ソースコードの変更によるデータフロー図の差異検出手法

長岡 源樹　上田 賀一　堀田 大貴　高橋 竜一

ソフトウェアの保守作業において設計書をもとにソフトウェアの概要を理解できることが望ましい．しかし，軽微な保守作業ではソースコードを変更しても，この変更を設計書に反映しないまま放置することも多い．これにより設計書とソースコードの対応が次第に取れなくなり，設計書によるソフトウェアの理解は困難になってしまう．そこで，ソースコードの変更を設計書に反映させる工程を支援することで，保守作業を支援することを考える．今回は業務システムの設計時に使用されることが多いデータフロー図を対象に，ソースコードの変更を設計書に反映させる手法を提案する．本手法は，初期開発時のデータフロー図とソースコードの対応関係から，変更後のソースコードの差分をデータフロー図に反映させる．これにより変更をデータフロー図に反映させ易くなり，保守作業のコストの削減が期待される．

It is desirable to be able to understand the outline of the software based on the design document in software maintenance work. However, in a minor maintenance work, even if you change the source code, you often leave this change without reflecting it in the design document. As a result, correspondence between design document and source code becomes gradually impossible, and it becomes difficult to understand software by reading design document. Therefore, we consider supporting the maintenance work by supporting the process of reflecting the change of the source code in the design document. In this paper, we propose a method for reflecting change of source code in design document, for data flow diagram which is often used when designing business system. In this approach, the difference of source code after change is reflected in the data flow diagram from the correspondence relationship between the data flow diagram and the source code which can be assumed that correspondence can be taken at the time of initial development. This makes it easy to reflect changes in the data flow diagram, and it is expected that the cost of maintenance work will be reduced.

1　はじめに

1.1　背景

近年におけるソフトウェア開発は，ソフトウェアの保守を行う割合が多い．保守工程では設計書を元にバグの修正や機能修正，機能拡張等が行われる．コストの割合では要件分析 3%，基本設計 3%，詳細設計 5%，実装 7%，製品段階では運用と保守 67%，テスト 15%とされている [5].

初期開発時には，まずソフトウェアの設計をし，設計書に沿ってソースコードを作成する．要求分析時や設計時には，ソフトウェアに何をさせたいのかを明確にしたり，どのような機能があり，仕様はどうなっているのかを明確にするため，UML やデータフロー図等を用いて設計書を作成する．

保守工程でバグの修正や機能拡張を行う場合，既存のソフトウェアが何をするためのソフトウェアなのか，どのような仕様なのかを理解しなければならない．理解するためには，ソースコードを読んだり，設計書を読む必要がある．ソースコードは人には理解しにくいものであるため，一般的にはまず設計書を読んでソフトウェアの概要を理解する．次にソースコードと設計書の変更を行う．しかし，軽微な保守作業ではソースコードのみ変更し，この変更を設計書に反映しないまま放置することも多い．これにより設計書とソースコードの対応が次第に取れなくなり，設計書からソフトウェアを理解することが困難になると，後の保守作業に支障が出てしまう．

Detection of difference in data flow diagram due to source code change

Motoki Nagaoka, Yoshikazu Ueda, Hiroki Horita, Ryuichi Takahashi, 茨城大学, Ibaraki University.

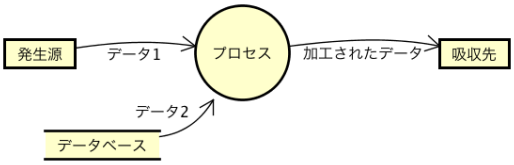

図 1　データフロー図の例

業務アプリケーションの設計にデータフロー図が良く使われている．データフロー図はデータがどのように変化していくかを表現した図である．

1.2　目的

業務アプリケーションの設計に良く使われるデータフロー図に注目し，ソースコードの変更をデータフロー図に反映させることで設計書の保守作業を支援する．

ソースコードの情報から完全なデータフロー図を作成することは困難であるため，データフロー図上での変更を提示し，開発者がデータフロー図を変更し易くすることを目的とする．

