2007－SE－157 (7)
2007／9／27

ソフトウェアの動的モデルに着目したラウンドトリップエンジニアリングの支援

今関雄人† 高田眞吾†

ラウンドトリップエンジニアリングは，モデリング段階とコーディング段階を往復しながらソフトウェア開発を行う手法である．ラウンドトリップエンジニアリングを支援するために，クラス図の変更をソースコードに自動的に反映し，またソースコードの変更をクラス図に自動的に反映するツールが提案されている．しかし，従来のツールでは，クラス図などの静的側面のモデルを扱うのみで，動的側面のモデル，すなわちシーケンス図やステートチャートとコード間のラウンドトリップエンジニアリングを扱うことはできない．そこで，本研究では動的モデルとソースコード間のラウンドトリップエンジニアリングを支援するツールを提案する．本ツールは MVC パターンに基づいたアプリケーションを対象とし，シーケンス図，ステートチャートとソースコード間のラウンドトリップエンジニアリングを支援する．本ツールを使用することにより，動的モデルとソースコード間の効率的なラウンドトリップエンジニアリングが可能となる．

Supporting Dynamic Models in Round-Trip Engineering

Yuto Imazeki † and Shingo Takada†

Round-trip engineering is a software development method that iterates between the modeling phase and coding phase. Conventional tools support class diagrams and code for round-trip engineering, i.e., they automatically reflect changes in class diagrams to code and vice versa. Thus, although they support static models such as class diagram, they do not support dynamic models such as sequence diagrams and statecharts. In this paper, we propose a tool that supports round-trip engineering between dynamic models and source code. This tool targets MVC pattern-based applications, and supports sequence diagrams, statecharts, and source code. We use information from the controller of the MVC pattern to conduct the transformations between the sequence diagrams, statecharts, and source code. This tool makes possible efficient round-trip engineering between dynamic models and source code.

1. はじめに

ラウンドトリップエンジニアリングは，フォワードエンジニアリングとリバースエンジニアリングを反復しながら開発を行う手法である[4)9)13)]．フォワードエンジニアリングでは，クラス図やシーケンス図などのモデルを基にしてソースコードを開発し，リバースエンジニアリングでは，ソースコードからモデルを生成する．すなわち，ラウンドトリップエンジニアリングとは，モデリング段階とコーディング段階を反復しながら開発を行う手法であると言える．ラウンドトリップエンジニアリングでは，モデリング段階とコーディング段階の反復を通して，モデルとソースコードを徐々に詳細化していく．このため，ラウンドトリップエンジニアリングは反復的なソフトウェア開発プロセス (プロトタイピングなど) に対して効率的な手法である．

ラウンドトリップエンジニアリングを行う際には，モデルとソースコードの整合性を保つ必要がある．したがって，モデルの変更をソースコードに反映する必要や，ソースコードの変更をモデルに反映する必要がある．ラウンドトリップエンジニアリングを支援するために，これらの反映を自動化するツールが提案されている[1)5)12)]．これらのツールは，クラス図の変更をソースコードに自動的に反映する機能や，ソースコードからクラス図を自動的に生成する機能を持っている．この機能により，クラス図とソースコードの整合性を保持しながら，モデリング段階とコーディング段階を往復することができる．しかしながら，これらのツールでは，静的な側面のモデルであるクラス図とコード間のみサポートしており，動的側面のモデル，すなわ

† 慶應義塾大学大学院理工学研究科
Graduate School of Science and Technology, Keio University

ちシーケンス図やステートチャートとソースコード間を扱うことができないという問題がある．

そこで，本研究では動的モデルとソースコード間のラウンドトリップエンジニアリングを支援するツールを提案する．本ツールは MVC パターンに基づいたアプリケーションを対象とし，シーケンス図，ステートチャートとソースコード間のラウンドトリップエンジニアリングを支援する．本ツールを使用することにより，動的モデルとソースコード間の効率的なラウンドトリップエンジニアリングが可能となる．

本論文の構成は次の通りである．まず 2 章でラウンドトリップエンジニアリングの関連研究と，その問題点について述べる．3 章では提案するラウンドトリップエンジニアリング支援ツールについて述べ，4 章で提案ツールの適用例を示し，考察を行う．最後に，5 章では結論を述べる．

