伝播する．汚染された情報は，コマンドの実行に利用することや，ファイル，ディレクトリ，プロセスを修正するようなコマンドにも利用できない．このようにして，情報漏洩を防ぐための機構を Perl は保持している．

これらの手法は，データフロー関係や制御フロー関係を用いて，文間の情報フローを利用した解析を行っている．文間の情報フローを用いることで，解析の各時点で各変数に今どういう機密度の情報が入りうるか（入っているか）という情報が得られるため，情報漏洩解析の精度を上げることができる．

しかし，これらの手法は，いずれも実行時に情報の漏洩を水際で食い止めるための手法で，あらかじめ実行前に解析を行って情報の漏洩の有無を検証するための手法ではない．またこれらの手法は，汚染されたか，汚染されていないかの 2 値の情報だけを扱っており，一般的なセキュリティモデルを対象とした手法ではない．そのため，一般的なセキュリティモデルを対象として，文間の情報フローを用いて実行前に情報漏洩の有無を検証するための手法が必要となる．

1.3 影響波及解析

ソフトウェア保守工程において，プログラム開発者はソフトウェアに対し多くの変更を行うが，その際，誤って欠陥を作り込んでしまう確率は 50%から 80%にも及ぶことが Hetzel により示されている [17]．要因としては，ソフトウェアに変更を加えたときに，変更していない部分に関しても何らかの影響が及ぶ可能性があることが挙げられる．

ソフトウェアに加えられた変更による影響を受ける部分（**被影響部分**と呼ぶ）を識別するための手法として，影響波及解析が提案されている．影響波及解析の適用分野の代表例として，変更後のソフトウェアが仕様通りに動作するかを確認するための，**回帰テスト**（*Regression Testing*）[16] への利用が挙げられる．回帰テストは，

Step 1: 被影響部分の識別

Step 2: 修正コンポーネントの再テスト方法の決定

Step 3: 再テストによる補償範囲の認識

Step 4: テストケースの選択，再利用，修正，新規作成

という過程をたどるが，**Step 1: 被影響部分の識別**において影響波及解析を利用することで，適用すべきテストケースを変更した部分に関係のあるものだけに限定し，必要最小限に抑えることができる．

また，現在のソフトウェア開発環境ではオブジェクト指向言語が多く利用されている．オブジェクト指向プログラムでは，従来の手続き型プログラムに比べ，変更箇所以外に影響を及ぼすような変更が数多く存在する．文献 [12, 25] では，オブジェクト指向プログラムにおける変更は，メソッドの**オーバーライド**（*Override*），フィールドの**隠蔽**（*Hide*）といったオブジェクト指向特有の概念により様々な影響が引き起こされることが述べられている．

1.3.1 影響波及解析の分類

これまでにオブジェクト指向プログラムに対する影響波及解析手法がいくつか提案されている [8, 22, 36, 26, 24]．ここでは，既存手法を解析の粒度で大きく 3 つに分類する．

クラス単位: クラスを被影響部分の単位とする [8]．メンバ単位・文単位の解析に比べて解析の粒度が大きく，数百クラスにもおよぶような大規模ソフトウェアに対して変更を行う際に有効な手法である．しかし，クラス内のどのメンバが影響を受けているかを特定できないため，影響の予測としては効果が薄く，回帰テストにおいて再テストの必要がないメンバも解析結果に含み得ることが指摘されている [33].

メンバ単位: クラスのメンバ（メソッド，フィールド）を被影響部分の単位とする [22, 26]. オブジェクトの構成要素であるメンバが解析結果となり，直感的に理解しやすく，クラス単位での解析より正確な結果を得ることができる．

文単位: 文を被影響部分の単位とする [24, 36]．**プログラムスライス**（*Program Slice*）[38] に基づいており，ある文に影響を及ぼす文の集合および，ある文が影響を及ぼす文の集合を抽出することで被影響部分を特定する．クラス単位およびメンバ単位の解析に比べて正確な結果を得ることができるが，文間の依存関係など，多くの解析が前提となり，解析コストは膨大になる．

以降では，メソッドの追加，削除，またシグニチャの変更といったメンバに関する変更を対象とし，メンバに関する変更による被影響部分の抽出を目的とした「メンバ単位での影響波及解析手法」に着目する．

影響波及解析は，回帰テストにおける，再試験が必要なテストケースを限定による効率化を主目的としているが，我々は，影響波及解析を保守工程などにおいてプログラム理解支援に利用できると考えている．例えば，保守工程においてプログラムの改変を行うときに,「プログラムの改変を行った際に、どの部分の計算結果が変わりうるかを知りたい」というようなケースだけでなく,「メソッドをオーバーライドしたときに，呼び出し関係が変わるメソッドを知りたい」というようなケースも考えられる．このような場合に，影響の定義を利用状況に応じて変化させ，解析結果をソースコード上で表示することで，影響が

起こる箇所を開発者に教えることができ，プログラムの振る舞いの理解支援にも利用できると考えられる．

1.4 再利用性評価手法

近年，大規模で複雑なプログラムを含む大量のソフトウェアが開発され，様々な場所で様々な目的で利用されている．これらのプログラムの中には似た種類のプログラムが多数存在し，開発期間の短縮や品質向上を目的として，ソフトウェアの再利用がよく用いられている．

一般にソフトウェア部品 (Software Comopnent) は再利用できるように設計された部品とされ，特に部品の内容を利用者が知る必要のないブラックボックス再利用ができるものを指すこともある [21, 49]．本論文ではより一般的に，ソースコードファイルやバイナリファイル，ドキュメントなどの種類を問わず，開発者が再利用を行う単位をソフトウェア部品，あるいは単に部品と呼ぶ．

再利用性とは，個々の部品がどれだけ再利用しやすいかを定量的に評価したものである．個々の部品の再利用性を評価する手法は数多く提案されている．Etzkorn らは, Modularity, Interface Size, Documentation, Complexity の 4 つの視点からオブジェクト指向ソフトウェアの再利用性を評価する手法を提案している．彼らはソースコードから計測される複数のメトリクスを正規化して足し合わせることで再利用性のメトリクスとし，C++のソースコードに対して実際にプログラマが評価した再利用性とメトリクス値を比較している [13].

また，山本らはソースコードが非開示な部品に対して，そのインターフェイスから再利用性を評価する手法を提案している．彼らは理解容易性，利用容易性，テスト容易性，可搬性の 4 つの視点から再利用性メトリクスを定義し，JavaBeans を対象として実際にプログラマがアプリケーションを実装した結果とメトリクス値を比較している [59].

これらの方法は全て，部品の構造やインターフェイスなど部品そのものの持つ静的な特性を計算して再利用性を評価するものとなっている．これらの手法は全て，部品そのものから読み取れる特性のみを計算して再利用性を評価するもので，実際のプログラマによる主観的な再利用性の評価の結果と似ているという結果が得られている．

1.4.1 再利用支援を目的とした部品検索システム

ソフトウェアの再利用による効果を最大限に引き出すためには，開発者が今から開発しようとするソフトウェアに必要な部品およびライブラリに関する知識を持つことが重要になる．しかし，知識の共有が満足になされていないために，同種のプログラムが別々の場所で，独立して開発されていることも多い．

一方でインターネットの普及により，SourceForge [46] などのソフトウェア開発に関する情報を交換するコミュニティが誕生し，大量のプログラムソースコードが簡単に入手できるようになった．これらのインターネット上で公開されている大量の部品（プログラムソースコード）の中から，開発者の必要としている機能を持つ部品，その機能の使い方を示している部品のような，再利用に有益な情報を提供する検索システムを実現することで，知識の共有が実現でき，再利用を促進することができると考えられる．

1.5 既存の各プログラム解析手法における問題点

以下では，本論文で注目する情報漏洩解析，影響波及解析，部品評価手法それぞれにおける問題点について論ずる．

1.5.1 情報漏洩解析手法における問題点

情報漏洩解析手法は，プログラムに入力される機密度の高い情報が，プログラムの実行を通してどのように処理および出力されるかを解析する手法である．

文内の情報フローのみを考慮して安全性の確認を行う手法 [2, 10, 11] の場合，変数が実行の各時点で持ちうる値の SC ではなく，変数に割り当てられた SC のみを用いて検証を行うため，再帰手続きや大域変数を考慮していない．また，関数呼び出し文に対する解析の際，戻り値の判定は実引数全体に対してのみで，戻り値の計算に実際にどの引数の値が利用されているかを考慮していないなど，不正確な面もある．そのため，現実的に適用可能となるプログラムが小規模なものに限られている．

一方で，文間の情報フローを用いて，入力値の機密度から各出力にどの機密度を持つ値が出力されうるかを解析し示す手法の場合，変数が実行の各時点で持ちうる値の SC に基づいて解析をおこなうため，再帰手続きや大域変数を考慮することができる．しかし，[29] で提案されている手法は解析対象言語に信頼度に関する構文を追加する必要があり，実際のプログラミング環境での利用を考えると現実的であるとはいえない．

一方で [50, 51] の手法は，解析対象言語を拡張する必要はないため，実用的なプログラムに対してもある程度の正確性を保ったまま，適用可能であると考えられる．しかし，手法の提案および健全性の証明のみで，実現がなされていないため，手法を実現する方法を提案し，その方法に基づいてシステムを構築する必要がある．

1.5.2 影響波及解析手法における問題点

影響波及解析手法はソフトウェアに加えられた変更による影響を受ける部分（**被影響部分**と呼ぶ）を識別する手法である．これまでにオブジェクト指向プログラムに対する影響波及解析手法がいくつか提案されているが, クラスを解析の粒度とした場合，解析の精度の面でに問題があり影響の予測としては効果が薄いことが知られている．また，文を解析の粒度とした場合，多くの依存関係解析を前提とするため，解析コストが莫大になるということが知られている．

解析コストと解析精度のバランスを考慮した手法として，クラスのメンバ（メソッド, フィールド）を被影響部分の単位とする [22, 26]，メンバ単位での影響波及解析が提案されている．この手法の場合，オブジェクトの構成要素であるメンバが解析結果となり，直感的に理解しやすく，クラス単位での解析より正確な結果を得ることができる．また解析コストも，メンバ間の呼び出しおよび参照関係を把握するだけでよいので，解析コストが現実的なものとなる．

そのため，オブジェクト指向言語を対象として影響波及解析をおこなうことを考えた場合，メンバ単位での解析が最善であると思われる．しかし，メンバを解析の単位とした影響波及解析手法では，オブジェクト指向言語の独自概念は考慮されているが，手法の実現

には触れられていないものが多く，満足な実装がなされているとはいえない．そのため，オブジェクト指向言語の独自概念を考慮した解析手法およびその実装が求められている．

また，従来の影響波及解析手法は回帰テストにおけるテストケース選択用に特化した形で実現されており，被影響部分の探索ルールもテストケース選択用に特化して定義されていた．しかし，プログラム理解，保守などのより広い範囲での影響波及解析の利用を考えた場合，影響の定義は利用目的に応じて異なる．例えば，回帰テストにおけるテストケースの選択 [37] を目的とした場合には，変更または削除されたメソッドをテストするテストケース，およびオーバーライド関係の変化したメソッドへの呼び出し経路をテストするテストケースの検出が必要となる．一方で，修正個所の特定を目的としてメソッドオーバーライドの変化による影響を求める場合には，オーバーライド関係の変化したメソッドを直接呼び出してる部分を検出できれば十分である．そのため，ユーザの目的に応じて被影響部分の探索ルールが選択可能な解析手法が必要である．

1.5.3 部品の再利用性評価手法における問題点

知識の共有を目的として，ソフトウェアの部品検索システムを構築することで，ソフトウェアの再利用による効果を最大限に引き出すことができると考えられる．このような部品検索システムを構築する際には，検索された部品を評価し順位付けし，選別して表示する仕組みが必要となる．このような部品検索システムにおける利用を考えた場合，目的にあった部品を選出することと同様に，いろいろなソフトウェアで使われ完成度が高い部品を選出することが必要であると考えられる．このとき，各部品の利用実績を定量的に評価した指標が必要となる．

部品の特性から個々のソフトウェア部品の再利用性を評価する手法が提案されているが [13, 59]，従来の再利用性評価手法は，部品そのものから読み取れる特性のみを評価したもので利用実績については考慮されていない．現実には，従来手法では再利用性が低いと評価されても，多くのシステムで再利用されている部品は数多く存在すると考えられ，このような部品検索システムにおける利用を考えた場合，実際によく利用されているかを考慮した，いわゆる利用実績に関する評価を行う手法が必要となる．

1.6 本論文の概要

本論文では，情報漏洩解析手法，影響波及解析手法，ソフトウェア部品の再利用性評価手法の 3 つのプログラム解析技術に着目し，保守におけるプログラムの理解支援や，部品の再利用時の部品理解支援を目的とした以下の解析手法を提案および実現する．

1.6.1 プログラムスライシングを利用した情報漏洩解析手法

2 章では，プログラムスライスにおけるプログラム依存グラフを用いて情報漏洩解析を行う手法を提案および実現する．これにより，プログラムスライスにおける PDG 構築手法を情報漏洩解析に利用することができ，より一般的な言語を対象として情報漏洩解析を適用することが可能となると期待できる．さらに，実現したシステムを用いて情報フロー

を用いた情報漏洩解析手法の適用事例を紹介する．適用事例では，同じようなプログラムでも，情報フローが違うことで高い SC のデータを出力する文の数が大きく違うことから，情報漏洩解析による安全性の確認が重要であること，関数の戻り値の SC を指定できる機能を組み込むことで情報漏洩解析が情報隠蔽を考慮すべき部分の推定にも有効であることを確認する．

1.6.2 オブジェクト指向プログラムの変更作業を支援する影響波及解析システム

3 章では，オブジェクト指向言語を対象とした影響波及解析手法を提案および実現する．提案手法では，オブジェクト指向言語 JAVA を対象に，クラスメンバ間の関係を表す 2 つのグラフに基づき解析を行う．影響の種類に応じて辺を定義し，実際に影響波及解析を行う際にユーザの利用目的に応じて影響波及ルールを定義することで，目的に応じた影響波及解析を行うことができる．さらに，実現した影響波及解析システムを，実際のソフトウェア開発における変更に適用する．適用では，本システムを用いることにより修正個所の特定を効果的に行うことができることを示すために，システムにより抽出された被影響部分が，実際の変更作業で修正が行われたかを確認する．

1.6.3 利用関係を用いたソフトウェア部品評価手法

4 章では，ソフトウェア部品の検索における部品の選別を目的として，ソフトウェア部品間の利用関係から各部品を順位付けし，評価する手法 (Component Rank 法, CR 法) を提案する．提案手法では，各ソフトウェア部品間に存在する利用関係に基づいてグラフおよび行列を構築し，構築された行列に対して繰り返し計算を行うことで，各部品の重みを測定し，順位を決定する．求められる順位は，開発者が利用関係に沿って参照を行うと仮定した場合の各部品の参照されやすさを表しており，よく利用される部品や，重要な部品から利用される部品の順位は高くなる．この CR 法をソフトウェア部品の検索における再利用部品の選択基準として用いることで，従来の再利用性評価手法で欠けている観点である，利用実績面からの評価を行うことができると考えられ，知識の共有が容易となり開発者の負担を軽減できることが期待できる．また，CR 法に基づいて部品利用度を評価する CR-システムを実装し，本手法で測定した部品順位が利用実績に関する評価としてふさわしいかどうかを，幾つかのソフトウェアシステムに適用することで確認する．

1.6.4 動的情報を利用したソフトウェア部品評価手法

5 章では，動的情報を用いて部品を評価する手法を提案する．提案手法では，ソフトウェア実行時に得られる動的支配関係を用いることで，単に各部品間の呼び出し関係の推移も得られるだけでなく，部品の支配関係を定量化した値を得ることができる．実行時に中心的な役割を果たしている部品ほど高い評価値を得ることができるため，提案手法は対象システムの振る舞いを理解するのに有効な手法であると考えられる．この提案手法をもとに，Java アプリケーションを対象として部品評価値を求めるシステムを実現し，適用事例から評価値の妥当性を確認する．

第2章　プログラムスライシングを利用した情報漏洩解析手法

2.1　導入

情報漏洩を防ぐアクセス制御法として，*Mandatory Access Control*[32] と呼ばれる制御法が存在する．しかし，このアクセス制御法のもとでは，プログラムの実行により機密度の高い情報が誰にでも参照可能な記憶域に書き出されてしまうと，不適切な情報漏洩が生じる．そのようなプログラムによって引き起こされる不適切な情報漏洩を防ぐために，**情報漏洩解析**（*Information Leak Analysis*）と呼ばれる手法が提案されている．

情報漏洩解析は，まず Denning らによって情報フローの矛盾を検出する手法が提案された [10, 11]．また [2] では，[11] の手法を理論的に再検討し解析手法の一般化を試みている．しかし，これらの手法では再帰手続きや大域変数を考慮していないこともあり，解析対象となるプログラムが単純な構造のものに限られていた．また，関数呼び出し文に対する解析の際，戻り値の判定は実引数全体に対してのみで，戻り値の計算に実際にどの引数の値が利用されているかを考慮していないなど不正確な面があり，実際に関数内部を解析すれば矛盾がないことがわかる場合でも，矛盾を検出してしまうなど，解析が厳密になりすぎてしまうという欠点があった．

そこで，国信らは，再帰を含む手続き型プログラムに対する情報フロー解析アルゴリズムを提案した [51]．この方法では，解析対象プログラム中すべての実行文について，その文の実行前後の各変数の SC 間で成り立つべき再帰的な関係式を定義する．

本章では，[51] の実現手法として，プログラムスライスの計算手法を利用した情報漏洩解析手法を提案する．プログラムスライスの計算では，プログラム文間の依存関係を表すために, プログラム依存グラフ (Program Dependence Graph,PDG) と呼ばれる有向グラフを利用し計算を行っている．PDG の各頂点はプログラム中の各文に対応し, 各文間の依存関係はそれぞれの頂点を結ぶ有向辺で表される. 提案手法においては，プログラムスライスにおける PDG 構築ルーチンを流用し，**データフロー方程式**（*Data Flow Equation*）[1] における繰り返し計算に基づいて行う．これにより，既存の PDG 構築ルーチンの流用が可能となるため，[51] の手法の適用対象となる言語の幅が大きく広がることが期待できる．

さらに，提案手法の改良手法として，プログラムスライスにおける PDG を利用した情報漏洩解析手法 [53] を提案する．提案手法では，PDG の依存関係と情報フローの等価性に着目することで，PDG 上での情報漏洩解析を行う．PDG 上での探索は言語の違いによる影響を受けにくいため，他のプログラム言語への手法の移植が容易となることが期待できる．一度 PDG を構築してしまえば，入力値の SC を変更して再度解析を行う際も，同じ PDG を探索することで解析できる．そのため，再解析の時間的コストを抑えることが可能となる．

以下では，プログラム依存グラフ（PDG）について述べる．以降 2.2 節で PDG 構築ルーチンを利用した情報漏洩解析手法について述べ，試作したシステムについて紹介する．また適用事例をもとに，情報フロー解析による安全性の確認が重要であることを確認する．2.3 節では PDG を利用した情報漏洩解析手法について述べ，試作したシステムについて紹介する．また適用事例をもとに，情報隠蔽を考慮すべき部分の推定が有効に行えることを確認する．

2.1.1 プログラム依存グラフ

プログラム依存グラフ(*Program Dependence Graph,PDG*) はプログラム内の文の依存関係を表す有向グラフである [34]．PDG の頂点はプログラム中の各文，及び if 文や while 文の条件判定部分を表す．PDG 中には 2 種類の辺が存在し，それぞれ変数の影響を表す**データ依存辺**，および条件文や繰り返し文の制御の影響を表す**制御依存辺**が存在する．依存関係には以下の制御依存関係とデータ依存関係の 2 種類がある．

- データ依存関係
  以下の 3 つの条件を全て満たすとき，文 s_1 から文 s_2 へ変数 v に関して**データ依存** *(Data Dependence，DD)* があると定義し，s_1 に対応する頂点から s_2 を表す頂点に制御依存辺を引く．
  - 文 s_1 で変数 v を定義している．
  - 文 s_2 で変数 v を参照している．
  - 文 s_1 から文 s_2 への実行可能なパスのうちで，s_1 から s_2 間に変数 v を再定義している文が存在しないパスが存在する．
- 制御依存関係
  以下の条件を全て満たしているとき，文 s_1 から文 s_2 への**制御依存** *(Control Dependence，CD)* があると定義し，s_1 に対応する頂点から s_2 を表す頂点に制御依存辺を引く．
  - 文 s_1 が条件文である．
  - 文 s_2 が実行されるかどうかは，文 s_1 の結果に依存する．

2.1.2 プログラムスライスの計算手順

プログラムスライス（*Program Slice*）は Weiser[38] により提案された．プログラムスライスとは，**スライス基準**（*Slicing Criterion*）（対 $<s, v>$ で示され，s は文，v は s で定義若しくは参照される変数を表す）に影響を与える可能性のある文の集合を指す．

一般的にプログラムスライスは以下の手順で計算される．(図 2.1 の (A))

(A) step 1 構文解析，意味解析を行い，各文において参照される変数，定義される変数を決定する．

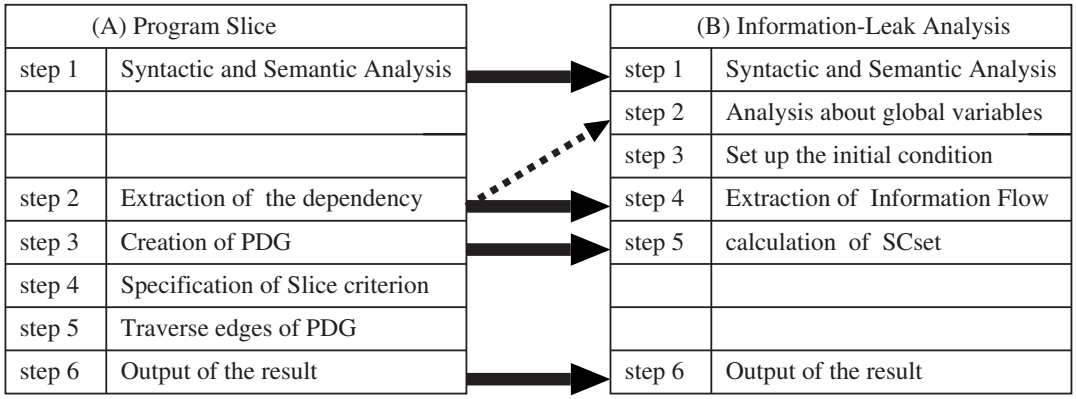


図 2.1: プログラムスライスの計算手順と情報漏洩解析手法の計算手順の比較

(A) step 2 各プログラム文間に存在するデータ依存関係および制御依存関係を，**データフロー方程式**（*Data Flow Equation*）による計算から求める．

(A) step 3 Program Dependence Graph (PDG) を抽出した依存関係から構築する．

(A) step 4 計算を行うために，スライス基準を指定する．

(A) step 5 スライス基準から PDG 辺を逆向きにたどることで，スライスを計算する．

(A) step 6 到達可能な節点の集合をプログラムスライスとして出力する．

2.2 PDG 構築ルーチンを利用した情報漏洩解析手法

ここでは，情報漏洩解析を行うためにプログラムスライスの計算手順をどのように変更するかについて述べた後で，データフロー方程式の計算に基づく解析手法について説明する．

2.2.1 情報漏洩解析のためのプログラムスライス計算手順の変更

[51] の手法は，手法の提案および健全性の証明のみで，実現がなされていない．そこで，プログラムスライスにおける PDG 構築ルーチンを流用し，**データフロー方程式**（*Data Flow Equation*）[1] における繰り返し計算に基づいて行う手法を提案する．これにより，プログラムスライスにおける PDG 構築手法を情報漏洩解析に流用することができ，より一般的な言語を対象として情報漏洩解析を適用することが可能となり，[51] の手法の適用対象となる言語の幅が大きく広がることが期待できる．

プロトタイプシステムにおける対象言語は Pascal のサブセットであり，表 2.1 にその BNF 表記の一部を示す．

また，実現手法における入力と出力を以下に示す．

入力

表 2.1: 解析対象プログラムの BNF 表記

```
型              = 標準型 | 配列型.
標準型          = “integer” | “boolean” | “char”
配列型          = “array”[ ] “of” 標準型 .

複合文          = “begin” 文の並び “end”.
文              = 基本文 |
                  “if” 式 “then” 限定文
                  “else” 文 |
                  “if” 式 “then” 文 |
                  “while” 式 “do” 文.
限定文          = 基本文 |
                  “if” 式 “then” 限定文 “else” 限定文 |
                  “while” 式 “do” 限定文.
基本文          = 代入文 | 手続き呼出文 |
                  入出力文 | 複合文 | 空文.
代入文          = 左辺 “:=” 式.
入力文          = “readln” [“(” 変数の並び “)”]|
出力文          = “writeln” [“(” 出力指定の並び “)”].
手続き呼出文= 手続き名 [“(” 式の並び “)”].
関数呼び出し= 関数名 [“(” 式の並び “)”].
```

プログラム: 解析対象プログラム

入力文における SC: 入力文で読み込まれる値の SC

手続き（関数）宣言部: 仮引数の SC

出力

プログラム中の各出力文における SC

図 2.1 の (A) のプログラムスライスの計算の手順を以下のように変えることで情報漏洩解析を行う．(B) step 2 中の大域変数に関する解析および，(B) step 5 中のデータフロー方程式を用いた解析は後述する．

(B) step 1 構文解析，意味解析を行い，各文において参照される変数，定義される変数を決定する．プログラムスライスにおける (A) step 1 と同じ計算手法が利用できる．

(B) step 2 (A) step 2 における解析の一部を流用し，各手続きにおける大域変数の参照および定義に関する関係を抽出する．

(B) step 3 情報漏洩解析における解析の前提（各入力文で代入される SC ）を入力する．

(B) step 4 (A) step 2 における解析を流用し，各プログラム文間に存在するデータ依存関係，制御依存関係から，explicit flow，implicit flow をそれぞれ抽出し，各プログラム文で出力される情報が持ちうる SC を求める．

(B) step 5 後述するデータフロー方程式を利用した情報フロー解析手法に基づいて，それぞれの手続きにたいして情報漏洩解析を行う．解析は全ての手続きの解析結果が収束するまで繰り返される．

(B) step 6 全ての出力文における SC を解析結果として表示する．

2.2.2 解析の手順

提案手法においては，step 5 において，各手続き毎に，解析の各時点で各変数の値が持ちうる SC を計算しながら各出力が出力しうる情報の SC を求める．

提案手法においては以下の点で [50] と異なる．

- 手続き間解析の効率化
  [50] では，手続きを解析する際，可能性のあるすべての引数の SC の組み合せを考慮し，それぞれの手続き内解析の結果を保持させている．
  しかし，実利用においては引数の SC の最小上界のみ考慮すればよく，実装ではそのようにしている．

