

# 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

丸 山 勝 久<sup>†</sup> 島 健 一<sup>†</sup>

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明確にすることは難しく、必要なすべての部品をあらかじめ用意することは不可能である。本稿では、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形を変える能動的部品を提唱し、その変化メカニズムを提案する。能動的部品は要求や環境に応じて自ら変化するため、利用者が行う部品変更の負担を軽減し、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可能である。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機能交換変化の2種類である。能動的部品の変化メカニズムは、プログラム・スライシングとグラフのラベル付き同型写像比較を用いて、プログラム合成アルゴリズムにより実現する。提案アルゴリズムは、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を導入し、スライスを合成する際に依存関係矢印の付けかえることで、従来アルゴリズムに比べて幅広いソースコードに適用可能であるという利点を持つ。よって、能動的部品はさまざまな形の新規部品に変化可能である。本稿では、変化メカニズムの概要、および具体的アルゴリズムとその適用例を示し、部品変更と部品作成に対する負担軽減の効果について考察する。

## A Mechanism for Automatically and Dynamically Changing Software Components

KATSUHIKA MARUYAMA<sup>†</sup> and KEN-ICHI SHIMA<sup>†</sup>

In source code reuse, reusable software components must be frequently modified to create required programs, since they take fixed codes in a library. Not all components to be reused can be prepared in a library before reusing, since the software development domain is difficult to specify. We have developed a new mechanism for automatically and dynamically changing software components that are called *active components*. These active components have the capability of modifying themselves into codes that meet users' requirements and that are in accordance with their existing environments. Therefore, the active components do not require many user modifications when they are composed, and can be reused in unspecified or floating domains. The mechanism of active components provides two kinds of changes by: 1) decomposing their functions, and 2) partially exchanging their functions with modification histories of other active components. The mechanism is achieved by a program integration algorithm based on program slicing and labeled graph matching. Our proposed algorithm can be applied to more kinds of source codes by introducing the complements of slices, label abstraction, and reconnection of dependence edges than the conventional algorithms can. This paper describes the new mechanism and the algorithm that achieves the mechanism, and gives examples of changing active components.

### 1. はじめに

プログラムのソースコードを再利用対象とする際には、まず部品を作成し、それらの部品を検索、変更、合成することで目的のプログラムを作成する<sup>1)</sup>。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、部品を無変更で再利用可能なとき、プログラム開発コストが減少する。しかし、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つの問題が存在する。

#### 問題1 部品変更に関する問題。

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よいのか、どのように変更すればよいのかを判断するのは難しい。

<sup>†</sup> NTT ソフトウェア研究所  
NTT Software Laboratories

**問題 2 部品作成に関する問題.**

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、我々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 components) を提唱し、その変化メカニズムの検討を進めている<sup>2),3)</sup>。本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仮定し、部品の**変化単位**として部品の機能に着目する。本稿では、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 種類の能動的部品変化を提案する。

- (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** (Cde: decomposing)
- (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** (Cex: exchanging)

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過程において、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、このライブラリ内に自動的かつ動的に生成される。問題 1 に対して、部品利用者は、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という利点を持つ。また、問題 2 に対して、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という利点を持つ。

さらに、本稿では、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<sup>4)</sup>による機能分割手法により実現する。機能交換変化は、上記の機能分割手法とプログラム依存グラフ (PDG: Program Dependence Graph)<sup>5)</sup>のラベル付き同型写像比較<sup>6)</sup>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により実現する。我々は、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、**区間限定スライス**<sup>7)</sup>と、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加える。

依存関係に基づきソースコードを合成する本アルゴリズムは、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。

本稿の構成は次のとおりである。2 章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、3 章で本変化メカニズムを従来手法と比較する。次に、4 章で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 章で部品変化の具体的アルゴリズムを示し、6 章で変化例を紹介する。さらに、7 章で能動的部品の効果および課題について考察する。

## 2. 能動的部品の**変化メカニズム**

本章では、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の詳細を説明する。

### 2.1 能動的部品の構造と動作

1 章で述べた問題 1, 2 を解決するためには、再利用部品が次のような性質を持てばよい。

**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、我々は、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**

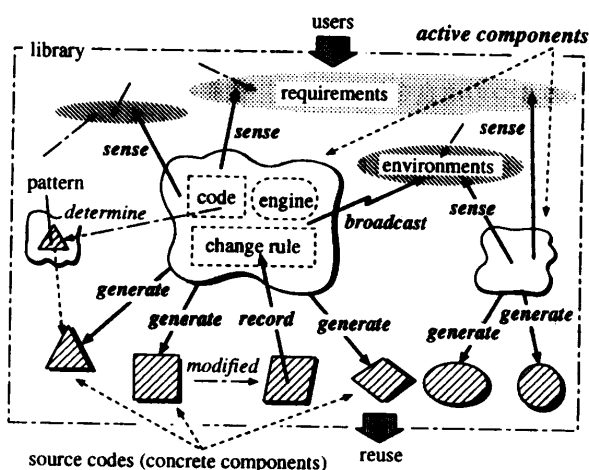


図1 能動的部品の構造および動作

Fig. 1 Structure and actions of active components.

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。

する。

- (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、適切に変更された変化後のソースコードを再利用し、目的のプログラムを作成する。

## 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可能である。プログラム・スライシングとは、着目する変数に影響を与える一部のコードだけを、もとのソースコードから抜き出すことである。本稿では、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結合されている。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結合しているスライスの記述から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

\* 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。

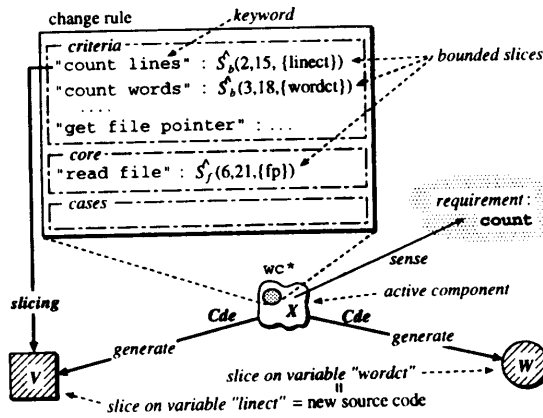


図2 機能分割変化

Fig. 2 Change by decomposing.

数を数える」機能を持つ  $V$  と「単語数を数える」機能を持つ  $W$  に機能分割変化可能である。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。

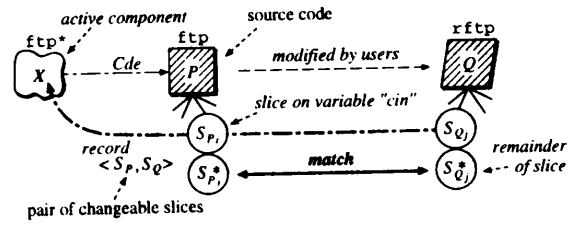
### 2.3 機能交換変化

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。我々は、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。

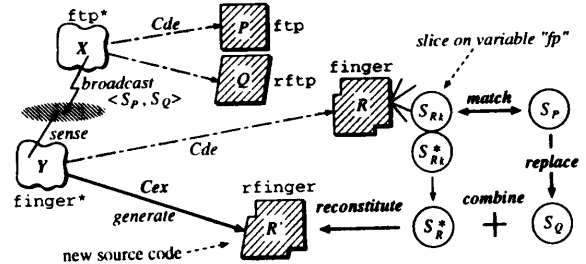
能動的部品は自分が生成したソースコードが部品利用者によって変更されたとき、その部品変更を変更事例として自分自身に記録する。変更事例を持つ能動的部品は、部品利用者がライブラリに要求を投入したとき、自分の持つ変更事例を他の能動的部品に広告する。能動的部品は、広告された他の部品の変更事例を感知し、自分の持つ機能の一部を他の部品の変更事例と交換および合成することで、新しい機能を持つ部品に機能交換変化可能である。このように、能動的部品は、機能交換変化により、他の部品が部品利用者から受けた変更を模倣することが可能で、自分が受ける可能性のある部品変更を自動的にかつ動的に行う。

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、ラベル付き同型写像比較を用いることで機能の一部だけを変



(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3 機能交換変化

Fig. 3 Change by exchanging.