2. 関連研究

本節ではラウンドトリップエンジニアリングを支援する関連研究とその問題点について述べる．

2.1 ラウンドトリップエンジニアリング支援ツール

ラウンドトリップエンジニアリングを支援するツールとして，Together[1]，Rational Software Modeler[5]，Eclipse UML[12] が挙げられる．これらのツールは，クラス図の変更をソースコードに自動的に反映する機能や，ソースコードからクラス図を自動的に生成する機能を持つ．この機能により，クラス図とソースコードの一貫性を保持することができ，クラス図とソースコードの効率的なラウンドトリップエンジニアリングが可能となる．しかしながら，これらのツールはクラス図などの静的側面のモデルを扱うのみで，動的側面のモデルについては扱えないという問題がある☆．

ソフトウェアの動的側面は主にシーケンス図とステートチャートを用いてモデリングを行う．そのため，動的モデルのラウンドトリップエンジニアリングを支援するには，モデルとソースコード間だけでなく，動的モデル間，すなわちシーケンス図とステートチャート間についても考える必要がある．次に，それぞれについての関連研究を挙げる．

2.2 シーケンス図とステートチャートの相互変換

Hasegawa らは，MSC (Message Sequence Charts)[6] と UML ステートチャートの相互変換を行う手法を提案している[3]．MSC は bHSC と呼ばれるシーケンス図と hMSC と呼ばれるシーケンス図の繋がりを表すモデルからなり，hMSC の情報を利用して変換を行う．

Hasegawa の手法では，シーケンス図におけるメッセージの送信と受信をステートチャートにおける状態遷移に対応させる．この対応関係により各シーケンス図から各オブジェクトのステートチャートを生成し，hMSC のシーケンス図の繋がり情報を利用して，生成したステートチャート同士を結合し，最終的なステートチャートを作成する．

また，ステートチャートの変更をシーケンス図に反映することが可能である．ステートチャートに行った変更に対応する差分 bMSC を記述し，その差分を既存の MSC に統合することにより，反映を行う．しかし，差分 bMSC は開発者が記述する必要があり，また，ステートチャートの変更は遷移の追加のみしか行うことができないという問題がある．さらに，シーケンス図がない状態では，ステートチャートからシーケンス図を生成することができない．

2.3 モデルからソースコードへの変換

動的側面のモデルからソースコードへ変換する関連研究として，Executable UML[10)14)] と，Harel の手法[2] が挙げられる．

Executable UML はステートチャートを拡張し，完全に実行可能なコード (雛形レベルのコードではない) を生成可能にしたものである．Executable UML では，アクション言語を導入しており，ステートチャートの状態内にメッセージの送信などの様々な処理を記述することができる．Executable UML の状態はメソッドに対応し，アクション言語による記述がメソッド内のソースコードに対応する．この対応関係を利用することにより，実行可能なコードが生成可能である．しかしながら，完全に実行可能にするためには詳細なコードまでアクション言語で記述しなければならず，モデルが肥大化し，読みにくくなるという欠点がある．

Harel の手法は，シーケンス図を用いた手法であり，LSC (Live Sequence Chart) というシーケンス図を拡張したモデルを用いる．LSC は，メッセージに加え，分岐構造やループ構造，一時変数などの構造を持っており，これらの要素をソースコードと対応させ，シーケンス図をひとつの関数のように表現する．また，シーケンス図のメッセージはメソッドが対応しており，メッセージに対応する実際のコードは LSC とは別に用意する必要がある．そのため，LSC のみでは完全に実行可能なコードを生成することはできず，LSC は関数を組み合わせるためのドライバのような役割を果たす．なお，Harel の手法では，LSC を直接実行す

