修 士 論 文

整合性を考慮したOCLからSQLへの変換に関する研究

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

吉積 邦浩

2002年3月

修 士 論 文

整合性を考慮したOCLからSQLへの変換に関する研究

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　落水 浩一郎 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

010123 吉積 邦浩

提出年月: 2002 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

システムの複雑化、大規模化に伴い、ソフトウェア開発における方法論としてオブジェクト指向の方法論が、1980年代の終りから1990年代の初めにかけて多く提案された。代表的なものとして、Grandy Booch によるBooch法、James Rumbaugh によるOMT法、Ivar Jacobson によるOOSE法などがある。これらの方法論は、類似した概念を多く含んでいるにも関わらず、異なった表記法が採用されていた。これら、一連のオブジェクト指向開発の方法論を継承し、表記法を統合したものとしてUMLが生まれた。UMLは，オブジェクト指向開発における表記法であり、OMG(Object Management Group)により標準化が行われた。

オブジェクト指向によるソフトウェア開発が盛んに行われてる現在、UMLは分析・設計段階におけるモデリング言語として広く使われている。UMLは理解の精度を高めるためにシステムの異なった側面を記述できる9種類のダイアグラムが提供されている。これら9種類のダイアグラムは、それぞれ独立性が高いものではあるが、完全に独立してはおらず、分析モデル・設計モデルを作成する際に、異なる種類のダイアグラム間、同種の複数のダイアグラム間において不整合が生じる可能性がある。また、複雑で大規模なシステムの開発では、そのモデルも複雑かつ大規模なものとなってしまう。その為、不整合が生じる可能性があるにもかかわらず、整合性が取れているかどうかのチェックを人間の手で行うには困難である。したがって、計算機によってモデルの不整合がチェックできる支援環境が必要であると考える。

また、整合性と言っても、二種類の整合性を考えることができる。第一に、モデルの意味にまで踏み込んだ整合性である。この整合性を計算機でチェックする場合、定理証明系等を用いた深い推論が必要であり、非常に困難な問題である。第二に、モデルの記述面の整合性である。記述面の整合性は計算機で比較的浅い推論を行うことで自動的にチェックすることが可能である。こういった記述面の整合性検出がシステム開発に貢献する部分もかなり存在する。たとえば、UMLのモデル要素の出現関係に対する整合性も記述面でチェックできる整合性の一つである。

そこで、UMLの標準制約記述言語であるOCLによりモデルの満たすべき制約を記述する。この制約は、そのモデルのインスタンスの整合性の定義と見なすことができる。そして、そのモデルを関係データベースで管理しておき、整合条件であるOCLを変換したSQLクエリーを関係データベースに発行することによって整合性チェックを行う。本研究

ではモデルの記述面に着目したこの整合性検出法を提案する。また、この整合性検出法における整合条件である OCL から等価な意味をもつ SQL への変換アルゴリズムを定義し、この変換アルゴリズムにより計算機によって自動変換が行えることを示すためにトランスレータの実装を行った。

1.2 論文の構成

本論文では、整合性検出法における OCL から等価な意味をもつ SQL への変換方法を提案する。本論文の構成は以下のとおりである。

2 章 本研究の提案する整合性検出法の主要な概念である UML、OCL、SQL についてその概要を述べる。

3 章 本研究で扱う整合性と提案する整合性検出法の概要を説明する。

4 章 モデルのテーブル化方法と OCL から SQL への変換アルゴリズムを定義する。また、実装したトランスレータの概要について述べる。

5 章 整合性検出法の適用例について、モデルとテーブルを提示し、実際に整合性の検出が行えることを示す。

第2章　諸技術の概要

第 2 章では、本研究の重要な概念となる UML、OCL、SQL についての概要を述べる。

2.1　UML

システム開発を行う際にシステムのモデルを設計することは、開発チームの知識の共有を可能とする。そのことにより、システムの構築・保守を円滑に行うことができる。システムが複雑になればそれだけ良いモデリング技術の重要性が増す。そういった要求にを背景に UML は開発された。

UML の開発は、1994 年に Rational Software の Grandy Booch と James Rumbaugh が Booch 法と OMT 法の統合を開始したことから始まる。当時、Booch 法と OMT 法はそれぞれが独立して成長しており、両者は世界中で主要なオブジェクト指向手法と認められていたが、統合が開始された。1995 年には Objectory 社の Ivar Jacobson がこの統合に参加し、OOSE 法の併合が行われた。その後、UML は大いなる発展を遂げ、オブジェクト指向でシステム開発を行う際に現在もっとも広く採用され、有効とされているモデリング言語である。

2.1.1　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もしくは、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 は計算機で支援することを意識して設計されている。しかし、特定の開発プロセス、プログラミング言語への依存を避け、さらには、拡張メカニズムと特殊化メカニズムを備えるなど、柔軟に対応できる設計がなされているため、UML の機能の全てを十分に行かすことができるツールを開発することがより困難になっている。

2.1.2 言語アーキテクチャ

UML のアーキテクチャは OMG の MOF(Meta-Object Facility) に基づいて、４つの層から成り立っている。この UML の言語アーキテクチャの層構造を図 2.1 に示す。

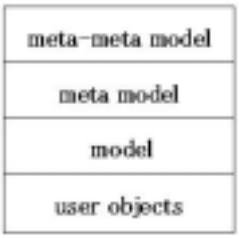

図 2.1: UML の層構造

最下層には、ユーザオブジェクト層 (user object layer) があり、この層は例やドメインに特化したインスタンスを扱う層である。ユーザオブジェクト層の 1 つ上の層はモデル層 (model layer) である。モデル層は、分析、設計段階での成果物として UML モデルを扱う層である。モデル層の上位の次の層は、メタモデル層 (meta model layer) と呼ばれる層である。この層は、UML を定義している層であり、通常、単なる UML のユーザはこの層を意識する必要はないがツール開発者はこの層を意識する必要がある。最上位層は、メタメタモデル層 (metameta model layer) である。この層は、メタモデルを定義している層である。つまり、この層で UML の構成要素を定義している。

OMG の Unified Modeling Language Specification の中で、メタモデルがクラス図で記述されており、メタモデルのトップレベルは、基盤 (Foundation)、動的要素 (Behavioral Elements)、モデル管理 (Model Management) の３つのパッケージで構成されている。これを図 2.2 に示す。

パッケージとは、ダイアグラム中の構成要素をグループ化したものである。トップレベルのそれぞれのパッケージについて説明を以下に示す。

- 基盤パッケージ (Foundaation Package)
  基盤パッケージには以下の要素が含まれている。

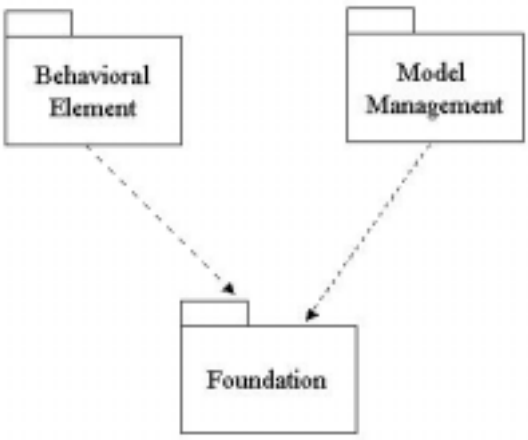

図 2.2: トップレベルのパッケージ

- コア (Core)
  コアパッケージは、UML に必要な抽象メタクラスと具象メタクラスを提供する。インスタンスを作成できるものが具象メタクラス、インスタンスを作成できなものは抽象メタクラスと呼ばれる。たとえば、抽象メタクラスにはモデル要素 (Model Element)、汎化要素 (Generalizable Element)、分類子 (Classifer) などを挙げることができる。また、具象メタクラスには、クラス (Class)、インタフェース (Interface)、関連 (Association)、データ型 (Data Type) などを挙げることができる。
- データ型 (Data Types)
  データ型パッケージでは、UML に integer、string、time などの列挙型を含む基本データ型を提供する。
- 拡張メカニズム (Extension Mechanisms)
  拡張メカニズムパッケージは UML に拡張方法を提供する。

