JAIST Repository
https://dspace.jaist.ac.jp/

Title	UML図面群の変更波及解析に利用可能な依存関係の自動生成法
Author(s)	小谷，正行；落水，浩一郎
Citation	Research report (School of Information Science, Japan Advanced Institute of Science and Technology), IS-RR-2006-002: 1-20
Issue Date	2006-02-07
Type	Technical Report
Text version	publisher
URL	http://hdl.handle.net/10119/8438
Rights	
Description	リサーチレポート（北陸先端科学技術大学院大学情報科学研究科）

JAIST
JAPAN ADVANCED INSTITUTE OF SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

UML図面群の変更波及解析に利用可能な依存関係の自動生成法

小谷 正行, 落水 浩一郎

2006年2月7日
IS-RR-2006-002

北陸先端科学技術大学院大学
情報科学研究科
923-1292 石川県能美市旭台1—1
{m-kotani, ochimizu}@jaist.ac.jp

UML図面群の変更波及解析に利用可能な依存関係の自動生成法

小谷 正行　落水 浩一郎

本論文では、UML1.5 版で定義されている依存関係のプリミティブなセマンティクスを検討することにより、変更波及解析に有用な依存関係 (情報共有、詳細化、作成順序、生存従属、コピーの 5 つ) を新たに定義する。また、そのような依存関係をメタモデルを利用して自動生成する、Model-based Translation 手法を提案する。メタモデルは、メタ関係とメタ要素からなり、メタ関係は上記 5 つの新しい依存関係である。メタ関係の両端にくるソースとターゲットの型を検討することにより、メタ要素を定義する。生成法の概要は以下の通りである。照合規則によって、比較対象となる 2 つの UML モデル要素を抽出し、抽出された UML モデル要素がどのメタ要素に属するかを判定する。特定されたメタ要素間に定義されているメタ関係のインスタンスにより、入力された UML モデル要素間に適切な依存関係を生成する。40 枚の図面からなる COMET 法によるエレベータ制御システムの事例研究を題材として、本論文で提案した新しい依存関係と変換手法の有用性を評価する。
キーワード Unified Modeling Language(UML)、依存関係、変更波及、メタモデル、Model-based Translation 法.

In this paper, we define a set of new dependency relationships by analyzing the primitive semantics of thirteen stereotypes related to dependency relationships of UML 1.5. Those are information sharing, refinement, order of production, existence dependency, and copy. We propose a model-based translation method that generates a new dependency relationship between input UML modeling elements by using a meta model. The meta model consists of meta relationships and meta elements. The meta relationships are the new relationships described above. The meta elements are defined by examining the types of source and target located at both ends of a meta relationship respectively. The outline of the translation algorithm is as follows: (1) extract two UML modeling elements to be compared by matching rule; (2) decide the meta element that the extracted UML modeling element belongs for each UML modeling element; (3) generate a proper dependency relationship between input UML modeling elements by using the instance of the meta relationship defined between two decided meta elements. We evaluate effectiveness of new dependency relationships and generating method proposed in this paper by examining the forty diagrams produced through a case study of elevator control system development by COMET method.
Keyword Unified Modeling Language(UML), dependency relationships, change impact analysis, meta model, model-based translation.

1 はじめに

ソフトウェア開発では、開発者は既存の成果物を参照しながら新たな成果物を作成するため、成果物間にさまざまな依存関係が発生する。このような依存関係を開発者自身が手動で付加することは、修正のコストがかかりまた誤りやすい。

本論文では、UML1.5 版で定義されている依存関係のプリミティブなセマンティクスを検討することにより、変更波及解析に有用な依存関係を新たに定義する。また、そのような依存関係をメタモデルを利用して、自動生成する手法を提案する。

UML1.5 版[17] では、ステレオタイプを利用して 13 種類の依存関係が定義されている。ステレオタイプ付きの各依存関係の意味は、開発者が遭遇する状況

Automatic generation method of dependency relationship among UML diagrams for impact analysis

Masayuki KOTANI, 北陸先端科学技術大学院大学情報科学研究科, School of Information Science, Japan Advanced Institute of Science and Technology.
Koichiro OCHIMIZU, 北陸先端科学技術大学院大学情報科学研究科, School of Information Science, Japan Advanced Institute of Science and Technology.

(例えば詳細化や仕様の実現など)、UML モデル要素間の構造的関係、名前空間の公開や可視性のスコープなどに基づいて定義されている。このような定義は開発者にとっては容易に理解できる意味を持つが、機械的に付加できるように定義されてないため、また、意味にあいまいな部分があり重複する部分もあるので、自動生成に直接利用することは困難である。我々は 2 節において、UML1.5 版における依存関係のプリミティブなセマンティクスを検討する。つぎに、3 節で、定義したセマンティクスをもとに変更波及の型を定義し、メタモデルにおけるメタ関係として形式化する。さらに、5 節で、メタ関係の両端にくる要素の型をメタ要素として定義する。それらを基にメタモデルを定義する (6 節)。

メタモデルに基づく新たな依存関係を UML モデル要素間に自動生成するため、Model-based Translation 手法[4] を用いる (図 1)。

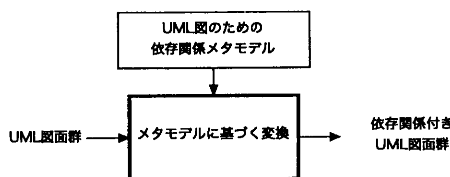

図 1 Model-based Translation 手法の概要

Model-based Translation 法の概要を述べる。自動生成の中核をなすものは、メタモデルである。

メタモデルは、照合規則、メタ規則およびメタ関係からなる。メタ関係は、開発者が UML モデル要素間に依存関係を設定するときに考えているプリミティブなセマンティクスを表現しており、2 つのメタ要素を関連付ける。メタ関係の両端に結合されるメタ要素は、対応するセマンティクスで結合されるべき UML モデル要素の抽象 (分類) である。

依存関係は以下のように生成される (図 2)。

UML 図面群中の 2 つのモデル要素を入力とする。まず、比較対象となる 2 つの入力モデル要素を、照合規則によって決定する。つぎに、それぞれのモデル要素がどのメタ要素に属するかを判定する。モデル要素

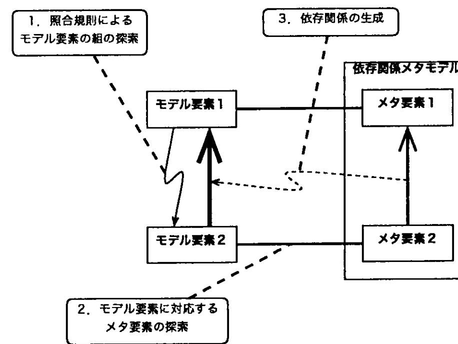

図 2 依存関係の自動生成法の概要

に対応するメタ要素間に、あらかじめ張られているメタ関係を入力モデル要素間の依存関係として採用する。出力は依存関係付き UML 図面群、すなわち、UML モデル要素間に上記の手順で設定されたメタ関係のインスタンスを設定したものである。

次節以降は、以下のように構成される。2 節では UML1.5 版で定義された依存関係のプリミティブなセマンティクス (変更波及の型) に関する考察結果を示す。3 節ではメタ関係を定義する。4 節では、本論文で自動生成の対象とする依存関係とその理由を述べる。5 節ではメタ要素を定義する。6 節ではメタモデルの設計結果を示す。7 節では、変換法を示す。8 節では、簡単な例を用いて本手法の有効性を示し、9 節では関連研究を述べ、10 節で本論文をまとめる。

2 依存関係のセマンティクスに関する考察

依存関係は図 3 のように表現され、「ソースがターゲットに依存する」と読み、「ターゲットが変更されるとソースも変更される場合がある」という意味を持つ。

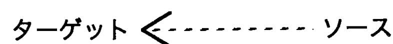

図 3 依存関係の表記

UML1.5 版においては、bind, derive, realize, refine, trace, use, call, create, instantiate, send, access, friend, import の依存関係のステレオタイプが

定義されている。それらは、UML メタモデルにおいて、Dependency のサブクラスとして Binding, Abstraction, Usage, Permission が分類されている (図 4)。以下、各ステレオタイプごとに UML1.5 版における定義、依存関係のソースとターゲット、依存の特徴 (ターゲットが持つ情報を変更したときに、ソースへ与える影響)、プリミティブなセマンティクス (変更波及の型) について検討する。

