UML ステートダイアグラムを用いたアスペクト指向デザインとその検証

Aspect-Oriented Design using UML State Diagram and its Verification

中島 震
Shin NAKAJIMA
国立情報学研究所*
NII†

玉井 哲雄
Tetsuo TAMAI
東京大学大学院
The Univ. of Tokyo

振舞い仕様を対象とするアスペクト指向デザインに的を絞って、形式的な意味論を与えることで、デザインの表現だけでなくモデル検査に基づく検証を可能とすることを目標とする。UML ステートダイアグラムのサブセットによって表現可能な振舞い仕様を対象として、アスペクト指向デザインの特徴を表現する方法を提案する。さらに、Promela に変換することで SPIN モデル検査ツールを用いた検証の方法を議論する。

1 はじめに

横断的な関心事 (cross-cutting concerns) の表現や段階的な機能追加の方法としてアスペクトの考え方が重要である。ある機能を実現したベース記述に対して、追加的にアスペクトを紡ぎ合わせる [7]。プログラミングの技術だけでなく、モデリング技術としてもアスペクトの考え方が試みられ、ソフトウェア開発の各段階で有効であることがわかった [4]。

開発の上流工程では、対象システムを複数の観点からモデル化する。モデリング言語 UML [15] ではクラスダイアグラムで表現可能な静的な情報構造と共にステートダイアグラム (STD) を中心とする動的な振舞い仕様が重要な視点を与える。表現の視点に応じてアスペクト概念の取り込み方法も多様である [2][4][10]。我々は UML/STD が表現する振舞い仕様を対象として、アスペクト概念を導入する方法を提案した [12]。特に、AspectJ [7] の JPM の考え方 [9][14] を UML/STD に導入する方法を議論した。

本稿では UML/STD によるデザインを対象としてアスペクト指向デザインの振舞い検証を行う方法を議論する。線形時相論理 (LTL) を用いて表現した性質が成り立つか否かをモデル検査の方法で検証する。アスペクト紡ぎ合わせの効果をどのように考えれば良いかも議論する。以下、第 2 節でアスペクト指向ステートダイアグラムを定義し、第 3 節で検証の問題を議論する。第 4 節で適用事例を説明し、第 5 節でまとめる。

*総合研究大学院大学ならびに科学技術振興機構
†Sokendai and SORST/JST

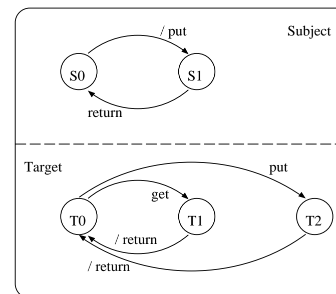

図 1: Base

2 アスペクト指向ステートダイアグラム

文献 [12] で提案した方法に基づきアスペクト指向ステートダイアグラムの概要を説明する。

2.1 アスペクトの考え方

本稿で提案するアスペクト指向ステートダイアグラムの概要を説明するために簡単な例題を考える。図 1 は `Subject` が `Target` にアクセスする簡単な例である。`Subject` は `put` を発行し `return` を受け取り初期状態に戻る。

次に、図 2 は `Subject` と `Target` の両者に跨るアスペクトの例である。本アスペクトを紡ぎ合わせることによって、`Subject` が `put` を発行した時、`Target` には `put-get` の列としてアクセスされる。

ベースの場合 (図 1) の振舞いを基準とすると、`put-return` の系列中に新たな部分系列 `return-get` が挿入され、その結果として振舞いが変化する。す

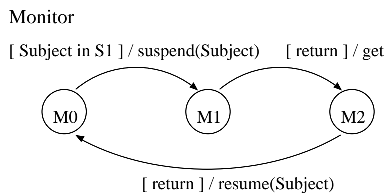

図 2: Aspect

なわち、アスペクトは、ベースが作り出す振舞いを変更するような効果をもたらす計算単位と考えることができる。

2.2 コンフィギュレーション

UML/STD は階層的な状態遷移システムである [15]。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で定式化する方法 [8] 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。UML/STD の定義として

(State, Event, Rule, ProvidedClauses)

を考える。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
- ProvidedClauses : 進行制御条件

である。すなわち、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。

図 1 の STM は、2 つのステートマシン `Subject` と `Target` から構成される And 階層を持つ。初期状態では

```
System(Subject(S0),Target(T0))
```