更箇所として自動的に特定する。ラベル付き同型写像比較とは、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、同型を判断することである。本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、ラベル付き同型写像比較により、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型写像比較により、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。交換後のスライスと無交換部分を指す補スライスを合成することで、能動的部品は機能交換変化を達成する。

図3に機能交換変化の概要を示す。図3(a)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  $P$ ,  $Q$  からそれぞれ複数のスライス  $S_{Pi}$ ,  $S_{Qj}$  とその補スライス  $S_{Pi}^*$ ,  $S_{Qj}^*$  を自動的に作成する。次に、補スライス  $S_{Pi}^*$  と  $S_{Qj}^*$  のラベル付き同型写像比較により、交換可能スライス対  $\langle S_{Pi}, S_{Qj} \rangle$  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3(b)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  $S_P$  がラベル付き同型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  $S_Q$  と補スライス  $S_{Rk}^*$  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。

たとえば、能動的部品  $X$  から UNIX 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  $rftp^*$  に変更する。部品検索時、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  $rfinger^*$  にも変化する。

### 3. 従来手法との比較

本章では、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、提案アルゴリズムの改良点を述べる。

#### 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<sup>8)</sup>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て、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。

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、利用者の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、利用者の変更後の部品の機能を理解しやすく、無条件に多くの部品が生成されることもない。

#### 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、Horwitz, Prins, Reps による HPR アルゴリズム<sup>9)</sup>, Yang, Horwitz, Reps による YHR アルゴリズム<sup>10)</sup>, 西村による PDI アルゴリズム<sup>11)</sup>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3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 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)<sup>10)</sup>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。3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。

本稿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

☆ 代理サーバを中継する *socks ftp* および *socks finger*

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と、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。本稿では、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、スライスと補スライス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。

## 4. 諸 定 義

本章では、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示す。

### 4.1 対象ソースコード

本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 を単純化した言語のプログラムを扱う。対象ソースコードのBNFを付録 A.1 に示す。プログラム構造としては、代入文、複合文、条件文、繰り返し文、入力文、出力文を持つ。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。

本変化メカニズムでは、ソースコードをPDGで表現する。PDGはソースコード内の代入文および条件式をラベルとして割り当てた節点の集合と、各節点間の依存を表す矢印の集合からなる有向グラフである。PDGの節点  $p$  から節点  $q$  への任意の変数に関するデータ依存関係 (data dependence) を  $p \rightarrow_d q$ 、節点  $p$  から節点  $q$  への制御依存関係 (control dependence) を  $p \rightarrow_c q$  と表す。依存関係の種類を区別する必要のないとき、 $p \rightarrow q$  と表す。また、節点  $p$  で定義されている変数集合を  $def(p)$ 、参照されている変数集合を  $use(p)$ 、節点  $p$  のラベルを  $label(p)$  と表す。さらに、PDG  $G$  の節点集合を  $node(G)$ 、矢印集合を  $edge(G)$  と表す。

### 4.2 プログラム・スライシング

本変化メカニズムでは、従来の静的スライス<sup>12)</sup>を任意の対象区間が指定可能となるように拡張した区間限定スライス (bounded slice)<sup>7)</sup>を用いる。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。

区間限定スライスは、依存をたどる向きによって順方向区間限定スライス  $\hat{S}_f$  と逆方向区間限定スライス  $\hat{S}_b$  の2種類がある。順方向とはプログラム制御の流れる向きで、逆方向とはその反対の向きを指す。区間限定スライスを作成するためには、スライシング区間を担う上限節点  $n_u$  および下限節点  $n_l$  と着目する変数  $v$  を同時に指定する。区間限定スライスでは、制御フローグラフ (CFG: Control Flow Graph)<sup>13)</sup>の上限節点  $n_u$  と下限節点  $n_l$  の制約付き到達可能経路<sup>7)</sup>に含まれる節点集合  $CRP(n_u, n_l)$  がデータ依存関係

および制御依存関係をたどる範囲となる。順方向および逆方向区間限定スライスの定義を以下に示す。

**定義 1** 順方向区間限定スライス  $\hat{S}_f(n_u, n_l, v)$  とは,  $CRP(n_u, n_l)$  において PDG 上で依存関係をたどり, 上限節点  $n_u$  の変数  $v$  から到達可能な節点を抜き出し, 抜き出した節点に関わる節点を集めた集合である。

**定義 2** 逆方向区間限定スライス  $\hat{S}_b(n_u, n_l, v)$  とは,  $CRP(n_u, n_l)$  において PDG 上で依存関係をたどり, 下限節点  $n_l$  の変数  $v$  に到達可能な節点を集めた集合である。

本稿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, 本変化メカニズムでは, 区間限定スライスに対する補スライスを 2 種類用意する。補スライスの定義を以下に示す。

**定義 3** 区間限定補スライス  $\hat{S}^*(n_u, n_l, v)$  とは,  $CRP(n_u, n_l)$  から  $\hat{S}(n_u, n_l, v)$  に含まれる節点を取り除いた集合である。

**定義 4** 全区間補スライス  $S^*(n_u, n_l, v)$  とは, もとのソースコード全体の節点集合から  $\hat{S}(n_u, n_l, v)$  に含まれる節点を取り除いた集合である。

機能交換変化では, 変更箇所を特定する際に区間限定補スライス  $\hat{S}^*$ , スライスと補スライスを合成する際に全区間補スライス  $S^*$  を用いる。本稿では, 区間限定補スライスと全区間補スライスを区別する必要のないとき, これらを単に補スライスと呼ぶ。補スライスを作成する際, 単純にスライスに含まれる節点を PDG から削除すると, 接続先を持たない矢印が生じ, PDG が不完全となる。そこで, 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ヌル節点を導入し, 補スライスに含まれない節点だけを PDG 上でヌル節点に置き換え, 矢印はそのままにする。

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 4 に示す。本アルゴリズムでは, 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

```

procedure slicing( $G$ )
declare  $G$ : PDG;  $n_u, n_l$ : 節点;  $N_u, N_l$ : 節点集合;
         $V$ : 変数集合;  $SC$ : スライスと補スライス対の集合;
begin
   $N_u := \emptyset$ ;  $N_l := \emptyset$ ;  $SC := \emptyset$ ;
  for each  $n_u \in \text{node}(G)$  do
    if  $\text{def}(n_u) \setminus \text{use}(n_u) \neq \emptyset$  then
       $N_u := N_u \cup \{n_u\}$ ; fi od
  for each  $n_l \in \text{node}(G)$  do
    if  $\text{use}(n_l) \setminus \text{def}(n_l) \neq \emptyset$  then
       $N_l := N_l \cup \{n_l\}$ ; fi od
  for each  $n_l \in N_l$  do
     $V := \{ V' \mid V' \subseteq [ \text{use}(n_l) \setminus \text{def}(n_l) ] \}$ ;
    for each  $n_u \in N_u$  do
      for each  $v \in V$  do
         $S := \hat{S}_b(n_u, n_l, v)$ ;  $C := \hat{S}_f^*(n_u, n_l, v)$ ;
         $SC := SC \cup \{ \{S, C, n_u, n_l\} \}$ ;
      od od od
    return ( $SC$ );
  end

```

/\*  $A \setminus B$  は  $A, B$  の差集合 \*/

図 4 スライスおよび補スライスの作成アルゴリズム

Fig. 4 Slicing algorithm.

