ソフトウェア工学 129－15
（2000．11．17）

コンポーネントベース・フレームワーク技術を利用したアプリケーションの形式的仕様記述

吉田和樹[†]　本位田真一[‡]
†（株）東芝 SI 技術開発センター
‡ 国立情報学研究所

本論文では、形式的仕様記述を前提にしたコンポーネントベース・フレームワーク技術を提案し、それを利用してアプリケーションプログラムを作成した場合の、アプリケーション全体に対する形式的仕様記述と、それに基づく機能的な検証について述べる。アプリケーションプログラムの例としては、トランザクション処理を対象にして、その中でも特に、オブジェクトモデルからリレーショナルモデルへのマッピング処理について取り上げ、これを業務トランザクション処理用のコンポーネントセット[1][2]を利用して作成することで、本論文におけるアプローチの有効性を示す。

A formal specification of application programs using component-based framework technology

Kazuki Yoshida[†]　Shinichi Honiden[‡]
† System Integration Technology Center, Toshiba Corporation
‡ National Institute of Informatics

This paper proposes a component based framework technology based on formal specification and explains the formal specification and verification of the application programs using it. A transaction processing, especially a mapping program from object oriented model to relational model which provides a general framework for making objects persistent in RDBMS, is taken up as an application example for this purpose.

1. はじめに

1.1 コンポーネントベース・フレームワーク技術

近年、多くの企業でアプリケーションフレームワークの必要性に対する認識が高まってきているが、その開発には、

- ドメインに対する深い知識や洞察力
- 汎用性を高めるための設計・実装の繰り返し

が必要とされるため、これらの点が開発を阻む要因になっているというのが実状である。そこで、[2]では、アプリケーションフレームワークを、予め用意されたコンポーネントの組み合わせで構築できるようにするコンポーネントベース・フレームワーク技術(CBF)を提案した。そこでは、予め用意されたコンポーネントの組み合わせによりフレームワークを構築することで、設計・実装作業の繰り返しから開発者を解放し、また、作業そのものを効率化することで、開発を阻んでいる上記の要因を克服することがねらいにある。CBFでは、コンポーネントの組み合わせ方を変えることにより、各種ドメインでフレームワークを構築可能な、ドメインに横断的なコンポーネントセットを開発していく[1]。

ところで、コンポーネントの組み合わせでフレームワークを構築する場合、コンポーネントの側でフレームワークのホットスポット[9]を問題なく扱えるようにする仕組みを考える必要がある。そのために、コンポーネントの抽出・設計に対しては次に挙げるような2つの限定が課されることになる[2]。

[限定1]
コンポーネントは、ルートを1つ持つDAG状の構成で組み合わされ、これらの間でメッセージはアークの向きに従って連鎖的に流れるものとする。

[限定2]
コンポーネントが提供する処理の中で、ホットスポットに相当する内容が存在する場合には、その部分を委譲の考え方に基づいて、別クラスに分けて実装する。このクラスを以後カスタマイズクラスと呼ぶことにする。

限定1により、ノードを増やしたり減らしたり、あるいは、入れ換えたりすることで、理解容易な形で、ホットスポット部分の処理のシーケンスのバリエーションを実現できるようにする。また、限定2のカスタマイズクラスは、ホットスポット部分の処理を実行するためのインタフェースを定義した抽象クラスとして提供する。利用者は、その抽象クラスを継承するクラスを作成して、その中でこのインタフェースに対して必要な処理を実装して、これをコンポーネントにプラグインして使うことになる[2]。

1.2 CBFと形式的仕様記述

CBFに関係する開発者には3種類ある。すなわち、先述のCBFの抽出・設計上の限定に基づいてコンポーネントを開発するコンポーネント開発者と、そのコンポーネントを組み合わせてフレームワークを開発するフレームワーク開発者、そして、そのフレームワークにカスタマイズクラスや周辺モジュールを組み合わせて最終的なアプリケーション

[1] ただし、ここで言う"ドメインに横断的な"というのは、事務処理システムという限定された枠の範囲内でのことである。

[2] フレームワーク内でホットスポットになり得る処理の対象となるデータの構成/型の違いなども、データの生成処理を行うコンポーネントの中でカスタマイズクラスに吸収させるようにする。

に仕上げていくアプリケーション開発者である。

ここで、コンポーネント開発者の観点からCBFを捉えてみる。コンポーネントの開発では、設計の規範として、パターンを適用することが開発上の指針になる。このパターンの適用により、コンポーネントを構成するクラスのインタフェースが明確になり、構造が良く見通せるようになる利点が得られるが、インタフェースの実装についての機能的な側面まではパターンから理解することはできない。したがって、コンポーネントの機能は別途しっかりと仕様化しておくことが必要になる。それにより、コンポーネント開発者どうしでの理解を促進し、また、提供する機能の正しさを実装前に十分に確認することもできるようになる。

一方、フレームワーク開発者の観点からCBFを捉えてみる。フレームワーク開発者は、適切なコンポーネントを選択して、それらの組み合わせによりフレームワークを構築するが、その際に、必要としている機能を提供するコンポーネントを選択できているのか、あるいは、意図するフレームワークが矛盾なく構築できているのかといったことが疑問として生じることになる。これらの疑問を解決するためには、コンポーネントの機能がしっかりと仕様化されていることが必要であり、さらに、コンポーネントの組み合わせ時の機能仕様を、個々のコンポーネントの機能仕様から容易に合成できるような記述上の仕組みと、無矛盾性を検証するための仕組みが必要になる。

アプリケーション開発者の観点からCBFを捉えると、コンポーネントどうしの組み合わせのときと同様に、それにより生成されるフレームワークと周辺モジュールやカスタマイズクラスとの組み合わせにおいて機能の無矛盾性を検証する仕組みが必要になる。