これらの依存関係を図 2.3 に示す。

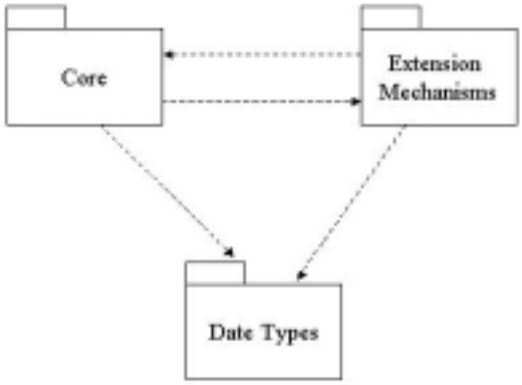

図 2.3: 基盤パッケージ

- 動的要素パッケージ (Behavioral Elements Package)
  動的要素パッケージには以下の要素が含まれる。
  - 共通の振る舞い (Common Behavior)
    共通の振る舞いパッケージは動的要素に要求される核となる概念を提供する。
  - コラボレーション (Collaborations)
    コラボレーションパッケージは特定のタスクを成し遂げるために協調して働くモデル要素の振る舞いを規定している。
  - ユースケース (Use Cases)
    ユースケースパッケージは、アクターとユースケースの振る舞いを規定する。
  - 状態マシン (State Machines)
    状態マシンパッケージは、遷移システムの最終状態へのシーケンスの振る舞いを規定している。
  - アクティビティグラフ (Activity Graphs)
    アクティビティグラフパッケージは、プロセスを表すのに使われる状態マシンの特定のケースを規定している。

  これらの依存関係を図 2.4 に示す。

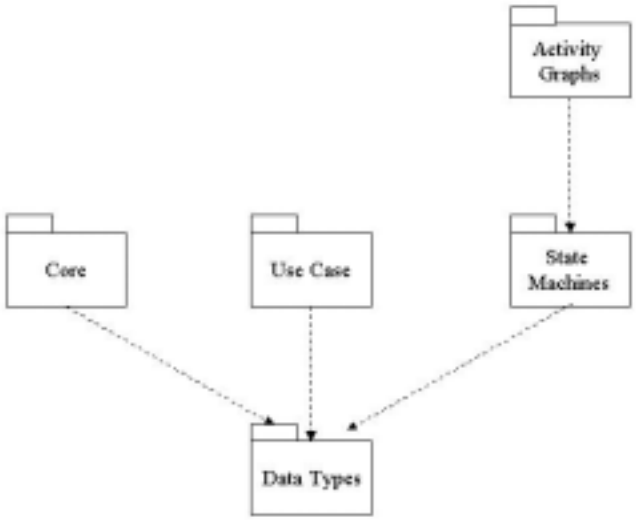

図 2.4: 動的要素パッケージ

- モデル管理パッケージ (Model Management)
  モデル管理パッケージは、どのようにモデル要素がモデルやパッケージ、サブシステムの中で組織されるかを規定するものである。

また、UML の構成要素の静的なセマンティクスは、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不変条件の集合として定義している。構成要素が意味をもつためには、この不変条件を満足し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規定している。これを Well-formedness rules と呼び OCL 式で記述されている。

2.1.3 UML のビューとダイアグラム

UML には、5 種類のビューと 9 種類のダイアグラムが用意されている。ビューとは、システムをモデル化する際に着目するある側面のことであり、それぞれのビューは、単数もしくは、複数のダイアグラムで構成される。以下にそれぞれのビューについて簡略に説明する。

- ユースケースビュー
  ユースケースビューはユースケース図により構成され、問題の所有者と解決策の用件の目標と目的を提供する。また、システムによって外部の相互作用者に提供される機能を記述する。

- 論理ビュー
  論理ビューはクラス図とオブジェクト図により構成され、システムが提供しなくてはならない機能を静的あるいは構造上の問題と解決策の側面を記述する。

- コンポーネントビュー
  コンポーネントビューはコンポーネント図により構成され、解決策実現の構造的で動的な側面を記述する。つまり、ソフトウェア実装コンポーネントの中の構成と依存関係を記述する。

- 動的モデルビュー
  動的モデルビューはシーケンス図とコラボレーション図とステートチャート図とアクティビティ図から構成され、動的な、あるいは振る舞いの問題と解決策の側面を記述する。たとえば、並行処理システムの通信や同期・非同期処理の問題を記述する。

- 配置ビュー
  配置ビューは配置図から構成され、物理的な装置の配置を記述する。

次に、それぞれのダイアグラムは、システムのある側面や特定の部分を表現するために使用され、様々なモデル要素が組み合わさって構成される。システム開発において、通常、複数の種類のダイアグラムを記述する。また、同種のダイアグラムについても、複数記述することができる。以下にそれぞれのダイアグラムについて簡略に説明する。

- クラス図
  クラス図は、クラスの静的な構造を表す。クラスとは、同一のデータ（属性）、手続き（操作）、関係および意味を共有するオブジェクトの集合である。オブジェクトは、それぞれが属するクラスによって生成、破壊される。クラス間には、関連、汎化、集約、依存などの関係が存在する。

- オブジェクト図
  オブジェクト図は、システムのある時点でのオブジェクトのスナップショットとして使用される。表記法はクラス図とほぼ同じであり、関連はインスタンスレベルではリンクと呼ばれる。

- アクティビティ図
  アクティビティとは、何らかの振る舞いを表している状態のことである。その、アクティビティの連続的な流れを表すためのダイアグラムがアクティビティ図である。

- コラボレーション図
  コラボレーション図は、メッセージを送受信するオブジェクトの構造的な構成の協調した相互作用を表す。

- コンポーネント図
  コンポーネントとはシステム内の物理的なコードモジュールのことで、それらの依存関係が示される。

- シーケンス図
  シーケンス図は、メッセージの送受信の時間順序を記述した相互作用図である。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。

- 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とりうるすべての状態と、状態から状態への変化を表すダイアグラムである。状態の変化のことを遷移とよび、その遷移にイベントとアクションを関連づける。イベントとは、オブジェクトの状態を変化させる事象のことである。アクションとは、状態遷移が発生した際に実行される手続きのことである。

- 配置図
  配置図は、システム中のハードウェア、または、ソフトウェアの物理的なアーキテクチャを表すための図である。

- ユースケース図
  ユースケース図は、ユースケースとアクターの相互関係を表す図である。ユースケースとは、システムが提供する機能の記述であり、アクターとは、モデリングする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。

2.2 OCL

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く、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されることが多い。しかし、自然言語による制約の記述は常に曖昧さを含んでしまう。そのため、曖昧さのない制約を記述するために形式言語が盛んに開発されてきた。現在までの多くの形式言語は、数学的な背景をもつ者には理解可能だが、普通のユーザやシステムのモデル化を行う者には理解が困難である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(Object Constraint Language : オブジェクト制約言語) は、UML の標準制約記述言語として採用されており、UML で記述されたモデルのインスタンスとなるシステムで成り立たなければならない不変条件を記述するために用いられる。また、UML モデル作成者は、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 を使用できる。特徴としては、以下の点が挙げられる。

- OCL 式が評価される際に副作用を伴わない。つまり、OCL 式の評価は実行中のシステムの状態を変化させない。
- プログラミング言語ではないのでプログラミングロジックやフロー制御を記述できない。OCL 内では、プロセスを起動したり、問い合わせでない操作を実行できない。OCL はモデル化言語で、OCL のすべてが直接実行できるわけではない。
- OCL は型付きの言語である。OCL 式が適格であるには言語の型適合規則に適合していなければならない。また、UML モデルで定義される分類子は異なる OCL の型を表す。

また、OCL は UML セマンティクスにおいて UML メタモデルを構成するメタクラスの well-fomedness rules を規定する。このことよって以前に比べて UML のメタモデル定義もより曖昧さが排除された整合的なものになった。

OCL の使用個所は次の物が挙げられる。

- クラスモデルにおける、クラスおよび型の不変条件を記述する。
- ステレオタイプのための、型の不変条件を記述する。
- 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。
- 誘導言語として使用する。

- 操作に関する制約を記述する。

また、well-fomedness rules で使用する「付加的な」操作の定義のために使用することもある。

2.2.2 事前定義の型

先に述べたとおり、OCL は型付きの言語であり、 OCL では事前に定義されている型がある。この定義済みの型は、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しており OCL の定義の一部である。また、定義済みの型にはそれぞれ多くの操作が定義されている。これも、モデル作成者が型適合規則に適合した形で使用できる。基本型としては、Boolean 型、Integer 型、Real 型、String 型がある。また、コレクション型としては以下のものがある。

- Collection 型
  Collection 型は、OCL の全てのコレクションの抽象スーパータイプである。つまり、それ自体のインスタンスをもたない。コレクション内のオブジェクトは要素と呼ばれる。

- Set 型
  Set 型は重複した要素を含まず、順不同のオブジェクトのグループである。

- Bag 型
  Bag 型は重複して要素を含むことが可能で、順不同のオブジェクトのグループである。

- Sequence 型
  Sequence 型は重複して要素を含むことが可能で、順序関係のあるオブジェクトのグループである。

その他に、列挙型である Enumeration 型と基本型とコレクション型を補間する型が事前に定義されている。基本型とコレクション型を補間する型として以下のものがある。