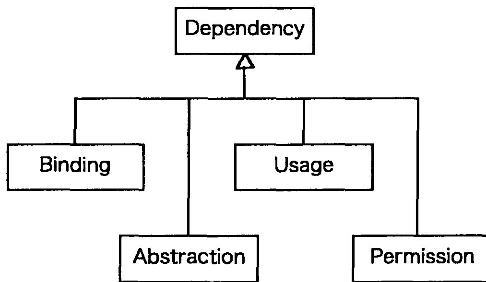

図 4 UML メタモデルにおける Dependency のサブクラス

2.1 Binding

Binding はテンプレートとそれから生成されたモデル要素間の関係を示す。これに対応する依存関係のステレオタイプは bind である。

2.1.1 bind

(UML1.5 版における定義) UML1.5 版において、bind は「テンプレートクラスのテンプレートパラメータに実際の値を束縛して、パラメータ無しの要素を作成する」と定義される。図 5[16] に bind を利用した例を示す。

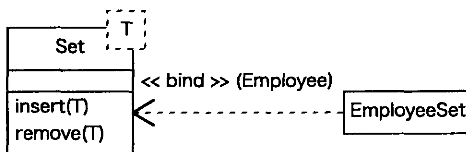

図 5 bind を利用した例

図 5 において、Set はテンプレートクラスであり、テンプレートパラメータ T を持つ。パラメータを持たない EmployeeSet クラスは、T に Employee を束縛して作成される。

(依存関係のソースとターゲット) テンプレートクラス EmployeeSet は依存関係のソースであり、クラス Set は依存関係のターゲットである。

(依存の特徴) ソースは、ターゲットが持つ情報（テンプレート）と、新たに付加される情報（パラメータの値）から構成されるので、ターゲットが持つ情報のどれかを変更したとき、または、付加される情報を変更したとき、ソースを見直す必要がある。

(プリミティブなセマンティクス) 2 つの変更波及の型がある。1 つは、ソースとターゲットは、ターゲットが持つ情報を共有し、ターゲット側の共有情報の変更はソースに波及する。また、ターゲットを変更せず、bind されるパラメータの値を変更したときもソースに波及する。この変更波及の型を「ターゲットが持つ情報の共有」と呼ぶ。この変更波及の型は、どのような情報が変更に関与するかを示すパラメータを持つ。「ターゲットが持つ情報の共有 (ターゲットが持つすべての情報、新たに付加される情報)」と記す。もう 1 つは、ソースの存在がターゲットの存在に依存することである。この変更波及の型を「生存従属」と呼ぶ。

2.2 Abstraction

Abstraction は異なる視点、または、異なる抽象レベルで同じコンセプトを表現する 2 つの要素、または、要素の集合間の関係を示す依存関係である。†1

2.2.1 derive

(UML1.5 版における定義) UML1.5 版において、derive は「ある要素は他の要素から計算できる」と定義される。図 6[12] に例を示す。図 6 において、クラス「銀行口座」のオブジェクト

†1 UML1.5 版におけるステレオタイプには、realize は定義されていないが、UML メタモデルには含まれているので、realize, derive, refine, trace の 4 つとも検討する。

とクラス「取引」のオブジェクト間のリンクと、クラス「取引」のオブジェクトとクラス「取引額」のオブジェクト間のリンクから、「銀行口座」のオブジェクトと「取引額」のオブジェクト間のリンクを導出できる。

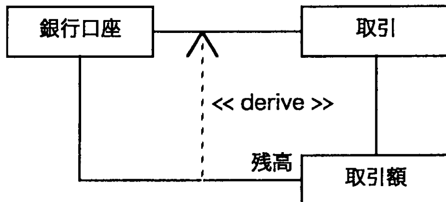

図 6　derive を利用した例

(依存関係のソースとターゲット) 計算された要素 (銀行口座と取引間の関連) が依存関係のソースであり、計算に使われた要素 (銀行口座と取引間の関連) が依存関係のターゲットである。

(依存の特徴) この依存関係の特徴は、ターゲットが持つ情報を利用してソースが計算されることである。ターゲットである関連の値 (リンク) が変化すれば、ソースの関連の値を計算し直さなければならない。また、ターゲットである関連が存在しなくなれば、ソースの関連は存在しえない。

(プリミティブなセマンティクス) 2 つの変更波及の型がある。1 つは、「ターゲットが持つ情報の共有 (計算に利用される値)」である。もう 1 つは、「生存従属」である。

2.2.2 realize

(UML1.5 版における定義) UML1.5 版において、realize は「仕様とその実現の関係」と定義される†2。図面上では、白塗りの三角形を一端に持つ破線として表現される。realize には 2 つの用途がある。1 つは、インタフェース (操作のシグネチャ) と、それを実現するクラスまたはコンポーネント間の関係を示すために利用する。もう 1 つは、ユースケースとそれを実現するコラボレーションの関係を示すために利用する。図 7 [5] に realize を利用した前者に対する例を示す。図 7 において、インタフェース IRuleAgent は 3 つの操作のシグネチャを定義しており、それを実現したクラスが AccountBusinessRules である。

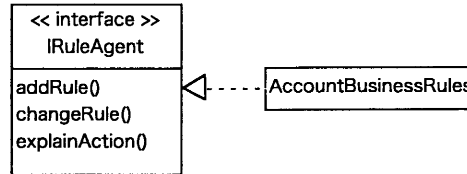

図 7　realize を利用した例

(依存関係のソースとターゲット) 実現 (how to) は依存関係のソースであり、仕様 (what) は依存関係のターゲットである。具体的には、以下の 2 つの例がある。(1) インタフェースの定義が依存関係のターゲットであり、それを実現したクラスは依存関係のソースである。(2) また、ユースケースは依存関係のソースであり、それを実現するコラボレーションは依存関係のターゲットである。

(依存の特徴) ターゲットが定義する仕様をソースは共有する。具体的には、インタフェースとその実現の場合、ソースはターゲットが持つ操作のシグネチャを共有する。ユースケースとコラボレーションの場合は、ユースケースで定義されたイベントフローを満たす機構がコラボレーションに実現されており、イベントフローが変化すれば機構も変化する。

(プリミティブなセマンティクス) 変更波及の型は、「ターゲットが持つ情報の共有 (仕様)」である。

2.2.3 refine

(UML1.5 版における定義) UML1.5 版において、refine は「異なる抽象レベル (例えば、分析と設計) におけるモデル要素間に対応がある。完全である必要はないが対応付けルールを持つ。2 つの要素間には歴史的または (対応付けルール による) 計算可能な関係がある」と定義され

†2 著者の見解によれば、what と how to の関係を表現する

る。対応付けルールとしては、詳細化ルール、変換ルール、最適化ルールなどがある。図 8[9] に refine を利用した例を示す。分析モデルにおけるクラス Chessboard の表現は、並列処理のための最適化の観点から詳細化され、設計モデルの Chessboard の表現となる。

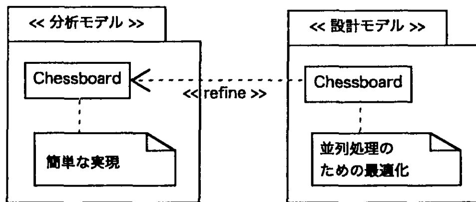

図 8 refine を利用した例

(依存関係のソースとターゲット) 詳細化後 (または変換後、最適化後) の抽象レベルが依存関係のソースであり、詳細化以前 (または変換前、最適化前) の抽象レベルが依存関係のターゲットである。

(依存の特徴) 対応付けルールの入力、または、対応付けルールそのものを変更すれば、出力が影響をうける。

(プリミティブなセマンティクス) 詳細化、変換、最適化などの対応付けがあるが、詳細化という表現で代表させる。変更波及の型は、「詳細化 (対応付けルールと、入力)」である。

作成順序 (開発履歴) の情報の必要性 歴史的な関係がある場合には、作成順序に関する情報を必要とする。

2.2.4 trace

(UML1.5 版における定義) UML1.5 版において、trace は「開発履歴、または、同じコンセプトを異なる意味レベルで表現する、2 つの要素間の時間的な順序関係 (**対応付けルールは必要ない**) を表現する」と定義される。このような時間的な順序関係で結合されるものとしては、同じ概念に対する抽象度の異なる表現や異なる版間の関係がある。

抽象レベルの異なる物を trace 依存関係で結合した例を図 9[8] の上段に示す。この例は、分析モデルの現金を引き出す機能を実現するためのコラボレーションと、設計モデルの同機能のコラボレーション間の異なる抽象概念間の trace 依存関係を示す。

また、歴史的な流れを示す trace 依存関係の例を図 9[8] の下段に示す。これは販売管理パッケージの 7.1 版と 7.2 版間の trace 依存関係を示している。