のコンフィギュレーション項で表現することができる。

また、`Subject` に定義されている遷移は、上記のコンフィギュレーション項間の書き換え規則として表現する。

```
System(Subject(S0),Target(?X))
--(/ put)--> System(Subject(S1),Target(?X))
System(Subject(S1),Target(?X))
--(return)--> System(Subject(S0),Target(?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。上の例では、`Subject` に関わる遷移を考える際、`Target` の状態は特に規定しないことを表す。

動作規則を考えるためには、ある時点で処理待ちになっているイベントを管理するイベントプールを導入する必要がある。この時、RTC の規則によると、イベントプール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、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[1]。さらに、ひとつのイベントは発火可能である限り、複数の遷移に関与してもよい (ブロードキャスト)。一方、その時点のコンフィギュレーションで発火可能な遷移がひとつもない場合、当該イベントは遷移に関与することなく消滅する (暗黙の消滅) [2]。

UML/STD の仕様書ではイベントプールの動作を定義していない。本稿では当該時点でイベントプールが含む全てのイベントを単一 RTC ステップで同時に評価することとする [11]。

2.3 振舞い仕様と JPM

前節の考え方を採用すると、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る。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。

ところで、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。したがって、振舞い仕様を次に示す拡張 Run で定義する。

[1] 予め決められた優先度を用いても衝突する場合がある [15]。
[2] 暗黙の消滅をさせないように遅延イベントとすることも可能である [15]。

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

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り、ξ_k は RTC 動作の規則で計算されるイベントプールとした。

アスペクトの概念を導入するために、ジョインポイントモデル (JPM) [9][14] を定義することを考える。本稿では、振舞い仕様を JPM と考える。すなわち、上記の拡張 Run を JPM とみなす。

2.4 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。

$$\mathrm{P} := \mathrm{M\ in\ S} \mid \mathrm{E} \mid \neg P \mid P \wedge P \mid P \vee P$$

命題 M in S は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。また、命題 E はイベント E が生成されてイベントプールにある時に $true$ となる。

図 2 の例では、`M0` から `M1` の遷移は `Subject` が状態 `S1` に到達する時に起こる。また、`M1` から `M2` の遷移はイベントプールに `return` イベントが生成された時に起こる。

2.5 進行制御の導入

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。

$$P := \mathrm{M\ in\ S} \mid false \mid true \mid \neg P \mid P \wedge P \mid P \vee P$$

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 となり、M $\neq$ N でもよい。たとえば、現在の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。

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。

図 2 で用いた 2 つのプリミティブ関数 `suspend` と `resume` は、次のように表現することができる。

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

2.6 動的な紡ぎ合わせ

本稿の提案方式では、紡ぎ合わせを動的に行う。すなわち、ベースのステートマシンの実行状況を監視し、指定されたポイントカットの条件が満たされると、アスペクトのステートマシンの当該遷移が起こり、したがって、アドバイスが実行される。

図 1 と図 2 の例題に対して、実行系列を具体的に追跡することで、紡ぎ合わせの効果を調べる。以下の説明を簡明に行うためにコンフィギュレーションの簡易記法を導入する。