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 1 つに必ず含まれる。

#### 4.3 ラベル付き同型写像比較

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, PDG のラベル付き同型写像比較を用いる。ラベル付き同型写像比較により, 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。ラベル付き同型写像比較における同型の定義を以下に示す。

**定義 5** 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 はラベル付き同型写像比較において同型であるという<sup>6)</sup>。

図 5 にラベル付き同型写像比較により 2 つの PDG の節点およびラベルどうしの対応関係を求めるアルゴリズム matching を示す。このアルゴリズムは, 同型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 mapping\_nodes (図 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 matching\_nodes (図 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 abstraction\_label (図 5 [3], [4]) では 4.4 節で述べるラベルの抽象化を行い, 関数 matching\_labels (図 5 [5]) で抽象化したラベルどうしを比較する。関数 mapping\_nodes, matching\_nodes を付録 A.2, A.3

```

procedure matching( $G_n, G_m, flag$ )
declare  $G_n, G_m$ : PDG;  $j$ : 整数;  $J$ : 整数集合;
 $n, m$ : 節点;  $F_r[]$ ,  $F$ : 節点およびラベル対集合;
 $D[]$ : 節点集合;  $B[]$ : 節点对集合;
 $a_n[], a_m[]$ : ラベル;  $V_r[], V$ : 変数対集合;
 $flag$ : 抽象化(TRUE) / そのまま(FALSE);
begin
  if #node( $G_n$ )  $\neq$  #node( $G_m$ ) then
    return (FAILURE,  $\emptyset, \emptyset, \emptyset$ ); fi
  [1]  $D := \text{mapping\_nodes}(G_n, G_m)$ ;
  [2] ( $B, J$ ) := matching_nodes(node( $G_n$ ),  $D$ );
  if  $J = \emptyset$  then return (FAILURE,  $\emptyset, \emptyset, \emptyset$ ); fi
  for each  $j \in J$  do
     $F_r[j] := \emptyset$ ;  $V_r[j] := \emptyset$ ;
    for each ( $n, m$ )  $\in B[j]$  do
      if  $flag = \text{TRUE}$  then
        [3]  $a_n := \text{abstraction\_label}(\text{label}(n))$ ;
        [4]  $a_m := \text{abstraction\_label}(\text{label}(m))$ ;
        ( $F, V$ ) := matching_labels( $n, a_n, m, a_m$ );
        if  $F = \emptyset \vee V = \emptyset$  then  $J := J \setminus \{j\}$ ; fi
        else /*  $flag = \text{FALSE}$  */
           $V := \emptyset$ ;
          if label( $n$ ) = label( $m$ ) then
             $F := \{(n, \text{label}(n), m, \text{label}(m))\}$ ;
          else  $J := J \setminus \{j\}$ ; fi
           $F_r[j] := F_r[j] \cup F$ ;  $V_r[j] := V_r[j] \cup V$ ;
        od od
      if  $J = \emptyset$  then return (FAILURE,  $\emptyset, \emptyset, \emptyset$ ); fi
      return (SUCCESS,  $F_r, V_r, J$ );
    end /* # $S$  は集合  $S$  の要素の数 */

```

図5 ラベル付き同型写像比較アルゴリズム  
Fig. 5 Matching algorithm for two labeled PDGs.

に示す。

#### 4.4 ラベルの抽象化

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型写像比較において、ラベルを加工する必要はない。しかし、広告し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型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、ラベル付き同型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。抽象化ラベルを持つグラフに対して、ラベル付き同型写像比較における同型の定義を以下に示す。

**定義6** 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、2つのPDGは抽象化ラベル付き同型写像比較において同

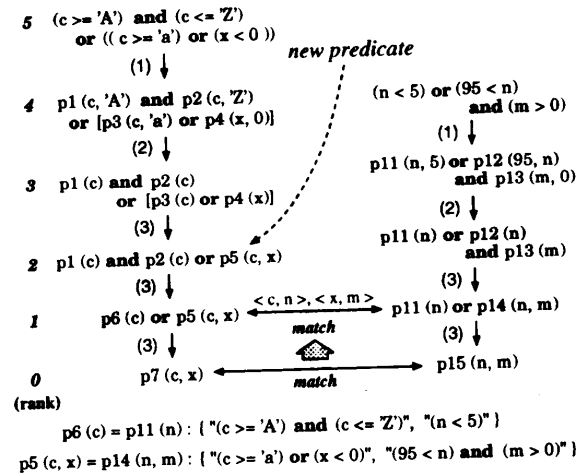


図6 ラベルの抽象化と一致の様子  
Fig. 6 Abstracted labels and matching them.

型であるという。

ラベルを抽象化する関数 abstraction\_label を付録 A.4 に示す。関数 abstraction (付録 A.4 [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、抽象度が最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 matching\_labels を付録 A.5 に示す。( $p$ )\_string (付録 A.5 [1], [2]) 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 6 に示す。図 6 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替えても変数(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6に示すようになる。

#### 4.5 交換可能スライス対

逆方向区間限定スライスおよび区間限定補スライスの定義2, 3と、ラベル付き同型写像比較における同型の定義5より、機能交換変化における交換可能スライス対を以下のように定義する。

**定義7** 逆方向区間限定スライス  $S_{Pi}$ ,  $S_{Qj}$  の区間限定補スライスをそれぞれ  $S_{Pi}^*$ ,  $S_{Qj}^*$ , ラベル付き同型写像比較において補スライスどうしが同型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型 ( $\langle S_P, S_Q \rangle \in U$ )、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 ( $\forall S_{Pi}^* (S_P^* \not\subseteq S_{Pi}^* \wedge \langle S_{Pi}, S_{Qj} \rangle \in U)$ ) とき、

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

### 5. 能動的部品変化アルゴリズム

本章では、能動的部品の2つの変化を実現する具体的なアルゴリズムを示す。

#### 5.1 基本動作

機能分割変化および機能交換変化が可能な能動的部品の基本動作アルゴリズムを図7に示す。図7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7[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7[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7[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7[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7[6])。変更前ソースコード  $P$  が自分の生成したソースコードであるとき、変更箇所の特定制を試み、変更事例を記録する(図7[7])。

##### (3) 変更事例を感知した際の動作 (図7[8], [9])

取得した変更事例を用いて、機能交換変化を行う

```

procedure active_component
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);
           $\langle S_P, S_Q \rangle$ : 交換可能スライス対;
          FPQ: 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  $\langle S_P, S_Q \rangle$  do
  [4]      $\langle S_P, S_Q \rangle, F_{PQ}$  の広告; od
  [5]   else if 部品変更  $P \rightarrow Q$  を感知 then
  [6]     Cde := decompose(all);
  [7]     if  $P \in Cde$  then record(P, Q); fi
  [8]   else if  $\langle S_P, S_Q \rangle, F_{PQ}$  を感知 then
  [9]     Cex := exchange(Cde,  $\langle S_P, S_Q \rangle, F_{PQ}$ ); fi
  od
end /* all はすべての変化条件を満たす予約語 */

```

図7 能動的部品の基本動作

Fig. 7 Basic algorithm of active components.

```

procedure decompose(Req)
declare Req, word: 文字列; S: ソースコード(スライス);
          nu, nl: 節点; 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; v: 変数集合;
begin
  Cde :=  $\emptyset$ ;
  for each word  $\in$  { 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]    $\langle n_u, n_l, v, dir \rangle :=$  スライシング基準(word);
  [3]   S :=  $\hat{S}_{dir}(n_u, n_l, v)$ ;
  [4]   if 核  $\subseteq S$  then Cde := Cde  $\cup$  { S }; fi
  fi od
  return (Cde);
end

```

図8 機能分割変化アルゴリズム

Fig. 8 Algorithm for decomposing functions.

(図7[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。

#### 5.2 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8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8[1])

部品利用者の要求文字列  $Req$  が、変化基準のキーワード  $word$  の部分文字列かどうか比較する。

##### (2) スライシングの適用 (図8[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8[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。

### 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 9 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 9 [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、定義 2 の逆方向区間限定スライス  $S_{P_i}, S_{Q_j}$  と、定義 3 の区間限定補スライス  $S_{P_i}^*, S_{Q_j}^*$  を作成する。

#### (2) 補スライスの変形 (図 9 [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  $S_{P_i}^*$  と  $S_{Q_j}^*$  は同型にならない。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 を付録 A.6 に示す。