抽象レベルの異なる物を trace 依存関係で結合するとき、ソースとなる抽象レベルの低いモデルには詳細な抽象概念をおき、ターゲットとなる抽象レベルの高いモデルには単純な抽象概念をおく。trace 依存関係でつなげた抽象概念や版をさかのぼることで、ソースより以前の抽象概念や版を取り出すことができるため、ターゲットはソースをさかのぼった祖先でもある。このため、同じ抽象概念や版を示すような同じモデル内の要素間には、trace 依存関係を適用しない。

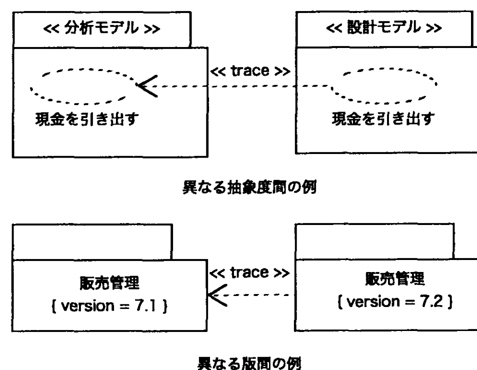

図 9 trace を利用した例

(依存関係のソースとターゲット) 時間的に後にできたもの (子孫) が依存関係のソースであり、時間的に先にできたもの (祖先) が依存関係のターゲットである。

(依存の特徴) 依存関係の特徴は trace 依存関係を設定する人の持つ分析・設計に関するセマンティクスに依存する。trace 依存関係におけるター

ゲットの変更はソースに波及する場合が多い。

(プリミティブなセマンティクス) 変更波及の型は、「時間的な作成順序」である。

作成順序 (開発履歴) の情報の必要性 上記の理由で、作成順序に関する情報を必ず必要とする。

2.3 Usage

Usage は、完全な実装や操作のために、ある要素が他の要素を必要とすることを示す。これに対応する依存関係のステレオタイプは use, call, create, instantiate, send である。

2.3.1 use

(UML1.5 版における定義) UML1.5 版において、use は「(正しい実装や機能定義のために) ある要素は別の要素の存在を要求する」と定義される。図 10[12] に use を利用した例を示す。図 10 において、クラス Client は操作の引数の型や戻り値の型として、クラス Supplier を利用する。

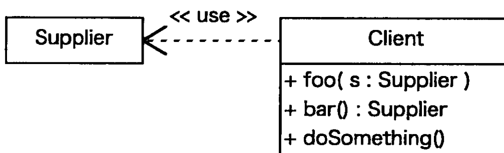

図 10 use を利用した例

(依存関係のソースとターゲット) 必要とするもの (クラス Client) が依存関係のソースであり、必要とされるもの (クラス Supplier) が依存関係のターゲットである。

(依存の特徴) この依存関係の特徴は、ターゲットが持つ情報 (インタフェース) をソースは利用する。ターゲットが持つ情報を変更したとき、ソースを見直す必要がある。

(プリミティブなセマンティクス) 変更波及の型は、「ターゲットが持つ情報の共有 (ターゲットのインタフェース)」である。

2.3.2 call

(UML1.5 版における定義) UML1.5 版において、call は「ある操作が、別の操作を呼び出す」と定義される。図面上では、下記の例に示すように、各操作を持つクラス間の関係として表現されることもある [9]。図 11[17] に call を利用した例を示す。図 11 において、クラス ShoppingCartImpl のある操作が、インタフェース Context のある操作を呼び出す。

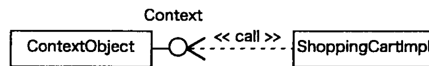

図 11 call を利用した例

(依存関係のソースとターゲット) 呼び出す方 (クラス ShoppingCartImpl) が依存関係のソースであり、呼び出される方 (インタフェース Context) が依存関係のターゲットである。

(依存の特徴) この依存関係の特徴は、ターゲットが持つ情報 (インタフェース) をソースは利用することである。ターゲットが持つ情報を変更したとき、ソースを見直す必要がある。

(プリミティブなセマンティクス) 変更波及の型は、「ターゲットが持つ情報の共有 (ターゲットのインタフェース)」である。

2.3.3 create

(UML1.5 版における定義) UML1.5 版において、create は「ある要素が、別の要素のインスタンスを生成する」と定義される。図 12 に create を利用した例を示す。図 12 において、クラス AccountHome によってクラス Account は生成される。

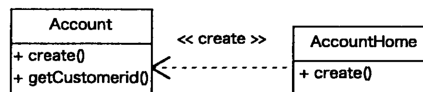

図 12 create を利用した例

(依存関係のソースとターゲット) 生成するもの (クラス AccountHome) が依存関係のソースであり、生成されるもの (クラス Account) が依存関係のターゲットである。

(依存の特徴) この依存関係の特徴は、以下の2つである。ソースはターゲットの生成タイミングを知っている。ターゲットの生成法はターゲット自身に定義されている。ターゲット生成にあたってのパラメータが変化したとき、ソースを見直す必要がある。ターゲットの生成タイミングはソース固有の情報であり、依存には無関係である。

(プリミティブなセマンティクス) 変更波及の型は、「ターゲットが持つ情報の共有 (ターゲットの生成法)」である。

2.3.4 instantiate

(UML1.5 版における定義) UML1.5 版において、instantiate は「ある要素の操作が、別の要素のインスタンスを生成する」と定義される。図 13[14] に instantiate を利用した例を示す。図 13 において、クラス AccountHome の操作によってクラス Account は生成される。

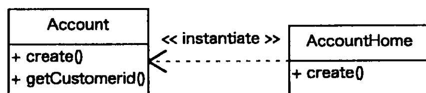

図 13 instantiate を利用した例

(依存関係のソースとターゲット) 生成するもの (クラス AccountHome) が依存関係のソースであり、生成されるもの (クラス Account) が依存関係のターゲットである。

(依存の特徴) この依存関係の特徴は、以下の2つである。ソースはターゲットの生成タイミングを知っている。ターゲットの生成法はターゲット自身に定義されている。ターゲット生成にあたってのパラメータが変化したとき、ソースを見直す必要がある。ターゲットの生成タイミングはソース固有の情報であり、依存に無関係である。

(プリミティブなセマンティクス) 変更波及の型は、「ターゲットが持つ情報の共有 (ターゲットの生成法)」である。

2.3.5 send

(UML1.5 版における定義) UML1.5 版において、send は「操作はシグナルを送り出す」と定義される。ただし、シグナルとは非同期イベントを示す。図 14[5] に send を利用した例を示す。図 14 において、クラス MovementAgent がシグナル Collision を送り出す。

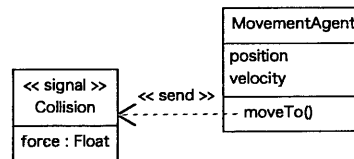

図 14 send を利用した例

(依存関係のソースとターゲット) 送出するもの (クラス MovementAgent) が依存関係のソースであり、送出されるもの (シグナル Collision) が依存関係のターゲットである。

(依存の特徴) この依存関係の特徴は、以下の2つである。ソースはターゲットの生成タイミングを知っている。ターゲットの生成法はターゲット自身に定義されている。ターゲット生成にあたって、パラメータが変化したとき、ソースを見直す必要がある。ターゲットの生成タイミングはソース固有の情報であり、依存に無関係である。

(プリミティブなセマンティクス) 変更波及の型は、「ターゲットが持つ情報の共有 (ターゲットの生成法)」である。

2.4 Permission

Permission は、モデル要素 (例えばクラスやパッケージ) が他の名前空間の要素にアクセス可能であることを認める。これに対応する依存関係のステレオタイプは access, friend, import である。

2.4.1 access

(UML1.5 版における定義) UML1.5 版において、access は「あるパッケージは、別のパッケージの名前空間の公開要素にアクセスする」と定義される。図 15[17] に access を利用した例を示す。図 15 において、パッケージ Controller はパッケージ GraphicsCore にアクセスする。

(依存関係のソースとターゲット) アクセスする方

図 15 access を利用した例

(パッケージ Controller) が依存関係のソースであり、アクセスされる方 (パッケージ GraphicsCore) が依存関係のターゲットである。

(依存の特徴) この依存関係の特徴は、ソースがターゲットの公開要素を知っていることである。ターゲットの公開要素を変更したとき、ソースを見直す必要がある。

(プリミティブなセマンティクス) 変更波及の型は、「ターゲットが持つ情報の共有 (ターゲットの公開要素)」である。

2.4.2 friend

(UML1.5 版における定義) UML1.5 版において、friend は「ターゲット要素に宣言された可視性にかかわらず、ターゲット要素がソース要素に特別な可視性を与える」と定義される。図 16[5] に friend を利用した例を示す。図 16 において、クラス CourseSchedule に宣言された可視性にかかわらず、クラス Iterator はクラス CourseSchedule にアクセスできる。

