修士学位論文

題目

UML メタモデルの変更に対応した
ダイアグラム間整合性検証環境の自動生成手法

指導教員
井上 克郎 教授

報告者
浜口 優

平成 20 年 2 月 8 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 19 年度 修士学位論文

UML メタモデルの変更に対応した
ダイアグラム間整合性検証環境の自動生成手法

浜口 優

内容梗概

オブジェクト指向型言語を用いたソフトウェア開発において広く利用される UML (Unified Modeling Language) では，複数種類のダイアグラムが定義されている．しかしこれら複数種類のダイアグラムは相互に依存しているため，システム設計の過程で矛盾したダイアグラムを作成するとシステムに不整合が生じることとなる．この問題に対して，UML メタモデルへ制約を付加することにより UML ダイアグラム間の整合性を保つ研究が行われている．しかし，これらの研究では UML メタモデルのバージョン変更に対応できないため，各バージョンの UML メタモデルに対して個別のモデル検証環境を用意する必要があった．本研究では，UML メタモデル及びそれに付加したオブジェクト制約記述言語 OCL (Object Constraint Language) を用いて記述した制約を，モデル駆動型アーキテクチャを支援するフレームワーク EMF (Eclipse Modeling Framework) に対して入力として与えることで，UML モデル検証環境を自動生成する手法を提案する．具体的には，EMF のモデル表現形式である Ecore モデルを UML メタモデルと整合性ルールを表現した OCL 制約から生成する．次に，生成された Ecore モデルを EMF に対して入力として与えることで UML モデル検証環境のソースコードを自動生成する．また提案手法に基づいて UML ダイアグラム間整合性検証を支援するツールを作成した．UML2.0 及び UML2.1.1 のメタモデルに基づく飲料生成システムを作成し，それぞれに対して適用した．その結果，提案手法による UML ダイアグラム間整合性検証が可能であることが確認でき，手法の有効性が示された．

主な用語

UML (Unified Modeling Language)
ダイアグラム間整合性検証 (Diagrams Verification)
メタモデル (Metamodel)
モデル駆動型アーキテクチャ (Model Driven Architecture)

目 次

1　まえがき

ソフトウェア開発において，UML (Unified Model Language) がソフトウェアのモデル記述に幅広く利用されている [20]．UML を用いることで，開発者は対象の情報を抽象化することにより，より明確に対象の特徴を把握することができる [1]．UML モデルには，様々な視点からソフトウェアシステムを表現するために複数の UML ダイアグラムが存在する．例えば，ソフトウェアシステムの静的な側面を表すクラス図やコンポーネント図，動的な側面を表すシーケンス図やステートマシン図がある．

UML の普及に伴い，モデル駆動型開発 (Model-Driven Development：MDD) が注目を集めている [2]．モデル駆動型開発とは，ソフトウェアのモデル化に重点を置いたソフトウェア開発手法である．モデル駆動開発により，様々な側面を抽象化したモデルを段階的に複数回に渡って変換していくことで，再利用性の高いソフトウェアシステムを構築することが出来る．UML モデルを用いたモデル駆動型開発では，開発工程を進める毎に UML ダイアグラムを修正，詳細化していく．例えば，開発の初期段階ではクラス図のみを作成し、開発の進行に伴なってシーケンス図など様々な種類の UML ダイアグラムが追加されていくことになる．

しかし，これら複数種類の UML ダイアグラムは相互に依存しているため，UML ダイアグラム間で名前要素の不一致などの不整合を起こす可能性がある．従って，UML ダイアグラム間で整合性を保つ必要がある．

この問題に対して，取り組んでいる既存研究やモデリングツールが存在する．Rational Rose [16] は UML ダイアグラム間の整合性を保つ機能を備えているが，通常のモデル記述に加えて，UML ダイアグラムの構成要素間に依存関係を定義する処理が必要になる．文献 [41] の研究や UML/Analyzer [9, 10, 11] は不整合の修正を目的とし，修正動作の選択肢を提示する．また，UML モデルと UML ダイアグラム間の整合性ルールを形式化することにより，整合性検証を行っている研究 [15, 22, 24, 25] も数多く存在する．しかし，上記の研究では UML モデルの仕様である UML メタモデルを変更することができない．UML メタモデルは頻繁に変更 [29]，もしくは拡張 [42] されるため，逐一対応するコストが生じる．

そこで本研究では UML メタモデルと OMG [28] の標準規格である制約記述言語 OCL (Object Constraint Language) [37] で記述された制約式を入力として与えることで，UML モデル検証環境を自動生成する手法を提案する．提案手法では，EMF (Eclipse Modeling Framework) [7] に基づいて，OMG より入手可能である.mdl ファイルまたは.uml ファイルで表現された UML メタモデルと，OCL で記述された整合性ルールを入力として与えることで EMF のモデル表現形式である Ecore モデルを生成する．次に Ecore モデルから EMF モデルを生成し，EMF モデルから UML モデル検証環境のソースコードを自動生成する．UML

メタモデルからEcoreモデルへは変換可能であり，EcoreモデルはOCL制約を付加することができる．ユーザは生成されたUMLモデル検証環境を用いて任意のUMLダイアグラムを編集し検証することができる．また提案手法に基づいて支援ツールを作成した．適用実験ではクラス図とシーケンス図間の整合性検証を2つの異なるUMLメタモデルの下で行う．

以降，2章で提案する手法の背景について述べ，3章でUMLモデル検証環境を構築する手法を述べる．4章で適用実験を行い，5章で実験の考察と評価を述べ，最後に6章でまとめと今後の課題を述べる．

2　準備

本章では，提案する手法を説明するにあたり必要な背景を述べ，既存研究とその問題点について考察する．

2.1　UML モデルと UML メタモデル

OMG は UML を 4 階層のアーキテクチャを用いて設計している．

1. MOF (Meta Object Facility) [34]：M3 層に相当し，UML メタモデルを記述するための言語を定義する．
2. UML メタモデル：MOF のインスタンス．M2 層に相当し，UML モデルを記述するための言語を定義する．
3. UML モデル：UML メタモデルのインスタンス．M1 層に相当し，オブジェクトモデルを記述するための言語を定義する．
4. オブジェクトモデル：UML モデルのインスタンス．M0 層に相当し，特定のオブジェクトを表現する．

図 1 は OMG のメタモデル 4 層の例を表している．M3 層である MOF において Class クラスと Property クラスを定義することで，M2 層の UML メタモデルにおいて Class クラスとその属性をインスタンス化することが可能になる．同様に，UML メタモデルにおいて Class クラスと Property クラスを定義することで，M1 層である UML モデルにおいて Class クラスとその属性をインスタンス化することが可能になる (図 1 では Person クラスとしてインスタンス化している)．

UML メタモデルを定義することで UML モデルの仕様が決定される．したがって UML メタモデルに制約を付加することで UML モデルを制御することが可能になる．図 2 はクラス図に関係する部分を簡略化した UML メタモデルである．クラス図のクラスは分類子を継承し，名前，属性，操作を持つことが定義されている．

2.2　オブジェクト制約言語 OCL

OCL [37] は OMG 標準の 1 つであり，制約式を第 1 階論理で記述する．OCL は UML モデル内のモデル要素に対して正確に制約を与えることを目的に導入された．具体的には，OCL 式をクラスに付与することで，図 3 のようにクラスのインスタンスに対して制約を課すことが可能になる．

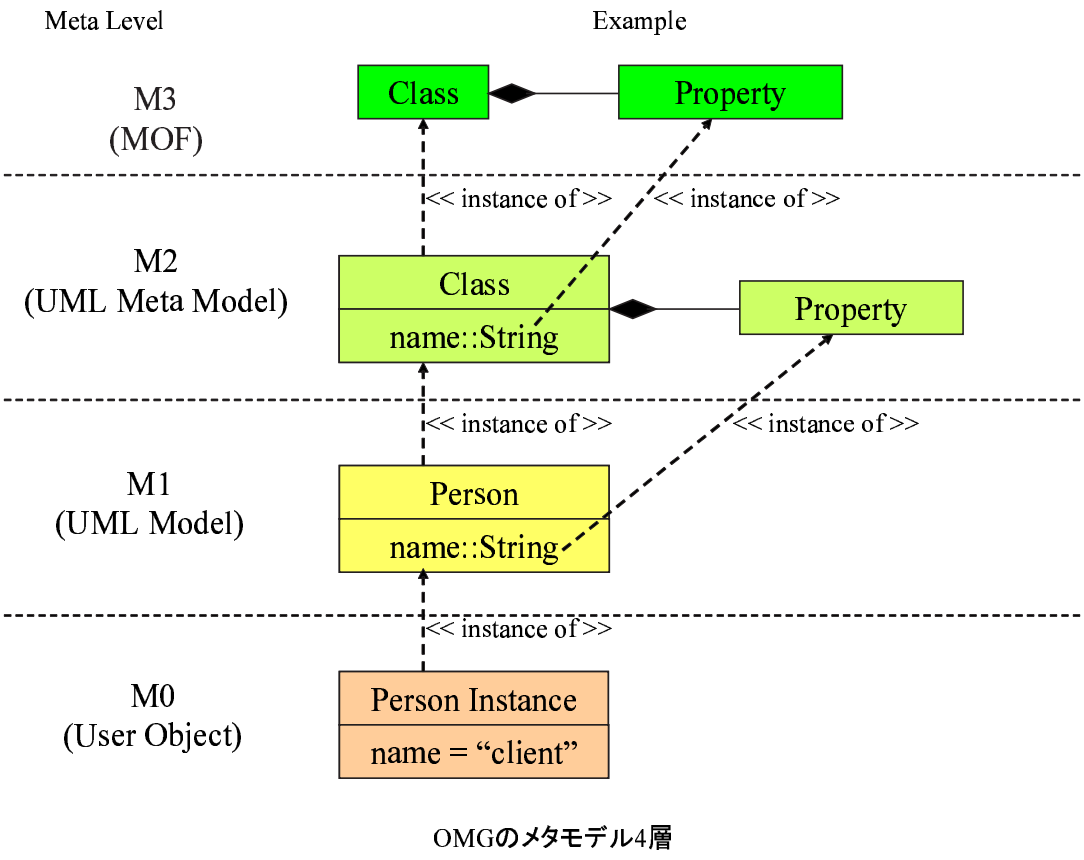

図 1: OMG のメタモデル 4 層

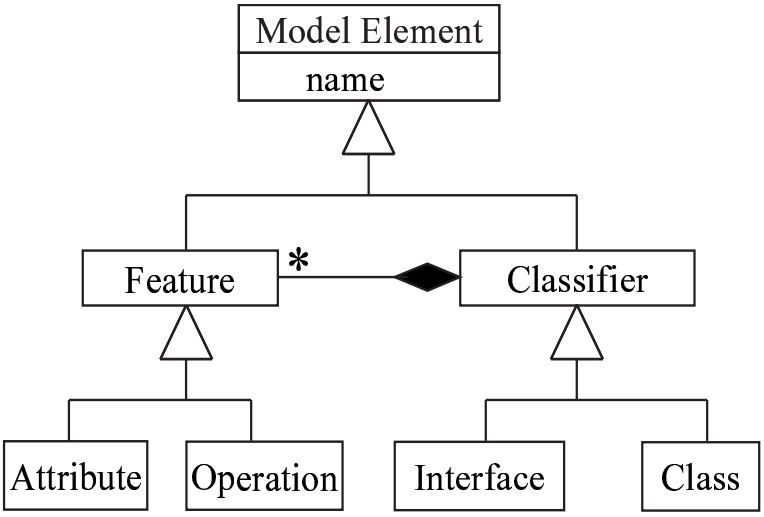

図 2: 簡略化した UML メタモデル

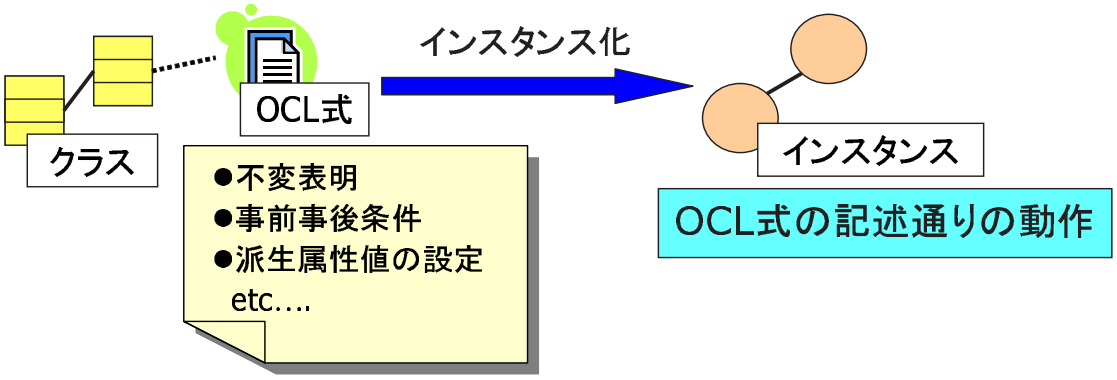

図 3: OCL 制約の実行

OCL は表 1 の条件式を宣言的な型付き言語で記述することにより，主にクラス仕様を表現する．OCL は，基本型，コレクション型，モデル型を持つ．基本型には，整数型，実数型，論理型，文字列型がある．コレクション型はクラスのインスタンスであるオブジェクト集合を表す．モデル型はクラスを表す [45]．OCL 演算子では基本型やコレクション型に関する演算が定義されている．例えば，Order (注文) クラスに不変式を用いて制約を与える場合，次のように定義する．

```
context Order inv:
sum > 10
```