#### (3) 交換可能スライス対の特定 (図 9 [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  $S_{P_i}^{*'}, S_{Q_j}^{*'}$  に対して、定義 5 のラベル付き同型写像比較を行い、補スライスが一致する

```

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;  $J$ : 整数集合;
 $S_P, S_Q$ : スライス (PDG);  $S, U$ : スライス対の集合;
 $S_P^*, S_Q^*, S_P^{*'}, S_Q^{*'}$ : 補スライス (PDG);
 $SC_P, SC_Q$ : スライスと補スライス対の集合;
 $F[], F_c, F_r$ : 節点およびラベル対の集合;  $j$ : 整数;
 $match$ : 一致 (SUCCESS / FAILURE);

begin
 $G_P := P$  の PDG;  $G_Q := Q$  の PDG;
[1]  $SC_P := \text{slicing}(G_P)$ ;  $SC_Q := \text{slicing}(G_Q)$ ;
 $U := \emptyset$ ;
for each  $\langle S_P, S_P^*, n_u, n_l \rangle \in SC_P$  do
for each  $\langle S_Q, S_Q^*, m_u, m_l \rangle \in SC_Q$  do
[2]  $S_P^{*'} := \text{reduce}(S_P^*, S_P)$ ;  $S_Q^{*' } := \text{reduce}(S_Q^*, S_Q)$ ;
[3]  $(match, F, V, J) := \text{matching}(S_P^{*'}, S_Q^{*'}, \text{FALSE})$ ;
if  $match = \text{SUCCESS}$  then
for each  $j \in J$  do
 $U := U \cup \{ \langle S_P, n_u, n_l, S_Q, m_u, m_l, F[j] \rangle \}$ ;
od fi
od od
[4]  $S := \{ \langle S_P, n_u, n_l, S_Q, m_u, m_l, F_c \rangle \in U \mid$ 
 $S_P^* \neq \emptyset \wedge S_Q^* \neq \emptyset \wedge \forall S_{P_i}^* (S_{P_i}^* \not\subseteq S_P^* \wedge$ 
 $\wedge \langle S_{P_i}, n_{ui}, n_{li}, S_{Q_j}, m_{uj}, m_{lj}, F_c \rangle \in U) \}$ ;
for each  $\langle S_P, n_u, n_l, S_Q, m_u, m_l, F_c \rangle \in S$  do
[5]  $F_r := \text{subgraph}(S_P, n_u, n_l, S_Q, m_u, m_l, F_c)$ ;
for each  $F_{PQ} \in F_r$  do
[6] 交換可能スライス対  $\langle S_P, S_Q \rangle, F_{PQ}$  を記録; od
od
end

```

図 9 変更事例を記録するアルゴリズム

Fig. 9 Algorithm for recording modification cases.

スライス対  $\langle S_{P_i}, S_{Q_j} \rangle$  と節点の対応関係  $F_c$  を決定する [改良 1]。このとき、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 7 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 9 [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` を付録 A.7 に示す。 $S_P, S_Q$  にそれぞれ包含されるスライス  $S_A, S_B$  のラベル付き同型写像比較 (付録 A.7 [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  $F_c$  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] (付録 A.7 [2]~[7])。

#### (5) 交換可能スライス対の記録 (図 9 [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。

### 5.4 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 10 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

```

procedure exchange( $Cde, \langle S_P, S_Q \rangle, F_{PQ}$ )
declare  $Cde, Cex$ : 生成されるソースコードの集合;
 $\langle S_P, S_Q \rangle$ : 交換可能スライス対;  $V$ : 変数対集合;
 $F[], F_{RP}, F_{PQ}$ : 節点およびラベル対の集合;
 $R, R'$ : ソースコード;  $N_r, T_r$ : 節点対の集合;
 $G_R, G_{R1}, G_{R2}, G_{R3}$ : PDG;  $match$ : 一致;
 $SC_R$ : スライスと補スライス対の集合;  $J$ : 整数集合;
 $S_R, S^*$ : スライスと補スライス (PDG);
 $n_{Ru}, n_{Rl}, n, m$ : 節点;  $N$ : 節点集合;

begin
 $Cex := \emptyset$ ;
for each  $R \in Cde$  do
[1]  $G_R := R$  の PDG;  $SC_R := \text{slicing}(G_R)$ ;
for each  $\langle S_R, S^*, n_{Ru}, n_{Rl} \rangle \in SC_R$  do
[2]  $(match, F, V, J) := \text{matching}(S_R, S_P, \text{TRUE})$ ;
if  $match = \text{SUCCESS}$  then
for each  $j \in J$  do
 $F_{RP} := F[j]$ ;
[3]  $G_{R1} := \text{replace}(G_R, F_{RP}, F_{PQ})$ ;
[4]  $N := \{ p \mid \langle r, l_r, p, l_p \rangle \in F_{RP} \wedge \langle p, l_p, q, l_q \rangle \notin F_{PQ} \}$ ;
[5]  $(G_{R2}, T_r) := \text{combine}(G_{R1}, S_Q, N, V, F_{RP})$ ;
for each  $N_r \in T_r$  do
 $G_{R3} := G_{R2}$  のコピー;
[6] for each  $\langle n, m \rangle \in N_r$  do
[7]  $\text{edge}(G_{R3}) := \text{edge}(G_{R3}) \cup \{ n \rightarrow m \}$ ; od
[8]  $R' := \text{ReconstituteProgram}(G_{R3})$ ;
if  $R' \neq \text{FAILURE} \wedge \text{核} \subseteq R'$  then
 $Cex := Cex \cup \{ R' \}$ ; fi
od od fi od od
return ( $Cex$ );
end

```

図 10 機能を交換するアルゴリズム

Fig. 10 Algorithm for exchanging functions.

を抽象化する改良2と、スライスと補スライス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10[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、定義2の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10[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 `matching` の引数を `TRUE` とする)。

### (3) 交換対象スライスの置換 (図10[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` を付録A.8に示す。

置換は2段階に分けられる。1段階目の置換 (付録A.8[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段階目の置換 (付録A.8[2]) は対応関係  $F_{PQ}$  を用いて、1段階目で置換された  $S_P$  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。

### (4) スライスと補スライスの合成 (図10[5]~[7])

交換可能スライス対の変更後スライス  $S_Q$  を、置換後の定義4の全区間補スライス  $G_{R1} (= S_R^*)$  に重ねることで合成を行う。合成を行う関数 `combine` を図11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11[1], [2])。重ね合わせの終了後、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、CFG 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 Dependence Graph)<sup>5)</sup> 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  $dom(n)$  とし、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  $pdom(n)$

```

procedure combine( $G_R, S_Q, N, V, F_{RP}$ )
declare  $G_R, S_Q$ : PDG;  $n, m$ : 節点;  $l_n, l_m$ : ラベル;
 $N, N_G, ID, PD$ : 節点集合;  $N_r, T_r$ : 節点対の集合;
 $e$ : 矢印;  $E$ : 矢印集合;
 $v_n, v_m$ : 変数;  $V$ : 変数対集合;

begin
[1]  $node(G_R) := node(G_R) \cup node(S_Q)$ ;
[2]  $edge(G_R) := edge(G_R) \cup edge(S_Q)$ ;
 $E := \{e' \mid src(e') \in N \wedge e' \text{ が制御依存矢印}\}$ ;
 $N_r := \emptyset$ ;
for each  $e \in E$  do
  if  $src(e) \in N$  and  $dst(e) \notin N$  then
     $m := src(e)$ ;
[3]  $ID(m) := \{q \mid q \in dom(m) \wedge q \neq m \wedge q \notin N$ 
       $\wedge \neg \exists r (q \in dom(r) \wedge r \in dom(m))\}$ ;
[4]  $PD(m) := \{q \mid p \in ID(m) \wedge q \in pdom(p)$ 
       $\wedge q \text{ が条件分岐 (if) 節点か繰り返し (while) 節点}\}$ ;
[5] for each  $n \in PD(m)$  do
[6] if  $e$  の種類 = T then  $N_r := N_r \cup \{\langle n, m \rangle\}$ ; fi
[7] if  $e$  の種類 = F  $\wedge m$  が条件分岐 (if) 節点 then
[8]  $N_r := N_r \cup \{\langle n, m \rangle\}$ ; fi od
fi od
 $T_r := \{T \mid T \subseteq N_r \wedge \forall \langle n, m \rangle \in T (n \text{ の接続元矢印の数} = 1)\}$ ;
[9]  $edge(G_R) := \{e' \mid e' \in edge(G_R)$ 
       $\wedge src(e') \notin N \wedge dst(e') \notin N\}$ ;
[10]  $node(G_R) := \{p \mid p \in node(G_R) \wedge p \notin N\}$ ;
 $L := \{\langle l_n, l_m \rangle \mid \langle n, l_n, m, l_m \rangle \in F_{RP} \wedge n \in N\}$ ;
for each  $n \in [node(G_R) \cap node(S_Q)]$  do
  for each  $\langle l_n, l_m \rangle \in L$  do
[11] if  $label(n) = l_n$  then  $label(n) := l_m$ ; fi od
  for each  $\langle v_n, v_m \rangle \in V$  do
[12]  $G_{R3}$  の  $label(n)$  の変数  $v_n$  を  $v_m$  に書き換え; od
od
return ( $G_R, T_r$ ); /*  $src(e)$  は矢印  $e$  の接続元節点 */
end /*  $dst(e)$  は矢印  $e$  の接続先節点 */

```

図11 スライスと補スライスを合成するアルゴリズム

Fig. 11 Algorithm for combining slices and the remainder of the replaced slices.