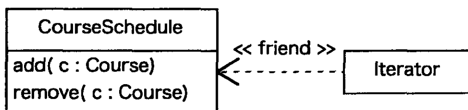

図 16 friend を利用した例

(依存関係のソースとターゲット) 参照するもの (クラス Iterator) が依存関係のソースであり、参照されるもの (クラス CourseSchedule) が依存関係のターゲットである。

(依存の特徴) この依存関係の特徴は、ソースはターゲットが持つすべての名前を参照できる。ターゲットが持つ名前を変更したとき、ソースを見直す必要がある。

(プリミティブなセマンティクス) 変更波及の型は、「ターゲットが持つ情報の共有 (ターゲットが持つすべての名前)」である。

2.4.3 import

(UML1.5 版における定義) UML1.5 版において、import は「あるパッケージが所有する公開要素への参照を別のパッケージに許可し、参照される公開要素の名前をその要素を参照する別のパッケージに加える」と定義される。図 17[5] に import を利用した例を示す。図 17 において、パッケージ Policies はその名前空間をパッケージ Client に共有させる。

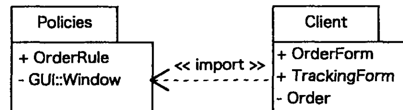

図 17 import を利用した例

(依存関係のソースとターゲット) 名前空間を取り込むもの (パッケージ Client) が依存関係のソースであり、名前空間を提供するもの (パッケージ Policies) が依存関係のターゲットである。

(依存の特徴) ソースの名前空間はソース自身の名前空間とターゲットが公開する名前空間を合わせたものである。ターゲットが持つ名前空間を変更したとき、ソースを見直す必要がある。

(プリミティブなセマンティクス) 変更波及の型は、「ターゲットが持つ情報の共有 (ターゲットが持つ名前空間)」である。

以上の分析結果を表 1 にまとめる。

3 メタ関係の定義

表 1 での分析結果をまとめると、変更波及に関わるプリミティブなセマンティクス (変更波及の型) は以下の 4 つである。

- ターゲットが持つ情報の共有 (パラメータ群)。以後、単に情報共有と記す。
- 詳細化 (対応付けルール、入力)
- 時間的な作成順序。以後、単に作成順序と記す。
- 生存従属

表 1 UML1.5 版における依存関係のステレオタイプの解析

	ステレオタイプ	UML1.5版における定義(文献[17])	依存関係のソースとターゲット	依存の特徴 (ターゲットが持つ情報を変更したときのソースへ与える影響)	プリミティブなセマンティクス	作成順序（開発履歴）の情報を必要とする物
(1)	bind	テンプレートクラスのテンプレートパラメータに実際の値を束縛して、パラメータ無しの要素を作成する	パラメータ無しの要素（ソース）が、テンプレートクラスのパラメータ（ターゲット）に依存する	ソースは、ターゲットが持つ情報（テンプレート）と、新たに付加される情報（パラメータの値）から構成されるので、ターゲットが持つ情報のどれかを変更したとき、または、付加される情報を変更したとき、ソースを見直す必要がある。また、ターゲットがなくなれば、ソースは存在しえない。	ターゲットが持つ情報の共有(ターゲットが持つすべての情報、新たに付加される情報)、生存従属	-
(2)	derive	ある要素は、他の要素から計算できる	計算された要素（ソース）は、計算に使われた要素（ターゲット）に依存する	ソースは、ターゲットが持つ情報を引数にして計算されるので、ターゲットの持つ情報を変更したとき、ソースは計算し直さなければならない。また、ターゲットがなくなれば、ソースは存在しえない。	ターゲットが持つ情報の共有(計算に利用される値)、生存従属	-
(3)	realize	仕様とその実現の関係	実現（ソース）は、仕様（ターゲット）に依存する	ターゲットが定義する仕様をソースは共有する。仕様を変更した場合、実現を見直す必要がある。	ターゲットが持つ情報の共有(仕様)	-
(4)	refine	異なる意味レベル（例えば、分析と設計）におけるモデル要素間に対応がある。完全である必要はないが対応付けルールを持つ。2つの要素間には歴史的または（対応付けルールによる）計算可能な関係がある。	詳細化後(または変換後、最適化後)の抽象レベルが依存関係(ソース)は、詳細化以前(または変換前、最適化前)の抽象レベル(ターゲット)に依存する	対応付けルール、または、その入力を変更すれば、出力が影響をうける。	詳細化(対応付けルールと、入力)	必要とする
(5)	trace	開発履歴、または、同じコンセプトを異なる意味レベルで表現する2つの要素間の関係（対応付けルールは必要ない）を表現する。	時間的に後にできたもの(ソース)は、時間的に先にできたもの(ターゲット)に依存する	依存関係の特徴はtrace依存関係を設定する人の持つセマンティクスに依存する。trace依存関係におけるターゲットの変更はソースに波及する場合が多い。	時間的な作成順序	必要とする
(6)	use	(正しい実装や機能定義のために、)要素Aが要素Bの存在を要求する	要素A（ソース）のセマンティクスが、要素B（ターゲット）のパブリック部分のセマンティクス（インタフェース）に依存する	ターゲットが持つ情報(インタフェース)を、ソースは利用するので、ターゲットが持つ情報(インタフェース)を変更したとき、ソースを見直す必要がある。	ターゲットが持つ情報の共有(ターゲットのインタフェース)	-
(7)	call	ある操作が、別の操作を呼び出す	呼ぶ操作（ソース）が、呼ばれる操作のインタフェース（ターゲット）に依存する	ターゲットが持つ情報(インタフェース)を、ソースは利用するので、ターゲットが持つ情報(インタフェース)を変更したとき、ソースを見直す必要がある。	ターゲットが持つ情報の共有(ターゲットのインタフェース)	-
(8)	create	ある要素が別の要素のインスタンスを生成する	インスタンスを生成される要素（ソース）は、生成する要素（ターゲット）に依存する	ソースはターゲットの生成タイミングを知っている。ターゲットの生成法はターゲット自身に定義されている。ターゲット生成にあたってのパラメータが変化したとき、ソースを見直す必要がある。ターゲットの生成タイミングはソース固有の情報であり、依存に無関係である。	ターゲットが持つ情報の共有(ターゲットの生成法)	-
(9)	instantiate	ある要素の操作が、別の要素のインスタンスを生成する	インスタンスを生成される要素（ソース）は、生成する要素の操作（ターゲット）に依存する	ソースはターゲットの生成タイミングを知っている。ターゲットの生成法はターゲット自身に定義されている。ターゲット生成にあたってのパラメータが変化したとき、ソースを見直す必要がある。ターゲットの生成タイミングはソース固有の情報であり、依存に無関係である。	ターゲットが持つ情報の共有(ターゲットの生成法)	-
(10)	send	操作は、シグナルを送りだす	操作（ソース）は、シグナル（ターゲット）の存在に依存する	ソースはターゲットの生成タイミングを知っている。ターゲットの生成法はターゲット自身に定義されている。ターゲット生成にあたってのパラメータが変化したとき、ソースを見直す必要がある。ターゲットの生成タイミングはソース固有の情報であり、依存に無関係である。	ターゲットが持つ情報の共有(ターゲットの生成法)	-
(11)	access	パッケージAは、パッケージBの名前空間の公開要素にアクセスする	パッケージA（ソース）は、パッケージB（ターゲット）の名前空間の公開要素（インタフェース）に依存する	ソースはターゲットの公開要素を知っているので、ターゲットの公開要素を変更したとき、ソースを見直す必要がある。	ターゲットが持つ情報の共有(ターゲットの公開要素)	-
(12)	friend	要素Bは、要素Bに明示された権限にかかわらず、要素Aに特別な可視性を与える	要素A（ソース）が要素B（ターゲット）に依存する	ソースはターゲットが持つすべての名前を参照できるので、ターゲットが持つ名前を変更したとき、ソースを見直す必要がある。	ターゲットが持つ情報の共有(ターゲットが持つすべての名前)	-
(13)	import	パッケージBが所有する公開要素への参照をパッケージAに許可し、参照される公開要素の名前をその要素を参照するパッケージAに加える	パッケージA（ソース）は、パッケージB（ターゲット）に依存する。	ソースの名前空間はソース自身の名前空間とターゲットが公開する名前空間を合わせたものなので、ターゲットが持つ名前空間を変更したとき、ソースを見直す必要がある。	ターゲットが持つ情報の共有(ターゲットが持つ名前空間)	-

