2008－SE－159 (19)
2008／3／18

レイヤーアーキテクチャのためのソフトウェアクラスタリング手法

加賀洋渡[†1] 新田直也[†2]

実規模ソフトウェアの保守, 再利用においてアーキテクチャの理解は必要不可欠である. しかし, 現実のプロジェクトではアーキテクチャに関する情報は文書化されていないか, 文書化されていても陳腐化している場合がほとんどであり, 保守, 再利用作業において多大なコストを発生している. そのため, ソースコードからアーキテクチャの情報を抽出するソフトウェアクラスタリング技術が広く研究されているが, 多くのソフトウェアクラスタリング手法では, ソフトウェア部品間の利用関係の向きに関する情報が捨象されているため, レイヤーアーキテクチャを持つ大規模なソフトウェアのクラスタリングには適さない. そこで本研究では, 利用関係の向きに着目したクラスタリング手法を考案し, クラスタリングツール SCALAR の実装を行った. またいくつかの実規模ソフトウェアを対象に SCALAR を適用し, その有効性を確認した.

Software Clustering for Layer Architecture

HITOTO KAGA and NAOYA NITTA

Architecture understanding is crucial for large scale software maintenance and reuse. However, in an actual project architectural documents are often obsolete or rather missing, and a maintainer often have to make a great effort to extract implicit architectural information from the source code. To address the problem, many works have been done in the field of software clustering, but most of the works are not sufficient for clustering large scale software which has a layer architecture because they omit the information of the directions of the use relation. Therefore, we consider a software clustering method which is aware of the directions of the use relation and implement it as a clustering tool SCALAR. Furthermore, we apply SCALAR to several real-world programs and confirm its effectiveness.

1. はじめに

実規模ソフトウェアの保守, 再利用においてアーキテクチャの理解は必要不可欠である. アーキテクチャを理解する上では, 適切に保守されたアーキテクチャ文書が非常に有用であるが, 現実のプロジェクトではその種の文書が存在しないか, 存在しても陳腐化している場合が多い. その結果, 開発者は保守, 再利用時にソースコードからアーキテクチャに関する情報を取り出さなければならず, そのことが大きな負担となっている.

そのため, ソースコードからアーキテクチャに関する情報を抽出する**アーキテクチャ抽出**と呼ばれるリバースエンジニアリング技術が広く研究されてきた (文献[1),5),6),8)–13)]). それらのうちのいくつかは, ソフトウェア全体の構造を高凝集, 疎結合などの設計原則に基づいて理解可能な小さなサブシステムに分解するものであり (**ソフトウェアクラスタリング**, 文献[8)–10)]), 他のいくつかはパターンマッチング技術を用いて, ソースコードからアーキテクチャに関する情報を抽出することを目指すものである (文献[1),13)]).

本研究では, ソフトウェアクラスタリングに着目する. 従来のソフトウェアクラスタリングでは, 対象となるシステムは比較的小規模なものが多かった. また一般にソフトウェアクラスタリングでは, モジュールやクラスなどのソフトウェア部品間の利用関係を元に解析が行われるが, 利用関係の向きが考慮されていないものがほとんどであった.

そこで本研究では, ソフトウェア部品間の利用関係の向きに注目し, 大規模なシステムが潜在的に有しているレイヤアーキテクチャ[2)] を抽出するソフトウェアクラスタリング手法を提案する. 一般にレイヤアーキテクチャの構造は非循環依存関係の原則に従うと考えられる. 非循環依存関係の原則とは, サブシステム間の依存関係がサイクルを持つべきではないとする設計原則である. このことから, レイヤアーキテクチャを抽出するソフトウェアクラスタリングは, ソフトウェ

†1 甲南大学 自然科学研究科 情報システム学専攻
†2 甲南大学 知能情報学部

ア部品間の利用関係を表す有向グラフ G を, クラスタ間の依存関係を表す非循環有向グラフ G' に簡約化する手続きとみなすことができる. 本論文では, G の各強連結成分をクラスタとみなして G' を構成する手法と, 上位クラスタが下位クラスタのできるだけ少ないインタフェースに依存するようにクラスタ分割を行って G' を求める手法を提案する.

