博 士 論 文

アスペクト指向的なモジュール記述を可能とする仕様記述言語

指導教官　鈴木 正人 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

山田 聖

2005 年 2 月 13 日

要旨

ソフトウエアを開発する場合，ソフトウエアの規模や複雑さに起因する困難を回避するために，それを幾つかのモジュールに分割し，それらを組み合わせたものとして構成することが一般的に行われる．このように構成されたソフトウエアは，独立した機能を実現する個々のモジュールと，他のモジュールが提供する機能の利用に基づくモジュール間の依存関係によってモデル化することができる．このモデルにおける，モジュール間の機能の提供・利用の関係を契約ととらえ，それに基づきソフトウエアを構成する手法に，契約による設計 (Design by Contract, DbC) がある. DbC は，ある処理の実行直前・直後で満たされるべき条件を明示し，それらを機能の提供者と利用者の間の契約とすることで，責任の切り分けを明確にする手法である．

DbC に基づくオブジェクト指向言語では，個々のメソッドの実行直前・直後で満たされるべき条件をそれぞれ，事前・事後条件と呼ばれる論理式で表現し，表明として記述する．表明が満たされない場合は，それが事前条件の場合はメソッドの呼び出し側に，事後条件の場合はメソッド実装者側に契約違反があることがわかる．このような表明の記述スタイルでは，プログラムコードが複雑化・大規模化するとともに表明の記述量も増加し，個々の表明の論理式も複雑になることから，表明の記述やプログラムコードの一貫性や整合性を保ちつつ，それらの修正や拡張を行うことが難しくなる．この問題の原因は，全ての表明の記述がメソッドに強く関連づけられており，そのメソッドが属するクラスやインターフェイスを単位としたモジュール化を強制させられることにある．このような記述スタイルでは，オブジェクトの振舞いを幾つかの独立した側面に分解して表現できるような場合であっても，個々の側面に関する表明の記述を独立にグループ化し，整理して記述することができない.

本論文では，アスペクト指向的な考えに基づき，表明の記述をメソッドやクラスといった単位から独立して記述することを可能とするモジュール化方式と記述言語を提案する．このモジュール化方式は，動的ジョインポイントモデルに従い，ジョインポイントと呼ばれるプログラムの実行の流れ上の位置を，ポイントカットと呼ばれる言語要素を用いて選択する．更に，ポイントカットで選択された時点で成立が期待される条件を表す論理式との組をアドバイスと呼び，アドバイスの

集合を表明アスペクトと呼ばれるモジュールとする．このモジュール化方式では，あるジョインポイントにおいて成立が期待される条件が複雑である場合にそれを複数のアドバイスに分割して記述できる．これを利用して，オブジェクトの振舞いが幾つかの独立した側面の合成として表現できる場合に，個々の側面に関する表明の記述を，それぞれ独立した表明アスペクトとしてモジュール化することができる．また，このモジュール化方式では，ポイントカットを利用することで，いくつかのジョインポイントで成立が期待される条件が共通である場合に，それらを一つのアドバイスとしてまとめて記述することができる．この表明のモジュール化方式を利用すると，クラスの大規模化に伴う表明の大規模化・複雑化を回避することができる．

謝辞

修士前期課程からこれまで御指導していただいた，東京工業大学大学院情報理工学研究科計算工学専攻助教授 渡部卓雄博士に心から感謝致します．また，北陸先端科学技術大学院大学情報科学研究科情報システム学専攻教授 片山卓也博士，同教授 落水浩一郎博士，同助教授 鈴木正人博士, 東京大学大学院情報理工学系研究科コンピュータ科学専攻教授 米澤明憲 博士には，本論文の査読をして頂き有益な助言を受けることができました．ここに深く感謝の意を表します．更に，北陸先端科学技術大学院大学情報科学研究科情報システム学専攻教授 二木厚吉博士は，副指導教官としてゼミを通して指導をしていただきました．ありがとうございます．

AnZenMail システムの設計・開発の指揮を執られた東京工業大学大学院情報理工学研究科数理・計算科学専攻教授 柴山悦哉博士には，AnZenMail クライアントの設計・開発に対して多大な支援をして頂きました．ありがとうございます．また，共に AnZenMail クライアントの開発に携わった佐々木明氏，望月智之氏に，感謝します．

ともに研究活動に打ち込んできた，北陸先端科学技術大学院大学情報科学研究科渡部研究室，二木研究室，東京工業大学情報理工学研究科計算工学専攻渡部研究室，権藤研究室，米崎研究室，西崎研究室のメンバ，そして，これらの研究室のスタッフの方々に感謝します．

最後に，博士課程での研究活動を応援し続けてくれた家族に (山田卓男，和子, 晶, き久)，そして友人に感謝します．ありがとう．

2005 年 早春

東京工業大学 大岡山キャンパスにて

山田聖

目 次

1 はじめに 1

1.1 契約による設計に基づく仕様記述 1

1.2 仕様記述の大規模化 2

1.3 アスペクト指向に基づく仕様記述のモジュール化 2

1.4 論文の構成 4

2 研究の背景 6

2.1 契約による設計 6

2.2 Java Modeling Language 7

2.2.1 JML による記述 7

2.2.2 JML の処理系 12

2.3 アスペクト指向プログラミング 13

2.4 AspectJ 14

3 AnZenMail クライアントの設計・開発 16

3.1 AnZenMail システム 16

3.2 AnZenMail クライアント 18

4 JML を利用した仕様記述の実践 21

4.1 Maildir プロバイダの開発 21

4.1.1 Maildir プロバイダ 21

4.1.2 JavaMail API 22

4.1.3 Maildir プロバイダの構成 23

4.2 JML を用いた Maildir プロバイダの実装の検査 26

4.2.1 Maildir プロバイダに求められる性質 26

4.2.2 Maildir プロバイダの実装の検査方法 27

4.2.3 振舞サブタイプ関係に基づく検査 28
4.2.4 保存されたメッセージの一貫性の保証 29
4.2.5 結果 30

5 アスペクト指向的な仕様記述のモジュール化方式 33
5.1 DbC に基づく表明の記述の問題点 33
5.2 DbC に基づく表明の記述に現れる横断的側面 35
5.3 DbC に基づく表明の記述へのアスペクト指向の適用 37

6 アスペクト指向的な仕様記述言語 40
6.1 概要 40
6.2 言語の定義 40
6.2.1 ジョインポイント 41
6.2.2 ポイントカット 41
6.2.3 アドバイス 42
6.2.4 表明アスペクト 45
6.3 アスペクト指向的な仕様記述言語の処理系 45
6.3.1 Moxa による仕様記述 46
6.4 実験的記述と評価 49
6.4.1 概要 49
6.4.2 仕様の記述の規模 50
6.4.3 変更・修正の容易さ 51
6.4.4 結論 53

7 まとめ 54
7.1 考察 54
7.2 今後の課題 56
7.3 関連研究 57

参考文献 61

本研究に関する発表論文 64

A JML による表明の記述例 (Service.jml) 65

B Moxa による表明の記述例 (Service.moxa) 69

図 目 次

3.1 AnZenMail クライアントのスクリーンショット 18
3.2 AnZenMail の構造 19

4.1 JavaMail API の構造 22
4.2 Maildir プロバイダの構造 24
4.3 JML を利用したユニットテストの手順 27
4.4 `MaildirManager.putMessage(Message)` メソッド (分割前) . 30
4.5 `MaildirManager.putMessage(Message)` メソッド (分割後) . 32

5.1 DbC に基づく表明の記述 34
5.2 JML を利用した場合の仕様の分割記述ができない例 35
5.3 JML の記述に現れる横断的側面 36
5.4 表明アスペクトを用いた表明のモジュール化の例 39

7.1 Moxa による `Folder` クラスの仕様の記述の一部 59
7.2 Maildir プロバイダの `Folder` クラスの `Folder_state` 表明アスペクトを状態遷移で表したもの 60

第 1章

はじめに

1.1　契約による設計に基づく仕様記述

ソフトウエアの開発においては，ソフトウエアを機能や性質に基づき幾つかの小モジュールに分割する事で，個々のモジュールの規模や複雑さを抑え，開発が困難になることを回避する．このように分割された個々のモジュールはインターフェイスを持ち，それを通して他のモジュールから利用可能なサービスを提供する．ソフトウエアのモジュール分割は，ソフトウエアを独立性の高い小規模なモジュール群の結合とし，それらの間の依存関係を低く抑える事ができることから，保守性，再利用性の向上が期待できる．