とする。このとき、関数 `combine` において、 $ID(m)$ ,  $PD(m)$  を求める (図11[3], [4])。  $ID(m)$  は消去される節点  $m$  に制御依存関係を持ち、他の節点を經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11[5]~[8]) し、ヌル節点およびヌル節点に接続を持つ矢印を消去する (図11[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11[11])。また、 $S_P$  と  $S_R$  のラベル付き同型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11[12])。

## (5) ソースコードへの再構成 (図 10 [8])

アルゴリズム *ReconstituteProgram*<sup>14)</sup>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、def-order dependence 矢印を追加し、データ依存関係矢印を loop independent 矢印と loop carried 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。

## 6. アルゴリズムの適用例

本章では、能動的部品が機能分割変化および機能交換変化アルゴリズムを適用して変化する例を示す。

## 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 12 に示す変化規

```

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 12 能動的部品の变化規則の例  
Fig. 12 A change rule.

則を持つ能動的部品 *Histogram* を考える。

要求を表すキーワード "count" をライブラリに投入すると、能動的部品 *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  $Sb(1,16,\{letter\})+(16)$ ,  $Sb(2,23,\{digit\})+\{23\}$ , ... に基づき、5つ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。能動的部品 *Histogram* が生成したソースコードのひとつ "count letters" を、図 13 (a) の  $P$  に示す。

## 6.2 機能交換変化の例

本節では、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。

## 6.2.1 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 13 (a), (b) に示す。図 13 (a), (b) において、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

|                                                                                                                                                                                                                                                                                                                                                                                                                                                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> procedure count_letter; var ch: char;     letter, digit: integer;  begin * p1  letter := 0;  p2  digit := 0; * p3  read(ch); * p4  while ch &lt;&gt; EOF do begin * p5    if (ch&gt;='A') and (ch&lt;='Z') or (ch&gt;='a') and (ch&lt;='z') then * p6      letter := letter + 1;  p7    if (ch&gt;='0') and (ch&lt;='9') then p8      digit := digit + 1; * p9    read(ch); end;  p10  writeln(letter);  p11  writeln(digit) end. </pre> | <pre> procedure count_letter_caps; var ch: char;     letter, digit: integer;     capital, small: integer;  begin * q1+  capital := 0; * q2+  small := 0; q3  digit := 0; * q4  read(ch); * q5  while ch &lt;&gt; EOF do begin * q6+    if (ch&gt;='A') and (ch&lt;='Z') then * q7+      capital := capital + 1 * q8+    else if (ch&gt;='a') and (ch&lt;='z') then * q9+      small := small + 1; q10  if (ch&gt;='0') and (ch&lt;='9') then q11    digit := digit + 1; * q12  read(ch); end; * q13+  letter := capital + small; q14  writeln(letter); q15+  writeln(capital); q16+  writeln(small); q17  writeln(digit) end. </pre> | <pre> procedure count_blank; var c: char;     line: integer;     blank, others: integer;     rate: real;  begin r1  line := 1; * r2  blank := 0; r3  others := 0; * r4  read(c); * r5  while c &lt;&gt; 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. </pre> |
| (a) $P$ (pre-modified)                                                                                                                                                                                                                                                                                                                                                                                                                         | (b) $Q$ (post-modified)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | (c) $R$ (to be exchanged)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |

図 13 変更前後のソースコードと機能交換対象のソースコード  
Fig. 13 Source codes that are modified and to be exchanged.

降の説明の例として,  $S_{P12} = \hat{S}_b(p1, p10, \{\text{letter}\})$ ,  $S_{P12}^* = \hat{S}_b^*(p1, p10, \{\text{letter}\})$ ,  $S_{Q35} = \hat{S}_b(q1, q14, \{\text{letter}\})$ ,  $S_{Q35}^* = \hat{S}_b^*(q1, q14, \{\text{letter}\})$  を取り上げる. 図 13(a), (b) 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  $S_{P12}^*$  を変形する様子を図 14 に示す. 図 14 において, N はヌル節点を指す. 変形後の補スライス  $S_{P12}'$  と  $S_{Q35}'$  はラベル付き同型写像比較において一致し,  $S_{P12}$ ,  $S_{Q35}$  は定義 7 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 15 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. ラベル付き同型写像比較により, 補スライスどうしの節点の対応関係  $F_c$  は図 15 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  $ch$  のスライスが

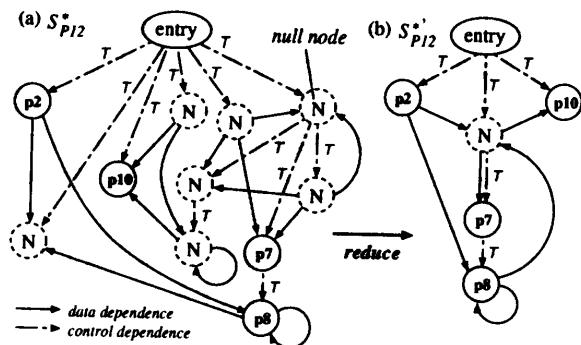


図 14 補スライスの変形 ( $S_{P12}^* \rightarrow S_{P12}'$ )  
Fig. 14 Reducing the PDG ( $S_{P12}^* \rightarrow S_{P12}'$ ).

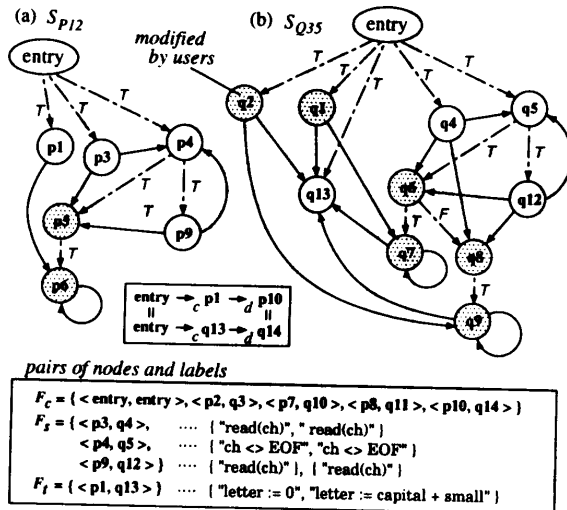


図 15 交換可能スライス対  $\langle S_{P12}, S_{Q35} \rangle$   
Fig. 15 Pair of changeable slices ( $S_{P12}, S_{Q35}$ ).

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型写像比較により, 節点の対応関係  $F_s$  は図 15 に示すように求まる. さらに,  $F_c, F_s$  を用いると,  $\text{entry} \rightarrow_c p1 \rightarrow_d p10$  と  $\text{entry} \rightarrow_c q13 \rightarrow_d q14$  より, 図 15 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. 網かけの節点に変更箇所を表す.

### 6.3 機能の交換

能動的部品 *TextFormat* が, 機能分割変化により 図 13(c) 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, 1 行あたりのスペースおよびタブの割合を求める」機能を持つ. ここでは, 以降の説明の例として  $S_{R36} = \hat{S}_b(r1, r13, \{\text{blank}\})$  を取り上げる. 図 13(c) 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型写像比較の様子を図 16 に示す.  $S_{P12}$  と  $S_{R36}$  はラベルを抽象化すると, ラベル付き同型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 16 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  $r2 \leftarrow p1$ ,  $r4 \leftarrow p3$ ,  $r5 \leftarrow p4$ ,  $r8 \leftarrow p5$ ,  $r9 \leftarrow p6$ ,  $r11 \leftarrow p9$  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  $p1 \leftarrow q13$ ,  $p3 \leftarrow q4$ ,  $p4 \leftarrow q5$ ,  $p9 \leftarrow q12$  と置き換える. 節

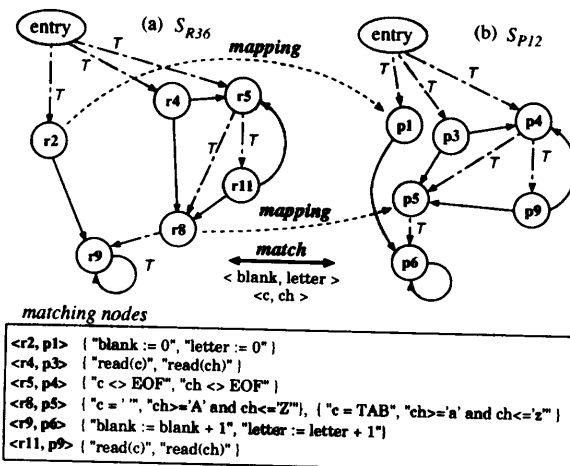


図 16 スライスどうしの抽象化ラベル付き同型写像比較  
Fig. 16 Matched slices with abstracted labels.

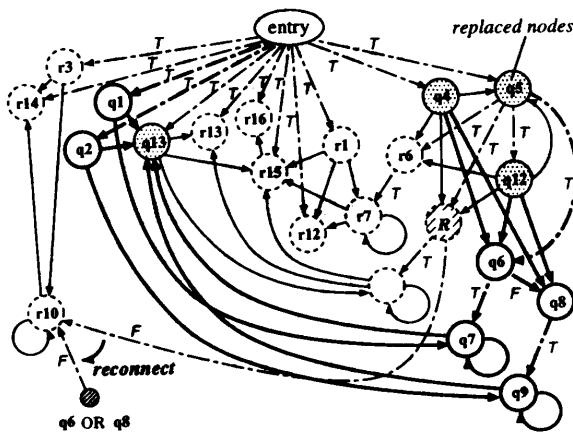


図 17 合成後の PDG ( $S_{Q35} + S_{R36}^*$ )  
Fig. 17 Combined PDG ( $S_{Q35} + S_{R36}^*$ ).

点 p5, p6 は、対応する節点が  $F_{PQ}$  に存在しないのでスル節点に置換する。