また本論文では, これらの手法をクラスタリングツール SCALAR として実装し, 複数の実規模アプリケーションを対象に評価実験を行った. その結果, 前者の手法によって抽出されるクラスタは実際のレイヤより粒度が細かい反面, 単一のレイヤ内の部品のみを含むことがわかった. これに対し, 後者の手法ではよりレイヤに近い粒度のクラスタを抽出することが可能である一方で, 抽出されたクラスタの中には複数レイヤの部品が混在する場合も見られた. 今後, 利用目的に応じてこれらの手法を使い分けることで保守および再利用の作業効率の向上が図れると考えられる. また, いずれの手法においても従来よりソフトウェアクラスタリングにおいて問題とされた偏在モジュールの存在が結果に悪影響を及ぼさないことが判明した. これも本手法の大きな特長といえる.

2. 諸 定 義

最初に本論文で用いる諸概念について定義する. 本論文では, ソースコード全体を**モジュール依存グラフ**で抽象化する. モジュール依存グラフは有向グラフ $G = (V, E)$ で, V はモジュール (ソフトウェア部品) を表す頂点の集合, $E \subseteq V \times V$ はモジュール間の依存関係を表す有向辺の集合である. モジュール依存グラフ G が与えられたとき, G の頂点の集合を $V(G)$ で表し, G の有向辺の集合を $E(G)$ で表す. G の 2 つの頂点 $v, v' \in V(G)$ が $e = \langle v, v' \rangle \in E(G)$ を満たすとき v と v' は**隣接する**といい, v および v' を e の**端点**という. G 中の隣接する頂点列 $p = \langle v_0, \ldots, v_k \rangle$ のうち頂点の重複を許さないものを**パス**といい, $v_0 = v_k$ かつ $v_0, \ldots, v_{k-1}$ が重複しないものを**サイクル**という. 特に頂点列を結ぶ各有向辺の向きが等しいパスおよびサイクルを, それぞれ**有向パス**, **有向サイクル**という. 頂点 $v \in V(G)$ の**入次数**とは, v に入ってくる有向辺の数であり $d_{\mathrm{IN}}(v)$ で表す. 同様に v の**出次数**とは, v から出ていく有向辺の数であり $d_{\mathrm{OUT}}(v)$ で表す. グラフ G 中の任意の異なる 2 点間にパスが存在するとき, G は**連結**であるという. また, G 中の任意の異なる 2 点間に有向パスが存在するとき, G は**強連結**であるという. 頂点の集合 $V' \subseteq V(G)$ に対して**誘導部分グラフ** $G[V']$ とは, V' を頂点集合に持つ有向グラフで G の有向辺のうち両端点が V' に属するものすべてを有向辺として持つグラフである. G の連結な誘導部分グラフを G の**連結成分**といい, 強連結な誘導部分グラフを G の**強連結成分**という. G から有向辺 e を取り除き, その両端点を 1 つの頂点にまとめたものを G の e による**縮約**といい G/e で表す. 同様に H を G の誘導部分グラフとしたとき, H のすべての有向辺による G の縮約を G/H で表す.

3. レイヤ構造抽出問題

本研究が対象とするソフトウェアクラスタリングを以下に定義する. モジュール依存グラフ G のクラスタとは G の連結な誘導部分グラフ $G[V']$ のことをいう. G のクラスタ分割とは,G のクラスタの組 $\Pi_{\mathrm{G}} = \langle G_1, \ldots, G_k \rangle$ で $1 \leq i, j \leq k, i \neq j$ のとき $V(G_i) \cap V(G_j) = \phi$ かつ

$$\bigcup_{1 \leq i \leq k} V(G_i) = V(G)$$

を満たすものをいう. Π_{G} のクラスタ構造 G/Π_{G} を $G/\Pi_{\mathrm{G}} = ((G/G_1)/G_2) \ldots /G_k$ で定義する. ただし $V(G/\Pi_{\mathrm{G}}) = \Pi_{\mathrm{G}}$ とする. 本研究ではレイヤアーキテクチャの抽出を目的とするため, G/Π_{G} はレイヤ間の依存関係を表すものと考える. 一般にレイヤ間には非循環依存関係の原則が成り立っているため, G/Π_{G} は有向サイクルを持たないと考えてよい. このような G/Π_{G} を特に**レイヤ構造**と呼び, このときの Π_{G} を**レイヤ分割**と呼ぶ. G が与えられたとき, あるレイヤ分割 Π_{G} を求める問題を**レイヤ構造抽出問題**という.

