ソースコードからCDFDへの変換によるSOFL仕様記述の支援ツールの提案

新城 汐里
法政大学情報科学研究科
shiori.araki.9n@stu.hosei.ac.jp

劉 少英
法政大学情報科学研究科
sliu@hosei.ac.jp

要旨

ソフトウェアを開発する際に機能を定義する仕様の記述が疎かになる場合がある．これはレガシーソフトウェアにも該当することであり，仕様記述が疎かであるどころか存在しない場合もある．この問題を解決するために*Java*ソースコードから*CDFD*を生成し，*SOFL*形式仕様の記述を支援するツールを提案する．これにより仕様記述にかかるコストを軽減し，ソースコード中のバグの検出を助ける．

1. はじめに

ソフトウェア開発において機能を定義する仕様の記述が疎かになる場合がある．これは現在開発されているソフトウェアに限らずレガシーソフトウェアにも当てはまる．加えて仕様の紛失や，仕様が初めから記述されていないというような場合もある．また，ソースコード中のバグを検出するためにレビューという手段が存在するが，レビューは行う人物に依存するという問題がある．

これらの問題を解決するためにリバースエンジニアリングという手段があり，研究がされてきた．例えばBenedusiらはPascalで書かれたソースコードからDFD(データフロー図)を生成するための方法とDFDがソフトウェアの理解と保守に対して持つ有用性についての考察について述べており[1]，A. B. O'HareらはRE-Analyzerというツールの開発によるC言語で実装されたソフトウェアに対するリバースエンジニアリングの支援を行っている[2]．また井垣らはDFDを用いてレガシーソフトウェアの依存解析を行い，サービス指向アーキテクチャのサービス候補を抽出する手法を提案している[3]．

本研究ではJavaソースコードからCDFD(条件データフロー図)およびSOFL形式仕様を半自動的に生成するための変換規則を述べ，その規則を用いたSOFL形式仕様を記述するための支援ツールの研究開発を行う．SOFL(Structured Object-Oriented Formal Language)[4, 5]は構造化手法とオブジェクト指向言語を融合した仕様記述言語を提供する形式工学手法であり，Javaから形式仕様への変換も容易であるため本研究にとって有用である．ツールの目的は過去に作成されたソースコードを構文解析することでCDFDの自動的な生成やSOFL形式仕様の半自動的な生成を行い，ソースコードとCDFDの両方を見ながら生成されなかった部分の仕様をユーザが手動で記述することによって仕様の不備を発見できるよう支援することである．

以降，2節ではSOFLの概要およびその内容を簡潔に述べる．3節ではJavaソースコードからCDFDへの変換規則を定義する．4節ではJavaソースコードからSOFL形式仕様へ変換する際の方針を述べる．5節では定義した変換規則を基に開発した支援ツールについて述べる．6節では開発した支援ツールを用いて簡単なソースコードによる事例研究を行う．7節では事例研究で得られた結果と今後の課題をまとめる．

2. SOFLの概要

本節ではSOFLの概要について述べる．SOFLはDFD[6]，ペトリネット[7]，VDM-SL[8, 9]を統合した形式工学手法[10]である．要件と仕様の設計に対して形式仕様記述言語を提供することによりソフトウェアシステムを開発するための実用的な方法を提供している．この手法を用いた仕様はJavaやC++のようなオブジェクト指向言語への変換に適している．以降，単純な自動販売機のシステムを事例としてCDFDとSOFLのモジュールの構造について述べる．

2.1. CDFD の概要および事例

CDFD は 2 節で述べた 3 つの手法を統合し形式化された DFD を指す．SOFL を構成するものの 1 つであり，SOFL のモジュールと関連付けられた図である．システムの構造を描画したものでもあり，あらかじめ定義された機能要件と機能間の依存関係を表現する．また，CDFD は階層構造である．1 つのプロセスを分解することで低レベルのより詳細な複数のプロセスを表現できる．数字のない長方形はプロセスを意味し，入出力を持った 1 つの操作を表す．名前の付いた矢印はデータフローを意味する．データフローにつけられた名前はそのデータの性質を，矢印はデータフローの方向を表す．数字の ID がつけられた長方形はデータストアを意味し，ファイルやデータベースのようなデータを与える役割を持つ．

DFD との主な相違点は 3 点である．1 点目は破線で表した制御データフローを用いている点である．このデータフローはプロセスにデータを渡すことなく実行の指示を出す場合に用いる．2 点目はプロセスの表現が詳細である点である．プロセスはプロセス名 (中央部)，入力ポート (左部)，出力ポート (右部)，事前条件 (上部)，事後条件 (下部) で構成される．また，入力ポートと出力ポートは複数存在する場合がある．これは入力あるいは出力が複数通り存在することを表す．3 点目は入力あるいは出力を持たずともプロセスとして表現できる点である．そのため入力はないが出力のあるプロセスや，入力はあるが出力のないプロセスを表現することもできる．

