依存解析に基づく
ソフトウェアの部品化再利用に関する研究

丸山 勝久

1999 年 3 月

目次

第 1 章

序論

1.1 ソフトウェア再利用と部品化

ソフトウェア再利用とは，ソフトウェアシステムを，新たに始めから構築するのでなく，既に存在するソフトウェアから作成することである [48, 77]．再利用により，ソフトウェア開発における生産性および信頼性の向上が期待できるため，従来から数多くの研究が行われている [18, 31, 48, 76, 91, 104]．これらの研究において，ソフトウェア開発の各工程 (分析，設計，実装，テスト，保守) に現れるさまざまなものが再利用可能であると考えられている [91]．例えば，ソースコードや仕様などといった開発時の生成物だけでなく，開発手順や開発構想なども再利用対象となる．本研究では，ソフトウェア開発における最終成果物 (artifacts) であるソースコードと，ソースコードを操作する際に用いる知識 (knowledge) を再利用対象として扱う．開発者は，ソースコードに付随する知識を参照することで，既存のソースコード，あるいは，その一部を変形させたものを再利用して，新しいソフトウェアを作成する．また，付随する知識を用いて計算機処理を行うことで，既存のソースコードを自動的に変形させ，開発者に提供することも可能である．ソースコードは，それを実装する言語や環境に強く依存するという欠点を持つ反面，記述の形式性が高いため，計算機で処理するのに適している．また，プログラムとして実行可能であるため，再利用時の効果が得やすいという利点を持つ．

一般的に，ソフトウェア再利用では，再利用対象を部品という形にまとめておく．特に，ソースコードを再利用対象とする際には，あらかじめ部品を作成 (抽象化) しておき，これらの部品を検索 (選択)，変更 (特殊化)，合成 (統合) することで，目的のプログラムを作成する [18, 48]．本論文では，ソースコードとそれに付随する知識を再利用可能な形にまとめたものを，部品 (ソースコード部品) と呼ぶ．このような再利用工程は，組立て型アプロー

チ (compositional approach) [18] と呼ばれ，手続き型プログラムに対する開発だけでなく，オブジェクト指向プログラミングや，近年注目されているコンポーネント組立て型ソフトウェア開発 (component-based software development) [4, 22, 101] においても適用されている．例えば，オブジェクト指向プログラミングでは，クラス (オブジェクト) が部品であり，継承 (implementation inheritance) や委譲 (delegation) により部品 (ソースコード) を再利用する [99, 113].

開発者の要求する部品が検索により見つかり，それらの部品が容易に変更および合成できるとき，再利用は成功し，新規のソフトウェアを重複して作成する手間を省略することが可能である．また，部品化がうまく行われていると，機能の追加 (拡張) や削除が部品単位で独立に行うことが可能となる．このような場合，ソフトウェアを作成するための開発者の負担が大幅に軽減される．

1.2 問題

前節で述べたように，組立て型アプローチによるソースコード部品化再利用では，再利用可能な部品があらかじめライブラリに用意されていることが前提となっている．さらに，部品は変更されることなく再利用され，それらの部品を組み合わせるだけで目的のソフトウェアを作成できることが理想的である．

しかしながら，あらゆる開発者が要求する部品，あるいは，あらゆるソフトウェアを開発する際に必要となる部品をすべて予測し，あらかじめライブラリに用意しておくことは不可能である．特に，開発者がエンドユーザの場合，再利用を行なう際の環境は動的に変化し，それに応じて要求する部品は多種多様に及ぶ．さらに，ある程度の大きさや機能を持った部品は，その用途が限られており，あらゆる場面で利用可能となることはない．開発者の要求する部品がライブラリに存在しない場合，再利用の効果が得られないことになる．そこで，部品提供者には，開発者や開発ソフトウェアの特性に合わせて，将来利用される可能性の高い部品を推測し，さまざまな種類の部品を数多く作成することが求められる．それにもかかわらず，従来の再利用手法の多くは，部品を利用する側の支援を目的としており，部品を提供する側の負担を考慮していない．現時点では，部品を自動的に作成するのに有効な手法は存在せず，部品作成過程の大部分を人間が経験的に行っているのが現状である．

また，変更することなく開発者の要求を満たす部品はごく一部に限られており，ライブラリに存在する部品を単純に組み合わせるだけでは，目的のソフトウェアを得ることができないことが多い．この場合，開発者は自分の要求に合わせて，部品を適時変更すること

になる．部品を変更するためには，その部品の機能や実装を理解した上で，変更が必要な範囲だけを特定したり，変更部分が他の部分へ与える影響を十分に考慮しなければならない．もし，変更されることを前提とせずに部品が作成されていたり，部品変更に対する支援がなければ，開発者は部品内部のソースコードの細部まで自分で調べなければならず，部品変更の負担は極めて大きくなる．また，人間が変更を行う場合，部品に誤りが混入する恐れがある．特に，部品変更を容認するホワイトボックス型再利用 (white-box reuse) では，部品の柔軟性や変更容易性，さらに，部品変更支援の欠如は大きな問題となる [83]．開発者の要求に合わせて自動的に部品を変更することが考えられるが [108, 6, 30, 13]，部品の変更箇所や変更方法を機械的に判断することは非常に困難である．どのようにして部品に柔軟性を持たせるのかという点，さらに，どのようにして部品変更を回避させるのかという点は，現在においても挑戦的な課題である．

ここで，実際のソフトウェア再利用においては，部品作成と部品変更以外の工程，例えば，部品検索や部品合成においても，さまざまな問題が存在する [18]．しかしながら，本研究では，部品作成と部品変更に関する問題に対象を絞り，これらの問題に対する解決を目的とする．

1.3 部品作成および部品変更に対する従来技術

前節で述べた部品作成と部品変更に関する問題より，本論文では，ソースコード部品化再利用に関して，次に示す 2 つの要求を扱う．

(1) 部品提供者が，再利用可能な部品を容易に作成できること．

(2) ソフトウェア開発者が，部品変更を回避，あるいは，部品を容易に変更できること．

要求 (1) に対して，既存のソフトウェアから再利用部品を切り出す方法が考えられている．文献 [1, 23, 25, 28] は，ソースコード再利用に関する問題の 1 つとして，再利用可能な部品の数が不足していることを指摘しており，数多くの部品を作成するためには，既存のソフトウェアから再利用部品を抽出することが有効であると述べている．しかしながら，既存のソフトウェアから再利用可能な断片だけを選択し，それらの断片を開発者が容易に再利用可能な形にまとめることは難しく，そのような作業は部品提供者の大きな負担となる．既存のソフトウェアから効果的な部品を抽出するためには，抽出対象のソフトウェアをどの程度理解することができるかどうかが鍵であり，リバースエンジニアリング (reverse engineering) 技術とリエンジニアリング (re-engineering) [17, 26] 技術が不可欠である．

要求 (2) に対しては，ドメイン分析 (domain analysis) [5] が有効であると考えられている．ソフトウェアが対象とするドメイン (分野) の性質やそのソフトウェアを開発する時に用いた知識などを，ドメイン分析によりモデル化して蓄積しておき，このドメインモデルを再利用する．過去のドメインモデルを参照することで，特定のドメインにおいて開発者が要求する可能性の高い部品を特定できると，そのような部品をあらかじめライブラリに用意することが可能となるため，部品変更を回避できる可能性が高い．しかしながら，開発者の要求や開発ソフトウェアの特性が頻繁に変化する場合，ドメインを分析することは極めて難しくなり，分析に費やす負担はかなり大きくなる．

また，要求 (2) に対して，部品にあらかじめ柔軟性を埋め込んでおくという観点から，型に関して部品をパラメータ化しておく方法が考えられている [34]．さらに，ソースコード部品を直接再利用するのではなく，あらかじめジェネレータに埋め込まれているソースコードの変換および合成に関するパターンを再利用することで，開発者の要求を記述したコード (仕様) からソフトウェアを生成する方法も考えられている [12, 81, 82]．パターンによる生成方法では，パターンの適用に成功すれば，開発者が部品変更を行う必要がない．しかしながら，部品 (ソースコードやパターン) をパラメータ化したり，ジェネレータを構築するためには，部品やジェネレータが利用されるドメインの特性を十分に理解し，部品に対する標準化を行う必要がある．よって，これらの手法は，その適用範囲が狭いという問題を持つ．また，柔軟性や変更容易性を部品に埋め込むことは，部品提供者の負担の増加につながる．さらに，開発者の要求するソフトウェアの種類の増加に合わせて，パラメータ化された部品やジェネレータを迅速に提供していくことは不可能に近い．

1.4 提案手法

本論文では，前節で述べた 2 つの要求に対して，プログラム依存解析に基づく部品作成手法および部品変更手法を論ずる．本研究では，部品作成に関する要求と部品変更に関する要求を切り離さずに考え，部品変更を前提とした上で容易に部品を作成する手法，および，部品作成時の負担を増加させずに部品変更を回避する手法の確立を目指す．特に，開発者の要求や開発ソフトウェアの特性が頻繁に変化するドメイン，すなわち，単純に従来の再利用手法を用いただけでは効果の得られないドメインを対象とする．コンピュータネットワークの発展により，不均質，かつ，流動的な環境でソフトウェアが開発および実行されるようになっており，本研究が対象とするようなドメインにおいても，ソフトウェア再利用技術の需要はますます高くなっている．

本論文で提案する部品作成手法および部品変更手法では，開発者が直接再利用するソースコードに対して，部品理解や部品変更に役立つ情報 (知識) を付加する．このような情報は，プログラム依存解析を適用することで，既存のソフトウェア，あるいは，開発者の部品変更から取得する．このため，再利用対象であるソースコードを，プログラム依存グラフ (program dependence graph: PDG) [29] と呼ぶ有向グラフ表現に変換して扱う．PDG とは，ソースコード内の各文 (命令) 間に存在する依存関係に着目し，ソースコードを抽象化したものである．

本論文では，部品作成および部品変更を支援するという観点から，以下に示す 4 つの手法を提案する．

(1) 区間限定プログラムスライシングを用いた部品作成手法.

(2) 変更履歴に基づく重み付き依存グラフを用いた部品洗練手法.

(3) 変更の模倣による部品の能動的変化手法.

(4) メソッド合成による新規クラスの自動生成手法.

以下に，それぞれの手法の概略を述べる．

(1) 区間限定プログラムスライシングを用いた部品作成手法

本論文では，まず，依存関係に基づくプログラムスライシング技法 [111] を用いて，既存のプログラムからソフトウェア部品を作成する手法を提案する．ソフトウェア部品とは，再利用の対象となるプログラム部品と部品理解に役立つ付加情報を合わせたものである．プログラムスライシング技法を部品抽出に適用するという考え方は従来から存在していたが，従来手法では，部品に含むソースコードを精度良く抽出できないという問題，および，部品理解や部品変更に有効な情報が部品抽出時に欠落するという問題があった．

これらの問題を解決するため，本論文では，従来のスライシング技法を，プログラムの任意の区間をスライシング対象として指定可能となるように拡張した順方向および逆方向区間限定スライスを定義する．これらのスライスを用いる部品作成手法は，作成した部品が実行可能性と抽出等価性という部品に不可欠な性質を満たすことを保証しながら部品抽出が可能であるという利点を持ち，従来の部品に比べて不要なコードを削除可能である．実行可能性とは抽出した部品が実行可能であること，抽出等価性とは部品がもとのプログラムを実行させたときと同じ結果を与えることを指す．また，本部品作成手法は直接再利用対象となる部

品本体ソースコードだけでなく，部品を実行するために必要な実行条件例と，部品がどのように利用可能であるかを示す利用例を与える．実行条件例と利用例をそれぞれ部品本体と組み合わせて得られるプログラムも実行可能性と抽出等価性という性質を満たすので，これらの付加情報は部品の実行動作や利用方法の確認を容易にし，部品理解や部品変更に役立つ．

(2) 変更履歴に基づく重み付き依存グラフを用いた部品洗練手法

前述の部品作成手法を用いて，既存のプログラムから部品を抽出することで，部品作成の負担が軽減できる．しかしながら，部品がどのような形で再利用されるのかを部品作成時に予測することは困難であり，抽出した部品が開発者の要求を常に満たすとはかぎらない．要求する部品が見つからない場合，開発者は部品の再利用をあきらめるか，自分の要求に合わせて類似の部品を変更することになる．よって，ソフトウェアの部品化再利用を促進するためには，部品作成を容易にするだけではなく，作成した部品が無変更あるいは少ない変更で再利用可能であることが重要である．

これに対して，前述の部品作成手法により抽出した部品を，開発者が行った過去の部品変更の履歴に基づき，将来再利用される可能性が高い部品に自動洗練する手法を提案する．本洗練手法では，部品ソースコードに対して，頻繁に再利用されたコード断片は開発者が将来も同じ形で再利用する可能性が高い部分，また，頻繁に変更を受けたコード断片は開発者が将来も同様の変更を行う可能性が高い部分であると仮定する．これらの仮定に基づき，PDG における節点間の依存関係の強度に重みを割り付け，開発者が変更後に再利用した部品の形および部品変更前後の部品の形の変化に基づき重みを変動させる．このようにして蓄積した依存関係矢印の重みを部品洗練の指標として用い，もとの部品ソースコードにおいて強い依存関係で結合されているコード断片を洗練後部品の候補として抜き出す．また，同時に，重みから計算した洗練コストにより洗練の妥当性を判定する．

本洗練手法により，ライブラリに存在する部品を，開発者あるいは開発ドメインに特化した形に自動的に変形することが可能である．よって，部品提供者が，微妙に異なる多種多様な部品をライブラリに用意しておく必要がなくなり，さらに，再利用時に開発者が部品を適時変更する手間も減少する．

(3) 変更の模倣による部品の能動的変化手法

本論文では，さらに，開発者がライブラリに存在しない部品を要求した場合においても，無変更で組み込み可能な部品が得られることを目的として，従来のソースコード部品とまっ

たく異なる性質を持つ能動的部品 (active component) を提案する．一般に，ソースコード再利用では，あらかじめ用意した部品を組み合わせることで目的のプログラムを作成する．このような部品化再利用においては，ライブラリ内の部品の機能は部品作成時に固定される．よって，さまざまな要求に応じるために，部品変更は不可欠である．また，開発ドメインの特性を完全に分析することは難しく，必要なすべての部品をあらかじめライブラリに用意しておくことは不可能である．

本論文で提案する能動的部品は，ソースコード再利用における従来の受動的な部品に変化メカニズムを組み込んだもので，開発者の要求や部品の存在する環境に応じて，部品検索時に自らその機能を変化させる．提案する部品変化は，1) 部品を分割することで，機能の一部を抽出する機能分割変化，2) 他の部品の変更事例を取り込み，機能の一部を交換することで，開発者の変更を模倣する機能交換変化の 2 種類である．機能交換変化における模倣メカニズムは，プログラムスライシングとグラフのラベル付き同形写像比較を用いた，プログラム合成アルゴリズム [40] により実現する．一般に，ソースコード部品の受けた変更を別の部品に単純に適用することは困難である．そこで，本変化手法では，変更箇所を特定する際に補スライスを，交換箇所を特定する際にラベルの抽象化を導入する．さらに，スライスとして閉じ込めた変更を統合する際に，依存関係を破壊しないように依存関係矢印の付けかえを行う．これらの拡張により，本変化手法における合成アルゴリズムは，従来のプログラム合成アルゴリズムに比べて，幅広いソースコードに適用可能である．

変更を自動化する手段として他の部品の受けた変更を模倣することで，過去に直接変更を受けた部品だけでなく，変更を直接受けていない部品も変更を学習することになる．よって，模倣メカニズムを備えた能動的部品を導入することで，再利用の要求が生じた際に，部品ライブラリ内にあらかじめ存在する部品と異なる機能を持つ部品を自動的に生成でき，開発者が行う部品変更の負担を軽減可能である．さらに，特性の分析が困難な開発ドメインでも環境に応じた部品を開発者に提供できる．

(4) メソッド合成による新規クラスの自動生成手法

前述した能動的部品の変化メカニズムは，手続き型プログラムを対象としている．本論文では，この変化メカニズムの対象範囲をオブジェクト指向プログラムにまで拡張し，既存クラスのメソッドとクラスの変更履歴から，開発者の要求するメソッドを持つ新規クラスを自動生成する手法を提案する．

いま，継承を用いてプログラムを作成した際，親クラスとその後継者クラスが持つメソッ

ド間の変更差異を特定し，その差異を変更履歴として親クラスに蓄積する．開発者が参照しているクラスにおいて，開発者の要求した呼出しメソッドが定義されていない場合，このメソッドに関連する変更差異を持つ類似クラスを見つけ，その差異を参照クラスに存在するメソッドと合成することで，呼出しメソッドを生成する．類似クラスは，ライブラリにおけるクラス階層関係とメソッドの名前およびシグニチャの等価判定に基づき決定する．また，変更差異の特定と合成は，オブジェクト指向プログラム対応の区間限定スライシングによる機能分割と，依存グラフの同形写像比較による等価機能特定を組み合わせたメソッド合成アルゴリズムにより実現する．

本手法により自動生成された新規クラスを再利用することで，継承におけるメソッドや変数の追加および再定義という開発者の負担が軽減される．

1.5 本論文の構成

本論文の構成は次の通りである．まず，2章で，本論文で提唱する各手法を説明する準備として，プログラム依存解析技術についてまとめておく．3章では，プログラムスライシングを用いた部品作成手法を提案し，実際のプログラムからソフトウェア部品を作成した評価実験に対する結果と考察を述べる．4章では，3章で作成した部品を，部品変更履歴に基づき洗練する手法を提案し，実際にソフトウェア部品を洗練した結果について考察を述べる．5章では，能動的部品の変化メカニズム，および，その具体的アルゴリズムと変化例を示し，部品作成と部品変更に関する負担軽減の効果について考察する．6章では，能動的部品変化メカニズムを，オブジェクト指向プログラムにおける新規クラスの自動生成に適用する手法を提案し，実際にクラスを自動生成する様子を示す．最後に，7章で，本論文のまとめを述べる．

第 2 章

プログラム依存解析技術

本論文で提唱する部品作成手法，部品洗練手法，部品変化手法，および，クラス生成手法はすべて，プログラム依存解析に基づいている．そこで，本章では，各手法で共通に用いる依存解析技術について述べる．まず，手続き型プログラムにおける制御の流れを表現する制御フローグラフ，および，プログラム文間の依存関係を表現するプログラム依存グラフの定義を示す．次に，依存関係に基づき，着目する変数に関連を持つ文集合をもとのプログラムから抜き出すプログラムスライシング技法，および，複数のプログラムを依存関係を保存したまま自動統合する技法について述べる．

2.1 制御フローグラフ

手続き型プログラムにおける基本ブロック間の制御の流れを表すために，制御フローグラフ (control flow graph: CFG) [3] を用いる．プログラム (ソースコード) の CFG とは，以下の条件を満たす有向グラフ $G_{CFG} = (N_{CFG}, E_{CFG}, start)$ である．

(1) N_{CFG} は基本ブロックを表す節点の集合である．本論文では，基本ブロックにプログラムの代入式 (代入文，入力文，出力文)，および，条件式 (分岐文，ループ文) を割り当てる． N_{CFG} は，プログラムを実行するときに最初に制御が移される開始節点 *start* と，制御が移る先を持たない終了節点 *stop* を必ず 1 つづつ含む．

(2) E_{CFG} は矢印集合である．節点 p から節点 q への矢印は，節点 p の文 (命令) を実行した後，プログラム制御がただちに節点 q に移る可能性があることを指す．このとき，節点 p を節点 q の直接先行 (predecessor)，節点 q を節点 p の直接後行 (successor) と呼ぶ．節点 p と節点 q の関係は $p = pred(q)$, $q = succ(p)$ となる．

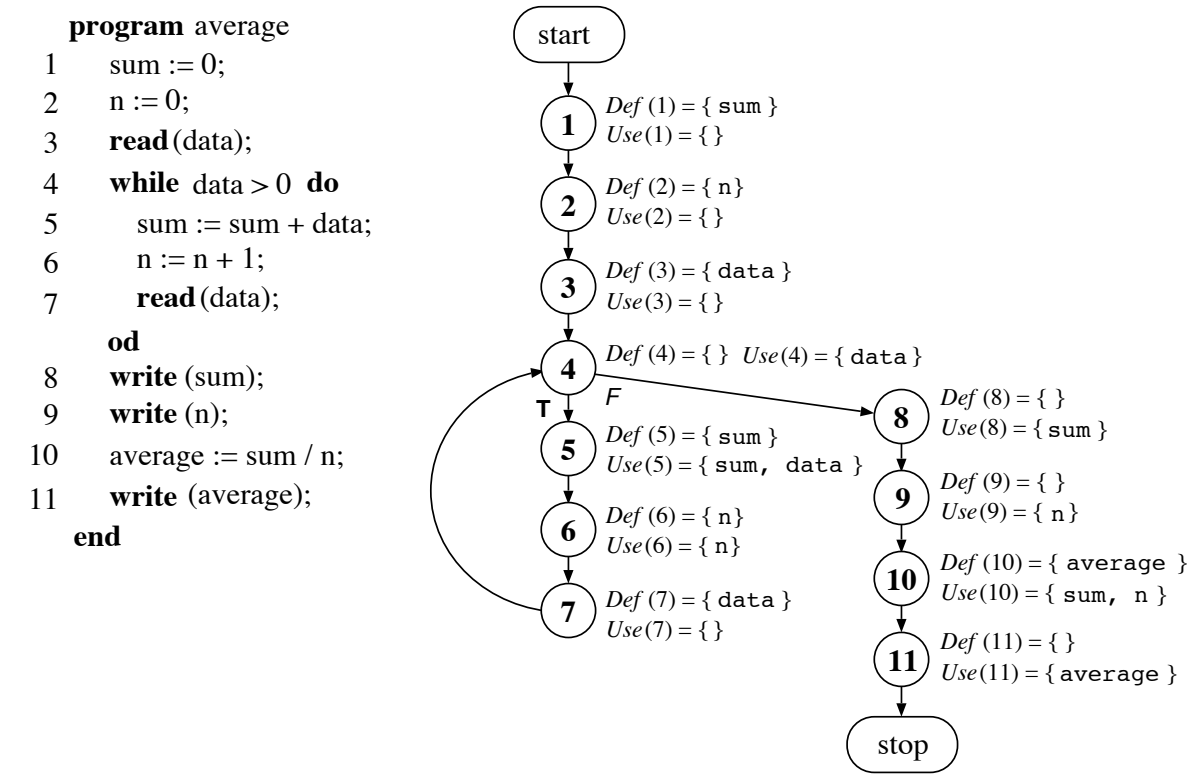


図 2.1: 制御フローグラフ (CFG) の例

(3) $start$ は，プログラム P を実行するときに最初に制御が移される開始文を表す節点である．

CFG は，ソースコードに対して，プログラム制御の流れを静的に解析することで得られる．ここでは， CFG の各節点に対応する文において定義および使用される変数の集合を，ラベルとしてその節点に割り付ける．変数 x を定義するとは，変数 x に値を設定することを指す．また，変数 x を使用するとは，変数 x の値を参照することを指す．節点 p に対して，定義変数の集合を $Def(p)$，使用変数の集合を $Use(p)$ と表す． CFG の例を図 2.1に示す．節点の右側に記述されているラベルは各節点における定義および使用変数集合である．左側のプログラムは Pascal 風の疑似言語で記述した．

CFG を用いることで，プログラムは逐次実行 (連接構造: sequence)，条件分岐 (選択構造: alternation)，繰り返し (反復構造: iteration) の 3 つで表現される．本論文では， CFG 上において，文 p から文 q まで矢印をたどる際に通過する節点の実行系列 $\langle p, \cdots, q \rangle$ を，節点 p から節点 q への到達可能経路 (reachable path) と呼ぶ．

2.2 プログラム依存グラフ

手続き型プログラムにおける文 (命令) 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) [29, 43, 49, 88] を導入する．プログラム P の PDG とは，以下の条件を満たす有向グラフ $G_{PDG} = (N_{PDG}, E_{PDG}, entry)$ である．

(1) N_{PDG} は，プログラムにおける代入式あるいは条件式を割り当てた節点の集合，および，$entry$ 節点である．N_{PDG} から $entry$ 節点を取り除いた集合は，N_{CFG} から $start$ 節点および $stop$ 節点を取り除いた集合に等しい．

(2) E_{PDG} は矢印集合である．節点 p から節点 q への矢印は，節点 p から節点 q に対して，制御依存関係 (control dependence) [29]，あるいは，データ依存関係 (data dependence) [29, 89] があることを指す．

(3) $entry$ は，どの節点にも依存しない入口節点である．(entry 節点を接続先とする依存関係矢印が存在しない．)

制御依存関係およびデータ依存関係の定義を以下に示す．

(a) 節点 p における条件式の実行結果が節点 q の実行を行うかどうかに直接影響を与える場合，節点 p から節点 q に制御依存関係があるという．すなわち，節点 p が条件分岐節点 (if 文に対応) であり，節点 q がその分岐内に直接含まれている場合，あるいは，節点 p がループ節点 (while 文に対応) であり，節点 q がそのループ内に直接含まれている場合を指す．節点 p が節点 q の分岐内あるいはループ内に直接含まれているとは，1) 節点 q が節点 p の分岐内あるいはループ内に含まれる，かつ，2) 節点 q が節点 p の分岐内あるいは繰り返し内に含まれる他の節点の分岐内あるいはループ内に含まれていないことを意味する．本論文では，条件式の実行結果の真偽に基づき制御依存関係矢印を区別し，各制御依存関係矢印に T 属性 (true) あるいは F 属性 (false) を付加する．本論文では，節点 p から節点 q への制御依存関係を $p \rightarrow_c q$ と表す．制御依存関係の属性を明記する場合は，$p \rightarrow_c^T q$, $p \rightarrow_c^F q$ と表す．

(b) 節点 p における変数 v の定義が変数 v を参照する節点 q に到達する場合，節点 p から節点 q にデータ依存関係があるという．すなわち，1) 節点 p において変数 v の値を定義し，節点 q において変数 v の値を参照している，かつ，2) 変数 v の値を再定義しない節点 p から節点 q への CFG における到達可能経路が存在することを指す．本

```
   program average
 1     sum := 0;
 2     n := 0;
 3     read(data);
 4     while data > 0 do
 5       sum := sum + data;
 6       n := n + 1;
 7       read(data);
       od
 8     write (sum);
 9     write (n);
10     average := sum / n;
11     write (average);
   end
```

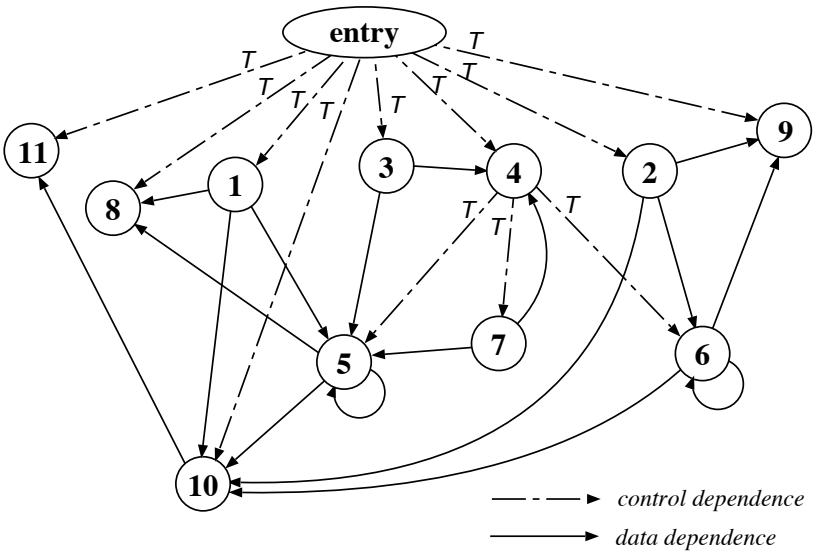


図 2.2: プログラム依存グラフ (PDG) の例

論文では，節点 p から節点 q への変数 v に関するデータ依存関係を $p \rightarrow_d^v q$，任意の変数に関するデータ依存関係を $p \rightarrow_d q$ と表す．

PDG は，ソースコードに対して，制御フローおよびデータフローを静的に解析することで作成可能である． PDG の例を図 2.2に示す．一点鎖線矢印は制御依存関係，実線矢印はデータ依存関係を表す．これら 2 種類の依存関係を区別する必要のないとき，節点 p から節点 q への依存関係を $p \rightarrow q$ と表す．

ここで，手続き型プログラムに対する依存グラフは，用途に応じていくつも存在する [29, 92, 43]．例えば，手続き呼出しを含むプログラムの依存関係を表現するために，システム依存グラフ (system dependence graph: SDG) [38] が提案されている． SDG では，個々の手続きを従来の PDG で表現し，それらの手続きを call 矢印で接続する．また，手続き (関数) 呼出しにおける引数 (パラメータ) の値の受渡しを担う節点 (actual-in, actual-out, formal-in, formal-out 節点) を追加し，それらの節点を parameter-in, parameter-out 矢印で接続する．さらに，呼び出された手続き内部での制御およびデータ依存関係を，呼出し側の手続きの actual-in 節点と actual-out 節点間のデータ依存関係に集約させ，それを summary 矢印で表現する． SDG の例を図 2.3に示す．本論文では，手続き呼び出しを含むプログラムやオブジェクト指向プログラムを扱う場合， SDG およびその変形グラフを用いる．

```
   program Sum                    procedure Add (a, b)          procedure Inc (z)
1     sum := 0;                7     a := a + b;           8     call Add (z, 1);
2     n := 1;                        return                      return
3     while  n < 11 do
4        call Add (sum, n);
5        call Inc (n);
      od
6     write (sum);
   end
```

control dependence
data dependence
call, parameter-in, parameter-out
summary

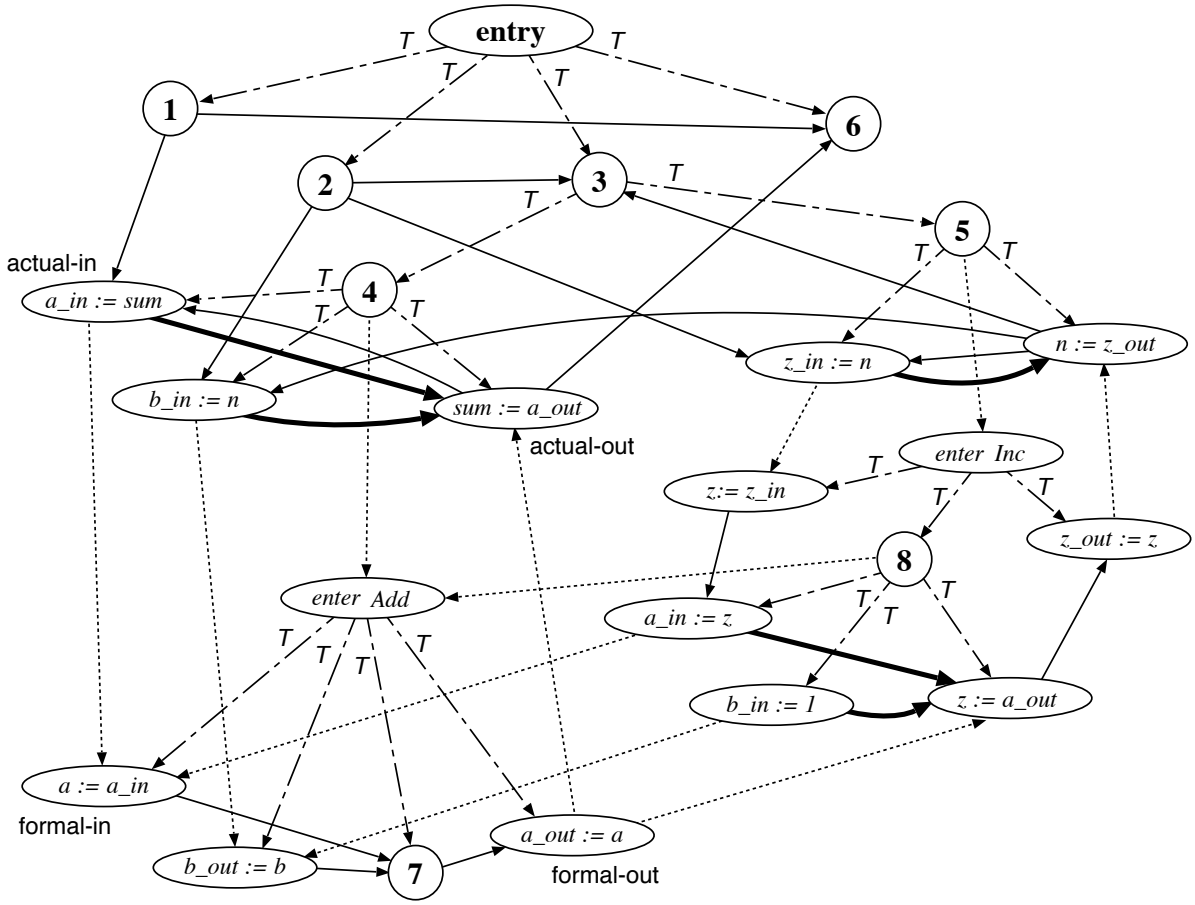


図 2.3: システム依存グラフ SDG の例

2.3 プログラムスライシング

本論文の部品作成手法，部品変化手法，および，クラス生成手法では，プログラムスライシング (program slicing) [111] を用いてソースコードを分割する．プログラムスライシングとは，プログラム内の任意の文において着目する変数に関連を持つ一部の文集合だけを，もとのプログラムから抜き出すことである．抜き出したコードの集まりをプログラムスライス (program slice)，あるいは，単にスライスと呼ぶ．スライシング技法は，プログラムデバッグを支援するために考案されたものであるが [109, 110, 111]，現在では，デバックだけでなく，テスト，保守，プログラム合成，プログラム理解，部品抽出など，さまざまな用途において利用されている [19, 43, 52, 96, 97, 102].

スライスには，すべての入力データに対して解析することで得られる静的スライス (static slice) [111] と，特定の入力データ (実行系列) に対して解析することで得られる動的スライス (dynamic slice) [2, 47] が存在する．本論文では，スライシング技法をソースコードの変形に用いるため，静的スライスのみを扱う (よって，スライスといった場合，静的スライスを意味する)．また，スライスの計算法には，データフロー方程式 (dataflow equation) による方法 [47, 111] と， information-flow 関係に基づく方法 [15]，依存グラフにおける到達可能性判定による方法 [2, 38, 88] がある．本論文では，依存グラフの到達可能性判定による方法を用いる．よって，スライシング実行の前に PDG (あるいは SDG) を作成し，データ依存関係矢印と制御依存関係矢印をたどることにより，スライスを求める．

ここで，節点 n における変数の集合 V に着目する場合 (スライシング基準 $\langle n, V \rangle$)，依存関係をたどる向きに応じて，逆方向スライス (backward slice) と順方向スライス (forward slice) の 2 種類が存在する [38, 80, 107]．逆方向スライスは，節点 n における変数 $v \in V$ の値に影響を与える可能性のある節点 (文) の集合となる．また，順方向スライスは，節点 n における変数 $v \in V$ の値が影響を与える可能性のある節点 (文) の集合となる．図 2.2に示すプログラムにおける逆方向スライスの例を図 2.4に示す．網掛け節点が節点 9 の変数 n に関する逆方向スライスとなる．

手続き呼出しを含むプログラムに対しては，ソースコードから SDG を作成し，次に示す 2 段階の経路探索により逆方向スライスを求める [38, 43]． 1 段目の経路探索では， parameter-out 矢印を除いて，着目する節点 n の変数 V から逆方向に依存関係矢印をたどり，到達する節点の集合 S_{phase1} を求める． 2 段目の経路探索では， call 矢印および parameter-in 矢印を除いて， 1 段目の到達節点 S_{phase1} 内の節点から逆方向に依存関係矢印をたどり，到達する節点の集合 S_{phase2} を求める． S_{phase2} が逆方向スライスとなる．順方向スライスを

```
   program average
2     n := 0;
3     read (data);
4     while  data > 0  do
6        n := n + 1;
7        read (data);
       od
9     write (n);
   end
```

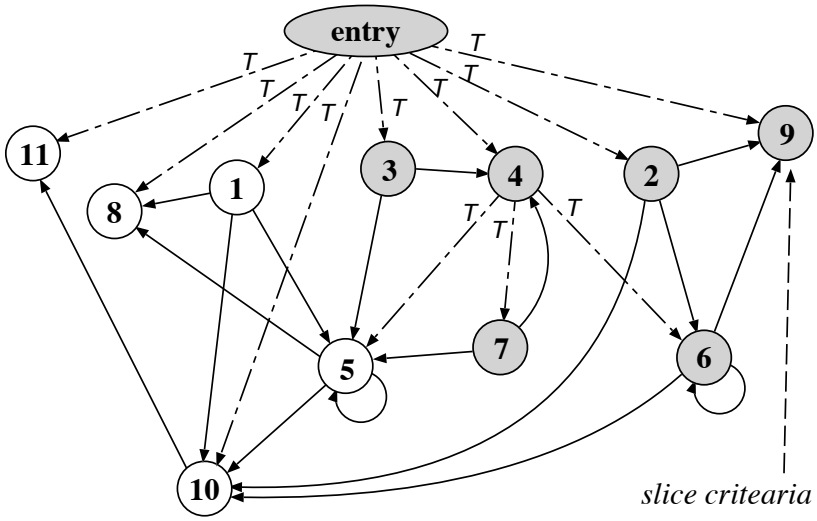


図 2.4: スライシング基準 $\langle 9, n \rangle$ に関する逆方向スライス

求める際には，1 段目で call 矢印および parameter-in 矢印を除き順方向に依存関係矢印をたどり，2 段目で parameter-out 矢印を除き順方向に依存関係矢印をたどればよい．図 2.3に示す手続き呼出しを含むプログラムにおける逆方向スライスの例を図 2.5に示す．網掛け節点が formal-out 節点の変数 z に関する逆方向スライスとなる．

2.4 プログラムの差分合成

プログラミングの過程において，1 つのプログラムから複数の改造版プログラムが作成されることがある．プログラムの差分合成とは，このように独立に変更された 2 つの異なるプログラムを 1 つに統合することを指す．差分合成アルゴリズムとしては，Horwitz, Prins, Reps による HPR アルゴリズム [40]，Yang, Horwitz, Reps による YHR アルゴリズム [114, 115]，西村による PDI アルゴリズム [85]，Binkley, Horwitz, Reps による BHR アルゴリズム [20]，Berzins による方法 [16] がある．本論文では，部品変化手法，および，クラス生成において，この HPR アルゴリズムを拡張したアルゴリズムを用いる．そこで，アルゴリズムを拡張する準備として，本節では HPR アルゴリズムの概略を述べる．

もとのプログラム $Base$ と，$Base$ に対して独立に変更を行うことで得られた 2 つのプログラムを A, B とする．HPR アルゴリズムは，これら 3 つのプログラムに対して，A および B において変更された機能 (動作) を保存し，かつ，$Base$ の無変更部分の機能 (動作) を

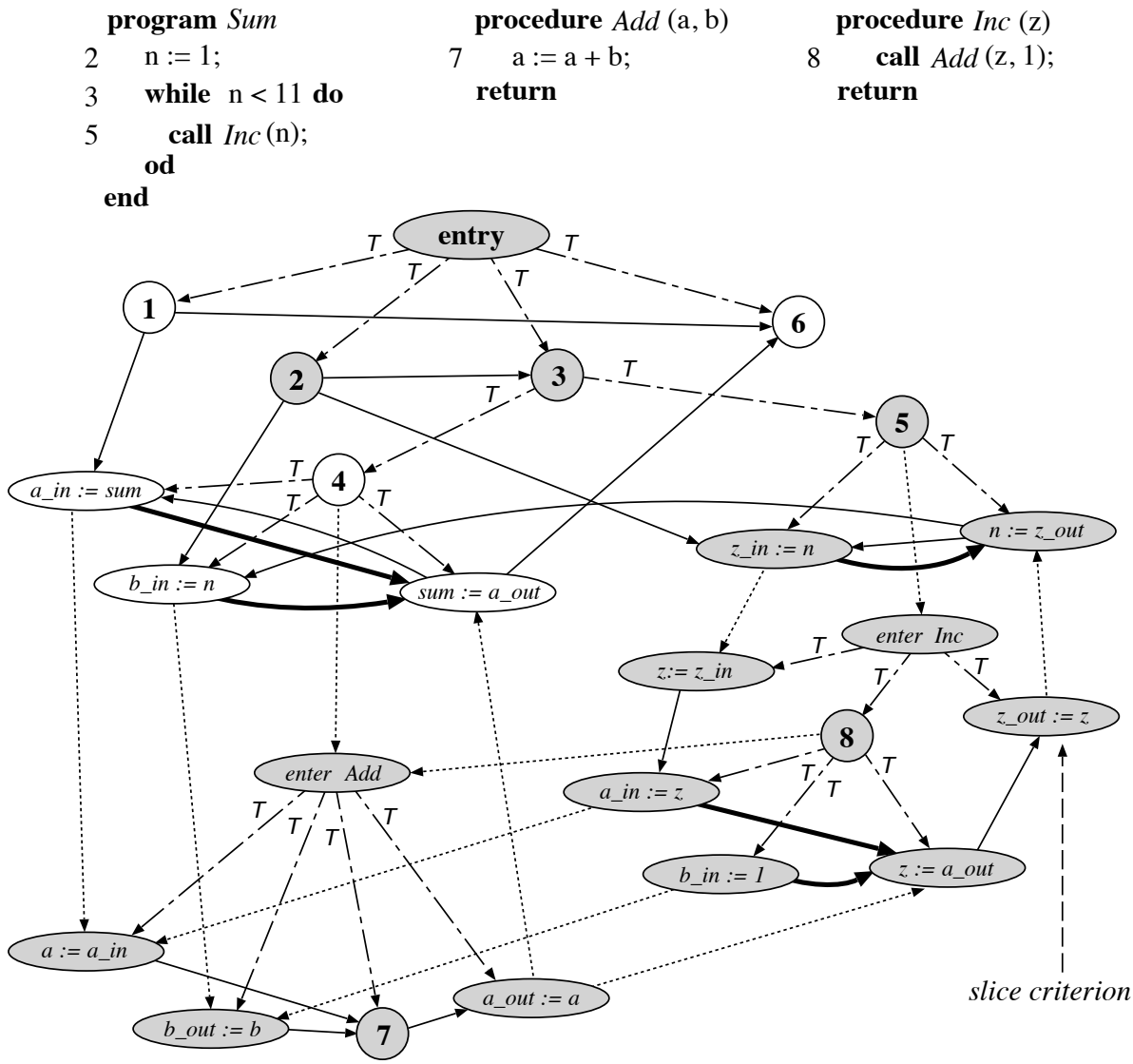


図 2.5: スライシング基準 $\langle z\_out, z \rangle$ に関する逆方向スライス

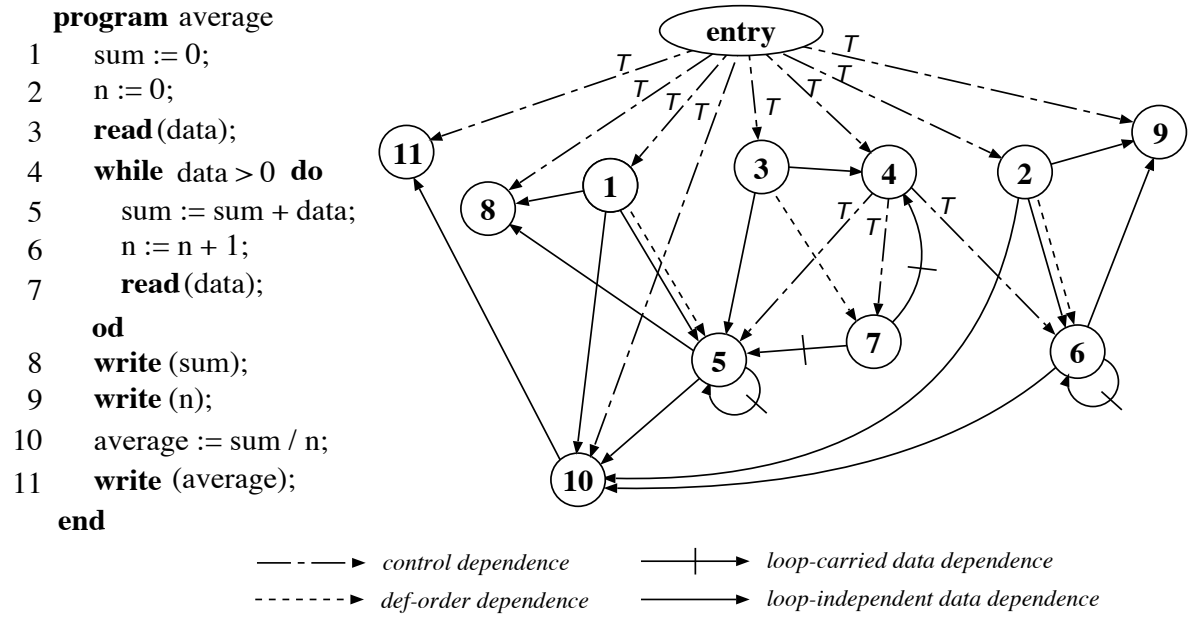


図 2.6: プログラム依存グラフ (PDG) の例

含む，統合プログラム M を生成する．ある変数の値の計算が $Base$, A, B において異なる場合， A と B は意味的に干渉 (お互いに矛盾) するといい，統合プログラム M は生成されない． HPR アルゴリズムの手順は次のようになる．

(1) $Base$, A, B のプログラムに対して，それぞれ PDG G_{Base}, G_A, G_B を作成する．

(2) $Base$ と A の差分 (改造部分) ΔA， $Base$ と B の差分 (改造部分) ΔB， $Base$, A, B における共通部分 (非改造部分) P_{Base} を，それぞれの PDG から取り出す．

(3) 3 つの部分 PDG ΔA, ΔB, P_{Base} を合成し，統合 PDG G_M を得る．

(4) PDG G_M からプログラム (ソースコード) M を再構成する．

PDG の同値性 [41] を保証するため， HPR アルゴリズムでは， 2.2節で述べた PDG のデータ依存関係を，ループ経由データ依存関係 (loop carried) とループ独立データ依存関係 (loop independent) に分類する [41]．さらに， PDG に対して，新たに定義順序依存関係 (def-order dependence) [41] (あるいは， output dependence [89]) を導入する．

ループ経由データ依存関係，ループ独立データ依存関係，定義順序依存関係の定義を以下に示す．

(b1) 節点 p と節点 q が同一のループ内にある，かつ，データ依存関係を担う節点 p から節点 q への到達経路が，このループのループ節点を含むとき，このデータ依存関係をループ経由データ依存関係と呼ぶ．

(b2) 節点 p と節点 q が同一のループ内にない，あるいは，節点 p と節点 q が同一のループ内にあっても，データ依存関係を担う節点 p から節点 q への到達経路が，このループのループ節点を含まないとき，このデータ依存関係をループ独立データ依存関係と呼ぶ．

(c) 節点 p における変数 v の定義が同じ変数 v を定義する節点 q に到達する場合，節点 p から節点 q に定義順序依存関係があるという．すなわち，1) プログラムにおいて，節点 p に対応する文と節点 q に対応する文が，この順序で記述されている，かつ，2) 節点 p と節点 q を含む条件分岐節点の同じ枝に含まれる (同じ属性を持つ制御依存関係矢印により同じ条件分岐節点を接続元とする)，かつ，3) 節点 q において同じ変数 v の値を定義している，かつ，4) 節点 p と節点 q が，同じ節点 r に対してデータ依存元となる ($p \rightarrow_d r$ かつ $q \rightarrow_d r$) ことを指す．

HPR アルゴリズムで用いる PDG の例を図 2.6に示す．一点鎖線矢印は制御依存関係，実線矢印はデータ依存関係 (ループ経由あるいはループ独立)，点鎖線矢印は定義順序依存関係を表す．

さらに，本論文で提案する拡張合成アルゴリズムでは，2 つのプログラムの機能に関する等価性を判定するために，PDG のラベル付き同形写像比較 [42] を用いる．この技法では，比較する 2 つのグラフにおいて，一方のグラフの節点と矢印を他方のグラフに写像することで，同形を判断する．

ラベル付き同形写像比較において，PDG G_1 と G_2 が等価であるとは，次の 4 つの条件を満たすことを指す．

(1) G_1 と G_2 が同数の節点と同数の矢印を持つ．

(2) G_1 の節点から G_2 の節点に，1 対 1 かつ上への写像 φ が存在する．

(3) G_1 の各矢印 $p \rightarrow q$ に対して，矢印 $\varphi(p) \rightarrow \varphi(q)$ が G_2 に存在し，対応する矢印の種類 (データ依存関係，真あるいは偽の制御依存関係) が等しい．

(4) G_1 の各節点 p に対して，p のラベルと G_2 内の節点 $\varphi(p)$ のラベルが等しい．

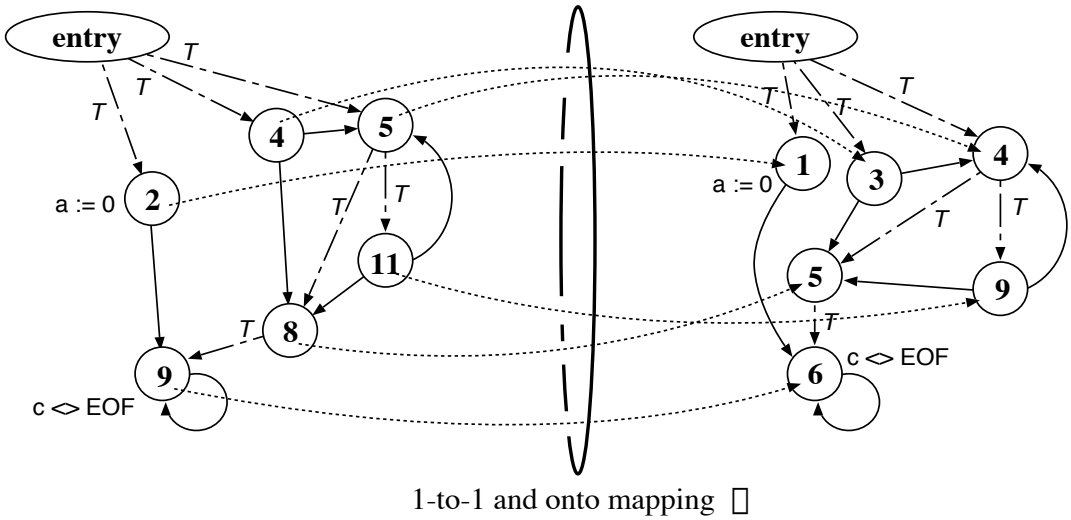


図 2.7: PDG の同形写像比較による一致

ラベル付き同形写像比較において，一致する PDG の例を 2.7に示す．一部の節点のラベルについては省略した．

第 3 章

プログラムスライシングを用いた部品の作成

3.1 はしがき

プログラムのソースコードを部品化して再利用する場合，必要な部品があらかじめ作成されていることが前提となっている．しかしながら，あらゆる開発領域で再利用可能な部品をすべて用意しておくことは不可能である．特に，個人あるいは小規模プロジェクトでは，プロトタイプの作成など短期的で実験的な開発を行なう機会が多く，必要な部品は頻繁に変化する．このため，自分が再利用する部品は自分で作成するということが多く要求される．また，現時点では部品を自動的に作成する手法は存在せず，部品作成工程の大部分を人間が経験的に行なっているのが現状である．このように，部品作成はソフトウェア開発者に大きな負担を与える．

これに対し，部品作成の負担を軽減するためには，既存のソフトウェアから再利用部品を抽出することが重要であると指摘されている [1, 23, 25, 28]．しかしながら，既存のソフトウェアから部品を抽出する場合， 1) 作成する部品の機能を満たすコード断片だけをプログラムから過不足なく抽出することは難しい， 2) 部品の機能を理解する際，抽出された部品のコードだけを参照することで，実行の条件や利用方法を取得および推測することは難しい，という問題が存在する．本研究では，これらの問題に対して，プログラムスライシング [111] を用いたソフトウェア部品の作成手法の検討を進めてきた [57, 58, 61, 62, 65]．本章で述べるソフトウェア部品とは，再利用の対象となるプログラム部品と部品理解に役立つ付加情報を合わせたものである．

問題 (1) に対して，プログラムスライシングにより依存関係を持つコード断片を，既存のプログラムから部品として抽出する手法を提案する．その際，従来のスライシングを単純に適用するのではなく，到達可能経路で定義する区間という概念をスライシングに導入して用

いる．これにより，部品作成者は抽出する部品の機能やサイズを従来より自由に決定することが可能となる．また，問題 (2) に対して，部品を実行する際や利用方法を取得する際に役立つコード断片を，スライシングによって抜き出し，部品作成時に付加情報として部品本体に付加する．この付加コードは，従来の部品作成手法では生成されていない情報である．以上より，提案する部品作成手法は， 1) 既存のプログラムにおいて，部品作成者が任意に指定した区間から，実行可能，かつ，もとのプログラムと同じ実行結果を与える部品を半自動抽出可能であり，従来の部品に比べて不要な記述コードが削除できる， 2) 部品の動作を確認するために必要な実行条件を提供することで多様なテストが可能，さらに，部品の利用例を提示することで部品を再利用する際の指針を与える，という利点を持つ．

これらの利点により，本部品作成手法は，個人あるいは小規模プロジェクトでの部品化再利用を支援する．すなわち，本部品作成手法では利用者が区間を指定する際に，既存プログラムの内部構造に対する知識を持つことを前提とする．個人あるいは小規模プロジェクトのような開発環境では，過去に自分で作成した部品やプロジェクト内で作成された部品を再利用する機会は多い．よって，利用者が部品作成対象プログラムに対する知識を持つことを前提とすることは可能である．さらに，本部品作成手法が対象とする開発環境では要求される部品が頻繁に変化するため，部品作成の負担は大きい．よって，このような開発環境で部品作成を支援することは，ソフトウェアの部品化再利用において有効である．また，本部品作成手法は利用者が指定した区間および変数に応じて自動的に部品を作成可能であることから，対話的な部品作成システムと相性がよい．よって，部品の機能やサイズを逐次調整しながら繰り返し部品を作成することで，頻繁に変化する要求に応じた部品を容易に得ることができる．

本章の構成は次の通りである． 3.2節で従来のプログラムスライシングを節点集合を用いて定義し，従来のスライシングを部品作成に適用した際の問題点を述べる． 3.3節で，これらの問題を解決するために従来のスライスを拡張した区間限定スライスを定義する．次に，3.4節で区間限定スライスを用いたソフトウェア部品の作成手法と例を示す．さらに， 3.5節では本部品作成手法に基づいて部品作成の評価実験を行なった結果を示し， 3.6節で部品抽出に関する利点 (1)，および部品理解に対して付加情報が担う利点 (2) に関して考察する．

3.2 スライシングによる部品作成手法

本節では，部品抽出に用いる拡張プログラムスライスを提案する準備として，静的な依存関係をたどることにより得られる通過節点集合を導入し，従来の逆方向および順方向実行可

能スライスを定義する．さらに，これらのスライスを部品作成に適用した際の問題点を述べる．

3.2.1 逆方向および順方向実行可能スライス

スライシング対象ソースコードのプログラム依存グラフ (PDG) [29] において，節点 n を依存先とする節点の集合を $D_b(n)$，節点 n を依存元とする節点の集合を $D_f(n)$ とすると，これらの集合の定義は以下のようになる．

定義 3.1 逆方向依存節点集合と順方向依存節点集合

$$\begin{aligned} D_b(n) &\stackrel{\text{def}}{=} \{m \mid m \rightarrow_c n\} \cup \{m \mid m \rightarrow_d n\} \\ D_f(n) &\stackrel{\text{def}}{=} \{m \mid n \rightarrow_c m\} \cup \{m \mid n \rightarrow_d m\} \end{aligned}$$

矢印 $\rightarrow_c$ は制御依存関係，矢印 $\rightarrow_d$ はデータ依存関係を表す．

節点 n において着目する変数の集合を V とする．PDG 上で節点 n から逆方向及び順方向に依存関係を k 回たどるとき，通過する節点の集合をそれぞれ $DS_b^k(n), DS_f^k(n)$ とおく．通過節点集合の定義を以下に示す．

定義 3.2 逆方向通過節点集合と順方向通過節点集合

$$DS_b^k(n,V) \stackrel{\text{def}}{=} \bigcup_{m \in DS_b^{k-1} \setminus DS_b^{k-2}} D_b(m) \cup DS_b^{k-1}(n,V)$$

$$\begin{cases} DS_b^0(n,V) = \bigcup_{v \in V} \{m \mid m \mapsto_d^v succ(n)\} \\ \\ DS_b^{-1}(n,V) = \emptyset \end{cases}$$

$$DS_f^k(n,V) \stackrel{\text{def}}{=} \bigcup_{m \in DS_f^{k-1} \setminus DS_f^{k-2}} D_f(m) \cup DS_f^{k-1}(n,V)$$

$$\begin{cases} DS_f^0(n,V) = \bigcup_{v \in V} \{m \mid pred(n) \mapsto_d^v m\} \\ \qquad \cup \{m \mid Def(m) \cap V \neq \emptyset \wedge m = n\} \\ \\ DS_f^{-1}(n,V) = \emptyset \end{cases}$$

$\bigcup_{m\in M} S(m)$ は，集合 M の各要素を $m_1, m_2, \cdots$ としたとき， $S(m_1)\cup S(m_2)\cup\cdots$ を表す．$A \setminus B$ は集合 A から集合 B の要素を取り除いたものである．また，$\mapsto$ は実際に PDG 上に現れる依存関係 ($\rightarrow$ で表す) でなく，ある仮定のもとで成立する一時的な依存関係を表す．$\{m \mid m \mapsto_d^v succ(n)\}$ は，節点 n の直後の節点 $succ(n)$ で変数 v が参照されていると仮定したとき，データ依存元となる節点の集合を指す．また，$\{m \mid pred(n) \mapsto_d^v m\}$ は，節点 n の直前の節点 $pred(n)$ で変数 v が定義されていると仮定したとき，データ依存先となる節点の集合を指す．さらに， $Def(m) \cap V \neq \emptyset$ は，変数集合 V 内の変数の少なくとも一つが節点 m で定義されていることを意味する．

定義 3.2において $k = 1, 2, \cdots$ とすることで，節点 n から逆方向および順方向に依存関係をたどり，通過節点集合を求める．逆方向通過節点集合が収束した ($DS_b^s = DS_b^{s-1}$) とき，収束後の集合を $DS_b(n, V)$ とする．順方向についても同様に，収束したときの順方向通過節点集合を $DS_f(n, V)$ とする．本論文では，通過節点集合を用いて，静的実行可能スライス [107, 79] を以下のように定義する．

定義 3.3 逆方向実行可能スライス

$$S_b(n, V) \stackrel{\text{def}}{=} DS_b(n, V)$$

定義 3.4 順方向実行可能スライス

$$S_f(n, V) \stackrel{\text{def}}{=} \bigcup_{i\in DS_f(n,V)} [\ S_b(i, Use(i)) \cup \{i\}\ ]$$

n は制御フローグラフ (CFG) [3] において着目する節点， V は着目する変数の集合である．スライスに対応するソースコードは，もとの CFG においてスライスに含まれない節点を実行前後でプログラムの状態を変化させないヌル節点に置換することで作成する．ここで，定義 3.3における逆方向スライスの作成手法は，着目する節点に対する扱いを除いて，文献 [88] の手法と同じである．