契約による設計 (Design by Contract, DbC) は，モジュール化されたソフトエアのインターフェイスを，サービスを提供するモジュールと，それを利用するモジュールとの間の契約と考えることで，モジュール間の責任の切り分けを明確にする．DbC では，サービスを提供するモジュールは，サービスを提供するにあたり，利用者側に期待する条件と実際に提供するサービスを，それぞれ，サービスの提供前に満たされるべき条件 (事前条件, precondition), サービス提供後に満たされる条件 (事後条件, postcondition) として，インターフェイスとともに提示する．サービスを利用するモジュールは，サービスの利用にあたり，事前条件を満たす状況を構成する責任を持ち，サービスを提供するモジュールは，事後条件を満たすようなサービスを提供する責任を持つ．DbC に基づきモジュール間の関係を構成すると，あるモジュールを構成する場合に，それが利用する別のモジュールの実装を

考慮せずに，契約に基づき事後条件を満たすような結果が得られることを期待できることから，DbC はモジュールの独立性を高める効果がある．

1.2　仕様記述の大規模化

DbC に基づき事前条件と事後条件を与えられたインターフェイスは，そのインターフェイスを持つモジュールが提供するサービスがどのようなものかを表現していることから，これはそのモジュールの仕様 (Specification) であると言える．仕様を記述するために，自然言語や図形が利用される場合があるが，そのような仕様は理解が容易である反面厳密性に欠けるため，厳密性を確保するためには仕様を数学的な論理に基づき記述する．このような仕様を，形式仕様 (Formal Specification) と呼ぶ．形式仕様は，それ自体を計算機で扱う事ができ，型式仕様に矛盾がないか，ソフトウエアが型式仕様を満たしているか等を機械的に検査する (Verification) のに役に立つ．しかし，記述の対象となるソフトウエアのモジュールが大規模で複雑なものである場合，型式仕様も大規模化・複雑化し，仕様の整合性や仕様とソフトウエアの一貫性の維持が難しくなる．記述対象であるモジュールを分割したり，サービスを細分化することで，仕様の複雑さを抑えることもできるが，この方法は本末転倒であり，好ましくない．仕様記述の複雑化に対応できる，仕様の記述方式が必要となる．

1.3　アスペクト指向に基づく仕様記述のモジュール化

DbC に基づきモジュールのインターフェイスに与えられた仕様は，次のような特徴を持つ場合が多い．

- モジュールが状態を持っており，サービスを提供可能な状態にあることを事前条件で検査する．
- サービスを提供するのに必要な引数が適切な値であることを事前条件で検査する．
- サービスの実行後，利用者に返す結果が適切なものであることを事後条件で

検査する．

- サービスの実行に伴い，モジュールの状態が変化した場合の，その状態の適切さを事後条件で検査する．

更に，モジュールが提供する機能や状態を，いくつかの独立した側面から捉える事ができる場合がある．具体的には，例えばあるモジュールが幾つかのサブモジュールを

結合したものとして実装されている場合に，そのモジュールの仕様は一つの仕様として表現されているが，それを個々のサブモジュールの仕様を結合したものとして捉える事ができる場合がある．このように，素朴な DbC に基づく仕様記述では，一つのインターフェイスの仕様は，モジュールの機能や状態の持ついくつかの側面を一つにまとめて記述したものとなっている．したがって，これらの一つにまとめられた仕様を状態や側面に関して別々に記述し，インターフェイスの仕様はそれらを結合したものであると指定できるようにすることで，個々の仕様の複雑さを抑えることができると考えられる．

本研究では，この，DbC に基づく仕様記述をモジュラに記述するために，アスペクト指向的を適用する方式と，それに基づく仕様記述言語 Moxa を提案する．アスペクト指向は，素朴なモジュール化方式では一つの独立したモジュールにまとめあげる事が難しく，複数のモジュール中に散在 (scattering) してしまうような機能や概念を，アスペクトと呼ばれる単位でモジュール化することを可能にする手法である．本研究で提案する Moxa のモジュール化機構は，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．表明アスペクトでは，表明をアドバイス，表明を記述するプログラム上の位置 (正確には，メソッドの事前・事後条件を検査するための制御流における位置) をポイントカットとして記述する．本機構により，プログラムの構造から独立に表明をモジュール化することができ，プログラムの大規模化に伴う表明の大規模化・複雑化を抑えることができる．仕様記述言語 Moxa は JML を拡張したものであり，JML と同様の仕様記述形式に加え，表明アスペクトを記述するための構文を導入したものである．

1.4 論文の構成

本論文の以降の章の構成を以下に示す.

第 2 章 研究の背景 では，DbC(Design by Contract, 契約による設計) と呼ばれるソフトウエアの構成手法について述べ，この手法に基づいて Java のソフトウエアを開発するための言語の一つである JML(Java Modeling Language) を紹介する．更に，アスペクト指向と呼ばれる方式に基づきコードのより自然なモジュール化を可能とするアスペクト指向プログラミングについての説明を行い，アスペクト指向プログラミング言語の一つである AspectJ の紹介を行う.

第 3 章 AnZenMail クライアントの設計・開発 では，我々が設計･開発した AnZenMail クライアントについての紹介を行う．このソフトウエアは，従来のメイルシステムとの互換性があり，一般的な電子メイルクライアントソフトウエアと同じく，GUI を通してメッセージの編集や送受信・整理を行うことができる．また，AnZenMail クライアントはプラグイン機構を持ち，新しい安全性向上技術を組み込む事に利用される.

第 4 章 JML を利用した仕様記述の実践 では，AnZenMail クライアントの実装の品質向上を目的として行った JML による仕様の記述とその仕様を用いた実装の正しさの検査についての説明を行う．仕様の記述対象は，Maildir プロバイダと呼ばれる AnZenMail クライアントがファイルシステム上に電子メイルメッセージを保存・保存されたメッセージを参照するために利用するモジュールである．本章では，Maildir プロバイダと，JavaMail API と呼ばれる Java で電子メイルメッセージを扱うための抽象的操作を定めた標準拡張 API を紹介し，Maildir プロバイダが JavaMail を継承しファイルシステムに対する具体的な操作を実装している事を述べる．更に，Maildir プロバイダに対して行った検査の内容と検査方法を説明する．最後に，この検査から得られた結果を述べる.

第 5 章 アスペクト指向的な仕様記述のモジュール化方式 では，DbC に基づいた仕様を記述した場合に直面する問題について述べる．クラスやインターフェイスといったプログラムモジュールの規模が大きい場合，それらに対する仕

様も大規模で複雑なものとなる．JML 等，従来の DbC に基づく仕様記述言語では，仕様の記述単位が記述対象であるプログラムの構造に依存して決まり，それらの構造から独立に仕様をモジュール化する事ができないため，仕様の大規模化に対応する事ができない点を，我々は問題であると考える．この問題を解決するために，我々は DbC に基づき記述された仕様が持つ，仕様記述対象のプログラムの構造から独立した構造を利用する方法を提案する．

本章では，まず，DbC に基づき仕様を記述する場合に，仕様が大規模化・複雑化する問題について述べる．次に，そのような大規模で複雑な仕様が，記述対象のクラスの持つ構造から独立した構造を持つ場合がある事を説明する．更に，そのような仕様の記述に対し，アスペクト指向を導入する事で，プログラムの構造から独立した仕様の持つ構造を，自然にモジュール化する方式の提案を行う．

第 6 章 アスペクト指向的な仕様記述言語 では，第 5 章で述べた仕様のモジュール化方式に基づいた仕様を記述のするための，アスペクト指向振舞インターフェイス仕様記述言語・Moxa について述べる．本章では，まず概要を述べ，それに続き Moxa の定義を示し，その処理系について述べる．更に，Moxa と JML のそれぞれを用いて，共通のコードに対して仕様の記述を行い，それらの比較を行う．

第 7 章 まとめ では，本研究に関する考察を示し，今後の課題を述べる．更に，関連研究の紹介を行う．

第 2章

研究の背景

本章では，本研究の背景となる，契約による設計，及び，アスペクト指向プログラミングについての説明を述べる．

2.1 契約による設計

契約による設計 (Design by Contract, DbC)[16] は，あるサービスの提供者と利用者の間に契約の概念を導入し，それに基づきソフトウエアを構成する手法である．ここで，サービスとは，関数型言語における関数，オブジェクト指向言語におけるメソッド，サーバ・クライアントモデルに基づくソフトウエアにおけるサーバが提供する機能等を表す．この手法では，あるサービスの提供者は利用者に対して，そのサービスを提供可能な状態を表す条件 (事前条件, precondition) と，そのサービスの提供直後に成立する条件 (事後条件, postcondition) を提示する．これが提供されるサービスの仕様となる．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が，利用者と提供者それぞれの責任となり，双方がこれらの条件を満たす事が契約となる．この手法に基づきソフトウエアを構成した場合，事前条件が満たされない場合は利用者側に，事後条件の場合は提供者側に問題があることがわかり，問題に対する責任の切り分けが，明確に行えるようになる．また，サービスの利用者は，サービスの事前条件を満たしている限り，提供者によるその実現方法を考慮すること無く事後条件を満たすような結果が得られることを仮定することがで

