2005－SE－149（1）
2005／7／28

アスペクト概念を持つステートダイアグラムの提案

中島 震　　玉井 哲雄
国立情報学研究所　　東京大学大学院

あらまし　アスペクト指向ソフトウェア開発の方法を開発上流工程に適用するために、モデリング言語 UML とアスペクトの関係を整理する研究が重要になってきている。本稿では、UML ステートダイアグラム (UML/STD) で表現する動的なモデルを対象として、アスペクトの考え方を考察する。UML/STD にジョイントポイントモデルの考え方を導入し、ポイントカットならびにアドバイスを定義することで、アスペクト記述を実現する方法を提案する。提案アスペクト拡張を実現する動作規則を標準 UML/STD への拡張として整理する。これによって、標準 UML/STD の動作規則の中核である RTC ステップ、ブロードキャストイベント、暗黙のイベント消滅、と密接な関係にあることがわかる。

キーワード　UML ステートダイアグラム、振舞い仕様、ジョインポイントモデル

A Proposal of Aspect-Oriented State Diagram

Shin NAKAJIMA　　Tetsuo TAMAI
NII　　Univ. Tokyo

Abstract　The notion of aspect is important to obtain design descriptions realizing the separation of concerns in early stages of the software development, which has raised questions on how UML is related to the aspect. This paper focuses on the notion of aspect in the case of UML State Diagram (UML/STD). It first introduces the Join Point Model (JPM) to UML/STD, which is based on the behavioral specification represented by a UML/STD design diagram. Then, the paper proposes Point Cut and Advice based on the JPM. The proposal includes an extension to the standard execution rule of UML/STD. And the extension is smoothly integrated with the core part of the standard consisting of the RTS step, the broadcast events, and the implicit consumption of the events.

Keywords　UML State Diagram, Behavioral Specification, Joint Point Model

1 はじめに

ソフトウェア・デザインの複雑さに対処するために、適切な” 関心事の分離 (separation of concerns)” が重要である [14]。ところが、ある観点から” 主要な関心事 (primary concerns)” を分離しクラスとして定義すると、複数のクラスにまたがる” 横断的な関心事 (cross-cutting concerns)” の取り扱いが難しい。そのために、アスペクト指向モデリングの考え方が試みられ、プログラミングから分析や設計を含むソフトウェア開発の各フェーズで有効であることがわかっている [1][4]。

一方、開発上流工程では、モデリング言語 UML [16] が標準として用いられるため、UML にアスペクトの概念を導入することが重要になってきている。UML はファミリー言語であり多様な側面を表現することができる。そのため、アスペクト概念の取り込み方法も多彩であるが、ステレオタイプ等の UML 拡張機能を用いてアスペクトのモデル要素を表現する方法が有力である [1][9]。しかし、ダイアグラムに基づくデザイン記法としての使い方に終る。形式的な解析ができない。

本稿では、アスペクト指向ソフトウェアのデザインに対して、デザイン記法のメカニズムの観点から議論する。UML ステートダイアグラム (UML/STD) で表現する振舞い仕様を対象とし、ジョイントポイントモデルを導入する。UML/STD の標準動作規則を拡張することで、アスペクトの考え方を容易に採り入れることができることを示す。

2 アスペクト指向ソフトウェア

2.1 モデリングとメカニズム

アスペクト指向ソフトウェアはモデリングとメカニズムに関わる技術である [1][4]。モデリングは、開発対象システムを分析し、見通しの良い関心・観点に分離することである。ある観点は主要な関心事であってベースとなる。別の観点は横断的な関心事を表現するアスペクトとみなされる。アスペクト指向プログラミングの例題として良く引合いに出されるログアスペクトやアクセス制御アスペクトでは、何をアスペクトとするかがわかりやすい。一方、開発上流工程では、ベースとアスペクトの切り分けが重要な課題であり、I.Jacobson によるユースケースを用いる方法 [6] などが提案されている。

