修 士 論 文

UML モデルに対するユーザー定義制約条件の整合性検査に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

伊東 恵輔

2003 年 3 月

修 士 論 文

UML モデルに対するユーザー定義制約条件の整合性検査に関する研究

指導教官　片山 卓也 教授

審査委員主査　片山 卓也 教授
審査委員　二木 厚吉 教授
審査委員　権藤 克彦 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

110013 伊東 恵輔

提出年月: 2003 年 2 月

目 次

図 目 次

第1章　はじめに

1.1　研究の背景と目的

現在、オブジェクト指向方法論に基づいたソフトウェア開発が盛んに行われている．UML はオブジェクト指向ソフトウェア開発の分析・設計段階におけるモデリング言語として広く用いられている．UML にはシステムの異なる側面を記述するための 9 種類のダイアグラムがある．これらのダイアグラムは多くの場合、互いに関連があり、独立していない．そのため、UML モデリングの際、UML モデルの各ダイアグラム間で不整合が生じる場合がある．今日のような大規模かつ複雑なシステムの開発においては、このような不整合が生じる可能性は極めて高い．しかし、人間が整合性検査を行なうことが困難であるのにもかかわらず、現在、ダイアグラム間での十分な整合性検査をおこなう CASE ツールがない．

関連した主な研究として、ダイアグラム間での構文面の整合性検査に関する研究がある [1]．しかし、この研究では、予め定義した検査項目に対する整合性検査に限定しており、ユーザードメイン特有の制約など、ユーザーが独自に制約条件を定義することはできない．一方、意味にまで踏み込んだモデルの検証に関する研究がある [2]．この研究の検証方法は、HOL 定理証明系を用いた深い推論が必要であり、一般ユーザーには向かない．

そこで、本研究では、ユーザーが独自に、UML メタモデルに対する制約条件を定義することが可能であり、実用的・汎用的な構文面のダイアグラム間整合性検査システムを提案する．本システムで検査をおこなう制約条件は、オブジェクト制約言語 OCL（Object Constraint Language）[4] で記述したものとする．そのため、ユーザーは、容易に曖昧さのない制約条件をシステムに与えることができる．また、構文面の整合性検査に限定しているため、比較的少ない計算機資源での検査が可能である．

本研究は、上記システムの実現を目指しておこなってきた．しかし、本整合性検査システムに非常に類似したシステムである、ocle(Object Constraint Language Evaluator)[3] が、ルーマニアのバヘシュ・ボヨイ（Babes-Bolyai）大学によって開発された（2003 年 1 月末）．この ocle については、第 6 章で説明する．

1.2 研究のアプローチ

本研究では、以下の特徴を持つ構文面のダイアグラム間整合性検査システムの実現を目指す.

- UML メタモデルに与えた制約条件に対する整合性検査
- ユーザーが独自に制約条件を記述可能
- 実用的・汎用的なシステム

本整合性検査システムでは、整合性検査の最初の手順として、まず OCL で記述した制約条件を入力としてとる．この OCL 制約条件を OCL パーザーが構文解析し、抽象構文木を生成する．抽象構文木生成時には、予め OCL パーザーに与えた UML メタモデルを用いて、OCL 制約条件の型チェックもおこなう．そして、生成された抽象構文木と検査対象モデルを入力として、整合性検査 Java プログラムが整合性検査をおこない、検査の結果を出力する.

検査対象のモデル、および OCL 制約条件の型チェックに必要な UML メタモデルは、モデリングツールから自動生成可能な UML XMI（XML Metadata Interchange）文書のモデルとする．UML XMI 文書とは、UML モデルを格納するための XML 文書である．現在、いくつかのモデリングツールが、描画したモデルからの UML XMI 文書の自動生成機能をサポートしており、モデル情報を格納する汎用的なファイル形式になりつつある.

上記手順で検査をおこなうための手法（コレクションの実現方法、UML XMI 文書からのオブジェクト生成方法など）を考案する．また、本整合性検査システムは構文面の整合性検査システムであり、すべての OCL 制約条件について整合性検査が可能であるわけではない．整合性検査可能な OCL 文法の範囲も明らかにする.

考案した整合性検査の手法に基づいて、システムの実装をおこなう．実装したシステムに対して、例題を用いて評価をおこない、本システムの有効性を示す.

1.3 研究成果

本研究では、目的とする構文面のダイアグラム間整合性検査システムを実現することができた.

システムの実現にあたり、コレクションの実現方法、UML XMI 文書からのオブジェクトの生成方法などの整合性検査の各手法を考案した.

考案した整合性検査の手法に基づいて、システムの実装をおこない、例題を用いて評価をおこなった．結果、与えたすべての OCL 制約条件に対して、正しく整合性検査がおこなえることを確認した.

よって、本研究のアプローチで、構文面のダイアグラム間整合性検査システムが実現可能であり、また、その有効性を示すことができた.

1.4 論文の構成

本論文の構成は以下のとおりである.

2 章 本研究における主要な技術である UML、OCL、XML、XMI について概要を述べる.

3 章 本整合性検査システムが対象とする UML メタモデルと OCL 文法の範囲について説明する.

4 章 本整合性検査システムの検査手法（コレクションの実現方法、UML XMI 文書からのオブジェクト生成方法など）について説明する.

5 章 実装した整合性検査システムの概要、および例題を用いた整合性検査システムの評価について述べる.

6 章 開発された整合性検査システム ocle（Object Constraint Language Evaluator）について述べる.

7 章 本研究のまとめと今後の課題について述べる.

第2章　諸技術の概要

第2章では、本研究における主要な技術であるUML、OCL、XMIについて概要を述べる.

2.1　UML（Unified Modeling Language）

オブジェクト指向分析設計における標準モデル化言語であるUMLについて説明する[5].

2.1.1　UMLを開発した動機

高層ビルを建てるには設計図が不可欠なように、強固なソフトウェアシステムの構築と保守には、モデルの設計が必要である．プロジェクトチームの円滑な通信と、堅牢なシステムを実現するためには、よいモデルの作成が鍵になる．プロジェクトの成功には他にも多くの要因があるが、厳密なモデル化言語の標準は必須である．モデル化言語は以下の3点を含まなければ成らない.

- モデル要素—基本的なモデル化概念と意味論
- 記法—モデル要素の図式的な表現
- ガイドライン—利用にあたっての慣用

UML以前には、標準化されたモデル化言語は存在しなかったので、多くは類似しているが細部が異なるモデル化言語の中から選択しなければならなかった．このような不統一が、新規ユーザーによるモデル化市場への参入やモデル化の利用を妨げる主要因であった.ユーザーは、モデル化の共通語を求めていた．UMLの開発は、1994年にGrady BoochとJames RumbaughがBooch法とOMT法を統合すべく作業を開始したことに始まった.1995年にはIvar Jacobsonが加わり、OOSE法の統合が行なわれた．彼らはUMLの草稿をOMG（Object Manegement Group）に何回も提出した．フィードバックとして得られた多くのアイデアと提案はUMLを改善するために取り入れられた．そして1997年にUMLの第1版が発表された.

2.1.2　UMLの開発目標

UMLの主な開発目標は以下のとおりである.

- ユーザーが分かりやすいモデルを開発・交換でき、すぐに使える図式的モデル化言語を提供する.
- 主概念を拡張するための拡張機構と特化機構を提供する.
- 特定のプログラミング言語や開発プロセスに依存しない.
- モデリング言語を理解するための形式基盤を提供する.
- オブジェクト指向ツール市場の成長を促進する.
- コラボレーション、フレームワーク、パターン、コンポーネントなどのより高次な開発概念を支援する.
- 最良の技術を統合する.

上記のように、UML は、ユーザーが分かりやすいモデルを開発・交換できる、すぐ使える図式的モデル化言語を提供する. 特定のプログラミング言語や開発プロセスへの依存を避け、さらには、拡張機構と特化機構を備えるなど、柔軟に対応できる設計がなされている.

2.1.3 ビュー、ダイヤグラム、モデル要素、一般的な機構

UML の初歩的な構成概念、構成要素について説明する.

- ビュー
  ビューとは、モデル化されるシステムのいろいろな側面を表すものである. ビューは、グラフではなく、多くのダイアグラムからなる抽象構造である. システムの全体像を描写するためには、システムの個々の側面をビューで説明し、そのビューをシステムの全側面に渡って定義する以外に方法はない. UML には 5 種類のビューがあり、各ビューは 1 種類または複数種のダイアグラムで構成される.
- ダイアグラム
  ダイアグラムは、ビューの内容を説明するグラフである. UML には 9 種類のダイアグラムがあり、それらを組み合わせて使用することによって、システムのビューをすべて構成することができる.
- モデル要素
  ダイアグラムで使用される概念は、モデル要素と呼ばれる. それらのモデル要素は、クラス、オブジェクト、メッセージ、モデル要素間の関係などのオブジェクト指向の概念を表現する. モデル要素間の関係には、関連、汎化、集約、依存などがある.

- 一般的な機構
  一般的な機構は、モデル要素に対してコメント、情報、セマンティクスなどを追加するためのものである．また、特定の方法論やプロセス、組織、ユーザーなどに対して UML を適合させ、または拡張するための仕組みも提供する．注釈や装飾などがある．

以下にそれぞれのビューについての説明を示す．

- ユースケースビュー
  ユースケースビューは外部のアクターの観点から、システムが提供しなければならない機能を記述したものである．ユースケースビューは顧客と、設計者、開発者、テスターのためのビューである．ユースケースビューにある多数のユースケースのそれぞれは、システムの各使用法を包括的に記述したものであり、機能用件となるものである．通常、ユースケースビューはユースケース図で記述される．

- 論理ビュー
  論理ビューは、システムがその機能をどのようにして提供するかを説明する．論理ビューは、主として設計者と開発者のためのビューである．ユースケースビューとは対照的に、論理ビューはシステムの内部に注目し、静的構造（クラス、オブジェクト、関係）および、動的コラボレーション（ある機能を実現するためのオブジェクト間でのメッセージのやり取り）を説明する．静的構造はクラス図とオブジェクト図を使って記述され、動的モデリングはステートチャート図、シーケンス図、コラボレーション図、アクティビティ図を使って記述される．

- コンポーネントビュー
  コンポーネントビューは、実装モジュールとその間の依存関係を説明する．コンポーネントビューは、主に開発者のためのビューである．コンポーネントビューはコンポーネント図から構成される．

- 並行性ビュー
  並行性ビューは、プロセスとプロセッサへのシステムの分割を取り扱う．効率的な資源の使用、並行実行、および実行環境からの非同期イベントの処理を可能にする．並行性ビューは、システム開発者とインテグレーターのためのビューであり、動的ダイアグラム（ステートチャート図、シーケンス図、コラボレーション図、アクティビティ図）と実装ダイアグラム（コンポーネント図、配置図）から構成される．

- 配置ビュー
  配置ビューは、コンピュータや装置などの、システムの物理的な配置とそれらの接続を表す．配置ビューは、開発者、インテグレーター、テスターのためのビューであり、配置図で表現される．

以下にそれぞれのダイアグラムについての説明を示す．

- ユースケース図
  ユースケース図は、複数の外部のアクター、およびシステムが提供するユースケースへのアクターの接続を表す．ユースケースとは、システムが提供する機能（ユーザーの目から見たシステムの振舞い）の記述である．アクターとは、システムの外部に存在して、システムに対して何らかの影響を与える抽象実体である．ユースケースの記述 は、外部からアクターが見るものに限られ、システムの機能の実現方法は記述しない．

- クラス図
  クラス図は、システム内のクラスの静的な構造を表す．クラスは、システムで扱われる「もの」を表す．クラス間の相互の関係には関連、汎化、集約、依存などの関係がある．クラス図には、属性と操作によるクラスの内部構造とともに、これらの関係が記述される．クラス図が静的と見られる理由は、システムのライフサイクルのどの時点をとっても、記述された構造が常に成り立つからである．システムは、通常、複数のクラス図によって表される．

- オブジェクト図
  オブジェクト図は、クラス図の変形であり、ほとんど同じ表記法を用いる．両者の違いは、オブジェクト図はクラスではなく、クラスから生成されたオブジェクト・インスタンスが表されるということである．オブジェクト図は、シーケンス図、コラボレーション図の一部として、一連のオブジェクトの動的なコラボレーションを表現するのにも使用される．

- ステートチャート図
  ステートチャート図は、クラスのオブジェクトが取りうるすべての状態と、状態の変化を引き起こすイベントを表す。状態の変化は遷移と呼ばれる．アクションには、状態の遷移が起こった時に実行しなければならないことが指定され、遷移にそのアクションを関連付けることができる．ステートチャート図は、クラスに複数のはっきりと定義された状態があり、そしてクラスの振舞いが状態に影響されたり、状態に応じて変化したりする時に作成される．

- シーケンス図
  シーケンス図は、多数のオブジェクト間の動的なコラボレーションを表す．シーケンス図の重要な点は、オブジェクト間でやり取りされるメッセージの順序（シーケンス）が表現されるということである．シーケンス図は、オブジェクトから下の垂直線を時間軸とし、その線に沿ってオブジェクト間で送受信されるメッセージの流れと順序を表現する．

- コラボレーション図
  コラボレーション図は、シーケンス図と同じように動的なコラボレーションを表す．協調動作をシーケンス図で表すかコラボレーション図で表すかは、選択の問題

である．もし、最も重要な側面として、時間や順序を強調したければ、シーケンス図を選ぶのがよいし、オブジェクトとその関係を強調したければ、コラボレーション図を選ぶのがよい．コラボレーション図は、オブジェクトとその関係が示されるオブジェクト図として描かれる．

- アクティビティ図
  アクティビティ図は、アクティビティの連続的な流れを表す．アクティビティとは、操作の中で実行される何らかの振る舞いを表している状態のことである．アクティビティ図は、アクション状態から構成され、それぞれのアクション状態は実行されるアクティビティ（アクション）の仕様を表す．

- コンポーネント図
  コンポーネント図は、コード・コンポーネントの観点からコードの物理的な構造を表す．コンポーネント間の依存関係を示すことによって、あるコンポーネントが変わった時に他のコンポーネントがどのような影響を受けるかを解析することができる．コンポーネント図は、プログラムを作成する際に使用される．

- 配置図
  配置図は、システム中のハードウェアとソフトウェアの物理的なアーキテクチャを表す．配置図を使って、コンピュータや装置（ノード）の接続とその種類を記述することができる．

2.1.4 言語アーキテクチャ

UML のアーキテクチャは４つの層から成り立っている（４層メタモデルアーキテクチャ）．この UML の言語アーキテクチャの層構造を図 2.1 に示す．

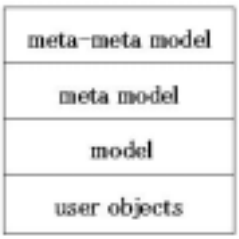

図 2.1: UML4 層メタモデルアーキテクチャ

最上位層のメタメタモデル層 (metameta model layer) は、メタモデルアーキテクチャの基盤となる．メタメタモデル層の基本的な責務は、メタモデルの仕様を記述する言語を定義することである．メタメタモデルは、モデルをメタモデルより高い抽象度で定義する．

メタメタモデル層の 1 つ下の層は、メタモデル層 (meta model layer) である．メタモデ

ルはメタメタモデルのインスタンスである．メタモデル層の基本的な責務は、モデルの仕様を記述する言語を定義することである．

メタモデル層の 1 つ下の層は、モデル層 (model layer) である．モデルはメタモデルのインスタンスである．モデル層の基本的な責務は、情報領域を記述する言語を定義することである．

最下位の層は、ユーザオブジェクト層 (user object layer) である．ユーザーオブジェクトはモデルのインスタンスである．ユーザーオブジェクト層の基本的な責務は、個別の情報領域を記述することである．

UML メタモデルは適度に複雑である．それは約 90 のメタクラスと 100 以上のメタ関連および約 50 のステレオタイプからなる．メタモデルの複雑さを扱うために、それを論理的なパッケージ群に体系化している．パッケージは、お互いに強い関連を持っていて、かつ他のパッケージのメタクラスとはあまり関係しないメタクラスのグループをひとまとめにする．UML のメタモデルの最上位パッケージは、Foundation（基盤)、Behavioral Elements（動的要素)、Model Management （モデル管理）の３つのパッケージで構成される．これを図 2.2 に示す．

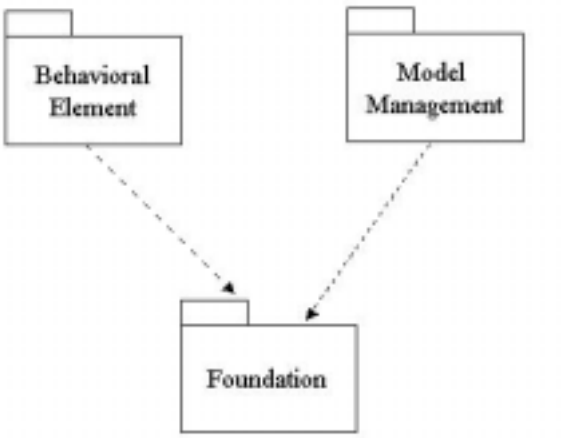

図 2.2: 最上位パッケージ

以下に最上位のそれぞれのパッケージについての説明を示す．

- Foundation（基盤）パッケージ
  Foundation パッケージは、モデルの静的構造を規定する言語インフラとなる．Foundation パッケージは、Core（コア)、Extension Mechanisms（拡張機構)、Data Types（データ型）という下位パッケージに分解される．図 2.3 に Foundation パッケージを示す．

  - Core（コア）
    Core パッケージは、UML の Foundation パッケージを構成する下位パッケー

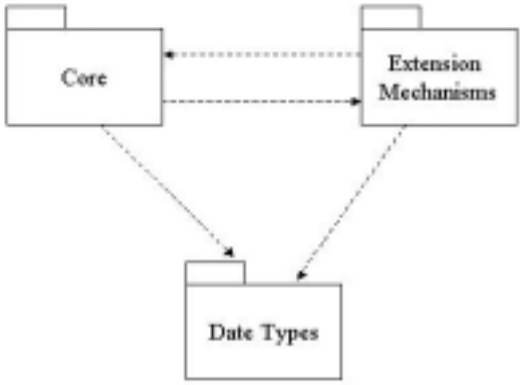

図 2.3: Foundation パッケージ

ジの中で最も基本的なものである．このパッケージは、オブジェクトモデルの開発に必要な基本的な抽象的および具象的メタモデル構成要素を定義する．抽象構成要素には、*ModelElement*（モデル要素）、*GeneralizableElement*（汎化可能要素）、*Classifier*（分類子）があり、具象構成要素には、Class（クラス）、Attribute（属性）、Operation （操作）、Association（関連）がある．

- Extension Mechanisms（拡張メカニズム）
  Extension Mechanisms パッケージは、モデル要素が新しい意味とともにどのようにカスタム化され拡張されるかを規定する下位パッケージである．これは、ステレオタイプ、制約、タグ付き値の意味論を定義する．
- Data Types（データ型）
  Data Types パッケージは、UML の定義に使われる異なるデータ型を規定する下位パッケージである．Integer、String、Boolean などのデータ型がある．

- Behavioral Elements（動的要素）パッケージ
  Behavioral Elements パッケージは、モデルの動的な振舞いを記述する言語の上位構造である．Behavioral Elements パッケージは、Common Behavior、Collaborations、Use Cases、State Machines、Activity Graphs の下位パッケージに分割される．図 2.4 に Behavioral Elements パッケージを示す．

  - Common Behavior（共通の振る舞い）
    Common Behavior パッケージは、Behavioral Elements パッケージを構成する下位パッケージ群の中で最も基本的なパッケージである．このパッケージは、動的要素に必要な核となる概念を規定するとともに、Collaborations、および State Machines、Use Cases を支える基盤を提供する．

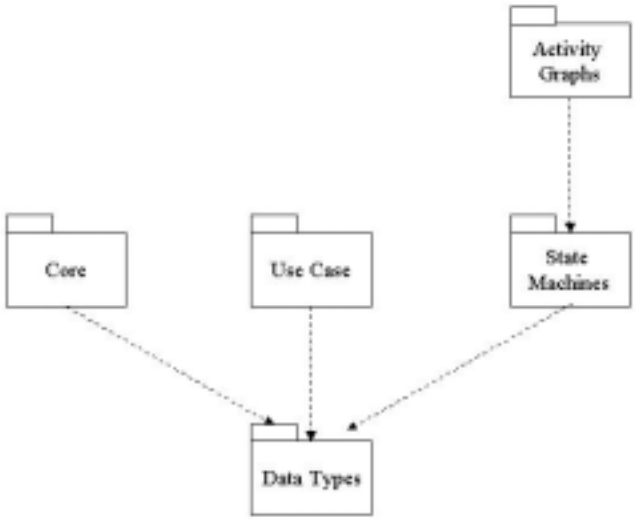

図 2.4: Behavioral Elements パッケージ

- Collaborations（コラボレーション）
  Collaborations パッケージは、モデル内の別々の要素が互いにどのように相互作用するかを、構造的な観点から表現するために必要な概念を規定する．このパッケージは、Foundation パッケージと Common Behavior パッケージとで定義された構成要素を使用する．
- Use Cases（ユースケース）
  Use Cases パッケージは、システムのような実体の機能性を定義するための概念を規定する．このパッケージは、Foundation パッケージと Common Behavior パッケージとで定義された構成要素を使用する．
- State Machines（状態マシン）
  State Machines パッケージは、独立した振舞いを有限状態遷移システムによりモデル化するために使うことのできる概念集合を記述する．このパッケージは、Foundation パッケージと Common Behavior パッケージとで定義された構成要素を使用する．
- Activity Graphs（アクティビティグラフ）
  Activity Graphs パッケージは、State Machines パッケージの拡張ビューを定義する．State Machine と Activity Graphs は、両者とも本質的に状態遷移システムであり、多くのメタモデル要素を共有する．

- Model Management（モデル管理）パッケージ

Model Management パッケージは、モデル、パッケージ、およびサブシステムを定義する．これらは他のモデル要素をグループ化する単位となる．

UML のメタモデルのダイアグラムそのものは、付録 A UML メタモデルに示した．

2.2 OCL（Object Constraint Language）

副作用のない制約記述形式言語であるオブジェクト制約言語 OCL について説明する [4, 5, 6].

