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フェース記述言語 Moxa

山田 聖[†,☆] 渡部 卓雄[††]

我々は，契約による設計（Design by Contract，DbC）に基づく，アスペクト指向振舞インタフェース仕様記述言語 Moxa の設計・開発を行っている．DbC は，サービスの提供者と利用者の間で，利用者が満たすべき条件（事前条件）と提供者がもたらす結果（事後条件）を契約として取り決め，責任の切り分けやサービスの利用方法と得られる結果を明確にすることで，ソフトウェアの品質を向上させる．Java のための DbC に基づく振舞インタフェース仕様記述言語の 1 つである JML（Java Modeling Language）は，メソッドごとに事前条件・事後条件の記述を可能にするが，メソッド数の増加や提供するサービスの高機能化がこれらの条件を複雑なものにし，表明の整合性・表明とプログラムの一貫性を維持しつつ，これらを修正・拡張することを難しくする．Moxa は，複雑なオブジェクトの振舞いをいくつかの独立した側面の合成としてとらえることができる場合に，それらを表明アスペクトと呼ばれる独立したモジュールに分割して記述することを可能にする．この，オブジェクトの振舞いのある側面は，JML による記述では複数のメソッドに対する表明を横断する形で存在していたものである．本論文では，Moxa が提供する表明記述のモジュール化機構と，その記述方式の説明を行う．

Moxa: An Aspect-oriented Behavioral Interface Specification Language

Kiyoshi Yamada[†,☆] and Takuo Watanabe[††]

In this paper, we report the design and implementation of Moxa — a behavioral interface specification language that supports DbC (Design by Contract) based specifications. In DbC methodology, a contract arranges responsibilities between a user and a supplier of the service. This improves software quality because this makes clear who has responsibility, how to use, and what is the result of the service. With JML (Java Modeling Language), a DbC based behavioral interface specification language for Java, we can declare assertions as pre/post-conditions for each method. However, as the number of methods increases and the functionalities of services become complex, assertions for each method gets complicated. This makes it difficult to keep the consistency between assertions and methods while we are modifying them. Using Moxa, we can divide and describe assertions into several modules called assertion-aspects, when we can regard the behavior of the object(s) as the composition of some independent aspects. In JML style specification declaration, the aspects are crosscut over the assertions declared for difference methods. In this paper, we show the modularization mechanism for assertion declarations and description method provided by Moxa.

1. はじめに

我々はこれまでに AnZenMail システムの設計・開発[6] に携わり，AnZenMail クライアントの開発を行った．この中で，Maildir フォルダサービスプロバイダ（以降 Maildir プロバイダ）と呼ばれる電子メイルメッセージ（以降メッセージ）を扱うライブラリの実装を行い，さらにそのライブラリの信頼性を高めるために JML（Java Modeling Language）[3] を利用してこの実装の検査を行った[7]．この検査は，JML を用いて契約による設計（Design by Contract，DbC）[5] に基づ

† 北陸先端科学技術大学院大学情報科学研究科
School of Information Science, Japan Advanced Institute of Science and Technology
†† 東京工業大学情報理工学研究科計算工学専攻
Department of Computer Science, Tokyo Institute of Technology
☆ 現在，独立行政法人産業技術総合研究所情報セキュリティ研究センター
Presently with Research Center for Information Security, National Institute of Advanced Industrial Science and Technology

いた表明として Maildir プロバイダの仕様を記述し，これを実装に強制したものを実行することで仕様に反する振舞いを検出し，実装の誤りを発見するものである．我々はこの作業の過程で，プログラムが複雑化・大規模化するにつれ，表明の整合性や表明とコードとの間の一貫性を維持しつつ，それらの修正や変更を行うことが困難になっていくという問題に直面した．この問題は直接的には表明数の増加と個々の表明の複雑化に起因するが，その原因はプログラムコードと表明のモジュール化の単位が一致しないこと，具体的には，いくつかの表明に共通に表れる性質がプログラムの持つ自然な構造を横断するような現象にあると我々は考える．

そこで我々は，アスペクト指向に基づいた表明のモジュール化機構と，その機構を提供する仕様記述言語 Moxa を提案する．このモジュール化機構は動的ジョインポイントモデルに従い，表明アスペクトと呼ばれる単位で，プログラムの構造を横断する性質のモジュール化を可能にする．このモデルでは，DbC に基づく表明の条件の成立を仮定することのできる制御流上のある時点をジョインポイントとし，それらの集合をポイントカット，ポイントカットとそこで成立する表明の条件をアドバイスとし，それらの集合をアスペクトとする．本機構により，プログラムの構造から独立した表明のモジュール化およびプログラムの大規模化にともなう表明の大規模化・複雑化の抑制が可能となる．Moxa は JML の言語仕様を拡張し，JML が提供する仕様記述形式に加え，アスペクト指向に基づいた表明のモジュール化を実現するための構文を持つ．

以降の章は，次のように構成される．次章では，DbC と，Java のための DbC に基づく振舞インタフェース仕様記述言語である JML の説明を行い，DbC に基づく仕様記述における問題点を明らかにする．続く 3 章では，DbC に基づき記述された仕様がプログラムの構造を横断する性質を持つ場合があることを述べ，アスペクト指向を導入することによりこの構造を自然にモジュール化する方法について述べる．4 章では，3 章で述べたモジュール化機構を提供するアスペクト指向振舞インタフェース仕様記述言語 Moxa の説明を行う．5 章では，Moxa と JML のそれぞれについて，Maildir プロバイダに対する仕様の記述の結果を通して比較する．6 章では，関連研究と今後の課題について述べる．7 章でまとめを述べる．

2. 契約による設計に基づく仕様の記述

2.1 契約による設計

契約による設計（Design by Contract，DbC）[5] は，あるサービスの提供者と利用者の間の契約に基づきソフトウェアを構成する手法である．この手法では，サービスの提供者は利用者に対して，サービスを提供することのできる状態を表す条件（事前条件，precondition）と，そのサービスの提供後に成立する条件（事後条件，postcondition）を提示する．サービスの利用者は，事前条件を満たすような状態を構成すること，提供者は事後条件を満たすようなサービスを提供することをそれぞれの責任とし，双方がこれらの責任を全うすることが契約となる．この手法の利点は，事前条件が満たされない場合はサービスの利用者側に，事後条件の場合は提供者側に問題があることが分かり，問題に対して明確に責任の切り分けができることにある．それに加え，サービスの利用者はサービスの提供者に対し，事前条件を満たしている限りその実現方法を考慮することなく事後条件を満たすような結果が得られることを期待することができることから，モジュールの独立性の向上につながる．

DbC で利用される事前条件および事後条件の記述には，表明（assertion）が利用される．表明とは，制御流上のある時点で，プログラムが満たすべき条件である．あるプログラムに対する仮定を表明の集合として表し，この仮定が満たされない場合（表明違反，assertion failure）を探すことで，そのソフトウェアの誤りを発見することができる．あるプログラムに対して，より多くの表明を指定することで，そのプログラムが正しく動作することをより確実なものとすることができる．

C や C++，Java といったプログラミング言語は，表明を文として記述するための構文を持ち，これを用いることで表明をプログラムテキスト中に埋め込むことができる．これらの言語では，埋め込まれた表明は動的に検査される．プログラムの実行が表明の埋め込まれた時点に到達した場合，表明として指定された条件が検査される．表明の条件が成立する場合にのみ計算が進み，条件が不成立の場合はただちにプログラムの実行を停止する．この動的な表明の検査は，実行時プログラムの中に表明の検査のためのコードを組み込むことで実現される．この表明の検査のためのコードの生成と実際の検査は開発中のソフトウェアに対し行われ，十分な検査が行われた後，完成版のソフトウェアからは取り除かれる．

```
1  public class Foo {
2   /*@ public behavior
3     @  requires P1;
4     @  ensures Q1;
5     @  signals (Exception1 e) R1(e);
6     @ also public behavior
7     @  requires P2;
8     @  ensures Q2;
9     @  signals (Exception2 e) R2(e);
10    @*/
11    public void foo() throws ExceptionX {...}
12 }
13
14 class Bar extends Foo {
15  /*@ also public behavior
16    @  requires P3;
17    @  ensures Q3;
18    @  signals (Exception3 e) R3(e);
19    @*/
20    public void foo() throws ExceptionY {...}
21 }
```

図 1 JML における表明記述例
Fig. 1 An example of assertion declarations in JML.

