2006－SE－153 (3)
2006／7／10

# Moxaによるアスペクト指向的仕様記述 ：プロトコルからのモジュラーなDbC記述に向けて

橋本 康範 (1), 渡部 卓雄 (1), 山田 聖 (2)
(1) 東京工業大学・大学院情報理工学研究科
(2) 産業技術総合研究所・情報セキュリティ研究センター

我々は現在までに、Javaのための振舞インターフェース仕様記述言語Moxaを開発してきた。MoxaはJava Modeling Language (JML)を拡張したものであり、表明アスペクトと呼ばれる機構を用いて表明間を横断する性質を分離することで、長大で複雑になりがちな実アプリケーションの仕様のモジュール化を可能にしている。本稿ではまずMoxaの設計について簡単に説明した後、その拡張 — プロトコルにもとづくアスペクト記述 — について議論する。この拡張は、メソッド呼び出し系列(プロトコル)にもとづいた仕様の記述・テスト・検証を容易にするためのものである。

# Aspect-Oriented Specification in Moxa : Toward Modular DbC from Protocol Descriptions

Yasunori Hashimoto(1), Takuo Watanabe(1), Kiyoshi Yamada(2)
(1)Graduate School of Information Science and Engineering, Tokyo Institute of Technology
(2)Research Center for Information Security, National Institute of Advanced Industrial Science and Technology

We have been developing Moxa, a behavioral interface specification language tailord for Java. Moxa provides a new modularization mechanism called assertion aspect that can capture the crosscutting properties among assertions in a specification. Using this mechanism, we can modularize large and complex specification of real-world applications. In this paper, we briefly explain the design of Moxa and then discuss its extension called protocol-oriented aspect description. The extension provides convenient way to describe, test and verify specifications that are described based on method invocation sequence.

## 1 はじめに

「契約による設計」(Design By Contract, DbC)の導入は、問題の責任の所在を明らかにし、安全性・保守性を向上させるという意味で有用である。Javaプログラミングにおける DbC を支援するツールとして、Java Modeling Langage (JML) がある。しかし、JMLによる仕様記述を大規模・複雑なプログラムに対して適用する場合、仕様記述自体の複雑化という問題が発生する。

仕様記述言語 Moxa は、JML を拡張し、アスペクト指向的に仕様を記述できるようにすることで、この問題を解決したものである。本稿では、この Moxa の現状を述べ、また、「プロトコル」記述に関する拡張を提案する。プロトコルを記述可能にすることにより、仕様記述の利便性を向上させることを目指すものである。

## 2 表記について

本稿では論理演算子として以下のものを使用する。

- $p \wedge q$ ： 論理積 (p かつ q)
- $p \vee q$ ： 論理和 (p または q)
- $\neg p$ ： 否定 (p ではない)

- p → q : 含意 (p ならば q)

## 3 JML (Java Modeling Language)

プログラミング言語 Java を対象とした仕様記述言語として、JML がある。これは、各メソッドが満たすべき事前条件・事後条件を、"requires" 節・"ensures" 節といった表明の形で記述し、それを利用して検査を行うものである。

### 3.1 JML による記述例

この例は、JML によって IntMath クラスの仕様を記述したものである。IntMath クラスのメソッドに関し、満たすべき事前条件・事後条件を表明として記述している。

```
public class IntMath{
/*@ public behavior
  @  requires i >= 0;
  @  ensures \result >= 0
  @  &&  \result * \result <= i
  @  && (\result+1)*(\result+1)>i;
  @*/
  public int intSqrt(int i);
}
```

JML において、表明は特別なアノテーションの形で記述される。この例では、"/*@" と "@*/" によって囲まれた部分が表明である。ここに含まれる、requires 節・ensures 節によって、それぞれ事前条件・事後条件を表すことができる。なお、表明中に存在する行頭の '@' は、読み易くするためだけに記述されるものである。