事例として図 1 に電子マネーで商品を購入できる自動販売機の CDFD を示す．ただし，この CDFD はあくまで図の流れを見るために簡易的に記述されたのものであり現実に使用されている自動販売機とは細部が異なる．この図 1 における動きは次の通りである．まず自動販売機で購入する商品を選択したことを表すデータフロー *button* はプロセス *Pressed* で処理を行う．このプロセスはデータストア *stock_of_drink* に在庫があるかどうかを確認する．在庫がないなら在庫がないメッセージを表すデータフロー *out_of_stock_message* を生成し，在庫があるなら在庫が存在することを伝えるデータフロー *confirmation* を生成してプロセス *Selling* に渡す．そしてデータフロー *confirmation* を受け取ったプロセス *Selling* は投入金額を表すデータフロー *money* も受け取る．このプロセスではデータストア *stock_of_drink* の在庫から商品を取り出す．投入金額がデータストア *stock_of_drink* に記録された商品の値段未満であればエラーメッセージを表すデータフロー *error_message* と返金を表すデータフロー *repayment* を生成し，投入金額が商品の値段以上であればデータストア *stock_of_drink* に該当する在庫を減らすような変更を加えてから商品を表すデータフロー *drink* とおつりを表すデータフロー *change* を生成する．

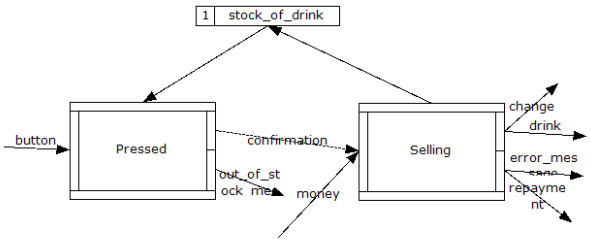


図 1. 自動販売機の CDFD

2.2. モジュールの構造

SOFL のモジュールは CDFD で表現したプロセス，データフロー，データストアを正確に定義するものである．事例として図 1 をモジュールとして定義したものを表 1 に示す．これらも例示のために自然言語を用いて簡単に記述したものである．表 1 において，モジュール名は「自動販売機」とする．このモジュール内で宣言する型は商品を表す Drink と選択した商品を表す Button であり，ストア変数は商品の在庫と値段を表す *stock_of_drink* である．このモジュールの振る舞いを表す CDFD は behav に番号が記述される．プロセス *Init* は変数の初期化を行うプロセスである．プロセス *Pressed* および *Selling* は 2.1 節の説明と同様である．

また，プロセス *Selling* を定義したものを表 2 に示す．表 2 において，プロセス名は *Selling* である．このプロセスに対する入力は *money* と *confirmation* であり，このプロセスからの出力は *drink* と *change* もしくは *error_message* と *repayment* のどちらかである．操作を実行するにあたってデータストア *stock_of_drink* に対し読み込みおよび書き込みを行う．事前条件として *money* が 0 より大きく，*confirmation* を受け取っていることが定義されている．事後条件として，2.1 節で説明したようにプロセスが実行すべき操作が定義されている．

表 1. SOFL モジュールの構造

module 自動販売機; **type** Drink=Drink の型の定義; Button=Button の型の定義; **var** stock_of_drink : stock_of_drink の型; **behav** CDFD_1; process Init; process Pressed; process Selling; **end_module**

表 2. プロセス仕様の構造

process Selling (money: int, confirmation: sign) drink: Drink, change: int | error_message: string, repayment: int **ext** wr stock_of_drink **pre** money>0 and confirmation を受け取っている **post** if money が商品の値段未満である then error_message の生成 and repayment=money else stock_of_drink から商品の取り出し and drink を生成 and change=money− 商品の値段 **end_process**

3. CDFD への変換規則

本節ではソースコード中の構造から CDFD への変換規則を定義し，その原理を述べる．ただし，定義する変換規則は構文解析を用いて機械的に変換することを前提とする．また，規則を用いた変換の事例を紹介する．

3.1. 順序構造の変換

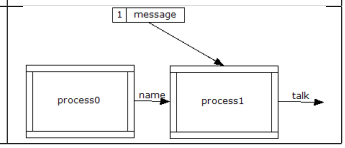
本節では順序構造における CDFD への変換規則の定義について述べる．これは本節で述べる変換規則の基礎となる定義である．順序構造の変換規則を以下に定義し，その理由を述べる．

- 1 つのステートメントを 1 つのプロセスとする．1 つのステートメントがそれ以上分解するのが難しい 1 つのまとまった処理だからである．
- 基本データ型もしくは見た目の挙動が基本データ型と同じように見える型を持つ 1 つのローカル変数を 1 つのデータフローとする．基本データ型の変数をメソッドに渡し，それに変更が加えられても元の変数には影響がなくフローとして表現しやすいためである．
- 参照型である 1 つのローカル変数，もしくは型に依らない 1 つのフィールド変数を 1 つのデータストアとする．参照型の変数をメソッドに渡し，それに変更が加えられた場合に元の変数にも影響がありフローとして表現しづらいためである．
- データフロー名およびデータストア名には変数名を用いる．
- データフローの方向やデータストアへの接続の方向はステートメントもしくは式の中における変数が使用された位置から決定する．例えば変数が宣言された場合はその変数を出力として扱い，式の中で計算に使用されたが値の変更がないような場合は入力として扱う．
- 同じデータストアへ接続するプロセス同士は実行順に制御データフローの始点と終点となる．データストアへ接続する順番を明確にする必要があるからである．