きる．これらの性質から，DbC はモジュール性が高く信頼性のあるソフトウエアを構成するための道具として役立つ手法である．

表明とは，プログラムの制御の流れ上のある時点で，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，表明が満たされない場合 (表明違反, assertion failed) を探すことで，そのソフトウエ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が，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．

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箇所に到達した場合，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直ちにプログラムの実行を停止する．動的な表明の検査は，表明の検査のためのコードを実行時プログラムの中に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エアに対し行われ，完成版のソフトウエアからは取り除かれる．

DbC に基づく表明の指定可能な時点は，関数やメソッドの事前条件・事後条件を検査する位置，つまり関数やメソッドの呼び出し及び復帰時に限られる．事前条件・事後条件を検査する時点以外の時点に表明を指定することが許されないのではなく，それ以外の時点に指定された表明は DbC の対象とはならず一般的な表明の記述となる．

2.2 Java Modeling Language

2.2.1 JML による記述

JML(Java Modeling Language)[13] は，Java のための振舞インターフェイス仕様記述言語 (Behavioral Interface Specification Language) の一つである．JML は，DbC, モデルベースの仕様記述，refinement calculus の概念に基づいた言語である．JML は Java のコードに対して，インターフェイスと振舞いの記述を可能とする．

JML の文法は Java の文法を拡張したものとなっており，Java のコードに対する JML を用いたインターフェイスの記述のために，Java のクラスやインターフェイスの記述のための構文が，ほぼそのまま利用される．一方，コードの振舞いは DbC に基づき表明として記述される．JML により Java の文法に加えられた拡張は，この DbC に基づく表明の記述のためのものである．JML により拡張された構文は，アノテーションと呼ばれる “@” を伴った特殊な Java のコメント中にのみ記述が許されるため，JML で記述された仕様は Java の文法にほぼ従うものとなる (JML では，クラスの仕様記述と実装を別々に記述する事ができる．このクラスの仕様の記述のために，Java のクラスの定義の構文がメソッドの実装を持たないよう変更が加えられている．この変更された構文は Java の文法を満たさない．) .

例えば，インスタンス変数 `int v` を持つ Java のクラス `C0` に属するメソッド `m0(int)` に対し，JML を用いて表明を指定する場合には次のように記述する．

```
public class C0 {
  private int v;

/*@ public behavior
  @  requires P0(a, v);
  @  ensures Q0(a, v);
  @  signals (Exception0 e) R0(a, v, e);
  @*/
 public int m0(int a) throws ExceptionX { ... }
}
```

表明を指定するメソッドの直前にアノテーションを置き，その中で，キーワード `requires` に続けて事前条件を表す論理式 `P0(a, v)` (`a` はクラス `C0` に属するメソッド `m0(int)` の仮引数, `v` はクラス `C0` が持つインスタンス変数) を，キーワード `ensures` に続けて事後条件を表す論理式 `Q0(a, v)` を，また例外時事後条件として，キーワード `signals` に続けて送出される例外を選択する型 `Exception0(e` は送出された例外) と条件を表す論理式 `R0(a, v, e)` を指定する．この記述の意味は,「クラス `C0` に属するメソッド `m0(int)` を条件 `P0(a, v)` を満たす状態で実行し，正常終了した場合の状態は条件 `Q0(a, v)` を満たし，例外を送出して終

了した場合は，その例外が Exception0 かそのサブクラスのインスタンスであった場合は，その状態は条件 R(a, v, e) を満たす」となる．

JML では表明の論理式を表すために，Java の boolean 型の式に拡張構文を追加した式を利用する．論理式中に現れる変数は，メソッドの引数，インスタンス変数を参照し，例外時事後条件を表す論理式の中では例外の型のマッチング時に束縛された例外オブジェクトを参照する．また，事後条件を表す論理式中では特殊な変数 \result を通してメソッドの返値を参照できる．更に，事後条件，例外時事後条件を表す論理式の中では \old(<式>) を用いてメソッド実行直前の<式>の値を参照できる．また，含意 (⊃) を表す演算子==>や限量子 \forall，\exists など表明の条件の記述を容易にするための演算子が用意されている．また，事前条件，事後条件，例外時事後条件を表す論理式の中に Java のメソッド呼び出し式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，副作用を持たないことを pure 修飾子を指定することで明示的に宣言されたものに限られる．

事前条件，事後条件，例外時事後条件は零回以上指定する事ができる．事前条件，事後条件，例外時事後条件の省略は，それぞれ次のように指定された場合と等しい．

```
requires true;
```

```
ensures true;
```

```
signals (Throwable) true;
```

一方，事前条件，事後条件が，それぞれニ度以上指定された場合は，全ての論理式が論理積で結ばれた一つの表明の指定と等価である．つまり，以下のような二つの事前条件の指定は等価である．また，事後条件の指定の場合も同様である．

```
requires r1;
requires r2;
```

```
requires r1 && r2;
```

また，例外時事後条件が二つ以上指定された場合も同様に全ての条件が論理積で結ばれる．つまり，以下のような二つの例外時事後条件の指定は等価なものとなる．

```
signals (Exception1 e1) r1(e1);
signals (Exception2 e2) r2(e2);
```

```
signals (Throwable e)
    ((ex instanceof Exception1) ==> r1(e))
 && ((ex instanceof Exception2) ==> r2(e))
```

複数の例外時事後条件の指定は，例外として送出されるオブジェクトの型にマッチする全ての条件の成立が期待される (最初にマッチした条件のみの成立が期待されるわけではない).

一方，JML における表明の記述は，`also` を用いて分割して記述することができる．例えば以下のような二つの表明の記述は等価なものとなる．

```
public class C1 {
 /*@ public behavior
   @  requires P1;
   @  ensures Q1;
   @  signals (Exception1 e) R1(e);
   @ also public behavior
   @  requires P2;
   @  ensures Q2;
   @  signals (Exception2 e) R2(e);
   @*/
 public void m1() throws ExceptionY {...}
}
```

```
public class C1 {
 /*@ public behavior
   @  requires P1 || P2;
   @  ensures (P1 ==> Q1) && (P2 ==> Q2);
   @  signals (Throwable e)
   @       (P1 ==> ((e instanceof Exception1) ==> R1(e)))
   @   && (P2 ==> ((e instanceof Exception2) ==> R2(e)));
   @*/
 public void m1() throws ExceptionY {...}
}
```

also を用いて記述された仕様と等価な also を含まない仕様は，事前条件は論理和により結合され，事後条件と例外時事後条件は，合成前の対応する事前条件が成立する場合を論理積で結合したものとなる．

also を用いた表明の分割は，クラスの継承によりオーバー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．つまり，以下に示すように仕様が与えられた場合，クラス C3 のメソッド m1 の事前条件，事後条件は

```
P1 || P2
```

```
P1 ==> Q1 && P2 ==> Q2
```

となり，例外時事後条件は送出される例外を e とすると

```
     (P1 ==> ((e instanceof Exception1) ==> R1(e)))
  && (P2 ==> ((e instanceof Exception2) ==> R2(e)))
```

となる．

```
public class C2 {
 /*@ public behavior
   @  requires P1;
   @  ensures Q1;
   @  signals (Exception1 e) R1(e);
   @*/
 public void m1() throws ExceptionZ1 {...}
}

public class C3 {
 /*@ also public behavior
   @  requires P2;
   @  ensures Q2;
   @  signals (Exception1 e) R2(e);
   @*/
 public void m1() throws ExceptionZ2 {...}
}
```

2.2.2 JML の処理系

JML には JML Tools と呼ばれるツール群があり，JML による記述の作成・検査のために利用される．以下に主なものを示す.

jml JML による記述と，対応する Java プログラムにをパースし，それらに対する型検査を行うチェッカ．構文エラー，型の誤り，未定義変数への参照といった誤り等を発見するために利用される.

jmlc JML による記述と，対応する Java プログラムから，実行時表明検査のためのコードが埋め込まれたクラスファイルを生成するコンパイラ．生成されたクラスファイルを実行することで，JML により記述された表明に対する表明違反の存在を検査できる.

jmlrac jmlc により生成されたクラスファイルを実行するための仮想マシン起

動コマンド．Java 言語における `java` コマンドに対応．

`jmlunit` Java のコードから単体テストフレームワーク JUnit[8] のためのテストケースのテンプレートを生成するツール．生成されたテンプレートに，テスト対象となるクラスの個々のメソッドの事前条件を満たすテストデータを生成するコードを埋め込むことで，ユニットテストの個々の試行と結果の妥当性検査が自動的に行われる．ユニットテストの結果の妥当性検査は `jmlc` により生成されたクラスファイルの実行による事後条件の検査により行われる．