この例は文脈 Order のクラス不変表明である．不変状態は OCL キーワード inv: に続く不変式で指定する．この式は論理式であり，Order クラスのオブジェクト全てについて真である必要がある．この例では Order オブジェクトの sum 属性は全て 10 を超える値が設定されていなければならないことを表している．

本研究で用いる整合性ルールは，OCL の不変式で表現する．

2.3 UML ダイアグラム間の整合性

UML モデルの欠陥の影響を文献 [13, 19] は調査している．調査の結果，欠陥は未検出のまま，誤った実装を引き起こすことが多く，欠陥の種類には被験者が検出しやすいものとそ

表 1: OCL の用途

分類	対象	説明
不変表明	クラス	すべてのオブジェクトが満たすべき条件を指定する
事前事後条件	操作	クラスの操作が実行前に満たすべき事前条件と，操作実行後に満たすべき条件を指定する
ナビゲーション	関連	互いに関連づけられているクラス間で，関連先に対する検索条件を指定する
派生	値	ある属性値が別の属性からどのように計算されるかを指定する
ガード条件	状態遷移メッセージ	状態遷移やシーケンス送信で，複数の選択肢がある場合，それぞれの選択肢が実行される条件を指定する

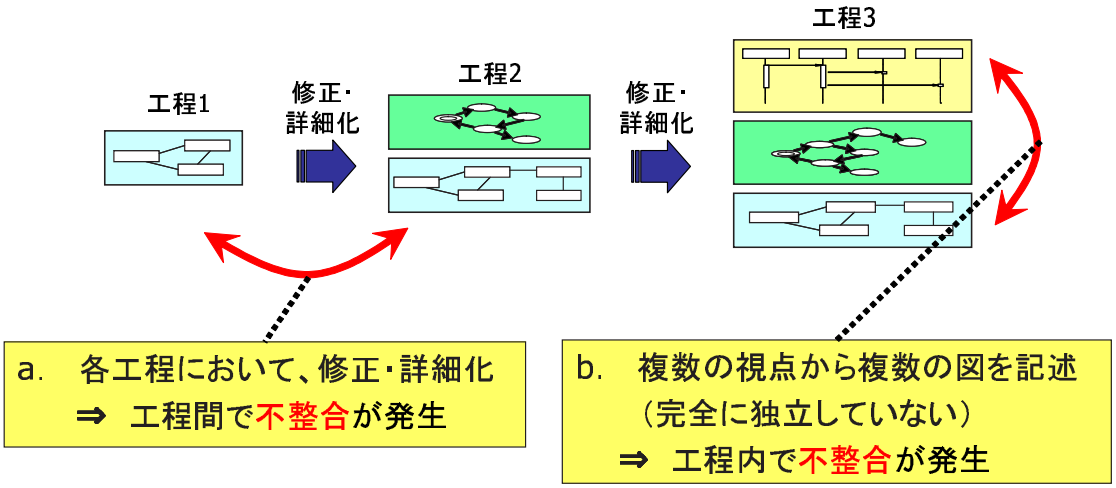

図 4: UML 開発工程における問題点

うでないものが存在することが述べられている．これらの文献から UML ダイアグラム間の不整合はできる限り早期に検出し，修正する必要があることが分かる．UML を用いたモデル駆動型開発では，段階的に UML ダイアグラムを修正，詳細化していく．この段階的な各工程の中で，図 4 のように UML ダイアグラム間には大別して 2 種類の不整合が発生する可能性がある [12].

- a. 各工程間において，同一種類の UML ダイアグラム間で発生する不整合
- b. 同一工程内において，異なる種類の UML ダイアグラム間と同一種類の UML ダイアグラム内で発生する不整合

a. の不整合に対して文献 [43] は各工程間に関する制約を与えて検証を行っている．文献 [27] は各バージョンを考慮した UML モデル管理システムを提案している．文献 [44] は UML メタモデルの依存関係を拡張し，形式化を行ってバージョン間の整合性を検証している．

本研究では b. の不整合に対してユーザが行う検証を目的とする．UML モデルに対してリファクタリングを行った場合も，異なる種類の UML ダイアグラム間で整合性を保つ必要があり [26]，b. の不整合を検出する重要性は高い．

2.4 既存研究

モデルに対する検証手法はいくつか存在する．整合性検証には大きく分けて 4 種類の方法が存在する [14].

1. 人を基準にした共同探査：非形式なモデルに対する唯一の整合性検証手法である．しかし，検証者の能力に依存し，大規模なモデルへの適用は難しい.

2. モデル検査：モデルを状態遷移モデルに変換し，意味的な整合性を検証する手法である．しかし，状態爆発を起こす可能性があり，また，到達可能性などの特別な種類の整合性ルールのみ検証可能である．したがって組込みシステムなどの比較的小さなモデルに対する検証に適している.

3. 特定のフォーマットを用いた解析：XML などの特定のフォーマットにモデルを変換して整合性を検証する手法である．構造的な整合性検証が可能である．しかし検証できない整合性ルールも存在する.

4. 形式手法：形式言語でモデルを表現して整合性を検証する手法である．任意の意味的な整合性を検証可能である．しかし，計算能率は他の手法に劣る.

大規模な UML モデルに対する検証では，主に 3. と 4. を用いた方法が研究されている．これらの研究では，UML メタモデルに対して制約を付加して検証を行っている．そうすることにより，UML メタモデルからインスタンス化された UML モデルに対して整合性検証が可能になる．この手法の利点は，同一仕様に基づく任意の UML モデルに対して整合性検証が可能な点である.

USE [40] は UML1.3 メタモデルの一部を実装し，UML モデルが OCL 式を満たすかをシミュレーションベースで検証する．OCLE [6] は UML1.5 メタモデル，OCL2.0 をサポートし，OMG が規定した WFR の整合性検証を行っている.

そして，2.3 で述べている同一工程内における UML ダイアグラム間整合性検証に関する研究も行われている．文献 [41] の研究では，XMI [36] 形式で保存された UML モデルに対して，独自の記述文法で記述した整合性ルールを用いて整合性検証を行っている．開発したツールでは，整合性ルールと違反した場合に対応する解消動作の組を保存しておくことで，不整合要素の位置情報とそれを修正する解消動作系列の集合をユーザに提示する．UML/Analyzer [9, 10, 11] は Rational Rose [16] に統合されており，独自の記述文法で記述した整合性ルールに影響する全てのモデル要素をスコープとして保存する．これにより整合性ルールに違反した場合は，違反したモデル要素の属するスコープを提示し，不整合の修正を支援する．文献 [24] は UML モデルと整合性ルールの形式化を行い，整合性検証を行っている．文献 [25]

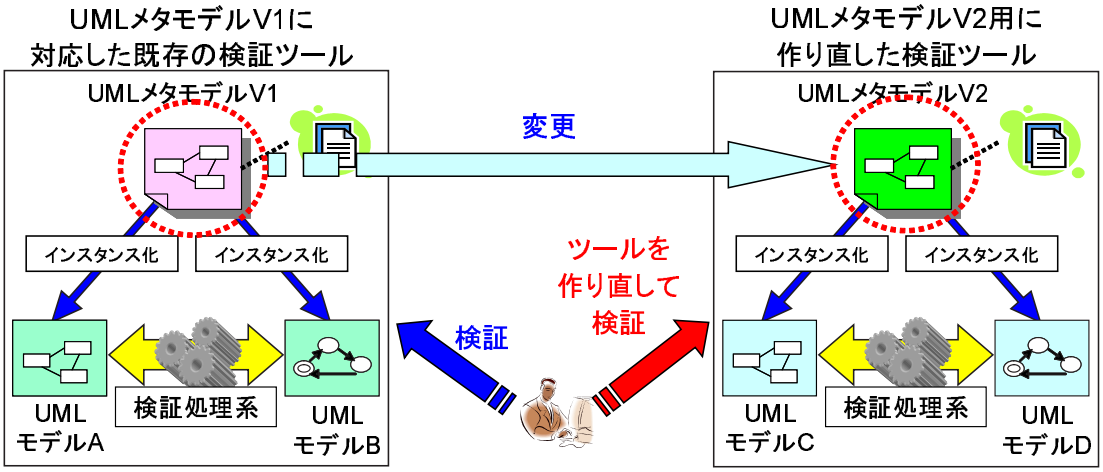

図 5: 既存研究の問題点

は記述論理を用いて整合性検証システムを実現している．文献 [22] は UML モデルを OOL (Object-Oriented specification Language) を用いて形式化し，同一時間軸と別時間軸の整合性検証を行っている．文献 [15] は USDP モデルに対して OCL 式を用いて整合性検証を行っている．文献 [38] はクラス図とシーケンス図を統合したモデルを生成し，統合モデルからテストケースを生成する．生成したテストケースに対して，クラス図に付加しておいた OCL 式を用いてテストを行うことで整合性検証を行っている．文献 [17] は抽象度の異なる UML モデル間の依存関係を定義し，同期することで整合性を保つ手法を提案している．

しかし，上記の研究で実装しているツールの UML メタモデルは変更することができない．開発チームが使用している UML メタモデルの変更や，ドメインに特化した UML メタモデルへの拡張など，UML メタモデルは変更される可能性がある．既存研究の手法では図 5 のように，変更された UML メタモデルに従って，別途検証ツールを作り直して検証する必要がある．UML メタモデルはバージョンチェンジや，組込みシステム向けなどへの拡張が頻繁に行われる．従って，検証ツールを作り直す作業は負担が大きい．

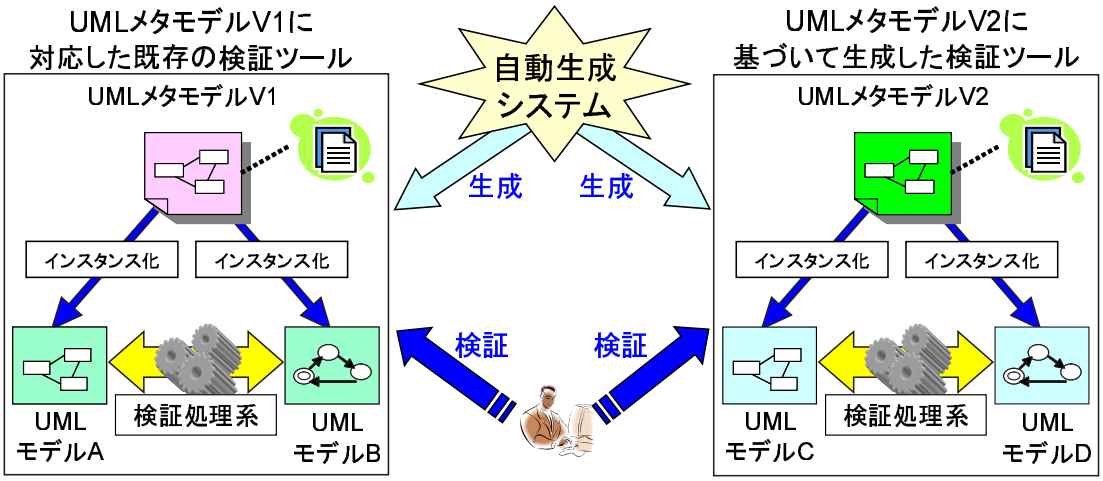

図 6: 本研究の目的

そこで本研究の目的は，UML メタモデルの変更に柔軟に対応できる UML ダイアグラム間整合性検証環境の生成手法の実現である．図 6 のように，自動生成システムを用意し，任意の UML メタモデルに対応した検証ツールを生成することで，検証ツールを作り直す負担を軽減する．

