ユースケースを用いた文脈指向ソフトウェア開発

紙名哲生[†1] 青谷知幸[†2]
増原英彦[†1] 玉井哲雄[†1]

近年，実行時文脈に依存した実行を行うシステムへの要求が高まっており，文脈指向プログラミング（Context-Oriented Programming: COP）が提案されている．本研究は，ユースケースから，COP 言語の一つである EventCJ へと変換する手法について提案する．本手法により，特定の文脈で実行されるユースケースや，文脈がいつどのように変化するかに関する仕様を，設計や実装を通して分離したままシステムの開発を行うことが可能になる．

Context-Oriented Software Development with Use Cases

Tetsuo Kamina,[†1] Tomoyuki Aotani,[†2]
Hidehiko Masuhara[†1] and Tetsuo Tamai[†1]

Context-awareness is becoming a major requirement in many application domains. Context-oriented programming (COP) has been proposed to modularly implement such applications. This paper proposes a method to convert specifications written in use cases and UML state machine models into a program written in EventCJ, a context-oriented language with the novel feature of event-based context transitions. This method enables us to modularly design and implement context-specific use cases and the specification of context changes.

1. はじめに

現在，実行時文脈に依存した実行を行うシステムへの要求が高まっている．こうしたシステムでは，文脈に依存した振る舞いが開発過程の成果物（ユースケースやモジュールなど）に横断的に存在する．さらに文脈が実行時に変化し，それに応じてシステムの振る舞いも変化する．これらの理由から，こうしたシステムの開発には，システムの分析・設計・実装が繁雑になるという問題がある．例として，カーソル位置と表示テキストの内容で異なる振る舞いを実行するプログラムエディタを考える[4]．このエディタでは，カーソル位置の違い（コード上にあるか，コメント上にあるか）で表示するメニューやツールバーが変化し，表示テキストの違いで表示のしかた（リッチテキスト表示するかタイプライタ体で表示するか）が変化する．しかし，表示のしかたは，カーソル位置の違いによっても変えたい要求がある．このように，文脈に依存した振る舞いはシステムの様々な場所に現れ，それらを分離したままシステムの開発を行うことは難しい．

こうした問題を解決するため，文脈指向プログラミング（Context-Oriented Programming: COP）が提案されている[8]．COP では，特定の文脈で実行される振る舞いの集合を層という単位でモジュール化することができ，動的に層を活性化・非活性化することにより実行時にシステムの振る舞いを変化させることができる．これまでに様々な COP 言語が提案されたが[3),5),7),10]，COP に基づくソフトウェア開発の方法は確立されていない．例えば，各文脈に依存した振る舞いを，各工程を通して分離したまま管理する方法，文脈の動的変化のモデル化方法，それに基づいて文脈依存な振る舞い同士の矛盾や衝突を起こさずにシステムの実現を行う方法はこれまで提案されていない．

本研究では，ユースケース上で文脈とそれに対応する振る舞いを明示的に示し，COP の言語要素へと対応づけることで，これらの問題を解決する手法を提案する．とくに，以下を実現する．

(1) 文脈依存の振る舞いや文脈の動的変化を，各工程を通して分離したまま管理する．
(2) 文脈の変化をモデル化し，その上でのある種の性質（例えばある二つの文脈は同時に有効にならない）を検証できるようにする．

本研究ではこれらを，ユースケースから EventCJ[10] プログラムへの変換方法を示すことで実現した．この方法では，特定の文脈で実行されるユースケースをユースケース図上で明記する．また，Jacobson らによるユースケースによるアスペクト指向ソフトウェア開発[9] で提案されているように，ユースケース中のポイントカットを用いて文脈を変化させるイベントを定義する．そしてイベントによって引き起こされる文脈の変化を UML の状態マシン図で記述する．特定の文脈で実行されるユースケースと状態マシン図を，それぞれ EventCJ の言語要素である層と層遷移規則へと変換することによって，ユースケースから

†1 東京大学
University of Tokyo
†2 北陸先端科学技術大学院大学
Japan Advanced Institute of Science and Technology