一般にレイヤ構造抽出問題は 1 つ以上の解を持つ. 特に以下の 2 種類の解は重要である.

例 3.1 (**自明な解**) クラスタ分割 $\Pi_{\mathrm{G}} = \langle G \rangle$ は, 常にレイヤ構造抽出問題の解となる. この解を自明な解という. 自明な解は明らかにクラスタ数を最小にする解である. □

例 3.2 (**極大強連結成分で構成された解**) G のすべての極大強連結成分を $G_1, \ldots, G_k$ とするとき, クラスタ分割 $\Pi_{\mathrm{G}} = \langle G_1, \ldots, G_k \rangle$ は, 常にレイヤ構造抽出問題の解となる. この解はクラスタ数を最大にする解である. □

4. 提案手法と評価実験

提案手法では, Java のソースコードを解析の対象とする. 任意の Java ソースコードが与えられたとき, 抽象型 (クラス, インターフェース) を頂点とみなし, 抽象型の間の継承関係, 実装関係, フィールドによる参

照, メソッドシグニチャによる参照などを含むあらゆる依存関係を有向辺とみなしてモジュール依存グラフを構成する. 構成されたモジュール依存グラフに対して, 本論文では以下の 2 種類の手法でレイヤ構造抽出問題を解く.

- **提案手法 1**: 例 3.2 で示した解を求める.
- **提案手法 2**: 上位クラスタが下位クラスタのできるだけ少ないインターフェースに依存するようにクラスタ分割を行う.

4.1 提案手法 1

提案手法 1 は G の各極大強連結成分をクラスタとみなして Π_G を構成する手法である. 得られるクラスタ構造 G/Π_G が有向サイクルを持たないことは極大強連結成分の定義より明らかである. この手法によって求められるレイヤ分割はレイヤ数を最大にする.

4.1.1 アルゴリズム

G の極大強連結成分の抽出には Tarjan アルゴリズムを用いる. Tarjan アルゴリズムによって G のあらゆる極大強連結成分を $O(|V|+|E|)$ で抽出することができる.

4.1.2 評価実験

評価実験に際して, 統合開発環境 eclipse[3] を使用し, 解析システムを実装した. 解析システムは以下の 2 つのツールから成る.

- **MDGExtractor**: Java のソースコードを解釈しモジュール依存グラフを生成する.
- SCALAR(**Software Cluster Analyzer for Layer ARchitecture**): モジュール依存グラフを元にクラスタリングを行う.

MDGExtractor は eclipse のプラグインとして Java により実装した. ソースコード内のクラスやインタフェースの利用関係を抽出する上で必要となる抽象構文木は eclipse が作成したものを利用するようにした.

解析システムを用いて, 本節では提案手法 1 がソフトウェアアーキテクチャの理解のために有効なレイヤ構造を抽出するかについて評価する. 解析対象として ArgoUML 0.19.2[14] と eclipse 3.2.2 のソースコードを用意した. ArgoUML はオープンソースの UML エディタで, ソースコードは約 10 万行である. eclipse はオープンソースの統合開発環境で, ソースコードは約 140 万行である. まず MDGExtractor を用いて解析対象のソースコードからモジュール依存グラフを抽出し, その際の処理時間を記録する. eclipse のワークスペース上のプロジェクトを入力ソースコードとし, 抽出したモジュール依存グラフはテキスト形式で出力される. 次に, MDGExtractor が生成したをモジュール依存グラフ SCALAR で読み込み解析する. 表 1 は ArgoUML と eclipse それぞれの頂点数, 辺数, MDGExtractor および SCALAR の実行時間を表している. 表 2 は解析手法 1 を用いた SCALAR の解析結果である. 抽出されたクラスタ C のうちサイズの大きいものに対して, C のサイズ, C に最も一致するレイヤ L, L に対する C の適合率および再現率を示す. 適合率は, クラスタ C があるレイヤ L に対応するとき, C に含まれる要素がどれくらいの割合で L に含まれているかを示す. 再現率は, L に含まれる要素がどれくらいの割合で C に含まれているかを示す. 同様に表 3 は eclipse を対象に解析を行った結果である. いずれのクラスタにおいても適合率は 100%と高い値を示していることがわかる.