DbC に基づく表明の指定可能なプログラムテキスト上の位置は，関数やメソッドの事前条件・事後条件を検査する時点に対応する位置，つまり関数やメソッドの呼び出し時および復帰時に限られる．それ以外の位置に指定された表明は，DbC の対象とはならない．

2.2 Java Modeling Language

JML（Java Modeling Language）[3)] は，Java に対し DbC に基づく表明記述のための拡張を導入する言語の 1 つである．JML を用いた表明記述の例を**図 1** に示す．JML の記述は，Java のプログラムコード中のアノテーションと呼ばれる "@" をともなったコメント中に記述される．たとえば 2–10 行目にあるアノテーションでは，直後で定義されるメソッド `void Foo.foo()` の事前条件・事後条件・例外時事後条件を記述している．キーワード `requires`，`ensures` に続く論理式がそれぞれ事前条件・事後条件の記述である．たとえば 3 行目は事前条件 P_1，4 行目は事後条件 Q_1 の記述である．また，5 行目にあるように `signals` に続く例外の型と論理式が例外時事後条件の記述である．ここでは，送出される例外の型として `Exception1` を，例外時事後条件として $R_1(e)$ を指定している．これは，送出された例外を e とすると，

$$((e\ \texttt{instanceof Exception1}) \supset R_1(e)$$

という条件を持つ例外時事後条件の記述となっている．

JML を利用して事前条件・事後条件・例外時事後条件を記述した場合，実際の事後条件・例外時事後条件は事前条件が成立することを前提としたものとなる．つまり JML では，事前条件 P，事後条件 Q と記述した場合，実際の事後条件は，

$$P \supset Q$$

となる．また，事前条件 P，例外時事後条件が例外の型 `Exception1` と条件 $R_1(e)$ として記述された場合，実際の例外時事後条件は，送出された例外の型を e とすると，

$$P \supset ((e\ \texttt{instanceof Exception1}) \supset R(e))$$

となる．

JML では，これらの条件を `also` を用いて分割して記述することができる．この分割は，クラスの継承によりオーバライドされるメソッドと，するメソッドの間でも有効である．`also` によって分割して記述された事前条件 P_i，事後条件 Q_i の意味は，事前条件が $\vee_i P_i$，事後条件が $\wedge_i Q_i$ となる．つまり，図 1 におけるメソッド `void Foo.foo()` の事前条件・事後条件はそれぞれ，

$$P_1 \vee P_2,$$
$$(P_1 \supset Q_1) \wedge (P_2 \supset Q_2)$$

となり，また，メソッド `void Bar.foo()` では，それぞれ

$$P_1 \vee P_2 \vee P_3,$$
$$(P_1 \supset Q_1) \wedge (P_2 \supset Q_2) \wedge (P_3 \supset Q_3)$$

となる．ここで，事後条件は事前条件が成立することを前提とする点に注意が必要である．また，例外時事後条件の意味は事後条件の場合と同様に，送出された例外を e とすると，メソッド `void Foo.foo()` は，

$$(P_1 \supset ((e\ \texttt{instanceof Exception1}) \supset R_1(e))$$
$$\wedge\ (P_2 \supset ((e\ \texttt{instanceof Exception2}) \supset R_2(e))$$

となり，メソッド `void Bar.foo()` は，

$$(P_1 \supset ((e\ \texttt{instanceof Exception1}) \supset R_1(e))$$
$$\wedge\ (P_2 \supset ((e\ \texttt{instanceof Exception2}) \supset R_2(e))$$
$$\wedge\ (P_3 \supset ((e\ \texttt{instanceof Exception3}) \supset R_3(e))$$

となる．

事前・事後条件には，Java の式を拡張した論理式を書くことができる．事後条件・例外時事後条件の中で利用できる `\old(<式>)` はメソッド実行前の<式>の値を表し，事後条件中に記述できる `\result` は，メソッドの返値を表す．その他，限量子 `\forall`，`\exists` 等を利用できる．事前・事後条件中に Java のメソッド呼び出しを書くこともできるが，呼び出すことができるメソッドは，`pure` 修飾子によって副作用を持たないことを宣言されたものに限られる．

JML で記述された表明の記述は，JML コンパイラを利用することで，実行時に表明検査を行うコードが埋め込まれたクラスファイルへ変換される．このクラスの実行は，JML により表明として指定された仕様

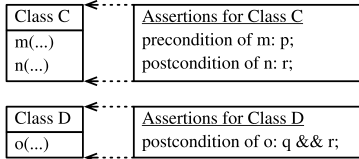

図 2　DbC に基づく表明の記述
Fig. 2　An image of DbC based assertion declarations.

が強制されたものとなる．

2.3　DbC による仕様記述の問題点

JML を利用して，Java のクラスやインタフェースに対する仕様を DbC に基づく表明として記述する場合，クラスやインタフェースが大規模で複雑なものになるにつれ，表明の記述も大規模化・複雑化する．この表明の記述の複雑化は，具体的には次のような形で現れる．

- 表明の条件を表す論理式が複雑化する．
- 1 つのクラスが持つメソッド数の増加にともない，表明の記述の数が増加する．

このため，表明の記述やそれに対応するメソッドの実装を修正した場合の影響が思いがけない広範囲な領域にわたり，表明の記述の一貫性や表明の記述とメソッドの実装との間の整合性を保ちつつ，それぞれを修正・改良していく作業が困難となる．この問題の原因は，JML がインタフェース振舞仕様のための表明の記述を適切にモジュール化する機構を提供していないことにある．JML を利用してクラスやインタフェースに対して仕様を記述する場合，たとえば**図 2** にあるように，メソッドごとに表明を 1 つずつ指定する．それらは，そのメソッドの属するクラスやインタフェースを単位としてモジュール化される．したがって，以下のような形の仕様を JML で記述することはできない．

- 1 つのメソッドに対する表明を 2 つ以上に分割して別々に指定する．
- 1 つのクラスやインタフェースのための仕様を 2 つ以上に分割してモジュール化する．
- 2 つ以上のクラスやインタフェースの仕様を 1 つにまとめてモジュール化する．

そのため，たとえばクラスやインタフェースの仕様がいくつかの小さな仕様の合成として表現できるような場合であっても，個々の小さな仕様ごとに別々の表明の記述のモジュールを作成し，クラスやインタフェースの仕様をこれらの合成として指定することができない．具体的には，**図 3** では，クラス Class A_m がクラス Class A_{1m}，Class A_{2m} の集約となる関係にあるようなモデルに対し，これらを 1 つのクラス Class A_i として実装とした場合を表している．JML では，モデルにおけるクラス Class A_{1m}，Class A_{2m} それぞれの仕様を JML Spec. A_{1m}，JML Spec. A_{2m} として記述した場合，実装 Class A_i に対する仕様をこれらの合成として表現することができず，これらの仕様を元に合成後の仕様 JML Spec. A_i を作成し，記述しなければならない．

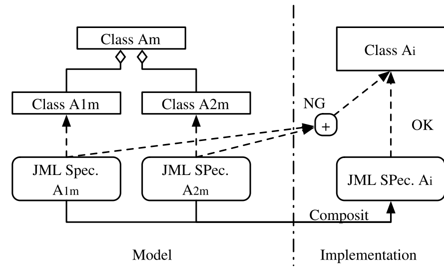

図 3　JML を利用した場合の仕様の分割記述ができない例
Fig. 3　An image of JML specification that can not be divided into some submodules.