☆ Together はシーケンス図に関しても一部サポートしている．

るため，コードの生成については考慮していないが，LSC を用いることでコードの自動生成も可能である．

2.4 ソースコードからモデルへの変換

Tonella らは，ソースコードを静的に解析し，シーケンス図 (およびコラボレーション図) を生成する手法を提案している[15]．Tonella の手法では，まずソースコードから，OFG (Object Flow Graph) と呼ばれる，オブジェクトが使用される流れを表したグラフを生成する．次に，ユーザが指定したメソッドから OFG をトレースすることで，シーケンス図を作成する．Tonella の手法をラウンドトリップエンジニアリングに用いる場合，既存のモデルと対応を取るためのメソッドを探す必要がある．すなわち，既存の全てのモデルについて，対応するモデルを生成するためのメソッドを指定しなければならない．また，各シーケンス図を別々に生成するため，生成した複数のモデル間の関連情報は存在せず，hMSC に対応するような情報も生成できない．そのため，ステートチャートのようなモデルを生成するのは困難である．そのような情報は解析の際にあらかじめ与えるか，実際に実行することにより解析する必要がある．

Malloy らはプログラムを実際に実行し，その結果を解析することによりクラス図やコールグラフ，シーケンス図などの様々なモデルを生成する手法を提案している[8]．しかしながら，実際に実行する手法では，特定の入力に対する実行結果のみに依存したモデルしか得ることができない．そのため，ある入力では発生しなかったメッセージが，別の入力では発生する場合などが考えられ，モデルに変換する際に情報が欠落してしまう可能性がある．変換の際に情報が欠落してしまうと，再変換した際に元に戻すことができなるという問題が発生する．このため，プログラムを実際に実行する手法は，ラウンドトリップエンジニアリングには不適切であると言える．

3. 動的モデルに着目したラウンドトリップエンジニアリング支援ツール

本論文では、動的モデルに着目したラウンドトリップエンジニアリング支援ツールを提案する．本ツールは MVC パターンに基づいたアプリケーションを対象とし，シーケンス図，ステートチャート，ソースコード間のラウンドトリップエンジニアリングを支援する．

3.1 概　　要

MVC (Model, View, Controller) パターン[7] は，ソフトウェアの設計パターンであり，ユーザへの入出力部分 (ビュー) と，ソフトウェアの内部的なロジック (モデル) を分離し，ビューとモデルをコントローラにより制御するというパターンである．近年，Web アプリケーションを中心として，広く用いられている．

MVC パターンアプリケーションのモデル部は通常のアプリケーションと同様であり，従来のオブジェクト指向開発を行うことができる．ここで，MVC パターンアプリケーションでは，モデル部がどのように実行されるかという情報をコントローラ部が保持しているという特徴がある．すなわち，モデル部の動的側面に関する情報の一部を，コントローラ部が保持していると言える．

そこで，本研究では，コントローラ部の情報を利用することにより，モデル部の動的側面のラウンドトリップエンジニアリングの支援を行う．さらに，本研究では，コントローラ部をモデリングするためのコントローラモデルを提案する．コントローラモデルにより，コントローラ部の開発と，ラウンドトリップエンジニアリングへの情報の利用を容易にする．なお，本研究ではビューに関しては扱わない．

3.2 機　　能

提案ツールは，次の機能を持つ．

(1) シーケンス図からステートチャートの生成
(2) ステートチャートからシーケンス図の生成
(3) ステートチャートからソースコードの生成
(4) ソースコードからシーケンス図の生成
(5) コントローラモデルからコントローラコードの生成

なお，本ツールでは，シーケンス図として UML シーケンス図，ステートチャートとして UML ステートチャートを扱う．また，ソースコードは Java を対象とする．なお，本ツールはクラス図に関してもサポートしているが，本論文では省略する．

本ツールでは，シーケンス図とステートチャートの間は完全に相互変換が可能であるため，動的モデルをひとまとめに扱うことが可能である．すなわち，シーケンス図とソースコード間，ステートチャートとソースコード間を別々に扱う必要はなく，動的モデルとソースコード間と，ひとまとめにして扱うことが可能である．そのため，シーケンス図から直接ソースコードを生成する機能はないが，この機能は一度ステートチャートに変換してからソースコードの生成を行うことにより実現している．同様に，ソースコードからステートチャートの生成は，一度シーケンス図を生成し，そのシーケンス図をステートチャートに変換することにより実現している．

これらの機能により，MVC パターンのモデル部に

関して，シーケンス図，ステートチャート，ソースコード間のラウンドトリップエンジニアリングを支援する．また，コントローラ部に関しては，完全に実行可能なソースコードがコントローラモデルから自動生成可能である．