- OclType 型
  モデルで定義されたすべての型と OCL の型は、OclType 型のインスタンスとなる。

- OclAny 型
  OclAny 型は、モデル内におけるすべての型のスーパータイプである。

- OclExpression 型
  OclExpression 型はすべての OCL 式の型である。

OCL は、型付きの言語であり、基本的には型階層で構成される。この階層が異なる型が適合するかどうかを決定する。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。クラス図における型適合規則は以下に示すとおりである。

- 各型は、その上位型に適合する。
- 型適合は、推移的である。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。

これは、型はその上位型、およびそれよりも上位の型全てに適合するという効果を生じる。たとえば、Real 型は Integer 型の上位型であり、Integer 型は Real 型に適合する。

2.2.3 ナビゲーションとプロパティの参照

OCL 式は、型、クラス、インターフェース、(型として動作する) 関連、データ型などの分類子を参照できる。これらの型で定義された、全ての属性、関連端、ならびに副作用のないメソッドおよび操作も使用できる。これらをオブジェクトのプロパティと呼ぶ。

クラス図で定義されるオブジェクトのプロパティの値は、ドットとそれに続くプロパティ名とで指定し参照する。

```
context [ AType ] inv:
 self.property
```

self がオブジェクトへの参照であるなら、self.property は、self に関するプロパティである property の値である。

また、OCL のコレクション型以外のそれぞれの型のもつ操作を参照する場合もドットを使って参照することが可能である。コレクション型の要素に対してコレクション型の操作を参照する場合は‘‘->’’ のあとに操作名を記述して参照する。C をコレクション型のインスタンスとし、コレクション型の操作 collection_operation を参照するには以下のように記述する。

```
C -> collection_operation
```

OCL では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ナビゲーションが可能である。このためには、関連終端ロール名を使いその関連でナビゲーションを行う。これは、以下のように記述する。

```
object.rolename
```

この式の値は、rolenameが指す関連の反対側のオブジェクトの集合である。関連端の多重度が最大１である場合、この式の値は１つのオブジェクトになる。もし、関連端にロール名がない場合は関連端における型の名前を小文字で始めることで、ロール名として使用する。また、ナビゲーションはナビゲーションを行った先でさらにナビゲーションを行うことも可能である。ナビゲーションを２度以上続ける場合はさらにドットを記述し、その後にナビゲーション先のロール名を記述することで実現する。ナビゲーションの一般形を以下に示す。

```
object.rolename_1.···.rolename_n
```

2.3 SQL

2.3.1 データベースシステム

データベースシステムとは、企業や学校などの組織体の運用上必要となるようなデータを統合的に管理するシステムである。データベースシステムを利用することで以下の利点を享受できる。

- データの冗長性を除去。
- データの一意性の確保。
- データの共有性の向上。
- データの安全性の向上。
- 組織体内の標準化。

データベースシステムは、実際にデータを格納するデータベース、データベースを管理する DBMS(DataBase Management System)、データの表現方法であるデータモデル、データベースを操作するデータベース言語によって構成される。関係データベースシステムの場合は、データモデルが関係データモデルであり、データベース言語が SQL である。

2.3.2 DBMS(DataBase Management System)

DBMS は、データベースにアクセスするユーザが、快適に利用できるようなサービスを行っているシステムである。DBMS の主な機能は以下のとおりである。

- データベースの管理
  データベースの定義と操作を行う。
- トランザクション管理
  データベースの操作履歴を管理する。
- 同時実行制御
  同時に複数のユーザがデータベースを操作してもデータの整合性が保たれるように管理する。
- 機密保護管理
  データベースに対するアクセスを管理する。たとえば、アクセスが許されていないユーザはデータの参照や変更を行えないように管理する。
- 障害回復管理
  障害発生時に、可能な限りデータの損失を最小限に押さえて復旧を行う機能。

2.3.3 関係データモデル

データのモデリング法としてテーブルの集合と関係を使ってデータを表現する関係データモデルが 1970 年代に IBM の E.F.Codd により提案された。関係データモデルはその単純性、抽象代数に基づいた数学的基盤の明解さを特徴とし、70 年代から 80 年代にかけて盛んに研究された。関係データモデルでは関係の集まりとしてモデル化を行っている。関係とは、定義域 $D_1, D_2, \cdots, D_n$ の直積 $D_1 \times D_2 \times \cdots \times D_n$ の部分集合として定義される。

関係データモデルの操作演算として関係代数が定義されている。関係代数は、関係を引数にとり関係を返す演算である。以下に 5 種の関係代数の基本演算子を示す。これらは、互いに独立である。

- 和演算 (union)
  和は 2 つの関係 $R(r_1, \cdots, r_n)$ および、$S(s_1, \cdots, s_n)$ の和集合をとる二項演算子である。また、以下のように定義できる。ただし、R と S は同じ次数と定義域をもつ。

$$R \cup S = \{t | t \in R \vee t \in S\}$$

- 差演算 (difference)
  差は 2 つの関係 $R(r_1, \cdots, r_n)$ および、$S(s_1, \cdots, s_n)$ の差集合をとる二項演算子である。また、以下のように定義できる。ただし、R と S は同じ次数と定義域をもつ。

$$R - S = \{t | t \in R \wedge \neg t \in S\}$$

- 直積 (cartesian product)
  直積は 2 つの関係 $R(r_1, \cdots, r_n)$ および、$S(s_1, \cdots, s_n)$ が与えられたとき以下の関係を導出する二項演算子である。

$$R \times S = \{(r_1, \cdots, r_i, s_1, \cdots, s_j) | (r_1, \cdots, r_i) \in R \wedge (s_1, \cdots, s_j) \in S\}$$

- 射影 (projection)
  射影は関係 $R(r_1, \cdots, r_n)$ がもつ属性のうち、指定した属性だけを残しほかを削除する単項演算子である。射影 $\pi r_1, \cdots, r_n R$ は以下の関係を導出する。

$$\pi r_1, \cdots, r_n R = \{t[r_1], \cdots, t[r_n] | t \in R\}$$

  ただし、$t[r_n]$ はタプル t の中の属性 r_n に対応する成分である。

- 選択 (selection)
  選択は、関係 $R(r_1, \cdots, r_n)$ がもつタプルの内条件を満たすものだけを残し、他を削除する単項演算子である。選択条件 F を用いた選択 $\sigma_F R$ は以下の関係を導出する。

$$\sigma_F R = \{t | t \in R \wedge F[r_n \setminus t[r_n]]\}$$

  選択条件 F は R の属性名 $(r_1, \cdots, r_i)$ と比較演算子 ($=, <, >, \leq, \geq$) と論理演算子 $(\neg, \wedge, \vee)$ からなる式である。

これらの基本演算子でほぼすべての一般的な問い合わせが表現可能である。これにより、具体的な計算方法が容易に実現できる。つまり、関係代数は5種の基本演算子からなる関係上の代数であると言える。以下に、基本演算子で表現可能だが良く使用される演算子を定義する。

- 共通部分 (intersection)
  共通部分は2つの関係 $R(r_1, \cdots, r_n)$ および、$S(s_1, \cdots, s_n)$ が与えられたとき以下の関係を導出する二項演算子である。

  $$R \cap S = \{t | t \in R \wedge t \in S\}$$

  共通部分演算子を基本演算子で表現すると以下のとおりになる。

  $$R \cap S = R - (R - S)$$

- 結合 (join) 2つの関係 $R(r_1, \cdots, r_n)$ および、$S(s_1, \cdots, s_n)$ における、結合条件 F を比較演算子 θ を用いて $r_i \theta s_j$ と表現すと、結合 $R \bowtie_F S$ は以下の関係を導出する二項演算子である。

  $$R \bowtie_F S = \sigma_F(R \times S)$$

- 自然結合 (natural join) 自然結合は2つの関係 $R(r_1, \cdots, r_i, s_1, \cdots, s_j)$ および、$T(s_1, \cdots, s_j, t_1, \cdots, t_k)$ に対して以下の関係を導出する二項演算子である。

  $$R \bowtie T = \pi_{r_1, \cdots, r_i, s_1, \cdots, s_j, t_1, \cdots, t_k}(\sigma_{R.s_1, =T.s_1, \cdots, R.s_j = T.s_j}(R \times T))$$