ところで、コンポーネント開発者、フレームワーク開発者、アプリケーション開発者の3者にとって必要になる、機能仕様の記述ということを課題とした場合、本研究では、これに取り組むために次のような選択を行った。すなわち、仕様に曖昧性のない厳密な記述を求める意味で、形式的仕様記述言語に着目し、その中でも、対象の機能的側面を記述するのに適しており、また、模擬的に実行可能で機能的な無矛盾性を検証可能な代数的仕様を選択した。

1.3 代数的仕様の拡張

[1]には、1.1節で述べたCBFの限定に基づいて、フレームワークの機能的な側面を記述するために代数的仕様に課せられる要件がまとめられている。これらの要件は、代数的仕様の理論的拡張を考えることで対処できるものと、記述上の工夫を考えることで対処できるものに分けることが可能である。

理論的拡張については、"Δ"という特別な定数記号を導入して、カスタマイズクラスでの未定義や、例外動作の記述の省略を表現できるようにする点と、含意記号"*imply*"を導入して、帰結部の等式が成立する場合の条件を、前提部に等式の集合を使って記述できるようにする点が挙げられる。

また、記述上の工夫については、項書き換えの処理系に対して、次に挙げるような拡張を考える必要がある。

- オブジェクト指向における継承関係を扱えること
- 項中の変数を、処理系内でも変数として扱われるように宣言できるようにして[3]、内部には項への参照を保持して、複数の項書き換えの系列から参照可能になるようにすること
- 処理系内で変数をグローバルに宣言して、任意の項書き換えでこの変数を参照できるようにすること
- 処理系内の変数に対して、ある項書き換えと別の項書き換えの間では、操作を同期化して扱うこと
- 必要ならば、処理系内の変数の値を直接書きかえるのではなく、コピーして利用することを、等式中に宣言できるようにすること

項書き換えにおいて、処理系内に変数を導入することは、コンポーネントの DAG 状の組み合わせによる処理の連鎖を検証時に項書き換えで正確にシミュレートするためには必要であり、代数的仕様の意味論として一般的な始代数意味論とは異なる操作的意味論が背後に形成されることになる[4]。

本論文では、以後の章において、本節で述べた拡張を前提にして、議論を進めていく。

2 アプリケーションの代数的仕様記述

2.1 対象とするアプリケーションについて

本論文では、仕様記述の対象とするアプリケーションの例として、データベースに対するトランザクション処理を扱うプログラムについて考える。トランザクション処理は、データの管理を主要目的とするビジネス系情報システムの開発においては、重要な役割を果たすものであり、ここで仕様記述の可能性を示すことができれば、それは実用的に大きな意義があると考える。

ところで、ここでアプリケーションの処理内容としては、オブジェクト指向モデルからリレーショナルモデルへのマッピング処理を考えることにする。このような処理を取り上げる理由としては、昨今のアプリケーションシステムの開発では、オブジェクト指向言語によるプログラミングが普及しつつある一方で、データベースには実績のあるリレーショナルデータベースが使用されるケースが多く、この2つのモデル間でのミスマッチをどのように解消するかが1つの技術的な問題になっていることが挙げられる。この問題に対して、Kyle Brownは、パターン" Crossing Chasms"[3]の中で、典型的なオブジェクト間の関係を次に挙げるように類型化して[5]、それぞれの場合に、リレーショナルデータベースのスキーマ設計をどのように行なえばよいのかを示している。

(R1)異なるクラスのオブジェクト間で1対1の関係
(R2)同一クラスのオブジェクト間で1対多の関係
(R3)異なるクラスのオブジェクト間で1対多の関係
(R4)異なるクラスのオブジェクト間で多対多の関係
(R5)異なるクラスのオブジェクト間で1対多の関係
(特に、多側に、ポリモルフィックな関係にある複数種類のクラスのオブジェクトが入るような場合)

例えば、(R1)の場合と(R5)の場合には、それぞれ次のようなスキー

[3] 本論文内では、イタリック体で記述する。

[4] このことは、逆に言うと、コンポーネントの代数的仕様を項書き換えによる実行を念頭において記述することを求めている。したがって、項書き換えの停止性や合流性を保証する形で代数的仕様を記述することも前提になる。

[5] オブジェクト間の関係だけでなく、クラス間の継承関係をどのように扱えばよいのかも示されている。

マ設計を行なうことが示されている。

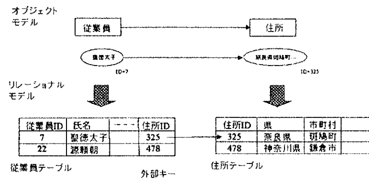


図1. 異なるクラスのオブジェクト間で1対1の関係(R1)

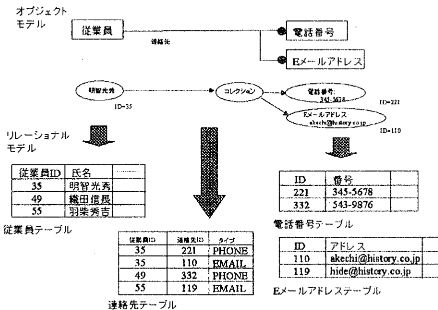


図 2.異なるクラスのオブジェクト間で1対多の関係(R5)

以後では、このようなオブジェクト間の関係の類型化とそれぞれに対するデータベースのスキーマ設計に基づいて、オブジェクト指向モデルからリレーショナルモデルへのマッピング処理を、CBFの1つである業務トランザクション処理用のコンポーネントセット[1][2]を使って、オブジェクト指向モデルの任意のクラス構造に適用可能な汎用的なフレームワークとして作成することを考える。そして、そのときの仕様記述と検証について説明していく。

仕様記述にあたっては、処理全体を次のような構成に分けて考えることにする。

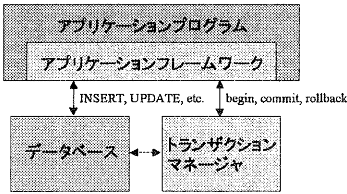


図 3. 処理の構成

2.2 データベースでの処理の記述

データベースでの処理を記述するにあたっては、次のように3つの hashtable から成る型を database として宣言する。

[hashtable, hashtable, hashtable]