本研究では，次の手法を提案する．まず始めに，初期開発で作られたデータフロー図，これと対応が取れているソースコードを利用し，データフロー図とソースコードの対応関係を調べる．このときメソッド間データ依存グラフを作成し，コールグラフと Graph Edit Distance の情報を用いて対応付けを行う．次に，保守工程で変更されたソースコードを解析し，元のソースコードとの差異を検出する．最後に，元のソースコードと変更後のソースコードの差異から，変更箇所をデータフロー図に反映させる．

2　関連知識

2.1　データフロー図（DFD）

データフロー図（DFD）[2] は，情報システムのデータの流れをグラフィカルに表現する図である．システム設計段階の初期に描かれることが多い [1]. また，構造化分析で利用され，業務上のシステムの設計時に使われることが多い．

データフロー図の構成要素を以下に示す．

プロセス

円 (図 1 の“ プロセス ”) で表す．データの処理・変換を示し，入力データを処理し，結果となるデータを出力するもの．

データフロー

矢印 (図 1 の“ データ 1 ”等) で表す．各要素間のデータの流れを表す．データの流れる方向を矢印で表し，データフローに名前をつけてデータを識別する．

データストア

2 本の平行線 (図 1 の“ データベース ”) で表す．データの格納場所を示す．

外部実体

四角形 (図 1 の“ 発生源等 ”) で表す．システム外部に存在するものであり，外部の情報源や外部への情報の出力先を示す．

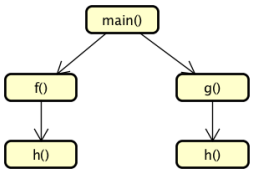

図 2　コンテキストを理解したコールグラフ

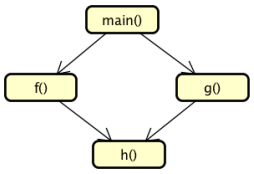

図 3　コンテキストを理解しないコールグラフ

2.2　コールグラフ

コールグラフとは，プログラムのメソッド同士の呼び出し関係を表現した有向グラフである．各ノードがメソッドを表現し，各エッジ (f, g) はメソッド f がメソッド g を呼び出すことを示す．

コールグラフは様々な形で表現できる．例えば図 2 のコンテキストを理解したコールグラフ (各メソッドについてメソッドが呼び出されるコールスタックごとに別々のノードを持つグラフ) や，図 3 のコンテキストを理解しないコールグラフ (各メソッドについてノードを 1 つしか持たないコールグラフ) 等が挙げられる．

2.3 Graph Edit Distance

Graph Edit Distance(以下 GED とする) とは，2 つのグラフの類似性を表すものである．片方のグラフを編集してもう片方のグラフを作り出すときの，編集コストの最小値である．グラフ g_1 と g_2 の GED を，$GED(g_1, g_2)$ と表す．

グラフの編集操作には以下のものがある．

- ノードの追加
- ノードの削除
- ノードの変更
- エッジの追加
- エッジの削除
- エッジの変更

3 提案手法

本研究の目的は，ソースコードの変更をデータフロー図 (以下 DFD とする) に反映させることで設計書の保守作業を支援することである．プログラムのみ変更されたソフトウェアに対し，変更前の DFD とソースコード，変更後のソースコードから，変更前と変更後の DFD 上での差異を検出し，変更を DFD に反映させる．

本手法の概要を DFD で表したものを図 4 に示す．本手法では，ソースコードから DFD を作成するため，メソッド間のデータ依存関係をグラフで表したメソッド間データ依存グラフ (3.1 節) を利用する．

初めに変更前のソースコードのメソッド間データ依存グラフを作成 (3.2 節) し，このグラフのノードと DFD 上のノード (プロセス) との対応関係を求める (3.3 節)．次に，変更後のメソッド間データ依存グラフを作成し，このグラフと変更前のグラフとの差異を検出する (3.4 節)．最後に，このメソッド間データ依存グラフ上での差異を DFD に反映させる (3.5 節)．