逆方向実行可能スライス S_b と順方向実行可能スライス S_f の例を図 3.1に示す．図 3.1の CFG において，矢印 DD は $3 \rightarrow_d^{\mathrm{a}} 4$ を，矢印 CD は $4 \rightarrow_c 5$ を表す．図 3.1の CFG から PDG を作成し，定義 3.1 – 3.4を適用することで， $S_b(5, \{\mathrm{sum}\})$, $S_f(5, \{\mathrm{sum}\})$ が求められる．

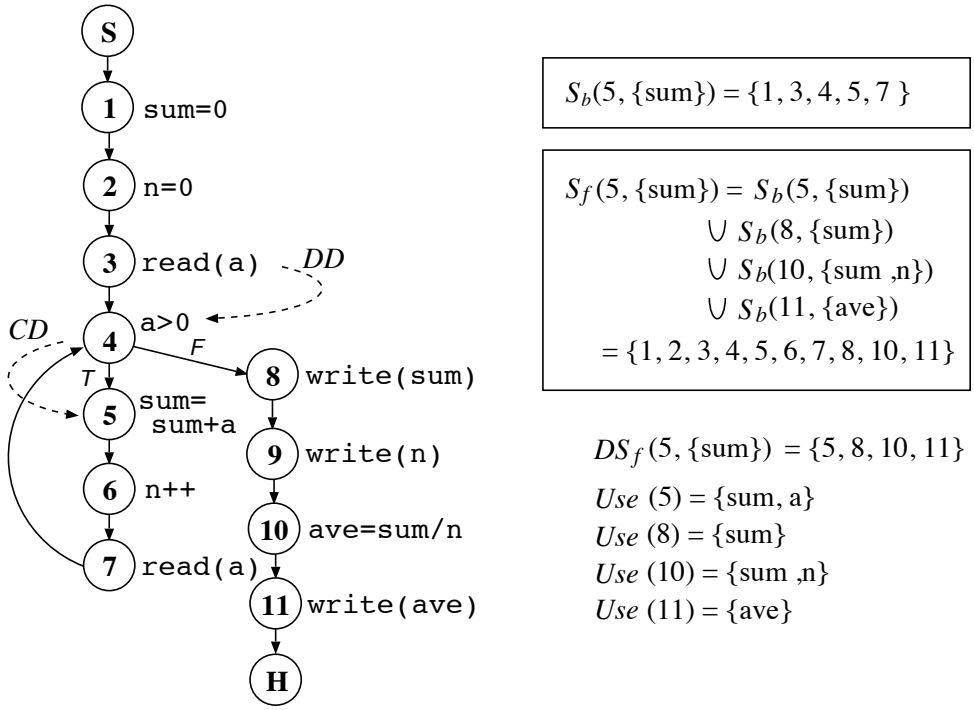


図 3.1: 逆方向スライスと順方向スライスの例

3.2.2 部品作成への適用における問題点

従来のソースコード再利用では，既存のプログラムの関数をそのまま，あるいは，単純に結合するだけで部品を作成する場合が多い．この場合，作成される部品の機能やサイズは必要以上に大きくなり，不要なコードが部品に含まれる．これに対して，定義 3.3, 3.4のスライスは依存関係を持つコードだけを集めたものなので，実行するのに不要なコードは抽出した部品に含まれない．さらに，スライスはスライシング基準が与えられた後は自動で作成できるので，部品作成の負担は軽減される．

プログラムスライシングを部品作成の手段に用いる研究は従来から行われている．文献 [32] では，変数のみをスライシング基準として指定し，スライス (decomposition slice) を作成する．また，文献 [27] では，指定した変数に直接影響を及ぼす節点と，これらの節点を枝に含む分岐節点をスライス (direct slice) として作成する．文献 [14] では，モジュール間の区切りを維持したまま，スライス (interface slice) を作成する．さらに，文献 [24] では，プログラム制御の流れを決定するための式をスライシング基準に追加し，スライス (conditioned slice) を作成する．

しかしながら，文献 [32, 27] のスライシングは，依存関係をたどる際に対象とするプログラムの範囲に関する制約を持たない．また，文献 [14] では関数の先頭，文献 [24] では分岐文だけがスライシングの範囲を決定する要素となる．このように，これらのスライシングでは，その対象範囲を部品作成者が自由に指定することはできない．よって，従来のスライシングを適用することで，依存関係だけに従い抽出した部品の機能やサイズは部品作成対象のプログラムの構造に大きく依存し，部品作成者が意図しない不要なコードが含まれることがある．これは，既存のプログラムが部品作成者の要求する機能を含んでいても，その機能だけを持つ部品を抽出することができないことを意味する．

また，作成するプログラムが頻繁に変化する領域では，部品をそのままの形であるいは他の部品と単純に合成するだけで目的のプログラムが得られる機会は少なく，部品変更が数多く生じる．部品を変更するためには，部品がどのような機能を持っているのかを理解することが不可欠である．しかし，従来の手法は部品を作成することだけにスライシングを適用しており，部品を再利用する際に役立つ支援情報の生成は行っていない．すなわち，3.1節で述べた部品抽出に関する問題 (2) は残されたままである．

3.3 区間限定スライス

本節では，3.2節で述べた問題を解決するために，区間という概念を導入して，従来の逆方向および順方向実行可能スライスを拡張した，逆方向および順方向区間限定スライスを定義する．

3.3.1 プログラムスライシングの拡張

抽出された部品を再利用して目的のプログラムを作成する場合，その部品が適当な入力データのもとで実行可能であることが不可欠である．また，既存のプログラムから抽出したにもかかわらず，もとのプログラムと関連を持たない部品は，その機能を特定することが難しく，再利用するのに適さない．このため，抽出した部品はもとのプログラムを実行させたときと同じ結果を出力する必要がある．本章では，これら部品の持つべき 2 つの性質をそれぞれ実行可能性および抽出等価性と呼ぶ．

いま，もとのプログラム P と抽出された部品 C 内に含まれる変数の集合を V_P, V_C とする．このとき，$V_P \supseteq V_C$ となり，$V'_P \equiv V_P \cap V_C (= V_C)$ とおく．実行可能性とは，V'_P 内の各変数の値が P の実行後に決定されるとき，V_C 内の各変数の値が C の実行後に必ず決定されることを指す．抽出等価性とは，実行前の P と C において V'_P と V_C 内の各変数の

値がお互いに等しいとき，実行後の P と C において V'_P と V_C 内の各変数の値がお互いにすべて等しいことを意味する．

本部品作成手法では，既存のプログラムにおいて部品作成者が任意に指定した範囲から部品を抽出可能とするため，従来の逆方向および順方向スライスに区間という概念を導入する．このとき，単純に指定した範囲のコードをプログラムから切り出し，スライシングを適用して部品を抽出すると，部品の実行可能性や抽出等価性が失われる．例えば，繰り返し構造の一部を含むコードを単純に切り出すと，繰り返し条件に関わる文が不用意に削除され，実行できない部品が作成される場合がある．また，実行できても抽出等価性は保証されない．

このため，本部品作成手法では部品作成者が CFG において任意の 2 節点を指定し，これらの節点間の到達可能経路を考える．このとき，繰り返し文の内部からでも一部の連続した節点だけを抜き出すことを可能とするため，到達可能経路に制約を追加する．これを制約付き到達可能経路と呼び，この経路によりスライシングの対象区間を表す．さらに，制約付き到達可能経路を用いて，区間の指定が可能となるように従来の逆方向および順方向スライスを拡張し，逆方向および順方向区間限定スライスを定義する．これにより，スライシングによって抽出される部品は実行可能性と抽出等価性を満たすことが保証される．

3.3.2 制約付き到達可能経路

本部品作成手法では，部品作成者が指定する区間を上限節点 n_u と下限節点 n_l で表現する． CFG 上を節点 n_u から節点 n_l に矢印をたどる際に通過する節点を制限することで，従来の到達可能経路に制約を与えた制約付き到達可能経路 (constrained reachable path: CRP) を導入する．制約付き到達可能経路は，節点集合 $CRP(n_u, n_l)$ と，もとの CFG 上において $CRP(n_u, n_l)$ に含まれる 2 つの節点を接続する矢印の集合で構成される部分グラフである．このように，もとの CFG があれば $CRP(n_u, n_l)$ から制約付き到達可能経路を求めることができるので，本論文では制約付き到達可能経路の代わりに $CRP(n_u, n_l)$ を用いる．この節点集合を制約付き到達可能経路集合と呼び，以下のように定義する．

定義 3.5 制約付き到達可能経路集合

$$CRP(n_u, n_l) \stackrel{\text{def}}{=} CRP_b(n_u, n_l) \cup CRP_f(n_u, n_l)$$

$CRP_b(n_u, n_l)$ ：CFG において，下限節点 n_l から上限節点 n_u に下限節点 n_l を中間節点として通らず，逆方向に経路をたどるとき通過する節点の集合．

```
procedure getCRP(G, n_u, n_l)
declare G: もとのプログラムの CFG;  n_u, n_l: 上限節点および下限節点;
        CRP: 制約付き到達可能経路 CRP(n_u, n_l);
        CRP_b, CRP_f, W: 節点集合;
  function BackwardWalk(n, m)
  input n, m: 探索開始節点および終了節点;
  declare n_p: 節点;
  begin
    W := W ∪ {n};
    if n = m then return; fi
    for each n_p ∈ pred(n) ∧ n_p ∉ W do
      BackwardWalk(n_p, m);
    od
  end
  function ForwardWalk(n, m)
  input n, m: 探索開始節点および終了節点;
  declare n_s: 節点;
  begin
    W := W ∪ {n};
    if n = m then return; fi
    for each n_s ∈ succ(n) ∧ n_s ∉ W do
      ForwardWalk(n_s, m);
    od
  end
begin
  W := ∅; ForwardWalk(n_u, n_l); CRP_b := W;
  W := ∅; BackwardWalk(n_l, S); CRP_b := CRP_b ∩ W;
  W := ∅; BackwardWalk(n_l, n_u); CRP_f := W;
  W := ∅; ForwardWalk(n_u, H); CRP_f := CRP_f ∩ W;
  CRP := CRP_b ∪ CRP_f;
  return (CRP);
end
```

図 3.2: 制約付き到達可能経路集合を求めるアルゴリズム

$CRP_f(n_u, n_l)$: CFG において，上限節点 n_u から下限節点 n_l に上限節点 n_u を中間節点として通らず，順方向に経路をたどるとき通過する節点の集合.

定義 3.5に基づき，制約付き到達可能経路を求めるアルゴリズム getCRP を図 3.2に示す．図 3.2において， S は CFG の開始節点， H は終了節点を指す．

節点 n_u から節点 n_l に制約をつけずに到達可能な経路上にある節点の集合を $RP(n_u, n_l)$ と表す．定義 3.5では，制約として中間節点の通過を禁止しているため，$CRP(n_u, n_l) \subseteq RP(n_u, n_l)$ となる．よって，節点 n_u や節点 n_l から到達不可能な節点が制約付き到達可能経路集合に含まれることはない．また，制約は経路をたどる際に利用されるだけで，上限節点 n_u および下限節点 n_l は CFG の任意の位置に指定することが可能である．さらに，制約付き到達可能経路集合は，経路をたどるとき通過する節点を集めたものなので，必ず一意に決定され，CFG 上で連続する節点集合となる．

逐次実行 (sequence)，条件分岐 (alternation)，繰り返し (iteration)，関数呼出し (function call) に対する制約付き到達可能経路集合 CRP の例を図 3.3に示す．逐次実行 (図 3.3a) では，2 節点間に存在する節点が単純に CRP となる．条件分岐 (図 3.3b) でも，2 節点間に存在する節点が CRP となり，どちらかの節点が分岐の枝に存在するとき，分岐の片側に存在する節点だけが CRP に含まれる．繰り返し (図 3.3c) では，2 節点がループの内側に存在するとき，2 節点間に存在する節点が CRP となり，どちらかの節点がループの外部に存在するとき，CRP はループ内のすべての節点を含む．また，関数呼出し (図 3.3d) に対しては，経路を探索する際に関数を動的に展開して CRP を求める．再帰を含む関数呼出しについては関数の展開ができないため，本部品作成手法では扱えない．

3.3.3 区間限定スライス

定義 3.5の制約付き到達可能経路を用いることにより，定義 3.3の逆方向実行可能スライスおよび定義 3.4の順方向実行可能スライスを拡張したものを区間限定スライス (bounded slice) と呼ぶ．ここで，もとのプログラムにおいて，指定した区間だけに関わる依存関係を区間限定依存とする．指定した区間に含まれる節点の集合を B とするとき，逆方向および順方向区間限定依存節点集合 $BD_b(B, n)$, $BD_f(B, n)$ を以下のように定義する．

定義 3.6 区間限定依存節点集合

$$BD_b(B, n) \stackrel{\text{def}}{=} D_b(n) \cap B$$
$$BD_f(B, n) \stackrel{\text{def}}{=} D_f(n) \cap B$$

区間限定依存節点集合を用いると，節点集合 B で表現された区間における節点 n の変数集合 V に関する通過節点集合 $BDS_b^k(B, n, V)$ と $BDS_f^k(B, n, V)$ の定義は以下のようになる．

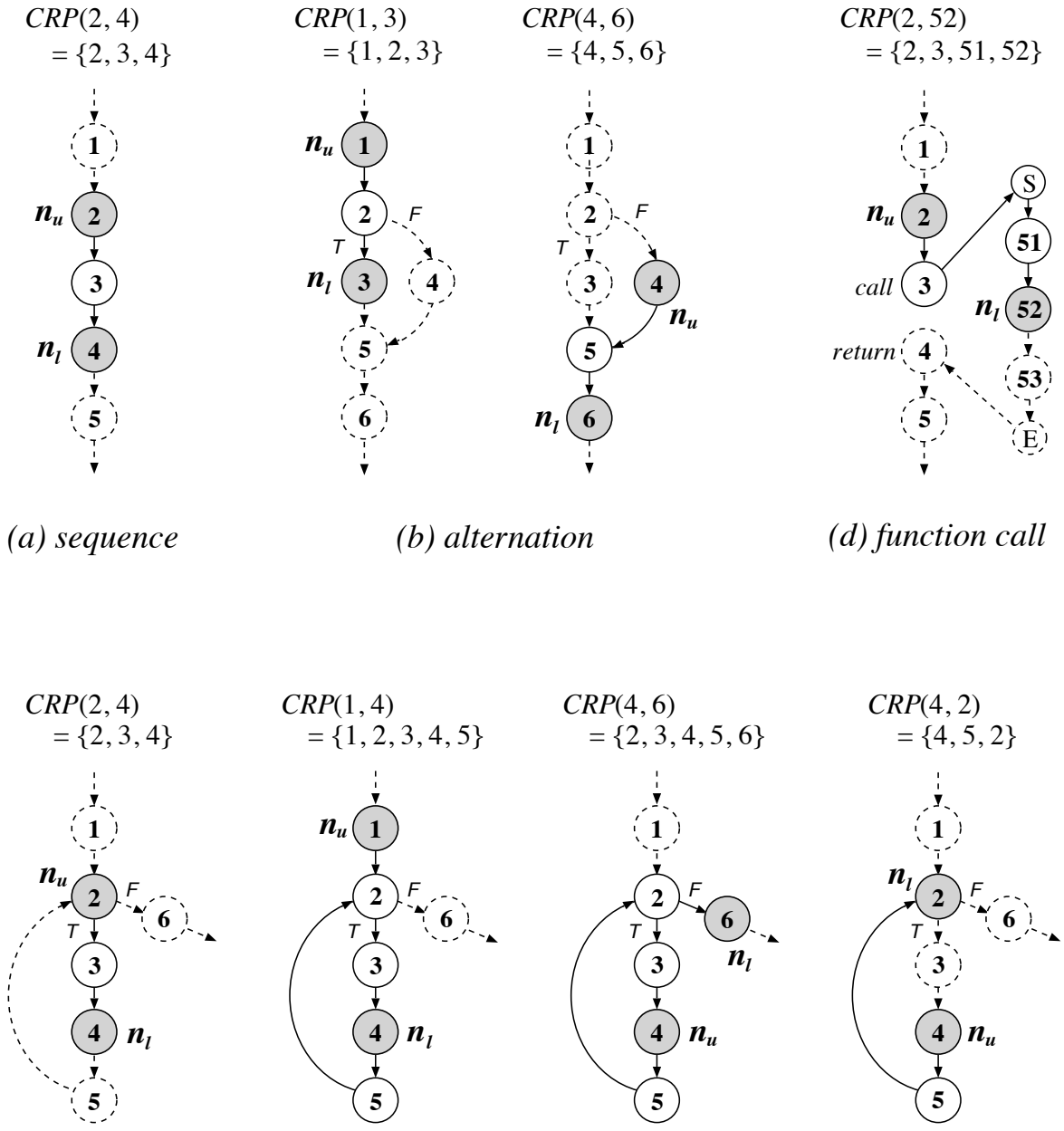


図 3.3: 制約付き到達可能経路 (CRP) の例

定義 3.7 区間限定通過節点集合

$$BDS_b^k(B,n,V) \stackrel{\text{def}}{=} \bigcup_{m \in BDS_b^{k-1} \setminus BDS_b^{k-2}} BD_b(B,m) \cup BDS_b^{k-1}(B,n,V)$$

$$\begin{cases} BDS_b^0(B,n,V) = DS_b^0(n,V) \cap B \\ \\ BDS_b^{-1}(B,n,V) = \emptyset \end{cases}$$

$$BDS_f^k(B,n,V) \stackrel{\text{def}}{=} \bigcup_{m \in BDS_f^{k-1} \setminus BDS_f^{k-2}} BD_f(B,m) \cup BDS_f^{k-1}(B,n,V)$$

$$\begin{cases} BDS_f^0(B,n,V) = DS_f^0(n,V) \cap B \\ \\ BDS_f^{-1}(B,n,V) = \emptyset \end{cases}$$

定義 3.2の通過節点集合と同様に，定義 3.7において収束後の逆方向および順方向区間限定通過節点集合を $BDS_b(B,n,V)$, $BDS_f(B,n,V)$ とする．これらの区間限定通過節点集合を用いて，逆方向区間限定スライス (backward bounded slice) および順方向区間限定スライス (forward bounded slice) を定義する．区間 $CRP(n_u,n_l)$ の変数集合 V に関する区間限定スライスの定義を以下に示す．

定義 3.8 逆方向区間限定スライス

$$\hat{S}_b(n_u,n_l,V) \stackrel{\text{def}}{=} BDS_b(CRP(n_u,n_l),n_l,V)$$

定義 3.9 順方向区間限定スライス

$$\hat{S}_f(n_u,n_l,V) \stackrel{\text{def}}{=} \bigcup_{i \in BDS_f(CRP(n_u,n_l),n_u,V)} [\ S_b(i,Use(i)) \cup \{i\}\ ]$$

n_u, n_l は CFG で指定する上限節点および下限節点，V は指定する変数集合である．また，$\hat{S}_b(n_u,n_l,\emptyset) = \hat{S}_f(n_u,n_l,\emptyset) = \emptyset$ である．

逆方向区間限定スライス $\hat{S}_b(n_u,n_l,V)$ は，もとのプログラムの節点 n_l において変数集合 V のすべての変数に影響を与えるコードを，区間 $CRP(n_u,n_l)$ に関して抜き出したものである．また，順方向区間限定スライス $\hat{S}_f(n_u,n_l,V)$ は，節点 n_u において変数集合 V のす

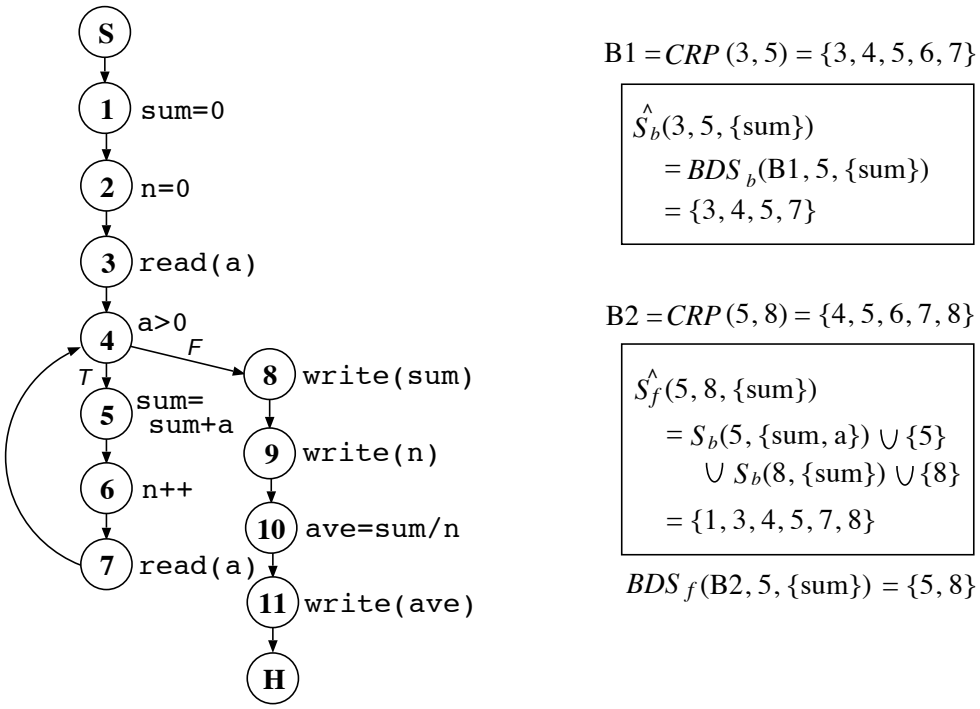


図 3.4: 区間限定スライスの例

べての変数が影響を与える節点を区間 $CRP(n_u, n_l)$ に関して集め，それらの節点に対する逆方向スライスを和集合で結合して作成する．ここで，順方向区間限定スライスは逆方向実行可能スライスの和集合により作成するので，スライス単体で実行可能である．

このように，区間限定スライス $\hat{S}_b, \hat{S}_f$ はデータ依存関係と制御依存関係をたどる際に区間による制約を与えて作成する． $\hat{S}_b$ は S_b に対して，逆方向に依存関係をたどる際の依存元節点に制約を与えることで作成し， $\hat{S}_b(n_u, n_l, V) \subseteq S_b(n_l, V)$ となる．また， $\hat{S}_f$ は S_f に対して，順方向に依存をたどる際の依存先節点に制約を与えることで作成し， $\hat{S}_f(n_u, n_l, V) \subseteq S_f(n_u, V)$ となる．

図 3.4に逆方向区間限定スライス $\hat{S}_b$ と順方向区間限定スライス $\hat{S}_f$ の例を示す．図 3.4に示す CFG において，上限節点を 3，下限節点を 5 とすると，区間 $CRP(3, 5)$ 内の節点を依存先とする依存関係のうち， $1 \rightarrow_d^{\mathrm{sum}} 5$, $2 \rightarrow_d^{\mathrm{n}} 6$ が除外される．よって，逆方向区間限定通過節点集合は，

$$BDS_b^0(\mathrm{B1}, 5, \{\mathrm{sum}\}) = \{\ 5\ \}$$
$$BDS_b^2(\mathrm{B1}, 5, \{\mathrm{sum}\}) = BDS_b^1 = \{\ 3, 4, 5, 7\ \}$$

となり，逆方向区間限定スライス $\hat{S}_b(3,\ 5,\ \{\text{sum}\})$ が求められる． $\hat{S}_b(3,\ 5,\ \{\text{sum}\})$ は，$S_b(5, \{\text{sum}\})$ に対して，節点 1 が取り除かれている．

また，上限節点を 5，下限節点を 8 とすると，区間 $CRP(5,\ 8)$ 内の節点を依存元とする依存関係のうち， $5 \rightarrow_d^{\text{sum}} 10$, $6 \rightarrow_d^{\text{n}} 9$, $6 \rightarrow_d^{\text{n}} 10$ が除外される．よって，順方向区間限定通過節点集合と逆方向実行可能スライスは，

$$
\begin{aligned}
&BDS_f^0(\text{B2}, 5, \{\text{sum}\}) = \{\ 5\ \} \\
&BDS_f^2(\text{B2}, 5, \{\text{sum}\}) = BDS_f^1 = \{\ 5, 8\ \} \\
&S_b(5, \{\text{sum}\}) = \{\ 1, 3, 4, 5, 7\ \} \\
&S_b(8, \{\text{sum}\}) = \{\ 1, 3, 4, 5, 7\ \}
\end{aligned}
$$

となり，順方向区間限定スライス $\hat{S}_b(5,\ 8,\ \{\text{sum}\})$ が求められる． $\hat{S}_f(5,\ 8,\ \{\text{sum}\})$ は，$S_f(5, \{\text{sum}\})$ に対して，節点 2, 6, 10, 11 が取り除かれている．

3.4 ソフトウェア部品の作成手法

3.4.1 対象プログラム

本部品作成手法では，部品作成対象プログラムの言語として手続き型言語 C のサブセットを用いる．プログラムを構成する文として，式 (代入文，関数呼出し文)，複文，選択文 (if, switch)，繰り返し文 (while, do, for)，ジャンプ文 (return, break, continue, goto) を持つ． 3.3節で述べたように，再帰呼出しに対しては制約付き到達可能経路を求めことができないので，本部品作成手法の対象から除く．ジャンプ文を含むプログラムを部品作成対象とする際には， ACFG (augmented CFG) [9] を利用し， APDG (augmented PDG) [9] を作成する．スライスは， APDG 上で依存関係をたどることで求める．

ここで，本部品作成手法の適用はデータ依存関係の解析後であるので，対象プログラム言語の変数の型は本部品作成手法に制限されない．しかし， C 言語のすべての変数の型をデータ依存関係解析可能とみなすのは難しいため，本部品作成手法ではスカラ型，配列型と一次ポインタのみを扱う．さらに， C 言語の標準ライブラリで提供される関数 (printf, getc など) については依存関係解析済みとし，引数間の依存関係はあらかじめ解析情報として持つこととする．

上記の言語は現実の C 言語に比べて一部簡単化が行われているが，再帰構造が扱えない点を除き基本的なデータ構造および制御構造をすべて含む．

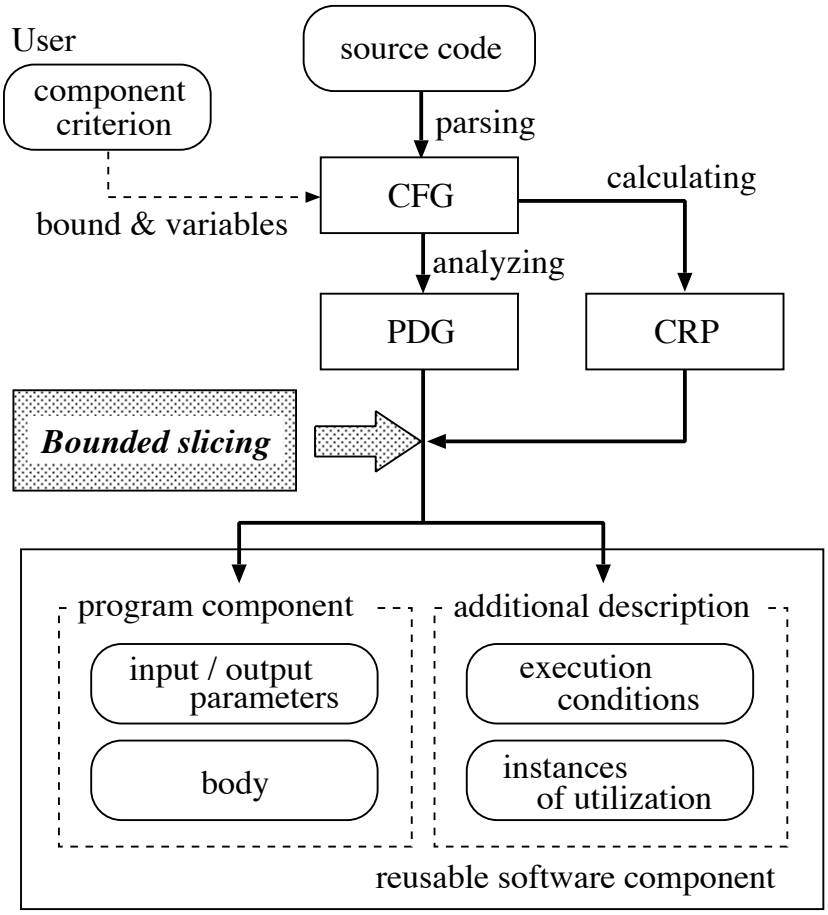


図 3.5: 本部品作成手法の概要

3.4.2 ソフトウェア部品

本部品作成手法の概要を図 3.5に示す．ソフトウェア部品は，ソフトウェア開発者が部品作成基準 C_C (component criterion) を指定し，スライシングを適用することで作成する．部品作成基準とは，作成する部品の機能を決定する際に着目する区間 $CRP(N_u, N_l)$ と変数 V を 3 つ組で表現したものである．部品本体は再利用されるソースコードを指し，入出力パラメータと合わせてプログラム部品となる．

ここで，部品本体に対する入出力パラメータを決定するために，入力データ依存関係 (incoming data dependence) と出力データ依存関係 (outgoing data dependence) を考える．入力データ依存関係は部品本体外部の節点から内部の節点へのデータ依存関係を表す．このとき，もとのプログラムにおける関数の引数は関数の先頭で定義されているとみなし，各引

数について一つずつ引数節点 (parameter node) [38] を作成し，関数内部の節点とのデータ依存関係を計算しておく．また，出力データ依存関係は部品本体内部の節点から外部の節点へのデータ依存関係を表す．部品本体を表す節点集合を C とするとき，入力データ依存節点集合 $DD_{in}(C)$ および出力データ依存節点集合 $DD_{out}(C)$ の定義を以下に示す．

定義 3.10 入出力データ依存節点集合

$$DD_{in}(C) \stackrel{\text{def}}{=} \{m \mid m \rightarrow_d n \wedge m \notin C \wedge n \in C\}$$
$$DD_{out}(C) \stackrel{\text{def}}{=} \{m \mid m \rightarrow_d n \wedge m \in C \wedge n \notin C\}$$

以上より，部品作成基準 $C_C = \langle N_u, N_l, V \rangle$ におけるソフトウェア部品を以下のように定義する．

定義 3.11 ソフトウェア部品 (SLICE)

部品本体 (body) [節点集合]:

$$C_{body} \stackrel{\text{def}}{=} \hat{S}_b(N_u, N_l, V)$$

入力パラメータ (input parameters) [変数集合]:

$$C_{in} \stackrel{\text{def}}{=} \bigcup_{m \in DD_{in}(C_{body})} Def(m)$$

出力パラメータ (output parameters) [変数集合]:

$$C_{out} \stackrel{\text{def}}{=} (V; V_d; V_s; V_e)$$

$$\begin{cases} V_d = \bigcup\limits_{m \in C_{body}} Def(m) \\ V_s = \{w \mid w \in V_d \wedge \hat{S}_b(N_u, N_l, w) \subseteq C_{body}\} \\ V_e = \bigcup\limits_{m \in DD_{out}(C_{body})} Def(m) \end{cases}$$

実行条件例 (execution conditions) [節点集合]:

$$C_{cond} \stackrel{\text{def}}{=} S_b(N_l, V) \setminus C_{body}$$

利用例 (instances of utilization) [節点集合]:

$$C_{inst} \stackrel{\text{def}}{=} \hat{S}_f(n_r, n_{rk}, V) \setminus C_{body}$$

$$\begin{cases} n_r \in \bigcup_{v \in V} \{n \mid m \rightarrow_{d(v)} n \wedge m \in C_{body} \wedge n \notin C_{body}\} \\ \\ n_{rk} \in DS_f^R(n_r, V) \end{cases}$$

出力パラメータにおいて， $A;B;C;D$ は A, B, C, D のうちどれか一つを選択することを意味する． V は部品作成時に指定した変数集合， V_d は部品本体内部で値が定義される変数集合， V_s は部品本体内部で値が一意に定まる変数集合， V_e は部品本体外部で参照される変数集合である．利用例の節点 n_r は，部品本体から外部への変数集合 V に関するデータ依存先となる節点の一つである． R は指定した変数集合 V が利用されている節点を見つけるために，節点 n_r から依存関係を順方向にたどる回数を表し， $DS_f^R(n_r, V)$ は節点 n_r から依存関係を順方向に R 回たどるとき通過する節点の集合を指す．

本部品作成手法では，実行条件例と利用例を合わせて付加情報 (additional information) と呼ぶ．これらは，部品本体と同時にスライシングによって生成される．定義 3.11より，実行条件例は部品を作成したときに着目した変数に関する逆方向実行可能スライスから部品本体を取り除くことで生成する．よって，部品本体と組み合わせて得られるプログラムは，必ず実行可能かつ抽出等価である．また，利用例は部品作成時に着目した変数がもとのプログラムにおいて影響を与えていた文と，その文を実行するために必要なコードを集めることで生成する．よって，作成した部品本体が実際にもとのプログラムでどのように利用されていたのかを示す一例となる．ここで，順方向区間限定スライスの性質より，利用例と部品本体を組み合わせて得られるプログラムは必ず実行可能性と抽出等価性を満たす．

次に，部品作成対象プログラムがジャンプ文を含む場合を考える．このとき，制約付き到達可能経路を求めるために ACFG を利用するか CFG を利用するかで作成されるソフトウェア部品は異なる． ACFG を用いた場合，区間内のジャンプ文から下限節点への経路が必ず存在するので，ジャンプ文節点およびその依存元節点は部品本体に含まれる． CFG を用いた場合，指定した区間外へ制御を移すジャンプ文は区間から無条件に排除されるので，APDG で依存関係をたどる際にジャンプ文節点で制御依存関係が途切れる．よって，ジャンプ文節点とその依存元節点は部品本体から削除される．

ここで，部品作成を考えると，作成された部品本体が不必要に例外処理などを含むと部品の汎用性が失われ，再利用される機会が減少する．さらに，部品理解は難しくなる．また，

CFG を用いることで削除された節点は，ソフトウェア部品内部に実行条件例 C_{cond} として必ず含まれるため，部品を実行することは可能である．そこで，本部品作成手法では制約付き到達可能経路を求める際には CFG を用い，部品に主要コードだけを含むようにする．

3.4.3 ソフトウェア部品の例

図 3.6に部品作成対象のプログラムとその CFG を示す． CFG において実線節点および実線矢印は $CRP(13,\ 34)$ を表す．プログラムにおいてコードの左側の数字は対応する CFG の節点番号を指す．本部品作成手法では，代入文が関数を含む場合， CFG 上で呼出し文節点と代入文節点に分割する (節点 2, 3 など)．また，区間として制御が合流する点を指定可能とするため，合流節点を導入する．プログラム中，( ) に入っている節点番号は合流節点を表す．さらに，標準ライブラリ関数 fopen, fclose, getch, printf に関しては，関数内部の節点を区間として指定できないことにし，関数の展開は行わない．

図 3.6のプログラムに対して，部品作成基準 $C_C = \langle\ 13, 34, \{\ \text{len}\ \}\ \rangle$ を指定し，「単語の長さを数える」機能を持つソフトウェア部品を作成すると以下のようになる．

$C_{body} = \{\ 23, 24, 25, 26, 27, 29, 31, 32\ \}$
$C_{in} = \{\text{fp}\}$
$C_{out} = (\{\text{len}\}; \{\text{ch, len}\}; \{\text{ch, len}\}; \{\text{ch, len}\})$
$C_{cond} = \{\ 1, 2, 3, 4, 9, 12\ \}$
$C_{inst1} = \{\ 10, 13\ \} \cup C_{cond}$
$C_{inst2} = \{\ 6, 10, 13, 14, 15, 18\ \} \cup C_{cond}$
$C_{inst33} = \{\ 28, 33\ \} \cup C_{cond}$

$R = 3$ とすると利用例は全部で 33 個作成され，そのうち 3 個を示した．これらの節点集合をソースコードに戻したものを図 3.7に示す．また，異なる部品作成基準で作成したときの部品本体を図 3.8a, 3.8b, 3.9c, 3.9d に示す．図中で変数宣言文の右側のコメント文 (input, output) は入出力パラメータを表す．

次に，指定した区間にジャンプ文が含まれる場合の例を示す．図 3.6に示すプログラムにおいて， ACFG および CFG を利用したときの制約付き到達可能経路は，それぞれ $CRP(1,6) = \{\ 1, 2, 3, 4, 5, 6\ \}$ (ACFG), $CRP(1,6) = \{\ 1, 2, 3, 5, 6\ \}$ (CFG) となる．依存関係をたどる際には APDG を利用するので， APDG を作成すると， $1 \rightarrow_c 2$, $1 \rightarrow_c 3$, $1 \rightarrow_c 4$, $4 \rightarrow_c 5$, $4 \rightarrow_c 6$ となり， $1 \rightarrow_c 5$, $1 \rightarrow_c 6$ は存在しない．よって， ACFG および CFG を利用した際の区間限定スライスは，それぞれ $\hat{S}_b(1, 6, \{\text{loc}\}) = \{\ 1, 4, 6\ \}$ (ACFG), $\hat{S}_b(1,$

```
      #include <stdio.h>
      #define  MAX_WORD_LEN  32
      #define  MAX_LINE_LEN  50

      pretty_text(int argc, char *argv[])
      {
          FILE *fp;
          char word[MAX_WORD_LEN];
          char ch;
          int len, loc;
          int lc, wc;

 1        if (argc >1)
 2, 3         fp = fopen(argv[1], "r");
          else
 4            exit(1);
(5)
 6        loc = 0;
 7        lc = 1;
 8        wc = 0;
 9        ch = ' ';
 10       len = 0;
 11       printf("%3d:  ", lc);
 12       while ch != EOF) {
 13           if (len != 0) {
 14               if (loc + len >= MAX_LINE_LEN) {
 15                   loc = len;
 16                   lc++
 17                   printf("%3d:  ", lc);
                  }
                  else {
 18                   loc += len +1;
 19                   printf("  ");
(20)              }
 21               printf("%s", word);
(22)          }
 23           len = 0;
 24, 25       ch = getc(fp);
 26           while (ch != ' ' && ch != '\n' && ch != EOF) {
 27               if (len < MAX_WORD_LEN) {
 28                   word[len] = ch;
 29                   len++;
(30)              }
 31, 32           ch = getc(fp);
              }
 33           word[len] = '\0';
 34           wc++;
          }
 35       fclose(fp);
 36       printf("\n");
 37       printf("total words = %d\n", wc);
      }
```

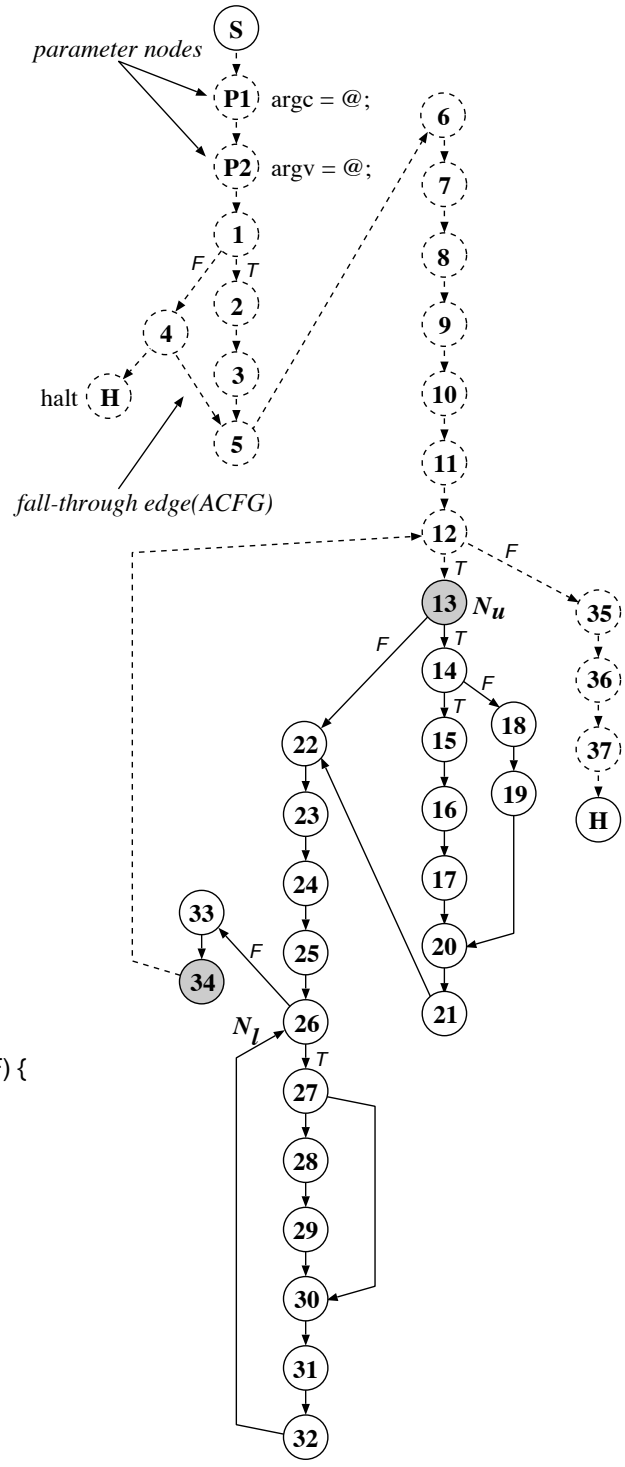


図 3.6: 部品作成対象のプログラムと CFG (ACFG)

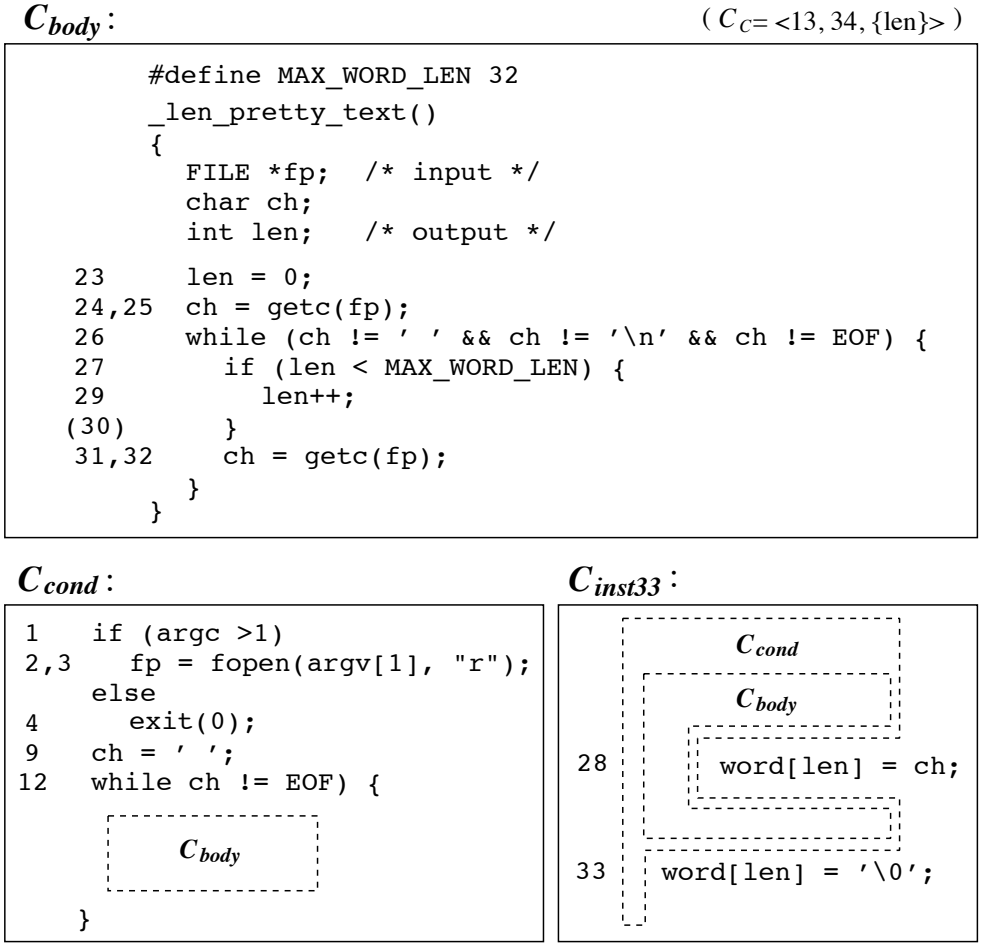


図 3.7: 図 3.6のプログラムから作成したソフトウェア部品

6, {loc}) = { 6 } (CFG) となる．本部品作成手法では，制約付き到達可能経路を求める際に CFG を用いるので，節点 1, 4 のように着目する下限節点に到達しない例外処理部分が削除される．

3.5 評価実験

本部品作成手法の有効性を確認するために，本部品作成手法に基づいてワークステーション上にソフトウェア部品作成実験システムを構築した．このシステムにより部品を抽出する様子を図 3.10, 3.11に示す．部品作成者は，図 3.10に示すウインドウの CFG において，区間 (上限節点と下限節点) を指定する．さらに，変数一覧テーブルから着目する変数を選択

```
      _loc_pretty_text()
      {  FILE *fp;  /* input */
         char ch;
         int len;   /* input */
         int loc;   /* input, output */
 13      if (len != 0) {
 14        if (loc + len >= MAX_LINE_LEN) {
 15          loc = len;
           }
           else {
 18          loc += len +1;
(20)       }
(22)    }
 23     len = 0;
 24,25  ch = getc(fp);
 26     while (ch != ' ' && ch != '\n' && ch != EOF) {
 27       if (len < MAX_WORD_LEN) {
 29         len++;
(30)       }
 31,32     ch = getc(fp);
        }
      }
```

(a) C_C= <13, 34, {loc}>

```
     _wc_ch_pretty_text()
     {
       FILE *fp;  /* input */
       char ch;   /* output */
       int wc;    /* output */
24,25  ch = getc(fp);
26     while (ch != ' ' && ch != '\n' && ch != EOF) {
31,32    ch = getc(fp);
       }
34     wc++;
     }
```

(b) C_C= <13, 34, {wc, ch}>

図 3.8: 図 3.6のプログラムから作成したその他の部品本体

し，Extract ボタンを押すと，図 3.11に示すようにソフトウェア部品のコード (部品本体，入力パラメータ，出力パラメータ，実行条件例，利用例) が提示される．

このソフトウェア部品作成実験システムを用いて，既存のプログラムからソフトウェア部品を作成する実験を行った．本部品作成手法に対する評価項目として以下の 4 点を設定した．実験結果については，3.6節で考察する．

```
     _len_pretty_text()
     {
        FILE *fp;  /* input */
        char ch;
        int len;   /* output */
 9      ch = ' ';
 10     len = 0;
 12     while ch != EOF) {
 23        len = 0;
 24,25     ch = getc(fp);
 26        while (ch != ' ' && ch != '\n' && ch != EOF) {
 27          if (len < MAX_WORD_LEN) {
 29             len++;
(30)         }
 31,32       ch = getc(fp);
          }
        }
     }
```

(c) C_C = <6, 22, {len}>

```
     _len_pretty_text()
     {
        int len;  /* input, output */
 27     if (len < MAX_WORD_LEN) {
 29        len++;
(30)    }
     }
```

(d) C_C = <27, 32, {len}>

図 3.9: 図 3.6のプログラムから作成したその他の部品本体

(1) 任意の区間を指定した際，部品本体から不要なコードが削除されるか (節点数の比較).

(2) 区間を変化させることで，機能の異なる部品が作成可能か (入力パラメータの比較).

(3) 異なるプログラムから作成した同じ機能を持つ部品本体のコードは等しいか (機能とコードの比較).

(4) 生成される付加情報は部品を再利用するのに役立つか (実行条件例と利用例のコード).

部品作成対象プログラムは，表 3.1に示す NetBSD のソースコードである． LOC はプログラムの行数を指す．本部品作成手法で扱わない再帰構造と高次のポインタ型変数については，前処理によって部品作成対象から除いた.

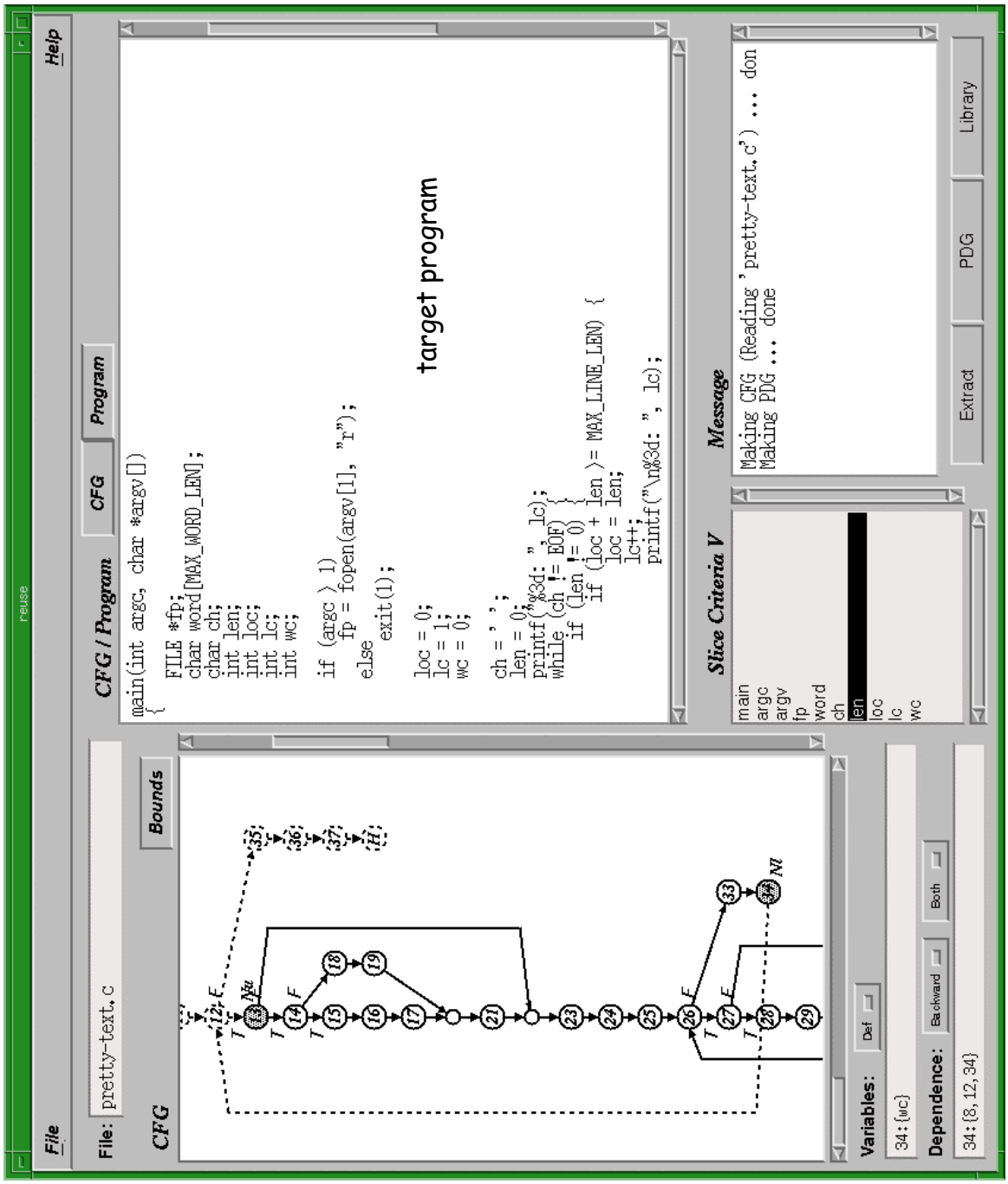


図 3.10: 部品作成実験システム (部品作成基準の指定)

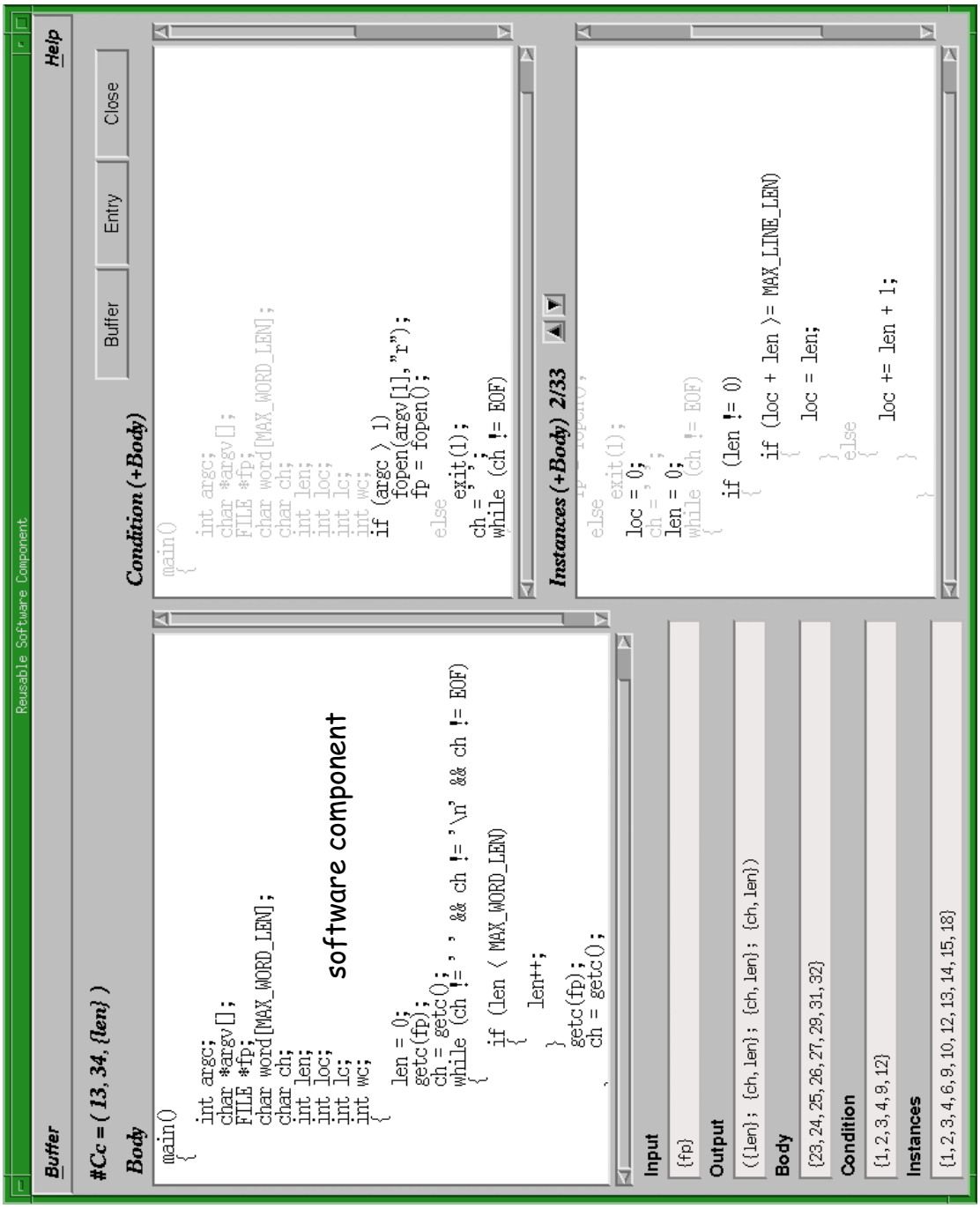


図 3.11: 部品作成実験システム (抽出されたソフトウェア部品)

表 3.1: 実験対象プログラム (ソースコード)

プログラム	LOC	関数の数	節点数	作成部品
whois	131	2	81	A1 ∼ A4
finger	1240	24	799	B1 ∼ B5
ftp	5595	128	4399	C1 ∼ C6

表 3.2: 表 3.1のプログラムから作成したソフトウェア部品

部品	部品作成基準	機能
A–1	⟨ w20, w49, {connect} ⟩	リモートサーバとの結合
A–2	⟨ w17, w29, {s} ⟩	ソケットのオープン
A–3	⟨ w30, w69, {sfi}⟩	データのポインタの設定
A–4	⟨ w20, w75, {ch}⟩	データの読み込み
B–1	⟨ n9, n42, {connect} ⟩	リモートサーバとの結合
B–2	⟨ n26, n36, {s} ⟩	ソケットのオープン
B–3	⟨ n9, n78, {fp} ⟩	データのポインタの設定
B–4	⟨ n54, n76, {c} ⟩	データの読み込み
B–5	⟨ n35, n76, {c}⟩	データの読み込み
C–1	⟨ f2, f30, {connect} ⟩	リモートサーバとの結合
C–2	⟨ c38, f30, {connect} ⟩	リモートサーバとの結合
C–3	⟨ f11, f66, {s} ⟩	ソケットのオープン
C–4	⟨ f22, f91, {cin} ⟩	入力データのポインタの設定
C–5	⟨ m68, c49, {port} ⟩	ポート番号の取得
C–6	⟨ m68, f92, {hostname} ⟩	ホスト名の取得

本部品作成手法に基づき作成したソフトウェア部品の機能を表 3.2，部品の各要素の節点数を表 3.3，入力パラメータを表 3.4に示す．表 3.2の部品作成基準において， w はソースコード whois.c (whois)， n は net.c (finger)， m, f, c は main.c, ftp.c, cmds.c (ftp) を指す．表 3.1, 3.2, 3.3, 3.4において，部品 A–1 ∼ A–4 は whois， B–1 ∼ B–5 は finger， C–1 ∼ C–6 は ftp から作成した部品を表す．表 3.3の C_{inst} は $R = 1$ で作成した．

表 3.3, 3.4において，本部品作成手法で作成した部品と次の 3 つの従来手法で作成した部品を比較する．従来手法 1 では，ソフトウェア部品を作成する際に指定した区間を含む範囲を関数単位で切り出して部品を作成する．従来手法 2 では，作成したソフトウェア部品と同じ節点および変数に着目し，従来のスライシングを単純に適用することで部品を作成する．

表 3.3: 表 3.2のソフトウェア部品の節点数

部品	ソフトウェア部品 (提案手法)				従来手法 1		従来手法 2		従来手法 3	
	CRP	C_{body}	C_{cond}	C_{inst}	$n1$	$C_{body}/n1$	$n2$	$C_{body}/n2$	$n3$	$C_{body}/n3$
A–1	21	12	32	1	81	15%	44	27%	44	27%
A–2	9	4	23	6	81	5%	27	15%	27	15%
A–3	29	10	39	3	81	12%	49	20%	49	20%
A–4	42	19	38	1	81	23%	57	33%	57	33%
B–1	28	21	67	1	83	25%	88	24%	37	57%
B–2	9	2	72	6	83	2%	74	3%	23	9%
B–3	61	23	70	4	83	28%	93	25%	42	55%
B–4	25	24	91	1	83	29%	115	21%	64	38%
B–5	39	27	88	1	83	33%	115	23%	64	42%
C–1	22	15	59	1	96	16%	74	20%	21	71%
C–2	32	23	51	1	185	12%	74	31%	40	58%
C–3	39	18	75	3	96	19%	93	19%	41	44%
C–4	39	17	85	5	96	18%	102	17%	50	34%
C–5	21	10	41	2	175	6%	51	20%	49	20%
C–6	77	28	77	1	271	10%	105	27%	103	27%

従来手法 3 は従来手法 1 と 2 を組み合わせた手法，すなわち，関数単位で区間を設定し，従来のスライシングを適用して部品を抽出する手法である．従来手法 1, 2, 3 により作成された部品の節点数 $n1$, $n2$, $n3$ に対して，本部品作成手法による部品本体の節点数 C_{body} がどの位減少しているのかを示すために，節点数の割合 $C_{body}/n1$, $C_{body}/n2$, $C_{body}/n3$ を表 3.3の各節点数 $n1$, $n2$, $n3$ の右欄に示す．また，表 3.4において，変数名 x が同じでも実行時に値が異なる可能性を持つ変数を x, x', x'' と表す．

