修 士 論 文

振舞仕様によるUMLの意味解析に関する研究

指導教官　二木厚吉 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

宗像一樹

2001年2月15日

要 旨

本研究の目的は、Unified Modeling Language(UML) と形式仕様言語との対応をとり、UML モデルの計算機による自動的な意味解析のための枠組みを提案することである。

UML は、オブジェクト指向モデリングのためのダイアグラムをベースとした表記法であり、事実上標準的なモデリング言語となっている。UML は開発プロセス、ドメインに依存しないため広範な利用が可能であり、ソフトウェアをグラフィカルに抽象化するため、開発者の直観的理解を助けるものとなっている。また、Object Constraint Language(OCL) によりダイアグラムベースの UML に対し整合性を高める機構も用意している。UML は巨大な表記法の集合であり、ダイアグラムも個別にいくつか用意され、現在ではそれらの形式的な意味論が整備されていない。それにより UML モデルの計算機による自動的な分析や、その振舞を確かめるべく仕様の実行等が困難なものとなっている。

本研究で対象とする UML は、クラス図に OCL による制約を加えたものとする。また、形式仕様言語として、代数仕様言語 CafeOBJ を使用する。CafeOBJ はオブジェクト指向の概念を扱うことが可能であることに加え、仕様の検証を機械的に支援する環境があるためである。特に隠蔽代数という枠組みに従えば、オブジェクトモデルを自然に扱うことができる。隠蔽代数で記述される仕様を振舞仕様と呼ぶ。

本研究では、1)UML のクラス図+OCL と振舞仕様の対応をとり、2) その対応にしたがって変換された振舞仕様を CafeOBJ の処理系を用いて解析する方法を提案する。振舞仕様による表現ではクラス、関連および OCL 式との対応をとり、いくつかの記述例を示した。CafeOBJ の処理系を用いた解析では、UML モデルのシミュレーション実行、OCL 制約のチェック方法について述べた。また、OCL 制約の一つである不変条件が、システムのすべての状態について成立することを証明する検証例を示した。

目 次

第 1 章

はじめに

形式手法とオブジェクト指向技術は、ソフトウェアの生産性と品質を向上させる主要な技術として、共に長い歴史を持つ。オブジェクト指向技術は、プログラミング言語、コンポーネント、パターン、フレームワーク等、多岐にわたり高度な再利用技術による生産性の向上とうい点においても、産業界では不可欠な技術となっている。また、方法論や仕様記述のための言語も整理が進み、ますます実用性の高いものとなっている。Unified Modeling Language(UML) は、オブジェクト指向モデリングのためのダイアグラムをベースとした表記法であり、事実上標準的なモデリング言語となっている。UML は開発プロセス、ドメインに依存しないため広範な利用が可能となっており、ソフトウェアをグラフィカルに抽象化するため、開発者の直観的理解を助けるものとなっている。また、Object Constraint Language(OCL) によりダイアグラムベースの UML に対し整合性を高める機構も用意している。

一方、形式手法は数学的な厳密性を持ち、形式仕様言語と共に多くの研究、開発が行われてきた。形式手法は、形式仕様言語による厳密なモデリングやその機械的な検証を提供し、ソフトウェアの信頼性向上に大きく貢献するものとなっている。形式手法は、これまで特別に高い信頼性を要求される、原子力発電所の制御システムや航空機の管制システム等に利用されてきたが、一般的なソフトウェアの開発現場で使用され始めている。これは、電子商取引等、高信頼であることが一般的にも求められるようになった他に、形式仕様言語が記述性、可読性および再利用性に対しても有効なものとなりつつあるためである。

そういった背景のもと、近年形式手法とオブジェクト指向技術の融合に関する研究がさまざまなレベルで行われている。これはオブジェクト指向技術に対して、形式手法による

厳密性や機械的な検証技術を組み合わせることで、高い信頼性を得ようという試みである。特に分析や設計の早い段階での検証が重要であり、UML を用いた仕様のレベルでの高度な計算機による支援が求められている。UML は巨大な表記法の集合であり、ダイアグラムも個別にいくつか用意され、現在ではそれらの形式的な意味論が整備されていない。それにより UML による仕様の計算機による自動的な解析や、その振舞を確かめるべく仕様の実行等が不可能となっている。

本研究の目的は、UML と形式仕様言語との対応をとり計算機による UML モデルの意味解析のための枠組を提案することである。そのことにより、開発の早い段階において実装言語によらない UML モデルのレベルでの仕様のチェックを行い、仕様の誤りの早期発見を促すことを目的とする。対象とする UML は、クラス図に OCL による制約を加えたものとする。形式仕様言語としては、代数仕様言語 CafeOBJ を使用する。代数仕様はデータ抽象の概念を基とし、オブジェクト指向との親和性が高いものとなっている。また、CafeOBJ は実行可能な仕様言語で、その処理系を用いてシミュレーションやラピッドプロトタイピングを行うことができる。この実行可能性を UML の解析に応用することを考える。

本研究では、クラス図からはシステムの構造に関する情報、OCL からは、チェックすべき表明および振舞に関する情報を取得し、CafeOBJ の振舞仕様として定義する。振舞仕様はその意味を隠蔽代数に持ち、システムの状態を抽象的に捉えデータとプロセスを統一的に扱う枠組を提供する。この定義により変換された振舞仕様を CafeOBJ の処理系を用いて、シミュレーション実行や OCL による表明のチェックを行う。

本論文は 5 章から成り、構成は次の通りである。第 2 章では UML、OCL、形式手法および形式仕様言語の概要について述べる。特に本研究で利用する振舞仕様と振舞仕様が記述可能な代数仕様言語 CafeOBJ の説明を行う。第 3 章ではクラス図に OCL により制約を加えた UML による仕様を、CafeOBJ の振舞仕様による記述方法について説明し、いくつかの例題を示す。第 4 章では UML から変換された振舞仕様を CafeOBJ の処理系を用いて解析する方法を、いくつかの例を用いて説明する。そして最後に第 5 章ではまとめと今後の課題について述べる。

第 2 章

背景

本章では、本研究の背景となるオブジェクト指向モデリングのための UML と OCL、および形式仕様について説明する。特に本研究で利用する代数仕様言語 CafeOBJ について説明する。

2.1 Unified Modeling Language

2.1.1 概要

オブジェクト指向分析・設計においては、対象となるシステムを図解してモデル化を行ってきた。80 年代終りから 90 年代始めにかけて多くのオブジェクト指向開発方法論が登場したが、各方法論が図解のための独自の記法を持っていた。そこでこういったダイアグラムをベースとしたモデルの記法をそのモデル構成要素の定義とともに統一したものが UML(Unified Modeling Language) である。UML は Grady Booch による Booch method[21]、James Rumbaugh による OMT[18]、Ivar Jacobson による OOSE[17] を基に Rational Software 社を中心に開発され、97 年にオブジェクト技術標準化団体 OMG(Object Management Group) によって、標準モデリング言語として正式に認められた。

開発方法論は原則的には、対象となるシステムを表現するために使用される記法と、開発を行うために踏むべき手順となるプロセスから構成されるが、UML は記法のみを定めるものである。よって UML 自体が特定の手法、プロセス、また対象とするドメイに縛られることがなく、一般的に広範囲な利用が可能となっている。また、UML はシステム開発の多くのフェーズをカバーしており、上位のフェーズで作成した UML モデルを下位のフェーズへ引き継ぐことができる。このことにより一貫性のあるドキュメント作成が可能

となる。

また、UML はその名が示す通り、モデル作成のための言語であり、他のプログラミング言語と同様、シンタックスとセマンティクスを定義している。これによって、単なるダイアグラムの描き方の統一ではなく、UML を利用してオブジェクト指向モデルを表現したダイアグラムに対する同一の解釈を試みている。

UML は、OMG Unified Modeling Language Specification[19] によりその仕様が与えられている。このドキュメントでは、UML のセマンティクス、シンタックスとしての各ダイアグラム、およびオブジェクト制約言語 OCL 等の記述がされている。

2.1.2 ダイアグラム

UML では９種類のダイアグラムを定義している。これらのダイアグラムは、対象となるシステムを様々な視点から表現する手段を与える。これらの９種類ダイアグラムは、構造的ダイアグラムと振舞的ダイアグラムに分類される。

- **構造的ダイアグラム**

  対象となるシステムの静的な側面に焦点を当てたダイアグラムである。構造的ダイアグラムは次のダイアグラムからなる。

  - **クラス図**

    クラス、インタフェースおよびコラボレーションとそれらの関係を示すもの。クラス図によりシステムの静的な側面を取り扱う。

  - **オブジェクト図**

    オブジェクトとそれらの相互の関係を示すもの。オブジェクト図によりインスタンスの静的なスナップショットを表現する。

  - **コンポーネント図**

    コンポーネントとそれらの相互の関係を示すもの。コンポーネントは通常、１つ以上のクラスやインターフェース等が割り当てられるが、それらコンポーネントの実装的な側面を表現する。

  - **配置図**

    ノードとそれらの相互の関係を示すもの。配置図によりシステムアーキテクチャの配置に関する静的な側面を表現する。

- **振舞的ダイアグラム**

  対象となるシステムの動的な側面や相互作用に焦点を当てたダイアグラムである。振舞的ダイアグラムは次のダイアグラムからなる。

  - **ユースケース図**

    ユースケースおよびアクター (クラスの一種) と、それらの関係を示すもの。システムの振舞いの組織化またはモデリングする際に重要となるもの。

  - **シーケンス図**

    メッセージの時間順序によるオブジェクト間の相互作用を示すもの。オブジェクトとそれらが送受信するメッセージによって表現する。

  - **コラボレーション図**

    メッセージを送受信するオブジェクトの構造的な側面を示すもの。オブジェクト、オブジェクト間のリンク、およびそれらのオブジェクトが送受信するメッセージによって表現する。

  - **ステートチャート図**

    状態、遷移、イベント等からなる状態マシンを示すもの。主にクラスの振舞を表現する場合に用いられ、オブジェクトのイベント順の振舞を表現する。

  - **アクティビティ図**

    システム内部のアクティビティ間のフローを示すもの。アクティビティ図によりシステムの機能をモデリングし、オブジェクト間での制御のフローをを表現する。

2.1.3 OCL(Object Constraint Language)

OCL(Object Constraint Language) は、モデル記述のための形式言語であり、UML により作成されたモデルが整合的であるためにモデル要素や関係の間で満たすべき制約を式として定義することができる。

OCL は 95 年に IBM 社によりビジネスモデルを記述する言語として開発され、オブジェクト指向開発方法論である Syntropy[25] の影響を強く受けているものとなっている。その後、IBM 社と Objectime 社によって UML の構成要素として OMG に提案され、UML1.1

からUMLの一部として導入された。また、UMLのセマンティクス記述にも積極的に使われ、UMLメタモデルの整合性の向上にも貢献している。

OCLは、ビジュアルな表現では規定することが困難な、クラスやオブジェクトに対する多くの制限に関する情報を、制約として表現する。この制約は、グラフィカルなオブジェクト指向モデルを補完するものとなる。

OCLは[20]において次のようなものであるとされている。

1. OCLはオブジェクト指向開発に用いられる各モデルおよびその他の成果物に対して、追加的な情報を表現することができる言語である。

2. OCLは正確で厳密な言語であることに加え、オブジェクト技術者やその顧客にも比較的容易に読み書きができるような言語である。

3. OCLは宣言的な言語である。OCLによる式は副作用を持つことができない。OCL式によってシステムの状態を変化することはない。

4. OCLは型を持つ言語である。

OCLはダイアグラムベースであるUMLモデルに対して、ビジュアルではない式として制約を加えるものである。OCLでは、クラスやインタフェースに対する不変条件、メソッドに対する事前/事後条件を制約として記述することができる。

- 不変条件

  クラスやインタフェースのすべてのインスタンスが、常に満たす条件

- 事前条件/事後条件

  メソッドが実行される前/後に常に満たす条件

先にも記した通り、OCLは型付き言語である。よってOCL式は常に型を持つ。OCL式のトップレベルはいつもBoolean型である。

OCLにおける型は、大きく次の２つに分類される。

- 標準ライブラリとして用意されるもの。このライブラリは次の２つに分類される。
  - **基本型**: Boolean,Integer,Real,String
  - **コレクション型**: Set,Bag,Sequence

- UML クラス図から導出されるもの。これらの型を**モデル型**と呼ぶ。これは、クラス図におけるあらゆるクラスが OCL 式内で型として扱えることを示す。
  - class
  - attribute
  - operation
  - association

  OCL では副作用を持つ式を書くことができず、OCL 式によりシステムの状態が変更することはない。よってモデル型のオペレーションは副作用を持たないもののみを扱うことができる。また、モデル型の関連は、関連するクラスへの誘導を可能とするものである。

2.2 形式仕様

1968 年 Garmisch にて開かれた NATO 会議にて、開発されるソフトウェアの危機的状況が認識され、既存の工学で行われていたことと同様のことをソフトウェア開発においても行いたいという願望の下、ソフトウェア工学 (Software Engineering) なる学問が提唱された。このソフトウェア工学の成り立ちからも、ソフトウェアの仕様化技術はソフトウェア工学の中核的なテーマとして位置付けられる。これは、ソフトウェアの仕様が、最終的に作成されるソフトウェアの品質に多大な影響を及ぼすためである。

ソフトウェアの**仕様** (Specification) とは、開発するソフトウェアの性質を定義したものであり、プログラムがいかに (How) を記述するのに対し、仕様は何を (What) を記述するものである。仕様は、ソフトウェア開発における契約としての役割、ソフトウェア自身の理解およびその適用の方法のためのドキュメンテーションとしての役割がある。

仕様は一般的に、形式的 (formal) であるものと非形式的 (informal) であるものとに分類される。形式的であるとは、数学を基礎として持つことを意味する。形式的に記述される仕様を**形式仕様** (formal specification) と呼ぶ。仕様が形式的であるか否かは、一般的にそれぞれ利点と欠点を持つ。自然言語やダイアグラムをベースとする非形式的な仕様では、実務者にも扱い易く可読性も高い反面、曖昧性、冗長性、整合性等で優れているとは言いがたく、一方形式的な仕様では厳密性や無矛盾性、および検証可能性の点で優れている。一般に仕様記述のための言語を仕様言語 (specification language) と呼び、形式仕様のための仕様言語を**形式仕様言語** (formal specification language) という。非形式的な仕様言語の代表的なものとして前章で説明した UML がある。

形式仕様言語は、その基盤に厳格な数学モデルを持ち、これまで多くの言語が研究、開発されてきた。多くの形式仕様言語は、その形式化に基づくソフトウェア工学的方法、すなわち**形式手法** (formal method) と共に発達してきた。

形式仕様言語には、その形式化によって代数的アプローチ、モデル指向的アプローチ、並行プロセス論的アプローチ等がある。

表 2.1: 各形式仕様言語

代数的アプローチ	Clear, OBJ, Larch, ...
モデル指向的アプローチ	VDM, Z, B, ...
並行プロセス論的アプローチ	CSP, CCS, ...