上記の変更波及の型に加えて、両端の要素がまったく同じであることを示す「コピー」という変更波及の型を付け加える。これは、ステレオタイプから導入されるものではないが、図面の大きさの制限等により、同じ内容がコピーされることがよくあるからである。

この 5 つの変更波及の型を、メタモデルにおけるメタ関係として採用する。メタ関係の図式表現を表 2 に示す。

表 2 メタ関係の定義

メタ関係	表記
情報共有	ターゲット (共有情報) ソース
詳細化	ターゲット ソース
作成順序	ターゲット ソース
生存従属	ターゲット ソース
コピー	ターゲット ソース

4 本論文で対象とする依存関係

表 1 の 13 個の依存関係を、「開発者が同じ図面上で明示的に記述することにより、CASE ツールの XML データベースから容易に解析できる自明な依存関係」と、「そうでないもの」に分類する。前者については下記に述べるように、依存関係を生成することが容易であるので、本論文の考察対象からはずす†3。すなわち、

- Binding に対応する依存関係 bind は、テンプレートクラスにパラメータ無しのクラスを、どのような値を入れて作られたか明示的にするための依存関係であり、1 枚の図面上に表現される。
- Usage と Access に対応する依存関係 use, call, create, instantiate, send, access, friend, import は、開発者が明示的に関係を定義する依存であり、1 枚の図面上で表現される。
- Abstraction に対応する依存関係 derive は、新しい関係や属性の導出理由を明示的にするための依存関係であり、1 枚の図面上で表現される。

本論文で考察対象とする依存関係は、以下の通りである。

- 依存関係 realize は、異なる図面のモデル要素間で発生する場合がある。
- 依存関係 refine や trace は、単純な表現を用いた要素から詳細な表現を用いた要素への依存関係であり、抽象度が異なる 2 つの図面にあるモデル要素間に発生する場合が多い。

表 3 に、対象とする依存関係 realize, refine, trace のソースとターゲットの型を再掲する。

表 3 に示す依存関係のソースとターゲットは、一般には特定できない。そこで、Unified Process において、上記依存関係が発生する場合を考察する。

図 18 [8] に、Unified Process における、UML モデル要素間の依存関係の一分析例を示す。図 18 において、長方形や雲形の記号は分析者や開発者が作成する図面を表している。図面間の破線の矢印は、realize, refine, trace のいずれかの依存関係を示し、コメントシンボルに記載された内容によりその理由を示す。例えば、図 18 の左上において、アクターとユースケースを洗い出すために、要求を参照してユースケース図を作成する。

依存関係の両端の要素の型の組み合わせを、図 18 に基づいて、以下の 4 種類に分類する。

Classifier とその振舞 例えば、クラスと状態遷移図のように、型とその振舞の間には依存関係がある。型の例としては、アクター、ユースケース、クラス、コンポーネント、ノード、オブジェクト図のオブジェクトなどがある。これらのメタ概念を Classifier と呼ぶことにする†4。振舞の表現としては、状態遷移図、アクティビティ図がある。また、クラス図とシーケンス図やコラボレーション図のように、構造を持った型とその振舞の間の依存関係もある。

振舞の抽象と具象 例えば、ユースケースとコラボレーション図のように、振舞の抽象表現と具象表現の間に依存関係がある。

†3 CASE ツールの XML データベースを実際に解析することにより、このことは確認済みである。

†4 UML1.5 版における Classifier の定義とは若干異なる。

要求
アクターと
ユースケースの
洗い出し
ユースケースの
詳細な振舞定義
ユースケース図
refine
ユースケースの
再構成
状態遷移図
アクティビティ図
シーケンス図
コラボレーション図
refine
分析クラス（概要）
の洗い出し
trace
ユースケースと
パッケージの対応
ユースケース図
trace
分析クラスを用いた
ユースケースの実現
クラス図
realize
シーケンス図
コラボレーション図
trace
refine
クラス図
trace
パッケージと
分析クラス間の包含、
パッケージ間の関係
クラス図
分析クラスの
責務の洗い出し
refine
refine
ユースケースに
参加する
設計クラス（概要）の
洗い出し
trace
クラス図
クラス図
パッケージ間の
インタフェースの
洗い出し
設計クラスを用いた
ユースケースの実現
realize
refine
シーケンス図
コラボレーション図
クラス図
refine
パッケージの
インタフェースとなる
設計クラスの決定
設計クラスの
操作の定義
refine
クラス図
trace
設計クラスの
振舞の定義
realize
クラス図
状態遷移図
コンポーネントと
パッケージの対応の
決定
trace
trace
コンポーネント図
設計クラスと
コンポーネントとの
対応
コンポーネント図
trace
システム
構成
コンポーネントを
ノードへ配置
trace
物理的構成の
設計
配置図
refine
配置図

図 18　Unified Process に従って生成された UML 図面間の依存関係

表3　セマンティクスを用いたUML1.5版における依存関係の分析

依存関係のステレオタイプ	基本的なセマンティクス	ターゲットの型	ソースの型
realize	情報共有	仕様	実現
refine	詳細化	詳細化以前の要素	詳細化後の要素
trace	時間順序	単純な抽象概念 (祖先)	詳細な抽象概念 (子孫)
		前の版	後の版

概念的構造の抽象と具象　例えば、クラス名だけ定義されたクラス図と操作のシグネチャや属性の型が定義されたクラス図との間のように、抽象概念構造と具象概念構造の間には依存関係がある。

物理的構造の抽象と具象　例えば、クラス図に定義されたクラスと、コンポーネント図中のコンポーネントとの間のように、抽象物理構造と具象物理構造の間には依存関係がある。

5　メタ要素の定義

前節における分析結果をふまえ、情報共有、詳細化、作成順序の3種類のメタ関係の両端にくるソースとターゲットの型を検討する。

realizeに対応するメタ関係「情報共有」　メタ関係「情報共有」の両端には、Classifier(表4のメタ要素「Classifier」)とその振舞(表5のメタ要素「振舞図」)がくる。また、メタ関係「情報共有」の両端には、構造を持った型(表5のメタ要素「関係図」)と、その振舞(表5のメタ要素「相互作用図」)が設定される場合もある。

refineに対応するメタ関係「詳細化」　メタ関係「詳細化」の両端には、ユースケースと相互作用図(シーケンス図およびコラボレーション図)がくる。

traceに対応するメタ関係「作成順序」　メタ関係「作成順序」の両端にくる要素は、一般に開発者の分析・設計に関するセマンティクスに依存し、特定できない。以下のように取り扱う。ここまで定義したメタ要素のスーパークラスとして、メタ要素「MetaElement」を定義する。「MetaElement」に再帰のメタ関係を導入する。

上記のメタ要素に加えて、いくつかのメタ要素が必要である。すなわち、関係図、振舞図、相互作用図は構造を持つ。関係図は、Classifierを関連、依存、集約、汎化、リンクなどで結合したものである。これらに対するメタ要素「関係」を定義する。振舞図は、状態遷移図とアクティビティ図の構成要素を一般化する。メタ要素として、メタ状態とメタ遷移を定義する。相互作用図は、シーケンス図やコラボレーション図の構成要素を一般化する。メタインスタンスおよびメタメッセージとなるメタ要素を定義する。

表4　モデル要素のメタ要素

メタ要素	図面要素
Classifier	アクター, ユースケース クラス, パッケージ, ノード, コンポーネント, オブジェクト (オブジェクト図)
関係	関連, 依存, 集約, 汎化, リンク
メタ状態	状態, アクション状態
メタ遷移	遷移, イベント, アクション
メタインスタンス	オブジェクト (シーケンス図、 コラボレーション図)
メタメッセージ	メッセージ