こうして記述された表明は、その直後に書かれたメソッドに対して適用される形になる。この場合は、intSqrt(int i) が該当するメソッドである。

上記の例においては、requires に続く "i > 0" が事前条件を表し、ensures に続く "\result>0"、"\result*\result<=i"、及び "(\result+1)+(\result+1)>i" の論理積が事後条件に関する記述となっている。表明においては Java で使用されるものと同様の論理演算子が使えるため、この例における "&&" は論理和を表している。また、\result はそのメソッドの返り値を表すため、結局この表明は、IntMath(int i) メソッドは、int 型の自然数を引数にとり、その正の平方根の小数点以下を切捨てたものを返す、という意味を示している。

### 3.2 表明が示す条件

**事前条件・事後条件**

各メソッドは、requires 節が条件が満たされていなかった場合、実行後に ensures 節の条件を守る責任はない。従って例えば、

```
/*@ public behavior
  @   requires r;
  @   ensures  e;
  @*/
```

(r,e : 論理式)

という記述があった場合、

- 事前条件 : r
- 事後条件 : r → e

という条件を与えられることになる。

## 4 アスペクト指向的仕様記述言語 Moxa

### 4.1 JML の問題点

契約による設計を支援するツールとして、JML の利用価値は高い。しかし、プログラムの大規模化・複雑化により、仕様記述も複雑化してしまう。JML には、こうした複雑化に対処しきれないという問題点が存在する。複雑な仕様記述はそれ自身の理解や変更を困難にし、仕様記述そのものの保守性の低下を導くからである。

山田 [1] は、実際に JML を利用して仕様を記述し、その複雑化の原因として、JML で以下のような記述ができないことを挙げている。

- 一つのメソッドに対する表明を二つ以上に分割して指定する
- 一つのクラスやインターフェイスのための仕様を二つ以上に分割してモジュール化する
- 二つ以上のクラスやインターフェイスの仕様を一つにまとめてモジュール化する

実際に記述する仕様の中には、クラスを横断する性質を持つものが存在する。それに対し JML では、上記の理由により、そうした仕様についてもクラスという単位で記述することになる。そのため、そうした仕様に関する記述はプログラム中に散在するこ

とになり、また、同じクラス中の別の仕様と混在することになる、などといった問題が発生する。

仕様記述言語 Moxa は、JML を拡張し、アスペクト指向的に仕様を記述可能にすることによって、このような問題を解決するものである。

### 4.2 表明アスペクト

Moxa においては、JML と同様の仕様記述方法に加え、表明アスペクトという単位で仕様を記述することが可能になっている。表明アスペクトとは、クラス間を横断するような性質をもつ仕様に着目し、モジュール化したものである。この表明アスペクトの内容は、処理系によって実際に適用される先のコードに挿入される形になる。

表明アスペクトによる仕様記述と JML による仕様記述の対応の一例を以下に示す。r と e はそれぞれ論理式とする。

```
public class C1{
  public void m1(int i);
}

spec A1{
  /*@ public behavior
    @   requires r;
    @   ensures  e;
    @*/
  void C1.m1(int i);
}
```

表明アスペクトを利用した仕様記述

```
public class C1{
  /*@ public behavior
    @   requires r;
    @   eusnres  e;
    @*/
  public void m1(int i);
}
```

JML による仕様記述

この例における、"spec A1{...}" が表明アスペクトである。"spec" は "class" などと同様のもので、表明アスペクトの宣言であることを示すキーワードである。

表明アスペクトは「アドバイス」の組によって構成される。アドバイスとは、挿入するべき表明と、その表明を挿入する位置の指定の組のことを言う。また、挿入する位置の指定のことを「ポイントカット」と呼ぶ。

この例においては、表明アスペクトは一つのアドバイスから構成され、そこに含まれるポイントカットで C1 クラスの m1(int i) メソッドが指定されている。なお、この例においてポイントカットは一箇所の挿入位置の指定だが、これは複数箇所設定することが可能である。