ここで，スライシングを部品作成に適用する手法は 3.2.2節で紹介したように複数存在するが，本実験では区間を指定することによる利点を明確にするため，従来のスライシングとして 3.2.1節で定義したものを用いた．

3.6 考察

本節では，3.4節のソフトウェア部品の例と 3.5節の評価実験をもとに，本部品作成手法の有効性について考察する．評価項目 (1) ～ (3) の部品本体および入出力パラメータに関す

表 3.4: 表 3.2のソフトウェア部品の入力パラメータ

部品	C_{in} (提案手法)	従来手法 1	従来手法 2	従来手法 3
A–1	host	argc, argv	argc, argv	argc, argv
A–2	host, argc′	argc, argv	argc, argv	argc, argv
A–3	s,hp	argc, argv	argc, argv	argc, argv
A–4	host	argc, argv	argc, argv	argc, argv
B–1	host	name	argc, argv	name
B–2	hp	name	argc, argv	name
B–3	host	name	argc, argv	name
B–4	s	name	argc, argv	name
B–5	sin, hp	name	argc, argv	name
C–1	host, port	host, port	argc, argv	host, port
C–2	port, argc′, argv′	sp, argc′, argv′	argc, argv	sp, argc′, argv′
C–3	hp, histaddr, port	host, port	argc, argv	host, port
C–4	hp, histaddr, port	host, port	argc, argv	host, port
C–5	argc″, argv″	argc, argv	argc, argv	argc, argv
C–6	argc″, argv″	argc, argv	argc, argv	argc, argv

る考察は 3.6.1節で，評価項目 (4) の付加情報に関する考察は 3.6.2, 3.6.3節で行う．

3.6.1 部品本体および入出力パラメータ

本部品作成手法で作成した部品本体と従来手法 1, 2, 3 で作成した部品を比較し，3.1節で述べた本部品作成手法の利点 (1) を確認する．本節では，従来手法 1, 2, 3 を用いて作成した部品をそれぞれ従来部品 1, 2, 3 と呼ぶ．また，図 3.7に示す部品本体と同じ節点および変数に着目して作成した従来のスライス (従来部品 2) を図 3.12に示す．図 3.12において，節点番号の左側の ∗ は本部品作成手法の部品本体に含まれない文を指す．

(1) 従来部品と部品本体の節点数

図 3.7, 3.12より，本部品作成手法による部品本体は従来のスライスと比較して節点 1, 2, 3, 4, 9, 12 を含まない．節点 1, 4 は例外処理コードである．節点 9, 12 が含まれないのは，部品作成基準として指定した区間が，もとのプログラムの繰り返し (節点 12 の while 文) 内部に含まれていたためである．また，表 3.3において，部品本体内のすべての節点は従来部

```
      #define  MAX_WORD_LEN  32
      _slice_pretty_text()
      {
        FILE *fp;
        char ch;
        int len;
*  1    if (argc >1)
*  2,3    fp = fopen(argv[1], "r");
*       else
*  4      exit(1);
*  9    ch = ' ';
* 12    while ch != EOF) {
   23      len = 0;
   24,25   ch = getc(fp);
   26      while (ch != ' ' && ch != '\n' && ch != EOF) {
   27        if (len < MAX_WORD_LEN) {
   29          len++;
  (30)       }
   31,32     ch = getc(fp);
           }
*       }
      }
```

図 3.12: 従来のスライシングにより抽出した部品 ($S_b(34, \{\text{len}\})$)

品 1, 2, 3 に含まれ，部品本体の節点数 C_{body} は $n1$, $n2$, $n3$ より必ず小さくなる．これは，部品の抽出対象を区間 CRP に限定しているためで，削除されたコードは着目した区間の外に存在していたコードか例外処理により部品外部に制御が移るコードである．

表 3.3において，C_{body} と $n1$, $n2$ を比較すると，節点数の割合は $C_{body}/n1 = 2\% \sim 33\%$, $C_{body}/n2 = 3\% \sim 33\%$ となる．よって，スライシングを用いずに関数単位で部品を切り出す場合や，単純にスライシングだけを適用した場合と比較すると，本部品作成手法は 2/3 以上の節点を従来部品から取り除いていることがわかる．さらに，C_{body} と $n3$ を比較すると，節点数の割合は $C_{body}/n3 = 9\% \sim 71\%$ となり，関数単位で区間を設定したときと比べても，本部品作成手法は要求する機能を満たすために不必要な節点を従来より多く取り除くことが確認できる．特に，A–1 ～ A–4, B–2, C–4 のように，着目する変数に依存関係を持つコードが関数内部で局所的に存在するとき，本部品作成手法は有効である．さらに，本部品作成手法では複数の関数にまたがるコードが関数呼出しを経由して密集している，つまり，関数呼出しの直前に呼び出す関数の入力パラメータを設定するコードが記述されているときにも有効である． C–5, C–6 は複数の関数にまたがり区間を指定して作成した部品で，

節点数の割合が小さいことより，上記の効果が確かめられる．

このように，スライシングを部品作成に適用するだけでなく，区間を指定可能とすることで，本部品作成手法は部品作成に対する効果を持つ．以上より，3.5節の評価項目 (1) に関して，次に示す効果が確認できる．

(a) 任意の区間を指定して，繰り返し文の内部や複数の関数からでも部品本体を抽出することが可能で，指定した区間において着目した変数に影響を与えない記述コードが部品から従来より多く削除される．

ここで，図 3.8a では，節点 34 の変数 loc を計算するのに不要なコード (節点 23, 24, 25, 26, 27, 29, 31, 32) が部品本体に含まれている．このように，指定した区間が繰り返しの内部にあり，指定した変数が区間の下限節点で定義あるいは参照されていないとき，不要なコードが含まれることがある．これに対しては，部品作成システムで区間の指定後に毎回依存関係の再計算を行うか，CFG において繰り返しの戻り矢印に自己代入文を持つダミー節点を自動的に挿入することで回避することができる．例えば，図 3.6の戻り矢印 (34 $\rightarrow$ 12) にダミー節点 (loc = loc) を挿入すると，ダミー節点は制約付き到達可能経路に含まれないので，データ依存関係が途切れ，上記の不要な節点が部品本体から削除される．

(2) 指定する区間と入力パラメータ

図 3.12に示す従来のスライスは必ず開始節点まで依存関係をたどって作成するため，節点 34 の変数 len に着目した場合，節点 1, 2, 3, 4 が無条件で部品内部に含まれる．このため，入力パラメータとして変数 fp を持つ部品は作成不可能である．これに対して，図 3.7に示す本部品作成手法で作成した部品本体は節点 1, 2, 3, 4 を含まない．よって，変数 fp をソフトウェア部品の入力パラメータとすることが可能である．

このように，従来のスライシングだけを適用して部品を作成した場合，依存関係をプログラムの開始節点までたどるため，作成される部品の入力パラメータは部品作成対象プログラムの入力パラメータに支配される．よって，部品の入力パラメータとしてプログラムの入力パラメータ以外の変数を設定できない．このことは，表 3.4において，着目変数を変化させても従来部品 2 の入力パラメータが部品作成対象プログラムの入力パラメータとすべて等しいことより確認できる．同様に，従来手法 3 では依存関係を関数の先頭節点までたどるため，作成される従来部品 3 の入力パラメータは関数の入力パラメータに支配される．よって，表 3.4に示すように，従来部品 3 の入力パラメータは関数，つまり，従来部品 1 の入力パラメータとすべて等しい．

これに対して，本部品作成手法では着目する変数を同じものに指定しても，指定する上限節点により入力パラメータ C_{in} をさまざまな変数に設定できる．これは，従来部品 1, 2, 3 の入力パラメータと C_{in} を比較すれば明確である．さらに，部品 B–4 と B–5 は着目する変数が節点 n76 の c で同じであるが，異なる節点 n54 と n35 を上限節点として指定することより，C_{in} を変えている．このように，本部品作成手法において指定する上限節点は作成する部品本体の入力パラメータに影響を与える．

部品の入力パラメータは部品の機能を決定する要素の一つであるとみなせるので，3.5節の評価項目 (2) に関して，次に示す効果が確認できる．

(b) 入力パラメータとなる変数の選択肢が広がり，従来のスライシングでは抽出できない機能を持つ部品が作成可能である．

(3) 部品の機能と部品本体コード

本部品作成手法により作成した部品本体の機能とコードとの関係を調べるために，同じ機能を持つように部品作成基準を指定して部品を作成した．表 3.2, 3.3, 3.4に示す A–1 と B–1 のように，部品作成基準を調節することで同じ機能の部品本体を抽出しても，2 つの部品本体の節点数 C_{body} およびコードは大きく異なることがある．また，A–2 と B–2 のように，C_{in} がまったく異なることもある．これは，部品作成対象プログラム A と B の指定した区間において，プログラム構造が大きく違っていたためである．逆に，A–1 と C–1 では，入力パラメータ C_{in} が異なりはするが，C_{body} は 12 と 15 でほぼ等しく，2 つの部品本体コードは類似していた．これは，部品作成対象プログラム A と C のプログラム全体で構造が大きく異なっていても，指定した区間において A と C のプログラム構造が似ていたためである．

以上より，3.5節の評価項目 (3) に関して，次に示す効果が確認できる．

(c) 本部品作成手法で作成される部品本体は部品作成対象のプログラムがどのように実現されていたかによって大きく影響を受けるが，指定する区間を調整することで異なるプログラムからでも同じ機能を持つ部品が作成可能である．

(4) 出力パラメータの選択

定義 3.11の出力パラメータについて考察する．V_d 内の変数は部品本体内部で値が定義されることだけしか保証されていないので，部品作成対象のプログラムと部品本体で意味が異

なる変数が存在する可能性を持つ．これに対して，V_s 内の変数に関連するコードは部品本体に必ず含まれる．よって，$V_d \setminus V_s$ 内の変数が部品作成対象プログラムと部品本体で，意味が保証されない変数を表す．V_e 内の変数は部品本体外部で少なくとも一度は利用されているので，一時変数である可能性は低い．逆に，$V_d \setminus V_e$ 内の変数は一時変数である可能性が高く，部品作成基準変数から取り除く方がよいといえる．

3.6.2 実行条件例

部品を再利用してプログラムを作成する場合，その部品を実行することで動作を確認したいという要求がある．一般に，部品を実行するためには部品の入力変数に適切な値を設定する必要がある．しかし，実行に必要な情報が部品に附属しているとは限らない．部品を実行させたいという要求に応じるために，本部品作成手法では部品本体だけでなく実行条件例を提示する．

実行条件例により部品の動作確認が可能になると，検索で見つかった類似部品から適切なものを選択したり，部品を変更するのに有利である．部品が目的の機能を満たしているかどうかを調べる場合，もとのプログラムのテストデータを部品本体と実行条件例を合わせたプログラムに入力し，実行経路を確認することが考えられる．また，任意のデータをそのプログラムに入力し，部品本体の入口で実行をとめる．その時の変数の値を記録しておくことで，部品本体に対するテストデータが生成される．このように，部品のテストデータを変数の値でなくコードとすることで，部品を再利用する際に動的にテストデータを生成可能である．さらに，変更により新しく作成した部品に対して，変更前と同じ実行条件例を組み合わせて実行を試みることで，以前の機能に対しても正しく動作するのか，以前の機能に対して変更が反映されているのかを検査することができる．

例えば，図 3.7において C_{body} と C_{cond} を組み合わせたプログラムを C_{prog} とおく．C_{prog} はファイル名を引数として指定可能な「単語の長さを数える」機能を持つプログラムであり，実際に実行して動作確認ができる．また，引数となるファイルを与えて C_{prog} を実行することで，部品本体のテストデータとして変数 fp の値が取得できる．さらに，部品本体の while (節点 26) 文の条件に式 `ch != '\t'` を新しく追加し，実行条件例と組み合わせて，もとのテストデータで実行を試みることで，部品内部の変更が部品外部にどのような影響を与えているのかを調べることができる．表 3.3に，本部品作成手法で生成した実行条件例の節点数 C_{cond} を示す．表 3.3において，各ソフトウェア部品の C_{cond} の数は異なり，それぞれ独自のテストパターンを持つことがわかる．

このように，実行条件例は部品本体を実行する際に変数の値を設定するコードとみなせ，部品本体を実行するために必要なコードの一例となる．よって，部品本体に対して少なくとも一つの実行パターンを保証する．また，コード自体をテストケースとみなすことで，多様で柔軟なテストが適用できる．

3.6.3 利用例

部品を再利用して作成中のプログラムに組み込む場合，どのように利用可能であるのかを知りたいという要求がある．この要求に応じるために，本部品作成手法は利用例を提示する．利用例は，部品がどの機能を担っていたのかを理解する指針となる．一般に，部品の機能を理解しようとする場合，部品内部のコードを解析することで出力を明確にする方法と，部品を利用している部品外部のコードから出力を推測する方法が考えられる．利用例は，後者の方法を支援する．

例えば，図 3.7のソフトウェア部品において，変数 len の担う機能を知らずに，部品作成基準に変数 len を指定したとする． C_{inst33} より，部品本体の出力変数 len は配列型変数 word における位置を指していて， word[len] に一文字変数 ch の値と終端を表す定数'\0' を代入していることが確認できる．これは，変数 len が変数 word の配列の添字に利用可能であることや，変数 len が変数 word の文字列の長さを表していることを推測する手がかりとなる．表 3.3に， $R = 1$ として本部品作成手法で生成した利用例の節点数 C_{inst} を示す． $R = 1$ では， C_{inst} が部品本体の利用パターンと等しくなる．

また，利用例は部品の機能を理解するためだけでなく，実際に部品を再利用する際の記述コードであり，部品を埋め込む際の結合コードとなる．よって，利用例まで含めて再利用対象として扱うことが可能である．特に，本部品作成手法によって作成した部品本体には定義変数集合が空集合になる関数 (printf など) が含まれないので，実際に利用する際には利用例を結合する機会は多いと考えられる．

3.7 あとがき

本章では，ソースコードを再利用対象とするとき，プログラムスライシングによって，既存のプログラムからソフトウェア部品を作成する手法を提案した．本部品作成手法は，抽出した部品が実行可能性と抽出等価性という部品に不可欠な性質を満たすことを保証しながら，部品作成対象プログラムに対して任意の区間を指定して，部品抽出が可能であるという点で従来の手法より有利である．また，本部品作成手法において，部品作成者は区間と変数

を指定するだけでよく，ソフトウェア部品の作成は半自動である．よって，ソフトウェア開発者の部品作成に対する負担を軽減している．さらに，本部品作成手法では部品理解を支援するために，実行条件例と利用例を部品利用者に提示する．これらの実行条件例と利用例をそれぞれ部品本体と組み合わせて得られるプログラムは実行可能性と抽出等価性を満たすので，部品の実行動作や利用方法の確認が容易である．

本部品作成手法では，部品作成基準の区間と変数を部品作成者が指定し，この部品作成基準は作成される部品の有用性に大きく影響を及ぼす．このため，本手法では，利用者が区間を指定する際に，部品作成対象プログラムに対する知識を持つことを前提とする．ここで，このような前提をおくことができない場合，すなわち，知識のないプログラムから部品を作成する場合に本部品作成手法を適用することを考える．本部品作成手法では，上限節点をプログラムの開始節点に設定することで，例外処理コードを削除する点を除いて従来と同じスライスを出力する．さらに，本部品作成手法では着目する節点および変数を適当に指定した場合でも，部品を理解するのに役立つ付加情報が提示される．また，着目する変数が特定された場合，その変数を含む関数を本部品作成手法の区間として設定し，上限節点および下限節点を調整しながら繰り返し部品を作成することが考えられる．このように，本部品作成手法を知識のないプログラムに適用した場合でも，従来のスライシングによる部品作成手法以上の支援は可能である．また，付加情報を利用することで，区間を特定するための負担は．部品を新規に作成する際の負担より少なくなる．

第 4 章

重み付き依存グラフを用いた部品の洗練

4.1 はしがき

部品を直接再利用するソースコード部品化再利用において，プログラムの開発コストを減少するためには，開発者の要求する部品があらかじめライブラリに用意されている必要がある．これに対して，既存のプログラムから部品を抽出することで，部品作成の負担を軽減する方法が提案されている [1, 27, 28, 84]．また， 3章でも，区間限定スライスを用いて，既存プログラムから部品を半自動抽出する手法を提案した．しかしながら，部品がどのような形で再利用されるのかを部品作成時に予測することは困難であり，抽出した部品が開発者の要求を満たすとはかぎらない．要求する部品が見つからない場合，開発者は部品の再利用をあきらめるか，自分の要求に合わせて類似の部品を変更することになる．よって，ソースコードの部品化再利用を促進するためには，部品作成を容易にするだけではなく，作成した部品が無変更あるいは少ない変更で再利用可能であることが重要である．

本研究では，部品作成者の負担を増加させずに，部品変更に対する開発者の負担を軽減するため，開発者の過去の部品変更の履歴から将来再利用される可能性の高いソースコード断片を予測し，ライブラリ内の各部品を自動的に洗練する手法の検討を進めてきた [59, 66, 72]．本章では，プログラム依存グラフ (PDG) [29] における各節点間の依存関係に対して，開発者が再利用した部品の形および部品変更前後の部品の形の変化に基づき変動する 2 種類の重みを割り付けた重み付き依存グラフ (weighted program dependence graph: WPDG) を定義し，この依存グラフを用いた部品洗練手法を提案する．部品の形とは，部品内部に含まれるプログラム文を指す．また，部品の形の変化とは，変更前の部品内部に含まれるプログラム文と，変更後の部品内部に含まれるプログラム文の違いを指す．

これら 2 種類の重みを開発者の再利用ドメインの特性に適した部品を決定する指標として

用い，もとの部品のソースコードから重み値が大きい強依存関係で結合したソースコード断片を，洗練後部品の候補として抜き出す．ここで，再利用ドメインの特性とは，開発者が部品を再利用してどのような機能を持つプログラムを頻繁に作成するのかを指す．さらに，本章では，開発者の再利用ドメインにおいて，現在の重みに基づき部品洗練を適用することが妥当であるかどうかを判定するため，洗練前後の部品の重みから計算する洗練コストと洗練条件を定義する．本手法により部品洗練を行うことで，ライブラリ内の部品が開発者の要求する機能を満たす可能性が高くなり，開発者が類似の要求に応じて部品を適時変更する手間が減少する．

本章の構成は次の通りである． 4.2節では，洗練対象とする部品と部品変更を示し，重み付き依存グラフの定義，および，変更履歴に基づく本洗練手法の概要を述べる． 4.3節では，部品内部の依存関係に割り付ける 2 種類の重みを定義し，重みの変動操作と計算式を示す．4.4節では，依存関係の強度に基づき，重み付き依存グラフから洗練部品に含まれる節点を抽出する手続きを述べ，洗練コストおよび洗練条件を定義する． 4.5節では，評価実験の結果より本洗練手法の有効性について考察し， 4.6節で本洗練手法の限界と拡張を述べる．

4.2 重みを用いた部品洗練手法

本節では，洗練対象とする部品と本洗練手法が扱う部品変更を示す．次に，重み付き依存グラフを定義し，本洗練手法の概要を述べる．

4.2.1 部品洗練における仮定

本部品洗練手法は，個人あるいは小規模プロジェクトでの部品化再利用によるプログラム開発を支援対象とする．このような開発では，開発者が作成するプログラムは類似しており，自分あるいはプロジェクト内で作成した部品を再利用する機会は多く，過去の部品変更を模倣する可能性は高い．よって，開発者が変更した後の部品を開発者の要求を満たす部品とみなすと，部品の洗練に関して，次に示す 2 つの仮定を導くことができる．

仮定 1 部品内部で過去に頻繁に再利用されたソースコード断片は，開発者のライブラリにおいて，将来も同じ形で再利用される可能性が高い．

仮定 2 部品内部で過去に頻繁に変更されたソースコード断片は，開発者のライブラリにおいて，将来も同様の変更を受ける可能性が高い．

本洗練手法では，仮定 1 により，部品内部において頻繁に再利用された部分を特定する．また，仮定 2 により，頻繁に変更を受けた (依存関係が結合あるいは分断された) 部分を特定する．

4.2.2 洗練対象部品

本洗練手法が扱う部品は，手続き型言語 C のサブセットで記述されたプログラムで，静的依存解析が可能であることが前提である．

部品は，作成者が再利用対象としてライブラリ内に登録した部品本体と，部品本体内部の文に依存関係を持つ付加ソースコード断片で構成される．本章では，部品本体を C_{body}，付加ソースコード断片を C_{add}，これら 2 つを合わせた部品全体を C_{whole} と表す．C_{body} は，開発者がそのままの形で再利用すると予測されるソースコード断片の集まりである．また，C_{add} は，開発者が C_{body} と組み合わせて再利用可能な付加ソースコード断片，あるいは，洗練により C_{body} に付加される可能性のあるソースコード断片の集まりである．本洗練手法が対象とする部品は，3章で提案した区間限定スライシングに基づく部品作成手法を用いることで，容易に抽出可能である．

図 4.1に，洗練対象部品の例を示す．各文の左側に示す数字は，このソースコードから作成した PDG における節点の識別番号である．太字の文 (節点 1, 2, 3, 5, 6, 7, 9, 10, 13) は C_{body} であり，「入力データの合計を表示する」機能を持つ．その他の文 (節点 4, 8, 11, 12) は C_{add} であり，「入力データの個数を求める」機能を持つソースコード断片と「平均値を求め，表示する」機能を持つソースコード断片を含む．このように，本洗練手法では，C_{body} と C_{add} は明確に区別されて，開発者に提示される．

4.2.3 部品変更

本洗練手法を用いたプログラム開発において，開発者は C_{whole} 内部の C_{body} をそのまま，あるいは，適時変更して再利用する．本洗練手法では，部品変更前後で同一な節点を識別するために，C_{whole} の各文に PDG の節点番号をタグとして割り付け，無変更な文に対して変更前後で共通のタグ番号を持たせる．タグ付け機構とバージョン管理機構を有するエディタを用いることで，以下に示す 5 つの部品変更操作を洗練システムが検出する．

(a) C_{body} に C_{add} 内のタグを持つ文を付加する．

(b) C_{body} 内の文を削除する．

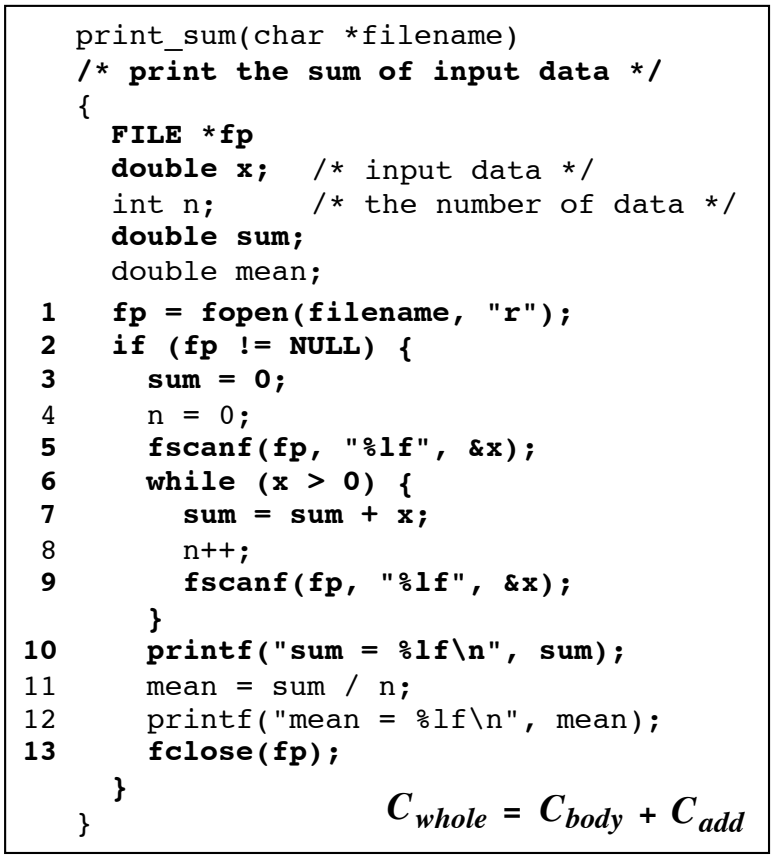


図 4.1: 洗練対象部品の例

(c) C_{body} にタグを持たない新規文を追加する．

(d) C_{body} 内の文をタグを変えずに移動する．ただし，移動した文と他のすべての文との依存関係が変わらないとき，その節点は無変更とみなす．他の文との依存関係に変化が生じたとき，この変更は移動文の削除と新規文の追加として扱う．

(e) C_{body} 内の文に対して，タグを変えずに内容だけ書き換える．この変更は書き換え文の削除と新規文の追加として扱う．

図 4.2に開発者が変更後に再利用した部品 C_{reuse} の例を示す． A, D, I, M, R は，それぞれ上記の (a) 付加操作， (b) 削除削除， (c) 追加操作， (d) 移動操作， (e) 書き換え操作を指す．ここで，本洗練手法が対象とする変更は，開発者の要求に合わせて部品のソースコードを改造することであり，誤りを含むソースコードの修正は変更として扱わない．

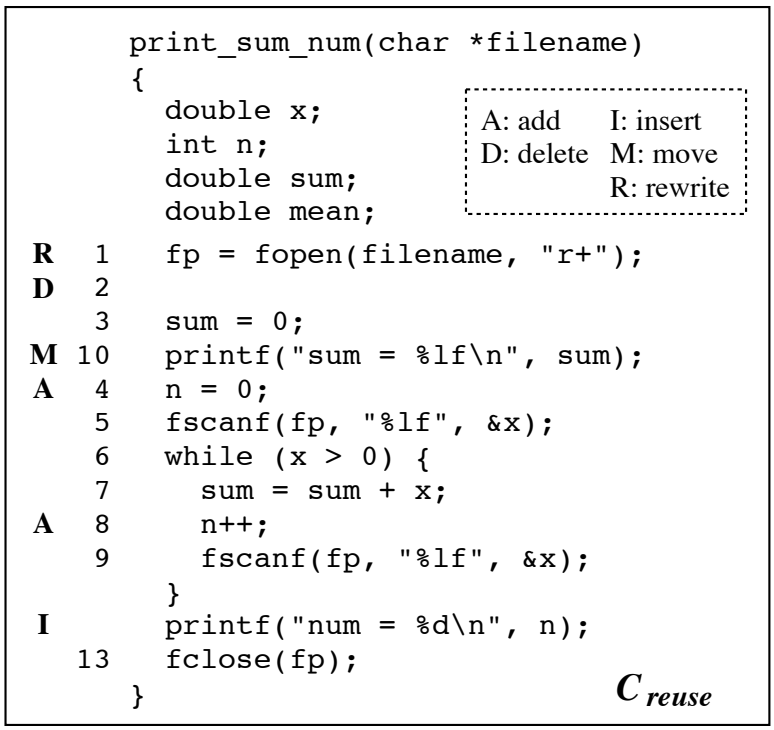


図 4.2: 変更後部品の例

4.2.4 重み付きプログラム依存グラフ

各部品に対する過去の利用および変更の履歴を蓄積するため，C_{whole} の PDG における節点間の依存関係矢印に重みを割り付ける．この PDG を，重み付き依存グラフ (WPDG) と呼ぶ．重みは矢印の接続元および接続先節点間の依存関係の強度を表す．

本洗練手法では，依存関係を持つ 2 つの節点を依存節点と呼び，WPDG 上の矢印を接続元および接続先の 2 つの依存節点により識別する．よって，依存関係矢印の重みを計算する際には，4.2.3節に示す 5 つの変更操作から，各矢印の依存節点の状態を調べ，C_{body} に付加する文に対応する付加節点の集合，および，C_{body} から削除する文に対応する削除節点の集合を求める．例えば，図 4.2に示す変更後部品において，付加節点は 4, 8，削除節点は 1, 2, 10 である．また，変更操作 (c), (d), (e) において，新規文の追加は新規節点と新規矢印の追加として扱い，新規矢印に新たに重みを割り付けることはしない．

本章では，節点 p から節点 q への重み w を持つデータ依存関係と制御依存関係を $p \xrightarrow{w} q$ と表す．強度に関係しない場合，節点 p と q の依存関係を $p \rightarrow q$ と表す．また，WPDG

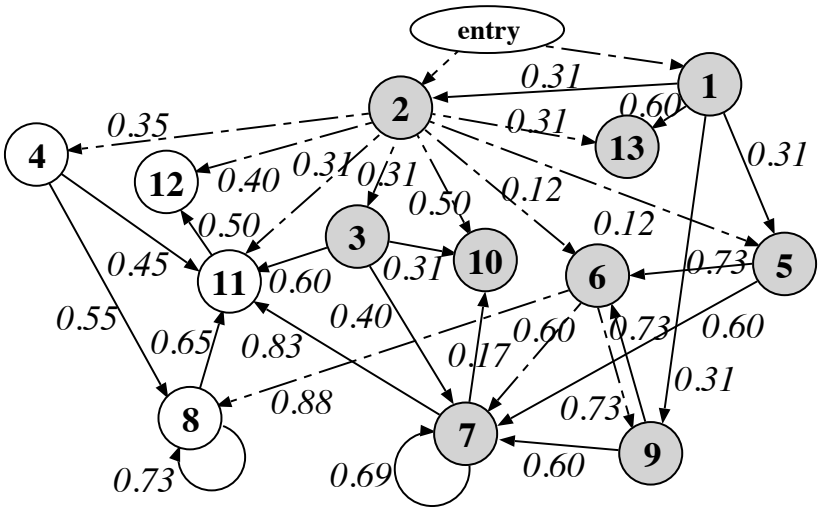


図 4.3: 重み付きプログラム依存グラフの例

(あるいは PDG) G 内の節点集合と矢印集合を $Node(G)$, $Edge(G)$ と表す．WPDG において $entry$ 節点は洗練対象とせず，この節点を接続元とする矢印には重みを割り付けない．

図 4.1に示す部品 C_{whole} に対して，部品の再利用と変更を何回か繰り返した後の WPDG を図 4.3に示す．制御依存矢印の属性 (T/F) は省略した．網かけの節点は部品本体 C_{body} を表す．

4.2.5 洗練手法の概要

部品洗練とは，部品変更履歴に基づく依存関係の強度に関する重みを指標として，部品本体 C_{body} の形を将来再利用される可能性が高いソースコード断片を含むように変形することを指す．図 4.4に本洗練手法の概要を示す．下側の WPDG は洗練前 (現在) の部品 C_{whole}，上側の WPDG は洗練後の部品 C'_{whole} を表す．各 WPDG において，網かけの節点は洗練前の部品本体 C_{body} と洗練後の部品本体 C'_{body} を指す．部品洗練の動作は，WPDG の各矢印に対する重み付けと，重みに基づく部品本体の抽出に分けられる．

開発者は，部品洗練システムの部品検索ブラウザを通して，ライブラリから要求部品を取得する．取得した部品 C_{whole} において，C_{body} をそのまま，あるいは，一部変更して再利用した場合を考える．C_{reuse} は，開発者が変更後に再利用した部品を指す．本洗練手法では，変更後の C_{reuse} の形から部品のどの部分を利用したのかに基づき変動する重み w^s と，変更前の C_{body} と変更後の C_{reuse} の形の変化から部品のどの部分を変更したのかに基づき変動す

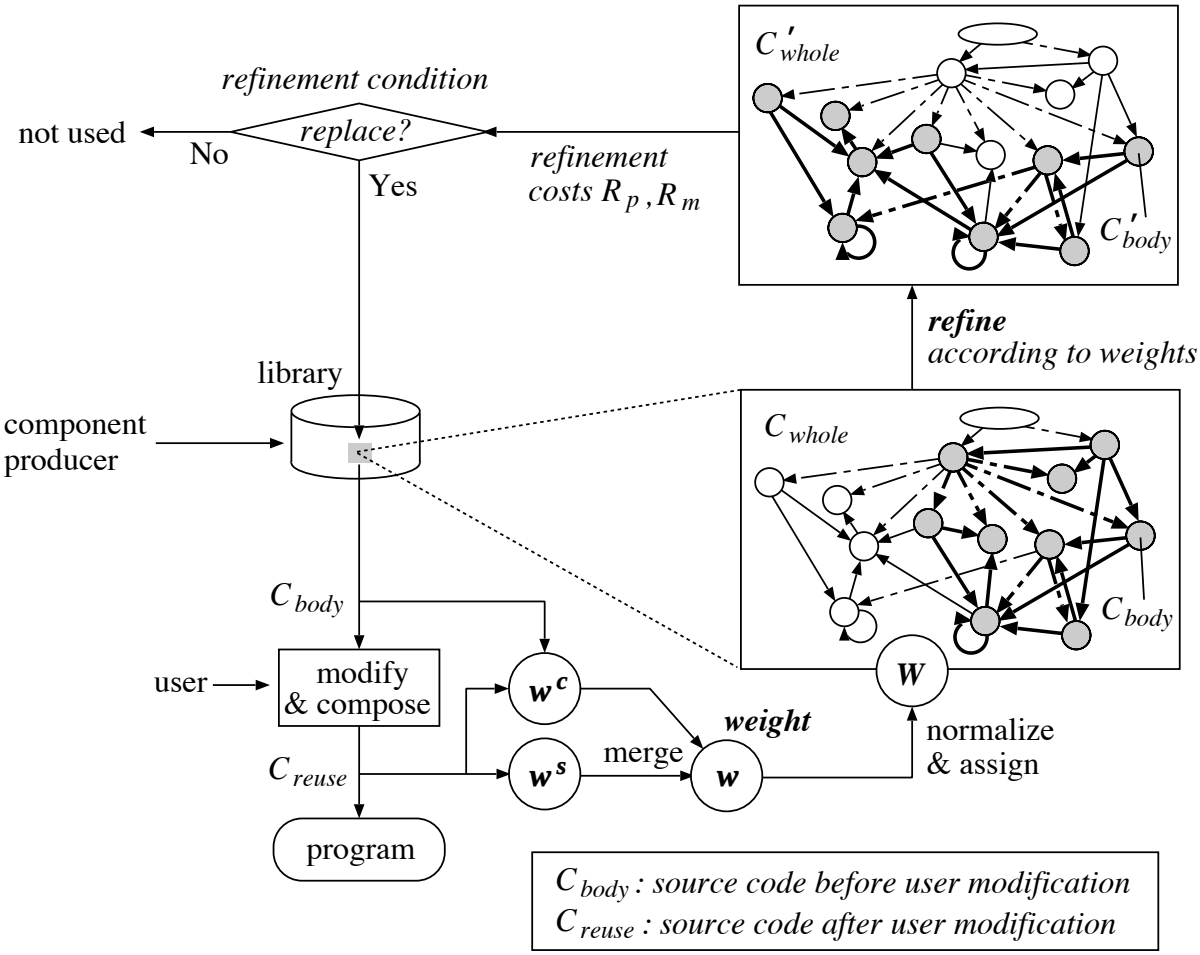


図 4.4: 部品洗練手法の概要

る重み w^c をそれぞれ求める．これらの重みに対して，合成演算および正規化演算を行った後の値 W を，C_{whole} の WPDG の各矢印に割り当てる．

重みが変動した際には，依存関係矢印の重みを指標として，C'_{body} に含まれる節点および矢印を C_{whole} から選択し，C'_{body} を含む C'_{whole} を依存関係を保存したまま作成する．C_{whole} と C'_{whole} は，内部に含む C_{body} と C'_{body} が異なるだけである．次に，洗練前後の部品 C_{body} および C'_{body} の形を維持するために必要なコスト R_p，依存関係の状態を修正するために必要なコスト R_m を計算し，これらのコストに関する洗練条件により洗練の妥当性を判定する．洗練条件が成立する場合，ライブラリ内の C_{whole} を C'_{whole} と置換し，洗練が終了する．洗練条件が成立しない場合は，C'_{whole} を破棄し，ライブラリ内の C_{whole} および C_{body} はそのままにする．

4.3 部品変更履歴に基づく重み付け

本節では，部品洗練における 2 つの仮定に基づく 2 種類の重みを提案し，それぞれの重みの変動操作と重みの計算式を定義する．

4.3.1 部品利用および変更に基づく重み

本洗練手法では，部品洗練における仮定 1, 2 に基づき， WPDG の各依存関係矢印に対して，次に示す 2 種類の重み w^s, w^c を定義する．

(1) **部品利用に基づく重み** w^s: 変更後に再利用した部品 C_{reuse} の PDG において，依存節点の接続関係に基づく重み．依存節点の接続関係とは，依存関係矢印の接続元節点と接続先節点が，変更後部品の内部に存在するのか，あるいは，外部に存在するのかという位置関係を指す．

(2) **部品変更に基づく重み** w^c: 変更前の部品本体 C_{body} と変更後に再利用した部品 C_{reuse} の PDG において，依存節点の接続関係の変化に基づく重み．依存節点の接続関係の変化とは，変更前後で依存関係矢印の接続元節点および接続先節点の位置関係がどのように変化したのか，つまり，依存関係矢印が結合あるいは分断されたのかを指す．

いま，将来再利用される可能性の高い部品の形を決定するという観点だけから部品洗練を行う場合，洗練における仮定 1 により，開発者が過去に頻繁に再利用したソースコード断片を特定すればよい．つまり，実際に開発者が再利用した変更後部品 C_{reuse} の形に応じて変動する重み w^s を導入するだけで十分である．

しかしながら，再利用した部品の形が同じでも，無変更で再利用した場合と変更を伴って再利用した場合では，開発者の負担に差があるはずである．例えば， C_{body} の形をまったく変えずに再利用した (C_{reuse} が C_{body} に等しい) 場合に比べて， C_{body} に文を追加して作成した C_{reuse} を再利用した場合の方が，開発者の負担は大きい．単純に変更後部品の形だけに注目して重みを変動させた場合，このような開発者の負担の差が部品洗練に反映されない．さらに，本洗練手法の目的は積極的に部品洗練を行うことで，開発者の部品再利用を支援することである．しかしながら，相反する再利用を繰り返した場合，例えば，ある 1 つの文を含む部品と含まない部品を順番に繰り返し再利用した場合，その文に対応する節点に接続された矢印の重み w^s は同じ値の間を振動する．このため，重み w^s の導入だけでは，開発者が過去に何度も部品変更を行っているにも関わらず，部品洗練がまったく行われない可能性がある．

そこで，本洗練手法では，同じ形の部品を再利用した場合でも開発者の負担の有無を区別し，洗練における仮定 2 に基づく重み w^c を導入する．仮定 2 において，同じ変更が将来も繰り返し行われることは，開発者に対して同じ部品変更の負担が将来も繰り返し生じることを指している．よって，仮定 2 に基づき，将来変更される可能性が高いソースコード断片を特定し，これらのソースコード断片が他のソースコード断片に比べて，部品洗練時に変更されやすくなるように重み付けを行うことは，開発者の部品変更に対する負担を軽減するという目的を達成するのに妥当である．さらに，開発者の変更を反映した重み w^c を変更後部品の形に基づく重み w^s と合成することで，相反する再利用を繰り返した場合でも，洗練により部品の形を変化させることが可能となる．

4.3.2 依存節点の位置関係

重み付けの際には， WPDG 上の接続元および接続先の 2 つの依存節点により，各依存関係矢印を識別する．つまり，接続元および接続先節点が変更前後でそれぞれ同一であるとき，それらを接続する矢印も同一であるとみなす．いま，矢印の方向を無視すると，依存関係矢印の接続関係は，部品内部と外部という 2 つの領域に対する依存節点の位置関係で表すことができる．依存関係矢印 $p \rightarrow q$ における依存節点の接続関係を表す論理式を以下に示す． G は WPDG (あるいは PDG) を指す．

結合: 依存節点 p, q が両方とも G に含まれる
$\Leftrightarrow BothIn(G,p,q) \equiv p \in Node(G) \wedge q \in Node(G)$.

分断: 依存節点 p, q が G の内部と外部に分断される
$\Leftrightarrow EitherIn(G,p,q) \equiv p \in Node(G) \wedge q \notin Node(G) \vee p \notin Node(G) \wedge q \in Node(G)$.

不定: 依存節点 p, q が両方とも G に含まれない
$\Leftrightarrow BothOut(G,p,q) \equiv p \notin Node(G) \wedge q \notin Node(G)$.
両依存節点とも G に含まれないとき，依存節点の関係は結合あるいは分断のどちらの状態か確定できないため，不定とする．

これらの論理式の真偽により， 4.3.1節の 2 種類の重みに対する変動操作を定義する．

図 4.5に，依存節点の接続関係の例を示す． WPDG G に存在する各矢印に対して，右側に示す論理式だけが真となる．ここで，矢印 $1 \rightarrow 5$ については，部品変更時に新規節点が追加されたため依存関係が破壊されている．しかし，これらの節点は新規に追加した節点

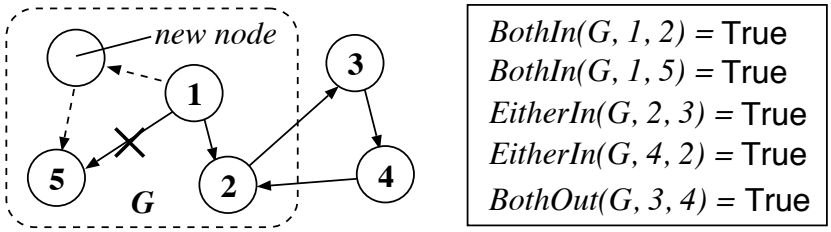


図 4.5: 依存節点の位置関係

を経由して依存関係を持つので，$BothIn(G, 1, 5)$ が真となる．このように，本洗練手法では，依存節点が新規に追加した節点を経由して間接的に依存関係を持つ場合も，もとの依存節点間に依存関係が存在するとみなし，依存関係矢印の状態を依存節点の位置関係だけから判定する．

4.3.3 依存節点の接続関係に基づく重み付け

洗練における仮定 1 に基づき，C_{whole} の WPDG G_{whole} において，過去に頻繁に再利用されたソースコード断片を特定する．このため，依存関係が保存される矢印の重みを増加させ，依存関係が破壊される矢印の重みを減少させる．G_{whole} に存在する依存関係矢印を $p \rightarrow q \in Edge(G_{whole})$，再利用時の部品 C_{reuse} の PDG を G_{reuse} と表す．G_{reuse} における依存節点の存在位置に基づき，重み w^s を次のように操作する．

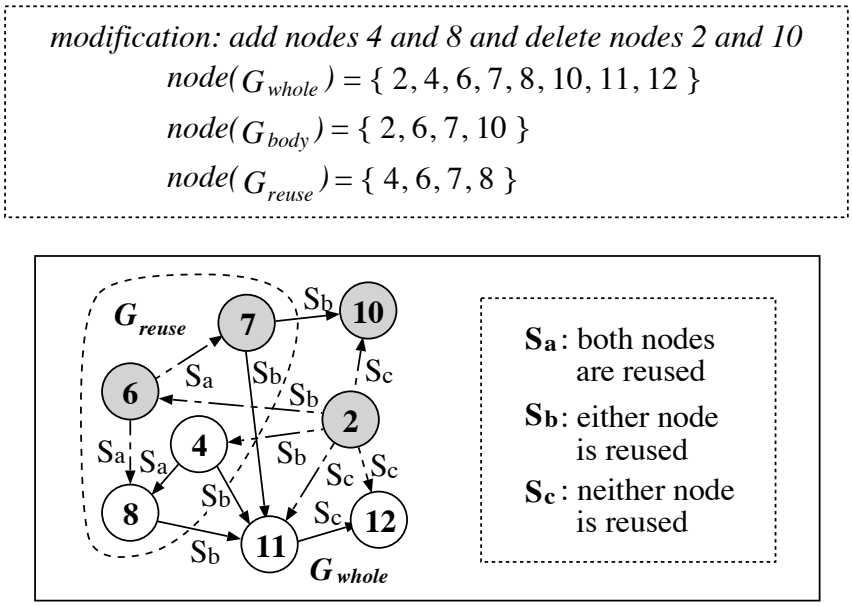
操作 Sa: G_{whole} 内の依存節点を両方とも再利用する ($BothIn(G_{reuse}, p, q)$ が真) とき，依存関係の強度を強くする (重み w^s 増加).

操作 Sb: G_{whole} 内の依存節点を片方だけ再利用する ($EitherIn(G_{reuse}, p, q)$ が真) とき，依存関係の強度を弱くする (重み w^s 減少).

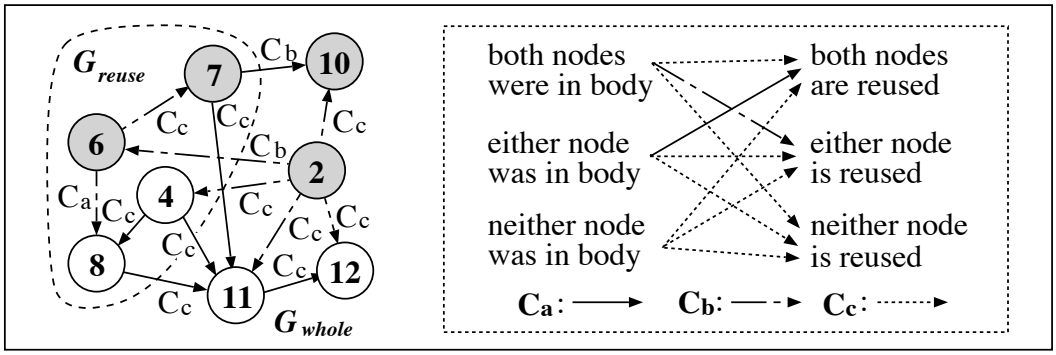
操作 Sc: G_{whole} 内の依存節点を両方とも再利用しない ($BothOut(G_{reuse}, p, q)$ が真) とき，依存関係の強度を変えない (重み w^s 変動なし).

図 4.6a に，重み付け操作 Sa, Sb, Sc を示す．C_{whole}, C_{body} は図 4.3の WPDG の一部である．網かけは C_{body} 内部の文に対応する節点を指す．

重み w^s は，開発者がどのような形で部品を再利用したかに影響を受ける．重み w^s の値が大きいほど，それらの依存節点を同時に再利用した割合が多いことを指す．逆に，重み w^s の値が小さいほど，それらの依存節点を片方だけ再利用した割合が多いことを指す．



(a) 依存関係矢印の接続関係に基づく重み w^s



(b) 依存関係矢印の接続関係の変化に基づく重み w^c

図 4.6: 依存関係矢印の接続関係と接続関係の変化に基づく重み付け操作

4.3.4 依存節点の接続関係の変化に基づく重み付け

洗練における仮定 2 に基づき，部品 C_{whole} の WPDG G_{whole} において，過去に頻繁に変更されたソースコード断片を特定する．このため，依存関係が分断から結合に変化する矢印の重みを増加，依存関係が結合から分断に変化する矢印の重みを減少させる． G_{whole} に存在する依存関係矢印を $p \rightarrow q \in Edge(G_{whole})$，変更前の部品本体 C_{body} の WPDG を G_{body}，再利用時の部品 C_{reuse} の PDG を G_{reuse} と表す． G_{body} と G_{reuse} における依存節点

の存在位置の変化に基づき，重み w^c を次のように操作する．

操作 Ca: G_{whole} において G_{body} の内部と外部に分断していた依存節点を両方とも再利用する ($EitherIn(G_{body}, p, q) \wedge BothIn(G_{reuse}, p, q)$ が真) とき，依存関係の強度を強くする (重み w^c 増加)．

操作 Cb: G_{whole} において G_{body} 内部で結合していた依存節点を片方だけ再利用する ($BothIn(G_{body}, p, q) \wedge EitherIn(G_{reuse}, p, q)$ が真) とき，依存関係の強度を弱くする (重み w^c 減少)．

操作 Cc: 部品変更前後で依存節点の位置関係が変化しない，または，依存節点を両方とも再利用しない ($BothIn(G_{body}, p, q) \wedge BothIn(G_{reuse}, p, q) \vee EitherIn(G_{body}, p, q) \wedge EitherIn(G_{reuse}, p, q) \vee BothOut(G_{reuse}, p, q)$ が真) とき，依存関係の強度を変えない (重み w^c 変動なし)．

図 4.6b に，重み付け操作 Ca, Cb, Cc を示す． C_{whole}, C_{body} は図 4.3の WPDG の一部である．網かけは C_{body} 内部の文に対応する節点を指す．

重み w^c は，開発者が部品に対してどのような変更を行ったかに影響を受ける．重み w^c の値が大きいほど，変更前の部品本体 C_{body} において内部と外部に分断していた依存節点を結合する変更を行った割合が多いことを指す．逆に，重み w^c の値が小さいほど， C_{body} において結合していた依存節点を分断する変更を行った割合が多いことを指す．依存節点の接続関係が変化しない，あるいは，変更前後の接続関係が不定のとき，重み w^c は変化しない．

このように，重み w^c は，現在 (変更前) の部品本体 C_{body} に対して，開発者が依存節点の接続関係を変更した場合にだけ変動する．現在の接続関係が結合のとき，重み w^c は負方向 (結合 $\rightarrow$ 分断) にのみ変動する．逆に，現在の接続関係が分断のとき，重み w^c は正方向 (分断 $\rightarrow$ 結合) にのみ変動する．よって，相反する再利用を繰り返した場合でも，重み w^c は必ず接続関係を変化させる方向にだけ変動し，部品洗練により接続関係が実際に変化するまでの間において，重み w^c がお互いに打ち消し合うことはない．重み w^c の絶対値が大きい依存関係矢印は過去に頻繁に変更を受けた部分，その符合は変更の方向 (分断 $\rightarrow$ 結合，あるいは，結合 $\rightarrow$ 分断) を表す．

4.3.5 重みの計算式

重みの変動定数を δ とすると，重み w^s, w^c の変動操作に対する式は，次のようになる．

(a) 重み増加： $w_n^t = w_{n-1}^t + \delta$

(b) 重み減少： $w_n^t = w_{n-1}^t - \delta$

(c) 変動なし： $w_n^t = w_{n-1}^t$

w^t は w^s あるいは w^c を指す．また， n は重み w^t の変動回数を指し， $n \geq 1$ の整数である．重みの初期値は， $w_0 = 0$ (中心値) に設定する．

本洗練手法では，洗練部品を作成する際，あるいは，洗練部品が開発者の再利用ドメインの特性に適しているかどうかを判定する際，重み w^s と重み w^c を組み合わせた指標を用いる．依存節点の接続関係に基づく重み w^s は，過去の部品再利用における依存関係の強度を示している．また，接続関係が変化するときに変動する重み w^c は，部品変更により依存関係矢印の結合あるいは分断が行われた部分に対して，重み w^s の変動を調整する役割を持つ．本洗練手法では，それぞれの重みの変動を δ の加算あるいは減算で達成している．よって，これらの重みを組み合わせた指標は，依存関係矢印の強度を示す主要素 w^s に，変動の調整要素 w^c を加算あるいは減算したものとなる．さらに，本洗練手法では， w^s が依存関係矢印の強度に与える影響の大きさと w^c が依存関係矢印の強度の変動を加速させる大きさを等しいとみなす．以上より，依存関係矢印の強度を表す重み w は，重み w^s と w^c を 1 対 1 の比で加算したものとなる．

$$w = \alpha \times w^s + \beta \times w^c = w^s + w^c \quad (\alpha = \beta = 1)$$

添字のない重み w, w^s, w^c は現在の値を指す．本論文では， $\alpha = 1$ および $\beta = 1$ と設定したが，本来は各ライブラリについて調整が必要である．

変更前後の部品において，依存節点の接続関係と重み w^s および w^c の変動の意味を表 4.1 に示す． $+$ は重み増加， $-$ は重み減少， 0 は重み変動なしを表す．表 4.1より，重み w^c が増加あるいは減少しているとき， w^s は必ず増加あるいは減少することが分かる．また， w^c は依存節点に関する矢印が結合あるいは分断されたときのみ $+$ あるいは $-$ となる．よって，無変更で依存節点を再利用した場合に比べて，変更を伴い依存節点を再利用した場合の重み w の変動は 2 倍になる．本論文における部品洗練の目的は部品変更の負担を軽減することであるので，過去に頻繁に変更された部分に対して，より早く洗練が行われるように重み w の変動を加速することは妥当である．

このように計算した重み w の変動範囲は $-2\delta n \sim +2\delta n$ となり，変動回数 n によって異なる．このため，再利用した回数が異なる部品どうしの重みを単純に比較できない．重みが

表 4.1: 重み w^s および w^c の変動値と変動の意味

変更前	変更後	w^s	w^c	変動値	意味
BI	BI	+	0	$+\delta$	無変更で再利用する
EI	BI	+	+	$+2\delta$	結合して再利用する
BO	BI	+	0	$+\delta$	変更は不定，再利用する
BI	EI	−	−	-2δ	分断して再利用しない (する)
EI	EI	−	0	$-\delta$	無変更で再利用しない
BO	EI	−	0	$-\delta$	変更は不定，再利用しない
BI	BO	0	0	0	変更は不定，再利用しない
EI	BO	0	0	0	変更は不定，再利用しない
BO	BO	0	0	0	変更は不定，再利用しない

BI(BothIn) は依存関係矢印が結合， EI(EitherIn) は矢印が分断，
BO(BothOut) は矢印の状態が不定を指す．

ライブラリ内の全部品に対して共通の指標となるためには，重みの変動が特定の範囲におさまるように正規化する必要がある．本洗練手法では，重みの変動の速度がパラメータによって設定できるという理由から，次の関数 [8] を用いて重み w を正規化する．

$$W = N(w) = \frac{1}{1 + e^{(-w/T)}} \quad (T > 0)$$

T は正規化関数のグラフの形を決定する定数である．正規化した重み W の変動範囲は $0 < W < 1$，重みの中心値は初期値 $W_0 = 0.5$ $(w_0 = 0)$ に等しい．定数 δ と T は，各ライブラリにおいて調節が必要である．

本洗練手法では，重み w^s と w^c の現在の値を蓄積し，これらを加算し正規化した W を部品 C_{whole} の WPDG G_{whole} の各依存関係矢印に割り付ける．図 4.3において， WPDG の各矢印に割り付けられている重みは，正規化後の値 W である．

4.4 重み付き依存グラフを用いた洗練部品の作成

本節では， WPDG の各依存関係の強度に基づき，部品 C_{whole} から強依存関係を持つソースコードを抽出することで，洗練部品を作成する手続きを示す．その際，部品内部の依存関係を維持するコストと洗練により依存関係を修正するコストを定義し，洗練の妥当性を判定する洗練条件を用いる．

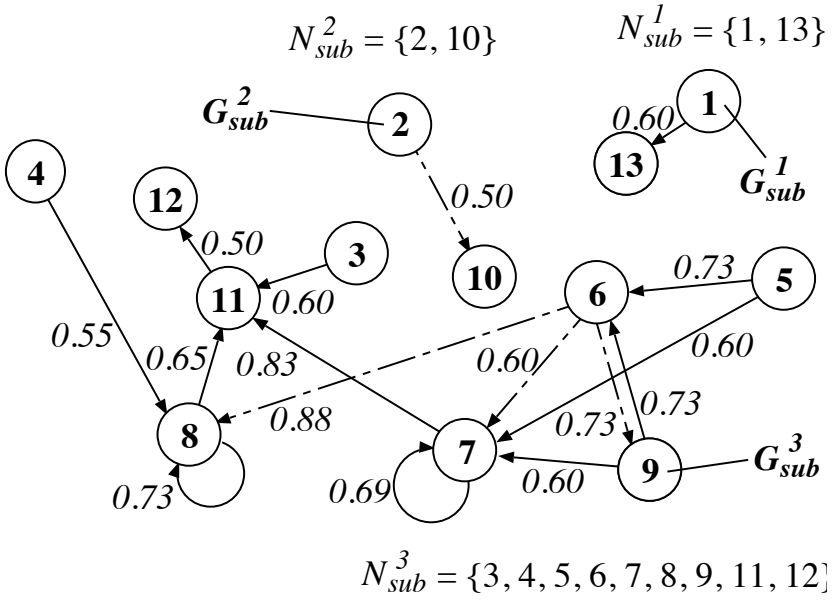


図 4.7: 強依存部分グラフと強依存節点集合

4.4.1 強依存部分グラフの抽出

過去の部品変更履歴に基づき，一定期間重みを蓄積した部品を考える．部品洗練に関する仮定と重みの定義より，開発者の再利用ドメインの特性に適した部品を作成する際，強度が強い依存関係を結合し，強度が弱い依存関係を切り離す．本洗練手法では，洗練後の部品本体に含む節点を選択する際，依存関係の強度にしきい値 θ を設定する．強結合 (重みがしきい値より大) と弱結合 (重みがしきい値より小) のしきい値は，正規化後の重みの中心値とする ($\theta = W_0 = 0.5$)．

C_{whole} の WPDG G_{whole} において，正規化後の W が θ を越える ($W \geq \theta$) 矢印だけを残し，W が θ を越えない ($W < \theta$) 矢印を消去する．これにより，強結合を持つ依存節点，つまり，強く影響を及ぼし合う節点だけを含み，弱結合を持つ依存節点を分断した複数の部分 WPDG が G_{whole} から抽出できる．抽出した部分 WPDG を強依存部分グラフ G_{sub}，G_{sub} 内に含まれる節点の集合を強依存節点集合 N_{sub} と表す．図 4.7に，図 4.3の WPDG から抽出した 3 つの G_{sub} と，それらの N_{sub} を示す．

本洗練手法では，N_{sub} 内の節点を洗練後の部品本体 C_{refine} 内部の節点とする．その際，部品本体の機能が単純になることを避けるために，これらの節点を結合する弱結合の矢印を

C_{refine} に追加する．洗練後の部品本体の WPDG G_{refine} を次のように定義する．

$$
\begin{aligned}
Node(G_{refine}) &\stackrel{\text{def}}{=} N_{sub} \quad (\#N_{sub} > 1) \\
Edge(G_{refine}) &\stackrel{\text{def}}{=} \{p \stackrel{W}{\rightarrow} q \in Edge(G_{whole}) \mid p \in N_{sub} \wedge q \in N_{sub}\}
\end{aligned}
$$

$\#N$ は集合 N の要素数を指す．G_{whole} に存在する N_{sub} 内の依存節点を結合する依存関係矢印は，たとえ依存関係の強度 W がしきい値 θ より小さくても，洗練後の部品本体に必ず現れる．図 4.7において，N^1_{sub}, N^2_{sub}, N^3_{sub} に対する G^1_{refine}, G^2_{refine}, G^3_{refine} が洗練後部品本体 $G_{body'}$ の候補となる．

4.4.2 重みの誤差による維持コストと修正コスト

依存関係の強度に基づき抽出した G_{refine} が洗練後の部品本体として妥当であるかどうかを判定するために，部品内部の依存関係に対して，維持コストと修正コストを定義する．本洗練手法では，これら 2 つのコストを，依存関係の強度に関する二乗誤差の和 (残差平方和) [98] で定義する．二乗誤差の和は，ニューラルネットワークにおける学習 [8] などで誤差を見積もる際に用いられ，本洗練手法のように重みの誤差でコストを定義するのに適している．

維持コストとは，部品本体の C_{body} が現在の形を維持するために必要なコストである．このコストは，洗練前の G_{body}，あるいは，洗練後の部品本体の候補 G_{refine} において，各矢印の現在の強度が現在の結合状態を維持する強度とどの程度異なるのかを指す．