2.3 代数仕様言語 CafeOBJ

2.3.1 概要

代数的アプローチにおける形式仕様は、1970 年代の半ばに**等式論理** (equational logic) と呼ばれる等式のみを述語に持つ論理に基づいて、**抽象データ型** (abstract data typ) を厳密に表現する方法として提案された。抽象データ型は、データの表現形式やデータ型上の演算の実現手段を記述するのではなく、演算の振舞を演算から生成される項の間の関係によって定める。抽象データ型はデータの集合 (台集合) とその上の演算を一体化させたものであり、台集合、台集合上の演算、演算から生成される項の間の等式の 3 つで表現される。このデータ型は、数学的に代数として捉えることができるため、このような形式仕様を、**代数仕様** (algebraic specification) といい、また代数仕様を記述するための仕様言語を**代数仕様言語** (algebraic specification language) という。

代数は、通常 1 つの台集合とその上に定義されたいくつかの演算との組で表される。**多ソート代数** (many sorted algebra) は、いくつかの種類 (**ソート；sort**) の要素から構成される台集合を含めるように拡張されたものである。**順序ソート代数** (order sorted algebra)[3] は、さらにソート間に半順序関係を定義できるように拡張したものである。また、現在では**隠蔽代数** (hidden algebra)[4] という新しい枠組が提案され、抽象データ型にとどまらず、抽象状態機械を表現する手法の試みがされている。

代数仕様言語 CafeOBJ[1] は、OBJ3[2] の流れを汲む実行可能な仕様言語である。OBJ3 は上述の等式論理と順序ソート代数を意味定義の基礎として持つ。これに対し CafeOBJ は、複数の論理の組合せを仕様の意味モデルとして持つことができるという特徴を持つ。CafeOBJ が仕様の意味モデルとして持つことができる論理は、多ソート代数 (many sorted algebra)、順序ソート代数 (order sorted algebra)、隠蔽代数 (hidden algebra)、書換え論理 (rewriting logic)[5] である。これらの複数の論理を任意の組合せにより開発対象のシステムに応じた意味モデルの選択が可能となっている。

CafeOBJ は仕様の強力なモジュール化機構を持つ。モジュールの輸入およびモジュールのパラメータ化機構が用意されており、抽象度、再利用性の高い仕様記述が可能となっている。また、CafeOBJ は実行可能な仕様言語である。仕様をそのまま機械的に実行可能であり、仕様のシミュレーション実行やラピッドプロトタイピングとして利用することができる。この操作的意味論は項書換えシステム (term rewriting system) に基づいている。

以降の節では、本研究に関連する CafeOBJ における仕様化パラダイムである、振舞仕様および計算機による検証の支援環境である処理系について説明する。

2.3.2 振舞仕様

隠蔽代数に基づいて記述される仕様を**振舞仕様** (behavioural specification)[1][4] と呼ぶ。隠蔽代数ではシステムをブラックボックスとみなして内部の構造ではなく、外部からみた振舞いを扱う。これは情報隠蔽の概念を自然に扱うことができることを意味する。

システムの状態空間は**隠蔽ソート** (hidden sort) によって表現される。この隠蔽ソート上にシステムの状態を遷移させる**操作演算** (action)、およびシステムの状態を観測する**観測演算** (observation) が定義される。観測の結果は抽象データ型で現され、隠蔽ソートと明示的に区別するために**可視ソート** (visible sort) と呼ばれる。このように隠蔽代数では、データを現す可視ソートとシステムの状態を現す隠蔽ソートの 2 種類のソートを扱う。操作演算と観測演算を**振舞演算** (behavioural operation) と呼ぶ。ここで振舞仕様の例として、カウンタの仕様を次に示す。カウンタは、整数を引数にとりその数を加える操作演算 (add) と、現在のカウンタの状態を得るための観測演算 (amount) を持つ。

```
mod* COUNTER{
  protecting(INT)
  *[ Counter ]*
  bop add : Int Counter -> Counter -- action
  bop amount : Counter -> Int      -- observation
  op init : -> initial state

  var I : Int
  var C : Counter
  eq amount(init) = 0 .
  eq amount(add(I, C)) = I + amount(C) .
}
```

CafeOBJ では、仕様をモジュールを単位として記述される。モジュール名は mod* の後ろに指定し (COUNTER)、モジュールの本体は {} によって囲む。*[]*により囲われた部分が隠蔽ソート (Counter) の定義であり、このソートによってカウンタの状態空間を扱う。カウンタのデータ、つまり可視ソートは整数 (INT) であり、protecting によって CafeOBJ

の組込モジュールを輸入している。bop および op に続く部分が演算子の定義であり、特に bop により振舞演算を定義する。カウンタでは観測演算として amount、操作演算として add が定義されている。op はそれ以外の演算の定義に用いられ、引数の列が空の演算子は定数の宣言となる。カウンタの例では、隠蔽ソートの定数として init が定義されており、これは初期状態を現す**隠蔽定数** (hidden constant) となっている。ソートと演算子の定義をその仕様の**指標** (signature) と呼び、システムとその外界のインタフェースを定義していると見ることができる。

等式の宣言は eq に続けて行い、条件付きの等式は ceq に続けて行う。等式内で使用する変数の宣言は、var に続けて行う。指標がシステムのインタフェースを表すのに対し、システムの振舞いを等式によって規定する。カウンタの例では amount に関する等式が 2 つ定義されているが、このように振舞仕様では操作演算によって状態がどのように変化するかということを、観測演算による観測結果によって規定していく。

代数仕様において指標をグラフィカルに記述する記法として **ADJ 図** (ADJ diagram) がある。

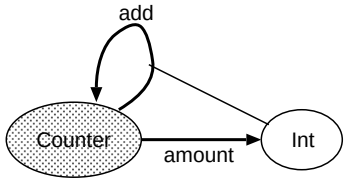


図 2.1: モジュール Counter の ADJ 図

2.3.3 射影演算

隠蔽代数として記述されたシステムをいくつか集めて**合成** (compose) することでより大きなシステムを記述することができる。これはあるシステムがサブシステム (もしくはコンポーネント) から構成されるような場合の仕様記述に有効である。これには、合成後のシステム全体の状態空間からサブシステムへの状態空間に射影する、振舞演算の一種である**射影演算** (projection operator)[6][7] を用いる。このようなシステムの合成の機構を、**オブジェクト合成 (オブジェクトコンポジション)**[6][7] と呼ぶ。

オブジェクトコンポジションによる振舞仕様の例として、前出の Counter が二つから合成されるシステム 2Counter の仕様を示す。

```
mod* 2COUNTER{
  protecting(COUNTER *{ hsort Counter -> Counter1,
                        op init -> init1 } +
     COUNTER *{ hsort Counter -> Counter2,
op init -> init2 })
  *[ 2Counter ]*
  bop add1 : Int 2Counter -> 2Counter  -- action
  bop add2 : Int 2Counter -> 2Counter  -- action
  bop amount1 : 2Counter -> Int        -- observation
  bop amount2 : 2Counter -> Int        -- observation
  bop counter1 : 2Counter -> Counter1  -- projection
  bop counter2 : 2COunter -> Counter2  -- projection
  op init : -> 2Counter                -- initial state

  var I : Int   var TC : 2Counter
  eq amount1(TC) = amount(counter1(TC)) .
  eq amount2(TC) = amount(counter2(TC)) .

  eq counter1(init) = init1 .
  eq counter1(add1(I, TC)) = add(I, counter1(TC)) .
  eq counter1(add2(I, TC)) = counter1(TC) .
  eq counter2(init) = init2 .
  eq counter2(add1(I, TC)) = counter2(TC) .
  eq counter2(add2(I, TC)) = add(I, counter2(TC)) .
}
```

この仕様では、protecting によって合成するモジュール COUNTER を輸入している。この際、隠蔽ソート Counter を Counter1、Counter2 に名前を変更し、各初期状態を示す隠蔽定数も名前を変更している。よって 2Counter は Counter1、Counter2 から構成され、それらを自由に操作する。操作演算 add1 によって Counter1 のカウントアップを、操作演算 add2 によって Counter2 のカウントアップを行う。観測演算 amount1 および amount2 は、Counter1 および Counter2 の状態を観測する演算である。射影演算として二つのカウンターの状態空間から、Counter1 の状態空間を取り出す演算 counter1、Counter2 の状態

空間を取り出す演算 counter2 を定義している。二つのカウンターは全く独立に動くもので、互いに影響を与えることはない。これらの意味を観測演算に関する等式、射影演算に関する等式として規定している。

2COUNTER を ADJ 図により示すと図 2.2 のようになる。

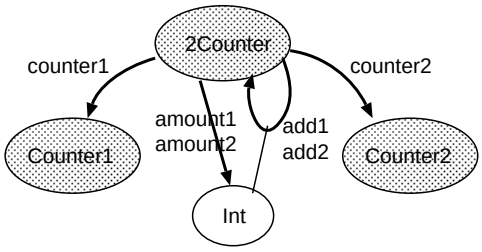


図 2.2: モジュール 2COUNTER の ADJ 図

2.3.4 項書換システム

代数仕様言語 CafeOBJ の操作的意味は、項書換システム (Term Rewriting System) によって与えられ、これに基づく処理系が実装されている。等式を左辺から右辺への書換え規則と見なすと、仕様は項書換えシステムとなる。等式による書換え規則をもとに、与えられた項を適用できる書換え規則が存在しない状態まで書換える、つまり正規形まで書換えることを**リダクション** (reduction) と呼ぶ。この項書換えにより、仕様の実行および検証を行う。CafeOBJ の処理系を用いることで計算機による支援が可能となっている。

CafeOBJ の処理系を用いてリダクションを行うには、red コマンドを用いる。カウンタの仕様による次の項をリダクションすることを例として示す。

```
amount(add(4, add(6, init)))
```

この項は、初期状態のカウンタに 6 を加え、さらに 4 を加えたときのカウンタの値を表している。この項に対して処理系を用いてリダクションした結果は次のようになる。

```
CafeOBJ> select COUNTER
COUNTER> red amount(add(4, add(6, init))) .
-- reduce in COUNTER : amount(add(4,add(6,init)))
10 : NzNat
(0.000 sec for parse, 5 rewrites(0.000 sec), 9 mches)
```

第 3 章

UMLの振舞仕様による定義

本章では、本研究の対象とするUMLについて述べ、CafeOBJの振舞仕様として扱う際の指針および実際の定義の仕方について説明し、その定義の枠組に従った振舞仕様による記述の例を示す。

3.1 対象とするUML

本研究では、対象とするUMLをクラス図とOCLとする。対象とするシステムの構造的側面をクラス図で表現したものに、OCLによる制約を加えたUMLによる仕様をCafeOBJの振舞仕様で対応づける。

3.1.1 クラス図

クラス図は、クラス、インタフェースおよびコラボレーションと、それらの関係を示すものであるが、本研究で想定するクラス図は、クラスとクラス間の関係から構成されるものとする。クラスは、クラス名、属性、操作から構成され、属性および操作の可視性はすべてpublicであるとし、他のクラスから参照可能であるものとする。関係は、関連、汎化、集約、依存があるが、関係として想定するのは関連とする。関連は、関連名、ロール名、多重度を想定する。

よって対象とするクラス図は、特別な拡張をおこなっていない、属性と操作の可視性がpuplicであるクラスと、多重度付きの関連から構成されるものを想定する。

3.1.2 OCL

OCL 型として、基本型、モデル型、コレクション型を扱う。後述するが、各型のオペレーションは基本的なオペレーションのみを想定する。また、if 文を使った制御文は想定しない。

3.2 振舞仕様による定義の指針

3.2.1 クラス図+OCL

オブジェクト指向方法論の一つである OMT[18] では、対象となるシステムのモデル化の異なる視点として、オブジェクトモデルと動的モデルを挙げている。オブジェクトモデルはシステム化対象の静的な構造をオブジェクトとして記述するものであり、動的モデルは時間とともに変化するシステムの状態を記述するものである[1] 。UML では、OMT におけるオブジェクトモデルを表現するダイアグラムとしてクラス図等の構造的ダイアグラム、動的モデルを表現するダイアグラムとしてステートチャート図等の振舞的ダイアグラムを用意している。