3　提案手法

本章では提案手法と適用した整合性ルールの詳細を述べ，実装したツールの詳細を述べる．

3.1　概要

研究目的を達成するために必要な機能について整理する．初めに，自動生成システムに必要な機能は以下の機能である．

- UML メタモデルの入力機能
- UML メタモデルに基づいて検証ツールを生成する機能

次に，生成される検証ツールに必要な機能は以下の機能である．

- UML モデルの入力機能
- UML モデルに対する検証処理系

そこで本研究では既存のモデリングフレームワークを応用することを考案した．提案手法ではモデリングフレームワークの 1 つである EMF に基づいて，UML メタモデルと OCL で記述された UML ダイアグラム間整合性ルールを入力として与えることで，UML モデル検証環境を自動生成する．ユーザは生成された UML モデル検証環境を用いて任意の UML ダイアグラムを編集し検証することができる．

3.2　UML モデル検証環境生成手法の提案

EMF [7] は Eclipse Modeling Project が作成したプロダクトの 1 つであり，モデル駆動型アーキテクチャ (Model-Driven Architecture：MDA) [3, 18] をサポートするモデル記述とコード生成のためのフレームワークである．図 7 は EMF の概要である．ユーザが XML スキーマ，注釈付き Java コード，.mdl ファイル，または.uml ファイルで表現されたモデルを EMF に入力することで業務モデル (Platform Independent Model，図 9 では PIM) である Ecore モデルと，実装依存モデル (Platform Specific Model，図 9 では PSM) である EMF モデルを生成する．EMF は MOF に準拠しており，モデル変換に関して OMG の策定したモデル変換規格 QVT [35] に従っている．したがって MOF 準拠した.mdl ファイルや.uml ファイルから Ecore モデル，EMF モデルへの変換が可能である．Ecore モデルを編集することで対応する EMF モデルは自動更新され，EMF モデルからモデルのインスタンス編集環境をコード生成することができる．インスタンス編集環境では Ecore モデルのインスタンスおよびインスタンス間の関連が XMI [36] に基づく木構造で管理されている．またこの環境は

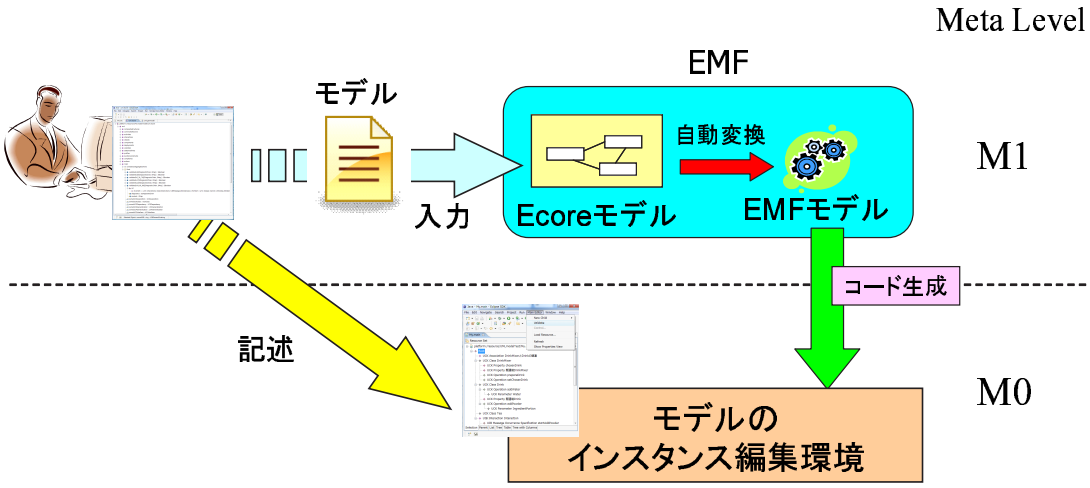

図 7: EMF の概要

木構造に基づくユーザインターフェースを提供している．根は Ecore モデルから選択した 1 クラスのインスタンス，その他の頂点は他のクラスのインスタンス，辺はインスタンス間の関連である．したがって EMF を利用することで，ユーザは Ecore モデルをメタモデルとした XMI 形式に基づく木構造のモデルエディタを作成することができる．

また EMFT_OCL [8] プラグインを利用することで，図 8 のように EMF モデルに付加された OCL 制約式をインスタンスに対して検証することが可能になる．

図 9 の左側は EMF の 3 階層アーキテクチャである．Ecore モデル，EMF モデルは MOF [34] のサブセットである Essential MOF のインスタンスであり，インスタンス編集環境は Ecore モデル，EMF モデルのインスタンスである．

提案手法では図 9 のように，UML メタモデルと Ecore モデル，EMF モデルを対応させた．そうすることにより UML モデル編集環境がコード生成される．また，UML メタモデルと対応する Ecore モデル，EMF モデルに対して OCL 制約式を付加することで，UML モデル編集環境において EMFT_OCL プラグインによる OCL 式を用いた検証が可能になる．図 10 は構築した UML モデル検証環境である．UML メタモデルを入力として EMF に与えると，Ecore モデルが生成され，Ecore モデルから EMF モデルが生成される．Ecore モデルに対して整合性ルールを OCL 式で追加することで，EMF モデルに対しても同期して制約が追加される．最後に EMF モデルからコード生成することで UML モデル検証環境が構築される．ユーザは生成された UML モデル検証環境で各 UML ダイアグラムを記述し，OCL 式を用いた整合性検証を行う．

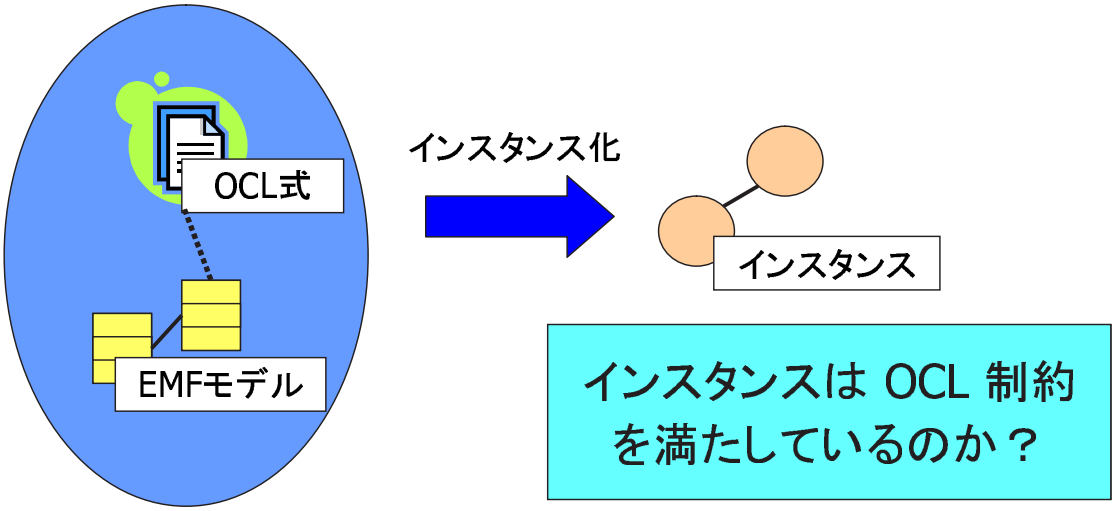

図 8: EMFT_OCL プラグインの概要

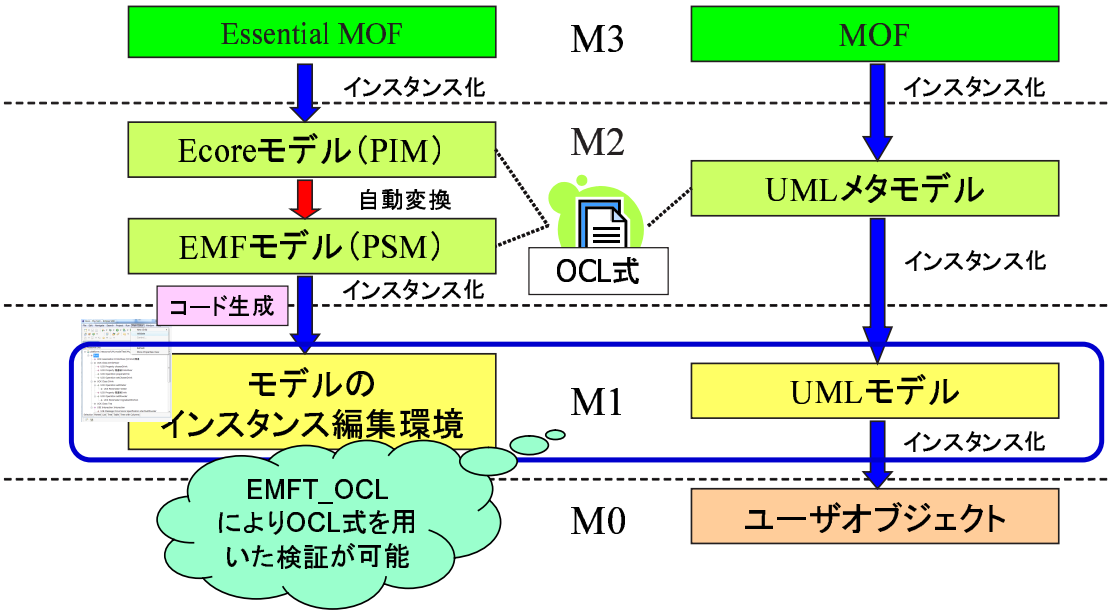

図 9: UML ダイアグラム間検証における EMF と UML の対応関係

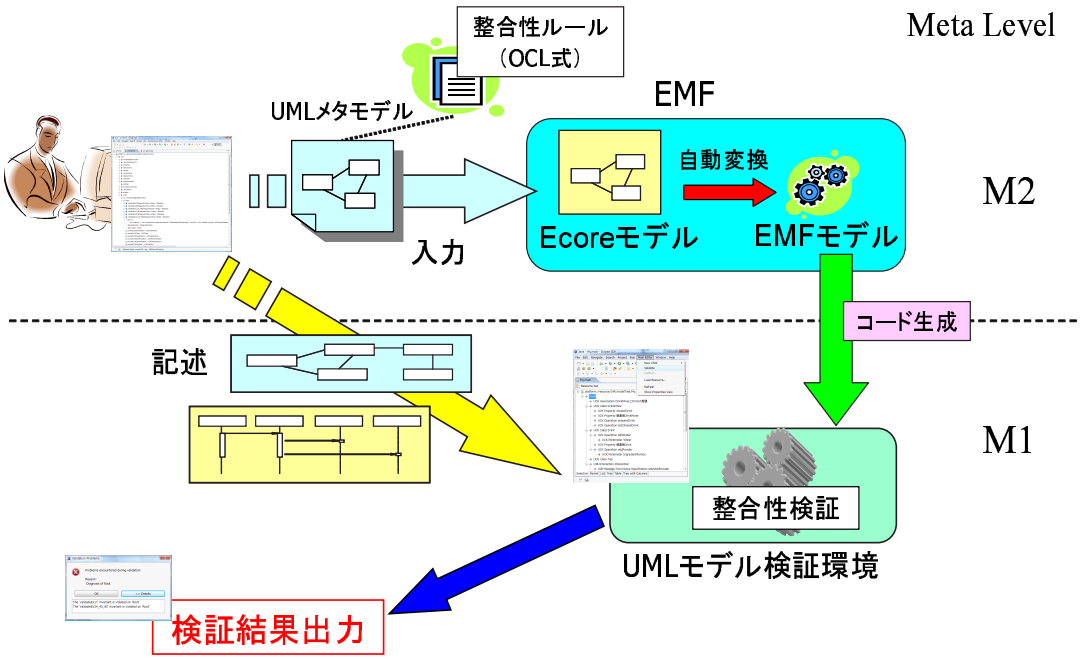

図 10: 提案するシステム

3.3 整合性ルール

UML モデル検証環境では，L. C. Briand らが定義した文献 [4, 5] に記述されている整合性ルールを利用して検証を行う．記述された整合性ルールは同一ダイアグラム内の整合性ルールとダイアグラム間の整合性ルールに大別されるが，本研究ではダイアグラム間の整合性ルールに着目して検証を行う．

3.4 実装

提案する手法を実現するために EMF の `org.eclipse.emf.ecore.presentation` パッケージ内に含まれる Ecore モデルエディタ (`EcoreEditor` クラス) を拡張した．拡張した Ecore モデルエディタにより，OCL 式の編集と，各 UML ダイアグラムに必要な UML メタモデル要素の管理を支援することができる．

3.4.1 検証システム実現の問題点とその解決

インスタンス編集環境は Ecore モデル内で定義されたパッケージ毎に生成され，選択したパッケージ内のクラスのインスタンスとそのインスタンス間の関連を XMI に基づく木構造で管理する．よって，別パッケージ内に存在する関連を持たないクラスのインスタンス同士