$$\sigma_{ij} \doteq \mathtt{System(Subject(S}i\mathtt{),Target(T}j\mathtt{))}$$

ただし、i = 0,1 であるが、j = 0,1,2 である。また、イベントプール内に存在するイベントを e_k として明示することとし、拡張 Run を構成するスナップショットを次のように表す。

$$\langle \sigma_{ij}, \{\ e_k\ \} \rangle$$

ベースだけの場合は、

$$\pi_{base} \doteq \langle \sigma_{00}, \{\ \} \rangle \langle \sigma_{10}, \{\ put\ \} \rangle \langle \sigma_{12}, \{\ \} \rangle \langle \sigma_{10}, \{\ return\ \} \rangle \langle \sigma_{00}, \{\ \} \rangle \langle \sigma_{10}, \{\ put\ \} \rangle \ldots$$

であるが、アスペクトが存在すると、σ_{10} の時点で `M0-->M1` の遷移 (図 2) を発生する。その効果として、`Subject` の動作進行を一時停止させると同時に、`get` イベントを発生する。このイベントにより、`Target` は動作進行し `return` を生成する。アスペクトはイベントプールを監視しており、この `return` イベント生成をきっかけとして `M2-->M0` の遷移を起こし、`Subject` の再開させる。`Subject` は `return` イベントによって遷移してもとに戻る。すなわち、

$$\pi_{weaved} \doteq \langle \sigma_{00}, \{\ \} \rangle \underline{\langle \sigma_{10}, \{\ put\ \} \rangle} \langle \sigma_{12}, \{\ \ \} \rangle \langle \sigma_{10}, \{\ return,\ get \} \rangle \langle \sigma_{11}, \{\ \ \} \rangle \langle \sigma_{10}, \{\ return \} \rangle \langle \sigma_{00}, \{\ \} \rangle \ldots$$

のような経路に変換されることになる。下線部が最初のポイントカットであり、以降、π_{base} に対して、部分列を挿入するような形で振舞いを変化させる。

このように経路変換あるいは振舞い変更の機構は、先に述べた進行制御が主役であるが、UML/STD の RTC ステップに基づく動作規則と融合して実現している。

3 アスペクト指向デザインの検証

3.1 モデル検査ツールを用いた検証

先に、振舞い仕様を拡張 Run として定義した。本稿では、振舞い仕様の性質表現として, 本稿では線形時相論理 (Linear Temporal Logic, LTL) の式 f を採用する． LTL の論理式 f は, 通常の論理結合子 ($\neg$, $\wedge$, $\vee$, $\Rightarrow$) に, 3 つの時相オペレータ $\Box$(always), $\Diamond$(eventually), U(strong until) を追加して表現するとする．なお, 3 つの時相オペレータの意味は通常の定義 [5] と同じなので省略する．

振舞い仕様の拡張 Run は RTC ステップからなる動作規則で生成される。一般には、書き換え規則が衝突する場合があり、衝突しない最大集合を求める必要がある。逆に, 当該の最大集合の具体的な構成方法を複数通り考えることが可能である。検証する立場では可能な全ての経路について与えられた性質が成り立つか否かを検査するので、 可能な方法で構成した全ての Run に対してチェックする。すなわち、RTC 動作規則によって網羅的に生成された拡張 Run の集合 $\{\hat{\pi}\}$ を作り, この集合の要素に対して与えられた式 f が成り立つか否かを検査する。

$$\forall\ \hat{\pi} \models \mathrm{f}$$

である。なお、LTL 式が参照するアトミックな命題としては、ポイントカットの項と同様に

- M in S
- E ($\in$ Event)

を考えることができる。

本稿ではモデル検査ツール SPIN[5] を用いて振舞い検証を行う．基本的には, 動作規則を反映する Configuration マシンを構成し, 遷移集合の計算処理を含むイベント評価ならびに RTC を実現する動作規則をモデル検査ツールの入力仕様言語 Promela で表現する．網羅的な経路生成を行うためには, 非決定的な遷移計算を行うように Promela 記述を生成すれば良い．

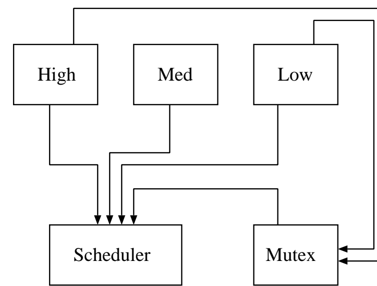

図 3: Process Configuration

3.2 紡ぎ合わせと LTL 検証

LTL 式で表現した性質の検証と紡ぎ合わせの関係について考察する。例として図 1 ならびに図 2 を考える。図 1 だけの場合、