6　メタモデルの定義

前節における考察結果に基づいて、メタ要素をメタ関係で関連付けたメタモデルを図19に示す。

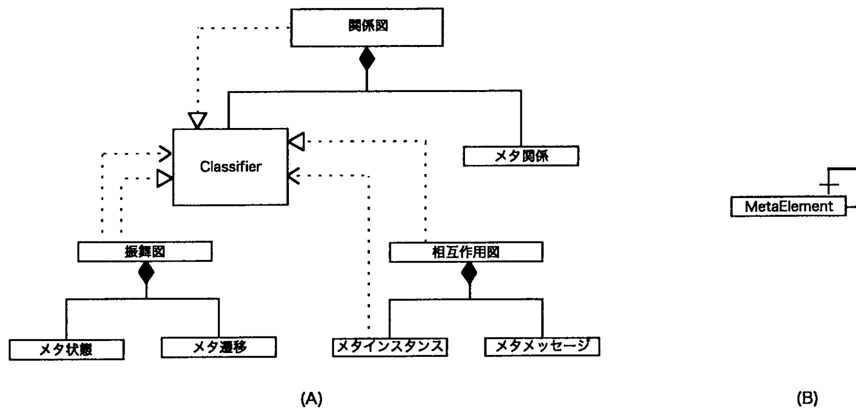

図 19 依存関係に関するメタモデル

表 5 UML 図のメタ要素

メタ要素	UML 図
関係図	ユースケース図, クラス図, オブジェクト図, コンポーネント図, 配置図
振舞図	状態遷移図, アクティビティ図
相互作用図	シーケンス図, コラボレーション図

図 19 において、

- メタ関係「生存従属」が、関係図と Classifier、関係図とメタ関係、振舞図とメタ状態、振舞図とメタ遷移、相互作用図とメタインスタンス、相互作用図とメタメッセージの間に設定される。
- メタ関係「情報共有」が、Classifier と振舞図、Classifier とメタインスタンスの間に設定される。
- メタ関係「詳細化」が、Classifier と関係図、Classifier と相互作用図の間に設定される。この関係は Classifier と関係図の詳細化の関係は、実際は、ユースケースとクラス図の間にのみ設定され、また、Classifier と相互作用図の詳細化の関係は、実際は、ユースケースとコラボレーション図、または、ユースケースとシーケンス図の間にのみ設定される。他の Classifier や関係図との間で情報共有の依存関係を設定しないためのルールは、次節で述べる照合規則によって実現される。
- trace に対応するメタ関係「作成順序」が MetaElement 間に設定される。

7 依存関係の自動生成

依存関係の自動生成は、以下の手順で行われる。

1. 入力された 2 つのモデル要素を比較すべきか否かを照合規則により決定する。
2. 照合規則を満たしたとき、各モデル要素に対応するメタ要素を決定する。
3. メタモデルを利用し、それらのメタ要素間に設定されているメタ関係を特定する。メタモデルの適用は 2 段階にわかれる。まず最初に、図 19 におけるメタモデル (A) を適用する。つぎに、メタモデル (B) を適用する。実際に作業順序のインスタンスを生成するか否かは、作業履歴に関する情報を必要とする。また、すでにメタモデル (A) によってメタ関係のインスタンスが生成されている場合には、メタ関係「作業順序」のインスタンスを生成しない。すでに存在従属や情報共有の依存関係が設定されているモデル要素間に、作業順序の依存関係を生成しないためである。
4. 特定されたメタ関係のインスタンスを、入力された 2 つのモデル要素間に張る

7.1 照合規則

変換手順の最初にやるべきことは、比較対象となる2つのUMLモデル要素の抽出である。これを照合規則として形式化する。メタ関係は、変更に関する本質的な情報を定義しているが、図面やモデル要素に直接表現されるわけではない。比較対象を特定するために手がかりとなる情報は、モデル要素に付けられた名前である。表6に、各UMLモデル要素のマトリックスを定義し、格子点には比較方法を記述する。

最左列および最上段は、UMLモデル要素やUML図の名前が同順に並んでいる。

行と列で交差する格子点には、照合の条件を示す。条件で扱う属性にはname、diagram、elementの3種類がある。0nameは要素の名前を、diagramは図面を、elementは図面要素を表す。また、属性間の関係には、equal、includeの2種類があり、equalは両要素が全く同じ値であることを、includeは列要素の属性が行要素の属性を含むことを示す。いくつかの例を説明する。(1) 2つのクラス間の照合にあたっての条件は、2つのクラスの名前が同じであることである。(2) クラスとオブジェクトの照合にあたっての条件は、オブジェクトの名前がクラスの名前を含むことである。(3) クラスがあるクラス図の要素であるかの照合にあたっての条件は、クラス図の構成要素の名前がそのクラスの名前と同じであることである。

7.2 生成例

次節で利用する例題を用いて、生成手順の例を示す。

クラス "ElevatorControl" とコラボレーション図のオブジェクト ":ElevatorControl" を入力したとする。表6に示す照合規則から、クラスとオブジェクトの格子点の規則を確認する。規則には「オブジェクトの名前がクラスの名前を含む」とある。この例ではクラス名ElevatorControlがオブジェクトの名前に含まれているため、照合規則を満たすことになる。次に、適合した各モデル要素に対応するメタ要素を決定する。表4に示すメタ要素とモデル要素の対応表より、クラスのメタ要素はClassifier、コラボレーション図のメタ要素はメタインスタンスであることが確認できる。次に、メタ要素間にあるメタ関係を特定する。図19に示すメタモデルには、Classifierとメタインスタンス間に情報共有と生存従属のメタ関係が定義されていることが確認できる。最後に、メタ関係「情報共有」のインスタンスとメタ関係「生存従属」のインスタンスを、それぞれ入力されたモデル要素間に張る。

8 評価

本節では、本論文で提案した手法の原理的な有効性を評価する。例として、Unified Processの一種である開発方法論COMET[6]により作成された、エレベータ制御システムの分析・設計例(40枚のUML図)を利用する。

クラス "ElevatorControl" に関する依存関係に着目した例を示す。本論文で提案した変換法を実現したプロトタイプシステムを開発し、実験を行った。クラス "ElevatorControl" に関係する図として、2枚のクラス図、そのクラスの状態遷移を示す5枚の状態遷移図、そのクラスのオブジェクトを持つ7枚のコラボレーション図が抽出された。

図20に示すように、対象となる2つのクラスと5枚の状態遷移図の間に、詳細化と情報共有の依存関係が生成された。また、クラス間に情報共有の依存関係が生成された。

図21に示すように、型が同じクラスの複数のオブジェクト間に、情報共有の依存関係が両方向に生成された。また、図では表現してないが、図20にある2つのクラスと、図21にある7つのオブジェクト間にも、クラスをターゲットとする情報共有の依存関係が生成された。

これらの実験結果に基づいて、照合規則および生成された依存関係の有効性の評価を行う。

4節で述べた、本論文で対象とする依存関係が適切に生成できたかどうかを確認した結果を、表7に示す。

表7の2列目に、図18を判断基準にして、生成されるべきモデル要素や図面間の依存関係を示す。3列目に、正しく照合されたか否かを示す。4列目に、2列目の各項目に対応するメタ要素間に定義されたメタ関係を示す。

表7の結果に基づいて、照合規則および生成され

表 6　照合規則

	アクター	ユースケース	クラス	パッケージ	コンポーネント	ノード	状態	アクション状態	オブジェクト
アクター	name/name equal								
ユースケース	-	name/name equal							
クラス	-	-	name/name equal						
パッケージ	-	-	name/name equal	name/name equal					
コンポーネント	-	-	-	-	name/name equal				
ノード	-	-	-	-	-	name/name equal			
状態	-	-	-	-	-	-	name/name equal		
アクション状態	-	-	-	-	-	-	-	name/name equal	
オブジェクト	-	-	name/name include	-	-	-	-	-	name/name equal
ユースケース図	diagram/element include	diagram/element include	-	-	-	-	-	-	-
クラス図	-	name/name include	diagram/element include	diagram/element include	-	-	-	-	-
オブジェクト図	-	name/name include	-	-	-	-	-	-	diagram/element include
コンポーネント図	-	-	-	-	diagram/elemnt include	-	-	-	-
配置図	-	-	-	-	-	diagram/elemnt include	-	-	-
状態遷移図	-	name/name include	name/name include	-	-	-	diagram/element include	-	-
アクティビティ図	-	name/name include	name/name include	-	-	-	-	diagram/element include	-
コラボレーション図	-	name/name include	-	-	-	-	-	-	diagram/element incolude
シーケンス図	-	name/name include	-	-	-	-	-	-	diagram/element include