これは、データベース内の状態を表現することを目的とした型であり、1つ目の hashtable では、テーブル名をキーにして、テーブルに記録されているレコードの集合を vector で表現して、これを値に持たせるようにしている。また、2つ目の hashtable では、テーブル名をキーにして、そのテーブルの主キー名を値に持たせるようにしている。3つ目の hashtable では、テーブル名をキーにして、そのテーブルの外部キー名を参照先のテーブル名と組にしたものを hashtable で表現して、これを値に持たせるようにしている。以後、本論文内では、$[\ h_{rec}, h_{pk}, h_{fk}\ ]$により、この database 型の任意の項を表現する。

また、データベース内で処理の例外が発生したことを表現するために、database 型の定数項 Exception を宣言しておく。

このような新たな型の導入を前提にして、データベースへのレコードの挿入処理を、主キー、および、外部キーに関する制約とともに記述すると次のようになる。

$$\begin{aligned}
&\mathrm{unique}(\ \mathrm{get}(\ h_{rec}, tn\ ), \mathrm{get}(\ h_{pk}, tn\ ), h_{in}\ ) = \mathrm{true},\\
&\mathrm{refer}(\ h_{rec}, h_{pk}, \mathrm{get}(\ h_{fk}, tn\ ), h_{in}\ ) = \mathrm{true}\quad \textit{imply}\\
&\qquad \mathrm{insert}(\ h_{in}, [\ h_{rec}, h_{pk}, h_{fk}\ ], tn\ ) =\\
&\qquad\qquad [\ \mathrm{addElement}(\ \mathrm{get}(\ h_{rec}, tn\ ), h_{in}\ ), h_{pk}, h_{fk}\ ];\\
&\mathrm{unique}(\ \mathrm{get}(\ h_{rec}, tn\ ), \mathrm{get}(\ h_{pk}, tn\ ), h_{in}\ ) = \mathrm{true},\\
&\mathrm{refer}(\ h_{rec}, h_{pk}, \mathrm{get}(\ h_{fk}, tn\ ), h_{in}\ ) = \mathrm{false}\quad \textit{imply}\\
&\qquad\qquad \mathrm{insert}(\ h_{in}, [\ h_{rec}, h_{pk}, h_{fk}\ ], tn\ ) = \mathrm{Exception};\\
&\mathrm{unique}(\ \mathrm{get}(\ h_{rec}, tn\ ), \mathrm{get}(\ h_{pk}, tn\ ), h_{in}\ ) = \mathrm{false}\quad \textit{imply}\\
&\qquad\qquad \mathrm{insert}(\ h_{in}, [\ h_{rec}, h_{pk}, h_{fk}\ ], tn\ ) = \mathrm{Exception};
\end{aligned}$$

すなわち、テーブルへのレコードの挿入を行なう際に、次の 2 点の制約が満たされていることを確認する。

- その主キーと同じ値を持つレコードがテーブル内に存在しないこと (unique により確認[6])
- レコード内に外部キーが存在する場合には、そのキーを持つもとのレコードが既に別テーブルに挿入されていること (refer により確認[7])

もしこの 2 点のいづれかが満たされなければ、挿入は行われず、例外が返されることになる。

項 $[\ h_{rec}, h_{pk}, h_{fk}\ ]$ はデータベースの状態を表す項として、処理系内部で、グローバルな変数 db_{state} に格納されて、任意の書き換え規則からこれを参照できるようにするが、内容の整合性を保つために、1つの書き換え規則がこれを使用している間は、他の書き換え規則はこれを使用できないようにする仕組みが必要である[8]。

2.3 トランザクションマネージャでの処理の記述

本節では、トランザクション処理の ACID 性の保証という問題を、代数的仕様を使ってどのように記述するのかを説明する。

まず、Atomicity について記述するために、ここでは最も一般的なフラットトランザクションを対象にして考える。データベースに対する一連の処理をトランザクションとして扱うことを項 $tx(\ db, p, h_{proc}\ )$ で表現する。ここで、db にはデータベースの状態を表す database 型の項が、また、p には一連の入力とそれに対する出力、および、戻り値を組にした process 型[9]の項が、また、h_{proc} には入力に応じて

[6] unique の仕様は付録を参照

[7] refer の仕様は付録を参照

[8] 含意記号を使って記述された仕様については、前提部の等式の書き換え時には、変数内に格納されている h_{rec}, h_{pk}, h_{fk} の内容のコピーが使用される。

[9] process 型は、[1]において、入力データを表す hashtable と、出力データを表す vector と、戻り値を表す object の3つ組[hashtable, vector, object]として定義されたが、この論文内では、前節で

呼び出される手続きの処理を表現した項がその手続き名と組にされて hashtable の形で渡される。これにより、以下の記述にあるように、proc で一連の入力を分解して、call[10]で各入力についての処理をその時点でのデータベースの内容のコピーとともに呼び出し、result によりその処理の成否が返されると、prepare で一連の処理が正常に終了していることを確認した後、そのときの出力結果を、db_{state} に代入する。もしも、例外が発生した場合には、db_{state} 内の値は元のまま残す。

$$\mathrm{tx}(\mathrm{db}, \mathrm{p}, h_{proc}) = \mathrm{prepare}(\mathrm{proc}(\mathrm{db}, \mathrm{p}, h_{proc}));$$