3.1 メソッド間データ依存グラフ

DFD のプロセスはデータを処理するものであるため，メソッドの組み合わせであると考えられる．そこで，メソッド間のデータ依存関係を表現したグラフを定義する．メソッドをノード，データ依存関係（3.1.1 節）をエッジとした有向グラフとする．

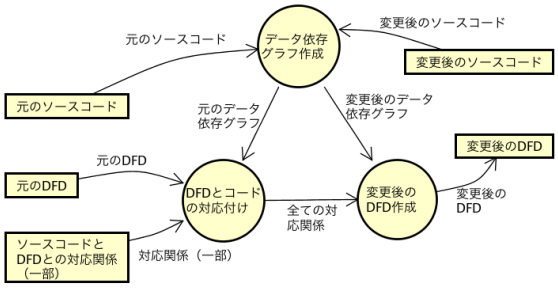

図 4　提案手法の概要（DFD）

3.1.1 データ依存関係

メソッド間のデータ依存関係を定義する．次のとき，データ依存関係があるものとする．

メンバ変数への代入・参照

メソッド f がメンバ変数 v へ値を代入している

メソッド g がメンバ変数 v の値を参照している

以上の 2 つを満たす時，メソッド f からメソッド g へのデータ依存関係があるとする．

メソッドへの引数での受け渡し

メソッド f が引数を取るメソッド g を呼び出している場合，メソッド f からメソッド g にデータ依存関係があるとする

メソッドの返り値での受け渡し

メソッド f が値を返すメソッド g を呼び出している場合，メソッド g からメソッド f へのデータ依存関係があるとする

3.2 メソッド間データ依存グラフの作成

ソースコードの静的解析によりデータ依存関係を解析し，メソッド間データ依存グラフ G を作成する．3.1.1 節でのデータ依存関係を見つける毎に，グラフ G にノードとエッジを追加していく．

3.3 DFD とメソッド間データ依存グラフの対応付け

3.2 節で作られたグラフ G と，DFD との対応付けを行う．

メソッド間データ依存グラフは DFD に比べて非常に抽象度が低く，このままでは DFD との対応が分かりにくい．そこで，メソッド間データ依存グラフの

ノードのグループ化を行い DFD の抽象度に近づけていく．また，ソースコードからではどの部分が外部実体・データストアへの入出力なのか特定できないため，外部実体・データストアへの入出力の箇所は手動で指定する必要がある．

初めに DFD の外部実体・データストアへの入出力箇所と対応付けを行い (3.3.1 節)，グラフ G のノードをグループ化して DFD の形に近づけ (3.3.2 節)，DFD のプロセスと対応付ける (3.3.3 節) ことで行う．

3.3.1 外部実体・データストアとの対応付け

ソースコードを解析するだけでは外部実体・データストアの対応を特定できないため，外部実体・データストアとの対応は手動で指定する．ここでは外部実体・データストアとの対応付けに必要となる情報と，グラフ G への対応付けの反映方法を述べる．

「DFD 上に現れている全ての外部実体・データストアと繋がっているデータフロー」に対し，「1 対 1 で対応する，そのデータフローを作り出しているメソッド」がソースコード上に存在していることを前提とし，この対応関係を必要とする．また，DFD ではデータフローの名前は通常一意なものであり，同じデータフロー名が存在する箇所はデータストアへの入出力の部分のみと考えられる．そのため，それぞれのデータフロー名とデータフローの方向（外部実体・データストアに対して入力か出力か) とそれに対応するメソッド名 (パッケージ名，クラス名，メソッド名) の対応関係の情報があれば外部実体・データストアとの対応付けが行える．

この対応関係をグラフ G に反映するためには，データフロー名をノードとしたものをグラフ G に追加し，対応するメソッドノードへの依存関係のエッジを追加することで行う．

3.3.2 メソッド間データ依存グラフのノードのグループ化

DFD のプロセスはメソッドの組み合わせと考えられるため，ノードをグループ化する (メソッドをまとめる) ことで，グラフ G を DFD と対応した形に変形できると考えられる．グループ化されたノード群を 1 つのノードとみなすことでグラフ G の抽象度を上げることができる．