次に，まず提案するコントローラモデルについて述べ，その後，各機能についての詳細を述べる．

3.3 コントローラモデル

コントローラモデルは，MSC[6] の hMSC を拡張し，遷移先のビューを記述可能にしたものであり，イベントにより発生するページおよびアクションの遷移を表現したものである．ここで，ページとは入出力が行われる ビュー部であり，JSP 等の Web アプリケーションでは Web ページに対応する．アクションとはコントローラが呼び出すロジックであり，各アクションとシーケンス図が 1 対 1 で対応する．また，イベントはビューから送られるリクエストに対応する．

コントローラモデルの例を図 1 に示す．図 1 はオンラインショッピングシステムの例である．例えば，select ページで add というイベントが発生すると，addItem というアクションを実行し，cart ページに遷移することを表現している．

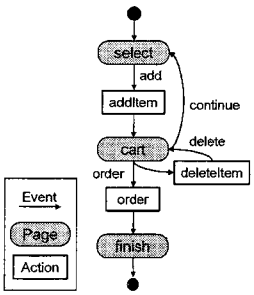

図 1 コントローラモデル

なお，図では省略しているが，各アクションについて呼び出すべきクラスおよびメソッドを指定する必要がある．他にも，本ツールではコントローラに対して様々な指定を行うことができるが，詳細は省略する．

3.4 シーケンス図 → ステートチャート

シーケンス図からステートチャートの生成は，Hasegawa の手法[3] に基づく．Hasegawa の手法では，メッセージの受信と送信がステートチャートの状態遷移に対応した．しかし，本手法では，ソースコードとの対応関係を考慮し，Harel の手法[2] と同様，メッセージの受信をメソッドの呼び出しと対応付ける．また，Executable UML[10][14] と同様，状態をメソッドに対応付ける．すなわち，本手法では状態の遷移はメッセージの受信のみと対応し，送信とは対応しない．また，本手法では Executable UML のアクション言語を導入する．本手法ではメソッド呼び出し，分岐，ループの 3 つの構造のみを扱う独自のアクションの記述法を導入し，それにより記述されたアクション (以降アクション記述と呼ぶ) を，それぞれシーケンス図のメッセージの送信，分岐，ループ構造にそれぞれ対応付ける．なお，自分自身へのメッセージの送信は，アクションの記述にのみ対応させ，状態の変化には対応させない．

ステートチャートの生成手順を以下に示す．

(1) 1 つのシーケンス図に対応するステートチャートの作成
前述の対応関係を使用し，1 つのシーケンス図分のステートチャートを作成する．メッセージの受信を状態の遷移とし，メッセージの送信元にはアクション記述を挿入する．分岐やループがある場合にも，同様にアクション記述を挿入する．これを全てのシーケンス図について行う．

(2) コントローラモデルから hMSC の作成
コントローラモデルに登場する全てのページを取り除き，1 つのアクションを 1 つの状態としたステートチャートを作成する (以後，これを便宜上 hMSC と呼ぶ)．ページを取り除く際に，ページに対する入力の遷移と出力の遷移をつなげた遷移を作成する．入力，出力が複数ある場合には，それらの全ての組み合わせの遷移を作成する．なお，同じ遷移が複数できる場合には，それらをまとめて 1 つにする．

(3) ステートチャートの組み立て
hMSC の各状態はシーケンス図に対応しているため，各状態を手順 1 で作成したステートチャートで置き換え，ステートチャートを組み立てる．なお，対応するステートチャートが存在しない場合には，単に状態を削除し，入力と出力の遷移をつなげる．手順 2，3 をシーケンス図に登場する全てのオブジェクトに対し繰り返し，ステートチャートを生成する．

図 2 は図 1 の各アクションに対応したシーケンス図の例である．“Page” は MVC のビューを表現している．各アクションはビューからのイベントにより呼び出されるため，各シーケンス図の先頭は常にビューからのメッセージとなる．

図 2 のシーケンス図から，本ツールは図 3 のステー

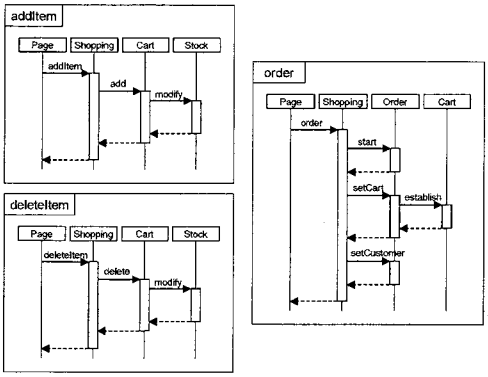

図 2 変換元のシーケンス図

トチャートを生成する☆.