このように，JML は仕様のモジュール化を自由に行えないことが個々の仕様を複雑化・大規模化させ，その結果，仕様やプログラムの修正・改良を難しくさせている．この問題は JML に特有のものではなく，DbC に基づきクラスやインタフェースの仕様をそれらの持つメソッドに対する表明として記述する場合に共通する問題点である．

3.　仕様記述へのアスペクト指向の適用

3.1　DbC に基づく仕様記述に現れる横断的側面

2.3 節で述べた，大規模なクラスやインタフェースに対して DbC に基づき表明を記述する場合の問題点に対処するために，大規模な仕様をいくつかの小さなモジュールに分割して記述することで，複雑度を緩和することを考える．大規模なプログラムに対しても無理なく仕様を与えられるようにするためには，仕様の記述を，対象となるプログラムの構造から独立した単位でモジュール化するための機構が必要となる．DbC に基づく表明の記述を分割するための手がかりとして，以下に示すような，仕様の記述が持つ特徴を利用する．

振舞いの多面性

仕様記述の対象であるクラスやインタフェースの振舞いは，いくつかの独立した側面として別々にとらえることができる場合がある．このクラスやインタフェースの振舞いは，これらの側面の合成としてとらえることができる．たとえば，2.3 節の図 3 に示した例では，実装におけるクラス Class A_i の振舞いは，モデルにおけるクラス Class A_{1m} と Class A_{2m} のそれぞれ

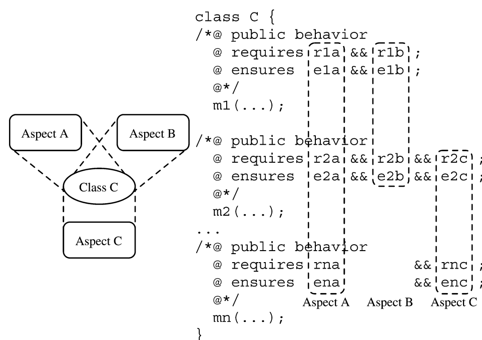

図 4 JML の記述に現れる横断的側面
Fig. 4 Crosscutting concerns seen in a JML specification.

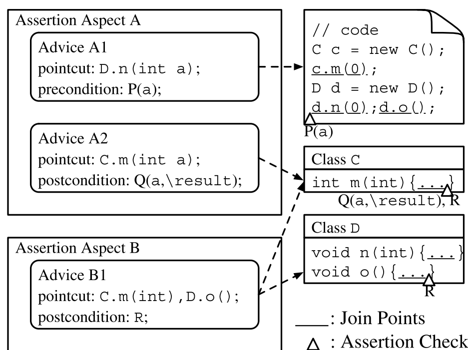

図 5 表明アスペクトを用いた表明のモジュール化の例
Fig. 5 An example of modularization of assertions based on aspect-orientation.

に対応する振舞いを合成したものとなる．一方，**図 4** では，クラス Class C の振舞いが，側面 Aspect A, B, C の合成としてとらえることができる場合を表す．

横断的側面の存在

クラスやインタフェースの振舞いのそれぞれの側面に対する表明の記述は，クラスやインタフェースが持つメソッドに指定される個々の表明の記述を横断する．図 4 では，クラス Class C に属する振舞いの側面 Aspect A がクラス Class C のメソッド m1(...), m2(...), ..., mn(...) の表明の論理式 r1a, e1a, r2a, e2a, ..., rna, ena として表され，同様に側面 Aspect B がメソッド m1(...), m2(...) の表明，側面 Aspect C が m2(...), ..., mn(...) の表明を横断している場合示す．

横断的側面と表明の記述の関係

あるメソッドに指定される表明の条件は，そのメソッドが属するクラスやインタフェースの振舞いの個々の側面に関する条件を表す論理式を論理積で結合したものとなる．図 4 では，クラス Class C のメソッド m1(...) の振舞いが，側面 Aspect A に関する条件 r1a，e1a，Aspect B に関する条件 r1b，r2b のそれぞれを論理積で結んだ形をとる．また，その他のメソッド m2(...), ..., mn(...) に関する表明の条件も同じような型式となっている．

3.2 DbC に基づく仕様記述のアスペクト指向的なモジュール化機構

3.1 節で述べたように，DbC に基づく表明の記述中には横断的側面が含まれている場合があり，それらを独立したモジュールとして表現できるようにするために，アスペクト指向を導入することは自然である．そこで，我々は，AspectJ で用いられている動的ジョインポイントモデルを利用し，これら DbC に基づく表明の記述における横断的側面をアスペクト化する機構を提案する．

DbC に基づきあるメソッドに表明を指定する場合，事前条件・事後条件の成立を仮定する制御流上の時点として，メソッドを呼び出す側と呼ばれる側の 2 つの時点が考えられる．本機構では，それぞれの条件の成立に責任を持つ側，つまり事前条件は呼び出す側，事後条件は呼ばれる側での条件の成立を仮定する．

この機構では，動的ジョインポイントモデルにおける，ジョインポイント，ポイントカット，アドバイス，アスペクトのそれぞれを以下のように定義する．

ジョインポイント

DbC に基づく表明の条件の成立を仮定することのできる，制御流上のある時点☆を表す．本機構では，クラスやインタフェースで定義されるメソッド（またはコンストラクタ）の呼び出し時点およびメソッド（またはコンストラクタ）本体の実行時点をジョインポイントとする．これらは，それぞれ AspectJ における Method/Constructor Call Join Points, Method/Constructor Execution Join Points に対応する．メソッドの呼び出し時点は事前条件の検査，メソッド本体の実行時点は事後条件の検査のために利用する．たとえば **図 5** では，下線で表されている，クラス Class `C` に属するメソッド `m` およびクラス Class `D` に属するメソッド `n`，`o` の本体の実行時点と，それらを利用するコード `code` 中にあるそれらの呼び出し時点がジョインポイントとなる．

ポイントカット

アドバイスが横断する範囲をジョインポイントの集合として選択，選択されたジョインポイントにおける

☆ ここでは時点と呼ぶが実際は制御流上の区間を表す．「メソッドの呼び出し時点」はメソッドの呼び出しから結果を得るまで，「メソッド本体の実行時点」はメソッド本体の実行開始から終了までの区間を表す．これらは，ジョインポイントモデルにおいてこれ以上分割できない区間であることから，これらを制御流上の点として扱う．

プログラムの状態の参照を行う．図 5 では，アドバイス Advice A1 において，“pointcut” の指定から，クラス Class `D` に属するメソッド `n` の呼び出し（コード `code` における `c.m(0)` によるメソッド呼び出し）時点およびそのメソッド本体の実行時点をまず選択し，さらにこのアドバイスが “precondition” の指定であることから，それらのうちのメソッドの呼び出し時点を採用する．また，選択されたジョインポイントにおけるメソッドの引数を変数 `a` として参照している．同様に，アドバイス Advice A2 で，クラス Class `C` に属するメソッド `m` の実行時を選択し，引数を変数 `a` に，返値を `\result` として参照しており，さらに，アドバイス Advice B1 で，クラス Class `C` に属するメソッド `m` とクラス Class `D` に属するメソッド `o` の実行時を選択している．