- SC の簡単化，入出力ファイルの単一化
  直観的な理解を容易にするため，SC は $\{high, low\}$，入力ファイル，出力ファイルは，それぞれ標準入力，標準出力に限定している．

大域変数への対応

本提案手法においては，step 2 において，各手続きにおける大域変数の参照および定義に関する関係を抽出する．これは，後の解析において，大域変数を

- 手続き呼び出し先に対する，仮想的な引数
- 手続き呼び出し元に対する，仮想的な戻り値

として扱うことで，その解析を可能にするためである．

しかし，すべての手続き呼び出し文（手続き）に対し，大域変数の数だけの引数（戻り値）を用意するのは効率が悪い．すべての大域変数が各手続きで使われるとは限らないためである．

そこで，前もって各手続きで直接もしくは間接的に定義，参照される大域変数を調べ，必要なだけの仮想的な引数（戻り値）を用意すればよい．手続き P で直接的に定義，参照される大域変数とは，P 内で扱われる大域変数を指す．手続き P で間接的に定義，参照される大域変数とは，P を起点とする手続き呼び出し経路上に存在する，P 以外の手続きで扱われる大域変数を指す．以降，手続き間解析，手続き内解析に分けそれぞれ述べる．

手続き間解析

手続き型プログラムは複数の手続きから構成されており，手続き呼び出し経路は複数存在するのが一般的である．また，再帰経路の存在も避けられない．そのため，ある手続き

P の解析結果が，P の呼び出し元手続き P' の解析結果に（引数や大域変数を介して）影響を与える可能性があり，すべての手続きの解析結果が安定するまで，手続き呼び出し経路上に存在する手続きを繰り返し解析する必要がある．そのため，手続きをまたぐ情報フロー解析では，以下のようなものを用意し解析を行う．SCset $\mathcal{C}$，$\mathcal{S}$ はすべての手続きに 1 つずつ存在する．SCset については後述する．

解析リスト:
手続き呼び出し経路に基づく，手続きの解析順リストである．解析リストは逐次更新され，空になるとプログラム全体の解析は終了する．具体的な更新アルゴリズムは [52] で述べられている．

手続き開始時点での SCset $\mathcal{C}$:
ある手続き呼び出し文 s があったとき，対応する手続き P を解析する際に，P が保持している SCset $\mathcal{C}$ と s により渡される SCset $\mathcal{C}'$ の最小上界をとる．その結果，$\mathcal{C}$ より高ければ $\mathcal{C}$ をその値で再定義し P の解析を行う．一方，$\mathcal{C}$ と等価であれば P の解析は行わない．

手続き終了時点での SCset $\mathcal{S}$:
手続き P の解析後，P が既に保持している $\mathcal{S}$ と解析終了時点での SCset $\mathcal{S}'$ の最小上界をとる．その結果，$\mathcal{S}$ より高ければ，$\mathcal{S}$ をその値で再定義し，P を呼び出すすべての手続きを解析リストに再登録する．一方，$\mathcal{S}$ と等価であれば何もしない．

手続き内解析

はじめに，手続き内で利用される可能性のある大域変数，局所変数，仮引数をもとに，**セキュリティクラス集合**（*Security Class Set*，以降，SCset と省略する）を用意する．その要素は (変数, SC) の組である．また，各変数の SC の初期値は以下の通りである．

局所変数: low

仮引数: 対応する手続き呼び出し文の実引数の SC の上界

大域変数: 対応する手続き呼び出し文の直前での大域変数の SC の上界

その後，手続き内の先頭の文からプログラムの実行順に従い解析を行う．SCset の計算は図 2.2 に基づき，手続きの先頭の文を P_{start} とすると，ALGORITHM($P_{start}, \emptyset$) から始まる．また，文 s の解析開始時点での SCset を SCset(s)，文 s の解析終了時点での SCset を SCset(s') と表す．アルゴリズムは文 s の種類に応じて定義され，それぞれ

$$\frac{\text{s の解析時の内部処理}}{\text{s の解析終了時の SCset および 出力文の持つ SC}}$$

の形式で記述している．図 2.4 には図 2.2 で利用される要素を示している．

(assignment statement)

cl := { c | x∈Ref(s)∧(x, c)∈SCset } ∪ imp;
$kill$:= { (x, c) | x∈Def(s)∧(x, c)∈SCset };
gen := { (x, $\sqcup_{c \in cl} c$) | x∈Def(s)};

SCset := SCset $- kill \cup gen$

(input statement)

$kill$:= { (x, c) | x∈Def(s)∧(x, c)∈SCset };
gen := SCset$_{input}$(s)
(* { x | x∈SCset$_{input}$(s) } = Def(s) *)

SCset := SCset $- kill \cup gen$

(output statement)

cl := { c | x∈Ref(s)∧(x, c)∈SCset } ∪ imp

SC$_{output}$:= $\sqcup_{c \in cl} c$; SCset := SCset

(if statement)(if E then B$_{then}$ else B$_{else}$)

cl := { c | x∈Ref(E)∧(x, c)∈SCset } ∪ imp;
SCset$_{pre}$:= SCset;
ALGORITHM(B$_{then}$, $\sqcup_{c \in cl} c$); SCset$_{then}$:= SCset;
SCset := SCset$_{pre}$;
ALGORITHM(B$_{else}$, $\sqcup_{c \in cl} c$); SCset$_{else}$:= SCset;

SCset := unite(SCset$_{then}$, SCset$_{else}$)

(while statement)(while E do B)

SCset$_{pre}$:= ∅;
while SCset <> SCset$_{pre}$ **begin**
  cl := { c | x∈Ref(E)∧(x, c)∈SCset } ∪ imp;
  ALGORITHM(B, $\sqcup_{c \in cl} c$);
  SCset := unite(SCset, SCset$_{pre}$)
end

SCset := SCset$_{pre}$

図 2.2: 手続き内解析アルゴリズム:ALGORITHM(s, imp)(1/2)

(block statement)(being B_1; ... B_n; end)

ALGORITHM(B_1, $\sqcup_{c \in cl} c$);
...
ALGORITHM(B_n, $\sqcup_{c \in cl} c$)

SCset := SCset

(procedure call)

statement s calls a procedure P.

SCset$_{next}$:= $\emptyset$
for i := 0 **to** |s$_{actuals}$| **begin**
  cl := { c | (s$_{actuals}$[i], c)∈SCset };
  SCset$_{next}$:= SCset$_{next}$ ∪ { ($P_{formals}$[i], cl) };
end;
foreach x∈Ref'(P) **begin**
  SCset$_{next}$:= SCset$_{next}$ ∪
    { (x, c) | (x, c)∈SCset }
end;
SCset := SCset$_{next}$;
analysis of procedure P;
$kill$:= $\emptyset$;
for i := 0 **to** |s$_{actuals}$| **begin**
  $kill$:= $kill$ ∪
    { ($P_{formals}$[i], c) | ($P_{formals}$[i], c)∈SCset }
end

SCset := SCset − $kill$

図 2.3: 手続き内解析アルゴリズム:ALGORITHM(s, imp)(2/2)

Ref(s): 文 s で参照される変数の集合

Def(s): 文 s で定義される変数の集合

Ref'(P): 手続き P で参照される大域変数の集合

Def'(P): 手続き P で定義される大域変数の集合

s$_{actuals}$**:** 手続き呼び出し文 s の実引数の集合

$P_{formals}$**:** 手続き P の仮引数の集合

SCset: 解析時点でのセキュリティクラス集合

SCset$_{input}$**:** 入力文 s において設定される，(変数, SC) を要素とする集合

SC$_{output}$**(s):** 出力文 s が持つ SC

⊔**:** 最小上界を求める演算子

unite(A, B): セキュリティクラス集合である A と B を一つにまとめる．各変数の SC は，A，B においてその変数の SC の最小上界とする．

図 2.4: ALGORITHM 内の要素の説明

手続き内解析の例

例として，図 2.5 の関数 f に対する解析を考える．大域変数，局所変数，引数から
SCset(8) := {(a, low), (x, low), (y, low)}
を定義し．先頭の文（8 行目）から解析を行う．図 2.2 より，
$kill$:= {(y, low)}; gen := {(y, $high$)}
SCset(8') := SCset(8) $- kill \cup gen$
:= {(a, low), (x, low), (y, $high$)}
を求め，これを SCset(9) として次の文（9 行目）の解析を行う．以下同様に最後の文（18 行目）まで解析していくと，次の結果が得られる．
SCset(18') := {(a, $high$), (x, low), (y, $high$), (f, $high$)}

2.2.3 提案手法の実現について

本節では，PDG 構築ルーチンを利用した情報漏洩解析手法の実現について述べる．ツールの実現は，我々が開発したスライスツールである *Osaka Slicing System*（，以下，*OSS*）[52] に，本節で述べた情報漏洩解析部を機能追加する形で行った．

```
1:  program sample;
2:  var a : integer;
3:  . . .
4:  function f(x : integer) : integer;
5:  (* 解析時の条件は a,x ← low と仮定 *)
6:  var y : integer;
7:  begin
8:      readln(y); (* ← high *)
9:      if a > 0 then
10:     begin
11:         a := y + 1;
12:         y := x − 1;
13:     end;
14:
15:     writeln(y);
16:     writeln(x);
17:
18:     f := y;
19:     end
20: . . .
21: end.
```

図 2.5: 手続き内解析の例

ツールの概要

ツールの解析の流れを図 2.6 に示す.

構文・意味解析部は，UI 部からの要求に応じて構文解析，意味解析を行う（図 2.6-1）. 次に，ユーザはソースファイル上で情報漏洩解析の前提条件を設定（図 2.6-2）し，UI 部を通じてセキュリティ解析部へ依頼する．情報漏洩解析部は，前提条件をもとに情報漏洩解析を行い（図 2.6-3），その結果を UI 部に渡す．UI 部は，SC の高い変数が使われる可能性のある文を強調する形で解析結果を表示する（図 2.6-4）.

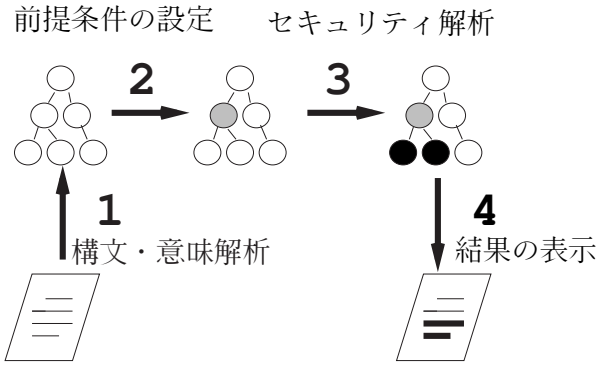


図 2.6: PDG の構築ルーチンを利用した手法の解析手順

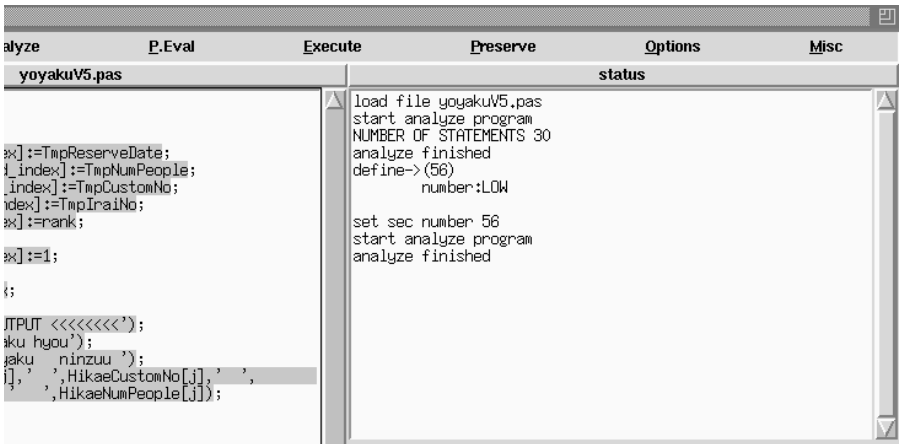


図 2.7: 変更前の予約システムの解析結果

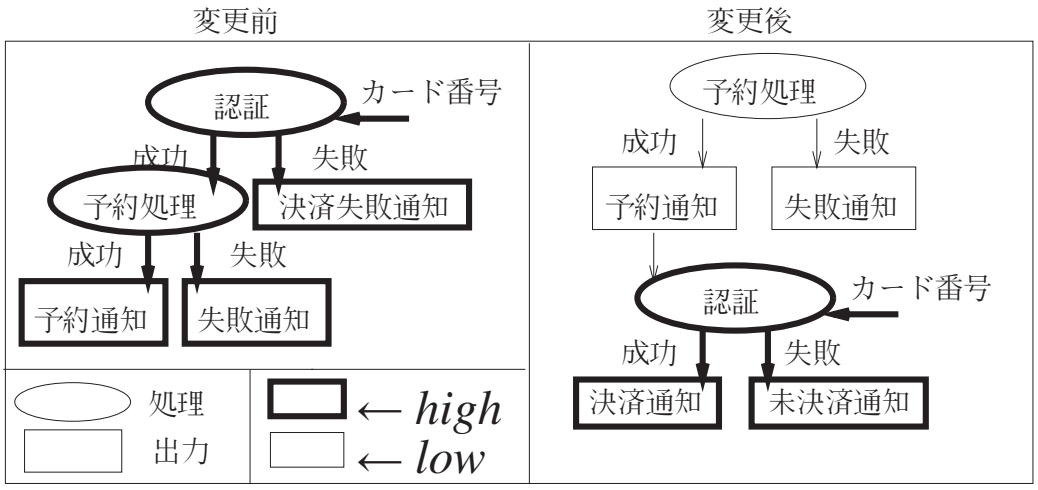


図 2.8: 予約システムの制御の流れ

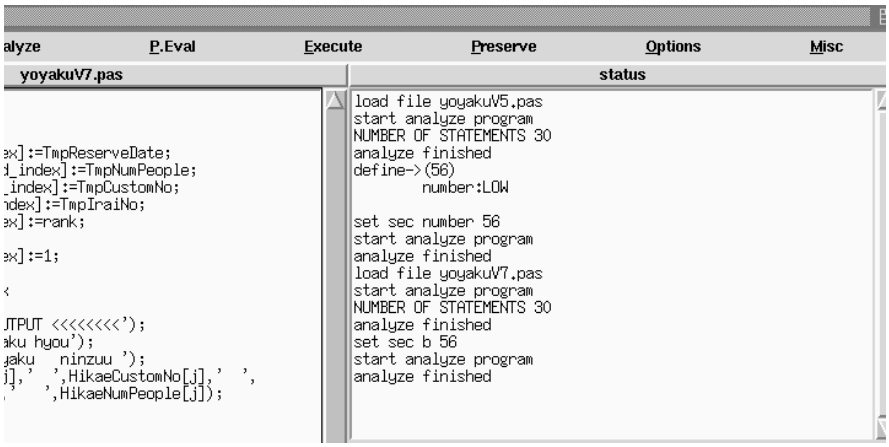


図 2.9: 変更後の予約システムの解析結果

ツールの適用事例

我々は本ツールの利用目的として，プログラムの安全性の確認を考えている．プログラムの安全性の確認とは，プログラムを静的に解析し SC の高い出力を事前に検出することで，予想外の情報漏洩を防ぐことをいう．プログラムの安全性を確認し対策を立てることで，SC の高い出力文を減らし，情報漏洩の可能性をより低くすることができる．

適用事例として，飛行機の予約システムを考える．このシステムにはクレジットカードの認証を行うモジュールが組み込まれている．

クレジットカード番号に関する情報にのみ高い SC を与えて解析を行なった結果，システム全体で 36 個ある出力文のうち，35 個の出力文が高い SC を持つという結果が得られた（図 2.7）．これらの出力文中には，予約処理モジュールの出力文も含まれていた．これは，このシステムが図 2.8 のように認証が成功した後に予約を行うよう実装されていたためである．このような実装では,「予約処理が行なわれる」ことが「与えられたクレジットカード番号が認証された」ことを示しており，クレジットカードの番号に関する情報が漏洩していることになる．

そこで，クレジットカード番号と予約処理モジュールの出力文に存在する情報フローを排除するため，認証前に予約処理を行う方法に変更した（図 2.8）．

ツールを用いて前回と同じ条件で解析を行なった結果，高い SC を持つ出力文は，クレジットカード認証に関する 13 個のみに減少させることができた（図 2.9）．また，予約処理モジュールの出力文が高い SC を持たないことも確認できた．このように，実装の方法によって情報の流れは大きく変わるため，情報フロー解析による安全性の確認は重要である．

2.3 PDG を利用した情報漏洩解析手法の提案

前節において，PDG の構築ルーチンを利用することで，各文内・文間における情報フローをデータフロー方程式で表現し，文実行前後において各変数の SC 間で成り立つべき関係式を定義し，繰り返し計算によって求める手法を提案した．しかし，この手法は，解析アルゴリズムを PDG の構築ルーチンから流用できるものの，言語ごとに定める必要があるため手法としての汎用性がまだ十分ではない．また，プログラムの実行順に従い繰り返し計算を行っているため，階層化しその中で判定される変数の数が多いプログラムに対しては，解析の繰り返しの条件となる変数が多くなることで繰り返し回数が増加し，効率が落ちる．

今節では改良手法として，PDG を探索することにより情報漏洩解析を行う手法を提案する．PDG によって表されるプログラム内部の依存関係は，情報フローの場合と同じようにデータ依存・制御依存の 2 種類に分類でき，それぞれの性質も情報フローの場合と非常に類似している．そこで，データ依存辺と明示的フロー，制御依存辺と暗示的フローを対応させ，アルゴリズム中のデータ依存辺を構築する部分を，明示的フローの計算部に，制御依存辺を構築する部分を暗示的フローの計算部に置き換え，解析を行なっている．

改良手法により，PDG の言語独立性を利用することによる言語独立な情報漏洩解析が可能となり，再解析における時間的コストの減少が可能となる．また，束構造を持つセキュリティモデルに対する情報漏洩解析を行うために，ユーザがあらかじめセキュリティモデルの束を記述し，それを参照して SC の和演算を行う仕組みを実現する．

2.3.1 情報漏洩解析のためのプログラムスライス計算手順の変更

図 2.1 の (A) のプログラムスライスの計算の手順を以下のように変えることで，PDG を利用した情報漏洩解析を行うことができる．(C) step 4，5，6 については後述する．

(C) step 1 構文解析，意味解析を行い，各文において参照される変数，定義される変数を決定する．

(C) step 2 各プログラム文間に存在するデータ依存関係および制御依存関係を，**データフロー方程式**（*Data Flow Equation*）による計算から求める．

(C) step 3 Program Dependence Graph (PDG) を抽出した依存関係から構築する．

(C) step 4 解析対象となるプログラムの 2 つの前提条件を入力する．

(C) step 5 PDG の探索をしながら，経路上の各頂点で SC の和演算を行う．

(C) step 6 各出力文における SC を情報漏洩解析の結果として出力する．

2.3.2 解析の手順

ここでは PDG を利用した，束構造のセキュリティモデルに対する情報漏洩解析の手順について説明する．初期状態として，全ての頂点における SC の初期値は low とする．

step 4: 前提条件の入力
解析対象となるプログラムの 2 つの前提条件を入力する．

- セキュリティモデル:
  二次元行列の形でセキュリティモデルの定義を入力する．
- 各入力文で読み込まれる値の SC:
  各入力文で読み込まれる値の SC を入力する．

step 5: 入力文を起点とした PDG の探索
PDG の探索をしながら，経路上の各頂点で SC の和演算を行う．

解析対象のプログラム中の各入力文を起点として，PDG の探索を行う．探索は，頂点から出るデータ依存辺および制御依存辺を順方向にたどる．変数 v の値が読み込まれる入力文を起点とした場合，探索で到着した頂点において，その頂点の現在の SC と $SC(v)$ の和演算を行い，頂点を演算結果の SC で更新する．その頂点が和演算を実行して更新されない場合，その頂点からの辺はたどらない．一方，更新される場合，その頂点からの辺をたどり次の頂点へ移る．SC が更新される頂点がなくなった場合，次の入力文を起点として PDG 探索を行い，全ての入力文に対して PDG の探索を行う．

step 6: 入力文を起点とした PDG の探索

各出力文において，どの SC を持つ値を出力しうるかを表示する．

解析の例

提案した手法により情報漏洩解析を行なった時の解析例を図 2.10 に示す．図 2.10-1 が前提条件を入力した直後の PDG である．SC の束構造は図の右下に示す．入力文を表す頂点にセットされた SC は頂点の色の濃さで示す．起点とした入力文に対応する頂点からの SC が辺を順方向にたどって PDG の各頂点の SC を更新していく．図 2.10-11 が最終的な結果となる．

2.3.3 提案手法の実現について

PDG を利用した情報漏洩解析手法の実現を，我々が開発したスライスツールである *Osaka Slicing System*，*OSS*[52] に情報漏洩解析部を機能追加する形で行った．入力ファイル，出力ファイルはそれぞれ標準入力，標準出力に限定している．また，SC 制約機能を追加し，一般的なセキュリティモデルに対応した．それぞれについて説明する．

SC 制約機能の追加

情報漏洩解析の問題点として，SC の高いデータに対してプログラム中で暗号化などを行って，もとのデータが隠蔽された場合でも，SC は高いままとなってしまうためことが考えられる．実際に運用する際には，暗号化などが信頼できるとユーザが判断したときには，プログラム中の関数による出力の SC を，ユーザが任意に設定できる機能が必要である．そこで今回の実装では，関数の戻り値の SC をユーザが任意に設定できる機構を実現した．Phase 3: 前提条件の設定時の際，関数の戻り値の SC を設定することで，その関数の戻り値を表す頂点から辺をたどる際に，その頂点で設定した SC をもとに，PDG を探索することができる．この機能により，SC の高いデータが関数内で隠蔽され，現実的にはその隠蔽されたデータからもとのデータを類推することが不可能であるとき，その値の SC を強制的に低くすることで，現実的な安全度に則した情報漏洩解析を行うことができる．

一般的なセキュリティモデルへの対応

束の任意の二元の最小上界を二次元行列を用いて表すことで，束構造を持つセキュリティモデルへの情報漏洩解析に対応する．

図 2.11-(a) のようなハッセ図で表される束構造を考える．low は 0，$high$ は 5 と対応している．このとき，和演算結果を格納した二次元行列を用いることで，図 2.11-(a) のハッセ図で表される束構造は，図 2.11-(b) の二次元行列として表現できる．二次元行列の m 行 n 列の要素が $SC(m) \vee SC(n)$ の結果を表す．

また $SC(a) \vee SC(b)$ の結果が $SC(c)$ であるとき，

- $SC(a) = SC(b) = SC(c)$ ならば $SC(a) = SC(b)$
- $SC(a) = SC(c), SC(b) \neq SC(c)$ ならば
  $SC(a) > SC(b)$

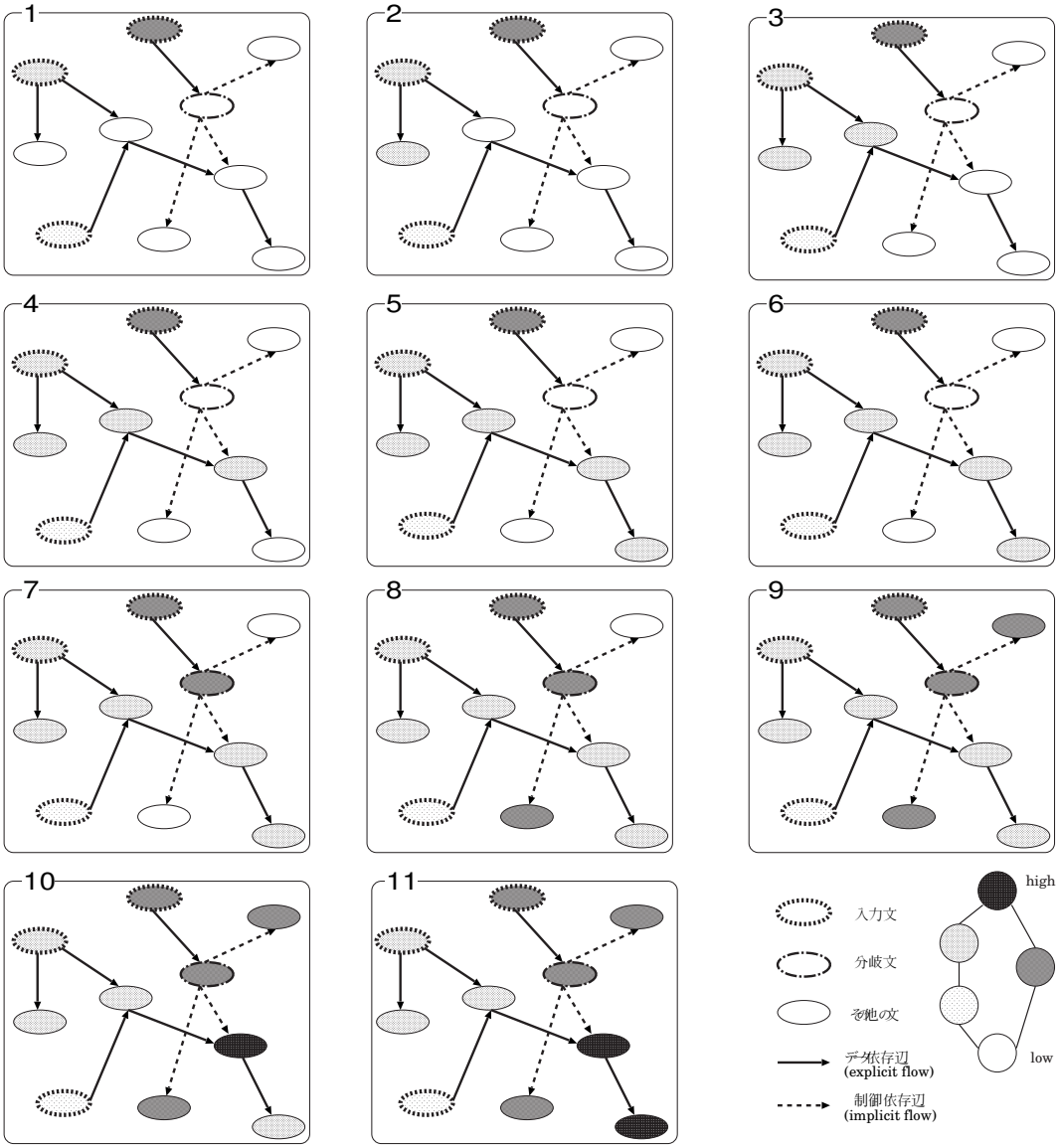
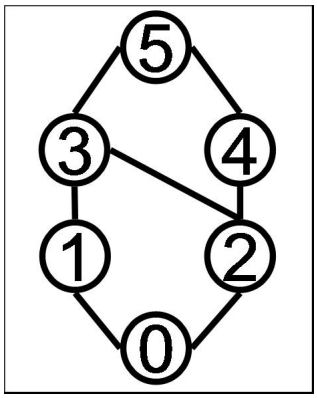


図 2.10: PDG を利用した解析の例



$low = 0, high = 5$

(a) ハッセ図

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	3	3	5	5
2	2	3	2	3	4	5
3	3	3	3	3	5	5
4	4	5	4	5	4	5
5	5	5	5	5	5	5

(b) 束構造の和演算を表す二次元行列

図 2.11: 束構造をもつ一般的なセキュリティモデルの表現方法

- $SC(b) = SC(c), SC(a) \neq SC(c)$ ならば
  $SC(b) > SC(a)$
- $SC(a) \neq SC(c), SC(b) \neq SC(c)$ ならば
  $SC(a), SC(b)$ 間に大小関係なし

というような関係が成り立っているため，大小関係の比較も二次元行列を用いて行うことができる．

ツールの概要

ツールの解析の流れを図 2.6 に示す.