メカニズムは、使用する言語あるいは記法に関わる。言語はベースとアスペクトを区別して表現する明確な言語要素を提供し、両者を組み合わせて全体を得る紡ぎ合わせ (weaving) の機構を持つ。アスペクト指向プログラミングの代表例である AspectJ [7] は Java プログラムとして作成したベースに対して、`aspect` と呼ぶ言語要素を用いてアスペクトを表現し、コンパイラが紡ぎ合わせ処理を自動的に行う。AspectJ を含む複数の言語を共通の観点から整理し紡ぎ合わせ処理の意味を与えることが可能である [10]。一方、モデリング言語 UML を用いる分析・設計では、ステレオタイプと呼ぶ UML の拡張機能を用いて、アスペクトを構成するモデル要素を明示する手法が提案されている [2][9]。紡ぎ合わせは設計者が行うモデル変換作業である。厳密な定義が望ましいが、UML 自身、明確な意味定義がないため、アスペクトまで含む議論を行うことが難しい。

2.2 ジョインポイントモデル

メカニズムの面から、アスペクト指向ソフトウェアの特徴を、quantification と obliviousness の 2 つに整理する [3]。前者はベースに対してアスペクトを紡ぎ合わせる条件のことであり、後者はベース自身にはアスペクトを紡ぎ合わせる情報が書かれていないことをいう。以下、AspectJ のジョインポイントモデル (JPM) [7][15] を基に、基本的な概念を整理する。

AspectJ のアスペクトは、アドバイスとポイントカットを用いて、ベースとなるプログラムの構造を横断するような振舞いを付加する。ベースプログラムの実行経路上、アドバイスが起動されるきっかけを与える点をジョインポイントと呼ぶ。ジョインポイントの集まりがポイントカットであり、あるポイントカットが指定するジョインポイントでアドバイスのアクションが実行される。

別の観点から述べると、ポイントカットは、ジョインポイントを表現する手段を与え、ベースプログラムの実行経路上のポイントを指定する条件であって quantification である。また、AspectJ では、ベースはポイントカットの有無に関わらず完結したプログラムであり obliviousness である。

モデリング言語 UML に対して、AspectJ の考え方にしたがってアスペクトの概念を導入することを考える。ジョインポイント、アドバイス、ポイントカット、をどのように定義するかが問題となる。UML では静的な情報構造を表現するクラスダイアグラムと動的な振舞いを与えるステートダイアグラムが代表的である。文献 [11] では静的な構造について、役割に基づくモデリングを基本とし、複数のモデル記述に登場する役割間の重ね合わせとして紡ぎ合わせを定義し、Alloy を用いることで形式的な解析が可能であることを示した。本稿では振舞いの側面に着目し、UML ステートダイアグラムを表現形式として用いる場合のアスペクトならびに紡ぎ合わせについて考察する。

3 振舞い仕様とアスペクト

3.1 UML/STD 動作規則と振舞い仕様

UML ステートダイアグラム (UML/STD) は階層的な状態遷移システムである。2 種類の階層を持ち、And 階層と Or 階層と呼ぶ。And 階層として展開される複数の状態遷移システムは並行実行すると考える。Or 階層内では遷移関係によって状態が移り変わり、いずれか一つの状態にある。UML/STD の実行とは、RTC と呼ぶ動作規則によって状態を遷移させていくことである。

本稿では、コンフィギュレーションマシンの考え方 [8] で定式化する方法を採用し、階層的な状態遷移システムとしてのステートダイアグラムに限定して議論する [12]。すなわち、UML/STD サブセットを

(State, Event, Rule)

と考える。ここで、

- State : (基底) コンフィギュレーション項の集合
- Event : イベントの集合
- Rule : 書き換え規則の集合

である。STD はコンフィギュレーション項を状態とする有限状態マシンである。

コンフィギュレーションは、ステートダイアグラムの実行スナップショットを表現する項表現である。トップレベルの状態をルートとし、And-Or 階層が作り出す木構造を簡明に表現することでスナップショットを表現する。表層のステートダイアグラムに示される遷移は、コンフィギュレーション項の書き換え規則と考えることができる。

コンフィギュレーションマシンはイベントプールと現在のコンフィギュレーションを管理する。イベントプールは以降に評価・実行されるイベントを保持する。UML の標準規則ではイベントプールの性質を規定していないが、一般的に FIFO キューと解釈される。本節ではイベントキューと考える。

RTC 動作は、イベントキューからイベントを取り出し、当該イベントによって発火可能な遷移を表す書き換え規則を全て実行する。当該イベントの処理が完了するまで、次のイベント処理を行わない。また、書き換え規則が衝突する場合は、予め決められた優先度によってどちらかを選ぶ、等の処理を行い、衝突しない書き換え規則の集合を求めることが特徴である。RTC 動作の切れ目では、コンフィギュレーション項とその時点でのイベントキューによって定義される平衡状態にある。