表 1 解析システムの実行時間 (提案手法 1)

	頂点数	辺数	実行時間 (ミリ秒) †	
			MDGExtactor	SCALAR
ArgoUML	2228	16601	119,359	18,593
eclipse	18092	222700	2442,766	8151,266

† OS:WindowsXP Home Edition, CPU:Intel Celeron 3.33GHz, メモリ:504MB, JVM build 1.6.0.03

表 2 提案手法 1 による ArgoUML の解析結果

クラスタサイズ	適合率	再現率	該当レイヤ
515	100.0%	41.3%	org.argouml
79	100.0%	28.9%	org.tigris.gef
27	100.0%	12.6%	org.argouml.model ru.novosoft

表 3 提案手法 1 による eclipse の解析結果

クラスタサイズ	適合率	再現率	該当レイヤ
596	100.0%	20.0%	org.eclipse.ui
454	100.0%	8.75%	org.eclipse.jdt
252	100.0%	4.87%	org.eclipse.jdt
159	100.0%	3.08%	org.eclipse.jdt org.eclipse.jdi
154	100.0%	10.3%	org.eclipse.pde
147	100.0%	12.0%	org.eclipse.team
138	100.0%	15.4%	org.eclipse.core
129	100.0%	8.66%	org.eclipse.pde

4.2 提案手法 2

提案手法 1 では, 適合率は 100%と高い値を示したが, 再現率が低い値に留まった. これは提案手法 1 で得られるクラスタ分割が, 実際のレイヤ分割よりも粒度が細かいことを示している. 一方, 3 節で議論した

ように, 最も粒度の粗いクラスタ分割はモジュール依存グラフ全体を唯一のクラスタとする自明なクラスタ分割であるため, 望ましいレイヤ分割はこれらの2つの分割の間に位置すると考えることができる.

そこで提案手法2では, 自明なクラスタ分割から始めて再帰的にクラスタを分割していくことで解に至るようなアルゴリズムを考える. クラスタの分割に際しては, 上位レイヤが下位レイヤのできるだけ少ないインタフェースに依存するような分割の方法を選ぶ. 具体的には以下の以下のような戦略をとる. まず, レイヤ L の**利用集中度** $\phi(L)$ を,

$$\phi(L) = \sum_{v \in \mathcal{E}(L)} \{d_{\mathrm{IN}}(v)\}^2$$

で定義する. ただし, $\mathcal{E}(L)$ は他のレイヤからの入力辺を持つ L 中の頂点の集合とする. 提案手法2は繰り返しの各ステージにおいて, 現在得られているレイヤ群のいずれか1つのレイヤを選んで分割を行うが, その際, 分割後の全レイヤの平均利用集中度ができる限り大きくなるような分割の方法を選ぶ.

4.2.1 アルゴリズム

平均利用集中度を最大にする分割を求める問題は, 計算に膨大なコストを要すると考えられるため, 以下では近似アルゴリズムを考える. 本論文で用いた近似アルゴリズムの概略を以下に示す. ここで, G を入力となるモジュール依存グラフ, Π_G を求めるクラスタ分割とする.