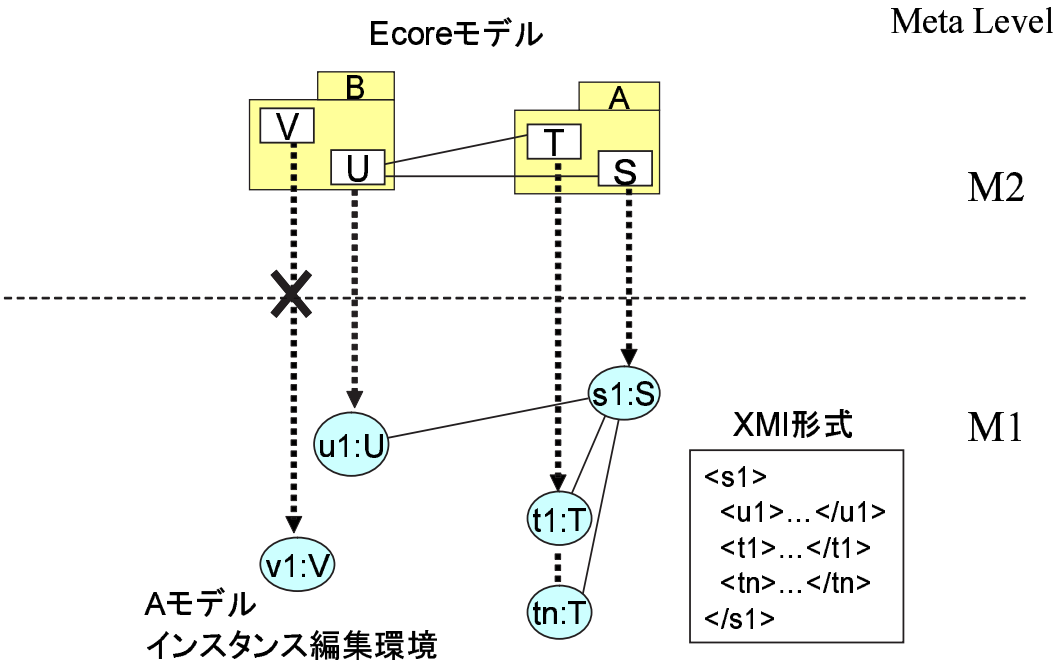

図 11: 検証システム実現の問題点

を同一インスタンス編集環境で管理することはできない．図 11 はこの問題点を端的に表現している．Ecore モデル内に A パッケージと B パッケージが存在する場合，A モデルインスタンス編集環境と B モデルインスタンス編集環境が生成されることになる．A モデルインスタンス編集環境で S クラスのインスタンスを根と選択した場合，S クラスと関連のある T クラスや U クラスのインスタンスは A モデルインスタンス編集環境でインスタンス化可能である．しかし，S クラスと関連を持たない V クラスを A モデルインスタンス編集環境でインスタンス化することはできない．

試しに EMF を用いて UML メタモデルから Ecore モデルを生成したところ，複数のパッケージに分割されており，その Ecore モデルから最終的に 46 個のインスタンス編集環境が生成された．実際の各 UML ダイアグラムを表現するために必要となる UML メタモデルのクラス間には，関連が存在しない場合がある．そのため，この分割されたインスタンス編集環境では適切に UML ダイアグラムを管理することができなかった．

したがって，我々はこの問題を解決するために Ecore モデルを拡張することにした．図 12 で示すように，各 UML ダイアグラムの構築に必要な全クラスと関連を持つクラスを Ecore モデルに追加することにした．必要な全クラスと関連を持つ Root クラスを追加し，Root クラスのインスタンスをルートとすることで，全てのクラスインスタンスを生成することが可

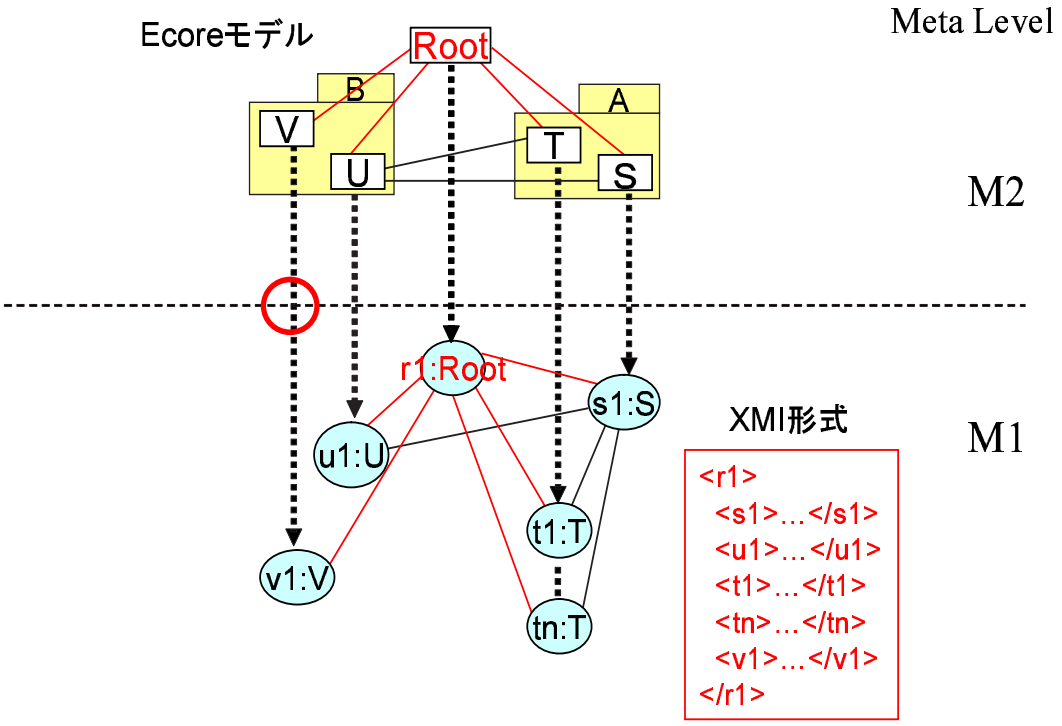

図 12: Root クラスの導入

能になる．

3.4.2 検証システム実現のためのプラグイン

一連の作業を支援するために Ecore モデルエディタを拡張し，提案手法をツール化した．図 13 で示すように，プラグインによって Root クラスを含む main パッケージが Ecore モデルに作成され，Root クラスは各 UML ダイアグラムに必要な UML メタモデル要素を包含する．main パッケージから生成される main モデル検証環境では，Root クラスのインスタンスを根に選択することで，別パッケージにある UML メタモデルを main モデル検証環境においてインスタンス化することが可能になる．