`jmlspec` Java のコードから JML の記述のテンプレートを生成するツール．

`jmldoc` JML のコードから HTML 型式のページを生成する．Java 言語における `javadoc` コマンドに対応．

JML Tools とは別に，JML による記述と Java のプログラムを対象としたツールやアプリケーションが存在する．以下に主なものを示す．

ESC/Java [3, 7, 14]Java のコード自体の誤りの検出と，Java のコードと JML による記述の一部との整合性の検査を静的に行う．

ESC/Java2 [12]ESC/Java の強化版．ESC/Java に比べ，扱う事のできる JML の構文が増え，検査項目の強化がなされている．

LOOP [20] アノテーションと指定された Java のプログラムから，定理証明系 PVS のための定理群を生成する．ESC/Java, ESC/Java と異なり JML の構文全てを扱う事ができるが，PVS[17] による定理証明はユーザによるインタラクションが必要となる．

Daikon [5, 6] Java のプログラムの実行時の振舞いを観測し，その結果を元に Java のクラスの不変条件を生成する．

2.3 アスペクト指向プログラミング

オブジェクト指向パラダイムは，物や概念といった対象をオブジェクトしてモデル化し，その振舞いをオブジェクト間の相互作用として捉えようという考え方で

ある．このパラダイムでは，モデル化の対象の状態や機能が，それぞれデータとメソッドとして抽象化され，それらは一まとまりのオブジェクトとしてモジュール化される．オブジェクト指向パラダイムを用いることで，ソフトウエアの構成要素の独立性を高め，それらの間の結合度を下げることができる．これを関心事の分離 (Separation of Concerns) と言う．

このように，オブジェクト指向パラダイムが関心事の分離を実現する一方で，適切にモジュール化できないような対象が存在する．オブジェクト指向的な方法では適切にモジュール化できずにメソッドの実装コードの中に散在 (scattering) してしまうものの典型的な例として，ロギング処理やトランザクション処理，セキュリティ機能がある．これらはどれも，本来の処理から独立しており，それらとは直接的な関連を持たず，多数のメソッドで利用されるという特徴を持つ．このような，ソフトウエアの構成要素を横断して様々な箇所に散在する関心事を，横断的関心事 (Crosscutting Concerns) と呼ぶ．この，横断的要素をアスペクト (Aspect) と呼ばれる独立した一つのモジュールとして表現できるようにすることで，関心事の分離を押し進めようという考え方がアスペクト指向である．アスペクト指向は，オブジェクト指向を置き換えるものでは無く，オブジェクト指向と組み合わせて利用するものである．アスペクト指向プログラミングは，オブジェクト指向プログラミングではモジュール化が困難であった横断的要素のモジュール化を可能とすることで，コードのモジュール性，保守性，再利用性などを改善する．

2.4 AspectJ

AspectJ[11] は，Java に対し，アスペクト指向に基づくモジュール記述のための要素が拡張された，アスペクト指向プログラミング言語 (AOP) である．AspectJ を利用すると，従来の Java の記述に加え，Java では複数のクラスやインターフェイス，メソッドを横断して存在していた機能を，本来のコードから独立したアスペクトと呼ばれるモジュールにまとめて記述することができる．AspectJ は，動的ジョインポイントモデルに基づき，アスペクト (Aspect)，ポイントカット (Pointcut)，アドバイス (Advice)，ジョインポイント (Join Point) と呼ばれる構成要素により，アスペクト指向パラダイムに基づくプログラミング環境を提供している．これらの

構成要素の説明を以下に示す.

ジョインポイント プログラムの実行の流れ上の明確に定義された時点を表す．ジョインポイントには，アドバイスの実行を割り込ませることができる．AspectJ において利用できるジョインポイントには，メソッドやコンストラクタ呼び出し時点，フィールドへのアクセス時点，例外ハンドラの実行時点等がある．

ポイントカット ポイントカットは幾つかのジョインポイントを選択したり，それらのジョインポイントにおける実行コンテキストからデータを取り出したりすためのプログラム要素である．ポイントカットは，アドバイスとともに利用される．AspectJ で利用できるポイントカットは，唯一のジョインポイントを選択する原始ポイントカットと，複数のポイントカットを組み合わせる論理演算子を用いて記述される．更に，実行コンテキストからデータを取り出すためのポイントカットを利用する事ができ，実行中のメソッドを持つオブジェクト，メソッド呼び出しやフィールドアクセス時のターゲットとなるオブジェクトを参照できる．

アドバイス アドバイスは，横断的な振舞いを定義する．アドバイスはポイントカットとコードから構成され，ポイントカットで選択される全てのジョインポイントにおいて，アドバイスとして指定されたコードが実行される．AspectJ で利用できるアドバイスには before, after, around の 3 種類があり，アドバイスとして指定されたコードが，ジョインポイントにおいてどのように実行されるかが異なる．before, after アドバイスはそれぞれ，ジョインポイントの直前・直後において実行される．around アドバイスはジョインポイントの周りで実行され，本来のジョインポイントの実行の制御が可能である．

アスペクト AspectJ におけるアスペクトは複数のアドバイスのモジュール化のための単位である．

第 3章

AnZenMailクライアントの設計・開発

3.1 AnZenMailシステム

AnZenMail システムは，文部科学省科学研究費補助金特定領域研究「社会基盤としてのセキュアコンピューティングの実現方式の研究」(平成 12 年度から平成 15 年度) における領域内共同研究として開発されたメイルシステムである．このシステムは，科学的なアプローチに基づき安全性を保証できるソフトウエアシステムの構築方式を確立することを目的として開発が行われたものである．

AnZenMail システムは，一般的なメイルシステムと同様に，AnZenMail サーバと呼ばれるサーバ部と AnZenMail クライアントと呼ばれるクライアント部から構成される．また，SMTP プロトコルによるメッセージの送受信，POP3, IMAP4 プロトコルを用いたメッセージストアへのアクセスが可能であり，従来インターネットで利用されるメイルシステムとの互換性がある．

一方，AnZenMail システムは従来の電子メイルシステムと異なり，安全性を保証することを目的として,「三重のセイフティネット」と呼ばれる防御戦略がとられている．この戦略は，システムの設計・実装の過程を三つの段階に分類し，それぞれの段階において以下に示す安全性を保証する技術を適用するというものである．

プログラムやプロトコルの理論的な検証・解析技術 プログラムやプロトコルを解析・検証する事で，実装以前にそれらの問題点や危険性を発見する．具体的には，次のような技術が利用されている．

プロトコル検証技術 電子メイルメッセージの配送経路の詐称を検出できるプロトコルを設計・実装されており，サーバ及びクライアントに組み込まれている．このプロトコルの正しさは数学的に証明されている．

ソフトウエア検証技術 サーバの実装の正しさについて，その一部分が証明されている．示されている性質は，サーバが SMTP プロトコルに従いメッセージを送受信していること，サーバがメッセージを受け取る場合，メッセージがストレージ上に保存された後に，メッセージの受信を送信者に通達することである．

安全なプログラミング言語・記述系の利用・改良 安全なプログラミング言語を設計・利用する事で，危険な操作をプログラムの開発段階で防ぐ．また，数学的に性質の良い言語を設計・利用する事で，ソフトウエアの安全性を示しやすくし，効率の良いコードの生成を可能とする．

AnZenMail システムでは，安全なプログラミング言語として Java を利用している．Java はメモリセーフ言語の一つであり，バッファオーバーフロー攻撃等に強い．

実行時検査 プログラミング言語レベルで検出する事が困難である問題に対処するために，OS やプログラミング言語のランタイム環境において，プログラムの振舞いを監視しつつ実行を行う．具体的に，組み込まれている技術は以下の通り．

サンドボックス技術 サーバのための拡張モジュールやメイルに添付された実行ファイルを安全に実行するために，SoftwarePot を利用してこれらのソフトウエアを OS やサーバ・クライアントから隔離して実行する．

未知ウイルス検知技術 プログラムの抽象実行とコード解析を行うことで，ウイルスのパターンデータベースを利用することなくウイルスの検知を可能とする．メイルに添付された実行ファイルを安全時実行するために利用される．IA32 アーキテクチャ・Win32 API を対象とする．

3.2 AnZenMail クライアント

AnZenMail クライアントは AnZenMail システムの Mail User Agent(MUA) である．このクライアントの設計目標は，メイルシステムの利便性と，メールに添付された実行コードの実行の安全性を両立する事となっている．従来の電子メイルメイルクライアントと同じく，このクライアント SMTP プロトコルによるメッセージ送信，POP3, IMAP4 プロトコルによるメッセージストアの参照ができ，また，GUI を通して MIME 形式のメッセージの編集及び表示ができる (図 3.1).