(1) G の各頂点 v に, "すでに切断した入力辺の数"($\nu(v)$ とおく) を記憶させ, 初期値を 0 に設定する.
(2) 以下の (3)~(5) を C 回繰り返す.
(3) G 中で $(d_{\mathrm{IN}}(v'))^2 + 2 \cdot d_{\mathrm{IN}}(v') \cdot \nu(v')$ を最大にする頂点 v' を選ぶ.
(4) v' から有向パスを通って到達可能な頂点全体からなる G の誘導部分グラフを G_{L} とおく.
(5) G_{L} は G のいずれかの極大連結成分 G_{C} に含まれている. G_{C} を G_{L} とそれ以外の部分に分割する. 分割に合わせて各頂点 v の $\nu(v)$ を更新する.
(6) (3)~(5) の繰り返しが終われば, G の各極大連結成分をクラスタとするクラスタ分割を Π_G として出力する.

本アルゴリズムが出力するクラスタ分割 Π_G に対して G/Π_G が常に有向サイクルを持たないことを容易に証明することができる. 本アルゴリズムの時間計算量は, $O(C \cdot (|V| + |E|))$ となる. 特に G が木であるとき, 本アルゴリズムは各繰り返しにおいて平均利用集中度を最大にする分割を求める.

4.2.2 評価実験

提案手法2を ArgoUML に適用した結果を示す. eclipse に対しても適用を試みたが, 実用時間内に処理が完了しなかった. ArgoUML の解析では, 分割の繰り返し回数 C を 8 回に設定した. 提案手法1と比較して, 適合率は低下したが再現率は上昇した. また解析に要する実行時間は大幅に増加した.

表4 解析システムの実行時間 (提案手法2)

	頂点数	辺数	実行時間 (ミリ秒) † SCALAR
ArgoUML	2228	16601	131,000
eclipse	18092	222700	N/A

† OS:WindowsXP Home Edition, CPU:Intel Celeron 3.33GHz, メモリ:504MB, JVM build 1.6.0.03

表5 提案手法2による ArgoUML の解析結果

クラスタサイズ	適合率	再現率	該当レイヤ
1183	71.5%	67.8%	org.argouml
358	55.0%	91.6%	org.argouml.model ru.novosoft
213	52.5%	41.2%	org.tigris.gef

5. 考　　察

提案手法1による解析では, ArgoUML, eclipse 共にサイズにおける上位クラスタの適合率が 100.0%となった. これはクラスタとして選び出された部品がすべて単独のレイヤに属していることを示している. すなわちクラスタに含まれるどの部品のソースコードを読んでも対応するレイヤの理解向上につなげることができる. さらに抽出された上位クラスタに該当するレイヤはいずれも ArgoUML や eclipse のアーキテクチャにとって重要な部分であった. たとえば eclipse の解析結果では, 文献[4] のアーキテクチャの解説にあるプラグイン開発環境 (PDE), Java 開発ツール (JDT), UI, コアに相当するクラスタがそれぞれ上位に位置している. これも提案手法1の特筆すべき性質である.

一方, 提案手法2では, 目的であった再現率をさらに高めることには成功したが, 適合率が下がってしまうという結果になった. また提案手法2は実行に多大な時間を要することが判明した. 図1は, 2つの手法によって抽出されたクラスタのサイズを比較したものである. 図より, 提案手法2の方が全体的により大きなクラスタを抽出できていることがわかる. 表5において, org.argouml.model と ru.novosoft は ArgoUML