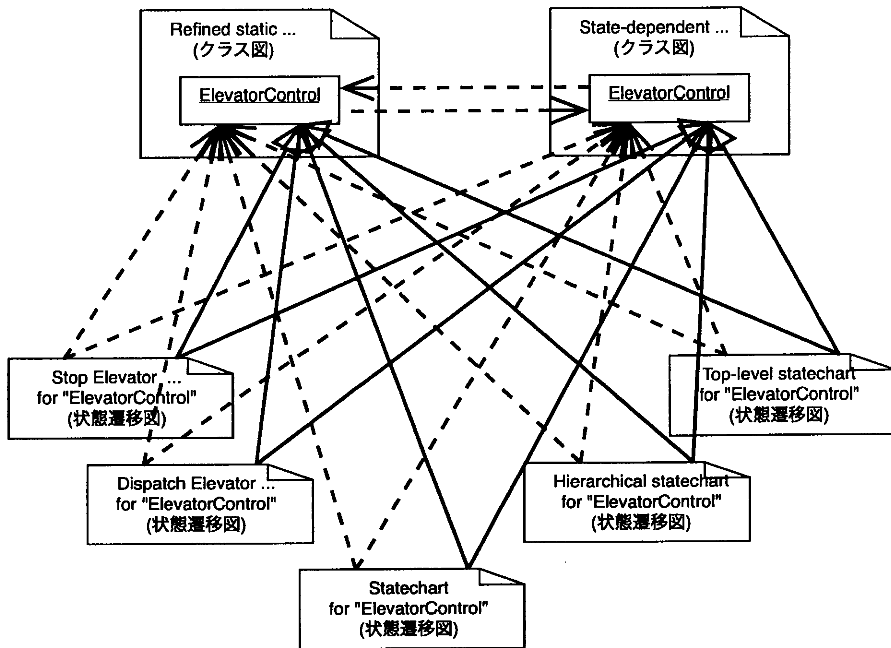

図 20　クラス "ElevatorControl" と状態遷移図間の依存関係

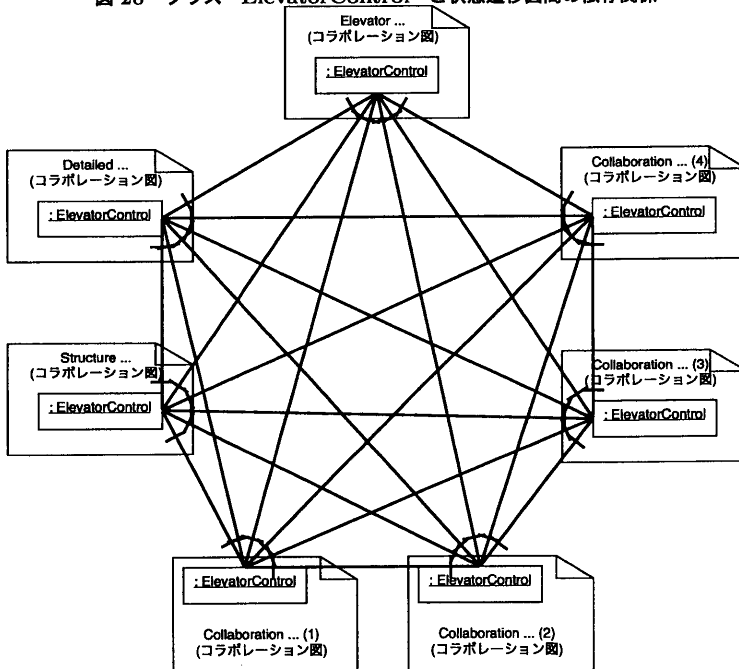

図 21　型が同じクラスの複数のオブジェクト間の依存関係

表 7 提案手法によって生成できる依存関係の種類

依存関係が発生する要素間の関係	UML 図を用いた例 (ターゲットからソース)	照合規則に含まれるか	自動生成された依存関係
対象とその振舞	クラスから状態遷移図	含まれる	情報共有、詳細化
	クラスから振舞オブジェクト (クラス図からコラボレーション図)	含まれる	情報共有
	ユースケースからクラス図	含まれる	詳細化
	ユースケースからパッケージ	-	なし
振舞の抽象と具象	ユースケースから状態遷移図	含まれる	情報共有、詳細化
	ユースケースからアクティビティ図	含まれる	情報共有、詳細化
	ユースケースからコラボレーション図	含まれる	詳細化
	ユースケースからシーケンス図	含まれる	詳細化
	ユースケースからユースケース	含まれる	作成順序 (双方向)
概念的構の抽象と具象	パッケージからパッケージ	含まれる	作成順序 (双方向)
	クラスからクラス	含まれる	作成順序 (双方向)
物理的構造の抽象と具象	クラスからコンポーネント	-	なし
	コンポーネントからノード	-	なし

た依存関係 (メタ関係のインスタンス) の効果と問題点を評価する。

8.1 照合規則の評価

8.1.1 有効性

表 7 に示すように、ユースケースからパッケージ、クラスからコンポーネント、コンポーネントからノードを除いて、適切な照合がなされている。これらの照合結果が必要十分なものであるかどうかを確認するため、別途、図面群の依存関係を調査した。その結果、依存関係があると判断されなかった図面群については、2 種類の特徴があった。1 つは、(1) 本来、依存関係が存在しない図面群、(2) ある図面の上位概念を図面要素として持つ図面群である。(1) により、あやまった照合がないことがわかる。(2) により、本来、依存関係が生成されるべきであるのに、生成されていないことがわかる。この例がユースケースからパッケージ、クラスからコンポーネント、コンポーネントからノードの 3 つに対応する。この問題について、原因を次節で検討する。

8.1.2 今後の改良点

抽象度の高いモデル要素 (上位モデル要素) が、より具体的なモデル要素群 (下位モデル要素群) を包含する場合は、図 22 に示すように、4 通りある。

1. 上位モデル要素と下位モデル要素が同じ名前である
2. 下位モデル要素群が上位モデル要素を示すパッケージで包含される
3. 上位モデル要素の名前を使った 1 枚の図面を使って、下位モデル要素群が表現される
4. 開発者の認識のみ

これらのうち、本手法で生成できない依存関係は 4 の場合だけである。例えば上位モデル要素 “Elevator-Control” から、下位モデル要素群 “FloorControl” と “BoxControl” に分割されたとする。この場合、当然、名前だけでは照合できない。

8.2 生成された依存関係の評価

メタモデルにより、UML モデル要素間に設定された依存関係 (メタ関係のインスタンス) を表 7 に示す。クラスから状態遷移図、ユースケースから状態遷移図、ユースケースからアクティビティ図、クラスから

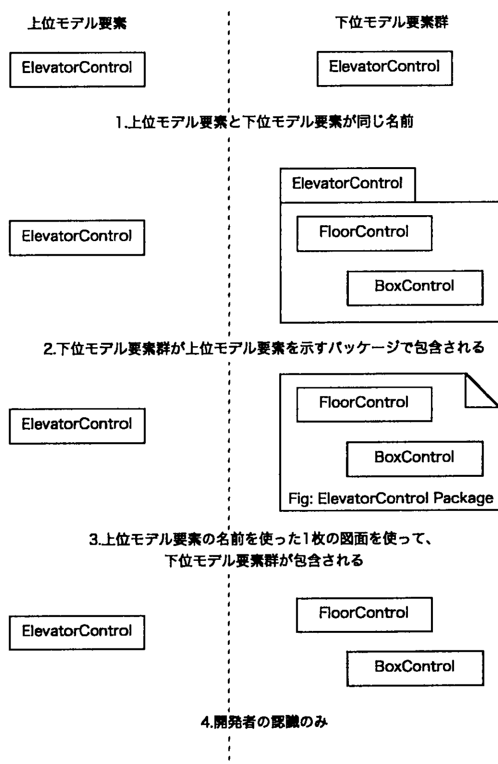

図 22　上位モデル要素と下位モデル要素群の表現方法

クラス、パッケージからパッケージを除き、適切な依存関係 (メタ関係のインスタンス) が生成されている。上記 5 つの問題点について、その原因を考察する。

1. クラスから状態遷移図：本来は、情報共有の依存関係のみが生成されるべきである。これは、ユースケースを Classifier の分類とは独立にメタモデルに定義することにより解決できる。
2. ユースケースから状態遷移図本来は、詳細化の依存関係のみが生成されるべきである。これは、ユースケースを Classifier の分類とは独立にメタモデルに定義することにより解決できる。
3. ユースケースからアクティビティ図本来は、詳細化の依存関係のみが生成されるべきである。これは、ユースケースを Classifier の分類とは独立にメタモデルに定義することにより解決できる。
4. クラスからクラス作成順序の依存関係が両方向に生成されていた。開発履歴に関する情報を別途、与えられる場合は、正しい作成順序関係になる。開発履歴に関する情報がない場合は、コピーの関係になる。
5. パッケージからパッケージ作成順序の依存関係が両方向に生成されていた。開発履歴に関する情報を別途、与えられる場合は、正しい作成順序関係になる。開発履歴に関する情報がない場合は、コピーの関係になる。