この時、UML/STD の振舞いは、RTC 動作が生成するコンフィギュレーション項の列のことであり、以下の定義による Run の集まりが、当該 STD の振舞い仕様と考える。

Run π はコンフィギュレーション項の列であり無限と考える。

$$\pi = s_0 s_1 \ldots s_n \ldots$$

s_{k+1} は s_k から RTC 動作の規則で計算されるコンフィギュレーション項である。なお、停止する STD では有限列の場合もあるが、Stutter 拡張の規則を適用することで、無限列と考えて良い。

先に述べたように、RTC ステップの切れ目での平衡状態は、コンフィギュレーション項とその時点でのイベントプールで規定することができる。コンフィギュレーション項が同じであってもイベントプール内のイベントが異なれば、以降の遷移進行経路が異なる。すなわち、RTC の平衡状態は

<Configuration, EventPool>

の組で表現される。このことを考慮した以下の拡張 Run を用いることでシステムの振舞いを一意に決定することができる。

$$\widehat{\pi} = \langle s_0, \xi_0 \rangle \langle s_1, \xi_1 \rangle \ldots \langle s_n, \xi_n \rangle \ldots$$

s_k は先の Run の定義と同様である。一方、ξ_k は RTC 動作の規則で計算される EventPool とした。

3.2 ジョインポイント

次に、アスペクトとの関係を考えるために、ジョインポイントを決める。ジョインポイントは実行経路上の点であるから、Run の構成要素である s_n (あるいは $\langle s_n, \xi_n \rangle$) とするのが良いであろう。この時、ポイントカットは、ジョインポイントを表現する手段であるから、s_n (あるいは $\langle s_n, \xi_n \rangle$) を指定するための条件式になる。

アドバイスは、指定のジョインポイントで起動されるアクションであった。アクションの効果を振舞い仕様の観点から考えるためにベースのみで生成する仮想的なな Run を想定する。この時、アドバイスはポイントカットが指定するジョインポイントに、新たな Run の部分列を挿入したり、仮想的な Run から部分列を削除することである。

たとえば、ベースに、アクセス制御アスペクトを紡ぎ合わせる場合、アクセス検査を実現する部分列が挿入される。さらに、アクセスが許可されない時は、アクセス動作が実現されないため、仮想的な Run から部分列を削除することに相当する。

言い替えると、Run を動的に変更するメタレベル操作がアドバイスである。デザイン表現の手段として UML/STD を用いている場合、振舞い仕様に影響を与えることが可能な方法は、イベントを生成して状態遷移を引き起こすことである。

UML/STD の標準動作規則で採用しているイベントの取り扱い方法を利用すると、ベースと同時並行的にアスペクトを評価することは容易である。ブロードキャストイベントであるため、ひとつのイベントが複数のマシンあるいは遷移に影響を与えることができ、その結果、ベースとアスペクトを並行動作させることが可能になる。これに関しては第 6.1 節で議論する。

一方、部分列の挿入や仮想的な Run からの部分列の削除は、UML/STD の標準動作では表現することができない。部分列を挿入する場合、一時的に当該マシンの進行を停止させ、挿入する部分列を生成した動作が完了した後に、再開させる必要がある。

部分列の削除は、あるイベントの評価前に進行を停止させて当該イベントを捨てる。ついで、進行を再開する際に、新たなイベントを生成し、停止前と異なる振舞いをおこせばよい。

アスペクトによって有効な振舞い仕様を作り出すためには、UML/STD の標準規則に加えて、ベースとなるステートダイアグラムの状態遷移進行を制御するメタ機構が必要となる。

4 提案のモデル

UML/STD の標準的な動作規則である RTC ステップに対する拡張として、先に述べたアスペクトを導入することが可能であることを示す。

4.1 ポイントカット

ポイントカットは拡張 Run の時点である組 $\langle s_n, \xi_n \rangle$ を指定する論理的な条件である。ポイントカット P の構文を示す。

P := M in S | E | $\neg P$ | $P \wedge P$ | $P \vee P$

命題 M in S は s_n に関連し、コンポーネントステートマシン M がコンフィギュレーション S にいる時に $true$ となる。たとえば、`currentState` が参照するコンフィギュレーション項が

```
System(..., M(S), ...)
```