$0 < W < 1$ に正規化されている依存関係の強度 W に関して，G_{body} の WPDG 上の各矢印の状態はしきい値 $\theta = 0.5$ を境界として，結合と分断に分かれる．よって，部品内部に含まれる弱結合の矢印に対して，結合するために補う強度は $\theta - W = 0.5 - W$ である．また，部品内外部に分断されている強結合の矢印に対して，分断するために奪う強度は $W - \theta = W - 0.5$ である．本洗練手法では，弱結合に対して補う強度および強結合に対して奪う強度を状態を維持するために必要なコストとみなし，それぞれの強度の二乗誤差の和で維持コストを定義する．C_{body} あるいは C_{refine} の WPDG を G とすると，維持コスト R_p は次のようになる．

$$
R_p(G) = \sum_{p \stackrel{W}{\rightarrow} q \in E1(G)} (0.5 - W)^2 + \sum_{p \stackrel{W}{\rightarrow} q \in E2(G)} (W - 0.5)^2
$$

$$
\begin{aligned}
E1(G) &= \{p \stackrel{W}{\rightarrow} q \in Edge(G_{whole}) \mid BothIn(G, p, q) \wedge W < 0.5\} \\
E2(G) &= \{p \stackrel{W}{\rightarrow} q \in Edge(G_{whole}) \mid EitherIn(G, p, q) \wedge W > 0.5\}
\end{aligned}
$$

$E1(G)$ は部品内部に含まれる弱結合の矢印の集合，$E2(G)$ は部品内外部に分断されている強結合の矢印の集合を指す．

修正コストとは，C_{whole} において，依存関係矢印の状態を結合および分断する際に必要なコストである．このコストは，各矢印の現在の強度が洗練による結合および分断をいう修正をどの程度妨げるのかを指す．

$0 < W < 1$ に正規化されている依存強度 W に関して，結合している C_{body} 内部の矢印の強度は 1 (強度最大)，分断している C_{body} 内部の矢印の強度は 0 (強度最小) であることが望ましい．よって，強度 W を持つ依存関係に対して，$|1 - W|$ は結合状態への変化を妨げる強度，$|W - 0|$ は分断状態への変化を妨げる強度である．本洗練手法では，洗練時に結合状態に変化する矢印に対して結合を妨げる強度および洗練時に分断状態に変化する矢印に対して分断を妨げる強度を修正に必要なコストとみなし，それぞれの強度の二乗誤差の和で修正コストを定義する．C_{body} の WPDG を G，C_{refine} の WPDG を G' とすると，修正コスト R_m は次のようになる．

$$R_m(G, G') = \sum_{p \xrightarrow{W} q \in E1(G,G')} (1 - W)^2 + \sum_{p \xrightarrow{W} q \in E2(G,G')} (W - 0)^2$$

$$\begin{aligned} E1(G, G') &= \{p \xrightarrow{W} q \in Edge(G_{whole}) \mid EitherIn(G, p, q) \wedge BothIn(G', p, q)\} \\ E2(G, G') &= \{p \xrightarrow{W} q \in Edge(G_{whole}) \mid BothIn(G, p, q) \wedge EitherIn(G', p, q)\} \end{aligned}$$

$E1(G, G')$ は洗練時に分断状態から結合状態に変化する矢印の集合，$E2(G, G')$ は洗練時に結合状態から分断状態に変化する矢印の集合を指す．

4.4.3 洗練条件

部品洗練の妥当性を考えたとき，重みが変動するごとに依存関係矢印の結合あるいは分断を行うと，ライブラリ内の部品の形は頻繁に変化することになり，部品検索や部品の機能に対する理解が困難になる．そこで，同様の部品変更が繰り返し行われる場合にのみ洗練が行われるように，維持コストと修正コストに基づく洗練条件を導入する．いま，洗練前の C_{body} が現在の結合状態を維持するのに必要な維持コスト $R_p(G_{body})$ が，洗練後の G_{refine} が結合状態を維持するのに必要な維持コスト $R_p(G_{refine})$ より大きいとき，部品の維持コストが減少するので洗練を行う方がよい．しかし，実際に洗練を行う際には，依存関係矢印の状態を変化させるための修正コスト $R_m(G_{body}, G_{refine})$ が必要である．よって，洗練後の維持

コストが洗練前より小さく，依存関係矢印の状態を変化させることが可能であるためには，以下の洗練条件を満たす必要がある．

$$R_p(G_{body}) - R_p(G_{refine}) > R_m(G_{body}, G_{refine})$$

本洗練手法では，上記の洗練条件が成立する際に洗練が妥当であると判定する．また，複数の候補から洗練後の部品本体を選択するために，部品洗練時の余り R を次のように定義し，R の値が大きいほど開発者の再利用ドメインの特性に適しているとみなす．

$$R = R_p(G_{body}) - R_p(G_{refine}) - R_m(G_{body}, G_{refine})$$

4.4.4 洗練部品の作成手続き

本洗練手法では，次に示す手順で C_{whole} を洗練し，洗練後の部品本体 C'_{body} に含まれる文を決定する．

Step 1. C_{whole} の WPDG の各依存関係矢印の重みに基づき強依存部分グラフ G_{sub} を抽出し，強依存節点集合 N_{sub} を求める． N_{sub} から G'_{body} の候補 G_{refine} を求める． G_{refine} が 1 つしか存在しないとき， Step 5 へ．

Step 2. 洗練による急激な形の変化を避けるため，洗練前の G_{body} と洗練後の候補 G_{refine} において，共有する節点の数が多いものを選択する．つまり，集合 $S = Node(G_{body}) \cap Node(G_{refine})$ に対して，要素数 $\#S$ が最大となる G_{refine} を G'_{body} の候補として選択する．選択した G_{refine} が 1 つしか存在しないとき， Step 5 へ．

Step 3. 余り R が最大となる G_{refine} を G'_{body} の候補として選択する．選択した G_{refine} が 1 つしか存在しないとき， Step 5 へ．

Step 4. 部品ライブラリの管理者，あるいは，開発者が複数の G_{refine} の中から 1 つを選択する．

Step 5. 洗練条件の真偽を判定する．選択した G_{refine} が洗練条件を満たす場合， G'_{body} に対応する C'_{body} を G_{refine} に対応する C_{refine} に置き換える．洗練条件を満たさない場合，現在の C_{body} をそのまま C'_{body} とする．

Step 1 において抽出した図 4.7に示す 3 つの N_{sub} を洗練の例として取り上げる． Step 1 において， N_{sub} から作成した 3 つの G_{refine} に対して，それぞれもとの部品本体との共有

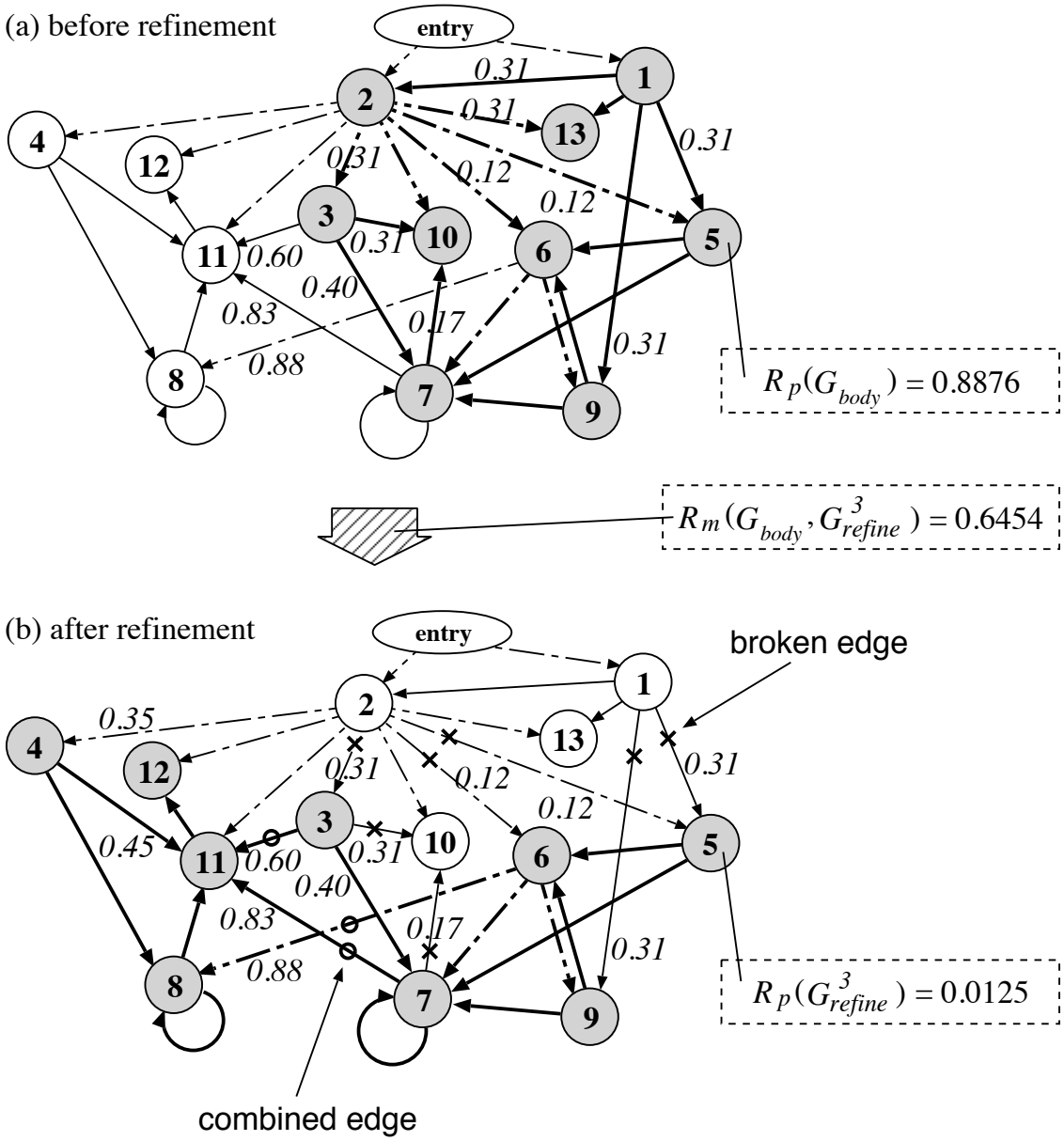


図 4.8: 洗練前後の部品と維持および修正コスト

節点の数は，$\#S^1_{refine} = 2$, $\#S^2_{refine} = 2$, $\#S^3_{refine} = 5$ となる．よって，Step 2 により，G^3_{refine} が G'_{body} の候補として選択される．G^3_{refine} を用いて洗練を適用する様子を図 4.8に

示す．図 4.8b において，○印は洗練により結合される矢印，×印は洗練により分断される矢印を指す．洗練前部品の維持コスト $R_p(G_{body})$，洗練候補の維持コスト $R_p(G^3_{refine})$，修正コスト $R_m(G_{body}, G^3_{refine})$ は，それぞれ図 4.8に示すようになる．よって，Step 5 において洗練条件は次のように成立し，

$$0.8876 - 0.0125 = 0.8751 > 0.6454$$

洗練対象部品 C_{whole} における C_{body} が C^3_{refine} に置換され，洗練後の C'_{body} となる．

図 4.8の例では，図 4.1の C_{whole} において，無変更で頻繁に再利用された，「入力データの合計を求める機能」を担う文 3, 5, 6, 7, 9 と，「入力データの個数を求める機能」を担う文 4, 8，「入力データの平均を求め表示する」機能を担う文 11, 12 が洗練後の部品本体に含まれる．また，部品変更時に頻繁に削除された，「入力データのファイルに関する処理」を担う文 1, 2, 13，および，「合計を表示する機能」を担う文 10 が部品本体から取り除かれる．

4.5 評価実験および考察

本洗練手法の有効性を確認するために，本手法に基づいて部品洗練システムを構築し，部品洗練実験を行った．部品洗練システムによる洗練の様子を図 4.9に示す．開発者は，このシステムの提供するエディタを通してライブラリから部品を取り出し，そのまま，あるいは，適時変更して再利用する．本節では，洗練の特性と洗練の効果という 2 つの観点から行った実験結果を示し，その実験結果について考察する．

4.5.1 部品洗練手法の特性

洗練後の部品がどのような形になるのか，および，洗練がどのような場合に行われるのかを確認するために，実際に部品を洗練した際の実験結果を用いて，本洗練手法における洗練の特性について考察する．

(1) 同様の変更を適用した際の結果 (実験 1)

過去の変更に応じて部品が洗練される様子を確認するため，図 4.1に示す部品を用いて，部品洗練に関する実験を行った．本実験では，図 4.1に示す部品から表 4.2に示す 7 つのソースコード断片を抽出し，これらの断片を組み合わせたものを変更後部品 C_{reuse} として用いた．洗練に関するパラメータの値は，$\delta = 1.0, T = 5.0$ とした．

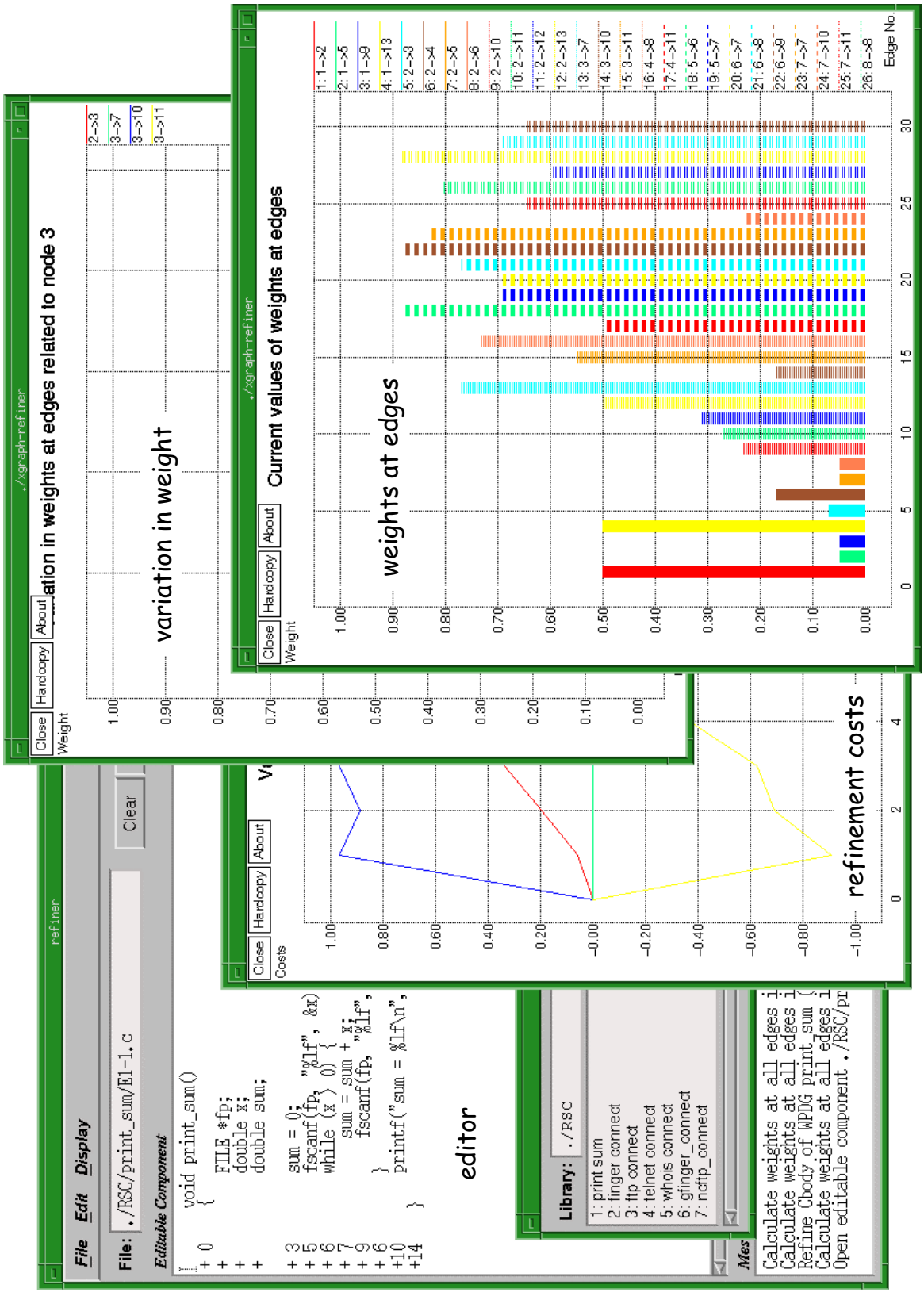


図 4.9: 部品洗練システム

表 4.2: 洗練対象部品から抽出したソースコード断片の機能と節点

断片	ソースコード断片の機能	節点
f1	ファイルのオープンとクローズ	1 2 13
f2	入力データを読み込む	5 6 9
f3	入力データの合計を表示する	3 5 6 7 9 10
f4	入力データの合計を求める	3 5 6 7 9
f5	入力データの個数を数える	3 5 6 8 9
f6	入力データの平均を表示する	3 4 5 6 7 8 9 11 12
f7	入力データの平均を求める	3 4 5 6 7 8 9 11

表 4.3: 適用した部品変更と洗練後の部品本体 C_{body}(実験 1)

変更	C_{reuse}/C_{body}	付加節点	削除節点
初期	f1+f2+f3	1 2 3 5 6 7 9 10 13	
M–1	f2+f3	なし	1 2 13
M–2	f2+f4	なし	1 2 10 13
M–3	f2+f5	4 8	1 2 3 7 10 13
M–4	f2+f3+f5+f6	4 8 11 12	1 2 13
M–5	f2+f3+f5+f7	4 8 11	1 2 13
洗練 A	f2+f3+f5+f6	3 4 5 6 7 8 9 10 11 12	
M–6	f2+f4	なし	8 10 11 12
M–7	f2+f5	なし	7 10 11 12
M–8	f2+f4+f5+f6	なし	10
M–9	f2+f4+f5+f7	なし	10
洗練 B	f2+f4+f5+f6	3 4 5 6 7 8 9 11 12	
M–10	f2+f4+f5+f6	なし	なし
最終	f2+f4b+f5+f6	3 4 5 6 7 8 9 11 12	

本実験において用いた C_{reuse} と，洗練後の C_{body} に含まれる節点を表 4.3に示す．表 4.3の C_{reuse}/C_{body} の欄において，記号 + は断片の組合せ (断片に含まれる節点の和集合) を表す．また，付加節点および削除節点の欄の番号は，図 4.1における節点識別番号を指す．初期，最終，洗練後の欄については，付加節点や削除節点ではなく，C_{body} 内部に含まれる節点の識別番号をそのまま記述した．M–1 ～ M–5 では，主に「ファイルポインタを入力引数とする (断片 f1 を削除する)」変更と，「入力データの個数を数える機能を追加する (断片 f5 を

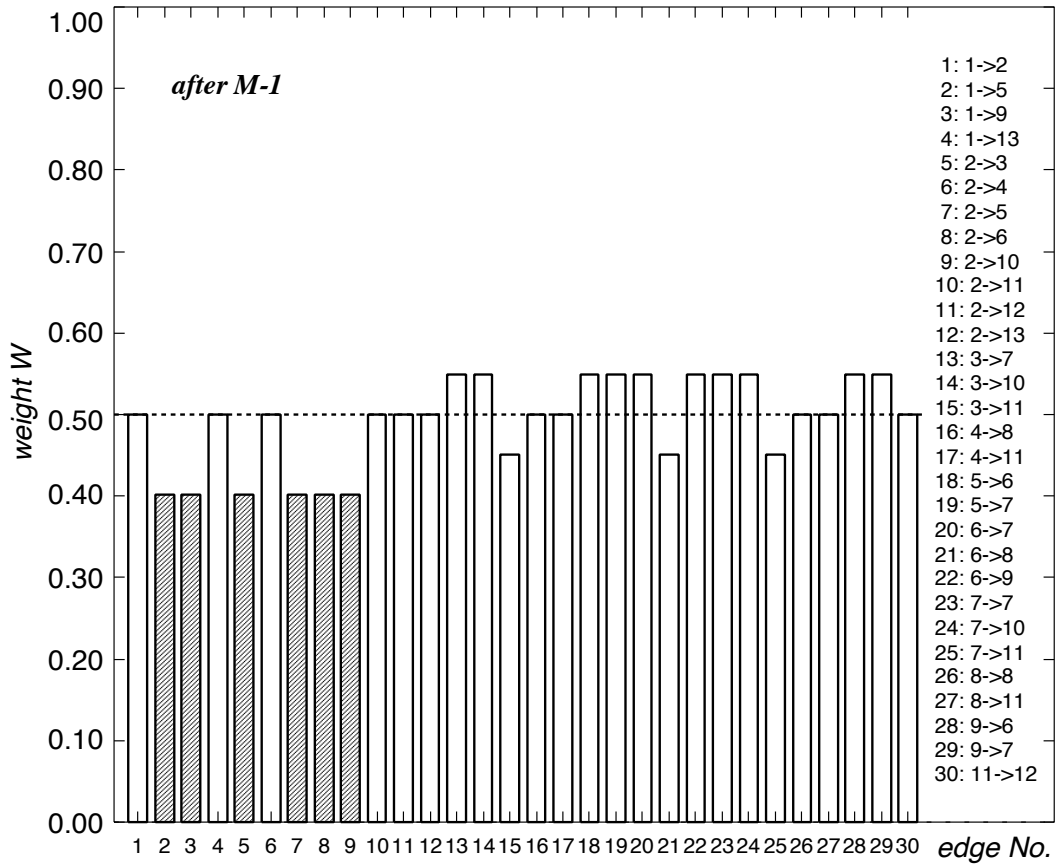


図 4.10: 依存関係矢印の重みの値 (変更 M–1 の直後)

付加する)」変更を適用した．また， M–6 ∼ M–10 では，主に「入力データの合計を求め，その値は表示しない (節点 10 を削除する)」形で部品を再利用した．以下， M–1 ∼ M–5 を変更パターン A， M–6 ∼ M–10 を変更パターン B と呼ぶ．

本実験では， M–5 と M–9 の直後に洗練が 2 回行われる．まず，洗練後の部品の形について考察する．表 4.3において，洗練 A 直後の C_{body} を C_A，洗練 B 直後の C_{body} を C_B とおく．また，初期状態の部品本体を $C_{initial}$ とおく．

表 4.3において， $C_{initial}$ と C_A に含まれる節点を比較すると，洗練 A により，断片 f4+f6 の節点 4, 8, 11, 12 が付加 (C_A の節点集合 $\setminus$ $C_{initial}$ の節点集合， $\setminus$ は集合の差を表す)，断片 f1 の節点 1, 2, 13 が削除 ($C_{initial}$ の節点集合 $\setminus$ C_A の節点集合) されていることが分かる．この洗練結果は，変更パターン A において，半分以上の割合で節点 4, 8 を付加した，および，すべての変更で節点 1, 2, 13 を削除したという変更操作に一致する．すなわち，洗

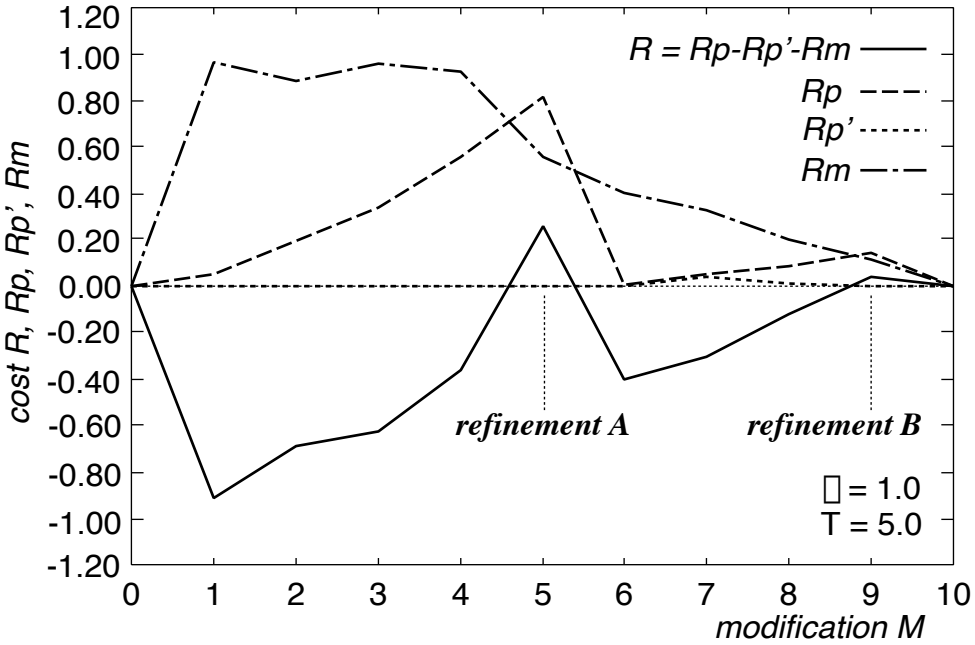


図 4.11: 洗練コスト R_p, R'_p, R_m と余り R の変動

練 A 前後の C_{body} の変化は，変更パターン A を反映して行われたと考えて良い．また，表 4.3において， C_A と C_B に含まれる節点を比較すると，洗練 B により，節点 10 が削除 (C_B の節点集合 $\setminus$ C_A の節点集合) されていることが分かる．洗練対象部品において，節点 10 は入力データの合計値を表示する文であり，この節点が削除されることは合計値の表示が不要，つまり，合計値は平均値を求めるための一時変数として用いられていることを指す．この洗練結果は，変更パターン B において，すべての変更で節点 10 を削除したという変更操作に一致する．すなわち，洗練 B 前後の C_{body} の変化は，変更パターン 2 を反映して行われたと考えて良い．

次に，部品洗練が行われる時期について考察する． M–1 の直後の部品 C_{whole} における各依存関係矢印の重みの値を図 4.10に示す．また， M–1 $\sim$ M–10 を適用した際の，洗練前部品の維持コスト R_p，洗練候補の維持コスト R'_p，修正コスト R_m，余り $R = Rp - R'_p - R_m$ の変動を図 4.11に示す．

本洗練手法では， 4.4.3節で定義したコストに基づく洗練条件を導入したことにより，ある節点に関する依存関係矢印の重みがしきい値 0.5 を越えた，あるいは，しきい値 0.5 を下まわったからといって，すぐに洗練が行われるわけではない．例えば，図 4.10を見ると，

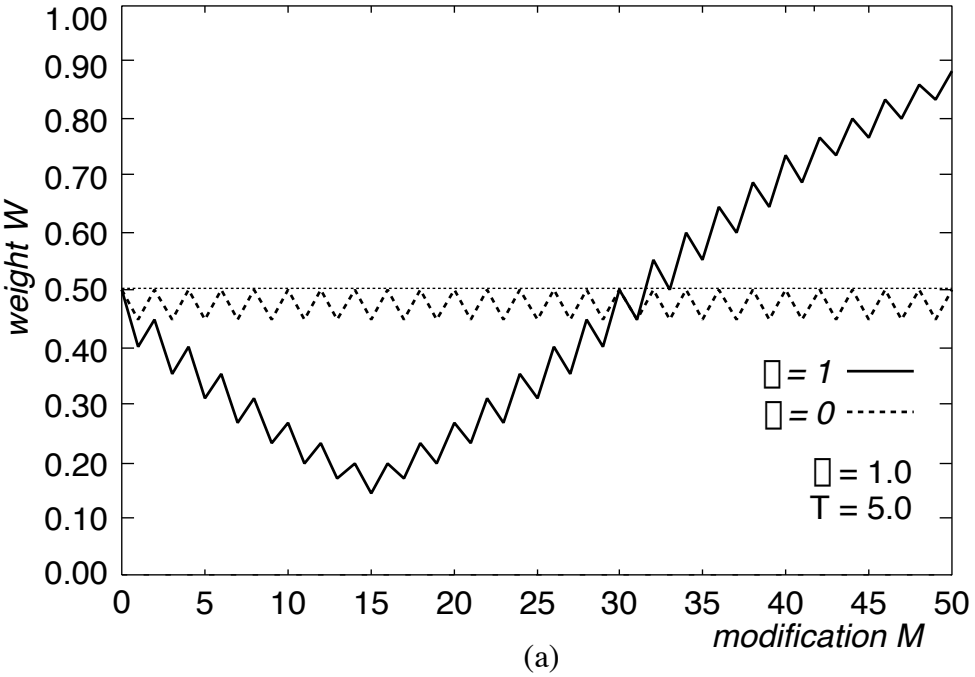
M–1 の適用時の削除節点 1, 2, 13 に関する矢印のうち，矢印 2, 3, 5, 7, 8, 9 の重みがしきい値 0.5 より小さい (図 4.10の網掛け部分)．よって，単純に現在の重みに基づいて部品を洗練すると， M–1 の直後に節点 1, 2, 13 が C_{body} から取り除かれるはずである．しかし，図 4.11を見ると， M–1 の直後において R の値は負であり，洗練条件が成立していない．このため，実際には M–1 の直後で洗練は行われない．このように，洗練条件によって洗練の妥当性を判定することで，同様の変更が繰り返し行われた場合にのみ洗練を行うことができ，瞬間的に行われる部品変更に対して C_{body} の形を緩やかに変化させることが可能である．また，図 4.11の M–1 $\sim$ M–5 と M–6 $\sim$ M–9 において，それぞれ R の勾配を見ると，洗練が行われるまで R は洗練条件が成立する方向 (正の値の方向) に単調に増加していることが確認できる．この結果は，本実験において，適用した部品変更操作がそれぞれの変更パターン A あるいは B 内で類似していたことを裏付けており，洗練後の部品本体の形が各変更パターンにおいて一意に収束していることを指す．

以上より，適用した部品変更と洗練に関して次のことがいえる．洗練条件を導入した本洗練手法では，同様の部品変更が繰り返し適用された場合に，その部品変更操作を反映して，緩やかに部品洗練を行うことができる．

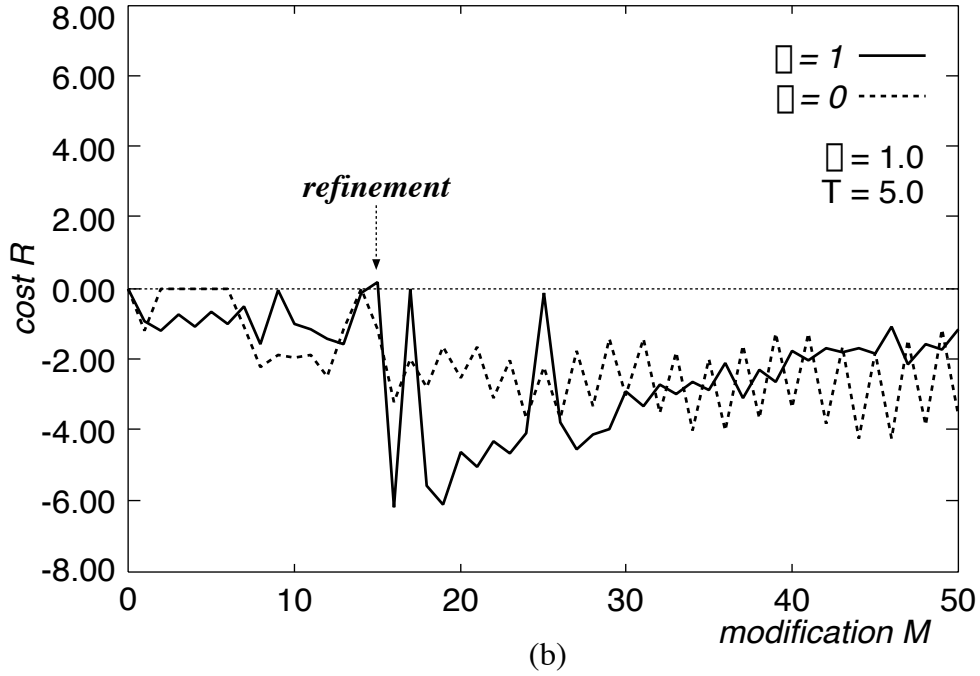
(2) 相反する変更を適用した際の結果 (実験 2)

本実験では，依存節点の接続関係の変化に基づく重み w^c を導入した際の効果の 1 つとして，相反する変更を繰り返し適用した場合でも部品洗練が行われることを確認する．このために， w^c を導入する場合 ($\beta = 1$) としない場合 ($\beta = 0$) について，同じ部品を洗練する実験を行った．表 4.3に示す M–1 $\sim$ M–5 に対して，断片 f1 (節点 1, 2, 13) を付加する変更 M–1$'$ $\sim$ M–5$'$ を適用した部品を用意する．これら相反する変更を含む 10 個の部品を M–1, M–1$'$, M–2, M–2$'$,$\cdots$, M–5, M–5$'$ の順序で 5 回繰り返し，全部で 50 回の重み変動を行った．洗練に関するパラメータの値は，実験 1 と同じ $\delta = 1.0, T = 5.0$ とした．

変更により依存関係が結合と分断を繰り返す矢印 7 (節点 2 と節点 5 を結合) の重みと，余り R の変動の様子を図 4.12に示す．図 4.12a を見ると，相反する変更を含む部品を適用することで，矢印 7 の重みが 1 回の重み計算ごとに増加と減少を繰り返していることが分かる． $\beta = 1$ のとき， 1 回ごとの振動の他に大きな周期が見られる．これは，重み w^c の導入により依存関係矢印の強度の変動に偏りが存在することに一致する．このように，相反する変更を適用した場合でも重みを少しずつ増加あるいは減少させることで，部品洗練が行われる可能性は高くなる．このことは，図 4.12b において， $\beta = 1$ の場合， M–15 で洗練が行



(a)



(b)

図 4.12: 矢印の重みと余り R の変動

われたことから確認できる．

以上より，重み w^c が洗練に与える影響に関して次のことがいえる．本洗練手法では，相反する変更を繰り返し適用した場合でも，重みの変動に一定方向の偏りが生じ，洗練が行われる可能性は高い．

(3) 重み付けパラメータの洗練への影響 (実験 3)

4.3.5節で定義した重み変動式および正規化関数の式に注目する． n 回の部品変更において，重みが増加した回数を n_i，減少した回数を n_d とすると， $w_0 = 0$ より，

$$w_n = w_0 + n_i\,\delta - n_d\,\delta = (n_i - n_d)\,\delta$$

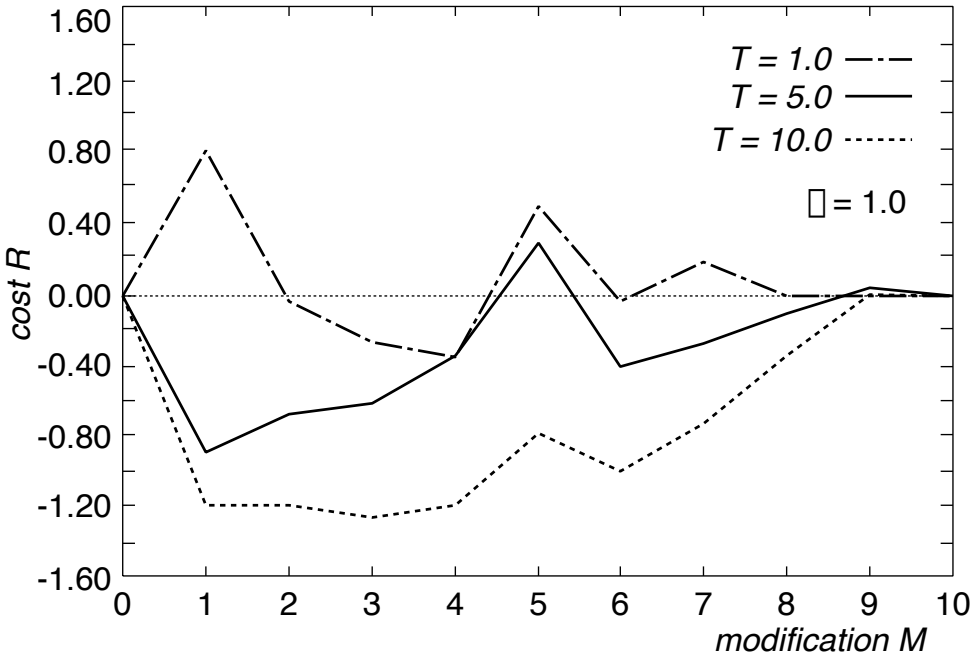
となる．よって，各依存関係矢印に割り当てられる正規化後の重み W は，

$$W = N(w) = \frac{1}{1 + e^{(-(n_i - n_d)\delta/T)}}$$

となる．上記の式より， W の変動はパラメータ δ, T 単独ではなく，これらの比 δ/T に影響を受けることが分かる．さらに，洗練時の余り R の計算式において， R は W^2 の加減算で定義されている．よって，洗練条件が成立する頻度や時期は δ/T に影響を受ける．そこで，重み付けの計算式におけるパラメータが洗練にどのような影響を与えるのかを調べるために， $\delta = 1.0$ と固定し， T の値を変えて，洗練実験を行った．

$T = 1.0, 5.0, 10.0$ において，表 4.3に示す変更 M–1 ～ M–10 を適用した際の，余り R の変動と洗練後の C_{body} に含まれる節点を図 4.13に示す．変更欄の数字は，洗練が行われるまでの変更の回数を指す． $T = 5.0$ のときの R と C_{body} は，実験 1 のときと同じである．

正規化関数において， T を小さくすると， $w = 0$ 付近での傾きが急になり， 1 回の重み変動操作に対して正規化後の重みの変動が大きくなる．よって， 1 回の重み変動で洗練条件が満たされる可能性が高くなり，洗練が頻繁に行われると推測できる．逆に， T の値を大きくすると， $w = 0$ 付近での傾きが緩やかになる．よって，洗練が行われるために多くの部品変更が必要となり，洗練の頻度は減少すると推測できる．これらの推測が正しいことは，図 4.13において，余り R の変動の形がほぼ等しいにもかかわらず，洗練の回数が $T = 1.0$ のとき 3 回， $T = 5.0$ のとき 2 回， $T = 10.0$ のとき 1 回と， T が大きくなるほど洗練の回数が減少していることから確認できる．特に， $T = 1.0$ のとき，実験 1 の変更パターン A の 1 回目直後，および，変更パターン B の 2 回目 (全体では 7 回目) 直後という早い段階で洗練が行われている．逆に， $T = 10.0$ においては，変更パターン B の 4 回目 (全



変更	節点集合 ($T = 1.0$)	節点集合 ($T = 10.0$)
初期	1 2 3 5 6 7 9 10 13	1 2 3 5 6 7 9 10 13
1	3 5 6 7 9 10	–
5	3 4 5 6 7 8 9 10 11 12	–
7	3 4 5 6 7 8 9 11 12	–
9	–	3 4 5 6 7 8 9 11 12
最終	3 4 5 6 7 8 9 11 12	3 4 5 6 7 8 9 11 12

図 4.13: $T = 1.0, 5.0, 10.0$ の際の余り R の変動と洗練後部品

体では 9 回目) 直後という遅い段階で洗練が行われている．さらに，$T = 10.0$ のとき，変更パターン A において，変更を繰り返し行ったにもかかわらず洗練が行われていない．

本実験だけから，δ/T の有効な値を導くことはできないが，重み付けパラメータ δ/T を設定する際の指針として，次のことがいえる．洗練条件を導入した本手法では，洗練の行われる頻度や時期を重み付けパラメータ δ/T の値によって調整可能である．δ/T を小さくすると，洗練の頻度は増加，かつ，洗練の時期は早くなる．逆に，δ/T を大きくすると，洗練の頻度は減少，かつ，洗練の時期は遅くなる．重み付けパラメータによる影響を明確にするためには，さまざまな部品変更パターンを用意し，実験と検証を数多く行う必要がある．

表 4.4: 洗練前部品と洗練後部品の節点および矢印の数

部品	FINGER	FINGER′	FTP	FTP′
節点数	21	20	22	16
矢印数	48	42	51	32
部品	TELNET	TELNET′	WHOIS	WHOIS′
節点数	37	19	26	22
矢印数	83	34	62	51

4.5.2 部品洗練の効果 (実験 4)

部品洗練の効果を確認するために，実在するソースコードから部品を抽出し，本手法に基づき洗練を行う場合と行わない場合における再利用時の変更操作数を測定した．洗練対象として，UNIX[1]の finger, ftp, telnet, whois のプログラムから，3章で述べた部品作成手法により抽出した「TCP ソケットのコネクションを確立する」という共通の機能を持つ部品 FINGER, FTP, TELNET, WHOIS を用意した．本実験では，これら 4 つの部品から 1 つを選択し，残りの 3 つの部品を変更後の部品とみなして，1 つの部品に対して 90 回 (3 部品 × 30 回) の変更を適用した．例えば，部品 FINGER を選択した場合，残りの FTP, TELNET, WHOIS を FINGER の変更後部品とみなし，FINGER → FTP, FINGER → TELNET, FINGER → WHOIS という変更を順番に合計 90 回適用した．このとき，洗練後部品の形が変更の順序から受ける影響を少なくするため，1 回の重みの変動幅を小さく設定し ($\delta = 0.1, T = 5.0$)，数多くの部品変更を行うことで洗練が行われるようにした．洗練は各部品とも 1 回づつ行われた．洗練前と洗練後の部品本体の節点数と矢印数を表 4.4に示す．FINGER′, FTP′, TELNET′, WHOIS′ は洗練後部品を指す．

洗練前および洗練後の各部品を再利用することで，もとの 4 つのプログラム finger, ftp, telnet, whois を作成するのに必要な変更操作数を表 4.5に示す．表中の A, D, I は，4.2.3節の部品変更における記号を指し，A は節点の付加，D は節点の削除，I は新規節点の追加を表す．例えば，洗練前の FINGER を再利用して ftp を作成するためには，FTP/finger の欄の A0 D6 I6 より，節点を 0 個付加，6 個削除，6 個追加の全部で 12 操作が必要である．節点の移動 M および節点におけるラベルの書き換え R は，節点の削除 D と新規節点の追加

[1]UNIX は X/Open カンパニリミテッドがライセンスしている米国ならびに他の国における登録商標である．

表 4.5: 洗練前部品と洗練後部品に対する変更操作の数 (finger, ftp, telnet, whois)

プログラム \ 部品	FINGER	FINGER′	FTP	FTP′
finger	– – –	A1 D0 I0	A0 D9 I6	A0 D3 I6
ftp	A0 D6 I6	A0 D6 I7	– – –	A6 D0 I0
telnet	A0 D4 I18	A0 D2 I17	A0 D8 I21	A0 D3 I22
whois	A0 D0 I5	A1 D0 I5	A0 D8 I10	A0 D1 I1
合計	39	39 (100%)	62	42 (68%)
減少率	–	0%	–	32%

プログラム \ 部品	TELNET	TELNET′	WHOIS	WHOIS′
finger	A0 D19 I1	A1 D3 I2	A0 D5 I0	A0 D1 I0
ftp	A0 D24 I6	A2 D4 I5	A0 D10 I5	A0 D6 I4
telnet	– – –	A18 D0 I0	A0 D6 I16	A0 D2 I16
whois	A0 D19 I6	A1 D2 I6	– – –	A4 D0 I0
合計	75	44 (59%)	42	33 (79%)
減少率	–	41%	–	21%

表 4.6: 洗練前部品と洗練後部品に対する変更操作の数 (GNU finger, ncftp)

プログラム \ 部品	FINGER	FINGER′	FTP	FTP′
GNU finger	A0 D12 I10	A0 D10 I10	A0 D13 I11	A0 D7 I12
合計	22	20 (91%)	24	19 (79%)
減少率	–	9%	–	21%
ncftp	A0 D10 I14	A0 D8 I14	A0 D7 I10	A0 D3 I13
合計	24	22 (92%)	17	16 (94%)
減少率	–	8%	–	6%

プログラム \ 部品	TELNET	TELNET′	WHOIS	WHOIS′
GNU finger	A0 D25 I7	A5 D9 I5	A0 D15 I9	A0 D11 I10
合計	32	19 (59%)	24	21 (88%)
減少率	–	41%	–	12%
ncftp	A0 D28 I12	A0 D22 I13	A0 D14 I13	A0 D10 I13
合計	40	35 (88%)	27	23 (85%)
減少率	–	12%	–	15%

I に分けて数えた．

もとのプログラム finger, ftp, telnet, whois を作成する際，洗練前部品を再利用した場合の変更操作数の合計と洗練後部品を再利用した場合の変更操作数の合計を比較する．FINGER と FINGER′ の変更操作の合計数はどちらも 39 回で，本実験から洗練の効果は確認できない．FTP, TELNET, WHOIS に関しては，洗練前より洗練後の変更操作の合計数が，それぞれ 32%, 41%, 21% 減少している．

さらに，本実験では，「TCP ソケットのコネクションを確立する」機能を持つプログラムとして GNU finger[2] と ncftp[3] を用意した．これらのプログラムは，部品 FINGER, FTP, TELNET, WHOIS の洗練に関わっていない．洗練前および洗練後の各部品を再利用することで，GNU finger, ncftp を作成するのに必要な変更操作数を表 4.6に示す．これらのプログラムに対しても，洗練前部品と洗練後部品を再利用した場合を比較すると，どの部品に対しても変更操作数が減少している (6%〜41%) ことが確認できる．

表 4.5, 4.6に示す実験 4 の結果から，実在するプログラムに対しても，部品洗練により変更操作数が減少することが確認でき，本洗練手法が有効であることが示された．特に，部品の規模 (節点数と矢印数) が他の部品より大きい TELNET に関しては，洗練前後の変更操作数の減少率が比較的大きい．本実験の一部の結果において，洗練前部品に対する洗練後部品の変更操作数の減少率がそれほど大きくならなかった原因として，部品および部品を再利用して作成したソースコードの規模が小さかったことと，洗練に用いた変更パターンの種類が少なかったことが考えられる．

4.6 拡張

本節では，本洗練手法の限界について考察し，本手法を拡張する方法について述べる．

4.6.1 洗練後部品の実行可能性

部品を再利用する際，そのままの形で組み込み可能であることや動作確認ができることより，ライブラリ内の部品は実行可能であることが望ましい．しかしながら，本洗練手法では，実際に依存関係が存在するにもかかわらず弱結合の依存関係を切断しているとみなすので，C_{body} に含まれる文だけを集めても実行可能な部品にはならないことがある．これは，弱結合の矢印を単純に切断すると，C_{whole} 内部に存在しなかった新しいデータ依存関

[2]Copyright 1990, 1991, 1992 Free Software Foundation, Inc.
[3]Copyright 1995 Mike Gleason, NCEMRSoft.

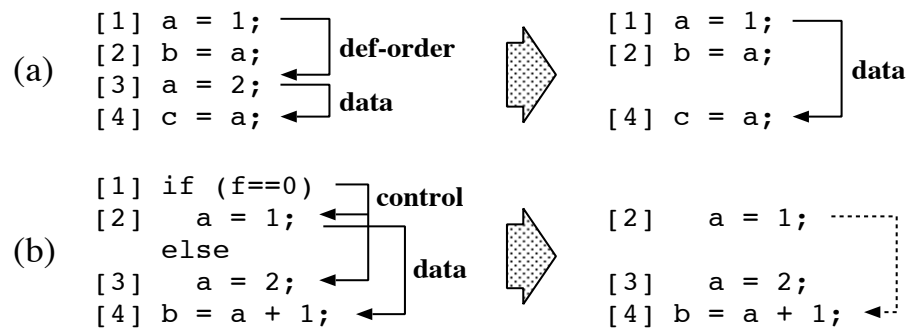


図 4.14: 洗練後部品における依存関係の変化

係が生じることや，本来含まれるはずのデータ依存関係が消滅するからである．例えば，図 4.14a のように，節点 1 と節点 3 に定義順序依存関係 (def-order dependence) が存在する場合，節点 3 の代入文を単純に削除すると，節点 1 と節点 4 に新しいデータ依存関係が生じる．また，図 4.14b のように，節点 1 と節点 2, 3 にそれぞれ真と偽の制御依存関係，さらに節点 2, 3 と節点 4 にデータ依存関係が存在する場合，節点 1 の条件文を単純に削除すると，節点 2 と節点 4 のデータ依存関係が消滅する．

この問題は，洗練後の C_{body} に含まれない節点を無条件に部品から削除するのではなく，削除する節点のラベル (節点に対応する文) を開発者に問い合わせを行う文に置換することで解決できる．問い合わせ文とは，削除対象節点での代入文において定義される変数の値，あるいは，削除対象節点での条件分岐において制御の移る方向を，実行時に開発者が入力する文を指す．

4.6.2 部品の拡大

本洗練手法では，あらかじめ用意した部品 C_{whole} から C_{body} に含まれる文を選択することで洗練を達成している．よって，C_{whole} に含まれない機能を持つ C_{body} が洗練により生成されることはない．そこで，C_{body} だけでなく C_{whole} も変更するような仕組みを，本洗練手法に導入する．開発者が変更した部品 C_{reuse} の PDG が C_{whole} の PDG を完全に包含するとき，つまり，C_{whole} の PDG の節点および矢印がすべて C_{reuse} の PDG に存在するとき，C_{whole} を C_{reuse} に置換する．この場合，PDG の包含関係より，置換前の C_{whole} に含まれるすべての機能が C_{reuse} から抽出可能なので，置換前の C_{whole} における C_{body} はそのままで良く，洗練後の C_{body} の種類を増加できる．

4.6.3 部品の分裂

本洗練手法は，1 つの部品 C_{whole} 内部に含まれる 2 つ以上の機能を独立に扱うことはできない．このため，機能の関連は少ないが共通な文を含む部品変更を繰り返した場合，過去に再利用したすべての文を含むソースコードが洗練後の C_{body} となり，適切な洗練が行われない可能性がある．この問題に対して，C_{whole} を頻繁に再利用する 2 つの部分に分裂させ，それぞれの分裂部品に対して独立に重み付けを行うことを考えている．これにより，1 つの C_{whole} から 2 つ以上の洗練後 C_{body} を生成することが可能である．このためには，部品変更のパターンを分類する必要がある．

4.7 関連研究

本洗練手法は，もとの部品 C_{whole} のソースコードから，開発者の再利用ドメインに適した機能を持つ一部のソースコードを部品本体 C_{body} として特定することで部品洗練を達成している．よって，プログラム全体を C_{whole} とすると，既存プログラムから再利用ドメインに適した部品を自動抽出することが 1 回の部品洗練とみなせる．このような手法として，プログラムスライシング [111] を用い，データの入出力に着目して部品を抽出する手法 [84] や，プログラム制御の条件分岐に着目して部品を抽出する手法 [24, 51] が提案されている．また，文献 [28] では，プログラム内部のデータの利用関係や制御の呼び出し関係の事実と推論により強結合の関数を特定し，それらの関数をまとめることで部品を作成する手法が述べられている．さらに，文献 [46] の手法では，もとのプログラムから複数のスライスを抽出し，抽出したスライスに含まれる文どうしの結合の強度を依存関係の存在する割合から算出し，部品に含まれる文を選択する．

このように，プログラムの構造や依存関係に基づき部品を洗練する手法は，従来から提案されている．また，ソフトウェアリストラクチャリング [7] におけるソースコード変換も洗練手法の一つであると捉えることができる．しかしながら，従来提案されている部品抽出および再構成手法は，プログラムの静的な解析だけによって達成されている．このため，部品のソースコードが与えられれば，洗練後の部品の形が一意に決まるが，開発者の再利用ドメインの特性に応じて洗練を行っているとはいえない．これに対して，本洗練手法は，過去の変更履歴を反映した依存関係の重みに基づき部品を洗練するため，同じプログラム構造や依存関係を持つ部品を，開発者の再利用ドメインにおける利用頻度に応じた形に洗練可能である．さらに，洗練の単位が関数やスライスではなく個々の文であるので，従来手法より粒度

の細かい機能変更が達成でき，開発者の部品変更の負担を軽減できる可能性が高い．

また，洗練の指標という観点に着目すると，部品が開発者の再利用ドメインにどの程度適しているのかを評価する手法 (メトリクス) が提案されている．例えば，文献 [23] では，再利用ドメインにおける標準的な部品から呼び出される頻度を指標として，注目する部品の有効性を見積る．このような手法に対して，本洗練手法では，開発ドメインとの適合度を，開発者が実際に再利用した部品の変更前後のソースコードから計算するため，ドメイン固有の標準的部品をあらかじめ設定しておく必要がない点で有利である．

4.8 あとがき

ソースコード再利用において，過去の部品変更の履歴に基づく重みを PDG の依存関係矢印に割り付け，重みを指標としてライブラリ内の部品を将来再利用される可能性が高い形に自動洗練する手法を提案した．さらに，この部品洗練手法に基づき構築した部品洗練システムを用いた洗練実験の結果より，実際のソフトウェア部品に対しても開発者が部品を変更する負担が減少することを示した．本洗練手法では，頻繁に再利用されたソースコード断片は開発者が将来も同じ形で再利用する可能性が高い部分，また，頻繁に変更を受けたソースコード断片は開発者が将来も同様の変更を行う可能性が高い部分であると仮定する．これらの仮定に基づき，重み付きプログラム依存グラフにおける 2 節点間の依存関係の強度を重みで定義し，開発者が変更後に再利用した部品の形および部品変更前後の部品の形の変化に基づき重みを変動させる．このようにして蓄積した依存関係矢印の重みを部品洗練の指標として用い，もとの部品ソースコードから強依存関係で結合されているソースコード断片を抜き出し，重みから計算した洗練コストにより洗練の妥当性を判定することで，開発者の再利用ドメインに適した洗練部品の形を決定する．本洗練手法を導入することで，部品作成者の負担を増加させずに，開発者が再利用時に部品を適時変更する負担を軽減できる．

現在，本洗練手法の対象をオブジェクト指向フレームワーク [113] にまで拡張し，開発者の変更履歴に基づく重み付き依存グラフを用いることで，フレームワークにおける固定部分 (フローズンスポット) と変更可能な可変部分 (ホットスポット) [93] を再構成する試みを行っている [70, 75]．オブジェクト指向フレームワークとは，特定のドメインにおいて再利用されることが予測されるクラスとクラス間の相互作用を内包する半構成アプリケーションである [44, 113]．開発ドメインの特性に適したフレームワークを提供することで，開発者は必要最小限のホットスポットだけを定義するだけで，要求するアプリケーションを作成可能となる．

第 5 章

変更の模倣による部品の能動的変化

5.1 はしがき

ソースコード部品化再利用においては，ライブラリ内の部品の機能は部品作成時に固定される．よって，要求や特性が頻繁に変化する開発ドメインでは，次に示す 2 つの問題が顕著になる． 1) さまざまな要求に応じてプログラムを作成するためには，部品変更が不可欠である．このとき，部品のどの部分を変更すれば良いのか，どのように変更すれば良いのかを判断するのは難しい． 2) プログラム開発ドメインの特性を明確にすることが難しく，必要なすべての部品を部品作成時にライブラリ内にあらかじめ用意することは不可能である．

これらの問題を解決するために，本研究では，部品利用者の要求や部品の存在する環境に応じて，部品検索時に自動的かつ動的に変化する能動的部品 (active component) を提唱し，その変化メカニズムの検討を進めてきた [60, 63, 64, 73, 74]．本部品変化メカニズムにおいて，機能とは，部品ソースコード内部に現れる変数の値を決定する計算であるとし，部品の変化単位として機能に着目する．本章では，上記の問題 (1), (2) に対して，機能の分割および合成という観点から，次に示す 2 種類の能動的変化を提案する．

(1) 部品を機能分割することで，機能の一部を抽出する機能分割変化.

(2) 他の部品の変更事例を取り込み，機能の一部を交換する機能交換変化.