2.2.1 OCL の概要

クラス図のような UML のダイアグラムは、仕様に必要なあらゆる側面を提供する程には詳細化されていないことが多い．モデルにおけるオブジェクトについて付加的な制約を記述する必要がある．これらの制約は自然言語で記述することが多い．しかし、実際には、これによって常に曖昧さが生じる．

曖昧さのない制約を書くために、いわゆる形式言語が開発されてきた．今までの制約言語は、数学的な背景を持つ者には使用可能だが、普通のビジネスおよびシステムのモデル化を行なう者には使用が困難という欠点があった．OCL は、この問題を解決するために開発された読み書きが容易な形式言語である．OCL は、IBM 保険部門でのビジネスモデル化言語として開発され、Syntropy 方法論にその起源を持つ．

UML の各構成要素の静的意味論は、多重度と順序制約を除いて、メタクラスのインスタンスの守らなければならない制約条件（不変表明）の集合として定義する．構成要素が意味を持つためには、この不変表明が満足されていなければならない．したがって、メタモデルで定義された属性と関連に関する制約を規則として定義する必要がある．これを適格性規則（well-formedness rules）という．OCL は、UML の意味論においてこの適格性規則を規定する．また、UML モデル作成者が、各自のモデルにおけるアプリケーション固有の制約を規定するためにも OCL を使用することができる．OCL は、以下の個所で使用される．

- クラスモデルにおける、クラスおよび型の不変表明を規定する．
- ステレオタイプのための、型の不変表明を規定する．
- 操作およびメソッドに関する事前条件および事後条件を記述する．
- ガードを記述する．
- 誘導言語（navigation language）として使用する．
- 操作に関する制約を規定する．

この節では、次の図 2.5 を例として使用する．

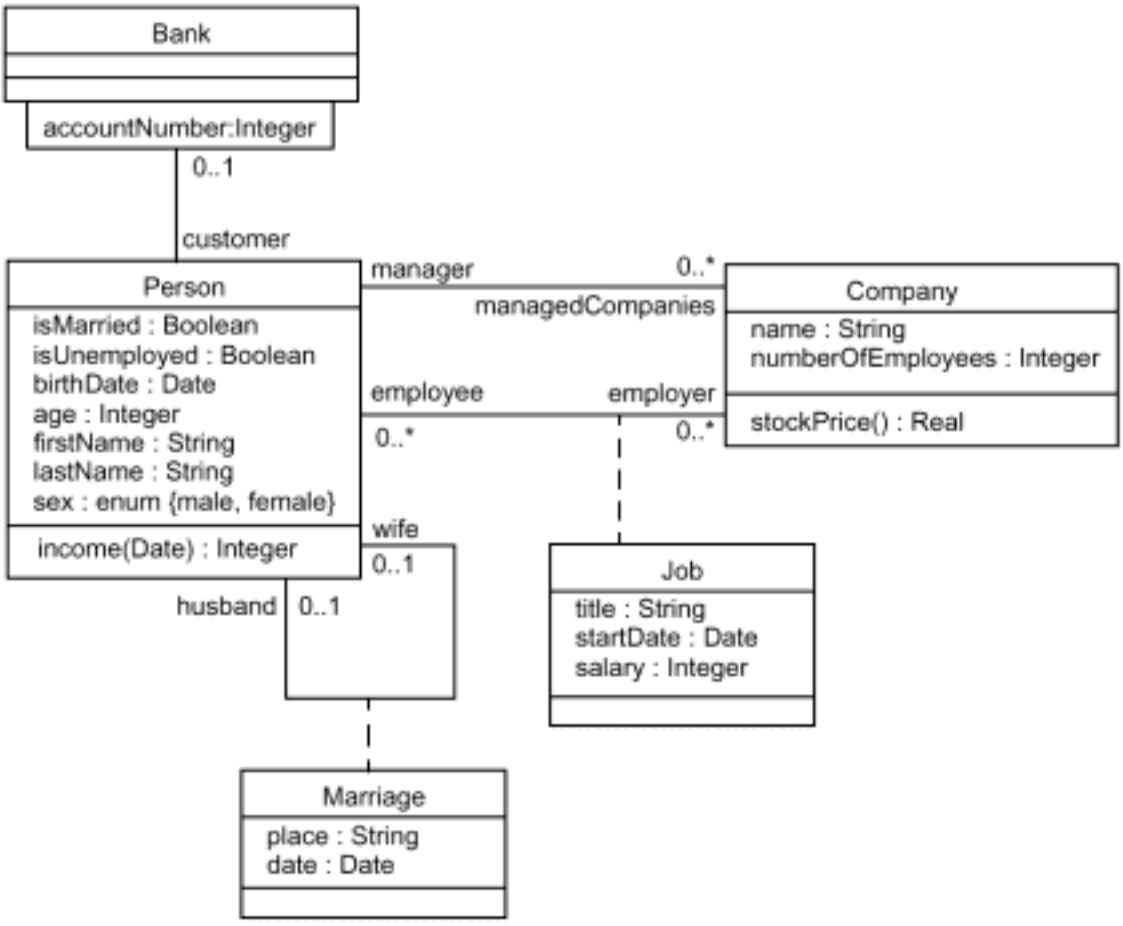
Bank
accountNumber:Integer
0..1
customer
Person
isMarried : Boolean
isUnemployed : Boolean
birthDate : Date
age : Integer
firstName : String
lastName : String
sex : enum {male, female}
income(Date) : Integer
manager
0..*
managedCompanies
Company
name : String
numberOfEmployees : Integer
stockPrice() : Real
employee
0..*
employer
0..*
wife
0..1
husband
0..1
Job
title : String
startDate : Date
salary : Integer
Marriage
place : String
date : Date

図 2.5: クラス図の例

2.2.2 UML メタモデルとのつながり

- UML 文脈（コンテキスト）の指定
  UML モデル内の OCL 式のコンテキストは、OCL 式の最初のいわゆるコンテキスト宣言によって指定できる．各 OCL 式は、特定の型のインスタンスのコンテキストを用いて記述する．OCL 式では、予約語 self を使ってそのコンテキストにおけるインスタンスを参照する．たとえば、コンテキストが Company の場合、self は Company のインスタンスを参照する．

- 不変表明
  OCL 式は、“ invariant ”でステレオタイプ化された Constraint（制約）である Invariant（不変表明）の一部となることができる．Invariant が Classifier（分類子）と関連づけられている場合、Classifier を型として参照する．この場合、OCL 式はその型の不変表明であって、常にその型のインスタンスに対して真でなければならない（不変表明を表現するすべての OCL 式は、論理型を持つ）．
  不変表明の一部である、OCL 式のコンテキストでのインスタンスの型は、context キーワードとそれに続く型の名前を使って記述する．ラベル inv:で、その制約が“ invariant ”制約であることを宣言する．たとえば、図 2.5 で、コンテキストが Company 型であり、従業員の数が常に 50 を越えなければならないという不変表明は次のように書くことができる．また、大抵の場合、キーワード self は省略することが可能である．

```
context Company inv:
 self.numberOfEmployees > 50
```

2.2.3 OCL の型

- 定義済みの型
  OCL では、多くの型が定義済みであり、モデル作成者が使用できる．定義済みの型は、任意のオブジェクトモデルから独立であり、OCL 定義の一部である．基本型として、Boolean 型、Integer 型、Real 型、String 型がある．また、コレクション型として、Collection 型、Set 型、Bag 型、Sequence 型がある．関連による 1 回のナビゲーション（後述）をおこなうと、Set となり、複合誘導は Bag となり、{ordered} で修飾された関連のナビゲーションは Sequence となる．以下に各コレクション型についての説明を示す．

  - Collection 型
    Collection 型は、抽象型であって、その下位型として、具象的なコレクション型、Set 型、Bag 型、Sequence 型を持つ．

- Set 型
  Set は数学でいう集合である．つまり重複した要素を含まず、順序付けのない要素の集まりである．
- Bag 型
  Bag 型は重複した要素を含むことが可能で、順序付けのない要素の集まりである．
- Sequence 型
  Sequence 型は重複した要素を含むことが可能で、順序付けのある要素の集まりである．

その他に、列挙型である Enumeration 型と、基本型を補足する OclExpression 型、OclType 型、OclAny 型がある．以下に OclExpression 型、OclType 型、OclAny 型についての説明を示す．

- OclExpression 型
  OCL 式自体は、OCL のコンテキストにおけるオブジェクトである．OCL 式自体の型を OclExpression 型とする．この型およびそのプロパティは、そのパラメタの 1 つとして式、たとえば、select、forAll などをとるプロパティの意味定義に使用する．
- OclType 型
  UML モデルで定義された型、および OCL 内で定義済みのすべての型は、1 つの型を持つ．この型は、OclType 型と呼ぶ OCL 型のインスタンスである．OclType 型へのアクセスによって、モデル作成者は、モデルのメタレベルに制限付きでアクセスできる．
- OclAny 型
  OCL のコンテキスト内で、OclAny 型は、モデル内のすべての型および基本定義済み OCL 型の上位型とする．定義済みの Collection 型は、OclAny 型の下位型ではない．OclAny 型のプロパティは、すべての OCL 式における各オブジェクトに関して利用可能である．

- UML モデルからの型
  OCL 式は、UML モデル、分類子（型/クラスなど）、関連などといったコンテキストで記述される．UML モデルの分類子はすべて、モデルに付いた OCL 式の型とする．

2.2.4 型の適合

型 Type1 のインスタンスが型 Type2 のインスタンスを期待する場所で置換できる場合、型 Type1 は型 Type2 に適合するという．

OCL は、型付き言語であり、基本的な型は型階層で構成される．この階層が異なる型の適合を決定する．すべての型が適合する OCL 式は、妥当（valid）である．型が適合しない OCL 式は、無効（invalid）であり、型適合エラーを含む．クラス図における型適合規則を以下に示す．

- 各型は、その上位型に適合する．
- 型適合は、推移的とする．つまり、Type1 が Type2 に適合し、Type2 が Type3 に適合する場合、Type1 は Type3 に適合する．

表 2.1 に基本型に対する型適合規則を示す．

表 2.1: 基本型の型適合規則

型	～に適合する/～の下位型である
Set	Collection
Sequence	Collection
Bag	Collection
Integer	Real

2.2.5 再型付け（キャスト）

状況によっては、下位型で定義されたオブジェクトのプロパティを使用したいことがある．そのプロパティは現在のオブジェクト型では定義されていないので、型適合エラーを生じる．オブジェクトの実際の型が下位型であることが確実な場合、そのオブジェクトを操作 oclAsType(OclType) を使ってキャストできる（oclAsType のような定義済みのプロパティについては、後述する）．型 Type1 のオブジェクト object が存在し、Type2 が Type1 と異なる型の場合、次のように記述することができる．

```
object.oclAsType(Type2)   --- object を型 Type2 で評価する
```

2.2.6 プロパティの参照とナビゲーション（誘導）

- プロパティの参照
  属性、関連端、ならびに副作用のないメソッドおよび操作を、オブジェクトのプロパティと呼ぶ．副作用のないメソッドおよび操作とは、その isQuery 属性が真であるものである．OCL 式で、このオブジェクトのプロパティを参照することができる．プロパティは次のいずれかである．

- 属性（Attribute）
- 関連端（AssociationEnd）
- isQuery が真である操作
- isQuery が真であるメソッド

クラス図で定義されるオブジェクトのプロパティ値は、ドットとそれに続くプロパティ名で指定する．

```
context TypeA inv:
 self.property
```

たとえば、図 2.5 の例に対する、Person の年齢（属性 age）が 0 歳以上であるという不変表明を以下のように書くことができる．

```
context Person inv:
 self.age > 0
```

- ナビゲーション（誘導）
  特定のオブジェクトから出発し、他のオブジェクトおよびそのプロパティを参照するために、クラス図上の関連による誘導が可能である．このためには、関連の反対側のロール名を用いて以下のように記述する．ロール名がない場合には、その関連端における型の名前を小文字で始めたものをロール名として使用する．

  ```
  object.rolename
  ```

  たとえば、図 2.5 の例に対して、Company のコンテキストで出発した、ナビゲーションを含む不変表明を以下のように書くことができる．

  ```
  context Company inv:
   self.manager.isUnemployed = false
  ```

2.2.7 定義済みプロパティ

前述した OCL の定義済みの型には、以下に示すような多くの定義済みプロパティがある．

- すべてのオブジェクトについての定義済みプロパティ
  次に示すように、すべてのオブジェクトに適用できる定義済みプロパティが存在する．これらのプロパティは、モデル内のすべての型および基本定義済み OCL 型の上位型である、oclAny 型のプロパティとして定義されている．

```
oclIsTypeOf(t: OclType)   : Boolean
oclIsKindOf(t: OclType)   : Boolean
oclInState(s: State)      : Boolean
oclIsNew                  : Boolean
oclAsType(t: OclType)     : instance of OclType
```

- コレクション操作
  OCL では、コレクションのプロパティとして、多くの操作が定義されている．コレクション操作は、'->' に操作名を続けて書くことによりアクセスできる．たとえば、図 2.5 の例に対して、Company のコンテキストで出発した、コレクション操作を含む不変表明を以下のように書くことができる．

```
context Company inv:
 self.employee->forAll(e | e.isUnemployed = false)
```

  forAll 操作は、コレクションの中のすべての要素において、引数となる OCL 式が成立する場合に真を返す．変数 e は、反復子（イテレータ）と呼ばれるもので、コレクションの中の 1 つの要素を参照する．変数 e は、コレクション上を順次割り当てられ、各 e に対して、引数の OCL 式が評価される．コレクション操作には、この他にも exists、size、select などの数多くのコレクション操作がある．いくつかの基本的なコレクション操作については、3.2 OCL 文法の範囲で説明する．また、付録 B に基本的なコレクション操作の一覧を示した．

2.2.8 OCL の文法

OCL の文法規則については、3.2 OCL 文法の範囲、および 4.2 OCL 文法の範囲制限にて説明する．また、OCL の文法規則を付録 C OCL の文法規則に示した．

2.3 XML（eXtensible Markup Language）

本研究で提案する整合性検査システムでは、検査対象モデルの UML XMI 文書、および OCL パーザーが型チェックをおこなう際に用いる、UML メタモデルの UML XMI 文書、これら 2 つの UML XMI 文書をシステムの入力として用いる．UML XMI 文書とは、XMI 標準により UML メタモデルに対して定義された、UML XMI DTD に従った XML 文書である．

そのため、本研究においては、XML と XMI の技術が重要である．本研究のメインテーマである整合性検査の手法を考案するためには、これらの技術に深く関わる必要がある．以下、XML 技術について説明する．

2.3.1 XML の概要

1998 に w3c（WWW Consortium）によって勧告された、拡張可能な（ユーザーがタグを定義可能な）マークアップ言語である [8]．XML は SGML（Standard Generalized Markup Language）をメタ言語としているが、SGML に対し簡略化された仕様になっている（同じインスタンス言語である HTML に近い）．XML には、以下の特徴がある．

- テキスト形式—人間が読める．
- 拡張が可能—ユーザーがタグを定義可能．
- 自己記述性—タグでデータの構造と意味を記述．文書としてだけでなく、データ交換にも有用．
- 表示を制御可能—スタイルシートにより、表示を制御できる．
- 多くの関連技術が存在—DTD（Document Type Declaration）、DOM （Document Object Model）、XPath（XML Path Language）など．WWW や Java との相性がよい．

2.3.2 DTD（Document Type Declaration）

DTD とは、XML 文書の構造を定義するスキーマ記述言語である．DTD は、次の 4 つからなる．

- 要素型宣言—要素（タグ）名と要素の構造を定義．
- 属性リスト宣言—要素の属性を定義．
- エンティティ宣言—XML 文書内で参照される実体（エンティティ）を定義．

- 記法宣言—外部に存在する XML 文書ではないファイルのデータ形式を伝える名前の宣言.

図 2.6 に DTD の例を示す．この DTD の例において、2 行目から 4 行目、6 行目および 7 行目は、要素型宣言である．例えば、4 行目の'<!ELEMENT student (name, email+)>'では、子要素として、要素 name を 1 つ、かつ要素 email を 1 つ以上持つ、要素 student を定義している（'+' 記号は、1 回以上の繰り返しを、また'#PCDATA' は文字列データを表す）.

5 行目は、要素 student の属性リスト宣言である．定義する属性は、1 行目で宣言されたエンティティを参照している．1 行目のエンティティ宣言では、参照された属性の、属性名が'studentID'、属性値の型が識別子である'ID'、そして'#REQUIRED' によって、この属性が必須属性であることを定義している（もちろん'<!ATTLIST student studentID ID #REQUIRED>' として、エンティティの参照を用いない属性リスト宣言を書くこともできる．しかし、次節 XMI（XML Metadata Interchange）における UML XMI DTD のエンティティ宣言の説明のために、敢えてエンティティの参照を用いるようにした）.

図 2.7 に、図 2.6 の DTD に従った XML 文書の例を示す．1 行目は XML 宣言と呼ぶものであり、XML のバージョンおよび文字コードを宣言する．2 行目は DOCTYPE 宣言と呼ぶものであり、この XML 文書が従うべき DTD を宣言する.

```
1 <!ENTITY % student.att 'studentID ID #REQUIRED'>
2 <!ELEMENT Katayama_lab (menber)>
3 <!ELEMENT member (student)+>
4 <!ELEMENT student (name, email+)>
5 <!ATTLIST student %student.att;>
6 <!ELEMENT name (#PCDATA)>
7 <!ELEMENT email (#PCDATA)>
```

図 2.6: DTD の例

2.3.3 DOM（Document Object Model）

DOM とは、プログラムによる XML 文書処理のための標準インタ-フェースである．DOM では、XML パーザーにより、XML 文書の木（DOM ツリー）をメモリ上に構築する．DOM は、この DOM ツリーを操作するための多くのインターフェースを提供する．DOM は扱いやすい反面、メモリ消費が大きいという欠点がある．一方、DOM ツリーを構築しない（メモリ消費の少ない）イベント駆動型の SAX（The Simple API for XML）もある．本整合性検査システムでは、次に説明する検索言語 XPath の利用のため、DOM を用いる.

図 2.8 に、図 2.7 の XML 文書を DOM ツリーで表した例を示す．DOM では、子ノード

```
 1 <?xml version = '1.0' encoding = 'UTF-8' ?>
 2 <!DOCTYPE Katayama_lab SYSTEM 'Katayama_lab.dtd'>
 3 <Katayama_lab>
 4   <member>
 5     <student studentID='110013'>
 6       <name>Keisuke Ito</name>
 7       <email>k-itou@jaist.ac.jp</email>
 8     </student>
 9   </member>
10 </Katayama_lab>
```

図 2.7: XML 文書の例

を削除したり、あるいはノードを生成して、DOM ツリーに追加するといったような多くのツリー操作が可能である.

2.3.4 XPath（XML Path Language）

XML に関する有用な関連技術の 1 つとして、検索言語 XPath がある．XPath は、XML 文書内の要素や属性の位置を指定してノードの集合を得る．以下、XPath について説明する.

- ロケーションパス
  ノードを条件付きのパスで指定して、ノード集合を得る.
  - ロケーションパスの構文
    - 絶対ロケーションパス ::= '/' 相対ロケーションパス?
      文書のルートノードを context node として評価.
    - 相対ロケーションパス ::= ステップ ('/' ステップ)*
      前のステップが返す集合の各ノードを context node として、次のステップを評価し、各結果集合の和を取る.
    - ステップ ::= 軸 '::' ノードテスト ('[' 述語 ']')*

- 軸（axis）
  パスをたどる方向を指定する．軸は 13 種類ある.
  - 軸の種類
    self（自分）、parent（親）、child（子．属性ノードと namespace は含まれな

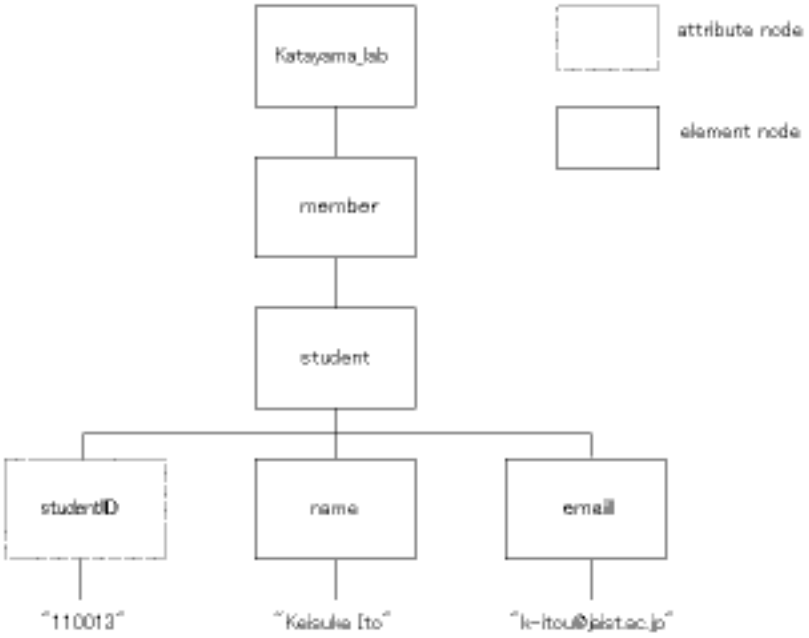

図 2.8: DOM ツリーの例

い)、preceding-sibling（parent の child のうち順序が自分より前のもの（兄））、following-sibling（parent の child のうち順序が自分より後のもの（弟））、ancestor（祖先)、preceding（自分や祖先の兄)、following（自分や祖先の弟)、descendant（子孫)、attribute （属性)、namespace（ネームスペース)、descendant-or-self（自分または子孫)、ancestor-or-self（自分または祖先）

- ノードテスト
  ノードの種類や名前を指定する.
  - ノードテスト
    - 名前—その名前のノード．namespace も使用可
    - *—主ノード型のすべてのノード．namespace も使用可
    - text()—文字列ノード
    - comment()—コメントノード
    - processing instruction(名前)—その名前を持つ処理命令ノード
    - node()—すべてのノード
  - 主ノード型
    - attribute 軸—属性
    - namespace 軸—namespace

    - その他の軸—要素

- 述語（predicates）
  述語が true となるノードからなるノード集合を返す.
  - 述語の形式—'[' expression ']'
  - 述語の評価手順
    - ノード集合中の各ノードを context として、述語を評価
    - その結果が、
      数字だったら、近接位置と等しければ true
      ノード集合だったら、空集合なら false. 非空集合は true.
      ...

- expression（式）
  - expression の形式
    - ロケーションパス
    - ノード集合 ('|' ノード集合)*
    - ノード集合 '[' 述語 '['
    - ノード集合 '/' ロケーションパス
    - expression ('or' | 'and' | '=' | '>' | ...) expression
    - 変数、定数文字列、数字定数
    - 関数呼び出し—last()、position() など.

- 省略記法
  以下の表 2.2 に XPath の省略記法を示す.

表 2.2: XPath の省略記法

省略記法	省略しない場合
（無し）	`child::`
@	attribute
//	`/descendant-or-self::node()/`
.	`self::node()`
..	`parent::node()`

これらの構文を用いることにより、XPath では、XML 文書内の様々なノード集合を得ることができる. 表 2.3 に、前述した図 2.7 の XML 文書に対する XPath 式の例を示す.

表 2.3: XPath 式の例

XPath 式	説明
`//member`	ルートノードの子孫であるノードのうち、要素 member であるもの
`//member/node()[@studentID]`	ルートノードの子孫であるノードのうち、親が要素 member であり、属性 studentID を持つノード
`//member/student[@studentID = '110013']`	ルートノードの子孫であるノードのうち、親が要素 member であり、値が'110013' に等しい属性 studentID を持つ要素 student

2.4 XMI（XML Metadata Interchange）

本整合性検査システムで用いるXML文書は、UML XMI DTDに従ったUML XMI文書である．前節で述べたように、本研究では、XMLとともに、XMIの技術が重要である．以下、XMI技術について説明する．

2.4.1 XMIの概要

XMIはOMGによる標準であり、XMLを使ったオブジェクトの交換をおこなうための構造を規定する[7, 4]．すなわち、モデルからどのようにXMLのDTD（XMI DTD）を生成するかを規定する．XMIには、以下の特徴がある．

- XMLを用いた標準的な表現方法を提供することで、効率的なオブジェクトの交換が可能．
- アプリケーションにより、XMI文書をさらに発展させることが可能．
  （データの拡張、クロスファイルリンク、または本整合性検査システムを用いて検査をおこない、修正を加えるなど）
- 多くのXML関連技術が利用可能．

2.4.2 データとメタデータ

しばしば、何らかのデータを用いる時、そのデータをメタデータと見なしたり、または単なるデータと見なしたりする．たとえば、モデルは、そのオブジェクト（インスタンス）を定義するメタデータと見なすことができる．しかし、一方で、モデリングツールは、モデルをメタデータではなく、単なるデータと見なす．

XMIでは、モデルのオブジェクトと同様に、モデルをXMI文書として格納することも可能である．前述したように、XMIは、モデルからどのようにXMI DTDを生成するかを規定する．そのため、モデルをXMI文書として格納するためには、XMI DTDを生成するためのメタモデルが必要となる．たとえば、UMLのモデルを格納する場合、UML XMI DTDを生成するために、UMLメタモデルが必要となる．

XMI文書は、メタデータとデータの両方を持つ．文書の型がメタデータであり、文書の要素が持つ情報がデータである．表2.4に、データとメタデータの例を示す．

2.4.3 UML XMI DTD

UML XMI DTDは、XMI仕様に基づいて、UMLのメタモデルから生成されたXMI DTDである[5]．まず、すべてのXMI DTDに共通な構造について説明し、後にUML XMI DTDにおける要素型宣言について説明する．

表 2.4: データとメタデータの例

メタデータ	生成される XMI DTD	データ
モデル	モデルに基づいた XMI DTD	オブジェクト
UML メタモデル	UML XMI DTD	モデル

- XMI DTD の構造
  すべての XMI DTD は、以下の宣言から構成される.
  - XML 宣言（例. `<?xml version = '1.0' encoding = 'UTF-8' ?>`）
  - 他のすべての妥当な XML 処理命令
  - 不可欠な XMI 宣言
- 不可欠な XMI 宣言
  XMI 仕様により、XMI DTD において宣言しなければならない、多くの要素型定義、属性リスト宣言、エンティティ宣言がある. これらのうち、本研究において関係のあるもの、重要であるものについて説明する.
  - 共通の要素
    XMI DTD において宣言しなければならない多くの要素がある. 以下に、その一覧を示す.

XMI	XMI.header	XMI.content
XMI.extensions	XMI.extension	XMI.documentation
XMI.owner	XMI.contact	XMI.longDescription
XMI.shortDescription	XMI.exporter	XMI.exporterVersion
XMI.exporterID	XMI.notice	XMI.model
XMI.metamodel	XMI.metametamodel	XMI.difference
XMI.delete	XMI.add	XMI.replace
XMI.reference	XMI.field	XMI.struct
XMI.seqItem	XMI.sequence	XMI.arrayLen
XMI.array	XMI.enum	XMI.discrim
XMI.union	XMI.any	

    * 本研究において関係のある共通の要素
      上記の要素のうち、本研究において関係のあるものについて、UML XMI DTD（UML1.1）の要素 XMI の定義を用いて説明する. これを図 2.9 に示す（図中の行番号は、UML XMI DTD における行番号を表す）.