$$\mathrm{proc}([h_{rec}, h_{pk}, h_{fk}], [h_{in}, \mathrm{db}_{out}, o_{ret}]+\mathrm{p1}, h_{proc}) =$$
$$\quad \mathrm{result}(\mathrm{call}([h_{in}, \mathrm{copy}(\mathrm{db}_{out}, [h_{rec}, h_{pk}, h_{fk}])^{11}, o_{ret}],$$
$$\qquad \mathrm{get}(h_{proc}, \mathrm{getProcId}(h_{in}))), \mathrm{db}_{out}, o_{ret}) + \mathrm{proc}(\mathrm{db}_{out}, \mathrm{p1}, h_{proc});$$

$$\mathrm{result}(\mathrm{p}, \mathrm{db}_{out}, \mathrm{null}) = \mathrm{db}_{out};$$

$$\mathrm{result}(\mathrm{p}, \mathrm{db}_{out}, \mathrm{Exception}) = \mathrm{Exception};$$

$$\mathrm{prepare}([h_{rec}, h_{pk}, h_{fk}]) =$$
$$\qquad \mathrm{commit}(\mathrm{substitute}(db_{state}, [h_{rec}, h_{pk}, h_{fk}])^{12});$$

$$\mathrm{prepare}(\mathrm{Exception}) = \mathrm{rollback};$$

$$\mathrm{commit}(\mathrm{db}) = \mathrm{db};$$

$$\mathrm{rollback} = db_{state};$$

$$[h_{rec1}, h_{pk1}, h_{fk1}] + [h_{rec2}, h_{pk2}, h_{fk2}] = [h_{rec2}, h_{pk2}, h_{fk2}];$$

$$[h_{rec1}, h_{pk1}, h_{fk1}] + \mathrm{Exception} = \mathrm{Exception};$$

$$\mathrm{Exception} + [h_{rec2}, h_{pk2}, h_{fk2}] = \mathrm{Exception};$$

Consistency については、前節で、データベースへのレコードの挿入処理を仕様化する際に、外部キーに関する制約を仕様の前提部に記述したが、このような形で、データベースの処理の中に含めて記述する。

Isolation については、代数的仕様が並行処理制御の詳細な記述には適していないため、Isolation レベルとして最も厳しい serializable の場合だけを考えて、次のように記述する。すなわち、グローバルな変数 db_{access} を設けて、ここに初期状態の db_{state} を代入する。

$$\mathrm{substitute}(db_{access}, db_{state});$$

そして、以後、データベースに対するトランザクション処理を行ないたいときには、次に示すように、この db_{access} と前述の tx を使って、

$$\mathrm{tx}(\mathrm{substitute}(db, db_{access}), \mathrm{p}, h_{proc})$$

のような項を生成し、この項を

$$\mathrm{substitute}(db_{access}, \mathrm{tx}(\mathrm{substitute}(db, db_{access}), \mathrm{p}, h_{proc}))$$

のように再び db_{access} の中に代入しておくことにより、変数 db_{access} の中に、待ち状態にあるトランザクション処理 tx から再帰的に構成される項を入れるようにしておく。この再帰的な項の書き換えは、常に最内戦略で行い、一番内側の tx 項の書き換え結果である database 型の項を次の tx に対する引数として与えるようにして、複数のトランザクションがシリアライズされて逐次に処理が進む様子を表現する。

Durability については、データベースによりトランザクションデータの永続性を確保することを考えるため、前節で取り上げたデータベースでの処理の記述が、この性質の記述に相当する。

2.4 フレームワークでの処理の記述

前節で述べたマッピング処理について、[1][2]でコンポーネントベース・フレームワークの例として取り上げた、業務トランザクション処理のコンポーネントセット(トランザクションタイプ、保管、分類、分割、変換)を使用して処理を実装することを考える[13]。

例えば、前節の(R1)の場合の処理は、次のようなコンポーネントの組み合わせで実装することができる。

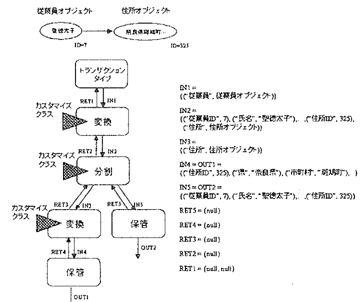


図 4. コンポーネントの組み合わせ --- (R1)の場合

すなわち、住所オブジェクトと1対1の関係を持つ従業員オブジェクトは、トランザクションタイプで予め設定されたデータ項目として Hashtableに格納され、変換に渡される。変換では、カスタマイズクラスの中で、従業員オブジェクトの属性について値を取得し、その値を属性名と組にしてHashtableに格納する。また、値が住所オブジェクトに対する参照の場合には、住所オブジェクトの識別子も取得して、これも適切なキー名と組にしてHashtableに格納する。そして、これを分割に渡す。分割では、カスタマイズクラスの中で、Hashtable内のキーと値の組を、住所オブジェクトの組と、それ以外のキーと値の組に分割して、前者を変換の側へ、また、後者を保管の側へそれぞれ渡す。変換では、上述の従業員オブジェクトの場合と同様に、カスタマイズクラスの中で、住所オブジェクトの属性について値を取得し、その値を属性名と組にしてHashtableに格納し、保管に渡す。最後にHashtableを渡される保管では、いずれの場合も、そのキーと値の組をデータベース内の従業員テーブルと住所テーブルにそれぞれ格納する処理を行なう。

その他の(R2)～(R5)の場合についても、これと同様にしてコンポーネントの組み合わせで処理を実装することが可能である。

ところで、[1]で説明している各コンポーネントの代数的仕様とその組み合わせによる処理の記述方法をもとにすれば[14]、マッピング処理のための組み合わせもまた項として表現することが可能である。例えば、(R1)の場合は次のようになる。

---

database 型を新たに定義したことにより、vector を database に置き換えて、[hashtable, database, object]のように再定義する。