### 4.3 表明の織り込み

Moxa において、メソッドの事前条件・事後条件は、複数のモジュールに分割して記述される可能性がある。このような場合、そのメソッドの満たす条件は、それらの条件の論理和となる。

例えば、

```
/*@ public behavior
  @   requires r1;
  @   ensures  e1;
  @*/
```

```
/*@ public behavior
  @   requires r2;
  @   ensures  e2;
  @*/
```

(r1,r2,e1,e2：論理式)

という二つの表明が適用されることになるメソッドが求められる条件は、

- 事前条件：r1 ∧ r2
- 事後条件：(r1 → e1) ∧ (r2 → e2)

となる。

なお、事前条件が成り立たないときの事後条件については考えなくて良いので、この事後条件は

$$(r1 \wedge r2) \rightarrow ((r1 \rightarrow e1) \wedge (r2 \rightarrow e2))$$

とすることができ、これは、$(r1 \wedge r2) \rightarrow (e1 \wedge e2)$ と等価である。よって、上記二つの表明の適用結果は、

```
/*@ public behavior
  @   requires r1 && r2;
  @   ensures  e1 && e2;
  @*/
```

と等価なものとなる。

### 4.4 Moxa による表明記述

表明アスペクトの利用によって仕様記述の複雑さを解消できるものとしては、次のようなものが考えられる。

横断的仕様が存在する場合の仕様記述

以下の仕様記述が JML でなされている場合を考える。r1、ra11 などは全て論理式だとする。また、ra11、ra21 などは、ある側面に関する仕様であり、クラス間を横断するようなものを表す。

```
public class C1{

  /*@ public behavior
    @   requires r1
    @          && ra11
    @          && ra12;
             :
    @*/
  public void m1();

  /*@ public behavior
    @   requires ra2;
             :
    @*/
  public void m2();

:

}
```

```
public class C2{

  /*@ public behavior
    @   requires ra3;
   :
    @*/
  public void m3();

:

}
```

これを Moxa で記述した場合、

```
public class C1{

  /*@ public behavior
    @   requires r1;
             :
    @*/
  public void m1();

  /*@ public behavior
             :
    @*/
  public void m2();

:

}
```

```
public class C2{

  /*@ public behavior
   :
    @*/
  public void m3();

:

}
```

```
spec A{

  /*@ public behavior
    @   requires ra11
    @          && ra12;
    @*/
  void C1.m1();

  /*@ public behavior
    @   requires ra2;
    @*/
  void C1.m2();

  /*@ public behavior
    @   requires ra3;
    @*/
  void C2.m3();

}
```

と書ける。ここで、例えば ra2 と ra3 が同じものだった場合は、表明アスペクトがはもっと簡潔に、

```
spec A{

  /*@ public behavior
    @   requires ra11
    @          && ra12;
    @*/
  void C1.m1();

  /*@ public behavior
    @   requires ra2;
    @*/
  void C1.m2();
  void C2.m3();

}
```

となる。

Moxa による記述の効果

この例からも分かる通り、多くの場合、Moxa による記述は行数を減らすことには貢献しない。Moxa による記述の利点としては、以下のことが挙げられる。

- 個々のモジュールの記述が明解
- 変更が複数のモジュールに及ぶことが少ない

これらの利点は、複数のクラスにまたがる仕様を一つのモジュールにまとめることにより得られている。

例として、横断的な仕様記述の一部を変更する際のことを考える。こうしたとき、その変更は同じ側面についての仕様に及ぶことが多い。よって、表明アスペクト A 内部のどこかに影響を与えるような変更があった場合、その変更は A 内部の別のものに対しても変更を与える可能性が高い。この変更を一貫性を保ちつつ行うためには、Moxa の場合は通常一つの表明アスペクトのみを見て行うことができる。しかし、JML の場合、プログラム全体から変更が及ぶ箇所を探す必要があり、大変手間がかかる。