要素 XMI は、XMI 文書のトップレベルに位置する要素である．22、23 行目に、要素 XMI が持ち得る子要素が定義されている．要素 XMI.header は、メタデータを識別する要素やモデルの変換に関する様々な情報を持つ要素などを含む．要素 XMI.content は、変換された実際のデータ、およびメタデータを含み、モデル、およびメタモデルに関する情報を表現する．要素 XMI.extensions は、メタモデルの拡張であるメタデータを含む．たとえば、メタデータに関する表示情報などである．

```
17 <!-- ________________________________________________________________ -->
18 <!--    -->
19 <!-- XMI is the top-level XML element for XMI transfer text    -->
20 <!-- ________________________________________________________________ -->
21
22 <!ELEMENT XMI (XMI.header, XMI.content?, XMI.difference*,
23     XMI.extensions*) >
24 <!ATTLIST XMI
25     xmi.version CDATA #FIXED "1.0"
26     timestamp CDATA #IMPLIED
27     verified (true | false) #IMPLIED
28 >
```

図 2.9: 要素 XMI の定義

* XML 文書の例
  実際の XML 文書の例として、本論文 5.2 システムの評価において、例題として用いたモデル（ニュースシステム）の XMI 文書の冒頭部分を図 2.10 に、要素 XMI.extensions の部分を図 2.11 に示す（これらの XMI 文書は、Rational Rose2001A を用いてモデルを作成した後、Rational Rose のアドイン、Unisys Rose/XMI interchange package（UML XMI DTD(UML1.1)）を用いて自動生成した 1 つの XMI 文書である）．

– 要素を識別する属性
  要素が互いに関連し合うことができるように、要素を識別するための 3 つの属性がある．これらの属性のうち、本研究では、属性 xmi.id が重要である．図 2.12 に、UML XMI DTD における属性 xmi.id の定義を示す．73 行目に定義された属性 xmi.id は ID 型であり、XMI 文書内においてユニークな値を持つ．

– リンクのための属性
  要素を識別する属性の値を用いることによって、他の要素への参照を可能にす

```
16 <XMI xmi.version = '1.0'>
17  <XMI.header>
18    <XMI.documentation>
19     <XMI.exporter>Unisys.IntegratePlus.2</XMI.exporter>
20     <XMI.exporterVersion>4.0.3</XMI.exporterVersion>
21    </XMI.documentation>
22    <XMI.metamodel xmi.name = 'UML' xmi.version = '1.1'/>
23  </XMI.header>
24  <XMI.content>
25    <Model_Management.Model xmi.id = 'G.1'>
```

図 2.10: 要素 XMI の例

```
4354     </Model_Management.Model>  <!-- End Model G.1 -->
4355   </XMI.content>
4356   <XMI.extensions xmi.extender = 'Unisys.IntegratePlus.2'>
4357     <UISModelElement xmi.id = 'G.350'>
4358       <uisOwnedDiagram>
4359         <UISDiagram xmi.id = 'S.10040'>
4360           <uisDiagramName>UseCaseDiagram</uisDiagramName>
4361           <uisToolName>Rational Rose 98</uisToolName>
4362           <uisDiagramStyle>ClassDiagram</uisDiagramStyle>
4363           <uisDiagramPresentation>
4364             <Foundation.Auxiliary_Elements.Presentation xmi.id='G.351'>
4365               <Foundation.Auxiliary_Elements.Presentation.geometry>
4366                 <Foundation.Data_Types.Geometry>
4367                   <Foundation.Data_Types.Geometry.body>
4368                     2192, 1744, 450, 314,
```

図 2.11: 要素 XMI.extensions の例

```
65 <!-- ____________________________________________________________ -->
66 <!--    -->
67 <!-- XMI.element.att defines the attributes that each XML element    -->
68 <!-- that corresponds to a metamodel class must have to conform to   -->
69 <!-- the XMI specification.    -->
70 <!-- ____________________________________________________________ -->
71
72 <!ENTITY % XMI.element.att
73     'xmi.id ID #IMPLIED xmi.label CDATA #IMPLIED xmi.uuid
74     CDATA #IMPLIED ' >
```

図 2.12: 属性 xmi.id の定義

る属性がいくつかある．これらの属性のうち、本研究では、属性 xmi.idref が重要である．図 2.13 に、UML XMI DTD における属性 xmi.idref の定義を示す．88 行目に属性 xmi.idref が定義されている．ある要素の属性 xmi.id の値をこの属性 xmi.idref の値とすることで、その要素を参照することができる．

```
76 <!-- ------------------------------------------------------------ -->
77 <!--    -->
78 <!-- XMI.link.att defines the attributes that each XML element that  -->
79 <!-- corresponds to a metamodel class must have to enable it to    -->
80 <!-- function as a simple XLink as well as refer to model    -->
81 <!-- constructs within the same XMI file.    -->
82 <!-- ------------------------------------------------------------ -->
83
84 <!ENTITY % XMI.link.att
85     'xml:link CDATA #IMPLIED inline (true | false) #IMPLIED
86     actuate (show | user) #IMPLIED href CDATA #IMPLIED role
87     CDATA #IMPLIED title CDATA #IMPLIED show (embed | replace
88     | new) #IMPLIED behavior CDATA #IMPLIED xmi.idref IDREF
89      #IMPLIED xmi.uuidref CDATA #IMPLIED' >
```

図 2.13: 属性 xmi.idref の定義

- UML XMI DTD における要素型宣言

  UML XMI DTD において、UML メタモデルのすべてのクラス、クラスの属性、およびクラスが持つ関連の反対側のロール（以後、ナビゲーション先ロール）が、XMI 文書の要素として定義される（UML メタモデルはプロパティとして操作を持たない）．

  ここでは、UML メタモデルのクラス Class を例にとり、説明する．図 2.14 にメタモデルの Core Package - Backbone を、図 2.15 に Core Package - Relationships を示す（図のメタモデルは UML1.3 から引用しており、UML XMI DTD のバージョン UML1.1 とは異なる．変更点を除いた範囲で説明する）．

  - クラスの要素型宣言

    クラスの要素型宣言では、そのクラスが持つ（上位型から継承したものも含む)、属性およびナビゲーション先ロールが子要素として定義される．図 2.16 にクラス Class の要素型宣言を示す．子要素として定義されるクラスの属性として、たとえば、1890 行目の要素 Foundation.Core.ModelElement.name がある．これは、上位型である *ModelElement* の属性 name を継承したものである（図 2.15）．ナビゲーション先ロールとしては、たとえば、1927 行目の要素 Foundation.Core.Classifier.feature がある．これは、Class の上位型である *Classifier* のナビゲーション先ロール feature を継承したものである（図 2.14、

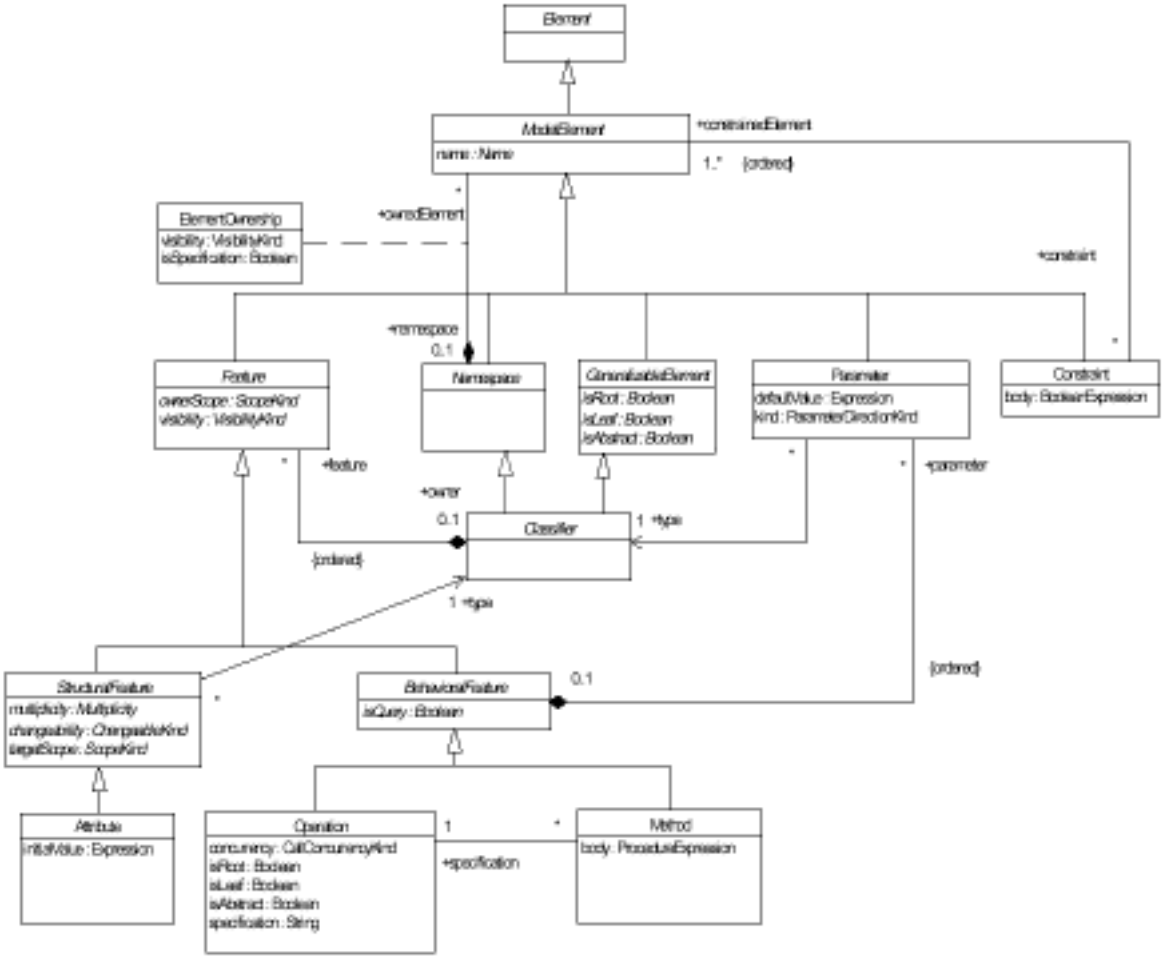

図 2.14: Core Package - Backbone

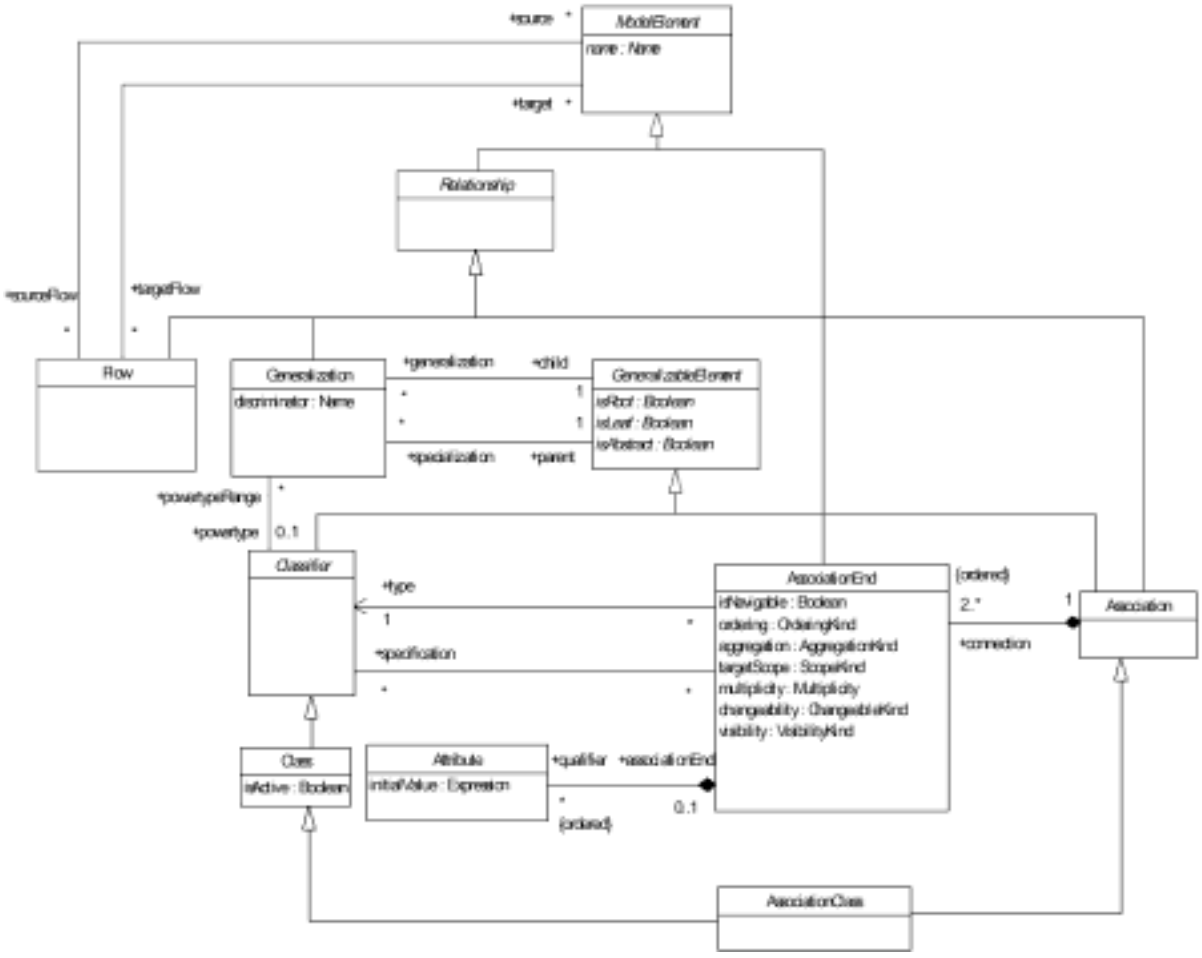

図 2.15: Core Package - Relationships

```
1889 <!ELEMENT Foundation.Core.Class (
1890     Foundation.Core.ModelElement.name,
1891     Foundation.Core.ModelElement.visibility,
1892     Foundation.Core.GeneralizableElement.isRoot,
1893     Foundation.Core.GeneralizableElement.isLeaf,
1894     Foundation.Core.GeneralizableElement.isAbstract,
1895     Foundation.Core.Class.isActive,
1896     XMI.extension*,
1897     Foundation.Core.ModelElement.constraint*,
1898     Foundation.Core.ModelElement.requirement*,
1899     Foundation.Core.ModelElement.provision*,
1900     Foundation.Core.ModelElement.stereotype*,
1901     Foundation.Core.ModelElement.elementReference*,
1902     Foundation.Core.ModelElement.collaboration*,
1903     Foundation.Core.ModelElement.partition?,
1904     Foundation.Core.ModelElement.template?,
1905     Foundation.Core.ModelElement.templateParameter*,
1906     Foundation.Core.ModelElement.view*,
1907     Foundation.Core.ModelElement.presentation*,
1908     Foundation.Core.ModelElement.namespace?,
1909     Foundation.Core.ModelElement.behavior*,
1910     Foundation.Core.ModelElement.binding?,
1911     Foundation.Core.ModelElement.implementation*,
1912     Foundation.Core.GeneralizableElement.generalization*,
1913     Foundation.Core.GeneralizableElement.specialization*,
1914     Foundation.Core.Classifier.parameter*,
1915     Foundation.Core.Classifier.structural Feature*,
1916     Foundation.Core.Classifier.specification*,
1917     Foundation.Core.Classifier.associationEnd*,
1918     Foundation.Core.Classifier.participant*,
1919     Foundation.Core.Classifier.createAction*,
1920     Foundation.Core.Classifier.instance*,
1921     Foundation.Core.Classifier.collaboration*,
1922     Foundation.Core.Classifier.classifierRole*,
1923     Foundation.Core.Classifier.realization*,
1924     Foundation.Core.Classifier.classifierInState*,
1925     Foundation.Core.ModelElement.taggedValue*,
1926     Foundation.Core.Namespace.ownedElement*,
1927     Foundation.Core.Classifier.feature*)? >
1928     <!ATTLIST Foundation.Core.Class
1929     %XMI.element.att;
1930     %XMI.link.att;
1931 >
```

図 2.16: Class の要素型定義

2.15）．1929、1930 行目には、前述した要素を識別するための属性、およびリンクのための属性が定義されている．これにより、他の要素を参照したり、されたりすることが可能となる．

– クラスの属性の要素型宣言
例として、*ModelElement* の属性 name の要素型宣言を図 2.17 に示す．要素 Foundation.Core.ModelElement.name は、子要素として、文字列データまたは他の要素への参照を持つ．

```
1329 <!ELEMENT Foundation.Core.ModelElement.name (#PCDATA | XMI.reference)* >
```

図 2.17: *ModelElement* の属性 name の要素型定義

– ナビゲーション先ロールの要素型宣言
例として、*Classifier* のナビゲーション先ロール feature の要素型宣言を図 2.18 に示す．ナビゲーション先ロールの要素は、クラスの要素のように、要素を識別するための属性、およびリンクのための属性を持たない．代わりに、下位型のクラスの要素を子要素として持つ．この例での子要素は、*Feature*、およびその下位型クラスである．例として、モデルにおけるクラス（つまり Class のインスタンス）に定義された属性や操作の情報を実際に格納する場合を考えると、要素 Foundation.Core.Class の子要素である要素 Foundation.Core.Classifier.feature は、その子要素として、要素 Foundation.Core.Attribute や要素 Foundation.Core.Operation を持ち、ここに属性や操作の情報を格納する．

```
477 <!ELEMENT Foundation.Core.Classifier.feature (
478     Foundation.Core.Feature |
479     Foundation.Core.StructuralFeature |
480     Foundation.Core.Attribute |
481     Foundation.Core.BehavioralFeature |
482     Foundation.Core.Operation |
483     Foundation.Core.Method |
484     Behavioral_Elements.Common_Behavior.Reception)*
485     >
```

図 2.18: *Classifier* のナビゲーション先ロール feature の要素型定義

第3章　UMLメタモデルとOCL文法の範囲

本研究でおこなう整合性検査は、構文面の整合性検査である．不変表明は、UML メタモデルに対して与え、OCL で記述する．不変表明が構文面のものとなるように、対象とするUML メタモデルと OCL 文法の範囲について検討する．

3.1　UMLメタモデルの範囲

UML のメタモデルは、モデルの記法を定義するものであるので、UML メタモデルのクラスを不変表明のコンテキストにとった OCL 式は、必然的に構文面の不変表明となる．

OCL では、ラベル pre:、post:を用いて、コンテキストのクラスの操作やメソッドに対する事前条件、事後条件を記述することもできる．このような OCL 式に対する検査をおこなうには、意味にまで踏み込む必要がある．しかし、UML メタモデルのクラスには、操作の定義が存在しないので、その必要がない．

本システムでは、全ての UML メタモデルを対象とする．図 3.1 に対象とする UML メタモデルの範囲の概念図を示す．

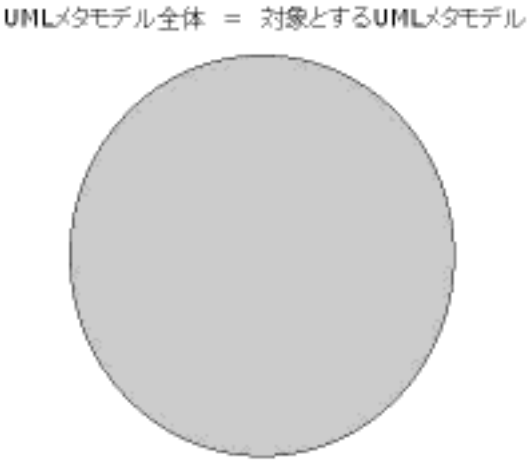

図 3.1: UML メタモデルの範囲

3.2 OCL 文法の範囲

上で述べたように、UML メタモデルのクラスには、操作の定義が存在しない．そのため、操作の事前条件（pre:）、事後条件（post:）に関する文法を対象外とする．図 3.2 に対象とする OCL 文法の範囲の概念図を示す．OCL の文法規則については、4.2 OCL 文法の範囲制限にて説明する．

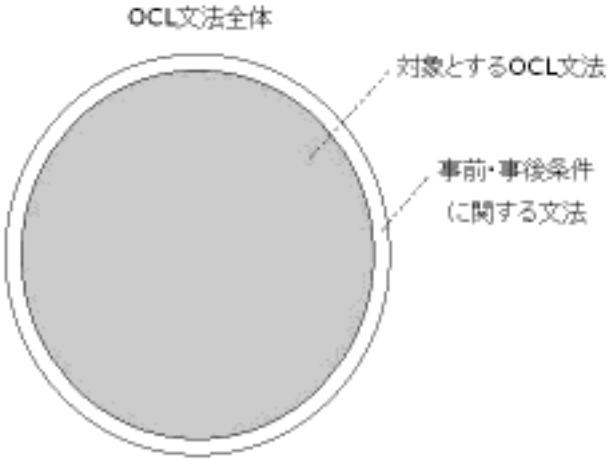

図 3.2: OCL 文法の範囲

第4章　整合性検査の手法

4.1　整合性検査システムの概要

本研究では、実用性・汎用性の高い整合性検査システムの実現を目指す．図 4.1 に本整合性検査システムの概略図を示す.

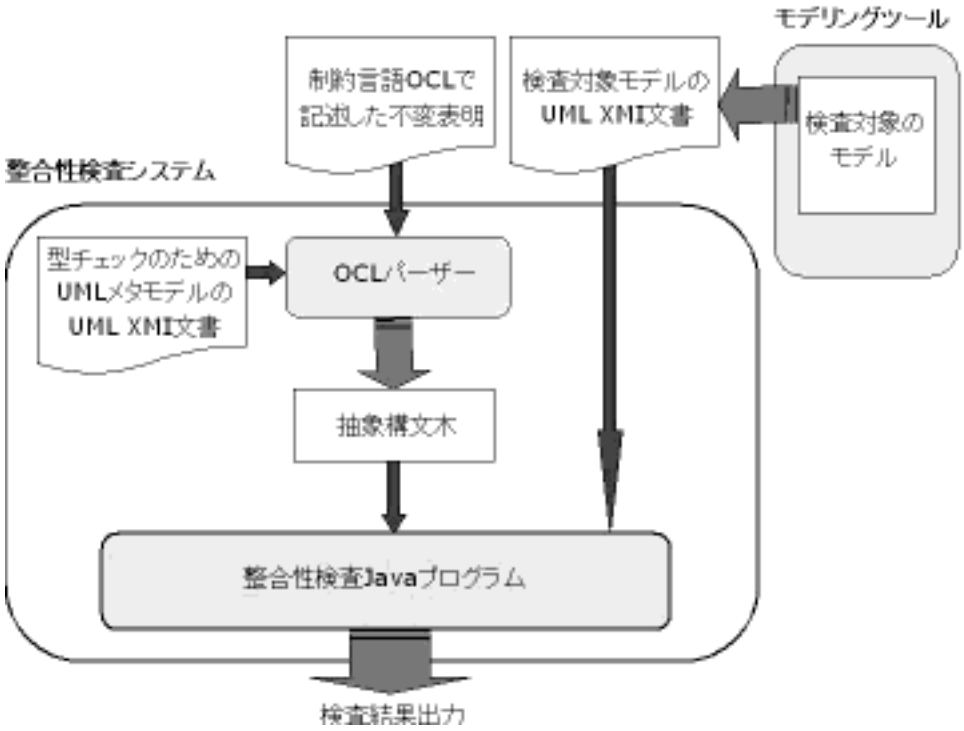

図 4.1: 整合性検査システム概略図

システムへの入力は３つある．１つは検査対象のモデルであり、モデリングツールから自動生成可能な UML XMI 文書のモデルとする．２つ目は、検査すべき不変表明であり、OCL で記述した UML メタモデルに対する不変表明とする．３つ目は、OCL パーザーが型チェックをおこなうため、予めシステムに与えておく UML メタモデルの UML XMI 文書である.

整合性検査システムは、まず与えられた OCL 不変表明を OCL パーザーが構文解析お

こない、抽象構文木を生成する．抽象構文木生成時、OCL パーザーは、予め与えられた UML メタモデルの UML XMI 文書を用いて、OCL 式の型チェックもおこなう．次に、生成された抽象構文木と検査対象モデルの UML XMI 文書を入力として、整合性検査 Java プログラムが整合性検査をおこない、検査の結果を出力する．

4.2 OCL 文法の範囲制限

前章で述べたように、本システムが対象とする OCL 文法は、操作の事前条件（pre:)、事後条件（post:）に関する文法を除いたものである．

しかし、本研究では、対象とする OCL 文法の範囲をさらに制限する．まず、制限した OCL 文法に対して、整合性検査の手法を確立する．その後、対象とするすべての OCL 文法に対して、整合性検査手法を確立するというアプローチをとる．

4.2.1 範囲制限の考え方

対象とする OCL 文法の範囲を制限する．しかし、ただ闇雲に OCL 文法の範囲を制限する訳にはいかない．OCL 式の基本的な構造、機能を損なわないように注意する必要がある．以下に、OCL 文法の範囲を制限する際の考え方を示す．

- プロパティの参照、およびナビゲーションが可能であること
- 基本的なプロパティ操作、およびコレクション操作が可能であること
- 再帰的な構造を持つ OCL 式が書けること

4.2.2 制限した OCL 文法の範囲

上記考え方に基づいて制限した、プロパティ操作、コレクション操作、および OCL 文法規則を以下に示す．

- プロパティ操作
  - object.oclAsType(type : OclType)
    オブジェクト object の実際の型が下位型であることが確実な場合、object を引数の型 type に再型付けする．下位型のプロパティ、およびナビゲーション先ロール名にアクセスしたい場合、この操作を用いる．
  - object.oclIsTypeOf(type : OclType)
    オブジェクト object の型が引数の型 type に等しい場合、真を返す．

- コレクション操作
  - `collection->size`
    コレクション collection の中の要素の数を返す.
  - `collection->exists( v : Type | boolean-expression-with-v )`
    `collection->exists( v | boolean-expression-with-v )`
    `collection->exists( boolean-expression )`
    コレクション collection の中の少なくとも 1 つの要素に対して、引数の ocl 式の評価が真の場合、真を返す.
  - `collection->forAll( v : Type | boolean-expression-with-v )`
    `collection->forAll( v | boolean-expression-with-v )`
    `collection->forAll( boolean-expression )`
    コレクション collection の中のすべての要素に対して、引数の ocl 式の評価が真の場合、真を返す.
  - `collection->collect( v : Type | expression-with-v )`
    `collection->collect( v | expression-with-v )`
    `collection->collect( expression )`
    引数の ocl 式をコレクションのすべての要素に適用することから生じる要素のコレクションを返す.
  - `collection->select( v | boolean-expression-with-v )`
    `collection->select( boolean-expression )`
    `collection->select( v : Type | boolean-expression-with-v )`
    `collection->select( v | boolean-expression-with-v )`
    `collection->select( boolean-expression )`
    引数の ocl 式が真となる部分コレクションを返す.