構文・意味解析部は，UI 部からの要求に応じて構文解析，意味解析を行う（図 2.12-1）. 次に，ユーザはソースファイル上で情報漏洩解析の前提条件を設定（図 2.12-2）し，UI 部を通じて情報漏洩解析部へ依頼する．情報漏洩解析部は，前提条件をもとに解析を行ない（図 2.12-3）その結果を UI 部に渡す．UI 部は，各々の文の SC に応じて背景色を変更することで SC を表示する（図 2.12-4）.

ツールの適用事例

次に複雑なセキュリティモデルを持つプログラムの安全性の確認に本システムを適用する．プログラムの安全性の確認とは，プログラムを静的に解析し SC の高い出力を事前に検出することで，予想外の情報漏洩を防ぐことである．プログラムの安全性を確認し対策を立てることで，SC の高い出力文を減らし，情報漏洩の可能性をより低くすることができる．

適用対象として学生の成績管理システムを考えた．成績は一般教養科目分野と，専門科目分野に分類され，それぞれ「可」または「不可」のいずれかとなる．全ての学生は自身

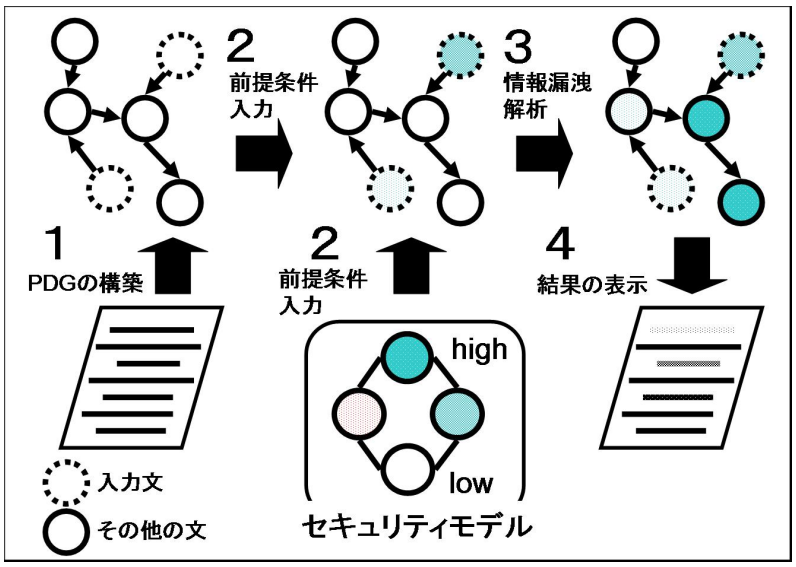


図 2.12: PDG を利用した手法における解析の手順

ユーザ	ユーザが参照できるデータ
学生	自身の一般教養科目分野の成績 自身の専門科目分野の成績 自身の認証番号
一般教養科目分野の 事務員	全ての学生の一般教養科目分野の成績 一般教養科目分野の事務員の認証番号
専門科目分野の 事務員	全ての学生の専門科目分野の成績 専門科目分野の事務員の認証番号

表 2.2: 成績管理システムで各ユーザが参照できるデータ

の成績を，分野に限らず参照できる．また，一般教養科目分野，専門科目分野にはそれぞれ事務員が居り，一般教養科目分野の事務員は全ての学生の一般教養科目の成績の参照と書き換えを行うことができるが，専門科目分野の成績を参照，書き換えすることはできない．同様に，専門科目分野の事務員は，全ての学生の専門科目分野の成績の参照と書き換えを行うことができるが，一般教養科目分野の成績を参照，書き換えすることはできない．

学生，一般教養科目分野の事務員，専門科目分野の事務員はそれぞれ認証番号をもっており，成績管理システムにアクセスする際にそれぞれの認証番号を入力する．入力された認証番号によりシステムはユーザが何者かを判断し，それぞれのユーザに応じた権限を与える．そのユーザは以後その権限に応じた操作を行うことができる．

各ユーザの参照できるデータを表 2.2 に整理する．

このシステムにおけるセキュリティモデルは図 2.13 のハッセ図のようになる．

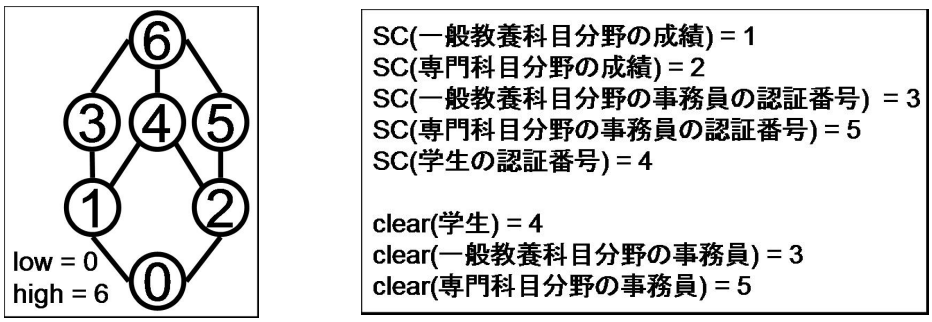


図 2.13: 成績管理システムのセキュリティモデル

図 2.13 をもとに，それぞれのデータの SC と，ユーザのクリアランスの SC に対するマッピングを考える．主要なデータとその SC は，

- SC(一般教養科目分野の成績) $= 1$
- SC(専門科目分野の成績) $= 2$
- SC(一般教養科目分野の事務員の認証番号) $= 3$
- SC(専門科目分野の事務員の認証番号) $= 5$
- SC(学生の認証番号) $= 4$

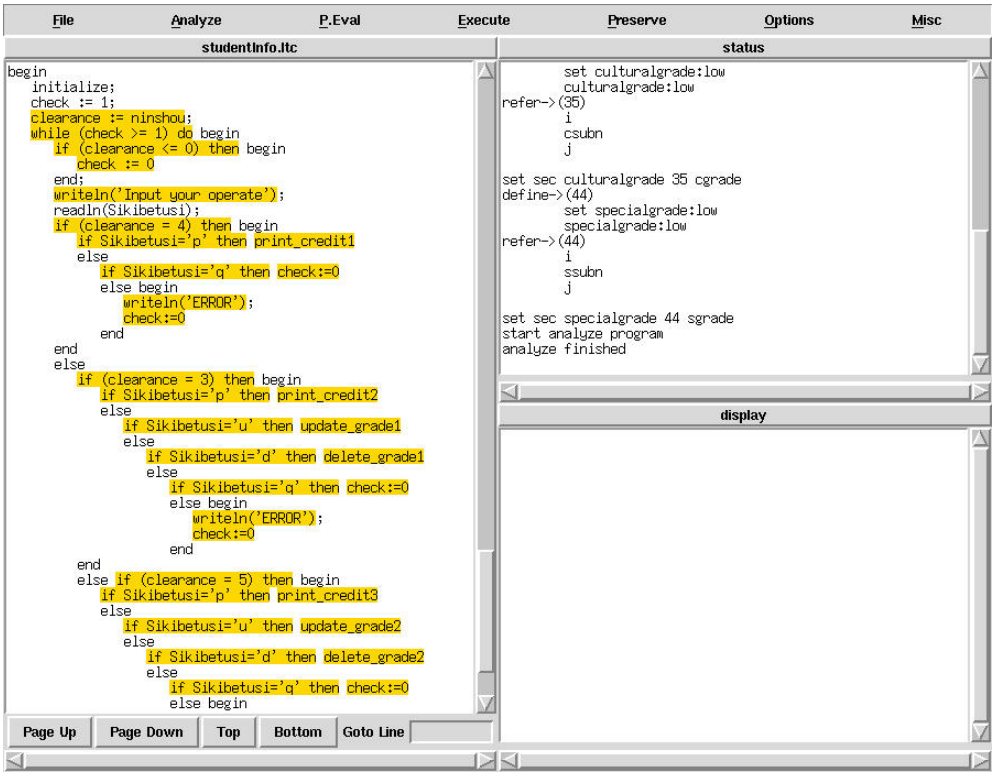
同様にユーザとそのクリアランスは，

- $clear$(学生) $= 4$
- $clear$(一般教養科目分野の事務員) $= 3$
- $clear$(専門科目分野の事務員) $= 5$

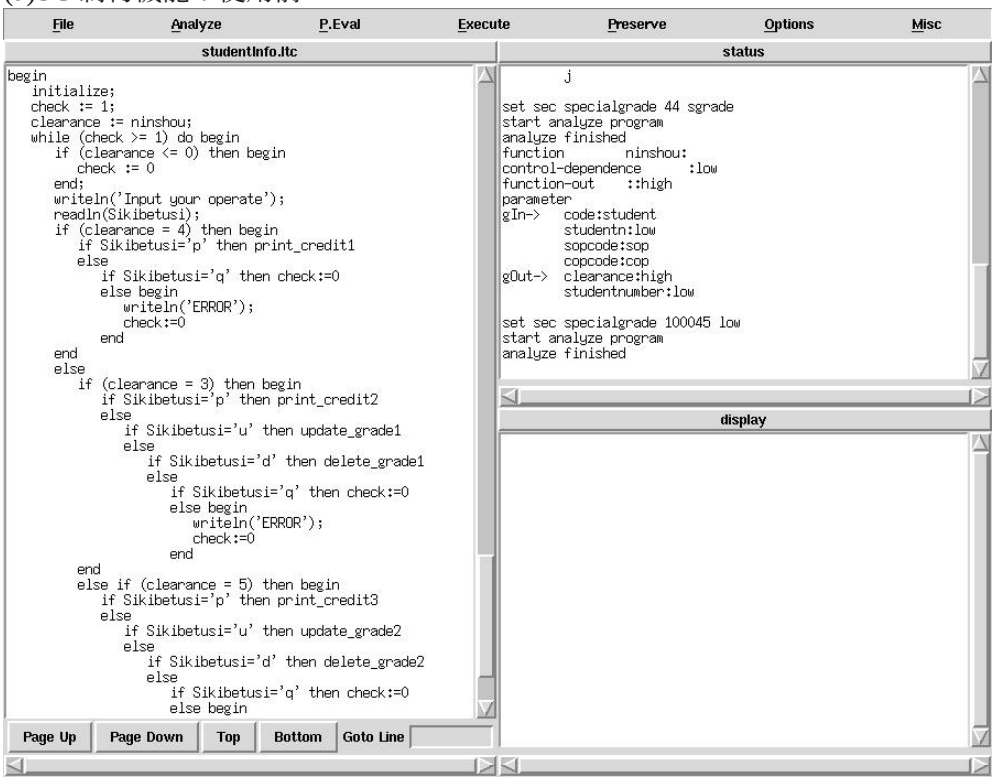
となる．

サンプルプログラムの各々のデータに対して上で述べた SC を与えて解析を行なった結果，33 個の出力文のうち，SC が 4 の出力文が 1 個，5 の出力文が 1 個，6 の出力文が 26 個という結果が得られた．（図 2.14-(a)）

これは，各ユーザの認証番号が様々な暗示的フローを引き起こし，そのためユーザの認証を行う関数が現在のユーザのクリアランスを判別した結果として返す値の SC が高くなり，その値が引き起こす暗示的フローを受ける文が多数あったために起こった．しかし，各ユーザの認証番号や，関数が返す値が十分に隠蔽されたときを想定して，認証関数の戻り値の SC を low に設定して再度解析を行うと，（図 2.14-(b)）33 個の出力文のうち，SC が 0 の出力文が 27 個，1 の出力文が 2 個，2 の出力文が 2 個，4 の出力文が 1 個，5 の出力文が 1 個という結果が得られた．このことから，適切なデータに隠蔽を行なえば，現実的に情報漏洩を考慮しなくてはならない部分の推定が容易になることがわかる．このように，情報漏洩解析と SC 制約機能を組み合わせることで，どのデータに隠蔽を施すべきかを判断する指針にすることができる．



(a)SC 制約機能の使用前



(b)SC 制約機能の使用後

図 2.14: 成績管理システムにおける解析結果

2.4　2章のまとめ

本章では，[50] で提案されたセキュリティモデルに基づく情報フロー解析アルゴリズムの実装を行い，手法の有効性を検証した．実装においては，プログラムスライスにおけるPDG 構築ルーチンを流用することで，より現実的な言語を対象とした情報漏洩解析手法を実現した．さらに，PDG を利用した，束構造を持つセキュリティモデルに対する情報漏洩解析を行う手法を提案し，実現した．本手法では，プログラムスライスなどに利用される，依存関係を示した PDG を使用することで，再利用性向上による解析コストの低下と，言語に依存しない情報漏洩解析を行うことができた．また，行列を用いて演算を表現することで，束構造を持つセキュリティモデルに対する情報漏洩解析に対応している．

また，提案手法を既存のスライシングツールに機能追加の形で実装し，適用事例を挙げた．適用事例では，同じようなプログラムであっても情報フローが異なるために機密度の高い出力文の数に大きな違いが出ることから，情報漏洩解析によるプログラムの安全性の確認が重要であることを確認した．また，一つの関数の返り値を隠蔽するだけで，機密度の高い出力文が大きく減少することから，関数の返り値の機密度を変更できる機能が情報隠蔽を考慮すべき部分の推定に利用でき，より現実的に情報漏洩解析が行えることを確認した．これにより，情報漏洩解析を用いることで，保守作業において改変前の注意を払うべき部分の推定や，改変後の安全性の確認が可能になると考えられる．

第3章　オブジェクト指向プログラムの変更作業を支援する影響波及解析システム

3.1　導入

ソフトウェアに加えられた変更による影響を受ける部分を識別し，回帰テストでのテストケースを選択するための手法として，影響波及解析が提案されている．我々はプログラム理解，保守といったより広い範囲でも影響波及解析が利用できると考えているが，従来の影響波及解析手法は回帰テストにおけるテストケース選択用に特化した形で実現されており，被影響部分の探索ルールもテストケース選択用に特化して定義されていた．しかし，プログラム理解，保守などのより広い範囲での影響波及解析の利用を考えた場合，影響の定義は利用目的に応じて異なる．

例えば，回帰テストにおけるテストケースの選択 [37] を目的とした場合には，変更または削除されたメソッドをテストするテストケース，およびオーバーライド関係の変化したメソッドへの呼び出し経路をテストするテストケースの検出が必要となる．一方で，修正個所の特定を目的としてメソッドオーバーライドの変化による影響を求める場合には，オーバーライド関係の変化したメソッドを直接呼び出してる部分を検出できれば十分である．そのため，ユーザの目的に応じて被影響部分の探索ルールが選択可能な解析手法が必要である．

また，現在のソフトウェア開発環境ではオブジェクト指向言語が多く利用されている．オブジェクト指向プログラムでは，従来の手続き型プログラムに比べ，変更箇所以外に影響を及ぼすような変更が数多く存在する．文献 [12, 25] では，オブジェクト指向プログラムにおける変更は，メソッドの**オーバーライド**（*Override*），フィールドの**隠蔽**（*Hide*）といったオブジェクト指向特有の概念により様々な影響が引き起こされることが述べられている．メンバを解析の単位とした影響波及解析手法では，これらの独自概念は考慮されているが満足な実装がなされていない [22, 26]．そのため，オブジェクト指向言語の独自概念を考慮した解析手法およびその実装が求められている．

本章では，JAVA[15] を対象言語とした影響波及解析手法の提案を行う．提案手法では，被影響部分の探索ルールをユーザの目的に応じて選択可能にすることで，回帰テストだけではなく，プログラム理解，保守といった，より広い範囲での影響波及解析の利用を実現している．さらに，クラスのメンバ（メソッド，フィールド）間の関係を表現する 2 種類のグラフを利用することで，メソッドのオーバーライド，フィールドの隠蔽を考慮した影響波及解析を実現する．また，提案手法を JAVA 影響波及解析システムとして実装し，適用事例をもとに本システムを用いることで修正個所の特定が効果的に行えるかを確認する．

以降，3.2 節では提案する影響波及解析手法について説明する．3.3 節で JAVA 影響波及解析システムについて述べ，3.4 節において，実際のソフトウェア開発履歴に適用した事

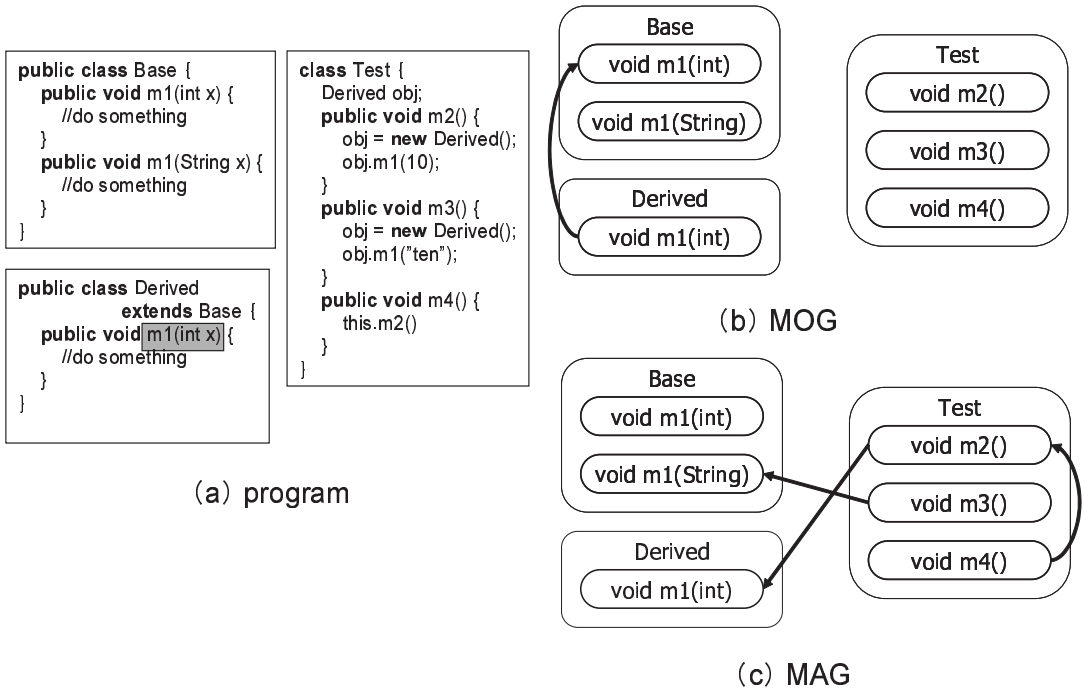


図 3.1: メンバオーバーライドグラフ（MOG) とメンバアクセスグラフ（MAG）

例を適用実験として紹介する．

3.2 MOG，MAG を用いた影響波及解析手法の提案

本節では，クラスのメンバ間の関係を表現する二つのグラフを利用した影響波及解析手法の提案を行う．提案手法により，メソッドのオーバーライド，フィールドの隠蔽を考慮した影響波及解析の実現，およびユーザの様々な目的に対応可能な影響の定義を行うことができる．

影響は，オーバーライドや呼び出し経路などに変化が生じたメンバから発生し，呼び出し経路を通じて波及するものである．回帰テストに利用されてきた既存の影響波及解析は，呼び出し経路を逆にたどる，つまり一部の呼び出し経路に限定した特殊な波及を考えていたとみなすことができる [37].

我々は，より一般的な影響波及解析の枠組，具体的には，メソッドのオーバーライドやフィールドの隠蔽などによって生じる様々な種類の影響の発生および波及のパターンを組み合わせできる枠組を実現するため，グラフを用いてオーバーライド関係および呼び出し関係を表現する．影響の発生はグラフの変化に，影響の波及はグラフ探索にそれぞれ置き換えることができる．

本論文では，これらを実現するために，**メンバオーバーライドグラフ**（*Member Override Graph*，*MOG*）および**メンバアクセスグラフ**（*Member Access Graph*，*MAG*）を利用した手法を提案する．MOG とは，メソッドオーバーライド，抽象メソッドの実装，フィール

ドの隠蔽などのメンバ間のオーバーライド関係をグラフで表現したもので，継承により親子関係となるクラスのメンバ間に存在する．従来の手法 [26, 22] では，グラフを構築する際に考慮されているのはクラスの継承までで，メンバ間の関係においては抽出アルゴリズム側で考慮していた．それに対して MOG では，グラフ構築時にクラスの継承に関する情報からあらかじめメンバ間の関係を求め，辺として表現している．

MAG とは，メソッドの呼び出し，フィールドの参照などのメンバ間のアクセス関係をグラフで表現したもので．クラスのメンバ間に存在する．

3.2.1 メンバオーバーライドグラフ（MOG）

ここでは，MOG の構成要素である MOG 節点および MOG 辺の定義，MOG の構築方法について述べる．図 3.1(b) は，図 3.1(a) のプログラムから構築される MOG である．

MOG 節点は，各クラスの各メソッドに対応した **MOG メソッド節点**と，各フィールドに対応した **MOG フィールド節点**の二種類から成る．静的初期化子およびコンストラクタはメンバではないが，メンバと同様のアクセス関係を持つことから，これらも MOG 節点として扱う．MOG 節点は後述する MAG 節点と一対一で対応する．

MOG 辺は，2 つの MOG 節点間のオーバライド関係を有向辺で表現したものであり，メソッドオーバーライドを表す **MOG オーバーライド辺**，抽象メソッドの実装を表現する **MOG 実装辺**，フィールドの隠蔽を表現する **MOG 隠蔽辺**の三種類から成る．各辺は，オーバーライドするメンバからオーバーライドされるメンバへ引かれる．図 3.1(b) では，MOG メソッド節点 `void Derived.m1(int)` から，`void Base.m1(int)` へ MOG オーバーライド辺が引かれている．MOG は，各クラスのメソッド，フィールド宣言の解析による MOG 節点の抽出，およびクラスの継承関係の解析による MOG 辺の抽出により構築される．

3.2.2 メンバアクセスグラフ（MAG）

ここでは，MAG の構成要素である MAG 節点および MAG 辺の定義，MAG の構築方法について述べる．図 3.1(c) は，図 3.1(a) のプログラムから構築される MAG である．

MAG 節点は，MOG 節点と同様に，**MAG メソッド節点**，**MAG フィールド節点**から成る．MAG 節点は MOG 節点と一対一で対応する．

MAG 辺は，2 つの MAG 節点間のアクセス関係を有向辺で表現したものであり，メソッド呼び出しを表す **MAG 呼び出し辺**，フィールドの参照を表す **MAG 参照辺**の二種類から成る．各辺は，アクセスするメンバからアクセスされるメンバへ引かれ，2 つの節点間に複数の辺が存在する場合もある．

MAG の構築時には，MAG 節点を抽出した上で，メソッド呼び出し式，フィールド参照式を解析し，各フィールドがどのクラスのオブジェクトを参照しうるかをあらかじめ解析した上で，各文において実際に参照しうるメンバの集合を求めることで MAG 辺の抽出を行う．このとき，参照変数が指すインスタンスの型が特定できないことにより，アクセスされるメンバが一意に決まらない場合がある．その際には，その参照変数の型から派生したクラスに存在する，同一シグニチャのすべてのメンバに対して辺を引く．

表 3.1: 直接被影響節点

直接被影響節点	概要
D-E1: 影響の発生元の節点	プログラム変更に対応する節点（発生，消失した節点も含む）. （図 3.2 の MOG 節点 {M8} および MAG 節点の {M8}）
D-E2: 辺の発生先の節点	プログラム変更により発生した辺の終節点. （図 3.2 の MOG 節点 {M7} および MAG 節点 {M6，M8}）
D-E3: 辺の発生元の節点	プログラム変更により発生した辺の始節点. （図 3.2 の MOG 節点 {M8} および MAG 節点 {M3，M4}）
D-E4: 辺の消失先の節点	プログラム変更により消失した辺の終節点. （図 3.2 の MOG 節点 {M6} および MAG 節点 {M7，M8}）
D-E5: 辺の消失元の節点	プログラム変更により消失した辺の始節点. （図 3.2 の MOG 節点 {M8} および MAG 節点 {M3，M4}）

表 3.2: 間接被影響節点

間接被影響節点	概要
I-E1: 順方向の間接影響波及節点	直接被影響節点から有向辺に従い到達可能な節点. 図 3.3 の MAG 節点 {M9，M10，F1, F2}
I-E2: 逆方向の間接影響波及節点	直接被影響節点から有向辺の逆向きに従い到達可能な節点. 図 3.3 の MOG 節点 {M10} および MAG 節点 {M1，M3，M4}

3.2.3 MOG，MAG を用いた影響波及解析

変更によって何らかの変化が生じた MOG 節点，MAG 節点を検出し，その節点から MOG 辺，MAG 辺をたどることで，被影響部分の抽出を行う．さらに，探索ルールを変更可能とすることにより，ユーザの様々な目的に対応することができる．

被影響部分の分類

提案する影響波及解析により抽出される被影響部分の単位は，プログラム上ではメンバ，MOG，MAG 上では節点となる．本論文では，これらをそれぞれ**被影響メンバ**（*Affected Member*），**被影響節点**（*Affected Node*）と呼ぶ.

ユーザの目的に応じた被影響メンバの抽出を行うため，被影響部分を表す節点を直接的な影響を表す**直接被影響節点**(*Direct Affected Node*) と間接的な影響を表す**間接被影響節点**(*Indirect Affected Node*) に分類する.