ここでのグラフ G には外部実体・データストアとの対応付け（3.3.1 節）は完了しているものとする．

纏まった処理を 1 つのノードとみなし，そのノードが DFD 上のプロセスと完全に対応していることが理想となる．コールグラフ上で深い位置にあるメソッドは細かい処理をしていると考えられるため，コールグラフ上での位置が深いメソッドから順にグループ化していく．グラフ G のノードは各メソッドに対して 1 つのノードを持っているため，同様の構造となる図 3 のコンテキストを理解しないコールグラフを利用する．手順を次に示す．

1. コールグラフ G_{call} を作成する
   (a) メソッド f の定義内でメソッド g を呼び出している場合,G_{call} にノード f からノード g へのエッジを追加する
   (b) 全てのメソッド定義に対して行う
2. G_{call} 上で最も深い位置にあるメソッド（G_{call} 上で先頭のノードとの直径が最大のノードのメソッド）を探し，このメソッドをメソッド A とする
3. G 上でメソッド A のノードのデータ依存元のノードがある場合，そのノードのうちの 1 つとグループ化する
   また，メソッド A が外部実体・データストアとの依存関係が存在する場合でも，この依存関係は存在しないものとして扱う（外部実体・データストアとのグループ化は行わない）
4. G 上でメソッド A のノードのデータ依存元のノードがない場合，データ依存先のノードのうちの 1 つとグループ化する
   また，メソッド A が外部実体・データストアとの依存関係が存在する場合でも，この依存関係は存在しないものとして扱う（外部実体・データストアとのグループ化は行わない）
5. G_{call} 上のメソッド A のノードを削除する
6. 2 〜 5 を G のノード数が DFD のプロセス数と一致するまで繰り返す

3.3.3 プロセスとの対応付け

グラフ G のノードと，DFD のプロセスの対応付けを行う．DFD と形が最も類似するように対応付けることで，プロセスとの対応付けできると考えられる．

DFD のプロセスをノード，データフローをエッジとして，G と DFD の Graph Edit Distance が最小となるように G のノードに DFD のプロセスを割り当てる．

グラフ G のノード数と DFD のノード数は一致しており，エッジの区別はしていない．そのため，GED を求める際のグラフの編集操作はエッジの追加，エッジの削除のみを利用し，それぞれのコストは 1 とする．

3.4 メソッド間データ依存グラフの差異検出

元のソースコードのメソッド間データ依存グラフ G_p，変更後のソースコードのメソッド間データ依存グラフ G_n を比較し，差異を検出する．G_p は DFD との対応付けまで行われているものとする．次の手順で差異を検出する．

1. 変更後のソースコードからメソッド間データ依存グラフ G_n を作成する
2. G_n のノードを，G_p 内で同じグループとなっているノード同士でグループ化する
3. グループ化されなかったノードがあった場合，そのノードは変更によって生じたことが分かる

3.5 変更後の DFD 作成

3.4 の差異検出により得られたグラフ G_n を，元の DFD と対応付けされたグラフ G_p に基づいて DFD のプロセスを対応付ける．このグラフ G_n に外部実体・データストアを追加し，プロセスと対応づいたノードにプロセスの名前を付けると，変更によって追加されたメソッドがプロセスとして追加された DFD が作れる．ただし，この追加されたプロセス名はメソッド間データ依存グラフでのノード名であり，データフロー名は付けることができない．

4 適用事例

対象とするプログラムと DFD に提案手法を適用し，プログラムと DFD の対応関係が正しく取れているか，プログラムの変更が DFD に正しく反映されるかを確認する．