© 2011 Information Processing Society of Japan

EventCJ への自然な変換を実現する．また，文脈の変化を状態マシン図で表現することにより，オートマトンに基づくモデル検査手法の適用を可能にする．なお，EventCJ はオブジェクトごとに異なる文脈を持たせることができることが特徴の一つであるが，本論文では簡単のため，大域的な文脈変化のモデル化のみを対象とする．

具体的事例として，CJEdit と呼ばれるプログラムエディタ[4] の例題を用いて，文脈に依存した振る舞いのユースケース分析を行い，特定の文脈で実行されるユースケースと文脈変化に関する状態マシン図を作成した．そしてそれらを EventCJ プログラムへと変換し，class-in-layer モデル（後述）の仮定のもとで，文脈および文脈の変化を，各工程を通して分離することが可能であることを示した．

以下，2 節で COP 言語と EventCJ に関する導入を行ったあと，3 節で本稿を通して使用する CJEdit プログラムエディタについて解説する．4 節で提案手法について述べ，5 節で本手法の有効性に関する議論を行う．6 節で関連研究について述べ，7 節で結論を述べる．

2. 文脈指向プログラミング

2.1 概　　要

文脈指向プログラミング（Context-Oriented Programming: COP）[8] は，実行時文脈に依存する振る舞いのモジュール化を実現するために考えられたプログラミング手法である．現在，それを実現する様々な COP 言語が提案されている[3),5),7),10)]．

COP 言語は主に以下の言語要素を提供する．

- 層（layer）: 特定の文脈で実行される振る舞いを抽象化したもの
- 部分メソッド（partial method）: 特定の文脈の個々の振る舞いを実現したコードの断片
- 層活性（layer activation）: 層を動的に活性化・非活性化させ，現在活性な層の部分メソッドのみを実行させる機構

図 1 は，Java に基づく文法で書いた層と部分メソッドの例である．これはプログラムエディタを実装するコードの断片であり，`TextEditor` クラスの中に `showToolbars`，`showWidgets`，`showMenu` という名前のメソッドが宣言されている．続く二つの層 `CursorOnCode` と `CursorOnComments` は，それぞれ「カーソルがコード上にある」という文脈と「カーソルがコメント上にある」という文脈を抽象化し，部分メソッドを宣言している．部分メソッドは，それを包含する層が活性な場合に実行される．例えば，`CursorOnCode` 内に宣言されている `after void showToolbars()` *Block* という宣言により，`CursorOnCode` が活性な場合，`TextEditor.showToolbars` の実行の直後で *Block* が実行されることになる．なお，

```
1  class TextEditor {
2    void showToolbars() { ... }
3    void showWidgets() { ... }
4    void showMenu() { ... }
5  }
6  layer CursorOnCode {
7    class TextEditor {
8      after void showToolbars() { ... }
9      after void showWidgets() { ... }
10   }
11 }
12 layer CursorOnComments {
13   class TextEditor {
14     after void showToolbars() { ... }
15     after void showMenu() { ... }
16   }
17 }
```

図 1　COP における層と部分メソッドの例
Fig. 1　An example of layers and partial methods

修飾子 `after` の代わりに `before` を付けると，オリジナルのメソッド実行の直前に部分メソッドが実行される．`before` や `after` を省略すると，オリジナルメソッドの代わりに部分メソッドが実行され，特殊メソッド `proceed` を用いて変更前のメソッドを部分メソッド内から呼び出すことができる．

なお，層とクラスの入れ子関係については，上記のようにクラスが層の中に宣言される class-in-layer 方式と，層がクラスの中に宣言される layer-in-class 方式，あるいはその両方を提供する言語が存在する．COP 言語の分類については文献 2) が詳しい．

層の活性状態は動的に変化し，部分メソッドは現在活性な層で宣言されているものだけが実行される．例えば，`CursorOnCode` が活性で `CursorOnComments` が非活性のときは，両方で `showToolbars` が宣言されているが，`CursorOnCode` の `showToolbars` のみが実行さ

© 2011 Information Processing Society of Japan

れる．

2.2 COP 言語 EventCJ における層の活性化機構

層を動的に活性化するために，COP 言語によって様々な方法が提供されている．EventCJ[10] では，いつ層活性の変化が起こり，それがプログラム実行中に存在するどのオブジェクトの振る舞いに影響を与えるかをイベントとして宣言し，各イベントに対応する層活性の規則を宣言的に記述する．これにより，文脈の変化をプログラムの他の部分から分離して実装することが可能になる．