直接被影響節点は変更が行われた節点に基づいて決定されるもので，表 3.1 に挙げる 5

表 3.3: 探索ルールの例

探索ルール	探索対象となる被影響節点	
	MOG	MAG
R1: アクセス変化メンバ抽出	ϕ	{D-{E1,E3,E5}, I-E2}
R2: 関係変化メンバ抽出	{D-{E1, E2, E3, E4, E5}}	{D-{E1, E2, E3, E4, E5}}
R3: 間接アクセスメンバ抽出	ϕ	{D-E1, I-{E1, E2}}

種類がある．例えば，メソッドの内容を変更した場合，表 3.1 の D-E1 から，変更されたメソッドに対応する MAG および MOG 節点が直接影響節点に含まれる．また，変更前後の MAG における辺を比較してアクセス関係に変更（追加・削除・更新）があった場合，D-E2 ～E5 に対応する MAG 節点が直接影響節点に含まれる．図 3.2 は，変更が行われた節点が M8 であるときの直接被影響節点の例を示している．

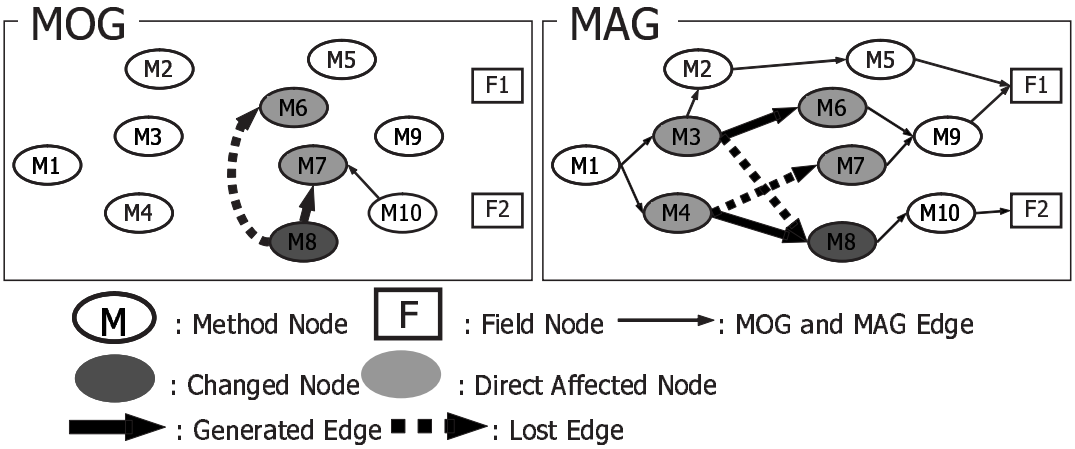


図 3.2: 直接被影響節点の例（表 3.1 参照）

間接被影響節点は表 3.2 に挙げる 2 種類があり，直接的な変更によって変化した値を参照する節点などの直接被影響節点から推移的に影響を受けうる節点を指す．そのため，変更によって直接影響を受けた節点に関係のあるメンバが全て含まれることになり，多くのメンバが間接被影響節点として抽出される．間接被影響節点によって，変更箇所で利用されるメンバの限定および，従来手法のテストケースの限定時において想定されていた，変更により実行結果が変わりうる場所の限定を行うことができる．図 3.3 は，MOG の直接被影響節点が M7，M8，MAG の直接被影響節点が M6，M8 であるときの間接被影響節点の例をそれぞれ示している．

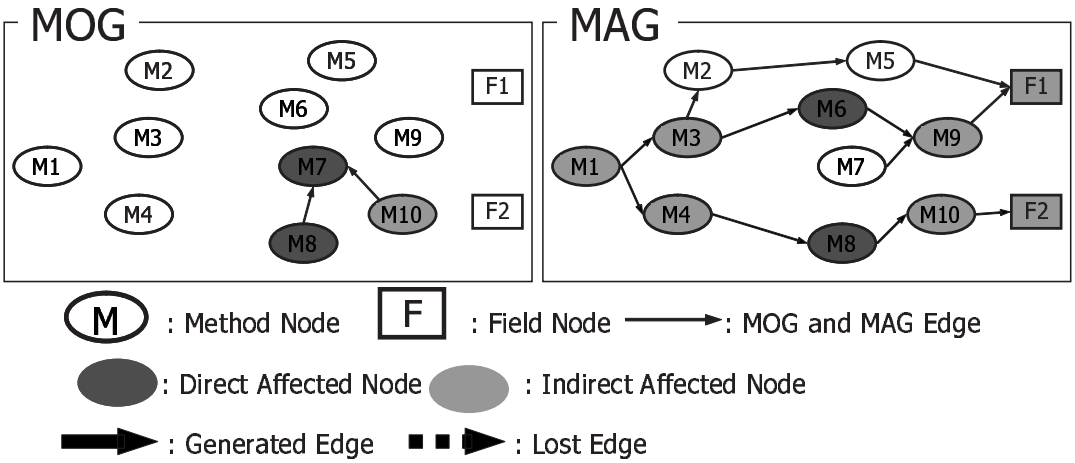


図 3.3: 間接被影響節点の例（表 3.2 参照）

被影響部分の抽出

提案手法では，前節で定義した各被影響節点の抽出の有無を組み合わせることにより，ユーザの目的に応じた被影響メンバの抽出が可能となる．組み合わせは数多く存在するが，ここでは代表的な 3 つの探索ルールについて述べる．なお，各ルールに対応する被影響節点の組み合わせの一覧を表 3.3 に示す．

R1: アクセス発生メンバ抽出

変更により発生した新たな実行経路上に存在する（すなわち新たにプログラムの実行結果に影響を及ぼす可能性が生じた）部分を抽出する．既存の影響波及解析手法が対象としている，回帰テストでの利用を目的としたものである．テストケース選択に用いる場合は，被影響節点の中でテストケースに対応するものだけを選択すればよい．図 3.4 に，M8 が変更メンバである場合の抽出例を示す．

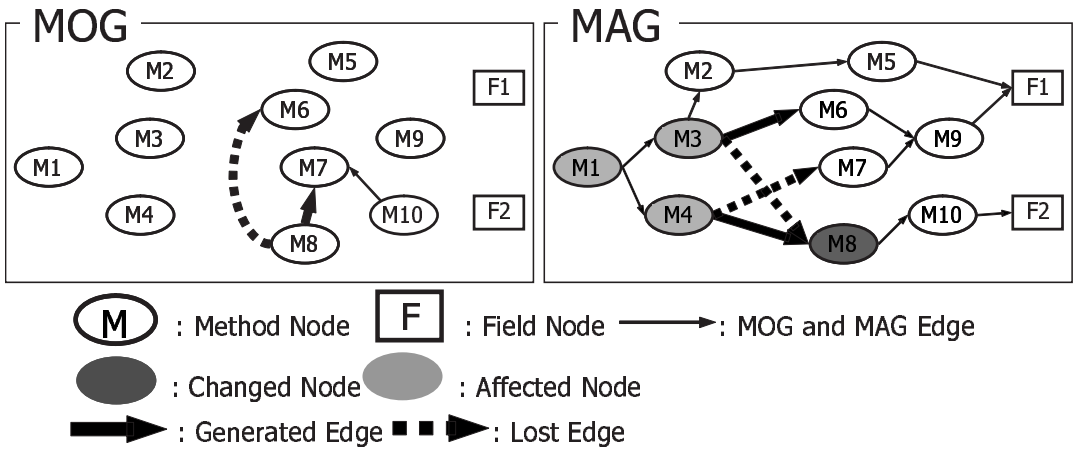


図 3.4: 影響探索ルール R1: アクセス発生メンバ抽出

R2: 関係変化メンバ抽出

オーバーライド関係の変化，およびそれに伴うアクセス関係の変化が発生するメンバを全て抽出する．プログラム変更によるメンバ間の関係の変化を把握し，変更に対応すべき修正個所を識別するために有効である．図 3.5 に，M8 が変更メンバである場合の抽出例を示す．

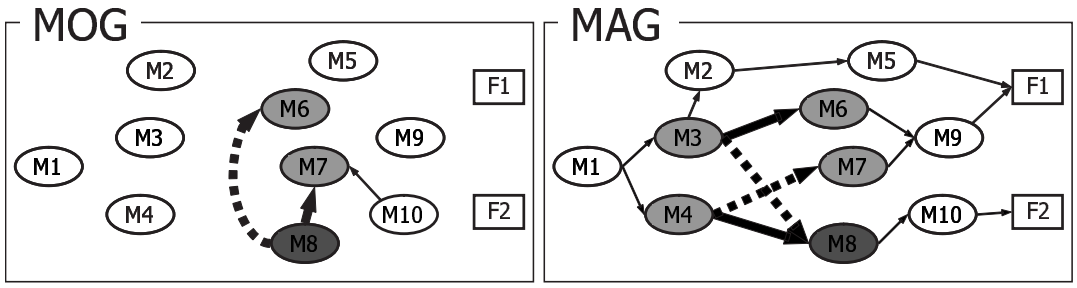


図 3.5: 影響探索ルール R2: 関係変化メンバ抽出（図 3.4 参照）

R3: 間接アクセスメンバ抽出

変更メンバに直接的および間接的にアクセスする可能性のあるメンバ，変更メンバが直接的および間接的にアクセスする可能性のあるメンバをすべて抽出する．メソッド本体やフィールドの初期値などを変更する場合，アクセス関係に変化は無いが，それらを使用するメンバの実行結果などが変化する可能性があるため，このルールによる被影響メンバの把握が有効となる．図 3.6 に，M8 が変更メンバである場合の抽出例を示す．

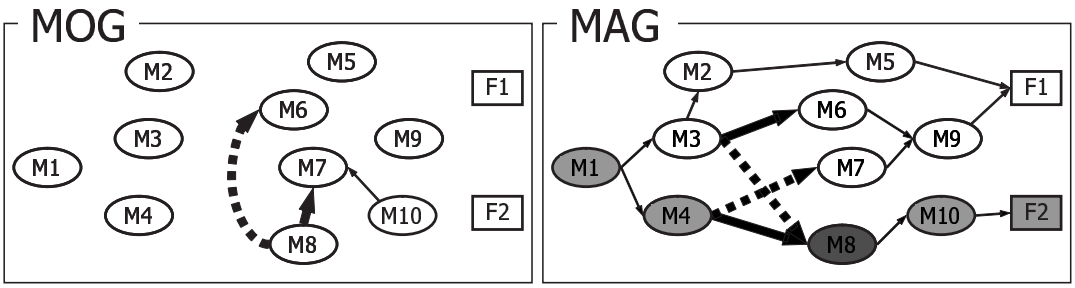


図 3.6: 影響探索ルール R3: 間接アクセスメンバ抽出（図 3.4 参照）

3.3 JAVA 影響波及解析システム

本節では，提案手法の実装である JAVA 影響波及解析システムの構成，およびその機能について述べる．

3.3.1 概要

本システムは，JAVA プログラムにおける，メンバ単位の変更で生じる影響を解析，表示するシステムである．本システムの利用の流れは次のようになる．

Step1: 変更前のプログラムに対するグラフの構築

Step2: ユーザによるソースファイルへの変更

Step3: 変更後のプログラムに対するグラフの再構築

Step4: 変更前後の差分からの被影響メンバ抽出

Step5: 抽出された被影響メンバの表示

3.3.2 システム構成

本システムは，実際に提案手法による影響波及を行う解析部と，ユーザインタフェースとなる GUI 部で構成されている．図 3.7 はシステムの各構成要素の関連を示したものである．以降，システムにおいて利用したツールと，解析部および GUI 部を説明する．また，以下の説明では，MOG，MAG を単にグラフと呼ぶ．

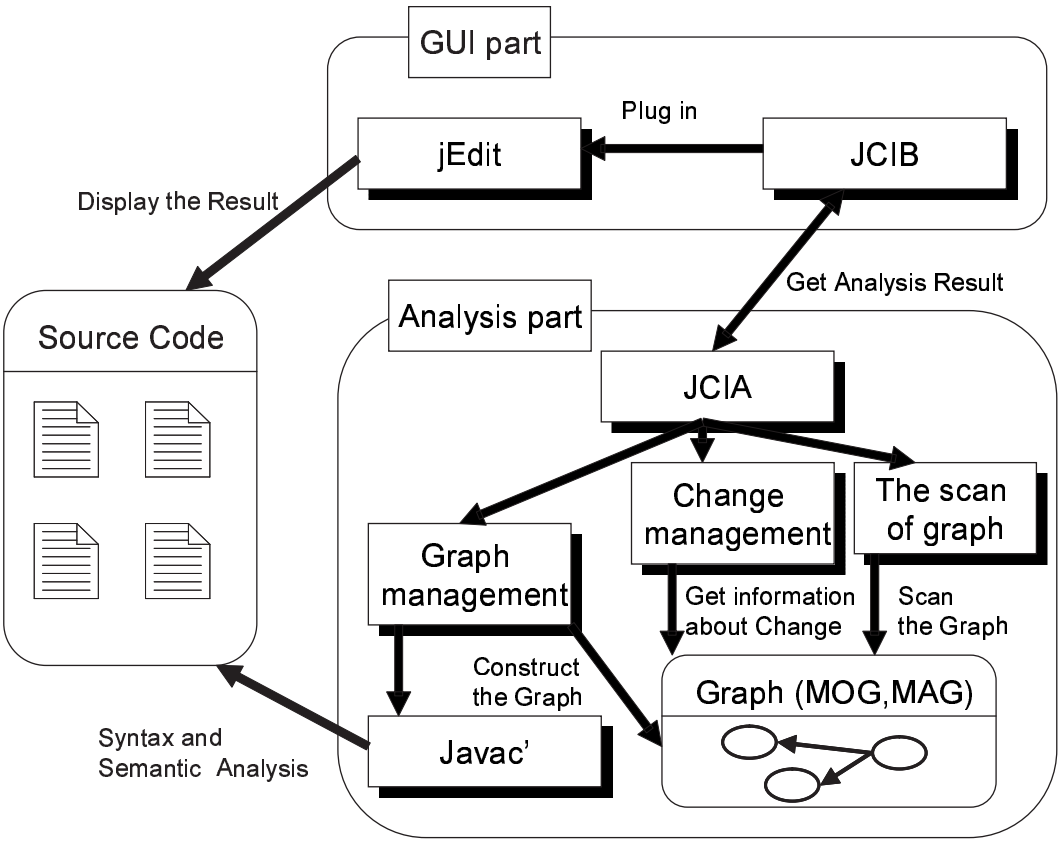


図 3.7: JAVA 影響波及解析システムの構成

利用したツールについて

本システムでは，Java プログラムの字句解析，構文解析，意味解析に javac[42] のコードを流用し，GUI に jEdit[44] を採用した．

javac は JDK[43] 付属の Java コンパイラであり，本システムで流用するにあたって行われたコード変更は 100 行未満に抑えられており，将来の JDK のバージョンアップにも容易に対応できる．

jEdit は Pestov らにより開発されているオープンソースのテキストエディタで，プラグイン機能の提供により高い拡張性を有している．本システムは，jEdit の 1 プラグインとして動作する．

解析部の説明

解析部は，次の 5 つの部分から構成されている．

JCIA： グラフ管理部，変更管理部，グラフ走査部に指示を出すことにより，影響波及解析を行う．

グラフ管理部： 指定された JAVA プログラムのグラフの構築を行う．

変更管理部： 変更前後のプログラムに対応するグラフを比較し，節点, 辺の発生, 消失といったグラフの変化を検出する．

グラフ走査部： 与えられた被影響メンバ探索ルールに従ってグラフを走査し，被影響メンバの抽出を行う．

javac’： JAVA プログラムのソースコードに対し，字句解析，構文解析，意味解析を行うと共に，グラフ構築に必要な情報を収集する．

GUI 部の説明

GUI 部は，JCIB と jEdit の 2 つから構成されている．

JCIB： jEdit のプラグインとして動作し，ユーザからの解析要求を解析部に伝え，解析結果を表示する．

jEdit： Java プログラムに変更を行う際のエディタとして動作し，JCIB と連携して解析結果を表示する．

3.4 適用実験

ここでは，実際にソフトウェアに対して実装システムの適用を行い，提案手法の有効性を検証する．

具体的には，ソフトウェアのある基準バージョンのソースコードと，何かしらの変更が行われた後のバージョンのソースコードに関し，システムが検出した被影響メンバと実際のソースコード差分との比較を行う．そして，その結果を考察することで，システムがユーザに対して適切な情報を提供できているか検証する．

3.4.1 適用対象ソフトウェア

本システムの適用対象ソフトウェアとして，Java ベースのビルドツールである Ant を選択した．Ant は Jakarta Project で開発されているオープンソースソフトウェアであり，Ant v1.1 は，96 個のクラス (約 20,000 行) から構成されている．

3.4.2 システムの適用事例

ここでは，Ant の開発過程において実際に行われた変更に対してシステムの適用を試みる．Ant は v1.1 から v1.2 において，様々な機能追加および修正が行われているが，その中の一つであるメソッド `Property::init()` の削除に着目し，このメソッドの削除による被影響メンバをシステムを用いて特定する．`Property::init()` の削除によって生じた全ての直接的な影響を調べるために「関係変化メンバ抽出」を探索ルールとして指定し，解析を行った結果，多くの被影響メンバが抽出された．そこで MOG および MAG ごとに分けることで,「関係変化メンバ抽出」ルールを「オーバーライド関係の変化」と「アクセス先の変化」に分解して再解析したところ，次のようなことを把握できた．

- オーバーライド関係の変化
  `Property` の親クラス `Task` のメソッド `init()` がオーバーライドされなくなった．
- メソッドのアクセス先の変化
  `TaskHandler::init()`，`Ant::execute()` 中のメソッド呼び出し式における呼び出し先が変更された．

3.4.3 実際の更新作業との比較に基づく考察

削除されたメソッド `Property::init()` の機能について考えると，その機能が不要な場合は削除され，必要な場合はプログラムの他の部分に移譲されるはずである．後者の場合，`Property::init()` の削除により影響を受けるメンバにおいて，何らかの対応を行う必要がある．そこで，上記の 3 つの被影響メンバ `Task::init()`，`TaskHandler::init()`，`Ant::execute()` が，v1.1 から v1.2 においてどのように修正されたかを調査した．

- `Task::init()`
  v1.1 から v1.2 において全く変更が行われておらず，`Property::init()` 中に存在していた機能は，オーバーライドの対象であった `Task::init()` に移譲されたわけではないことがわかった．
- `TaskHandler::init()`
  v1.1 から v1.2 においてメソッドに変更が加えられており，各バージョンにおけるソースコードは図 3.8 のようになっていた．`Property::init()` の削除とは関係のない，機能追加と考えられる部分を除くと，図 3.8 の網掛部が削除の影響により変更された部分であると思われる．v1.2 では，`Task` 型の参照変数 `task` が指し得るオブジェクト (`Task` クラスの子クラスである `Property` クラスのオブジェクトな

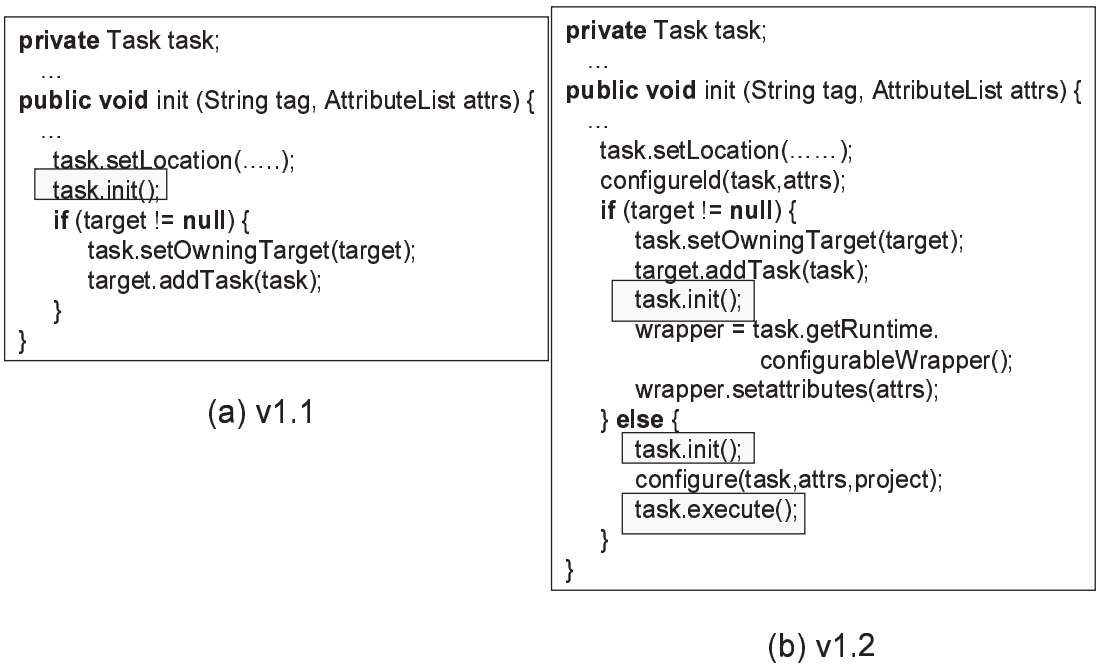


図 3.8: v1.1 と v1.2 における TaskHandler::init() の違い

ど) に対して，v1.1 と同様に `init()` の呼び出しを行うだけでなく，場合によっては `execute()` の呼び出しも行うよう変更されていることがわかる．

- `Ant::execute()`
  v1.1 から v1.2 においてメソッドに変更が加えられており，各バージョンにおけるソースコードは図 3.9 のようになっていた．これにより，`Property::init()` の呼び出し式が，`Property::execute()` の呼び出し式に置換されていることがわかる．

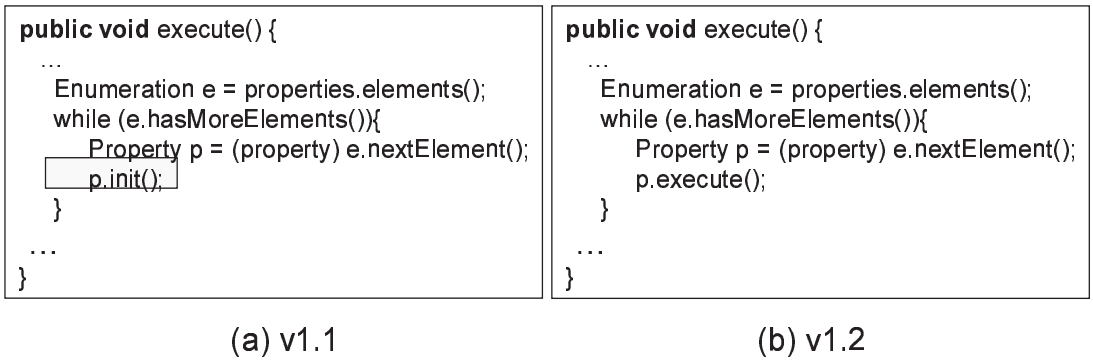


図 3.9: v1.1 と v1.2 における Ant::execute() の違い

上記の変更点より，`Property::init()` の機能は，`Property::execute()` に移譲されたと推測できる．実際に，v1.1 における `Property::init()` と v1.2 における

`Property::execute()` の間には機能の一致が確認できた．

以上より，システムにより抽出された被影響メンバは，実際の変更作業において確かに修正が行われていたことがわかり，ソフトウェアに対して変更を行う際に本システムによって修正個所の特定を行うことは有効であると考えられる．今後は，本システムを用いることで変更作業に要する時間が短縮できることを，実際のソフトウェア開発者を被験者とした評価実験を行うことによって検証することが必要である．

3.5 3章のまとめ

本章では，オブジェクト指向言語である JAVA の特性を考慮した影響波及解析手法の提案を行った．本手法では，クラスのメンバ間の関係を表現する 2 種類のグラフを定義し，その上でグラフの変化の種類によって変更の影響を受けるメンバを分類し，それらの探索ルールを選択して適用できる枠組を実現した．これにより，ユーザの目的に応じた影響の定義，抽出を行うことが出来る．

また，提案手法を JAVA 影響波及解析システムとして実装し，実際のソフトウェアの更新履歴に適用した．適用したところ，システムにより抽出された被影響部分は，実際の変更作業において修正が行われていた．そのため，本システムを用いることにより修正個所の特定を効果的に行うことができると考えられる．

第4章　利用関係を用いたソフトウェア部品評価手法

4.1　導入

近年，大規模で複雑なプログラムを含む大量のソフトウェアが開発され，様々な場所で様々な目的で利用されている．これらのプログラムの中には似た種類のプログラムが多数存在し，開発期間の短縮や品質向上を目的として，既存のソフトウェア部品を同一システム内や他のシステムで利用する，いわゆるソフトウェアの再利用がよく用いられている．ソフトウェアの再利用による効果を最大限に引き出すためには，開発者が今から開発しようとするソフトウェアに必要な部品およびライブラリに関する知識を持つことが重要になってくるが，知識の共有が満足になされていないために，同種のプログラムが別々の場所で，独立して開発されていることも多い．

一方でインターネットの普及により，SourceForge [46] などのソフトウェア開発に関する情報を交換するコミュニティが誕生し，大量のプログラムソースコードが簡単に入手できるようになった．これらの公開されている大量の部品の中から，開発者の必要としている機能を持つ部品，その機能の使い方を示している部品のような，再利用に有益な情報を提供する検索システムを実現することで，知識の共有が実現でき，再利用を促進することができると考えられる．

知識の共有を目的としたソフトウェアの部品検索システムを構築する際，検索された部品を評価し順位付けし，選別して表示する仕組みが必要となる．部品の特性から個々のソフトウェア部品の再利用性を評価する手法が提案されているが [13, 59]，このような部品検索システムにおける利用を考えた場合，目的にあった部品を選出することと同様に，いろいろなソフトウェアで使われ完成度が高い部品を選出することが必要であると考えられる．このとき，各部品の利用実績を定量的に評価した指標が必要となる．

そこで本章では，ソフトウェア部品間の利用関係から各部品を評価し，順位付けする手法 (Component Rank 法, CR 法) を提案する．この手法では，十分な時間が経過し利用関係が収束したソフトウェア部品の集合に対して，各ソフトウェア部品間に存在する利用関係に基づいてグラフおよび行列を構築し，構築された行列に対して繰り返し計算を行うことで各部品の重みを求め，順位付けする．求められる重みは，開発者が利用関係に沿って参照を行うと仮定した場合の各部品の参照されやすさを表しており，よく利用される部品や，重要な部品から利用される部品の順位は高くなる．この順位を利用実績として部品検索システムで用いることで，既存の再利用性評価手法で欠けている観点である，利用実績面からの評価を行うことができると考えられる．