今回の適用事例では，Java 言語により開発された料理レシピ管理を行う料理レシピ帳プログラム (筆者が作成したもの) を対象とする．

表 1　料理レシピ帳プログラムの規模

定義されたメソッド数	102
コード行数	1,523
ファイル数	21

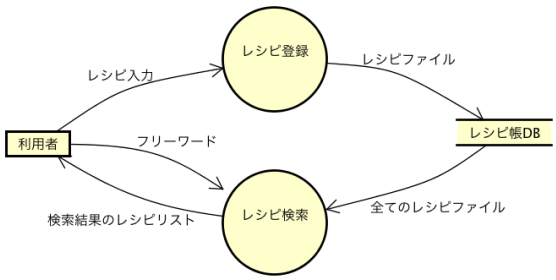

図 5　料理レシピ帳プログラムの DFD

4.1 料理レシピ帳プログラム

料理のレシピを管理するプログラムである．レシピの登録と，レシピの検索が行える．レシピの登録では，レシピ名，材料名，材料の分量，料理の手順を記入する．レシピの検索では，フリーワードで検索し，レシピ名，材料名，料理の手順の中で，一致する箇所が存在するレシピを表示する．

この料理レシピ帳プログラムの DFD を図 5，プログラムの規模を表 1 に示す．

4.2 メソッド間データ依存グラフの作成

レシピ帳プログラムのソースコードからメソッド間データ依存グラフ G_p を作成する．この結果，G_p は 202 個のノードを持つグラフとなった．定義されたメソッドのノードは 102 個であり，残りの 100 個はライブラリ等のメソッド呼び出しのノードとなる．

4.3 DFD とメソッド間データグローグラフの対応付け

4.3.1 外部実体・データストアとの対応付け

グラフ G_p 上に外部実体・データストアのノードを追加し，外部実体・データストアとのデータフローを作り出しているメソッドのノードと依存関係のエッジを追加する．

図 5 の外部実体・データストアと繋がっているデー

表 2　図 5 の外部実体・データストアとの対応付け

データフロー	対応するメソッド
レシピ入力	RegistrationPage.register()
レシピファイル	RecipeData.writeObject()
全てのレシピファイル	FileManager.getRecipeData()
フリーワード	SearchPage.getText()
検索結果のレシピリスト	SearchPage.showResult()

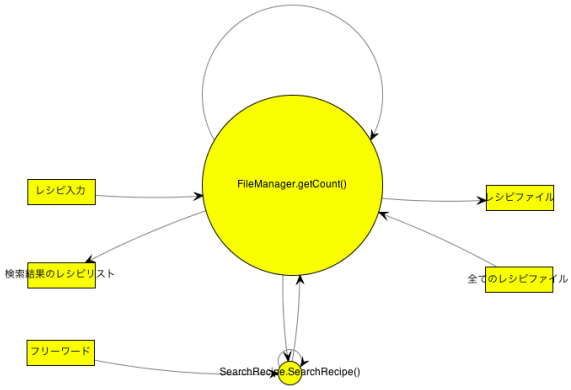

図 6　グラフ G_p のグループ化結果

タフローを作り出しているメソッドを表 2 に示す．

表 2 から，5 個のノードをグラフ G_p に追加した．

4.3.2　メソッド間データ依存グラフのノードのグループ化

DFD との対応を取るため，グラフ G_p のノードをグループ化する．4.3.1 節で追加したノード以外のノードの数が，DFD のプロセスの数と同じ数になるまでグループ化する．

グループ化した結果を図 6 に示す．4.3.1 節で追加した，外部実体・データストアとの対応付ノードは四角形で表し，円の大きさはグループに含まれるノード数を表している．fileManager.getCount() ノードは 194 個のノードのグループとなり，SearchRecipe.SearchRecipe() ノードは 8 個のノードのグループとなった．

4.3.3　プロセスとの対応付け

グラフ G_p のノードと，DFD のプロセスの対応関係を求める．

1. *FileManager.getCount*() = レシピ登録，*SearchRecipe.SearchRecipe*() = レシピ検索とするとき $GED(G_p, G_n) = 8$
2. *FileManager.getCount*() = レシピ検索，*SearchRecipe.SearchRecipe*() = レシピ登録とするとき $GED(G_p, G_n) = 10$

以上から，GED が最小となる，1 の対応付けを行う．

表 3　変更後のレシピ帳プログラムの規模

定義されたメソッド数	107
コード行数	1,623
ファイル数	23

4.4　メソッド間データ依存グラフの差異検出

4.4.1　ソースコードの変更

変更前のソースコードと，変更後のソースコードの差異検出の準備として，ソースコードの変更を行う．レシピ帳プログラムにレシピのお気に入り登録機能を追加する．お気に入りに登録でき，お気に入りリストからレシピにアクセスできるようにする．変更後のレシピ帳プログラム