イベントは，`declare event` 構文を用いて以下のように宣言される．

```
declare event MoveOnCode(TextEditor edit)
  :after execution(void TextEditor.onCursorPositionChanged())
    && this(edit) && if(edit.isCode());
```

このイベント宣言では，まずイベント `MoveOnCode` がいつ生成されるかを，AspectJ[12] のポイントカットを用いて記述している．つまり，`void TextEditor.onCursorPositionChanged()` の起動後で，且つ `edit`（これは `this` ポイントカットで束縛されてあるとおり，メソッドの実行主体である）に対する `isCode` 呼び出しが `true` だった場合に `MoveOnCode` が生成される．なお，EventCJ では `sendTo` 構文を用いてどのインスタンスにイベントを通知するかを指定することができる（指定しない場合はアプリケーション全体で，大域的に文脈が変化する）．本論文では，大域的な層活性のみを扱うため，`sendTo` については割愛する．

イベントに対応する層活性の規則は以下のように記述する．

```
transition MoveOnCode:
    CursorOnComments ? CursorOnComments -> CursorOnCode
  | -> CursorOnCode
```

ここでは，`MoveOnCode` に対応する層活性の規則を宣言している．この宣言では二つの規則を `|` 演算子で連結している．個々の規則は,「*Guard*?*Layers1*->*Layers2*」の形式をしている．*Guard* の部分には層の名前を書き，その層が活性であれば真と評価される．*Layers1* の部分には非活性化される層を書く．*Layers2* の部分には活性化される層を書く．指定する層がない場合は単に層の名前を書かない．`|` で連結された規則は左側から評価され，一番最初に適用可能な規則のみが実行される．従って，上の層活性の規則は，`CursorOnComments` が活性であればそれを非活性にして `CursorOnCode` を活性にする，そうでなければ単に `CursorOnCode` を活性にする，という意味になる．

このような遷移規則を用いる利点の一つは，モデル検査を用いた層活性に関する仕様の検査が容易になることである．EventCJ コンパイラは，層遷移規則から Promela によるプロセス記述を生成することにより，モデル検査を支援している．

3. CJEdit プログラムエディタの要求仕様と開発方法の問題点

この節では，本稿を通して使用する CJEdit プログラムエディタ[4] の例題を示す．このエディタの要求仕様をまとめると，以下のようになる．

> CJEdit プログラムエディタは，文法ハイライトによるコードの可読性と，リッチテキストフォーマットによるコメント文の可読性を高めるプログラムエディタである．このエディタは，新規作成・開く・保存・名前を付けて保存のファイルメニューと，元に戻す・繰り返し・切り取り・コピー・貼り付けの編集メニューを提供する．さらに，それぞれの機能に対応するツールをツールバーに表示する．
>
> メイン画面にはリッチテキストフォーマットで整形されたコメントが付いたソースコードが表示される．ソースコードはタイプライタ体で表示される．ユーザがコードを編集しているときは，ソースコードの文法ハイライトを行い，キーワードを強調してプログラムの可読性を高める．コメント編集時には，コード部分の文法ハイライトはオフにされ，コード部分全体がグレー色で目立たなく表示される．
>
> ユーザがコードを編集しているときは，メイン画面の横にコードのアウトラインビューを表示し，プログラム構造の理解を助ける．またツールバーには「プログラム実行」ツールを表示し，編集したプログラムを実行できるようにする．一方，ユーザがコメントを編集しているときは，フォント・書体・文字サイズ・段落のアラインメントの調整を行うメニューやツールがそれぞれメニューバーとツールバーに表示される．コード編集時に表示していたアウトラインビューと「プログラム実行」ツールは，コメント編集時には隠される．

この例題には次のような難しさが存在する．

- 特定の文脈で実行される振る舞いが横断的に存在し，分離して扱うことが難しい．例えば，カーソルがコード上にある場合の振る舞いは，ツールバーやメニューバーの表示と文法のハイライト処理に影響を与える．
- 文脈に依存した振る舞い同士の干渉を理解する必要がある．例えば，コード編集時とコメント編集時に使用される機能は排他的で，同時に有効になるものではないが，それが