4 と 5 の問題について、開発履歴に関する情報を仮に与えて[†5]生成される依存関係を確認してみた。

作成順序に関する情報を図 23 のように与えたとき、図 21 が図 24 のように変化し、コピー関係が正しく生成されていることが確認された。

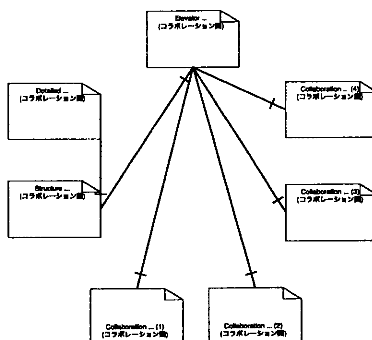

図 23　図面間に付加した作成順序

9　関連研究

変更波及は論理的な波及と、システムの性能に関する波及の 2 種類にわけられる [10] [19]。論理的な波及とは、他の成果物との一貫性を論理的に保証するための変更波及であり、システムの性能に関する波及とは、システム性能の向上を目的として構造や振舞を再構成するための変更波及である。本論文で提案する依存関係は、論理的な波及解析に利用するためのもので

†5　開発方法論に依存する作業モデルの定義と、作業モデルの実行結果から開発履歴に関する情報を自動的に生成する部分の形式化は今後の課題であるが、ここでは、作業順序に関する情報を仮に与えることにより、4,5 の問題がどの程度解決できるか確認した。

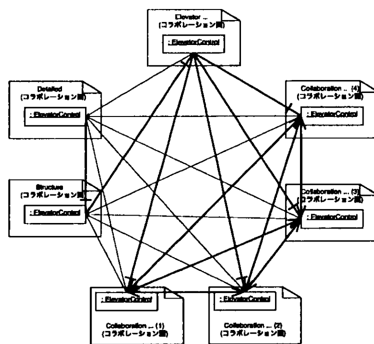

図 24　図 21 の改良図

ある。

論理的な変更波及解析手法としては、スライシングや形式手法の利用、依存関係を追跡する手法などが提案されてきた。スライシング技法を用いたプログラムのための波及解析 [11] [13] は、変更されたメソッドや変数と同じ名前を持つものを捜す手法であり、おもにソースコードに対して適用される。形式的意味論を用いたクラス図と状態遷移図間 [7] の波及解析に関する研究や、クラス図とコラボレーション図間 [18] の波及解析に関する研究は、図面要素間の整合性検証を行い発生した矛盾を検知する手法である。また、依存関係を追跡して図面の波及解析をする手法もあり、図面の完成度やメトリクスを用いて、追跡する依存関係の範囲を決定する方法が提案されている [2] [3]。我々の研究は依存関係を追跡することにより、変更の波及を解析する立場をとる。著者らの知見の範囲では、そのような依存関係を自動生成しようとする試みは新しいものである。このとき、どのような依存関係を定義すれば効果的な追跡が行えるかが問題となる。依存関係の種類を定義する方法としては、コンセプトマップを用いた解析 [15] や、分散管理におけるコンポーネント間の特徴を用いた解析 [1] などが提案されている。本論文では、現在普及しつつあるユースケース駆動型プロセスを対象とし、分析図面や設計図面は UML により表現されることを前提としたアプローチを行った。具体的には UML1.5 版において定義された 13 個の依存関係のプリミティブなセマンティクスを分析し、5 つの新しい依存関係を定義した。このようなアプローチもきわめて新しいものと考える。

10　まとめ

本論文では、UML1.5 版で定義されている依存関係のプリミティブなセマンティクスを検討することにより、変更波及解析に有用な依存関係 (情報共有、詳細化、作成順序、生存従属、コピーの 5 つ) を新たに定義した。また、そのような依存関係を Model-based Translation 手法を適用して自動生成する手法を提案した。

COMET 法によるエレベータ制御システムの事例研究を題材として、提案した依存関係と生成法が原理的には有用であることを示した。

生成法については、照合規則やメタモデルを本論文で指摘した点について改良し、精度を向上させる必要がある。また、図面間の対応だけではなく、図面とソースコード間の依存関係も検討する必要があり、現在考察中である。

今後の課題は以下の通りである。

- 本論文で提案した手法により、UML モデル要素が依存関係で結合されたグラフ構造を生成することができる。そのグラフ構造を変更作業手順を示すワークフローに変換するアルゴリズムを開発する予定である。そのためには、作業モデルを定義し、作業モデルの実行結果から変更に関する作業順序の情報を抽出するアルゴリズムの開発が必要であり、今後の課題である。
- 一般に、変更作業は複数の変更作業スレッドからなる。複数の変更作業スレッドは、一般に、図面や図面の一部を共有する。共有図面の存在により同期をとるような作業実行の形態は非効率である。また、ソフトウェア開発作業は複数人による共同作業であり、各図面ごとにアクセス権が異なる。本手法を実世界で利用するにあたっては、効率のよい同期方式、適切なアクセス権の設定法などを考案する必要があり、今後の課題である。

謝辞

本研究は文部科学省科研費特定領域研究 (2) 課題番号 16016239 の補助を基に実施された。

参考文献

[1] Alexander Keller, Uri Blumenthal and Gautam Kar.: Classification and Computation of Dependencies for Distributed Management, *Proc. of the 5th IEEE Symposium on Computers and Communications (ISCC 2000)*, 2000, pp78-83.

[2] Anna Rita Fasolino, Giuseppe Visaggio.: Improving Software Comprehension through an Automated Dependency Tracer, *7th International Workshop on Program Comprehension*, 1999, pp.58-65.

[3] Antje von Knethen, Mathias Grund.: QuaTrace : A Tool Environment for (Semi-) Automatic Impact Analysis Based on Traces, *19th IEEE International Conference on Software Maintenance (ICSM'03)*, 2003, pp.246-255.

[4] Dragon Milićev.: Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments, *IEEE Trans. Software Eng., vol. 28, no. 4*, April, 2002, pp.413-431.

[5] Grady Booch, James Rumbaugh, Ivar Jacobson.: *The Unified Modelig Language User Guide*, Addison Wesley Longman, Inc., 1999.

[6] Hassan Gomma.: *Designing concurrent, distributed, and real-time applications with UML*, Addison Wesley, Inc. 2000.

[7] Holger Rasch and Heike Wehrheim.: Consistency between UML Classes and Associated State Machines, *5th International Conference on the UML*, 2002, pp.46-60.

[8] Ivar Jacobson, Grady Booch and James Rumbaugh.: *The Unified Software Development Process*, Addison Wesley Longman, Inc., 1999.

[9] James Rumbaugh, Ivar Jacobson and Grady Booch.: *The Unified Modeling Language Reference Manual*, Addison Wesley Longman, Inc., 1999.

[10] James S. Collofello, Mikael Orn.: A Practical Software Maintenance Environment, *Proc. of Conf. on Software Maintenance*, 1988, pp. 45-51.

[11] Jianjun Zhao.: Change Impact Analysis for Aspect-Oriented Software Evolution, *International Workshop on Principles of Software Evolution*, 2002, pp.107-112.

[12] Jim Arlow, Ila neustadt.: *UML and the Unified Process*, Peason Education Limited, 2002.

[13] Keith Brian Gallagher and James R. Lyle.: Using Program Slicing in Software Maintenance, *IEEE Trans. Software Eng., Vol. 17, No. 8*, 1991, pp.751-761.

[14] Khawar Zaman Ahmed and Cary E. Umrysh.: *Developing Enterprise Java Applications with J2EE^{TM} and UML*, Addison-Wesley, 2002.

[15] Lisa Cox, Dr. Harry S. Delugach.: Dependency Analysis Using Conceptual Graph, *Proc. 9th Intl. conf. on Conceptual Structures*, 2001, pp.117-130.

[16] Martin Fowler with Kendall Scott.: *UML Distilled Second Edition*, Addison Wesley Longman, Inc., 2000.

[17] Object Management Group.: *Unified Modeling Language (UML), version 1.5*, http://www.omg.org/cgi-bin/doc?formal/03-03-01

[18] Richard F.Paige, Jonathan S. Ostroff, and Phillip J. Brooke.: Checking the Consistency of Collaboration and Class Diagrams using PVS, *Proc. of 4th Workshop on Rigorous Object-Oriented Methods(ROOM4)*, 2002,

[19] S. S. Yau, J. S. Collofello, and T. MacGregor.: Ripple Effect Analysis of Software Maintenance, *Proc. IEEE COMPSAC*, 1978, pp. 60-65.