このようなことから、Moxa による仕様記述では行数が増えることは多いが、適切なモジュール化により、保守性が高まることが分かる。

### 4.5 処理系

Moxa の処理系については、直接実行時検査をするためのコードを生成するようなものではなく、JML へのトランスレータとしての実装を行った。これは、JML の処理系として存在する、様々なツールを利用するためである。

## 5 状態遷移モデルと仕様記述

本節では、メソッド呼出の順番を表すために以下の表記を使用する。

- `M1 -> M2` : `M1` の終了後に `M2` の実行
- `(M)*` : M の 0 回以上の繰り返し
- `(M)+` : M の 1 回以上の繰り返し
- *method-name*`()` : メソッド呼出

### 5.1 表明アスペクトからの状態遷移モデルの抽出

Moxa では、表明アスペクトを利用し、ある側面に関する仕様の情報をまとめている。複数のクラスにまたがる記述をまとめることによって扱いやすくすることが目的であるが、これによって同時に関連の深いメソッドとそれらの表明が分かる記述になっている。

こうした情報を利用し、表明アスペクトから状態遷移モデルを抽出する方法が山田ら [2] によってが提案されている。手順は以下の通りである。

■状態の抽出　表明アスペクト中の事前条件の記述から、状態遷移モデルの状態集合を決定する。事前条件ー記述は、オブジェクトの内部状態を取り出すメソッド呼び出しと、その値に対する条件の記述という形で記述してあると仮定し、この事前条件の記述と内部状態を取り出すメソッドに指定された事後条件とを利用して、オブジェクトの内部状態を状態集合へと分割する。

■遷移の検出　表明アスペクトの記述に含まれるメソッドに対し、それぞれの事前・事後条件により選択される状態を遷移元・遷移先とする遷移を状態遷移モデルに追加する。事後条件が選択する状態が、複数の状態にまたがる場合は、それぞれの状態を遷移先とする遷移を追加する。

これによって、メソッド呼出によって状態を遷移させる状態遷移モデルが得られ場合がある。そのような時は、これを利用し、表明の記述の整合性の検査等を行うことが可能である。

### 5.2 JML の拡張としてのプロトコル記述

一方で、メソッド呼出のシーケンスそのものを仕様として記述する方法が、JML の拡張として、Yoonsik ら [6] によって実装されている。この拡張は、メソッド呼出のシーケンスを規定するための表明として

"call_sequence" という表明を記述可能にしたものである。

このシーケンスの規定は「プロトコル」と呼ばれ、例えばアプレットの動作については以下のようなプロトコルが存在する。

```
init()
  -> ( start() -> stop() )+
  -> destroy()
```

これは、以下のような記述によって定義される。

```
public class Applet{
/*@ public call_seauence
  @  init()
  @  : (start() : stop())+
  @  : destroy();
  @*/
       :
}
```

こうした定義は、元々の JML の機能のみで行うことは可能ではある。しかし、複雑なシーケンスを事前条件・事後条件の形に手で正確に変換するのは困難である。また、単純なものであったとしても、複数のメソッドに対する記述に変更を加える必要があり、手間がかかる。

プロトコル記述の利点は、その書き易さにある。特にその内容に変更を加えるときを考えれば、call_sequence の表明だけの変更で済むことになる。こうした意味で、プロトコルを直接記述できるようにすることは有用だと言える。

### 5.3 複雑なプロトコルの分割記述

プログラムの大規模化・複雑化に際し、複雑なプロトコルを記述する必要性が生じる可能性がある。だが、それを直接扱うのが困難なこともある。

**プロトコルの分割定義**

複雑なプロトコルを扱う方法として、Yoonsik[6] は、各部分を分けて定義することを提案している。例えば、

```
s1() -> a1() -> a2() -> s2()
```