© 2011 Information Processing Society of Japan

満たされているかを調べるために，モデル化により文脈がいつどのように変化するかを理解する必要がある．
- 上述のモデルから実装に移る際，実装との対応関係が分かりやすくなるように文脈の変化を分離したまま管理するのは難しい．

4. ユースケースから EventCJ プログラムへの変換

本研究では，以下に示すユースケースから EventCJ プログラムへの変換手順を与えることにより，前節で示した問題を解決する．

4.1 文脈依存なユースケースの分析

まず，ユースケース分析中に，文脈の種類と文脈に依存した振る舞いを識別する．

CJEdit には,「プログラムを書く」というユースケースが存在し，それが包含するユースケースとして,「テキスト領域の表示」が存在する．これらに対する文脈として，カーソル位置と表示するテキストの種類を識別する．それぞれカーソルがコード上にある文脈とコメント上にある文脈，及びコード領域をレンダリングする文脈とコメント領域をレンダリングする文脈で実行するユースケースが次のように変化する．
- 「プログラムを書く」ユースケースは，カーソルがコード上にある文脈では「コードを書く」を実行し，カーソルがコメント上にある文脈では「コメントを書く」を実行する．
- 「テキスト領域の表示」ユースケースは，コードをレンダリングする文脈では，カーソルがコード上にあるという文脈で「文法ハイライト」を実行し，カーソルがコメント上にあるという文脈で「ハイライトなし」を実行する．また，コメントをレンダリングする文脈では，カーソル位置に関わらず「RTF」を実行する．

本手法では，これら文脈に依存したユースケースは代替ユースケースではなく，独立した別のユースケースとして識別する．

4.2 汎化によるユースケース図上での表現

識別したユースケースをユースケース図を用いて表現する．このとき，文脈に依存したユースケース（例えば「コードを書く」）は，より一般的なユースケース（例えば「プログラムを書く」）の特殊なケースなので，ユースケース図上では汎化関係によって表現する．

図 2 に CJEdit のユースケース図を示す．特定の文脈で実行されるユースケースは，どの文脈で実行されるかをメモによって明示してある．

4.3 シナリオの作成

ユースケースシナリオを作成する．このとき，文脈に依存したユースケースが，汎化関係

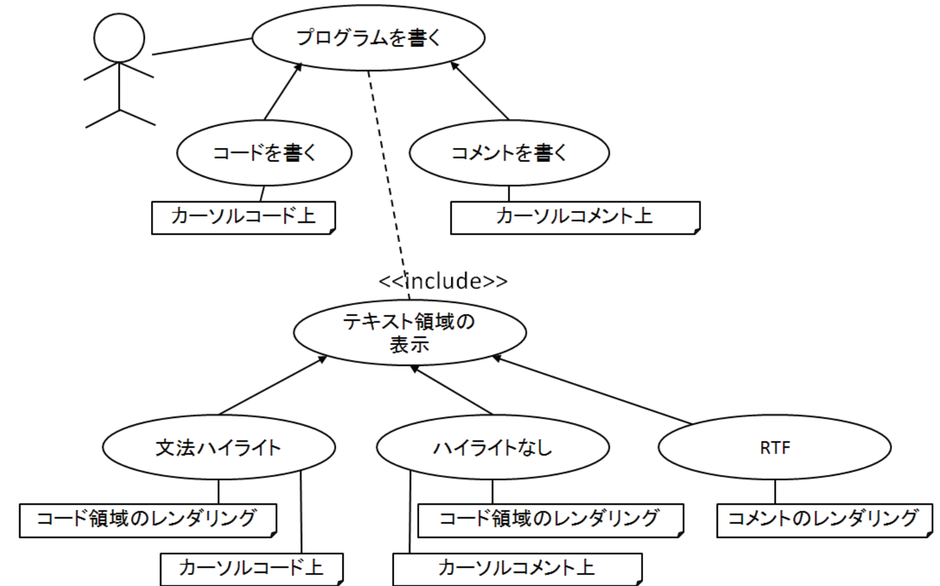


図 2　CJEdit のユースケース図
Fig. 2　Use case diagram of CJEdit

における親ユースケースのどの振る舞いを拡張するかを決定する．**図 3** に，例として「プログラムを書く」ユースケースと「コードを書く」ユースケースのシナリオを示す.「コードを書く」は,「プログラムを書く」の「プログラムを編集する」という動作を拡張したものとして描かれている．