オブジェクト指向開発においてはこれらの２つのモデルのインタラクションによって開発を進めていくというのが基本的指針である。UML においては、システムの静的側面を構造的ダイアグラム、主にクラス図により表現し、システムの動的側面を振舞的ダイアグラムから必要に応じてダイアグラムを選択し、システムのモデル化を行っていく。

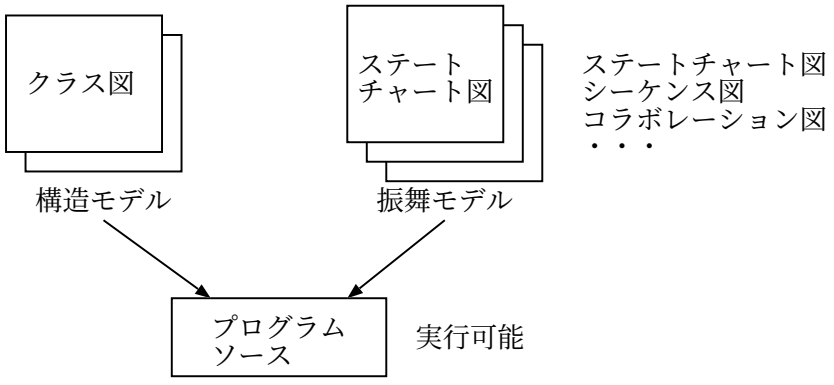


図 3.1: UML の構造モデルと振舞モデル

[1] UML では、構造モデル、振舞モデルと呼んでいる

一般的にオブジェクト指向開発モデルおよびUMLモデルの形式化を行い、検証/ モデルのシミュレーションを行う場合、構造モデルと振舞モデル両方の形式化が必要となる。

本研究ではクラス図にOCLにより制約を加えた仕様を対象とする。クラス図に対するOCLの役割を本研究では次のように捉える。

- 表明的なもの
- 振舞に関するもの

表明的なものとは、例えば、「年齢は0歳以上100歳以下である」「客の総数は部屋のベッドの総数を越えない」といったようなもので、主にOCL制約における不変条件に現れる。振舞に関するものとは、オブジェクトがどのように変化していくかを記述するものであり、OCL制約ではメソッドに対する事前/事後条件に現れる。

クラス図＋OCLを振舞仕様として定義する指針として、OCLの振舞に関する情報を振舞仕様における公理系、つまり等式として表現し、表明的なものはCafeOBJの処理系を用いた検証事項として扱う。

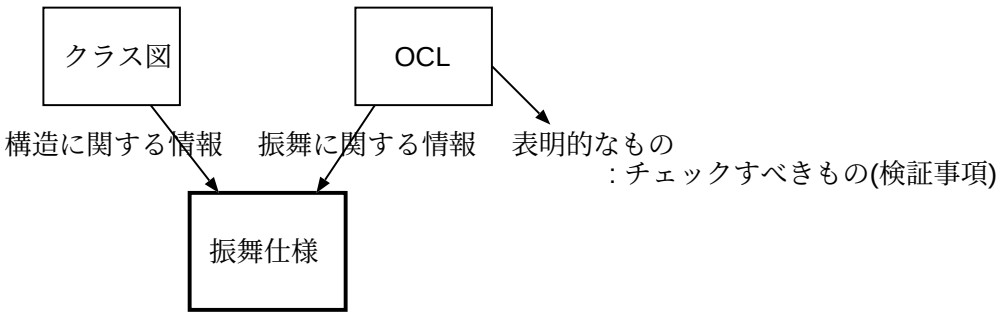


図 3.2: クラス図+OCLと振舞仕様

クラス図より、振舞仕様の指標としての情報を取得し、OCLによる制約を振舞仕様の等式として反映させる。

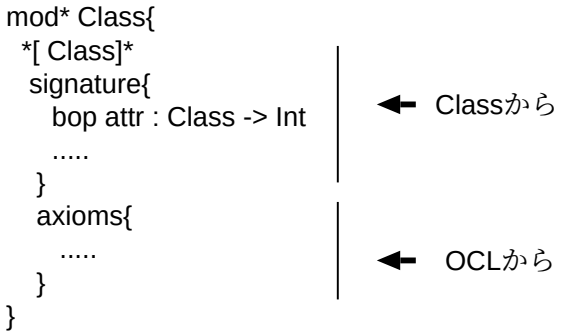


図 3.3: クラス図+OCL と振舞仕様

3.2.2 OCL と振舞仕様の仕様記述スタイル

代数仕様はデータ抽象概念を基礎とし、システム化の対象を代数として厳密にモデル化していくものである。オブジェクト指向の基礎概念はデータ抽象と捉えられるため、代数仕様とオブジェクト指向は親和性が高いものと考えられる。

振舞仕様は、操作演算によってシステムの状態がどのように遷移していくかを観測演算の観測結果によって規定していく、観測的な仕様記述スタイルといえる。

また、OCL は、OCL 式においてシステムの状態を変化させるような副作用を記述することはできない。クラスの属性値およびクエリー的なオペレーションの値[2] のみを扱って制約を記述してくため、OCL も観測的な仕様記述スタイルであるといえる。

よってクラス図＋ OCL は、振舞仕様によって自然に扱うことができる。

3.3 クラスの定義

クラスは隠蔽代数に基づくモジュールによって表現する。ここでクラス名は、隠蔽ソートに対応させる。クラスにおける属性は、振舞仕様における観測演算、操作は操作演算に対応させる。ここでクラスの属性および操作の可視性は、public であることを仮定する。このことによりクラスの属性および操作は、外部から全てアクセス可能であることを意味し、振舞仕様として扱うのは実装の詳細を隠蔽したクラスを扱う。

[2] 例えば、コレクション型における要素数を返すような演算値

クラスの操作の引数は、操作演算のアリティの可視ソート部分に対応させる。クラスの属性および操作の型は、基本的にはCafeOBJの組み込みモジュールを利用するが、ユーザ定義の型や列挙型等は対応するモジュールを適宜用意し、そのクラスにモジュールを輸入して対応させる。

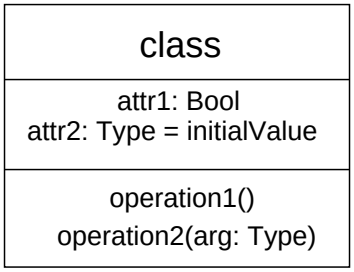


図 3.4: クラス図

図3.4のクラスを振舞仕様により記述すると次のようになる。

```
mod* CLASS{
  protecting(TYPE)
  *[ Class ]*
  -- observation
  bop attr1 : Class -> Bool
  bop attr2 : Class -> Type
  -- action
  bop operation1 : Class -> Class
  bop operation2 : Type Class -> Class
  -- initial state
  op init : -> Class
  -- equations
  var C : Class var T : Type
  eq attr2(init) = initialValue .

}
```

基本的には、クラス図からは振舞仕様の指標部分に対応させることができる。ただし属性の初期値が指定してあった場合は、初期状態の観測に観測に関する等式をに対応させることができる。

3.4 関連の定義

本研究では、関連として、単純で装飾のない関連と、多重度付きの関連を扱う。問題の簡略化のために、継承、集約を扱わない。

3.4.1 関連

２つのクラス間の単純で装飾のない関連とは、一方のオブジェクトからもう一方のオブジェクトへの双方向の誘導可能性 (navigability) を持つということである [19][22][23]。 誘導可能性とは、一方のオブジェクトへの参照[3] がなんらかの形で保証されていることを意味する。特に装飾のない関係については、誘導が双方向である。

振舞仕様において相互の誘導可能性を実現するために、射影演算によるオブジェクトコンポジションを利用する。

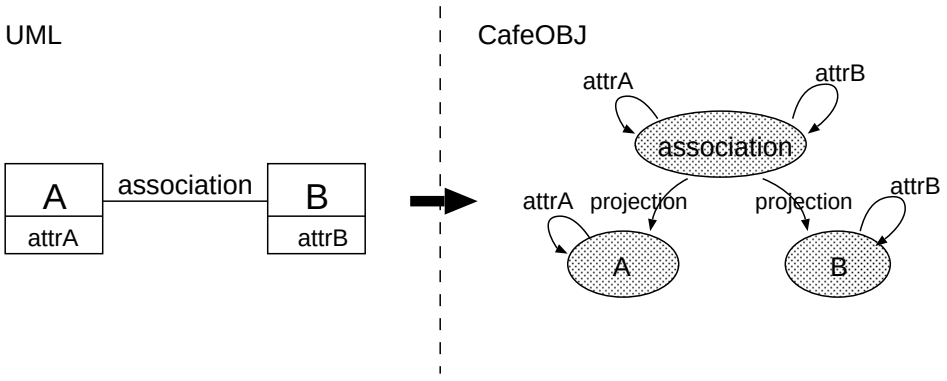


図 3.5: 関連の扱い

関連名のモジュールをコンポジションの上位モジュールとして用意し、このモジュール内において関連するクラス同士の相互参照を行う。次の関連のモジュールの仕様例のように、クラスを表す下位モジュールの観測演算、操作演算を関連モジュール内で再定義する。

```
mod* ASSOCIATION{
  pr(CLASS-A + CLASS-B)
  *[ Association ]*
  -- projection
```

[3] 参照が可能なのは、可視性が public なものである

```
  bop classA : Association -> ClassA
  bop classB : Association -> ClassB
  -- liftup operation from componet
  bop attrA : Association -> Type
  bop attrB : Association -> Type
  -- equation for component operation
  var ASSOC : Association
  eq classA(attrA(ASSOC)) = attrA(classA(ASSOC)) .
  eq classB(attrB(ASSOC)) = attrB(classB(ASSOC)) .
  ...
}
```

3.4.2 多重度付き関連

多重度付き関連は、ダイナミックオブジェクトコンポジションにより表現する。ダイナミックオブジェクトコンポジションは、コンポーネントを識別子によってパラメータ化することで動的にコンポーネントを加える機構である。関連の終端の多重度が0以上である関連をオブジェクトの識別子 $OBJECT-ID$ によって、クラスが0以上であることを表現する。この際、オブジェクトを加えるメタ操作として関連を表すモジュールに操作addを用意する。

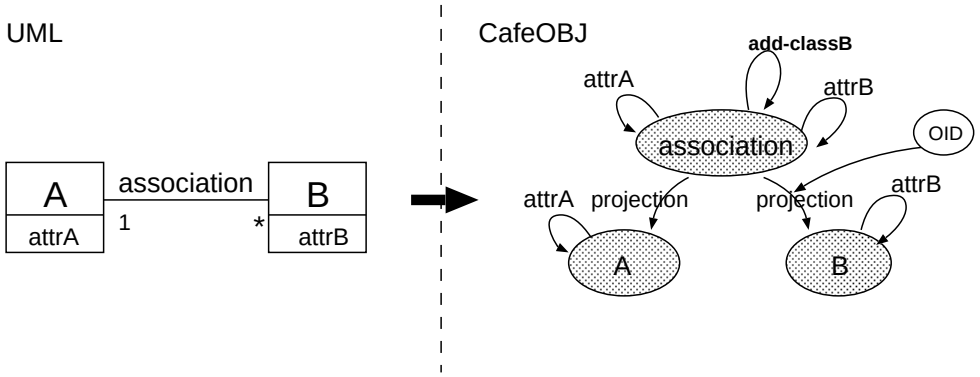


図 3.6: 多重度付き関連の扱い

次の関連のモジュールの仕様例のように、classBへの射影演算を識別子oidによりパラ

メータ化し、classB を加えるメタ操作 add を定義する。

```
mod* ASSOCIATION{
pr(CLASS-A + CLASS-B)
  pr(OBJECTID)
  *[ Association ]*
  -- projection
  bop classA : Association -> ClassA
  bop classB : Oid Association -> ClassB
  -- meta action
  bop add-classB : Oid Association -> Association
  -- liftup operation from component
  ...
  -- equation for component operation
  ...
}
```

3.5 OCL の定義

OCL 式の振舞仕様における表現の仕方、および OCL の制約を仕様としてどのように取り込むかを説明する。

3.5.1 OCL 型

2 章で説明したように、OCL 型は基本型、コレクション型、モデル型に分けられる。各型と振舞仕様との対応を次に示す。

- **基本型： Boolean,Integer,Real,String**

  CafeOBJ の組み込みモジュールである BOOL,INT,STRING モジュールのソートに対応させる。これらの組み込みモジュールは抽象データ型として表現されており、振舞仕様においては可視ソートとして扱われる。尚、数値として扱うのは Integer のみとし、Real については扱わない。OCL 型上で定義されている各演算については、組み込みモジュールにおいて定義されている演算のみを扱うものとする。

- **コレクション型： Collection,Set,Bag,Sequence**

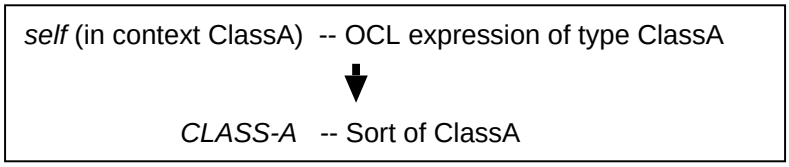
  OCL のコレクション型は、Set、Bag、Sequence 型の抽象スーパータイプである Collection 型の主要な演算のみを扱う。振舞仕様において COLLECTION モジュールを用意することで対応させる。COLLECTION モジュールのシグニチャは次の通りである。

```
mod* COLLECTION(X :: TRIV){
pr(INT)
*[ Collection ]*
-- initial state
op empty : -> Collection
-- behavioural constructors
op add : Elt Collection -> Collection { coherent }
-- atributes
bop includes : Elt Collection -> Bool
bop size : Collection -> Nat
 ...
}
```