2.3.4 SQL

SQL(Structured Query Language) は関係データベースを操作するプログラミング言語である。1970年代にIBMで開発され、数多くの関係データベース製品で採用されたことで、事実上の標準操作言語となった。しかし、各社で実装されているSQLには少しずつ異なった仕様になっていた。それを受け、ANSIは1986年にISOは1987年に規格の制定を行った。

SQLには、大きく分けてデータ定義言語 (Data Definition Language, DDL) とデータ操作言語 (Data Maniqulation Language, DML) がある。DDLはテーブル名や属性名、データ型などの定義、つまりデータベース自体の定義を行う。また、DMLはテーブルに対してデータの追加、修正、削除、検索などの操作を行う。

SQLを利用する方法を大まかに分類すると、以下のように分けることができる。

- 直接問い合わせ
  著癖いつ問い合わせは、関係データベースに対して直接SQLを実行することを意味する。直接問い合わせは、簡単な問い合わせを行うには有用だが、複雑な問い合わせを行うときには向かない。

- 埋め込み
  埋め込は様々なプログラミング言語中でSQLを使用することを言う。埋め込む言語をホスト言語と呼ぶ。埋め込みSQLは問い合わせに用いる引数を変更しながら問い合わせを行う。

第3章　UMLモデルの整合性検出手法

3.1　整合性検出

オブジェクト指向によるソフトウェア開発における、分析・設計段階でのモデリング言語としてUMLが広く用いられている。大規模で複雑なシステムを分析・設計するときには、それぞれのモデルやダイアグラムは別の人間が独立に作成することが多い。ただ単に、UMLの記法にそって記述されただけでは、その作成されたモデル間やダイアグラム間の整合性をUMLでは保証していない。これらの、起りうる不整合を分析・設計段階で放置して、分析・設計や実装を進めると、深刻なバグや大幅な手戻りが生じる結果となる。こうなれば、時間的コストや経済的なコストが増大してしまう。

以上の理由から、作成されたそれぞれのモデルやダイアグラム間の整合性検出が必要であることは明らかである。しかし、大規模で複雑なシステムのモデルはこれも大規模で複雑なものになってしまう。このモデルに対して人間の手で整合性検出を行なう場合、たいへん重要な作業であるにも関わらず、単調で退屈な作業であり、見落としやすく、やはり、時間的コストや経済的なコストが増大してしまう。そのため、整合性検出を計算機で支援することが必要である。

モデルの整合性といっても、大きく二つに分けることができる。

- セマンティクス
  モデルの意味にまで踏み込んだ整合性。
  定理証明系等を用いた深い推論が必要。

- シンタックス
  モデルの記述面の整合性。
  浅い推論で自動的に検出が可能。

本研究では、モデルの記述面の整合性に注目する。この記述面の整合性が貢献する部分はかなりある。例えば、モデル要素の出現関係などである。

OCLはUMLのクラス図に制約を記述し、その制約が満たされているかの検出は制約を記述したクラス図のインスタンスのレベルで行われる。たとえば、2章で述べたとおりUMLのメタモデルはクラス図で記述されており、OCLでメタモデル上に制約を記述することができる。UMLモデルが、メタモデルのインスタンスであることからこの制約は、UMLモデルの整合性の定義と見なすことができる。すでに、UMLを使用する上での

最低限の制約として、“OMG Unified Modeling Language Specification” において、Well-FormednessRules として、OCL で記述されている。たとえば、Interface というモデル要素には、次のような Well-FormednessRule が定義されている。

1. An Interface can only Operations.

```
self.allFeatures->forAll( f | f.oclIsKindOf(Operation)
                                    or f.oclIskindOf(Reception) )
```

2. An Interface cannnot contain any ModelElements.

```
self.allContents->isEmpty
```

3. All Features defines in an Interface are public.

```
self.allFeatures->forAll( f | f.visibility = #public )
```

これらの Well-FormednessRules は、上でも述べたように UML を使用してモデリングする際に最低限守るべき規則である。モデリングをする際には、他にも多くの暗黙的な制約が存在する。これらの暗黙的な制約は、開発方法論やモデリングを行っている組織などに依存することがあるであろう。本環境では、これらの、暗黙的な制約を OCL で記述し、その整合性を実際の UML モデルで成立するかどうかを調べることで、記述面の整合性を検出できる。本研究では、問題を簡略化するために抽象度を下げて UML モデルのある特定のクラス図に対して OCL で制約を記述する。そして、そのインスタンスである実際のオブジェクト図で整合性検出を行う。この対応を図 3.1 に示す。

図 3.1: メタレベルとインスタンスレベルの対応

3.2 整合性検出のためのアプローチ

本研究で提案する環境では、UML で記述されたモデルを単一の関係データベースで管理する。関係データベースを利用することの利点は以下のことが挙げれる。

- 関係データベースは広く普及しており、導入が比較的容易に行える。

- 既存のシステムを利用することができるので開発のコストが大幅に削減できる。
- 実績のあるシステムを利用できるので、信頼性が高い。

関係データベースシステム内はUMLで記述されたモデルの各クラスごとに管理する。これは、各クラスごとにテーブルを作成しそのインスタンスがそのテーブルの要素となることである。モデルの各クラスごとを関係データベースシステムで管理することで、そのシステムの論理的な構造に基づいてテーブルを作成することができ、矛盾なくモデルのもつ情報をテーブルに納めることができる。論理的で矛盾なくモデルをテーブルかできることは、UMLの標準制約記述言語であるOCLで記述されたモデルに対する制約を特に大幅な意味の変化をなくして構文上の変換のみで関係データベースシステムで管理されているモデルにアクセスできることを意味する。UMLモデルのテーブル化の詳細は第4章で述べる。本環境では、ユーザがOCLで記述したUMLモデルへの制約を等価な意味をもつSQLクエリに変換しそれをモデルを管理している関係データベースに発行することによって整合性判定を行う。つまり、図3.2のようにUMLの領域を関係データベースの領域にマッピングを行う。

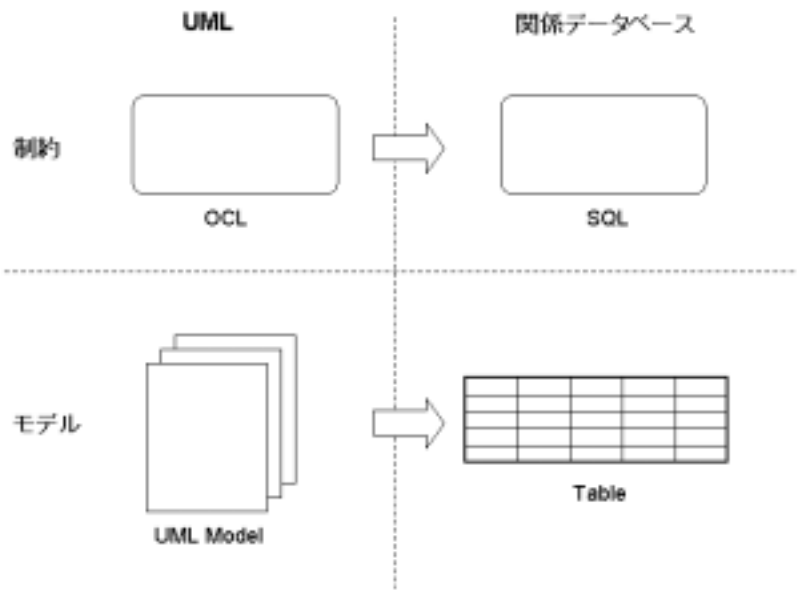

図 3.2: UML領域から関係データベース領域へのマッピング

UMLのモデルと制約を関係データベースの領域にマッピングを行うことによって、上記したような関係データベース技術の利点を享受することができる。以下に上記の整合性検出方法の手順を簡略にまとめたものを示す。また、図3.3に整合性判定システムの概要を示す。

1. UML で記述されたクラス図に対して OCL で制約条件を記述する。
2. クラス図によって関係データベースのテーブルの仕様を与える。
3. OCL で記述された制約条件を SQL クエリーに変換する。
4. 実際のインスタンスを関係データベースで管理する。
5. インスタンスが制約条件を満たしているか変換された SQL クエリーを発行して整合性判定を行う。