の時に、命題 (M in S) は $true$ になる。一方、命題 E は ξ_n に関連し、イベント E が生成されてイベントセットにある時に $true$ となる。

4.2 進行制御の導入

進行制御の基本機構を一般化し、UML/STD の標準実行規則である RTC ステップに統合する意味論を説明する [12]。先の UML/STD サブセットを拡張し、

(State, Event, Rule, ProvidedClauses)

と考える。

ProvidedClauses は階層を構成するステートマシンに対して定義する遷移進行の条件である. N をステートマシンの名前とする時, 次の構文で指定する.

N provided P

ここで, P は以下の構文で表現する命題である.

P := M in S | $true$ | $false$ | $\neg P$ | $P \wedge P$ | $P \vee P$

命題 (M in S) はステートマシン M がコンフィギュレーション S にいる時に true となる。通常は M $\neq$ N である。

直観的には、ステートマシン N が条件 P を満たす場合のみ遷移可能なイベントを実行する。簡単な例を示す。

```
System(TaskA(A1),TaskB(A1))
```

であって、TaskA が次の provided 句を持つ場合を考える。

```
TaskA provided (TaskB in A2)
```

この時、TaskA は、TaskB が遷移して状態 A2 に移るまで、たとえ実行可能なイベントがイベントプールにあっても実行されない。

関係 ProvidedClauses は名前 (N) と命題 (P) の対応関係を与え、この対応関係を変更するプリミティブを仮定する。

```
provided(N) := P
```

は、ステートマシン N の進行制御命題を P に変更することを意図する。同時に、本変更プリミティブを状態遷移ラベルの作用記述部に持つ事を許す。これにより、状態遷移に伴って起こす効果のひとつとして進行制御命題の変更が可能となる。

最後に、提案モデルでは、イベントプールを FIFO キューとしないで、ひとつの RTC ステップ内で、イベントプール内の全てのイベントを評価対象とする”複数イベント評価方式” を採用した。この設計方針に関しては文献 [12] を参照されたし。

5 記述例

5.1 タスクの交互実行制御

最初の例を図 1 と図 2 に示す。図 1 は独立に並行実行する 2 つのタスクからなるシステムを示す。2 つのタスクは同じ振舞いを示し、状態 A1(B1) と状態 A2(B2) の間を自発的な遷移で交互に移る。コンフィギュレーション項を用いると初期状態を

```
System(TaskA(A1), TaskB(B1))
```

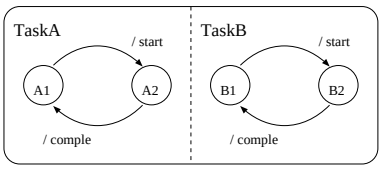

図 **1. Two Tasks**

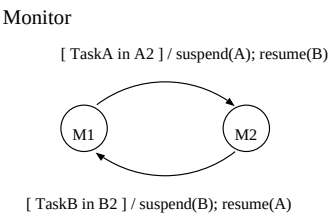

図 **2. Alternating Execution**

のように簡明に表現することができる。また、TaskA の遷移は次のような書き換え規則で表す。

```
[R1] System(TaskA(A1),TaskB(?X))
        --(/ start)-->
        System(TaskA(A2),TaskB(?X))
[R2] System(TaskA(A2),TaskB(?X))
        --(/ complete)-->
        System(TaskA(A1),TaskB(?X))
```

ここで、?X は $don't\ care$ を表し、適当なサブ項が束縛されることを意図する。

さて、上記の初期コンフィギュレーション項から開始すると、[R1] ならびに TaskB の対応する書き換え規則が発火可能であり、両者は衝突しないので、遷移を実行して、次のコンフィギュレーション項に移る。

```
System(TaskA(A2), TaskB(B2))
```

以下の説明で記述を簡明にするために簡易記法を導入する。

$\sigma_{ij} \doteq$ System(TaskA(Ai),TaskB(Bj))

簡易記法を用いると、初期コンフィギュレーション項 σ_{11} から開始する次のような経路 π_{free} を生成する。

$\pi_{free} \doteq \sigma_{11}\sigma_{22}\sigma_{11}\sigma_{22}\sigma_{11}\sigma_{22}\ldots$