  Collection 型の演算として、対応させるものは、includes および size とする。

- **モデル型： Attribute,operation,association**

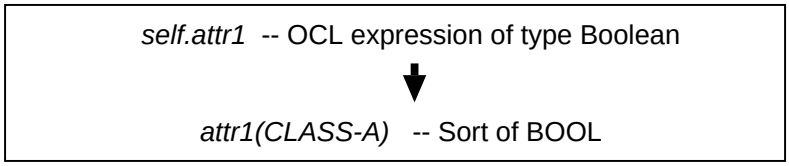
  モデル型は、UML クラスダイアグラム内のクラスを型として扱える機構である。OCL は各クラスに対して OCL 式を記述していくが、このクラスをコンテキストとして捉え、$self$ 識別子においてそのクラスを OCL 型として扱うことができる。このコンテキストは、クラスを定義したモジュールに相当し、このモジュールにおけるクラスと対応する隠蔽ソートを $self$ に対応させる。



OCLでは、$self$の後のドットノーテーションにより、attribute,operation,associationを利用する。

- **attribute**

attributeは、selfのコンテキストと対応するモジュールの属性を表す観測演算に対応させる。



- **operation**

観測演算に対応させる。ここで、振舞仕様における操作演算として扱わないのは、OCLでは副作用をもたらすものを扱わないため、クラスのメソッドをOCLとして参照できるのは、クエリー的なオペレーションのみであるため、振舞仕様としては観測演算として現れるからである。

- **association**

OCLでは関連するクラスへの誘導を可能としている。関連の終端の多重度により評価結果が変わる。

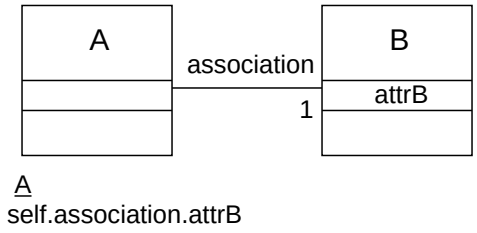


図 3.7: 多重度1の場合の関連による誘導

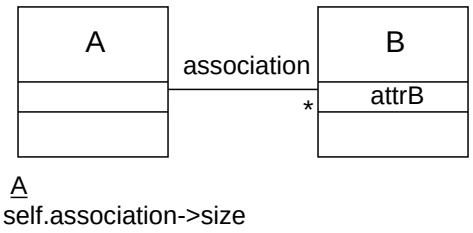
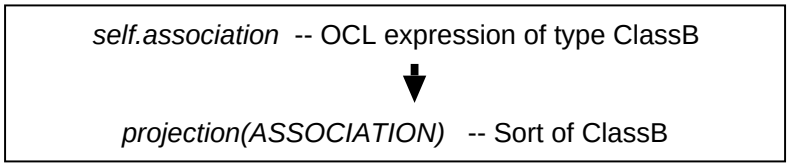


図 3.8: 多重度 1 以上の場合の関連によるコレクション型

関連終端の多重度が 1 の場合は、association による OCL 式の結果は、関連するオブジェクトとなり、型はそのクラスとなる。振舞仕様においては association によるクラスの誘導を、関連を示すモジュールからの射影演算に対応させる。



型がクラスになるために、関連先の attribute、operation の参照が可能となる。

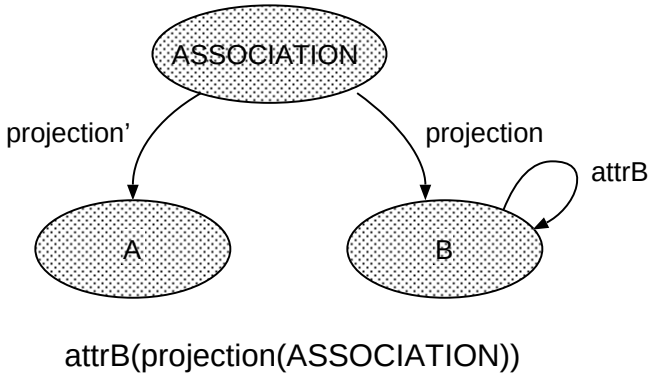


図 3.9: 多重度 1 の場合の関連による誘導の振舞仕様による扱い

関連終端の多重度が 1 以上の場合は、association による OCL 式の結果は、関連するオブジェクト群となり、関連するクラスのコレクション型となる。よってコレクション型の演算を利用することができる。

先に示した通り、多重度付きの関連はダイナミックオブジェクトコンポジションにより表現する。ここで、コレクション型として扱えるようにするために、関連を示

すモジュールにCOLLECTIONモジュールを輸入し、コレクション型の演算を使用できるようにする。

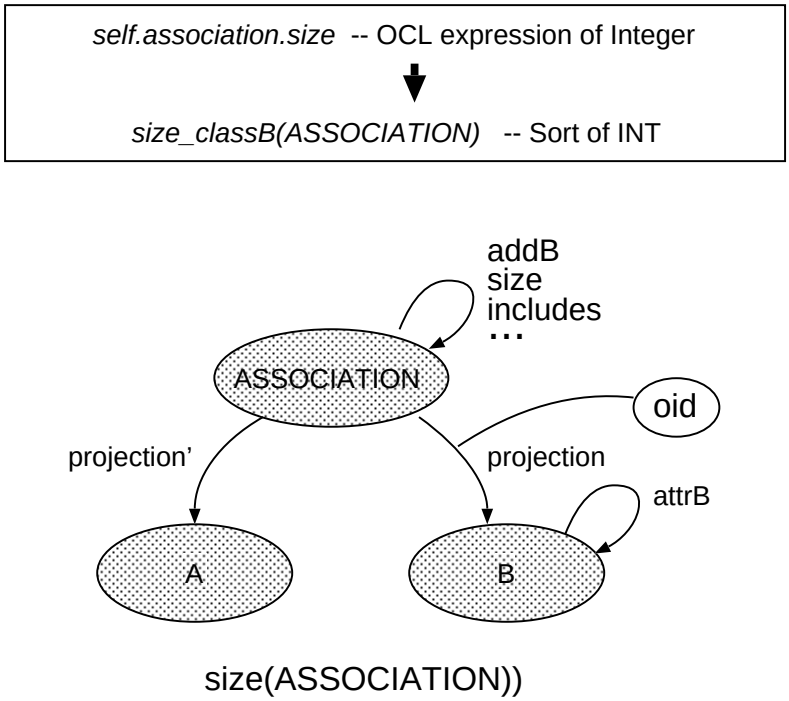


図 3.10: 多重度 1 以上の場合の関連によるコレクション型の振舞仕様による扱い

次の多重度を加味した関連のモジュールの仕様例のように、ダイナミックオブジェクトコンポジションにおけるコンポーネントの追加操作 (add 操作) と COLLECTION モジュールの同期をとるようにする。

```
mod* ASSOCIATION{
 pr(CLASS-A + CLASS-B)
  pr(COLLECTION(OBJECT-ID))
  *[ Association ]*
  -- projection
  bop classA : Association -> ClassA
  bop classB : Oid Association -> ClassB
  bop collection : Association -> Collection
  -- meta action
  bop add-classB : Oid Association -> Association
```

```
    -- collection type operation
    bop _exists_ : Oid Association -> Bool
    bop size_ : Association -> Nat
    ...
    -- equation for Collection type's operation
    var ASSOC : Association var OID : Oid
    eq collection(add-classB(OID,ASSOC)) = add(OID, collection(ASSOC)) .
    eq OID exists ASSOC = 0 exists collection(ASSOC) .
    eq size(ASSOC) = size(collection(ASSOC)) .
    ...
}
```

3.5.2 制約 (不変条件、事前条件、事後条件)

OCL 式によって表現される不変条件、事前条件、事後条件は、等式および条件付き等式として表現する。事前/事後条件は、クラスのメソッドについての制約であるため、振舞仕様の操作演算に関する等式として記述する。メソッドの事後条件に関する等式は、隠蔽ソートに操作演算を適用した項の観測結果がどのようになるかということを規定するものとなる。

```
var C : Class
eq attr(operation(C)) = ...
```

OCL では、アットマーク記号 (@) によってメソッドが適用される前の属性を参照することができる。例えば、self.attr@pre というような式を書くことができる。これは等式においては次のように表される。

```
var C : Class
eq attr(operation(C)) = attr(C) ...
```

また、事前条件は、条件付き等式によって表すことができる。

```
var C : Class
ceq attr(operation(C)) = ... if ...
```

これらの等式は、その OCL 式のコンテキスト、つまり制約が付けられているクラスを表すモジュール内に記述する。しかし関連による導出を利用している OCL 式に関しては、関連を表すモジュール内の等式として反映させる。

不変条件は、基本的にはモジュール内の等式として記述せずに、モジュールを読み込ませたあとの CafeOBJ 処理系による検証事項として利用する。等式として表現すべく不変条件も存在する (後述の記述例を参照)。

3.6 振舞仕様による記述例

これまでに説明したクラス図+OCL の振舞仕様による定義の枠組に従って、クラス図+OCL による UML の仕様を CafeOBJ による振舞仕様への変換例を示す。尚、これらの例題の振舞仕様による記述は付録に示した。

3.6.1 例題 (1): 関連による誘導のない例

OCL において関連による誘導のない場合、つまり一つのクラスにたいして OCL による制約が記述されている例を説明する。

クラス図+OCL による仕様を図 3.11 に示す。これは旅行者 (passenger) というクラスがあり、年齢が 80 才以上であればケアが必要 (care?) になり、支払いが行われていれば、支払い済 (paid?) になるというものである。

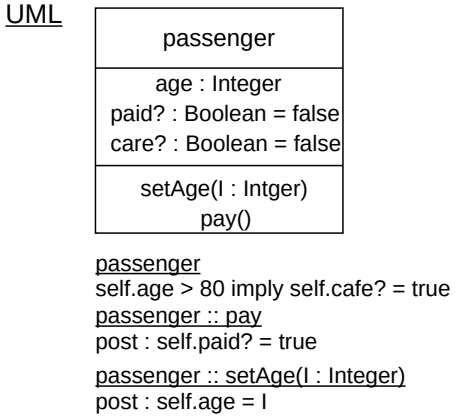


図 3.11: 例題 (1)

まずクラスから導出される、振舞仕様の指標は次のとおりである。クラスを隠蔽ソー

ト、属性およびメソッドを観測演算、操作演算に対応させ、初期状態を現す隠蔽定数 init を用意する。また、passenger クラスのデータとして Integer、Boolean があるが、これらは CafeOBJ の組み込みモジュール INT、BOOL を輸入する[4] 。

```
mod* PASSENGER{
  protecting(INT)
  *[ Passenger ]*
  -- observation
  bop age : Passenger -> INT
  bop care? : Passenger -> Bool
  bop paid? : Passenger -> Bool
  -- action
  bop setAge : Nat Passenger -> Passenger
  bop pay : Passenger -> Passenger
  -- initial state
  op init : -> Passenger
```

クラスの属性の初期値に対する等式は次のようになる。

```
eq age(init) = 0 .
eq care?(init) = false .
eq paid?(init) = false .
```

次に OCL から導出される等式について説明する。passenger クラスの不変条件、メソッド pay、setAge の事後条件に関する等式は次のようになる。

```
-- equations for invariant of passenger
ceq care?(setAge(N,P)) = true if N > 80 .
ceq care?(setAge(N,P)) = false if not(N > 80) .
-- equation for post condition of pay
eq paid?(pay(P)) = true .
-- equation for post condition of setAge
eq age(setAge(N,P)) = N .
...
}
```

3.6.2 例題 (2):関連による誘導のある例 (多重度 1)

クラス図において関連が存在し、OCL 制約において関連による誘導がある場合の例について説明する。この際、関連の多重度はともに 1 である場合である。

[4] モジュール BOOL は標準で読み込まれているため、明示的に輸入する必要はない

クラス図+OCL による仕様を図 3.12 に示す。これは、明細書 (expense-sheet) と客 (customer) というクラスがあり、注文 (order) という関連が 2 つのクラスの間に多重度 1 対 1 であるものとする。明細書の金額 (price) が客の所持金 (pocket) よりも大きい場合に支払いはカード使用 (card?) となるものである。

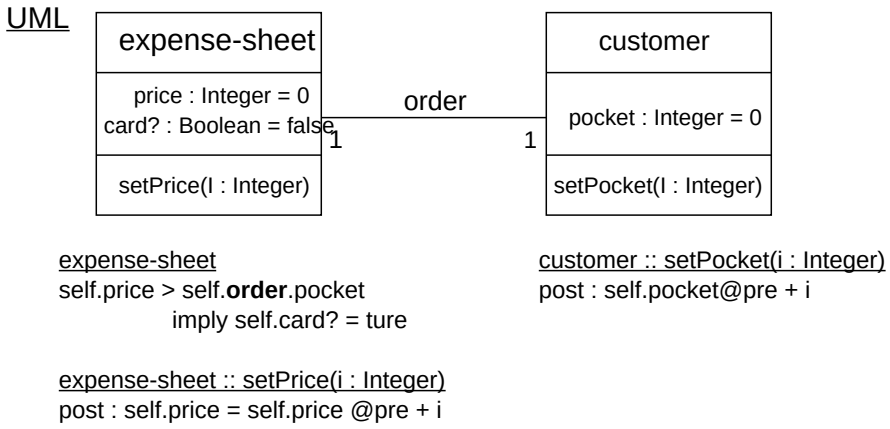


図 3.12: 例題 (2)