以上の定義を用いて Java ソースコードから CDFD に変換した場合の事例を表 3 に示す．

表 3. 順序構造の変換例

ソースコード	CDFD への変換例

`public static String message;`
`public static void function(){`
`  String name="I";`
`  String talk=name+" say "+message;`
`}`
	

3.2. 選択構造の変換

本節では選択構造における CDFD への変換規則の定義について述べる．選択構造のステートメントは条件と処理の 2 種類から構成される．ここで述べる選択構造は if 文についてであり，switch 文は本稿では言及しない．

CDFD への変換が変則的かつ正しくは選択構造に分類されないからである．以上を踏まえて選択構造の変換規則を以下に定義し，その理由を述べる．

- 1 つのプロセスはそのプロセスを親プロセスとした複数の子プロセスを持つ場合がある．1 つの選択構造が複数の処理を持つことが可能だからである．この規則を適用することによって CDFD の階層構造を表現することが可能となる．
- 条件は子プロセスとしてではなく，親プロセスの一部と見なして変換する．条件はステートメントではなく式として扱われるからである．
- 子プロセスに相当するステートメントは該当する変換規則に沿って変換を行う．
- 選択された処理を表すプロセスに制御データフローを接続する．選択しなかった処理を実行しないようにするためである．
- 選択に関係なくプロセスにつながる可能性のある全てのデータフローを表現する．静的なコード解析のみでは選択の特定がほぼ不可能なためである．
- 子プロセスを持つ親プロセスは子プロセスへ接続データフローも親プロセスに接続しているものとして表現する．子プロセスは親プロセスを構成する一部だからである．

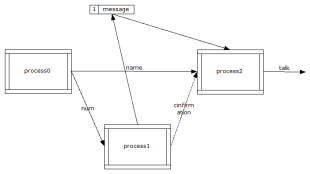
以上の定義を用いて if 文を用いた Java ソースコードから CDFD に変換した場合の事例を表 4 に示す．この時，連続した変数宣言は結合して 1 つのプロセスとして表現し，制御データフローは変数として表されていないため適当な名前を用いている．また，if 文の処理をプロセスに変換する際に記述した制御データフロー *then*，*else* および if 文全体をプロセスに変換する場合に記述した制御データフロー *confirmation* は親プロセスである if 文全体が生成したものとして見なす．そのため条件を表すプロセスは処理を表すプロセスより上の階層に存在するとして考える．なお，本稿における選択構造の表現は本来の SOFL における表現とは異なる．本来の SOFL では選択構造はひし形で分岐を表現している．

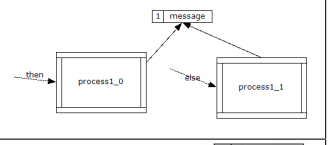
3.3. 反復構造の変換

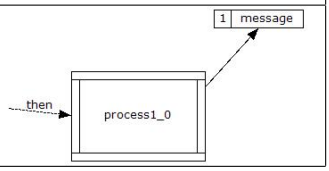
本節では反復構造における CDFD への変換規則の定義について述べる．ここで述べる選択構造は for 文，while 文，do-whle 文についてである．これらのステートメントは for 文なら初期化，条件，処理，更新の 4 種類から構成され，while 文と do-whle 文なら条件と処理の 2 種類から構成される．以上を踏まえて反復構造の変換規則を以下に定義し，その理由を述べる．なお，この規則は選択構造と一部重複する．

表 4. 分岐構造の変換例

ソースコード	CDFD への変換例

`public static String message;`
`public static void function(){`
`  String name="I";`
`  double num=Math.random()*10;`
`  if(num>5){`
`  message="Bye.";`
`  }else{`
`  message="Hello.";`
`  }`
`  String talk=name+" say "+message;`
`}`
	

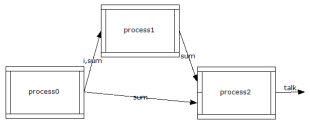
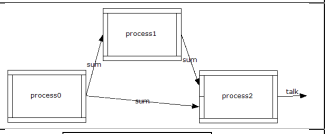
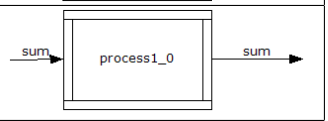
`if(num>5){`
`  message="Bye.";`
`}else{`
`  message="Hello.";`
`}`
	

`if(num>5){`
`  message="Bye.";`
`}`
	

- 子プロセスを持つ親プロセスは子プロセスへ接続データフローも親プロセスに接続しているものとして表現する．理由は選択構造の場合と同様である．

以上の定義を用いて反復構造を用いた Java ソースコードから CDFD に変換した場合の事例を表 5 に示す．こちらも連続した変数宣言は結合して 1 つのプロセスとして表現している．

表 5. 反復構造の変換例

ソースコード	CDFD への変換例
`public static void function() { int sum = 0; int i=0; while(i<=5){ sum ++; i++; } String talk="I have "+sum+" pens"; }`	
`public static void function() { int sum = 0; int i=0; do{ sum ++; i++; }while(i<=5); String talk="I have "+sum+" pens"; }`	
`public static void function() { int sum = 0; for (int i = 1; i <= 5; i++) { sum += i; } String talk="I have "+sum+" pens"; }`	
`do{ sum ++; i++; }while(i<=5);` / `while(i<=5){ sum ++; i++; }`	
`for (int i = 1; i <= 5; i++) { sum += i; }`	