- OCL 文法規則
  制限した OCL 文法規則を以下に示す（フルセットの OCL 文法規則は、付録 C OCL の文法規則に示した）. 文法の記述は、EBNF 構文を用いている. ここで、“ | ”は選択、“ ? ”はオプション、“ * ”はゼロ回以上、“ + ”は 1 回以上、を意味する. name、typeName、および string の記述では、JavaCC パーザー生成器からの字句トークンに対する構文を示す.

```
constraint              := contextDeclaration
                           (stereotype name? “ : ” expression)
contextDeclaration      := “ context ”classifierContext
classifierContext       := <typeName>
stereotype              := “ inv ”
```

```
expression              := logicalExpression
logicalExpression       := relationalExpression
relationalExpression    := additiveExpression
                           ( relationalOperator additiveExpression )?
additiveExpression      := multiplicativeExpression
multiplicativeExpression:= unaryExpression
unaryExpression         := postfixExpression
postfixExpression       := primaryExpression ( ("." | "->") featureCall )*
primaryExpression       := literalCollection
                           | literal
                           | pathName featureCallParameters?
featureCallParameters   := "(" ( declarator )?
                           ( actualParameterList )? ")"
literal                 := <STRING> | <number>
simpleTypeSpecifier     := pathTypeName
featureCall             := pathName featureCallParameters?
declarator              := <name> ( ":" simpleTypeSpecifier )? "|"
pathTypeName            := <typeName>
pathName                := ( <typeName> | <name> )
actualParameterList     := expression
relationalOperator      := "=" | ">" | "<" | ">=" | "<=" | "<>"
typeName                := ( [“ a ”- ”z ”] | ["A"-"Z"] | “ _ ” )
                           ( ["a"-"z"] | ["0"-"9"] | ["A"-"Z"] | "_")*
name                    := ( [“ a ”- ”z ”] | ["A"-"Z"] | “ _ ” )
                           ( ["a"-"z"] | ["0"-"9"] | ["A"-"Z"] | "_")*
number                  := ["0"-"9"] (["0"-"9"])*
string                  := "'" ( (~["'","\\","\n","\r"])
                           | ("\\"
                           ( ["n","t","b","r","f","\\","'","\""]
                           | ["0"-"7"] ( ["0"-"7"] )?
                           | ["0"-"3"] ["0"-"7"] ["0"-"7"]
                           )))*
                           "'"
```

4.3 OCL パーザー

本システムでは、OCL パーザーは、ドレスデン工科大学が開発した OCL パーザー、OCL compiler を用いる [9]．OCL compiler は、OCL 式を構文解析して抽象構文木を生成するだけでなく、OCL 式の型チェック、OCL 式のノーマライゼーションなどの機能を持つ．以下、OCL compiler について説明する．

4.3.1 OCL compiler

OCL compiler は、SableCC という Java 言語のコンパイラコンパイラ（LALR(1)）を用いて生成した OCL コンパイラである [10]．SableCC は、文法規則にアクションコードを用いない代わりに、パーザーと Tree Walkers のコンパイラフレームワークを提供する．

パーザーが生成する抽象構文木を構成するオブジェクトのクラスは、文法規則から自動的に生成される．Tree Walkers は、抽象構文木を走査するために生成された複数のクラスである．Tree Walkers を用いることにより、抽象構文木における様々な処理が可能である．

4.3.2 OCL compiler の OCL 文法規則

4.2.2 制限した OCL 文法の範囲で示した EBNF 構文の OCL 文法規則に対する、OCL compiler の文法規則（LALR(1)）を以下に示す（OCL compiler のフルセットの OCL 文法規則については、付録 D OCL compiler の文法規則に示した）．これらの文法規則から、抽象構文木のノードであるオブジェクトのクラスが、SableCC によって自動的に生成される（下記文法規則の右辺の非終端記号、または終端記号の前に付けられた'{''}' 内の名前は、SableCC が生成するクラスための名前である）．

```
Helpers

  uppercase = ['A'..'Z'];
  lowercase = ['a'..'z'];
  digit = ['0'..'9'];
  number = digit+;


Tokens

  dot = '.';
  arrow = '->';

  context = 'context';
  t_inv = 'inv';
```

```
  equal = '=';
  n_equal = '<>';
  lt = '<';
  gt = '>';
  lteq = '<=';
  gteq = '>=';

  l_par = '(';
  r_par = ')';

  colon = ':';
  comma = ',';
  bar = '|';
  apostroph = ''';

  bool = 'true' | 'false';
  simple_type_name =
  ( uppercase (lowercase | digit | uppercase | '_')* )
  name = lowercase (lowercase | digit | uppercase | '_')*;

  int = number;
  real = number '.' number;


Productions

  constraint =
    context_declaration constraint_body ;

  constraint_body =
    stereotype name? colon expression ;

  context_declaration =
    context context_body ;

  context_body =
    {classifier} classifier_context |
    {operation} operation_context ;

  classifier_context =
    type_name ;

  stereotype =
    {inv} t_inv ;

  expression =
```

```
  logical_expression ;

logical_expression =
  relational_expression ;

relational_expression =
  additive_expression relational_expression_tail? ;

relational_expression_tail =
  relational_operator additive_expression ;

additive_expression =
  multiplicative_expression ;

multiplicative_expression =
  unary_expression ;

unary_expression =
  {postfix} postfix_expression ;

postfix_expression =
  primary_expression postfix_expression_tail* ;

postfix_expression_tail =
  postfix_expression_tail_begin feature_call ;

postfix_expression_tail_begin =
  {dot} dot |
  {arrow} arrow ;

primary_expression =
  {literal} literal |
  {feature} path_name feature_call_parameters? ;

feature_call_parameters =
  l_par declarator? actual_parameter_list? r_par ;

literal =
  {integer} int |
  {boolean} bool ;

simple_type_specifier =
  {path} path_type_name ;

feature_call =
  path_name feature_call_parameters? ;
```

```
declarator =
  {standard} name declarator_type_declaration? bar ;

declarator_type_declaration =
  colon simple_type_specifier ;

path_type_name =
  type_name ;

type_name =
  {non_collection} simple_type_name ;

path_name =
  path_name_begin ;

path_name_begin =
  {type_name} type_name |
  {name} name ;

actual_parameter_list =
  expression ;

relational_operator =
  {equal} equal |
  {n_equal} n_equal |
  {gt} gt |
  {lt} lt |
  {gteq} gteq |
  {lteq} lteq ;
```

4.3.3 抽象構文木

抽象構文木のノードであるオブジェクトのクラスは、OCL 文法規則に基づいて、SableCC により自動的に生成される．全ての文法規則 production_name からは、抽象クラス PProductionName が生成される．全ての文法規則の右辺 alternative からは、具象クラス AAlternativeProduction-Name が抽象クラス PProductionName のサブクラスとして生成される．また、すべての終端記号 token_name からは、クラス TTokenName が生成される．例として、図 4.2 に文法規則 post-fix_expression から生成されたクラスを示す.

4.3.4 Tree Walkers

Tree Walkers は、オブジェクト指向のデザインパターンの１つである“ visitor ”パターンを用いアプリケーションである．Tree Walkers には、抽象構文木を深さ優先で文法規則の左から右に走

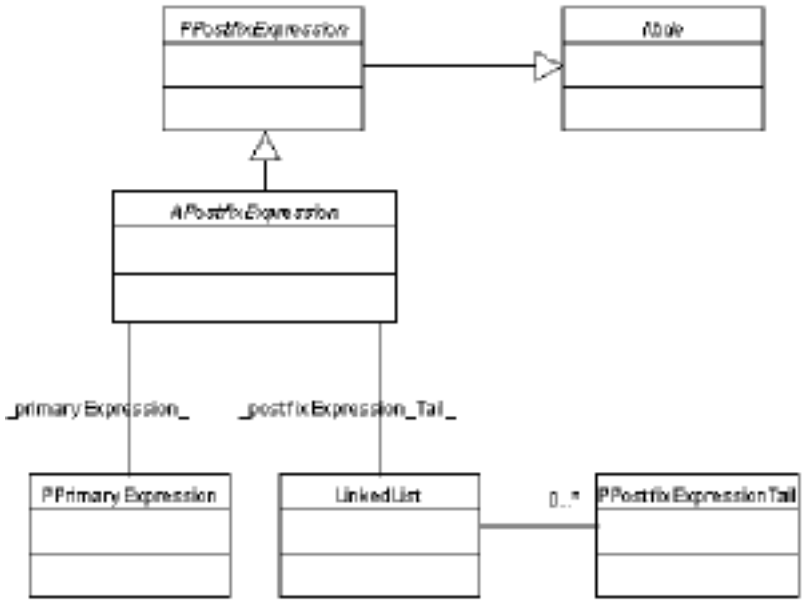

図 4.2: 文法規則 postfix_expression から生成されたクラス

査するクラス DepthFirstAdapter と、右から左に走査するクラス ReverseDepthFirstAdapter の２つのクラスがある．これら Tree Walkers クラスは、“ visitor ”クラス AnalysisAdapter をサブクラス化したものである．図 4.3 に Tree Walkers における visitor パターンのクラス階層を示す．

インターフェース Analysis は、全ての文法規則の右辺 alternative、および終端記号 token に対するメソッド caseXxx を持つ．このインターフェース Analysis を実装したクラス AnalysisAdapter（visitor クラス）は、すべてのメソッドで空のメソッド defaultCase を呼び出す．

Tree Walkers は、クラス AnalysisAdapter をサブクラス化したものである．Tree Walkers のクラスのすべてのメソッド caseXxx は、抽象構文木の走査をおこなうように、またメソッド inXxx、outXxx を呼び出すようにオーバーライドされている（token のメソッド caseXxx は、オーバーライドされていない）．そして、メソッド inXxx、outXxx はそれぞれ、空のメソッド defaultIn、defaultOut を呼び出す．

また、クラス AnalysisAdapter は、抽象構文木の走査時にデータの受け渡しをおこなうために、２つのハッシュテーブルを用意している．これらのハッシュテーブルは、以下のメソッドでアクセスできる．

```
Object getIn(Node n)
Object getOut(Node n)
void setIn(Node n, Object o)
void setOut(Node n, Object o)
```

4.3.5 型チェック

2.2.3 OCL の型で説明したように、OCL には、定義済みの型と UML モデルからの型の２つがある．OCL compiler は、これら２つの型に対して型チェックをおこなう．

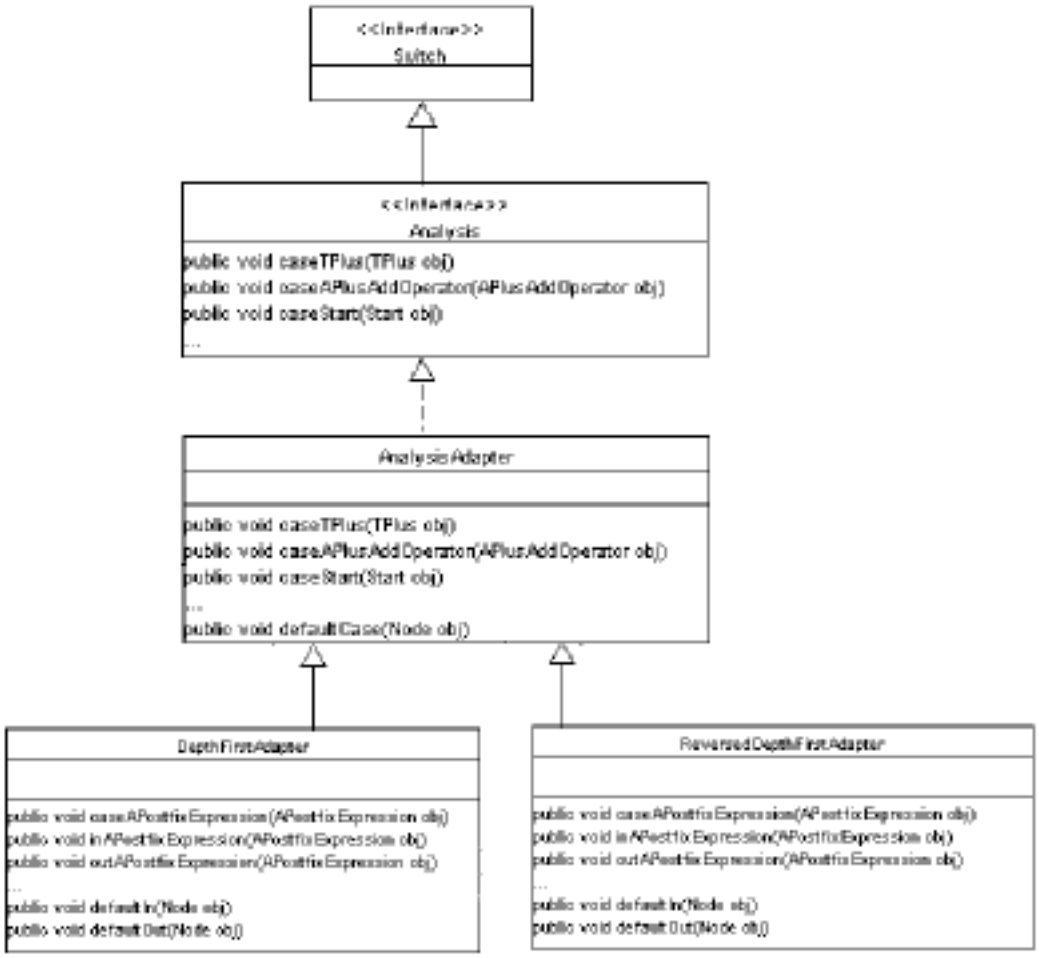

図 4.3: Tree Walkers における visitor パターン

UML モデルからの型に対する型チェックをおこなうためには、UML モデルの型情報を与える必要がある．OCL compiler では、型情報を UML XMI 文書として与えることができる．但し、UML XMI 文書の仕様は、モデリングツール ArgoUML から自動生成されたものである．

4.3.6 ノーマライゼイション

ノーマライゼイションは、その後処理をおこないやすくするために、抽象構文木に対して施す処理である．ノーマライゼイションには、選択可能な７つの処理が用意されている．これらの内、本システムで重要な３つの処理について説明する．

- DefaultContextInsertion
  OCL では、大抵の場合、コンテキストのインスタンスを参照するキーワード self は、省略可能である．DefaultContextInsertion は、この省略されたキーワード self を補う．

```
context Person inv: age > 10 ⟹  context Person inv: self.age > 10
```

- IteratorInsertion
  コレクションのイテレータ操作では、イテレータの宣言、または引数となる OCL 式のイテレータを省略することが可能である．IteratorInsertion は、この省略されたイテレータを補う（イテレータ宣言がない場合、ユニークな名前のイテレータが補われる）．

```
context Company inv: self.employee->forAll(isUnemployed = false)
⟹  context Company
    inv: self.employee
         ->forAll(tudOclIter0 | tudOclIter0.isUnemployed = false)
```

- ValiableClarification
  OCL 式に現れるすべての変数（イテレータなど）が、ユニークな名前であった方が、その後の処理がおこないやすくなる場合がある．ValiableClarification は、OCL 式に現れるすべての変数をユニークな名前に付け替える．

```
context Company
   inv: self.employee
        ->select(e | e.age < 25)
        ->exists(e | e.isMarried = true)

⟹  context Company
    inv: self.employee
         ->select(e | e.age < 25)
         ->exists(tudOclIter0 | tudOclIter0.isMarried = true)
```

4.4 コレクションの実現方法

実際に、UML モデルに対して、OCL 式の評価をおこなうためには、評価すべきオブジェクトを格納するコレクションをどのように実現するのかを考える必要がある．

本節では、考案したコレクションのモデルを説明する．説明を分かりやすくするために、まず、OCL 式のイテレータのスコープ内に他のイテレータが存在しない場合でモデルを説明し、後に、イテレータのスコープ内に他のイテレータが存在する場合でモデルを説明する．

4.4.1 イテレータのスコープ内に他のイテレータが存在しない場合

まず、イテレータのスコープ内に、他のイテレータが存在しない場合で、考案したコレクションのモデルについて説明する．

本システムにおける整合性検査の処理は、OCL パーザーが生成した抽象構文木を、深さ優先かつ最左優先で 1 回のみ走査する間にすべておこなう（1 パス、インタプリタ方式）．

以下に、説明で用いる OCL 式の例を示す．まず、コンテキスト宣言まで読み終えた場合を考える．OCL 式にある記号'^' が、現時点で読み終えた位置を表している（OCL 式を左から右へ読んだ後、右から左へ読んで戻ることと、抽象構文木を 1 回走査することは同じである）．

```
context Company inv: self.employee->forAll(e | e.isUnemployed = false)
                ^
```

ここでは、コンテキスト Company のすべてのオブジェクトを UML モデルから集め、すべてのオブジェクトを１つの集合（コレクション）として格納する．この時のオブジェクトの状態を図 4.4 に示す．

図 4.4: 左端から context Company まで読んだ時の状態

次に、左端から不変表明のラベル inv:の後の self まで読んだ場合を考える．以下に OCL 式を、図 4.5 にオブジェクトの状態を示す．self のコレクションは、各 Company オブジェクトに対して、その Company オブジェクトと同一のオブジェクトを 1 つだけ要素として含むコレクションとなる．図中の破線矢印は、オブジェクトの参照を表す．self の各コレクションが評価された場合、参照先オブジェクトへ結果の真理値が返される．

```
context Company inv: self.employee->forAll(e | e.isUnemployed = false)
                     ^
```

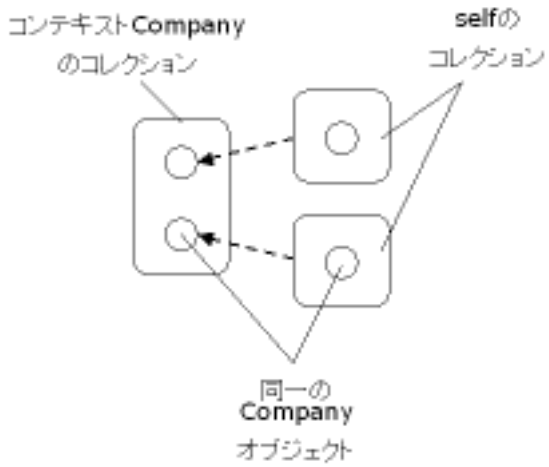

図 4.5: 左端から self まで読んだ時の状態

次に、左端から self の後の employee まで読んだ場合を考える．以下に OCL 式を、図 4.6 にオブジェクトの状態を示す．各 employee コレクションは、1 つの self オブジェクトからナビゲーションして得られたすべての empoloyee オブジェクトを格納したものである．

```
context Company inv: self.employee->forAll(e | e.isUnemployed = false)
                                ^
```

次に、左端から forAll 操作のイテレータ宣言 e まで読んだ場合を考える．以下に OCL 式を、図 4.7 にオブジェクトの状態を示す．ここで、イテレータ変数 e に、イテレータ変数 e が参照するオブジェクトを格納する、すべてのコレクションを対応付ける．これは、イテレータが、イテレータ宣言のあるコレクション操作の引数スコープ内の、任意の場所で用いられるからである．

```
context Company inv: self.employee->forAll(e | e.isUnemployed = false)
                                          ^
```

次に、左端からイテレータ宣言 e の後のイテレータ変数 e まで読んだ場合を考える．これは、コンテキスト Company の集合から、self のコレクションを生成した場合と同じである．以下に OCL 式を、図 4.8 にオブジェクトの状態を示す．

```
context Company inv: self.employee->forAll(e | e.isUnemployed = false)
                                              ^
```

次に、左端からイテレータ宣言 e の後の e.isUnemployed まで読んだ場合を考える．これは、self オブジェクトからナビゲーションをして、employee コレクションを生成した場合と同じである．以下に OCL 式を、図 4.9 にオブジェクトの状態を示す．

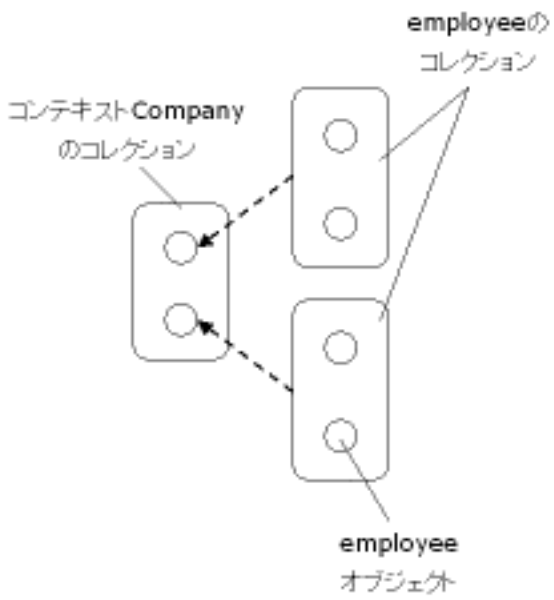

図 4.6: 左端から self.employee まで読んだ時の状態

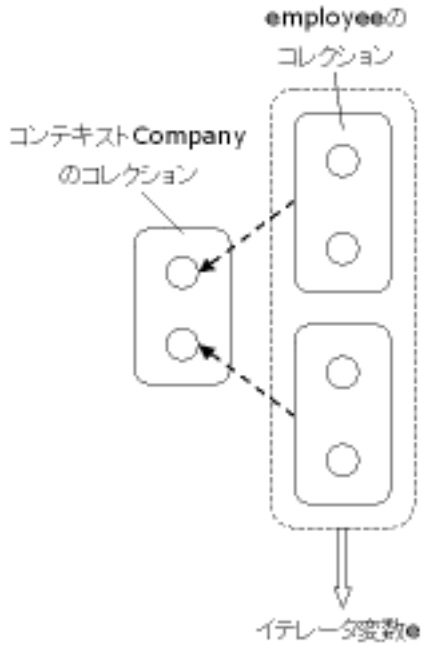

図 4.7: 左端からイテレータ宣言 e まで読んだ時の状態

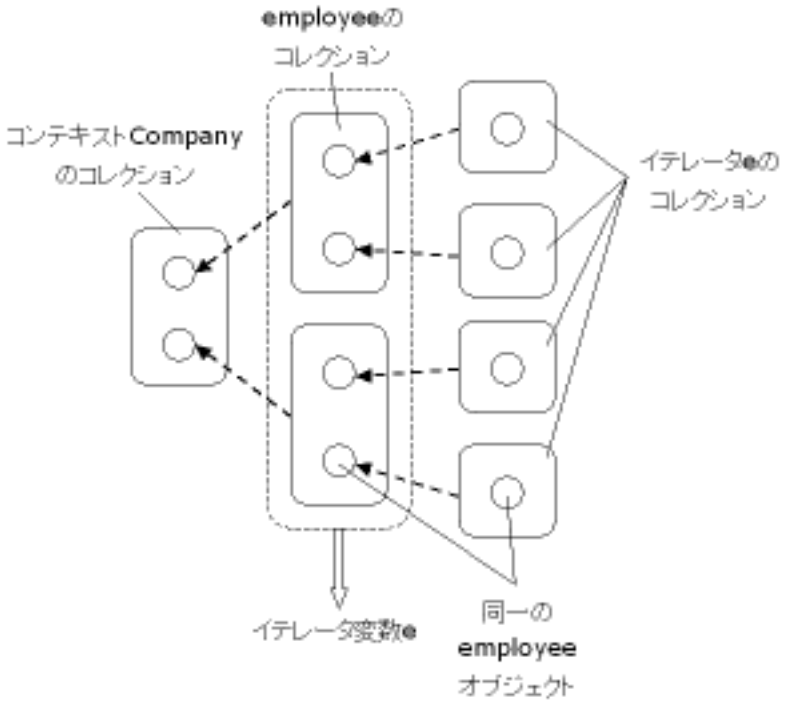

図 4.8: 左端からイテレータ宣言 e の後の e まで読んだ時の状態

```
context Company inv: self.employee->forAll(e | e.isUnemployed = false)
                                                ^
```

次に、左端から OCL 式の右端まで読んだ後、イテレータ宣言 e の前まで戻った場合を考える．以下にその OCL 式を示す．ここでは、employee オブジェクトのプロパティである isUnemployed オブジェクトに対する評価をおこなう．評価結果は、評価された isUnemployed オブジェクトに持たせる．そして、さらに評価結果を isUnemployed オブジェクトのコレクションが参照するオブジェクトへ返す．図 4.10 にこの時のオブジェクトの状態を示す．図中の黒く塗りつぶしたオブジェクトは、評価済みのオブジェクトであることを示す．

```
context Company inv: self.employee->forAll(e | e.isUnemployed = false)
                                               ^
```

次に、右端から forAll 操作まで読んだ場合を考える．以下にその時の OCL 式を示す．ここでは、forAll 操作が、イテレータ変数 e に対応付けられたすべてのコレクションに対し、評価をおこなう．forAll 操作は、コレクションの中のすべてのオブジェクトの評価結果が真であれば真を、そうでなければ偽を返す．評価した結果は、評価されたコレクションが参照するオブジェクトへ返す．図 4.11 にこの時のオブジェクトの状態を示す．この時点で、コンテキストのすべての Company オブジェクトが評価されている．

```
context Company inv: self.employee->forAll(e | e.isUnemployed = false)
                                             ^
```