アドバイス

ポイントカットと条件の組であり，ポイントカットによって選択された時点で成り立つべき条件を定義する．アドバイスには事前条件アドバイス，事後条件アドバイスの 2 つの種類がある（ここでは例外発生時における事後条件アドバイスは事後条件アドバイスの一種と考える）．事前条件アドバイスとして指定された条件は，そのアドバイスと組で指定されるポイントカットが選択する全ジョインポイントの直前で成立することが仮定され，事後条件アドバイスとして指定された条件は，ジョインポイントの直後で成立することをが仮定される．条件にはポイントカットで選択した時点のプログラムの状態の参照を利用できる．図 5 では，Advice A1 が “precondition” の指定を持つことから事前条件アドバイスであり，クラス Class `D` に属するメソッド `n` の呼び出し時点直前で条件 P(`a`) の成立が仮定されることを表している．同様に，Advice A2 は “postcondition” の指定を持つことから事後条件アドバイスであり，クラス Class `C` に属するメソッド `m` の本体の実行時点直後で条件 Q(`a`, `\result`) の成立を仮定している．さらに，事後条件アドバイス Advice B1 ではクラス Class `C` に属するメソッド `m` とクラス Class `D` に属するメソッド `o` の本体の実行時点直後で条件 `R` の成立を仮定している．

表明アスペクト（アスペクト）

アドバイスの集合であり，複数の横断的な条件を 1 つの側面としてモジュール化する．図 5 では，アドバイス Advice A1 と Advice A2 を 1 つの表明アスペクト Assertion Aspect A にモジュール化し，さらにアドバイス Advice B1 を表明アスペクト Assertion Aspect B にモジュール化している．

ここで述べた DbC に基づく仕様記述のアスペクト指向的なモジュール化機構は，表明アスペクトを利用することで，仕様の記述対象であるクラスやインタフェースの構造から独立した単位で表明の記述をモジュール化することを可能とする．さらに，複数のジョインポイントを横断するようなプログラムに関する仮定を，ポイントカットを利用し 1 つのアドバイスとして指定することも可能とする．このモジュール化機構を利用することで，従来の素朴な DbC に基づく表明の記述方式の問題点である仕様の複雑化・大規模化に対処することができる．

4. アスペクト指向振舞インタフェース仕様記述言語 Moxa

4.1 言語の定義

本章ではアスペクト指向振舞インタフェース仕様記述言語 Moxa の説明を行う．

以下ではメタ記号として以下のものを利用する．

- [...]：省略可能
- {...}：0 回以上の繰返し
- A|B：A，B のどちらか一方を選択

4.1.1 ジョインポイント

ジョインポイントは，DbC に基づく表明の条件を検査することのできる制御流上のある時点を表す．Moxa が定義するジョインポイントには 2 つの種類があり，それぞれメソッド呼び出しが行われる時点およびメソッドの本体の実行時点である．メソッド呼び出しが行われる時点は事前条件の成立を仮定，メソッドの本体の実行時点は事後条件の成立を仮定するために利用される．個々のジョインポイントはポイントカットの指定の中で，原始ポイントカット（4.1.2 項参照）として記述される．

4.1.2 ポイントカット

ポイントカットは，ジョインポイントを選択し，選択された時点のコンテキストにおけるプログラムの状態を参照する．ポイントカットの指定は，原始ポイントカットと呼ばれるジョインポイントの選択のための記述と，それらを組み合わせるための構文からなる．

原始ポイントカットはジョインポイントを選択するためのポイントカットの記述の最小単位である．原始ポイントカットは次のように記述される．

Type *OnType*.*Id*(*Formals*) [*ThrowsClause*];

この記述は，*OnType* で指定されるクラスやインタフェースに属する，次に示すシグニチャを持つメソッドの呼び出し時点とメソッド本体の実行時点を示す．

Type Id(*Formals*) [*ThrowsClause*];

原始ポイントカットの記述における *Formals* は次のような形でメソッドの引数に関するシグニチャを指定する．

[{*Type* [*Id*],} *Type* [*Id*]]

Id を指定することで，その原始ポイントカットの記述が選択するジョインポイントにおけるメソッドの引数の値を参照する．また，メソッド本体の実行を表す原始ポイントカットは，選択されたメソッドの返値を `\result` として暗黙のうちに参照する．さらに，例外時事後条件の指定のために利用される次のような記述は，メソッド本体の実行が *Type* 型またはそのサブタイプとなる例外を送出した場合に，その例外を *Id* として参照する．

`signals` (*Type* [*Id*]) *s*;

ポイントカットは，次のように原始ポイントカットを並べて記述し空行で終える形で指定する．

```
{⟨ 原始ポイントカットの記述 ⟩}
⟨ 原始ポイントカットの記述 ⟩
⟨ 空行 ⟩
```

このように指定されたポイントカットは，それぞれの原始ポイントカットの記述により選択されるジョインポイントすべてを選択する．

メソッドの呼び出し時点とメソッド本体の実行時点は，原始ポイントカットの記述が属するアドバイスが定義する表明の種類（`requires`，`ensures`，`signals`）により選択される．アドバイスが事前条件を指定する場合はメソッドの呼び出し時点，事後条件・例外時事後条件を指定する場合メソッド本体の実行時点が選択される．`requires`，`ensures`，`signals` は，アドバイスの種類の決定とジョインポイントの選択の 2 つの機能をあわせ持つ点に注意が必要である．

4.1.3　アドバイス

アドバイスは，横断的な表明の条件を定義する．アドバイスは，ポイントカットと論理式から構成され，ポイントカットで選択されるすべてのジョインポイントに対し表明を指定する．Moxa で記述できるアドバイスの標準的な形は次のようになる．

```
/*@ public behavior
{ @  requires r;
| @  ensures  e;
| @  signals (Type [Id]) s;}
  @*/
⟨ ポイントカットの記述 ⟩
```

JML による事前条件・事後条件の記述スタイルと同様に，`@`付きのコメントをポイントカットの記述の直前に配置し，その中に `requires` r と記述することで事前条件 r を，`ensures` e と記述することで事後条件 $r \supset e$ を，さらに `signals` (*Type* [*Id*]) s と記述することで，例外時事後条件として，送出された例外の型を e とすると，$r \supset ((e$ `instanceof` *Type*$) \supset s)$ を指定する．事後条件および例外時事後条件は，事前条件の成立を前提とする点に注意が必要である．これらはそれぞれ 0 回以上繰り返して記述できる．

事前条件 `requires` が 2 回以上指定された場合，それらすべての `requires` の条件を論理積で結んだものと等価である．つまり，以下の 2 つの記述は等価なものとなる．

```
/*@ public behavior
  @  requires r1;
  @  requires r2;
…
```

```
/*@ public behavior
  @  requires r1 && r2;
…
```

また，事前条件 `requires` が 1 つも指定されない場合，それは `requires true` と指定した場合と等価である．事後条件 `ensures`，例外時事後条件 `signals` に関する記述も，事前条件 `requires` の場合と同様である．

特に例外時事後条件 `signals` が 2 回以上指定された場合，すべての指定が検査される点に注意が必要である．たとえば，例外 $E1$，$E2$ が次のような関係にある場合を考える．

```
class E1 extends Exception { ...  }
class E2 extends E1 { ...  }
```

さらに，例外時事後条件が以下のように指定されていたとする．

```
/*@ public behavior
  @  signals(E1) s1;
  @  signals(E2) s2;
…
```

このとき，$E2$ 型の例外が送出されると，1 つ目の signal の指定において $E2$ は $E1$ のサブタイプであるため条件 $s1$ の成立が期待され，それだけでなく 2 つ目の signal の指定において $E2$ は送出される例外の型に等しいため，条件 $s2$ の成立も期待される．また，例外時事後条件が 1 つも指定されない場合，それは signal (Throwable) true と指定した場合と等価である．

2 つ以上のアドバイスは，ジョインポイントが共通の場合に，also を利用してまとめて記述することができる．以下に 2 つのアドバイスを also を用いてまとめて記述した例を示す．

```
/*@ public behavior
  @  requiers  r1;
  @  ensures   e1;
  @  signals  (E1) s1;
  @ also
  @ public behavior
  @  requiers  r2;
  @  ensures   e2;
  @  signals  (E2) s2;
  @*/
〈ポイントカットの記述〉
```

also でまとめられたアドバイスの事前条件は論理和で結合され，事後条件・例外時事後条件は論理積で結合される．したがって，上記のように指定されたアドバイスでは，事前条件・事後条件はそれぞれ，