4. 形式仕様への変換

本節ではソースコードから SOFL のプロセス仕様への変換における所見と方針について述べる．3 節で述べた CDFD への変換は自動的に行うことが可能だが，プロセス仕様への変換は全てを自動化することは難しい．機械的に作成した仕様は自然言語のような曖昧性は存在しないが，システムの概念や目的は機械的に判断できないため変換された仕様を開発目的に合致させるのは難しいからである．例えばプロセス仕様内に事前条件を記述する場合，あるプロセスに分岐するための条件を事前条件とするか他の箇所で決めた制限を事前条件とするかを機械に判断させるのは困難である．

以上よりプロセス仕様におけるプロセス名，入力，出力，データストアのような機械的に判断しやすい部分はツールが自動的に生成し，事前条件と事後条件のような人間の思考が必要な部分はユーザ自身が記述するものとする．これによりソースコードの内容を理解しながら記述を行うため，ユーザがバグを発見しやすくなる，あるいはエラーが発生しなくともバグとなるような使用する変数が過不足である場合に気づきやすくなるという利点がある．なお自動生成には 3 節の変換規則の利用が可能だが，データフローやデータストアに相当する変数の型を明確にする必要がある．

5. 支援ツールの紹介

定義した変換規則を用いて CDFD を作成する支援ツールを開発した．本ツールの開発環境は Eclipse Neon 3(4.6.3)[11] を用い，Eclipse プラグイン [12] としてエディタに付随させたビューを拡張する形で実装した．そのためこのツールを利用するためには開発したプラグインを Eclipse にインストールし，Java ファイルを開発したエディタで開けばよい．CDFD の表示には Zest Visualization Toolkit[13] というプラグインを使用した．プログラミング言語は Java を用いた．ソースコードを解析するために Eclipse JDT[14] が提供している Java のソースコードを抽象構文木として保持する AST を，解析結果を記録するためにデータベースとして Java DB[15] を用いた．

ツールの機能は 3 つに分けられる．図 2 に支援ツールの画面を示しつつ説明する．1 つ目は Java エディタ (赤枠) である．これは Eclipse に初めから備わっているものを使用している．2 つ目は CDFD ビュー (黄枠) である．Java ファイルを支援ツールで開くと同時にソースコードの読み込みと変換を行い，画面上に表示する機能である．このツールではプロセスはグレーの長方形で表現し，データフローはグレーの矢印で表現する．データストアはプロセスとの区別のために青色の長方形で表現し，それに対する接続は緑の矢印で表現する．データフロー名は重複を許すこととする．始点もしくは終点が同じ階層に存在しない場合はダミーとして作成したプロセスであるプロセス *No_To* もしくはプロセス *No_From* にデータフローを接続する．プロセス *No_To* はデータフローが存在するがフローの終点が不明な場合の終点として用い，プロセス *No_From* はデータフローが存在す

るがフローの始点が不明な場合の始点として用いる．ただし，終点のプロセスがどの階層にも存在しないデータフローは表現しないものとする．例えばソースコード内で変数を宣言したがその変数が後に使われなかった場合を考える．この時，変数を用いたフローの始点となるプロセスは存在するが，終点となるプロセスは存在しない．用いたプラグインの関係でフローは始点と終点の双方が存在しないと表示できないため，終点のプロセスがどの階層にも存在しないデータフローは表現しないこととする．プロセスの名前はステートメントの記述を用いるものとする．例えば $int\ a = 0;$ というステートメントが存在する場合，「$int\ a = 0;$」という名前のプロセスが存在することになる．ただし，メソッドの仮引数を表現する場合に限り「$main(String[]\ args)$」のように「メソッド名 (仮引数)」をプロセス名として扱う．ソースコード内の記述をプロセスの名前とすることにより，表示されたプロセスがソースコード上のどの部分であるかを直感的に理解しやすくするためである．また，可視性を向上させるために連続した変数宣言のプロセスは結合して 1 つのプロセスとし，if 文のプロセスの子プロセスには $then$ もしくは $else$ をプロセス名の頭に追加する．3 つ目はモジュールの構造ビュー (青枠) である．こちらも Java ファイルを開くと同時にソースコードからモジュールの構造を木構造として変換し，画面上に表示する．この機能は CDFD ビューに表示する階層の指定にも用いる．また，仕様記述エディタとしての機能も備える予定である．この 3 つの機能を用いることにより，CDFD を見ることでシステムの動きの把握を助け，ソースコードを確認することで詳細を理解しつつ，同時に仕様を記述を行うことが実現できる．