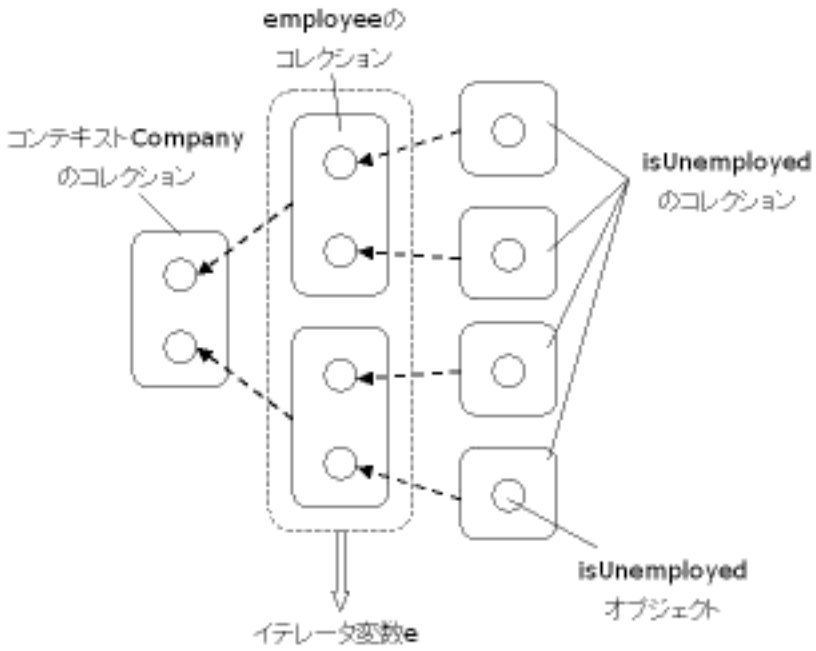

図 4.9: 左端から e.isUnemployed まで読んだ時の状態

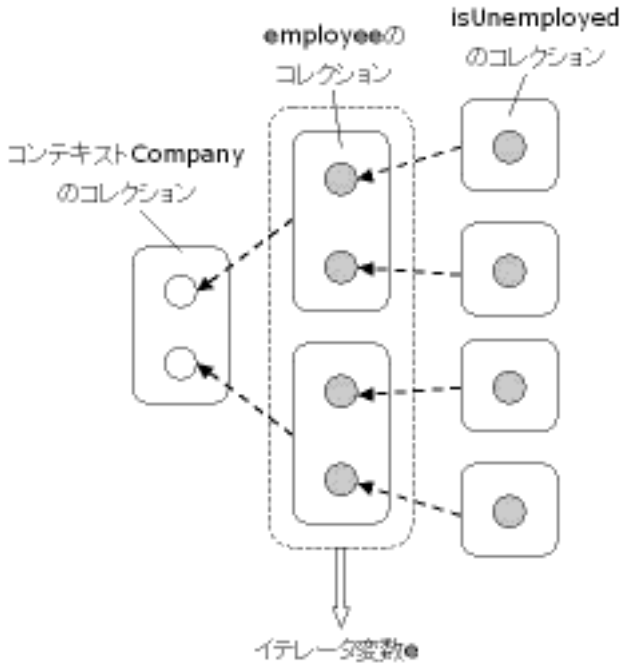

図 4.10: 右端からイテレータ宣言 e の前まで読んだ時の状態

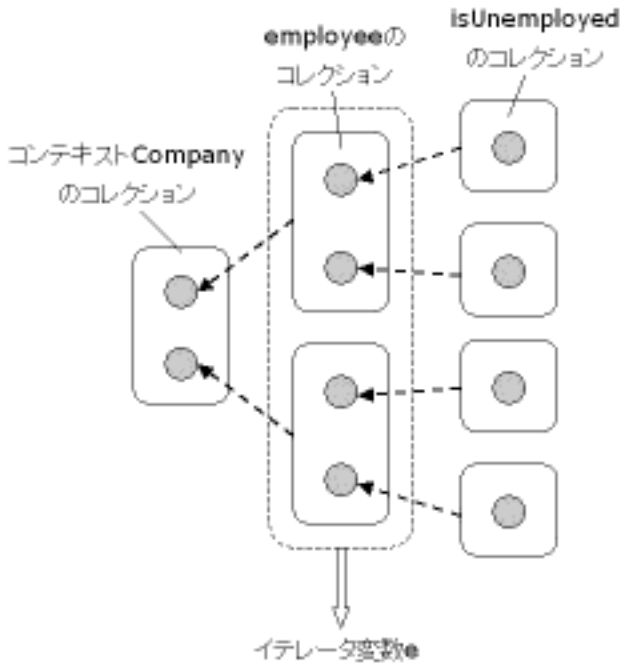

図 4.11: 右端から forAll 操作まで読んだ時の状態

この後、OCL 式の左端まで戻り、最終的にコンテキストの集合に対する評価をおこなう．先に述べた forAll 操作と同様に、コンテキストの中のすべてのオブジェクトの評価結果の論理積をとり、最終的な整合性検査の結果として出力する．

4.4.2 イテレータのスコープ内に他のイテレータが存在する場合

イテレータのスコープ内に、他のイテレータが存在する場合で、考案したコレクションのモデルについて説明する．例として、以下のような OCL 式の構造を考える．この時のオブジェクトとコレクションの状態を図 4.12 に示す．

```
->forAll(a | a. ･･･
            ->exists(b | b. ･･･
                        ->forAll(c | c = a)))
```

上記 OCL 式における最後の forAll 操作では、イテレータ変数 c が参照するオブジェクトとイテレータ変数 a が参照するオブジェクトが同一のオブジェクトであるかどうかを評価し、イテレータ変数 c のオブジェクトに結果を返す．しかし、イテレータ変数 a のどのオブジェクトと比較をすべきなのかがわからなければ、評価することができない．

そのため、各コレクションに、参照先のオブジェクトの情報を格納するテーブルを用意する．たとえば、図中のオブジェクト C1 に対して、イテレータ変数 a のオブジェクトと同一であるかどうかの評価をおこなう場合、テーブルを用いて、イテレータ変数 a とオブジェクト A1 の対応関係がわかれば、評価することができる．以下に、コレクション C1 が持つべきテーブルのタプルを示す．

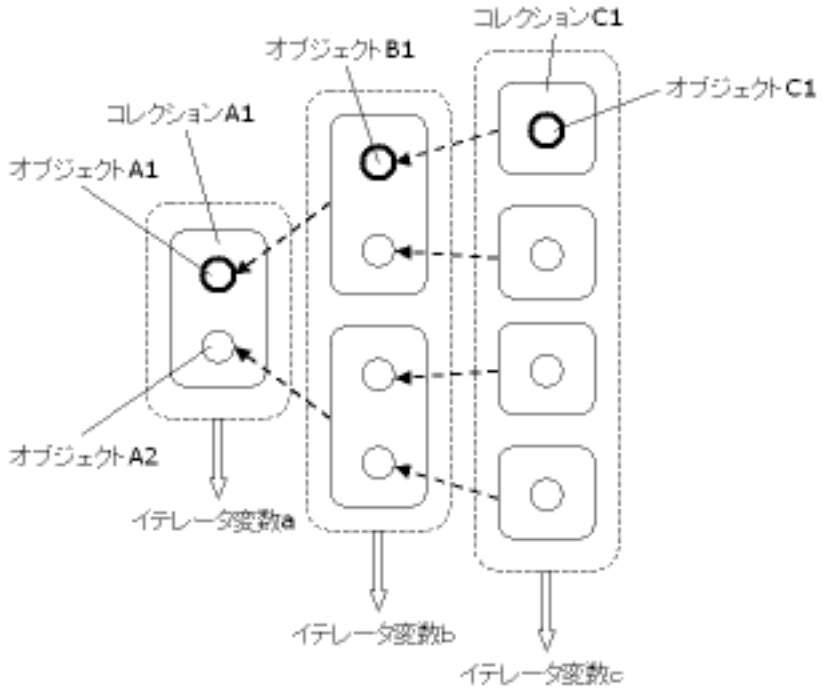

図 4.12: イテレータのスコープ内に他のイテレータが存在する場合の状態

```
a, A1
b, B1
```

4.5 UML XMI 文書からのオブジェクト生成方法

本システムでは、検査対象の UML モデルは、UML XMI 文書に格納されたモデルである．そのため、OCL 式の評価で用いるオブジェクトは、UML XMI 文書から生成された DOM ツリーから取得したノードとなる．また、取得するノードの種類は、UML メタモデル上のクラスのインスタンス、またはクラスのプロパティ（属性値）のノードである．

オブジェクトを生成する方法は、2 つに分けられる．1 つは、コンテキストのクラスのインスタンスであるオブジェクトを生成する場合である．もう 1 つは、プロパティ、またはナビゲーションロールのオブジェクトを生成する場合である．以下、それぞれの場合について、下記の不変表明の例を用いて説明する．

```
context Class
inv: self.feature->exists(f | f.oclIsTypeOf(Operation))
```

- コンテキストクラスのオブジェクト生成方法
  前述したように、OCL 式の評価では、まずコンテキストとなるクラスのオブジェクトを生成する．上の不変表明の例では、コンテキストクラスは Class である．このクラス名 Class

を用いてオブジェクトを生成するには、まず UML XMI 文書におけるクラス Class のタグ名を知らなければならない．そのため、以下に示すように、クラス名とタグ名を対応付けるテーブルを用意する．

```
Class, Foundation.Core.Class
```

テーブルから得たタグ名を用いて、実際に DOM ツリーからノードを取得するにはさまざまな方法がある．しかし、XPath を用いれば、簡単にノードを取得することができる（Xalan が XPath API を実装 [11]）．以下に、タグ名 Foundation.Core.Class のノードを取得するための XPath 式を示す．

```
//Foundation.Core.Class[@xmi_id]
```

- プロパティ参照、またはナビゲーション時のオブジェクト生成方法
  コンテキストクラスのオブジェクト生成方法と同様に、プロパティ名、またはナビゲーション先ロール名から、そのタグ名を取得する必要がある．
  しかし、ここで注意しなければならないのは、UML メタモデルにおいては、同じ名前のナビゲーション先ロール名が、メタモデル上の複数の関連で用いられている場合があることである．つまり、ロール名だけでは、対応するタグ名が決められない．そのため、ナビゲーション元オブジェクトのクラス名と、ナビゲーション先ロール名を用いて、タグ名を取得する．
  テーブルは、ナビゲーション先ロール名またはプロパティ名と、そのタグ名、およびプロパティのタグかどうかの真理値を 1 つのタプルとしたものとする．このテーブルを各クラス毎に用意する．このテーブルからタグ名を取得するには、ナビゲーション先ロール名またはプロパティ名と、ナビゲーション元オブジェクトのクラス名の 2 つを用ればよい．
  以下に、クラス Class のテーブルにおけるプロパティ名 name、およびナビゲーション先ロール名 feature のタプルを示す．

```
name, Foundation.Core.ModelElement.name, true
feature, Foundation.Core.Classifier.feature, false
```

また、テーブルから取得したナビゲーション先ロールのタグ名 Foundation.Core.Classifier.feature を用いて、オブジェクトを取得するための XPath 式を以下に示す．タグ名 classTagName は、ナビゲーション元オブジェクトのタグ名であり、属性 `xmi_id` の値とともに、ナビゲーション元オブジェクトから取得する．

```
//classTagName[xmi_id = 'xxxx']/Foundation.Core.Classifier.feature
```

プロパティではなく、ナビゲーションロールでオブジェクトを取得した場合では、さらに、取得したオブジェクトの子ノードであるクラスのオブジェクトをすべて取得して、それを新たなコレクションの要素とする．また、子ノードであるクラスのオブジェクトを取得するには、子ノードの内、要素型で、属性値 `xmi_id`、または `xmi_idref` を持つものを選べばよい．

上で述べてきたように、OCL 式の評価で用いるオブジェクトは、UML XMI 文書から生成された DOM ツリーのノードである．しかし、オブジェクトを OCL 式の評価で用いるためには、オブジェクトが評価されたかどうかを判定する真理値、評価結果の真理値など、いくつかのプロパティを持たせたる必要がある．そのため、実際には、DOM ノードをラップしたものとなる．

4.6 抽象構文木の走査方法

本システムでは、OCL パーザーとして、4.3 OCL パーザーで説明した OCL Compiler を用いる．モデルに対する整合性検査は、パーザーが生成した抽象構文木を、深さ優先かつ最左優先で、1 回のみ走査する間で、すべておこなう．

抽象構文木を走査するために、SableCC のコンパイラフレームワークとして提供される Tree Walkers のクラス DepthFirstAdapter をサブクラス化したクラスを用いる．

走査時におこなう整合性検査の処理は、そのノードに入った直後と、出る直前におこなう．おこなう処理は抽象構文木のノードの種類（非終端記号）毎に記述できる．本論文では、考案した各ノード毎での具体的な処理内容は示さない．

4.7 モデリングツールに依存する問題

現在、いくつかのモデリングツールが、作成したモデルからの UML XMI 文書の自動生成をサポートしている．しかし、1 つのモデルに対して、さまざまな UML XMI 文書が生成され得る．たとえば、以下のような問題がある．

たとえば、図 4.13 の上段に示すように、メタモデルにおけるクラス A とクラス B が、互いに関連を持っていたとする．そして、このメタモデルに対するインスタンスを、ユーザーがモデリングツールで作成した場合を考える．

もし、このモデルから、正しく UML XMI 文書を自動生成がされたならば、生成された UMI XMI 文書において、クラス A、およびクラス B に関する UML XMI 文書の要素（タグ）は、以下のようになるはずである．

クラス A

```
<ClassA>
  <roleB>
    <ClassB>
    </ClassB>
  </roleB>
</ClassA>
```

クラス B

```
<ClassB>
  <roleA>
    <ClassA>
    </ClassA>
  </roleA>
</ClassB>
```

しかし、片方のクラスについて、メタモデルにおける関連のロールに関するタグが欠落している場合がある．たとえば、クラス B で欠落があった場合の UML XMI 文書を以下に示す．

クラス A

```
<ClassA>
  <roleB>
    <ClassB>
```

クラス B

```
<ClassB>
</ClassB>
```

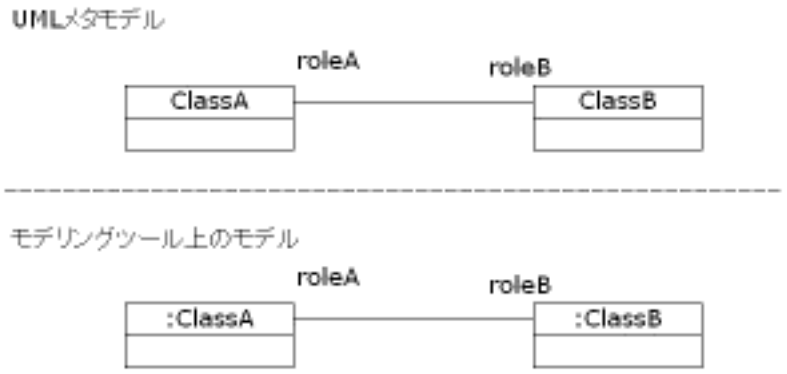

図 4.13: メタモデルと対応するモデルの例

```
    </ClassB>
  </roleB>
</ClassA>
```

この時のメタモデル上の関係をクラス図を用いて表すと、以下の図 4.14 のようになる.

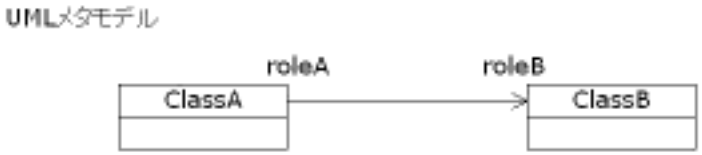

図 4.14: 片方向の関連が欠落したメタモデルの例

上記のように、メタモデル上の片方向の関連が欠落していても、モデリングツールがメタモデルの情報を持っていれば、UML XMI 文書のモデルから、モデリングツール上のモデルへと再現することができる.

しかし、本整合システムの場合、情報が欠落している場合とそうでない場合で、検査の結果が変わってくる. そこで、以下のアルゴリズムを用いて、欠落した情報の補完をおこなう.

- UML XMI 文書の欠落した情報を補完するアルゴリズム
  - もし、UML XMI 文書に以下のような構造があったら、

    クラス A

    ```
    <ClassA>
      <roleB>
        <ClassB>
        </ClassB>
      </roleB>
    </ClassA>
    ```

* UML XMI 文書のクラス B に関するタグに、以下のような構造があるかどうかをチェックする.
  - もし、以下のような構造があったら、何もしない.
  - もし、以下のような構造がなかったら、タグの補完をおこなう.
    クラス B

```
<ClassB>
  <roleA>
    <ClassA>
    </ClassA>
  <roleA>
</ClassB>
```

第5章　整合性検査システムの実装および評価

5.1　システムの実装

第 4 章で述べた整合性検査の手法に基づき、整合性検査システムを実装した．以下、実装した整合性検査システムについて説明する．

5.1.1　システムの入力

本システムには、３つの入力がある．

- OCL 不変表明
  検査をおこないたい OCL の不変表明をユーザーが独自に記述できる．javax.swing で実装したツールのテキストエリアから入力する．
- 検査対象モデルの UML XMI 文書
  検査対象とするモデルの UML XMI 文書は、Rational Rose により自動生成した UML XMI 文書を入力とする．(Rational Rose2001A でモデルを作成した後、Rational Rose のアドイン、Unisys Rose/XMI interchange package（UML XMI DTD(UML1.1)）を用いて自動生成したもの）．
- 型チェックのための UML メタモデルの UML XMI 文書
  本システムの OCL パーザーは、後述するドレスデン工科大学が開発した OCL compiler を用いている．そのため、UML XMI 文書の仕様は、OCL compiler がサーポートしているモデリングツール ArgoUML[12] から自動生成したものとなる（本実装では、ArgoUML ベースのモデリングツール IIOSS を用いてメタモデルを作成した．IIOSS の UML XMI DTD の仕様は、UML1.1 に基づいて IBM が独自開発した DTD である）．

5.1.2　実装した整合性検査システム

以下、実装した整合性検査システムについて説明する（実装したシステムの GUI は、OCL compiler に改良を加えたものである）．

- Constraint タブ
  図 5.1 に Constraint タブを示す．Constraint タブでは、検査をおこないたい OCL の不変表明をテキストエリアに記述する．右上の Parse ボタンを押すと、記述した OCL の不変表明がパーズされる．Parse ボタン以外のボタンは、後述する例題の OCL 不変表明をテキストエリアに読み込ませるためのボタンである．

図 5.1: Constraint

- Model タブ
  図 5.2 に Model タブを示す．Model タブには、ファイルのパスを指定する個所が 4 つある．1 番上の欄には、型チェックをおこなうための UML メタモデルの UML XMI 文書のパスを指定する．2 番目の欄には、検査対象モデルの UML XMI 文書のパスを指定する．3 番目、4 番目の欄は、4.7 モデリングツールに依存する問題で説明した、情報の欠落のある UML XMI 文書を補完したい場合に指定する．3 番目の欄に補完したい UML XMI 文書のパスを、4 番目の欄に補完後の UML XMI 文書のパスを指定する．そして、右下の Complement ボタンを押すと、指定したファイルに対する補完がおこなわれる．

図 5.2: Model タブ

- Lexer タブ
  図 5.3 に Lexer タブを示す．Lexer タブは、構文解析で得られたトークンとその型を表示する．

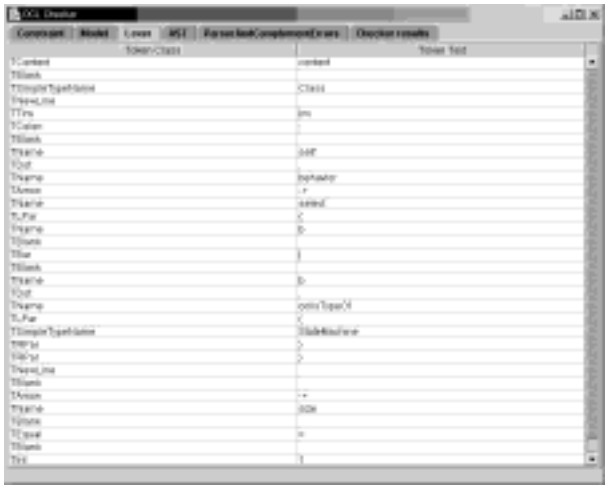

図 5.3: Lexer タブ

- AST タブ
  図 5.4 に AST タブを示す．AST タブは、構文解析の結果得られた抽象構文木を表示する．右上の Show Leaves ボタンを押すと、すべての抽象構文木のノードが表示される．その下の OCL Checker ボタンは、整合性検査をおこなうためのボタンである．ボタンを押すと、検査対象のファイルから DOM ツリーを生成し、検査を行なう．1 番下のラジオボタンは、検査結果を詳細表示にするか、しないかを指定するものである．

図 5.4: AST タブ

- ParserAndComplementErrors タブ
  パーズ時、または UML XMI 文書の補完をおこなっている時に発生したエラーを表示する．

- Checker results タブ
  図 5.5 に Checker results タブ示す．Checker results タブには、整合性検査の結果が表示される．

図 5.5: CheckerResults タブ

5.2 システムの評価

実装した整合性検査システムに対しておこなった、例題を用いた評価について説明する.

5.2.1 例題モデルの選定

例題は、関連研究であるダイアグラム間での構文面の整合性検査に関する研究の論文 [1] におけるモデル、ニュースシステムを用いた. 本評価は、ニュースシステムのダイヤグラムの内、クラス図-ステートチャート図間、およびクラス図-コラボレーション図間に対しておこなった. 図 5.6 から図 5.11 に、ニュースシステムのダイアグラムを示す.

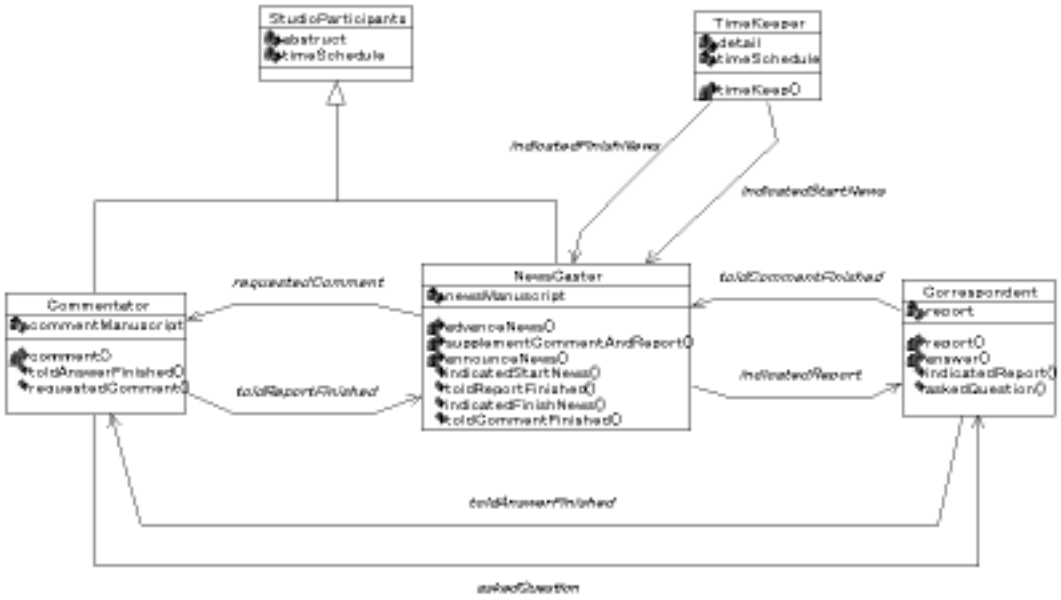

図 5.6: ニュース放送システムのクラス図

5.2.2 検査する OCL 不変表明

上記論文の整合性検査システムでは、予め規定した検査項目に対してのみ整合性検査をおこなうものとなっている. 本評価では、これら検査項目の内の 4 つについて、検査をおこなった. 以下に、評価した 4 つの検査項目を示す.