クラスおよび関連から導出される振舞仕様の各指標は次の通りになる。ここで、クラス 2 つと関連からなる 3 つのモジュールを用意する。

```
mod* EXPENSE-SHEET{
  protecting(NAT)
  *[ Expense-sheet ]*
  bop price : Expense-sheet -> Nat                      -- observation
  bop card? : Expense-sheet -> Bool                     -- observation
  bop setPrice : Nat Expense-sheet -> Expense-sheet     -- action
  op init : -> Expense-sheet                            -- initial state
...
}
mod* CUSTOMER{
  protecting(NAT)
  *[ Customer ]*
  bop pocket : Customer -> Nat              -- observation
  bop setPocket : Nat Customer -> Customer  -- action
  op init : -> Customer                     -- initial state
...
}
mod* ORDER{
  protecting(CUSTOMER + EXPENSE-SHEET)
  *[ Order ]*
```

```
  bop order-price : Order -> Int              -- observation
  bop order-card? : Order -> Bool             -- observation
  bop order-pocket : Order -> Int             -- observation
  bop order-setPocket : Int Order -> Order    -- action
  bop order-setPrice : Int Order -> Order     -- action

  bop expense-sheet : Order -> Expense-sheet  -- projection
  bop customer : Order -> Customer            -- projection

  op init-order : -> Order                    -- initial state
...
}
```

ORDER モジュールでは、EXPENSE-SHEET モジュールおよび CUSTOMER モジュールの観測演算、操作演算をともに再定義している。その際の名前付けは order-を演算名の前に付けている。

次に OCL から導出される等式について説明する。setPrice メソッドおよび setPocket メソッドの事後条件に関しては、EXPENSE-SHEET モジュールおよび CUSTOMER モジュールの等式として反映させる。これらは、前出の例と同様に記述するためここでは省略する。expense-sheet クラスに対する不変条件は、関連による導出を利用した制約となっている。このような制約は、関連をあらわすモジュール内の等式として反映させる。ORDER モジュール内にこの制約として次のような等式を記述する。

```
var O : Order
var I : Int
-- equations for invariant of expense-sheet
ceq order-card?(O) = true if price(expense-sheet(O)) > pocket(customer(O)) .
ceq order-card?(O) = false if price(expense-sheet(O)) <= pocket(customer(O)) .
```

また、オブジェクトコンポジションを構成するために次のような等式が必要となる。

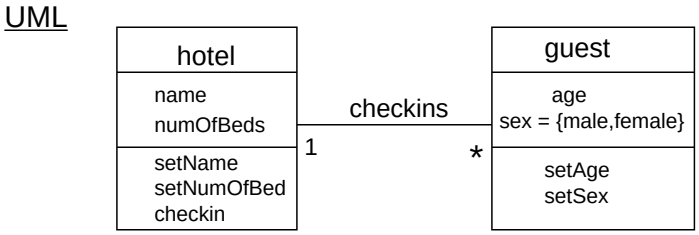
```
-- equations for observation
eq order-price(O) = price(expense-sheet(O)) .
eq order-card?(O) = card?(expense-sheet(O)) .
eq order-pocket(O) = pocket(customer(O)) .
-- equations for projection
eq expense-sheet(init-order) = init .
eq expense-sheet(order-setPrice(I,O)) = setPrice(I,expense-sheet(O)) .
eq expense-sheet(order-setPocket(I,O)) = expense-sheet(O) .
eq customer(init-order) = init .
eq customer(order-setPrice(I,O)) = customer(O) .
eq customer(order-setPocket(I,O)) = setPocket(I,customer(O)) .
```

これらの等式に関してはOCL制約の記述とは特に関係してこないため、ここでは説明しない。詳しくは、[6][7]を参照されたい。

3.6.3 例題(3):コレクション型を利用する例

クラス図において関連の多重度が0以上の場合で、OCL制約において関連先のクラスの集合としてコレクション型を利用する例について説明する。

クラス図+OCLによる仕様を図3.13に示す。これはホテル(hotel)と客(guest)というクラスがあり、チェックイン(checkins)という関連が2つのクラスの間に1対多(0以上)の多重度であるものとする。ホテルにチェックインするたびに、関連する客の数(つまりguestクラスのインスタンス数)が1増加される。チェックインの際は同じ客は重複してチェックインすることができないというものである。また、ホテルの客数はホテルが所持するベッド数(numOfBed)をこえない。



hotel
self.checkins**->size** <= self.numberOfBeds
hotel :: checkIn(g : guest)
pre : not self.checkins**->inclueds(g)**
post : self.checkins**->size** = (self.checkins@pre**->size**) + 1 and
self.checkins->**includes(g)**
hotel :: setName(s : string)
post : self.name = s
hotel :: setNumOfBed(i : integer)
post : self.numOfBed = i

図 3.13: 例題(3)

まず、guestクラスにおいて性別(sex)が列挙型となっているため、それに対応するデータ型を次のように定義する。

```
mod! SEX{
  [ Sex ]
  ops Male Female : -> Sex .
}
```

クラスおよび関連から導出される振舞仕様の各指標は次の通りになる。ここで、クラス2つと関連からなる3つのモジュールを用意する。

```
mod* HOTEL{
  pr(INT + STRING + OBJECTID)
  *[ Hotel ]*
  bop numOfBed : Hotel -> Int          -- observation
  bop name : Hotel -> String           -- observation
  bop setNumOfBed : Int Hotel -> Hotel -- action
  bop setName : String Hotel -> Hotel  -- action
  bop checkIn : Oid Hotel -> Hotel     -- action
  op init : -> Hotel                   -- initial state
...
}
mod* GUEST{
  pr(INT + STRING + SEX + OBJECTID)
  *[ Guest ]*
  bop age : Guest -> Int           -- observation
  bop sex : Guest -> Sex           -- observation
  bop setAge : Int Guest -> Guest  -- action
  bop setSex : Sex Guest -> Guest  -- action
  op init : Oid -> Guest           -- initial state
  op no-guest : -> Guest           -- error
...
}
mod* CHECKINS{
  pr(GUEST + HOTEL + COLLECTION(OBJECTID))
  *[ Checkins ]*
  bop numOfBed : Checkins -> Int        -- observation
  bop name : Checkins -> String         -- observation
  bop age : Oid Checkins -> Int         -- observation
  bop sex : Oid Checkins -> Sex         -- observation
  bop setNumOfBed : Int Checkins -> Checkins  -- action
  bop setName : String Checkins -> Checkins   -- action
  bop checkIn : Oid Checkins -> Checkins      -- action
  bop setAge : Oid Int Checkins -> Checkins   -- action
  bop setSex : Oid Sex Checkins -> Checkins   -- action
  bop hotel : Checkins -> Hotel               -- projection
  bop guest : Oid Checkins -> Guest           -- projection
  bop collection : Checkins -> Collection     -- projection
  op init-checkins : -> Checkins              -- initial state
  op errar-assoc : -> Checkins                -- errar state

  -- meta operation
  bop add : Oid Checkins -> Checkins
```

```
  -- collection type operations
  bop includes : Oid Checkins -> Bool
  bop size_ : Checkins -> Int
...
}
```

hotel クラスの checkin メソッドは、guest クラスを引数に取るが、ここでは Guest クラスのオブジェクトの識別子である Oid ソートで対応させている。CHECKINS モジュールでは、HOTEL モジュールおよび GUEST モジュールの観測演算、操作演算を再定義する。また guest クラスのオブジェクトを追加するためのメタ操作である add 演算子、コレクション型のための includes および size 演算子を用意している。

CHECKIN モジュール内には、add 演算およびコレクション型の演算の振舞いを規定する等式を次のように用意する。

```
-- equations for meta actions and collection operator
eq collection(init-checkins) = empty .
eq collection(add(O, C)) = add(O, collection(C)) .
eq includes(O, C) = includes(O, collection(C)) .
eq size(C) = size(collection(C)) .
eq collection(setNumOfBed(N, C)) = collection(C) .
eq collection(setName(S, C)) = collection(C) .
eq collection(setAge(O, N, C)) = collection(C) .
eq collection(setSex(O, SE, C)) = collection(C) .
```

これらの等式は、add 演算によるオブジェクトの追加操作がオブジェクトの集合、つまりコレクション型として扱えるように同期をとるためのものである。

次に OCL から導出される等式について説明する。hotel クラスの setName メソッドおよび setNumOfBed メソッド、guest クラスの setAge メソッドおよび setSex メソッドの事後条件に関しては、HOTEL モジュール、GUEST モジュールの等式として反映させる。これらは前出の例と同様に記述するためここでは省略する。hotel クラスの checkin メソッドの事前/事後条件は、関連による導出、しかもコレクション型を利用した制約となっている。この制約に関しては前出の例と同様に関連をあらわすモジュール内の等式として反映させる。CHECKINS モジュール内にこの制約として次のような等式を記述する。

```
-- equations for pre/post condition of checkin method
ceq checkIn(O, C) = add(O, C) if not(includes(O, C)) .
ceq checkIn(O, C) = errar-checkins if includes(O, C) .
```

この等式は checkIn メソッドをメタ操作である add 演算に対応させている。OCL 制約は checkIn メソッドの適用後に、hotel クラスと関連する guest クラスの集合に加えるとい

う意味を表しており、これを add 演算によって行うことを示している。また事前条件は、条件付き等式の if 以降に対応させている。

ここで、hotel クラスに関する不変条件、つまりホテルの客数はホテルの所持するベッド数を越えないという制約は、CHECKIN モジュール内の等式として反映させない。この制約は表明を表すものであるため、これらのモジュールを読み込ませた後のチェックすべき式として抽出する。この式は次のような項として表現する。

```
size(C: Checkins) <= numOfBed(C)
```

チェックの方法については第 4 章で説明する。

第 4 章

CafeOBJ によるUML モデルの解析

本章では、4 章における UML の振舞仕様による定義の枠組をもとに、CafeOBJ の処理系を用いたチェック方法について説明する。

4.1 概要

クラス図に OCL による制約を加えた UML モデルのチェックとして、OCL のチェックおよびシミュレーション実行を行う。次節において OCL のチェックとしてなにをすべきか、またシミュレーション実行について説明を行い、CafeOBJ 処理系を用いたチェックの全体像について述べる。

4.1.1 OCL のチェック

本研究では、UML による仕様として、クラス図に OCL の制約を加えたものを対象としている。現在、OCL は UML メタモデルの定義に活用が注目されているが [19]、UML ベースによる実際のソフトウェア開発における品質向上のためにも十分な能力を持つ。OCL を十分に活用した開発方法論である Catalysis[24] 等も存在するが、これらの方法論においても OCL のツールによるサポートの欠如の問題があることは否めない [10]。[10] では、UML ツールにおける実際的な OCL のサポートが進まないのは、OCL ツールとしてどのような機能をサポートすべきか、つまり何をチェックすべきかが未整理であることを挙げている。

本研究では、[10][12] における調査を基に OCL のチェックを次のように捉える。

- **Type Checking**

  OCL 式のシンタクティカルなチェックである。OCL は型付き言語であるため、他の型付き言語と同様な方法で型チェックを行うアプローチが考えられる。しかしながら、OCL 式はモデル型により、クラスモデルへの参照が可能となっている。よって OCL 式の型チェックを行うには、UML モデルの情報にアクセスできることが必要となる。

- **Dynamic Checking**

  システムの実行時における OCL 制約のチェックである。これは、システムのある実行の時点、つまりスナップショットにおいて不変条件等が満たされているかをチェックすることである。このシステムはプログラミングにより実際に実装されたシステムをベースとする場合や、その仕様をベースとするプロトタイプやシミュレーションをベースとすることも可能である。このチェックは、OCL 制約および UML モデルの実行可能性 (executability) が重要なポイントとなる。

- **Consistency Checking**

  OCL の制約の一貫性に関するチェックである。OCL は論理的な言語であるため、OCL 制約のなかに矛盾が含まれているかどうか発見できる可能性がある。

本研究では、OCL のチェックとして、Type Checking、Dynamic Checking を対象とする。また、Dynamic checking では、ある実行時点でのチェックであるが、それをさらに拡張したチェックとして、すべての状態に対して OCL の制約の一つである不変条件がすべての状態についてなりたつことを証明する検証も行う。

クラス図+OCL の仕様から振舞仕様への変換の枠組をもとに、CafeOBJ の実行可能性、検証可能性によりチェックを行うアプローチをとる。

4.1.2 UML のシミュレーション実行

UML のシミュレーション実行を次のように捉える。クラスのインスタンスが属性値を保持している状態をスナップショットと呼ぶ。あるスナップショットにおいて、インスタンスの属性値を変える、つまり別のスナップショットに変化することを、実行として捉える。ここで実装言語によらない状況で実行を行うことがシミュレーション実行となる。

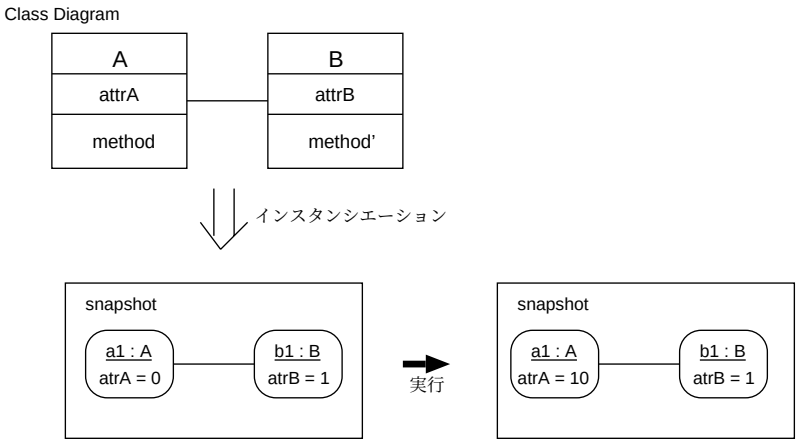


図 4.1: シミュレーション実行

クラス図においてスナップショットを作成するとは、クラスおよび関連のインスタンシエーションを行うことである。本研究では、OCL のメソッドに関する事前/事後条件を振舞仕様の等式として与えることで振舞に関する情報を得ている。項書換えシステムに基づく CafeOBJ の処理系による簡約 (reduction) によって、シミュレーション実行と同等のことを行う。