ステートチャート "Cart" の状態 "add" はシーケンス図 "addItem" のメッセージ "add" に対応し，アクション記述 "Stock ->modify()" はシーケンス図 "addItem" でのメッセージ "modify" の送信に対応する．また，"Cart" での "add" から "delete" への遷移は，コントローラモデルでの "addItem" から "deleteItem" へのパスに対応している．

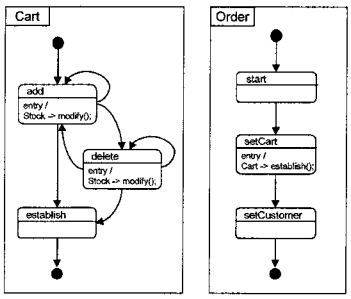

図 3 自動生成されたステートチャート

3.5 ステートチャート → シーケンス図

ステートチャートからシーケンス図への変換は，状態内のアクション記述をトレースし，メッセージの流れを解析することにより行う．ここで，コントローラの情報を利用することにより，トレースの開始位置を決定することができる．具体的な手順は以下のとおりである．

(1) 解析開始状態の決定

コントローラモデルを使用し，解析の開始位置を決定する．シーケンス図はコントローラの各アクションに対応するため，各アクションが呼び出すメソッドが解析開始位置となる．解析開始位置の決定後，"Page" からのメッセージをシーケンス図に挿入する．

(2) アクション記述のトレース

決定した解析開始位置から状態内のアクション記述を再帰的に解析しながらシーケンス図を作成する．メッセージの送信を発見した場合，まず，本ツールはシーケンス図にメッセージを挿入する．次に，そのメッセージに対応する状態をステートチャートから検索し，アクション記述の解析位置をその状態に移す．また，分岐を発見した場合には，分岐構造をシーケンス図に挿入し，分岐が閉じられるまでは，メッセージなどの挿入先をその分岐構造内にする．ループの場合も同様である．

例えば，図 3 の "add" 状態の解析では，"modify" メッセージが "Stock" オブジェクトに送信されているため，本ツールは "Cart" から "Stock" への "modify" メッセージの送信をシーケンス図に挿入し，解析対象を "Stock" の "modify" へと再帰的に移す．

手順 1，2 を全てのアクションに対し繰り返し，シーケンス図を生成する．

3.6 ステートチャート → ソースコード

本ツールでは，Executable UML[10)14)] と同様，各ステートチャートがクラスに対応し，状態がメソッドに対応する．そして，アクション記述がメソッド内のコードに対応する．しかし，完全なソースコードをステートチャートから生成するのは不可能である☆☆．そのため，本ツールでは既存のソースコードが存在しない場合 (すなわち，ステートチャートからソースコードを最初に生成する場合) には，ソースコードのテンプレートのみを生成する．テンプレートコードの生成では，前述の対応関係を使用し，クラスおよびメソッド定義，そしてアクション記述に対応するメソッド内のコードを生成する．

ソースコードが既に存在する場合には，本ツールはステートチャートと既存のソースコードの比較を行い，整合性を検証する．まず，比較を行う際，まず本ツールはソースコードからシーケンス図を経由して，現在のソースコードに対応するステートチャートの生成を

☆ 実際には "Shopping", "Cart", "Stock", "Order" の 4 つのステートチャートが生成されるが，ここでは二つのみを示す．

☆☆ ただし，アクション記述を非常に詳細に記述すれば可能である．

行う☆．次に，本ツールは生成したステートチャートと現在のステートチャートの比較を行う．比較では，各ステートチャートについて対応する状態が存在しているかどうかを確認し，存在している場合には，アクション記述が一致しているかどうかの比較を行う．本ツールでは記述可能なアクション記述をメッセージ送信，分岐，ループに限定しているため，アクション記述の比較は単純であり，一行ずつ比較を行う．