- クラス図-ステートチャート図間の検査項目
  - ステートチャート図に対するクラスの存在

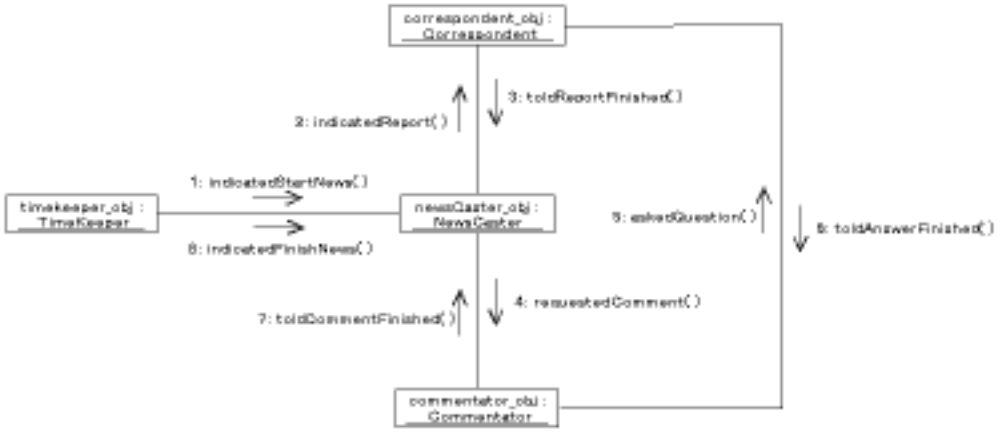

図 5.7: ニュース放送システムのコラボレーション図

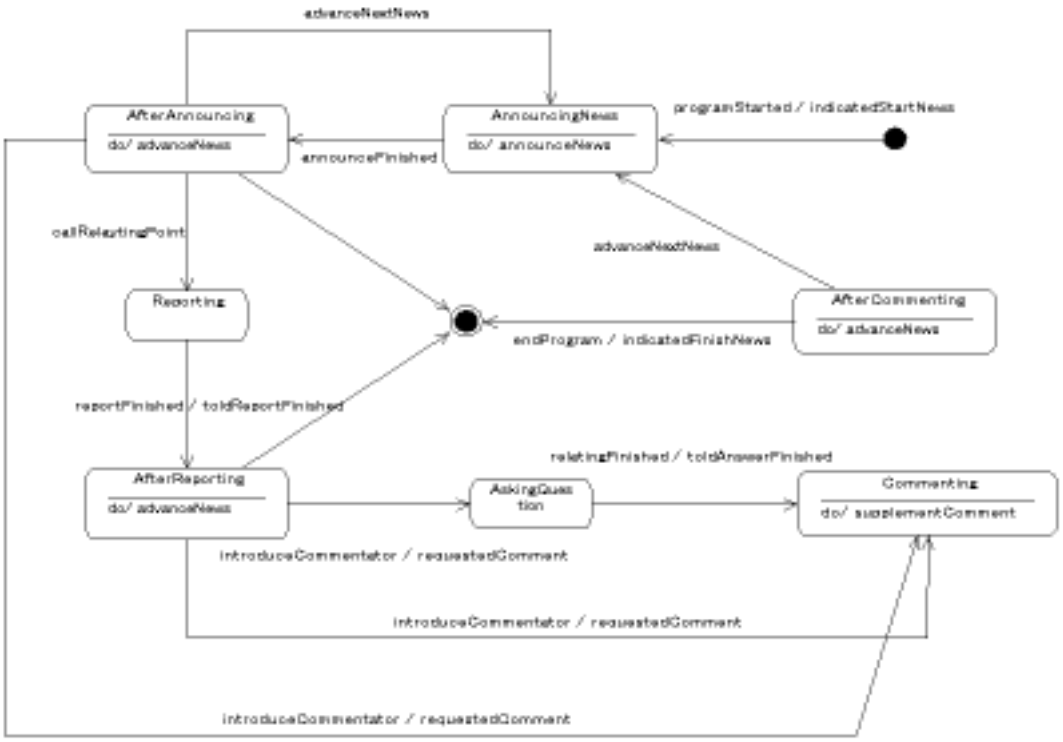

図 5.8: NewsCaster のステートチャート図

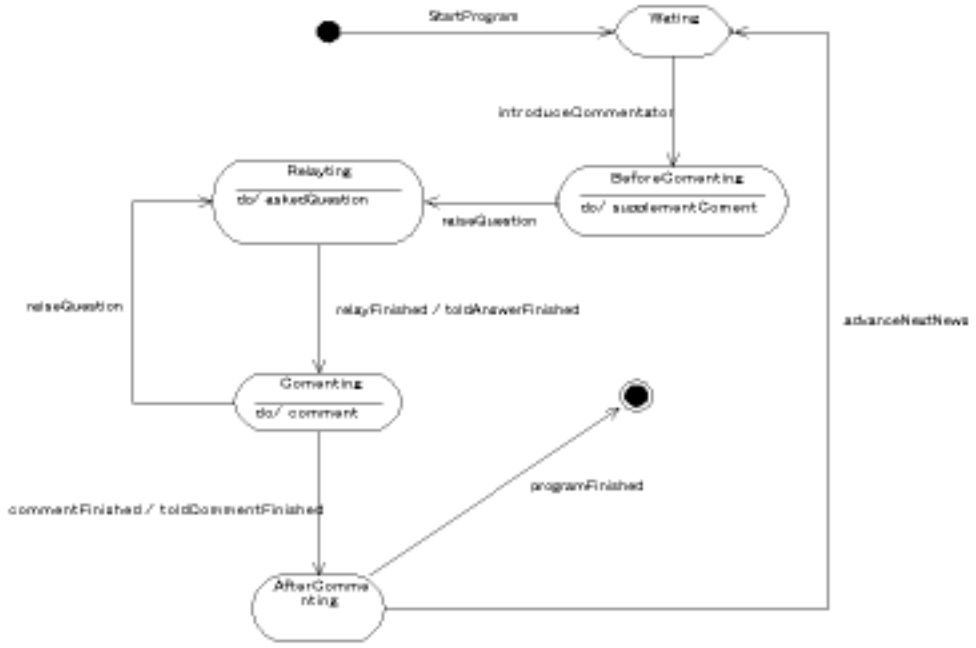

図 5.9: Commentator のステートチャート図

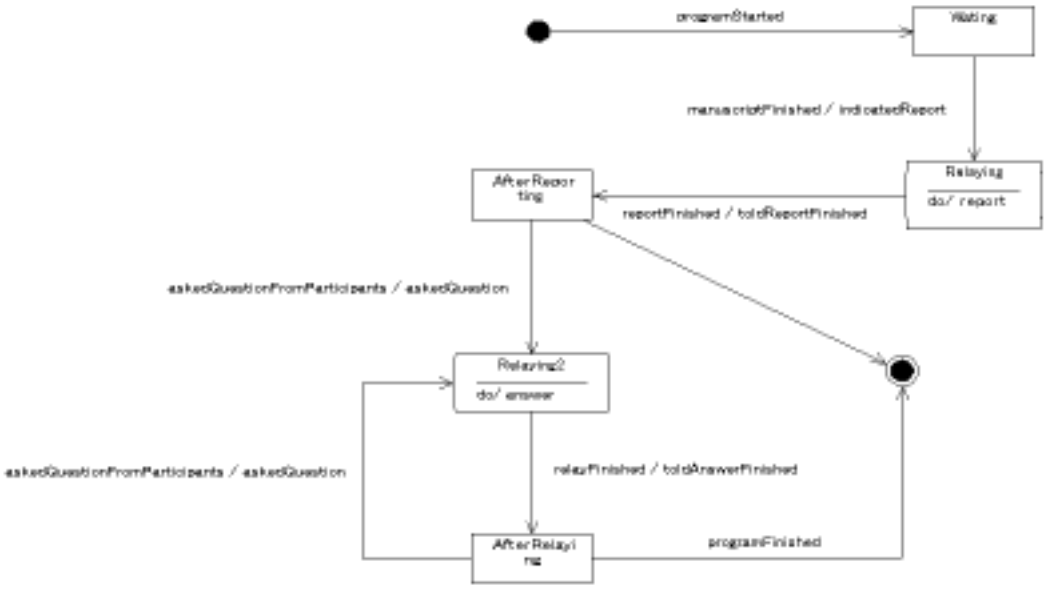

図 5.10: Correspondent のステートチャート図

図 5.11: TimeKeeper のステートチャート図

– ステートチャート図を持っていないクラスの存在

- クラス図-コラボレーション図間の検査項目

– コラボレーション図のオブジェクトとクラス図のクラスとの対応

– メッセージと操作の対応

これらの検査項目を OCL で記述し直した不変表明を以下に示す.

- クラス図-ステートチャート図間の検査項目

– ステートチャート図に対するクラスの存在

不変表明 1：

```
context StateMachine
inv: self.xmi_context.oclIsTypeOf(Class)
```

– ステートチャート図を持っていないクラスの存在

不変表明 2：

```
context Class
inv: self.behavior->select(b | b.oclIsTypeOf(StateMachine))
                  ->size = 1
```

- クラス図-コラボレーション図間の検査項目

  - コラボレーション図のオブジェクトとクラス図のクラスとの対応

不変表明 3：

```
context Collaboration
inv: self.ownedElement
          ->select(oe | oe.oclIsTypeOf(ClassifierRole))
          ->forAll(cr| cr.oclAsType(ClassifierRole).base
                       ->select(b | b.oclIsTypeOf(Class))
                       ->size = 1)
```

  - メッセージと操作の対応

不変表明 4：

```
context Collaboration
inv: self.interaction.message.oclAsType(Message)
         ->forAll(m| m.receiver.oclAsType(ClassifierRole).base
                     ->select(b | b.oclIsTypeOf(Class))
                     ->collect(c | c.oclAsType(Class).feature)
                     ->select(f | f.oclIsTypeOf(Operation))
                     ->exists(o | o.name = m.name))
```

5.2.3 確認結果

上記４つの OCL 不変表明に対し、整合性検査をおこなった結果を以下に示す.

- 不変表明 1： ステートチャート図に対するクラスの存在
  図 5.12 に、不変表明 1 に対する確認結果を示す．整合性検査の結果は true であり、すべてのステートチャート図が対応するクラスを持っていることがわかる.

- 不変表明 2： ステートチャート図を持っていないクラスの存在
  図 5.13 に、不変表明 2 に対する確認結果を示す．整合性検査の結果は true である．しかし、図 5.14 に示す詳細結果をよく見てみると、コンテキストの Class のすべてのインスタンスで self.behavior のナビゲーションをした結果、すべてのコレクションが空になっている（naviFailedObjList は、ナビゲーションに失敗したオブジェクトを格納するリストである．ナビゲーションのパス（図中'NL:[]'）における記号'^' は、ナビゲーションに失敗したパスの位置を表す）．これは、4.7 モデリングツールに依存する問題で述べた UML XMI 文書の情報の欠落によるものである.
  そこで、UML XMI 文書に対する補完をおこない、その後、もう一度検査をおこなった.

図 5.12: 不変表明 1 の確認結果

結果は先と同様に true であるが、図 5.15 の詳細結果を見てみると、状態遷移図をもっていないクラス StudioParticipants （`xmi_id` = 'S.10006'）以外は、ナビゲーションに成功していることがわかる.

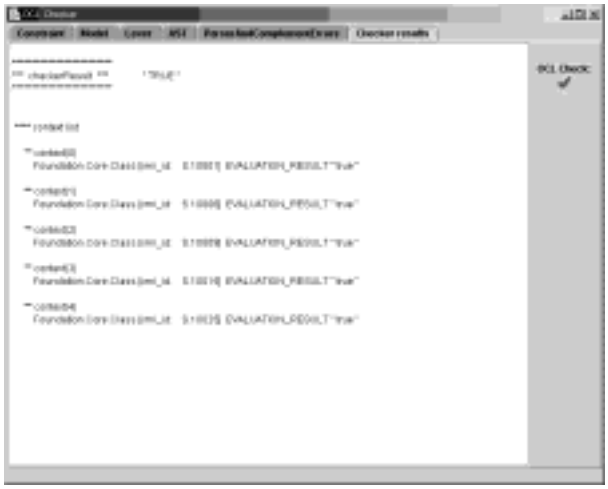

図 5.13: 不変表明 2 の確認結果

- 不変表明 3： コラボレーション図のオブジェクトとクラス図のクラスとの対応
  図 5.16 に、不変表明 3 に対する確認結果を示す. 整合性検査の結果は true であり、すべてのコラボレーション図のオブジェクトが対応するクラスを持っていることがわかる.

- 不変表明 4： メッセージと操作の対応
  図 5.17 に、不変表明 4 に対する確認結果を示す. 整合性検査の結果は true であり、コラボレーション図のすべてのメッセージは、メッセージを受けるオブジェクトのクラスの操作と

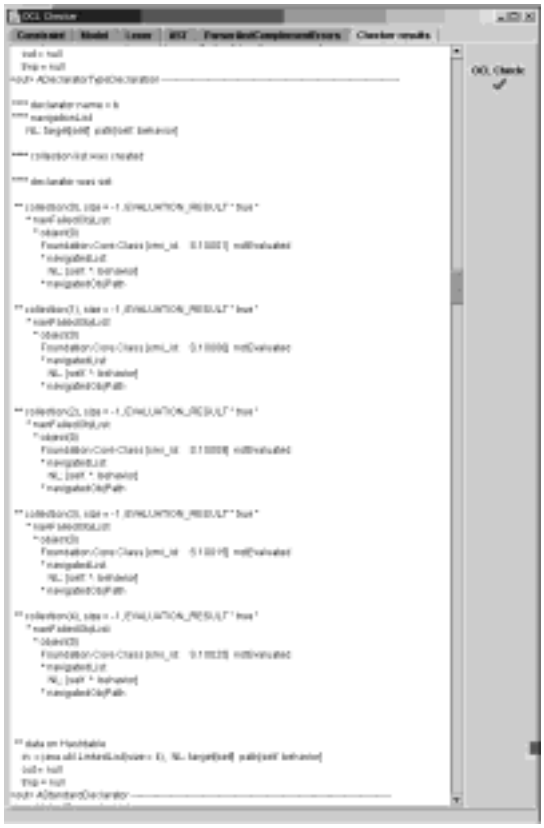

図 5.14: 不変表明 2 の詳細結果（補完前）

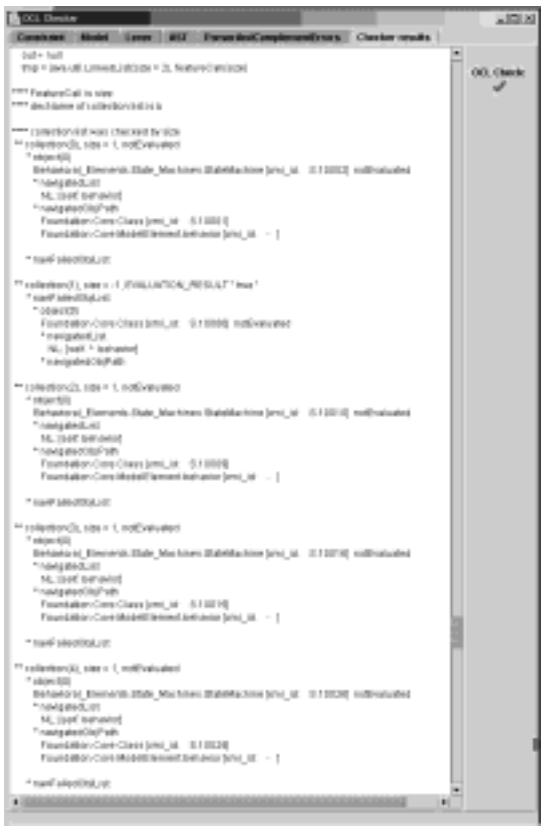

図 5.15: 不変表明 2 の詳細結果（補完後）

図 5.16: 不変表明 3 の確認結果

名前が同じものが存在していることがわかる.

図 5.17: 不変表明 4 の確認結果

5.2.4 結論

本システムを用いて、4 つの OCL 不変表明を作成し、クラス図-ステートチャート図間、およびクラス図-コラボレーション図間の整合性検査をおこなった. 結果、すべての不変表明に対し、正しく整合性検査がおこなえることを確認した.

本評価により、本研究のアプローチで、構文面のダイアグラム間整合性検査システムが実現可能であり、また、その有効性を示すことができた.

第6章　ocle(Object Constraint Language Evaluator))

本整合性検査システムに極めて類似した、ルーマニアのバヘシュ・ボヨイ（Babes-Bolyai）大学で開発された整合性検査システム ocle[3] について述べる．ocle には、以下の特徴がある．

- OCL1.4 のフルセットに対応
- XMI1.0、1.1、1.2 に対応
- UML モデルおよび UML メタモデルに与えた制約に対する検査が可能．
- UML メタモデルおよびモデルのブラウザ、プロパティシートを用いたデバッギング情報の表示が可能．

図 6.1: ocle(Object Constraint Language Evaluator)

ocle では、UML メタモデルの情報はシステムに組み込まれている．そのため、本整合性検査システムのように、UML メタモデルへの何らかの変更を簡単には加えることができない．

第7章　まとめと今後の課題

7.1　まとめ

本研究は、UML メタモデル対する OCL の不変表明をユーザーが独自に定義することが可能であり、実用的・汎用的な構文面のダイアグラム間整合性検査システムの実現を目指しておこなってきた.

上記システムを実現するために、整合性検査の各手法（コレクションの実現方法、UML XMI 文書からのオブジェクトの生成方法など）を考案した.

考案した整合性検査の手法に基づいて、システムの実装をおこない、例題を用いて評価をおこなった．結果、すべての不変表明に対し、正しく整合性検査がおこなえることを確認した.

よって、本研究のアプローチで、構文面のダイアグラム間整合性検査システムが実現可能であり、また、その有効性を示すことができた.

7.2　今後の課題

本研究における課題を以下に示す.

- OCL 文法のフルセットへの対応
  OCL 文法のフルセットに対し、整合性検査可能にする.
- UML メタモデルを用いた OCL 不変表明作成の支援
  一般に、検査すべき OCL 不変表明の記述は長くなる．UML メタモデルを用いて OCL 不変表明作成の支援をおこなう.
- 整合性検査可能な OCL 文法範囲の明確化
  どういう不変表明が検査可能で、どういう不変表明が検査できないのか、その範囲を明確にする.
- 大規模な例題への適用
  実際の大規模な例題（例．ITS(Intelligent Transport Systems) のモデル) を用いて、本システムの有効性を示す.

謝辞

本研究をおこなうにあたり、終始ご指導を賜った片山卓也教授、青木利晃助手に心から感謝いたします．また、研究ゼミ等でお世話になった権藤克彦助教授、立石孝彰氏、岡崎光隆氏に深く感謝いたします．最後に、日頃お世話になったソフトウェア基礎講座の皆様に感謝申し上げます．

関連図書

[1] 大西淳, “ UML モデルにおけるモデル整合性検証支援システム ”, 電子情報通信学会論文誌 D-I,Vol.J84-D-I No.6, pp.671-681, 2001 年 6 月

[2] 立石孝影, 青木利晃, 片山卓也, “ HOL を用いたオブジェクト指向分析モデルの検証 ”, 日本ソフトウェア科学会第 7 回ソフトウェア工学の基礎 (FOSE2000) ワークショップ論文集, pp.117-124, 近代科学者, 2000

[3] http://lci.cs.ubbcluj.ro

[4] OMG, http://www.omg.org/technology/documents/modeling_spec_catalog.htm

[5] OMG, UML 仕様書, アスキー, 2001

[6] Jos Warmer, Anneke Kleppe, The Object Constraint Language, Addison Wesley Longman, 1999

[7] Timothy J. Grose, Gary C. Doney, Stephen A. Brodsky, Ph.D., Mastering XMI, OMG PRESS, 2001

[8] w3c, http://www.w3.org/XML/

[9] http://dresden-ocl.sourceforge.net/

[10] E.Gagnon: SableCC; an object-oriented compiler framework. Master Thesis, McGill University, Montreal, 1998

[11] http://xml.apache.org/xalan-j/

[12] Tigris.org, http://argouml.tigris.org/

付 録A　UMLメタモデル

UML のすべてのメタモデルを図 A.1 から図 A.15 に示す.