図 14 は実装したツールの UI である．生成する UML ダイアグラムの選択と，UML メタモデルに付加する制約を表現する OCL 式の編集ができる．

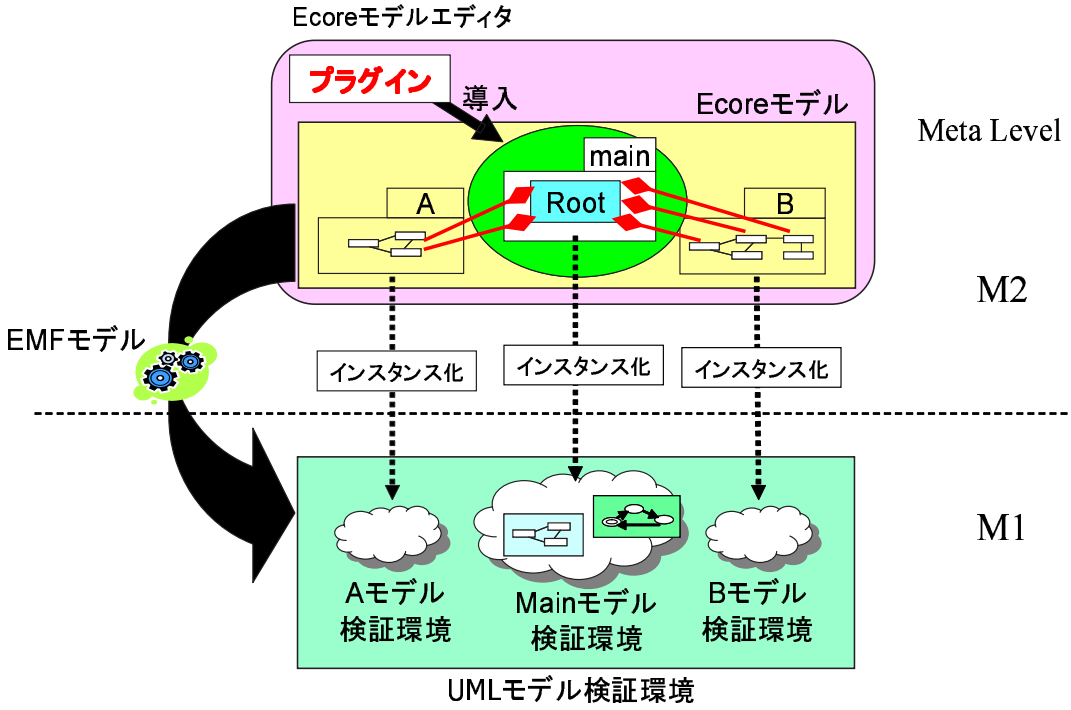

図 13: 提案手法が行うインスタンス化

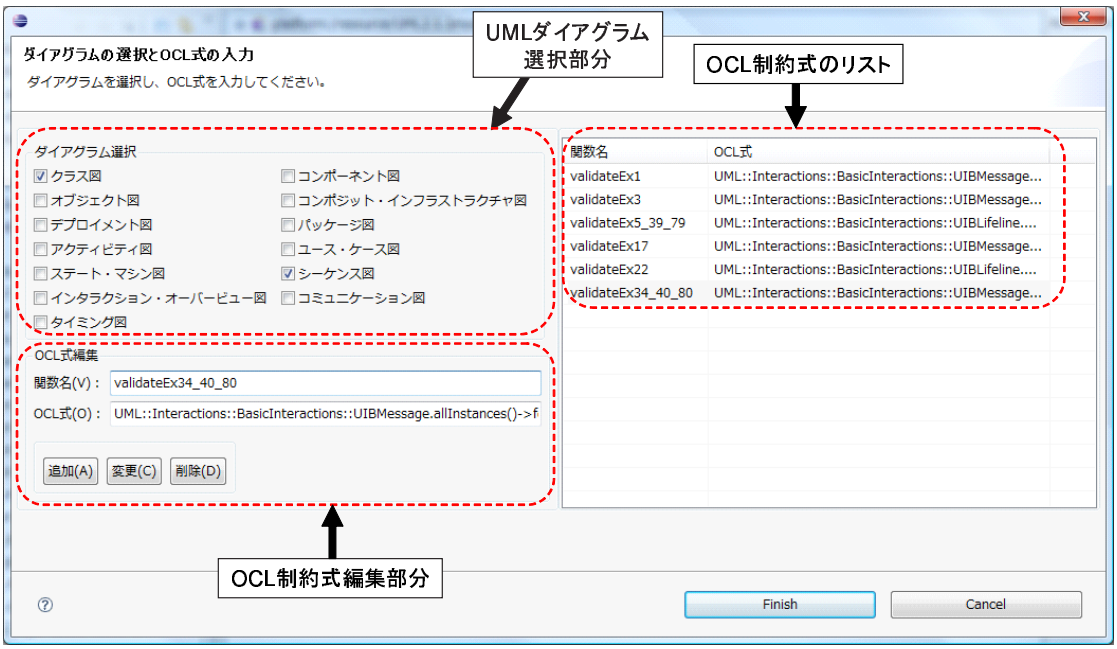

図 14: プラグインの UI

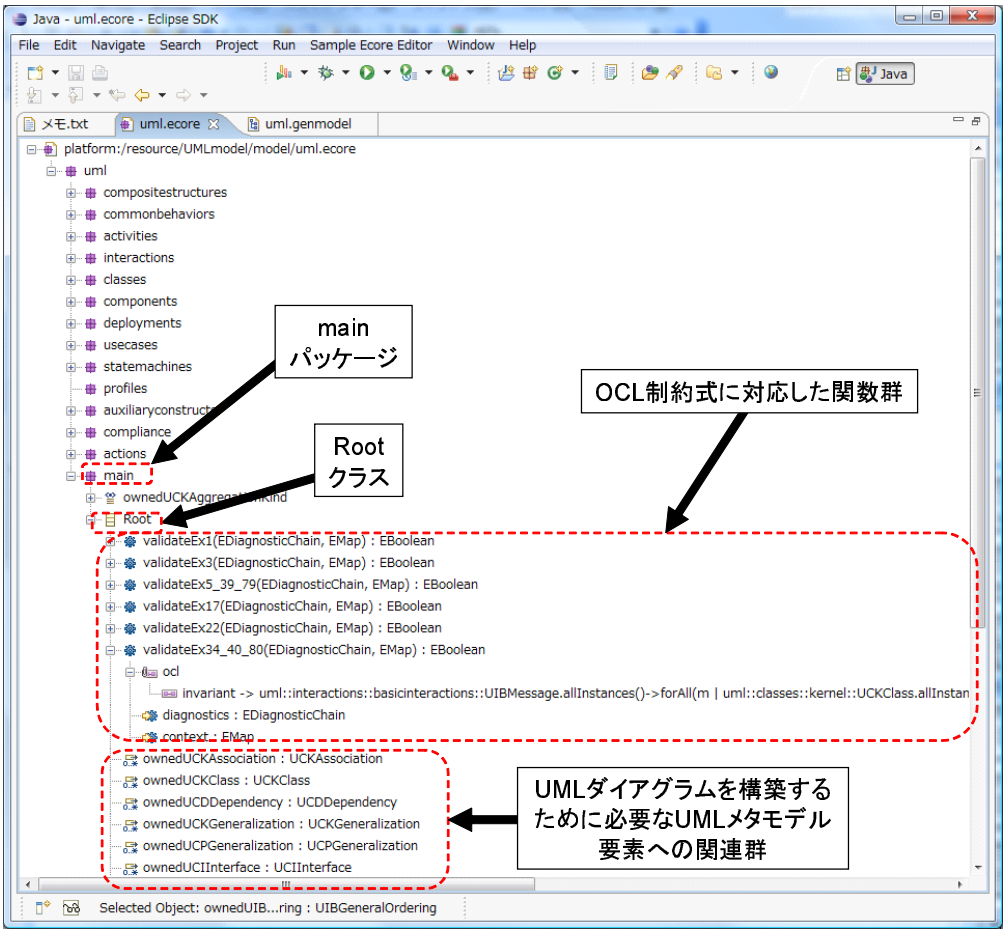

図 15: プラグインの出力

図 15 はツールの出力結果である．Ecore モデルに対して main パッケージと Root クラスが追加されている．Root クラスでは以下の項目が定義される．

- ツールの UI で選択した UML ダイアグラムに対応する UML メタモデル内のモデル要素への包含関係
- ツールの UI で記述した OCL 式とその関数名

これにより図 13 で示している main モデル検証環境では，選択した UML ダイアグラムの表現に必要な UML メタモデルのインスタンスを生成することが可能になる．また，それぞれの UML ダイアグラムを構築するために必要な UML メタモデル内のモデル要素は UML の仕様書 [30, 31, 32, 33] から特定した．

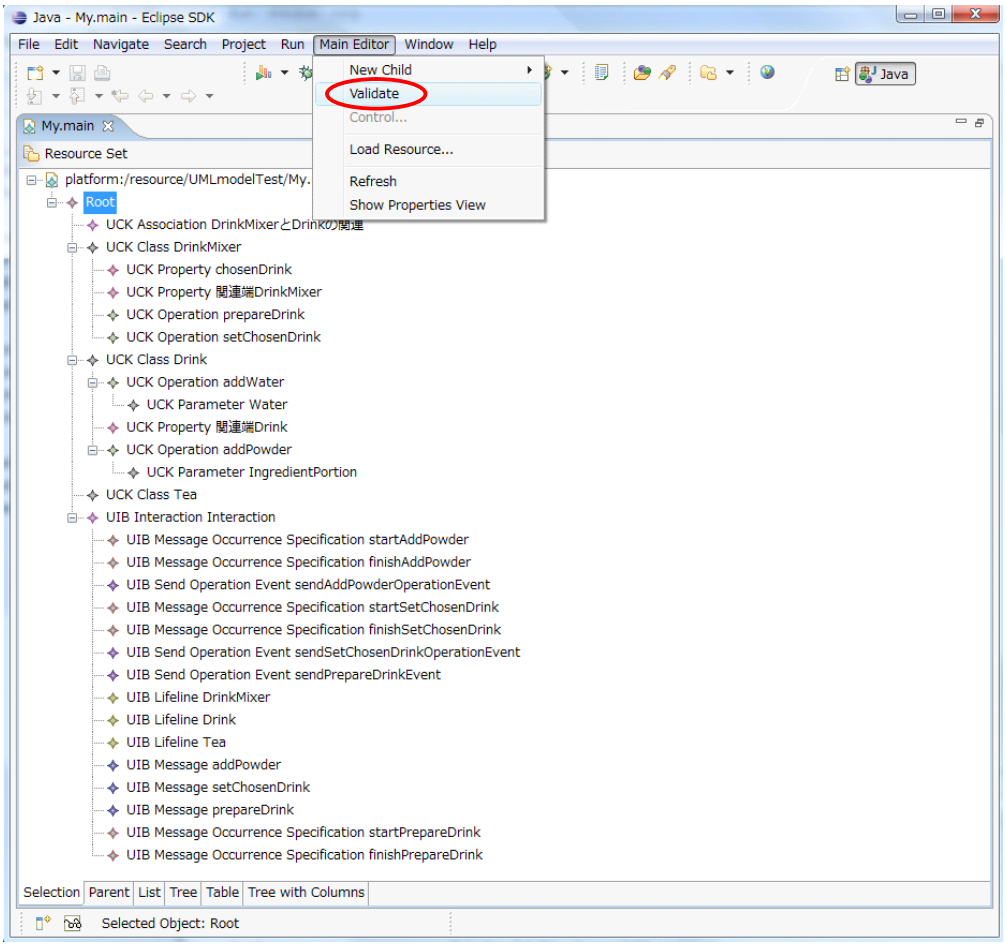

図 16: main モデル編集環境

図 16 は main モデル検証環境である．Root クラスを起点としてツリー構造のエディタで編集を行っていく．Root クラスを選択し validate ボタンを押すことで，Root クラスに定義した OCL 制約式を検証することができる．

検証結果はダイアログで出力される．違反が検出された場合は，図 17 のように追加した OCL 式と対応する関数名が表示される．

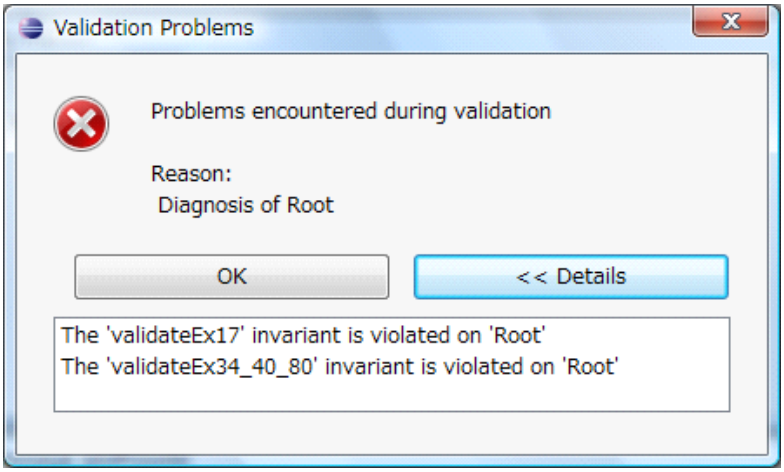

図 17: 検証結果例

4 実験

本章では提案した手法を評価するために行った実験とその結果について述べる．

4.1 実験設定

行った実験は 2 段階に分かれており，それぞれ以下のように設定する．

- 実験対象 1：図 18 のように，ユーザが検証システムに与える入力は，UML2.0 メタモデルとクラス図とシーケンス図間に関する整合性ルールである．また，記述する UML ダイアグラムはクラス図とシーケンス図である．実験の目的は既存研究と同様に UML ダイアグラム間整合性検証が可能かどうかを評価することである．
- 実験対象 2：図 19 のように，ユーザが検証システムに与える入力は，UML2.1.1 メタモデルと実験対象 1 と同じクラス図とシーケンス図間に関する整合性ルールである．また，記述する UML ダイアグラムは実験対象 1 と同じクラス図とシーケンス図である．実験の目的は実験対象 1 と同様に UML ダイアグラム間整合性検証が可能かどうかを評価することと，整合性ルールの OCL 制約式はどのくらい再利用可能であるかを評価することである．

また，実験対象 1 で入力する UML2.0 メタモデルは.mdl ファイルで表現されており，一方，実験対象 2 で入力する UML2.1.1 メタモデルは.uml ファイルで表現されている．これはバージョン変更の際に，より XMI 形式に準拠した.uml ファイルに表現形式が変更されたためである．

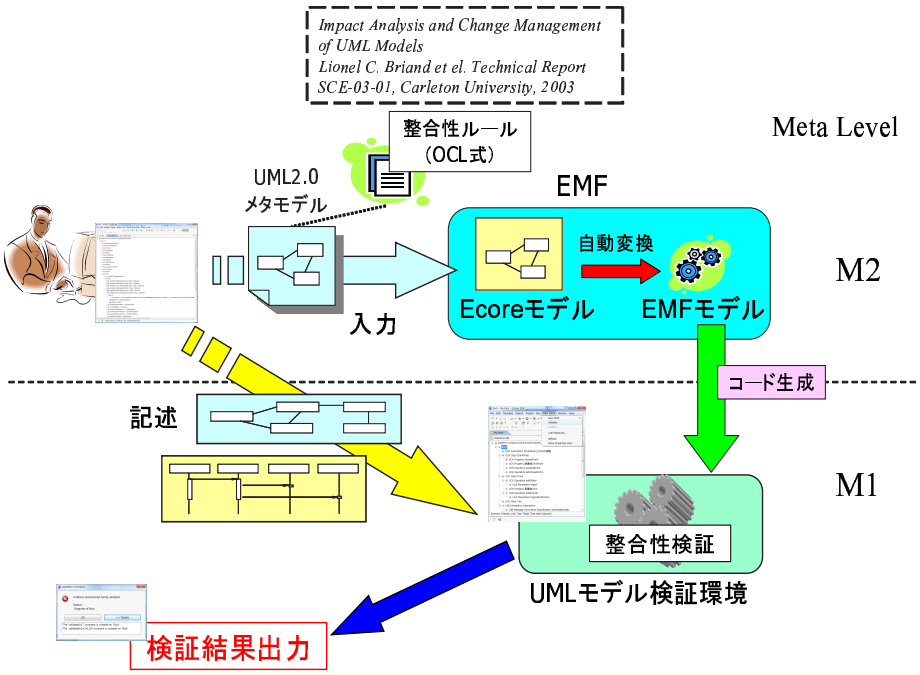

図 18: 実験対象 1 の設定

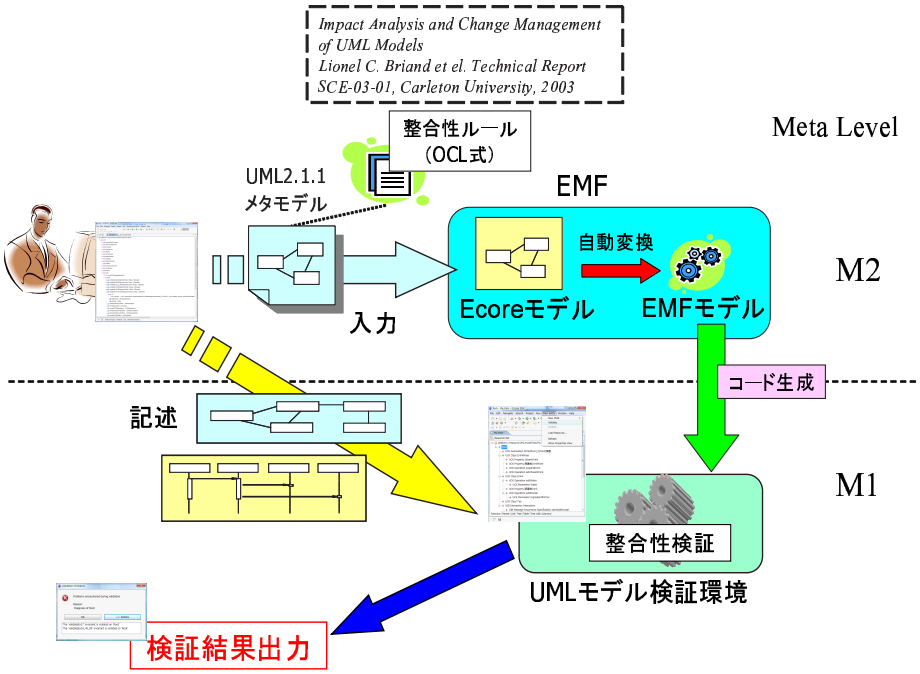

図 19: 実験対象 2 の設定

実験環境は以下の通りである．
OS：Windows Vista Business,
CPU：Intel(R) Core(TM) 2 CPU 1.87GHz,
Memory：2.0GB RAM,
Eclipse Version：3.2.0,
EMF Version：2.2.0