[10] call の仕様記述については[1]を参照。

[11] copy は変数に対するコピー操作を行なう演算子。

[12] substitute は、変数に対する代入操作を行なう演算子。

[13] この問題を考えるにあたり、オブジェクトには、クラス内で一意に識別できるような識別子が必ず付与されているものとする。

[14] 各コンポーネントの仕様については、このマッピング処理への適用に際して、[1]で挙げたものに変更を加える必要がある。これらの変更点を付録に載せる。

$invoke_{split}(invoke_{transform}(invoke_{ttype}(h_{in}, v_{item}, h_{type}, db_{out}, o_{ret})),$
h_{map}, put(put(h_{empty},
pair(“住所オブジェクトへの処理”,
$invoke_{store}(invoke_{transform}([h_{in1}, db_{out1}, o_{ret1}])$, “住所”))),
pair(“従業員データへの処理”,
$invoke_{store}([h_{in2}, db_{out2}, o_{ret2}]$, “従業員”))));

h_{map} = put(put(h_{empty},
pair(“従業員データへの処理”,
addElement(…addElement(addElement(v_{empty},
“従業員 ID”), “氏名”),…“住所 ID”))),
pair(“住所オブジェクトへの処理”,
addElement(v_{empty}, “住所”)));

その際、カスタマイズクラスでの処理の仕様は、マッピングの対象となるクラス構造に応じて、記述することになる。例えば、図 4 に挙げている例で、一番上の変換コンポーネントに挿されているカスタマイズクラスでの処理は、次のように仕様化される。

$Customize_{transform1}(h_{in})$ = makeHash (h_{empty}, get(h_{in}, “従業員”));
makeHash(h, oneToOne(o_1, o_2, ”従業員”, “住所”, “カスタマイズ1”)) =
putPairs(put(put(h, pair(“住所”, o_2)),
pair(“住所ID”, getID(o_2))), o_1);
putPairs(h, set(o, pair(o_{key}, o_{val}))) =
putPairs(put(h, pair(o_{key}, o_{val})), o);
putPairs(h, o_{empty}) = h;

マッピング処理の仕様の中で、現在扱っている従業員オブジェクトと住所オブジェクトの関係に特化した部分は、すべてこのようにカスタマイズクラスの中で記述する。したがって、対象が従業員と住所以外の関係に変わった場合にも、このカスタマイズクラスの仕様を書き直すだけで、対処することが可能である[15]。一方、[1]の中では、カスタマイズクラスの仕様は、Δを使って未定義として宣言されているが、もしも、その状態で項書き換えを実行すれば、全体の結果もΔになり、これはまさに処理を行なえる状態になっていないということを表している。

3 アプリケーション処理の無矛盾性の検証

3.1 複合オブジェクトを対象にした処理の記述

2.1 節で挙げた5種類の関係から構成される任意の複合オブジェクトについて、2.4 節に示したような業務トランザクション処理のコンポーネントセットの組み合わせにより、データベースへのマッピング処理が問題なく行なえることを検証する。そのために、このマッピング処理を、次のような項で表現し、前章で説明したデータベース処理、トランザクション処理、コンポーネント処理の仕様の組み合わせに基づく項書き換えシステムで、これの書き換えを行い、意図した結果が得られるかどうかを確認する。

map(db_{access}, ro)

ここで、演算子 map のシグニチャは次の通りである。

map : database, rel_object → database

rel_object は、複合オブジェクトを表す型である。具体的には、object 型の項を基底項として[16]、2.1 節の5種類の関係を表現した演算子(説明は後述)を再帰的に施すことにより生成される項の集合である。

したがって、上記の項は、5種類の関係で結ばれた任意の複合オブジェクト ro に対して、データベース db_{acces} へのマッピング処理 map を施すことを表している。

ところで、rel_object の項を構成する5種類の関係を表現する演算子を、それぞれ次のように定義する。

(R1)の場合
oneToOne : rel_object(関係基オブジェクト),
rel_object(関係先オブジェクト),
string(関係基オブジェクトに付与するキー),
string(関係先オブジェクトの参照名),
string(カスタマイズ処理に付与するキー)
→ rel_object(複合オブジェクト)
(R2)の場合 oneToMany1 (以後、シグニチャは省略)
(R3)の場合 oneToMany2
(R4)の場合 manyToMany
(R5)の場合 oneToMany3

これを基にして、map の仕様を以下のように記述する。

map(db_{access}, ro) =
substitut(db_{access}, tx(substitute(db, db_{access}) makeProcess(ro, db_{out}, o_{ret}),
h_{proc}));

すなわち、makeProcess により ro の内部を図 5 のように1つ1つの関係に分解して、2.3 節で定義した演算子 tx を使い、1つのトランザクションの内部でそれらを連続的に処理していくように記述する。

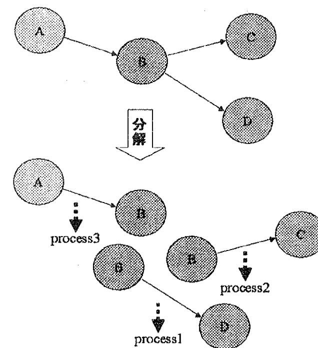


図 5. 複合オブジェクトの分解

[15] 仕様には反映させていないが、このマッピング処理においては、各コンポーネントのカスタマイズクラスでの処理を、いくつかのタイプ[6]に分けて実装することも可能である。それにより、関係に特化した部分をパラメータとして与えるだけで済ませることができるようになる。

[16] object 型の項は、オブジェクト間の関係を一切内部には保持しない。

このとき、makeProcess 以下の仕様は、任意の複合オブジェクトが次の 2 種類の方法で構成されていることに基づいて、記述される。

① 1つのオブジェクトが複数種類の関係を持つ

② 1つのオブジェクトの関係先になるオブジェクトが再び他の関係を持つ

ただし、紙面の都合上、ここでは上述の関係が、特に、先に挙げた oneToOne のみで構成されている場合の仕様を示す。仕様中に現れる atom、$atom_1$、$atom_2$ は、rel_object 型の基底項になる object 型の項のための変数である。(以後のすべての仕様について同様)

makeProcess(oneToOne(ro_1, ro_2, n_1, n_2, n_3), db_{out}, o_{ret}) =
makeProcess(ro_1, db_{out}, o_{ret}) + makeProcess(ro_2, db_{out}, o_{ret}) +
[toHash(oneToOne(ro_1, ro_2, n_1, n_2, n_3)), db_{out}, o_{ret}];

makeProcess(atom, db_{out}, o_{ret}) = p_{empty};