変更後の料理レシピ帳プログラムの規模を表 3 に示す．

4.4.2　差異検出

変更後のソースコードから作成したメソッド間データ依存グラフを G_n とする．グラフ G_p でグループ化されているノードを，グラフ G_n でもグループ化していく．グループ化を行った結果を図 7 に示す．この結果から，FileManager.getCount() ノード内のメソッドとデータ依存関係がある変更が行われたことが分かる．

4.5　変更後の DFD 作成

図 7 を，グラフ G_p と DFD との対応関係から外部実体・データストアを追加し，グループ化されたノードを 1 つのプロセスとしプロセス名を付け，データフローに名前をつける．こうして作られた DFD を図 8 に示す．また，変更により生じたプロセスは水色で示している．

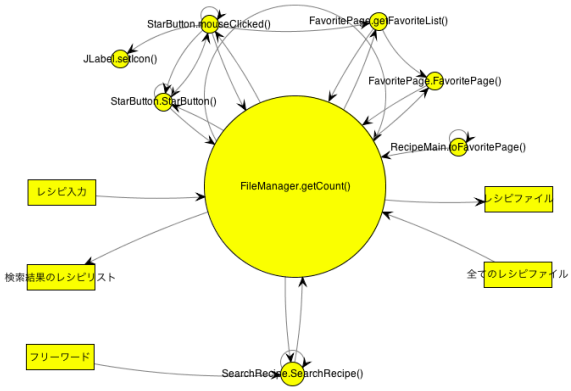

図 7　グラフ G_n のグループ化結果

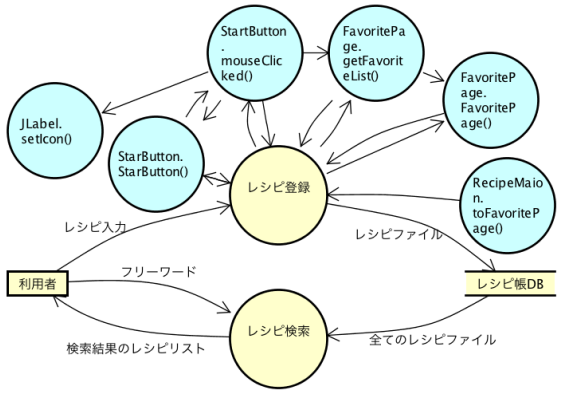

図 8　変更後の DFD の作成結果

5　考察

今回の実験では，ソースコードの変更によるデータ依存関係の差異を検出し，可視化することができた．しかし，差異を DFD に反映できたとは言えない結果となった．

5.1　ノードのグループ化について

ノードをグループ化した結果，194 個のノードからなるグループと 8 個のノードからなるグループが出来上がった．2 つのプロセスで共通して利用するメソッドが FileManager.getCount() ノードにグループ化されてしまい，この影響で SearchRecipe.SearchRecipe() ノードにグループ化されるべきであったノードも FileManager.getCount() ノードにグループ化されてしまったと考えられる．そのため，正しく (DFD 上のプロセスに近づくように) グループ化されたとは言えない結果になった．改善する必要があるため，良い結果が出なかった原因を考察する．

この料理レシピ帳の場合，登録されたレシピデータをファイルで保存している．このレシピデータ (ファイル) を読み書きするメソッドは，レシピ登録・レシピ検索の両方で使われている．そのため，このメソッドのノードによって別のプロセスのノードもグループ化されてしまい，プロセスに合わせたグループ化ができなかったと考えられる．このような，複数のプロセスで使われる汎用的なメソッドは他にも様々なものが考えられる．汎用的なメソッドを識別してこのノード経由でのグループ化を防ぐか，オブジェクトを識別してオブジェクトごとにノードを分ける等の対策が考えられる．