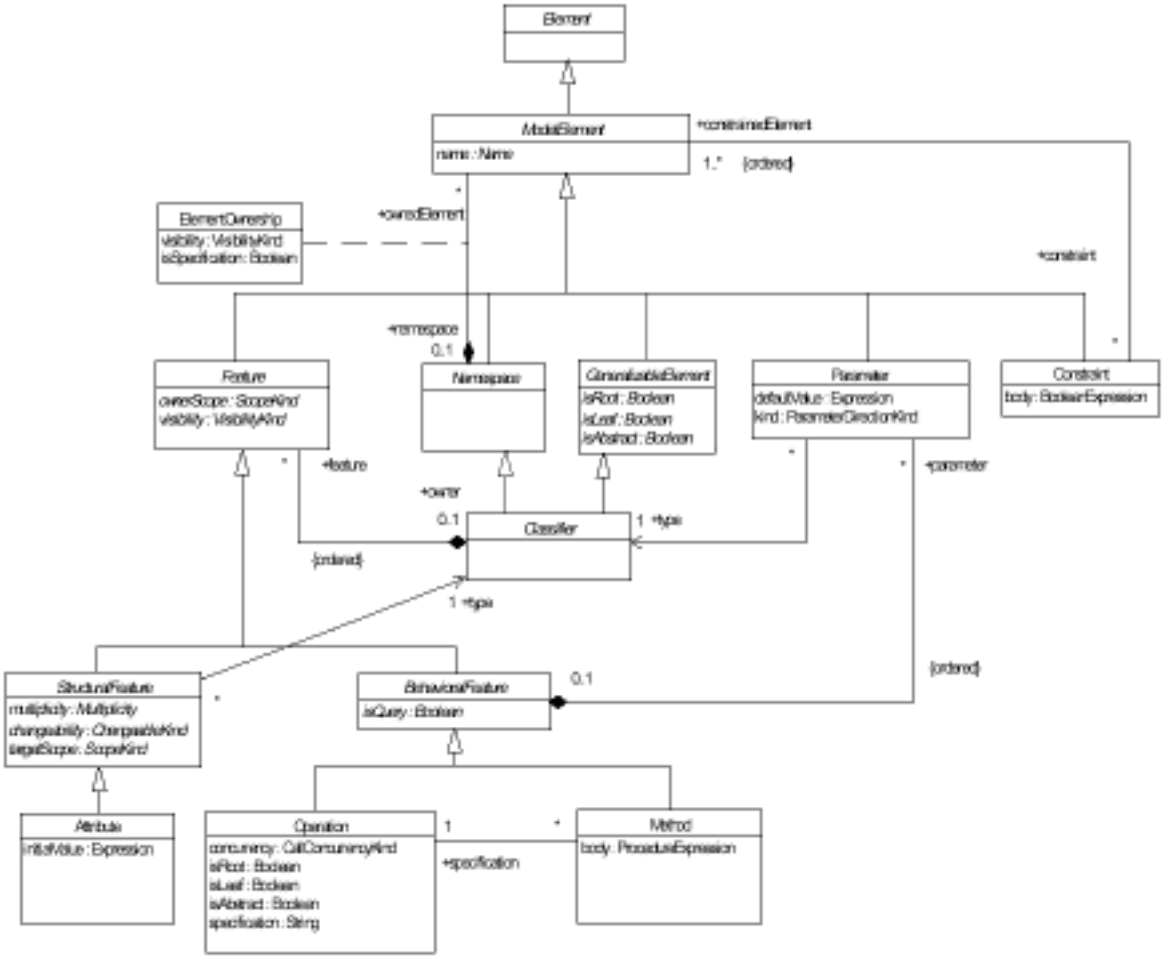

図 A.1: Core Package - Backbone

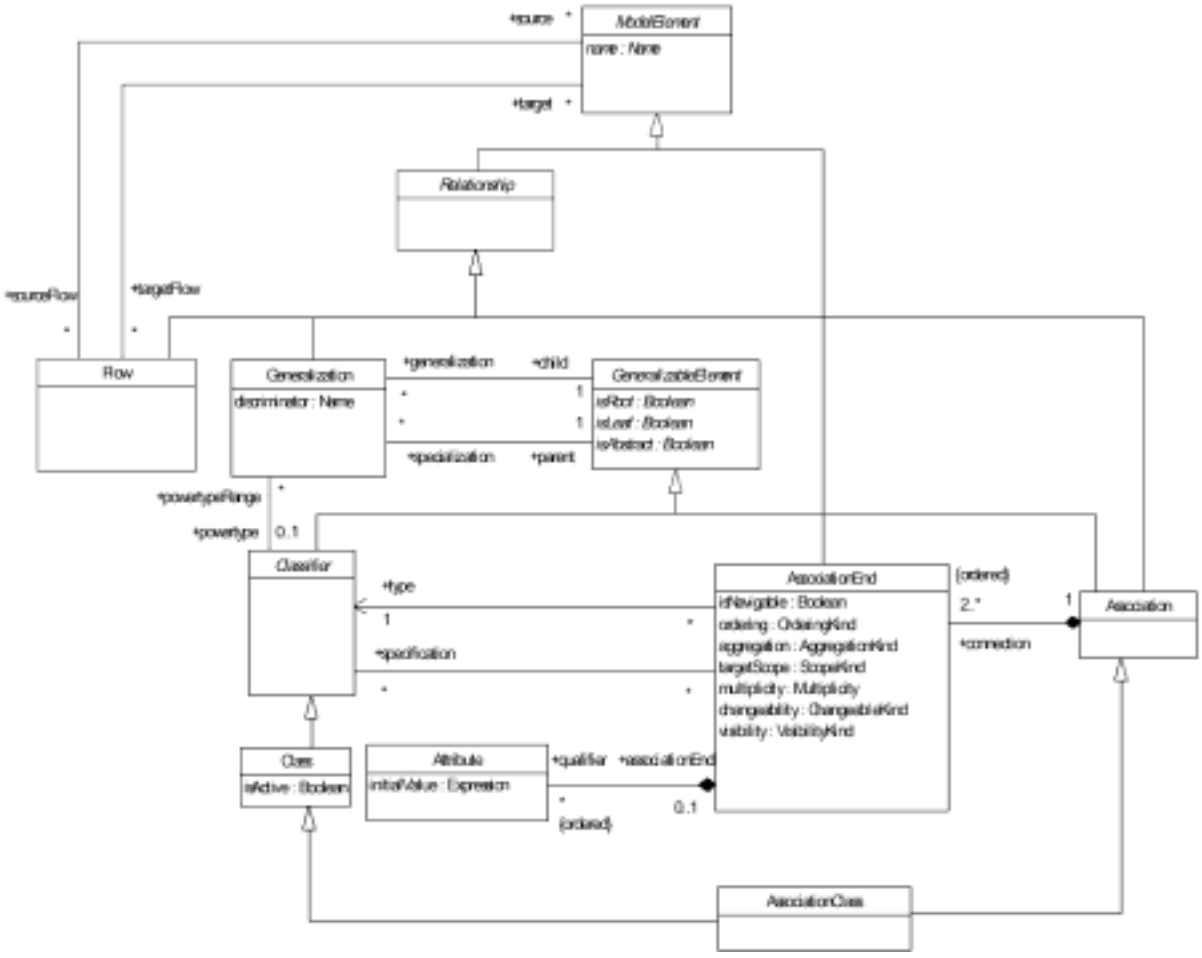

図 A.2: Core Package - Relationships

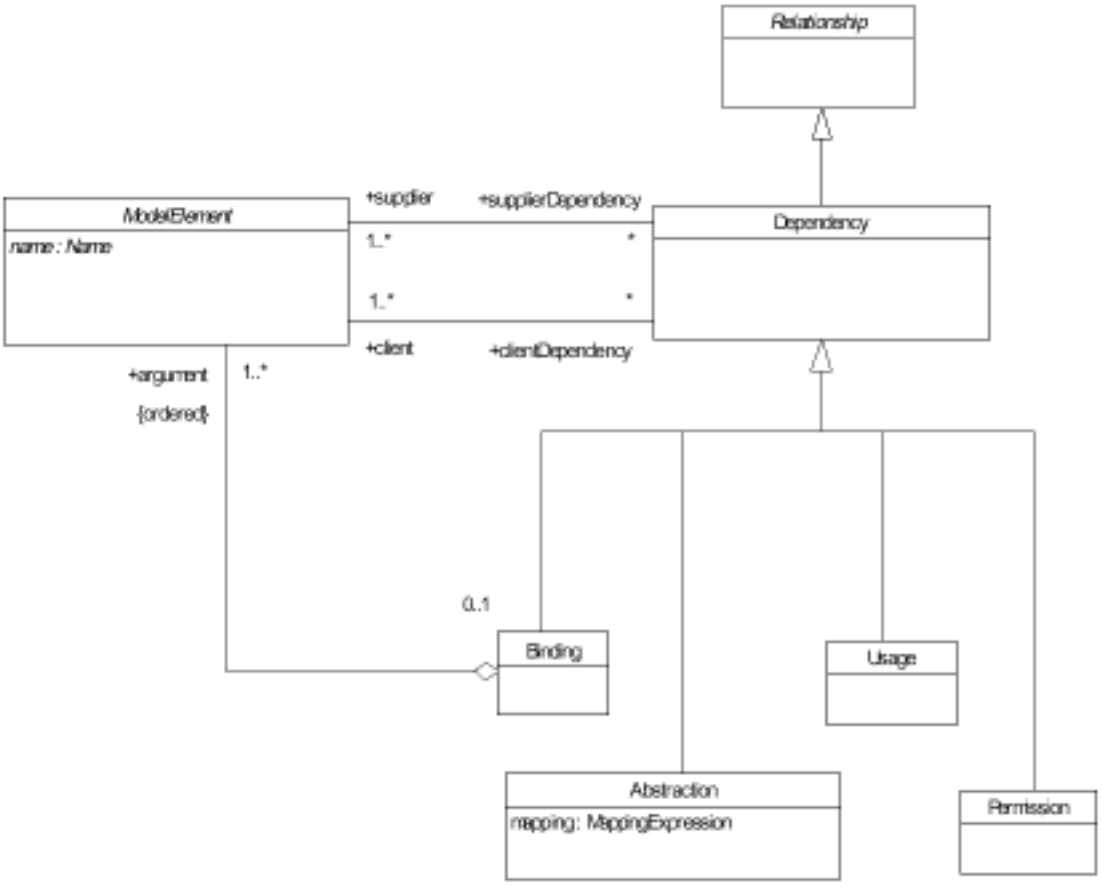

図 A.3: Core Package - Dependencies

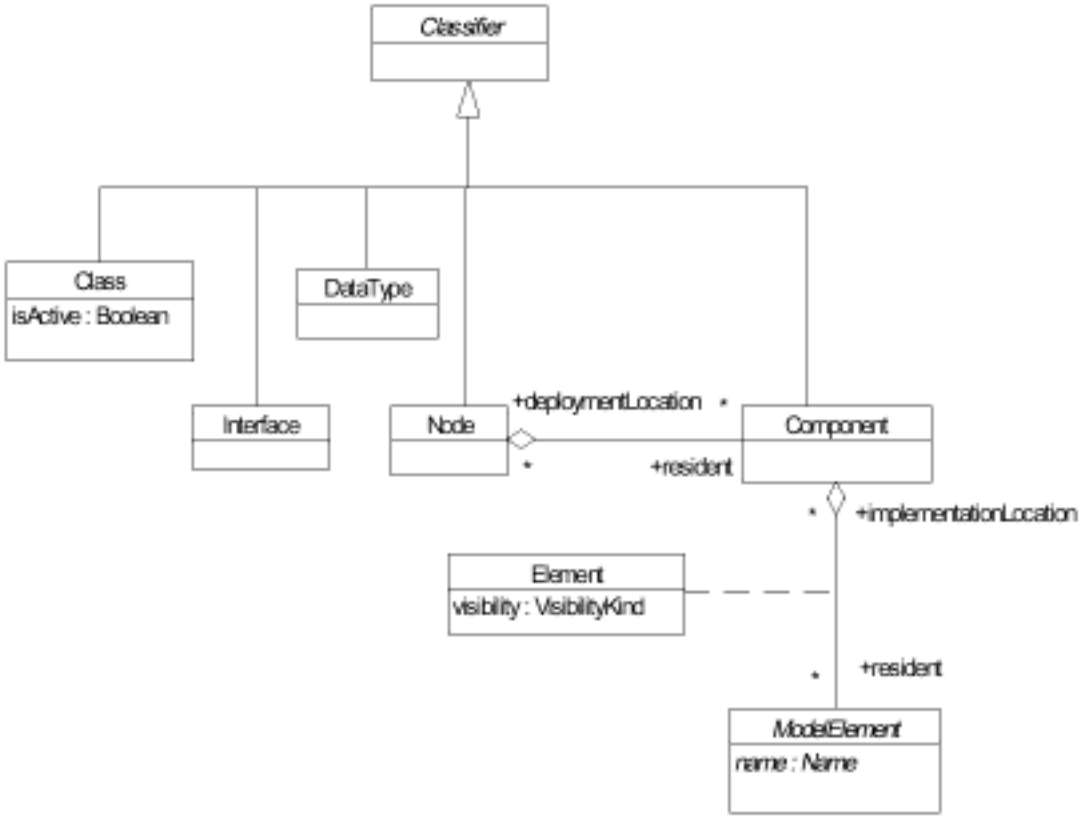

図 A.4: Core Package - Classifiers

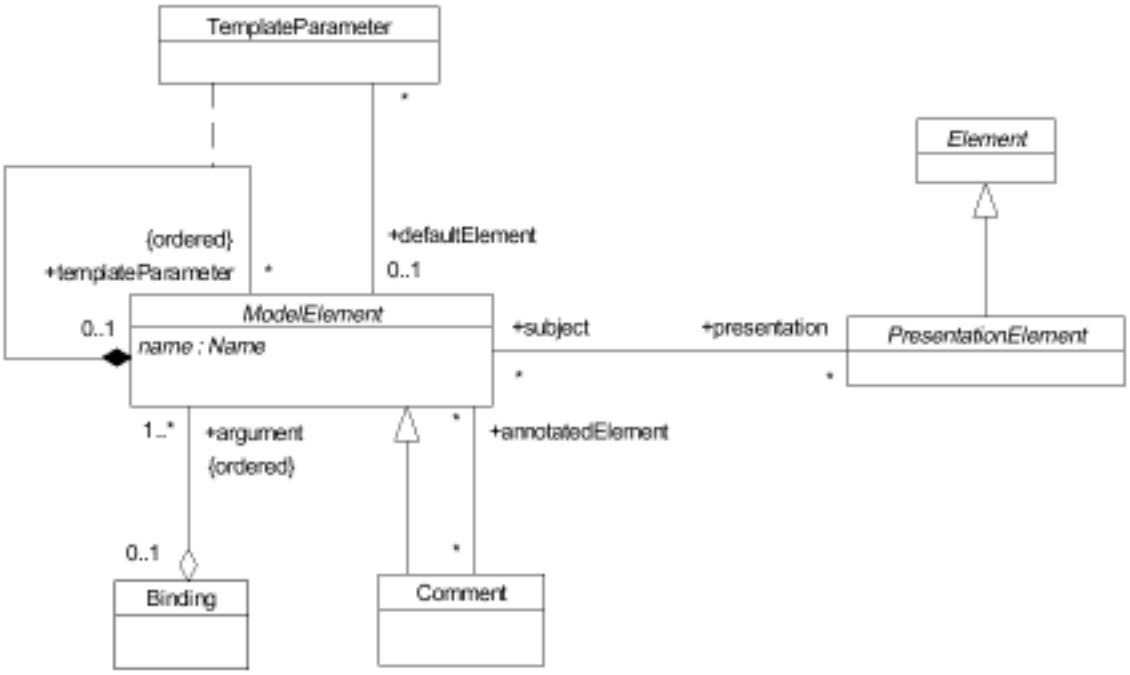

図 A.5: Core Package - Auxiliary elements

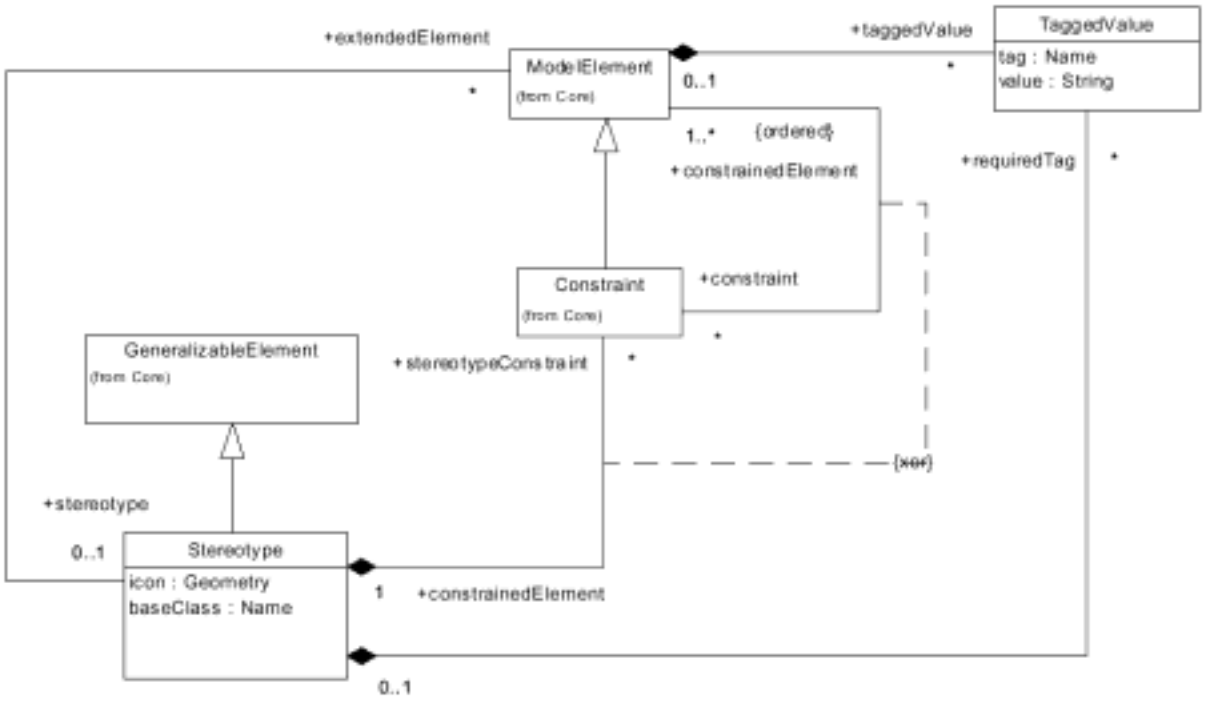

図 A.6: Core Package - Extension Mechanisms

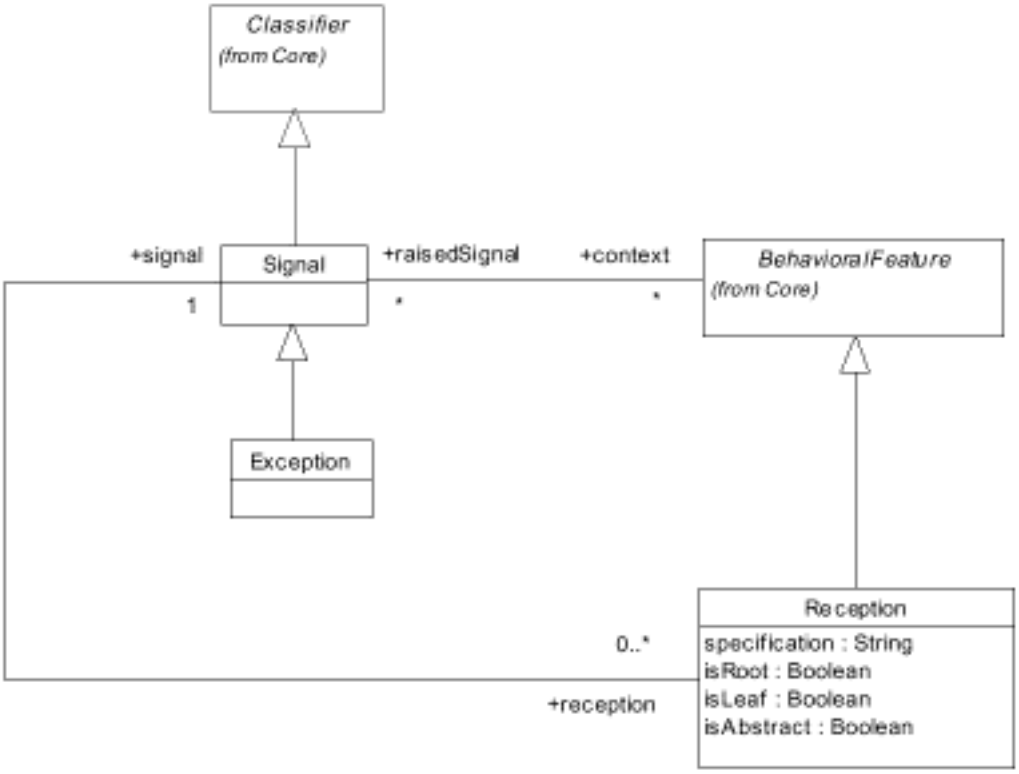

図 A.7: Common Behavior - Signals

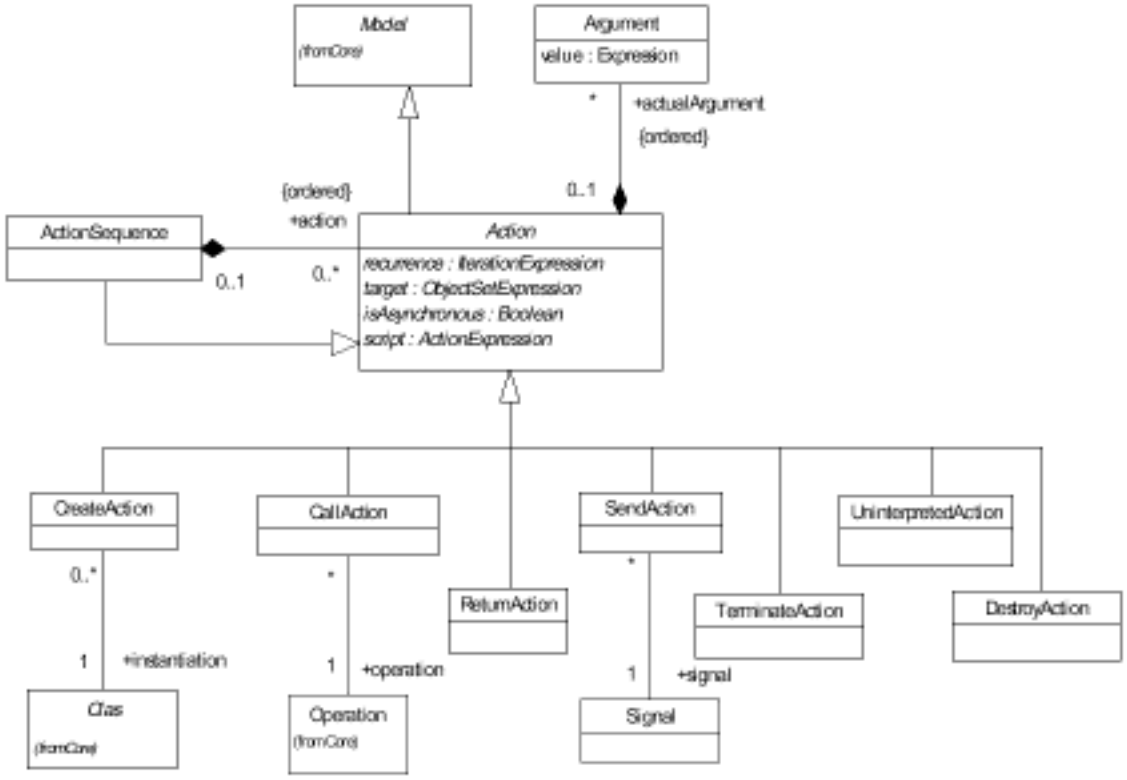

図 A.8: Common Behavior - Actions

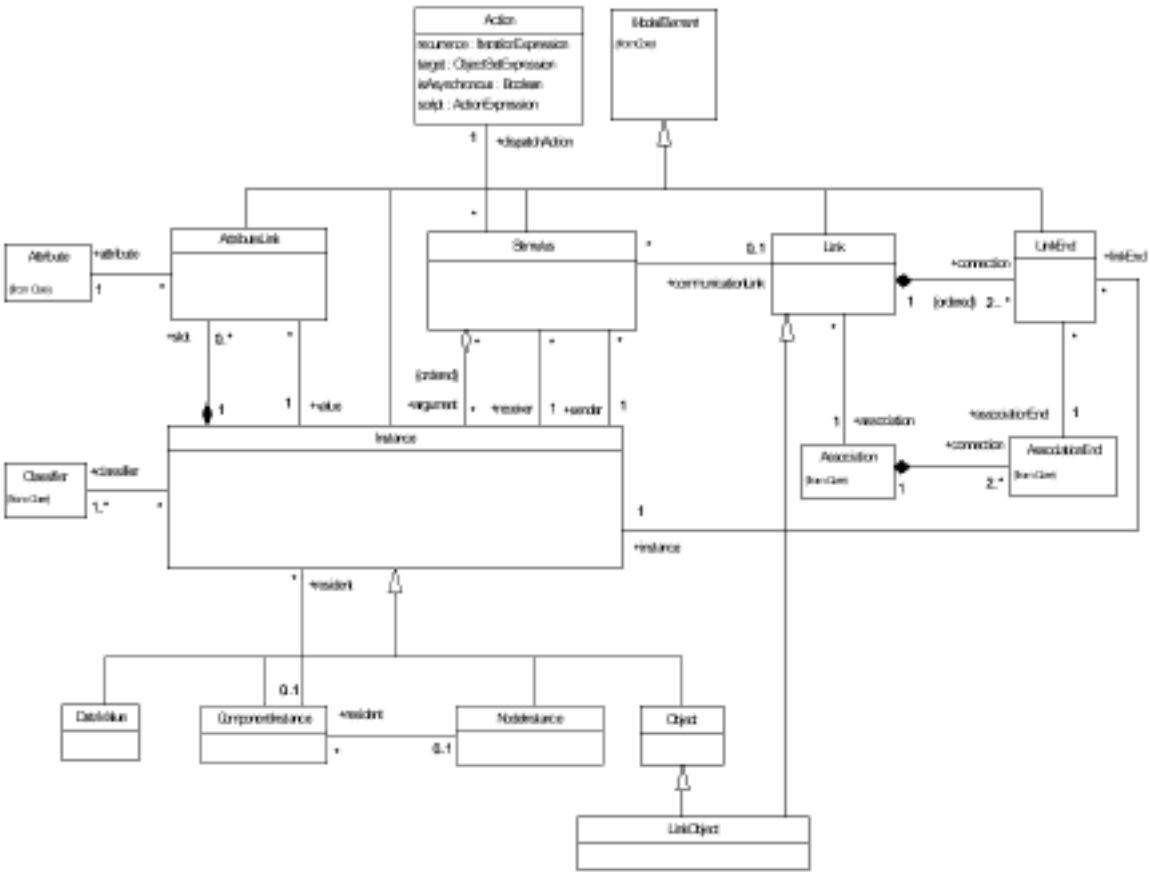

図 A.9: Common Behavior - Instances and Links

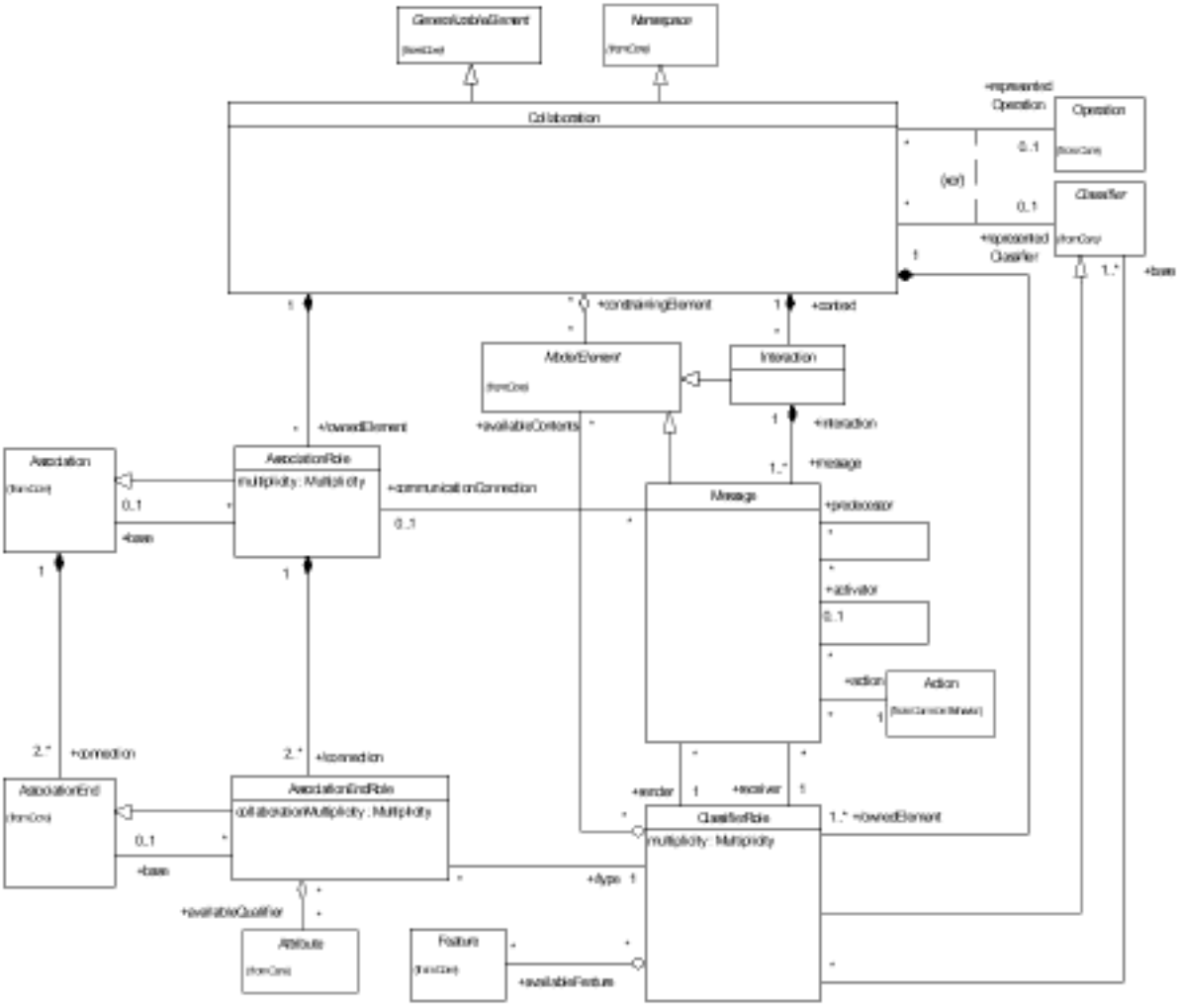

図 A.10: Collaborations

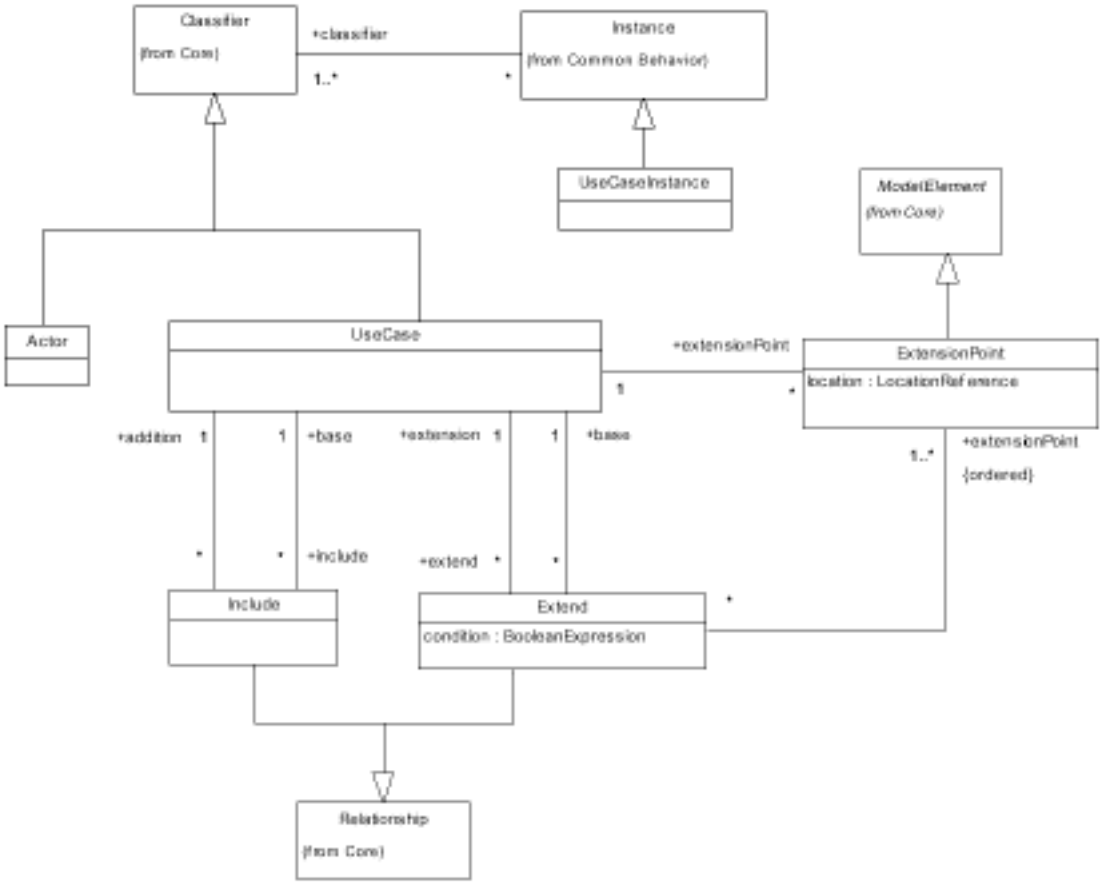

図 A.11: Use Cases

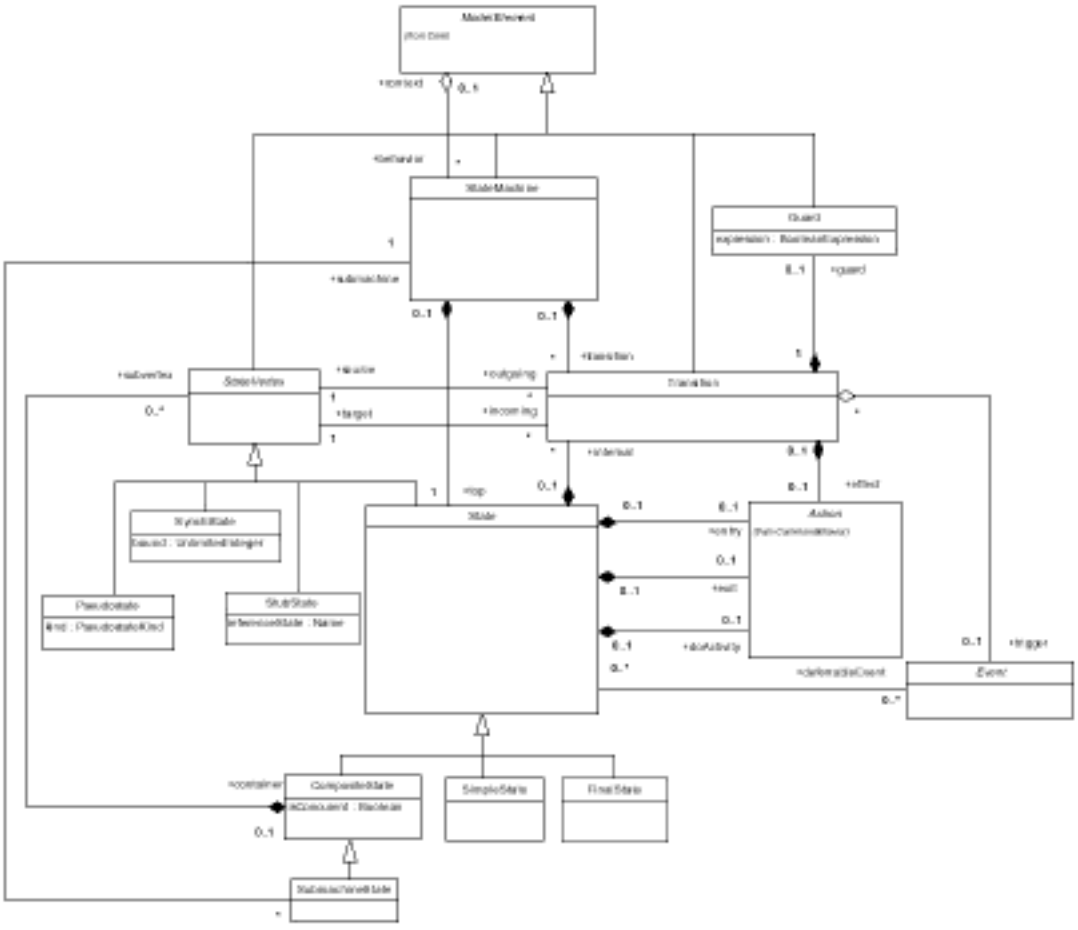

図 A.12: State Machines - Main

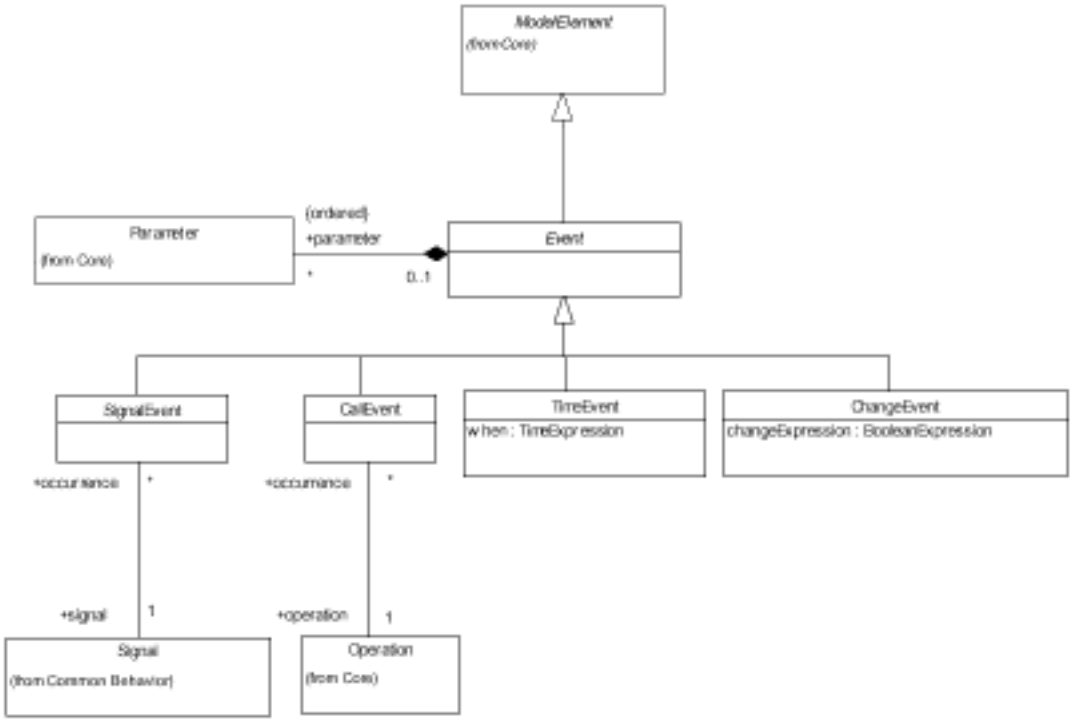

図 A.13: State Machines - Events

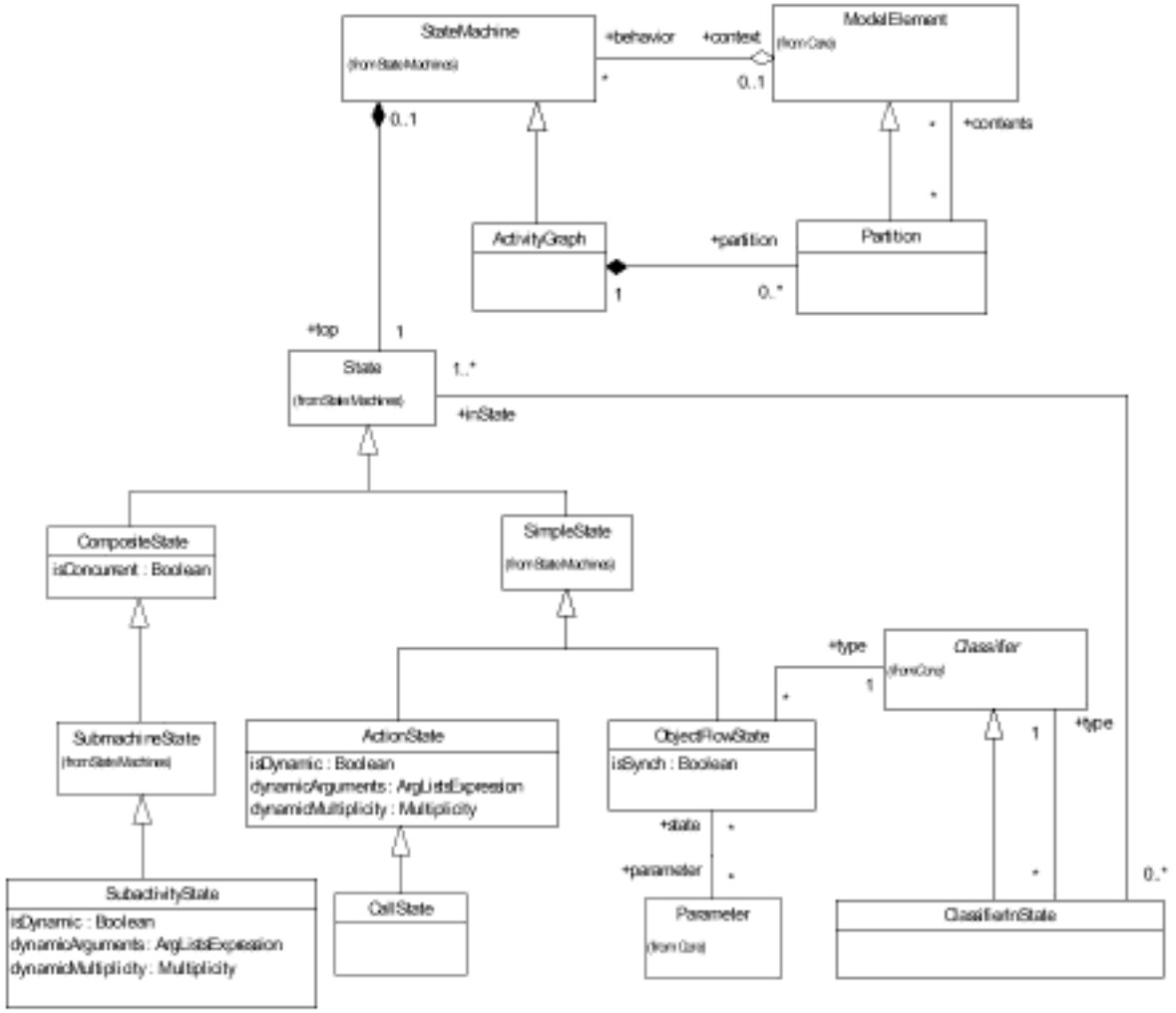

図 A.14: Activity Graphs

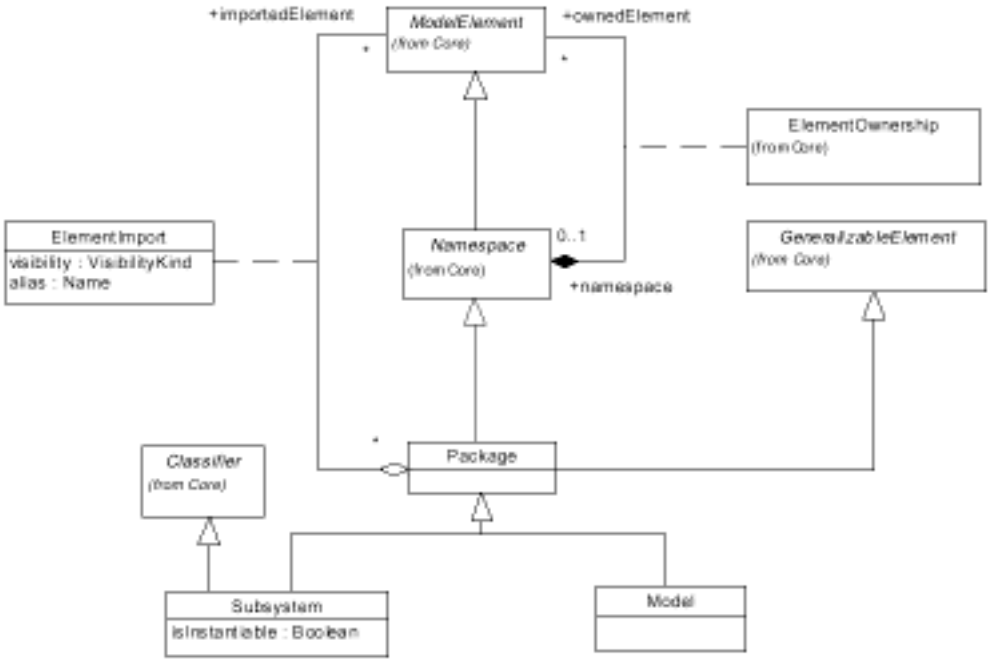

図 A.15: Model Manegement

付 録B　基本的なコレクション操作の一覧

以下に基本的なコレクション操作の構文、および使用例を示す.

- Size 操作
  - 構文
    ```
    collection->size
    ```
  - 使用例
    ```
    context Company inv:
    self.employee->size < 50
    ```

- Select 操作
  - 構文
    ```
    collection->select( v | boolean-expression-with-v )
    collection->select( boolean-expression )
    collection->select( v : Type | boolean-expression-with-v )
    collection->select( v | boolean-expression-with-v )
    collection->select( boolean-expression )
    ```
  - 使用例
    ```
    context Company inv:
    self.employee->select(age > 50)->notEmpty
    self.employee->select(p | p.age > 50)->notEmpty
    self.employee.select(p : Person | p.age > 50)->notEmpty
    ```

- Reject 操作
  - 構文
    ```
    collection->reject( v : Type | boolean-expression-with-v )
    collection->reject( v | boolean-expression-with-v )
    collection->reject( boolean-expression )
    ```

- 使用例

```
context Company inv:
self.employee->reject( isMarried )->isEmpty
self.employee->reject(e | e.isMarried )->isEmpty
self.employee->reject(e : Employee | e.isMarried )->isEmpty
```

- Collect 操作
  - 構文

```
collection->collect( v : Type | expression-with-v )
collection->collect( v | expression-with-v )
collection->collect( expression )
```

  - 使用例

```
context Company inv:
self.employee->collect( birthDate )
self.employee->collect( person | person.birthDate )
self.employee->collect( person : Person | person.birthDate )
```

    Collect 操作は、次のように省略して書くこともできる.
    self.employee.birthdate

- ForAll 操作
  - 構文

```
collection->forAll( v : Type | boolean-expression-with-v )
collection->forAll( v | boolean-expression-with-v )
collection->forAll( boolean-expression )
```

  - 使用例

```
context Company inv:
self.employee->forAll( forename = 'Jack' )
self.employee->forAll( p | p.forename = 'Jack' )
self.employee->forAll( p : Person | p.forename = 'Jack' )
```

- Exists 操作
  - 構文

```
collection->exists( v : Type | boolean-expression-with-v )
collection->exists( v | boolean-expression-with-v )
collection->exists( boolean-expression )
```

  - 使用例

```
context Company inv:
self.employee->exists( forename = 'Jack' )
```

```
self.employee->exists( p | p.forename = 'Jack' )
self.employee->exists( p : Person | p.forename = 'Jack' )
```

- Iterate 操作
  - 構文

    ```
    collection->iterate( elem : Type; acc : Type = <expression>
                        | expression-with-elem-and-acc )
    ```

    collect、select、reject、forAll、exists などの操作はすべて、Iterate 操作を用いて記述することができる.
  - Collect 操作の構文を Iterate 操作で記述した例

    ```
    collection->collect(x : T | x.property)
    --- is identical to:
    collection->iterate(x : T; acc : T2 = Bag{}
                        | acc->including(x.property))
    ```

付 録C　OCLの文法規則

文法の記述は、EBNF 構文を用いている．ここで、“ | ”は選択、“ ? ”はオプション、“ * ”はゼロ回以上、“ + ”は 1 回以上、を意味する．name、typeName、および string の記述では、JavaCC パーザー生成器からの字句トークンに対する構文を示す.

```
constraint              := contextDeclaration
                           (stereotype name? “ : ” expression)+
contextDeclaration      := “ context ”
                           (classifierContext | operationContext)
classifierContext       := (<name> “ : ”)? <typeName>
operationContext        := <typeName> “ :: ” <name>
                           “ (“  formalParameterList? “ ) ”
                           ( “ : ” <typeName> )?
formalParameterList     := formalParameter (“ ; ” formalParameter)*
formalParameter         := <name> “ : ” <typeName>
stereotype              := “ inv ” | “ pre ” | “ post ”
expression              := letExpression* logicalExpression
ifExpression            := "if" expression
                           "then" expression
                           "else" expression
                           "endif"
logicalExpression       := relationalExpression
                           ( logicalOperator relationalExpression )*
relationalExpression    := additiveExpression
                           ( relationalOperator additiveExpression )?
additiveExpression      := multiplicativeExpression
                           ( addOperator multiplicativeExpression )*
multiplicativeExpression:= unaryExpression
                           ( multiplyOperator unaryExpression )*
unaryExpression         := ( unaryOperator postfixExpression )
                           | postfixExpression
postfixExpression       := primaryExpression ( ("." | "->") featureCall )*
primaryExpression       := literalCollection
                           | literal
                           | pathName timeExpression? qualifier?
                             featureCallParameters?
                           | "(" expression ")"
                           | ifExpression
```

```
featureCallParameters   := "(" ( declarator )?
                           ( actualParameterList )? ")"
letExpression           := “ let ” <name>
                           ( “ : ” pathTypeName )?
                           “ = ” expression “ in ”
literal                 := <STRING> | <number> | "#" <name>
enumerationType         := "enum" "{" "#" <name> ( "," "#" <name> )* "}"
simpleTypeSpecifier     := pathTypeName
                           | enumerationType
literalCollection       := collectionKind "{" expressionListOrRange? "}"
expressionListOrRange   := expression
                           ( ( "," expression )+
                             | ( ".." expression )
                           )?
featureCall             := pathName timeExpression? qualifiers?
                           featureCallParameters?
qualifiers              := "[" actualParameterList "]"
declarator              := <name> ( "," <name> )*
                           ( ":" simpleTypeSpecifier )? "|"
pathTypeName            := <typeName> ( "::" <typeName> )*
pathName                := ( <typeName> | <name> )
                           ( "::" ( <typeName> | <name> ) )*
timeExpression          := "@" <name>
actualParameterList     := expression ( "," expression )*
logicalOperator         := "and" | "or" | "xor" | "implies"
collectionKind          := "Set" | "Bag" | "Sequence" | "Collection"
relationalOperator      := "=" | ">" | "<" | ">=" | "<=" | "<>"
addOperator             := "+" | "-"
multiplyOperator        := "*" | "/"
unaryOperator           := "-" | "not"
typeName                := ( [“ a ”- ”z ”] | ["A"-"Z"] | “ _ ” )
                           ( ["a"-"z"] | ["0"-"9"] | ["A"-"Z"] | "_")*
name                    := ( [“ a ”- ”z ”] | ["A"-"Z"] | “ _ ” )
                           ( ["a"-"z"] | ["0"-"9"] | ["A"-"Z"] | "_")*
number                  := ["0"-"9"] (["0"-"9"])*
string                  := "'" ( (~["'","\\","\n","\r"])
                           | ("\\"
                           ( ["n","t","b","r","f","\\","'","\""]
                           | ["0"-"7"] ( ["0"-"7"] )?
                           | ["0"-"3"] ["0"-"7"] ["0"-"7"]
                           )))*
                           "'"
```

付 録D　OCL compilerの文法規則