というプロトコルを定義したい場合、

```
A : a1() -> a2()
```

などと記述できるようにし、この A によってシーケンスの一部を表現することができるようになれば、

```
s1() -> A -> s2()
```

と記述できるようになり、これを用いることによって記述の複雑化を避けられる。

しかし、これだけでは必ずしも問題の解決にはならない。なぜなら、この方法で解決される複雑さは表記上のものだけだからである。このような記述の場合、全体を見なければ、結局その記述が何を表しているのかを把握することが不可能である。したがって、理解や変更の容易さは得られていない。

**プロトコルの側面による分割**

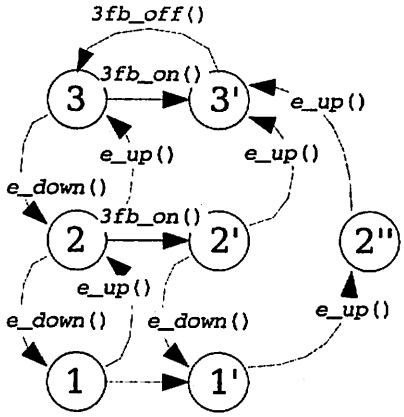
こうした複雑化の問題は、JML のみによる事前条件・事後条件の仕様記述が直面した問題と同質のものである。したがって、Moxa のモジュール化機構を使ってプロトコルを分割することにより、こうした問題の解決が可能だと考えられる。

例えば、エレベーターの仕様について、プロトコルの形で記述することを考える。ここでは、三階建ての建物に対し、一基のエレベーターが存在する状況とする。このような単純な状況でも、状態数が非常に多くなることから、仕様を正確に反映した状態遷移図を作るのは困難である。

これに対し、仕様の側面ごとに分割して記述することを考える。例として、「三階のボタンが押下された際、エレベーターが問題なく動作する」ことに関する仕様は、例えば図 1 に表す状態遷移図のような定義が可能である。こうした形で各仕様に関するプロトコルの定義をし、それらの合成として、全体のプロトコルを定義することにする。

このような方法で分割することの利点は、理解や変更が容易であることである。この記述においては、プロトコルは独立した仕様ごとに分割されているため、それぞれを分けて考え、理解することが可能であり、また仕様を変更する際に関しても、変更が他の部分に及ぶのを防げる。

Moxa に対してこの拡張を行う場合、JML に対す




(e_up(),e_down()：エレベーターの上昇・下降)
(3fb_on(),3fb_off()：3階のボタンのオン・オフ)

図1　三階のボタンの動作に関する仕様

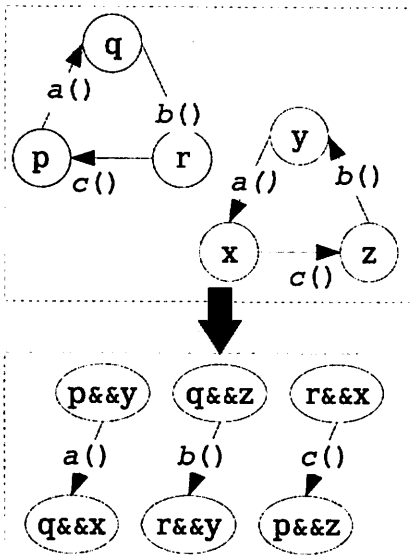



図2　シーケンスが途切れる例

る拡張との差としては、表明アスペクトに記述する場合は適用するクラスを明記しなければならない点である。だが、それ以外は基本的に同様の方法で拡張可能である。

### 5.4　分割したプロトコルからの検査

こうして分割したプロトコルに対し、次のような検査を行う。

- 表明アスペクトから抽出したモデルとの比較
- 合成によって生じる途切れたシーケンスの検出
- 合成結果のプロトコルの検証

**表明アスペクトから抽出したモデルとの比較**

プロトコルを仕様の側面によって分割した場合、そのプロトコルが示す状態遷移モデルは、それが含まれる表明アスペクトから抽出されたモデルと関係の深いものになるはずである。よって、これらのモデルに矛盾がないことの確認を含め、比較を行うことで、有用な検査ができると考えられる。