能動的部品は，機能分割変化により，自分の持つ機能から要求に対して必要な部分だけを抽出する．また，機能交換変化により，ライブラリ内で頻繁に行われる部品変更事例を環境として取得し，他の部品が受けた機能変更を模倣する．このようにして，能動的部品は，部品利用者が再利用時に行う部品変更を自分自身で行う．よって，部品化再利用によるプログラム作成工程において，能動的部品を用いることで，利用者が能動的部品ライブラリに要求

を投入するだけで，要求や環境に応じたソースコード部品が自動的かつ動的に生成される．問題 (1) に対して，部品利用者が能動的部品自らが適切に変更を行った変化後の部品を用いることで，部品変更に対する負担を軽減できる．また，問題 (2) に対して，部品作成者がライブラリ内に多種多様な部品をあらかじめ用意しておく必要はなく，特性分析が困難な開発ドメインでも，環境に応じた部品を利用者に提供できる．

さらに，本章では，能動的部品の変化メカニズムを実現する具体的アルゴリズムを提供し，提案する部品変化が機械実行可能であることを示す．能動的部品の機能分割変化は，プログラムスライシング [111] による機能分割手法により実現する．機能交換変化は，機能分割手法とプログラム依存グラフ (PDG) [29] のラベル付き同形写像比較 [42] による等価機能の特定手法を組み合わせた，プログラム合成アルゴリズム [40] により実現する．本部品変化メカニズムでは，従来のアルゴリズムでは合成不可能なソースコードが合成できるように，提案する合成アルゴリズムに対して，次の 3 つの改良を加えた．1) 機能の変更箇所を特定する際に，3章で提案した区間限定スライス，および，もとのプログラムからスライスを取り除いた際に残る補スライスを用いる，2) 機能の交換箇所を特定する際に，PDG の各節点のラベルを抽象化する，3) 変更箇所と無変更箇所を合成する際に，PDG 上のすべての節点にプログラム制御が必ず到達するように制御依存関係矢印を付けかえる．

依存関係に基づきソースコードを合成する本部品変化アルゴリズムは，部品内の文を無条件に入れ換える手法と異なり，合成後の部品が合成前の機能の一部を持つことを保証する．また，従来の合成アルゴリズムを改良することで，能動的部品はライブラリに存在しない数多くの新規部品を生成可能で，部品利用者の要求を無変更で満たす可能性は高くなる．

本章の構成は次の通りである．5.2節で能動的部品の動作と 2 つの変化メカニズムの詳細を述べる．次に，5.3節で本部品変化メカニズムを従来手法と比較し，本変化メカニズムにおける改良点を述べる．5.4節では，部品変化メカニズムを実現するために必要な諸定義を示す．5.5節で部品変化の具体的アルゴリズムを示し，5.6節で変化例を紹介する．さらに，5.7節で能動的部品の効果および拡張について考察する．

5.2 能動的部品の変化メカニズム

本節では，能動的部品の構造と動作を示し，機能分割変化および機能交換変化を述べる．

5.2.1 能動的部品の構造と動作

5.1節で述べた 2 つの問題を解決するためには，再利用部品が次の性質を持てばよい．

性質 1 部品利用者の要求を満たす部品がライブラリ内に存在しないときでも，部品利用者の要求を満たすソースコード (形を固定した部品) に変化する．

性質 2 特性分析が行われていない開発ドメインにおいても，自分の存在する環境に合わせたソースコードに変化する．

性質 1, 2 を満たすアプローチとして，要求に対して必要な部分だけを自分自身の機能から抽出したり (機能分割変化)，過去の変更事例を環境として取り込み，他の部品が受けた変更を模倣する (機能交換変化) メカニズムを能動的部品に持たせる．

機能分割変化および機能交換変化を達成するためには，能動的部品が従来の再利用ソースコード (code) に加えて，変化エンジン (engine) と変化規則 (change rule) を持つ必要がある．変化エンジンは， 1) 要求や環境を自ら取得するための感知機構， 2) 他の部品と交換する変更事例を蓄積するための記録機構， 3) 蓄積した変更事例を環境として他の部品に伝搬させるための広告機構， 4) 実際に部品の形を変更する生成機構で構成する．変化規則は，a) 変化パターンを決定する際の基準となる変化基準 (criteria)， b) どのように変化しても必ず部品内部に含まれる共通な機能 (スライス) を指す核 (core)， c) 過去に部品利用者が行った部品変更を差分で表現した変更事例 (cases) で構成する．変化基準と核は，能動的部品作成者が部品作成時に記述する．

図 5.1に能動的部品の構造および動作を示す．能動的部品のライブラリは，従来のライブラリと比較して，要求や環境を含む．要求とは，部品利用者が再利用する部品の機能を表すキーワードである．環境とは，ライブラリ内で自分の周りに存在する能動的部品と，それらの部品の機能や部品変更の事例を指す．能動的部品は次に示す 4 つの動作を行い，機能分割変化および機能交換変化を達成する．

(1) 能動的部品は，部品利用者が与えた要求や自分の存在するライブラリの環境を感知 (sense) する．感知とは，能動的部品がライブラリの状態をたえず監視し，ライブラリ内に存在する要求や環境 (変更事例，変更後ソースコード) を取得することを指す．

(2) 能動的部品は，自分の生成したソースコードが変更を受けたことを感知し，変化規則に変更事例を記録 (record) する．

(3) 能動的部品は，自分が変化する際に，自分の持つ変更事例をライブラリ全体に広告 (broadcast) する．

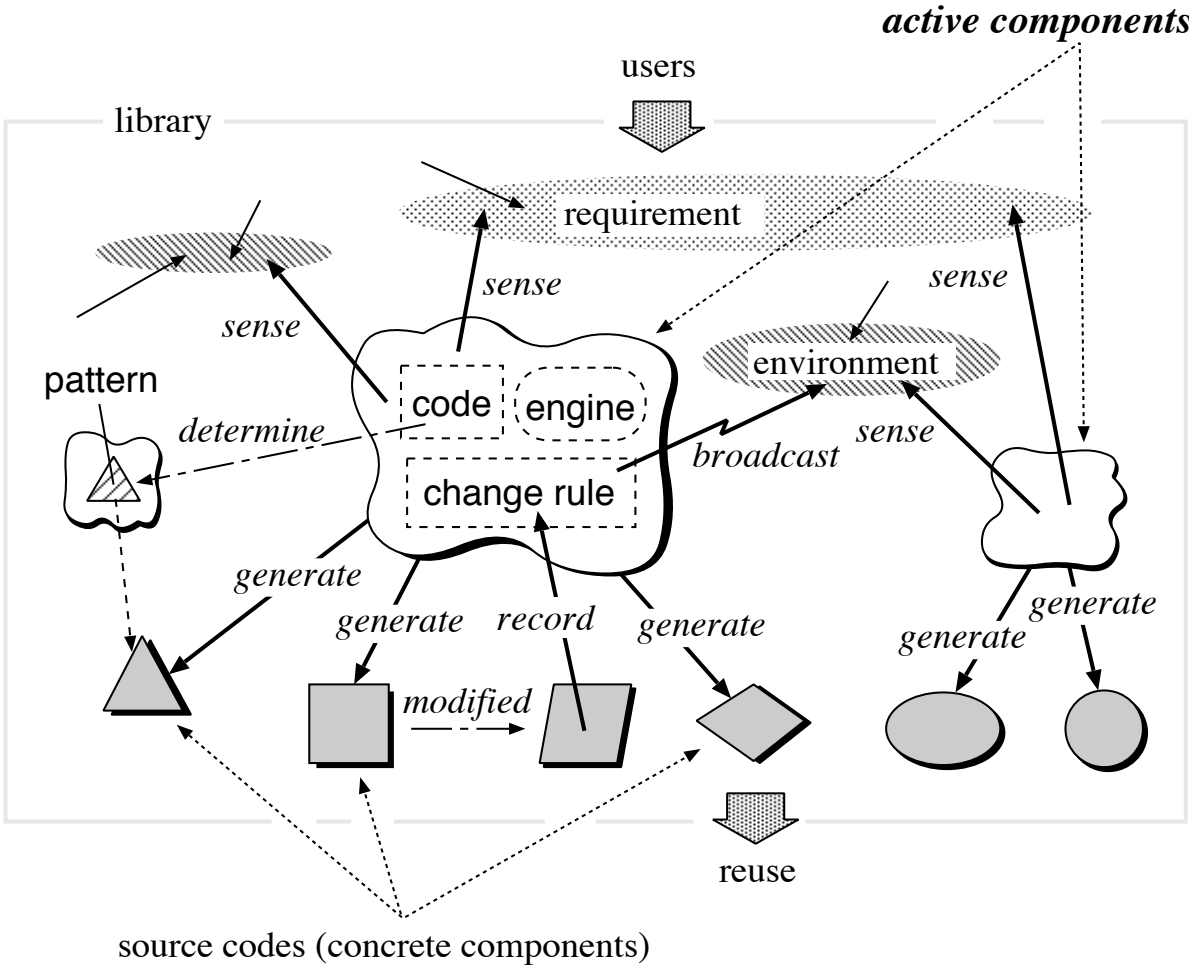


図 5.1: 能動的部品の構造および動作

(4) 能動的部品は，自分の変化規則や感知した要求および環境に基づき，機能分割変化および機能交換変化において，変化パターンを決定する．各能動的部品は，自分の持つ変化パターンに基づき新規部品を生成 (generate) する．

新しい変化パターンが決定不可能なとき，部品変化が終了する．部品利用者は，変更された変化後のソースコードから適切なものを再利用し，目的のプログラムを作成する．

5.2.2 機能分割変化

能動的部品は，自分自身にプログラムスライシングを適用することで，自分の持つ機能を分割し，感知した要求に対して必要なコードだけを集めた新規部品に機能分割変化 (Cde:

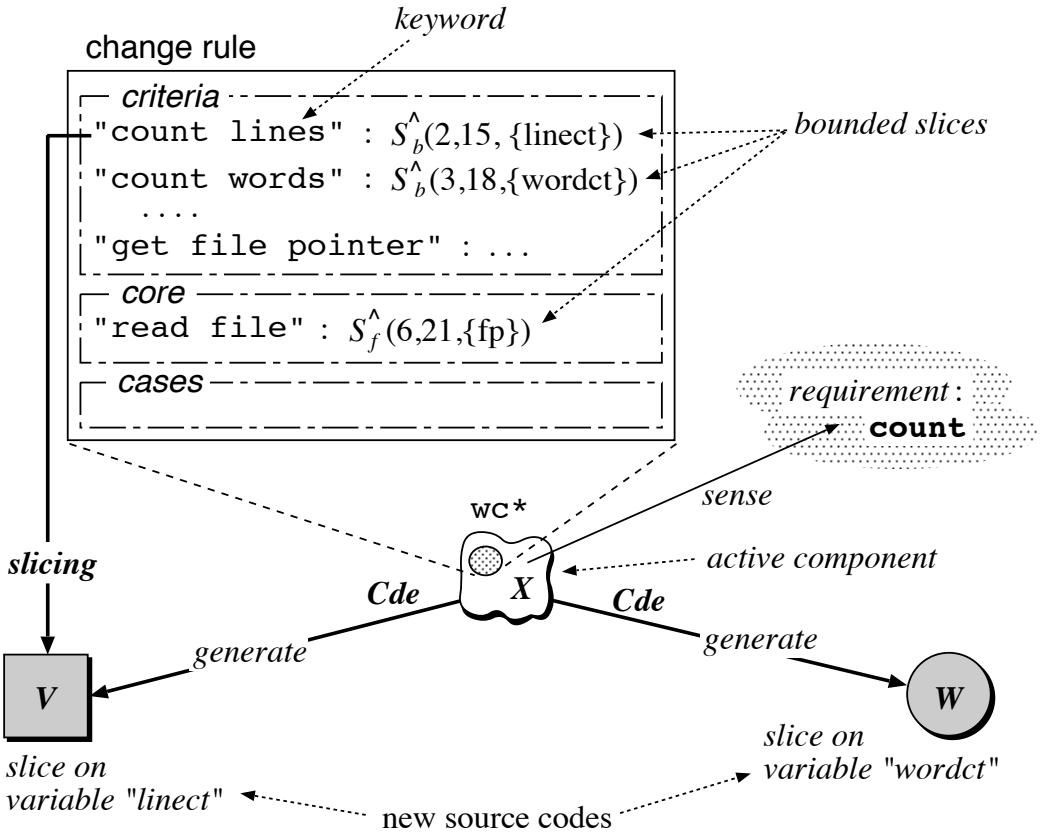


図 5.2: 機能分割変化 (Cde)

decomposing) 可能である．本章では，機能を変数の値を決定することだと仮定したので，部品内部の変数に基づきスライシングを適用することは，能動的部品から特定の機能を抽出していることに等しい．変化後の固定的部品 (スライス) は，スライシング基準変数に関して実行可能かつ実行結果がもとの能動的部品のソースコードと等価である．

能動的部品の変化基準において，ソースコードを機能分割することで抽出可能なスライスは，そのスライスの機能を表すキーワードに関連付けられている．キーワードとは，各スライスにおいて着目した変数 (スライシング基準変数) を説明する文字列である．能動的部品の作成者は，ソースコード内の任意の変数に着目し，その変数に関して作成したスライスにキーワードを割り付け，変化基準を記述する．能動的部品は，部品利用者の要求と変化基準内のキーワードを文字列比較し，要求に対応するキーワードを決定する．変化基準において，決定したキーワードに関連のあるスライシング基準を取り出し，この基準に基づきスライシングを適用することで，機能分割変化を達成する．

機能分割変化の概要を図 5.2に示す．変化基準において，キーワードは "count lines", "count words", ･･･であり，それぞれキーワードの右側にスライスが記述されている．いま，UNIX の wc ユーティリティのソースコード wc を持つ能動的部品 X が，要求として "count"を感知すると，キーワードが文字列 "count"を含むので，「行数を数える」機能を持つ V と「単語数を数える」機能を持つ W が機能分割変化により生成される．ソースコード V, W は能動的部品 X を機能分割したスライスなので，X の機能の一部を持つ．また，V, W は，核であるスライス "read file"の機能「ファイルを読み込む」を含む．

5.2.3 機能交換変化

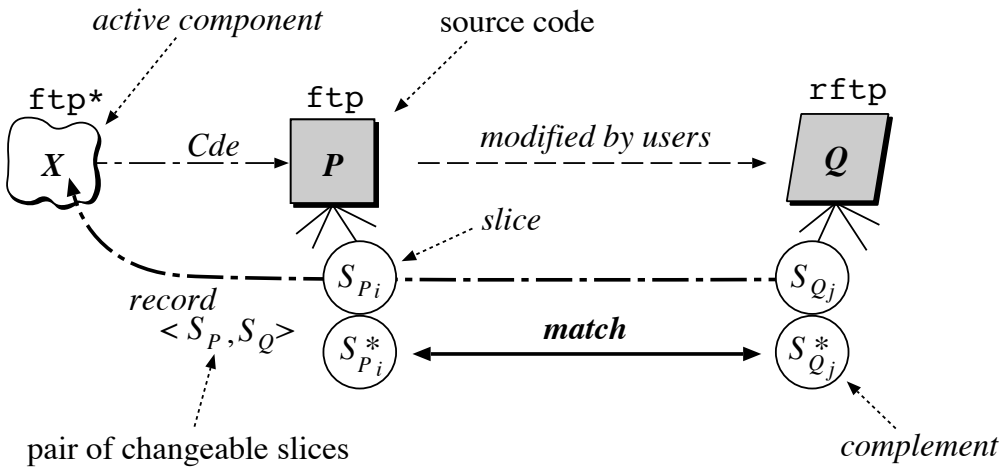
能動的部品が生成したソースコード部品は，再利用時に利用者によって，さまざまな形に変更される．そこで，過去の変更は類似ソースコード部品に対しても将来行われる可能性が高いと判断し，部品変更の事例を用いた変化機構を能動的部品に組み込む．

能動的部品は，自分が生成したソースコードが部品利用者によって変更されたとき，その部品変更を変更事例として自分自身に記録する．変更事例を持つ能動的部品は，部品利用者がライブラリに要求を投入したとき，自分の持つ変更事例を他の能動的部品に広告する．能動的部品は，広告された他の部品の変更事例を感知し，自分の持つ機能の一部を他の部品の変更事例と交換および合成することで，新しい機能を持つ部品に機能交換変化 (Cex: exchanging) 可能である．このように，能動的部品は，機能交換変化により，他の部品が部品利用者から受けた変更を模倣し，自分が受ける可能性のある部品変更を自動的に行う．

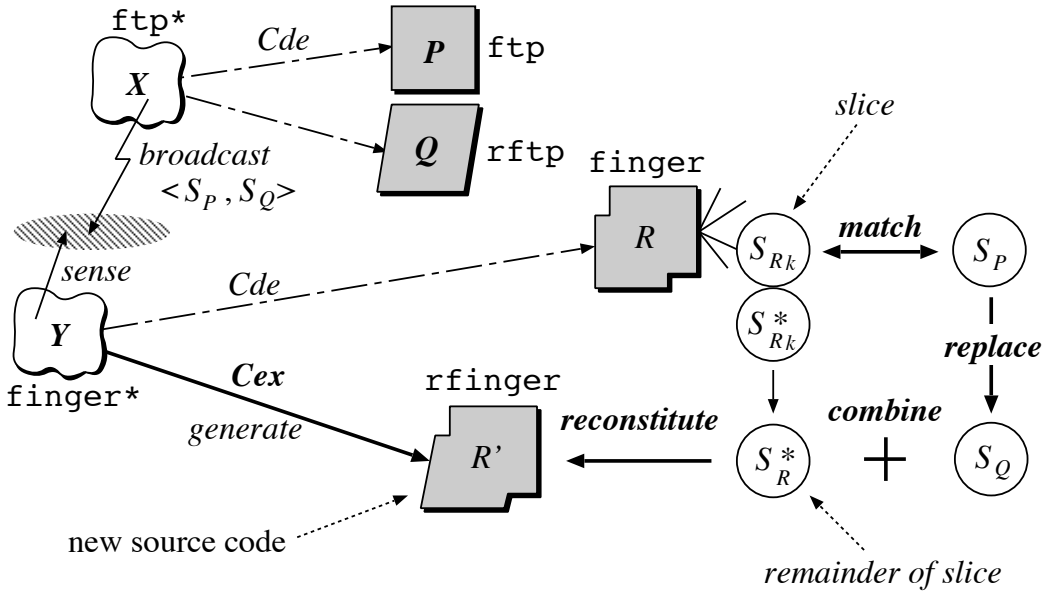
機能交換変化において交換対象は，機能分割変化により生成されたソースコードである．機能交換変化は，部品利用者が部品変更を行った際に変更事例を記録する動作と，部品検索の際に機能の一部を交換する動作に分けられる．

変更事例を記録する動作では，プログラムスライシングにより対象ソースコードを機能分割し，ラベル付き同形写像比較を用いることで機能の一部だけを変更箇所として自動的に特定する．本部品変化メカニズムでは，変更箇所を特定する際に，無変更部分を補スライスで表す．補スライスとは，もとのプログラムからスライスを取り除いたものである．変更前後のソースコードにおいて，補スライスどうしが等価であることは無変更部分が等しいことを指し，変更影響箇所はそれぞれのスライスに閉じ込められる．よって，ラベル付き同形写像比較により，等価な補スライスを見つけることで，変更箇所を特定可能である．

機能の一部を交換する動作では，機能分割とラベル付き同形写像比較により，機能の交換箇所を自動的に特定し，特定したスライスと変更事例に含まれるスライスを交換する．交換



(a) 部品変更時に変更事例を記録する動作



(b) 部品検索時に機能の一部を交換する動作

図 5.3: 機能交換変化 (Cex)

後のスライスと無交換部分を表すスライスの残りを合成することで，能動的部品は機能交換変化を達成する．

図 5.3に機能交換変化の概要を示す．図 5.3a において， P は能動的部品 X が機能分割変

化したソースコードである．部品利用者がソースコード P を Q に変更したとき，能動的部品 X は P, Q からそれぞれ複数個のスライス S_{Pi}, S_{Qj} とその補スライス S^*_{Pi}, S^*_{Qj} を作成する．次に，補スライス S^*_{Pi} と S^*_{Qj} のラベル付き同形写像比較により，交換可能スライス対 $\langle S_P, S_Q \rangle$ を特定し，能動的部品 X の変化規則に変更事例として記録する．交換可能スライス対とは，変更箇所を含む変更前後のスライスの組を指す．

図 5.3b において，ライブラリに要求が与えられたとき，能動的部品 X, Y はそれぞれソースコード P, R に変化する．このとき，能動的部品 X は交換可能スライス対 $\langle S_P, S_Q \rangle$ を持つので，これをライブラリに広告する．交換可能スライス対を感知した能動的部品 Y は，自分の生成したソースコード R にスライシングを適用し，複数個のスライス S_{Rk} を作成する．作成したスライス S_{Rk} と交換可能スライス対の変更前スライス S_P がラベル付き同形写像比較により一致するとき，スライス S_{Rk} と S_Q を交換し，スライス S_Q とスライスの残り S^*_{Rk} の合成を行う．このようにして，能動的部品 Y はソースコード R' に機能交換変化する．

例えば，能動的部品 X から UNIX の ftp ユーティリティのソースコード *ftp* が生成され，部品利用者がソースコード *ftp* を *rftp*[1] に変更した場合を考える．能動的部品 Y は，能動的部品 X の広告した変更事例を感知し， UNIX の finger ユーティリティのソースコード *finger* だけでなく， *finger* の機能の一部を *rftp* と交換したソースコード *rfinger*[1] にも変化する．

5.3 従来手法との比較

本節では，能動的部品の変化メカニズムと，事例に基づきプログラムの自動変更を行う従来手法を比較し，本変化メカニズムの利点について述べる．さらに，本変化メカニズムを実現する際に，従来のプログラム合成アルゴリズムを用いた場合の問題点を示し，提案アルゴリズムにおける改良点を述べる．

5.3.1 事例に基づくプログラム自動変更

能動的部品の機能交換変化のように，過去のプログラム変更の事例に基づき，新しいプログラムを作成するシステムに TA [112] がある． TA は，利用者が与えた仕様 (論理式) に対応する類似プログラムを類推により見つけ，利用者に変更を要求し，変更前後のコードを事例として蓄積する．別の仕様が与えられた際，仕様に対する類推により類似プログラムを見つけ，蓄積した事例が適用できる場合に，プログラムを自動変更する．

[1]代理サーバーを中継する *socks ftp* および *socks finger*.

TA は，利用者が直接変更した部分を変更箇所とし，変更コードが無変更コードに与える影響や，入れ換えたコードがもとのプログラムのコードに与える影響を考慮していない．さらに，TA では，仕様変更とプログラム変更が正しく対応づけられている必要があるが，この対応づけは難しい．よって，この手法では，変更後のソースコードにおいて，無変更部分の機能が破壊されたり，変更事例が正しく反映されない可能性があり，利用者が変更後の部品の機能を理解することは困難である．さらに，機能を保存せずにコードを入れ換えるため，もとの部品と関連を持たない部品が大量に生成されるおそれがある．

これに対して，能動的部品の変化メカニズムは，プログラムの各文の依存関係に基づくプログラム合成アルゴリズムにより実現する．本合成アルゴリズムは，変更コードや入れ換えコードが，もとのソースコードに与える影響範囲を解析する．よって，本変化メカニズムにおいて，過去の部品変更は正しく反映され，変更後の部品は無変更部分に関する機能を保存する．このように，本部品変化アルゴリズムは，変更後の部品が特定の機能を保存することを保証するので，利用者が変更後の部品の機能を理解し易く，無条件に多くの部品が生成されることもない．

5.3.2 合成アルゴリズムの改良

依存関係に基づきプログラムを合成する手法に，HPR アルゴリズム [40]，YHR アルゴリズム [114]，PDI アルゴリズム [85]，BHR アルゴリズム [20] がある．これらのアルゴリズムは，別々に変更された 2 つのプログラムを，合成後のプログラムに追加した差分が互いに矛盾しないことを保証しながら自動合成する．上記の 4 つのアルゴリズムは，変更箇所を特定する方法に関して違いを持つ．HPR および BHR アルゴリズムは，PDG において変更後に追加された節点および変更前後で依存関係が異なる節点を静的に求め，これらの節点に基づきスライスを作成することで変更箇所を特定する．YHR アルゴリズムは，PRG (program representation graph) [114] において，節点のラベルと依存関係矢印の接続関係から変更前後で動作が等価な節点を求め，変更後の全節点を追加，変更，無変更節点に分類する．また，PDI アルゴリズムは，HPR アルゴリズムで用いる後方 (逆方向) スライスに加えて，前方 (順方向) スライスを導入することで，変更箇所と無変更箇所を決定する．4 つのアルゴリズムは，PDG (あるいは PRG) の節点および矢印を重ね合わせることで，変更箇所と無変更箇所を合成する．

本論文で提唱する能動的部品は，ライブラリに存在しない新規部品に自動的かつ動的に変化するため，他の部品の変更事例コードを自分の持つコードと合成する．よって，能動的部

品は，派生元が異なる 2 つのソースコードを合成可能なアルゴリズムを備える必要がある．しかし，上記の合成アルゴリズムは，同一のソースコードから変更により派生した 2 つのプログラムだけを扱い，それぞれの合成対象ソースコードはもとのソースコードと一部の節点を共有し，共有節点の対応関係が特定されている必要がある．よって，派生元が異なる 2 つのソースコードを合成する際には，プログラム構造やラベルの違いにより，共有節点どうしの対応関係を見つけることが難しく，これらのアルゴリズムをそのままの形で機能交換変化に用いることはできない．また，上記の合成アルゴリズムは，合成後のソースコードが意味的な矛盾を含まないことを厳密に保証する．このため，合成可能なソースコードのプログラム構造に対する制約がきつく，合成が成功するソースコードの種類は少ない．よって，これらのアルゴリズムは，部品利用者のさまざまな要求に応じて，数多くの新規ソースコードを自動生成するという目的には適さない．

このように，従来の合成アルゴリズムは，派生元が異なるソースコードにおいて，変更および交換箇所を特定する際に共有節点が求められない，合成後の機能の一部がもとの機能に矛盾することを許さない，という点で機能交換変化を実現するには不十分である．

そこで，能動的部品を実現するために，従来の合成アルゴリズムに対する制約を緩め，より多くの種類のソースコードを合成可能であるアルゴリズムを用意する．本章で提案する合成アルゴリズムでは，従来の合成アルゴリズムに対して，次に示す改良を加えた．

改良 1 プログラム構造が異なる変更前後のソースコードに対して，変更箇所を特定する際に補スライスを導入し，より多くの共有節点の対応関係が決定できるように，スライスだけでなく補スライスどうしの節点の対応関係も用いる．

改良 2 異なるラベルを持つスライスどうしを比較対象とし，機能の交換可能な箇所をより多く特定できるように，ラベルを抽象化する．

改良 3 合成が成功するソースコードの種類を多くするため，交換スライスと無交換部分を合成する際に依存関係が切断される場合でも合成不可能とせず，変更により追加される機能が，もとのソースコードと無矛盾であることを保証したまま，プログラム制御が必ず到達するように制御依存関係矢印を付けかえる．

改良アルゴリズムを備えることで，合成が成功する可能性は高くなり，能動的部品は過去の部品変更を反映した数多くの新規部品に変化可能である．よって，無変更で部品利用者の要求を満たす可能性が高くなる．

5.4 能動的部品に関する諸定義

本節では，能動的部品変化を実現するアルゴリズムに必要な諸定義を述べる．

5.4.1 対象ソースコード

本部品変化メカニズムでは，機能を分割する際にプログラムスライシングを適用するため，能動的部品の持つソースコードとして手続き型言語 Pascal を簡単化した言語のプログラムを扱う．プログラム構造としては，代入文，複合文 (**begin-end**)，条件文 (**if-then**, **if-then-else**)，繰り返し文 (**while**)，入力文 (**read**, **readLn**)，出力文 (**write**, **writeLn**) を持つ．プログラムは逐次実行，条件分岐，繰り返し構造のみで，条件分岐や繰り返しのネストが交差しないこととする．変数の型としては，スカラ型のみで配列型やポインタ型は扱わない．また，本章では，対象ソースコードの PDG G に対して，節点集合を $Node(G)$，矢印集合を $Edge(G)$ と表す．

5.4.2 スライスと補スライス

本変化メカニズムでは，3章で提案した区間限定スライス (bounded slice) を用いる．区間限定スライスとは，従来の静的スライスを任意の対象区間が指定可能となるように拡張したものである．区間限定スライスを用いることで，大規模なソースコードにおいても，一部のコードだけをスライシング対象として扱うことが可能であり，抽出可能な機能の種類が増加する．

本部品変化メカニズムでは，区間限定スライスを節点集合ではなく，PDG の部分グラフで表現する．3.3節で定義した順方向区間限定スライス $\hat{S}_f(n_u, n_l, v)$, $\hat{S}_b(n_u, n_l, v)$ に含まれる節点の集合をそれぞれ N_f, N_b とする．本変化メカニズムで扱う順方向区間限定スライスとは，N_f 内の節点と N_f 内の節点を接続する矢印によって構成される，もとの PDG の部分グラフである．また，逆方向区間限定スライスとは，N_b 内の節点と N_b 内の節点を接続する矢印によって構成される，もとの PDG の部分グラフである．ここで，n_u, n_l, v は，スライシングにおける上限節点，下限節点，着目変数である．本章で扱うスライスはすべて区間限定スライスであるため，区間限定スライスを単にスライスと呼び，方向を区別する必要のないとき，$\hat{S}_f$, $\hat{S}_b$ を $\hat{S}$ と記述する．

さらに，本変化メカニズムの機能交換変化では，変更箇所を特定する際に補スライスを用いる．区間限定スライス $\hat{S}(n_u, n_l, v)$ を作成する際の制約付き到達可能経路集合 $CRP(n_u, n_l)$ から $\hat{S}(n_u, n_l, v)$ に含まれる節点を取り除いた集合を N_c とする．補スライス $\hat{S}^*(n_u, n_l, v)$ と

```
procedure slicing(G)
declare G: 対象プログラムの PDG;  SC: スライスと補スライスの対;
        n_u, n_l: 節点;  N_u, N_l: 節点集合;  V: 変数集合;
begin
  N_u := ∅;  N_l := ∅;  SC := ∅;
  for each n_u ∈ Node(G) do
    if Def(n_u) \ Use(n_u) ≠ ∅ then  N_u := N_u ∪ {n_u}; fi
  od
  for each n_l ∈ Node(G) do
    if Use(n_l) \ Def(n_l) ≠ ∅ then  N_l := N_l ∪ {n_l}; fi
  od
  for each n_l ∈ N_l do
    V := { V' | V' ⊆ [ Use(n_l) \ Def(n_l) ] };
    for each n_u ∈ N_u do
      for each v ∈ V do
        S := Ŝ_b(n_u, n_l, v);  C := Ŝ*_b(n_u, n_l, v);
        SC := SC ∪ {⟨S, C, n_u, n_l⟩};
    od od
  od
  return (SC);
end                                   /* A \ B は A, B の差集合 */
```

図 5.4: スライスおよび補スライスの作成アルゴリズム

は，もとの PDG において， N_c に含まれない節点をヌル節点に置換したグラフである．すなわち，補スライスとは，制約付き到達可能経路外の節点とスライス内部の節点をヌル節点に置換したグラフである．ヌル節点とは，節点の実行前後でプログラムの各変数の値を変化させないラベルを持つ節点である．

機能分割変化では，変化規則に記述された基準に基づき単純にスライシングを行う．機能交換変化において，スライスおよび補スライスを作成するアルゴリズム slicing を図 5.4に示す．このアルゴリズムでは，ソースコード内部に現れる変数に着目し，区間限定スライシングにより各変数のデータ依存関係を途中で分断しないように，ソースコード内で変数の初期値を定義する節点を上限節点 n_u，最後に参照される変数を持つ節点を下限節点 n_l とする．さらに，下限節点 n_l において再定義されない参照変数の集合 V' のベキ集合 V を基準変数に設定する．このようにスライシング基準を設定することで，ソースコード内で定義される各変数の値を決定する計算手続きは，もとのソースコードを機能分割したスライスのどれか一つに必ず含まれる．

```
   procedure matching(G_n, G_m, flag)
   declare G_n, G_m: PDG;  flag: 抽象化 (TRUE) / そのまま (FALSE);
             n, m: 節点;  D[ ]: 節点集合;  B[ ]: 節点対集合;
             a_n[ ], a_m[ ]: ラベル;  F_r[ ], F: 節点およびラベル対集合;
             V_r[ ], V: 変数対集合;  j: 整数;  J: 整数集合;
   begin
     if #Node(G_n) ≠ #Node(G_m) then return (FAILURE, ∅, ∅, ∅); fi
[1]  D := mapping_nodes(G_n, G_m);
[2]  (B, J) := matchingNodes(Node(G_n), D);
     if J = ∅ then return (FAILURE, ∅, ∅, ∅); fi
     for each j ∈ J do
       F_r[j] := ∅;  V_r[j] := ∅;
       for each ⟨n, m⟩ ∈ B[j] do
         if flag = TRUE then
[3]        a_n := abstractionLabel(label(n));
[4]        a_m := abstractionLabel(label(m));
[5]        (F, V) := matchingLabels(n, a_n, m, a_m);
           if F = ∅ ∨ V = ∅ then J := J \ {j}; fi
         else  /* flag = FALSE */
           V := ∅;
           if label(n) = label(m) then
             F := {⟨n, label(n), m, label(m)⟩};
           else J := J \ {j}; fi
         fi
         F_r[j] := F_r[j] ∪ F;  V_r[j] := V_r[j] ∪ V;
     od od
     if J = ∅ then return (FAILURE, ∅, ∅, ∅); fi
     return (SUCCESS, F_r, V_r, J);
   end                                  /* #S は集合 S の要素の数 */
```

図 5.5: ラベル付き同形写像比較アルゴリズム

5.4.3 ラベル付き同形写像比較とラベルの抽象化

本部品変化メカニズムでは，2 つのソースコードの依存関係に関する等価性を判断するために，2.4節で述べた PDG のラベル付き同形写像比較を用いる．各節点のラベルおよび依存関係を表す矢印の種類と接続関係が一致するとき，2 つの PDG はラベル付き同形写像比較において同形であるという．ラベル付き同形写像比較により，変更前後の PDG において無変更な部分，あるいは，交換対象の PDG において交換可能な部分が特定できる．

図 5.5にラベル付き同形写像比較により 2 つの PDG の節点およびラベルどうしの対応関

```
procedure mappingNodes(G_n, G_m)
declare G_n, G_m: PDG; type: 依存関係の種類 (T,F,D);
        v: 節点; D[ ], D_T[ ], D_F[ ], D_D[ ]: 節点集合; W: 節点対集合;
  procedure candidateNodes(n, m, t)
  declare n, m, p, q: 節点; D_n, D_m: 節点集合;
          t: 依存関係の種類;
  begin
    D_n := { G_n の節点 n の t 依存先節点集合 };
    D_m := { G_m の節点 m の t 依存先節点集合 };
    if #D_n ≠ #D_m then return; fi
    D_t[n] := D_t[n] ∪ {m}; W := W ∪ {⟨n, m⟩};
    for each p ∈ D_n do
      for each q ∈ D_m do
        if ⟨p, q⟩ ∉ W then
          for each t ∈ { T, F, D } do
            candidateNodes(p, q, t);
    od fi od od
  end
begin
  W := ∅;
  for each v ∈ Node(G_n) do
    D_T[v] := ∅; D_F[v] := ∅; D_D[v] := ∅; od
  for each type ∈ { T, F, D } do
    candidateNodes(entry, entry, type); od
  for each v ∈ Node(G_n) do
    D[v] := D_T[v] ∩ D_F[v] ∩ D_D[v]; od
  return (D);
end                    /* D はデータ依存関係, T, F は制御依存関係の種類 (真 / 偽) */
```

図 5.6: 節点を写像するアルゴリズム

係を求めるアルゴリズム matching を示す．$label(p)$ は，節点 p のラベルを表す．このアルゴリズムは，同形写像比較が成功したか失敗したか，成功時の節点の対応関係の配列，変数の対応関係の配列，節点および変数対配列において対応関係が存在する要素の集合を返す．関数 mappingNodes (図 5.5[1]) では，依存関係に基づき節点を写像することで，一致する可能性がある節点集合を求める．関数 matchingNodes (図 5.5[2]) では，これらの節点集合から一致する節点の組を作成する．関数 abstractionLabel (図 5.5[3][4]) では，この後で述べるラベルの抽象化を行い，関数 matchingLabels (図 5.5[5]) で抽象化したラベルどうしを比較する．関数 mappingNodes, matchingNodes を図 5.6, 5.7に示す．

```
procedure matchingNodes(N, D)
declare n, m: 節点;  N, M, D[ ]: 節点集合;  B[ ]: 節点対集合;
        i, j: 整数;  J, tmpJ, JJ[ ]: 整数集合;
begin
  i := 0;  J := {0};
  B[0] := {⟨p, q⟩ | p ∈ N ∧ q ∈ D[p]};
  for each n ∈ N do
    for each j ∈ J do
      M := {q | p = n ∧ ⟨p, q⟩ ∈ B[j]};
      JJ[j] := ∅;
      if #M ≠ 1 then
        for each m ∈ M do
          i := i + 1;  JJ[j] := JJ[j] ∪ {i};
          B[i] := B[j] \ {⟨p, q⟩ | p = n ∧ q ∈ D[n]};
          B[i] := B[i] ∪ {⟨n, m⟩};
    od fi od
    tmpJ := J;  J := {k | k ∈ JJ[j] ∧ j ∈ J};
    if J = ∅ then  J := tmpJ;  fi
  od
  tmpJ := J;
  for each j ∈ tmpJ do
    M := {q | ⟨p, q⟩ ∈ B[j]};
    if #M ≠ #N then  J := J \ {j};  fi
  od
  return (B, J);
end
```

図 5.7: 一致する節点を求めるアルゴリズム

ここで，機能交換変化において，変更前後のソースコードの無変更部分では変数名，定数，演算子は等しい．よって，変更箇所を特定する際のラベル付き同形写像比較において，ラベルを加工する必要はない．しかしながら，広告された変更事例を用いて機能の交換箇所を特定する際には，派生元が異なるソースコードがラベル付き同形写像比較の対象となる．よって，ソースコード内部の変数名，定数，演算子は異なり，多くの場合ラベルの不一致により比較が失敗する．そこで，派生元が異なるソースコードを幅広く比較対象とするため，ラベル付き同形写像比較を行う前にラベルの抽象化を行う．さらに，抽象化されたラベルを比較する際には変数名の置換を行う．ラベルの抽象化と変数名の置換により，変更事例が適用できる機会は多くなり，機能交換変化において，新規部品が生成される可能性は高くなる．ラベル付き同形写像比較において，各節点の抽象化ラベルおよび依存関係を表す矢印

の種類と接続関係が一致，かつ， PDG の全節点で成立する置換変数の組合せが存在するとき，抽象化ラベルを持つ 2 つの PDG は同形であるという．

ラベルを抽象化する関数 abstractionLabel を図 5.8に示す．関数 abstraction (図 5.8[1]–[3]) は呼び出しごとに，以下の抽象化操作を (1), (2), (3) の順に行い，ラベルが単一述語になるまで (3) を繰り返す．

(1) 関係演算子 (=, <>, <, >, <=, >=)，代入文 (:=)，入力文，出力文を述語に置換し，論理式の形に形式化する．このとき，演算の優先順位を表す “(”, “)” は “[”, “]” に置換する．

(2) 定数 (定項) と符合 (+, −) を削除する．このとき，前後の項が存在しない演算子も削除する．

(3) 論理演算子 (**or**, **and**, **not**) や “[” “]” による優先順位に基づき新規述語を導入することで，論理演算子と前後の項を結合する．

この抽象化操作では，変数の消去を行わないため，抽象度がもっとも高いラベルはもとのラベルの全変数を変項として持つ．ラベルの一致を判定する際には，抽象化したラベルどうしを抽象度が高いランク 0 から順に比較する．抽象化したラベルを比較する関数 matchingLabels を図 5.9に示す．図 5.9[1][2] の $(p)_{string}$ は抽象化したラベルの述語 p に対応するもとのラベルを指す．変数名の置換は，対応する 2 節点のラベルに現れる変数どうしのすべての組合せについて行い，ラベルが一致するときの変数の組合せを保存する．ラベルの対応関係と置換変数の組合せは，機能交換の際に節点に割り当てられた演算子や変数名の整合性を保つために利用する．

ラベルの抽象化とラベルが一致する様子を図 5.10に示す．図 5.10において，矢印の左側の数字はラベルの抽象化操作を表す．いま，変数名 c を n， x を m に置換すると， 2 つのラベルはランク 0, 1 で一致する．ランク 2 では述語の順序を入れ換えても変数 (変項) の数が一致しないので比較は終了する．このとき，ラベルと置換変数の対応関係はランク 1 の述語の対応関係より，図 5.10に示すようになる．

5.4.4 交換可能スライス対

逆方向区間限定スライスおよび補スライスと，ラベル付き同形写像比較における同形の判定より，機能交換変化における交換可能スライス対を次のように定義する．

```
   procedure abstractionLabel(orig_label)
   declare orig_label, a[ ], l[ ]: ラベル; rank, r: 整数;
   begin
     l[0] := orig_label; rank := 0;
[1]  while l[rank] ≠ 単一述語 do
[2]    rank := rank + 1;
[3]    l[rank] := abstraction(l[rank − 1]);
     od
     for r := 0 to rank do a[r] := l[rank − r]; od
     a[rank + 1] := ⊤;
     return (a);
   end
```

図 5.8: ラベルを抽象化するアルゴリズム

```
   procedure matchingLabels(n, l_n, m, l_m)
   declare l_n[ ], l_m[ ], l'_n, l'_m: ラベル; v_n, v_m: 変数; V[ ]: 変数対集合;
           n, m: 節点; F[ ]: 節点およびラベル対集合; r, i: 整数;
   begin
     F[n] := ∅; V[n] := ∅;
     for each v_n ∈ [Def(n) ∪ Use(n)] do
       for each v_m ∈ [Def(m) ∪ Use(m)] do
         l_n[0] の変数 v_n を変数 v_m に書き換え;
         l_n[0], l_m[0] の v_m 以外の変数を dummy に書き換え;
         r := 0;
         while l_n[r] = l_m[r] ∧ l_n[r] ≠ ⊤ ∧ l_m[r] ≠ ⊤ do
           r := r + 1;
           l_n[r] の変数 v_n を変数 v_m に書き換え;
           l_n[r], l_m[r] の v_m 以外の変数を dummy に書き換え;
         od
         if r > 0 then
           for i := 1 to l_n[r − 1] 内の述語の数 do
[1]          l'_n := (l_n[r − 1]内のi番目の述語)_string;
[2]          l'_m := (l_m[r − 1]内のi番目の述語)_string;
             F[n] := F[n] ∪ {⟨n, l'_n, m, l'_m⟩};
           od
           V[n] := V[n] ∪ {⟨v_n, v_m⟩}; fi
     od od
     return (F, V);
   end                                  /* dummy は予約語 */
```

図 5.9: 抽象化したラベルを比較するアルゴリズム

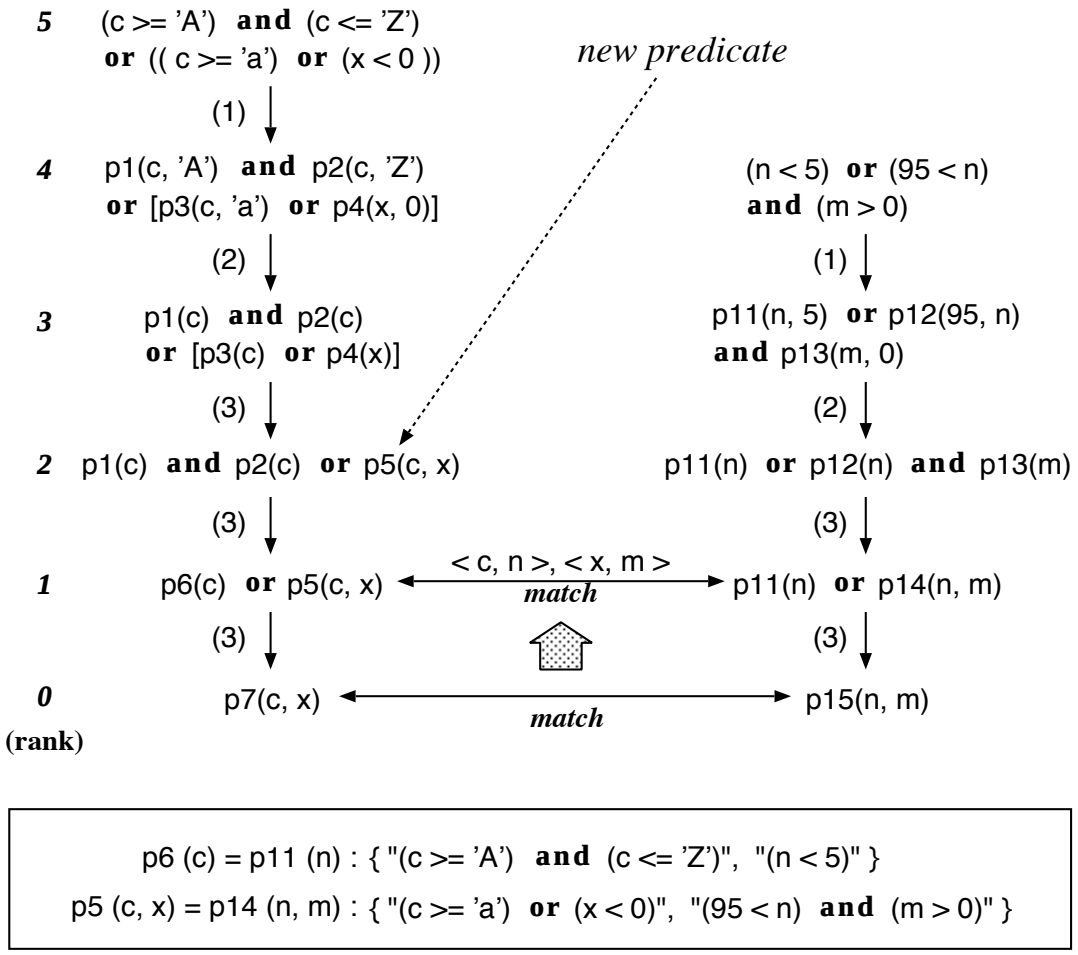


図 5.10: ラベルの抽象化と一致の様子

定義 5.1 逆方向区間限定スライス S_{Pi}, S_{Qj} の補スライスをそれぞれ S^*_{Pi}, S^*_{Qj}，ラベル付き同形写像比較において補スライスどうしが同形であるスライス対の集合を
$U = \{\langle S_{Pi}, S_{Qj}\rangle \mid S^*_{Pi} = S^*_{Qj}\}$ とおく．

- 補スライスどうしが同形 ($\langle S_P, S_Q\rangle \in U$)，かつ，
- 変更前後の補スライスが空集合でなく ($S^*_P \neq \emptyset \wedge S^*_Q \neq \emptyset$)，かつ，
- 変更前補スライスが他の変更前補スライスに含まれない
  ($\forall S^*_{Pi}(S^*_P \not\subset S^*_{Pi} \wedge \langle S_{Pi}, S_{Qj}\rangle \in U)$) とき，

$\langle S_P, S_Q\rangle$ を交換可能スライス対とする．

```
   procedure activeComponent
   declare Cde, Cex: 生成されるソースコードの集合;
           P, Q: ソースコード; Req: 要求 (キーワード);
           ⟨S_P, S_Q⟩: 交換可能スライス対; F_PQ: 節点およびラベルの対応関係;
   begin
     do /* 能動的部品はたえず動作中 */
[1]    if Req を感知 then
[2]      Cde := decompose(Req);
[3]      for each 変化規則内の変更事例 ⟨S_P, S_Q⟩ do
[4]        ⟨S_P, S_Q⟩, F_PQ の広告; od
[5]    else if 部品変更 P → Q を感知 then
[6]      Cde := decompose(all);
[7]      if P ∈ Cde then record(P, Q); fi fi
[8]    else if ⟨S_P, S_Q⟩, F_PQ を感知 then
[9]      Cex := exchange(Cde, ⟨S_P, S_Q⟩, F_PQ); fi
       fi
     od
   end                                  /* all はすべての変化条件を満たす予約語 */
```

図 5.11: 能動的部品の基本動作

5.5 能動的部品変化アルゴリズム

5.5.1 基本動作

機能分割変化および機能交換変化を行う能動的部品の基本動作アルゴリズム activeComponent を図 5.11に示す．図 5.11に示す動作は，各能動的部品で並行に実行される．以下にアルゴリズムの詳細を述べる．

(1) 要求を感知した際の動作 (図 5.11[1]–[4])

取得した要求に応じて，機能分割変化を行い，新規ソースコードを生成する (図 5.11[2])．さらに，交換可能スライス対を持つとき，変更事例を自分の存在するライブラリに広告する (図 5.11[4])．変更事例とは，交換可能スライス対 $\langle S_P, S_Q \rangle$ と節点およびラベルの対応関係 F_{PQ} を指す．

(2) 部品変更を感知した際の動作 (図 5.11[5]–[7])

ソースコード P が Q に変更されたことを感知したとき，機能分割変化により生成可能なすべてのソースコードを生成する (図 5.11[6])．変更前ソースコード P が自分の生成したソー

```
  procedure decompose(Req)
  declare Req, word: 文字列;  S: ソースコード (スライス);
          n_u, n_l: 節点;  v: 変数集合;  dir: スライシングの方向;
          Cde: 生成されるソースコードの集合;
  begin
    Cde := ∅;
    for each  word ∈ {変化基準のキーワード集合}  do
[1]   if  Req が word の部分文字列  then
[2]     ⟨n_u, n_l, v, dir⟩ := スライシング基準 (word);
[3]     S := Ŝ_dir(n_u, n_l, v);
[4]     if 核 ⊆ S then Cde := Cde ∪ { S };  fi
    fi od
    return (Cde);
  end
```

図 5.12: 機能分割変化アルゴリズム

スコードであるとき，変更箇所の特定を試み，変更事例を記録する (図 5.11[7])．

(3) 変更事例を感知した際の動作 (図 5.11[8][9])

取得した変更事例を用いて，機能交換変化を行う (図 5.11[9])．能動的部品は，一度感知した変更事例に唯一の識別子を付加することで，同じ機能交換変化を繰り返し行わない．

5.5.2 機能分割変化アルゴリズム

能動的部品が，機能分割変化する際のアルゴリズム decompose を図 5.12に示す．以下にアルゴリズムの詳細を述べる．

(1) キーワードと変化基準の比較 (図 5.12[1])

部品利用者の投入した要求文字列 Req が，変化基準のキーワード $word$ の部分文字列かどうか判定する．

(2) スライシングの適用 (図 5.12[2][3])

要求との文字列比較が成立するキーワード $word$ に対応するスライシング基準を取得し，スライシングを実行する．

(3) 核に対する検査 (図 5.12[4])

生成するスライスが核を含むかどうか検査し，核を含むソースコードのみ生成する．

```
procedure record(P, Q)
declare P, Q: ソースコード;  n_u, n_l, m_u, m_l: 節点;
        G_P, G_Q: PDG;  V: 変数対集合;  F[ ], F_c, F_r: 節点およびラベル対の集合;
        S_P, S_Q: スライス (PDG);  S*_P, S*_Q, S*_P', S*_Q': 補スライス (PDG);
        SC_P, SC_Q: スライスと補スライス対の集合;  S, U: スライス対集合;
        match: 一致 (SUCCESS / FAILURE);  j: 整数;  J: 整数集合;
begin
  G_P := P の PDG;  G_Q := Q の PDG;
[1] SC_P := slicing(G_P);  SC_Q := slicing(G_Q);
  U := ∅;
  for each ⟨S_P, S*_P, n_u, n_l⟩ ∈ SC_P do
    for each ⟨S_Q, S*_Q, m_u, m_l⟩ ∈ SC_Q do
[2]   S*_P' := reduce(S*_P, S_P);  S*_Q' := reduce(S*_Q, S_Q);
[3]   (match, F, V, J) := matching(S*_P', S*_Q', FALSE);
      if match = SUCCESS then
        for each j ∈ J do
          U := U ∪ {⟨S_P, n_u, n_l, S_Q, m_u, m_l, F[j]⟩};
      od fi
  od od
[4] S := {⟨S_P, n_u, n_l, S_Q, m_u, m_l, F_c⟩ ∈ U |
          S*_P ≠ ∅ ∧ S*_Q ≠ ∅ ∧ ∀S*_Pi(S*_P ⊄ S*_Pi ∧ ⟨S_Pi, n_ui, n_li, S_Qj, m_uj, m_lj, F_c⟩ ∈ U)};
  for each ⟨S_P, n_u, n_l, S_Q, m_u, m_l, F_c⟩ ∈ S do
[5]   F_r := subgraph(S_P, n_u, n_l, S_Q, m_u, m_l, F_c);
    for each F_PQ ∈ F_r do
[6]     交換可能スライス対 ⟨S_P, S_Q⟩, F_PQ を記録;
  od od
end
```

図 5.13: 変更事例を記録するアルゴリズム

5.5.3 変更事例の記録アルゴリズム

能動的部品が，変更事例を記録するアルゴリズム record を図 5.13に示す．本アルゴリズムには，変更箇所を特定する際と変更箇所における節点の対応関係を求める際に，補スライスを用いる改良 1 が加えられている．以下にアルゴリズムの詳細を述べる．

(1) スライスと補スライスの作成 (図 5.13[1])

変更前後のソースコード P, Q に対するスライシング基準を決定し，逆方向区間限定スライス S_{Pi}, S_{Qj} と補スライス S^*_{Pi}, S^*_{Qj} を作成する．

```
procedure reduce(G, G_N)
declare G, G_N: PDG;  n_s, n_d: 節点;  e: 矢印;
begin
  Node(G) := Node(G) ∪ { NULL };
  for each e ∈ Edge(G) do
    n_s := src(e);  n_d := dst(e);
    if n_s∈Node(G_N) ∧ n_d∈Node(G_N) then
      Edge(G) := Edge(G) \ {e};
    else if n_s ∈ Node(G_N) then
      Edge(G) := Edge(G) ∪ { NULL→ n_d} \ {e};  fi
    else if n_d ∈ Node(G_N) then
      Edge(G) := Edge(G) ∪ {n_s → NULL } \ {e};  fi
    fi
  od
  Node(G) := {n | ∃p(p → n) ∨ ∃q(n → q)};
  return (G);
end              /* src(e), dst(e) は矢印 e の接続元および接続先節点， NULL はヌル節点 */
```

図 5.14: 補スライスを変形するアルゴリズム

(2) 補スライスの変形 (図 5.13[2])

変更部分に含まれるヌル節点どうしの結合関係まで含めると，補スライス S^*_{Pi} と S^*_{Qj} は同形にならない．そこで，ヌル節点以外の節点に関する結合関係を破壊しないように，ヌル節点どうしの接続関係だけを取り除き，ヌル節点を 1 つに集めることで， PDG で表現された補スライスを変形する．補スライスを変形する関数 reduce を図 5.14に示す．

(3) 交換可能スライス対の特定 (図 5.13[3][4])

変更により影響を受けた範囲を特定するために，変形後の補スライス $S^*_{Pi}{}'$， $S^*_{Qj}{}'$ に対して，ラベル付き同形写像比較を行い，補スライスが一致するスライス対 $\langle S_{Pi}, S_{Qj}\rangle$ と節点の対応関係 F_c を決定する (改良 1)．この際，ラベルの抽象化は行わない (関数 matching の引数を FALSE とする)．さらに，定義 5.1を用いて，補スライスが一致するスライス対から交換可能スライス対 $\langle S_P, S_Q\rangle$ を特定する．

(4) 交換可能スライス対の節点の対応 (図 5.13[5])

交換可能スライス対 $\langle S_P, S_Q\rangle$ において，節点の対応関係を求める関数 subgraph を図 5.15に示す． S_P および S_Q にそれぞれ包含されるスライス S_A， S_B のラベル付き同形写像比較 (図 5.15[1]) により，対応関係 F_s を決定する．さらに，補スライス内部の節点の対応関係 F_c

```
   procedure subgraph(S_P, n_u, n_l, S_Q, m_u, m_l, F_c)
   declare S_P, S_Q, S_A, S_B: スライス (PDG);  n_u, n_l, m_u, m_l: 節点;
           N, N_t: 節点対集合;  F[ ], F_c, F_s, F_r, F_t: 節点およびラベル対集合;
           V_P, V_Q, V_d, V: 変数集合;  V: 変数対集合;  j: 整数;  J: 整数集合;  match: 一致;
   begin
     F_r := ∅;
     V_P := ⋃_{p∈Node(S_P)} Def(p);  V_Q := ⋃_{q∈Node(S_Q)} Def(q);
     V_d := {V | V ⊆ [V_P ∩ V_Q]};
     for each V ∈ V_d do
       S_A := Ŝ_b(n_u, n_l, V);  S_B := Ŝ_b(m_u, m_l, V);
       if S_A ⊆ S_P ∧ S_B ⊆ S_Q then
[1]      (match, F, V, J) := matching(S_A, S_B, FALSE);
         if match = SUCCESS then
           for each j ∈ J do  F_r := F_r ∪ F[j];  od
     fi fi od
     F := ∅;
     for each F_s ∈ F_r do
[2]    N := {⟨p, q⟩ | ⟨p, l_p, q, l_q⟩ ∈ [F_c ∪ F_s]};
[3]    N_t := {⟨p, q⟩ | p_1 → p → p_2 ∧ q_1 → q → q_2 ∧ ⟨p_1, q_1⟩ ∈ N ∧ ⟨p_2, q_2⟩ ∈ N};
[4]    for each ⟨p, q⟩ ∈ N_t do
[5]      if ∃q_i(⟨p_i, q_i⟩ ∈ N_t ∧ p = p_i ∧ q ≠ q_i) ∨ ∃p_j(⟨p_j, q_j⟩ ∈ N_t ∧ p ≠ p_j ∧ q = q_j) then
[6]        N_t := N_t \ {⟨p, q⟩};
       fi od
[7]    F_t := {⟨p, label(p), q, label(q)⟩ | ⟨p, q⟩ ∈ N_t};
       F := F ∪ { F_s ∪ F_t };
     od
     return (F);
   end
```

図 5.15: 節点の対応関係を求めるアルゴリズム

と，包含されるスライス内部の節点の対応関係 F_s における節点間の接続関係から，対応関係が未決定な節点の対応関係 F_t を求める (改良 1) (図 5.15[2]–[7]).

(5) 交換可能スライス対の記録 (図 5.13[6])

交換可能スライス対 $\langle S_P, S_Q \rangle$ と節点の対応関係 $F_{PQ} = F_s \cup F_t$ を変化規則に記録する.

5.5.4 機能の交換アルゴリズム

能動的部品が，機能の一部を交換し，新規ソースコードを生成するアルゴリズム exchange を図 5.16に示す．本アルゴリズムには，交換箇所を特定する際にラベルを抽象化する改良 2

```
  procedure exchange(Cde, ⟨S_P, S_Q⟩, F_PQ)
  declare Cde, Cex: 生成されるソースコードの集合; G_R, G_R1, G_R2, G_R3: PDG;
          ⟨S_P, S_Q⟩: 交換可能スライス対; R, R': ソースコード;
          F[ ], F_RP, F_PQ: 節点およびラベル対の集合; N_r, T_r: 節点対集合; V: 変数対集合;
          S_R, S*: スライスと補スライス (PDG); SC_R: スライスと補スライス対の集合;
          n_Ru, n_Rl, n, m: 節点; N: 節点集合; J: 整数集合; match: 一致;
  begin
    Cex := ∅;
    for each R ∈ Cde do
[1]   G_R := R の PDG; SC_R := slicing(G_R);
      for each ⟨S_R, S*, n_Ru, n_Rl⟩ ∈ SC_R do
[2]     (match, F, V, J) := matching(S_R, S_P, TRUE);
        if match = SUCCESS then
          for each j ∈ J do
            F_RP := F[j];
[3]         G_R1 := replace(G_R, F_RP, F_PQ);
[4]         N := {p | ⟨r, l_r, p, l_p⟩ ∈ F_RP ∧ ⟨p, l_p, q, l_q⟩ ∉ F_PQ};
[5]         (G_R2, T_r) := combine(G_R1, S_Q, N, V, F_RP);
            for each N_r ∈ T_r do
              G_R3 := G_R2 のコピー;
[6]           for each ⟨n, m⟩ ∈ N_r do
[7]             Edge(G_R3) := Edge(G_R3) ∪ {n → m}; od
[8]           R' := ReconstituteProgram(G_R3);
              if R' ≠ FAILURE ∧ 核 ⊆ R' then Cex := Cex ∪ { R' }; fi
    od od fi od od
    return (Cex);
  end
```

図 5.16: 機能を交換するアルゴリズム

と，交換スライスと無交換部分を合成する際に制御依存矢印の付けかえを行う改良 3 が加えられている．以下にアルゴリズムの詳細を述べる．

(1) スライスの作成 (図 5.16[1])

変化後のソースコード R に対するスライシング基準を決定し，逆方向区間限定スライス S_{Rk} を作成する．

(2) 交換対象スライスの特定 (図 5.16[2])

スライス S_{Rk} と交換可能スライス対の変更前スライス S_P のラベル付き同型写像比較により，交換対象スライス S_R を決定する．このとき，ラベルの抽象化を行う (改良 2) (関数

```
   procedure replace(G_R, F_RP, F_PQ)
   declare G_R, G_R4, G_R5: PDG;
           F_RP, F_PQ: 節点対およびラベル対の集合;
   begin
[1]  G_R4 := relaceNodes(G_R, F_RP);
[2]  G_R5 := relaceNodes(G_R4, F_PQ);
     return (G_R5);
   end
   procedure replaceNodes(G_R, F)
   declare G_R: PDG;  F: 節点およびラベル対集合
           n, m: 節点;  N: 節点対集合;  e_t: 矢印;
   begin
     N := {⟨n, m⟩ | ⟨n, l_n, m, l_m⟩ ∈ F};
     for each ⟨n, m⟩ ∈ N do
       Node(G_R) := Node(G_R) ∪ {m};
       for each e_t ∈ {e' | n = src(e')} do
         Edge(G_R) := Edge(G_R) ∪ {m →_t dst(e_t)};  od
       for each e_t ∈ {e' | n = dst(e')} do
         Edge(G_R) := Edge(G_R) ∪ {src(e_t) →_t m};  od
       Edge(G_R) := Edge(G_R) \ {n → dst(e_t)}
       Edge(G_R) := Edge(G_R) \ {src(e_t) → n}
       Node(G_R) := Node(G_R) \ {n};
     od
     return (G_R);
   end                                  /* →_t は e_t と同種類の依存関係矢印 */
```

図 5.17: スライスを置換するアルゴリズム

matching の引数を `TRUE` とする).