4.4 ユースケース中のイベントの定義

次に，いつ文脈の変化が起こるのかを分析する．CJEdit では，カーソル位置が文脈として扱われており，4.1 節における分析結果により，それがコード上とコメント上の場合があることが分かっている．そこで，カーソル位置がコード上からコメント上，あるいはコメント上からコード上へと変化する時点を，ユースケースシナリオ中から探し出し，イベントとして定義する．これは Jacobson らの方式によるユースケース中のポイントカットを用いて定義できる[9]．例えば，図 3 の「プログラムを書く」からは以下のイベントを見つけ出すことができる．

© 2011 Information Processing Society of Japan

ユースケース：プログラムを書く
(1)　プログラマは,「開く」メニューよりプログラムを開く.
(2)　メイン画面にプログラムが表示される.
(3)　カーソルをコード領域に移動させたり，コメント領域に移動させたりしながら，プログラムを編集する.
(4)　編集が終われば,「保存」メニューよりプログラムを保存する.
ユースケース：コードを書く
(1)　メイン画面の横に，コードのアウトラインビューが表示され，ツールバーに「プログラム実行」ツールが表示される.
(2)　プログラマは，プログラミング言語の構文要素をメイン画面に書き込んでいく.
(3)　プログラマは,「保存」メニューによりプログラムの自動ビルドを行う.

図 3　ユースケースシナリオの定義
Fig. 3　Definitions of use case scenarios

- `MoveOnCode` = カーソルをコード領域内に移動させる.
- `MoveOnComments` = カーソルをコメント領域内に移動させる.

ここでは，Jacobson のポイントカット記法に倣い，等式の右辺にユースケースシナリオ内のアクションを自然言語で記述し，左辺にイベントの名前を記している.「カーソルをコード領域内に移動させる」ときに MoveOnCode イベントが生成されると読む.

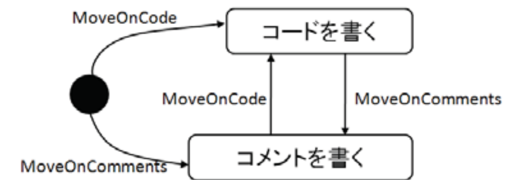
なお，以降では「テキスト領域の表示」(本論文ではシナリオの定義は省略する）から識別された以下のイベントも用いる.

- `StartCodeRendering` = コード領域のレンダリングを開始する.
- `StartCommentsRendering` = コメント領域のレンダリングを開始する.

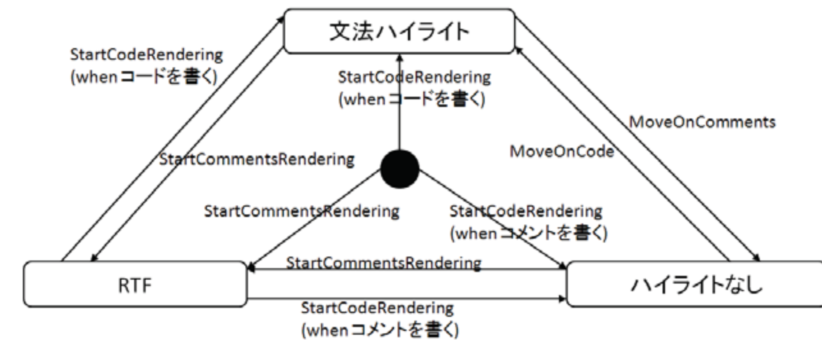
4.5　実行するユースケースの動的変化

ユースケースシナリオ中で定義したイベントを用いて，実行するユースケースがどのように変化するかを UML の状態マシン図を使って記述する.