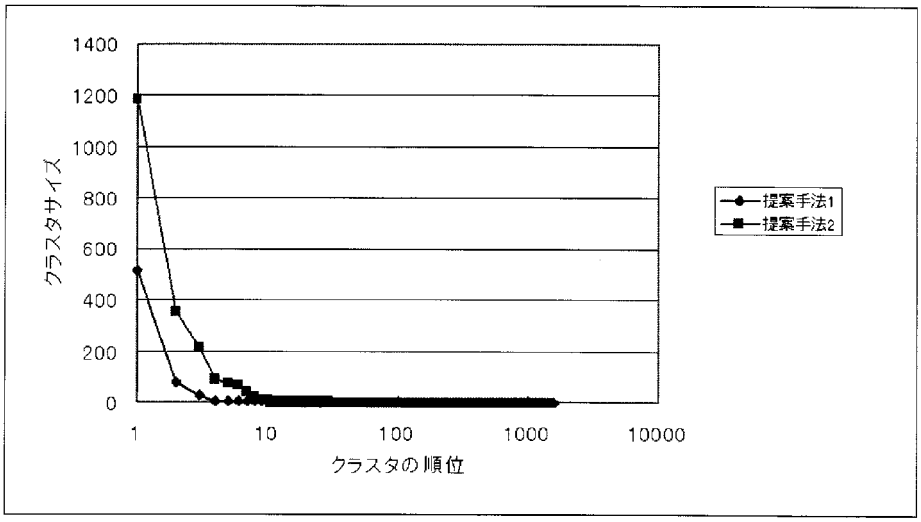

図 1 ArgoUML クラスタサイズ

の中の UML モデルを表現するフレームワークとそのラッパー部分に相当するが, これを再現率が 91.6%と高い割合で抽出できたことは効果的であったといえる. 今後の課題として, 高い適合率を保持したまま再現率を上げていくための効率の良いアプローチを考えたい.

提案手法 1 を以下のように利用することでアーキテクチャ理解に役立てることができると考えられる. まず, 上でも述べたように生成されたクラスタの適合率が 100.0% で, かつ取り出された部品数が多いので, それらの部品を読むことで, 個々のレイヤを効率的に理解することができると考えられる. さらに, 上位クラスタにはアーキテクチャを構成する主要なレイヤが抽出される傾向が強いので, 上位クラスタから読んでいくことで全体の構造を順を追って理解することができる. このように提案手法 1 の結果を利用すれば, おそらくアーキテクチャの理解の手助けとなるだろうと考えられる.

一方, 提案手法 1 は設計の不備の検出にも有用であると考えられる. 図 2 は, 本研究室で開発したゲーム (Tetris) を提案手法 1 で解析した結果である. この図では, 設計者が予想した以上に大きな強連結成分が抽出されている. これは, Block クラスがより具体的な Tetris クラスを参照しているためである. この参照を取り除けば, 抽出される強連結成分は設計者の予想通り小さなものとなる. また, Tetromino1~Tetromino7 クラスの中で Tetromino6 クラスだけが同じ強連結成分に所属していないが, これはこのプログラムに混入している不具合によるものである. このように本手法は設計やプログラムに不備がある場合の発見にも役立つと考えられる.

6. 関連研究

Bunch[9] は疎結合, 高凝集の設計原理に基づいてクラスタリングを行う非常に強力なツールである. しかし, 文献[9] で例題として用いられている対象システムはいずれも規模が小さく, 実用規模のソフトウェアに対して有効であるか否かは不明である. また, この手法ではモジュール依存グラフの有向辺の向きが考慮されていないため, レイヤアーキテクチャなど依存関係の向きが本質的な役割を果たすようなアーキテクチャの抽出に適さないと考えられる.

なお, 文献[8)-10)] 等で指摘されている偏在モジュールが解析結果に与える影響は, 本手法では大幅に緩和される. たとえば解析手法 1 では偏在モジュールの有無に関係なく強連結成分のみが抽出され, 解析手法 2 ではアルゴリズムの性質として偏在モジュールが最初の段階で除外される.

文献[7),13)] では, モジュール依存グラフの有向辺の向きを積極的に考慮して支配頂点とそれに続く部分グラフを抽出する手法が紹介されているが, 本論文で提案しているいずれの手法とも異なるものである.

7. おわりに

レイヤアーキテクチャを持つソフトウェアのためのクラスタリング手法を提案した. 本論文では 2 つの手法を考案し, それらを解析ツール SCALAR として実装

し実規模ソフトウェアを対象に評価実験を行った．その結果，手法の 1 つでは，抽出された大きなクラスタにおいて適合率がすべて 100%となるという非常に顕著な結果を得た．また，クラスタのサイズが大きくなるほどアーキテクチャ上で占める役割も大きくなるという傾向も認められた．このことから，本手法を実規模ソフトウェアのアーキテクチャ理解の支援に用いることができると期待される．一方，再現率の向上を目指したもうひとつの手法は実行に多大な時間を要し，現時点では実用的であるとは言い難い．適合率と再現率を共に高水準で満たし，かつ実用的な時間で実行可能なクラスタリングアルゴリズムを考案することが今後の課題である．