5.2　差異検出について

今回の実験では，メソッドの追加によるデータ依存関係の差異を検出できた．追加されたメソッドについて，そのメソッド同士にどのようなデータ依存関係があるかがが分かりやすくなり，保守作業の支援に役立てられると考えられる．

しかし，この差異は全てレシピ登録プロセスとデータ依存があるとして検出された．これはレシピ登録プロセスのノードグループにノードが偏ってしまったためであり，ノードのグループ化を改善することで正しく検出されるようになると考えられる．

また，グループ化されたノード内のデータ依存関係の差異も可視化することによって，保守作業に有益な情報が得られると考えられる．

5.3　変更後の DFD 作成について

生成された変更後の DFD を見ると，今回の適用事例ではソースコードの変更は小規模なものであったが，メソッド名がそのまま出てくるので見難いものになっている．ソースコードの変更の規模が大きくなるとこの問題は顕著になると考えられる．ある程度グループ化したものを反映させたり，Javadoc 等のコメントをプロセス名にする等で対応できる可能性が

ある．

6 関連研究

変数間のデータフローを表す変数間データフローグラフを作成してエディタに表示し，ソースコード間の移動を支援することで開発者を支援する研究 [3] が行われている．

丸山らは，ソースコードの変更によるシーケンス図の差異を検出し，変更をシーケンス図に反映させることでソフトウェア保守作業を支援する手法 [4] を提案した．

本研究では業務アプリケーションの設計でよく使われる DFD に着目し，ソースコードの変更によ DFD の差異を検出し，変更を DFD に反映させることでソフトウェア保守作業を支援する．メソッド間のデータ依存関係を解析し，この依存関係の差異を DFD に反映させる．

7 おわりに

7.1 まとめ

本研究では，ソースコードのみ変更されたソフトウェアに対し，変更前のデータフロー図（DFD) とソースコード，変更後のソースコードから，変更前と変更後の DFD の差異を検出する手法を提案した．ソースコードの解析の際にメソッド間データ依存グラフを作成し，コールグラフの情報を利用したノードのグループ化によって DFD の形に近づけ，DFD と対応付けを行う．これにより，ソースコードと DFD の対応関係を解析し，ソースコードのみ変更されたソフトウェアに対し，ソースコードの変更を DFD に反映させる．

適用事例では，料理レシピの管理を行う小規模なソースコードに対して本手法を適用した．その結果，様々なメソッドが追加されたことが検出できたが，変更された箇所が正しく DFD に反映されたとは言えない結果となった．ノードのグループ化を改善していくことで変更の DFD への反映が改善できると考えられる．

7.2 今後の課題

7.2.1 ノードのグループ化の改善

現状ではグループ化した結果のノードが，DFD のプロセスと対応しているとは言えない．汎用的なメソッドを識別してこのノード経由でのグループ化を防ぐか，オブジェクトを識別してオブジェクトごとにノードを分ける等の対策が考えられる．

7.2.2 抽象化

生成された DFD の，変更箇所のプロセスは抽象度が非常に低く見難いものとなっている．この抽象度を高めることで，DFD を見た時にどの部分でどのような変更があったのか分かりやすくできると考えられる．

参考文献

[1] Bruza, P. D. and Van der Weide, T. P.: *The semantics of data flow diagrams*, University of Nijmegen, Department of Informatics, Faculty of Mathematics and Informatics, 1989.

[2] Demarco, T., and 高橋智弘: 黒田 順一郎 (訳): 構造化分析とシステム仕様, 1994.

[3] 悦田翔悟, 石尾隆, 井上克郎, ほか: 変数間データフローグラフを用いたソースコード間の移動支援, 研究報告ソフトウェア工学 *(SE)*, Vol. 2011, No. 12, pp. 1–8 (2011).

[4] 丸山翔平, 上田賀一: ソフトウェア保守のための UML シーケンス図維持支援, 日本ソフトウェア科学会大会論文集, Vol. 32, 9p (2015).

[5] Zelkowitz, M. V., Shaw, A. C., and Gannon, J. D.: *Principles of software engineering and design*, Prentice-Hall Englewood Cliffs, 1979.