次に、スライス  $S_{Q35}$  と補スライス  $S_{R36}^*$  の合成を行う。合成後の PDG を図 17 に示す。図 17 において、網かけは置換後の節点を表す。太線は  $S_{Q35}$  を重ねること追加された節点および矢印を表す。また、 $R \rightarrow_{c(F)} r10$  が付けかえ対象の矢印である。図 17 の例では、 $ID(R) = \{q5\}$ 、 $PD(R) = \{q5, q6, q8\}$ 、矢印の属性が F 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 に接続したときに生成されるソースコード  $R'_1, R'_2$  を図 18 に示す。図 18 において、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。図 18(a) の  $R'_1$  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、1 行あ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、図 18(b) の  $R'_2$  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、1 行あ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 18 の  $R'_1$  と  $R'_2$  は、変数 others の実行結果に関して違いを持つ。 $R, R'_1, R'_2$  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分かる。

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 1 に示

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> program count_blank; var c: char; line: integer; blank, others: integer; capital, small: integer; rate: real;  begin   r1 line := 1;   r3 others := 0; + q4 read(c); + q1 capital := 0; + q2 small := 0; + q5 while c &lt;&gt; EOF do   begin     r6 if c = CR then     r7 line := line + 1; + q6 if c = ' ' then + q7 capital := capital + 1     else     begin       r10 others := others + 1; + q8 if c = TAB then + q9 small := small + 1     end; +q12 read(c);     end;   r12 writeln(line); +q13 blank := capital + small;   r13 writeln(blank);   r14 writeln(others);   r15 rate := blank / line;   r16 writeln(rate); end. </pre> | <pre> program count_blank; var c: char; line: integer; blank, others: integer; capital, small: integer; rate: real;  begin   r1 line := 1;   r3 others := 0; + q4 read(c); + q1 capital := 0; + q2 small := 0; + q5 while c &lt;&gt; EOF do   begin     r6 if c = CR then     r7 line := line + 1; + q6 if c = ' ' then + q7 capital := capital + 1 + q8 else if c = TAB then + q9 small := small + 1     else     others := others + 1; +q12 read(c);     end;   r12 writeln(line); +q13 blank := capital + small;   r13 writeln(blank);   r14 writeln(others);   r15 rate := blank / line;   r16 writeln(rate); end. </pre> |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

(a)  $R'_1$ (reconnect q6)

(b)  $R'_2$ (reconnect q8)

図 18 変化後のソースコードの例

Fig. 18 New source codes.

表 1 生成したスライス対およびソースコードの数  
Table 1 Number of slice pairs and source codes.

| 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 ( $S_{P_i}, S_{P_i}^*$ )  | 20        | —          |
| 変更後スライス対 ( $S_{Q_j}, S_{Q_j}^*$ )  | 56        | —          |
| 交換可能スライス対 ( $S_P, S_Q$ )           | 6         | —          |
| 交換対象スライス対 ( $S_{R_k}, S_{R_k}^*$ ) | —         | 51         |
| 新規ソースコード (Cex)                     | —         | 12         |

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 2 に生成した 12 個のソースコードの機能を示す。12 個のソースコードは、変数 others の実行結果、変数 capital, small 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。

## 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型写像比較により、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

表2 生成したソースコードの機能

Table 2 Functions of original and new source codes.

| 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、1行あたりの割合を求める                          |
| R' <sub>1</sub>  | 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' <sub>2</sub>  | 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' <sub>3</sub>  | R' <sub>1</sub> において、capitalの初期値が入力引数                          |
| R' <sub>4</sub>  | R' <sub>2</sub> において、capitalの初期値が入力引数                          |
| R' <sub>5</sub>  | R' <sub>1</sub> において、capital, smallの初期値が入力引数                   |
| R' <sub>6</sub>  | R' <sub>2</sub> において、capital, smallの初期値が入力引数                   |
| R' <sub>7</sub>  | R' <sub>1</sub> において、blankの出力結果がつねに初期値                         |
| R' <sub>8</sub>  | R' <sub>2</sub> において、blankの出力結果がつねに初期値                         |
| R' <sub>9</sub>  | R' <sub>1</sub> において、capitalの初期値が入力引数、blankの出力結果がつねに初期値        |
| R' <sub>10</sub> | R' <sub>2</sub> において、capitalの初期値が入力引数、blankの出力結果がつねに初期値        |
| R' <sub>11</sub> | R' <sub>1</sub> において、capital, smallの初期値が入力引数、blankの出力結果がつねに初期値 |
| R' <sub>12</sub> | R' <sub>2</sub> において、capital, smallの初期値が入力引数、blankの出力結果がつねに初期値 |

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** 既存の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。たとえば、6章の適用例において、表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。たとえば、6章の適用例において、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。

能動的部品を実際に再利用に適用して効果を得るためには、以下に示す課題を解決する必要がある。

**課題1** 現在の変化メカニズムは、手続き呼出し（再帰呼出し）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、かなり制限された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。また、ラベルの抽象化だけでは、プログラム構造の違いまでは吸収しておらず、生成される新規ソースコードの種類が少ない。

**課題2** キーワードだけで部品利用者の要求を表現することは難しい。このため、要求を容易かつ正確に記述する手法とそれを感知するメカニズムを確立する必要がある。

**課題3** 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、ラベル付き同型写像比較を行うことは時間的、リソース的観点から現実的でない。たとえば、6章の適用例において、 $20 \times 56 + 6 \times 51 = 1426$ 回の同型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。

課題1に対して、言語の制限を取り除くために、文献15)~18)の手法を本変化メカニズムに組み込むことが考えられる。また、プログラム構造に関する抽象化手法も、同時に考案中である。課題2に対しては、部品利用者の要求の一部を論理式で記述することを考えている。これにより、キーワードより複雑な要求を形式的に記述することができる。さらに、課題3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<sup>19)</sup>。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型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。

## 8. おわりに

ソースコード再利用において、従来になかった新しい部品として、部品検索時に自ら形を変える能動的部品を提唱した。本稿では、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、適用例を示した。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型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に変化可能となる。よって、部品変更の負担を軽減でき、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。現在、7章で述べた課題に対する考察と、ソースコード再利用に能動的部品を適用する実験を進めている。

謝辞 日頃ご指導ご討論いただく伊藤正樹リーダーはじめ、グループの皆様に深く感謝します。また、本研究の初期段階から本稿の推敲までアドバイスをいただいた高橋直久氏、鈴木英明氏に感謝します。

## 参考文献

- 1) Biggerstaff, T. and Richter, C.: Reusability Framework, Assessment, and Directions, *IEEE Software*, Vol.4, No.2, pp.41-49 (1987).
- 2) 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, 情報処理学会ソフトウェア工学研究会 (95-SE-102), Vol.95, No.11, pp.71-76 (1995).
- 3) 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, レクチャーノート/ソフトウェア工学 15, ソフトウェア工学の基礎 II, 日本ソフトウェア科学会, pp.31-40 (1996).
- 4) Weiser, M.: Program Slicing, *IEEE Trans. Softw. Eng.*, Vol.10, No.4, pp. 352-357 (1984).
- 5) Ferrante, J., Ottenstein, K.J. and Warren, J.D.: The Program Dependence Graph and Its Use in Optimization, *ACM Trans. Prog. Lang. Syst.*, Vol.9, No.3, pp.319-349 (1987).
- 6) Horwitz, S. and Reps, T.: Efficient Comparison of Program Slices, *Acta Informatica*, Vol.28, pp.713-732 (1991).
- 7) 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 (1996).
- 8) Williams, R.S.: Learning to Program by Examining and Modifying Cases, *Proc. Case-Based Reasoning Workshop*, pp.463-474 (1988).