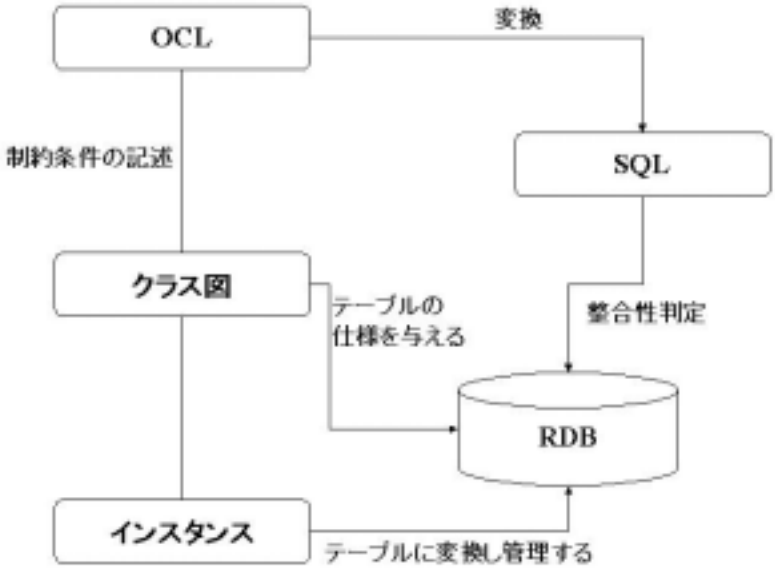

図 3.3: 整合性判定システムの概要

第4章　OCLからSQLへの変換規則

4.1　使用する例

本稿では今後の説明をわかり良くするために、例となるクラス図を導入する。このモデルを図 4.1 に示す。

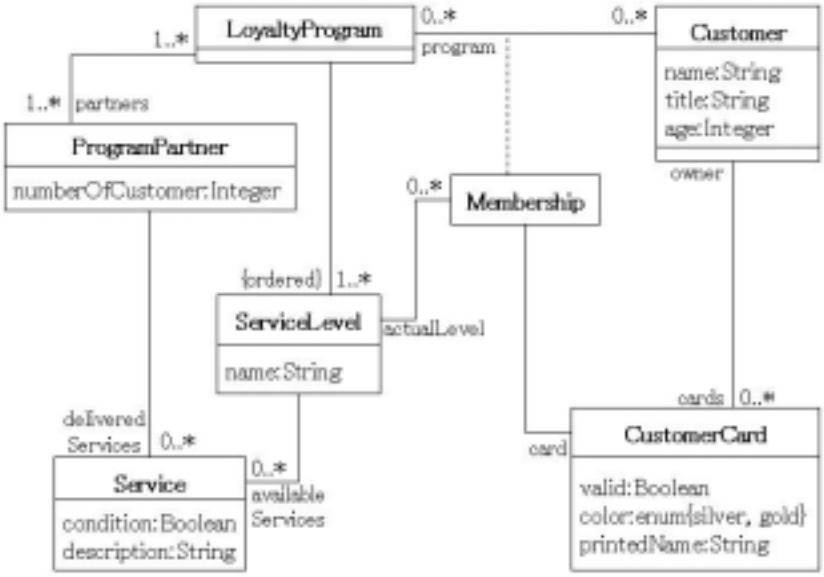

図 4.1: 例に使用するモデル

このモデルは、“THE OBJECT CONSTRAINT LANGUAGE - PRECISE MODELING WITH UML” 内で使用されるモデルの一部を抜粋したものである。このモデルの中心となるクラスは LoyaltyProgram クラスであり、このシステムは必ず１つ以上の LoyaltyProgram オブジェクトを含む。このモデルは顧客にメンバーズカードを発行し LoyaltyProgram に参加している企業からサービスを受けることができるシステムをモデル化したものである。

4.2 モデルのテーブル化

モデルを関係データベースとして、その情報を管理するにあたり、テーブル化の方法を定義する。まず、テーブルはクラス図に現れるそれぞれのクラスごとに作成する。例のクラス図では、LoyaltyProgram、Customer、CustomerCard、ProgramPartner 、 Membership、Service、ServiceLevel の 7 種のテーブルが用意されることになる。それぞれのテーブルにはまずオブジェクトの識別を行うために、それぞれのオブジェクトに対して独特の名前をつける。これを、属性 id としてテーブルに格納する。また、テーブルの属性となる他の値はクラスのもっている属性と操作そして、OCL においてそのクラスからなナビゲート可能なナビゲート先とする。ロール名があればロール名とし、ロール名がなければナビゲート先のクラスの頭文字を小文字に変換したものとする。例のモデルの Customer クラスのテーブルを図 4.2 に示す。

id	name	title	age	program	cards	membership

図 4.2: Customer テーブル

このテーブルの属性 name、title、age はぞれぞれ Customer クラスの属性である。これらの属性には Customer オブジェクトがもつ属性値が格納される。残りの属性 program、cards、membership は Cusotmer クラスからたどれるナビゲート先のクラスを表している。テーブルの属性 program、cards はそれぞれ LoyaltyProgram クラス、CustomerCard クラスのロール名である。また、テーブルの属性名 membership は Membership クラスのクラス名の先頭の一文字を小文字に変換したものである。これらの属性にはリンクの先のオブジェクトの id がそれぞれ格納される。ここで、注意しなければならないのがこのテーブル化方法は、全てのリンクに対して、その先にあるオブジェクトを値としてもつ。そのため、そのテーブルの属性 id は実際のオブジェクトに対しては一意であるが、テーブル内においては一意に定まらない。以上のテーブル化の方針をまとめた対応は以下のとおりである。

テーブル名 → クラス名
属性名 → 特定のオブジェクトを表す id とクラスのもつ属性と
操作およびナビゲート可能なロール名
属性 → 特定のプロパティの値
組 → あるオブジェクトのリンクを含んだ情報

4.3 変換の方針

モデルの整合条件を記述する OCL は評価を行うと、Boolean 型の値が返ってくる。つまり、制約を満たしていれば”True” を、制約を満たしていなければ”False” と言う意味を返す。モデルが整合している状態とは、定められたすべての OCL 式の戻り値の意味が “True” であることである。OCL から SQL へ変換した後も、SQL クエリーのトップレベルの式は、制約を満たしていれば”True” という意味を返し、制約を満たしていなければ”False” の意味を返すものとする。しかし、SQL の仕様では、トップレベルで”True” または、”False” を SQL クエリーの戻り値として返すようにはなっていない。SQL クエリーはすべてテーブルとして結果を返す。そこで、OCL の戻り値である Boolean 型の値を何らかのテーブルに意味を対応づけなければならない。そこで、提案する整合性検出法では、属性値が”True” または、”False” である boolean という属性だけをもつテーブル Result を用意する。この Result テーブルを図 4.3 に示す。

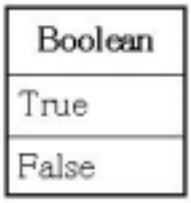

図 4.3: Result テーブル

そして、SQL クエリーを発行した結果としてこの Result テーブルの属性 boolean の属性値の”True” または、”False” を戻り値とすることで、OCL で記述されたモデルの整合条件との対応を取る。これは、以下のような SQL クエリーで実現できる。

```
SELECT *
FROM Result
WHERE Result.boolean =
              NOT EXISTS [ 不整合を起こしているオブジェクトの集合 ]
```

SQL のキーワードである EXISTS 演算子は続くサブクエリーから返された結果の集合が存在すれば”True” を戻り値とし、存在しなければ”False” を戻り値とする演算子である。ちなみに角カッコはサブクエリーを表し、角カッコ内がそのサブクエリーの意味を示す。つまり、この SQL クエリーの意味としては、不整合を起こしているオブジェクトが存在していなければ、Result テーブルから属性値”True” を返し、不整合を起こしているオブジェクトが存在すれば、Result テーブルから属性値”False” を返すということになる。

次に、上記したクエリーの NOT EXSISTS 以下のサブクエリーである、[不整合を起こしているオブジェクト] について説明する。これは、存在してるオブジェクトすべての集合と整合条件を満たしているオブジェクトの集合との差をとったもので実現できる。NOT EXSISTS 以下のサブクエリーを実現した SQL クエリーを以下に示す。

```
SELECT Context.id
FROM Context
WHERE Context.id NOT IN [ 整合条件を満たしているオブジェクトの集合 ]
```

SQL のキーワードである IN 演算子は指定された式がサブクエリー、またはリスト内の値と一致するかどうかを判断する演算子である。構文は以下のとおりである。

$$< exp > \quad [NOT] \quad IN \quad (< subquery > | < exp > [, < exp > \cdots])$$

IN 演算子の前に与えられた式が、演算子の後ろで指定されているサブクエリーの実行結果のリスト、また数値式のリストと一致するものがあった場合には “True” を返し、一致する値がなかった場合には”False” が返される。この SQL クエリーの意味は実際に存在するオブジェクトの集合のなかで整合条件を満たしていないオブジェクトの集合となる。