```
[]<>put && !([]<>get)
```

であるが、図 2 を紡ぎ合わせると、

```
[]<>put && []<>get
[](put -> <>get)
```

などが成り立つ。すなわち、紡ぎ合わせによって、ベースが持っていた性質を壊すことがある。むしろ、これらは、アスペクトによって実現したかった要求性質と考えるべきである。

ところで、例題の場合に紡ぎ合わせ前後で、`Subject` の振舞い仕様は

```
[](put -> <> return)
```

を満たす。検査対象のどの部分に着目するかで、紡ぎ合わせ前後で保存される性質もあれば壊れる性質もある。本稿の方法では、適切な LTL 式を用いることで、アスペクトの効果を議論できると考えるのが良い。

4 スケジューラへの応用

次に興味ある例題としてスケジューリングが関係するデザインの表現と解析の問題を扱う [11][13]。一般に、スケジューリングは、複数の構成コンポーネントに跨る横断的な関心事であるとされることが多い。特に、本稿では、Mars Pathfinder の事故で有名になったプライオリティ逆転の状況を扱う。プライオリティ逆転は、優先度スケジューリングと排他的

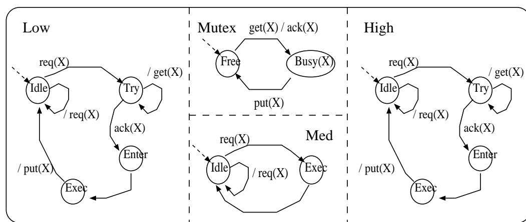

図 4: Priority Inversion

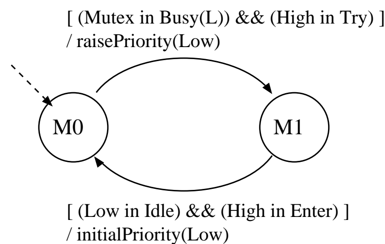

図 5: Aspect for Priority Inheritance

な共有資源の両方がある場合に起こり得る不具合である。

図 3 は全体の構造を表現する模式図である。3 つのタスクがあり各々名前が示すような優先度を持ち、`High`、`Med`、`Low` の順番とする。タスク `High` と `Low` は資源 `Mutex` を共有する。また、スケジューラを明示した。

プライオリティ逆転は以下のような状況で発生する。`Low` が `Mutex` を獲得して、`High` が資源解放待ちであって、さらに、`Med` が実行要求する時を考える。`Med` と `Low` の優先度比較によって、`Med` に実行権が与えられる。`Med` が占有実行すると、`Low` は待ち状態となる。`Low` は共有資源 `Mutex` をロックしているため、`High` は資源解放待ちになり動作しない。すなわち、`Med` が実行し `High` が実行できないという優先度が逆転した状況に陥る。

標準の UML/STD 等の状態遷移に基づくデザイン技法を用いる場合、図 3 に示した構成を忠実に反映したモデルを作成する。タスク等の構成要素をステートマシンとして表現し、ステートマシン間の情報交換をメッセージ通信で行う等を考えることができる。しかし、基本となるプライオリティ逆転の状況を再現するためにはスケジューラを明示する必要がある。スケジューラはタスクや共有資源と情報交換を行うため、独立性が低い。

本稿の方法では、スケジューリングに関する処理を、より高いモジュラリティで表現することができる [11]。第 1 に、タスクの優先度の違いを表現するために provided 句を導入する。

$$
\begin{aligned}
&\text{Med provided } \neg((\text{High in Exec}) \vee (\text{High in Enter}))\\
&\text{Low provided } \neg((\text{High in Exec}) \vee (\text{High in Enter}))\\
&\qquad\qquad \wedge \neg(\text{Med in Exec})
\end{aligned}
$$

これによって、たとえば、`Med` は、`High` が `Exec` あるいは `Enter` 状態でない時にのみ状態遷移を進行できることを表現する。

次に、本システム全体に対して、性質を時相論理 LTL の式として表現して検証する。

$$\Box((\text{High in Try}) \rightarrow \Diamond(\text{High in Exec}))$$

この式は進行性、あるいは leads-to の性質を表現するもので、`High` は共有資源獲得要求 (`Try` 状態) の後、いつか作動状態 (`Exec`) に移行することを示す。正しく共有資源を獲得する場合、この式は満たされる。しかし、本例の記述では、中間優先度の `Med` が作動することによって、`Low` が共有資源を保有したままの状態にいるという状況が反例となる。`High` よりも優先度の低い `Med` が実行することを示し、したがって、プライオリティ逆転の現象を生じることがわかる。

プライオリティ逆転の問題を避ける方法がいくつか知られており、たとえば、プライオリティ継承と呼ぶ解決方法がある [3]。プライオリティ継承は、共有資源を獲得した状態で `Ready` 状態になっているタスクの優先度を一時的に高くすることで問題を解決する。

図 3 を素朴に実現したステートマシンの基本機能に加えて、プライオリティ継承の機能を追加することを考える。共有資源をロックしているタスク (`Low`) の優先度を一時的に高める処理と元に戻す処理が必要である。これは、`Scheduler` と `Mutex` 間に新たなイベントを導入し、両者に適切な処理を追加することになる。複数のステートマシンの振舞いに影響を与える” 横断的な関心事 (Cross-cutting Concerns)” になっていることがわかる。

アスペクトの考え方でプライオリティ継承を実現するためには、図 5 に示す監視オートマトン [13] を追加すればよい。すなわち、図 4 のステートダイアグラムに当該アスペクトを And 階層の要素として結合する。

監視オートマトン (図 5) は定常状態では M0 にあり、他のステートマシンの状況を監視している。共有資源 Mutex が Low タスクにロック (Busy(L)) されると同時に High タスクが共有資源を要求する状況に達すると M1 状態に遷移する。遷移の際に、Low タスクの優先度を一時的に高める (raisePriority(Low))。次に、Low タスクが共有資源を解放した後に Idle 状態に移り、同時に、High タスクが共有資源を獲得して動作処理を開始した時点 (Enter 状態) で、Low タスクの優先度を元に戻す (initialPriority(Low))。

具体的には、raisePriority(Low) として、