4.1.3 全体像

ここで、CafeOBJ の処理系を用いた UML モデルの解析方法の全体像を図 4.6 に示す。

まず、クラス図+OCL による UML の仕様を振舞仕様に変換する。この際、4 章で説明した定義に従い変換を行う。変換された振舞仕様を CafeOBJ の処理系に読み込ませる。ここで等式として反映されている OCL 式の Type Checking を行う。振舞仕様のモジュール内に定義されない、表明的な OCL 式は処理系に読み込まれたモジュール上で parse することにより Type Checking を行う。次に CafeOBJ の処理系を用いてシミュレーション実行、OCL 式の Dynamic Checking および OCL 不変条件の検証を行う。

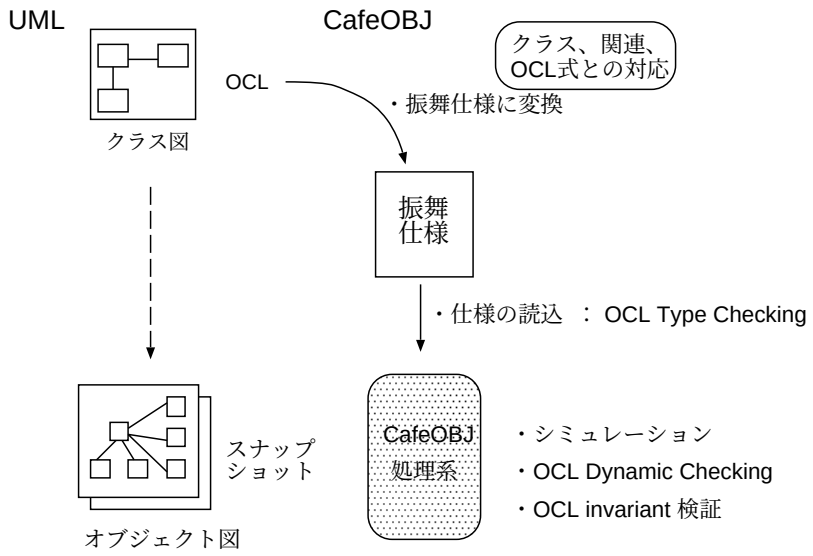


図 4.2: 全体像

4.2 UML モデルのシミュレーション実行

4 章で説明した通り、OCL の事前/事後条件より振舞いに関する情報を取得する。これはクラスのメソッドの事前/事後条件を振舞仕様における操作演算に関する等式として反映させることである。これらの等式を左辺から右辺への書換え規則とみなせば仕様は項書換えシステムとして捉えることができ、操作演算を繰り返し適用した項をリダクションすることによって実行に相応することを行う。この際、CafeOBJ の処理系を利用すれば計算機による自動的な実行を行うことができる。CafeOBJ の処理系を用いてリダクションを行うには、red コマンドを使用する。

先にも説明した通り、クラス図におけるスナップショットはクラス図から生成することのできる、ある状態である。クラスに対応する隠蔽ソートはそのクラスの状態空間を現すソートであり、ある時点でのスナップショットを項として表現することができる。ある Class に対してメソッド operation1、operation2 を順に適用した時点でのスナップショットは次の項によって表現することができる。

```
operation2(operation1(C : Class))  -- hidden sort of Class
```

では、例を用いてクラス図+OCL のシミュレーション実行について説明する。まず第 4 章で説明した例題 (2) を用いて説明する。

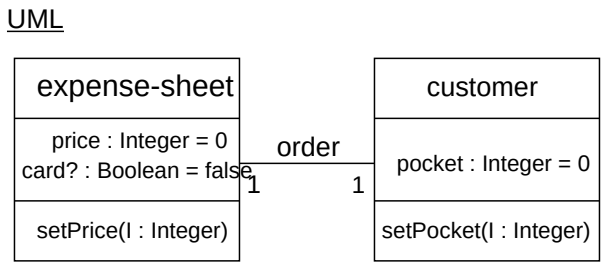


図 4.3: クラス図: シミュレーション実行例 (1)

図 4.3 のクラス図において、まず客の所持金が 5000 円で明細書の額が 3000 円である場合のスナップショットを考え、このスナップショットを snapshot1 と名づける。snapshot1 は、次の項によって表現できる。

```
setPrice(3000,setPocket(5000,init-order))  -- snapshot1
```

この際、このスナップショットにおける各クラスの属性値をリダクションを行うことで次のように取り出すことができる。

```
CafeOBJ> open ORDER
-- opening module ORDER.. done.
%ORDER> red card?(setPrice(3000,setPocket(5000, init-order))) .
false : Bool
```

次に、snapshot1 から明細書の額が 7000 円に更新された場合のスナップショットを考える。このスナップショットを snapshot2 と名づける。snapshot2 は次の項によって表現できる。

```
setPrice(7000,setPrice(3000,setPocket(5000,init-order)))  -- snapshot2
```

同様にしてリダクションによりクラスの属性値を次のように取り出す。

```
%ORDER> red card?(setPrice(7000,setPrice(5000,setPocket(5000,init-order)))) .
false : Bool
%ORDER> red price(setPrice(7000,setPrice(5000,setPocket(5000,init-order)))) .
7000 : NzNat
```

次に、snapshot2 から所持金が 10000 円に増えた場合のスナップショットを考え、これを snapshot3 と名づける。snapshot3 は次の項によって表現される。

```
setPocket(10000,setPrice(7000,setPrice(3000,setPocket(5000,init-order)))) -- snapshot3
```

同様にしてリダクションによりクラスの属性値を次のように取り出す。

```
%ORDER> red card?(setPocket(10000,setPrice(7000,setPrice(3000,
                                  setPocket(5000,init-order))))) .
false : Bool
%ORDER> red pocket(setPocket(10000,setPrice(7000,setPrice(3000,
                                  setPocket(5000,init-order))))) .
10000 : NzNat
```

以上のようにシミュレーション実行を行う。

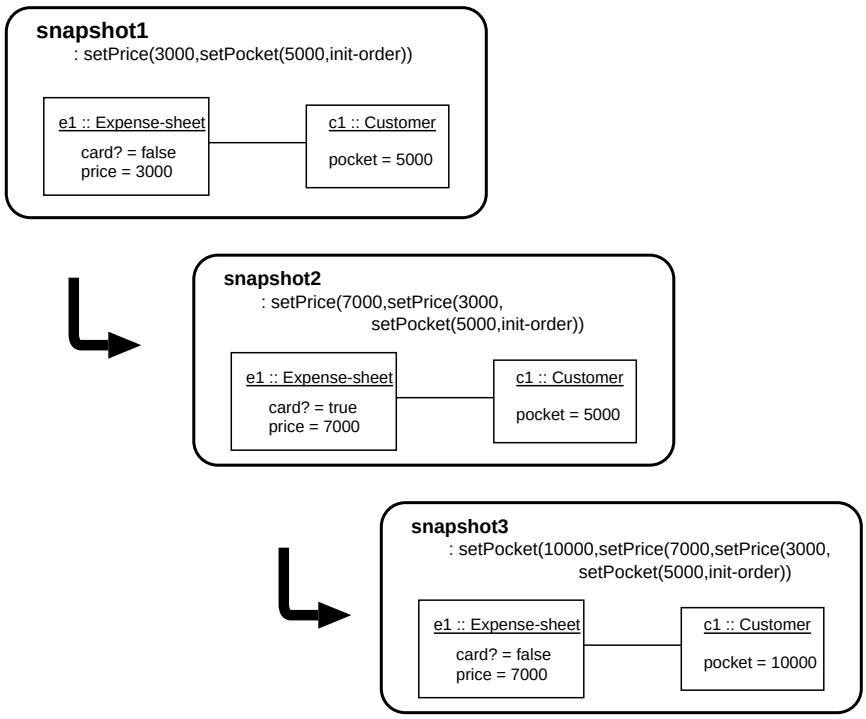


図 4.4: シミュレーション実行例 (1)

OCL の事前/事後条件は等式として記述したが、これらの制約が正しいかどうかは以上のようにシミュレーション実行してみることで調べることができる。

4.3 OCL チェック

OCL 式と振舞仕様との対応については第 4 章において説明した。この対応関係をもとに OCL 式を振舞仕様として取り込み、CafeOBJ の実行可能性を組み合わせて OCL のチェックを行う。先に説明した OCL のチェックのうち、type チェックと dynamic チェックおよび OCL 不変条件の検証を本研究では扱う。

4.3.1 type チェック

クラス図+OCL を振舞仕様として変換し、その仕様を CafeOBJ の処理系に読み込ませることで OCL 式の type チェックを行う。CafeOBJ は、強く型付けされた言語であると言えるため、処理系において不整合な項の検出が可能である。よって OCL 式を振舞仕様化することで OCL 式の型チェックが原理的に可能である。OCL 型のチェックには、基本型、コレクション型に加え、クラス図から導出されるモデル型も扱う必要があるが、クラスおよび関連に関する情報も同様の振舞仕様として扱うため、モデル型のチェックも可能となる。ただし、OCL のすべてのライブラリとの対応がとれているわけではないため、対応がとれている範囲内での型チェックとなる。

また CafeOBJ の処理系に用意されている parse コマンドを用いて、OCL の type チェックを行うこともできる。図 4.5 のようなクラス図に対して OCL 式を記述していく場合を想定する。

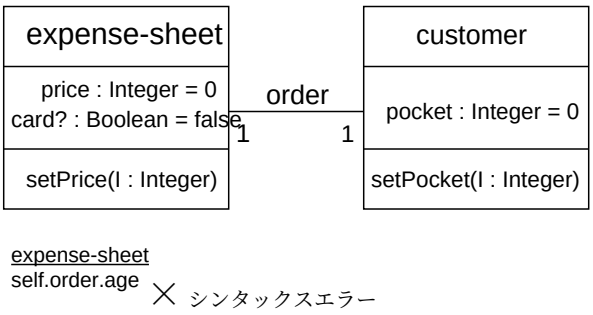


図 4.5: Type チェックのイメージ

この際、図 4.5 にあるように存在しない属性を参照するような OCL 式を記述しているものとし、この OCL 式に相当する項を次のように parse コマンドでチェックを行う。

```
CafeOBJ> open ORDER
-- opening module ORDER.. done
```

```
%ORDER> parse age(O:Order) .
[Error] no successfull parse
  (age ( O:Order ))
("age" "(" "O:Order" ")") : SyntaxErr
```

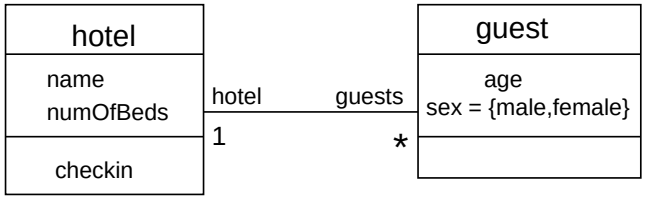
上記のようにシンタックスエラーを検出する。

4.3.2 dynamic チェック

dynamic チェックは、OCL の不変条件 (表明) がシステムのある実行時点、つまりシステムのあるスナップショットにおいて満たされているかどうかチェックすることである。このチェックを行うにはシステムが実行可能であることが求められるが、先に説明した UML モデルのシミュレーション実行と組み合わせることで OCL の dynamic チェックを行う。

シミュレーション実行において示した通り、ある隠蔽ソートに対し操作演算を繰り返し適用することである時点のスナップショットを現す項を作成し、その状態において表明として抽出した OCL 式をチェックする。

以降では、第 4 章の例題 (3) をもとに dynamic チェックについて説明する。まず、hotel



hotel
self.guests->size <= self.numOfBeds ◀ 表明
hotel :: checkIn(g : guest)
pre : not self.checkins->inclueds(g)
and self.numOfBeds > self.guests.size
post : self.checkins->size = (self.checkins@pre->size) + 1 and
self.checkins->includes(g)

図 4.6: dynamic チェックの例

クラスに関する不変条件をチェックすべき表明として抽出する。よってこの不変条件は、等式として反映させない。この不変条件は、ホテルのベッド数が客数よりも多くなることを示すもので、この OCL 式がシステムのあるスナップショットにおいて成り立つかどうかをチェックする。この不変条件は次のように表される。

```
numOfBed(s) >= size(s)  -- s : System
```

シミュレーション実行の際と同様に、あるスナップショットを項として表現し、上記の式を調べ、OCL 制約の dynamic チェックを行う。

dynamic チェックの例を次に示す。まず、ホテルの部屋数を 5 部屋とする。

```
CafeOBJ> open SYSTEM .
-- opening module SYSTEM.. done.
%SYSTEM> eq numOfBed(init) = 5 .
```

宿泊客が 3 人チェックインした状態で、条件が成り立つかどうかチェックする。

```
%SYSTEM> beq s1 = checkIn(o2, 27, Male, checkIn(o1, 24, Female,
init-system)) .
%SYSTEM> bred numOfBed(s1) >= size(s1) .
true : Bool
```

次に宿泊客が 6 人チェックインした状態で、条件が成り立つかどうかチェックする。

```
%SYSTEM> beq s2 = checkIn(o6, 26, Female,
                checkIn(o5, 22, Female, checkIn(o4, 21, Male, checkIn(o3, 30, Male, s1))))
%SYSTEM> bred numOfBed(s2) >= size(s2) .
true : Bool
```