以上の部分は OCL 式の内容が変化しても Context の値を返るだけで変換が行える。したがって、OCL から SQL クエリーへの変換を考える場合にもっとも重要となるのは、NOT IN 以下のサブクエリーである。つまり、整合条件を満たしているオブジェクトの集合を求める SQL クエリーが変換の際の主要な部分となる。以下にこの主要な部分である NOT IN 以下のサブクエリーの概要を示す。

```
SELECT Context.id
FROM < 整合条件の判定に必要な要素を含むテーブルの作成 >
WHERE < 整合条件の判定を行う >
```

このクエリーはFROM句で整合条件の判定に必要な要素を集め1つのテーブルにまとめる。そして、WHERE句で整合条件を満たしているかどうかの判定を行い、整合条件を満たすオブジェクトのidをテーブルとして返すものである。

以上の、流れでOCLの変換は行われる。次節では、形式的に変換方法を定義する。

4.4 変換アルゴリズム

前節までに、説明をした変換方法の形式化を行う。先ほど示したとおり、NOT INまでの部分については、Contextの値を与えるだけで変換が可能である。この部分を以下に示す。

```
SELECT *
FROM Result
WHERE Result.boolean = NOT EXISTS
        SELECT Context.id
        FROM Context
        WHERE Context.id
                NOT IN [ 整合条件を満たしているオブジェクトの集合 ]
```

先に述べたとおり、変換の際に最も重要なのはNOT IN以下のサブクエリーである。このNOT IN以下のサブクエリーは整合条件の判定を行い、整合条件を満たしているオブジェクトの集合を生成するクエリーである。今後、このNOT IN以下のサブクエリーについて述べる。

まず、このクエリーはオブジェクトを識別するための属性である、idのみを含むテーブルを戻り値とすれば良いのでSELECT句はContext.idでよい。したがって、実質のこのクエリーの重要な部分はFROM句とWHERE句の構造である。

4.4.1 FROM 句

NOT IN 以下のサブクエリーの FROM 句の目的は整合判定に必要な要素を全て含んだテーブルの作成であり、実際の働きは以下のものがある。

- ナビゲーションを行う。
- コレクション型（Collection 型、Set 型、Bag 型、Sequence 型) の戻り値が Boolean 型でない操作の適用を行う。(たとえば、size : int など。）
- OCL 式内のそれぞれの参照で生成されたテーブルを Context.id によって内部結合を行い 1 つのテーブルにまとめる。

OCL のナビゲーションはクラス図の関連に沿って他のオブジェクトやそれらの属性等を参照できるようにする操作である。まず、ナビゲーションは Context から始まり、ドットに続くプロパティにターゲットが移っていく。ナビゲーションの一般形を図 4.4 と以下の OCL 式にて示す。

```
context
self.navigation_1. ... .navigation_n
```

図 4.4: Navigation の一般形

モデルを関係データベースで管理する際に各クラスごとにテーブルを作成し、その属性としてナビゲーションでたどれるロール名をもっている。したがって、そのロール名を内部結合で結合することで、ナビゲーションは実現できる。この意味は、ナビゲーション先のオブジェクトをすべて抽出することと同じである。先ほど示した OCL のナビゲーションの一般形を SQL クエリーの一般形として以下に示す。

```
SELECT Context.id
FROM Context INNER JOIN Navi_1 ON Context.navigation_1 = Navi_1.id
                          ...
             INNER JOIN Navi_n ON Navi_n_1.navigation_n = Navi_n.id
```

次に、Collection 型の戻り値が Boolean 型でないプロパティについて述べる。以下にその変換規則を示す。c は Collection 型のプロパティが適用されるインスタンスとし、c_T はそのインスタンスを含んだテーブルである。

- Collection 型 (戻り値が Boolean 型でないもの)
  - c -> size
    c の要素数を返す。

```
SELECT Context.id
FROM ( SELECT Context.id, COUNT(*) AS size
       FROM ( SELECT DISTINCT Context.id, c
              FROM c_T )
        GROUP BY id )
WHERE size
```

  - c -> count (object)
    c の中での object の数を返す。

```
SELECT Context.id
FROM ( SELECT Context.id, COUNT(*) AS object_count
       FROM ( SELECT DISTINCT Context.id, c
              FROM c_T
              WHERE c = object )
        GROUP BY id )
WHERE object_count
```

  - c -> sum
    c の全ての要素の合計値を返す。この場合、要素は加算可能な数値型でないといけない。

```
SELECT Context.id
FROM ( SELECT Context.id, SUM(c) AS total
       FROM ( SELECT DISTINCT Context.id, c
              FROM c_T )
        GROUP BY id )
WHERE total
```

テーブル化方法の定義の際に述べたように、テーブルは全てのリンクに対して、その先にあるオブジェクトを値としてもつ。そのため、そのテーブルの属性 id は実際のオブジェクトに対しては一意であるが、テーブル内においては一意に定まらない。したがって、この Collection 型の戻り値が Boolean 型でないプロパティの変換において、SQL のキーワード DISTINCT によって同一の要素を含まないテーブルにした上で、それぞれの演算を行っている。これにより、同一のオブジェクトを繰り返して演算の対象にしないように制御している。そして、SQL のキーワード GROUP BY によってそれぞれのオブジェクトに対して演算を行っている。

そして、最終的にナビゲーションと Collection 型の戻り値が Boolean 型でないプロパティの参照で得られたテーブルを一つのテーブルにまとめる。今、これらの結果得られたテーブルを T_1,T_2, ... ,T_n とすると、FROM 句は以下の形となる。

```
T_1 INNER JOIN T_2 ON T_1.id = T_2.id
              ...
    INNER JOIN T_n ON T_n-1.id = T_n.id
```

こうして、整合判定に必要な要素を 1 つのテーブルでもつことによって、テーブルのそれぞれの組が同一のオブジェクトのリンクをたどった要素の集合となる。このことにより、複数のオブジェクトが存在する際にもリンクでつながっているオブジェクト同士の比較が容易に行える。

4.4.2 WHERE 句

NOT IN 以下のサブクエリーの WHERE 句は OCL で記述された整合条件を満たしているかどうかの判定を行う。実際に適応する操作は以下のものがある。

- WHERE 句では、OCL の文法で定義されている、logicalOperator と relationalOperator を適用する。
- 数値型（Real 型、Integer 型) や文字列型 (String 型) の操作を行う。
  (たとえば、数値型の四則演算や絶対値、文字列型の連結や文字数計算。)

この logicalOperator(and, or, xor, implies) と relationalOperator(=, >, <, >=, <=, <>) は OCL で事前に定義されている Boolean 型のプロパティであり、これらの変換規則を以下に示す。b は Boolean 型のプロパティが適用されるインスタンスとし、b_T はそのイン

スタンスを含んだテーブルである。

- logicalOperator
  - b and b2

    ```
    SELECT Context.id
    FROM b_T INNER JOIN b2_T ON Context.id = Context.id
    WHERE b and b2
    ```

  - b or b2

    ```
    SELECT Context.id
    FROM b_T INNER JOIN b2_T ON Context.id = Context.id
    WHERE b or b2
    ```

  - b xor b2

    ```
    SELECT Context.id
    FROM b_T INNER JOIN b2_T ON Context.id = Context.id
    WHERE (b AND NOT b2) or (NOT b AND b2)
    ```

  - b implies b2

    ```
    SELECT Context.id
    FROM b_T INNER JOIN b2_T ON Context.id = Context.id
    WHERE NOT b or b2
    ```

- relationalOperator
  relationalOperator (=, >, <, >=, <=, <>) を relationa_operator で表す。
  - b relational_operator b2

    ```
    SELECT Context.id
    FROM b_T INNER JOIN b2_T ON Context.id = Context.id
    WHERE b relational_operator b2
    ```

logicalOperator の and と or、relationaOperator の全てはSQL にも同一の意味で用意されており、そのまま同じ形で使用可能である。しかし、logicalOperator の xor と implies はSQL で用意されていないので、論理式によって実現している。

次に、Real 型と Integer 型の数値演算を行うプロパティと String 型の文字列操作を行うプロパティの適用規則を以下に示す。n はReal 型と Integer 型のプロパティが適用されるインスタンスである。また、s はString 型のプロパティが適用されるインスタンスである。s_T、b_T はそれぞれそのインスタンスを含んだテーブルである。

- 数値型
  四則演算子 $(+, -, *, /)$ を arithmetic_operator で表す。
  - n arthmetic_operatorand b2

    ```
    SELECT Context.id
    FROM n_T INNER JOIN n2_T ON Context.id = Context.id
    WHERE n arithmetic_operator n2
    ```