(3) 交換対象スライスの置換 (図 5.16[3][4])

交換可能スライス対 $\langle S_P, S_Q \rangle$ を用いて，交換対象ソースコードの PDG G_R における交換対象スライス S_R の節点を置換する．スライスを置換する関数 replace を図 5.17に示す．

置換は 2 段階に分けられる． 1 段目の置換 (図 5.17[1]) は対応関係 F_{RP} を用いて， S_R の一部の節点を S_P の一部の節点に置き換える． 2 段目の置換 (図 5.17[2]) は対応関係 F_{PQ} を用いて， 1 段目で置換された S_P の一部の節点を S_Q の一部の節点に置き換える．対応関係 F_{PQ} において対応する節点が存在しない S_P に含まれる節点はヌル節点に置き換える．また，置換後の節点には， G_R のラベルをそのまま割り当てる．

```
   procedure combine(G_R, S_Q, N, V, F_RP)
   declare G_R, S_Q: PDG;  n, m: 節点;  N, N_G, ID, PD: 節点集合;  N_r, T_r: 節点対集合;
           e: 矢印;  E: 矢印集合;  l_n, l_m: ラベル;  v_n, v_m: 変数;  V: 変数対集合;
   begin
[1]  Node(G_R) := Node(G_R) ∪ Node(S_Q);
[2]  Edge(G_R) := Edge(G_R) ∪ Edge(S_Q);
     E := {e' | src(e') ∈ N ∧ e'が制御依存矢印};
     N_r := ∅;
     for each e ∈ E do
       if src(e) ∈ N and dst(e) ∉ N then
         m := src(e);

[3]      ID(m) := {q | q ∈ dom(m) ∧ q ≠ m ∧ q ∉ N ∧ ¬∃r(q ∈ dom(r) ∧ r ∈ dom(m))};
[4]      PD(m) := {q | p ∈ ID(m)
                       ∧q ∈ pdom(p) ∧ qが条件分岐(if)節点あるいは繰り返し(while)節点};

[5]      for each n ∈ PD(m) do
[6]        if e の種類 = T then  N_r := N_r ∪ {⟨n, m⟩};  fi
[7]        if e の種類 = F ∧ mが条件分岐(if)節点 then  N_r := N_r ∪ {⟨n, m⟩};  fi
     od fi od
     T_r := {T | T ⊆ N_r ∧ ∀⟨n, m⟩ ∈ T(nの接続矢印の数 = 1)};
[8]  Edge(G_R) := {e' | e' ∈ Edge(G_R) ∧ src(e') ∉ N ∧ dst(e') ∉ N};
[9]  Node(G_R) := { p | p ∈ Node(G_R) ∧ p ∉ N};
     L := {⟨l_n, l_m⟩ | ⟨n, l_n, m, l_m⟩ ∈ F_RP ∧ n ∈ N};
     for each n ∈ [ Node(G_R) ∩ Node(S_Q) ] do
       for each ⟨l_n, l_m⟩ ∈ L do
[10]     if label(n) = l_n then  label(n) := l_m;  fi  od
       for each ⟨v_n, v_m⟩ ∈ V do
[11]     G_R3のlabel(n)の変数v_nをv_mに書き換え;  od
     od
     return (G_R, T_r);
   end
```

図 5.18: 交換スライスとスライスの残りを合成するアルゴリズム

(4) 交換スライスと無交換部分の合成 (図 5.16[5]–[7])

交換可能スライス対の変更後スライス S_Q を，無交換 (無置換) 部分を表すスライスの残り $G_{R1}(= S_R^*)$ に重ねることで合成を行う．合成を行う関数 combine を図 5.18に示す． S_Q と G_{R1} の重ね合わせは， S_Q と G_{R1} の節点および矢印集合の和集合を作成することで行う (図 5.18[1][2])．重ね合わせ終了後，接続元や接続先がヌル節点となる矢印を消去する．このとき，無条件に PDG の矢印を消去すると，制御フローグラフ (CFG) [3] において制御が

移らない節点が生じることがあるので，G_{R1} 上の節点のデータ依存関係に影響を与えない，かつ，すべての節点にプログラム制御が移るように制御依存矢印を付けかえる (改良 3)．

PDG から制御依存関係だけを抜きだして作成した CDG (control depdendence graph) [29] において，節点 `entry` から節点 n に矢印を順方向にたどるときの経路上の節点 (支配元節点) 集合を $dom(n)$，節点 n から接続先がなくなるまで矢印を順方向にたどるときの経路上の節点 (支配先節点) 集合を $pdom(n)$ とおく．このとき，関数 combine において，$ID(m)$, $PD(m)$ を求める (図 5.18[3][4])．$ID(m)$ は消去される節点 m に制御依存関係を持ち，他の節点を経由せずに到達できる節点集合である．$PD(m)$ は消去節点 m と同じ制御依存元 (直接および間接依存元) 節点を持つ最小の節点集合である．このように，$PD(m)$ を選択することで，変更を確実に反映でき，もとの機能に矛盾する範囲を最小限に抑えることが可能である．$PD(m)$ と節点 m を接続元とする矢印 e の依存関係の種類 (T/F) により，矢印 e の付けかえ候補節点集合 N_r を決定 (図 5.18[5]-[8]) し，ヌル節点及びヌル節点に接続を持つ矢印を消去する (図 5.18[9][10])．

合成により追加されたスライス S_Q の節点のラベルは，合成時に消去される S_P 内の節点のラベルと文字列比較することで，S_R 内の節点の抽象化していないもとのラベルと置換する (図 5.18[11])．また，S_P と S_R のラベル付き同形写像比較により求めた変数の対応関係を用いて，S_Q 内の節点のラベルにおける変数名も書きかえる (図 5.18[12])．

(5) ソースコードへの再構成 (図 5.16[8])

アルゴリズム ReconstituteProgram [10] を用いて，合成後の PDG をソースコードに再構成する．このとき，定義順序依存関係 (def-order dependence) 矢印を追加し，データ依存関係矢印をループ経由データ依存関係 (loop carried) 矢印とループ独立データ依存関係 (loop independent) 矢印に分類する．再構成が成功し，再構成後のソースコード R' が核を含む場合，R' を新規ソースコードとして生成する．

5.6 アルゴリズムの適用例

本節では，能動的部品が前述のアルゴリズムを適用して変化する例を示す．

5.6.1 機能分割変化の例

「入力された英文テキストに対して，英字，数字，特殊文字の出現回数を求め，結果を数値とヒストグラムで表示する」ソースコードと，図 5.19に示す変化規則を持つ能動的部

active component 'Histogram'

```
# CRITERIA
"count letter"   :Sb(1,16,{letter})+{16}
"count digit"    :Sb(2,23,{digit})+{23}
"count letters"  :Sb(1,23,{letter,digit})+{16,23}
"count special"  :Sb(3,30,{special})+{30}
"count visible"  :Sb(1,35,{letter,digit,special})
"display letter" :Sf(15,21,{letter})
"display digit"  :Sf(22,28,{digit})
"display special":Sf(22,28,{digit})
# CORE
"read characters":Sb(5,14,{ch})
```

図 5.19: 能動的部品の変化規則の例

```
     procedure Histogram.count_letters;
     var ch: char;
         letter,digit: integer;
     begin
*p1    letter := 0;
 p2    digit := 0;
*p3    read(ch);
*p4    while ch <> EOF do
         begin
*p5        if (ch>='A') and (ch<='Z')
           or (ch>='a') and (ch<='z') then
*p6          letter := letter + 1;
 p7        if (ch>='0') and (ch<='9') then
 p8          digit := digit + 1;
*p9        read(ch)
         end;
 p10   writeLn(letter);
 p11   writeLn(digit)
     end.
```

図 5.20: 変更前のソースコード P

品 *Histogram* を考える．要求を表すキーワード "count"を部品ライブラリに投入すると，*Histogram* は，"count"を部分文字列とするキーワードに対応する変化基準に基づき，5 つ

```
     procedure Histogram.count_lettrs;
     var ch: char;
         letter,digit: integer;
         uppercase,lowercase: integer;
     begin
*q1+   uppercase := 0;
*q2+   lowercase := 0;
 q3    digit := 0;
*q4    read(ch);
*q5    while ch <> EOF do
         begin
*q6+       if (ch>='A') and (ch<='Z')
*q7+         uppercase := uppercase + 1
*q8+       else if (ch>='a') and (ch<='z') then
*q9+         lowercase := lowercase + 1;
 q10       if (ch>='0') and (ch<='9') then
 q11         digit := digit + 1;
*q12       read(ch)
         end;
*q13+  letter := uppercase + lowercase;
 q14   writeLn(letter);
 q15+  writeLn(uppercase);
 q16+  writeLn(lowercase);
 q17   writeLn(digit)
     end.                          + : modified statements
```

図 5.21: 変更後のソースコード Q

のソースコードを生成する．これらのソースコードは，核 CORE に示すスライスを含む．*Histogram* が生成したソースコードの一つ "count letters"を，図 5.20に示す．

5.6.2 機能交換変化の例

機能交換変化により，能動的部品が変更事例を記録する例と，機能を交換することで新規ソースコードを生成する例を示す．

(1) 変更事例の記録

能動的部品 *Histogram* が，機能分割変化より生成したソースコード P が，部品利用者によりソースコード Q に変更されたときを考える． P, Q を図 5.20, 5.21に示す．各文の左の番号は節点識別子，識別子の右側の + は変更箇所を指す． P は「入力されたテキストに対して，英字と数字の出現回数を求める」機能を持つ． Q は P に対して，「英字の大文字と

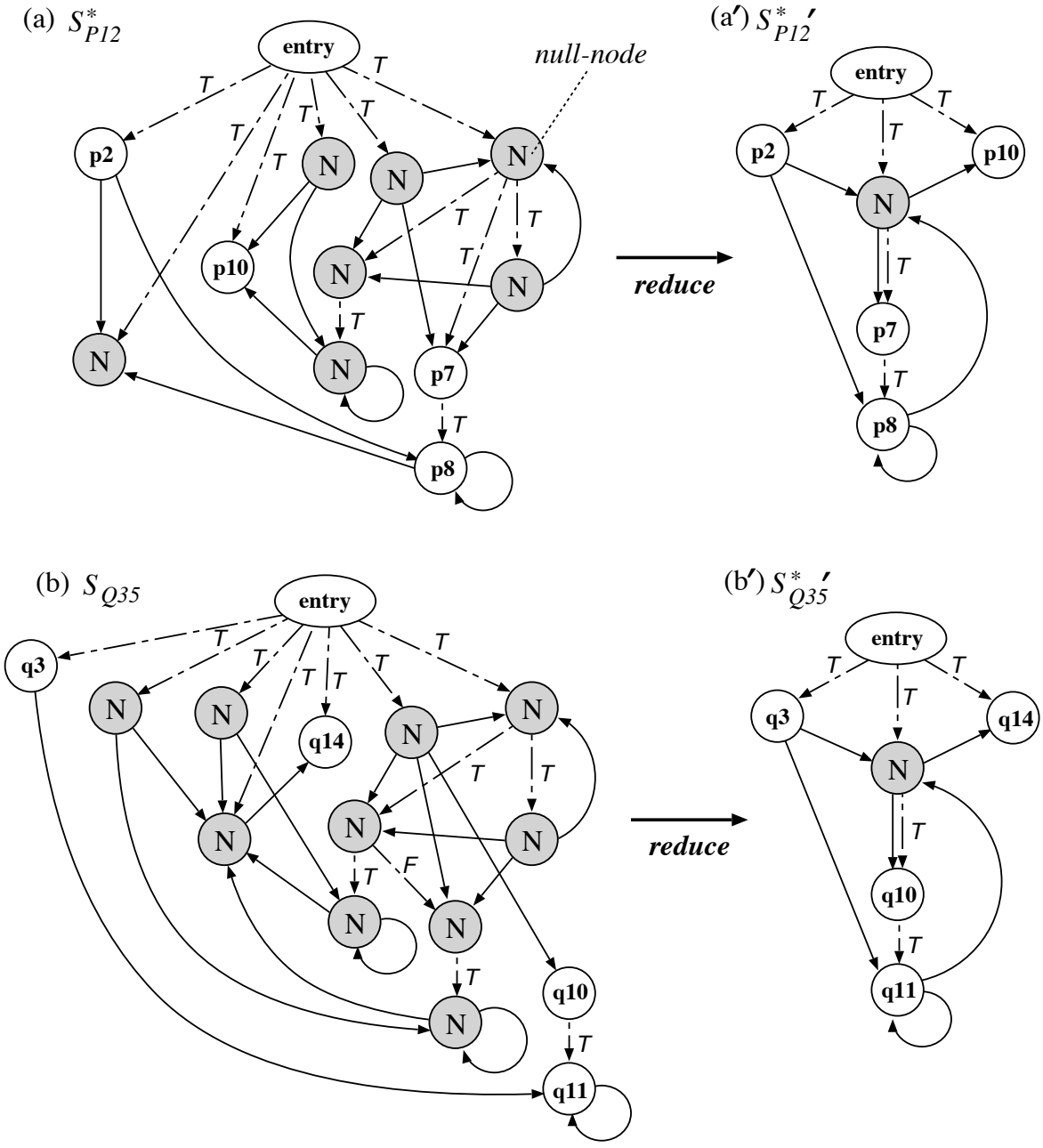


図 5.22: 補スライスの変形 ($S^*_{P12} \rightarrow S^*_{P12}{}'$, $S^*_{Q35} \rightarrow S^*_{Q35}{}'$)

小文字を区別して出現回数を求める」ように変更を加えたソースコードである．P, Q に対して上限および下限節点集合と基準変数を求め，スライスと補スライスの対を作成する．ここでは，以降の説明の例として，$S_{P12} = \hat{S}_b(\mathrm{p1},\ \mathrm{p10},\ \{\mathrm{letter}\})$, $S^*_{P12} = \hat{S}_b{}^*(\mathrm{p1},\ \mathrm{p10},$

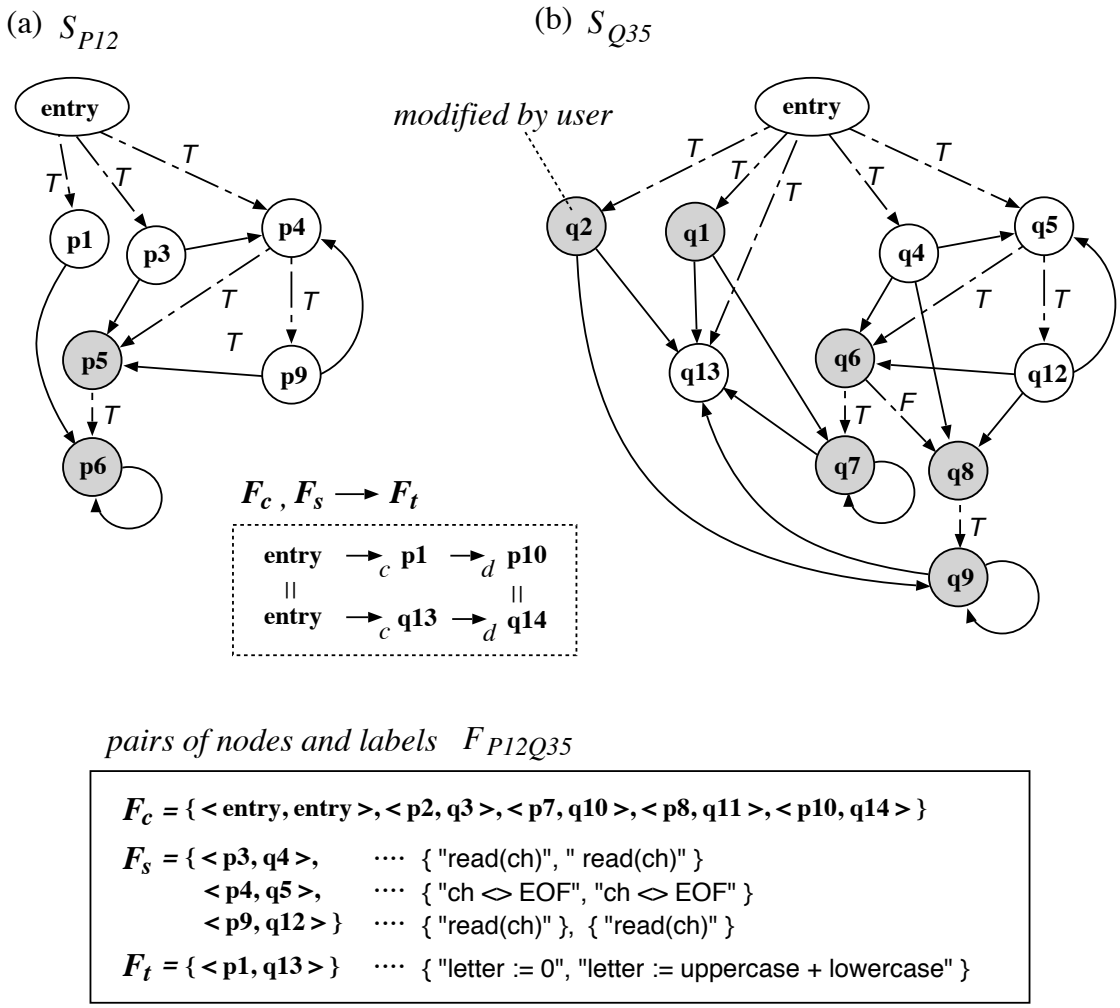


図 5.23: 交換可能スライス対 $\langle S_{P12}, S_{Q35} \rangle$

{letter}), $S_{Q35} = \hat{S}_b$(q1, q14, {letter}), $S^*_{Q35} = \hat{S}_b{}^*$(q1, q14, {letter}) を取り上げる．図 5.20, 5.21において，識別子の左側の $*$ はそれぞれ S_{P12}, S_{Q35} に含まれる文を指す．

補スライス S^*_{P12} および S^*_{Q35} を変形する様子を図 5.22に示す．N はヌル節点を指す．変形後の補スライス $S^*_{P12}{}'$ と $S^*_{Q35}{}'$ はラベル付き同形写像比較において一致し，S_{P12}, S_{Q35} は定義 5.1を満たすので，$\langle S_{P12}, S_{Q35} \rangle$ が交換可能スライス対となる．図 5.23に P, Q から得られる交換可能スライス $\langle S_{P12}, S_{Q35} \rangle$ を示す．ラベル付き同形写像比較により，補スライスどうしの節点の対応関係 F_c は図 5.23に示すように求まる．

次に，P, Q で同時に利用されている変数 ch, digit についてスライスを作成する．変数

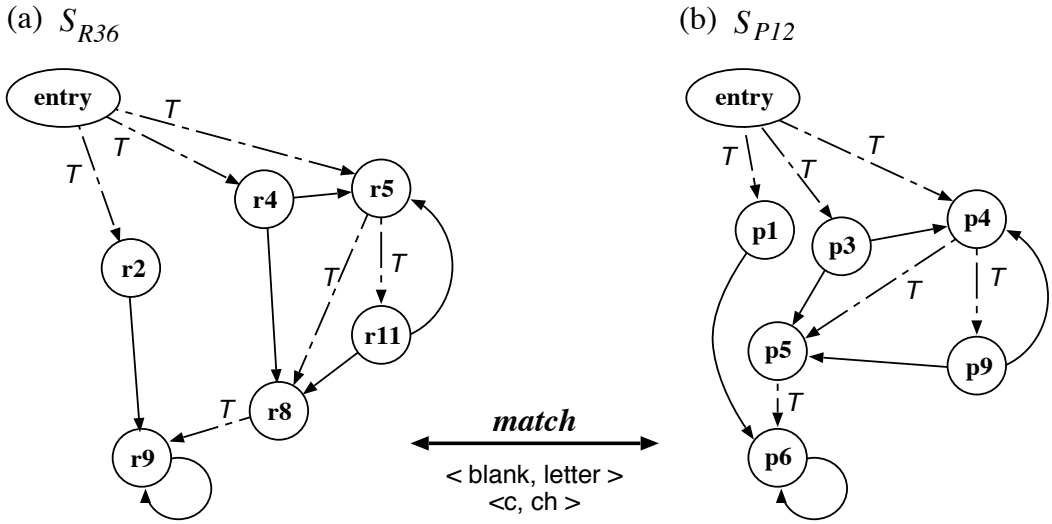
```
     procedure TextFormat.count_rate;
     var c: char
         line: integer;
         blank,other: integer;
         rate: real;
     begin
 r1    line := 1;
*r2    blank := 0;
 r3    others := 0;
*r4    read(c);
*r5    while c<> EOF do
         begin
 r6        if c = CR then
 r7          line := line + 1;
*r8        if (c = ' ') or (c = TAB) then
*r9          blank := blank + 1
           else
 r10         others := others + 1;
*r11       read(c)
         end;
 r12   writeLn(line);
 r13   writeLn(blank);
 r14   writeLn(others);
 r15   rate := blank / line;
 r16   writeLn(rate)
     end.
```

図 5.24: 機能交換対象のソースコード R

ch のスライスがそれぞれ S_{P12}, S_{Q35} に包含されるので，包含されるスライスどうしのラベル付き同形写像比較により，節点の対応関係 F_s は図 5.23に示すように求まる．さらに， F_c, F_s を用いると， entry $\rightarrow_c$ p1 $\rightarrow_d$ p10, entry $\rightarrow_c$ q13 $\rightarrow_d$ q14 より，図 5.23の対応関係 F_t が求まる．交換可能スライス対 $\langle S_{P12}, S_{Q35} \rangle$ の節点の対応関係は $F_{PQ} = F_s \cup F_t$ となる． F_c, F_s, F_t において，対応する節点のラベルは同じである．網かけの節点は変更箇所を表す．

(2) 機能の交換

能動的部品 *TextFormat* が，機能分割変化により図 5.24に示すソースコード R に変化した際に， *Histogram* により広告された交換可能スライス対 $\langle S_{P12}, S_{Q35} \rangle$, F_{PQ} を感知したときを考える． R は「入力されたテキストに対して，スペースおよびタブ，その他の文字の出現回数，テキストの行数，一行当たりのスペースおよびタブの割合を求める」機能を持



matching nodes F_{R36P12}

<r2, p1>	{ "blank := 0", "letter := 0" }
<r4, p3>	{ "read(c)", "read(ch)" }
<r5, p4>	{ "c <> EOF", "ch <> EOF" }
<r8, p5>	{ "c = ' '", "ch>='A' and ch<='Z'"}, { "c = TAB", "ch>='a' and ch<='z'" }
<r9, p6>	{ "blank := blank + 1", "letter := letter + 1"}
<r11, p9>	{ "read(c)", "read(ch)" }

nodes to be replaced in F_{R36P12} *and* F_{P12Q35}

r2 ← p1 ← q13	**r5 ← p4 ← q5**	**r8 ← p5 ← null**
r4 ← p3 ← q4	**r11 ← p9 ← q12**	**r9 ← p6 ← null**

図 5.25: スライスどうしの抽象化ラベル付き同形写像比較

つ．ここでは，以降の説明の例として $S_{R36} = \hat{S}_b(\mathrm{r1}, \mathrm{r13}, \{\mathrm{blank}\})$ を取り上げる．図 5.24において，識別子の左側の $*$ は S_{R36} に含まれる文を指す．

スライスどうしのラベル付き同形写像比較の様子を図 5.25に示す． S_{P12} と S_{R36} はラベルを抽象化すると，ラベル付き同形写像比較で一致するので，スライス S_{R36} が交換対象スライスになる． S_{P12} と S_{R36} の節点，ラベル，変数の対応関係を図 5.25に示す．

ソースコード R の PDG G_R の節点を置換する． 1段目の置換は対応関係 F_{RP} を用い，節点を r2 ← p1, r4 ← p3, r5 ← p4, r8 ← p5, r9 ← p6, r11 ← p9 と置き換える． 2段目の置換は対応関係 F_{PQ} を用い，節点を p1 ← q13, p3 ← q4, p4 ← q5, p9 ← q12 と置き換え

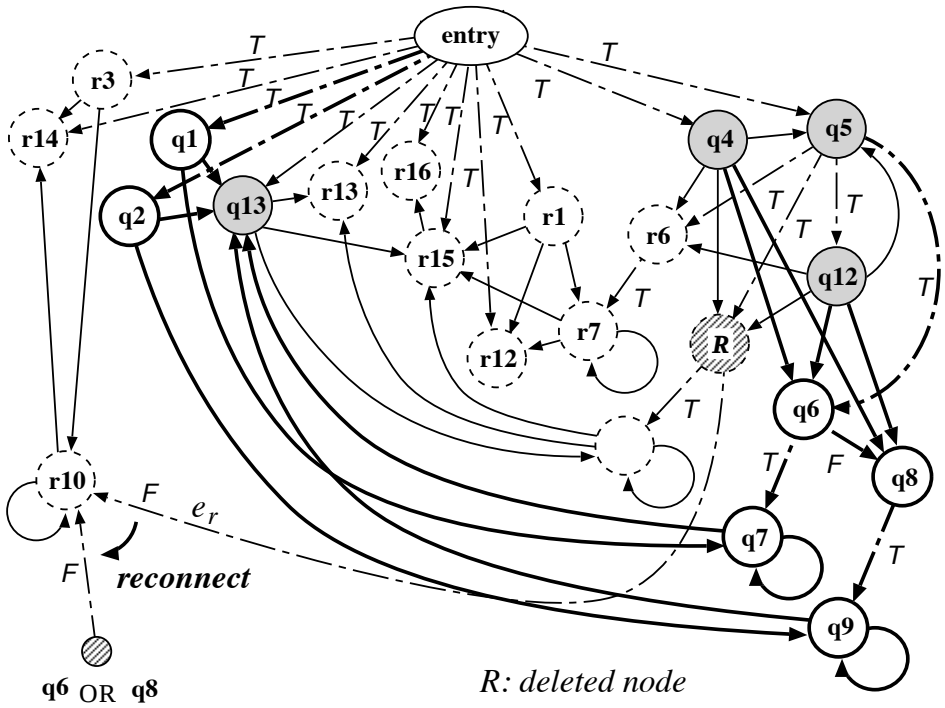


図 5.26: 合成後の PDG($S_{Q35} + S^*_{R36}$)

る．節点 p5, p6 は，対応する節点が F_{PQ} に存在しないのでヌル節点に置換する．

次に，交換スライス S_{Q35} と無交換部分に対応する部分グラフ S^*_{R36} の合成を行う．合成後の PDG を図 5.26に示す．図 5.26において，網かけは置換後の節点を表す．太線は S_{Q35} を重ねることで追加された節点および矢印を表す．また，$e_r = R \rightarrow^F_c$ r10 が付けかえ対象の矢印である．図 5.26の例では，$ID(R) =$ { q5 }, $PD(R) =$ { q5, q6, q8 }，矢印の属性が偽 (F: false) より，条件分岐節点 { q6, q8 } が付けかえ候補節点集合となる．

節点 r10 をそれぞれ q6, q8 に接続した時に生成されるソースコード R'_1, R'_2 を図 5.27, 5.28 に示す．識別子の左側の + は機能交換変化により自動変更された節点を指す．R'_1 は「入力されたテキストに対して，スペースとタブを区別し，スペース，タブ，スペース以外の文字の出現回数，テキストの行数，一行当たりのスペースおよびタブの割合を求める」機能を持つ．また，R'_2 は「入力されたテキストに対して，スペースとタブを区別し，スペース，タブ，スペースおよびタブ以外の文字の出現回数，テキストの行数，一行当たりのスペースおよびタブの割合を求める」機能を持つ．

図 5.27の R'_1 と，図 5.28の R'_2 は，変数 others の実行結果に関して違いを持つ．R, R'_1,

```
     procedure TextFormat.count_rate_1;
     var c: char;
         line,blank,others: integer;
         uppercase: integer;
         lowercase: integer;
         rate: real;
     begin
 r1    line := 1;
 r3    others := 0;
+q4    read(c);
+q1    uppercase := 0;
+q2    lowercase := 0;
+q5    while c <> EOF do
         begin
 r6        if c = CR then
 r7          line := line + 1;
+q6        if c = ' ' then
+q7          uppercase := uppercase + 1
           else
             begin
 r10           others := others + 1;
+q8            if c = TAB then
+q9              lowercase := lowercase + 1
             end;
+q12       read(c)
         end;
 r12   writeLn(line);
+q13   blank := uppercase + lowercase;
 r13   writeLn(blank);
 r14   writeLn(others);
 r15   rate := blank / line;
 r16   writeLn(rate)
     end.
```

図 5.27: 変化後のソースコードの例 R'_1 (節点 q6 への結合時)

R'_2 において，変数 line の実行結果は等価である．これは，「テキストの行数を求める」機能は保存されたままで合成が行われたことを指す．また， R, R'_1, R'_2 における変数 blank, rate の計算手続きより，「大文字と小文字を区別する」という P から Q への変更が，「スペースとタブを区別する」という形で R に反映されているのがわかる．

本例の機能交換変化において，能動的部品が生成する各種スライス対，新規ソースコードの数を表 5.1に示す．変更事例を持つ能動的部品 *Histogram* は， 6 個の交換可能スライス対をライブラリに広告する．この変更事例を感知した能動的部品 *TextFormat* は，ソースコー

```
     procedure TextFormat.count_rate_2;
     var c: char;
         line,blank,others: integer;
         uppercase: integer;
         lowercase: integer;
         rate: real;
     begin
 r1    line := 1;
 r3    others := 0;
+q4    read(c);
+q1    uppercase := 0;
+q2    lowercase := 0;
+q5    while c <> EOF do
         begin
 r6        if c = CR then
 r7          line := line + 1;
+q6        if c = ' ' then
+q7          uppercase := uppercase + 1
+q8        else if c = TAB then
+q9          lowercase:= lowercase + 1
           else
 r10         others := others + 1;
+q12       read(c)
         end;
 r12   writeLn(line);
+q13   blank := uppercase + lowercase;
 r13   writeLn(blank);
 r14   writeLn(others);
 r15   rate := blank / line;
 r16   writeLn(rate)
     end.
```

図 5.28: 変化後のソースコードの例 R'_2 (節点 q8 への結合時)

ド R からそれぞれ 2 個 (付けかえ節点の数)，全部で 12 個の新規ソースコードを生成する．表 5.2に生成した 12 個のソースコードの機能を示す．これらのソースコードは，変数 others の実行結果，変数 uppercase, lowercase の初期化，変数 blank の出力結果に関してそれぞれ機能の違いを持つ．

5.6.3 プロトタイプによる変化の様子

5.5節で示した提案アルゴリズムに基づき能動的部品のプロトタイプを構築し，実際に能動的部品の動作確認を行った．能動的部品の変化の様子を図 5.29, 5.30, 5.31, 5.32に示す．

表 5.1: 生成したスライス対およびソースコードの数

スライス対 / ソースコード	*Histogram*	*TextFormat*
新規ソースコード (Cde)	5	1
変更前スライス対 $\langle S_{Pi}, S^*_{Pi}\rangle$	20	—
変更後スライス対 $\langle S_{Qj}, S^*_{Qj}\rangle$	56	—
交換可能スライス対 $\langle S_P, S_Q\rangle$	6	—
交換対象スライス対 $\langle S_{Rk}, S^*_{Rk}\rangle$	—	51
新規ソースコード (Cex)	—	12

表 5.2: 生成したソースコードの機能

No.	ソースコードの機能
P	英字と数字の出現回数を求める
Q	P において，英字の大文字と小文字を区別する
R	スペースおよびタブ，その他の文字の出現頻度， 行数，一行当たりの割合を求める
R'_1	R において，スペースとタブを区別し， スペース，タブ，スペース以外の文字の出現回数を求める
R'_2	R において，スペースとタブを区別し， スペース，タブ，スペースおよびタブ以外の文字の出現頻度を求める
R'_3	R'_1 において， uppercase の初期値が入力引数
R'_4	R'_2 において， uppercase の初期値が入力引数
R'_5	R'_1 において， uppercase, lowercase の初期値が入力引数
R'_6	R'_2 において， uppercase, lowercase の初期値が入力引数
R'_7	R'_1 において， blank の出力結果が常に初期値
R'_8	R'_2 において， blank の出力結果が常に初期値
R'_9	R'_1 において， uppercase の初期値が入力引数， blank の出力結果が常に初期値
R'_{10}	R'_2 において， uppercase の初期値が入力引数， blank の出力結果が常に初期値
R'_{11}	R'_1 において， uppercase, lowercase の初期値が入力引数， blank の出力結果が常に初期値
R'_{12}	R'_2 において， uppercase, lowercase の初期値が入力引数， blank の出力結果が常に初期値

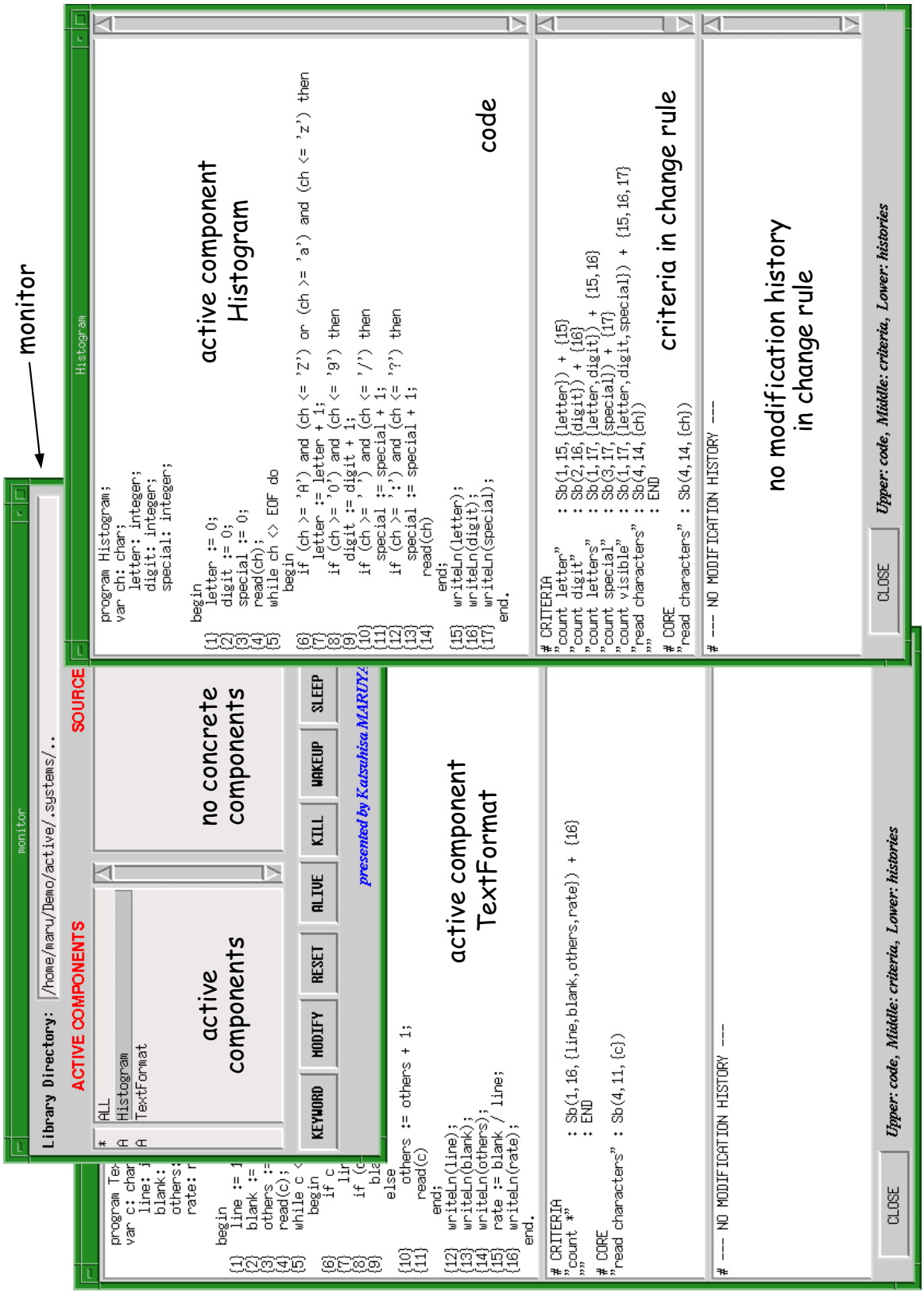


図 5.29: 能動的部品の動作 (初期状態)

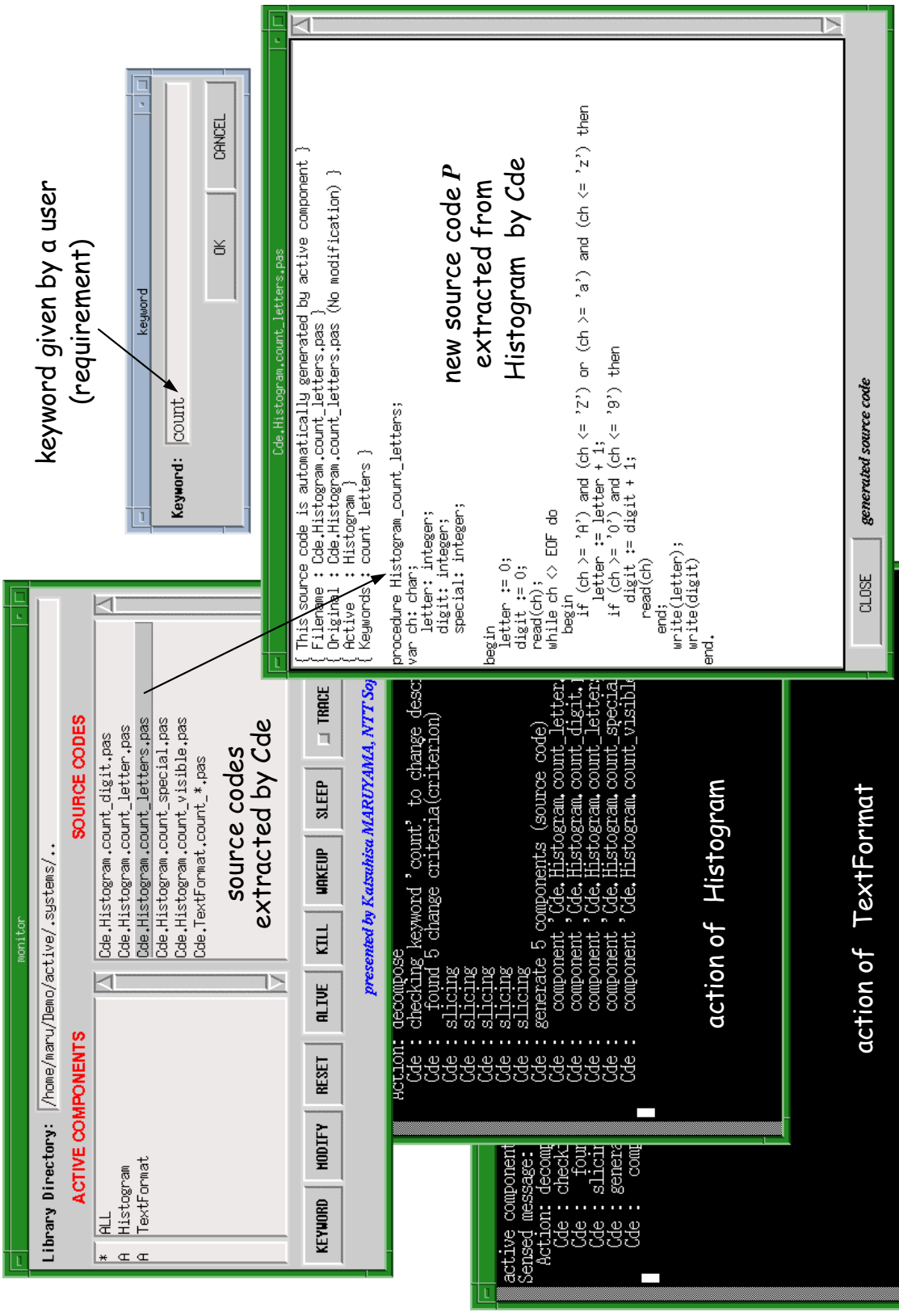


図 5.30: 能動的部品の動作 (機能分割変化 Cde)

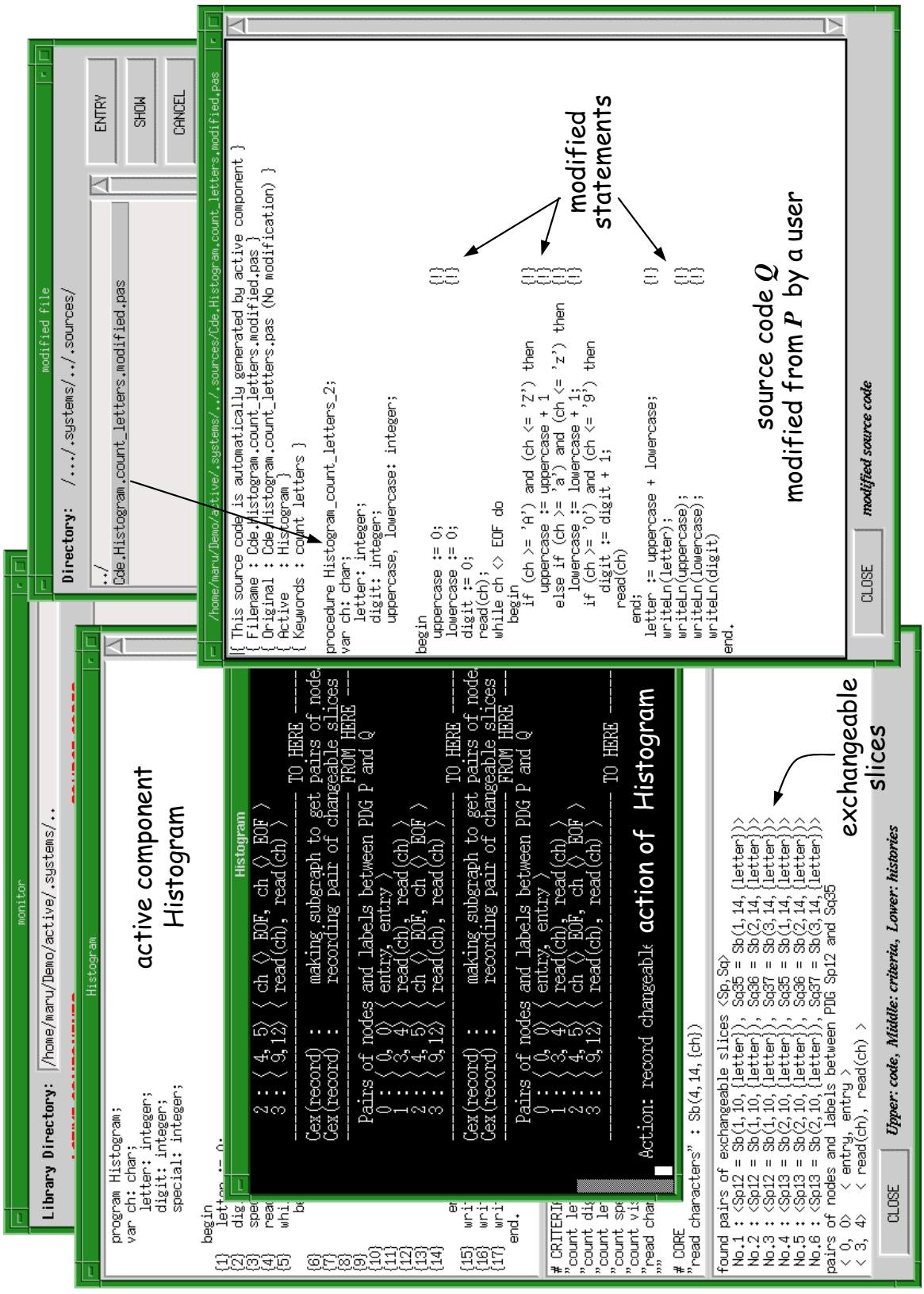


図 5.31: 能動的部品の動作 (生成ソースコードの変更 $P \rightarrow Q$)

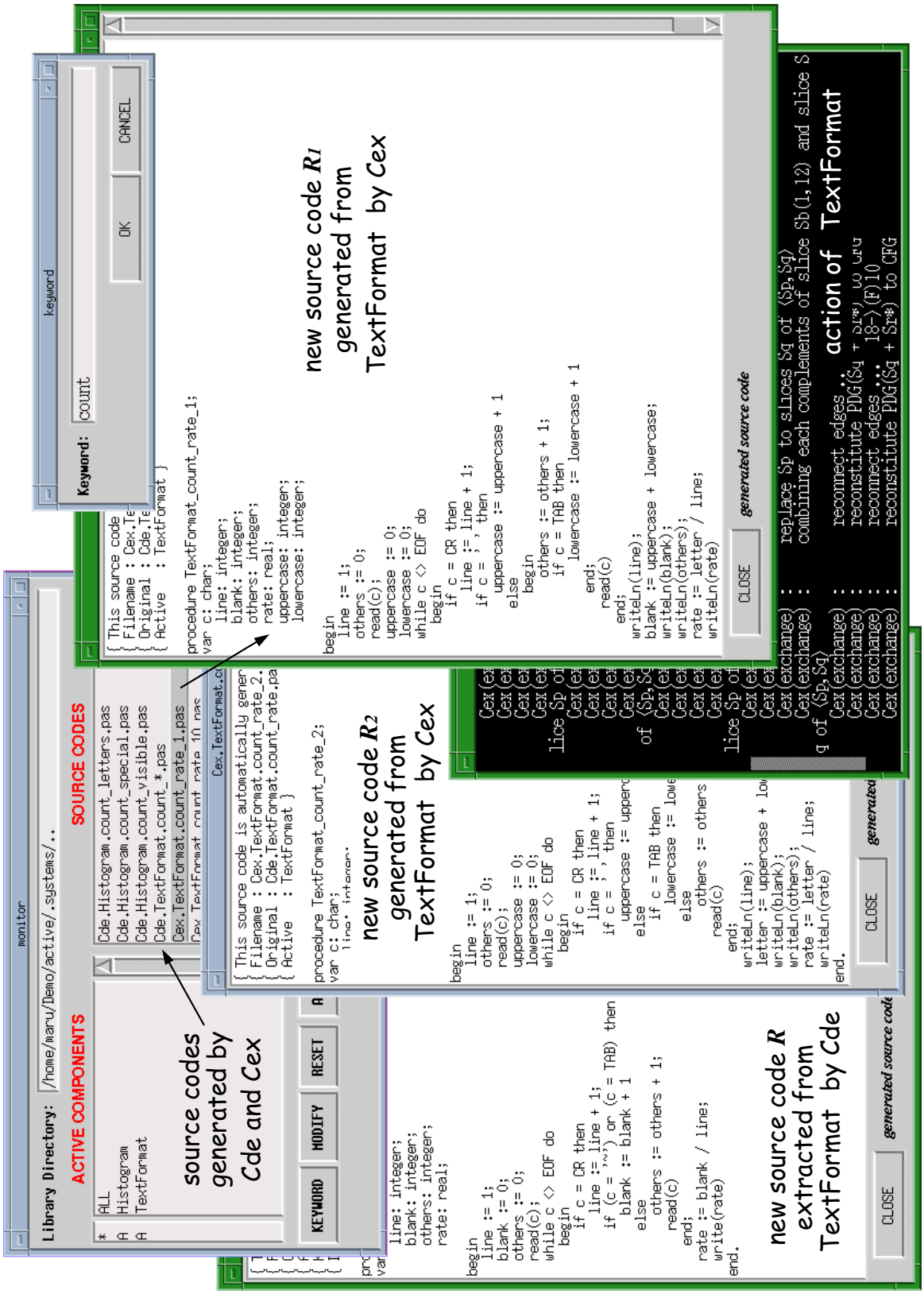


図 5.32: 能動的部品の動作 (機能分割変化 + 機能交換変化)

図 5.29において，能動的部品 *Histogram* と *TextFormat* は変更履歴を持っていない．この状態で，図 5.30に示すように， *Histogram* と *TextFormat* に要求キーワード `"count"`を与えると， 2 つの能動的部品はそれぞれ機能分割変化を行い，それぞれ 5, 1 個のソースコードを生成した．次に，図 5.31では， *Histogram* から生成されたソースコード P に対して，変更ソースコード Q を作成し， Q の所在を *Histogram* に通知する．図 5.31より， *Histogram* の変化規則に交換可能スライス対が記録されているのがわかる．この状態で，図 5.32に示すように， *Histogram* と *TextFormat* に再び要求キーワード `"count"`を与えると， *TextFormat* は *Histogram* により広告された変更差異を適用して機能交換変化を行い， 12 個のソースコードを生成した．能動的部品のプロトタイプが生成したソースコードは，前述のアルゴリズムの適用例と同じものであった．

5.7 考察

機能分割変化および機能交換変化メカニズムを備えた能動的部品は，スライシング基準を変化規則や依存関係矢印の接続の有無から自動的に決定し，区間限定スライスとその補スライスを作成する．さらに，ラベルの抽象化とラベル付き同形写像比較により，変更の影響を受けた範囲および機能を交換する範囲を自動的に特定できる．能動的部品は再利用される環境に応じてさまざまな変更を受けるので，同じ能動的部品でも変更事例はそれぞれ異なる．よって，変更事例はプログラム開発ドメインの特性の一部を担い，能動的部品は自分の存在する環境の特性を動的に取得し，自分の変化パターンを決定可能である．

このように，能動的部品は機能分割変化と機能交換変化を備えることで，部品検索時に要求や環境に応じて自動的かつ動的に変化する能力を持つ．この点で，能動的部品は，作成時に機能が固定される従来の部品と大きく異なり，ソースコード再利用に対して，次に示す効果が期待できる．

効果 1 部品ライブラリに存在しない部品を要求した場合においても，類似部品に対する変更が自動で行われるため，変化後の部品が部品利用者の要求を満たす場合には，部品利用者が部品を変更する負担が軽減する．例えば， 5.6節の適用例において，表 5.2に示す 12 個のソースコードは自動的に生成される．よって，部品利用者が「英字の大文字と小文字を区別する」という機能変更を，既存の類似部品ごとに行う手間を削減できる．

効果 2 環境を動的に取得することより，ライブラリ内に多種多様な部品をあらかじめ用意

しておく必要がなく，特性分析が困難な開発ドメインでも，環境に応じた部品を利用者に提供できる．例えば，5.6 節の適用例において，部品作成者が用意した 2 つの能動的部品から，環境に応じた表 5.2に示す 12 個のソースコードが動的に生成される．また，変更を積み重ねるたびに，能動的部品に過去の変更事例が蓄積され，開発ドメインに適した部品に変化可能となる．

実際のソースコード再利用において，能動的部品の効果を得るために，以下に示すアルゴリズムの拡張を試みている．

拡張 1 現在の変化メカニズムは，手続き呼出しを含まない，制御構造が単純である，配列型およびポインタ型変数を扱わない等，制限されたプログラミング言語において定義されている．これらの制限は，主に依存解析能力の低さに起因する．言語の制限を取り除くために，文献 [9, 38, 39, 105] の手法を本変化メカニズムに組み込むことを行っている．

拡張 2 機能交換変化において，生成されたスライスおよび補スライスどうしのすべての組合せに対して，ラベル付き同形写像比較を行うことは時間的およびリソース的観点から現実的でない．例えば，5.6 節の適用例において，$20\times56+6\times51 = 1426$ 回の同形写像比較が行われる．また，変更の種類によっては，大量の交換可能スライス対が記録され，生成されるソースコードが爆発的に増加する可能性がある．よって，要求に無関係なスライスの生成を抑える手法や，変化によって生成された部品から適切な部品を選択する手法を提供する必要がある．これに対して，能動的部品のソースコードに論理表明を付加し，生成した各スライスの機能を論理式で表す [55, 56, 86]．これにより，要求を満たす可能性のあるスライスだけを同形写像比較の対象とし，比較回数を削減することができる．また，生成したソースコードの意味を明確にすることや，生成した新規ソースコードから要求を満たすものだけを選択することが可能となる．

5.8 あとがき

ソースコード再利用において，従来とまったく異なる新しい部品として，部品検索時に自ら形を変える能動的部品を提唱した．本章では，能動的部品の機能分割変化および機能交換変化のメカニズムを明確にし，その適用例を示した．部品変化メカニズムは，プログラムスライシングとグラフのラベル付き同形写像比較により実現可能である．

機能分割変化および機能交換変化を備えた能動的部品は，要求に応じて自分自身を機能分割する．また，他の部品が受けた変更を模倣するために，自分の周りに存在する他の能動的部品の変更事例を取得し，機能の一部を他の部品と交換する．機能分割と機能交換により，能動的部品は要求や環境に応じて自動的かつ動的に変化可能となる．よって，部品利用者の部品変更に対する負担を軽減でき，特性分析が困難な開発ドメインでも環境に応じた部品を提供可能であるという効果を持つ．

第 6 章

メソッド合成による新規クラスの自動生成

6.1 はしがき

オブジェクト指向プログラミングでは，継承 (inheritance) を用いることで，既存のクラスから派生させた後継者クラスにおいて，開発者がメソッドや変数を追加および再定義するだけで，要求するプログラムを作成することが可能である [45, 78, 93]．継承により，ライブラリ内に存在する既存クラスの再利用性は高くなり，開発者が同じコードを繰り返し記述することが避けられる．

しかしながら，後継者クラスで再定義したメソッドが派生元である親クラスのメソッドと関係を持つことがあるため，開発者は再定義メソッドの変更部分が他の部分へ与える影響を十分に考慮してコード変更を行う必要があり，その変更は容易であるとはいえない．さらに，継承における再定義の最小単位はメソッドである．このため，要求する機能が既存クラスで定義されているメソッドの持つ機能と微妙に異なる場合，例えば，メソッド引数の 1 つの入力条件だけが異なる場合でも，開発者はメソッド全体を再定義することになり，コード変更が要求される場面は多い．このように，継承により既存クラスを再利用できる機会は増加するが，再利用時のコード変更に関する開発者の負担はそれほど軽減していない．

このような問題に対して，本研究では，開発者がオブジェクト生成 (instantiation) とメソッド呼出し (メッセージ送信， message passing) のコードを記述するだけでプログラムを作成できる手法の検討を進めてきた [67, 68, 69, 71]．本章では，クラス変更の履歴を積極的に利用することで継承におけるコード変更を自動化し，ライブラリ内に存在する既存クラスから開発者の要求する機能を満たす可能性の高い新規クラスを自動生成する手法を提案する．本クラス生成手法は，1）新規クラスを生成する際に，過去のクラス変更を模倣する，2）独立に変更された 2 つのプログラムを，依存関係に関する無矛盾性を保ちながら合成可

能なプログラム合成アルゴリズム [40] に基づく，という点が特徴である．ここで，本章で扱う変更は，メソッドや変数の追加および再定義というクラス内部のコードだけを書き換えることを指し，クラス階層の再構成を含まない．

過去のクラス変更を模倣するため，開発者が既存クラスを変更した際に，その変更の履歴を既存クラスに蓄積する．次に，開発者が再利用しようとした参照クラスにおいて呼出しメソッドが定義されていないとき，過去において類似クラスが受けた変更の履歴を参照クラスに適用する．これらの動作は，5章で提案した能動的部品の機能交換変化メカニズムを拡張することで実現可能である．本クラス生成手法をオブジェクト指向プログラム開発に導入することで，継承におけるメソッドや変数の追加および再定義というコード変更に関する開発者の負担の軽減が期待でき，開発者が容易にプログラムを作成可能となる．

本章の構成は次の通りである．6.2節では，新規クラス生成手法の概要を示す．6.3節では，本生成手法を実現する準備として，オブジェクト指向プログラムのスライシングと同形写像比較を述べる．6.4節では，継承の特性によりクラス生成時に生じる問題を示し，この問題を回避するメソッド複写を提案する．6.5節で本生成手法を実現する 3 つの手続きを，6.6節で新規クラスの生成例を示す．6.7節では，本生成手法を分散ネットワーク環境に適用する方式を紹介する．6.8節で，本生成手法に関する利点および限界を考察する．

6.2 新規クラス生成手法

新規クラスの自動生成を実現するにあたり，オブジェクト指向プログラム開発において，既存クラスに対する過去の変更は類似のクラスに対しても同様に行われる可能性が高いという仮定をおく．この仮定は，開発者が新規のプログラムを作成する際に，類似のプログラムに対して過去に行った変更を模倣することから導かれ，新規クラスの生成という目的において妥当である．このような仮定に基づき，本クラス生成手法は，たとえ開発者の要求するクラスがライブラリ内に存在しない場合でも，開発者の要求を満たす可能性の高いクラス候補を開発者に提供する．本生成手法では，開発者の記述したメッセージ送信に対して，そのメッセージを処理可能なメソッドを持つクラスを，開発者の要求を満たすクラスとみなし，要求をメソッドインターフェイスで表現する．メソッドインターフェイス[1]とは，メソッドの名前およびシグニチャ (戻り値の型および引数の数と型) を指す．

類似クラスにおける過去の変更を模倣することで，開発者の要求を満たす新規クラスを自動生成する手法は，次に示す仕組みにより実現可能である．

[1] メソッドシグニチャとも呼ばれる．

(1) 親クラスとそのクラスから派生した後継者クラスの機能に関する変更差異を特定し，その差異を変更履歴として蓄積する仕組み．

(2) 開発者が参照している既存クラスに対して，類似な機能を持つ類似クラスを見つける仕組み．

(3) 類似クラスの持つ履歴に含まれる変更差異を，開発者の参照している既存クラスの機能に組み込む仕組み．

機能とは，メソッドにおいて，特定の入力変数の値から着目する出力変数の値を計算する際の動作を指す．(1) と (3) の仕組みを，能動的部品の機能交換変化メカニズムにより実現する．この際，オブジェクト指向プログラムから新規クラスを生成できるようにするため，手続き型プログラムから新規部品を生成する従来の能動的部品における合成アルゴリズムに対して，次の 2 つの改良を加えた．1）オブジェクト指向プログラム対応の区間限定スライシングを用いて変更コードを抽出する，2）生成した新規クラスにおいて，呼出しメソッドの同一性と参照可能性を保証するためにメソッドを複写する．

また，(2) の仕組みを，単一継承により階層化されたライブラリにおけるクラスの位置とメソッドインターフェイスの同一性に基づくクラス探索手続きにより実現する．ここで，2 つのクラスが類似であるとは，それらのクラスが祖先から継承したメソッドや変数を数多く共有していることを指す．

クラス自動生成手法の概要を図 6.1に示す．四角形はクラスを表す．楕円は上記の 3 つの仕組みをそれぞれ実現する，1) 変更差異特定部 (Specifier)，2) 類似クラス探索部 (Finder)，3) 変更差異合成部 (Integrator) である．

開発者が，継承により親クラス (*Parent*) から後継者クラス (*Heir*) を派生させ，メソッドや変数の追加および再定義を行った場合を考える．変更差異特定部は，*Perent* と *Heir* のメソッドから変更差異を抽出し，この差異を *Parent* の履歴 (history) に蓄積する．次に，開発中のクラス (*Client*) から呼び出されたメソッド M が，参照クラス (*Referenced*) において定義されていない場合を考える．参照クラスとは，メッセージの受信先として指定されたインスタンス (オブジェクト) の生成元クラスを指す．実際のプログラミングにおいて，開発者が要求だけから，*Referenced* に存在しないメソッド M を呼び出す文を *Client* に記述することは少ない．しかしながら，開発者が *Referenced* の変更を前提に開発を進めている場合，他のクラスに存在するメソッドの名前と機能を参考にして，*Client* から呼び出すメソッドのインターフェイスを (呼び出されるメソッドの実体を記述する前に) 決定しておくことは