この全体 $\Pi = \{\ \pi_{free}\ \}$ が図 1 の振舞い仕様である。

次に、図 1 に制限を加えて、2 つのタスクが交互に実行するような順序性を導入することを考える。たとえば、

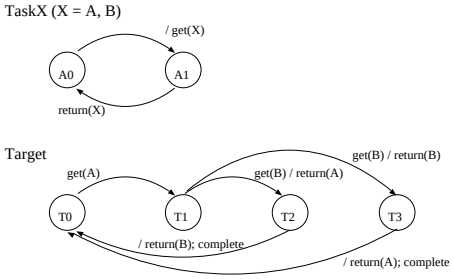

図 **3. Base**

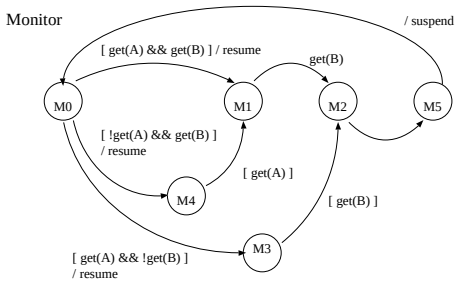

図 **4. Aspect**

$$\pi_{alt} \doteq \sigma_{11}\sigma_{21}\sigma_{11}\sigma_{12}\sigma_{11}\sigma_{21}\cdots$$

のような経路だけを生成するようにしたい。そのために、図 2 に示した Monitor マシンを 2 つのタスクの実行状態を横断的に監視するアスペクトとしてシステム直下の And 階層に追加する。システム全体のコンフィギュレーション項は

```
System(TaskA(A1),TaskB(B1),Monitor(M1))
```

のようになる。

Monitor マシンは以下のような動作を示す。2 つのタスクマシンの状態を監視し、TaskA が状態 A2 に達すると、自身が状態遷移を起こし、同時に、2 つのタスクに対する進行制御処理を起動する。

- suspend(A) により TaskA の状態遷移進行を一時停止させる
- resume(B) により TaskB の状態遷移進行を再開させる

さらに、TaskB が状態 B2 に達すると、上記と逆の進行制御を行う。ただし、TaskB は System の実行開始時点では、suspend(B) になっていることを仮定した。これ以外は、単に、図 2 を” 並行結合” するだけで、自由に動作する 2 つのタスクの順序性の制限を導入することができる。

上記の 2 つのプリミティブ関数は先に述べた進行制御機構を用いて容易に実現することができる。

```
suspend(N) ≒ provided(N) := false
resume(N) ≒ provided(N) := true
```

5.2 生成イベントの順序制御

図 3 と図 4 に 2 つめの例を示す。先の例と異なり、ベース (図 3) だけでは意図通りの振舞いを示さない。順序制御機能をモジュール性をよくするためにアスペクトとして導入する例である。

第 1 に、簡易表現を導入する。

$$\sigma_{ijk} \doteq \texttt{System(TaskA(A}i\texttt{),TaskB(A}j\texttt{),Target(T}k\texttt{))}$$

ベースシステムは全く同じ振舞いを示す 2 つのタスク (TaskA と TaskB) ならびに、両者から共有資源としてアクセスされる Target からなる。タスクは get(X) を出すと return(X) を待つ単純な構成である。一方、Target は受け付けるイベントに順序性を要請し、TaskA からの get(A) が TaskB からの get(B) よりも先行すること、ならびに、両方のイベントを受け取って後、はじめて実行を完了する (complete)。さらに、Target は 2 つのタスクに返す return(X) の順序を非決定的に決める。

このベースシステムに対して、常に、Target が動作完了するかを確認する。すなわち、complete イベントが生成されることを確認すればよく、LTL の式であれば

□◇(complete)

のように簡明に表現することができる。ところが、ベースシステムは、たとえば、次のような経路で動作しデッドロックに陥る。

$$\widehat{\pi}_{orig} \doteq \langle\sigma_{000},\{\,\}\rangle\,\underline{\langle\sigma_{110},\{\,get(A),get(B)\,\}\rangle}\,\langle\sigma_{111},\{\,\}\rangle$$