  - n.abs

    ```
    SELECT Context.id
    FROM n_T
    WHERE ABS(n)
    ```

- String 型
  - s.size

    ```
    SELECT Context.id
    FROM s_T
    WHERE LEN(s)
    ```

  - s.concat(s2)

    ```
    SELECT Context.id
    FROM s_T INNER JOIN s2_T ON Context.id = Context.id
    WHERE s + s2
    ```

四則演算はSQL でも事前に用意されておりそのまま使用することができる。SQL クエリーで使用されている、ABS、LEN はSQL に組み込まれている関数であり、それぞれ、ABS() は絶対値を返し、LEN は文字列を返す関数である。

4.5 トランスレータの実装

前節までに述べた変換アルゴリズムによる、トランスレータの実装を行った。このことで、計算機で自動的にOCLからSQLへの変換が行えることを示す。、実装言語はC言語を採用した。そして、字句解析の部分にはflexを、構文解析の部分にはbisonを用いた。

4.5.1 bisonとflex

bisonとflexの前身となる、yaccとlexは1970年代にベル研究所で開発された。現在、yaccもlexも標準UNIXユーティリティとなっている。yaccの互換プログラムのbisonと、lexの改良版としてのflexをFree Software FoundationのGNUプロジェクトが配布を行っている。bisonとflexを使用すると、アプリケーションのプロトタイプがすばやく作成でき、修正がたやすくなり、プログラムの保守が単純になるという利点を享受できる。

bisonは形式的に文法規則を定義してやることで、その文法に沿った構文解析器を生成する。つまり、文脈自由文法の仕様を入力として受取り、その文法の正しいインスタンスを認識するC言語の関数を生成する。形式的に文法規則を定義する,最も一般的な方法はBNF(Backus-Naur form)である。BNFで表現された任意の言語は、文脈自由文法である。bisonへの入力は、本質的には、機械可読なBNFである。しかし、全ての文脈自由文法が扱えるわけではなく、bisonが扱えるのは、文脈自由文法のLALR(1)文法だけである。bisonの文法の全体像は以下のとおりになる。

```
%{
C宣言部
%}
bison宣言部
%%
文法規則部
%%
追加のCプログラム部
```

C宣言部には、マクロ定義と、文法定義で使用するための関数と変数の宣言がある。bison宣言部は、終端記号と非終端記号の宣言、演算子の優先順位の指定などを含む。文法規則部1つ以上のbison文法規則を含む。また、追加のCプログラム部は、C宣言部が構文解析器ファイルの先頭に複写されるのと同じように、構文解析器ファイルの末尾にそのまま複写される。bisonの文法規則を定義する場合に、気をつけなければならないのが衝突である。2つの文法規則があったときにどちらを適用して良いか明確でないときに起

るのが衝突である。衝突には、以下の 2 種類がある。

- シフト/還元衝突
  ある 1 つの入力に対して、シフトと還元の両方が有効なときに起る。演算子優先規則宣言で特に指定されない限り、シフトを選ぶことで衝突を解決するように設計されている。

- 還元/還元衝突
  ある 1 つの入力に対して、2 個以上の規則が適用可能であると起る。この衝突は、文法の重大なエラーを意味する。

flex は入力された情報を意味のあるシンボルであるトークンに分割する字句解析器を生成する。flex の定義の全体的な定義の構造は以下のようになる。

```
定義と初期 C プログラム部
%%
ルール部
%%
追加の C プログラム部
```

定義と初期 C プログラム部と追加の C プログラム部に記述された、C プログラムのコードはそのまま、字句解析器ファイルに複写される。定義の部分では、ある文字のグループに一意な識別子を与え、その識別子がその文字グループに置き換えられるようにする。ルール部はパターンとアクションを記述する。パターンは何を認識するべきかを定義し、アクションがそのパターンを認識したときに何を実行するかを定義する。もし、2 つ以上のパターンに合致するものを見つけた場合以下の規則を適用してどのパターンを認識するかを決定する。

- 後続コンテキストも含めて最も長いものを受け入れる。

- 合致するものが全て同じ長さの場合、定義中に最初に記述されたものを認識する。

bison と flex の間で情報を渡す基本的な方法は、関数 yylex() を使用することである。これは、flex により生成される字句解析器において、字句解析処理を実行する関数の名前である。bison はこの関数 yylex() を呼び出して、構文解析を行う言語のある要素を表す整数値を受け取り、解析をおこなう。これにより、bison と flex の間の非常に明確なインタフェースを作成することが可能である。

4.5.2 実装

以下に、定義した変換規則に対する bison の文法規則部を示す。

```
%%
constraint: /* empty */
          | context_declaration expression
;
context_declaration: CONTEXT STRING
;
expression: relational_expression
          | expression LOPE relational_expression
          | expression IMPLIES relational_expression
          | expression XOR relational_expression
;
relational_expression: primary_expression
          | relational_expression ROPE primary_expression
;
primary_expression: NUM
          | SELF DOT reference
          | primary_expression ARROW SIZE
          | primary_expression ARROW COUNT LB literal RB
          | primary_expression ARROW SUM
;
reference: literal
          | literal DOT reference
          | literal property
;
property: DOT CONCAT LB primary_expression RB
          | DOT SIZE
          | AOPE primary_expression
          | DOT ABS
;
literal: STRING
;
```

ここで定義される OCL のサブセットの言語のグループは、まず完全な入力の写しである constraint から始まる。この constraint の意味は、「入力 (constraint) は空文字列もしくは

コンテキスト宣言（context_declaration）と 式（expression）で構成される」とうことである。この規則から、構文解析が始まり、先ほど示した変換規則に適応する文法規則のアクションで SQL クエリーへの変換を行う。その際、C 宣言部で宣言した構造体 query を使用する。構造体 query の宣言を以下に示す。

```
struct query
{
  char *from;
  char *where;
};
```

構造体 query は 2 つの char 型へのポインタだけをもつ単純な構造である。構造体 query の要素 from と where には、SQL クエリーの FROM 句と WHERE 句をそれぞれ対応させている。bison 宣言部において型が ¡qptr¿ と宣言されている非終端記号の文法規則を適用する際に生じる変換部分を構造体 query に格納し、それをその文法規則の意味値として還元する。bison 宣言部を以下に示す。

```
/* Bison 宣言部  */
%union{
  char strval[10000];
  struct query *qptr;
}
%token CONTEXT SELF LB RB
%token CONCAT SIZE COUNT SUM ABS
%token STRING NUM
%left  IMPLIES
%left  LOPE XOR
%left  ROPE
%left  AOPE
%left  DOT ARROW
%type <strval> context_declaration literal
%type <strval> CONTEXT ROPE STRING LOPE IMPLIES XOR NUM
%type <qptr> expression relational_expression primary_expression
%type <qptr> reference property
```

第5章　不正合検出手法の適用例

本研究で提案する不整合検出手法を先ほど示したモデル例 (図 4.1) を使用して説明する。まず、モデルのオブジェクトがもつ属性に関する制約についてである。以下に SQL へ変換する OCL によって記述された整合条件を示す。

OCL 整合条件 - 例題 1

「 Customer クラスの属性 age は 18 以上。」

```
Context Customer
inv: self.age >= 18
```

この整合条件を定義した変換方法によって変換した結果の SQL クエリーは以下のとおりである。

SQL クエリー - 例題 1

```
SELECT *
FROM Result
WHERE Result.boolean = NOT EXISTS
   (SELECT Customer.id
    FROM Customer
    WHERE Customer.id NOT IN
       (SELECT Customer.id
        FROM Customer
        WHERE Customer.age >= 18));
```

キーワード NOT IN までのクエリーと整合条件を満たすオブジェクトを取り出すクエリーの SELECT 句はこの例のコンテキストである、Customer を変換規則の context の部分に代入するだけで良い。適当な Customer クラスのオブジェクトの情報を格納したテーブルを図 5.1 に示す。

id	name	title	age	program	cards	membership
cu1	A	a	25	lp1	cc1	m1
cu1	A	a	25	lp2	cc7	m7
cu2	B	b	38	lp2	cc2	m2
cu3	C	c	22	lp1	cc3	m3
cu4	D	d	15	lp2	cc4	m4
cu5	E	e	63	lp1	cc5	m5
cu6	F	f	35	lp2	cc6	m6