4.2 検証する整合性ルール

整合性ルールは文献 [4, 5] に記述された 115 個の整合性ルールの中からクラス図とシーケンス図間に関する全てのルール 10 個である．10 個のルールは重複する内容を考慮すると実際は 6 個である．以下が検証に用いた整合性ルールである．

1. シーケンス図中において，protected なオペレーションはそのコンテナクラスを継承していないクラスに属するオペレーションからは呼び出されない．
2. シーケンス図中において，private なオペレーションは他のクラスに属するオペレーションからは呼び出されない．
3. シーケンス図中において，オブジェクトはクラス図中のクラスのインスタンスでなければならない．
4. シーケンス図中において，2 つのオブジェクト間の各メッセージには，適切なパスがなければならない．
5. シーケンス図中において，抽象オペレーションは呼び出すことができない．
6. シーケンスメッセージ中で呼ばれる各オペレーションはクラス図で定義されていなければならない．

記述する OCL 式は，実験対象 1 の場合は UML2.0 仕様書から，実験対象 2 の場合は UML2.1.1 仕様書から，上記の整合性ルールに関係する UML メタモデル集合を特定し，次にそのインスタンスに関する不変式で表現する．評価結果は真偽値である．図 20 は整合性ルール (3) の OCL 表記例である．シーケンス図に存在する全てのオブジェクトはクラス図に存在するクラスと同じ名前を持たなければいけないことを示している．

```
uml::interactions::basicinteractions::Lifeline.allInstances()->forAll(l |
    uml::classes::kernel::Class.allInstances()
    ->select(oclIsTypeOf(uml::classes::kernel::Class))->collect(Name)
    ->includesAll(l->collect(Name))
)
```

図 20: 整合性ルールの OCL 表記例

4.3 適用例題

実験対象 1，実験対象 2 で適用するドリンクメーカー (DrinkMaker) はコーヒーや紅茶などの飲料を作る，レストランなどに置かれている機械である [23]．この例題に関するクラス図とシーケンス図を用いて提案手法の適用を試みる．図 21 と図 22 が例題の UML ダイアグラムであり，それぞれクラス図と，カップに注入された紅茶にパウダーを入れるというユースケースをモデル化したシーケンス図の一部である．また，元のクラス図には Tea クラスに root 特性が指定されていたが，UML2.0 と UML2.1.1 ではクラスに root 特性の指定を行わないため省略している．

4.4 適用例題検証例

文献 [21] によると最もよく起こりやすい不整合は，関連の定義されていないクラス間でメッセージのやりとりをしてしまうという不整合である．図 21 の UML ダイアグラムの中にもこれと同じ不整合が埋め込まれている．つまり，例題として挙げた上記の 2 つの UML ダイアグラムには，以下の 2 つの不整合が埋め込まれていることになる．

1. シーケンス図中の DrinkMixer クラスのオブジェクトと，Tea クラスのオブジェクトとの間でやりとりされているメッセージ addPowder が，その受信オブジェクトのクラスであるクラス Tea 中のオペレーションで定義されていない．

2. シーケンス図中で，DrinkMixer クラスのオブジェクトと Tea クラスのオブジェクトとの間でメッセージがやりとりされているが，クラス図中でそれらのクラスの間に関連が引かれていない．

実験対象 1 と実験対象 2 において，それぞれこの例題に対して main モデル検証環境で検証を行うと，整合性ルール (4)，(6) について違反しているという結果を出力できた．次に図 23 のように，Tea クラスが Drink クラスを継承し，DrinkMixer クラスと Drink クラス間に関連を追加するという修正を行ったところ違反は出力されなかった．したがって提案手法は

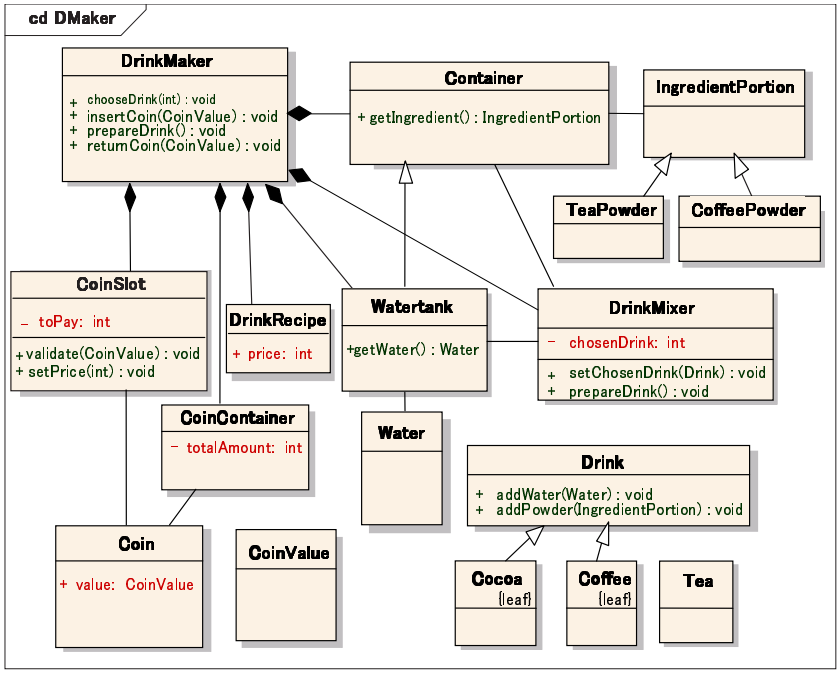

図 21: 例題クラス図

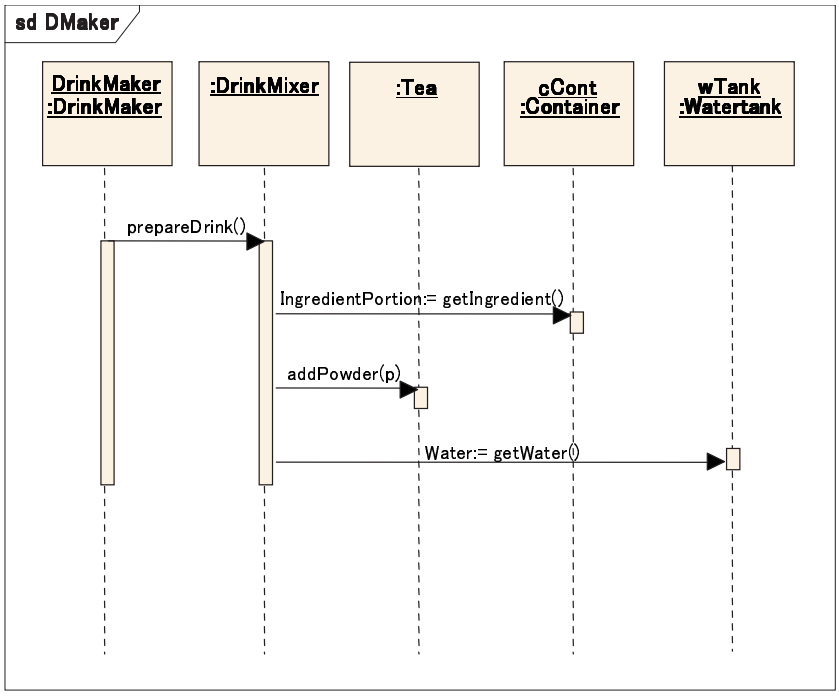

図 22: 例題シーケンス図

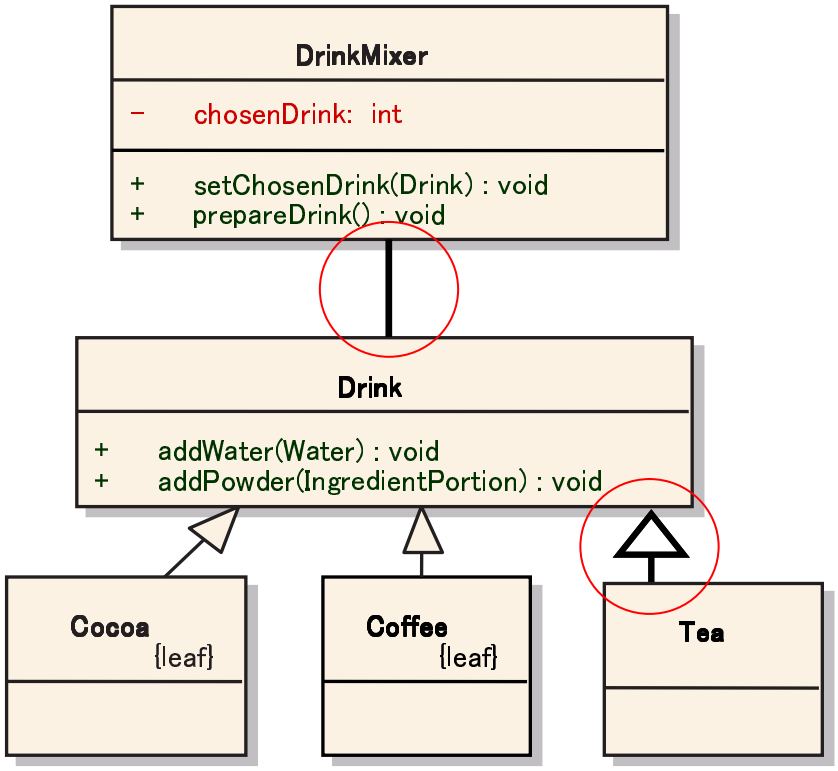

図 23: 修正したクラス図

整合性ルール (4)，(6) に関する整合性検証に関しては正しく機能していると考えられる．なお検証時間は約 0.2 秒であった．

4.5 実験結果

上記の例題を用いて，整合性ルール (4)，(6) 以外の 4 つの整合性ルールに対しても同様に実験を行った．全ての整合性ルールについて，違反する設計をした場合は正しく違反を出力するかどうかを確認し，違反を修正した場合は違反を出力しないかどうかを確認した．図 24 は実験結果である．左の表が実験対象 1 の結果で，右の表が実験対象 2 の結果である．表の項目はそれぞれ，適用した整合性ルールとその整合性ルールを表現する OCL 制約式とその検証結果である．整合性ルールや検証式に記述してある番号はそれぞれ文献 [4] 中の制約番号を引用したものであり，ルール 1 は整合性ルール (1)，ルール 3 は整合性ルール (2)，ルール 5，39，79 は整合性ルール (3)，ルール 17 は整合性ルール (4)，ルール 22 は整合性ルール (5)，ルール 34，40，80 は整合性ルール (6) に該当する．

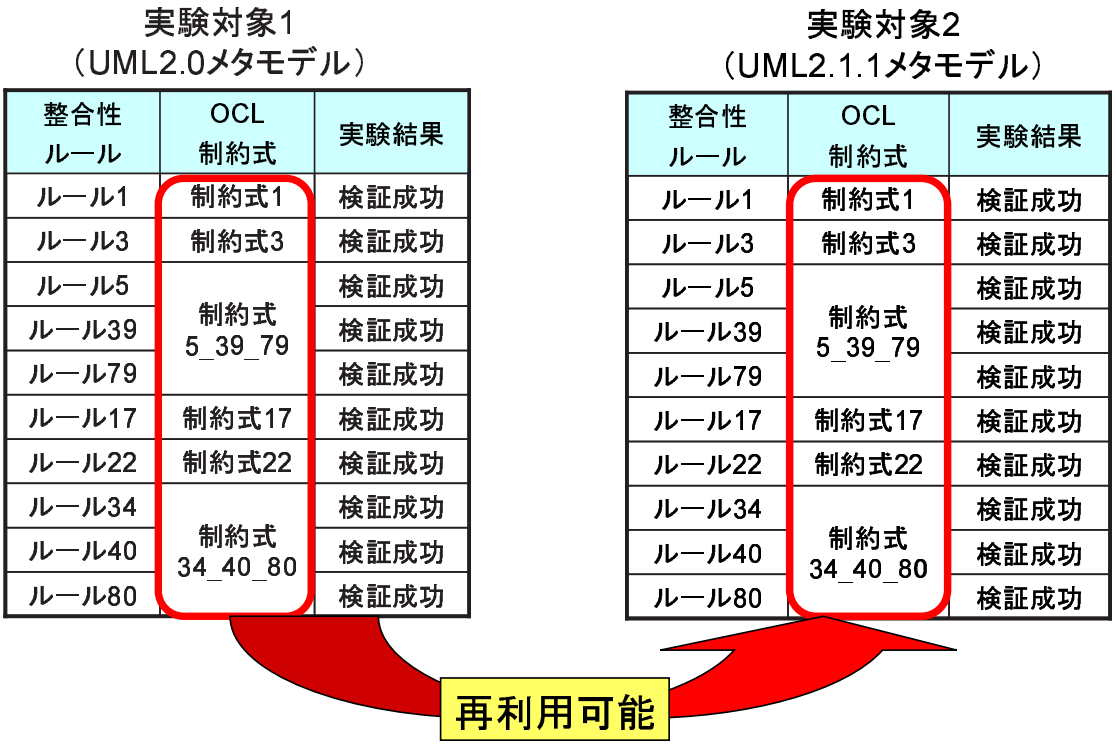

図 24: 実験結果

実験の結果，実験対象 1 と実験対象 2 において全ての整合性ルールの検証が成功した．検証時間も適用例題の結果とさほど変化しなかった．また，実験対象 1 で用いた OCL 制約式を全て実験対象 2 で再利用することができた．