ここで、チェックイン数が 6 になっても、宿泊者数が部屋数を越えない。これは checkIn メソッドの事前条件が効いているためである。最後に追加された客のオブジェクト ID(o6) は含まれていないことが次のようにわかる。

```
%SYSTEM> bred includes(o6,s2) .
false : Bool
```

以上のようにシミュレーション実行と組み合わせることで、システムの実行時における OCL 制約のチェックを行う。

4.3.3 OCL 不変条件の検証

4.3.2 では、OCL 制約があるスナップショット、つまりある実行時点で成り立つかどうかをリダクションにより調べるものであった。しかしながら、不変条件は、システムのすべての状態に対して成立すべき条件を宣言として与えたものである。

ここでは、OCL の事前/事後条件で与えられたシステムの振舞いにおいて、表明として抽出した不変条件がシステムのすべての状態について成立するかどうかを検証することを考える。

検証の例として、4.3.2 と同様に例題 (3) のホテルの例を使用する (図 [?])。検証すべき OCL 不変条件は

```
context hotel : self.numOfBed >= self.guests->size
```

である。これを CafeOBJ の処理系を用いて、

```
numOfBed(s) >= size(s) -- for all s : System
```

が成立することを証明する。証明の流れは次のようになる。尚、proof score を付録に添付する。

`s ( : System )` の構造に関する帰納法により解く。

Base case (`s = init-system`) :

```
**> base case ( s = init-system ) : numOfBed(init-system) >= size(init-system)
red numOfBed(init-system) >= size(init-system) .
-- should be true
```

I.H. (`s = sys`) : `numOfBed(sys) >= size(sys)` が成り立つと仮定する。ただし、このままでは書換えルールとして使いにくいため、次のように等式を定義する。

```
**> induction hypothesis ( s = sys ) : numOfBed(sys) >= size(sys)
--  in CafeOBJ => numOfBed(hotel(sys)) >= size(collection(hotel(sys)))
eq numOfBed(hotel(sys)) >= size(collection(sys)) = true .
```

I.S. (`s = checkIn(o, n, s, sys)`) : `include(o,sys)` ついて場合分けをする。

case (`binclude(o,s) = true`) :

```
**> case : include(o,sys) = true
eq includes(o, sys) = true .
red numOfBed(checkIn(o, n, s, sys)) >= size(checkIn(o, n, s, sys)) .
-- should be true
```

case (`binclude(o,s) = false`) : この場合では、証明を容易にするため、I.H. を分割を次のように分割し、おのおのについて調べる。

```
**> case : include(o,sys) = false
eq includes(o, sys') = false .
-- I.H.(1)
eq numOfBed(hotel(sys')) > size(collection(sys')) = true .
red numOfBed(checkIn(o, n, s, sys')) >= size(checkIn(o, n, s, sys')) .
-- should be true
eq includes(o, sys'') = false .
-- I.H.(2)
eq numOfBed(hotel(sys'')) = size(collection(sys'')) .
red numOfBed(checkIn(o, n, s, sys'')) >= size(checkIn(o, n, s, sys'')) .
-- should be true
```

以上のようにユーザが証明の指針について手を加えながらCafeOBJの処理系を用いて半自動的に行うことができる。

第 5 章

まとめと今後の課題

5.1 まとめ

5.1.1 UML の振舞仕様による定義

本研究では、UML による仕様と代数仕様言語 CafeOBJ の振舞仕様として記述する際の対応関係を定義した。その際、UML としてクラス図に OCL による制約を付加した仕様を対象とした。定義の指針としては、クラス図より静的構造に関する情報を得、OCL のメソッドに関する事前/事後条件より振舞いに関する情報を得る。クラス図から抽出される静的構造を仕様の指標とし、OCL の事前/事後条件を振舞いを規定する等式として反映させた。クラス図を構成するものとしてクラスとその関連があるが、クラスは隠蔽代数に基づくモジュールとして表現し、関連はそれらのモジュールを合成する射影演算に基づくオブジェクトコンポジションとして表現した。このことにより関連の誘導可能性を表すことができる。

また、OCL 型の表現と制約の等式としての表現について述べた。OCL 型は、モデル型の表現が重要となるが、先のクラスおよび関連の定義により関連による導出やコレクション型を扱う方法提を提案し、関連も含めたオブジェクトの集合に対して制約を記述する仕組みを示した。

OCL による制約は、クラスの属性値やコレクション型によるクエリー的なオペレーションの値をを利用して記述していく。これは観測的な仕様記述スタイルである。振舞仕様は観測演算を通して振舞いを記述していくため観測的な仕様記述スタイルといえ、クラス図+OCL の仕様を自然に表現することができる。

5.1.2 UML モデルの解析

UML との対応関係をもとに変換された振舞仕様を CafeOBJ の処理系を用いて解析する方法を提案した。解析としてシミュレーション実行と OCL のチェック方法を考察した。シミュレーション実行の際の、あるシステムの実行時点を示すスナップショットを項として表し、その項に観測演算を適用したものを機械的にリダクションすることでその時点での属性値を取り出すことができる。このような実行により、その仕様の振舞いを確認することで、OCL のメソッドに対する事前/事後条件を確認することが可能である。

また、OCL のチェックとして type チェックと dynamic チェックを行う枠組みを考察した。OCL の type チェックを行う際に重要なポイントとなるモデル型のチェックであるが、クラスや関連も同じ振舞仕様として表現しているため、CafeOBJ の処理系を用いて型チェックを行うことができることを示した。また、OCL の dynamic チェックのためには仕様の実行可能性が重要となるが、これは先のシミュレーション実行と組み合わせることでチェックが可能となることを示した。

また、OCL 不変条件が、あらゆるシステムの状態に対して成立することを示す検証を例題を通して示した。この検証では、システムの状態に関する構造帰納法により証明を行った。これは、OCL のメソッドに対する事前/事後条件のもとで、不変条件が常に満たされることが示される。このような検証を可能とするところも、UML モデルを CafeOBJ によって記述する大きな利点であると言える。

本稿で示した例は、限定的なものではあるが、UML モデルを形式仕様言語により表現し、機械的な解析を行うための足がかりを示した。

5.2 関連研究

オブジェクト指向モデル (もしくは UML モデル) に対して形式仕様の概念を応用する研究がさまざまなレベルで行われている。Alloy[8] は、オブジェクトモデルに対して形式仕様言語 Z の概念を応用したモデリング言語である。Alloy はオブジェクトモデルに対する簡明な制約の記述により、オブジェクトの状態遷移を表現できる。Alloy で扱うオブジェクトモデルは、UML ではクラス図と OCL に相当する。また、Alloca[9] というツールによって分析することが可能である。

vUML[16] は、UML クラス図から構造に関する情報を、ステートチャート図から振舞

いに関する情報を得、仕様言語 PROMELA で表現し、ツール SPIN によるモデル検査を行うものである。UML モデルの実行という点では、Action Semantics[13] という動きがあり、UML モデルを実行させるべく UML の拡張を OMG に対して提案している。

OCL の解析に関する研究では、Dresden 大学において OCL コンパイラの開発が行われ、Java によるコード生成をすることで OCL 制約の動的な解析を行うアプローチがある [10]。また、彼らは OCL を SQL で表現し RDBMS によって解析を行うアプローチも提案している [11]。また、[14] では、OCL 制約によるシステムの状態の変化を視覚化するツールにより、OCL 制約のチェックを行う。[15] では、時相論理により形式化を行い、モデル検査法によって OCL を解析する試みが紹介されている。

Catalysis[24] は、これまでの多くの関連技術を取り込んだオブジェクト指向方法論であり、OCL の利用が随所で行われている。OCL の有効活用を方法論としてまとめていると捉えることもできる。

5.3 今後の課題

本研究の今後の課題としては以下のことが考えられる。

- より複雑な例の記述

  クラスや関連が複雑になった場合に、本研究で提案した対応関係が自然に適用できるか実証する必要がある。例えばクラスや関連の数が増えた場合にオブジェクトコンポジションを重ねていくのは仕様が重くなりがちである。また、不変条件の検証においても、より実際的な例で示すことで、さらなる CafeOBJ の有効活用を示す必要がある。

  また、より複雑な OCL 式に対応することで、より多くの例題に対応することができるようになると考えられる。特にコレクション型のオペレーションは本研究において未対応のものも多いため、それらに対応することでより実質的な OCL による仕様を対象とすることができるようになるであろう。

- UML の他のダイアグラムの考慮

  本研究では、対象とする UML がクラス図と OCL であった。クラス図から構造に関する情報、OCL から振舞に関する情報を抽出するアプローチを取ったが、UML ダイアグラムにおいて振舞モデルを表現するものに、ステートチャート図、シーケン

ス図およびユースケース図等がある。これらのダイアグラムと CafeOBJ による仕様との対応を考察することが課題としてあげられるであろう。

- CafeOBJ の検証技術のさらなる応用

  本研究で使用した振舞仕様では、余帰納法を用いた振舞等価性に関する検証 [4] が可能となっている。これらの検証技術を利用して、UML モデルの検証に応用することが課題としてあげられる。

- UML ツールとしての応用

  本研究で提案した枠組みは、UML ツールとしての応用が見込まれる。現在市販されている UML ツールは、UML モデルのシミュレーションや検証等が困難であった。UML のようなモデリング言語は、開発環境としてサポートすることが重要であり、UML モデルの解析の視覚化を含めてツール化することにより、本研究の枠組みが有効活用されると考えられる。

謝辞

本研究を進めるにあたり終始御指導頂いた二木厚吉教授に深く感謝致します。また、ゼミを通じて有益な助言をしてくださった渡部卓雄助教授、緒方和博先生、森彰先生、松本充広氏に感謝致します。特に様々な相談に応じて頂いた飯田周作氏に御礼を申し上げます。最後に公私ともどもお付き合いいただいた言語設計学講座の皆様に感謝致します。

参考文献

[1] Răzvan Diaconescu and Kokichi Futatsugi, CafeOBJ Report, World Scientific, 1998

[2] Kokichi Futatsugi, Fundamentals of algebraic modeling, Computer Software, 13(1):pp.4-22, 1996, in Japanese

[3] Joseph Gogune and José Meseguer, Order-sorted algebra : Equational deduction for multiple inheritance, overloading, exceptions and partial operations, Theoretical Computer Science,1992

[4] Joseph Goguen and Grant Malcolm, A hidden agenda, Technical Report CS97-538, UCSD Technical Report, 1997

[5] José Meseguer, Conditional rewriting logic as a unified model of concurrency, Theoretical Computer Science, 1992

[6] Shusaku Iida, An Algebraic Formal Method for Component based Software Developments, PhD thesis, Japan Advanced Institute of Science and Technology, 1999

[7] Shusaku Iida, Michihiro Matsumoto, Răzvan Diaconescu and Kokichi Futatsugi, Concurrent Object Composition in CafeOBJ, JAIST Research Report, IS-RR-98-0009S,1998

[8] D.Jackson, Alloy:A Lightweight Object Modeling Notaion, Technical Report 797, MIT Laboratory for Computer Science, Cambridge, Mass, 2000

[9] Daniel Jackson, Ian Schechter and Ilya Shlyakhter, Alcoa: the Alloy Constraint Analyzer Proc. International Conference on Software Engineering, Limerick, Ireland, 2000

[10] H.Hussmann, B.Demuth and F.Finger, Modular Architecture for a Toolset Supporting OCL, UML'2000 The Unified Modeling Language Conference, 2000

[11] H.Hussman and B.Demuth, Using OCL Constraints for Relational Database Design, UML'99 The Unified Modeling Language, Second Int. Conference Fort Collins, Co, USA, Springer, 1999

[12] OCL FAQ(draft), http://www.cs.ukc.ac.uk/research/sse/oclws2k/oclfaq.txt

[13] Stephen J. Mellor, Software-platform-independent,Precise Action Specifications for UML, UML'99 The Unified Modeling Language, Second Int. Conference Fort Collins, Co, USA, Springer, 1999

[14] M.Richters and M.Gogolla, Validating UML Models and OCL Constraints, UML'2000 The Unified Modeling Language Conference, 2000

[15] D.Distefano, J.Katoen and A.Rensink, Towards model checking OCL, ECOOP'2000 workshop on Defining Precise Semantics for UML, 2000

[16] VUML,http://www.abo.fi/ iporres/vUML/

[17] I.Jacobson, Object-Oriented Software Engineering, Addison-Wesley,1992

[18] J.Rumbaugh, M.Blaha, W.Premerlani, F. Eddy and W.Lorensen, Object-Oriented Modeling and Design, Prentice-Hall, 1991

[19] OMG, OMG Unified Modeling Language Specification version 1.3, http://www.rational.com/uml/index.html

[20] J.Warmer and A.Kleppe, The Object Constraint Language Precise Modeling with UML, Addison-Wesley, 1999

[21] G.Booch, Object Oriented Design with Applications, Benjamin Cummings, 1991

[22] G.Booch et al, The Unified Modeling Language User Guide, Addison-Wesley, 1998

[23] M.Fowler and K.Scott, UML Distilled: Applying the Standard Object Modeling Language, Addison-Wesley, 1997

[24] D'Souza, A.Wills, Objects, Componets, and Frameworks with UML - The Catalysis Approach, Addison-Wesley, 1999

[25] S.Cook and J.Daniels, Designing Object Systems -Object Oriented Modeling with Syntropy, Prentice-Hall, 1994