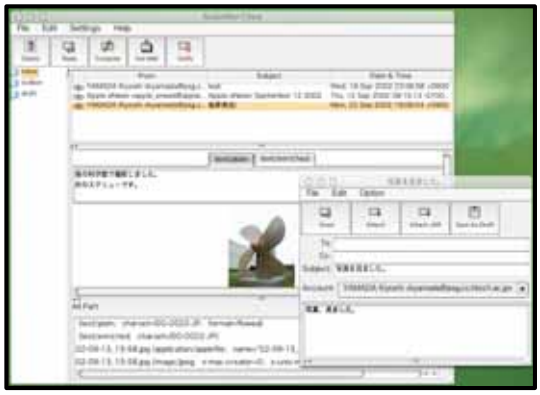

図 3.1: AnZenMail クライアントのスクリーンショット

AnZenMail クライアントは，4 名の開発者で開発を行った 5 万行程度の規模のソフトウエアである．開発の期間は約 2 年間であるが，これらの期間をすべて開発に費やしたわけではなく，作業は断続的に行われた．このクライアントは，GUI を実現するために Java の Swing ライブラリを利用し，また電子メイルメッセージを扱うために JavaMail ライブラリを利用している．これらのライブラリ及び AnZenMail のコードに関する形式的な安全性の検証は行われていないが，一部のモジュールに対する安全性の検査の作業は行われている (第 4 節).

AnZenMail クライアントの大きな特徴は，プラグインアーキテクチャを持ち，それに基づき複数のモジュールを結合した形で構成されている点にある．そのため，AnZenMail クライアントは高い拡張性を持つ．また，自身を幾つかのモジュールの組み合わせとして実現する事で，個々のモジュールの独立性・データの局所性を高めている．また，このプラグインアーキテクチャは，外部モジュールとして

開発された安全性向上技術を組み込むために利用されている点にある．図 3.2 に AnZenMail クライアントが添付ファイルを持つ電子メイルメッセージを受け取る時の動作の大まかな流れを示す．AnZenMail クライアントは，まず，受信部を通して電子メイルメッセージを取得する．受信したメッセージの MIME タイプに従い，それが表示可能なアタッチメントである場合には，適切な表示モジュールが選択され表示される．アタッチメントが実行ファイルであった場合，そのコードは検証系に渡される．検証系は，プラグインとして組み込まれた検証モジュールにコードを渡しその危険性を検査させる．ここで，全ての検証モジュールが危険性を報告しなかった場合，そのコードは次の実行系に渡される．任意の検証モジュールがコードの危険性を報告した場合，そのコードは実行系には渡されず，実行される事は無い．実行系は，そのコードの MIME タイプに応じて適切な実行モジュールにそのコードを渡し実行させる．実行モジュールはコードの実行を監視し，危険な動作を行おうとした場合に，そのコードの実行を停止する．具体的に，組み

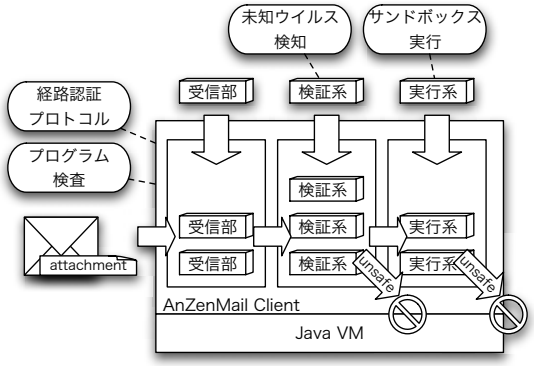

図 3.2: AnZenMail の構造

込み可能なプラグインとして，以下に示すようなモジュールが開発されている．

SoftwarePot プラグイン メッセージに添付された実行ファイルをサンドボックス中で実行するためのモジュール．Linux/x86, Solaris/Sparc のコードを対象とし，実行時にシステムコールを捕捉，検査することでサンドボックスを実現している．

未知ウイルス検知プラグイン メッセージに添付された実行ファイルに対し，静的

解析と抽象実行を組み合わせ，ソフトウエアの動作をシミュレートし，その振舞いから悪意のあるコードを発見する．

Java Signature 検査プラグイン メッセージに添付された Java のアーカイブファイルが適切なシグニチャを持っていることを確認するモジュール．

Resource Usage Analysis プラグイン リソースの消費量に関する型アノテーション付きの Java コードがメールに添付されている場合，コードとその型アノテーションの整合性を静的に検査するモジュール．このモジュールは Windows/x86 を対象とする．

第 4章

JML を利用した仕様記述の実践

我々はこれまでに，AnZenMail システム [18] の設計・開発に携わり，AnZenMail クライアントの開発を行なった．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，更にそのライブラリの信頼性を高めるために，JML を利用してこの実装の検査を行った [21]．この検査は，JML を利用して DbC に基づく表明として 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．この章では，JML を利用した Maildir プロバイダの検査について述べる．

4.1 Maildir プロバイダの開発

4.1.1 Maildir プロバイダ

Maildir プロバイダは，電子メイルシステムにおいてやり取りされるメッセージを，ファイルシステム上へ保存する機能と，保存されたメッセージを参照・操作する機能を提供する Java のモジュールの一つである．Maildir プロバイダは，メッセージをファイルシステム上に保存する形式として，メールサーバ qmail[2] で用いられている Maildir フォルダ形式 [1] を利用する．この保存形式は，一般的なファイルシステムと同様に，フォルダを利用してメッセージを階層的に分類できることや，ファイルの保存形式や手順の工夫によりメッセージが不用意に破壊される

可能性が抑えられていることが，mbox 形式や MH フォルダ形式等の他の型式に比べ優れている．

また，この Maildir プロバイダは，Java 上で電子メイルやネットニュース等のメッセージを扱うための API である JavaMail API(第 4.1.2 節参照) のライブラリに対するプラグインモジュールとなっている．そのため，JavaMail API を利用するソフトウエアは，容易に Maildir プロバイダを利用することができる．

4.1.2 JavaMail API

JavaMail API[19] は，電子メイルシステムやネットニュースシステムをモデル化した API の集合であり，Java の上でメッセージを編集・送信・保存するための，プロトコルや形式に依存しない抽象的なフレームワークを提供する，Java の標準拡張 API の一つである．JavaMail API は抽象層と実装層の二つの層から構成される (図 4.1).

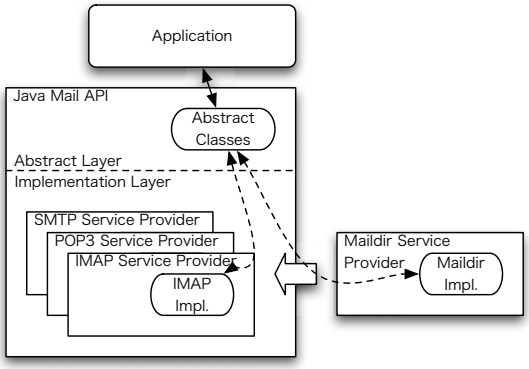

図 4.1: JavaMail API の構造

抽象層では全てのメイルシステムに共通する概念や操作のためのクラスやインターフェイス，抽象メソッドが定義される．抽象層で定義されるクラスの主なものを以下に示す.

`Service` メッセージングサービスに共通する機能を持つ抽象クラス.

`Store` (extends `Service`) メッセージストアとそれにアクセスするためのプロト

コルをモデル化したクラス，メッセージの保存・参照機能を提供する．

`Transport` (extends `Service`) メッセージトランスポートをモデル化したクラス．メッセージを送信する機能を提供する．

`Folder` メッセージを保存するフォルダをモデル化したクラス．

`Message` メッセージをモデル化したクラス．

一方，実装層は抽象層を構成するクラスやインターフェイスを拡張し，特定の保存型式やプロトコルに従った，メッセージの保存・参照・送信といった操作や，メッセージそれ自体の作成・修正を行う操作が実装されたクラスで構成される．特定の保存形式やプロトコルを扱うための実装は，サービスプロバイダと呼ばれるモジュールとしてまとめられ，JavaMail に対するプラグインとして自由に追加できる．JavaMail API を利用するソフトウエアは JavaMail API の抽象層が定めるインターフェイスを通して間接的にサービスプロバイダを利用することで，異なった型式やプロトコルを共通の操作で扱うことができる．JavaMail は，電子メイルのやりとりに主に利用されるプロトコルをサポートする以下のサービスプロバイダとともに公開されている．

IMAP ストア サービスプロバイダ IMAP 形式のメッセージストアへのアクセスを提供する．

POP3 ストア サービスプロバイダ POP3 形式のメッセージストアへのアクセスを提供する．

SMTP トランスポート サービスプロバイダ SMTP プロトコルによるメッセージ送信を提供する．

4.1.3 Maildir プロバイダの構成

Maildir プロバイダは四つの主要クラスと，それらをサポートする幾つかのクラスで構成されている (図 4.2)(ここでは，主要クラスについての説明を行い，その他のクラスの説明は省略する．)．Maildir プロバイダの主要クラスは，`MaildirStore`,