3.7 ソースコード → シーケンス図

ソースコードからシーケンス図の生成は，ステートチャートからシーケンス図の生成と類似した手法を用いる．まず，本ツールは解析開始位置をコントローラモデルから決定し，その後，アクション記述の代わりにソースコードを静的に解析する．

アクション記述ではメッセージの送信先オブジェクトがシーケンス図のオブジェクト名に対応していたが，ソースコードの場合はエイリアスがあるため，必ずしも対応しない．そこで，本ツールでは Tonella の手法[15] に基づいた静的解析を行う．Tonella の手法では OFG は1つのみ作成するが，本手法では，複数のシーケンス図間でのオブジェクトの同一性を確認するため，各解析位置からそれぞれ OFG を作成し，最後に統合を行う．しかしながら，実行パスによって変数の参照するオブジェクトが変化する場合には，OFG の統合は困難である．そこで，本研究では，クラスのフィールドが参照するオブジェクトは変化しないという仮定の基に OFG の統合を行った．

シーケンス図の生成アルゴリズムは以下の通りである．

(1) 解析開始位置の決定
ステートチャートからシーケンス図を生成するのと同様の手順で解析開始位置を決定し，View からのメッセージをシーケンス図に挿入する．

(2) ソースコードの解析
各解析開始位置からソースコードを再帰的に解析し，最初のシーケンス図を作成する．

(3) 冗長な分岐とループの除去
ソースコードの解析では，内部にメッセージ送信を含まない，冗長な分岐やループが現れてしまうため，それを取り除く．

(4) OFG の統合
インスタンス生成した場所の情報が含まれない OFG (すなわち，インスタンス生成した場所が不明なオブジェクト) を検索する．発見した場合，その OFG のルートノード (最初のオブジェクトへのアクセス) と同じノードを他の OFG から検索し，OFG を結合する．

(5) オブジェクトの決定
シーケンス図の各オブジェクトについて，OFG 内の対応するノードを検索し，OFG のルートノード名で，シーケンス図のオブジェクト名を置き換える．

☆ ソースコードからシーケンス図の生成については 3.7 節を参照

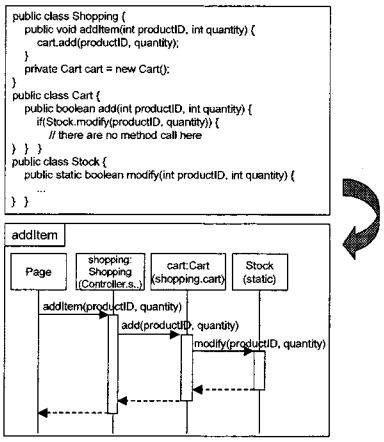

図 4 ソースコード → シーケンス図

図 4 はソースコードからシーケンス図を生成した例である．各メソッド呼び出しはシーケンス図のメッセージに対応し，メソッドの引数もメッセージの引数に対応する．なお，“Cart.add()” 内の “if” は，内部にメッセージ呼び出しがないため，シーケンス図には登場しない．また，シーケンス図のオブジェクト名はインスタンス生成された時点の名前と型により置き換えられる．例えば，“cart:Cart (shopping.cart)” は “shopping.cart()” 内でインスタンス生成された “Cart” クラスのインスタンス “cart” を表現している．また，“Stock (static)” は “Stock” が静的クラスであることを表現している．

3.8 コントローラモデル → コントローラコード

コントローラコードは，受信したイベントと現在のビューを基に，ページやアクションへの遷移を行うためのコードからなる．3.3 節で述べたとおり，各アクションには呼び出すべきメソッドなどの様々な情報が指定されている．そのため，アクションに遷移する場合にはそのメソッドを呼び出すコードを生成する．