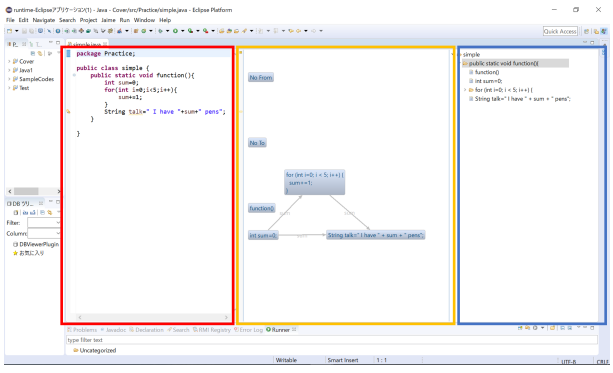


図 2. 支援ツールの画面

5.1. データベースの設計

変換に用いるためのデータを記録する必要がある．そこでリレーショナルデータベースを用いてデータの記録を行うためにデータベースの設計を行った．設計したデータベースの ER 図を図 3 に示す．

プロセステーブルはプロセスを記録するために用いる．バインディングのキーは AST がメソッド名やクラス名ごとに割り振ったキーであり，階層の異なる同名のメソッドやクラスを区別するために用いる．ただし，プロセスがメソッドやクラスではなくステートメントの場合はバインディングのキーは NULL である．また，プロセス名の文字数が型の制限である長さ 255 を超えた場合には一部を切り捨てて記録する．データフローテーブルはデータフローを記録するために用いる．これは同名の変数であっても始点となるプロセスと終点となるプロセスが異なれば別のデータフローとして扱うためである．そのため同じバインディングのキーを持っていても異なるデータフロー ID を持つ場合がある．バインディングのキーは AST が変数名ごとに割り振ったキーであり，階層の異なる同名の変数を区別するために用いる．始点プロセス ID はデータフローの始点がどのプロセスであるかを表す．データストアテーブルはデータストアを記録するために用いる．バインディングのキーは AST はデータフローテーブルと同様である．また，親プロセス ID はデータストアがどのプロセスの下で宣言されているかを表す．

階層関係テーブルはプロセス間の階層関係を記録するために用いる．親プロセス ID は子プロセスの 1 つ上の階層にあたるプロセス ID を表し，子プロセス ID は 1 つ上の階層にあたるプロセスのプロセス ID を表す．親プロセス ID は 1 つ以上の子プロセス ID を持つが，子プロセス ID は 1 つの親プロセス ID を持つ．データストア接続テーブルはプロセスからデータストアへの接続関係を記録するために用いる．データストア ID はプロセスに使用されるデータストアを表し，プロセス ID はデータストアを使用するプロセスを表す．書き込みは値が true であるならデータストアに対する書き込みが発生することを表し，値が false であるならデータストアの値を参照するのみであることを表す．データフロー接続テーブルはデータフローの終点を記録するために用いる．終点プロセス ID はデータフロー ID を持つデータフローの終点がどのプロセスであるかを表す．

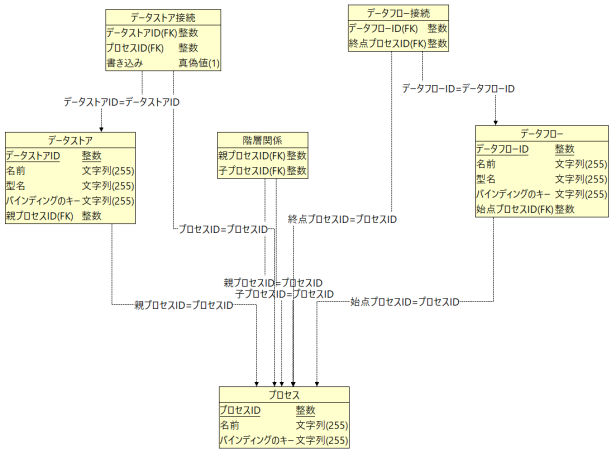


図 3. データベースの ER 図

5.2. クラスの作成

ツールの実装のためにいくつかのクラスを作成した．これらのクラスは主に 4 種類に分類される．すなわち Eclipse の画面に表示を行うクラス (AutoEditor クラス)，データベースに対して操作を行うクラス (DataManagement クラス，UseDB クラス)，SQL 文を生成してデータベースに対して操作を行うクラスに渡すクラス (Ext_DB クラス，Flow_DB クラス，Link_DB クラス，Parent_DB クラス，Process_DB クラス，Store_DB クラス)，ソースコードに対して構文解析を行うクラス (DecomExpression クラス，MyVisitorwithDecom クラス) である．作成したクラスの UML クラス図を図 4 に記す．ただしメソッド等は省略している．

AutoEditor クラスはプラグインを動かす際に最初に動くクラスである．このクラスでは DataManagement クラスのインスタンスを作成して必要なテーブルを作成する，対象となるソースコードを AST で抽象構文木とし，DataManagement クラスのインスタンスと併せて MyVisitorwithDecom クラスに渡す．MyVisitorwithDecom クラスでの解析が終了したら DataManagement クラスに CDFD を生成するように指示を出す．DataManagement クラスと UseDB クラスはデータベースのテーブルに対する操作を行うクラスである．UseDB クラスは MyVisitorwithDecom クラスと DecomExpression クラスから得た情報をテーブルに記録し，DataManagement クラスは CDFD を生成するためにテーブルから情報を得る目的で用いる．また，DataManagement クラスは UseDB クラスを継承している．DecomExpression クラスと MyVisitorwithDecom クラスは抽象構文木を探索するクラスである．CDFD を生成するために必要な情報を DataManagement クラスを用いて記録する．MyVisitorwithDecom クラスはステートメントを探索し，DecomExpression クラスはステートメント内にある式を探索するために MyVisitorwithDecom クラスから呼び出される．Ext_DB クラス，Flow_DB クラス，Link_DB クラス，Parent_DB クラス，Process_DB クラス，Store_DB クラスは DataManagement クラスと UseDB クラスから呼び出され，各テーブルを操作するための SQL 文を文字列として生成して返すクラスである．