<u>基底構成のための仕様記述</u>

toHash(oneToOne($atom_1$, $atom_2$, n_1, n_2, n_3)) =
put(put(put(h_{empty}, pair(n_1, oneToOne($atom_1$, $atom_2$, n_1, n_2, n_3))),
pair("procId", "R1")), pair("customizeProcName", n_3));

<u>①の構成に基づく仕様記述</u>

toHash(oneToOne(oneToOne(ro_1, ro_2, n_1, n_2, n_3), atom, n_1, n_4, n_5)) =
toHash(oneToOne(ro_1, atom, n_1, n_4, n_5));

<u>②の構成に基づく仕様記述</u>

toHash(oneToOne(atom, oneToOne(ro_1, ro_2, n_1, n_2, n_3), n_4, n_1, n_5)) =
toHash(oneToOne(atom, ro_1, n_4, n_1, n_5));

<u>①、②の両構成に基づく仕様記述</u>

toHash(oneToOne(oneToOne(ro_1, ro_2, n_1, n_2, n_3), oneToOne(ro_3, ro_4, n_4, n_5, n_6), n_1, n_4, n_7)) = toHash(oneToOne(ro_1, ro_3, n_1, n_4, n_7));

すべての関係の場合に、toHash により生成される hashtable の中には、その関係の種類を明示する識別子が pair("procId", "R1") の形で追加される。

ところで、tx 内の h_{proc} には、(R1)～(R5)の各関係について、2.4 節で挙げた処理のためのコンポーネントの組み合わせを項の形で表現したものを、それぞれ"R1"～"R5"などの識別子と組にして格納した hashtable が渡される。これにより、tx 内に、toHash により生成された hashtable が入力されると、その中から "procId"をキーにして識別子を get し、これをもとに、h_{proc} の中から適切な項を選択して、call によりマッピング処理を表現する項書き換えを開始することになる。

また、各ベースコンポーネントで、カスタマイズクラスでの処理を表す項を、キーと対応付けて管理させるようにする。この中でどのキーを選択するのかは、入力データの中に "customizeProcName" と対応付けて設定されている値をもとに、決められることになる。例えば、先にも取り挙げた図4の一番上の変換コンポーネントに挿されているカスタマイズクラスでの処理は、次のように仕様化される。

$Customize_{transform1}$(h_{in}) = get($h_{transform1}$, get(h_{in}, "customizeProcName"));

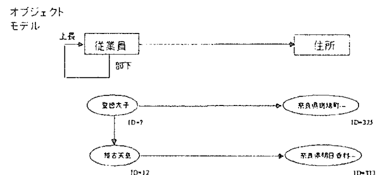
右辺を書き換えることにより、$h_{transform}$内に設定されている

makeHash(put(h_{empty}, pair("customizeProcName", get(h_{in}, "customizeProcName"))), get(h_{in}, "従業員"))

が得られることになる。このような記述上の工夫から、仕様レベルでコンポーネントの組み合わせの部分を共通化した上で、対象とする入力データ (複合オブジェクト) の種類に応じてカスタマイズクラスでの処理の仕様を切り替えることにより、検証が容易に実現できるようになる。

3.2 検証例

例えば、図 6 に示すような 1 対 1 の関係を3組持つ複合オブジェクトを次のように項で表現する。



<u>図 6. 1 対1の関係を 3 組持つ複合オブジェクト</u>

ro =
oneToOne(oneToOne(set(…set(set(o_{empty},
pair("従業員ID", 7)), pair("氏名", "聖徳太子")),…
pair("役職", "摂政")),
set(set(set(o_{empty},
pair("住所ID", 325)), pair("県", "奈良県")),
pair("市町村", "斑鳩町")),
"従業員", "住所", "カスタマイズR1"),
oneToOne(set(…set(set(o_{empty},
pair("従業員ID", 12)), pair("氏名", "推古天皇")),…
pair("役職", "天皇")),
set(set(set(o_{empty},
pair("住所ID", 333)), pair("県", "奈良県")),
pair("市町村", "明日香村")),
"従業員", "住所", "カスタマイズ R1"),
"部下", "上長", "カスタマイズR2");

そして、これを、map(db_{state}, ro)に与えて項書き換えを行うと、db_{state} 内部の h_{rec} について、次のような結果が得られる。

h_{rec} = put(put(h_{empty},
pair("従業員", addElement(addElement(v_{empty},
put(...put(put(h_{empty}, pair("従業員ID", 7)), pair("氏名", "聖徳太子")),…pair("住所ID", 325))), put(...put(put(h_{empty}, pair("従業員ID", 12)), pair("氏名", "推古天皇")),…pair("住所ID", 333))))),
pair("住所", addElement(addElement(v_{empty},
...put(put(put(h_{empty}, pair("住所ID", 325)), pair("県", "奈良県")),
pair("市町村", "斑鳩町"))…),
...put(put(put(h_{empty}, pair("住所ID", 333)), pair("県", "奈良県")),
pair("市町村", "明日香村"))…)));