$r1 \vee r2$,

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

となり，また，例外時事後条件も同様に，送出される例外を e とすると，

$$(r1 \supset ((e \texttt{ instanceof } E1) \supset s1)) \wedge (r2 \supset ((e \texttt{ instanceof } E2) \supset s2))$$

となる．つまり，上記の記述は，以下の記述と等価である．

```
/*@ public behavior
  @  requiers  r1 || r2;
  @  ensures  (r1==>e1) && (r2==>e2);
  @  signals  (Exception e)
  @     (r1==>((e instanceof E1)==>s1))
  @  && (r2==>((e instanceof E2)==>s2));
  @*/
〈ポイントカットの記述〉
```

ここでは，含意（⊃）を，JML により拡張された含意を表す二項演算子==>を用いて記述している．

4.1.4 表明アスペクト

表明アスペクトは，複数のアドバイスを 1 つのグループとしてモジュール化するものである．表明アスペクトの記述は，次のように，アドバイスの記述を並べて記述したものとなる．

```
spec Id1 {
 { depends Id2; }
 { <アドバイスの記述> }
}
```

この記述では，0 個以上のアドバイスの記述を $Id1$ で指定される名前を持つ 1 つの表明アスペクトとして定義している．また，depends は表明アスペクト間の依存関係を表しており，$Id2$ で指定される複数の表明アスペクトの適用を前提としている．

4.2 Moxa による仕様記述

4.2.1 ポイントカットの利用

複数のジョインポイントを選択するポイントカットを記述することで，複数のメソッドに対する共通の表明を 1 つのアドバイスから指定することができる．たとえば，次のようなアドバイスを考える．

```
/*@ public behavior
  @  requires r;
  @*/
  void C.foo();
  void C.bar();
```

このような形のアドバイスは，以下に示すように，唯一のジョインポイントを選択するポイントカットを持つアドバイスに分解して記述することができる．

```
/*@ public behavior
  @  requires r;
  @*/
  void C.foo();

/*@ public behavior
  @  requires r;
  @*/
  void C.bar();
```

この形は，JML におけるメソッドに対する表明の記述の構文とほぼ同じものになる．

また，Moxa では，異なるクラスに属するジョインポイントを選択するポイントカットを構成することができる．たとえば，以下のように記述することで，クラス C に属するメソッド foo() と，クラス D に属す

るメソッド bar() に対して共通の事前条件 r を一度に指定することができる．

```
/*@ public behavior
  @  requires r;
  @*/
  void C.foo();
  void D.bar();
```

4.2.2　継承の扱い

あるクラスを継承し，メソッドの振舞いを変更する場合，振舞サブタイプ（behavioral subtyping）関係[4]を満たさなければならない．振舞サブタイプ関係とは，あるクラスのインスタンスを，そのサブクラスのインスタンスで置き換え可能であることを表すクラス間の関係である．クラス C と D が，C をスーパクラスとする振舞サブタイプ関係にある場合，クラス C のインスタンスを利用するプログラムは，クラス C のインスタンスをクラス D のインスタンスに置き換えた場合でも支障なく動作する．これは，サブクラスでメソッドをオーバライドする場合に，事前条件は弱く事後条件を強くできる，といい換えることもできる．

Moxa では次のように，2 つのクラスの間のサブタイプ関係を振舞サブタイプ関係と考える．

```
class C { void foo() {...} ...}
class D extends C { void foo() {...} ...}
```

そのため，以下のような仕様が与えられた場合，メソッド void D.foo() の事前条件は $r1 \vee r2$，事後条件は $(r1 \supset e1) \wedge (r2 \supset e2)$ となる．

```
/*@ public behavior
  @  requires r1;
  @  ensures  e1;
  @*/
  void C.foo();

/*@ public behavior
  @  requires r2;
  @  ensures  e2;
  @*/
  void D.foo();
```

4.2.3　アドバイスの重なり

あるメソッドに対し，複数のアドバイスが指定された場合，それらの持つ表明の条件は論理積で結合される．たとえば以下のように，メソッド C.foo() に対し，2 つのアドバイスが指定されている場合を考える．

```
/*@ public behavior
  @  requires r1;
  @*/
  void C.foo();

/*@ public behavior
  @  requires r2;
  @*/
  void C.foo();
```

この記述と以下の記述は等価である．ただし，条件 $r1$, $r2$ の評価の順序は未定となる．

```
/*@ public behavior
  @  requires r1 && r2;
  @*/
  void C.foo();
```

このような，1 つのメソッドに表明を指定するアドバイスが 2 つの異なる表明アスペクトに所属し，それらの表明アスペクトの間に depends により依存関係が定義されている場合，依存先の表明の条件が先に評価される．つまり以下のような表明の記述では，クラス C に属するメソッド foo の事前条件は，r1 && r2 となり，必ず r1 の評価が r2 の評価の前に行われる．

```
spec AspectA {
/*@ public behavior
  @  requires r1;
  @*/
  void C.foo();
}
spec AspectB {
 depends AspectA;
/*@ public behavior
  @  requires r2;
  @*/
  void C.foo();
}
```

5．実験的記述と評価

5.1　概　　要

我々が提案する表明のアスペクト指向的モジュール化方式の有効性を明らかにするために，JML と Moxa のそれぞれを利用して記述した仕様の比較を行った．

表 1 JML と Moxa による仕様記述の規模の比較
Table 1 Comparison of the scale of specification descriptions in JML and Moxa.

	JML		Moxa	
	Service	Store	Service	Store
モジュール数	1	1	3	5
表明，アドバイス数	42	53	13	18
行数	190	149	152	286
行数/モジュール数	190	149	51	57

モジュール：JML では仕様，Moxa では表明アスペクトを指す.

仕様記述の対象は，AnZenMail クライアントがメッセージの保存・参照するために利用する Maildir プロバイダモジュールである．この比較では，このモジュールの実装の正しさを検査するために必要となる仕様を JML と Moxa のそれぞれを用いて記述し，その記述の規模（5.2 節）と変更・修正の容易さ（5.3 節）についての比較を行った．本比較で扱うのは，Maildir プロバイダの実装のスーパクラスであり，Maildir プロバイダの実装の振舞いを規定するインタフェースのうちの一部分（javax.mail.Service，javax.mail.Store）に対するものである．付録 A.1，付録 2 に，JML と Moxa それぞれによる Service クラスに対する仕様の記述を添付した（付録に示された仕様の記述は，コメント等を編集したため行数等が評価時の値と若干異なる）．

5.2 仕様の記述の規模

JML，Moxa それぞれにより記述した仕様の規模についての比較結果を**表 1** に示し，そこに見られる特徴を以下に示す．比較項目はモジュール数（JML ではクラス数，Moxa では表明アスペクト数），表明数（JML では事前・事後条件数，Moxa ではアドバイス数），行数（コメントや空行もカウント）とした．

モジュール数

JML による仕様の記述では，クラス Service とクラス Store のそれぞれに対する記述のモジュール数がどちらも 1 となっている．これは，仕様のモジュール化の単位が記述対象である Java のクラスやインタフェースに一致しなければならないためである．一方 Moxa による仕様の記述では，クラス Service に対する表明記述のモジュール数は 3，クラス Store の場合は 5 となっている．これはクラス Service，クラス Store の振舞いにおけるいくつかの側面を，別々の表明アスペクトに分割して記述したためである．具体的にはクラス Service に対する表明アスペクトとして，オブジェクトの状態（ServiceSpec_isConnected），オブジェクトの名前（ServiceSpec_getURLName），その他（ServiceSpec）の 3 つの側面を記述し，さらにクラス Store に対する表明アスペクトとして，クラス Service の 3 つの側面に対する表明アスペクトの拡張のための記述（StoreSpec_isConnected，StoreSpec_getURLName，StoreSpec）に加え，Store が扱うフォルダ（StoreSpec_getFolder），名前空間に関する側面（StoreSpec_getNamespaces）についての記述を行った．