6. 事例研究

本節では実際に Java ソースコードから支援ツールを用いて CDFD への変換を行う．テストに使用した Java ソースコードをプログラム 1 に示す．これは *main* メソッド内で台形則を用いた数値積分を計算し，コンソールに表示するプログラムである．フィールド変数 A, B は区間を表し，フィールド変数 N は区間の分割数を表す．ローカル変数 sum, h, t, w はそれぞれ積分の結果，幅，積分点，重みを表す．また，ローカル変数 i は for 文の条件に用いる．

プログラム 1. Java ソースコード

```
1  package Practice;
2
3  public class Trap {
4    public static final double A=0.,B=3.;
5    public static final int N=100;
6    public static void main(String[] args) {
7
8      double sum,h,t,w;
9      int i;
10
11     h=(B-A)/(N-1);
12     sum=0.;
13
14     for(i=1;i<=N;i=i+1){
15       t=A+(i-1)*h;
16       if(i==1||i==N)w=h/2.;else w=h;
17       sum=sum+w*t*t;
18     }
19     System.out.println(sum);
20   }
21 }
```

プログラム 1 の内，main メソッドを順序構造の変換規則で変換した CDFD を図 5 に，14 行目から 18 行目の繰り返し処理を反復構造の変換規則で変換した CDFD

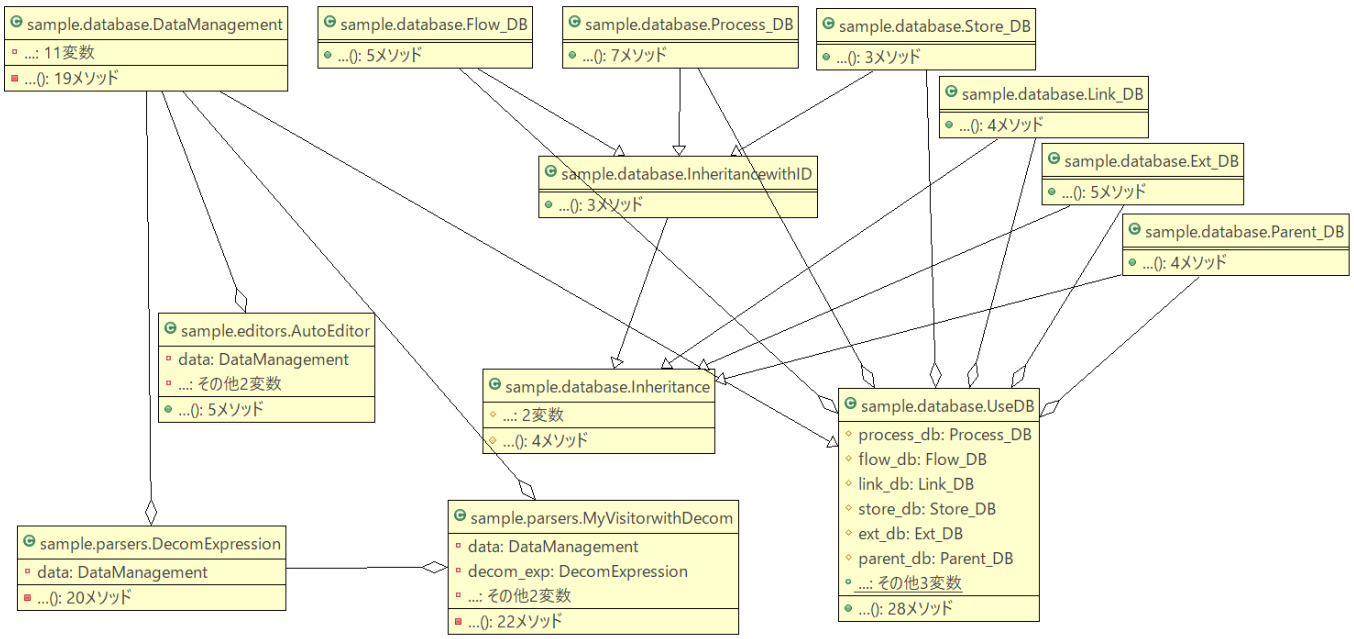


図 4. UML クラス図

を図 6 に，16 行目の条件分岐を選択構造の変換規則で変換した CDFD を図 7 に示す．これらの図からソースコードを分割して階層ごとに見ることが可能であり，プロセス間の依存関係が見て取れる．特に図 5 のプロセス $sum = 0;$ とプロセス $h = (B - A)/(N - 1);$，図 6 のプロセス $t = A + (i - 1) * h;$ と選択構造のプロセス，図 7 のプロセス $then\ w = h/2;$ とプロセス $else\ w = h;$ は互いに関係を持たず，依存関係がないと分かるからである．しかしツールとしての問題点も見受けられる．以下に問題点を記す．