提案する手法に基づいて，プログラミング言語 Java で開発されたソフトウェア群から各部品の Component Rank を計測するシステムを開発し，求めた順位が利用実績として妥当

であるかを検証するために幾つかのソフトウェアシステムに適用する．また，CR 法を用いて順位付けし検索結果を表示する部品検索システム (Software Product Archiving, analyzing, and Retrieving System ,*SPARS*) について説明する．

以降，4.2 節では，提案手法「CR 法」の定義と計算方法について述べる．4.3 節では，CR 法に基づく部品評価システム「CR-システム」の実現について述べる．4.4 節では，Java ソースコードへの適用実験について述べる．4.5 節では，関連研究およびソフトウェア部品検索への考察について述べる．最後に 4.6 節では，まとめと今後の課題について述べる．

4.2 Component Rank 法の提案

ここでは，部品の概念をモデル化し，その上で利用関係から部品を評価する手法 (Component Rank 法, CR 法) について説明する．

4.2.1 部品と部品間の利用関係

一般にソフトウェア部品 (Software Component) とは再利用できるように設計された部品とされている [21, 49]．ここではより一般的に，ソースコードファイルやバイナリファイル，ドキュメントなどの種類を問わず，開発者が再利用を行う単位をソフトウェア部品，あるいは単に部品と呼ぶ．これらの部品間には互いに利用する，利用されるという利用関係が存在する．本論文では各部品を頂点，部品間の利用関係を利用する側からされる側への有向辺として，グラフ上に表現する．以下，この部品間の利用関係を表現したグラフを**部品グラフ** *(Comopnent Graph)* と呼ぶ．以下では，V を部品（頂点）の集合，E を有向辺の集合として，$G = (V, E)$ と表現する．

図 4.1 は二つのソフトウェアシステム X と Y を部品グラフ化したものである．X は A から E の 5 つ，Y は F から I の 4 つの部品で構成されており，部品 C は 部品 A および B を利用し，部品 D および E は 部品 C を利用している．同様に部品 H と I は 部品 G を利用し, 部品 G と F はお互いに利用しあっている．

4.2.2 部品と利用関係の重みの定義

CR 法では，部品グラフ $G = (V, E)$ 上の個々の辺および頂点に対して重みを計算し，対応する頂点の重みをもとに各部品を評価する．最初に，頂点の重みを次のように定義する．

定義 1 (頂点の重みの和) 部品グラフ G 上の各頂点 v は $0 \leq w(v) \leq 1$ の重みを持ち，G の頂点の重みの総和は，*1* とする．

$$\sum_{v \in G} w(v) \qquad = 1$$

また，部品グラフ G 中の頂点 v の重みを求めるために，辺の重みを次のように定義する．

定義 2 (辺の重み) 頂点 v_i から v_j への辺 e_{ij} に関する辺の重み $w'(e_{ij})$ を次のように定義する．

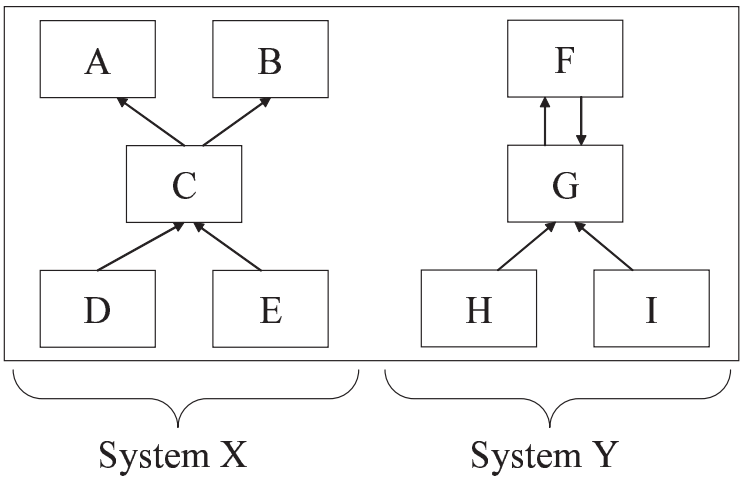


図 4.1: 部品グラフの例

$$w'(e_{ij}) = d_{ij} \times w(v_i)$$

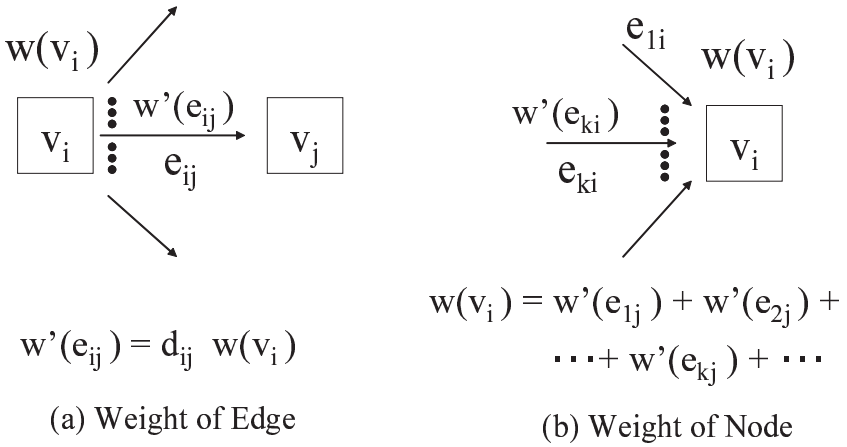


図 4.2: 部品グラフ上の頂点および辺における重みの定義

図 4.2 (a) はこの定義を図示したものである．d_{ij} は配分率とよび，$0 \leq d_{ij} \leq 1$ かつ $\sum_i d_{ij} = 1$ を満たす値とする．頂点 v_i から v_j へ利用関係が存在しない場合，ここでは $d_{ij} = 0$ とする．この配分率 d_{ij} は，次に示す頂点の重みの定義において，有向辺の終点となる頂点の重みの決定に利用される．図 4.2 (b) は次の定義を図示したものである．

定義 3 (頂点の重み) *IN(v_i)* を v_i を終点とする有向辺の集合とする．この時，頂点 v_i の重みは v_i が終点となる有向辺 e_{ki} の重みの総和とする．

$$w(v_i) = \sum_{e_{ki} \in \text{IN}(v_i)} w'(e_{ki})$$

4.2.3 重みの計算と Component Rank

2.2 節の重みの定義に基づいて，頂点 $w(v_i)$ に対して次の方程式が生成できる．

$$w(v_i) = \sum_{e_{ki} \in \text{IN}(v_i)} d_{ki} \times w(v_k)$$

各頂点に関してこの方程式を立てることで，$n(= |V|)$ 個の連立方程式が生成できる．また，各頂点の重みをベクトル W として次のように表現し，

$$W = \begin{pmatrix} w(v_1) \\ w(v_2) \\ \vdots \\ w(v_n) \end{pmatrix}$$

配分率を次の行列 D として表現することで，

$$D = \begin{pmatrix} d_{11} & d_{12} & ... & d_{1n} \\ d_{21} & d_{22} & ... & d_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n1} & d_{n2} & ... & d_{nn} \end{pmatrix}$$

$n(= |V|)$ 個の連立方程式を次のように表すことができる．ここで 行列 D^t は D の転置行列を表す．

$$W = D^t W \tag{4.1}$$

定義 1 から，行列 D^t は推移確率行列の性質を満たしているため，D^t を用いて開発者が部品をどのように参照するかをマルコフ連鎖 [5] でモデル化し表現する．すなわち，開発者はある部品を参照した後に，辺に沿って利用関係のある部品の一つを参照するとみなす．ここで，式 (1) を満たす解 W は D^t に関する定常分布となり，対象とする部品の集合の中での参照を十分長く繰り返した場合の，開発者によって各部品が参照されている確率を示すことになる．つまり，重みが高い部品ほど開発者によって頻繁に参照されることになる．この値を用いることで，ただ単に利用数が多い部品だけでなく，利用数が多い部品が利用している部品も評価することができる．定常分布が一意に求められることを示すためには，D^t が既約であることが必要となるが，CR 法では，既約であることを保証するために後述する補正を加えている．

この場合，D^t における絶対値最大の固有ベクトルを求めることで，定常分布を求めることができるが，行列 D^t の絶対値最大の固有値は 1 であるため，累乗法 (power method)

を用いて適当な初期ベクトルに対して繰り返し D^t を掛けることで，近似解を容易に求めることができる．図 4.3 は，与えられたグラフにおいて各頂点の重みを計算した結果である．v_1 は 2 つの有向辺の始点で，v_1 の重み 0.4 は二つの辺に 0.2 ずつ等分されている (つまり，$d_{12} = d_{13}$ = 0.5)．また，v_3 は 2 つの有向辺の終点で，それぞれの辺が 0.2 の重みを持つため，v_3 の重みは 0.4 であることがわかる．

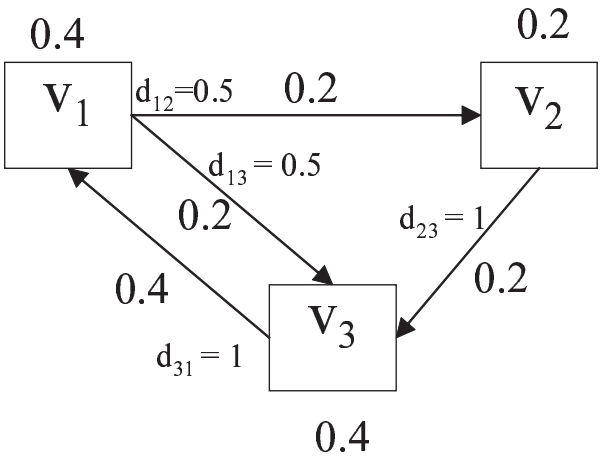


図 4.3: 部品グラフ上の頂点の重みの計算例

収束した時点での各頂点の重みを，その部品の重みとする．その際，部品の持つ重みの和が 1 と固定されているため，同一の部品であっても，求められる値は解析対象となるソフトウェア部品の集合によって大きく変わり，値自体の値にはあまり意味がない．そのため，CR 法では解析対象における順位をもとに評価を行う．この順位を Component Rank とよぶ．

4.2.4 繰り返し計算における補正

擬似辺による補正

前節で部品（頂点）の重みの計算に関する説明を行ったが，実際に運用する場合不具合が起こる場合がある．例えば，部品 i がどの部品も利用していない場合，頂点 v_i から全ての頂点への配分率 d_{ix} が全て 0 になり，配分率に関する定義（頂点からの辺への配分率の総和が 1）を満たさなくなる．また，グラフが図 4.4(a) のように強連結でない場合，頂点 v_1 の重みが 0 になってしまい，v_1 から v_2 への利用関係を正しく評価することができない．

このような場合，与えられた行列が既約であることを保証できず，定常分布が一意に定まらない場合がある．そこで，全ての頂点を図 4.4 (b) のように低い配分率の擬似辺で結ぶことで，与えられたグラフを強連結なグラフに変換し，行列が既約であることを保証する．この擬似辺は，各部品における自分を含めた全ての部品への明示的でない参照を意図している．ここで p は実際の辺と擬似辺の重みの配分比率を指す．

定義 4 (修正配分率) 頂点 v_i を始点とする辺への配分率 $d'(e_{ij})$ を次のように定義する．

$$d'_{ij} = \begin{cases} p \times d_{ij} + (1-p)/n & \textit{if } v_i \text{ からの辺が存在} \\ 1/n & \textit{if } v_i \text{ からの辺がない} \end{cases}$$

この補正は，前節で説明したマルコフ連鎖を用いたモデル上での参照行為を,「ある部品を参照した後に，辺に沿って利用関係のある部品の一つを参照するが，ある確率（$1-p$）で，ランダムに全部品の中の一つを参照する」のように修正する．この修正により，参照行為をより自然な形でモデル化することができる．

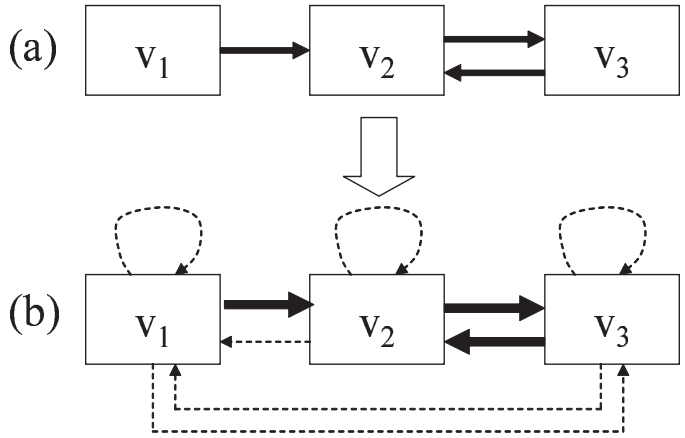


図 4.4: 部品の重みの計算に関する補正（擬似辺の追加）

部品のグループ化による補正

システムを構築する際には，過去に開発したシステムの部品の一部を再利用して開発が行われることがよくある．これらの部品は，ライブラリとして再利用される他に，コピーされたり，コピーされた上で大小様々な変更が加わった上で再利用されることが考えられる．そのため，様々なソフトウェアから部品を集めた場合，その部品集合にはコピーした部品や，コピーして一部を変更しただけの部品（類似部品）が数多く存在すると考えられる．

このとき，異なるシステムをまたいで類似部品が現れる場合を考える．再利用という観点から考慮すると，これらの類似部品は類似部品中のある一つの部品を利用してコピーして作成されたと推測することができる．そのため，それらの類似部品間に利用関係を定義するのが妥当であると考えられる．ライブラリとして再利用された場合には，部品グラフ上で利用関係を有向辺として定義できるが，コピーの場合はコピー元の特定が難しく，部品グラフ上の有向辺として定義するのは難しい．

そこで，提案手法では似た部品をグループ化し，部品群を頂点として，部品群それぞれに対して重みを与える．このとき各部品の重みは，その部品が属する部品群の重みとする．その際，それぞれの類似部品へ指されていた辺が，グループ化を行うことで一つの部品群への辺とみなされる．そのため，コピーして再利用される回数が多いほど，その部品群はたくさんの部品から利用されることになる．グループ化によりコピーされた部品に対応す

る頂点の重みが高くなるため，コピーされたという利用関係を Component Rank に反映させることができる．

グループ化を行う際には，2 つの部品がどの程度類似しているかを定量的に評価するために部品間の類似度 (Similarity) を評価するメトリクスを利用する．以下，部品 v_1，v_2 間の類似度を $s(v_1, v_2)$ と表す．類似度は 0 以上 1 以下の範囲に正規化され，値が高いほど部品は良く類似しているとし，類似度 1 を完全に部品が一致した場合（コピーした部品）とする．また，グループ化の閾値として t という値を定義し，$s(v_1, v_2) \geq t$ であるときに $s(v_1, v_2)$ は類似関係にあるとする．部品集合 V 内で類似関係にある部品を同一の部品群とすることで，V の商集合からなる類似部品群の集合 V' を求める．この時，G の部品群グラフ（*clustered component graph*）$G' = (V', E')$ を次のように定義する．

定義 5 (部品群グラフ $G' = (V', E')$) V の商集合からなる集合 V' を G' の頂点集合とする．また v_i，v_j が属する V' 中の集合をそれぞれ v'_i，v'_j としたときに，$E' = \{(v'_i, v'_j) | (v_i, v_j) \in E\}$ を G' の辺集合とする．

図 4.5 は 部品のグループ化を行った際に，コピーされた部品の重みがどのように変わるかを例で示したものである．3 つのシステムのうち，2 つのシステムには $X.A$ および $Y.A$ という同一の部品が存在する．A という部品がライブラリとして再利用された場合は，部品 A が部品 $X.B$ および $Y.C$ の両方と利用関係にあることを把握できる．しかし一方で，部品がコピーされて再利用された場合は，部品が属する名前空間（パッケージ名）が異なる，または部品名が異なるなどの理由から $X.A$，$Y.A$ は別々の部品として扱われてしまう．このとき，グループ化を行わないと，コピーされた部品は別々の部品として評価されるため，図 4.5 の場合すべての部品の重みが等しくなる．

そこで，提案手法では部品間の類似度を測定することでグループ化を行う．これにより $X.A$ および $Y.A$ が同一部品であると判定され，部品 $X.A$ および $Y.A$ へのそれぞれの有向辺が一つの部品群 A' への有向辺に統合されるため，コピーが存在する部品 A の重みが他の部品よりも高くなることがわかる．以降の実際の部品の重みの計算では，部品グラフではなく部品群グラフを対象としており，部品群グラフに対して修正配分率に関する補正を行った上で部品群に対して各頂点の重みを計算している．

4.3 Component Rank システム（CR-システム）

2 節で提案した CR 法に基づいて，我々は Java プログラムを対象としてソースコード部品を評価するシステム「Component Rank システム（，以下 CR-システム）」を構築した．ここでは，構築した CR-システムについて説明する．

4.3.1 Java の概念との対応

Java に適用する際の，モデルと Java の概念との対応を以下に示す．

部品: オブジェクト指向言語 Java は原則として 1 つのソースファイルには 1 つのクラスを記述している．そこで，クラス単位での利用が行いやすいソースコードファイルを部品の単位とする．

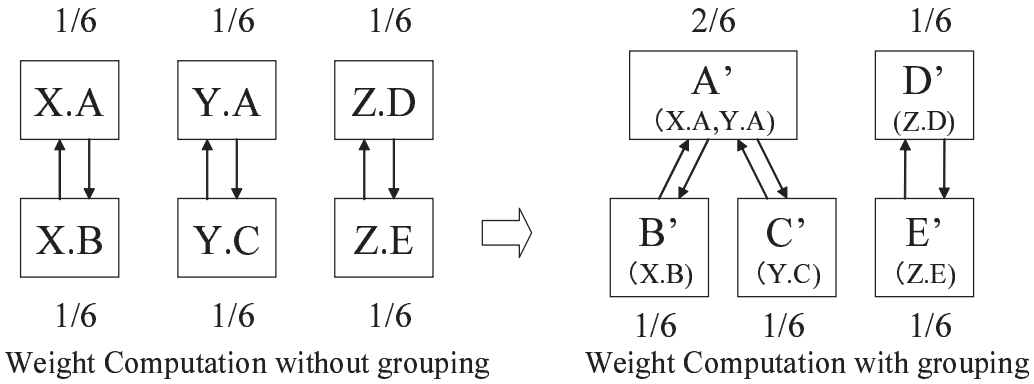


図 4.5: 部品のグループ化の効果

利用関係: 部品間の利用関係をクラスの継承，インターフェースおよび抽象クラスの実装，メソッドの呼びだし，フィールド参照とする．

辺と擬似辺の重みの配分比率 p: 配分比率 p として $p = 0.85$ を採用している．この場合，部品の重みの 85%が実際の辺に分配され，15%が擬似辺の重みとして全ての部品に均等に分配される．

分配率: 現在は計算の効率化のため，分配率を利用関係の種類にかかわらず全て等しくしている．例えばある部品が 5 つの部品を利用した場合，それぞれの辺は 17%の重みを持つ．

類似度の閾値 t: グループ化において部品間の類似度を評価するメトリクスとして，文献 [58] で提案されている「S_{line}」を用いる．S_{line} は 2 つのソースコードファイル間で一致する行の割合で，2 つのファイル間の類似度を表す.「SMMT(Similarity Metrics Mesuring Tool)」は S_{line} をソースコードファイルから計測するシステムである．閾値 t については $t = 0.8$ を採用している．

4.3.2 CR-システムの構成

CR-システムの構成を図 4.6 に示す．解析の手順は以下の通りである．

(1) 入力された M 個の部品に対して，SMMT を用いることで，各部品間の類似度を計測する．

(2) SMMT で得られた類似度をもとに，閾値 t 以上の類似度を持つ部品をまとめることで，M 個の部品を，N 個の部品群にグループ化する．

(3) M 個の Java ソースコードファイルを解析し，利用関係を抽出する．Java ソースコードの構文解析には，ANTLR[39] を利用している．

(4) 部品群に関する情報とファイル間の関係抽出部の結果（部品間の関係）から，部品群間の利用関係を求め，部品群グラフを形成する．

(5) 部品群グラフから行列を構築し，行列における繰り返し計算をもとに定常分布を求め，それを各頂点（部品）の重みとする．

(6) 各部品が属する部品群の重みから，各部品の重みを求め，順位付けを行い，Component Rank を出力する．

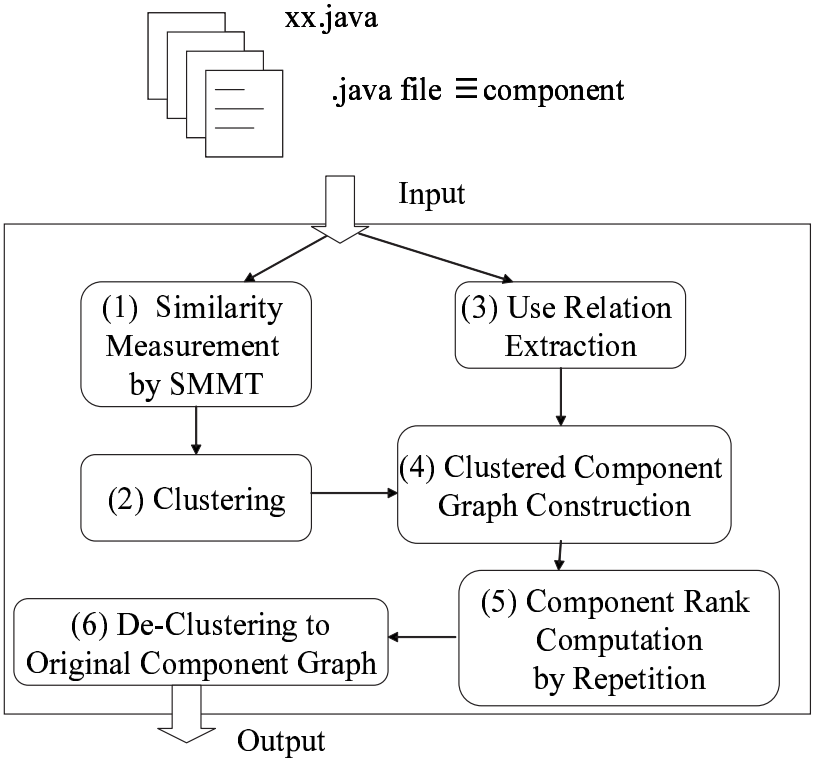


図 4.6: Component Rank システム（CR-システム）の構成

4.4 評価実験

本節では，CR-システムによって求められた部品の順位が利用実績として妥当であるかを確認するために，3 つのソフトウェア部品の集合に対して適用実験を行う．

4.4.1 JDK 1.3.0 への適用

Sun Microsystems [47] から配布されている Java 2 Software Development Kit, Standard Edition 1.3.0 (以下 JDK) のソースコードを対象として評価実験を行った．JDK は 1877 ファイルで構成され，総行数は約 575000 行である．Pentium 4 2GHz の CPU, 2GBytes のメモリーを持つ PC 上で，JDK に対して CR-システムを用いて順位付けを行ったところ，約 7 分の時間を必要とした．

表 4.1: JDK 1.3.0 における Component Rank

C.Rank	Class Name	Weight
1	java.lang.Object	0.16126
2	java.lang.Class	0.08712
3	java.lang.Throwable	0.05510
4	java.lang.Exception	0.03103
5	java.io.IOException	0.01343
6	java.lang.StringBuffer	0.01214
7	java.lang.SecurityManager	0.01169
8	java.io.InputStream	0.01027
9	java.lang.reflect.Field	0.00948
10	java.lang.reflect.Constructor	0.00936
⋮	⋮	⋮
1256	sunw.util.EventListener	0.00011
1256	⋮	⋮

表 4.1 は得られた順位の上位 10 位までを表にしたものである．上位 10 クラスについて見てみると，Object, Class, Throwable など，Java の言語仕様 [15] で利用しなければならないクラスが大半を占めている．例えば，java.lang.Class クラスは実行中のクラスおよびインターフェイスを表すクラスで，このクラスを直接継承するようなクラスはないが，実行時のオブジェクト型の情報を取得するために頻繁に呼び出される．一方で最下位は 1256 位で，最下位に属する 622 クラスはどのクラスからも利用されていない．

これらの Java において重要な役割を担っているクラスが上位を占めているということから，CR 法による部品の順位付けは利用実績として妥当であると考えられる．

4.4.2 研究室内ソースコードへの適用

研究室内で過去に開発された Java アプリケーションのソースコード (582 ファイル) を適用対象として実験を行った．適用対象について簡単に説明する．

C-K メトリクス計測ツール: Java ソースコードから C-K メトリクス [9] を計測するツールとそのバージョンアップ版．パッケージ名は cktool および cktool_new(29+29 ファイル)．ANTLR[39] を使用してソースコードの構文解析を行っている．

CR-システム: 本論文で作成した CR-システム．下記で示したライブラリを利用している．
パッケージ名は
jp.ac.osaka_u.es.ics.iip_lab.metrics．(68 ファイル)

ライブラリ群: ANTLR(パッケージ名 antlr, 188 ファイル), JAMA[40](パッケージ名 Jama, 9 ファイル), Caffe Cappuccino Class Library[54](パッケージ名 jp.gr.java_conf.keisuken,

245 ファイル).