表 明 数

JML で記述した場合の表明数が Service では 42，Store では 53 であるのに対し，Moxa で記述した場合は Service では 13，Store では 18 であり，どちらのクラスに対する仕様も表明数が少なくなっている．これは，表明アスペクトとして取り出されたオブジェクトの振舞いに関する仕様記述において，複数の表明の条件が共通の論理式を持つ場合，すなわち，ある振舞いに関する条件が複数の表明の記述を横断する場合，Moxa ではこれらを 1 つのアドバイスとして記述できるためである．

表明記述の規模

JML による表明の記述の行数は，Service では 190 行，Store では 149 行であるが，Moxa での行数は Service では 152 行，Store では 286 行となっており，Moxa の利用は表明記述の行数を抑えることに貢献していない．しかし，1 つのモジュールあたりの行数は，JML の場合が Service では 190 行，Store では 149 行であるが，Moxa の場合は Service では 51 行，Store では 57 行となり，Moxa による記述の方が，モジュールあたりの行数が少なくなる．Moxa では，表明を複数の表明アスペクトに分割して記述しているため，表明アスペクトの定義のための記述や，ジョインポイントの選択のためのアドバスの記述が増えることが，Moxa による記述の記述量が JML の場合よりも多くなる原因である．また，モジュールあたりの平均行数が JML による記述よりも Moxa による記述の方が少なくなるのは，複数のジョインポイントに対する共通の表明の指定を，ポイントカットを利用して 1 つのアドバイスにまとめて記述できることが影響している．

5.3 変更・修正の容易さ

JML と Moxa それぞれにより記述した仕様を修正する場合の作業の容易さについての比較結果を**表 2** に示す．ここでは，クラス Service，Store に対する仕様の中でメソッド boolean Service-.isConnected() を利用している部分をメソッド boolean Service.notConnected() を利用するように修正する場合についての比較を行った．メソッド

表 2 仕様の変更にともなう修正の波及範囲の比較
Table 2 Comparison of the amount of region on the specification descriptions affected by a modification.

	JML		Moxa	
	Service	Store	Service	Store
修正の波及箇所	42	53	6	4
修正の波及箇所の総行数	190	149	54	40

boolean Service.isConnected() は，これらのクラスのインスタンスの状態の 1 つを得るためのメソッドであり，このメソッドの値と論理値が逆となる値を返すメソッドが boolean Service.notConnected() である．比較項目は，修正の波及箇所の数と行数とした．ここで，修正の波及箇所とは，上記のように仕様の中で利用するメソッドを変更するのにともない修正の必要のある表明（JML の場合）やアドバイス（Moxa の場合）の候補を指す．つまり，修正の波及箇所が指すものの中に，実際に修正の必要がないものも含まれる．実際の修正作業には，修正の波及箇所が指す表明やアドバイスの中から実際に修正の必要のあるものを探し出す必要がある．

まず，JML によって記述された仕様（付録 A.1）の場合は，修正の波及箇所が，表 2 に示したように，クラス Service では 42 個，Store では 53 個となっており，JML によって記述された仕様が持つ表明のすべてとなっている（表 1 参照）．これは，仕様の記述対象であるクラスやインタフェースの構造から独立に仕様を構造化することができないため，メソッド boolean Service.isConnected() の値に関する条件を含む表明の記述と含まないものとを区別して記述することができないことに起因する．

一方，Moxa によって記述された仕様（付録 2）の場合は，クラス Service に対するアドバイスのうちの 6 個，Store に対するアドバイスのうちの 4 個が修正の波及箇所となる．このように，JML による仕様の記述に比べ修正の波及箇所の数が大幅に減少しているのは，Moxa を利用して仕様を記述する場合，仕様記述の対象となるクラスやインタフェースの振舞いを，それが持つ側面に基づきいくつかの独立した表明アスペクトとして別々に記述できることが効いている．我々が記述したクラス Service，Store の仕様では，メソッド boolean Service.isConnected() の値に関する条件は，Service クラスの状態に関する振舞いを表す表明アスペクト（ServiceSpec_isConnected）と Store クラスの状態に関する振舞いを表す表明アスペクト（StoreSpec_isConnected）に局所化されている．さらに，複数のジョインポイントを横断する，メソッド boolean Service.isConnected() の値に関する共通する表明の条件を，1 つのアドバイスとして指定している点も，修正の波及箇所を減らすことに貢献している．

5.4 結論

5.2 節および 5.3 節で行った比較の結果から，DbC に基づく仕様の記述に Moxa を利用することが，以下のような点において有利であるといえる．

- クラスの仕様を表明アスペクトを利用し分割記述することによって，個々の仕様の規模を小さく抑えることができる．
- プログラムや仕様の変更時の影響の波及範囲を明確にすることができる．

これらの利点から，大規模なプログラムの開発に対して DbC に基づく仕様記述を行う場合に問題となる仕様の大規模化・複雑化を回避し，仕様やプログラム本体の修正・開発作業をより効率的なものにするために，Moxa を利用することは有効であると予想される．

6. 関連研究と今後の課題

6.1 関連研究

我々の提案と同様に，アスペクトを利用して表明を記述する方法として石尾らの方法[2)]，Diotalevi の方法[1)] がある．どちらの方法も，プログラム中に表明の記述が埋め込まれることの問題点を指摘し，それを解決するために表明をアスペクトとしてプログラムから分離して記述する方法を提案している．我々の提案は，彼等の提案と同じく表明をアスペクトとしてプログラムから独立して記述できるようにするだけでなく，表明間を横断する性質について着目し，それをアスペクト指向の考え方を用いてモジュール化することで表明記述をコンパクトにして扱いやすくするものである．このような仕事は，我々が知る限り本研究が最初のものである．

JML の対象言語を Java から AspectJ へと拡張した言語として Pipa[8)] がある．この言語を利用すると，AspectJ のアドバイスやイントロダクションの記述に対し表明を記述できる．この言語の場合も，表明はクラス・アスペクトのモジュール化の単位でグループ化しなければならず，JML の場合と同じく表明の大規模化・複雑化に対応できない．我々の提案するモジュール化機構におけるジョインポイントを，AspectJ のアドバイスやイントロダクションを選択できるように拡張することで，Pipa を用いた表明記述を分割・モジュール化することが可能となる．

6.2 今後の課題

今後の課題を以下に示す．

状態遷移モデルの抽出

表明アスペクトを用いるとオブジェクトのある側面だけに注目して仕様を記述できることから，オブジェクトの状態のある側面の変化に関する仕様のみを1つの表明アスペクトに記述することで，JML 等従来の DbC に基づく仕様に比べて状態遷移モデルを抽出することが容易になると考えている．そこで，状態遷移モデルを抽出するのに十分となる表明アスペクトの形を明らかにすることや，そのような表明アスペクトから状態遷移モデルを抽出する方法，抽出されたモデルを検査する機構の作成等を今後の課題とする．

対象言語の AOP 化

現在，Moxa とそのモデルが対象としている言語は Java のようなオブジェクト指向言語であるが，AspectJ のようなアスペクト指向言語を扱うような拡張を考える．アスペクト指向言語を利用したプログラム開発では，プログラム自体がオブジェクトおよびアスペクトを利用してモジュール化される．そのため，Moxa とそのモデルの利点である，プログラムの構造とは独立に仕様を構造化できるという性質の有効性が，大半の場合失われてしまうと予想される．しかし，仕様と実装における側面の取り出し方や粒度を別々に設定できる点が有効となる場合も考えられる．このような点を明らかにすることを今後の課題とする．