下線部でイベントプールに 2 つのイベントが生成されるが、Target は状態 T0 にいるため受け付け可能なイベントは get(A) だけである。イベント get(B) は遷移に寄与することなく暗黙の消滅の規則によって、イベントプールから削除される。一方、期待している経路の 1 例として以下がある。

$$\pi_{expected} \doteq \sigma_{000}\,\sigma_{100}\,\sigma_{111}\,\sigma_{112}\,\sigma_{010}\,\cdots$$

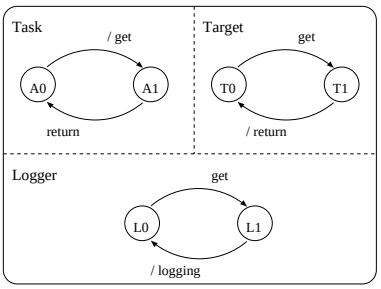

図 **5. Logging Aspect**

次に図 4 のアスペクトをベースに紡ぎ合わせたシステムの振舞いを考える。

$$\pi_{weaved} \doteq \sigma_{0000}\,\sigma_{1101}\,\sigma_{1112}\,\sigma_{1125}\,\sigma_{0100}\,\sigma_{1003}\cdots$$

ここで、関数 drop を導入する。これは、各コンフィギュレーション項から、アスペクトに相当するステートマシンの状態を除去するもので、紡ぎ合わせ前のベースに相当するコンフィギュレーション項を抽出することを可能にする。経路 π_{weaved} に関数 drop を作用させると、経路を構成するコンフィギュレーション項に drop を適用することとする。この時、

$$\mathrm{drop}(\pi_{weaved}) \doteq \sigma_{000}\,\underline{\sigma_{110}}\,\sigma_{111}\,\sigma_{112}\,\sigma_{010}\,\sigma_{100}\cdots$$

となり、意図通りに進行することがわかる。イベントプールまで考慮した拡張 Run の一部を示す。

$$\langle\sigma_{110},\{\,get(A),get(B)\,\}\rangle \rightarrow \langle\sigma_{111},\{\,get(B)\,\}\rangle \rightarrow \langle\sigma_{112},\{\,return(A)\,\}\rangle$$

6 考察

6.1 ログアスペクト

UML/STD の標準動作規則だけで、アスペクトを表現できるか否かを考える。図 5 はログアスペクトをベースに紡ぎ合わせる例である。ベースは `Target` とこれに `get` アクセスするタスク `TaskA` からなる。

アスペクトは `Logger` である。UML/STD の標準動作規則に従うと、`Logger` を And 階層として取り込むだけでよい。タスク `TaskA` が生成したイベント `get` はブロードキャストされるため、`Target` だけでなく `Logger` の遷移にも寄与することができる。その結果、ログ機能を追加することが可能となる。

$\sigma_{ijk} \doteq$ `System(TaskA(A`i`),Target(T`j`),Logger(L`k`))`

とする時、以下の経路を得る。

$$\pi_{Logger} \doteq \langle\sigma_{000},\{\,\}\rangle\,\underline{\langle\sigma_{100},\{\,get\,\}\rangle}\,\underline{\langle\sigma_{111},\{\,\}\rangle} \langle\sigma_{100},\{\,return,logging\,\}\rangle\,\langle\sigma_{000},\{\,\}\rangle\,\langle\sigma_{100},\{\,get\,\}\rangle\cdots$$

これから、たしかに、`Logger` マシンが作動し、イベントプールを監視することで、`logging` の生成を確認することができる。

遷移 $\sigma_{100} \rightarrow \sigma_{111}$ はイベント `get` のブロードキャストによって同時に状態を変化させることを示している。また、生成された 2 つのイベントの中で、`return` は遷移 $\sigma_{100} \rightarrow \sigma_{000}$ に寄与するが、イベント `logging` は発火遷移がないため暗黙の消滅の規則によってイベントプールから消える。

上記の系列を σ_{ij} が生成する π_{base} と比較すると全く同じである。

$$\mathrm{drop}(\pi_{Logger}) \equiv \pi_{base}$$

このように、簡単な Log アスペクトは、標準動作規則の範囲で、自然に追加することができる。

UML/STD では並行性に起因する複雑な振舞い仕様が問題となりやすい。並行性に関する側面のモジュール性を高めるためにアスペクトの考え方を用いることが大切を思われる。第 5 節は並行性に関する例題を議論した。