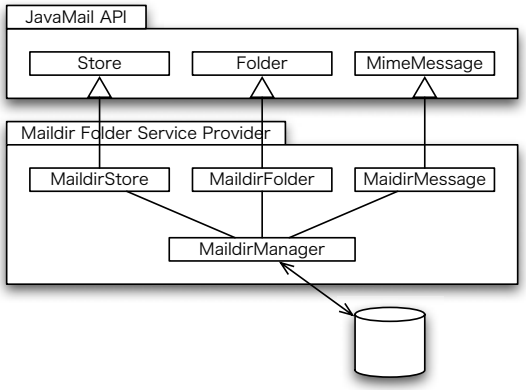

図 4.2: Maildir プロバイダの構造

`MaildirFolder`, `MaildirMessage` の三つのクラスとクラス `MaildirManager` である．`MaildirStore`，`MaildirFolder`，`MaildirMessage` の三つのクラスの説明を以下に示す．

`MaildirStore` (extends `Store`) ファイルシステム上の Maildir フォルダ形式のディレクトリを表現するクラス．

`MaildirFolder` (extends `Folder`) Maildir フォルダ形式が提供する，メッセージを保存するためのフォルダを表現するクラス．

`MaildirMessage` (extends `MimeMessagae`) フォルダに保存されるメッセージを表現するクラス (クラス `MimeMessage` は RFC822 形式のメッセージを扱うクラス，JavaMail の実装層で定義される．クラス `Message` を拡張)．

これらのクラスは，JavaMail API の抽象層が定義するクラス `Store`，`Folder`，`MimeMessage` を継承し，JavaMail API の定めるメッセージストアに対する抽象的な操作に従う形で Maildir フォルダに関する操作を提供する．

一方，クラス `MaildirManager` は，ファイルシステム上にある Maildir フォルダ形式のディレクトリや保存されているメッセージファイルに関する具体的な操作を実装する．クラス `MaildirManager` が持つ主なメソッドと，その機能を以下に示す．

MaildirMessage putMessage(MaildirFolder, Message) 指定されたフォルダにメッセージを追加し，追加されたメッセージに対応する MaildirMessage オブジェクトを返す．

boolean deleteMessage(MaildirMessage) メッセージを削除する．

File updateFlags(MaildirMessage, Flags) 指定されたメッセージのフラグを変更する．メッセージに対応するファイルのファイル名を返す (Maildir 形式ではメッセージのフラグをファイル名にエンコードするため，フラグの変更によりファイル名が変化する．)．

boolean createFolder(MaildirFolder) フォルダを作成する．

boolean deleteFolder(MaildirFolder) フォルダを削除する．

boolean existsFolder(MaildirFolder) フォルダが存在するか調べる．

boolean renameToFolder(MaildirFolder, MaildirFolder) フォルダの名前を変更する．

boolean hasNewMessages(MaildirFolder) 新規メッセージがあるか調べる．

Folder[] listFolders(MaildirFolder) 指定されたフォルダに属するサブフォルダを得る．

ArrayList createMessagesList(MaildirFolder) 指定されたフォルダに属するメッセージのリストを得る．

このクラスは Maidir プロバイダの内部でのみ利用され，JavaMail API を通して直接参照・利用されることはない．また上記の三クラスは，ファイルシステム上のディレクトリやファイルに対して直接操作を行わず，必ずこの MaildirManager クラスに属するメソッドを通して行う．これは，第 4.2 節で説明する，Maildir プロバイダのメッセージをファイルシステム上に保存する操作や，ファイルシステム上に保存されたメッセージに対する参照・移動等の操作の適切さを検査するに

あたり，ファイルシステムに対する操作を一つのクラスにまとめることで，検査しやすいものとするためである．

4.2 JML を用いた Maildir プロバイダの実装の検査

4.2.1 Maildir プロバイダに求められる性質

一般的にソフトウエアは仕様に基づいて構成され，仕様に規定された通りに動作することが期待される．しかし，ソフトウエアが仕様に従い動作するように実装されていなかったり，仕様通りに動作するように実装したつもりであっても，実装が不適切であることが原因で動作が仕様を逸脱してしまう場合がある．我々が開発した Maildir プロバイダは，第 4.1 節で述べたように，JavaMail API のためのサービスプロバイダとして提供され，JavaMail API が定めるメッセージストアに対する抽象的な操作を，Maildir 型式のディレクトリを扱うように具象化したものとなっている．したがって，この Maidlir プロバイダの実装が JavaMail API を通して問題なく利用できるためには，これがメッセージストアに対する抽象的操作についての JavaMail API が定める仕様が規定する振舞いに従う必要がある．

また，この Maildir プロバイダは電子メイルメッセージを扱うソフトウエアであり，コミュニケーションの道具として利用されるものである．そのため，利用者の意図に反して操作中のメッセージが壊れたり失われない事が強く求められる．この性質は JavaMail API の仕様として記述されていないが，常識的に考えて Maildir プロバイダが満たしてほしい性質である．以上をまとめると，Maildir プロバイダの実装に期待される性質は以下のようになる．

a. Maildir プロバイダの実装が JavaMail API の定める振舞いに従っており，Maildir プロバイダによってファイルシステム上に保存されたメッセージや処理中のメッセージが破壊・紛失される事が無い．

また，ソフトウエアが仕様通りに動作するだけでなく，OS やハードウエアといったソフトウエアが動作する環境の異常が原因でソフトウエアの動作が突然停止してしまうような場合であっても，操作中のデータが破壊されないことが期待される．Maildir プロバイダの場合，例えば，あるメッセージをフォルダへ格納する作

業を行っている最中にアプリケーションが突然停止した場合であっても，そのフォルダが破壊され既に格納されたメッセージが取り出せなくなることがないことを保証したい．つまり，上記 (a) に加え，以下の性質も，Maildir プロバイダの実装に期待される．

b. Maildir プロバイダのコードの実行中にアプリケーションの実行が停止しても，操作中のフォルダの整合性が失われることが無い．

4.2.2 Maildir プロバイダの実装の検査方法

我々が実装した Maildir プロバイダの信頼性向上のために，Maildir プロバイダの実装の検査を行った．検査の内容は，第 4.2.1 節で述べた，Maildir プロバイダに求められる性質 (a), (b) の二つである．Maildir プロバイダが性質 (a), (b) を満たすことの検査は，どちらも図 4.3 に示す手順に従う．Maildir プロバイダの検査には，ま

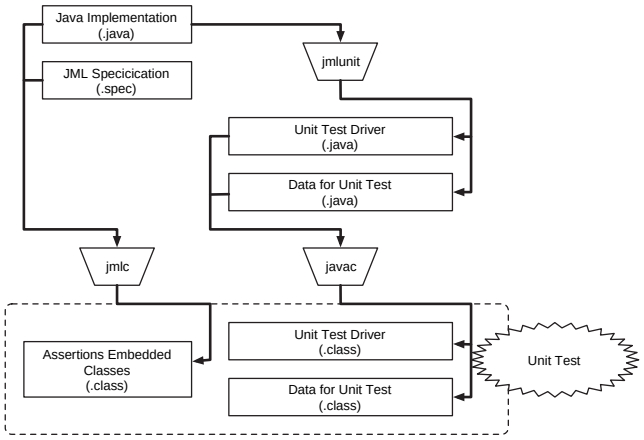

図 4.3: JML を利用したユニットテストの手順

ず，どちらの性質の検査の場合でも第 2.2 節で説明した JML を利用し，Maildir プロバイダのコードに対して DbC に基づき表明の記述を体系的に与えた (図の JML Specification)．ここで指定した表明の記述の詳細については第 4.2.3 節および第 4.2.4 節で述べる．次に，この JML による表明の記述を Maildir プロバイダのソースコードと共に JML コンパイラ `jmlc` を用いて実行時表明検査のためのコードが埋め込まれたクラスファイルにコンパイルした (図の Assertions Embedded Classes)．更に，

Maildir プロバイダのソースコードから `jmlunit` を利用して JUnit のためのテストケースのテンプレートを生成しこれにテストデータを与え (図の Unit Test Driver, Data for Unit Test)，jmlc で生成した実行時表明検査のためのコードが埋め込まれたクラスファイルを対象とした単体テストを行い，表明違反の検出を行った．テストケースのテンプレートには，個々のメソッドの事前条件を満たすテストデータを生成するコードを埋め込んだ．

Maildir プロバイダのプログラム全体を余すこと無く検査するために，カバレッジ検査ツール jcoverage[10] を利用して，事前条件違反の検査のためのコードブロックを除く全コードブロックを実行するように，単体テストのための初期状態を複数選択した．