これは、図 7 に示すような形で複合オブジェクトの内容がテーブル内にレコードとして格納されたことを表している。

リレーショナル
モデル

従業員ID	氏名	------	住所ID	上長ID
7	聖徳太子		325	12
12	推古天皇		333	

従業員テーブル　　外部キー

住所ID	県	市町村	
325	奈良県	斑鳩町	
333	奈良県	明日香村	

住所テーブル

図7. テーブルに格納された複合オブジェクトの内容

このようにして、仕様レベルでアプリケーションの機能に矛盾がないことを確かめていく。

4. 関連研究

コンポーネントの利用者の立場から、ブラックボックス化されたコンポーネントの状態に関する等価性を、代数的仕様の振舞意味論として定義し[5][7][8]、その等価性を検証する手法が提案されている[7][8]。そして、特に、コンポーネントがツリー状に組み合わされたアーキテクチャにおいては、上下コンポーネントの演算子間での詳細化の対応関係を、検証にどのように取り込むかによって、検証時の計算量を減らすことができることも示されている[8]。

これに対して、本論文では、コンポーネントの作成者の立場で、状態を直接的に表現して、処理に応じた状態の遷移を個々のコンポーネントの仕様として記述する。そして、利用者がコンポーネントを組み合わせてフレームワークを構築する際に、処理の連鎖を項書き換えによりシミュレートする形で、機能的な検証を行えるようにすることを主眼とした内容になっている。

5. まとめ

本論文では、コンポーネントベース・フレームワーク技術におけるアプリケーションの形式的仕様記述について、次の2点が可能であることを明らかにした。

- CBF に基づくフレームワークを始めとする複数モジュールの形式的仕様を合成して、アプリケーション全体の処理を記述すること
- アプリケーションの機能的側面の検証を、項書き換えにより実現すること

また、CBF の適用例として次の点を示すことができた。

- 業務トランザクション処理のコンポーネントセットを使って、オブジェクト指向モデルからリレーショナルモデルへのマッピング処理を行うフレームワークを構成すること

今後の予定としては、本論文で説明した代数的仕様や項書き換えの拡張を扱うことができるような処理系を実現して、検証実験を行うことを考えている。その際、これまでにも多くの実績があり、現在も北陸先端科学技術大学院大学で開発が進められているCafeOBJ[5]を処理系の候補とすることも検討中である。また、本論文で取り上げたオブジェクトモデルからリレーショナルモデルへのマッピング処理だけでなく、その逆のリレーショナルモデルからオブジェクトモデルへの復元処理についても、CBFのコンポーネントセットを使ってフレームワークを作成し、それを含むアプリケーションについて、その形式的仕様の記述と検証が可能かどうかを確かめたいと考えている。

参考文献

[1] 吉田, 本位田: コンポーネントベース・フレームワーク技術におけるコンポーネントの形式的仕様記述について, ソフトウェア科学会第17回大会, 講演論文集 (2000).

[2] 吉田, 本位田 : コンポーネントベース・フレームワーク技術におけるコンポーネントの抽出/設計方法論, 情報処理学会研究報告(2000－SE－128), Vol.2000, No.70, pp.25－35 (2000).

[3] Brown, K., Whitenack, B.G, : Crossing Chasms : A Patterns Language for Object-RDBMS Integration, http://www.ksc.com/article5.htm

[4] Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1, Equations and Initial Semantics. Springer-Verlag (1985).

[5] Diaconescu, R., Futatsugi, K.: CafeOBJ Report, AM AST Series in Computing 6, World Scientific (1998).

[6] Fowler,M., Analysis Patterns : Reusable Object Models, Addison-Wesley (1997). 堀内, 児玉, 友野 (訳):アナリシスパターン:再利用可能なオブジェクトモデル, 星運社, 東京 (1998).

[7] Goguen, J.A., Malcolm, G: A Hidden Agenda, Technical Report CS97-538, UCSD Technical Report (1997).

[8] Matsumoto, M., Futatsugi, K.: Test Set Coinduction – Toward Automated Verification of Behavioural Properties -, Proceedings of Second International Workshop on Rewriting Logic and It's applications, Electronic Notes in Theoretical Computer Science, Elsevier Science (1998).

[9] Pree,W.: Design Patterns for Object-Oriented Software Development, Addison-Wesley (1994). 佐藤, 金澤 (訳):デザインパターンプログラミング, トッパン (1996).

付録

1. [1]で示した業務トランザクション処理のコンポーネントセットの仕様に対する変更点

その1) 「保管」

database 型の導入に伴い、これに対する出力処理を行なう「保管」の仕様も、次のように変更になる。

database に対する挿入処理は insert の中で記述され、そこで例外が発生したかどうかは、process 型内部の戻り値で返すことにする。

また、主キーとなる識別子の値はオブジェクト生成時にクラスより付与されるものとし、その段階で一意性の制約は守られていることを前提として、もしもテーブル内に同じ主キーの値を持つレコードが既に存在していれば、挿入ではなくて、更新処理(update)を行うことにする。update の仕様については、付録 2.を参照のこと。

$invoke_{store}$: process, String → process

$invoke_{store}^{-1}$: object → object

$invoke_{store}([h_{in}, db_{out}, o_{ret}] + p, tn) =$
$\quad invoke_{store}([h_{in}, db_{out}, o_{ret}], tn) + invoke_{store}(p, tn);$

$invoke_{store}([h_{in}, db_{out}, o_{ret}], tn) =$
$\quad tryCatch(insertOrUpdate(h_{in}, db_{out}, tn), db_{out}, o_{ret});$

$unique(get(h_{rec}, tn), get(h_{pk}, tn), h_{in}) = true$ *imply*
$\quad insertOrUpdate(h_{in}, [h_{rec}, h_{pk}, h_{fk}], tn) = insert(h_{in}, [h_{rec}, h_{pk}, h_{fk}], tn)$

unique(get(h_{rec}, tn), get(h_{pk}, tn), h_{in}) = false *imply*

insertOrUpdate(h_{in}, [h_{rec}, h_{pk}, h_{fk}], tn) = update(h_{in}, [h_{rec}, h_{pk}, h_{fk}], tn)

tryCatch([h_{rec}, h_{pk}, h_{fk}], db_{out}, o_{ret}) =

[h_{empty}, substitute(db_{out}, [h_{rec}, h_{pk}, h_{fk}]),

return(o_{ret}, $invoke_{store}^{-1}$(null))];

tryCatch(Exception, db_{out}, o_{ret}) =

[h_{empty}, substitute(db_{out}, Exception),

return(o_{ret}, $invoke_{store}^{-1}$(Exception))];