図 4 に CJEdit における例を示す．ここでは，図 2 で書かれたユースケースが，上で定義されたイベントによってどのように変化するかが書かれてある．状態マシン図の各状態は，実行するユースケースを意味する．イベント `MoveOnCode` が生成されると「コードを書く」が実行可能になり，イベント `MoveOnComments` が生成されると「コメントを書く」が実行可能になる．それぞれ，遷移前に実行可能だったユースケースは無効化される．イベント



「プログラムを書く」に関するユースケースの変化



「テキスト領域の表示」に関するユースケースの変化

図 4　状態マシン図による文脈活性状態の変化
Fig. 4　Transitions of active contexts by state machine daiagram

`StartCodeRendering` が生成されたときは,「コードを書く」と「コメントを書く」のどちらが実行中かで実行されるユースケースが異なる．前者が実行中ならば「文法ハイライト」が実行され，後者が実行中ならば「ハイライトなし」が実行される．`StartCommentsRendering` が生成されると「RTF」が実行される.

システム全体のユースケース実行の仕様はこれらの状態マシン図を並行合成したものである⋆1．一つのイベントが，複数のユースケース実行を引き起こすことに注意する．例えば，`MoveOnCode` は「コードを書く」だけでなく,「文法ハイライト」も同時に起動する.

4.6　ユースケース・状態マシン図から EventCJ への変換

ユースケース，イベント，状態マシン図は，それぞれ EventCJ の言語要素である層，イ

⋆1 開発者は合成した状態マシン図を実際に作る必要はない.

© 2011 Information Processing Society of Japan

ベント，層遷移規則にマップさせて実装する．

4.6.1　クラスの識別

ユースケースを実装する際には，ユースケース記述を調べて，ユースケース実行の各段階で必要なクラスを決定する．これは Jacobson らの手法[9] と同じである．例えば,「プログラムを書く」ユースケースから以下のクラスと操作を識別する．

- TextEditor クラスは CJEdit の UI 全般を担い，メニュー，ウィジェット，ツールバーを表示する操作 showMenu，showWidgets，showToolbars を持つ．
- FileHandler クラスはファイルに対する操作を担う．例えば，プログラマの「保存」操作に対して，save を提供する．

4.6.2　層 の 実 装

特定の文脈で実行されるユースケースには，親ユースケースの振る舞いを拡張・変更する動作が記述されているが，実装時にはそれらを部分メソッドに対応させる．

以下に,「コードを書く」と「コメントを書く」を実現した層を示す．

```
1  layer CursorOnCode { /* 「コードを書く」ユースケースの実装 */
2    class TextEditor {
3      after void showWidgets() { .. }
4      after void showToolbars() { .. }
5    }
6    class FileHandler {
7      void save() { .. }
8    }
9  }
10 layer CursorOnComments { /* 「コメントを書く」ユースケースの実装 */
11   class TextEditor {
12     after void showMenu() { .. }
13     after void showToolbar() { .. }
14   }
15 }
```

「コードを書く」ユースケースが実行されると，ウィジェットとして新たにコードのアウトラインビューが追加され，ツールバーに「プログラム実行」ツールが追加される．これは TextEditor クラスの showWidgets，showToolbars を拡張する部分メソッドとして実現されている．また，FileHandler クラスの save を上書きし，保存後にソースコードの自動ビルドを行うようにする．これらの部分メソッドは，層を用いて一つのモジュールに実装されている．同様に,「コメントを書く」ユースケースに対応した層も実装されている．この場合は，それぞれ TextEditor の showMenu と showToolbar を拡張して RTF 編集用のメニューやツールバーの追加を実現している．

4.6.3　イベントの宣言

ユースケースシナリオ中で定義されたイベントによって示される操作に対応する条件を，クラス定義の中から見つけ出すことで，層遷移を引き起こすイベントを宣言する．CJEdit における各イベントの宣言を以下に示す．

```
1  declare event MoveOnCode(TextEditor edit)
2    :after execution(void TextEditor.onCursorPositionChanged())
3      && this(edit) && if(edit.isCursorOnCode());
4  declare event MoveOnComments(TextEditor edit)
5    :after execution(void TextEditor.onCursorPositionChanged())
6      && this(edit) && if(edit.isCursorOnRTF());
7  declare event StartCodeRendering(SyntaxHighlighter sh)
8    :before execution(void SyntaxHighlighter.highlightBlock(..))
9      && this(sh) && if(!sh.isCode());
10 declare event StartCommentsRendering(SyntaxHighlighter sh)
11   :before execution(void SyntaxHighlighter.highlightBlock(..))
12     && this(sh) && if(!sh.isRTF());
```