**合成によって生じる途切れたシーケンスの検出**

プロトコルを分割して記述した場合、各々には矛盾がなくても、全体として見たときに、途切れたシーケンスが生成されている可能性がある。図2に例を示す。こうしたものを意図してプロトコルを定義することは考えられないため、このようなシーケンスの検出を行う必要がある。

**合成結果のプロトコルの検証**

これは、意味を変えずにプロトコルの分割を行うことが困難なためである。利用者が分割を上手に行ったつもりでも、その合成結果が利用者の期待通り、あるいはそれに近いものになる保証はない。したがって、合成結果を示し、それを利用者が確認できるような機能が求められる。

## 6　今後の課題

**プロトコルによる記述の実装**

今回提案した、Moxaに対するプロトコルによる仕様記述の実装が求められる。プロトコルを記述するための構文を確定し、実際に使用して効果を検証すべきと考える。

**Moxaによる仕様記述の具体例**

Moxaは大規模で複雑なプログラムに対処するための仕様記述言語である。その特性上、Moxaの適用の効果を示す、具体的な例が簡単には作れない。多くの複雑で規模の大きいプログラムに対して仕様を記述し、問題点や拡張に対しての考察を進める必要がある。

**クラス間を横断するプロトコル**

エレベーターの例で考えた場合、各階のボタンとエレベータは別のクラスとして実装されることが考

えられる。このような場合、プロトコルはクラスを横断するものとなる。こうしたプロトコルを、どのように実装すれば良いかを考えなければならない。

**受理状態・非受理状態**

現在考えている方法では、すべてを受理状態として記述することになる。だが実際は、途中状態の一部を非受理状態としたい場合があると考えられる。例えば、

```
(start() -> finish())*
```

というシーケンスがあったとする。このとき、`start()` した場合は必ず `finish()` の操作を実行したい、という状況が考えられる。つまり、`start()` したのに `finish()` することなくプログラムが終了するようなものを検出する必要がある。こうした機構の実装方法が求められる。

## 7 まとめ

大規模で複雑なプログラムの開発において、DbCの導入は保守性・安全性を高めるという意味で有用である。特にJavaプログラミングに関しては、JMLが持つ様々な機能は魅力的である。ただし、その記述の複雑化という問題があった。

MoxaはJMLにおける記述の複雑化の原因となる横断的仕様に着目し、それを扱うための表明アスペクトというモジュール単位を導入することにより、この問題の解決方法を与えている。今回はこれに加え、プロトコル記述を表明アスペクトによって行い、その複雑に対処することを提案した。今後はMoxaそのものの考察に加え、こうした拡張の実装についても考えていく予定である。

## 参考文献


[1] 山田聖, 渡部卓雄, 契約による設計を支援するアスペクト指向的振舞インターフェース記述言語Moxa 情報処理学会論文誌 (プログラミング), Vol. 46, No. SIG 11 (PRO 26), pp. 27-44, Aug., 2005. (2005)

[2] Kiyoshi Yamada and Takuo Watanabe, An Aspect-Oriented Approach to Modular Behavioral Specifications, 1st International Workshop on Aspect-Based and Model-Based Separation of Concerns in Software Systems (ABMB 2005), Nov., 2005.

[3] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby,David Cok, Peter Müller, and Joseph Kiniry.: "JML Reference Manual", ftp://ftp.cs.iastate.edu/pub/leavens/JML/jml refman.pdf (2005).

[4] Gary T. Leavens, Albert L. Baker, and Clyde Ruby.: "JML:A Notation for Detailed Design", Behavioral Specifications for Businesses and Systems, chapter 12, pp. 175-188 (1999).

[5] Gary T. Leavens and yoonsik Cheon.: "Design by Contract with JML", ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf (2005).

[6] Yoonsik Cheon and Ashaveena Perumandla.: "Specifying and Checking Method Call Sequences in JML", Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP '05), pp. 511-516 (2005).
(extended version : http://www.cs.utep.edu/~cheon/techreport/tr05-36.pdf)