コントローラコードはビューの実装に依存するため，ビューの実装ごとに異なるコード生成器が必要となる．本ツールは現在，JSP Web アプリケーション用のコ

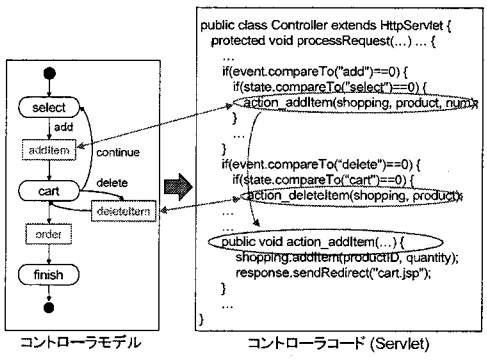

図 5 Controller model to controller code

ントローラコードの生成をサポートしている．図 5 に生成したコードの例を示す．図 5 のコントローラでは，呼び出すメソッドをアクション名と同名のメソッドとしており，クラスは “Shopping” クラスのインスタンスである “shopping” としている☆．コントローラは “HttpServlet” クラスのサブクラスとして実装した．“doGet” と “doPost” メソッドをオーバーライドし，リクエストを共通の “processRequest” メソッドで処理しており，イベント名と現在のビューにより呼び出すアクションを分岐している．本ツールは各アクションについて “action_addItem” のようなメソッドを作成し，分岐からはそのメソッドを呼び出す．

なお，本ツールはコントローラモデルからコントローラコードを生成するのみで，コードからモデルの生成については考慮していない．しかしながら，コントローラモデルから完全なコードを生成できるため，開発者はコントローラコードを編集する必要はないと考えられる．

4. 適用事例，考察

本節では，本研究で提案するツールの適用例を示し，考察を行う．本ツールの適用例として，オンラインショッピングシステムの開発を行った．反復型の開発プロセスを想定し，まずプロトタイプを作成し，プロトタイプに対して様々な変更を行った．

4.1 プロトタイプ開発

プロトタイプの開発では，まずコントローラモデルを使用して全体の処理の流れを設計した．これにより，システムの機能が明確となり，シーケンス図の作成単位が明確になるという利点がある．次に，コントローラモデルの各アクションをシーケンス図を用いてモデリングし，本ツールを使用してシーケンス図からステートチャートを作成した．その後，本ツールを使用してシーケンス図とステートチャートを相互に変換しながら，モデルを詳細化していった．具体的には，状態の追加や分割をステートチャートレベルで行い，クラスの追加や引数の追加をシーケンス図レベルで行った．この際，片方のモデルに変更を加えた後，すぐに本ツールを使用してもう片方のモデルに反映を行った．これにより，常にシーケンス図とステートチャートが同期を保った状態とすることができ，また，片方のモデルの変更がもう片方のモデルに与える影響も容易に知ることができ，効率的な開発が可能であった．

モデルを十分に詳細化した後，本ツールを使用してステートチャートからソースコードのテンプレートを作成した．その後，テンプレートを詳細化し，プロトタイプを完成させた．最後に，本ツールを使用してソースコードからシーケンス図を生成し，設計通りに実装できていることを確認した．

4.2 プロトタイプの拡張

プロトタイプに対し，様々な拡張や変更を行った．例えば，カートにある商品の個数を変更する機能の追加は，直接ソースコードを修正することにより行った．また，カートを空にする機能では，新たにシーケンス図を記述し，その後ソースコードを記述した．他にも，注文の最終確認ページの追加では，新たなシーケンス図の記述と既存のシーケンス図の修正を行い，変更をソースコードに反映した．他にも，支払い方法の選択やメール送信機能など，さまざまな拡張を行った．

本ツールを使用することで，機能の追加や要求の変更に対し，モデリング段階に戻って設計をモデルレベルで変更し，それをソースコードに反映することができた．さらに，ソースコードを直接編集し，それをモデルに反映することもできた．このように，本ツールを使用することにより，シーケンス図，ステートチャート，ソースコードから適切なものを選択して修正を行うことができ，効率的に開発を行うことができた．

4.3 考　　察

本節では，変換の妥当性および本ツールの適用範囲についての考察を行う．

まず，シーケンス図からステートチャートへの変換では，シーケンス図の全ての要素がステートチャートの要素に対応しているため，変換により情報が失われることはない．一方，ステートチャートからシーケンス図の変換では，本ツールは状態や遷移ではなく，ア

☆ “shopping” インスタンスはコントローラの初期化時に生成されるように指定している．このようなクラスはセッション内で保持され，マルチユーザアプリケーションでユーザを区別するのに使用可能である．

クション記述を解析して変換を行う．このため，状態の追加や削除を行った際，対応するアクション記述も追加/削除しないと，シーケンス図から再生成した際に元に戻らなくなってしまうという問題がある．この問題を解決するために，正しく変換できるステートチャートかどうかを検査する機能や，対応するアクション言語を自動的に修正する機能が必要である．