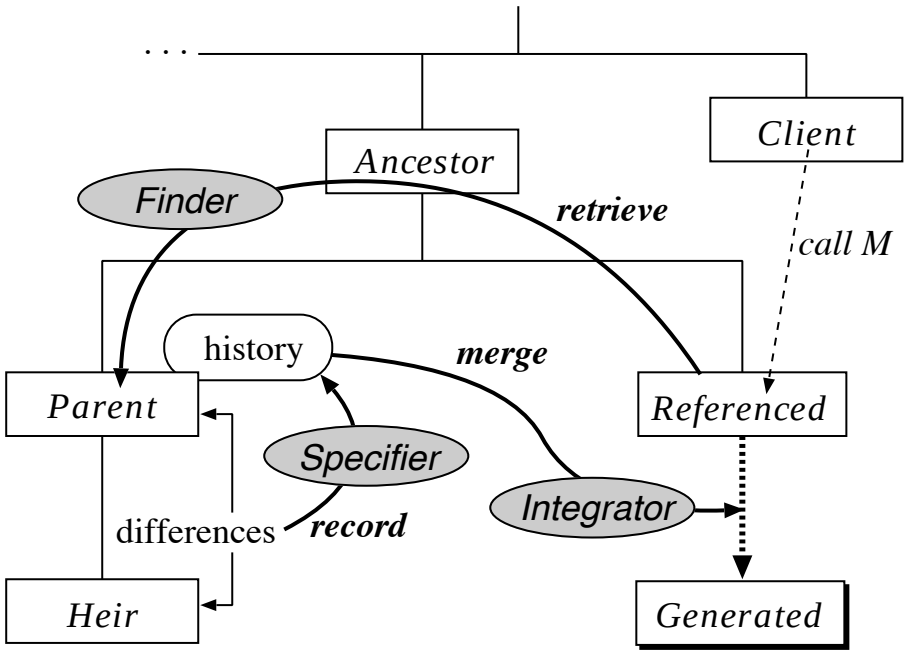


図 6.1: クラス生成手法の概要

多い．よって，このようなメソッド呼出し文をソースコード内に記述することを奨励した場合，その記述は多いと考えられる．類似クラス探索部は，ライブラリの階層関係をたどることにより類似クラス (この例では， *Parent*) を見つける．変更差異合成部は， *Referenced* に類似クラスの持つ変更差異の一部を組み込むことで，新規クラス (*Generated*) を生成する．

このように，本生成手法では，開発者が親クラス *Parent* から後継者クラス *Heir* を派生させた際の変更を模倣することで，既存の参照クラス *Referenced* から新規クラス *Generated* を生成する． *Generated* は，開発者の要求したメソッド M と， M に関連するために変更の影響を受けたメソッドを持つ．開発者は， *Generated* を無変更あるいは少ない変更により再利用することで，作成中のクラス *Client* を実行可能である．

6.3 クラス生成手法における諸定義

本節では，まず，メソッドの合成を実現するための準備として，クラス依存グラフについて述べ，このグラフを用いた区間限定スライシングおよび同形写像比較を定義する．ここで，本クラス生成手法は，型宣言のある静的型推論機構を持つ単一継承のオブジェクト指向言語を扱う．本章では， Java 言語 [35] の記法を用いて，ソースコードを記述する．

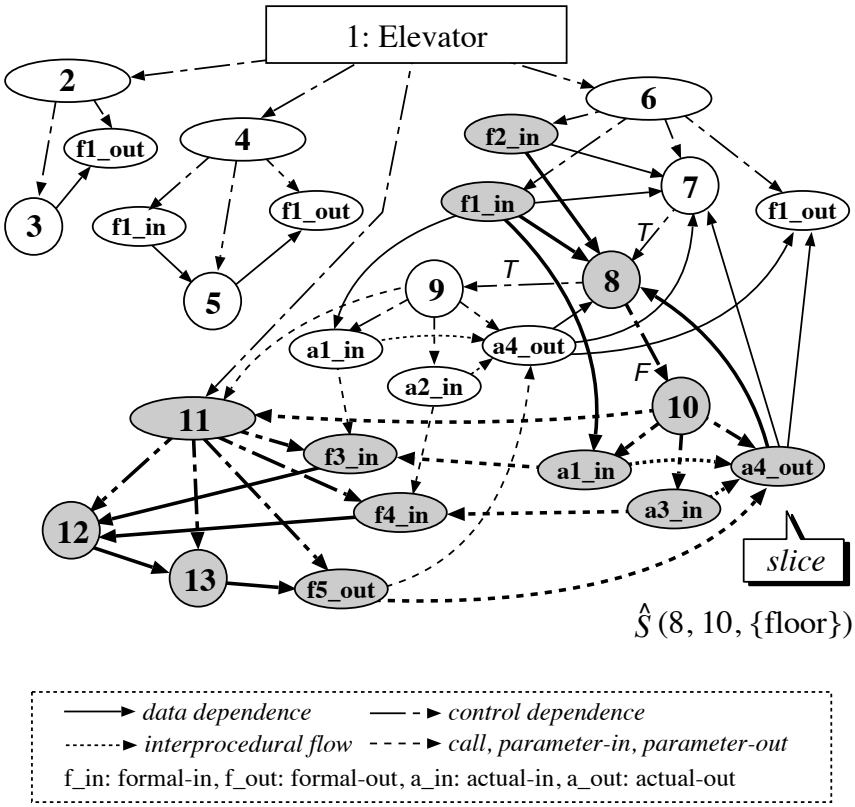
6.3.1 クラス依存グラフ

オブジェクト指向プログラムのクラス内部のコードを抽出，比較，合成する際，プログラム依存グラフ (PDG) [29] を拡張したクラス依存グラフ (class dependence graph: ClDG) [94, 53] を用いる．ClDG において，クラス内の各メソッドは手続き依存グラフ (procedure dependence graph [29, 38] あるいは interprocedural program dependence graph: IPDG [94]) で表現し，手続き型プログラムにおける関数とみなす．ClDG のデータ依存関係は，各メソッドごとに作成した CFG を集めたクラス制御フローグラフ (class control flow graph: CCFG) [37] に対して，メソッド内部の定義 – 参照関係 (intra-method define-use pair) [90]，メソッド間の定義 – 参照関係 (inter-method define-use pair) [90]，クラス内部の定義 – 参照関係 (intra-class define-use pair) [36] に関するデータフローを解析することで得られる．さらに，ClDG は，メソッド呼出しの引数やインスタンス変数 (および global 変数) の受け渡しを担う入出力変数節点 (formal-in 節点, actual-in 節点, formal-out 節点, actual-out 節点) とそれらの節点間のデータ依存関係 (parameter-in, parameter-out dependence)，メソッド呼出し関係 (call dependence)，および呼出しメソッドにおける引数のデータ依存関係 (interprocedural flow dependence) を表す矢印を含む．これらの節点および矢印は，システム依存グラフ (SDG) [38] に存在するものと同じである．

ライブラリに存在するクラスのメソッドに関しては，あらかじめ内部の依存解析を行い，メソッド引数に関するデータ依存関係を求めておく．本生成手法では，スライシングおよび同形写像比較における計算量を抑えるため，これら解析済みメソッドの内部コードを ClDG において展開せず，メソッド呼出し節点に関する依存関係に置き換える．

6.3.2 オブジェクト指向プログラム対応の区間限定スライシング

本クラス生成手法では，オブジェクト指向プログラムに対して，ClDG を用いる文献 [53] のスライシング技法を採用する．この技法では，2.3節で述べた手続き呼び出しを含むプログラムに対するスライシングアルゴリズム [38, 53] を用い，ClDG に対する 2 段階の経路探索によりスライスを求める．さらに，本生成手法では，3.3節で定義した制約付き到達可能経路を用いて，従来のオブジェクト指向プログラム対応のスライシング技法を対象区間を指定できるように拡張する．本章において，区間限定スライス (bounded slice) $\hat{S}(n_u, n_l, V)$ とは，節点 n_u, n_l によって指定された区間内において，着目する変数 v $(\in V)$ に関して依存関係をたどることによって得られる静的スライスである．オブジェクト指向プログラム対応の区間限定スライシングは，任意のプログラムの同一のメソッド内において n_u と n_l を自




```
1 class Elevator {
    int floor;
2   Elevator() {
3     floor = 1;
    }
4   public int which_floor() {
5      return floor;
    }
6   public void go(int f) {
7     while (floor != f) {
8       if (floor < f)
9         floor = add(floor, 1);
        else
10        floor = add(floor, -1);
      }
    }
11  private int add(int a, int b) {
12      int c = a + b;
13      return c;
    }
  }
```

CRP (8, 10)
= {8, 10} + { add }
= {8, 10, 11, 12, 13}

f1_in: floor = floor_in
f1_out: floor_out = floor
f2_in: f = f_in
f3_in: a = a_in
f4_in: b = b_in
f5_out: c_out = c
a1_in: a_in = floor
a2_in: b_in = 1
a3_in: b_in = -1
a4_out: floor = c_out

図 6.2: クラス依存グラフと区間限定スライスの例

由に設定でき，指定した区間内の依存関係だけに基づきスライスを抽出可能であるという利点を持つ．よって，このような区間限定スライシングをプログラム合成に導入することで，より多くの変更差異を特定および合成可能である．本章では，オブジェクト指向プログラムの区間限定スライスを単にスライスと呼ぶ．

ClDG と区間限定スライスの例を図 6.2に示す．網掛け節点および太矢印が，節点 10 を実行した後の変数 `floor` の値に影響を与えるスライス $\hat{S}(8, 10, \{\texttt{floor}\})$ である．

6.3.3 クラス依存グラフの同形写像比較

本生成手法では，2 つのプログラムの機能に関する等価性を判定するために，2.4節で述べたグラフの同形写像比較を用いる．この技法では，比較する 2 つの ClDG において，一方の ClDG の節点と矢印を他方の ClDG に写像することで，同形を判断する．

ここで，同形写像比較において，同一のクラスから生成したインスタンスは同じメソッドや変数を持つので，その機能は等価であるとみなせる (同一の基本データ型から生成したインスタンスも，その機能は等価であるとみなす)．しかしながら，実際に比較を行う際には，インスタンスを参照する変数名の違いによって，同形写像比較におけるラベルの等価性 (2.4節のラベルに関する条件 (4)) が満たされないことがある．さらに，変数名が同じでも，その変数の指しているインスタンスの機能が異なる場合がある．

そこで，機能の等価性を判定可能とするために，比較前に ClDG の各節点のラベルにおける変数名を型に基づき置換する．まず，比較する 2 つの ClDG に対して，型が一致する変数名の対を置換変数の候補として求める．このとき，比較対象のクラス (ClDG) に宣言を持たない変数については，その変数が指すインスタンスが同一機能の場合に必ず同じ変数名で参照されるため，置換対象候補としない．また，比較対象のクラスに型が異なる同名の変数が存在する場合，クラス名を変数名の前に付加することで，それぞれ別の変数名に変更する．次に，置換変数の候補に基づき一方の ClDG において変数の置換を行った際，2 つの ClDG の各節点のラベルが一致するかどうかを検査する． ClDG のすべての節点のラベルにおいて成立する置換変数の対応関係 φ_v が存在するとき，実際に変数名を対応関係 φ_v に基づき置換する．以上より，本章では，2.4節の条件 (4) を次のように再定義する．

(4) G_1 の各節点 p に対して， φ_v により変数名を置換した p のラベルと G_2 内の節点 $\varphi(p)$ のラベルが等しい．

変数名を置換する様子を図 6.3に示す．クラス A あるいは B で宣言される変数に対して，型に基づき置換変数の対応関係の候補を求めると， $\varphi_v = \{\langle \texttt{gr}, \texttt{g}\rangle, \langle \texttt{pen}, \texttt{p}\rangle\}$ となる．これ

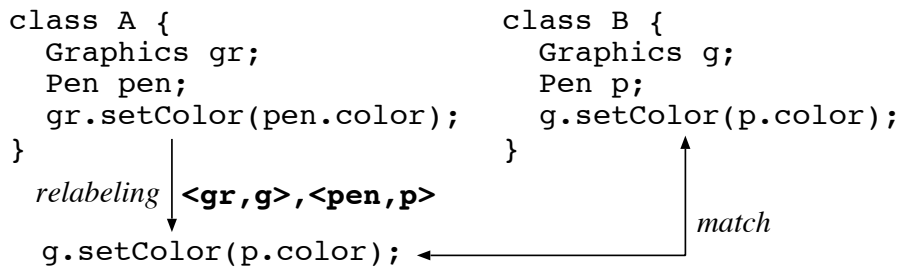


図 6.3: 変数名の置換

らの候補に従ってクラス A において実際に置換を行った場合，2 つのラベルは一致する．変数 color については，比較するクラス A および B のどちらでも宣言されていないため置換対象としない．

6.4 メソッドの複写

親クラスと後継者クラスから求めた変更差異を単純に参照クラスに合成すると，オブジェクト指向の持つ継承の特性より，参照インスタンスを明に指定しないメソッド呼出しに関して 2 つの問題が生じる．

これらの問題を説明するため，クラス間のメソッド呼出しの様子を図 6.4に示す． *Parent*, *Heir*, *Referenced*, *Generated* は，それぞれ図 6.1における親クラス，後継者クラス，参照クラス，新規生成クラスを指す． method{} はメソッドの定義， method(); はメソッドの呼出しを表す． *Common* は *Parent* と *Referenced* の共通の祖先において最下位の (*Parent* あるいは *Referenced* に最も近い) クラスを指す． H は *Parent* と *Heir* の変更差異を集めた履歴を指す．いま， *Common* から *Parent* へ到達する経路上にあるクラスの集合 (*Common* を除く) を R_p， *Common* から *Referenced* へ到達する経路上にあるクラスの集合 (*Common* を除く) を R_r とおく． *Common* が *Parent* あるいは *Referenced* と同一であるとき， *Parent* や *Referenced* は， R_p あるいは R_r から削除する． $CSET_p$ は R_p 内に存在するクラスの子孫クラスの集合， $CSET_r$ は R_r 内に存在するクラスの子孫クラスの集合である． $CSET_p$ および $CSET_r$ を単純に *Parent* あるいは *Referenced* の祖先クラスだけとせず， R_p あるいは R_r の子孫クラスの集合としたのは，呼び出されたメソッドが抽象メソッドで，対応する具象メソッドが抽象メソッドを含むクラスの子孫で定義されている可能性があるからである．

メソッド呼出しに関する 2 つの問題を次に示す．

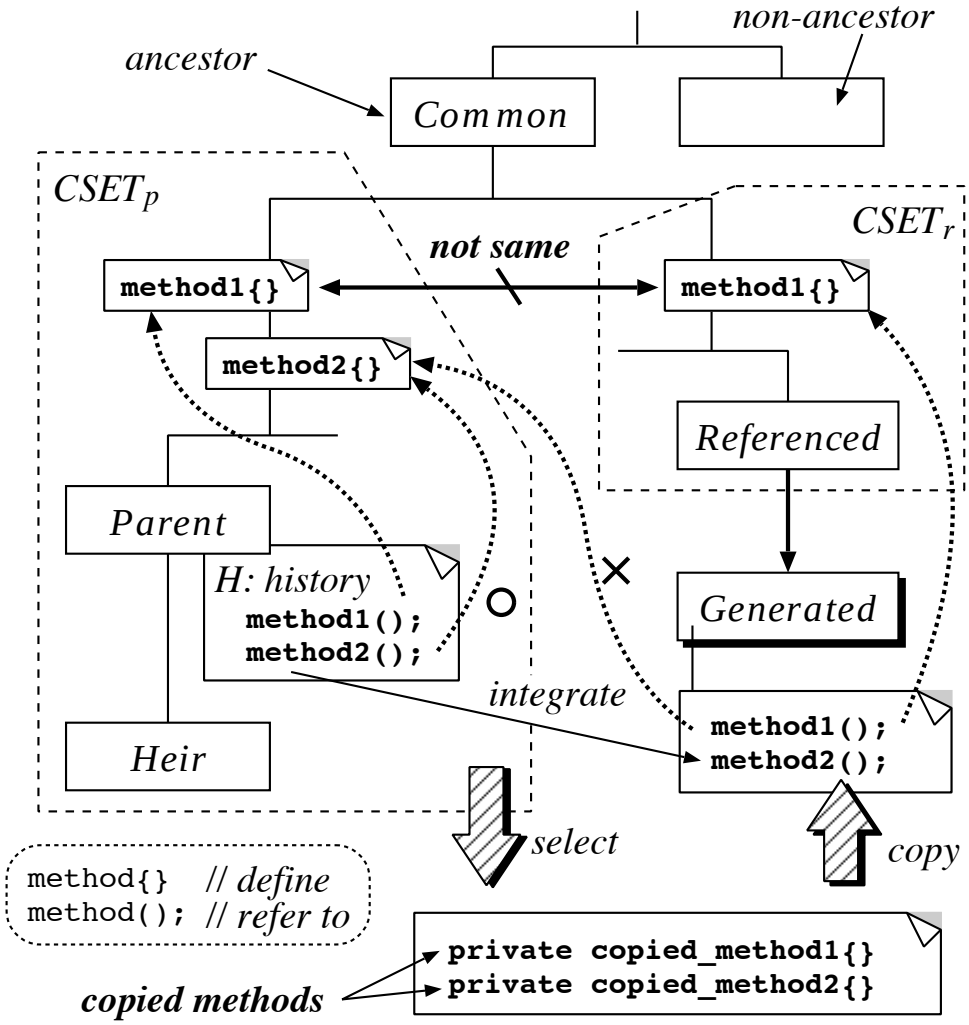


図 6.4: メソッドの呼出しに関する問題

同一性に関する問題 変更差異 H と新規クラス *Generated* から呼び出される同名のメソッドが同一のものを指していないことがあり，これらのメソッドの機能が等しいとは限らない．このため，H に含まれる機能が新規クラスに正しく合成されず，H 内に含まれていた文と実際に *Generated* から呼び出されたメソッド内の文が矛盾することがある．例えば，*Generated* におけるメソッド `method();` の呼出し先は，$CSET_r$ 内に存在するメソッド `method1{}` であり，H から呼び出される $CSET_p$ 内に存在するメソッド `method1{}` と実体が異なる．この場合，H に含まれる `method1{}` の機能が *Generated* に正しく合成されない．

参照可能性に関する問題 参照クラス *Referenced* から呼出し不可能なメソッドが変更差異 H に含まれる場合，実行不可能なメソッドの呼出しが新規クラス *Generated* に存在することになる．例えば，$CSET_p$ 内に存在するメソッド `method2{}` が *Referenced* に対して非公開である場合， *Generated* はこのメソッドを呼び出すことはできない．

呼出しメソッドの型が静的かつ一意に決定可能な場合を考える (型が動的に特定される場合については，6.8章で考察する)．この場合，上記の問題を回避するためには，実際にメッセージを処理するメソッドの実体が必ず *Generated* に存在するように， *Parent*, *Heir*, *Referenced* から呼び出されるすべてのメソッドの実体を，新規クラス *Generated* に複写すればよい．なぜなら，実際にメッセージを処理するインスタンスは，呼び出されたメソッドの実体を含むクラスから生成したものであるからである．

しかしながら，メソッドを複写することは，新規生成クラスのソースコードの量の増加につながる．さらに，メソッドを複写すると，祖先クラスにおける複写対象元メソッドへの変更が複写メソッドを含む新規生成クラスに伝搬しなくなる．このように， *Parent*, *Heir*, *Referenced* から呼び出されるすべてのメソッドを無制限に複写することは，新規生成クラスに対する理解あるいはクラス階層の管理という面で得策ではない．

そこで，本手法では，同名の呼出しメソッドの実体が必ず等しくなる，かつ，新規生成クラス *Generated* から呼出し可能なメソッドを，クラス階層木における *Parent* と *Referenced* の位置関係に基づいて決定しておき，メソッド複写の対象から除く．いま，図 6.4において，$CSET_p$ あるいは $CSET_r$ のどちらにも含まれないクラスを考える．これらのクラスは，*Common* の祖先であるクラスと祖先でないクラスに分けられる． *Common* の祖先クラスで定義されている公開 (public) メソッドおよび保護 (protected) メソッドは， *Generated* から呼出し可能である．さらに，これらのメソッドが *Parent*, *Heir*, *Generated* から呼び出された場合，そのメソッドの実体は必ず同じになる．すなわち，呼出しメソッドの同一性および参照可能性は保証される．ここで，公開メソッドとはどのクラスからでも参照可能なメソッドを，保護メソッドとはその子孫クラスからのみ参照可能なメソッドを指す． *Common* の祖先でないクラスについては，そのクラスで定義されているメソッドの呼出し時に必ず参照インスタンス名が指定されるため，同一性および参照可能性に関する問題を引き起こさない． *Parent*, *Heir*，あるいは， *Referenced* が非公開メソッドの呼出しを含む場合には，呼出しメソッドが *Generated* から呼出し不可能なので必ず複写する必要がある．非公開メソッドとはそのメソッドが定義されるクラスからのみ参照可能なメソッドを指す．

以上より，$CSET_p$ あるいは $CSET_r$ のどちらかに存在するクラスで定義されているメソッ

ドに関してのみ，呼出しメソッドの同一性と参照可能性を保証し，できるだけ複写数が小さくなるように複写メソッドを決定すればよい．本クラス生成手法では，変更履歴 H を参照クラス *Referenced* に合成する際，$CSET_p$ (および $CSET_r$) を決定し，次に示すメソッドの呼出しに対してのみ，メソッド複写を行う．

複写操作 C1: $CSET_p$ に含まれるクラスで定義されており，参照インスタンスを指定せずに呼び出された公開メソッド．

複写操作 C2: $CSET_p$ に含まれるクラスで定義されている保護メソッド．

複写操作 C3: *Parent*, *Heir*，あるいは， *Referenced* で定義されている非公開メソッド．

ただし，同名のメソッドが変更差異の合成により新規に生成されている場合，そのメソッドの複写は行わない (新規生成メソッドでない場合，複写を行う)．複写対象のメソッドが抽象メソッドの場合，そのメソッドに静的に束縛される実体を複写する．また，複写されたメソッドが，さらに上記のメソッドに対する呼出しを含む場合，メソッド複写を再帰的に行う．すべての複写メソッドは，他のクラスから呼び出されることがないため，非公開とする．さらに，メソッド名の衝突をさけるため，複写メソッドの名前は，もとのメソッド名に接頭語 (例えば， “copied_”) を付けたものとする．複写メソッドで利用されるインスタンス変数が新規クラスにおいて宣言されていない場合，その変数の宣言部も同時に複写する．図 6.4では， $CSET_p$ に含まれるクラスのメソッド `method1{}` と `method2{}` が， *Generated* に複写される．

6.5 新規クラス生成手法における各手続き

本節では，新規クラスを生成する 3 つの手続きについて詳細を述べる．

6.5.1 変更差異特定部

変更差異特定部は，継承において開発者がメソッドの追加および再定義を行った際，図 6.5に示す動作を行う．

親クラス *Parent* と後継者クラス *Heir* において，インターフェイスが一致する 2 つのメソッド (再定義メソッドと再定義されたメソッド) `method` から，スライス S_{Pi} と S_{Qj} を抽出する．次に， S_{Pi} および S_{Qj} に対する補スライス S^*_{Pi} および S^*_{Qj} を求め，それらの ClDG に関して同形写像比較を行う．補スライスとは， 5.4.2節で定義したものである．これは，も

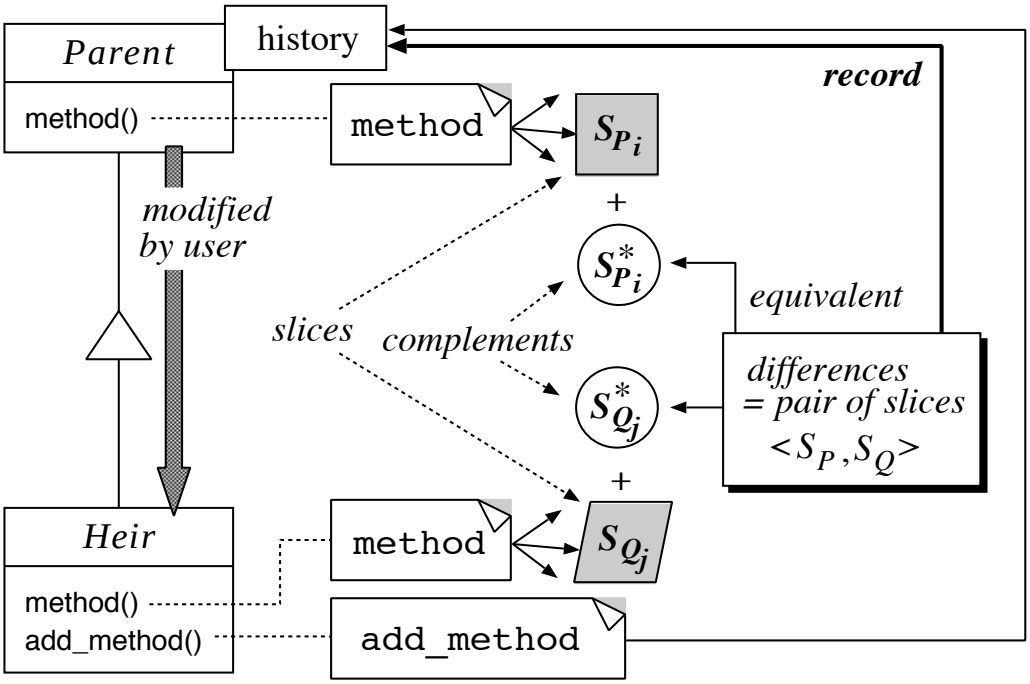


図 6.5: 変更差異の特定

とのソースコードにおける抽出対象区間の節点からスライスに含まれる節点を取り除いたものであり，スライスおよび区間外の全節点を 1 つの節点に集めることで作成する．比較する 2 つの ClDG において変更部分がそれぞれのスライス S_{Pi} および S_{Qj} に閉じ込められた場合，無変更部分を表す補スライス S^*_{Pi} と S^*_{Qj} が等価になる．よって，同形写像比較により等価な補スライスを見つけることで，もとのメソッドの ClDG の一部を変更部分として特定可能である．最後に，変更部分を内包するスライス S_P と S_Q の対 $\langle S_P, S_Q \rangle$ を，それらに含まれる節点の対応関係とともに親クラスの履歴に蓄積する．節点の対応関係は，スライス S_P と S_Q の部分スライス (部分グラフ) どうしの同形写像比較により得られる．メソッド `add_method` は， *Heir* だけに含まれるため，メソッド全体が変更部分となる．

6.5.2 類似クラス探索部

類似クラス探索部は，参照クラスに開発者の要求する呼出しメソッドの定義が存在しないとき，参照クラスと合成可能な変更履歴を持つ類似クラスの探索を行う．

本クラス生成手法では，メソッドのインターフェイスにより開発者の要求を定義しているため，要求メソッドとインターフェイスが一致するメソッドを履歴に含むクラスを類似クラ

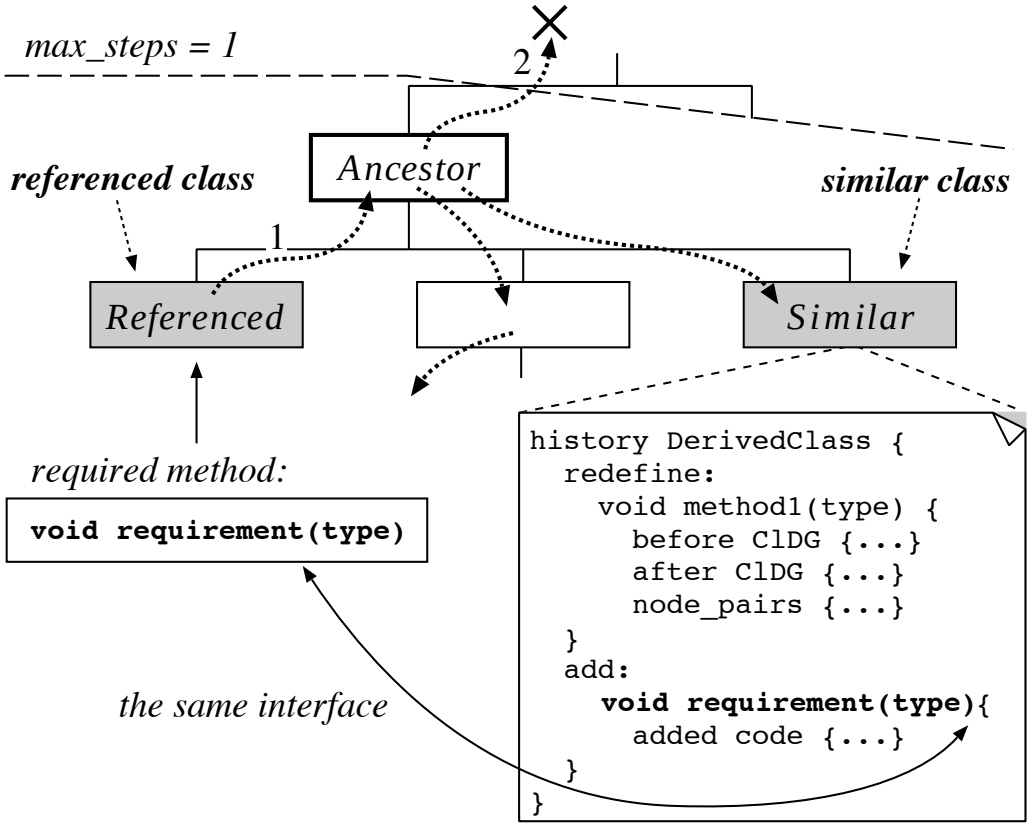


図 6.6: 類似クラスの特定

スの候補とする．しかし，インターフェイスの一致だけに基づき類似クラスを特定すると，合成対象の候補が大量に見つかる恐れがある．さらに，これらの候補のほとんどは，要求メソッドの生成に役立たない可能性が高い．なぜなら，メソッドの合成を成功させるためには，参照クラスと類似クラスが同様の変更を受け入れる，つまり，同様のプログラム構造 (ClDG) を含む同名のメソッドを持たなければならないからである．

そこで，本生成手法では，メソッド合成の試行回数を減少させるため，クラス階層における参照クラスとの位置関係を利用して類似クラスの探索範囲を限定する．いま，クラス階層木内の 2 つのクラスに対して，共通の祖先クラスまでの継承段数 (階層数) が少ないほど継承する共通な性質は多い．よって，これらのクラスにおいて，同じインターフェイスを持つメソッドが再定義されている可能性，さらに，それらのメソッドが同様の変更を受ける可能性が高いと判断できる．このことより，クラス階層木において，参照クラスと近い位置に存在するクラスほど，参照クラスと合成可能な変更差異を履歴に含む可能性は高いといえる．

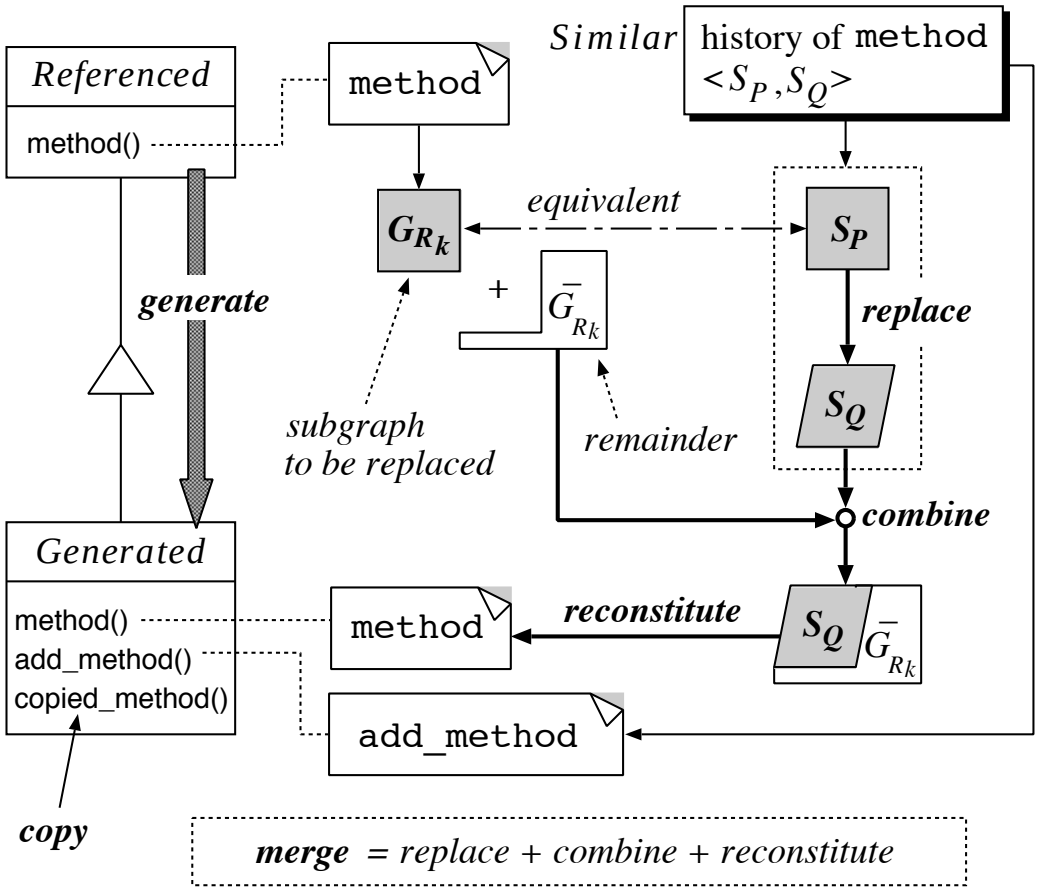


図 6.7: 変更差異の合成

類似クラス探索部の動作を図 6.6に示す．参照クラスは *Referenced*，開発者が呼び出した要求メソッドのインターフェイスは `requirement(type)` である．類似クラス探索部は，クラス階層における継承関係を *Referenced* から 1 段階づつ上がり，到達したクラス (1 段階上位の場合， *Ancestor*) の子孫を類似クラスの候補とする．このとき，クラス階層木ごとに探索段数をあらかじめ設定しておく．これらの候補に対して，変更履歴が要求メソッドと同じインターフェイスを持つメソッドを含むかどうかを検査することで，類似クラス *Similar* を特定する．図 6.6では，探索段数の最大値を $max\_steps = 1$ と設定したので， *Ancestor* より上位のクラスに対しては探索を行わない．

6.5.3 変更差異合成部

変更差異合成部は，類似クラスが特定された後，図 6.7に示す動作を行う．

変更差異の合成は，参照クラス *Referenced* において，類似クラス *Similar* の履歴に含まれるメソッドとインターフェイスが一致するメソッドに対して行われる． *Referenced* の合成対象メソッド `method` の ClDG において， *Similar* の履歴内の変更前スライス S_P に等価な部分グラフ G_{Rk} を抽出する．この部分グラフを置換対象グラフと呼ぶ．置換対象グラフは， S_P の抽出元メソッドのインターフェイスと S_P の各節点のラベルに基づき抜き出し，S_P との同形写像比較により等価性を検査する．次に，履歴内の節点の対応関係を用いて，置換対象グラフ G_{Rk} を変更前スライス S_P に置換し，さらに， S_P を変更後スライス S_Q に置換する．置換後の S_Q を，参照クラス内の合成対象メソッドにおける無置換部分 $\bar{G}_R$ と合成する．合成後の ClDG $(S_Q + \bar{G}_R)$ は，ソースコードに再構成され，新規クラス *Generated* のメソッド `method` となる．合成対象メソッドが *Referenced* に存在しない `add_method` については，そのコードがそのまま *Generated* に追加される．最後に，クラス階層木における *Referenced* および *Similar* の位置に基づき， 6.4節で示したメソッド複写操作 C1, C2, C3 を適用し， `copied_method` が複写される．

6.6 新規クラス自動生成の例

新規クラスを自動生成する例を， Java 言語で記述したプログラムを用いて示す．本例で用いるクラスの関係を図 6.8に示す．

(1) 変更差異の特定

開発者が，継承により，親クラス DisplayArea にスクロールバーの機能を追加して，後継者クラス DisplayAreaScrollbar を作成した場合を考える． DisplayArea のソースコードを図 6.9， DisplayAreaScrollbar のソースコードを図 6.10に示す． Java パッケージ内の Graphics クラスにおいて定義されるメソッド (`setColor`, `drawLine`, `drawString`) と Component クラスにおいて定義されるメソッド (`size`) 内部の依存関係は解析済みであるとする．本例では，メソッド内部のデータ依存関係が分かるように，各節点における定義および参照変数を図 6.9, 6.10に示す． `lines.X1/Y1/X2/Y2` は， `lines.X1`, `lines.Y1`, `lines.X2`, `lines.Y2` の短縮形である．また，スクリーンへの書き込みを表すため，描画領域を指す仮想的な変数 `gImage` を導入した．

DisplayArea と DisplayAreaScrollbar のメソッド `paint` の ClDG と変更差異を図 6.11に示す．メソッド引数およびインスタンス変数に関する入出力節点は省略した．変更差異特定部は，これら 2 つの ClDG から，それぞれスライス $S_P = \hat{S}(\text{p7}, \text{p11}, \{\texttt{gImage}\})$ およ

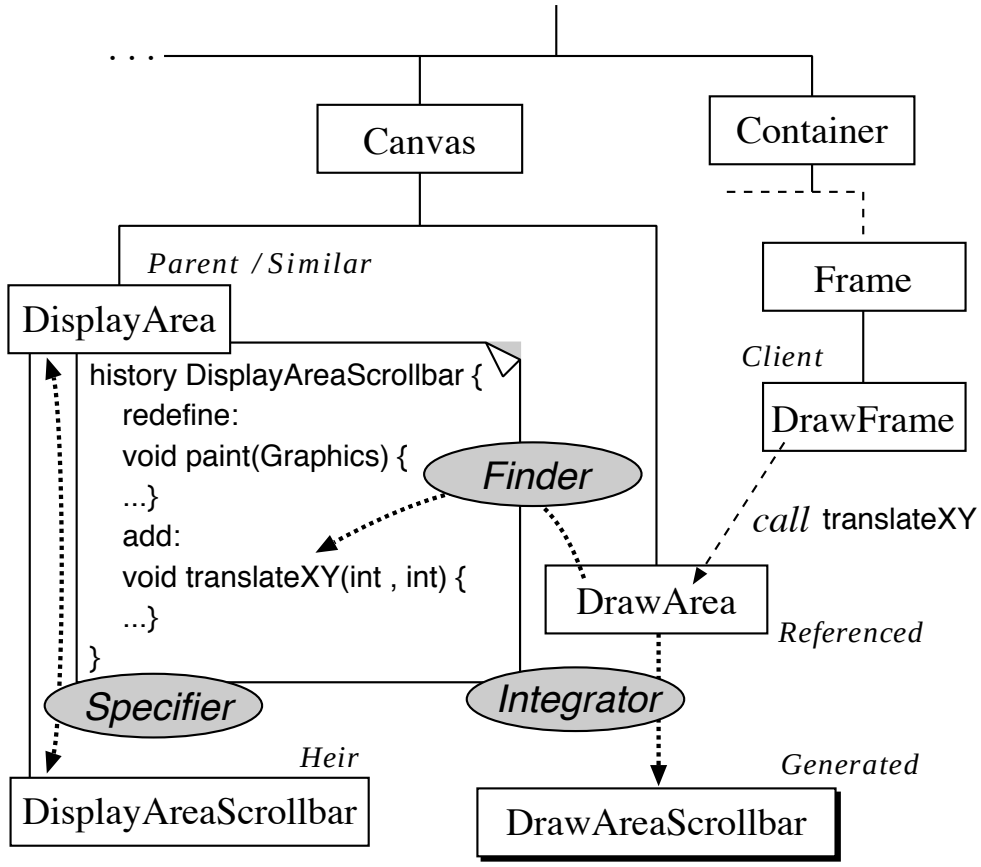


図 6.8: 新規クラス生成の例

び $S_Q = \hat{S}(\text{q11}, \text{q19}, \{\texttt{gImage}\})$ を抽出し，それらの補スライス S_P^* と S_Q^* を作成する．いま， S_P と区間外の節点を節点 N_P， S_Q と区間外の節点を N_Q に集約し， S_P^* および S_Q^* 内の節点とそれらの節点どうしを結ぶ矢印を残した 2 つの ClDG (図 6.11における右側の ClDG) は，同形写像比較において等価である．よって，網かけ節点で表現されたスライスの対 $\langle S_P, S_Q \rangle$ が DisplayArea と DisplayAreaScrollbar のメソッド `paint` の変更差異となる．$\langle S_P, S_Q \rangle$ における節点の対応関係を図 6.11の ClDG の下に示す． $-$ は，対応する節点が存在しないことを表す．

(2) 類似クラスの探索

参照クラス DrawArea で定義されていないメソッド `translateXY` に対する呼出し文を，開発者がクラス DrawFrame に記述した場合を考える． DrawFrame のソースコードを図 6.12,

```
p1 class DisplayArea extends Canvas {
     Line line[]; int lineN;
p2   DisplayArea() {...}
p3   public void paint(Graphics g) {
p4     g.setColor(Color.blue);
p5     int hmax = size().width;
p6     int vmax = size().height;
p7     boolean outRange = false;
p8     for (int l = 0;
p9     l < lineN;
p10    l++) {
p11      g.drawLine(line[l].x1,line[l].y1,
                    line[l].x2,line[l].y2);
p12      if (line[l].x1 < 0 || line[l].x1 > hmax
          || line[l].x2 < 0 || line[l].x2 > hmax
          || line[l].y1 < 0 || line[l].y1 > vmax
          || line[l].y2 < 0 || line[l].y2 > vmax)
p13        outRange = true;
       }
p14    if (outRange)
p15      g.drawString("out of range",5,size().height);
     }
     ...
   }
```

Node	*defined variables*	*used variables*
p4	g.color	g
p5	hmax	size().width
p6	vmax	size().height
p7	outRange	
p8	l	
p9		l, lineN
p10	l	l
p11	gImage	g, g.color, l, line[].x1/y1/x2/y2
p12		l, hmax, vmax, line[].x1/y1/x2/y2
p13	outRange	
p14		outRange
p15	gImage	g, g.color, size().height

gImage : virtual variable for storing a image

図 6.9: 親クラス DisplayArea のソースコード

```
q1 class DisplayAreaScrollbar extends DisplayArea {
     int transX, transY; int hscmax, vscmax;
q2   DisplayAreaScrollbar(int h, int v) {...}
q3   public void translateXY(int x, int y) {
q4     transX = x;
q5     transY = y;
q6     redraw();
     }
q7   public void paint(Graphics g) {
q8     g.setColor(Color.blue);
q9     int hmax = size().width+hscmax;
q10    int vmax = size().width+vscmax;
q11    boolean outRange = false;
q12    for (int l = 0;
q13    l < lineN;
q14    l++) {
q15      g.drawLine(line[l].x1-transX,line.y1-transY,
                    line[l].x2-transY,line.y2-transY);
q16      if (line[l].x1 < 0 || line[l].x2 > hmax
          || line[l].x2 < 0 || line[l].x2 > hmax
          || line[l].y1 < 0 || line[l].y1 > vmax
          || line[l].y2 < 0 || line[l].y2 > vmax)
q17        outRange = true;
       }
q18    g.color(Color.black);
q19    g.drawString("X="+transX+";Y="+transY, 5, 15);
q20    if (outRange)
q21      g.drawString("out of range",5,size().height);
     }
q22  public void redraw() {...}
   }
```

Node	*defined variables*	*used variables*
q8	g.color	g
q9	hmax	size().width, hscmax
q10	vmax	size().height, vscmax
q11	outRange	
q12	l	
q13		l, lineN
q14	l	l
q15	gImage	g, g.color, l, transX, transY, line[].x1/y1/x2/y2
q16		l, hmax, vmax, line[].x1/y1/x2/y2
q17	outRange	
q18	g.color	g
q19	gImage	g, g,color, transX, transY
q20		outRange
q21	gImage	g, g.color, size().height

図 6.10: 後継者クラス DisplayAreaScrollbar のソースコード

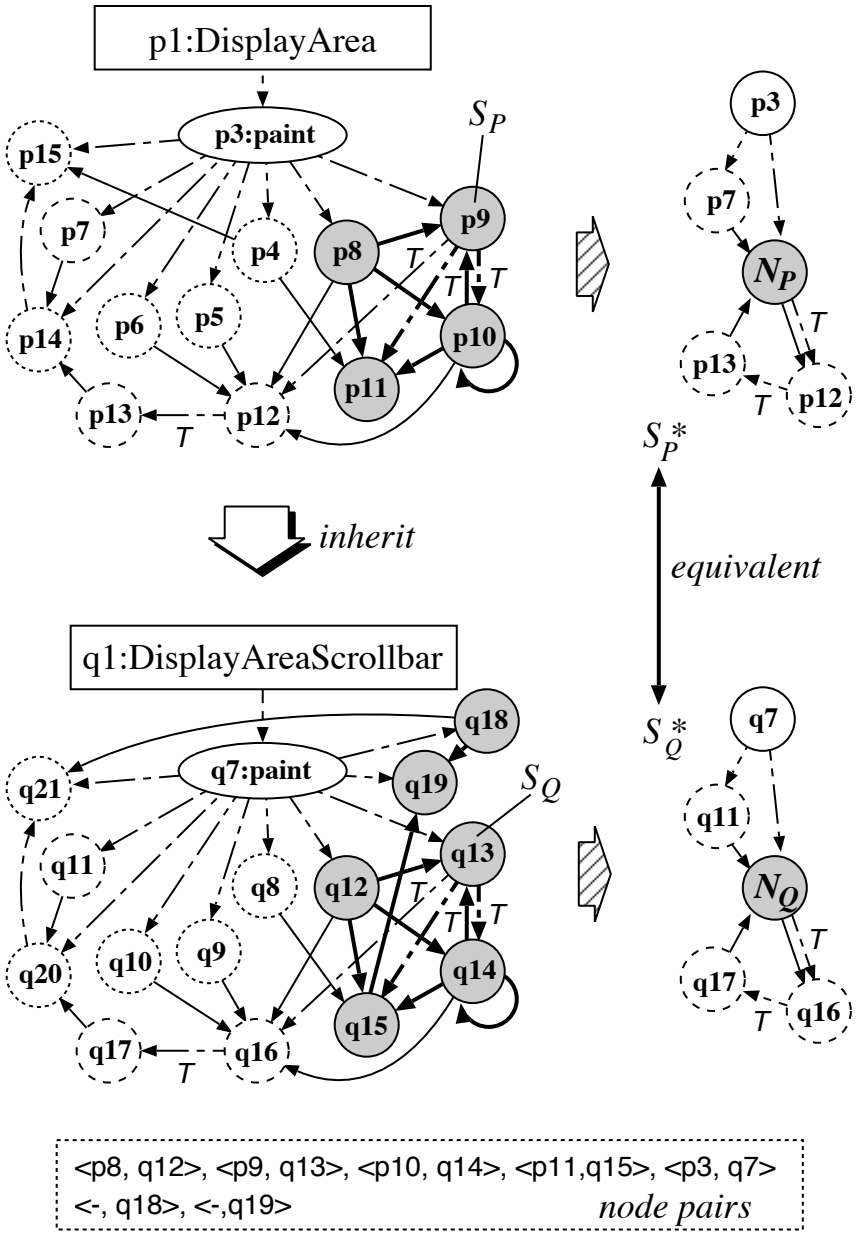


図 6.11: 親クラスと後継者クラスにおけるメソッド paint の ClDG と変更差異 S_P と S_Q

DrawArea のソースコードを図 6.13に示す．このような状況は，開発者が，他のクラス (例えば， DisplayAreaScrollbar) に存在する `translateXY` の名前と機能から， DrawArea の派生クラスに `translateXY` を追加する必要があると考えた場合に生じる．この場合，開発者は， `translateXY` の存在を想定して，このメソッドを呼び出す文を DrawFrame に記述す

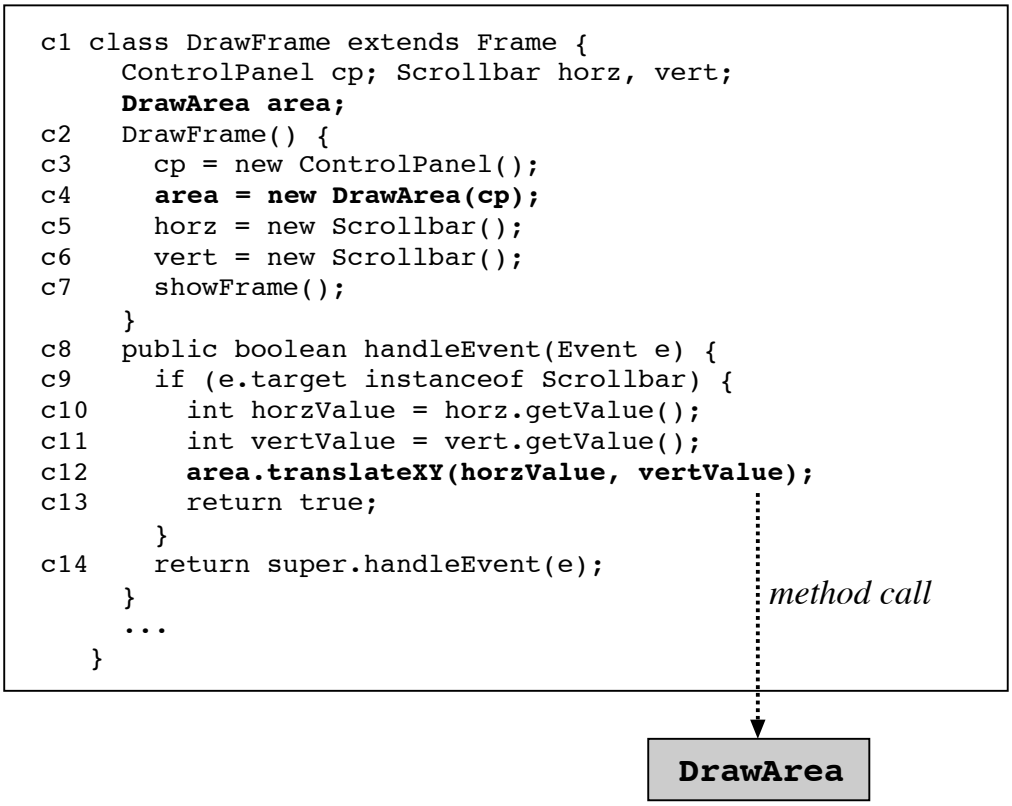
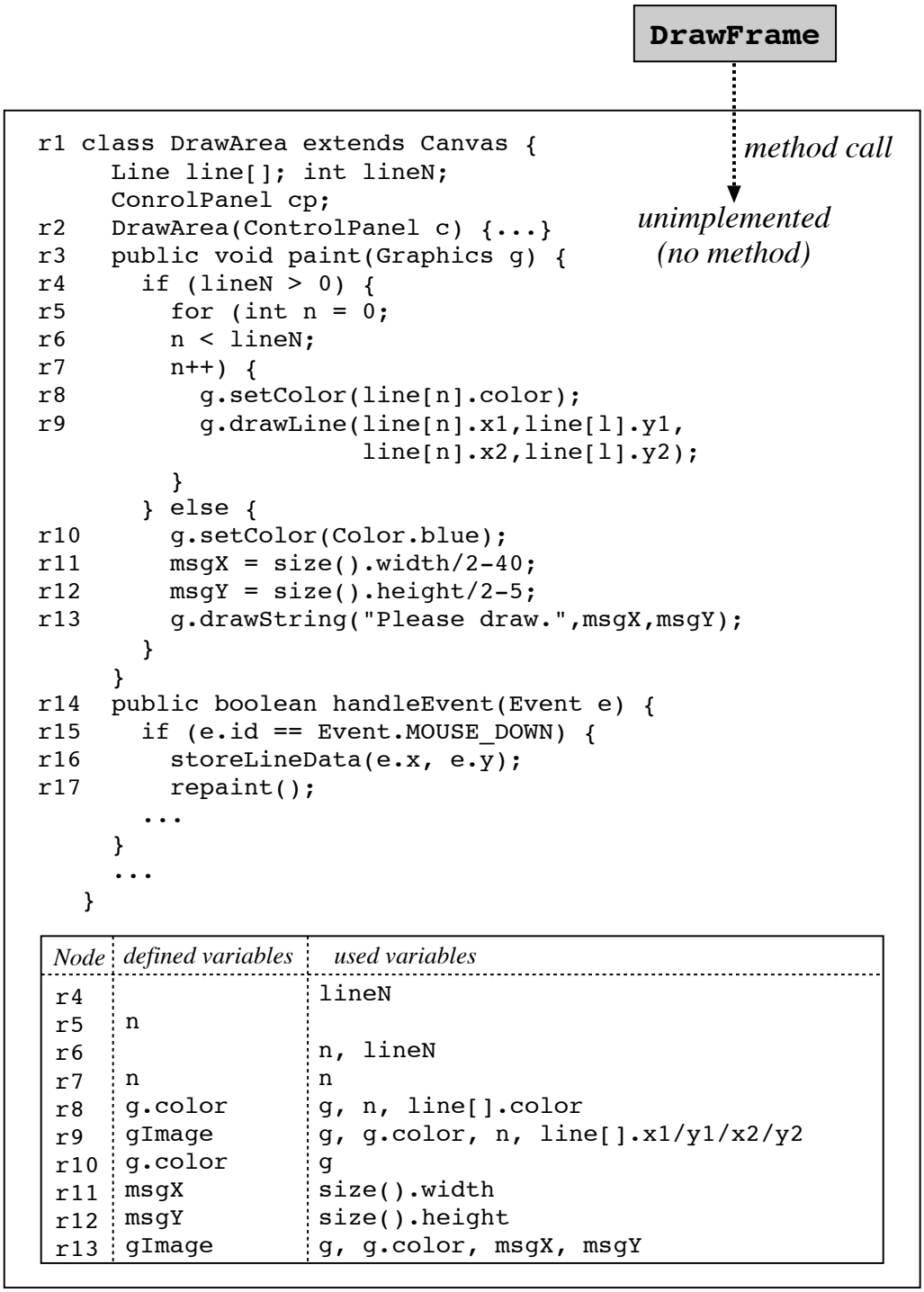


図 6.12: 呼出し側クラス DrawFrame のソースコード




```
r1 class DrawArea extends Canvas {
     Line line[]; int lineN;
     ConrolPanel cp;
r2   DrawArea(ControlPanel c) {...}
r3   public void paint(Graphics g) {
r4     if (lineN > 0) {
r5       for (int n = 0;
r6       n < lineN;
r7       n++) {
r8         g.setColor(line[n].color);
r9         g.drawLine(line[n].x1,line[l].y1,
                      line[n].x2,line[l].y2);
         }
       } else {
r10      g.setColor(Color.blue);
r11      msgX = size().width/2-40;
r12      msgY = size().height/2-5;
r13      g.drawString("Please draw.",msgX,msgY);
       }
     }
r14  public boolean handleEvent(Event e) {
r15    if (e.id == Event.MOUSE_DOWN) {
r16      storeLineData(e.x, e.y);
r17      repaint();
       ...
     }
     ...
   }
```

Node	*defined variables*	*used variables*
r4		lineN
r5	n	
r6		n, lineN
r7	n	n
r8	g.color	g, n, line[].color
r9	gImage	g, g.color, n, line[].x1/y1/x2/y2
r10	g.color	g
r11	msgX	size().width
r12	msgY	size().height
r13	gImage	g, g.color, msgX, msgY

図 6.13: 参照クラス DrawArea のソースコード

る．

本例では，文 c12 におけるメソッド translateXY の呼出しが失敗し，類似クラスの探索が行われる．類似クラス探索部は，図 6.8に示すように，呼出しメソッド translateXY とインターフェイスが一致するメソッドを持つ変更差異を DisplayArea の履歴内に見つける．

(3) 変更差異の合成

変更差異合成部では，発見した変更差異を参照クラス DrawArea に合成することで，最初にメソッド translateXY を生成する．さらに，変更差異に含まれる他のメソッドに関しても合成を適用し，メソッド paint を生成する． translateXY については，合成対象となる同一インターフェイスを持つメソッドが DrawArea 内に存在しないので，メソッドのコード全体を新規クラスに追加する．一方， paint については，変更差異に含まれる paint のコードと DrawArea に含まれる paint のコードを合成することで生成する．

DrawArea における合成対象メソッド paint と本手法により生成した新規メソッド paint の ClDG を図 6.14に示す．本例では， DrawArea 内の変数名 n を l に置換することで，同形写像比較において， G_R と類似クラス DisplayArea における変更前スライス S_P が等価になる．よって，網かけ節点で表現された G_R が置換対象グラフとなる．次に，図 6.14に示す節点の対応関係を用いて， G_R 内の節点を $r \rightarrow p$, $p \rightarrow q$ の順に置換する．さらに， S_Q に含まれていた他の節点および矢印を追加し， S_Q と置換対象グラフ G_R の残り $\bar{G_R}$ を合成する．このようにして，新規クラス DrawAreaScrollbar のメソッド paint の ClDG が生成される．本クラス生成手法により自動生成された新規クラス DrawAreaScrollbar のソースコードを図 6.15に示す．

メソッド paint (13–31 行目) は，図 6.14の下段の ClDG をソースコードに変換したものである．メソッド translateXY (8–12 行目) と，このメソッドで用いられるインスタンス変数 transX, transY の宣言文 (2 行目) が，類似クラス DispalyArea の変更履歴から追加された．また，参照可能性を保証するために，メソッド copied_redraw (32–36 行目) が DispalyArea から複写 (複写操作 C1) され，このメソッドに対する呼び出しが copied_redraw() (11 行目) のように書き換えられる．呼出し文 (35 行目) に対応するメソッド paint の実体は，同名のメソッドが直前に生成されているため複写されない．

新規クラス DrawAreaScrollbar のソースコードを見ると，開発者の要求したメソッド translateXY(int, int) が存在することが確認できる．また， DisplayAreaScrollbar において座標変換を行うように再定義したメソッド paint の変更部分 (19–21 行目，節点 q15) が，