```
provided(Low) := true
```

を選び、Low タスクが常にスケジューリングされるように設定した。これは、プライオリティ継承の方法よりも強い条件であるが、ここで示した状況では、Med よりも高い優先度を一時的に与えるという目的に合っているので、これでも良いだろう。

このアスペクトを紡ぎ合わせた全体システムは意図通りの振舞いを示し、High タスクに対する先の leads-to 性を満たすことがわかる。

5 議論とおわりに

本稿では UML/STD の RTC ステップを中心とする動作規則と整合する形でアスペクトの考え方を導入した。振舞い仕様を JPM と考えてポイントカットを定義した。ここで、アスペクトの効果はベースが作る振舞い仕様への部分列挿入などの変更であるとした。この振舞い仕様の変更は進行制御機構を導入することで可能となった。進行制御機構のアイデアは SPIN [5] にあるが、UML/STD との融合は文献 [11] が最初である。この方法によってスケジューリングの例題を作成する過程で、アスペクトの考え方との関連性が生じ [13]、アスペクトの基本機構としての使い方を提案した [12]。本稿ではモデル検査ならびに LTL 検証との関連を議論した。

UML とアスペクトの関連研究はいくつかある。UML の stereotype と呼ぶ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][4]。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。

J. Araujo [1] たちは、シナリオを用いる要求仕様の研究にアスペクトの考え方を導入している。ベース機能ならびにアスペクトと考えるシナリオを準備し、これら複数のシナリオ群から状態遷移モデルを合成する。シナリオは実行履歴であるため JPM そのものと考えることができるが、ポイントカットを基本的な言語要素として持たない。シナリオからの状態遷移モデル合成アルゴリズムに暗に組み込まれている。

E. Katz たち [6] は合成したシナリオを対象とする振舞い検証の方法を提案している。ポイントカットは LTL で表現する。本質的に本稿の方法と同様である。振舞い検証の目的は、紡ぎ合わせ前後で、意味のあるシナリオ断片が分離されないことを確認することである。いわば、シナリオ合成の手法の妥当性を確認することを目的とする。一方、本稿ではアプリケーションから要求される機能を LTL 式として表現して検証することを目的とした。

本稿で提案したアスペクトに関して、いくつかの課題が残っている。第 1 はベースに複数アスペクトを紡ぎ合わせることが可能であるか否かである。紡ぎ合わせが可能なアスペクトに何かの条件が必要と考えている。第 2 は性質検証との関係である。第 3.2 節では簡単な例を用いて、ベースの性質が紡ぎ合わせ後に満たされないことを述べた。LTL 式を用いる方法はアプリケーション要求を表現するものであり、紡ぎ合わせ前後の関係を調べるためには粗いと考えている。紡ぎ合わせ前後の振舞いを細かい関係で調べる方法が必要であろう。たとえば、E.Katz たちの方法がある。また、このような方法は、第 1 の課題として示した紡ぎ合わせ可能な複数アスペクトに対する条件とも関係しそうである。

参考文献

[1] J. Araujo, J. Whittle, and D. Kim. Modeling and Composing Scenario-Based Requirements with Aspects, In *Proc. RE2004*, pages 58–67, September 2004.

[2] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design*. Addison-Wesley 2005.

[3] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real-time Systems*. Wiley 2002.

[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] E. Katz and S. Katz. Verifying Scenario-Based Asect Specifications. In *Proc. FM 2005*, pages 432–447, July 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] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *Proc. ECOOP 2003*, 2003.

[10] 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.

[11] 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.

[12] 中島 震, 玉井 哲雄. アスペクト概念を持つステートダイアグラムの提案. 情報処理学会ソフトウェア工学研究会, July 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, August 2005.

[14] 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.

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