また，本ツールではソースコードを解析する際に，クラスのフィールドが参照するオブジェクトは変化しないという制限を設けた．制限が守られている限り，OFGの統合によりオブジェクトを特定できるため，必要な情報が欠落することはないが，違反する場合には正しい結果を得ることができない．本適用事例においてはこの制限が問題となることはなかったが，制限に違反するオブジェクトがあった場合にも，違反するオブジェクトがメッセージを送信していないオブジェクトに関しては解析が可能である．

また，本ツールでは，各種変換において，コントローラモデルが必須である．このため，本ツールを使用する場合には，基本的には最初から本ツールを使用して開発を行う必要がある．しかし，コントローラモデルを用意することにより，既存のアプリケーションに対しても使用可能である．この際，コントローラとモデル(ロジック)が明確に分離している必要はなく，各ロジックが任意のメソッドが呼び出されることにより起動する構造になっていれば，本ツールに対応するコントローラモデルを(手動で)作成することが可能である．このため，既存のアプリケーション等に対しても本ツールによる各種変換が可能であるが，この場合，コントローラモデルが正しいかどうかを検証することができず，コントローラコードの自動生成も行うことができない．

5. 結　　論

動的モデルとソースコード間のラウンドトリップエンジニアリングを支援するツールを提案した．本ツールはMVCパターンに基づいたアプリケーションを対象とし，シーケンス図，ステートチャートとソースコード間のラウンドトリップエンジニアリングを支援する．本ツールを使用することにより，動的モデルとソースコード間の効率的なラウンドトリップエンジニアリングが可能となる．

今後の課題としては，第4.3節で述べたステートチャートの検証およびアクション記述の自動修正や，ソースコードの静的解析で設けた制限の解決，既存のMVCフレームワークとの連携などが挙げられる．

参 考 文 献

1) Borland: "Borland Together Technologies", *http://www.borland.com/us/products/together/* (2006).
2) D. Harel, R. Marelly: "Specifying and Executing Behavioral Requirements: The Play In/Play-Out Approach", Software and System Modeling, pp.82-107 (2003).
3) H. Hasegawa, S. Takada, N. Doi: "Supporting the Iterative Development of Sequence Diagrams and Statecharts", Proc. of the 8th IASTED Int'l Conf. on Software Engineering and Applications, pp.736-742 (2004).
4) A. Henriksson, H. Larsson: "A Definition of Round-trip Engineering", Technical Report, *http://www.ida.liu.se/~andhe/re.pdf* (2003).
5) IBM: "Rational Software Modeler", *http://www-306.ibm.com/software/rational/* (2006).
6) ITU-T. Recommendation Z.120 (1999).
7) G. E. Krasner, S. T. Pope: "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system", Journal of Object Oriented Programming, Vol.1, No.3, pp.26-49 (1988).
8) B. A. Malloy, J. F. Power: "Exploiting UML Dynamic Object Modeling for the Visualization of C++ Programs", Proc. of the ACM Symposium on Software Visualization, pp.105-114 (2005).
9) N. Medvidovic, A. Egyed, D. S. Rosenblum: "Round-Trip Software Engineering Using UML: From Architecture to Design and Back", Proc. of the 2nd Workshop on Object-Oriented Reengineering, pp.1-8 (1999).
10) S. J. Mellor, M. J. Balcer: "Executable UML - A Foundation for Model-Driven Architecture", Addison Wesley (2002).
11) OMG: "Unified Modeling Language (UML), version 2.1.1", *http://www.omg.org/* (2007).
12) Omondo: "EclipseUML", *http://www.eclipseuml.com/* (2006).
13) S. Sendall, J. Kuster: "Taming Model Round-Trip Engineering", Proc. of Workshop on Best Practices for Model-Driven Software Development (2004).
14) L. Starr: "Executable UML - How To Build Class Models", Prentice-Hall, Inc. (2002).
15) P. Tonella, A. Potrich: "Reverse Engineering of the Interaction Diagrams from C++ Code", Proc. of the 19th Int'l Conf. on Software Maintenance, pp.159-168 (2003).