その他: (14 ファイル)

適用結果を表 4.2 に示す．2 位，8 位に同じ順位の部品がそれぞれ存在している．これは，それぞれ二つのクラスがよく似ているため同一部品群に存在していることを示している．

複数のアプリケーションで共通に使用される ANTLR のようなパッケージが，研究室内で作成されたアプリケーションよりも全体的に上位に来る傾向が見られる．また，データを格納する Token，Vector，Matrix，Array クラスやデバッグ情報を出力するための Event クラスのような，ソフトウェア全体で広く利用されるクラスが高く評価されている．これらのクラスは，ソフトウェア内で参照される回数が多く，様々なクラス内で利用されていることが予想できる．ソフトウェア内での利用回数が多く，汎用的で重要な役割を担っていることを予想できるようなクラスが上位に来ており，これらの結果から CR 法による順位付けは利用実績として妥当であると考えられる．

表 4.2: 研究室内ツールにおける Component Rank

C.Rank	Class Name	Weight
1	antlr.Token	0.10727
2	antlr.debug.Event	0.06189
2	antlr.debug.NewLineEvent	0.06189
4	antlr.collections.impl.Vector	0.05434
5	jp.gr.java_conf.keisuken. text.html.HtmlParameter	0.05246
6	jp.gr.java_conf.keisuken. net.server.ServerProperties	0.03699
7	Jama.Matrix	0.01564
8	jp.gr.java_conf.keisuken. util.IntegerArray	0.01390
8	jp.gr.java_conf.keisuken. util.LongArray	0.01390
10	jp.ac.osaka_u.es.ics.iip_lab.metrics. parser.IdentifierInfo	0.01365
⋮	⋮	⋮
418	cktool_new.examples.Main	0.00050

4.4.3 部品検索への利用例

今回提案した手法の応用例として，インターネット上で公開されている大量の部品の中から，開発者の必要としている機能を持つ部品や，その機能の使い方を示している部品の

ような，再利用に有益な情報を提供するための部品検索システムの実現が考えられる．このとき，あらかじめ全ての部品に対して CR 法を用いてその部品の Component Rank を利用実績として一元的に定義し，検索結果において部品が複数検出されたときに，Component Rank の順に並び替えて表示を行う．これによりよく利用される汎用性の高い部品を先頭に表示することができるため，知識の共有を円滑にでき，再利用を促進することができると考えられる．

表 4.3: 検索結果（Component Rank 順）

Order (C.Rank)	Class Name	weight
1(67)	enhydra3.1..dom.Node	0.029110
2(169)	saxon7_0..saxon.om.NodeInfo	0.000969
3(275)	saxon7_0..saxon.pattern.NodeTest	0.000437
4(316)	enhydra3.1..dom.DocumentImpl	0.000368
5(355)	saxon7_0..saxon.pattern.Pattern	0.000324
6(382)	saxon7_0..saxon.Controller	0.000296
7(437)	enhydra3.1..xslt.XSLTEngineImpl	0.000241
8(446)	enhydra3.1..dom.ElementImpl	0.000235
9(500)	saxon7_0..saxon.style.StyleElement	0.000202
10(506)	saxon7_0..saxon.tree.NodeImpl	0.000198
⋮	⋮	⋮
125(4441)	enhydra3.1..FuncID	0.000029
125(4441)	⋮	⋮

Section 4.2 節で紹介したツールおよび，テキストエディタ *JEDIT* ，アプリケーションサーバー *Enhydra*，XSLT プロセッサー *saxon*，gnutella client*phex* などの SourceForge[46] で公開されている Java アプリケーション群を対象に検索の例を示す．総ファイル数は 7171 個で，これらのアプリケーションでは XML 文書に関する操作を行っている．

これらの部品を対象として，DOM ツリーの中で現れるノードの種類を得るためのメソッドである”getNodetype” について検索を行う場合を考える．今回の検索では UNIX のコマンドである *grep* を用いて検索を行った．ただし，コメントにのみ現れる場合は除外している．検索の結果，181 のクラスが該当し，CR-システムの順位をもとに検索結果を順位づけしたところ，表 4.3 のようになった．

部品全体の順位は JDK に対する適用の場合と同様に，java.lang.Object のような汎用的な部品が上位を占めるが，検索結果においてはそれらの部品はヒットせず，部品定義を行う部品とその利用方法に関する部品のみが抽出された．今回は，1 位と 2 位にそのメソッドの定義が出現し，その他のクラスは利用例であった．利用例を見ていくと，上位の利用例は DOM の解析などにおけるノードの操作（3〜5，10 位），スタイルシートなどの XML 文書の解析（6〜9 位）など一般的なものであったが，下位になればなるほど，XML で書

かれた内容を解釈して実行するための FuncID クラスのような特殊な利用例が現れやすくなっていた．

今回提案する Component Rank を部品検索システムにおける検索結果の表示順位決定に用いることで，今回の例のように大量の部品の中からでも部品定義に関する情報を上位に配置できる．さらに，ある機能の利用方法を知りたい場合にも，一般的な利用方法から参照できるようになるため，知識の共有を円滑に行うことができる．順位付けがない場合，最悪の場合全ての部品を見る必要があることを考えると，CR 法を部品検索システムの表示順位の決定に用いることは有益であると考えられる．

4.5 考察

4.5.1 関連研究

利用回数による評価手法

本論文で提案した CR 法は，利用関係をもとに各部品を評価する手法である．利用実績を評価する方法としては，各部品の利用回数をもとに評価する手法も考えられる．[48] では、クラスの被利用クラス数からクラスの人気度を測定する測定法として，CP（class popularity）が提案されている．

この手法の場合，CR 法のような繰り返し計算は必要としない．しかし，今回想定しているようなソフトウェアの部品検索システムにおける利用を考えた場合，実際に機能を実現している部品を発見するのが困難になる場合があると考えられる．例えば図 4.7 のような，複数の部品である機能 X を実現しているような部品の集合 A，B，C，D が存在するとする．この部品の集合では，B，C，D は機能 X の一部をそれぞれ実現している一方で，A は機能 X に関して外部とのやりとりをするためのラッパークラスで，B，C，D で実現されているメソッドを呼び出すだけである．この機能 X がよく利用されている場合に評価を行う場合を考える．この場合，利用回数で評価してしまうと．検索を行ったときに橋渡しをしている A のみが評価され，実際に機能 X を実現している B，C，D が評価されない．

一方で CR 法では，A の重みが利用関係を通じて B，C，D に反映されるため，B，C，D も比較的上位として評価することができる．実際に機能 X を実現している部品 B，C，D はただ機能 X の使い方を知りたい際にはさほど重要ではないが，機能 X を理解したり機能 X を拡張する際には極めて重要であると考えられる．再利用支援を目的とした部品検索システムの場合，前者の利用法の取得だけでなく，後者のような機能理解や機能拡張も検索の目的として考えられる．そのため，部品検索に利用する際には CR 法を用いる方法が適切であると考えられる．Java のようなオブジェクト指向言語においては，複数の部品で一つの機能を生成することが多く，そのような部品内では他のメソッドを呼び出すだけのメソッドが定義されることが多いため，今回のように Java で記述された部品に対して CR 法を適用することは適切であると思われる．

他分野における評価方法

今回提案した CR 法のような，利用関係をもとに行列を生成し繰り返し計算を行うことで評価を行う手法は様々な分野において採用されている．例えば，計量社会学において，ある論文誌を引用する回数および引用される回数をもとに行列を構築し，繰り返し計算を

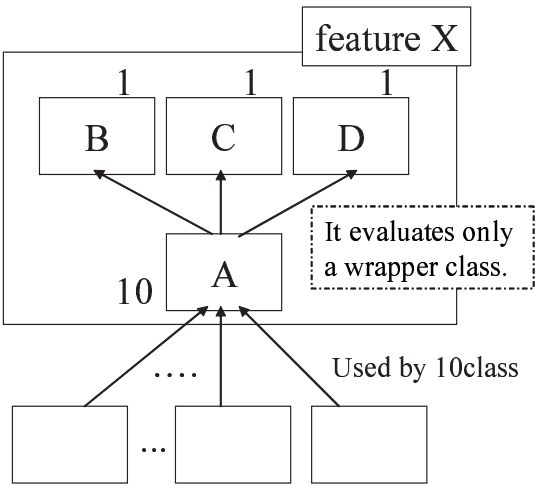


図 4.7: 利用回数による評価における問題

行うことで各学術論文誌の重要度（Influence Weight）を評価する手法が提案されている [30]．また，サーチエンジン Google では，検索結果の表示順を決めるために，リンク構造から行列を構築し，繰り返し計算を行うことであらゆるページの重要度（PageRank）を評価している [55, 28]．Influence Weight や PageRank と CR 法は，利用関係をグラフおよび行列に変換したのち各ノードの重みを計算しているなど，よく似ている．また，JDK の API の一部（java.lang パッケージ）の文書中からリンク構造を抽出し，PageRank 手法を適用した例 [57] では本論文における JDK への適用結果とよく似た結果が示されている．

しかし，これらの手法の対象はソフトウェア部品ではなく，参考文献，ハイパーテキスト，ソフトウェア部品の内容を記述した文献など，全て文書である．現在のところ，ソフトウェア部品に対して直接適用した事例はない．

さらに，ソフトウェア部品に対してこのような手法を適用する際には，コピーの存在を考慮する必要がある．実際のソフトウェア開発現場では，既存の部品からコピーされ修正されることによって出来上がることが多く，コピーは開発期間の短縮などソフトウェアの実現に大きな意味を持つ．一方で学術論文や Web ページのコピーにはほとんど意味がない．部品においてはこのような類似部品の存在が特性となっており，評価を行う際には類似部品の存在を前提としなければならない．

インターネット上から様々なソフトウェア部品を集めた場合，収集された部品の中には，流用したためであるとか，実行に必要なライブラリを添付したためであるなどの理由で，類似部品が大量にあると考えられる．部品のグループ化を行わずにただ単に部品に記述されている利用関係を追跡するだけでは，ライブラリと各ソフトウェア間の利用関係は解析できるが，異なるソフトウェア間のコピーした，されたという関係は解析できない．この場合でも，ライブラリ中の部品に対する Component Rank は利用関係を反映していると考えられるが，ソフトウェア内の部品はソフトウェア毎に利用関係が独立することが考えられるため，利用関係を反映したものにならない．

ソフトウェア内の部品の利用関係を評価するためには，類似部品の存在を念頭においてどのようにその部品が生成されたかを考慮して，異なるソフトウェア間に存在するコピー

に関する利用関係を定義する必要がある．ただ，コピーの関係を部品グラフ上の有向辺として定義するのは難しいため，提案手法では部品のグループ化を行うことでコピーの関係を Component Rank に反映させている．コピーして再利用される回数が多い（類似部品が多い）ほどその部品群を利用している部品の数は多くなるため，類似部品に対応する頂点の重みを高くすることができる．

再利用性の評価

前述の通り CR 法は部品グラフを構築する際，部品の再利用についても考慮している．我々は，CR 法を部品が実際に多くのソフトウェア中に再利用されているという実績を定量的に評価した指標として，ソフトウェア部品の再利用性評価に利用できると考えている．

再利用は，一般に生産性と品質を改善し，結果としてコストを削減するといわれており，コストを削減することができた事例の報告も行われている [4][19][23]．これまでに個々のソフトウェア部品の再利用性を評価する方法がいくつか提案されている．例えば，Etzkorn らは，レガシーソフトウェア中の部品 (C++のクラス) に対して，様々なメトリクス値を計算し，それらの値を正規化して足し合わせることで，再利用性とすることを提案した [13]．また，山本らは，ソースコードが非開示なソフトウェア部品に対して，インタフェース部分の情報のみを用いて再利用性を評価する方法を提案している [59]．

これらの手法は全て，部品そのものから読み取れる特性のみを計算して再利用性を評価するもので，実際のプログラマによる主観的な再利用性の評価の結果と似ているという結果が得られている．しかし従来の手法は，部品そのものから読み取れる特性のみを評価したもので，利用実績については考慮されていない．現実には，従来手法では再利用性が低いと評価されても，多くのシステムで再利用されている部品は数多く存在すると考えられる．CR 法は，従来の再利用性評価手法において考慮されていない部分である，利用実績に関する評価を補完するものであり，ソフトウェア部品の再利用性評価に利用できると考えている．

4.5.2 配分比率および閾値について

今回提案した CR 法では，パラメータとして辺と擬似辺の重みの配分比率 p およびグループ化における閾値 t が存在する．以下では，値の決定について考察する．

配分比率 p に関して p の値を変えて順位の変動を検証したところ，$p = 0$（補正のみしかないため全ての部品が同順位になる）と $p = 1.0$（補正無し）の時以外は，値を変更しても結果として得られる順位に違いはほとんど見られなかった．また，$p = 1.0$（補正なし）の場合は補正を行った場合と比べて 10 倍以上の計算時間を必要とした．

このことから，p は順位の決定に影響を与えるパラメーターではなく，計算の収束にのみ必要なパラメータで，p はどの値でも問題ないことがわかった．そこで，現在のところ最初に採用していた値である $p = 0.85$ を利用している．

一方で，閾値 t に関しても t の値を変えて順位の変動を検証したところ，似ていると判定される部品が増えるため一部の部品の順位は変わるが，結果として得られる順位に大きな影響を与えるものではなかった．しかし閾値を低くした場合，判定回数が増え計算時間が膨大になるため．効率化を行うためにできるだけ高い閾値を用いて判定対象を絞り込む必要がある．そこで，閾値を $t = 0.8$ まであげたところ，閾値を 0.1 にした場合と比べて

時間が 1/10 以下に抑えられたにもかかわらず，閾値 0.1 において類似と判定された部品に関してその半分以上は類似していると判定できた．

グループ化を行う際の目的が「コピーされた部品もしくは一部修正が加わった部品を検出すること」であることを考慮すると,「コピーした上で一割強程度の変更がなされた部品」までを把握できれば十分な精度が得られると考えられ，現在のところ $t = 0.8$ を用いている．

4.5.3 SPARS

私たちは CR-システムを用いたソフトウェア部品検索エンジンとして，現在 SPARS（*Software Product Archiving, analyzing, and Retrieving System*）を開発している．図 4.8 は SPARS の構成図である．

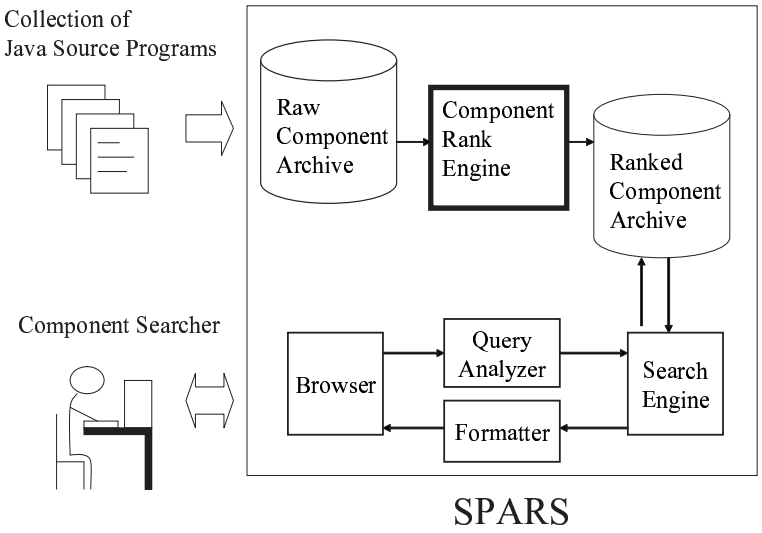


図 4.8: 部品検索システム SPARS の構成

SPARS は収集してきた Java のソースプログラムに対して解析を行い，リポジトリに保存し，CR-システムを用いて各部品を順位付けする．部品を検索したい利用者はブラウザを通じて必要な部品に関する情報を検索キーとして入力する．現在のところ，キーワードによる検索を想定しているが，コード断片による検索も考慮する予定である．部品検索部は，検索キーを解析しリポジトリ内を検索し．検索にヒットした部品を CR-システムによる順位をもとに並び替えて出力する．これにより利用者はよく利用される部品を容易に取得することができる．

4.6 4 章のまとめ

本章では，ソフトウェア部品間の利用関係から各部品を評価し，順位付けする手法として CR 法を提案した．提案手法では，各ソフトウェア部品間に存在する利用関係に基づい

てグラフおよび行列を構築し，構築された行列に対して繰り返し計算を行う．

また，実際に CR 法に基づいて CR-システムを実装し，幾つかのソフトウェアシステムに適用した．いくつかの例に適用した結果，よく利用されるクラスや一般的な使い方をしているクラスが集中して上位のランクに現れ，利用実績として妥当であると同時に，ソフトウェア部品の検索のための順位付けにも有用なものであった．この CR 法を再利用する部品を選択する基準として用いることで，知識の共有をスムーズに行うことができ，開発者の負担を軽減できると考えられる．

第5章　動的情報を利用したソフトウェア部品評価手法

5.1　導入

ソフトウェア開発において既存のソフトウェア部品を再利用することで，生産性と品質を改善し，結果として開発コストを削減することができる．4 章で説明した CR 法は，再利用を用いたソフトウェア開発モデルに従い，利用実績に基づいて部品評価値を求めている．しかし，再利用を行うためには，部品選択のための評価値を示すだけでなく，それら部品がどのような機能の実現において重要な役割を果たしているかを示す必要がある．

CR 法による部品評価値は，様々なソフトウェアで利用されるライブラリのような汎用性の高い部品を検索するのに有効な手法である．しかし，開発者が再利用を行う際，これら汎用性の高い部品の検索が可能となっても，それらが機能的に「何を」実装している部品なのかわからなければ,「再利用できるであろう」部品でしかなく，効率のよい開発を行うことができない．また，個々の部品が「何を」実装しているか理解できても，それが他の部品から「いつ」「どのように」利用されているか十分に想定できなければ，再利用を行っても不具合を検出する恐れがある．よって，ある機能を実装している部品を再利用したいと考えたとき，個々の部品だけでなく，機能を実装している部品が相互に「いつ」「どのように」利用しているかという情報を加えて開発者に示す必要があると考えられる．

そこで本節では，動的情報を利用した部品評価手法 (*Dynamic Component Rank* **法，以下** *DCR* **法**) の提案を行う．ソフトウェア実行時に得られる動的支配関係を用いることで，単に各部品間の呼び出し関係の推移も得られるだけでなく，部品の支配関係を定量化した値が得られ，対象システムの振る舞いを理解するのに有効な手法であると考えられる．この提案手法をもとに，Java アプリケーションを対象として，部品評価値を求めるシステムを実現し，求められる値の妥当性を確認する．

また動的情報を利用した部品評価手法として，求めた部品評価値でフィルタリングを行い，重要な部品間のみのシーケンス図を作成するツールを実現する．これにより注目すべき部品を絞り込んで開発者に表示することで，ソフトウェア理解・再利用を支援することが可能となる．

以降，5.2 節において提案手法である DCR 法について説明を行う．5.3 節では提案手法に基づいて評価を行う DCR システムを紹介する．5.4 節では，DCR システムを用いた評価実験を行い，動的情報を利用した部品評価値の利用例としてシーケンス図の自動生成について例を挙げて説明する．

5.2　動的情報を利用した部品評価値計算法 (DCR 法）の提案

4 節における利用関係を用いた部品グラフから部品を評価する手法をもとに，提案手法では支配部品グラフを入力とし，繰り返し計算により各頂点の重みを求めることで，頂点に対応する部品の評価値を定める．以下では支配部品グラフについて説明し，解析の概略を示す.

5.2.1　動的支配関係

一般に**ソフトウェア部品**(Software Component) とは，再利用できるように設計された部品とされている [21]．本論文ではより一般的に，開発者が再利用を行う単位をソフトウェア部品，あるいは単に部品と呼ぶ.

また，ソフトウェア部品間には実行時互いに呼び出す，呼び出されるという 2 項関係が存在する．この関係のことを**支配関係**と呼ぶ.

さらに，**動的支配関係**とは，ソフトウェア実行時に生じる，部品の呼び出し関係のことをいう．入力データに依存して実行経路も異なるため，実際に実行時に呼び出された部品にのみ動的支配関係が存在するため，実行によって部品間の動的支配関係は異なる．また，静的な利用関係が存在する部品でも，実行時に呼び出されなければ，利用関係は存在しないものとして扱う.

5.2.2　動的支配関係を用いた支配部品グラフ

ソフトウェア部品を頂点，動的支配関係を呼び出される部品から呼び出した部品への有向辺としたものを，**支配部品グラフ**(*Domination Component Graph*) と呼ぶ．図 5.1 にその例を示す.

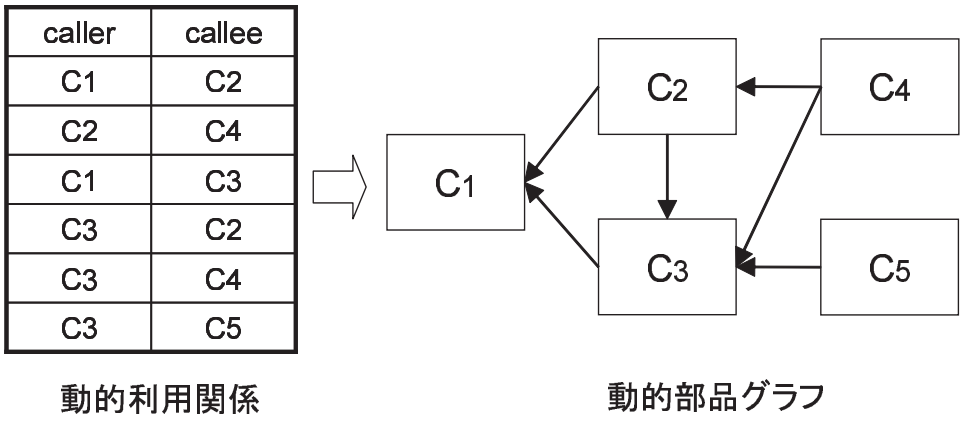


図 5.1: 支配部品グラフの例

動的支配関係では，実行されるまでどの部品が支配されるかわからず，呼び出しイベントが発生して初めて，部品は支配される．また，支配する部品は，支配される部品の実行

結果を用いて機能を実装する．そこで提案手法では，呼び出した部品の実行結果を利用して，機能を実装するというモデルから，支配部品グラフ作成の際，有向辺を支配される部品から支配する部品へ引くものとしている．

5.2.3 支配部品グラフを用いた部品評価値の計算手順

1. ソフトウェアを実行し，動的支配関係を取得
2. 動的支配関係をもとに，支配部品グラフを生成
   (部品評価値計算が必ず収束するようにするための補正を加えた擬似辺も含む.)
3. 支配部品グラフの各頂点の和が 1 になるように重みを分配する
4. 支配部品グラフの各有向辺の重みを求める
   (各頂点の重みを，その頂点から出ていく辺で分配する)
5. 頂点に入ってくる辺の重みの総和を，その頂点の重みとして再定義する
6. 頂点の重みが収束するまで，4,5 を繰り返し計算する
7. 収束した頂点の重みを，その頂点に対応する部品の評価値として出力する

部品評価値計算例を，図 5.2 に示す．部品 C_1，C_2，C_3 の重みは，それぞれ 0.4，0.2，0.4 で収束しており，この値を部品評価値として出力する．

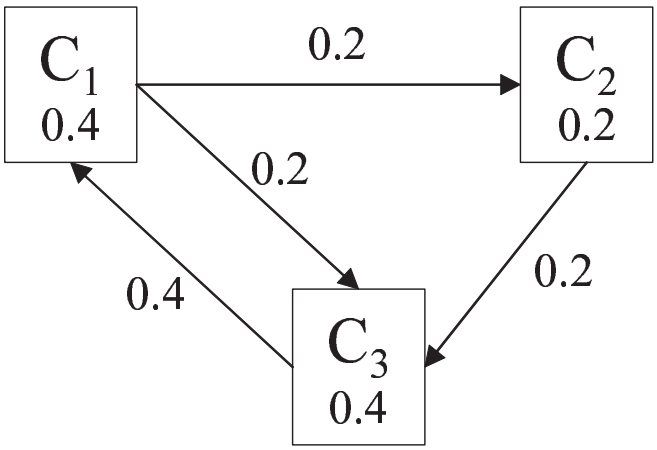


図 5.2: 支配部品グラフを用いた部品評価値の計算例

5.3 Dynamic Component Rank システム

5.2 節で説明した動的情報を利用したソフトウェア部品評価手法をもとに，オブジェクト指向言語 Java[15] を対象として，我々の研究グループにおいて構築したソフトウェア部

品評価システム「**Component Rank System**」(以降，CR システム) に対して機能拡張を行い，動的支配関係による部品評価システム **DCR システム**を構築した．以降，DCR システムにおける Java の概念への対応，システムの構成，について説明を行う．

5.3.1 Java の概念への対応

DCR 法を Java に適用する際の，DCR システムにおける Java の概念への対応を以下に示す．

- Java の概念への対応

  部品: クラスを部品の単位とする．
  ただし，Java は原則として 1 つのソースコードファイルには 1 つのクラスを記述するため，ソースコードファイルも部品単位とみなすことが可能である．

  利用関係: クラスの継承，インターフェースおよび抽象クラスの実装，メソッドの呼びだし，フィールド参照とする．

  分配率: 利用関係の種類・頻度にかかわらず，分配率は全て等しくする．

  辺と擬似辺の重みの配分比率 p: p として 0.85 を採用する．

5.3.2 システム構成

DCR システムによる部品評価値計算手順を以下に示し，各解析部との対応を，以下に示す．図 5.3 は DCR システムの構成図である．

1. プログラムを実行し，実行履歴から，利用関係を抽出
   - 利用関係解析部
2. 部品間の利用関係から，部品グラフを生成
   - 部品グラフ生成部
3. DCR 法による部品評価値計算し，順位付けを行い出力
   - 部品評価値計算部

Java 実行ファイル (クラスファイル) を JavaVM 実行する際，利用関係解析部 (Profiler) を用いて実行履歴を取得し，利用関係 (prof file) を抽出する．次に抽出された利用関係から部品グラフ生成部 (Dynamic Relation Analyzer) が部品グラフを生成する．最後に生成された部品グラフから部品評価値計算部 (Ranker) で部品評価値を計算し，出力する．以降，利用関係解析部，部品グラフ生成部，部品評価値計算部の説明を行う．

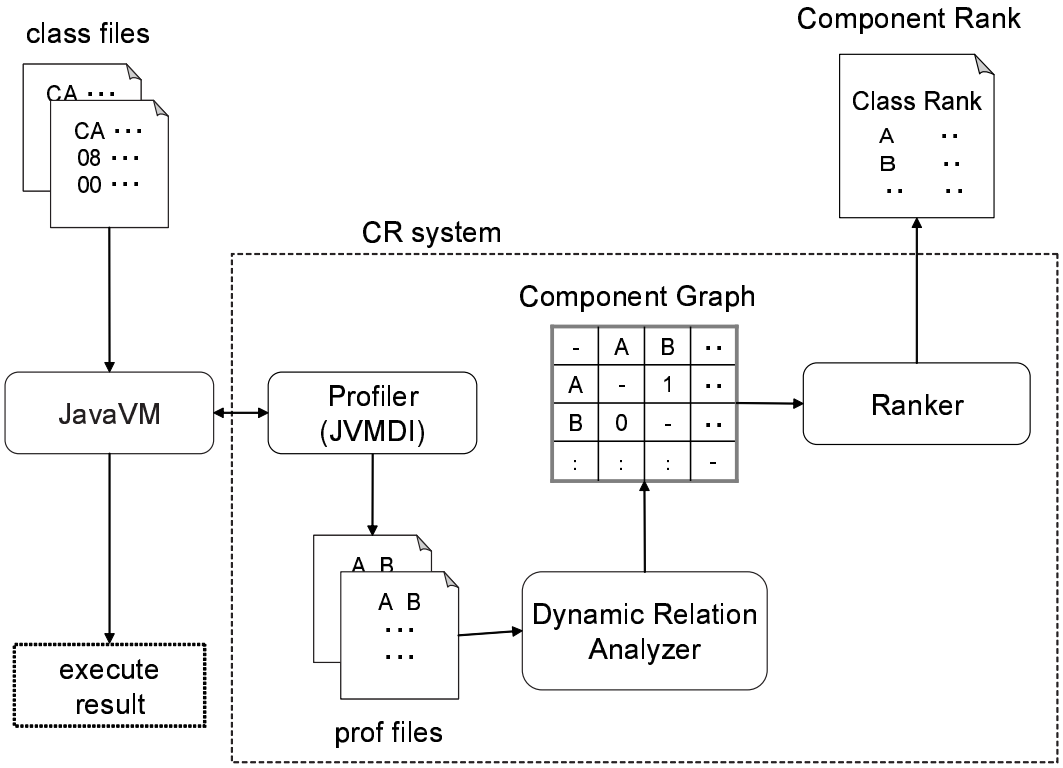


図 5.3: DCR システムの構成

利用関係解析部

Java では，Java Virtual Machine Debug Interface (JVMDI) と呼ばれる，VM によって実装されているネイティブインタフェースが用意されており，これを利用することで容易にメソッド呼び出しの情報を得ることができる [45].