5　考察と評価

適用実験により，OCL 式で整合性ルールを記述することができれば，提案手法によりクラス図とシーケンス図間の整合性を検証することができることが確認できた．DrinkMaker は文献 [4] で述べられている全ての整合性ルールを含んでいるため，他のクラス図，シーケンス図間の整合性検証にも提案手法を適用可能であると考えられる．本研究と同様に文献 [41] も DrinkMaker [23] を用いて整合性検証法の評価を行っている．本研究と文献 [41] の実験結果を比較すると，本研究でも同等の検証を行えることがわかる．また，UML メタモデルに制約を与えて整合性検証を行っている既存研究に比べて UML メタモデルの制限なしに UML モデル検証環境を構築できることは大きな利点である．

また，UML2.0 メタモデルから UML2.1.1 メタモデルへの変更の場合は，OCL 制約式をそのまま再利用して整合性を検証することができることが分かった．このバージョン間の変更ではクラス図とシーケンス図間に関係するメタモデル構造がほとんど変更されなかったためだと考えられる．実際に関係するメタモデル構造がほとんど変更されなかったかどうかを確認するために，UML2.0[31] の仕様書と UML2.1.1[33] の仕様書を比較した．クラス図の生成に必要なメタモデル要素は UML メタモデルにおいて Classes パッケージ内に存在する．UML2.0 の Classes パッケージ内に存在するメタモデル要素と UML2.1.1 の Classes パッケージ内に存在するメタモデル要素は同一のものであり，要素数はそれぞれ 55 個であった．シーケンス図の生成に必要なメタモデル要素は UML メタモデルにおいて Interactions パッケージ内に存在する．UML2.0 の Interactions パッケージ内に存在するメタモデル要素と UML2.1.1 の Interactions パッケージ内に存在するメタモデル要素を比較したところ，2 クラスだけ UML2.1.1 の Interactions パッケージの方が多くメタモデル要素を含み，それぞれ 29 個と 31 個であった．このことから UML2.0 から UML2.1.1 のバージョン変更では若干シーケンス図の表現方法が増加しているが，OCL 検証式を変更する必要があるほどの大きな変更ではなかったことが推測される．

そして，別の種類の UML ダイアグラム間に対しても提案手法により同様に検証可能だと考えられる．クラス図とシーケンス図間の制約と同様に OCL 制約式を記述可能であれば，クラス図とステートマシン図などの整合性ルールも検証可能だと考えられる．

短所として，検証者は UML メタモデル構造と OCL 記述文法を深く理解する必要があることがあげられる．UML ダイアグラムのモデル記述環境は UML メタモデルを入力として与えることでツールと EMF が自動で生成するが，OCL 式に関しては UML メタモデルのバージョン毎に検証者が書き直さなければならない可能性がある．制約を与える OCL 式は UML メタモデルのモデル要素にアクセスするので，モデル構造が変化すると OCL 式自体も変更する必要があるからである．また，文献 [4] の中には OCL 式を適用できない整合性

ルールも存在する．オペレーション内部の情報が必要な制約や，別の OCL 式の内容を参照する必要がある制約は適用が難しい，他にも大量の UML メタモデルにアクセスする必要がある制約を OCL 式で記述するのは現実的ではない．OCL 式で表現するのが適切な制約のみを実装するのが望ましい．上記の問題を解決するためには，EMFT_OCL プラグインを拡張し，独自の述語を用意するか，もしくは文献 [39] のように OCL の表現能力を拡張する手法が有効だと考える．

6　あとがき

本論文では，UML メタモデルと UML ダイアグラム間整合性ルールを記述した OCL 式を用いて UML モデル検証環境を構築する手法を提案し，適用実験を行った．提案手法では EMF に基づいて，UML メタモデルと OCL 制約式を入力として与えることで Ecore モデルを生成し，Ecore モデルから UML モデル検証環境のソースコードを自動生成する．また支援ツールを作成し，作業の効率化を行った．UML2.0 及び UML2.1.1 のメタモデルに基づく飲料生成システムを作成し，それぞれに対して適用した．その結果，提案手法による UML ダイアグラム間整合性検証が可能であることが確認でき，手法の有効性が示された．

今後の課題は，より多くのバージョンやドメインに特化した UML メタモデルを用いた評価実験である．また，DrinkMaker 以外の例題への適用も行いたい．他にも，各工程間における同一種類の UML ダイアグラム間整合性検証や，UML ダイアグラム間の不整合修正支援機能を追加したい．

謝辞

本研究の全過程を通して，常に適切な御指導，御助言を賜りました 大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻 井上 克郎 教授に心より深く感謝致します.

本論文を作成するにあたり，常に適切な御指導，御助言を賜りました 大阪大学 大学院情報科学研究科コンピュータサイエンス専攻 楠本 真二 教授に心から感謝致します.

本論文を作成するにあたり，適切な御指導，御助言を賜りました 大阪大学大学院情報科学研究科 コンピュータサイエンス専攻 岡野 浩三 准教授に心から感謝致します.

本論文を作成するにあたり，適切な御指導，御助言を賜りました 大阪大学大学院情報科学研究科 コンピュータサイエンス専攻 松下 誠 准教授に心から感謝致します.

本研究を通して，逐次適切な御指導，御助言を頂きました 大阪大学大学院情報科学研究科 コンピュータサイエンス専攻 石尾 隆 助教に深く感謝致します.

本研究において，様々な御協力を頂きました 株式会社 NTT データ 我妻 智之 氏，ならびに梅村 晃広 氏に深く感謝します.

最後に，その他様々な御指導，御助言等を頂いた 大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻 井上研究室の皆様に深く感謝いたします.

参考文献

[1] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche. The Impact of UML Documentation on Software Maintenance: An Experimental Evaluation. *IEEE Transactions on Software Engineering*, Vol. 32, No. 6, pp. 365–381, 2006.

[2] J. Bezivin. Model Driven Engineering: Principles, Scope, Deployment and Applicability. In *Generative and Transformational Techniques in Software Engineering (GTTSE 2005), Tutorials*, pp. 1–33, 2005.

[3] G. Booch, A. Brown, S. Iyengar, J. Rumbaugh, and B. Selic. The IBM MDA Manifesto. *The MDA Journal*, 2004.

[4] L. C. Briand, Y. Labiche, and L. O'Sullivan. Impact Analysis and Change Management of UML Models. Technical Report SCE-03-01, Carleton University, 2003. `http://squall.sce.carleton.ca/pubs/tech_report/TR_SCE-03-01.pdf`.

[5] L. C. Briand, Y. Labiche, and L. O'Sullivan. Impact Analysis and Change Management of UML Models. In *In Proceeding of the 19th International Conference on Software Maintenance (ICSM 2003)*, pp. 256–265, 2003.

[6] D. Chiorean, M. Pasca, A. Carcu, C. Botiza, and S. Moldovan. *Ensuring UML models consistency using the OCL Environment*, pp. 99–110. Electr. Notes Theor. Comput. Sci., 2004.

[7] Eclipse Foundation. Eclipse Modeling Framework. `http://www.eclipse.org/modeling/emf/`.

[8] Eclipse Foundation. Model Development Tools. `http://www.eclipse.org/modeling/mdt/`.

[9] A. Egyed. Instant consistency checking for the UML. In *In Proceeding of the 28th International Conference on Software Engineering (ICSE 2006)*, pp. 381–390, 2006.

[10] A. Egyed. Fixing Inconsistencies in UML Design Models. In *In Proceeding of the 29th International Conference on Software Engineering (ICSE 2007)*, pp. 292–301, 2007.

[11] A. Egyed. UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models. In *In Proceeding of the 29th International Conference on Software Engineering (ICSE 2007)*, 2007.

[12] M. Elaasar and L. C. Briand. An Overview of UML Consistency Management. Technical Report SCE-04-18, Carleton University, 2004. `http://squall.sce.carleton.ca/pubs/tech_report/TR-SCE-04-18.pdf`.

[13] M. Elaasar and L. C. Briand. Experimentally investigating Effects of Defects in UML Models. CS-Report 05-07, Technische Universiteit Eindhoven, 2005. `http://alexandria.tue.nl/extra1/wskrap/publichtml/200507.pdf`.

[14] G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. `http://citeseer.ist.psu.edu/479796.html`.

[15] B. Hnatkowska and A. Walkowiak. Consistency Checking of USDP Models. In *In Proceeding of the 3rd International Workshop, Consistency Problems in UML-based Software Development III*, pp. 59–70, 2004.

[16] IBM. Rational Rose. `http://www-306.ibm.com/software/rational/`.

[17] I. Ivkovic and K. Kontogiannis. Tracing Evolution Changes of Software Artifacts through Model Synchronization. In *In Proceeding of the 20th International Conference on Software Maintenance (ICSM 2004)*, pp. 252–261, 2004.

[18] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[19] C. F. J. Lange and M. R. V. Chaudron. Effects of defects in UML models: an experimental investigation. In *In Proceeding of the 28th International Conference on Software Engineering (ICSE 2006)*, pp. 401–411, 2006.

[20] C. F. J. Lange, M. R. V. Chaudron, and J. Muskens. In practice: UML software architecture and design description. *IEEE Software*, Vol. 23, No. 2, pp. 40–46, 2006.

[21] C. Lange, M. R. V. Chaudron, J. Muskens, L. J. Somers, and H. M. Dortmans. An Empirical Investigation in Quantifying Inconsistency and Incompleteness of UML Designs. In *In Proceeding of the Workshop on Consistency Problems in UML-based Software Development II*, pp. 26–34, 2003.

[22] Z. Liu, H. Jifeng, X. Li, and Y. Chen. Consistency and Refinement of UML Models. In *In Proceeding of the 3rd International Workshop, Consistency Problems in UML-based Software Development III*, pp. 23–40, 2004.

[23] K. Ludwik and S. Miroslaw. Inconsistencies in Student Designs. In *In Proceeding of the Workshop on Consistency Problems in UML-based Software Development II*, pp. 9–17, 2003.

[24] H. Malgouyres and G. Motet. A UML model consistency verification approach based on meta-modeling formalization. In *In Proceedings of the 2006 ACM symposium on Applied computing (SAC 2006)*, pp. 1804–1809, 2006.

[25] T. Mens, R. V. D. Straeten, and J. Simmonds. Maintaining Consistency between UML Models with Description Logic Tools. In *In Proceeding of the 4th International Workshop on Object-Oriented Re-engineering (WOOR 2003)*, 2003.

[26] T. Mens and T. Tourwe. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, Vol. 30, No. 2, pp. 126–139, 2004.

[27] L. Murta, H. Oliveira, C. Dantas, L. G. Lopes, and C. Werner. Odyssey-SCM: An integrated software configuration management infrastructure for UML models. *Sci. Comput. Program.*, Vol. 65, No. 3, pp. 249–274, 2007.

[28] Object Management Group. `http://www.omg.org/`.

[29] Object Management Group. OMG Modeling Specifications. `http://www.omg.org/technology/documents/vault.htm#modeling`.

[30] Object Management Group. UML2.0 Infrastracture. `http://www.omg.org/docs/formal/05-07-05.pdf`.

[31] Object Management Group. UML2.0 Superstracture. `http://www.omg.org/docs/formal/05-07-04.pdf`.

[32] Object Management Group. UML2.1.1 Infrastracture. `http://www.omg.org/docs/formal/07-02-06.pdf`.

[33] Object Management Group. UML2.1.1 Superstracture. `http://www.omg.org/docs/formal/07-02-05.pdf`.

[34] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification, 2004. http://www.omg.org/docs/ptc/03-10-04.pdf.

[35] Object Management Group. MOF QVT Final Adopted Specification, 2005. http://www.omg.org/docs/ptc/05-11-01.pdf.

[36] Object Management Group. UML 2.0 Diagram Interchange Specification, 2005. http://www.omg.org/docs/ptc/05-06-04.pdf.