表明アスペクトの再利用性に関する考察

表明は一般的に，プログラムの開発時にそのプログラムの動作に関する過程をプログラム中に埋め込まれ，プログラムの動作の検査のために利用されるものであり，また，DbC に基づく表明の記述に関しても，プログラムの開発段階でそのプログラムを構成する関数やメソッドの振舞いを規定するために利用され，それ自体を再利用することは稀であった．これは，従来の表明がプログラムコードの中に埋め込まれ，Moxa を利用すれば表明アスペクトとして別々に記述できたいくつかの側面を1つに合成した形で表明を記述しなければならなかったため，表明の記述を再利用することが困難であったためである．Moxa を利用することで，従来の表明記述のこれらの性質を取り除くことができるが，表明の記述を再利用することが実際の開発においてどのように効いてくるかについての分析は行っていない．以上から，表明記述の再利用が有効である場合の発見，再利用に適した形への言語デザインの変更等を今後の課題とする．

様々な仕様の Moxa による記述

様々な問題を Moxa を用いて記述することにより，Moxa とそのモデルの有効性と問題点をより明確にする．この作業を通して得られた結果をもとに，Moxa の改良を行うことを今後の課題とする．

7. まとめ

本論文で，我々はアスペクト指向に基づく表明記述の新しいモジュール化機構と，その機構のための仕様記述言語 Moxa について述べた．我々が提案するこのモジュール化機構は，実用レベルのソフトウェアモジュール（Maildir プロバイダ）の開発に JML を用いた経験から，表明記述にアスペクト指向の考え方を適用するという着想に基づいてデザインされたものである．さらに本論文では，研究の動機となった Maildir プロバイダの仕様記述に JML と Moxa のそれぞれを適用し，記述量および変更の容易さという観点からその有効性を示している．

Moxa を用いることで，プログラムの構造を横断するような表明中の性質を表明アスペクトという形で分離して記述することができ，クラスの大規模化にともなう表明記述の大規模化と複雑さをおさえることができる．さらに同じ視点の表明が1つの表明アスペクトにまとめられることが，表明記述の見通しを良くすることに貢献している．

DbC は，特に信頼性が必要なソフトウェアの開発に適しているとされている[5] が，ある程度規模の大きいソフトウェアの開発においては，表明の大規模化と複雑化がその有効な利用を阻んでいるのが現状である．本論文で提案する表明のモジュール化機構により，高信頼ソフトウェアの開発に貢献することが期待できる．

謝辞 本研究は科学研究費補助金特定領域研究（安全な情報基盤）12133207 および科学研究費補助金基盤研究（C）15500028 の補助を受けている．

参考文献

1) Diotalevi, F.: Contract enforcement with AOP. http://www-106.ibm.com/developerworks/java/library/j-ceaop/
2) 石尾 隆，神谷年洋，楠本真二，井上克郎：アスペクトを用いた表明の記述，情報処理学会研究報告（2004-SE-144），Vol.2004, No.30, pp.75–82 (2004).
3) Leavens, G.T., Baker, A.L. and Ruby, C.: Preliminary Design of JML: A Behavioral Interface Specification Language for Java, Technical Re-

port 98-06y, Department of Computer Science, Iowa State University (1998). Revised June 2004.
4) Liskov, B. and Wing, J.: A Behavioral Notion of Subtyping, *ACM Trans. Prog. Lang. Syst.*, Vol.16, No.6, pp.1811–1841 (1994).
5) Meyer, B.: Applying "Design by Contract", *Computer*, Vol.25, No.10, pp.40–51 (1992).
6) Shibayama, E., Hagihara, S., Nisizaki, S., Taura, K. and Watanabe, T.: AnZenMail: A Secure and Certified E-Mail System, *Software Security: Theories and Systems*, Okada, M., Pierce, B., Scedrov, A., Tokuda, H. and Yonezawa, A. (Eds.), Lecture Notes in Computer Science, Vol.2609, pp.201–216, Springer-Verlag (2003).
7) 山田 聖, 佐々木明, 望月智之, 渡部卓雄：JMLによるアプリケーションの安全性保証：Maildirフォルダライブラリの一貫性保証，日本ソフトウェア科学会第 20 回大会論文集 (2003). 2B-4 (5 pages).
8) Zhao, J. and Rinard, M.: Pipa: A Behavioral Interface Specification Language for AspectJ, *Proc. Fundamental Approaches to Software Engineering (FASE'2003)*, No.2621, svlncs, pp.150–165 (2003).

付　録

A.1 JML による表明の記述例 (Service.jml)

```
1   package javax.mail;
2
3   import java.util.Vector;
4
5   import javax.mail.event.ConnectionListener;
6   import javax.mail.event.MailEvent;
7   //@ import javax.mail.event.ConnectionEvent;
8
9   public abstract class Service {
10      //====================================================================
11      // Instance Variables
12      //====================================================================
13      // protected Session session;
14      // protected URLName url;
15      // protected boolean debug;
16
17      // private boolean connected;
18      // private Vector connectionListeners;
19
20      //====================================================================
21      // Constructor
22      //====================================================================
23      /*@ protected behavior
24        @     requires   session != null;
25        @     ensures    !this.isConnected();
26        @*/
27      protected Service(Session session, URLName urlname);
28
29      //====================================================================
30      // Connection Management
31      //====================================================================
32      /*@ public behavior
33        @
34        @     requires !this.isConnected();
35        @     ensures  this.isConnected();
36        @     ensures  (* open ConnectionEvent is delivered *);
37        @     signals  (AuthenticationFailedException) !this.isConnected();
38        @                                              // subclassof MessagingException
39        @     signals  (MessagingException)            !this.isConnected();
40        @*/
41      public void connect() throws MessagingException;
```

```
42
43      /*@ public behavior
44        @
45        @     requires !this.isConnected();
46        @     ensures  this.isConnected();
47        @     ensures  (* open ConnectionEvent is delivered *);
48        @     signals  (AuthenticationFailedException) !this.isConnected();
49        @                                            // subclassof MessagingException
50        @     signals  (MessagingException)            !this.isConnected();
51        @*/
52      public void connect(String host, String user, String password)
53          throws MessagingException;
54
55      /*@ public behavior
56        @
57        @     requires !this.isConnected();
58        @     ensures  this.isConnected();
59        @     ensures  (* open ConnectionEvent is delivered *);
60        @     signals  (AuthenticationFailedException) !this.isConnected();
61        @                                            // subclassof MessagingException
62        @     signals  (MessagingException)            !this.isConnected();
63        @*/
64      public void connect(String host, int port, String user, String password)
65          throws MessagingException;
66
67      /*@ public behavior
68        @     requires this.isConnected();
69        @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
70        @     signals (MessagingException)
71        @               URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
72        @     ensures !this.isConnected();
73        @     ensures (* close ConnectionEvent is delivered *);
74        @     signals (MessagingException) !this.isConnected();
75        @*/
76      public synchronized void close() throws MessagingException;
77
78      /*@ public behavior
79        @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
80        @     assignable \nothing;
81        @*/
82      public /*@ pure @*/
83      boolean isConnected();
84
85      //--------------------------------------------------------------------
86      /*@ protected behavior
87        @     requires !this.isConnected();
88        @     ensures this.isConnected() == \old(this.isConnected());
89        @     signals (AuthenticationFailedException)
90        @               this.isConnected() == \old(this.isConnected());
91        @     signals (MessagingException)
92        @               this.isConnected() == \old(this.isConnected());
93        @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
94        @     signals (AuthenticationFailedException)
95        @                URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
96        @     signals (MessagingException)
97        @                URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
98        @*/
99      protected boolean protocolConnect(
100         String host,
101         int port,
102         String user,
103         String password)
```

```
104            throws MessagingException; // , AuthenticationFailedException
105
106        /*@ protected behavior
107          @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
108          @     ensures  this.isConnected() == connected;
109          @*/
110        protected void setConnected(boolean connected);
111
112        //====================================================================
113        // URLName Management
114        //====================================================================
115
116        /*@ public behavior
117          @     ensures this.isConnected() == \old(this.isConnected());
118          @     ensures  \result == null || \result.getPassword() == null;
119          @*/
120        public /*@ pure @*/
121        URLName getURLName();
122
123        /*@ protected behavior
124          @     ensures this.isConnected() == \old(this.isConnected());
125          @*/
126        protected void setURLName(URLName url);
127
128        //====================================================================
129        // Event Management
130        //====================================================================
131        /*@ public behavior
132          @     requires l != null;
133          @     ensures this.isConnected() == \old(this.isConnected());
134          @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
135          @*/
136        public synchronized void addConnectionListener(ConnectionListener l);
137
138        /*@ public behavior
139          @     requires l != null;
140          @     ensures this.isConnected() == \old(this.isConnected());
141          @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
142          @*/
143        public synchronized void removeConnectionListener(ConnectionListener l);
144
145        /*@ protected behavior
146          @     requires type == ConnectionEvent.CLOSED
147          @           || type == ConnectionEvent.DISCONNECTED
148          @           || type == ConnectionEvent.OPENED;
149          @     ensures this.isConnected() == \old(this.isConnected());
150          @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
151          @*/
152        protected void notifyConnectionListeners(int type);
153
154        //====================================================================
155        // methods inherit from Object
156        //====================================================================
157        /*@ also public behavior
158          @     ensures this.isConnected() == \old(this.isConnected());
159          @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
160          @     ensures  \result != null;
161          @*/
162        public String toString();
163
164        //====================================================================
165        // Event Management
```

```
166     //====================================================================
167     // private EventQueue q;
168     // private Object qLock;
169
170     /*@ protected behavior
171       @     requires   event != null;
172       @     requires   vector != null;
173       @     ensures this.isConnected() == \old(this.isConnected());
174       @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
175       @*/
176     protected void queueEvent(MailEvent event, Vector vector);
177
178     // private void terminateQueue();
179
180     //====================================================================
181     // methods inherit from Object
182     //====================================================================
183     /*@ also protected behavior
184       @     requires true;
185       @     ensures  !this.isConnected();
186       @*/
187     protected void finalize() throws Throwable;
188 }
```