JVMDI のクライアントは、アプリケーションプログラム内で発生する多くのイベントについての通知を受けることができ，必要なイベントを監視することで利用関係を取得する．5.3.1 節で定義した，Java における利用関係の取得と，イベントとの関係を以下に示す.

- クラス継承
  - JavaVM ではクラスインスタンスが生成される際，インスタンス初期化メソッド $\langle init \rangle$ が実行される．サブ (子) クラスの $\langle init \rangle$ メソッドが実行されると，そのメソッドからスーパー (親) クラスの $\langle init \rangle$ メソッドが実行される．よって，クラス継承の利用関係の取得は，メソッド呼び出しの利用関係と同じ処理で取得可能である.
- インタフェースならびに抽象クラスの実装
  - 新しいクラスが実行されるには，必ずそのクラスがロードされる．この時，VM 内でクラスロードイベント (JVMDI_EVENT_CLASS_LOAD) が発生する．JVMDI

ではこのクラスロードイベント発生時に，クラスの情報を取得することができ，実装しているインタフェース，抽象クラス名を取得できる．

- メソッド呼び出し
  - メソッドが呼び出される時には，メソッド呼び出しイベント (JVMDI_EVENT_METHOD_ENTRY) が，終了するときには，メソッド終了イベント (JVMDI_EVENT_METHOD_EXIT) が発生する．これらイベントを監視し，メソッドが呼び出される毎に，スタックの先頭にその情報を保存し，メソッド終了時に，スタックの先頭を削除する．スタックを用意することで，新しくメソッドが実行された際に，スタックの先頭がそのメソッドを呼び出したクラス (メソッド) であることがわかり，支配関係の取得が行える．また，スレッド毎にスタックを用意することで，マルチスレッドに対応した支配関係の取得も可能である．
- フィールド参照
  - フィールドがアクセスまたは変更される際，フィールドアクセスイベント (JVMDI_EVENT_FIELD_ACCESS) が発生する．この時，メソッド呼び出しで作成したスタックの先頭クラスから，アクセスされたフィールドのクラスへの支配関係の取得が可能である．

これらの支配関係は，プログラムの開始 (JVMDI_EVENT_VM_INIT) から終了 (JVMDI_EVENT_VM_DEATH) までの間に取得される．支配関係とそれが取得できる VM イベントとの対応表を, 表 5.1 に示す.

表 5.1: 支配関係と VM 上のイベントとの関連

支配関係	イベント名	発生方法
クラスの継承	JVMDI_EVENT_METHOD_ENTRY	メソッドが呼び出される
インターフェース・抽象クラス実装	JVMDI_EVENT_CLASS_LOAD	クラスがロードされる
メソッド呼び出し	JVMDI_EVENT_METHOD_ENTRY JVMDI_EVENT_METHOD_EXIT	メソッドが呼び出される メソッドが終了する
フィールド参照	JVMDI_EVENT_FIELD_ACCESS	フィールドの参照・変更
プログラムの開始	JVMDI_EVENT_VM_INIT	VM の初期化が完了する
プログラムの終了	JVMDI_EVENT_VM_DEATH	VM が終了する

また，JVMDI を利用したプロファイリング機能の実装構成図は図 5.4 のようになる．

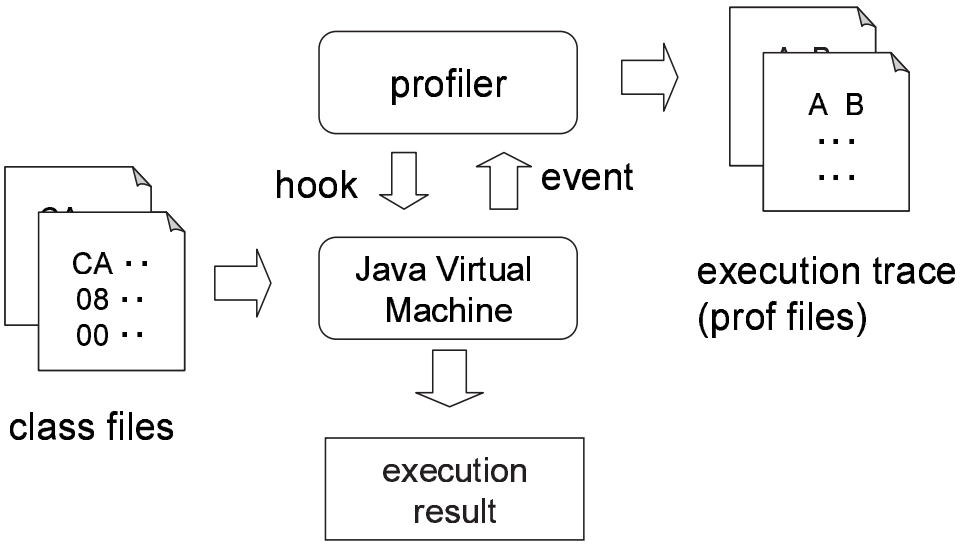


図 5.4: JVMDI を利用したプロファイリング機能の実装構成図

部品グラフ生成部

部品グラフは，$n \times n$ の行列 D 上 (部品数 n) に表現する．部品 C_i が部品 C_j を利用するとき，行列 D(i,j) の値を 1 とし，部品 C_i が部品 C_j を利用しないとき，行列 D(i,j) の値を 0 とする．

- 部品グラフの生成

  for($i = 1; i <= n; i++$)
  for($j = 1; j <= n; j++$)
  if (C_i が C_j を利用)
  $D(i, j) = 1$
  else
  $D(i, j) = 0$

部品評価値計算部

提案手法は，5 節における手法と同様に，D を転置した行列である D^t に対して，既約であることを保証するために後述する補正を加えた後に，実行が開発者が部品をどのように参照するかをマルコフ連鎖 [5] でモデル化したものである．この場合も，D^t における絶対値最大の固有ベクトルを求めることで，定常分布を求めることができるが，行列 D^t の絶対値最大の固有値は 1 であるため，累乗法 (power method) を用いて適当な初期ベクトルに対して繰り返し D^t を掛けることで，近似解を容易に求めることができる．

5.4 評価実験

提案手法を実現した DCR システムを用いて評価実験を行った．以降，実験結果ならびに考察について述べる．

5.4.1 実験内容

対象プログラムとして，Java 2 Software Development Kit, Standard Edition 1.4.0 付属の j2sdk1.4.0_01.demo.jfc.Notepad と，javac を選んだ．前者プログラムは，19 クラス (内部クラスも含む) から成り立つ，単純なエディタアプリケーションであり，後者プログラムは，全 159 クラスから成り立つ，Java コンパイラである．図 5.5 はこの前者 Notepad プログラムのスナップショットである．

動的支配関係の抽出を行うには，対象プログラムを実行する必要がある．Notepad アプリケーションの実行を行う際，以下の実行方法を用いた．

1. いくつかの機能をランダムに実行
2. 「ファイルを開く」のみ実行

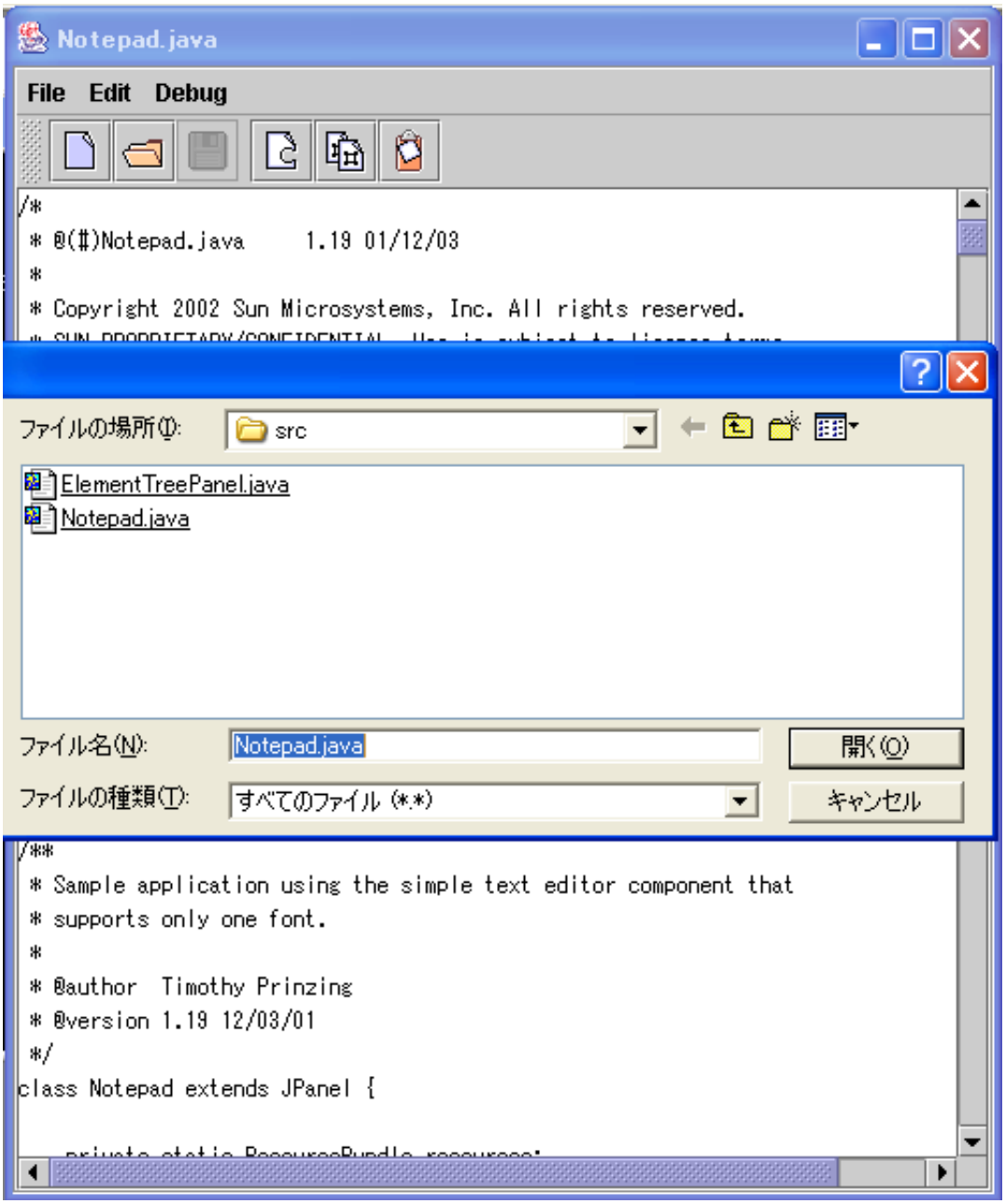


図 5.5: Notepad アプリケーションのスナップショット

3. 「コピー＆ペースト」のみ実行

また，javac プログラムの実行には，次の引数を与えた．

1. 正常にコンパイルが終了するプログラム
2. 構文エラーを出力し終了するプログラム
3. 存在しないプログラム
4. 引数に何も与えない

5.4.2 実験結果

Notepad アプリケーションにおける 3 種類の実行方法により得られた部品評価値ならびに，その部品 (クラス) が利用する部品数を，それぞれ表 5.2，表 5.3，表 5.4 に示す．また javac の 4 種類の実行方法により得られた部品評価値ならびに，その部品が利用する部品数を，それぞれ上位 15 位まで表 5.5，表 5.6，表 5.7，表 5.8 に示す．

順位（DCR）は提案する部品評価値による評価順位，順位（利用）は他の部品の利用回数による順位をそれぞれ表す．なお，部品評価値は，従来小数で表示されるが，わかりやすく表示するため 1 億倍した値を記述している．また，$\langle System \rangle$ クラスは，ユーザが記述したクラスではなく，VM 内部で実行されるクラス (例えば，クラスをロードする ClassLoader クラスや，イベントを扱う EventHandler クラスなど) を表している．

5.4.3 考察

計算手法の有効性

実験結果より，利用する部品数の多い部品ほど上位に位置することがわかる．しかし，javac で上位に位置する部品に注目した場合，決して利用部品数が多いほうが評価値が高いわけではないことがわかる．特に，com.sun.tools.javac.Main クラスは，全ての実行方法で利用部品数が 2 個であるにも関わらず，上位に位置している．com.sun.tools.javac.Main クラスは，この javac プログラムを起動する際，1 番初めに呼び出されるクラスであり，このクラスが存在しなければ，プログラムが実行されることはない．よって，プログラム内で重要な部品の 1 つであると考えられる．しかし，単純に利用部品数だけで部品の評価を行ったならば，com.sun.tools.javac.Main クラス 50 位前後に位置する．

提案手法を用いた部品評価値では，利用部品数に加え，利用している部品の重要度も加味しているため，ほとんど他の部品を利用しないようなシステムの末端で利用される部品 (クラスの) 評価値をおさえると共に，利用数は少ないが，プログラムを実行する上で重要となる部品の評価値を高くなる．よって，本提案手法は，機能の再利用を実現する上で，有効な手法であると考えられる．

今回の実験では，対象プログラムが小規模なプログラムであったため，大規模プログラムに適応した場合においても，有効な手法であるか検証する必要がある．また，他の部品評価手法と比較する必要もあると考えられる．

表 5.2: 部品評価順位：Notepad：ランダム

順位（DCR）	クラス名	評価値 ($\times 10^8$)	利用数	順位（利用）
1	$\langle System \rangle$	16207350	16	1
2	Notepad	13356511	12	2
3	Notepad$UndoAction	5732073	5	3
4	Notepad$OpenAction	5361253	4	4
5	Notepad$ShowElementTreeAction	5063501	3	6
6	Notepad$UndoHandler	4803297	3	6
7	ElementTreePanel	4740721	4	4
8	Notepad$3	4732369	3	6
9	Notepad$FileLoader	4731713	2	10
10	Notepad$2	4358630	2	10
11	ElementTreePanel$ElementTreeModel	4306154	3	6
12	ElementTreePanel$1	3811945	2	10
13	Notepad$NewAction	3430323	1	13
14	Notepad$RedoAction	3189917	1	13
15	Notepad$ExitAction	2695707	0	15
15	Notepad$ActionChangedListener	2695707	0	15
15	Notepad$1	2695707	0	15
15	Notepad$StatusBar	2695707	0	15
15	Notepad$AppCloser	2695707	0	15
15	ElementTreePanel$2	2695707	0	15

実行方法による部品評価値の変化について

実行方法が異なれば，当然動的な支配関係も変化する．実行方法がどの程度部品評価値に影響を与えているか考察する．

Notepad プログラムの実行方法の 2 では「ファイルを開く」という動作を行っている．その時の部品評価値の結果では，Notepad$OpenAction クラスや Notepad$FileLoader クラスが，他の結果と比較して上位に位置していることがわかる．これらクラスは,「ファイルを開く」という機能の動作を行った際に，実際に呼び出されるクラスであり，部品評価値計算でこれら部品が上位に位置することは，正しい結果であると考えれる．

また，javac プログラムでも，エラーを出力して終了するプログラムを引数に与えて実行すれば，エラー出力を行う com.sun.tools.javac.v8.util.Log が正常に終了する場合と比較して，上位に位置していることがわかる．

実験を行ったサンプル数は少ないが，以上の実験結果よりでも，本提案手法を用いて部品評価を行うことで，実行方法に依存した部品評価値が得られることがわかる．この結果を用いることにより，特定の機能を実現している部品の再利用が可能になるのではないか

表 5.3: 部品評価順位：Notepad：ファイルオープン

順位（DCR）	クラス名	評価値 ($\times 10^8$)	利用数	順位（利用）
1	Notepad	28475829	12	1
2	$\langle System \rangle$	14920808	4	2
3	Notepad$OpenAction	10882916	3	3
4	Notepad$FileLoader	7715919	1	4
5	Notepad$ShowElementTreeAction	3800453	0	5
5	Notepad$UndoHandler	3800453	0	5
5	Notepad$UndoAction	3800453	0	5
5	Notepad$NewAction	3800453	0	5
5	Notepad$RedoAction	3800453	0	5
5	Notepad$ExitAction	3800453	0	5
5	Notepad$ActionChangedListener	3800453	0	5
5	Notepad$1	3800453	0	5
5	Notepad$StatusBar	3800453	0	5
5	Notepad$AppCloser	3800453	0	5

表 5.4: 部品評価順位：Notepad：コピー&ペースト

順位（DCR）	クラス名	評価値 ($\times 10^8$)	利用数	順位（利用）
1	Notepad	30736333	11	1
2	$\langle System \rangle$	14178601	3	2
3	Notepad$UndoHandler	12237346	3	2
4	Notepad$UndoAction	5178274	1	4
4	Notepad$OpenAction	5178274	1	4
6	Notepad$ShowElementTreeAction	4061397	0	6
6	Notepad$NewAction	4061397	0	6
6	Notepad$RedoAction	4061397	0	6
6	Notepad$ExitAction	4061397	0	6
6	Notepad$ActionChangedListener	4061397	0	6
6	Notepad$1	4061397	0	6
6	Notepad$StatusBar	4061397	0	6
6	Notepad$AppCloser	4061397	0	6

表 5.5: 部品評価順位：javac：正常終了

順位（DCR）	クラス名	評価値 ($\times 10^8$)	利用数	順位（利用）
1	com.sun.tools.javac.v8.Main	6230715	17	11
2	com.sun.tools.javac.v8.JavaCompiler	5401857	24	7
3	com.sun.tools.javac.v8.comp.Gen	3308156	36	1
4	com.sun.tools.javac.Main	2911060	2	53
5	com.sun.tools.javac.v8.comp.Attr	2844601	34	2
6	com.sun.tools.javac.v8.comp.Enter	2588095	28	3
7	com.sun.tools.javac.v8.comp.TransInner	2007288	27	4
8	com.sun.tools.javac.v8.code.ClassReader	1949322	26	5
9	com.sun.tools.javac.v8.comp.Enter$CompleteEnter	1821256	16	14
10	com.sun.tools.javac.v8.comp.TransTypes	1797646	26	5
11	com.sun.tools.javac.v8.tree.TreeMaker	1791180	19	10
12	com.sun.tools.javac.v8.parser.Parser	1779365	8	20
13	com.sun.tools.javac.v8.comp.Enter$MemberEnter	1699103	17	11
14	com.sun.tools.javac.v8.comp.Flow	1671419	21	8
15	com.sun.tools.javac.v8.comp.Symtab	1581438	15	15

表 5.6: 部品評価順位：javac：エラー終了

順位（DCR）	クラス名	評価値 ($\times 10^8$)	利用数	順位（利用）
1	com.sun.tools.javac.v8.Main	7410380	17	7
2	com.sun.tools.javac.v8.JavaCompiler	6999128	22	4
3	com.sun.tools.javac.v8.comp.Attr	3939811	34	1
4	com.sun.tools.javac.v8.comp.Enter	3435564	28	2
5	com.sun.tools.javac.Main	3429715	2	43
6	com.sun.tools.javac.v8.comp.Enter$CompleteEnter	2500022	16	10
7	com.sun.tools.javac.v8.code.ClassReader	2486254	26	3
8	com.sun.tools.javac.v8.comp.Flow	2466759	22	4
9	com.sun.tools.javac.v8.tree.TreeMaker	2444630	19	6
10	com.sun.tools.javac.v8.parser.Parser	2234598	8	16
11	com.sun.tools.javac.v8.comp.Enter$MemberEnter	2180470	17	7
12	com.sun.tools.javac.v8.comp.Symtab	2037506	15	11
13	com.sun.tools.javac.v8.comp.Resolve	1691420	15	11
14	com.sun.tools.javac.v8.util.Log	1516791	7	17
15	com.sun.tools.javac.v8.code.Symbol	1497920	11	13

表 5.7: 部品評価順位：javac：存在しないプログラム

順位（DCR）	クラス名	評価値 ($\times 10^8$)	利用数	順位（利用）
1	com.sun.tools.javac.v8.JavaCompiler	11287694	19	1
2	com.sun.tools.javac.v8.Main	10479894	17	3
3	com.sun.tools.javac.Main	4768826	2	23
4	com.sun.tools.javac.v8.comp.Symtab	4212854	15	5
5	com.sun.tools.javac.v8.comp.Enter	3954821	17	3
6	com.sun.tools.javac.v8.code.ClassReader	3149856	18	2
7	com.sun.tools.javac.v8.util.Log	2028882	5	12
8	com.sun.tools.javac.v8.comp.Enter$CompleteEnter	1966526	7	10
9	$\langle System \rangle$	1886880	1	42
10	com.sun.tools.javac.v8.tree.Tree$TopLevel	1771217	3	18
11	com.sun.tools.javac.v8.comp.Gen	1588532	9	6
12	com.sun.tools.javac.v8.code.Symbol	1586177	9	6
13	com.sun.tools.javac.v8.code.Type	1532599	9	6
14	com.sun.tools.javac.v8.code.Symbol$PackageSymbol	1457690	5	12
15	com.sun.tools.javac.v8.comp.Attr	1362777	8	9

表 5.8: 部品評価順位：javac：引数なし

順位（DCR）	クラス名	評価値 ($\times 10^8$)	利用数	順位（利用）
1	com.sun.tools.javac.v8.Main	28144312	15	1
2	com.sun.tools.javac.Main	10803013	2	2
3	com.sun.tools.javac.v8.Main	8341994	2	2
4	com.sun.tools.javac.v8.comp.Symtab	5643229	1	6
5	com.sun.tools.javac.v8.Main	3689524	2	2
5	com.sun.tools.javac.v8.Main	3689524	2	2
7	com.sun.tools.javac.v8.util.Log	3686118	1	6
7	com.sun.tools.javac.v8.comp.Enter$CompleteEnter	3686118	1	6
7	$\langle System \rangle$	3686118	1	6
10	com.sun.tools.javac.v8.Main$1	3182210	1	6
10	com.sun.tools.javac.v8.Main$2	3182210	1	6
10	com.sun.tools.javac.v8.Main$3	3182210	1	6
10	com.sun.tools.javac.v8.Main$4	3182210	1	6
10	com.sun.tools.javac.v8.Main$5	3182210	1	6
10	com.sun.tools.javac.v8.Main$6	3179750	1	6

と考えられる．

5.4.4　DCR 法の応用:シーケンス図の自動生成

動的支配関係モデルによる部品評価値の応用として，リバースエンジニアリングへの適応が考えられる．リバースエンジニアリングとは，通常のソフトウェア開発プロセスの逆の流れでソフトウェアの開発等をサポートしようとするものである．既存のソフトウェアを解析し，そのソフトウェアがどのようなモデルであるかを示すことで，ソフトウェア理解につながり，その結果再利用支援にもつながると考えれる．

部品評価値のリバースエンジニアリングへの応用として，**シーケンス図**(Sequence Diagram) への適応を行う．シーケンス図とは，部品間の利用関係を表すと共に，実行方法によって変化する動的な利用関係を同時に表現する関係図である．図 5.6 にシーケンス図の簡単な例を示す．

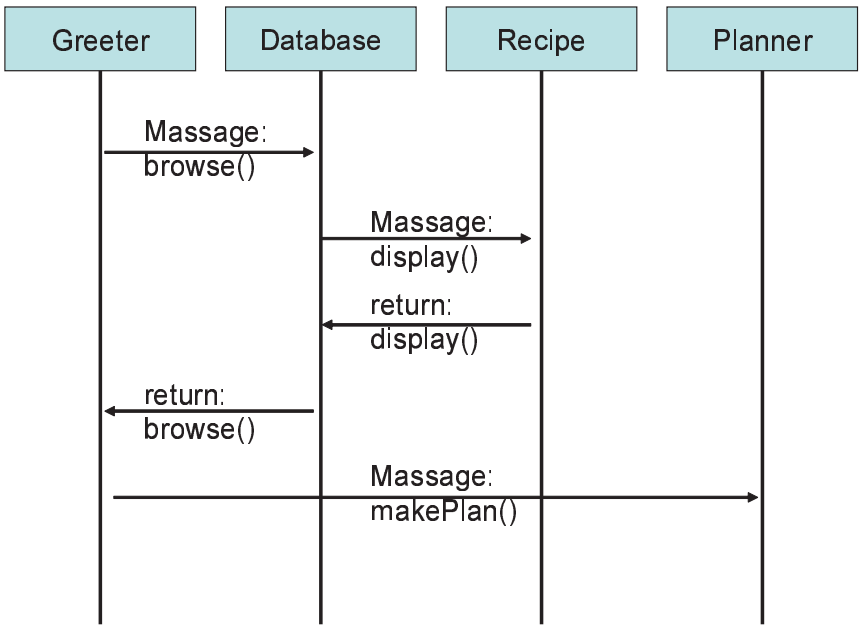


図 5.6: シーケンス図の例

この図に部品から上下に伸びている線は，時間軸を表し，時間軸から左右にラベルつきで伸びた有向辺は，部品間利用関係 (メッセージのやり取り) を表している．

単純に，ソフトウェアを実行した履歴だけからこのシーケンス図を作図したとする．小規模なソフトウェアの場合なら，部品数，部品間のメッセージのやり取りも少ないために，全ての部品間の利用関係を出力しても問題ない．しかし，少し大規模なソフトウェアになると，部品数，部品間のメッセージが増えるため，全ての部品間の利用関係を出力すれば，複雑な利用関係のためソフトウェア理解支援するどころか，かえって困難にしてしまう恐れがある．