1. データフロー名が見づらい．これは全ての図で見受けられる．これではデータが関係するか否かが明確にならない．解決策としてデータフロー名を設定する箇所でプロセスの始点と終点が同じデータフローがすでに存在する場合はデータフローを追加するのではなくデータフロー名を「a, b, c」のように更新するという方法が挙げられる．

2. 全ての選択を考慮して余分なデータフローを表現している．これは図 5 で見受けられる．今回の例では反復構造が必ず処理を行うためプロセス $sum = 0;$ で生成されたデータフローは反復構造のプロセス以外につながることはないはずである．解決策として反復構造および選択構造の条件を解析を行う際に真偽を計算し，最低 1 回は満たすかによってデータフローの終点を判断するということである．判断の結果をデータベースに記録すれば CDFD を表示する際に余計なデータフローを減らすことができる．ただし 4 節で述べたように静的なコード解析のみでは分岐先の特定がほぼ不可能なため推奨しない．

3. プロセスに CDFD のプロセスとしての要素が欠けている．これは全ての図で見受けられる．特に入力ポートと出力ポートが表現されていないという問題がある．プロセス $System.out.println(sum);$ のように同じ変数を表現するデータフローが複数入力もしくは出力された場合はポートを分割する必要がある．解決策としてまずプロセスを表す図を SOFL におけるプロセスを表すものに差し替える．その上で同じ変数を表すデータフローが複数入力もしくは出力された時にデータベースに記録し，プロセスを表す図に反映させるということが挙げられる．これを CDFD を生成する箇所に追加すれば表現可能である．

4. プロセス $System.out.println(sum);$ においてデータストア out へ書き込みがされている．これは図 5

で見受けられる．変数 *out* がデータストアとなっているため変換規則には反していないが，このプロセスはコンソールへの出力を行うプロセスであり表現としては不自然である．解決策として解析時にデータストアとなりうる変数の型を確認し，PrintStream 型であるならコンソールを表すプロセスを終点として設定する方法が挙げられる．

5. 選択構造や反復構造における条件が表現されず，理解の助けになりにくくなっている．これは図 6 と図 7 で見受けられる．解決策として条件を上のプロセスに含むのではなく条件もプロセスとして扱う，もしくは 4 節で触れた本来の CDFD における条件構造を用いることが挙げられる．条件をプロセスとして扱う場合は一度条件のプロセスをデータフローの終点とした上でそれぞれの分岐先に改めて新しくデータフローを生成するという方法がとれる．CDFD の条件構造を用いる場合は条件構造用のテーブルを新たに用意する必要がある．

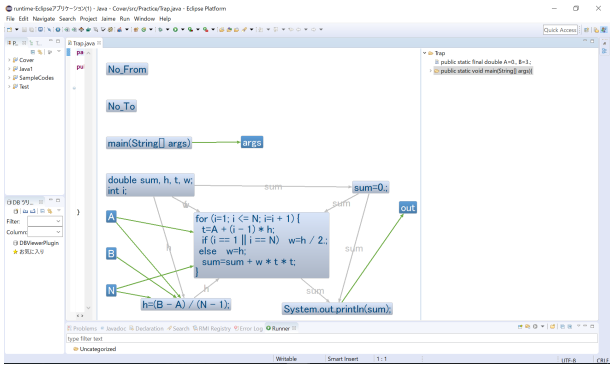


図 5. main メソッド (順序構造) の CDFD

7. 結論

本稿では Java ソースコードから CDFD を生成することで SOFL 形式仕様の記述を支援を行うための変換規則とツールの提案を行い，事例研究によってその有効性を確かめた．その結果，ソースコード中のプロセスにおいて関係性を視覚的に確認することができた．今後の課題は変換規則の追加とツールの完成の 2 つが挙げられる．1 つ目の課題である変換規則の追加について述べる．まず CDFD で表現できる範囲を広げる必要がある．今回の規