参 考 文 献

1) D. Beyer, A. Noack and C. Lewerents: Simple and efficient relational querying of software structures, In proc. of WCRE'03, 2003.
2) F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal: Pattern-oriented software architecture: a system of patterns, John Wiley & Sons, 1996.
3) Eclipse.org: eclipse, http://www.eclipse.org/, 2007.
4) E. Gamma and K. Beck: Contributing to eclipse: principles, patterns, and plug-ins, Addison Wesley, 2004.
5) R. L. Krikhaar: Reverse architecting approach for complex systems, In proc. of ICSM'97, 1997.
6) J. Knodel, D. Muthig and M. Naab: Understanding software architectures by visualization – an experiment with graphical elements, In proc. of WCRE'06, 2006.
7) S. Li and L. Tahvildari: A service-oriented componentization framework for Java software systems, In proc. of WCRE'06, 2006.
8) J. Luo, L. Zhang and J. Sun: A hierarchical decomposition method for object-oriented systems based on identifying omnipresent clusters, In proc. of ICSM'05, 2005.
9) B. S. Mitchell and S. Mancoridis: On the automatic modularization of software system using the Bunch tool, IEEE trans. on Software Engineering, 32(3), pp.193–208, 2006.
10) H. A. Müller, M. A. Orgun, S. R. Tilley and J. S. UHL: A reverse engineering approach to subsystem structure identification, Journal of Software Maintenance: Research and Practice, 5, pp.181–204, 1993.
11) H. A. Müller, K. Wong and S. R. Tilley: Understanding software sustems using reverse engineering technology, ACFAS'94, 1994.
12) R. W. Schwanke: An intelligent tool for reengineering software modularity, In proc. of ICSE'91, 1991.
13) V. Tzerpos and R. C. Holt: ACDC: An algorithm for comprehension-driven clustering, In proc. of WCRE'00, 2000.
14) Tigris.org: ArgoUML, http://argouml.tigris.org/, 2007.

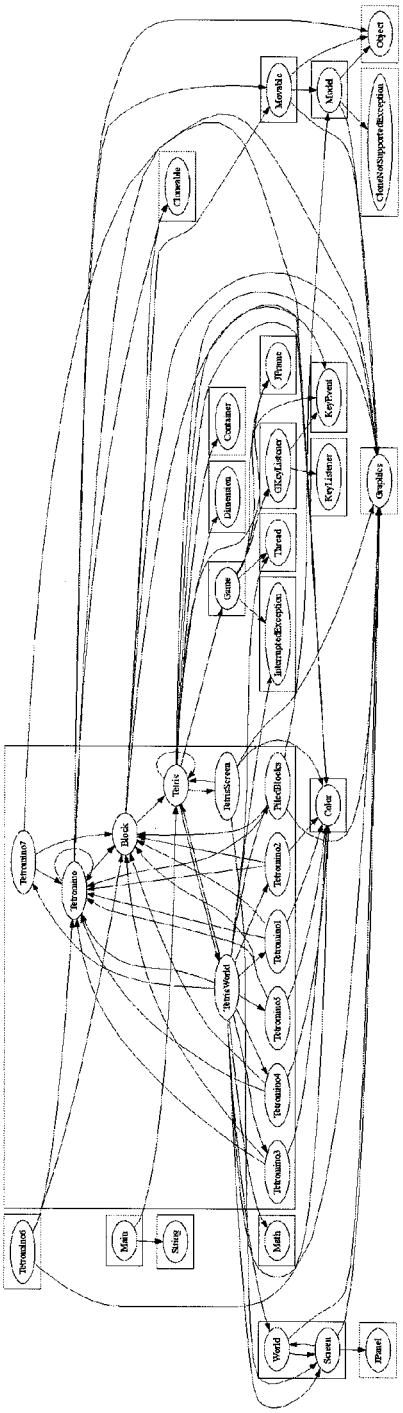

図 2　提案手法 1 による Tetris の解析結果