そこで，提案手法で計算された部品評価値でフィルタリングを行い，重要な部品間のみ

シーケンス図上に表現する．部品 C_i に対応する頂点 v_i の重み $w(v_i)$ とし，フィルタリング値として w_limit が与えられたとする．この時 $w(v_i) \geq w_limit$ を満たす部品 C_i のみ，シーケンス図のノードとして出力する．また，部品間の利用関係については，$w(v_i) \geq w_limit$ かつ $w(v_j) \geq w_limit$ を満たす，部品 C_i と部品 C_j との利用関係のみ出力する．

5.4 節の実験で利用した，Notepad プログラムのファイルオープンのみ実行した結果についてシーケンス図を作図する．図 5.8 にフィルタリングを行っていないシーケンス図の一部を，図 5.7 に Notepad$FileLoader クラスの部品評価値 0.07715919 でフィルタリングを行ったシーケンス図の一部を示す．なお，部品の上の数字は，部品評価値 ($*10^8$) を表している．

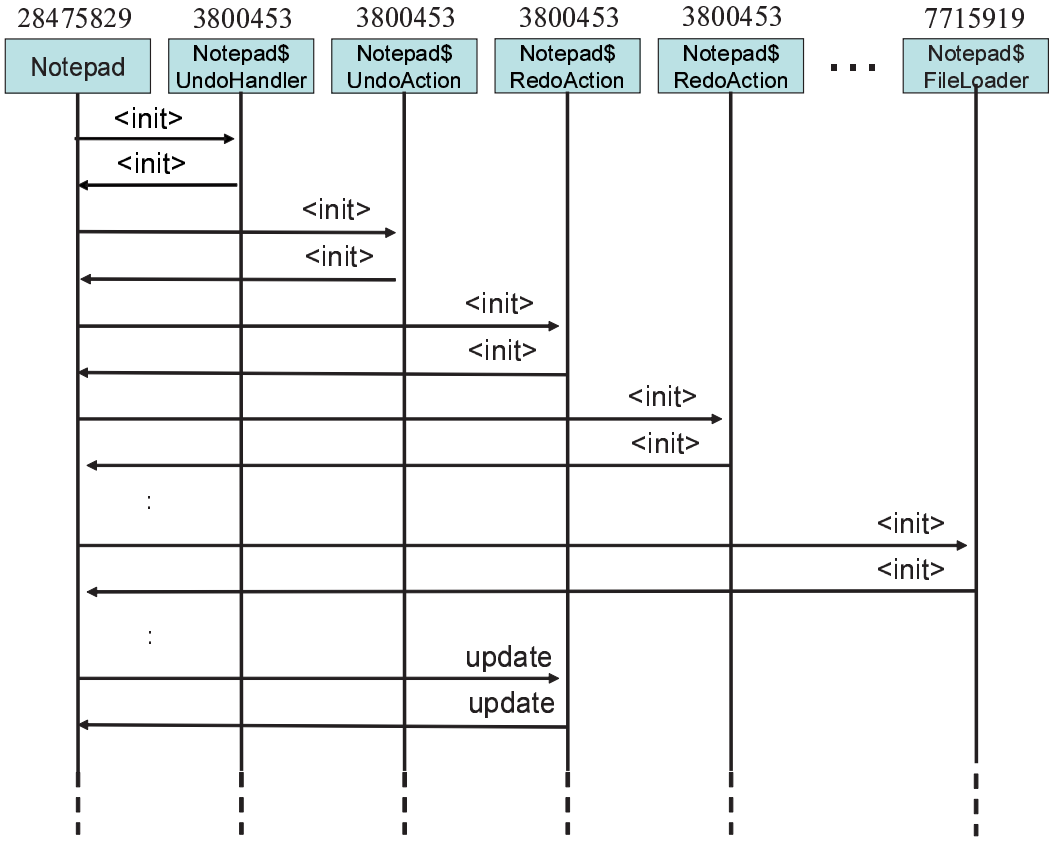


図 5.7: Notepad シーケンス図 (フィルタリングなし)

出力結果からもわかるように，フィルタリングを行ったシーケンス図では，3 つの部品ならびにそれら部品間の呼び出し関係のみ出力されている．適当な部品評価値でフィルタリグを行うことで，出力する部品を減らすことができ，開発者が注目すべき部品を絞り込める．

また，単純に表示する部品数を減らしているわけではなく，部品評価値でフィルタリングを行っているので，実行した機能実装に必要な部品を出力し，あまり重要でない部品を除くことが可能となる．Notepad プログラムのファイルオープン機能のみ実行した結果をフィルタリングした例では，Notepad，Notepad$OpenAction，Notepad$FileLoader クラス

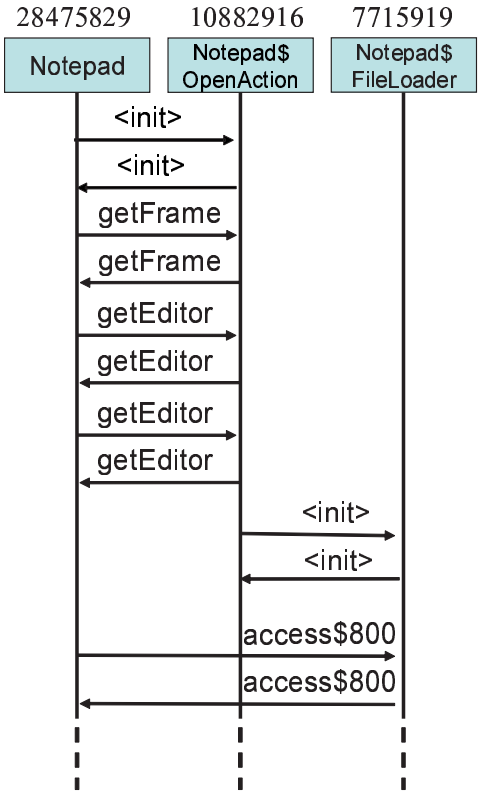


図 5.8: Notepad シーケンス図 (フィルタリングあり)

のみが出力されているが，ファイルオープンを機能実装している部品はこの 3 部品であり，開発者が注目したいはずの部品のみが出力されている．

今回の例では，部品数が少ないため機能実装に必要な部品のみが出力されている．ソフトウェアが大きくなれば，必ずしも機能実装に必要な部品のみが出力されるわけではないが，適当なフィルタリング値を選ぶことで，少なくとも重要でない部品を除いて出力することは可能であると考えられる．

以上のことから，適当な評価値でフィルタリングを行うことで，ユーザが注目すべき部品を絞り込め，結果ソフトウェア理解支援につながると考えられる．

しかし，部品評価値でフィルタリングを行う際，どのような値でフィルタリングを行うのか，という問題がある．今回実装したプロトタイプシステムでは，ユーザがその値を選ぶというかたちで実装を行っているが，実利用を考えたとき，対象プログラムがどのようなプログラムか全く知識がないユーザが，適当な値を選択できるのかどうかわからない．そのため，重要な部品がどれかを判断し，自動的にフィルタリングする部品評価値を決定する機構が必要であると考えられる．また，今回の実装方法では，部品評価値のみフィルタリングの対象としており，支配関係 (有向辺) の重みについては対象としていない．その結果，ループが存在するようなソフトウェアについては，何度も同じメッセージのやり取りが出力されてしまい，利用関係図が大きくなってしまう．そのため，辺の重みについてもフィルタリングを行う必要があると考えられる．

さらに，今回の実験結果は，比較的小規模なプログラムを対象としている．その理由として，実行履歴の保存に大きなコストがかかるため，今回実装を行ったプロトタイプシステムでは，大規模プログラムに適応できないためである．そのため，大規模なソフトウェアで同様の実験を行い，有効性を評価する必要があると考えられる．

DCR システムへの追加実装について

動的な支配関係と，部品評価値，フィルタリング値をもとに，シーケンス図を出力する機能を，CR システムに追加実装した．図 5.9 に出力結果を示す．

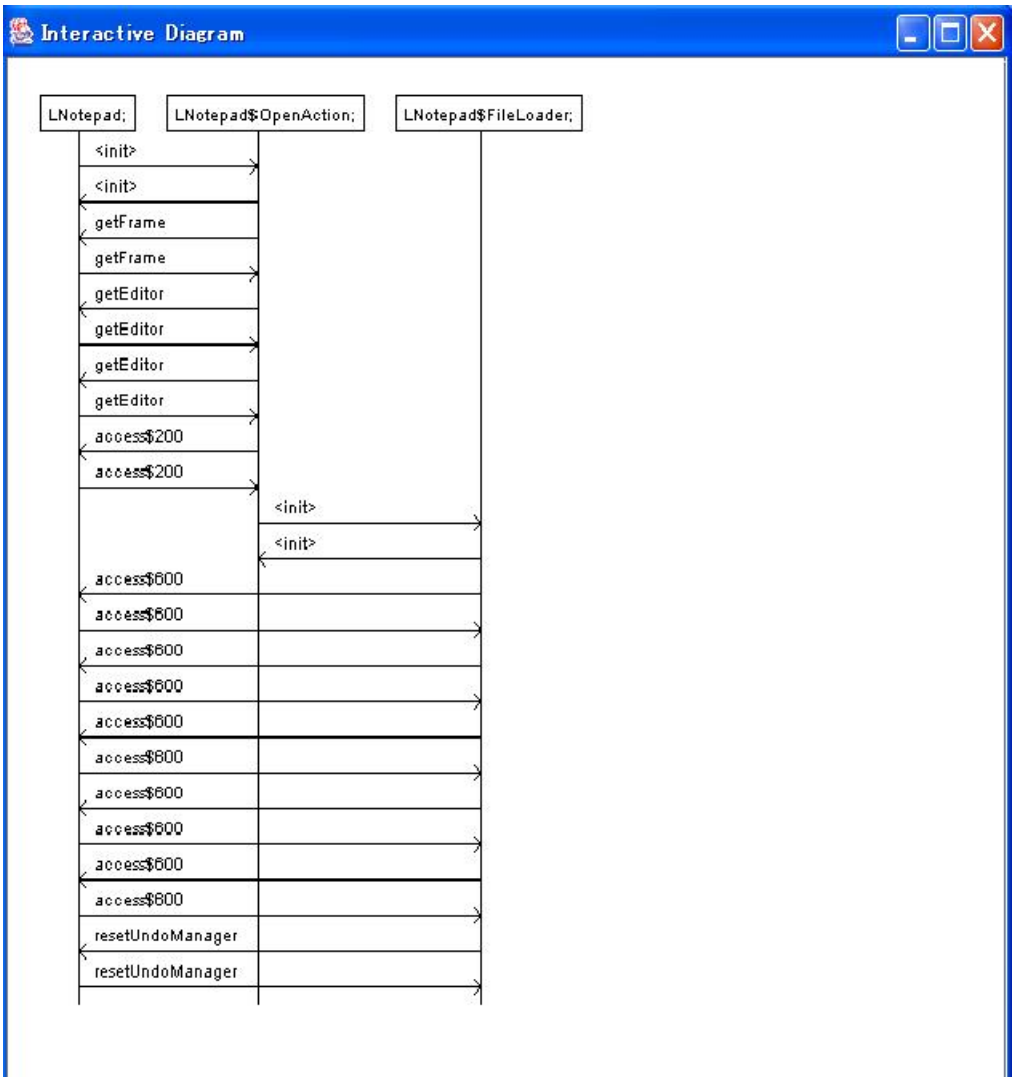


図 5.9: シーケンス図出力機能におけるシーケンス図の出力例

5.5　5 章のまとめ

本章では，動的な情報を利用した部品評価手法を提案した．提案手法を評価システムに実装し，実際に Java アプリケーションに適用実験を行い，本手法の有効性について確認を行った．また，部品評価値の利用方法として，シーケンス図への適用を行った．評価値でフィルタリングを行い作図することで，ソフトウェア理解支援につながることを確認した．適用実験では，プログラムの実行時に中心的な役割を果たしている部品が利用回数にかかわらず上位を占め，本手法が対象システムの振る舞いを理解するのに有効な手法であることを確認した．

本提案手法を用いることで，重要な部品の検索だけでなく，それら部品の相互利用関係も取得できるため，開発者は機能単位での再利用を行うことが可能となり，効率よく再利用が行えると考えている．

第6章　あとがき

6.1　まとめ

本論文では，情報漏洩解析手法，影響波及解析手法，ソフトウェア部品の再利用性評価手法の 3 つのプログラム解析技術に着目し，保守や再利用における支援を目的とした解析手法を提案および実現した．

まず，プログラムにおける入力値が持つ機密度から各出力文が出力しうる機密度を解析する手法として，プログラムスライスにおける手法を流用することによる情報漏洩解析の実現手法を提案した．

次に，プログラムの変更による被影響部分を特定する影響波及解析手法として，クラスのメンバ間の関係を表現する 2 種類のグラフを定義し，ユーザの目的に応じ探索ルールを選択して適用することによる，オブジェクト指向言語を対象とした影響波及解析手法を提案した．

また，再利用支援を目的とした部品検索システムにおける順位付け手法として，利用関係からソフトウェア部品を評価する手法を提案し，得られた Component Rank が利用実績として妥当であることを確認した．さらに，実行時に中心的な役割を果たす部品の抽出を目的として，動的情報を利用したソフトウェア部品評価手法を提案した．

6.2　今後の研究方針

本論文では，様々なプログラム解析技術の中から，情報漏洩解析手法，影響波及解析手法，ソフトウェア部品の再利用性評価手法の 3 つのプログラム解析技術に着目し，保守や再利用における支援を目的とした実現手法を提案および実現した．

情報漏洩解析手法，影響波及解析手法は本来，それぞれ，検証とテストケースの限定を目的として考案されていたが，解析結果情報をソースコード上で表示したり，目的に応じた解析を行うことで，開発および保守作業を支援することができると考えられる．論文においては，適用事例をもとに有効性を確認したが，現実のソフトウェア開発への適用を行うためには，実際の作業者による保守およびデバッグ工程における実験を通した評価を行う必要があると考えられる．

そのため，情報漏洩解析手法における今後の課題として，

- 他言語に対する情報漏洩解析アルゴリズムの実装
- 適用実験による情報漏洩解析手法の有効性の確認

が，影響波及解析手法における今後の課題として，

- Rapid Type Analysis[3] の利用による参照変数の指すオブジェクトの型の限定
- 現実のソフトウェア開発を想定した評価実験

が考えられる．

従来の再利用性評価手法は，部品単体の特性のみから評価を行っており，部品の利用実績については考慮されておらず、部品検索システムにおける利用にそぐわないものであった．本論文で提案した CR 法は，部品検索システムにおける利用を想定した手法である．現在我々の研究グループでは，ソフトウェア部品検索システム SPARS について研究を行っており，Java を対象として SPARS-J を実現した．SPARS-J においては，部品検索結果の表示順位として本論文で提案した Component Rank が用いられている．我々は Component Rank を用いることで効果的に部品を取得できると考えており，SPARS-J の検索性能評価を行うことで，有効性を検証したいと考えている．

また，CR 法は従来の再利用性評価手法において考慮されていない部分である，利用実績に関する評価を補完しており，ソフトウェア部品の再利用性評価に利用できるとも考えている．現実には，従来手法では再利用性が低いと評価されても，多くのシステムで再利用されている部品は数多く存在すると考えられる．従来の再利用性評価手法との結果の比較を行うことで，再利用性の評価精度向上を図りたいと考えている．

さらに，本論文で提案した CR 法における今後の課題として，

- さらに多くの部品への適用
- 継承，実装，メソッド呼びだしの各関係の部品グラフ上での重みづけの検討

などが挙げられる．

また，動的情報を用いた DCR 法における今後の課題として，

- より多くの大規模アプリケーションへの適応
- 実行履歴取得の効率化
- 他の部品評価手法との比較
- フィルタリングしたシーケンス図の有効性評価

などが挙げられる．

プログラムの理解支援とは，解析結果情報にもとづいてそれを効果的に提示することで，開発および保守作業を支援することをさす．実際のデバッグ作業において，プログラムスライスを用いることで，フォールト位置の特定が有効に行えることを確認した例が報告されている．本論文で提案している手法は，いずれもプログラムに記述されている内容から，クラス間の利用関係，メソッドの呼び出し関係，プログラム文中の制御およびデータフロー関係などを取り出すことで，プログラム中に存在する個々の要素（プログラム文，メソッドやメンバ，クラス）間の関係を抽象化する手法で，いずれも，保守や再利用などのプログラム理解支援を必要とする状況における利用目的を考察し，それに基づいてプログラム解析技術を利用した手法である．

近年大規模化しつつあるプログラムにおいては，プログラムを直接見ることで個々の要素間の関係を正確に把握することは不可能に近く，これらの関係情報を抽出し見やすい形式で出力することで開発者に情報を提供する仕組みが極めて重要となる．プログラムのソースコードを効果的に見せる研究や，プログラムの振る舞いを効果的に視覚化する研究がなされており，これらの手法を既存のプログラム解析技術と組み合わせることで，大規模ソフトウェアに対しても効果的に活用できるシステムが構築できると考えられる．

参考文献

[1] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman: “Compilers : Principles, Techniques, and Tools“, Addison-Weseley, 1986.

[2] J.Banâtre, C.Bryce and D.Le Mêtayer: “Compile-Time Detection of Information Flow in Sequential Programs”, Proc. of the 3rd ESORICS, LNCS 875, pp. 55–73, 1994.

[3] D. F. Bacon: “Fast and Effective Optimization of Statically Typed Object - Oriented Languages”, Ph.D. Thesis, Computer Science Division, University of California, Berkeley, December, UCB/CSD-98-1017, 1997.

[4] V. R. Basili, G. Caldiera, F. McGarry, R. Pajerski, G. Page and S. Waligora: “The software engineering laboratory - an operational software experience”, Proc. of ICSE14, pp. 370-381, 1992.

[5] G. Blom, L. Holst and D. Sandell, 森真 [訳]: “確率問題ゼミ”, シュプリンガー・フェアラーク東京, 1995.

[6] G. Booch: “ Object-Oriented Analysis and Design with Applications”, The Benjamin/Cummings Publishing, 1994.

[7] C. Braun: “ Reuse”, Encyclopedia of Software Engineering, Vol. 2, John Wiley & Sons, pp. 1055-1069, 1994.

[8] X. Chen, W. T. Tsai, and H. Huang: “Omega - an Integrated Environment for C++ Program Maintenance”, Proc. of the International Conference on Software Maintenance, pp. 114-123, Monterey, USA, 1996.

[9] S. R. Chidamber and C. F. Kemerer: “A metrics suite for object-oriented design”, IEEE Trans. on Software Eng., Vol. 20, No. 6, pp. 476-493, 1994.

[10] D.E.Denning: “A Lattice Model of Secure Information Flow”, Communication of the ACM,Vol. 19, No. 5, pp. 236–243, 1976.

[11] D.E.Denning and P.J.Denning: “Certification of Programs for Secure Information Flow”, Communication of the ACM,Vol. 20, No. 7, pp. 504–413, 1977.

[12] S. Eisenbach and C. Sadler: “Changing Java Programs”, Proc. of the International Conference on Software Maintenance (ICSM 2001), pp. 479-487, Florence, Italy, 2001.

[13] L. H. Etzkorn, W. E. Huges Jr., C. G. Davis: “Automated reusability quality analysis of OO legacy software”, Information and Software Technology, Vol. 43, Issue 5, pp. 295-308, 2001.

[14] L. H. Etzkorn, J. Bansiya, C. G. Davis: “Design and code complexity metrics for OO classes”, Journal of Object-Oriented Programming, Vol. 12, No. 1, pp. 35-40, 1999.

[15] J. Gosling, B. Joy, and G. Steele, 村上 雅章 [訳], “The Java 言語仕様”

[16] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel: “An Empirical Study of Regression Test Selection Techniques”, Proc. of the 20th International Conference on Software Engineering, pp.188-197, Kyoto, Japan, 1998.

[17] W. Hetzel: “The Complete Guide to Software Testing”, QED Information Sciences, Wellsley, Mass., 1984.

[18] T. H. Haveliwala: “Efficient Computation of PageRank”, Stanford Technical Report, 1999.

[19] S. Isoda: “Experience report on a software reuse project: Its structure, activities, and statistical results”, Proc. of ICSE14, pp. 320-326, 1992.

[20] D. Jackson, and M. Rinard: “Software Analysis: A Roadmap”, The Future of Software Engineering, pp. 135–145, 2000.

[21] I. Jacobson, M. Griss and P. Jonsson: “Software Reuse”, Addison Wesley, 1997.

[22] Y. K. Jang, H. S. Chae, Y. R. Kwon, and D. H. Bae: “Change Impact Analysis for A Class Hierarchy”, Proc. of the Asia Pacific Software Engineering Conference (APSEC’98), pp. 304-311, Taipei, Taiwan, 1998.

[23] B. Keepence and M. Mannion: “Using patterns to model variability in product families”, IEEE Software, Vol. 16, No. 4, pp. 102-108, 1999.

[24] A. Krishnaswamy: “Program Slicing: An Application of Object-Oriented Program Dependency Graphs”, Technical Report TR94-108, Department of Computer Science, Clemson University, 1994.

[25] D. Kung, J. Gao, P. Hsia, and F. Wen: “Change Impact Identification in Object Oriented Software Maintenance”, Proc. of the International Conference on Software Maintenance, pp. 202-211, Victoria, Canada, 1994.

[26] L. Li, and A. J. Offutt: “Algorithmic Analysis of the Impact of Changes on Object-Oriented Software”, Proc. of International Conference on Software Maintenance (ICSM ’96), pp. 171-184, Monterey, USA, 1996.

[27] F. Narin, G. Pinski, and H. H. Gee: “Structure of the Biomedical Literature,” Journal of the American Society for Information Science, Vol. 27, No. 1, 25-45, 1976.

[28] L. Page, S. Brin, R. Motwani, T. Winograd: "The PageRank Citation Ranking: Bringing Order to the Web", http://www-db.stanford.edu/ backrub/pageranksub.ps

[29] J. Palsberg and Peter K: "Trust in the lambda-calculus", Proc. of the 1995 Static Analysis Symposium, pp.314–329, 1995.

[30] G. Pinski and Francis Narin: "Citation influence for journal aggregates of scientific publications: Theory, with application to the literature of physics", Information Processing and Management, Vol. 12, N0. 5, pp. 297-312, 1976.

[31] Pressman, R.S: "Software Engineering A Practitioner's Approach, fourth edition", 1997.

[32] G.Purnul: "Database Security", Advances in Computers(M.Yovits Ed.),Vol. 38, pp. 1–72, 1994.

[33] K. Rangarajan, P. Eswar, and T. Ashok: "Retesting C++ Classes", Proc. of the Ninth International Software Quality Week, San Francisco, USA, 1996.

[34] T. Reps, S. Horwitz, M. Sagiv and G. Rosay: "Speeding up Slicing", Proc. of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering, pp. 11–20, 1994

[35] D.J. Robson, K.H. Bennet, B. J. Cornelius and M. Munro: "Approaches to Program Comprehension", J.System Software, Vol. 14, No. 1, 1991.

[36] G. Rothermel and M. J. Harrold: "Selecting Regression Tests for Object-Oriented Software", Proc. of the International Conference on Software Maintenance, pp.14-25, Victoria, Canada, 1994.

[37] B. G. Ryder and F. Tip: "Change Impact Analysis for Object-oriented Programs", Proc. of the Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001), pp.46-53, Snowbird, USA, 2001.

[38] M. Weiser: "Program slicing", Proc. of the 5th International Conference on Software Engineering, pages.439–449, San Diego, California, 1981.

[39] http://www.antlr.org/ , "ANTLR Website"

[40] http://math.nist.gov/javanumerics/, "JAMA : A Java Matrix Package"

[41] "Google," http://www.google.com/

[42] http://java.sun.com/j2se/1.3/docs/tooldocs/win32/javac.html, "javac - Java programming language compiler."

[43] http://java.sun.com/j2se/1.3/, "The Java^{TM} 2 Platform, Standard Edition."

[44] http://www.jedit.org/, "jEdit - Open Source programmer's text editor."

[45] “JavaTM Platform Debugger Architecture” Sun Microsystems, http://java.sun.com/j2se/1.4/ja/docs/ja/guide/jpda/index.html

[46] http://sourceforge.net/, “SourceForge”

[47] http://java.sun.com/, “The Source For Java(TM) Technology”, Sun Microsystems

[48] 青木淳: オブジェクト指向システム分析設計入門, ソフトリサーチセンター, 1992.

[49] 青山幹雄, 中所武司, 向山博: “コンポーネントウェア”, 共立出版, 1998.

[50] 國信茂太, 高田喜朗, 関 浩之, 井上克郎: “束構造のセキュリティモデルに基づくプログラムの情報フロー解析”, 電子情報通信学会技術研究報告, Vol. 100, No. 472, pp. 25-32, 2000.

[51] 國信茂太, 高田喜朗, 関 浩之, 井上克郎: “束構造をもつセキュリティクラスに基づく再帰的プログラムに対する情報フロー解析法”, 電子情報通信学会論文誌 D-I, Vol.J85-D-I, No. 10, pp. 961-973, 2002.

[52] 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システムの試作”, 情報処理学会論文誌, Vol. 37, No. 4, pp. 536–545, 1996.

[53] 西 秀雄, 横森 励士, 井上克郎: “プログラム依存グラフを利用した情報漏洩解析手法の提案と実装”, 電子情報通信学会技術研究報告, SS2002-14, pp.19-24, 2002.

[54] 西本圭佑: “Java 言語について”, http://cappuccino.ne.jp/keisuken/ java/

[55] 馬場肇: “Google の秘密 - PageRank 徹底解説”, http://www.kusastro.kyoto-u.ac.jp/baba/wais/pagerank.html

[56] 汎道: “Think! Software Reuse ソフトウェアの部品化・再利用”, http://www5.airnet.ne.jp/handoh/Think_Reuse.htm

[57] 山本篤: “Google の基礎技術 PageRank について”, KTYY Seminar, 2001, http://aglaia.c.u-tokyo.ac.jp/ỹamamoto/KTYY_Seminar/20010220/

[58] 山本哲男, 松下誠, 神谷年洋, 井上克郎: “ソフトウェアシステムの類似度とその計測ツール SMMT”, 電子情報通信学会論文誌 D-I, Vol.J85-D-I, No.6, pp.503-511, 2002.

[59] 山本浩数, 鷲崎弘宜, 深澤良彰: “再利用特性に基づくコンポーネントメトリクスの提案と検証”, ソフトウェア工学の基礎ワークショップ (FOSE2001), 2001.

[60] 山本浩数, 鷲崎弘宜, 深澤良彰: “静的側面から見たソフトウェアコンポーネントの品質”, 電子情報通信学会技術研究報告, Vol. 101, No. 98, pp. 33-40, 2001.