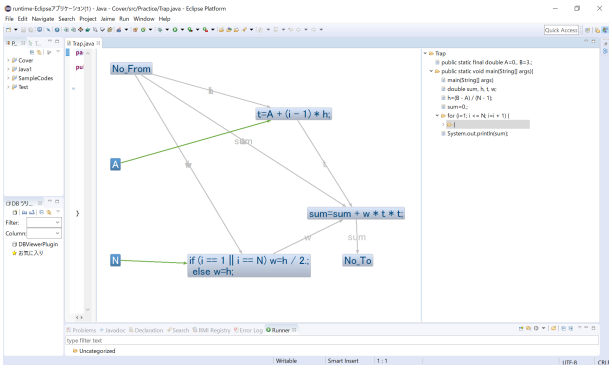


図 6. for 文 (反復構造) の CDFD

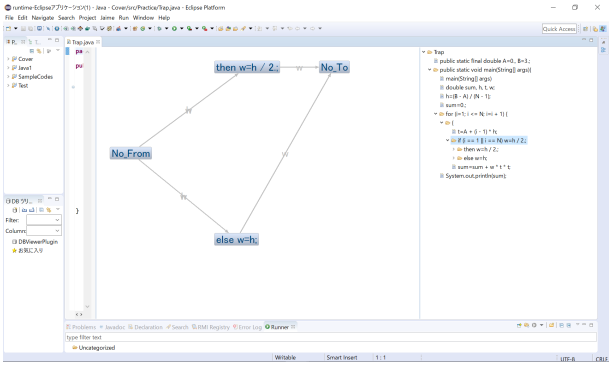


図 7. if 文 (選択構造) の CDFD

則では 1 つのメソッドに対する変換規則のみを定義している．そのためクラス，他メソッドとの関係，継承，カプセル化，コンストラクタといった 1 つのメソッドでは表現しきれない場合に対する変換を定義していない．これは選択構造に対する変換規則へ追加をすべきである．また，private や final といったメソッドや変数につける修飾子を考慮していない．これらはデータフローやデータストアに対する矢印の向きに直接関係する．例えば final は変更不可能な変数として扱うため，データストアとして扱う場合は変数の宣言時以外では書き込みが常に false でなくてはならないという制限を設ける必要がある．これはデータベースのテーブルに該当する列を設ければよい．そしてソースコード中のコメントに対する変換を無視している．これは SOFL 形式仕様の記述の際に自動で変換することの可能な箇所であるためコメントを抽出することで自動的な変換を実現する必要がある．このように，変換規則が不足していることが課題である．2 つ目の課題であるツールの完成について述べる．まず 6 節

で問題点として挙げた入力ポートと出力ポートに加え，データフローを表現する箇所に制御データフローを表現する記述を追加することで制御データフローを実装する必要がある．そして先に述べた変換規則の追加を随時実装していくこととなる．また，テストで使用したソースコードはコンパイルエラーが起きないものであり，コンパイルエラーが発生するようなソースコードに対するテストを行えていない．そのため本稿ではバグの検出に対する明確な有用性を示すことができていない．さらに紹介した支援ツールでは CDFD の表示のみで SOFL 仕様記述を行うための機能が実装されていない．そのため 5 節でも触れたように画面上のモジュールの構造ビューの下部に SOFL 仕様記述を行うためのエディタを設置する必要がある．エディタを設置すれば事前条件と事後条件を適用した場合の事例研究を行うことも可能となる．このように，画面上に SOFL 仕様記述のためのエディタを実装時に正しく CDFD および SOFL 形式仕様が表示されるように改善していくことが課題である．今後は以上の課題を解決した上で複数のメソッドやクラスに関わる事例研究および参考文献として示した先行研究との具体的な比較を行う必要がある．

8. 謝辞

本研究は JSPS 科研費 26240008 の助成を受けたものです．

参考文献

[1] Paolo Benedusi, Aniello Cimitile, and U De Carlini. A reverse engineering methodology to reconstruct hierarchical data flow diagrams for software maintenance, 11 1989.

[2] A.B.O ’ Hare, E.W.Troan. Re-analyzer: From source code to structured analysis. *IBM Systems Journals*, Vol. 33, No. 1, pp. 110–130, 1994.

[3] 井垣宏, 木村隆洋, 中村匡秀, 松本健一. Dfd を利用したプログラム処理間の依存解析によるレガシーソフトウェアからのサービス抽出. ソフトウェアエンジニアリング最前線 ２００９ 情報処理学会ＳＥ, pp. 89–96. 近代科学社, September 2009.

[4] Shaoying Liu. *Formal Engineering for Industrial Software Development*. SpringerVerlag, 2004.

[5] 劉少英. 実用性が高い形式工学手法と支援ツールの研究開発, 2012. https://www.ipa.go.jp/sec/softwareengineering/reports/20130422.html.

[6] 葛山善基, 神代知範. データフロー図の構造化記述法について. 情報処理学会研究報告ソフトウェア工学（SE）, pp. 9–16, sep 1996.

[7] W. Reisig. *Petri Nets: An Introduction*. Monographs in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2012.

[8] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005.

[9] 小田朋宏, 荒木啓二郎. Vdm-sl 仕様からの smalltalk プログラムの自動生成. ソフトウェア・シンポジウム 2016 in 米子 論文集, pp. 1–10, Jun 2016.

[10] 漢明張, 昌満野呂, 篤史沢田, 敦吉田, 吉成蜂巣, 励士横森. アーキテクチャ指向開発における形式手法の適用に関する考察 (ディペンダブルコンピューティング組込み技術とネットワークに関するワークショップ etnet2013). 電子情報通信学会技術研究報告 : 信学技報, Vol. 112, No. 482, pp. 61–66, mar 2013.

[11] Eclipse. https://www.eclipse.org/.

[12] 大城正典, 永井保夫. Eclipse 視覚化プラグインによる総合的なプログラミング教育支援システム. 情報教育シンポジウム 2015 論文集, 第 2015 巻, pp. 23–30, aug 2015.

[13] Zest visualization tool kit. http://www.eclipse.org/gef/zest/.

[14] Eclipse jdt project. https://projects.eclipse.org/projects/eclipse.jdt.

[15] Java db. http://download.oracle.com/javadb/index.html.