生成規則の右辺の非終端記号、または終端記号の前に付けられた'{''}' 内の名前は、SableCC が生成するクラスための名前である.

```
Helpers

  all = [0..127];
  lf = 10;
  cr = 13;
  uppercase = ['A'..'Z'];
  lowercase = ['a'..'z'];
  digit = ['0'..'9'];
  number = digit+;
  line_terminator = lf | cr | cr lf;
  input_character = [all - [cr + lf]];


  simple_escape_sequence = '\' ''' | '\"' | '\?' | '\\' |
  '\a' | '\b' | '\f' | '\n' | '\r' | '\t' | '\v';
  octal_digit = ['0' .. '7'];
  octal_escape_sequence = '\' octal_digit octal_digit? octal_digit?;
  hexadecimal_digit = [digit + [['a' .. 'f'] + ['A' .. 'F']]];
  hexadecimal_escape_sequence = '\x' hexadecimal_digit+;
  escape_sequence = simple_escape_sequence |
  octal_escape_sequence |
  hexadecimal_escape_sequence;
  s_char = [all - [''' + ['\' + [lf + cr]]]] | escape_sequence;
  s_char_sequence = s_char+;
  h_set = 'Set';
  h_bag = 'Bag';
  h_sequence = 'Sequence';
  h_collection = 'Collection';



Tokens

  comment = '--' [[all - 10] - 13]* [10 + 13]?;
```

```
dot = '.';
arrow = '->';

not = 'not';

mult = '*';
div = '/';
plus = '+';
minus = '-';

context = 'context';
enum = 'enum';
t_pre = 'pre';
t_post = 'post';
t_inv = 'inv';

equal = '=';
n_equal = '<>';
lt = '<';
gt = '>';
lteq = '<=';
gteq = '>=';

and = 'and';
or = 'or';
xor = 'xor';
implies = 'implies';

l_par = '(';
r_par = ')';
l_bracket = '[';
r_bracket = ']';
l_brace = '{';
r_brace = '}';
semicolon = ';';

dcolon = '::';
colon = ':';
comma = ',';
channel = '#';
at = '@';
bar = '|';
ddot = '..';
apostroph = ''';
```

```
  t_let = 'let';
  t_in = 'in';

  t_if = 'if';
  t_then = 'then';
  t_else = 'else';
  endif = 'endif';

  t_set = h_set;
  t_bag = h_bag;
  t_sequence = h_sequence;
  t_collection = h_collection;

  bool = 'true' | 'false';
  simple_type_name =
  ( uppercase (lowercase | digit | uppercase | '_')* ) |
  h_set | h_bag | h_sequence | h_collection;
  name = lowercase (lowercase | digit | uppercase | '_')*;

  new_line = line_terminator;
  int = number;
  real = number '.' number;
  blank = 9 | ' '*;
  tab = 9;

  string_lit = ''' s_char_sequence? ''';



Ignored Tokens

  comment,
  new_line,
  blank,
  tab;



Productions

  constraint =
    context_declaration constraint_body+ ;

  // added (SableCC)
```

```
constraint_body =
  stereotype name? colon expression ;

context_declaration =
  context context_body ;

// added (SableCC)
context_body =
  {classifier} classifier_context |
  {operation} operation_context ;

classifier_context =
  classifier_head? type_name ;

// added (SableCC)
classifier_head =
  name colon ;

operation_context =
  type_name dcolon name l_par formal_parameter_list? r_par
    return_type_declaration? ;

// added (SableCC)
return_type_declaration =
  colon type_name;

formal_parameter_list =
  formal_parameter formal_parameter_list_tail* ;

// added (SableCC)
formal_parameter_list_tail =
  semicolon formal_parameter;

formal_parameter =
  name colon type_name ;

stereotype =
  {inv} t_inv |
  {pre} t_pre |
  {post} t_post ;

expression =
  let_expression* logical_expression ;

if_expression =
  t_if [if_branch]:expression t_then [then_branch]:expression
```

```
    t_else [else_branch]:expression endif ;

logical_expression =
  relational_expression logical_expression_tail* ;

// added (SableCC)
logical_expression_tail =
  logical_operator relational_expression ;

relational_expression =
  additive_expression relational_expression_tail? ;

// added (SableCC)
relational_expression_tail =
  relational_operator additive_expression ;

additive_expression =
  multiplicative_expression additive_expression_tail*

// added (SableCC)
additive_expression_tail =
  add_operator multiplicative_expression ;

multiplicative_expression =
  unary_expression multiplicative_expression_tail* ;

// added (SableCC)
multiplicative_expression_tail =
  multiply_operator unary_expression ;

unary_expression =
  {unary} unary_operator postfix_expression |
  {postfix} postfix_expression ;

postfix_expression =
  primary_expression postfix_expression_tail* ;

// added (SableCC)
postfix_expression_tail =
  postfix_expression_tail_begin feature_call ;

// added (SableCC)
postfix_expression_tail_begin =
  {dot} dot |
  {arrow} arrow ;
```

```
primary_expression =
  {lit_col} literal_collection |
  {literal} literal |
  {feature} path_name time_expression? qualifiers?
              feature_call_parameters? |
  {parentheses} l_par expression r_par |
  {if} if_expression ;

// this rule was changed to make the grammar LALR(1)-parsable;
// while the concrete syntax used by the parser contains only
// the first two alternatives, the abstract syntax tree
// produced by the class OclParser contains only instances of
// classes generated from the third, unnamed alternative
feature_call_parameters =
  {empty} l_par r_par |
  {concrete} l_par expression fcp_helper* r_par |
  ( l_par declarator? actual_parameter_list? r_par ) ;

// this production is used by the concrete syntax only; instances of
// corresponding classes will not be found in abstract syntax trees generated
// by the class OclParser
fcp_helper =
  {comma} comma expression |
  {colon} colon simple_type_specifier |
  {iterate} semicolon name colon simple_type_specifier equal expression |
  {bar} bar expression ;

let_expression =
  t_let name let_expression_type_declaration? equal expression t_in ;

// added (SableCC)
let_expression_type_declaration =
  colon path_type_name ;

// added alternatives real and boolean
literal =
  {string} string_lit |
  {real} real |
  {integer} int |
  {boolean} bool |
  {enum} channel name ;

enumeration_type =
  enum l_brace channel name enumeration_type_tail* r_brace ;

// added (SableCC)
```

```
enumeration_type_tail =
  comma channel name ;

simple_type_specifier =
  {path} path_type_name |
  {enum} enumeration_type ;

literal_collection =
  collection_kind l_brace expression_list_or_range? r_brace ;

expression_list_or_range =
  expression expression_list_or_range_tail? ;

// added (SableCC)
expression_list_or_range_tail =
  {list} expression_list_tail+ |
  {range} ddot expression ;

// added (SableCC)
expression_list_tail =
  comma expression ;

feature_call =
  path_name time_expression? qualifiers? feature_call_parameters? ;

qualifiers =
  l_bracket actual_parameter_list r_bracket ;

// "iterate" alternative added (missing in specification)
declarator =
  {standard} name declarator_tail* declarator_type_declaration? bar |
  {iterate} [iterator]:name [iter_type]:declarator_type_declaration
    semicolon [accumulator]:name
    [acc_type]:declarator_type_declaration equal expression bar;

// added (SableCC)
declarator_tail =
  comma name ;

// added (SableCC)
declarator_type_declaration =
  colon simple_type_specifier ;

path_type_name =
  type_name path_type_name_tail* ;
```

```
// added (SableCC)
path_type_name_tail =
  dcolon type_name ;

// added to solve keyword conflict
type_name =
  {non_collection} simple_type_name |
  {collection} collection_type l_par simple_type_name r_par;

collection_type =
  {set} t_set |
  {bag} t_bag |
  {sequence} t_sequence |
  {collection} t_collection;

path_name =
  path_name_begin path_name_tail* ;

// added (SableCC)
path_name_begin =
  {type_name} type_name |
  {name} name ;

// added (SableCC)
path_name_tail =
  dcolon path_name_end ;

// added (SableCC)
path_name_end =
  {type_name} type_name |
  {name} name ;

// changed from "at name" to "at t_pre" to solve keyword conflict
time_expression =
  at t_pre ;

actual_parameter_list =
  expression actual_parameter_list_tail* ;

// added (SableCC)
actual_parameter_list_tail =
  comma expression ;

logical_operator =
  {and} and |
  {or} or |
```

```
  {xor} xor |
  {implies} implies;

collection_kind =
  {set} t_set |
  {bag} t_bag |
  {sequence} t_sequence |
  {collection} t_collection ;

relational_operator =
  {equal} equal |
  {n_equal} n_equal |
  {gt} gt |
  {lt} lt |
  {gteq} gteq |
  {lteq} lteq ;

add_operator =
  {plus} plus |
  {minus} minus;

multiply_operator =
  {mult} mult |
  {div} div;

unary_operator =
  {minus} minus |
  {not} not;
```