MoveOnCode，MoveOnComments はそれぞれ，TextEditor クラスの onCursorPositionChanged メソッドが起動した直後で，前者は isCursorOnCode 呼び出しが真の場合，後者は isCursorOnRTF が真の場合に生成される．StartCodeRendering と StartCommentsRendering は，それぞれ SyntaxHighlighter クラスの highlightBlock メソッドが起動される直前で，前者はそれまで isCode 呼び出しが偽である場合，後者はそれまで isRTF 呼び出しが偽である場合に生成される．

4.6.4　層遷移規則の実装

状態マシン図で書かれた文脈活性状態の変化を，EventCJ の層遷移規則を使って実装す

© 2011 Information Processing Society of Japan

る．この変換は機械的に行える．図 4 で書かれた状態マシン図を実装する層遷移規則は以下のようになる．

```
transition MoveOnCode:
    CursorOnComments ?
      CursorOnComments -> CursorOnCode, RenderWithHighlighting
  | -> CursorOnCode, RenderWithHighlighting
transition MoveOnComments:
    CursorOnCode ?
      CursorOnCode -> CursorOnComments, RenderWithoutHighlighting
  | -> CursorOnComments, RenderWithoutHighlighting
transition StartCodeRendering:
    RenderComments, CursorOnCode ?
      RenderComments -> RenderWithHighlighting
  | RenderComments, CursorOnComments ?
      RenderComments -> RenderWithoutHighlighting
  | CursorOnCode ? -> RenderWithHighlighting
  | CursorOnComments ? -> RenderWithoutHighlighting
transition StartCommentsRendering:
    RenderWithHighlighting ?
      RenderWithHighlighting -> RenderComments
  | RenderWithougHighlighting ?
      RenderWithoutHighlighting -> RenderComments
  | -> RenderComments
```

5. 議　論

1 節で示した目的に対し，本手法が解決できた点について議論する．また，本手法に向く COP 言語の設計や本手法の限界と今後の課題についても議論する．

5.1 関心事の分離

本手法では，特定の文脈に関するユースケースの実装が一つの層で実現されている（ただし class-in-layer モデルを仮定した場合）．よって，ユースケース分析時に分離した文脈に依存する振る舞いを，分離したまま実装することを達成した．さらに本手法では，いつ文脈依存の振る舞いが活性化・非活性化するかについて，イベントと状態マシン図を用いてユースケースシナリオから分離して記述し，それを EventCJ のイベント宣言と層遷移規則に対応づけている．よって，モデルから実装に移る際には，実装との明らかな対応を保持するように文脈の変化を分離したまま管理することを達成した．

5.2 モデル検査の適用

本手法では，とくに文脈の変化に関するモデルと EventCJ で書かれたコードとの間に，自明な対応関係が成り立っている．このことは，コード上で行われる解析手法が，そのままモデルにも適用できることを意味している．実際，EventCJ は Promela コードの自動生成機能を提供しており，実装がある種の性質（例えば「コードを書く」と「コメントを書く」は同時に実行されない，など）を満たしているかに関するモデル検査の支援を行っている．よって本手法は文脈の変化に関するモデル検査の適用を容易にする．

5.3 COP 言語の設計に対する議論

本研究では，クラスと層の入れ子関係について，class-in-layer モデルを仮定して議論した．実際には，多くの COP 言語は layer-in-class モデルを採用しており，EventCJ も先行の言語に倣ってこの方式を採用している．layer-in-class 方式では，文脈を実装するコードが様々なクラスに分散されるため，本手法のようにユースケースを一つのモジュール単位として扱う場合には不向きである．class-in-layer モデルも同時にサポートするような言語拡張か，またはツールサポートによってユースケースごとにソースコードを管理できるようにすることが望ましいと考えられる．

5.4 オブジェクトごとの文脈変化

本論文では，文脈の変化を大域的なもののみに限っている．そのため，オブジェクトごとに文脈が異なる場合を扱えない．例えば GUI アプリケーションの場合，タブなどで複数の画面を切り替えて使う場合に，各画面ごとに文脈が異なる場合がある．そのような場合は，ユースケースシナリオを分析することで層活性を起こすオブジェクトを特定し，それを EventCJ の `sendTo` に対応づけることで扱っていけると考えられる．