4.2.3 振舞サブタイプ関係に基づく検査

この節では，第 4.2.1 節で示した Maildir プロバイダに期待される性質 (a)「Maildir プロバイダの実装が JavaMail API の定める振舞いに従っており，Maildir プロバイダによってファイルシステム上に保存されたメッセージや処理中のメッセージが破壊・紛失される事が無い.」についての，JML を用いた検査手法について述べる．

Maildir プロバイダが実装するクラスである `MaildirStore`, `MaildirFolder`, `MaildirMessage` は，JavaMail API の抽象層で定義されるクラス `Store`，`Folder`, `MimeMessage` を継承し実装されているため，インターフェイスが正しく継承されていることは明らかである．従って Maildir プロバイダが JavaMail API の仕様に従っていることを示すには，Maildir プロバイダの実装の振舞いが JavaMail API が想定している振舞いに従うこと示せばよい．つまり，Maildir プロバイダが実装するクラスと JavaMail API の抽象層で定義されるクラスの間に振舞サブタイプ関係が成り立つことを示せばよいことになる．ここで，振舞サブタイプ (behavioral subtyping)[15] とは，あるクラスのインスタンスをそのクラス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意味する．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，と言い換えることもできる．

この検査のために，JavaMail API の抽象層で定義されるクラスの持つ各メソッ

ドの振舞いを JML を用いて事前条件・事後条件として記述し，実行時検査で違反を検出することにした．具体的な作業の流れは，

1. JavaMail API の抽象層で定義されるクラスの自然言語で書かれた仕様を元に，その振舞いを JML を用いて表明として記述し，
2. Maildir プロバイダの実装と，JML による表明の記述を JML ツールを用いてコンパイルし，仕様が強制されたバイトコードを生成し，
3. 実際にプログラムを実行して違反を検出する

となる．

4.2.4 保存されたメッセージの一貫性の保証

この節では，第 4.2.1 節で示した Maildir プロバイダに期待される性質 (b)「Maildir プロバイダのコードの実行中にアプリケーションの実行が停止しても，操作中のフォルダの整合性が失われることが無い.」についての，JML を用いた検査手法について述べる．

Maildir プロバイダは，Maildir フォルダ形式のディレクトリに対する実際の操作をクラス `MaildirManager` を通して行う．このクラスは，Maildir フォルダに対するメッセージの追加・削除・取得，メッセージのフラグの変更等の機能を提供するメソッドを持つ．Maildir プロバイダが性質 (a) を満たすことを調べるには，クラス `MaildirManager` が性質 (a) を満たすことを検査すればよい．

この，クラス `MaildirManager` の検査のために，このクラスが持つメソッドそれぞれに対し，削除や追加といった操作の対象となったメッセエージを除く，その他全てのフォルダ中に存在するメッセージが，メソッドの実行前後で変化しないことを，事後条件として JML を用いて記述した．更に，性質 (b) を検査するために，ファイルシステムに対する操作を，１メソッド内ではたかだか１回しか行わないようにメソッドを分割した (図 4.4，4.5)．これは，あるメソッドの実行中にファイルシステムに対する操作を 2 度以上行う場合，それらの操作の間に Maildir フォルダの一貫性が一時的に崩れる可能性があり，JML を用いて DbC に基づいた表明の記述方式ではこのような状態に関する表明を記述できないためである．こ

のようにメソッド分割を行うことで，クラス MaildirManager が持つ全てのメソッドが，フォルダに対するメッセージの追加・削除等で扱われるメッセージを除いたフォルダに保存されているメッセージを破壊・紛失しないことを検査した.

```
1  ...
2  class MaildirManager {
3   /*@ public behavior
4      @ requires message!=null;
5      @ ensures \result!=null
6      @ && \result.exists()
7      @ && \result.isFile()
8      @ && this.inFolder(\result, "new")
9      @ && this.getContentsOf(\result).equals(
10     @    this.getContentOf(message));
11     @ ...
12     @*/
13   public File putMessage(Message message) throws ... {
14    File tmpFile = // create tmpFile
15    // write message to tmpFile
16    File newFile = // move tmpFile to newFile
17    return newFile;
18   }
19   ...
20   /*@ public pure boolean int inFolder(File file,
21         String name); *@/
22   //@ public pure File newFileOf(File file);
23   //@ public pure byte[] getContentOf(File file);
24   //@ public pure byte[] getContetOf(Message message);
25   }
```

図 4.4: MaildirManager.putMessage(Message) メソッド (分割前)

4.2.5 結果

これまでに述べてきた方法に従い，Maildir プロバイダのコードの検査を行った結果，幾つかの実装上の問題点を発見することができた．例えば，フォルダの位置を指定するために利用される URL 名と，それに対応するパスの間の変換時のエス

ケープ処理を 2 重に行っていた点，フォルダに格納されたメッセージのインデックスの扱いの誤り (インデックスは 1 以上，`int Folder.getMessageCount()` の値以下の値であり，0 以上 `int Folder.getMessageCount()` の値未満ではない) 等の他，JavaMail API が提供するクラス `URLName` の実装上の問題を発見できた．実際の検査作業は，Maildir プロバイダの開発の中でインクリメンタルに行われたため，JML で表明を書く段階で Maildir プロバイダの実装上の問題に気づくことが多く，単体テストの段階で問題が発見されることは稀であった．この記述の規模であるが，Maildir プロバイダの Java コードが 2,500 行であるのに対し，JML による記述はこの Java コードを除いて 3,500 行であった．

```
1  ...
2  class MaildirManager {
3   /*@ ... 分割前と同じ ... @*/
4   public File putMessage(Message message) throws ... {
5    File tmpFile = this.createNewFile();
6    this.writeMessageToFile(tmpFile, message);
7    return this.moveTmpToNew(tmpFile);
8   }
9
10  /*@ private behavior
11    @ ensures \result!=null && !\result.exists()
12    @  && this.inFolder(\result, "tmp");
13    @ ...
14    @*/
15
16  public File createNewFile() throws ...;
17  /*@ private behavior
18    @ requires file != null && !file.exists() && message != null;
19    @ ensures file.exist() && file.isFile()
20    @  && this.getContentsOf(file).equals(
21    @    this.getContentsOf(message));
22    @ ...;
23    @*/
24  private void writeMessageToFile(File file,
25    Message msg) throws ...;
26
27 /*@ private behavior
28   @ requires tmpFile!=null && tmpFile.exists() && tmpFile.isFile()
29   @  && this.inFolder(tmpFile, "tmp")
30   @  && !inFolder(this.newFileOf(tmpFile), "new");
31   @ ensures !tmpFile.exists()
32   @  && \result!=null && \result.exists() && \results.isFile()
33   @  && \old(this.getContentsOf(tmpFile)).equals(
34   @    this.getContentsOf(\result));
35   @ ...
36   @*/
37   private File moveTmpToNew(File tmpFile) throws ...;
38  ...
39  }
```

図 4.5: `MaildirManager.putMessage(Message)` メソッド (分割後)

第 5 章

アスペクト指向的な仕様記述のモジュール化方式

本章では，契約による設計に基づく仕様記述について，記述量の増加が一貫性のある仕様を書くことを難しくする事を説明する．記述された仕様が持つ構造上の特徴として，記述対象であるクラスやインターフェイスが幾つかの側面から捉える事ができ，個々の側面に関する仕様が複数の事前条件・事後条件の指定を横断することを述べ，これらを独立したモジュールとして記述できるようにするための，アスペクト指向の適用について述べる．

5.1 DbC に基づく表明の記述の問題点

第 2.2 節では，Java のための振舞インターフェース仕様記述言語 JML の紹介をし，第 4 章では，Java で電子メイルメッセージを扱うためのライブラリである Maildir プロバイダの検査について述べた．この JML を利用して，Java のクラスやインター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，クラスやインター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伴い，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 がインター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ーフェースに対して仕様を記述する場合，例えば図 5.1 にあるように，メソッド毎に表明を一つずつ指定する．それらは，そのメソッドの属するクラスやインター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

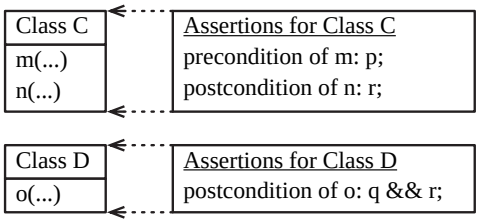

図 5.1: DbC に基づく表明の記述

ような形の仕様を JML で記述することはできない．

- 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ー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ースの仕様を一つにまとめてモジュール化する．

そのため，例えばクラスやインターフェースの仕様が幾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，個々の小さな仕様毎に別々の表明の記述のモジュールを作成し，クラスやインターフェースの仕様をこれらの合成として指定する事ができない．具体的には，図 5.2 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，これらを一つのクラス 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．