2. Moxa による表明の記述例（Service.moxa）

```
 1 package jp.ac.jaist.kyamada.goodies.maildir;
 2
 3 import java.util.Vector;
 4
 5 import javax.mail.Service;
 6 import javax.mail.Session;
 7 import javax.mail.MessagingException;
 8 import javax.mail.URLName;
 9 import javax.mail.event.ConnectionEvent;
10 import javax.mail.event.ConnectionListener;
11 import javax.mail.event.MailEvent;
12
13 //=======================================
14 // for public boolean Service.isConnected()
15 //=======================================
16 spec ServiceSpec_isConnected {
17 //----requires
18     /*@ public behavior
19       @  requires !isConnected();
20       @*/
21     void Service.connect() throws MessagingException;
22     void Service.connect(String, String, String) throws MessagingException;
23     void Service.connect(String, int, String, String) throws MessagingException;
24     boolean Service.protocolConnect(String, int, String, String) throws MessagingException;
25
26     /*@ public behavior
27       @  requires isConnected();
28       @*/
29     public void Service.close() throws MessagingException;
30
31 //----ensures
32     // changes
```

```
33      /*@ public beahvior
34        @  ensures isConnected();
35        @*/
36      void Service.connect() throws MessagingException;
37      void Service.connect(String, String, String) throws MessagingException;
38      void Service.connect(String, int, String, String) throws MessagingException;
39
40      /*@ public beahvior
41        @  ensures !isConnected();
42        @*/
43      boolean Service.protocolConnect(String, int, String, String) throws MessagingException
44      void Service.connect() throws MessagingException;
45      void Service.connect(String, String, String) throws MessagingException;
46      void Service.connect(String, int, String, String) throws MessagingException;
47
48      /*@ public behavior
49        @  ensures isConnected() == connected;
50        @*/
51      void Service.setConnected(boolean connected);
52
53      // no changes
54      /*@ public behavior
55        @   ensures isConnected() == \old(isConnected());
56        @*/
57      URLName Service.getURLName();
58      void Service.setURLName(URLName);
59      void Service.addConnectionListener(ConnectionListener);
60      void Service.removeConnectionListener(ConnectionListener);
61      void Service.notifyConnectionListeners(int);
62      void Service.queueEvent(MailEvent, Vector);
63      void Service.toString();
64
65      // constructor
66      /*@ public behavior
67        @  ensures !isConnected();
68        @*/
69      Service.new(Session, URLName);
70  }
71
72  //=======================================
73  // for public URLName Service.getURLName()
74  //=======================================
75  spec ServiceSpec_getURLName {
76  //---requires
77
78  //---ensures
79      // changes
80
81      // no changes
82      /*@ public behavior
83        @  requires MaildirUtility.checkEqualsURLNames(getURLName(), \old(getURLName()));
84        @*/
85       boolean Service.protocolConnect(String, int, String, String) throws MessagingException;
86       boolean Service.isConnected();
87       void Service.setConnected(boolean);
88       void Service.close() throws MessagingException;
89       void Service.addConnectionListener(ConnectionListener);
90       void Service.removeConnectionListener(ConnectionListener);
91       void Service.notifyConnectionListeners(int);
92       String Service.toString();
93       void Service.queueEvent(MailEvent, Vector);
94  }
```

```
95
96  //=======================================
97  // extra preconditions
98  //=======================================
99  spec ServiceSpec {
100 //---requires
101     /*@ public behavior
102       @  requires session != null && urlname != null;
103       @*/
104     Service.new(Session sesison,URLName urlname);
105
106     /*@ public behavior
107       @  requires l != null;
108       @*/
109     void Service.addConnectionListener(ConnectionListener l);
110     void Service.removeConnectionListener(ConnectionListener l);
111
112     /*@ public behavior
113       @  requires type == ConnectionEvent.CLOSED
114       @         || type == ConnectionEvent.DISCONNECTED
115       @         || type == ConnectionEvent.OPENED;
116       @*/
117     void Service.notifyConnectionListener(int type);
118
119     /*@ public behavior
120       @  requires event != null && vector != null;
121       @*/
122     void Service.queueEvent(MailEvent event, Vector vector);
123
124 //---ensures
125     /*@ public behavior
126       @  ensures \result != null && \result.getPassword() != null;
127       @*/
128     URLName Service.getURLName();
129 }
```

(平成 16 年 12 月 22 日受付)
(平成 17 年 4 月 28 日採録)

山田　聖

1971 年生．2000 年北陸先端科学技術大学院大学情報科学研究科博士前期課程修了．2005 年北陸先端科学技術大学院大学情報科学研究科博士後期課程修了．博士（情報科学）．現在，産業技術総合研究所情報セキュリティ研究センター研究員．オブジェクト指向プログラミングおよびアスペクト指向プログラミングに興味を持つ．日本ソフトウェア科学会学生会員．

渡部　卓雄（正会員）

1963 年生．1986 年東京工業大学理学部情報科学科卒業．1991 年同大学大学院理工学研究科情報科学専攻博士後期課程修了．理学博士．1990 年 4 月～1992 年 3 月日本学術振興会特別研究員（DC・PD）．1991 年 9 月～1992 年 3 月イリノイ大学計算機科学科客員研究員．1992 年 4 月より北陸先端科学技術大学院大学情報科学研究科助教授．2001 年 1 月より東京工業大学情報理工学研究科計算工学専攻助教授．現在に至る．2002 年 4 月から 2004 年 3 月まで国立情報学研究科ソフトウェア研究系（流動部門）に在籍．プログラミング言語の設計と実装，自己反映計算，分散システムに興味を持つ．日本ソフトウェア科学会，ACM，IEEE Computer Society，USENIX 各会員．