[37] Object Management Group. OCL 2.0 Specification, 2006. http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf.

[38] O. Pilskalns, A. Andrews, A. Knight, S. Ghosh, and R. France. Testing UML designs. *Inf. Softw. Technol.*, Vol. 49, No. 8, pp. 892–912, 2007.

[39] M. Richters and M. Gogolla. A Metamodel for OCL. In *In Proceeding of 2nd International Conference on the Unified Modeling Language (UML 1999)*, pp. 156–171, 1999.

[40] M. Richters and M. Gogolla. Validating UML models and OCL constraints. In *In Proceeding of 3rd International Conference on the Unified Modeling Language (UML 2000)*, pp. 265–277, 2000.

[41] 佐々木亨, 岡野浩三, 楠本真二. 制約指向に基づいた UML モデルの不整合検出・解消手法の提案. 電子情報通信学会論文誌 D, Vol. J90-D, No. 4, pp. 1005–1013, 2007.

[42] P. Selonen and J. Xu. Validating UML models against architectural profiles. In *In Proceedings of the 9th European software engineering conference (ESEC 2003)*, pp. 58–67, 2003.

[43] W. Shen, Y. Lu, and W. Liong Low. Extending the UML Metamodel to Support Software Refinement. In *In Proceeding of the Workshop on Consistency Problems in UML-based Software Development II*, pp. 35–42, 2003.

[44] R. V. D. Straeten. Formalizing Behaviour Preserving Dependencies in UML. In *In Proceeding of the 3rd International Workshop, Consistency Problems in UML-based Software Development III*, pp. 71–82, 2004.

[45] 山本修一郎. UML の基礎と応用, 2002. http://www.bcm.co.jp/site/2002/uml/uml15.htm.

付録

1. シーケンス図中において，protected なオペレーションはそのコンテナクラスを継承していないクラスに属するオペレーションからは呼び出されない.

```
1 UML::Interactions::BasicInteractions::Message.allInstances()->forAll(m |
2   if
3      not m->collect(ReceiveEvent)->collect(Name)->includesAll(
4              m->collect(SendEvent)->collect(Name)
5          )
6      then
7          if
8             not UML::Classes::Kernel::Class.allInstances()
9             ->select(oclIsTypeOf(UML::Classes::Kernel::Class))->select(c1 |
10                UML::Interactions::BasicInteractions::Lifeline.allInstances()->select(l |
11                   l->collect(CoveredBy)->collect(Name)->includesAll(
12                      m->collect(ReceiveEvent)->collect(Name)
13                   )
14                )->collect(Name)->includesAll(c1->collect(Name))
15             )->collect(c2 | c2->union(c2->collect(SuperClass)))->includesAll(
16                UML::Classes::Kernel::Class.allInstances()
17                ->select(oclIsTypeOf(UML::Classes::Kernel::Class))->select(c3 |
18                   UML::Interactions::BasicInteractions::Lifeline.allInstances()->select(l |
19                      l->collect(CoveredBy)->collect(Name)->includesAll(
20                         m->collect(SendEvent)->collect(Name)
21                      )
22                   )->collect(Name)->includesAll(c3->collect(Name))
23                )
24             )
25             and
26             not UML::Classes::Kernel::Class.allInstances()
27             ->select(oclIsTypeOf(UML::Classes::Kernel::Class))->select(c1 |
28                UML::Interactions::BasicInteractions::Lifeline.allInstances()->select(l |
29                   l->collect(CoveredBy)->collect(Name)->includesAll(
30                      m->collect(SendEvent)->collect(Name)
31                   )
32                )->collect(Name)->includesAll(c1->collect(Name))
33             )->collect(c2 | c2->union(c2->collect(SuperClass)))->includesAll(
34                UML::Classes::Kernel::Class.allInstances()
35                ->select(oclIsTypeOf(UML::Classes::Kernel::Class))->select(c3 |
36                   UML::Interactions::BasicInteractions::Lifeline.allInstances()->select(l |
37                      l->collect(CoveredBy)->collect(Name)->includesAll(
38                         m->collect(ReceiveEvent)->collect(Name)
39                      )
40                   )->collect(Name)->includesAll(c3->collect(Name))
41                )
42             )
43             then
```

```
44                 UML::Classes::Kernel::Class.allInstances()
45                 ->select(oclIsTypeOf(UML::Classes::Kernel::Class))->select(c1 |
46                    UML::Interactions::BasicInteractions::Lifeline.allInstances()->select(l |
47                       l->collect(CoveredBy)->collect(Name)->includesAll(
48                          m->collect(ReceiveEvent)->collect(Name)
49                       )
50                    )->collect(Name)->includesAll(c1->collect(Name))
51                 )->collect(c2 | c2->union(c2->collect(SuperClass)))->collect(OwnedOperation)
52                 ->select(o1 | o1->collect(Name)->includesAll(m->collect(Name)))
53                 ->forAll(o2 |
54                    o2->notEmpty()
55                    and
56                    not o2->collect(Visibility)
57                        ->includes(UML::Classes::Kernel::VisibilityKind::Protected)
58                 )
59              else true
60           endif
61       else true
62    endif
63 )
```

このOCL式はシーケンス図中の各メッセージに対する制約を表現している (1〜63行目)．メッセージのやり取りを行う各オブジェクトどうしが同じクラスではなく (3〜5行目)，且つ互いに継承関係が存在しない (8〜42行目) 場合は，受け取り側のオブジェクトに相当するクラスとその親クラスのいずれかがメッセージと同じ関数名を持ち (44〜54行目)，且つその関数のVisibility属性はProtectedであってはならない (56〜58行目) という制約を記述している．このように記述することにより，メッセージに相当する関数がProtectedであり，且つメッセージのやり取りを行っている各オブジェクトに相当するクラス間で継承関係が無い場合は，falseを返すようになる．

2. シーケンス図中において，private なオペレーションは他のクラスに属するオペレーションからは呼び出されない．

```
1 UML::Interactions::BasicInteractions::Message.allInstances()->forAll(m |
2    if
3       not m->collect(ReceiveEvent)->collect(Name)->includesAll(
4               m->collect(SendEvent)->collect(Name)
5          )
6      then
7          UML::Classes::Kernel::Class.allInstances()
8          ->select(oclIsTypeOf(UML::Classes::Kernel::Class))->select(c1 |
9             UML::Interactions::BasicInteractions::Lifeline.allInstances()->select(l |
10                l->collect(CoveredBy)->collect(Name)->includesAll(
11                   m->collect(ReceiveEvent)->collect(Name)
12                )
13             )->collect(Name)->includesAll(c1->collect(Name))
14          )->collect(c2 | c2->union(c2->collect(SuperClass)))->collect(OwnedOperation)
15          ->select(o1 | o1->collect(Name)->includesAll(m->collect(Name)))
16          ->forAll(o2 |
17                   o2->notEmpty()
18                   and
19                   not o2->collect(Visibility)
20                       ->includes(UML::Classes::Kernel::VisibilityKind::Private)
21          )
22      else true
23   endif
24 )
```

この OCL 式はシーケンス図中の各メッセージに対する制約を表現している (1〜24 行目)．メッセージのやり取りを行う各オブジェクトどうしが同じクラスではない (3〜5 行目) 場合，受け取り側のオブジェクトに相当するクラスとその親クラスのいずれかがメッセージと同じ関数名を持ち (7〜17 行目)，且つその関数の Visibility 属性は Private であってはならない (19〜21 行目) という制約を記述している．このように記述することにより，メッセージに相当する関数が Private であり，且つメッセージのやり取りを行っている各オブジェクトが同じクラスではない場合は，false を返すようになる．

3. シーケンス図中において，オブジェクトはクラス図中のクラスのインスタンスでなければならない．

```
1 UML::Interactions::BasicInteractions::Lifeline.allInstances()->forAll(l |
2   UML::Classes::Kernel::Class.allInstances()
3   ->select(oclIsTypeOf(UML::Classes::Kernel::Class))->collect(Name)
4   ->includesAll(l->collect(Name))
5 )
```

この OCL 式はシーケンス図中でメッセージのやり取りを行う各オブジェクトに対する制約を表現している (1〜5 行目)．各オブジェクトに相当するクラスがクラス図中に全て存在する (2〜4 行目) 場合は true を返す．

4. シーケンス図中において，2 つのオブジェクト間の各メッセージには，適切なパスがなければならない．

```
1 UML::Interactions::BasicInteractions::Message.allInstances()->forAll(m |
2   UML::Classes::Kernel::Class.allInstances()
3   ->select(oclIsTypeOf(UML::Classes::Kernel::Class))->select(c1 |
4      UML::Interactions::BasicInteractions::Lifeline.allInstances()->select(l1 |
5         l1->collect(CoveredBy)->collect(Name)->includesAll(
6            m->collect(ReceiveEvent)->collect(Name)
7         )
8      )->collect(Name)->includesAll(c1->collect(Name))
9   )->collect(c2 | c2->union(c2->collect(SuperClass)))->collect(OwnedOperation)
10   ->collect(Name)->includesAll(m->collect(Name))
11   and
12   UML::Classes::Kernel::Class.allInstances()
13   ->select(oclIsTypeOf(UML::Classes::Kernel::Class))->select(c3 |
14      UML::Interactions::BasicInteractions::Lifeline.allInstances()->select(l2 |
15         l2->collect(CoveredBy)->collect(Name)->includesAll(
16            m->collect(SendEvent)->collect(Name)
17         )
18      )->collect(Name)->includesAll(c3->collect(Name))
19   )->collect(c4 | c4->union(c4->collect(SuperClass)))->collect(OwnedAttribute)
20   ->collect(Association)->exists(a |
21      a->notEmpty()
22      and
23      UML::Classes::Kernel::Class.allInstances()
24      ->select(oclIsTypeOf(UML::Classes::Kernel::Class))->select(c5 |
25         UML::Interactions::BasicInteractions::Lifeline.allInstances()->select(l3 |
26            l3->collect(CoveredBy)->collect(Name)->includesAll(
27               m->collect(ReceiveEvent)->collect(Name)
28            )
29         )->collect(Name)->includesAll(c5->collect(Name))
30      )->collect(c6 | c6->union(c6->collect(SuperClass)))->collect(OwnedAttribute)
31      ->collect(Association)->includes(a)
32   )
33 )
```

この OCL 式はシーケンス図中の各メッセージに対する制約を表現している (1〜33 行目)．メッセージの受け取り側のオブジェクトに相当するクラスとその親クラスのいずれかがメッセージと同じ関数名を持ち (2〜10 行目)，且つメッセージの送り側のオブジェクトに相当するクラスとその親クラスの Association 集合 (12〜20 行目) の中には，受け取り側のオブジェクトに相当するクラスとその親クラスの Association 集合と同じものが 1 つ以上存在している (21〜32 行目) 場合は true を返す．このように記述することにより，メッセージのやり取りを行う各オブジェクトどうしには関連が存在しなければならないことを表現している．

5. シーケンス図中において，抽象オペレーションは呼び出すことができない.

```
1 UML::Interactions::BasicInteractions::Lifeline.allInstances()->forAll(l |
2   UML::Classes::Kernel::Class.allInstances()
3   ->select(oclIsTypeOf(UML::Classes::Kernel::Class))->select(c |
4      l->collect(Name)->includesAll(c->collect(Name))
5   )->collect(IsAbstract)->includes('false')
6 )
```

この OCL 式はシーケンス図中でメッセージのやり取りを行う各オブジェクトに対する制約を表現している (1〜6 行目)．各オブジェクトに相当するクラスがクラス図中に全て存在し (2〜4 行目)，且つそのクラスの IsAbstract 属性が false である (5 行目) 場合は true を返す.

6. シーケンスメッセージ中で呼ばれる各オペレーションはクラス図で定義されていなければならない.

```
1 UML::Interactions::BasicInteractions::Message.allInstances()->forAll(m |
2   UML::Classes::Kernel::Class.allInstances()
3   ->select(oclIsTypeOf(UML::Classes::Kernel::Class))->select(c1 |
4      UML::Interactions::BasicInteractions::Lifeline.allInstances()->select(l |
5         l->collect(CoveredBy)->collect(Name)->includesAll(
6            m->collect(ReceiveEvent)->collect(Name)
7         )
8      )->collect(Name)->includesAll(c1->collect(Name))
9   )->collect(c2 | c2->union(c2->collect(SuperClass)))->collect(OwnedOperation)
10   ->collect(Name)->includesAll(m->collect(Name))
11 )
```

この OCL 式はシーケンス図中の各メッセージに対する制約を表現している (1〜11 行目). メッセージの受け取り側のオブジェクトに相当するクラスとその親クラスのいずれかがメッセージと同じ関数名を持つ (2〜10 行目) 場合は true を返す.