$invoke_{store}^{-1}$(null) = null;

$invoke_{store}^{-1}$(Exception) = Exception;

その 2)　全コンポーネント

1)で述べたように、例外発生の有無を戻り値で返すために、次のような仕様を各コンポーネントに追加する。

$invoke_{classify}^{-1}$(Exception) = Exception;

ただし、子コンポーネントからの戻り値の受け渡しに vector が使われる「分割」と「変換」では、追加される仕様は次のようになる。

$invoke_{split}^{-1}$(addElement(v, Exception)) = Exception;

$invoke_{split}^{-1}$(addElement(v, null)) = $invoke_{split}^{-1}$(v);

$invoke_{split}^{-1}$(addElement(v_{empty}, null)) = null;

その 3)　「分割」

レコード間の関連を外部キーとして持つような複数のレコードの挿入処理の際に、[1]で示した「分割」の仕様では、parallel により複数の子コンポーネントから連鎖する処理が並行に進められるため、3.2 節で記述した外部キーの制約を必ずしも満たすことを保証できない。そこで、parallel を、子コンポーネントでの処理を逐次に行なうことを表現する sequential に置き換える。

$invoke_{split}$([h_{in}, v_{out}, o_{ret}], h_{map}, h_{child}) =

sequential($customize_{split}$(h_{in}, h_{map}), v_{out},

return(o_{ret}, $invoke_{split}^{-1}$(substitute(x, v_{empty}))), h_{child});

その 4)　「トランザクションタイプ」

2.3 節で示したトランザクション処理に関する仕様の中で、「トランザクションタイプ」を起点とするコンポーネントの組み合わせから成る処理を呼び出すことができるようにするために、$invoke_{ttype}$ の仕様を次のように変更する。

$invoke_{ttype}$: process, vector, hashtable → process

$invoke_{ttype}$([h_{in}, db_{out}, o_{ret}], v_{item}, h_{type}) =

[assign(h_{in}, v_{item}, h_{type}, h_{empty}), db_{out}, return(o_{ret}, $invoke_{ttype}^{-1}$(x))];

$invoke_{ttype}$ 以外の演算子については、変更はない。

2. 共通部分の仕様化

<u>オブジェクトに対する属性の設定、取得について</u>

set : object, association → object

get : object, object → object

get(set(o, pair(o_{key}, o_{val})), o_{key}) = o_{val};

<u>process 型のコンスタント p_{empty} について</u>

p_{empty} + p = p;

p + p_{empty} = p;

<u>vector どうしの連接について</u>

v_1 + addElement(v_2, e) = addElement(v_1, e) + v_2;

v_1 + addElement(v_{empty}, e) = addElement(v_1, e);

<u>database に対する更新処理(update)について</u>

unique(get(h_{rec}, tn), get(h_{pk}, tn), h_{in}) = false,

refer(h_{rec}, h_{pk}, get(h_{fk}, tn), h_{in}) = true *imply*

update(h_{in}, [h_{rec}, h_{pk}, h_{fk}], tn) =

[updateRecord(get(h_{rec}, tn), get(h_{pk}, tn), h_{in}), h_{pk}, h_{fk}];

unique(get(h_{rec}, tn), get(h_{pk}, tn), h_{in}) = false,

refer(h_{rec}, h_{pk}, get(h_{fk}, tn), h_{in}) = false *imply*

update(h_{in}, [h_{rec}, h_{pk}, h_{fk}], tn) = Exception;

unique(get(h_{rec}, tn), get(h_{pk}, tn), h_{in}) = true *imply*

update(h_{in}, [h_{rec}, h_{pk}, h_{fk}], tn) = Exception;

eq(get(h, key), eq(h_{in}, key)) = true *imply*

updateRecord(addElement(v, h), key, h_{in}) =

valsChanges(h, h_{in});

eq(get(h, key), eq(h_{in}, key)) = false *imply*

updateRecord(addElement(v, h), key, h_{in}) =

updateRecord(v, key, h_{in});

valsChanges(h_1, put(h_2, pair(k, v))) =

valsChanges(change(h_1, k, v), h_2);

valsChanges(h, h_{empty}) = h;

eq(k_1, k_2) = true *imply*

change(put(h, pair(k_1, v_1)), k_2, v_2) = put(h, pair(k_1, v_2));

eq(k_1, k_2) = false *imply*

change(put(h, pair(k_1, v_1)), k_2, v_2) =

put(change(h, k_2, v_2), pair(k_1, v_1));

change(h_{empty}, k, v) = put(h_{empty}, pair(k, v));

<u>database の整合性チェックのための演算について</u>

unique(addElement(v, h), key, h_{in}) =

unique(v, key, h_{in}) × eq^{-1}(get(h, key), get(h_{in}, key));

unique(v_{empty}, key, h_{in}) = true;

eq(get(h_{in}, fk), null) = true *imply*

refer(h_{rec}, h_{pk}, put(h, pair(fk, rtn)), h_{in}) = refer(h_{rec}, h_{pk}, h, h_{in});

eq(get(h_{in}, fk), null) = false *imply*

refer(h_{rec}, h_{pk}, put(h, pair(fk, rtn)), h_{in}) =

refer(h_{rec}, h_{pk}, h, h_{in}) × include(get(h_{rec}, rtn),

get(h_{pk}, rtn), get(h_{in}, fk));

refer(h_{rec}, h_{pk}, h_{empty}, h_{in}) = true;

include(addElement(v, h), key, val) =

include(v, key, val) + eq(get(h, key), val);

include(v_{empty}, key, val) = false;