6. 関 連 研 究

比較的新しいプログラミング言語パラダイムを開発方法論の中に取り入れる研究は，主にアスペクト指向技術やフィーチャ指向技術[13) の分野で盛んに行われてきた．本研究は文脈指向に基づくアプローチを提案しており，これらの研究とは対象としている問題が異なる．

© 2011 Information Processing Society of Japan

これらは互いに補い合う関係にあると考えられる．

特に Jacobson らによるユースケースによるアスペクト指向ソフトウェア開発[9] は，本研究との類似点が多い．この手法では，ユースケースの実装が様々なクラスに跨る問題を，アスペクトを用いて解決している．また本研究のユースケース中でのイベント定義は，Jacobson らの手法における拡張ポイントカットに基づいている．本研究との大きな違いは，本研究が動的に変化する振る舞いに注目し，文脈の動的な変化に関する生産物（仕様・設計・ソースコードなど）を分離した点である．一方で，Jacobson らの手法では，拡張ポイントカットを定義することにより，既存の生産物に全く手を加えずに拡張を追加する方法が提案されているが，本研究では議論されていない．

フィーチャ指向ソフトウェア開発[1] は，分析段階で得られるフィーチャ図[11] を実装へと対応づける開発手法である．フィーチャ図はソフトウェアを構成する各フィーチャの依存関係の分析に適しており，とくにソフトウェアの製品ファミリーを作るのに向いている．文脈依存の振る舞いをフィーチャ図で分析することによって，層間の依存関係を分析する手法も提案されている[6]．本研究ではこうした依存関係の分析は考慮に入れておらず，むしろ動的に切り替わる文脈の分析に重きを置いている．そのため，シナリオに基づいて振る舞いを記述できるユースケースに基づくアプローチを採用した．

7. おわりに

本研究では，ユースケース上で文脈とそれに対応する振る舞いを明示的に示し，さらに文脈の動的な変化を状態マシン図でモデル化し，それを EventCJ で書かれたプログラムへと変換する手法が提案された．本手法により，特定の文脈で実行される振る舞いを，仕様・設計・ソースコードを通して分離することができ，いつ文脈が変化するかについても同様に分離することが可能になった．CJEdit の例題により，動的にシステム中の様々な箇所で振る舞いの変化する複雑なシステムであっても，モデルと実装との間に明らかな対応関係を形成し，統制のとれた開発が容易になることが示された．

今後は，より大規模なシステム開発を通して，またより多くの事例研究を積み上げることにより，本手法の有効性を検証していくことが望まれる．また，要求からいかに文脈やそれに依存した振る舞いを見つけ出すかに関する手順やガイドラインなどの開発も，重要な今後の研究課題として残されている．

参 考 文 献

1) Sven Apel and Christian Kästner. On overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.
2) Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *COP '09*, 2009.
3) Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. **コンピュータソフトウェア**, 28(1):272–292, 2011.
4) Malte Appeltauer, Robert Hirschfeld, and Hidehiko Masuhara. Improving the development of context-dependent Java application with ContextJ. In *COP'09*, 2009.
5) Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *SC'10*, volume 6144 of *LNCS*, pages 50–65, 2010.
6) Pascal Costanza and Theo D'Hondt. Feature descriptions for context-oriented programming. In *2nd International Workshop on Dynamic Software Product Lines (DSPL'08)*, 2008.
7) Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming – an overview of ContextL. In *DLS'05*, pages 1–10, 2005.
8) Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.
9) Ivar Jacobson and Pan wei Ng. *Aspect-Oriented Software Development with Use Cases*. Pearson Education, 2005.
10) Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: a context-oriented programming language with declarative event-based context transition. In *AOSD '11*, pages 253–264, 2011.
11) Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
12) Gregor Kiczales, Erik Hilsdale, Jim Hugunin, M ik Kersten, Jeffrey Palm, and William G. Grisword. An overview of AspectJ. In *ECOOP 2001*, pages 327–353, 2001.
13) Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP'97*, volume 1241 of *LNCS*, pages 419–443, 1997.

© 2011 Information Processing Society of Japan