図 5.1: Customer テーブル

例の OCL 制約ではプロパティの参照が 1 度だけ行われている。また、この整合判定に必要な要素はすべて Customer テーブルに含まれている。したがって、FROM 句は Customer テーブルのみを指定している。そして、整合判定を行う WHERE 句で Customer テーブルの属性 age の値が 18 以上の要素を含むテーブルを返すようになっている。図 5.1 の Customer テーブルにこのクエリーを発行すると、整合条件を満たすオブジェクトを取り出すクエリーは図 5.2 のテーブルを返す。Customer テーブルにおいて属性 age が 18 以下である cu4 だけが存在しない Customer テーブルの属性 id がテーブルとして返る。この結果を受けて SQL クエリーのトップレベルでは、Boolean 型の “false” のみを含んだテーブル Result が結果としてでる。したがって、この整合条件は満たされていないことがわかる。変換の流れをこの例題で確認することができる。

id
cu1
cu2
cu3
cu5
cu6
cu1

図 5.2: 結果テーブル

第 2 の例題は複数の参照を含み、また String 型の操作である concat を適用する。この OCL で記述した整合条件を以下に示す。

OCL 整合条件 - 例題 2

「 CustomerCard クラスの属性 printedName は
Customer クラスの属性 title と name を連結したものである。」

```
Context CustomerCard
inv: self.printedName = self.customer.title.concat( self.customer.name )
```

先ほどの例と同様に、キーワード NOT IN までのクエリーと整合条件を満たすオブジェクトを取り出すクエリーの SELECT 句はコンテキストを代入するだけで良い。適当な CustomerCard クラスのオブジェクトの情報を格納したテーブルを図 5.3 に示す。

id	valid	color	printedName	owner	membership
cc1	True	silver	Aa	cu1	m1
cc2	True	silver	Bb	cu2	m2
cc3	True	gold	Cc	cu3	m3
cc4	True	silver	Dd	cu4	m4
cc5	True	silver	Ee	cu5	m5
cc6	True	gold	Ff	cu6	m6
cc7	True	silver	Aa	cu1	m7

図 5.3: CustomerCard テーブル

この例題では、3 つのプロパティを参照する式が含まれている。その式のそれぞれで、ナビゲーションを行い必要な要素を含んだテーブルを作成し、最終的に 1 つのテーブルに内部結合したものが FROM 句となる。この対応関係以下に示す。

```
self.printedName
    CustomerCard

self.customer.title
    CustomerCard AS CC1 INNER JOIN Customer ON CC1.owner = Customer.id

self.customer.name
    CustomerCard AS CC2 INNER JOIN Customer AS C1 ON CC2.owner = C1.id
```

CustomerCard や Customer が 2 度、3 度それぞれ FROM 句内で使用されるので、SQL のキーワードである AS を使用して別名をつけている。そして、先ほど示した規則どおり、String 型のプロパティ concat は SQL では、'+' 演算子で実現できる。これを WHERE 句で適用している。この整合条件を SQL クエリーに変換した結果は以下のとおりである。

SQL クエリー - 例題 2

```
SELECT *
FROM Result
WHERE Result.boolean = NOT EXISTS
   (SELECT CustomerCard.id
    FROM CustomerCard
    WHERE CustomerCard.id NOT IN
       (SELECT CustomerCard.id
        FROM (CustomerCard
              INNER JOIN
              (CustomerCard AS CC1 INNER JOIN Customer
                                   ON CC1.owner = Customer.id)
              ON CustomerCard.id = CC1.id)
              INNER JOIN
              (CustomerCard AS CC2 INNER JOIN Customer AS C1
                                    ON CC2.owner = C1.id)
               ON CustomerCard.id = CC2.id
        WHERE CustomerCard.printedName = (Customer.title + C1.name)));
```

SQL クエリーのトップレベルでは、Boolean 型の “true” のみを含んだテーブル Result が結果として返ってくる。したがって、この整合条件は満たされていることがわかる。この例では、OCL にいくつかの参照を行う式を含んだときの FROM 句の動きと、WHERE 句にのみ影響をおよぼすプロパティの参照の変換法を確認できる。

以下に、第 3 の例題の OCL 整合条件と変換後の SQL クエリーを示す。この例題は、FROM 句に影響をおよぼすプロパティ size を適用している例である。先に示した 2 つの例と同様に、整合条件を満たすオブジェクトの集合を検索するサブクエリの SELECT 句までの変換は、context 部分の代入を行う。

OCL 整合条件 - 例題 3

「 各 LoyaltyProgram クラスのオブジェクトの ServiceLevel は 2 である。」

```
Context LoyaltyProgram
inv: self.serviceLevel->size = 2
```

SQL クエリー - 例題 3

```
SELECT *
FROM Result
WHERE Result.boolean = NOT EXISTS
   (SELECT LoyaltyProgram.id
    FROM LoyaltyProgram
    WHERE LoyaltyProgram.id NOT IN
       (SELECT LoyaltyProgram.id
        FROM (SELECT LoyaltyProgram.id COUNT(*) AS size
              FROM (SELECT DISTINCT
                           LoyaltyProgram.id,LoyaltyProgram.serviceLevel
                    FROM LoyaltyProgram)
                GROUP BY id )
        WHERE size = 2));
```

OCL 式の self.serviceLevel までの部分は LoyaltyProgram テーブルのみで参照可能であるので使用するテーブルはこの 1 つだけである。Collection 型の size の意味は、それぞれのオブジェクトごとのコレクションの要素数である。したがって、`self.serviceLevel -> size` の意味は、LoyaltyProgram のオブジェクトごとつまり、属性 id ごとの serviceLevel の数である。だが、図 5.4 の LoyaltyProgram テーブルからわかるように、このテーブル化手法ではリンクの組み合わせをすべてテーブルに格納するので、他のリンクの関係上 id と serviceLevel だけに着目すると重複要素が存在する。したがって、SQL の DISTINCT キーワードにより、id と serviceLevel だけに着目したテーブルを作成し、その後に、id ごとの serviceLevel の数を数える。以上のことを FROM 句で実現している。WHERE 句では、その要素数を数えた属性である size を比較に用いて整合判定を行っている。図 5.4 の LoyaltyProgram テーブルではこの整合条件を満たしている。

id	partners	customer	membership	serviceLevel
lp1	pp1	cu1	m1	sl1
lp1	pp1	cu3	m3	sl2
lp1	pp1	cu5	m5	sl1
lp1	pp2	cu1	m1	sl1
lp1	pp2	cu3	m3	sl2
lp1	pp2	cu5	m5	sl1
lp2	pp2	cu1	m7	sl3
lp2	pp2	cu2	m2	sl3
lp2	pp2	cu4	m4	sl3
lp2	pp2	cu6	m6	sl4

図 5.4: LoyaltyProgram テーブル

第6章　まとめと今後の課題

6.1　まとめ

本研究では、UML モデルの記述面の整合性に着目した整合性検出手法を提案した。この整合性検出手法のための、モデルのテーブル化方法を示した。テーブル化は、UML モデル上の OCL 制約を記述する際に必要な情報をくまなくテーブル化できる。また、モデルの整合条件である、OCL 制約を SQL クエリーへの変換方法を定義した。OCL のサブセットに対して形式化を行った。そして、この変換方によって計算機による自動変換が可能であることを示すためにトランスレータの実装を行った。このトランスレータは flex と bison によって実現されており、bison の文法規則は 22 個となった。そして、この整合性検出手法による実例を示した。

6.2　今後の課題

現在、手作業で行ったモデルのテーブル化を計算機により自動化する。また、OCL のフルセットに対して変換規則を明確にし、トランスレータの実装を行う。そして、この整合性検出法により、OCL で記述できる整合条件の内で検出が可能な範囲を明らかにする必要がある。

謝辞

本研究を行うにあたり終始ご指導を賜った片山卓也教授、青木利晃助手に深謝致します。また、日頃から有用で適切なご助言をいただきました立石孝彰氏、岡崎光隆氏には心から感謝申し上げます。

最後に、本論文をまとめるにあたりご協力いただいたソフトウェア基礎講座の皆様に深く感謝致します。

関連図書

[1] “OMG Unified Modeling Language Specification Version 1.3”, OMG(Object Management Group), 2000.

[2] Joseph Schmuller 著, 多摩ソフトウェア 訳, 永瀬嘉秀 監訳, “独習 UML”, 翔泳社, 2000.

[3] Sinan si Alhir 著, オブジェクト指向研究会 訳, “UML クイックリファレンス”, オライリージャパン, 1999.

[4] Jos B. Warmer, Anneke G. Kleppe 著, “The Object Constraint Language”, ADDISON-WESLEY, 1999.

[5] C.J.Date, Hugh Darwen 著, QUIP LIC 訳, “標準 SQL ガイド 改訂第 4 版”, アスキー出版社, 1999.

[6] 宮坂 雅輝, “SQL ハンドブック”, ソフトバンクパブリッシング, 2001.