- 9) Horwitz, S., Prins, J. and Reps, T.: Integrating Noninterfering Versions of Programs, *ACM Trans. Prog. Lang. Syst.*, Vol.11, No.3, pp.345-387 (1989).
- 10) Yang, W.: A New Algorithm for Semantics-Based Program Integration, Technical Report TR-962, Computer Sciences Dept., Univ. of Wisconsin, Madison, WI (1990).
- 11) 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 (1995).
- 12) Ottenstein, K.J. and Ottenstein, L.M.: The Program Dependence Graph in a Software Development Environment, *Proc. ACM SIGPLAN/SIGSOFT Symp. Practical Programming Development Environments*, *ACM SIGPLAN Notices*, Vol.19, No.5, pp.177-184 (1984).
- 13) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley (1986).
- 14) Ball, T., Horwitz, S. and Reps, T.: Correctness of an Algorithm for Reconstituting a Program from a Dependence Graph, Technical Report TR-947, Computer Sciences Dept., Univ. of Wisconsin, Madison, WI (1990).
- 15) Horwitz, S., Pfeiffer, P. and Reps, T.: Dependence Analysis for Pointer Variables, *Proc. SIGPLAN Conf. on Prog. Lang. Desi. and Impl.*, pp.28-40 (1989).
- 16) Horwitz, S., Ball, T. and Binkley, D.: Interprocedural Slicing Using Dependence Graphs, *ACM Trans. Prog. Lang. Syst.*, Vol.12, No.1, pp.26-60 (1990).
- 17) Ball, T. and Horwitz, S.: Slicing Programs with Arbitrary Control Flow, *Proc. Int. Workshop on Automated and Algorithmic Debugging, Lecture Notes in Computer Science*, Vol.749, pp.206-222 (1993).
- 18) 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, 信学論, Vol.J78-D-I, No.1, pp.11-22 (1995).
- 19) 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, 信学論, Vol.J77-D-I, No.11, pp.747-758 (1994).

## 付 録

## A.1 対象ソースコードのBNF

**active-component-code:**  
**program** identifier ; **block** .  
**procedure** identifier ; **block** .  
**block:**  
**var** variable-declaration **begin** statement-list **end**  
**begin** statement-list **end**  
**statement-list:**  
statement  
statement-list ; statement  
**statement:**  
empty-statement  
variable := expression  
**begin** statement-list **end**  
**if** expression **then** statement  
**if** expression **then** statement **else** statement  
**while** expression **do** statement  
**read** (variable)  
**readLn** (variable)  
**write** (variable)  
**write** ('string')  
**writeln** (variable)  
**writeln** ('string')  
**writeln**  
**expression:**  
simple-expression relational-operator simple-expression  
simple-expression  
**simple-expression:**  
sign term addition-operator term  
sign term **or** term  
sign term  
**term:**  
factor multiplication-operator factor  
factor **and** factor  
factor  
**factor:**  
unsigned-constant  
variable  
(expression)  
**not** factor  
empty-statement ... 空文 variable-declaration ... 変数宣言部  
relational-operator ... 関係演算子(=, <, <=, >, >=)  
multiplication-operator ... 乗除演算子(\*, /, div, mod)  
addition-operator ... 加減演算子(+, -)  
sign ... 符号(+, -, 空文字) variable ... 変数 string ... 文字列  
unsigned-constant ... 符号なし定数 identifier ... 識別子

## A.2 節点を写像するアルゴリズム

**procedure** mapping\_nodes( $G_n, G_m$ )  
**declare**  $G_n, G_m$ : PDG;  $v$ : 節点;  $W$ : 節点对集合;  
type: 依存関係の種類(T, F, D);  
 $D$ ,  $D_T$ ,  $D_F$ ,  $D_D$ : 節点集合;  
**procedure** candidate\_nodes( $n, m, t$ )  
**declare**  $n, m, p, q$ : 節点;  $D_n, D_m$ : 節点集合;  
 $t$ : 依存関係の種類;  
**begin**  
 $D_n := \{ G_n \text{ の節点 } n \text{ の } t \text{ 依存先節点集合} \};$   
 $D_m := \{ G_m \text{ の節点 } m \text{ の } t \text{ 依存先節点集合} \};$   
**if**  $\#D_n \neq \#D_m$  **then return**; **fi**  
 $D_t[n] := D_t[n] \cup \{m\}; W := W \cup \{(n, m)\};$   
**for each**  $p \in D_n$  **do**  
**for each**  $q \in D_m$  **do**  
**if**  $(p, q) \notin W$  **then**  
**for each**  $t \in \{T, F, D\}$  **do**  
candidate\_nodes( $p, q, t$ );  
**od fi od od**  
**end**  
**begin**  
 $W := \emptyset;$   
**for each**  $v \in \text{node}(G_n)$  **do**

$D_T[v] := \emptyset; D_F[v] := \emptyset; D_D[v] := \emptyset;$  **od**  
**for each**  $\text{type} \in \{T, F, D\}$  **do**  
candidate\_nodes(entry, entry, type); **od**  
**for each**  $v \in \text{node}(G_n)$  **do**  
 $D[v] := D_T[v] \cap D_F[v] \cap D_D[v];$  **od**  
**return** ( $D$ ); /\*  $D$  はデータ依存関係 \*/  
**end** /\*  $T, F$  は制御依存関係の種類(真/偽) \*/

## A.3 一致する節点を求めるアルゴリズム

**procedure** matching\_nodes( $N, D$ )  
**declare**  $n, m$ : 節点;  $i, j$ : 整数;  $N, M, D$ : 節点集合;  
 $J, tmpJ, JJ$ : 整数集合;  $B$ : 節点对集合;  
**begin**  
 $i := 0; J := \{0\};$   
 $B[0] := \{(p, q) \mid p \in N \wedge q \in D[p]\};$   
**for each**  $n \in N$  **do**  
**for each**  $j \in J$  **do**  
 $M := \{q \mid p = n \wedge (p, q) \in B[j]\};$   
 $JJ[j] := \emptyset;$   
**if**  $\#M \neq 1$  **then**  
**for each**  $m \in M$  **do**  
 $i := i + 1; JJ[j] := JJ[j] \cup \{i\};$   
 $B[i] := B[j] \setminus \{(p, q) \mid p = n \wedge q \in D[n]\};$   
 $B[i] := B[i] \cup \{(n, m)\};$   
**od fi od**  
 $tmpJ := J; J := \{k \mid k \in JJ[j] \wedge j \in J\};$   
**if**  $J = \emptyset$  **then**  $J := tmpJ;$  **fi**  
**od**  
 $tmpJ := J;$   
**for each**  $j \in tmpJ$  **do**  
 $M := \{q \mid (p, q) \in B[j]\};$   
**if**  $\#M \neq \#N$  **then**  $J := J \setminus \{j\};$  **fi**  
**od**  
**return** ( $B, J$ );  
**end**

## A.4 ラベルを抽象化するアルゴリズム

**procedure** abstraction\_label(orig\_label)  
**declare** orig\_label,  $a$ ,  $l$ : ラベル; rank,  $r$ : 整数;  
**begin**  
 $l[0] := \text{orig\_label}; \text{rank} := 0;$   
[1] **while**  $l[\text{rank}] \neq \text{単一述語}$  **do**  
[2]  $\text{rank} := \text{rank} + 1;$   
[3]  $l[\text{rank}] := \text{abstraction}(l[\text{rank} - 1]);$  **od**  
**for**  $r := 0$  **to** rank **do**  $a[r] := l[\text{rank} - r];$  **od**  
 $a[\text{rank} + 1] := \top;$   
**return** ( $a$ );  
**end**

## A.5 抽象化したラベルを比較するアルゴリズム

**procedure** matching\_labels( $n, l_n, m, l_m$ )  
**declare**  $l_n, l_m$ : ラベル;  $r, i$ : 整数;  
 $n, m$ : 節点;  $F$ : 節点およびラベル対集合;  
 $v_n, v_m$ : 変数;  $V$ : 変数対集合;  
**begin**  
 $F[n] := \emptyset; V[n] := \emptyset;$   
**for each**  $v_n \in [\text{def}(n) \cup \text{use}(n)]$  **do**  
**for each**  $v_m \in [\text{def}(m) \cup \text{use}(m)]$  **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**  $l_n[r] = l_m[r] \wedge l_n[r] \neq \top \wedge l_m[r] \neq \top$  **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]  $l'_n := (l_n[r - 1] \text{ 内の } i \text{ 番目の述語})_{\text{string}};$   
[2]  $l'_m := (l_m[r - 1] \text{ 内の } i \text{ 番目の述語})_{\text{string}};$   
 $F[n] := F[n] \cup \{(n, l'_n, m, l'_m)\};$   
**od**