Appendix

```
-- ****************************
--  Library for OCL
-- ****************************

mod* OBJECTID{
  [ Oid ]
}


mod* COLLECTION(X :: TRIV){
  pr(INT)

  *[ Collection ]*

  -- initial state
  op empty : -> Collection

  -- behavioural constructors
  op add : Elt Collection -> Collection { coherent }

  -- atributes
  bop includes : Elt Collection -> Bool

  bop size : Collection -> Nat
```

```
  -- variables
  vars E E' : Elt
  vars S : Collection

  -- equations for attributes
  eq includes(E, empty) = false .
  eq includes(E, add(E', S)) = (E == E') or (includes(E, S)) .
  eq size(empty) = 0 .
  eq size(add(E, S)) = size(S) + 1 .

}

-- ****************************
--  example1 : Passenger
-- ****************************

mod* PASSENGER{
  protecting(NAT)

  *[ Passenger ]*

  -- observation
  bop age : Passenger -> Nat
  bop care? : Passenger -> Bool
  bop paid? : Passenger -> Bool
  -- action
  bop setAge : Nat Passenger -> Passenger
  bop pay : Passenger -> Passenger

  op init : -> Passenger

  var P : Passenger
```

```
  var N : Nat
  eq age(init) = 0 .
  eq age(setAge(N,P)) = N .
  eq age(pay(P)) = age(P) .
  eq care?(init) = false .
  ceq care?(setAge(N,P)) = true if N > 80 .
  ceq care?(setAge(N,P)) = false if not(N > 80) .
  eq care?(pay(P)) = care?(P) .
  eq paid?(init) = false .
  eq paid?(setAge(N,P)) = paid?(P) .
  eq paid?(pay(P)) = true .
}

-- ****************************
--  example2 : Order
-- ****************************

mod* CUSTOMER{
  protecting(INT)
  *[ Customer ]*
  bop pocket : Customer -> Int               -- observation
  bop setPocket : Int Customer -> Customer  -- action
  op init : -> Customer                      -- initial state

  var C : Customer
  var I : Int

  eq pocket(init) = 0 .
  eq pocket(setPocket(I,C)) = I + pocket(C) .

}
```

```
mod* EXPENSE-SHEET{
  protecting(INT)
  *[ Expense-sheet ]*
  bop price : Expense-sheet -> Int                      -- observation
  bop card? : Expense-sheet -> Bool                     -- observation
  bop setPrice : Int Expense-sheet -> Expense-sheet     -- action
  op init : -> Expense-sheet                            -- initial state

  var E : Expense-sheet
  var I : Int

  eq price(init) = 0 .
  eq price(setPrice(I,E)) = I .
  eq card?(E) = false .

}

mod* ORDER{
  protecting(CUSTOMER + EXPENSE-SHEET)
  *[ Order ]*
  bop order-price : Order -> Int              -- observation
  bop order-card? : Order -> Bool             -- observation
  bop order-pocket : Order -> Int             -- observation
  bop order-setPocket : Int Order -> Order    -- action
  bop order-setPrice : Int Order -> Order     -- action

  bop expense-sheet : Order -> Expense-sheet  -- projection
  bop customer : Order -> Customer            -- projection

  op init-order : -> Order                    -- initial state

  var O : Order
```

```
  var I : Int

  -- equations for observation
  eq order-price(O) = price(expense-sheet(O)) .
  eq order-card?(O) = card?(expense-sheet(O)) .
  eq order-pocket(O) = pocket(customer(O)) .

  -- equations for projection
  eq expense-sheet(init-order) = init .
  eq expense-sheet(order-setPrice(I,O)) = setPrice(I,expense-sheet(O)) .
  eq expense-sheet(order-setPocket(I,O)) = expense-sheet(O) .
  eq customer(init-order) = init .
  eq customer(order-setPrice(I,O)) = customer(O) .
  eq customer(order-setPocket(I,O)) = setPocket(I,customer(O)) .

  -- equations for invariant of expense-sheet
  ceq order-card?(O) = true if price(expense-sheet(O)) > pocket(customer(O)) .
  ceq order-card?(O) = false if price(expense-sheet(O)) <= pocket(customer(O)) .

}

-- ****************************
--  example3 : Hotel .v1
-- ****************************

mod! SEX{
  [ Sex ]

  ops Male Female : -> Sex .
}
```

```
mod* HOTEL{
  pr(INT + STRING + OBJECTID)
  *[ Hotel ]*
  bop numOfBed : Hotel -> Int          -- observation
  bop name : Hotel -> String           -- observation
  bop setNumOfBed : Int Hotel -> Hotel -- action
  bop setName : String Hotel -> Hotel  -- action
  bop checkIn : Oid Hotel -> Hotel     -- action
  op init : -> Hotel                   -- initial state

  var I : Int
  var S : String
  var H : Hotel

  eq numOfBed(init) = 0 .
  eq numOfBed(setNumOfBed(I, H)) = I .
  eq name(init) = "" .
  eq name(setName(S, H)) = S .

}


mod* GUEST{
  pr(INT + STRING + SEX + OBJECTID)
  *[ Guest ]*
  bop age : Guest -> Int           -- observation
  bop sex : Guest -> Sex           -- observation
  bop setAge : Int Guest -> Guest  -- action
  bop setSex : Sex Guest -> Guest  -- action
  op init : Oid -> Guest           -- initial state
  op no-guest : -> Guest           -- error
```

```
  -- equations
  var G : Guest
  var I : Int
  var S : Sex
  var OID : Oid

  eq age(init(OID)) = 0 .
  eq age(setAge(I, G)) = I .
  eq age(setSex(S, G)) = age(G) .
  eq sex(init(OID)) = Male .
  eq sex(setSex(S, G)) = S .
  eq sex(setAge(I, G)) = sex(G) .
}


mod* CHECKINS{
  pr(GUEST + HOTEL + COLLECTION(OBJECTID))
  *[ Checkins ]*
  bop numOfBed : Checkins -> Int        -- observation
  bop name : Checkins -> String         -- observation
  bop age : Oid Checkins -> Int         -- observation
  bop sex : Oid Checkins -> Sex         -- observation
  bop setNumOfBed : Int Checkins -> Checkins  -- action
  bop setName : String Checkins -> Checkins   -- action
  bop checkIn : Oid Checkins -> Checkins      -- action
  bop setAge : Oid Int Checkins -> Checkins   -- action
  bop setSex : Oid Sex Checkins -> Checkins   -- action
  bop hotel : Checkins -> Hotel               -- projection
  bop guest : Oid Checkins -> Guest           -- projection
  bop collection : Checkins -> Collection     -- projection
  op init-checkins : -> Checkins              -- initial state
  op errar-checkins : -> Checkins                -- errar state
```

```
-- meta operation
bop add : Oid Checkins -> Checkins
-- collection type operations
bop includes : Oid Checkins -> Bool
bop size_ : Checkins -> Int

vars O O' : Oid
var C : Checkins
var N : Int
var S : String
var SE : Sex

-- equations for meta actions and collection operator
eq collection(init-checkins) = empty .
eq collection(add(O, C)) = add(O, collection(C)) .
eq includes(O, C) = includes(O, collection(C)) .
eq size(C) = size(collection(C)) .
eq collection(setNumOfBed(N, C)) = collection(C) .
eq collection(setName(S, C)) = collection(C) .
eq collection(setAge(O, N, C)) = collection(C) .
eq collection(setSex(O, SE, C)) = collection(C) .

-- equations for projection operator
eq guest(O, init-checkins) = init(O) .
eq guest(O, add(O', C)) = guest(O, C) .
ceq guest(O, setAge(O', N, C)) = setAge(N, guest(O, C)) if O == O' .
ceq guest(O, setAge(O', N, C)) = guest(O, C) if O =/= O' .
ceq guest(O, setSex(O', SE, C)) = setSex(SE, guest(O, C)) if O == O' .
ceq guest(O, setSex(O', SE, C)) = guest(O, C) if O =/= O' .
eq guest(O, setNumOfBed(N, C)) = guest(O, C) .
eq guest(O, setName(S, C)) = guest(O, C) .
```

```
  eq age(O,C) = age(guest(O, C)) .
  eq sex(O,C) = sex(guest(O, C)) .

  eq hotel(init-checkins) = init .
  eq hotel(add(O, C)) = hotel(C) .
  eq hotel(setNumOfBed(N, C)) = setNumOfBed(N, hotel(C)) .
  eq hotel(setName(S, C)) = setName(S, hotel(C)) .
  eq hotel(setAge(O, N, C)) = hotel(C) .
  eq hotel(setSex(O, SE, C)) = hotel(C) .
  eq numOfBed(C) = numOfBed(hotel(C)) .
  eq name(C) = name(hotel(C)) .

  -- equations for pre/post condition of checkin method
  ceq checkIn(O, C) = add(O, C) if not(includes(O, C)) .
  ceq checkIn(O, C) = errar-checkins if includes(O, C) .

}

-- ***************************************************************
--  example3 : Hotel .v2
--    for the example of dynamick cheking and invariant verification
-- ***************************************************************
mod* HOTEL{
  pr(NAT + OBJECTID)
  *[ Hotel ]*
  bop numOfBed : Hotel -> Nat          -- observation
  bop checkIn : Oid Hotel -> Hotel     -- action
  op init : -> Hotel                   -- initial state

  var H : Hotel  var N : Nat  var O : Oid
  eq numOfBed(checkIn(O, H)) = numOfBed(H) .
```

```
}

mod* GUEST{
  pr(NAT + SEX + OBJECTID)
  *[ Guest ]*
  bop age : Guest -> Nat           -- observation
  bop sex : Guest -> Sex           -- observation
  op init : Oid Nat Sex -> Guest          -- initial state
  op no-guest : -> Guest           -- error

  -- equations
  var G : Guest  var N : Nat  var S : Sex  var OID : Oid
  eq age(init(OID, N, S)) = N .
  eq sex(init(OID, N, S)) = S .
}

mod* SYSTEM{
  pr(GUEST + HOTEL + COLLECTION(OBJECTID))
  *[ System ]*
  bop numOfBed : System -> Nat        -- observation
  bop age : Oid System -> Nat         -- observation
  bop sex : Oid System -> Sex         -- observation
  bop checkIn : Oid Nat Sex System -> System      -- action
  op hotel : System -> Hotel               -- projection
  op guest : Oid System -> Guest           -- projection
  op collection : System -> Collection     -- projection
  op init-system : -> System              -- initial state
  -- collection type operations
  bop includes : Oid System -> Bool
  bop size_ : System -> Nat

  vars O O' : Oid  var C : System  var N : Nat  var S : Sex
```

```
  -- observation
  eq includes(O, C) = includes(O, collection(C)) .
  eq size(C) = size(collection(C)) .
  eq age(O,C) = age(guest(O, C)) .
  eq sex(O,C) = sex(guest(O, C)) .
  eq numOfBed(C) = numOfBed(hotel(C)) .
  -- equations for meta actions and collection operator
  eq collection(init-system) = empty .
  ceq collection(checkIn(O, N, S, C)) = add(O, collection(C))
                                if not(includes(O,C)) and (numOfBed(C) > size(C)) .
  ceq collection(checkIn(O, N, S, C)) = collection(C)
                                if not(includes(O,C)) and (numOfBed(C) == size(C)) .
  ceq collection(checkIn(O, N, S, C)) = collection(C)
                                if not(includes(O,C)) and (numOfBed(C) < size(C)) .
  ceq collection(checkIn(O, N, S, C)) = collection(C)
                                if includes(O,C) .
  -- equations for projection operator
  eq guest(O, init-system) = no-guest .
  ceq guest(O, checkIn(O', N, S, C)) = init(O, N, S) if O == O' .
  ceq guest(O, checkIn(O', N, S, C)) = guest(O, C) if O =/= O' .

  eq hotel(init-system) = init .
  eq hotel(checkIn(O, N, S, C)) = checkIn(O, hotel(C)) .
}

-- ********************************************************
-- proof of the following OCL invariant of example3(hotel)
--    context hotel : self.numOfBed >= self.guests->size
--    in CafeOBJ : numOfBed(S) >= size(S) . ( S : System )
-- ********************************************************

-- opening module SYSTEM
```

```
open SYSTEM

-- declaring constraints for induction
op o : -> Oid .
op n : -> Nat .
op s : -> Sex .
ops sys sys' sys'' : -> System .

equation for the arbitary number of beds of the hotel
var n' : Nat .
eq numOfBed(init) = n' .

**> prove numOfBed(s) >= size(s)
**> by induction on s

**> base case ( s=init-system ) : numOfBed(init-system) >= size(init-system)
red numOfBed(init-system) >= size(init-system) .
-- should be true

**> induction hypothesis ( s = sys ) : numOfBed(sys) >= size(sys)
--   in CafeOBJ => numOfBed(hotel(sys)) >= size(collection(hotel(sys)))
eq numOfBed(hotel(sys)) >= size(collection(sys)) = true .
**> induction step ( s = checkIn(o, n, s, sys) )

  -- equations for inequality
  vars N M : Nat
  ceq N >= (M + 1) = true if N > M .
  ceq N >= M = true if N == M .

  **> case analysis for checkIn operation
  **> case : include(o,sys) = true
  eq includes(o, sys) = true .
```

```
  red numOfBed(checkIn(o, n, s, sys)) >= size(checkIn(o, n, s, sys)) .
  -- should be true

  **> case : include(o,sys) = false
  eq includes(o, sys') = false .
  -- the division I.H. into I.H.(1) and I.H.(2)
  -- I.H.(1)
  eq numOfBed(hotel(sys')) > size(collection(sys')) = true .
  red numOfBed(checkIn(o, n, s, sys')) >= size(checkIn(o, n, s, sys')) .
  -- should be true
  eq includes(o, sys'') = false .
  -- I.H.(2)
  eq numOfBed(hotel(sys'')) = size(collection(sys'')) .
  red numOfBed(checkIn(o, n, s, sys'')) >= size(checkIn(o, n, s, sys'')) .
  -- should be true

**> QED for OCL invariant
close
```