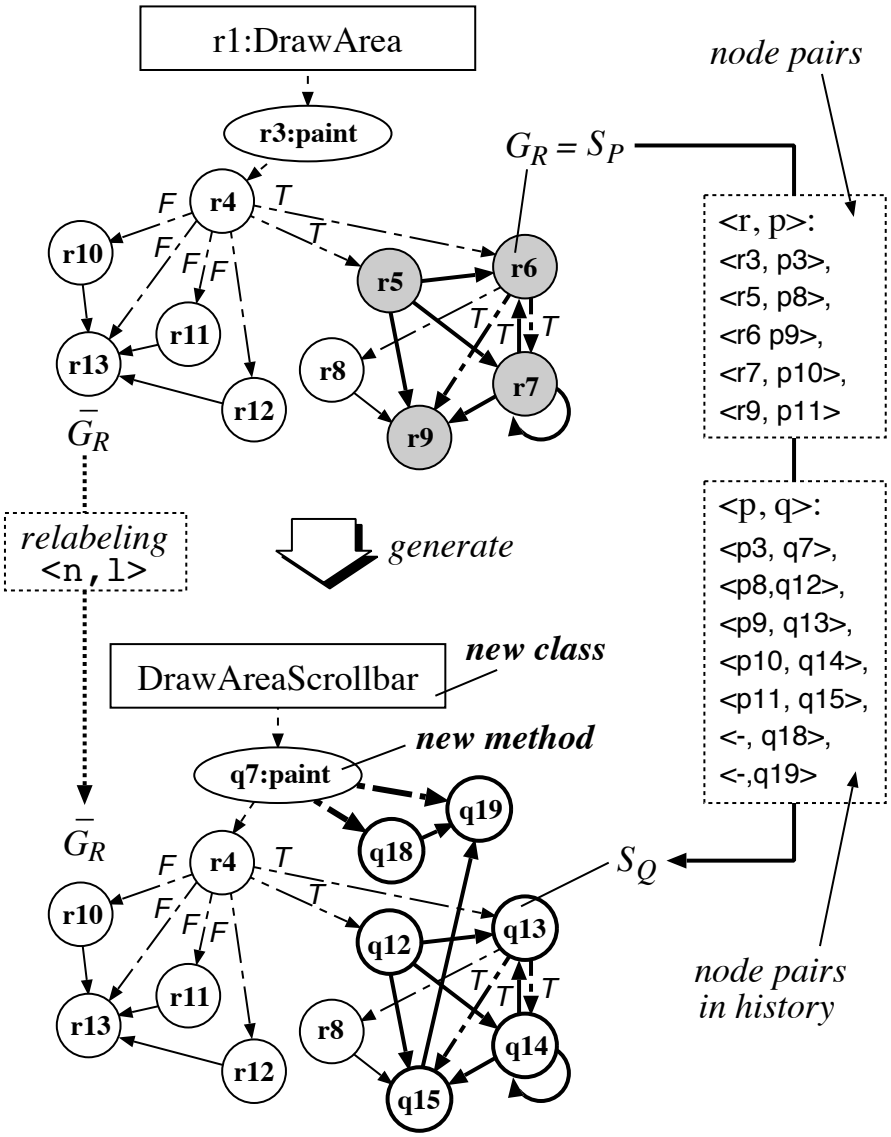


図 6.14: 参照クラスと新規クラスのメソッド paint の ClDG

DrawAreaScrollbar の新規メソッド `paint` に反映されているのが分かる．さらに，参照クラス DrawArea のメソッド `paint` において個々の線に色を付ける機能 (18 行目， DrawArea の節点 r8, r9) と，類似クラス DisplayAreaScrollbar のメソッド `paint` におけるスクロールバーの現在座標を表示する機能 (29–30 行目， DisplayAreaScrollbar の節点 q18, q19) が保存されていることも確認できる．本手法では， DrawAreaScrollbar の名前 (1 行目) およびコンストラクタは (3–7 行目) は自動生成されないため，これらのソースコードは開発者が

```
 1:     class DrawAreaScrollbar extends DrawArea {
 2:       int transX, transY;  <------------------------------ added
 3:       DrawAreaScrollbar(ControlPanel c) {
 4:         super(c);
 5:         transX = 0;
 6:         transY = 0;                          implemented by user
 7:       }
 8:  q3   public void translateXY(int x, int y){
 9:  q4     transX = x;
10:  q5     transY = y;
11:  q6     copied_redraw();                                 added
12:       }
13:  q7   public void paint(Graphics g){
14:  r4     if (lineN > 0) {
15:  q12      for (int l = 0;
16:  q13      l < lineN;
17:  q14      l++) {                                         merged
18:  r8         g.setColor(line[l].color);
19:  q15        g.drawLine(
20:               line[l].x1-transX,line[l].y1-transY,
21:               line[l].x2-transX,line[l].y2-transY);
22:           }
23:         } else {
24:  r10      g.setColor(Color.blue);
25:  r11      msgX = size().width/2 - 40;
26:  r12      msgY = size().height/2 - 5;
27:  r13      g.drawString("Please draw.",msgX,msgY);
28:         }
29:  q18    g.setColor(Color.black);
30:  q19    g.drawString("X="+transX+";Y="+transY,5,15);
31:       }
32:  q22  private void copied_redraw(){
33:  q22    Graphics g = getGraphics();
34:  q22    g.clearRect(0,0,size().width,size().height);
35:  q22    paint(g);
36:       }                                                 copied
37:     }
```

図 6.15: 新規クラス DrawAreaScrollbar のソースコード

記述する必要がある．このように，本例では，既存クラスおよび変更履歴内の依存関係を保存，かつ，合成により追加した機能が既存クラスの機能に無矛盾のままで，開発者の要求するメソッドを持つ新規クラスの生成に成功した．この新規クラスは，コンストラクタの追加という少ない変更で実行可能であった．

6.7 分散ネットワーク環境に対するクラス生成手法の適用

従来の部品化再利用において，アプリケーション開発者は，自分のライブラリに存在する部品を自分で変更し，ソフトウェアを開発していた．これに対して，近年の部品組立て型ソフトウェア開発 (component-based software development: CBSD) [4, 22] では，開発者 (エンドユーザ) と部品提供者が区別され，開発者はネットワーク上に分散している部品を組み合わせることで，要求を満たすソフトウェアを作成する．このような分散環境において，開発者各自が現在利用している部品は，開発者のアプリケーション利用環境に応じて，独自の変更が行われている可能性が高い．よって，各開発者が要求する部品は，たとえ類似の機能を持っていたとしてもその実装が微妙に異なり，部品提供者の行う更新だけで開発者の要求に迅速に応じることは困難である．

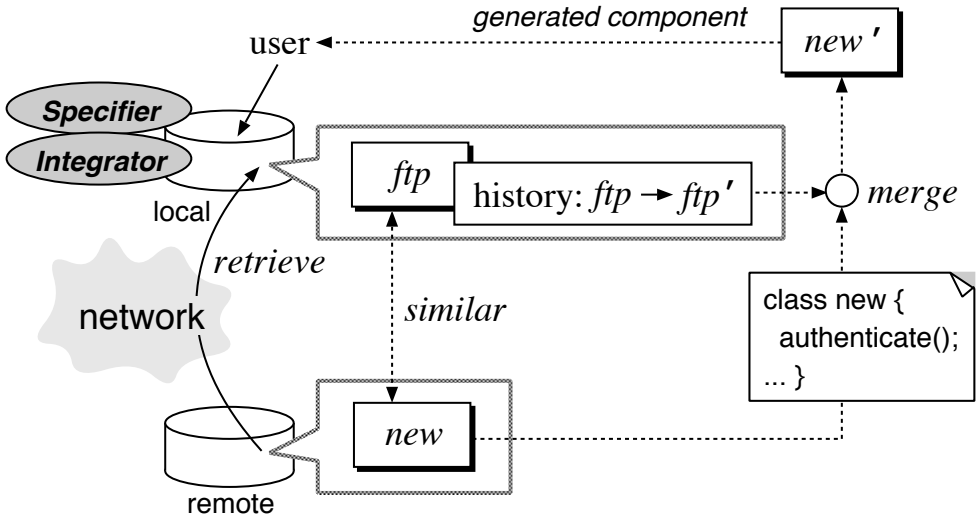
本章で提案するクラス生成手法は，過去のクラス継承における変更を模倣することで新規クラスを生成するため，ライブラリに存在しないクラスを要求した場合でも，その要求を満たすメソッドを持つ再利用可能なクラスを提供できるという利点を持つ．本節では，本クラス生成手法をネットワーク上に分散する部品に対して適用する方式を提案し，本手法が分散ネットワークにおける部品変更や部品更新に対しても有効であることを示す．

本クラス生成手法を適用する場合，ネットワーク上を配送する対象が部品自身であるか，あるいは，部品の変更履歴であるかによって，次に示す 2 種類の方式が考えられる．

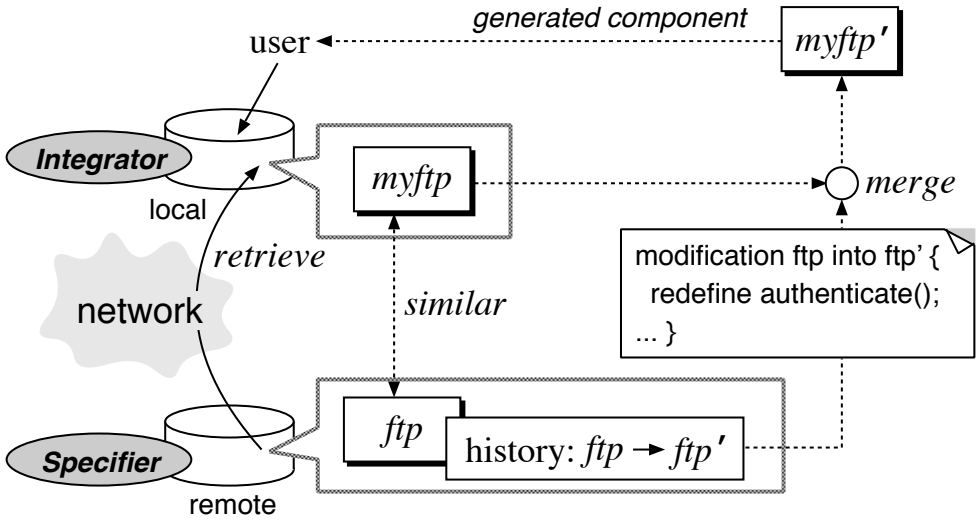
(a) 開発者の要求を部分的に満たす部品をネットワーク上において配送し，開発者のライブラリに存在する変更履歴と合成する方式．

(b) 開発者の要求を満たすために必要な変更履歴をネットワーク上において配送し，開発者のライブラリに存在する部品と合成する方式．

部品および変更履歴の両方を配送対象とする場合は，これら 2 つの方式の組み合わせとみなす．ここでは，オブジェクト指向プログラムのソースコードを部品と呼ぶ．よって，新規部品の生成とは，開発者の要求を満たす機能を実現したソースコードを生成することを指す． 2 つの適用方式を図 6.16を示す．以下に， 2 つの適用方式をそれぞれ述べる．



(a) 部品を配送する方式



(b) 変更履歴を配送する方式

図 6.16: 分散ネットワーク環境に対する適用方式

6.7.1 部品を配送する方式

図 6.16a に部品を配送する方式を示す．本方式では，開発者がネットワークを通して，要求を部分的に満たす部品をリモートライブラリから取得し，それらの部品をカスタマイズ

することで，目的のアプリケーションを作成する．変更差異特定部 (Specifier) は，あらかじめ開発者のライブラリ内の部品に対して変更履歴を蓄積しておく．リモートライブラリから取得した部品に対して，開発者が過去と同様の変更を要求すると，変更差異合成部 (Integrator) は，それら変更対象部品と開発者のローカルライブラリに存在する類似部品の変更履歴を合成することで，開発者の要求する新規部品を生成する．

例えば，開発者の環境ではアプリケーションに独自の認証機構を採用している場合を考える．このような環境では，開発者は，汎用部品 *ftp* などに対して，独自の認証機構を組み込む変更を過去に行っているはずである．いま，新しいサービスに対応した汎用部品 *new* が登場し，開発者がこの部品を自分の環境で利用しようと考えた場合，開発者は過去と同様の変更を部品 *new* に行う必要がある．本方式では，このような状況において，開発者の環境に特化した変更履歴 ($ftp \rightarrow ftp'$) を汎用的な部品 *new* に適用することで，独自の認証機構を既に組み込んだ新規部品 new' を自動生成する．

一般的に，開発者が行う変更は，アプリケーション利用環境に応じた特殊な修正である．よって，このような変更の履歴は開発者の環境の特性を反映しており，開発者のローカルライブラリに数多く蓄積するべきである．本方式により，独自の変更履歴を用いて部品の自動カスタマイズが可能になると，開発者は部品変更を意識せずに，ネットワーク上のすべての部品を共有することができる．プログラムをネットワーク上で検索および配送するという概念は，従来から考えられている．よって，方式 (a) は，クラス生成手法を分散ネットワーク環境に対して適用する場合の自然な拡張であり，本クラス生成手法と従来の部品検索および配送機構を組み合わせることで比較的容易に実現可能であると考えられる．

6.7.2 変更履歴を配送する方式

図 6.16b に部品の変更履歴を配送する方式を示す．本方式では，開発者がネットワークを通して，部品の更新履歴をリモートライブラリから取得し，それらの履歴を用いてローカルライブラリに存在する部品を更新することで，目的のアプリケーションを作成する．部品提供者が汎用部品を更新した場合，変更差異特定部 (Specifier) は，その部品に対する変更履歴を部品提供者のライブラリに蓄積しておく．ローカルライブラリに存在する特殊な部品に対して，開発者が同様の部品更新を要求すると，変更差異合成部 (Integrator) は，開発者の部品と部品提供者の変更履歴を合成することで，開発者の要求する新規部品を生成する．

例えば，開発者が自分の環境に合わせて汎用部品 *ftp* に GUI を付加する独自の変更を行った部品 *myftp* を利用している場合を考える．いま，従来と異なる新しい認証機構が提案さ

れた場合，部品提供者は，以前に供給した部品 *ftp* に対して新しい認証機構を組み込む更新を行い，更新後の部品 *ftp′* を供給する．この場合，開発者が独自の GUI を持つ部品に新しい認証機構を組み込む更新を要求する可能性は高いが，このような部品が部品提供者から供給される可能性は低い．よって，開発者は，部品提供者が行った変更と同様の変更を部品 *myftp* に行う必要がある．本方式では，このような状況において，部品提供者が行った汎用的な更新に関する変更履歴 (*ftp* → *ftp′*) を，開発者の環境に特化した特殊な部品 *myftp* に適用することで，新しい認証機構を既に組み込んだ独自の GUI を持つ新規部品 *myftp′* を自動生成する．

一般的に，部品提供者が行う変更は，さまざまな環境で利用されている部品に対する汎用的な更新である．よって，このような変更の履歴は，開発者の環境に特化した部品に対しても適用できる可能性があり，積極的に公開することに意義がある．本方式により，ネットワーク上に存在する汎用的な部品変更履歴を用いて部品の自動更新が可能になると，開発者は将来の部品更新を意識せずに，部品を特殊化することができる．

方式 (b) は，変更履歴をネットワーク上で検索および配送するという新しい概念を含んでいる．従来も，配送するソースコードの量を減らすという観点で，ソースコード全体ではなく更新における差分だけを配送することが行われていた．しかし，変更時の知識を積極的に配送し，各開発者のライブラリに存在する部品を自動更新するという仕組みは，本クラス生成手法を適用することで可能となる．また，方式 (b) を実現する本手法を各ライブラリに組み込むことで，従来部品だけを供給していた部品提供者は，部品変更時の知識の供給という新しいサービスを行うことができる．

6.8 考察

本節では，関連研究との比較により本クラス生成手法の有効性を述べる．また，本手法の課題および拡張について考察する．

6.8.1 関連研究との比較

本研究と同様に，コンポーネントウェア [4] や部品組立て型ソフトウェア開発 (CBSD) [22] の目的は，開発者 (エンドユーザ) が部品を組み合わせるだけで，要求プログラムを作成できるようにすることである．また，ドメインに合わせて変換可能な部品を組合せることで，ソフトウェアを作成する GenVoca ジェネレータ [11] がある．これらの研究では，組合わせ可能な部品の形態やそれらの部品を利用するアーキテクチャの提案に主眼がおかれ，開発者

の要求する部品があらかじめ構築できることを前提としている．しかしながら，開発者のさまざまな要求を完全に予測し，無変更で再利用可能なクラスや部品をあらかじめ構築しておくことは難しい．

オブジェクト指向プログラミングに限定した場合でも，新規メソッドを含むクラスを静的に作成するのではなく，多重継承 (multiple inheritance) や mixins [21] のようなクラスコンポジションを用いることで，新規メソッドを含まないクラスを作成するだけで，機能を拡張する方法がある．また，オブジェクトコンポジション [33] により既存オブジェクトの機能を動的に合成することで，開発者の要求する機能を実現する方法がある [101]．さらに，機能を拡張するという観点から，クラスの継承階層を任意に結合するアプローチ [87]，あるいは，テンプレート機能により継承の順序を任意に変更可能なアプローチ [106] が提唱されている．

しかしながら，これらの方法では，合成対象のオブジェクトや派生元のクラスが適切に設計されている必要があり，開発者の要求する機能が既存クラス (オブジェクト) の機能の組合せだけで実現できる場合にのみ有効である．特に，オブジェクトコンポジションでは，既存クラス (オブジェクト) の機能の一部を洗練するような機能変更は実現できない．また，多重継承においても，既存クラスの組合せだけで開発者の要求が満たせない場合，メソッドの再定義が必要である．

6.6節の例では，クラス DisplayArea のメソッド `paint` とクラス DrawArea のメソッド `paint` において，描画する図形 (線) に対する色の付け方が異なり，色を指定する文と実際に図形を描画する文が密接に関連している．このような場合，DisplayArea にスクロールバーの機能を追加する際に用いた合成オブジェクトを，単純に DrawArea に合成することはできない．よって，オブジェクトコンポジションでは，開発者の要求する機能を備えたクラス DrawAreaScrollbar の実現は不可能である．また，多重継承を用いた場合でも，DrawAreaScrollbar において，メソッド `paint` の再定義が必要となる．

本章で提案するクラス生成手法は，継承におけるコード変更を自動化し，開発者の要求するメソッドを持つ再利用可能なクラスを生成する．これにより，従来のコンポジション手法に比べて，次に示す利点を持つ．

(1) 過去のクラス継承における変更を模倣することで新規クラスを生成するため，ライブラリに存在しないクラスを要求した場合でも，その要求を満たすメソッドを持つ再利用可能なクラスを提供できる可能性がある．さらに，同様の変更を適用した多種多様なクラスが，既存クラスから生成されるため，あらかじめ予測できない要求に対して

数多くのクラスをライブラリに用意しておく必要がない．

(2) 区間限定スライシングを用いて特定される変更の差分はメソッドより小さいため，微妙に異なる機能を持つさまざまな新規メソッドを生成できる．よって，既存のクラスに対してメソッドや変数を追加するだけでは作成できないクラス，つまり機能の合成だけでは実現できない派生クラスを生成可能である．

(3) プログラムスライシングによる機能分割手法と依存グラフの同形写像比較による等価機能の特定手法に基づくメソッド合成アルゴリズムにより，メソッドを合成する．よって，既存クラスの機能の一部を保存し，合成により付加した変更差異の機能が既存クラスのメソッドの機能に無矛盾であることを保証する．

一般に，本クラス生成手法で採用したメソッド再定義による機能拡張と，オブジェクトコンポジション (あるいは，メソッド再定義のないクラスコンポジション) による機能拡張は，相反する再利用技法ではなく，協調して利用可能である．よって，本生成手法は，従来のコンポジション手法と置換されるものではない．例えば，合成だけで実現可能な機能をコンポジションで，洗練により実現可能な機能を持つメソッドを本生成手法で生成することが考えられる．また，最初に本生成手法により開発者の要求を満たす可能性の高い新規クラスを生成し，このクラスを修正あるいは洗練することにより，機能合成において利用可能なクラス (オブジェクト) を作成することが考えられる．

6.8.2 拡張

(1) メソッドの整合性

本クラス生成手法では，開発者の要求を呼出しメソッドのインターフェイスで表現し，インターフェイスの一致により類似クラスを特定する．さらに，新規クラスを生成する際には，類似クラスの変更履歴と参照クラスにおいて，インターフェイスが一致するメソッドどうしを合成対象とする．これらの動作は，クラスライブラリにおいてメソッドの整合性が保たれるべき，つまり，類似の機能を持つメソッドは同じ名前およびシグニチャを持つべきである [95] という前提に基づいている．しかしながら，実際のプログラミングにおいて，このようなメソッドの整合性は必ずしも保証できない．

本生成手法では，インターフェイスの綴りを比較することにより，類似メソッドの探索あるいは合成対象メソッドの特定を実現している．よって，上記の前提が成立していないライブラリにおいても，新規メソッドの生成は可能である．しかし，この前提が成立しているか

どうかは，本手法の能力に大きな影響を与える．例えば，同じ機能を持つメソッドが異なるインターフェイスを持つ場合，類似クラスの発見が困難になる．また，異なる機能を持つメソッドが同じインターフェイスを持つ場合，生成したメソッドが開発者の要求する機能を満たさない可能性は高くなる．

このような問題を解決するためには，メソッドインターフェイスの綴りでなく，メソッドの持つ機能の等価性あるいは類似性を判定可能とする手法が有効である．例えば，シグニチャマッチング (signature matcing) [116] や仕様マッチング (specification matching) [117] を類似クラス探索手続きやメソッド合成手続きに組み込むことで，変更を適用するメソッドの範囲を広げることを考えている．これらのマッチング手法では，シグニチャや (一階の述語論理で書かれた) 仕様の正確な一致だけでなく，それらの緩やかな一致 relaxed match) を判定可能である．さらに，生成したメソッドが開発者の要求する機能を満たすことを検証するため，クラス内の各メソッドに表明 (例えば， Eiffel 言語 [78] における，前提条件，終了条件，クラス不変表明) を付け，メソッド合成と同時に表明の合成を行う手法を，本クラス生成手法に組み込む試みを行っている．

(3) メソッドの動作に関する互換性

本クラス生成手法は，インターフェイスが一致するメソッドどうしだけを合成対象とする．つまり，あるメソッドに関する過去の変更が，異なるインターフェイスを持つメソッドに適用されることはない．このため，開発者の要求する機能を完全に満たさないクラスが生成されてしまうことがある．例えば， 6.6節で述べたクラス生成例において，新規クラス DrawAreaScrollbar を実際に実行すると，図形描画時にマウスで指定した座標と実際に図形が描画される座標にずれがあることに気づく．これは， DrawArea のメソッド `handleEvent` (節点 r14–r17) が，そのまま DrawAreaScrollbar に継承されたために生じる．座標のずれを取り除くには，節点 r16 の `e.x` と `e.y` に対して，メソッド `paint` に適用した座標変換と同様の変更 (`e.x` → `e.x+transX`, `e.y` → `e.y+transY`) を適用し， `handleEvent` を再定義する必要がある．このように，本生成手法は，新規に生成したメソッドの動作が，新規生成クラスの祖先クラスの既存メソッドの動作と互換性を持つことまでは保証していない．

このような変更を自動的に行うことは一般的に難しいため，新規に生成したクラスを検査する仕組みが必須である．特定の動作に関連を持つコードだけを抽出し，変更の不要なメソッドを特定するという点で，クラス階層を扱えるプログラムスライシング [53, 103] が利用可能である．また．クラス内で関連のあるメソッドをグループ化しておくことにより，変

更が必要となるメソッドを特定する手法 [50, 100] も有効である．

(3) 多相性とメソッド複写

多相性 (polymorphism) を有するメソッドに対して，6.4節で述べたメソッド複写操作を単純に適用することはできない．なぜなら，動的束縛 (dynamic binding) により実行時に呼び出される可能性を持つ複数のメソッドを同一クラス内に複写することはできないからである．このため，本クラス生成手法では，複写対象のメソッドを含むインスタンスに対して，その生成元クラスが静的かつ一意に決定可能であることを前提とし，多相性を有するメソッドが複写対象となるとき新規クラスの生成を行わない．複写対象でないメソッドが多相性を有する場合は，新規クラスの生成が行われる．

本生成手法において，多相性を扱う方法の一つとして，多相性を有するメソッドを直接複写するのではなく，複写メソッドの多相性を保存したままで，束縛される可能性のある関連クラス全体を複写することが考えられる．複写対象のクラスを特定する場合，クラス階層スライシング [103] が利用できる．

(4) 本手法の実用性

本章では，本生成手法によりクラスの自動生成が成功する一例を示しただけである．実際のプログラミングにおいて，どの程度生成に成功するのか，さらに，新規生成クラスが開発者の負担をどの程度軽減可能かを確認するためには，実在するクラスライブラリに対して本手法を適用し，その適用結果を評価することが必須である．

本クラス生成手法により生成される新規クラスの有用性の一部は，オブジェクト指向メトリクス [54] により評価できると考えている．例えば，新規クラスにおけるメソッドおよび変数の数，追加再定義，継承メソッドの数から，本生成手法の有効性を確認することができる．さらに，これらの評価基準は，本生成手法における類似クラスの特定，あるいは，新規生成クラスの候補から妥当なクラスを選択する際の指標に用いることもできる．

(5) クラス生成ツールの実現

本クラス生成手法の計算量を見積もる際に参考となる情報として，手続き呼出しを含むプログラムを合成するアルゴリズムの計算量が文献 [20] に示されている．これによると，SDG からソースコードを再構成する過程を除いて，合成アルゴリズムの計算コストは SDG を構築するコストに対して多項式時間で抑えられる (ソースコード再構成部分については，

NP 完全となることが指摘されている [40]). オブジェクト指向プログラムに対する ClDG と手続き型プログラムに対する SDG は，ClDG が多相的なメソッド呼出しを含むことを除いて，類似している．よって，ClDG を構築するコストと SDG を構築するコストは，ほぼ同じであると考えられる．さらに，ClDG を操作するアルゴリズム (例えば，スライシングや同形写像比較) は，SDG を操作するアルゴリズムと等しい．

しかしながら，本クラス生成手法を実現するアルゴリズムは，変更差異の特定あるいは合成を，対象となるメソッドから抽出したスライスおよび部分グラフのすべての組合せに対して行うため，その試行回数はかなり多い．さらに，このツールは，プログラムの記述とクラスの生成を繰り返すインタラクティブなプログラミング開発で用いられることが多いと考えられ，頻繁にクラスを生成することが求められる．よって，本クラス生成手法を実現するツールを構築する際には，ツールの性能に関する検討および計算量に関するアルゴリズムの改良が必要である．

6.9 あとがき

オブジェクト指向プログラミングにおいて，開発者が容易にプログラムを作成できるようにするためには，少ない変更で再利用可能なクラスを提供する必要がある．本章では，開発者が再利用している参照クラスで呼出しメソッドが定義されていない場合，参照クラスのメソッドとクラス変更履歴から，開発者が要求するメソッドを持つ新規クラスを自動生成する手法を提案した．さらに，本手法を実現する，変更差異特定手続き，類似クラス探索手続き，変更差異合成手続きの詳細を述べ，実際にこれらの手続きを用いて新規クラスが生成できることを示した．本論文で取り上げた例では，開発者の要求するメソッドを持つ新規クラスが得られ，このクラスはコンストラクタの追加だけで実行可能であった．このように，本手法により，継承におけるメソッドや変数の追加および再定義という開発者の負担軽減が期待できる．

第 7 章

結言

本論文は，筆者が日本電信電話株式会社 (NTT) ソフトウェア研究所において行った，プログラム依存解析に基づくソフトウェアの部品化再利用に関する研究の総合報告である．

ソフトウェア再利用は，ソフトウェア開発における生産性と保守性を飛躍的に向上させる効果を持つといわれており，その技術に対する期待は大きい．本論文では，再利用対象をソースコードとし，その部品化による再利用を支援する手法，および，それらの手法の有効性について論じた．以下，各章ごとにその内容をまとめる．

1章では，ソフトウェアの部品化再利用における問題と，部品作成および部品変更に対する要求について述べた．次に，部品作成および部品変更に対する従来手法を紹介し，従来手法において残されている課題に対する本研究のアプローチを示した．さらに，本論文で提唱する 4 つの手法，1) 区間限定プログラムスライシングを用いた部品作成手法，2) 部品変更履歴に基づく重み付き依存グラフを用いた部品洗練手法，3) 変更の模倣による部品の能動的変化手法，4) メソッド合成による新規クラスの自動生成手法の概略を述べた．

2章では，本論文で提唱する 4 つの手法で共通して用いるプログラム依存解析技術についてまとめた．具体的には，手続き型プログラムにおける制御の流れを表現する制御フローグラフ，および，プログラム文間の依存関係を表現するプログラム依存グラフの定義を示した．さらに，依存関係に基づき，着目する変数に関連を持つ文集合をもとのプログラムから抜き出すプログラムスライシング技法，および，複数のプログラムを依存関係を保存したまま自動統合する技法について述べた．

3章では，区間限定スライシングを用いた部品作成手法を提案し，この部品作成手法に基づき構築した部品作成システムを用いて，実際のプログラムからソフトウェア部品を作成した際の実験結果について考察を述べた．具体的には，プログラムの任意の区間をスライシング対象として指定可能となるように従来のスライスを拡張した，順方向および逆方向区間限

定スライスを定義し，これらのスライスを用いて既存のプログラムから再利用可能なソフトウェア部品を作成する手法を提案した．本部品作成手法は，抽出した部品が実行可能性と抽出等価性という部品に不可欠な性質を満たすことを保証しながら，部品作成対象プログラムに対して任意の区間を指定して，部品抽出が可能であるという点で従来の手法より有利である．また，本部品作成手法において部品作成者は区間と変数を指定するだけでよく，ソフトウェア部品の作成は半自動である．よって，ソフトウェア開発者の部品作成に対する負担を軽減できる．さらに，本部品作成手法では部品理解を支援するために，実行条件例と利用例を部品利用者に提示する．これらの実行条件例と利用例をそれぞれ部品本体と組み合わせて得られるプログラムは実行可能性と抽出等価性を満たすので，部品の実行動作や利用方法の確認が容易となる．

4章では，過去の部品変更の履歴に基づきプログラム内の依存関係に割り付けた重みを用いることで，既存のプログラムから抽出した部品を将来再利用される可能性が高い部品に自動洗練する手法を提案した．さらに，この部品洗練手法に基づき構築した部品洗練システムを用いて，実際にソフトウェア部品を洗練した結果について考察を述べた．本洗練手法では，頻繁に再利用されたソースコード断片は開発者が将来も同じ形で再利用する可能性が高い部分，また，頻繁に変更を受けたソースコード断片は開発者が将来も同様の変更を行う可能性が高い部分であると仮定する．このような仮定に基づき，重み付きプログラム依存グラフにおける節点間の依存関係の強度を重みで定義し，開発者が変更後に再利用した部品の形および部品変更前後の部品の形の変化に基づき重みを変動させる．このようにして蓄積した依存関係矢印の重みを部品洗練の指標として用い，もとの部品ソースコードから重み値が大きい強依存関係で結合されているソースコード断片を抜き出す．抜き出したソースコード断片に対して，重みから計算した洗練コストにより洗練の妥当性を判定することで，開発者の再利用ドメインに適した洗練部品の形を決定することが可能である．本洗練手法を導入することで，部品提供者が微妙に異なる多種多様な部品をライブラリに用意しておく必要がなくなり，さらに，再利用時に開発者が部品を適時変更する負担も減少する．

5章では，従来とまったく異なる新しい部品として，部品検索時に自ら形を変える能動的部品を提唱し，その変化メカニズム，および，具体的アルゴリズムと変化例を示した．機能分割および機能交換のためのメカニズムを備えた能動的部品は，要求に応じて自分自身を機能分割する，および，他の部品が受けた変更を模倣するために自分の周りに存在する他の能動的部品の変更事例を取得し，機能の一部を他の部品と交換する．機能分割と機能交換により，要求や環境に応じて自動的かつ動的に変化可能となる．能動的変化メカニズムは，プロ

グラムスライシングとグラフのラベル付き同形写像比較を用いたプログラム合成アルゴリズムにより実現した．さらに，本章で提案したアルゴリズムでは，変更箇所および交換箇所を特定する際に補スライスとラベルの抽象化，スライスを合成する際に依存関係矢印の付けかえを導入することで，従来アルゴリズムに比べて幅広いソースコードに対して変化が適用可能である．提案アルゴリズムに基づき構築した能動的部品のプロトタイプを用いて，動作確認と適用実験を行った結果より，1) ライブラリに存在しない部品を要求した場合においても，変化後の部品が部品利用者の要求を満たす可能性があり，利用者が適時部品を変更する負担が軽減される，2) 開発ドメインの特性を分析する必要がなく，多種多様にわたるすべての部品をライブラリに用意する必要がない，という能動的部品の利点が確認できた．

6章では，開発者の過去のクラス変更の履歴を用いることで，オブジェクト指向プログラムにおける新規クラスを自動生成する手法を提案した．本生成手法では，継承を用いてプログラムを作成した際，親クラスとその後継者クラスが持つメソッド間の変更差異を特定し，その履歴を親クラスに蓄積する．開発者が参照しているクラスにおいて，開発者の要求した呼出しメソッドが定義されていない場合，このメソッドに関連する変更差異を持つ類似クラスを見つけ，その差異を参照クラスに存在するメソッドと合成することで，呼出しメソッドを生成する．類似クラスは，ライブラリにおけるクラス階層関係とメソッドの名前およびシグニチャの等価判定に基づき決定する．また，変更差異の特定と合成は，5章に示す能動的部品の機能交換変化メカニズムを拡張することで実現した．本手法により生成される新規クラスは，親クラスに対して単純に差分コードを追加するだけでは生成不可能なメソッドを含む．また，依存関係に基づきコードを合成する本生成手法は，プログラム内のコードを命令単位で無条件に入れ換える手法と異なり，合成により追加した変更差異がもとのプログラムに矛盾しないことを保証する．本生成手法をオブジェクト指向プログラム開発に導入することで，継承におけるメソッドや変数の追加および再定義という開発者の負担が軽減され，開発者が容易にプログラムを作成可能となる．

謝辞

本論文をまとめるにあたり，懇切なる御指導，御鞭撻，ならびに，御助言を頂いた早稲田大学理工学部 深澤良彰教授に謹んで感謝の意を表します．本研究の一部は筆者が早稲田大学理工学部在学中に始めたものであり，深澤教授には，本研究のきっかけを与えて頂いたとともに，研究に対する基本的な姿勢を直接御指導頂きました．また，本論文の執筆にあたり，有益なる御教示，御示唆を頂きました早稲田大学理工学部 大須賀節雄教授，村岡洋一教授，ならびに，上田和紀教授に深く感謝の意を表します．

本論文は，筆者が日本電信電話株式会社 (NTT) ソフトウェア研究所において行った研究成果をまとめたものであり，本研究の機会を与えてくださった，細谷僚一 前所長 (現， NTT ソフトウェア株式会社取締役営業本部長)，伊土誠一 所長に感謝致します．また，研究を遂行するにあたり，御指導御鞭撻を頂いた，後藤滋樹早稲田大学理工学部教授 (当時，広域コンピューティング研究部長)，市川晴久 広域コンピューティング研究部長，伊藤正樹 NTT Multimedia Communication Laboratories 所長 (General Manager) (当時，知的ソフトウェア研究グループリーダ)，ならびに，後藤厚宏 知的ソフトウェア研究グループリーダに感謝致します．

高橋直久 光ネットワーク研究所分散ネットワークシステム研究部並列分散アーキテクチャ研究グループリーダ (当時，ソフトウェア研究所広域コンピューティング研究部超並列プログラミング研究グループリーダ)，ならびに，島健一 主任研究員には，本研究の当初より多岐にわたり御指導，ならびに，御討論頂きました．心から感謝致します．また，数多くの有益な御討論，ならびに，御助言を頂いた，内藤昭三 主任研究員をはじめとする広域コンピューティング研究部の皆様に感謝致します．

また，筆者が早稲田大学理工学部在学時に，懇切なる御指導御鞭撻を頂いた (故) 門倉敏夫 早稲田大学名誉教授，ならびに，小野康一氏 (現在， IBM 東京基礎研究所勤務) に深く感謝致します．

最後に，著者の論文執筆を支援してくれた家族に感謝致します．

参考文献

[1] Salwa K. Abd-El-Hafiz, Victor R. Basili, and Gianluigi Caldiera. Towards automated support for extraction of reusable components. In *Proc. Intl. Conf. Software Maintenance*, pp. 212–219, October 1991.

[2] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proc. ACM Conf. Programming Language Design and Implementation*, pp. 246–256, June 1990.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] 青山幹雄. コンポーネントウェア: 部品組立て型ソフトウェア開発技術. 情報処理, Vol. 37, No. 1, pp. 71–79, January 1996.

[5] Guillermo Arango. Domain analysis methods. In Wilhelm Schäfer, Rubén Pieto-Díaz, and Masao Matsumoto, editors, *Software Reusability*, chapter 2. Ellis Horwood, 1994.

[6] Guillermo Arango, Ira D. Baxter, Peter Freeman, and Christopher Pidgeon. TMM: Software maintenance by transformation. *IEEE Software*, Vol. 3, No. 5, pp. 27–39, May 1986.

[7] Bobert S. Arnold. Software restructuring. In *Proc. IEEE,* Vol. 77, No. 4, pp. 607–617, April 1989.

[8] 麻生英樹. ニューラルネットワーク情報処理. 産業図書, 1988.

[9] Thomas Ball and Susan B. Horwitz. Slicing programs with arbitrary control flow. In *Proc. Intl. Workshop on Automated and Algorithmic Debugging,* Lecture Notes in Computer Science, Vol. 749, pp. 206–222, May 1993.

[10] Thomas Ball, Susan B. Horwitz, and Thomas W. Reps. Correctness of an algorithm for reconstituting a program from a dependence graph. Technical Report TR-947, Computer Science Dept., University of Wisconsin, Madison, WI, July 1990.

[11] Don Batory and Bart J. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Trans. Software Engineering*, Vol. 23, No. 2, pp. 67–82, February 1997.

[12] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Software Engineering and Methodology*, Vol. 1, No. 4, pp. 355–398, October 1992.

[13] Ira D. Baxter. Design maintenance systems. *Communications of the ACM*, Vol. 35, No. 4, pp. 73–89, April 1992.

[14] Jon Beck and David Eichmann. Program and interface slicing for reverse engineering. In *Proc. Intl. Conf. Software Engineering*, pp. 509–518, May 1993.

[15] Jean-Francois Bergeretti and Bernand A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Programming Languages and Systems*, Vol. 7, No. 1, pp. 37–61, January 1985.

[16] Valdis Berzins. Software merge: Models and methods for combining changes to programs. *Journal of System Integration*, Vol. 1, No. 2, pp. 121–141, August 1991.

[17] Ted J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, Vol. 22, No. 7, pp. 36–49, July 1989.

[18] Ted J. Biggerstaff and Charles Richter. Reusability framework, assessment, and directions. *IEEE Software*, Vol. 4, No. 2, pp. 41–49, March 1987.

[19] David Binkley and Keith Brian Gallagher. Program slicing. *Advances in Computers,* Vol. 43, 1996.

[20] David Binkley, Susan B. Horwitz, and Thomas W. Reps. Program integration for language with procedure calls. *ACM Trans. Software Engineering and Methodology*, Vol. 4, No. 1, pp. 3–35, January 1995.

[21] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 303–311, October 1990.

[22] Alan W. Brown, editor. *Component-Based Software Engineering*. IEEE Computer Society Press, 1996.

[23] Gianluigi Caldiera and Victor R. Basili. Identifying and qualifying reusable components. *IEEE Computer*, Vol. 24, No. 2, pp. 61–70, February 1991.

[24] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, and G. A. Di Lucca. Software salvaging based on conditions. In *Proc. Intl. Conf. Software Maintenance*, pp. 424–433, September 1994.

[25] Gerardo Canfora, Aniello Cimitile, and Malcolm Munro. RE^2: Reverse engineering and reuse re-engineering. Technical report, School of Engineering and Computer Science, University of Durham, 1992.

[26] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, Vol. 7, No. 1, pp. 13–17, January 1990.

[27] Filippo Cutillo, Piernicola Fiore, and Giuseppe Visaggio. Identification and extraction of "domain independent" components in large programs. In *Proc. Working Conf. Reverse Engineering*, pp. 83–92, May 1993.

[28] Michael F. Dunn and John C. Knight. Automating the detection of reusable parts in existing software. In *Proc. Intl. Conf. Software Engineering*, pp. 381–390, May 1993.

[29] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Programming Languages and Systems*, Vol. 9, No. 3, pp. 319–349, July 1987.

[30] Peter Freeman. A conceptual analysis of the Draco approach to constructing software systems. *IEEE Trans. Software Engineering*, Vol. 13, No. 7, pp. 830–844, July 1987.

[31] Peter Freeman, editor. *Tutorial: Software Reusability*. IEEE Press, 1987.

[32] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Software Engineering*, Vol. 17, No. 8, pp. 751–761, August 1991.

[33] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. (本位田 真一，本田和樹 監訳: デザインパターン，ソフトバンク (1995)).

[34] Joseph Goguen. Principles of parameterized programming. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability Volume I*, pp. 159–225. Addison Wesley, 1989.

[35] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[36] Mary Jean Harrold. Efficient computation of interprocedural definition-use chains. *ACM Trans. Programming Languages and Systems*, Vol. 16, No. 2, pp. 175–204, March 1994.

[37] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. In *Proc. ACM Symp. Foundations of Software Engineering*, pp. 154–163, December 1994.

[38] Susan B. Horwitz, Thomas Ball, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Programming Languages and Systems*, Vol. 12, No. 1, pp. 26–60, January 1990.

[39] Susan B. Horwitz, Phil Pfeiffer, and Thomas W. Reps. Dependence analysis for pointer variables. In *Proc. ACM Conf. Programming Languages Design and Implementation*, pp. 28–40, June 1989.

[40] Susan B. Horwitz, Jan Prins, and Thomas W. Reps. Integrating noninterfering versions of programs. *ACM Trans. Programming Languages and Systems*, Vol. 11, No. 3, pp. 345–387, July 1989.

[41] Susan B. Horwitz, Jan Reps, and Thomas W. Reps. On the adequacy of program dependence graphs for representing programs. In *Proc. ACM Symp. Principles of Programming Languages*, pp. 146–157, January 1988.

[42] Susan B. Horwitz and Thomas W. Reps. Efficient comparison of program slices. *Acta Informatica*, Vol. 28, pp. 713–732, June 1991.

[43] Susan B. Horwitz and Thomas W. Reps. The use of program dependence graphs in software engineering. In *Proc. Intl. Conf. Software Engineering*, pp. 392–411, May 1992.

[44] Ralph E. Jonson. Documenting frameworks using patterns. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 63–76, October 1992.

[45] Ralph E. Jonson and Brian Foote. Designing reusable classes. *Journal on Object-Oriented Programming*, Vol. 1, No. 2, June/July 1988.

[46] Hyeon Soo Kim, In Sang Chung, and Yong Rae Kwon. An approach to restructuring programs through program slicing. In *Proc. Joint Conf. Software Engineering*, pp. 307–314, November 1993.

[47] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, Vol. 29, No. 3, pp. 155–163, October 1988.

[48] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, Vol. 24, No. 2, June 1992.

[49] David J. Kuck, R. H. Kuhn, David A. Padua, and Michael Wolfe. Dependence graphs and compiler optimization. In *Proc. ACM Symp. Principles of Programming Languages*, pp. 207–218, January 1981.

[50] John Lamping. Typing the specialization interface. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 201–214, September 1993.

[51] Filippo Lanubile and Giuseppe Visaggio. Function recovery based on program slicing. In *Proc. Intl. Conf. Software Maintenance*, pp. 396–404, September 1993.

[52] Filippo Lanubile and Giuseppe Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Trans. Software Engineering*, Vol. 23, No. 4, pp. 246–259, April 1997.

[53] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *Proc. Intl. Conf. Software Engineering*, pp. 495–505, March 1996.

[54] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics*. Prentice Hall, 1994. (宇治邦昭監訳: オブジェクト指向ソフトウェアメトリクス，トッパン (1995)).

[55] 丸山勝久, 小野康一, 門倉敏夫, 深澤良彰. プログラム変更に対する正当性検証技法の適用. 情報処理学会 第 44 回全国大会 2K-2, pp. 293–294, March 1992.

[56] 丸山勝久, 小野康一, 門倉敏夫, 深澤良彰. プログラム変更に対する正当性検証技法と分割技法の適用. 情報処理学会 研究報告， SIG-SE-90, pp. 89–96, February 1993.

[57] 丸山勝久, 高橋直久. プログラムスライシングによるソフトウェア部品の作成. 情報処理学会 第 48 回全国大会 7J-5, pp. 319–320, March 1994.

[58] 丸山勝久, 高橋直久. プログラムスライシングによるソフトウェア部品の作成. 情報処理学会 研究報告， SIG-SE-98, pp. 1–8, May 1994.

[59] 丸山勝久, 島健一, 高橋直久. 重み付き依存グラフを用いたソフトウェア部品の洗練. 情報処理学会 第 49 回全国大会 5M-8, pp. 193–194, September 1994.

[60] 丸山勝久, 島健一. ソースコード再利用における能動的部品変化メカニズム. 情報処理学会 研究報告， SIG-SE-102, pp. 71–76, January 1995.

[61] 丸山勝久, 島健一, 高橋直久. 区間限定スライスを用いた再利用部品抽出手法の評価. 情報処理学会 第 51 回全国大会 5M-10, pp. 147–148, September 1995.

[62] 丸山勝久, 島健一, 高橋直久. 区間限定スライスを用いた部品作成システムの評価. 情報処理学会 研究報告， SIG-SE-105, pp. 49–56, September 1995.

[63] 丸山勝久, 島健一. 能動的部品における機能交換変化メカニズム. ソフトウェア科学会 FOSE'95，ソフトウェア工学の基礎 II，近代科学社, pp. 31–40, December 1995.

[64] 丸山勝久, 島健一. ソースコード再利用における能動的部品変化メカニズム. 情報処理学会 論文誌, Vol. 37, No. 12, pp. 2334–2351, December 1996.

[65] 丸山勝久, 高橋直久. 区間設定可能なプログラムスライシングを用いたソフトウェア部品の作成. 情報処理学会 論文誌, Vol. 37, No. 4, pp. 520–535, April 1996.

[66] 丸山勝久, 島健一, 高橋直久. 重み付き依存グラフを用いたソフトウェア部品の洗練. ソフトウェア科学会 FOSE'96，ソフトウェア工学の基礎 III，近代科学社, pp. 26–33, December 1996.

[67] 丸山勝久, 島健一. オブジェクト指向における変更履歴を用いた新規クラスの自動生成. 情報処理学会 第 54 回全国大会 1K-3, pp. 289–290, March 1997.

[68] 丸山勝久, 島健一. オブジェクト指向プログラミングにおける変更履歴を用いた新規クラスの自動生成. 情報処理学会 オブジェクト指向'97 シンポジウム，オブジェクト指向最前線，朝倉書店, pp. 76–83, July 1997.

[69] 丸山勝久, 島健一. 分散ネットワーク環境に対する能動的変化メカニズムの適用. 情報処理学会 第 55 回全国大会 2S-5, pp. 319–320, September 1997.

[70] 丸山勝久, 島健一. 変更履歴に基づく重みを用いたメソッドの再構成. ソフトウェア科学会 FOSE'97，ソフトウェア工学の基礎 IV，近代科学社, pp. 143–150, December 1997.

[71] Katsuhisa Maruyama and Ken-ichi Shima. A new class generation mechanism by method integration. In *Proc. Intl. Conf. Software Reuse*, pp. 196–205, June 1998.

[72] 丸山勝久, 島健一. 部品変更履歴に基づく重み付き依存グラフを用いた部品の洗練. 情報処理学会 論文誌, Vol. 39, No. 3, pp. 725–740, March 1998.

[73] Katsuhisa Maruyama and Ken-ichi Shima. New software components with an autonomous changing mechanism. In *Proc. Asia-Pacific Software Engineering Conf.*, pp. 154–165, December 1996.

[74] Katsuhisa Maruyama and Ken-ichi Shima. A mechanism for automatically and dynamically changing software components. In *Proc. ACM Symp. Software Reusability,* Software Eng. Notes, Vol. 22, No. 3, pp. 169–180, May 1997.

[75] Katsuhisa Maruyama and Ken-ichi Shima. Automatic method refactoring using weighted dependence graphs. In *Proc. Intl. Conf. Software Engineering*, May 1999. (to appear).

[76] Carma McClure. *The Three Rs of Software Automation: Re-engineering, Repository, Reusability.* Prentice Hall, 1992.

[77] M. D. McIlroy. Mass-produced software components. In J. M. Buxton, P. Naur, and B. Randell, editors, *Software Engineering Concepts and Techniques, Proc. 1968 NATO Conf. Software Eng.*, pp. 88–98. Petrocelli/Charter, 1969.

[78] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988. (二木厚吉 監訳: オブジェクト指向入門，アスキー出版局 (1990)).

[79] 直井邦彰, 高橋直久. 経路依存フローグラフを用いた infeasible path 検出法. 電子情報通信学会論文誌, Vol. J76-D-I, No. 8, pp. 429–439, August 1995.

[80] 直井邦彰, 高橋直久. 経路依存フローグラフを用いたプログラムスライシング. 電子情報通信学会論文誌, Vol. J76-D-I, No. 7, pp. 607–621, July 1995.

[81] James M. Neighbor. *Software Construction Using Components.* PhD thesis, Dept. of Information and Computer Science, University of California, Irvine, 1980.

[82] James M. Neighbor. Draco: A method for engineering reusable software systems. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability Volume I*, pp. 295–319. Addison Wesley, 1989.

[83] James M. Neighbor. An assessment of reuse technology after ten years. In *Proc. Intl. Conf. Software Reuse*, pp. 6–13, November 1994.

[84] Jim Q. Ning, Andre Engberts, and Wojtek Kozaczynski. Recovering reusable components from legacy systems by program segmentation. In *Proc. Working Conf. Reverse Engineering*, pp. 64–82, May 1993.

[85] 西村進. PDI アルゴリズム: プログラム差分合成のためのアルゴリズム. コンピュータソフトウェア, Vol. 12, No. 5, pp. 85–99, September 1995.

[86] 小野康一, 丸山勝久, 深澤良彰. プログラム変更に対する正当性検証技法と分割技法の適用. 電子情報通信学会 論文誌, Vol. J77-D-I, No. 11, pp. 747–758, November 1994.

[87] Harold Ossher and William Harrison. Combination of inheritance hierarchies. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 25–40, October 1992.

[88] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proc. ACM Symp. Practical Programming Development Environments,* ACM SIGPLAN Notices, Vol. 19, No. 5, pp. 177–184, May 1984.

[89] David A. Padura and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, Vol. 29, No. 12, pp. 1184–1201, December 1986.

[90] Hemant D. Pande, William A. Landi, and Barbara G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Trans. Software Engineering*, Vol. 20, No. 5, pp. 385–403, May 1994.

[91] Rubén Pieto-Díaz. Status report: Software reusability. *IEEE Software*, Vol. 10, No. 3, pp. 61–66, May 1993.

[92] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Software Engineering*, Vol. 16, No. 9, pp. 965–979, September 1990.

[93] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1994. (佐藤啓太，金澤典子 訳: デザインパターンプログラミング, トッパン (1996)).

[94] Gregg Rothermel and Mary Jean Harrold. Selecting regression tests for object-oriented software. In *Proc. Intl. Conf. Software Maintenance*, pp. 14–25, August 1994.

[95] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991. (羽生田栄一監訳: オブジェクト指向方法論 OMT，トッパン (1992)).

[96] 下村隆夫. Program slicing 技術とテスト，デバッグ，保守への応用. 情報処理, Vol. 33, No. 9, pp. 1078–1086, September 1992.

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

[98] 篠崎信雄. 統計解析入門, 第 11 章, pp. 253–268. サイエンス社, 1994.

[99] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 38–45, September 1986.

[100] Raymie Stata and John V. Guttag. Modular reasoning in the presence of subclassing. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 200–214, October 1995.

[101] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[102] Frank Tip. A survey of program slicing techniques. Technical Report CS-R9438, Centrum voor Wiskunde en Informatica, July 1994.

[103] Frank Tip, Jong-Deck Chio, John Field, and G. Ramalingam. Slicing class hierarchies in C++. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 179–197, October 1996.

[104] Will Tracz, editor. *Tutorial: Software Reuse: Emerging Technology*. IEEE Press, 1988.

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

[106] Michael VanHilst and David Notkin. Using C++ templates to implement role-based designs. In *Proc. Intl. Symp. Object Technologies for Advanced Software*, pp. 22–37, March 1996.

[107] G A Venkatesh. The semantic approach to program slicing. In *Proc. ACM Conf. Programming Language Design and Implementation*, pp. 107–119, June 1991.

[108] Richad C. Waters. The programmer's apprentice: A session with KBEmacs. *IEEE Trans. Software Engineering*, Vol. 11, No. 11, pp. 1296–1320, November 1985.

[109] Mark Weiser. Program slicing. In *Proc. Intl. Conf. Software Engineering*, pp. 439–449, May 1981.

[110] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, Vol. 25, No. 7, pp. 446–452, July 1982.

[111] Mark Weiser. Program slicing. *IEEE Trans. Software Engineering*, Vol. 10, No. 4, pp. 352–357, July 1984.

[112] Robert S. Williams. Learning to program by examining and modifying cases. In *Proc. Case-Based Reasoning Workshop*, pp. 463–474, March 1988.

[113] Rebecca J. Wirfs-Brock and Ralph Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, Vol. 33, No. 9, pp. 104–124, September 1990.

[114] Wuu Yang. A new algorithm for semantics-based program integration. Technical Report TR-962, Computer Science Dept., University of Wisconsin, Madison, WI, August 1990.

[115] Wuu Yang, Susan B. Horwitz, and Thomas W. Reps. A program integration algorithm that accommodates semantics-preserving transformations. In *Proc. ACM Symp. Software Development Environments*, pp. 133–143, December 1990.

[116] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: A tool for using software libraries. *ACM Trans. Software Engineering and Methodology*, Vol. 4, No. 2, pp. 146–170, April 1995.

[117] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Trans. Software Engineering and Methodology*, Vol. 6, No. 4, pp. 333–369, October 1997.

研究業績

種類別	題名，発表・発行掲載誌名，発表・発行年月，連名者
論文○	Automatic Method Refactoring Using Weighted Dependence Graphs Proc. IEEE Intl. Conf. Software Engineering, 1999 年 5 月 (to appear) Katsuhisa Maruyama, Ken-ichi Shima
論文○	A New Class Generation Mechanism by Method Integration Proc. IEEE Intl. Conf. Software Reuse, pp.196–205, 1998 年 6 月 Katsuhisa Maruyama, Ken-ichi Shima
論文○	部品変更履歴に基づく重み付き依存グラフを用いた部品の洗練 情報処理学会 論文誌, Vol.39, No.3, pp.725–740, 1998 年 3 月 丸山 勝久, 島 健一
論文○	A Mechanism for Automatically and Dynamically Changing Software Components Proc. ACM Symp. Software Reusability, Software Eng. Notes, Vol.22, No.3, pp.169–180, 1997 年 5 月 Katsuhisa Maruyama, Ken-ichi Shima
論文○	New Software Components with an Autonomous Changing Mechanism Proc. Asia-Pacific Software Eng. Conf., pp.154–165, 1996 年 12 月 Katsuhisa Maruyama, Ken-ichi Shima
論文○	ソースコード再利用における能動的部品変化メカニズム 情報処理学会 論文誌, Vol.37, No.12, pp.2334–2351, 1996 年 12 月 丸山 勝久, 島 健一
論文○	区間設定可能なプログラムスライシングを用いたソフトウェア部品の作成 情報処理学会 論文誌, Vol.37, No.4, pp.520–535, 1996 年 4 月 丸山 勝久, 高橋 直久
論文○	プログラム変更に対する正当性検証技法と分割技法の適用 電子情報通信学会 D-I, Vol.J77, No.11, pp.747–758, 1994 年 11 月 小野康一, 丸山勝久, 深澤良彰
(論文) (英訳)	Application of Verification Method and a Decomposition Method to Program Modification Systems and Computers in Japan, Vol.27, No.1, pp.12-26, 1996 年 11 月 Kouichi Ono, Katsuhisa Maruyama, Yoshiaki Fukazawa
講演	変更履歴に基づく重みを用いたメソッドの再構成 ソフトウェア科学会 FOSE'97, ソフトウェア工学の基礎 IV, 近代科学社, pp.143–150, 1997 年 12 月 丸山 勝久, 島 健一
講演	分散ネットワーク環境に対する能動的変化メカニズムの適用 情報処理学会 第 55 回全国大会 (2S-5), pp.319–320, 1997 年 9 月 丸山 勝久, 島 健一

種類別	題名，発表・発行掲載誌名，発表・発行年月，連名者
講演	オブジェクト指向プログラミングにおける変更履歴を用いた新規クラスの自動生成 情報処理学会 オブジェクト指向'97 シンポジウム, オブジェクト指向最前線, 朝倉書店, pp.76–83, 1997 年 7 月 丸山 勝久, 島 健一
講演	オブジェクト指向における変更履歴を用いた新規クラスの自動生成 情報処理学会 第 54 回全国大会 (1K-3), pp.289–290, 1997 年 3 月 丸山 勝久, 島 健一
講演 (パネル)	New Components with Mechanisms for Automatically and Dynamically Modifying Themselves Asia-Pacific Software Eng. Conf. Panel: If Software Reuse Can Lead IT, How?, 1996 年 12 月 Katsuhisa Maruyama
講演	重み付き依存グラフを用いたソフトウェア部品の洗練 ソフトウェア科学会 FOSE'96, ソフトウェア工学の基礎 III, 近代科学社, pp.26–33, 1996 年 12 月 丸山 勝久, 島 健一
講演	能動的部品における機能交換変化メカニズム ソフトウェア科学会 FOSE'95, ソフトウェア工学の基礎 II, 近代科学社, pp.31–40, 1995 年 12 月 丸山 勝久, 島 健一
講演	区間限定スライスを用いた部品作成システムの評価 情報処理学会 研究報告, SIG-SE-105, pp.49–56, 1995 年 9 月 丸山 勝久, 島 健一, 高橋 直久
講演	区間限定スライスを用いた再利用部品抽出手法の評価 情報処理学会 第 51 回全国大会 (5M-10), pp.147–148, 1995 年 9 月 丸山 勝久, 島 健一, 高橋 直久
講演	ソースコード再利用における能動的部品変化メカニズム 情報処理学会 研究報告, SIG-SE-102, pp.71–76, 1995 年 1 月 丸山 勝久, 島 健一
講演	重み付き依存グラフを用いたソフトウェア部品の洗練 情報処理学会 第 49 回全国大会 (5M-8), pp.193–194, 1994 年 9 月 丸山 勝久, 島 健一, 高橋 直久
講演	プログラムスライシングによるソフトウェア部品の作成 情報処理学会 研究報告, SIG-SE-98, pp.1–8, 1994 年 5 月 丸山 勝久, 高橋 直久
講演	プログラムスライシングによるソフトウェア部品の作成 情報処理学会 第 48 回全国大会 (7J-5), pp.319–320, 1994 年 3 月 丸山 勝久, 高橋 直久

種類別	題名，発表・発行掲載誌名，発表・発行年月，連名者
講演	プログラム変更に対する正当性検証技法と分割技法の適用 情報処理学会 研究報告, SIG-SE-90, pp.89–96, 1993 年 2 月 丸山 勝久, 小野 康一, 門倉 敏夫, 深澤 良彰
講演	プログラム変更に対する正当性検証技法の適用 情報処理学会 第 44 回全国大会 (2K-2), pp.293–294, 1992 年 3 月 丸山 勝久, 小野 康一, 門倉 敏夫, 深澤 良彰
その他 (論文○)	A Network Dependence Graph for Modeling Network Services and its Use in Fault Location 電子情報通信学会 E-D (採録決定, 1999 年 4 月) 丸山 勝久, 内藤 昭三
その他 (講演)	ネットワークに潜在する依存関係のグラフ表現とその応用 情報処理学会 第 56 回全国大会 (2K-4), pp.663–664, 1998 年 3 月 丸山 勝久, 内藤 昭三
その他 (講演)	ソフトウェアアーキテクチャの洗練 情報処理学会 ウインターワークショップ, pp.29–30, 1998 年 1 月 丸山 勝久
その他 (講演)	第 19 回ソフトウェア工学国際会議報告 情報処理学会 研究報告, SIG-SE-115, pp.119–124, 1997 年 8 月 丸山 勝久
その他 (講演)	能動的変化メカニズムを備えたソフトウェア部品 情報処理学会 ウインターワークショップ, pp.49–50, 1997 年 1 月 丸山 勝久, 島 健一
その他 (報告書)	ソースコード再利用における新規プログラム部品の自動生成手法 NTT 研究開発資料 第 11932 号, 1995 年 9 月 丸山 勝久, 島 健一
その他 (特許 出願)	変更ネットワークの統合における干渉検出方法および装置と 干渉検出プログラムを記録した記録媒体 平成 10 年 特許願 243635 号, 1998 年 8 月 丸山 勝久, 内藤 昭三
その他 (特許 出願)	ネットワークにおける障害箇所検出方法及び装置 及びネットワークにおける障害箇所検出プログラムを格納した媒体 平成 10 年 特許願 63284 号, 1998 年 3 月 丸山 勝久, 内藤 昭三
その他 (特許 出願)	プログラム部品自動生成方法および装置 平成 7 年 特許願 167737 号, 1995 年 7 月 丸山 勝久, 島 健一