```

    od
     $V[n] := V[n] \cup \{v_n, v_m\}$ ; fi
  od od
  return (F, V);
end

```

#### A.6 補スライスを変形するアルゴリズム

```

procedure reduce( $G, G_N$ )
declare  $G, G_N$ : PDG;  $n_s, n_d$ : 節点;  $e$ : 矢印;
begin
  node( $G$ ) := node( $G$ )  $\cup$  { NULL };
  for each  $e \in \text{edge}(G)$  do
     $n_s := \text{src}(e)$ ;  $n_d := \text{dst}(e)$ ;
    if  $n_s \in \text{node}(G_N) \wedge n_d \in \text{node}(G_N)$  then
      edge( $G$ ) := edge( $G$ )  $\setminus$  { $e$ };
    else if  $n_s \in \text{node}(G_N)$  then
      edge( $G$ ) := edge( $G$ )  $\cup$  { NULL  $\rightarrow$   $n_d$  }  $\setminus$  { $e$ }; fi
    else if  $n_d \in \text{node}(G_N)$  then
      edge( $G$ ) := edge( $G$ )  $\cup$  { $n_s \rightarrow$  NULL }  $\setminus$  { $e$ }; fi fi
  od
  node( $G$ ) := { $n \mid \exists p(p \rightarrow n) \vee \exists q(n \rightarrow q)$ };
  return ( $G$ );
end

```

#### A.7 節点の対応関係を求めるアルゴリズム

```

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;  $N, N_t$ : 節点对集合;
 $F[\ ], F_c, F_s, F_r, F_t$ : 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;
match: 一致;  $j$ : 整数;  $J$ : 整数集合;
begin
   $F_r := \emptyset$ ;
   $V_P := \bigcup_{p \in \text{node}(S_P)} \text{def}(p)$ ;
   $V_Q := \bigcup_{q \in \text{node}(S_Q)} \text{def}(q)$ ;
   $V_d := \{V \mid V \subseteq [V_P \cap V_Q]\}$ ;
  for each  $V \in V_d$  do
     $S_A := \hat{S}_b(n_u, n_l, V)$ ;  $S_B := \hat{S}_b(m_u, m_l, V)$ ;
    if  $S_A \subseteq S_P \wedge S_B \subseteq S_Q$  then
      [1] ( $\text{match}, F, V, J$ ) := matching( $S_A, S_B, \text{FALSE}$ );
      if match = SUCCESS then
        for each  $j \in J$  do  $F_r := F_r \cup F[j]$ ; od fi
      fi od
     $F := \emptyset$ ;
    for each  $F_s \in F_r$  do
      [2]  $N := \{ \langle p, q \rangle \mid \langle p, l_p, q, l_q \rangle \in [F_c \cup F_s] \}$ ;
      [3]  $N_t := \{ \langle p, q \rangle \mid p_1 \rightarrow p \rightarrow p_2 \wedge q_1 \rightarrow q \rightarrow q_2$ 
         $\wedge \langle p_1, q_1 \rangle \in N \wedge \langle p_2, q_2 \rangle \in N \}$ ;
      [4] for each  $\langle p, q \rangle \in N_t$  do
      [5] if  $\exists q_i (\langle p_i, q_i \rangle \in N_t \wedge p = p_i \wedge q \neq q_i)$ 
         $\vee \exists p_j (\langle p_j, q_j \rangle \in N_t \wedge p \neq p_j \wedge q = q_j)$  then
      [6]  $N_t := N_t \setminus \{ \langle p, q \rangle \}$ ; fi
      od
      [7]  $F_t := \{ \langle p, \text{label}(p), q, \text{label}(q) \rangle \mid \langle p, q \rangle \in N_t \}$ ;
       $F := F \cup \{ F_s \cup F_t \}$ ;
    od
    return ( $F$ );
  end
end

```

#### A.8 スライスを置換するアルゴリズム

```

procedure replace( $G_R, F_{RP}, F_{PQ}$ )

```

```

declare  $G_R, G_{R4}, G_{R5}$ : PDG;
 $F_{RP}, F_{PQ}$ : 節点对およびラベル対の集合;
begin
  [1]  $G_{R4} := \text{relace\_nodes}(G_R, F_{RP})$ ;
  [2]  $G_{R5} := \text{relace\_nodes}(G_{R4}, F_{PQ})$ ;
  return ( $G_{R5}$ );
end

```

```

procedure replace_nodes( $G_R, F$ )
declare  $G_R$ : PDG;  $F$ : 節点およびラベル対集合;
 $n, m$ : 節点;  $N$ : 節点对集合;  $e_t$ : 矢印;
begin
   $N := \{ \langle n, m \rangle \mid \langle n, l_n, m, l_m \rangle \in F \}$ ;
  for each  $\langle n, m \rangle \in N$  do
    node( $G_R$ ) := node( $G_R$ )  $\cup$  { $m$ };
    for each  $e_t \in \{e' \mid n = \text{src}(e')\}$  do
      edge( $G_R$ ) := edge( $G_R$ )  $\cup$  { $m \rightarrow_t \text{dst}(e_t)$ }; od
    for each  $e_t \in \{e' \mid n = \text{dst}(e')\}$  do
      edge( $G_R$ ) := edge( $G_R$ )  $\cup$  { $\text{src}(e_t) \rightarrow_t m$ }; od
    edge( $G_R$ ) := edge( $G_R$ )  $\setminus$  { $n \rightarrow \text{dst}(e_t)$ }
    edge( $G_R$ ) := edge( $G_R$ )  $\setminus$  { $\text{src}(e_t) \rightarrow n$ }
    node( $G_R$ ) := node( $G_R$ )  $\setminus$  { $n$ };
  od
  return ( $G_R$ );
end /*  $\rightarrow_t$  は  $e_t$  と同種類の依存関係矢印 */

```

(平成 8 年 3 月 15 日受付)

(平成 8 年 9 月 12 日採録)

#### 丸山 勝久 (正会員)



1967 年生。1991 年早稲田大学理工学部電気工学科卒業。1993 年同大学院理工学研究科修士課程修了。同年日本電信電話株式会社入社。現在、NTT ソフトウェア研究所 広域コンピューティング研究部に所属。ソフトウェア・リエンジニアリング、プログラム自動合成技術の研究に従事。電子情報通信学会、日本ソフトウェア科学会、IEEE-CS、ACM 各会員。

#### 島 健一 (正会員)



1976 年北海道大学工学部電気工学科卒業。1978 年同大学院情報工学専攻修士課程修了。同年 NTT 武蔵野電気通信研究所入所。現在、NTT ソフトウェア研究所主任研究員。主に、知識ベース構築用システムの基礎研究、ソフトウェア設計での知識獲得、学習システムなどの研究開発に従事。また、WWW でのユーザモデル研究に興味を持つ。電子情報通信学会、人工知能学会各会員。