6.2 議論

本稿で提案した記述と動作規則が、アスペクトの特徴を持つことを確認する。

第 1 に、高い抽象レベルでは、アスペクトは 2 つの特徴、quantification と obliviousness を持つ。前者は、実行履歴全体を対象として起動の条件を表現することである。本稿の方法では、システムの実行スナップショットはコンフィギュレーション項とイベントプールの組でユニークに表現でき、この組の情報に対する条件を指定することができる。そのため、quantification の特徴を満たす。また、後者は、ベースの記述に何ら変更を加える必要がないことであり、少なくとも本稿の例題は、この特徴を持つ。

第 2 に、ジョインポイントモデルの観点から考える。アドバイスはベースに対する操作を実現することに関係する。本稿の方法は、ベースの計算モデルである UML/STD の動作規則にしたがってイベン

トを処理することができる。ポイントカットは、先の quantification で述べたように、振舞い仕様の上で定義している。

次の問題として、複数アスペクトを紡ぎ合わせることが可能であるかがあるが、さらに検討が必要である。

アスペクト指向ソフトウェアのデザインに対して、デザイン記法のメカニズムの観点から議論する研究は少ない。研究の焦点はモデリングの面であり、紡ぎ合わせは設計者が行うモデル変換作業であるとする。

一方、本稿では、UML/STD の動作規則を拡張する方法によって、振舞い仕様の上にアスペクトの概念を導入する機構が簡単に統合できることを示した。本稿では振舞い仕様とジョイントポイントモデルの関係に議論を絞り、その結果、比較的単純な例題を用いて基本的な考え方を説明した。

文献 [12][13] では振舞い仕様に関する一つの典型的な横断的な関心事として議論されるタスクのスケジューリングの問題を例題として取り扱った。プライオリティ逆転現象を示すデザイン、ならびにプライオリティ継承プロトコルを導入した解決方法をアスペクトの考え方で簡明に表現する例を示した。

7 おわりに

UML/STD にジョイントポイントモデルの考え方を導入し、ポイントカットならびにアドバイスを定義することで、アスペクト記述を実現する方法を提案した。アスペクトがベースに及ぼす効果は振舞い仕様の変換として理解できることを説明し、これを実現するための動作規則を標準 UML/STD への拡張として整理した。また、標準 UML/STD の動作規則の中核である RTC ステップ、ブロードキャストイベント、暗黙のイベント消滅、の考え方と密接な関係にあることを述べた。

なお、本稿記載の例題はすべて Promela/SPIN [5] で動作を確認した。本稿で提案したアスペクトの考え方とモデル検査との関係については別稿に譲る。

参考文献

[1] T. Elrad, R. Filman, and A. Bader. Aspect-Oriented Programming. *Comm. ACM*, Vol.44, No.10, October 2001.

[2] T. Elrad, O. Aldawud, and A. Bader. Expressing Aspects Using UML Behavioral and Structural Diagrams. In [4], pages 459–478, 2005.

[3] R. Filman and D. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In [4], pages 21–35, 2005.

[4] R. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development*. Addison Wesley 2005.

[5] G. Holzmann. *The SPIN Model Checker*. Addison-Wesley 2004.

[6] I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development with Use Cases*. Addison Wesley 2005.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97*, 1997.

[8] J. Lilius and I.P. Paltor. The Semantics of UML State Machines. TUCS TR No.273, May 1999.

[9] M. Mahoney, A. Bader, T. Elrad, and O. Aldawud. Using Aspects to Abstract and Modularize Statecharts. UML 2004 Workshop on Aspect-Oriented Modeling, October 2004.

[10] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *Proc. ECOOP 2003*, 2003.

[11] 中島 震, 玉井 哲雄. アスペクト指向モデリングにおける紡ぎあわせ. 日本ソフトウェア科学会第 21 回大会, September 2004.

[12] 中島 震. 状態遷移システムを用いたデザインのモデル検査. 電子情報通信学会技術研究報告 SS2004-59, March 2005.

[13] 中島 震. プライオリティ概念のあるステートダイアグラムのモデル検査. 電子情報通信学会技術研究報告 SS2005-XX, August 2005.

[14] D. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Comm. ACM*, Vol.15, No.12, pages 1053-1058, December 1972.

[15] M. Wand, G. Kiczales, and C. Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. ACM TOPLAS, Vol.26, No.5, pages 890–910, September 2004.

[16] OMG – Unified Modeling Language, v1.5, March 2003.