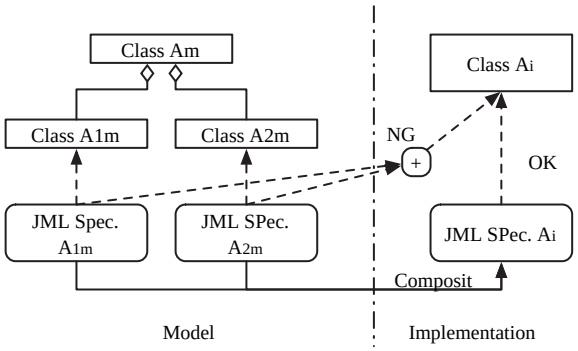

図 5.2: JML を利用した場合の仕様の分割記述ができない例

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ー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．

5.2 DbC に基づく表明の記述に現れる横断的側面

第**??**節で述べた，大規模なクラスやインターフェースに対して 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，大規模な仕様を幾つかの小さなモジュールに分割して記述する事で，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無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．

振舞の多面性　仕様記述の対象であるクラスやインターフェースの振舞は，いくつかの独立した側面として別々に捉える事ができる場合がある．このクラスやイ

ンターフェースの振舞は，これらの側面の合成として捉える事ができる．例えば，第**??**節の図 5.2 に示した例では，実装におけるクラス Class A_i の振舞は，モデルにおけるクラス Class A_{1m} と Class A_{2m} のそれぞれに対応する振舞を合成したものとなる．一方，図 5.3 では，クラス Class C の振舞が，側面 Aspect A, B, C の合成として捉える事ができる場合を表す.

横断的側面の存在　クラスやインターフェースの振舞のそれぞれの側面に対する表明の記述は，クラスやインター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 5.3 では，クラス Class C に属する振舞の側面 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ーフェースの振舞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 5.3 では，クラス Class C のメソッド m1(...) の振舞が，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.

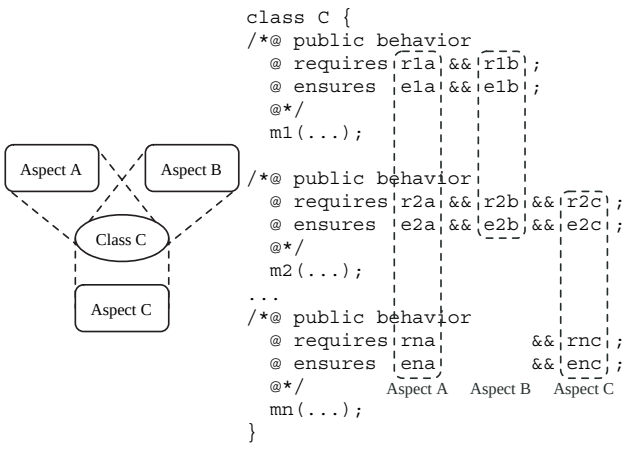

図 5.3: JML の記述に現れる横断的側面

5.3 DbC に基づく表明の記述へのアスペクト指向の適用

第**??**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二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[1]を表す．本機構では，クラスやインターフェースで定義されるメソッド (又はコンストラクタ) の呼び出し時点及びメソッド (又はコンストラクタ) 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．例えば図 5.4 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` 及びクラス 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プログラムの状態の参照を行う．図 5.4 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

[1] 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．

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) 時点及びそのメソッド本体の実行時点をまず選択し，更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，更に，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二つの種類がある (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事が仮定され，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 5.4 では，Advice A1 が “precondition” の指定を持つ事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” の指定を持つ事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．更に，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一つの側面としてモジュール化する．図 5.4 では，アドバイス Advice A1 と Advice A2 を一つの表明アスペクト Assertion Aspect A にモジュール化し，更にアドバイス 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事で，仕様の記述対象であるクラスやインターフェース

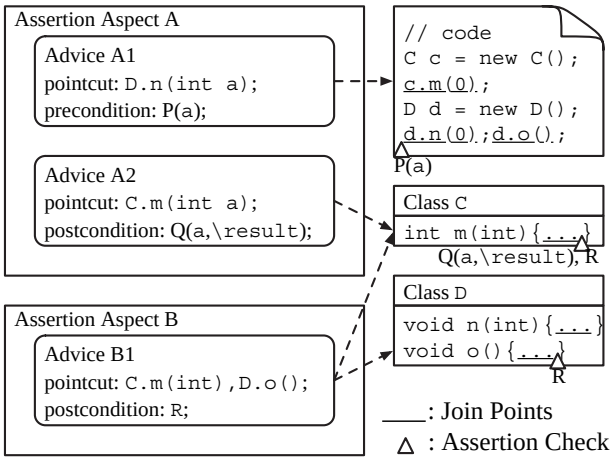

図 5.4: 表明アスペクトを用いた表明のモジュール化の例

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事を可能とする．更に，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一つのアドバイスとして指定する事も可能とする．このモジュール化機構を利用する事で，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事ができる．

第 6 章

アスペクト指向的な仕様記述言語

本章では，アスペクト指向的なモジュール記述を可能とする DbC に基づく Java のための表明記述言語 Moxa の説明を行う．

6.1 概要

Moxa は Java のためのアスペクト指向振舞インターフェイス仕様記述言語である．Moxa は JML を拡張した言語となっており，Moxa が提供するアスペクト指向的な表明の記述型式に加え，JML が提供する，クラスやインターフェイスに対して表明を与える型式もサポートする．

6.2 言語の定義

本節ではアスペクト指向振舞インターフェイス仕様記述言語 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 回以上の繰り返し
- A|B: A, B のどちらか一方を選択

6.2.1 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二つの種類があり，それぞれメソッド呼び出しが行われる時点及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 (第 6.2.2 節参照) として記述される．

6.2.2 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ーフェースに属する，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．更に，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* 型又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全てを選択する．

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二つの機能を併せ持つ点に注意が必要である．

6.2.3 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全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 と記述する事で事前条件 r を，ensures e と記述する事で事後条件 $r \supset e$ を，更に signals（*Type* [*Id*]）s と記述する事で，例外時事後条件として，送出された例外の型を ex とすると，$r \supset ((ex$ instanceof *Type*$) \supset s)$ を指定する．事後条件及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 が二回以上指定された場合，それら全ての requires の条件を論理積で結んだものと等価である．つまり，以下の二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 が一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 が二回以上指定された場合，全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 { ... }
```

更に，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，一つ目の 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，それだけでなく二つ目の 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，例外時事後条件が一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．

二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事ができる．以下に二つのアドバイスを 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．従って，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 ex とすると，

$$\begin{aligned} & (r1 \supset ((ex \texttt{ instanceof } E1) \supset s1)) \\ \wedge \quad & (r2 \supset ((ex \texttt{ instanceof } E2) \supset s2)) \end{aligned}$$

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 ex)
  @     (r1==>((ex instanceof E1)==>s1))
  @  && (r2==>((ex 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．

6.2.4 表明アスペクト

表明アスペクトは，複数のアドバイスを一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物となる．

```
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一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．

6.3 アスペクト指向的な仕様記述言語の処理系

moxa2jml は，Moxa で記述された表明アスペクトを元に JML の仕様を生成する処理系である．この処理系は，入力として Moxa で記述された表明アスペクト及び，JML で記述されたインターフェイス振舞仕様を読み込み，表明アスペクトと

して指定されてた全てのアドバイスを，ポイントカットで指定された位置に対する表明の記述とする JML の記述として出力する．

6.3.1 Moxa による仕様記述

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一つのアドバイスから指定する事ができる．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．例えば，以下のように記述する事で，クラス C に

属するメソッド foo() と，クラス D に属すメソッド bar() に対して共通の事前条件 r を一度に指定する事ができる．

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

継承の扱い

あるクラスを継承し，メソッドの振舞を変更する場合，振舞サブタイプ (behavioral subtyping) 関係 [15] 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事を表すクラス間の関係である．クラス C と D が，C をスーパー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無く動作する．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，と言い換えることもできる．

Moxa では次のように，二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();
```

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．例えば以下のように，メソッド C.foo() に対し，二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，一つのメソッドに表明を指定するアドバイスが異なる二つの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 r2 の評価が r1 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();
}
```

6.4 実験的記述と評価

6.4.1 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．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 (第 6.4.2 節) と変更・修正の容易さ (第 6.4.3 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を規定するインター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, 付録 B 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).

6.4.2 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 6.1 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 のクラスやインター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` の振舞における幾つかの側面を，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 (`Service-Spec_isConnected`)，オブジェクトの名前 (`ServiceSpecgetURLName`)，その他 (`ServiceSpec`) の三つの側面を記述し，更にクラス `Store` に対する表明アスペクトとして，クラス `Service` の三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`, `Store-Spec`) 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 ではこれらを一つのアドバイスとして記述できるだめである.

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，一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一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.

6.4.3　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 6.2 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() の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 boolean Service.isConnected() は，これら

表 6.1: JML と Moxa による仕様記述の規模の比較

	JML		Moxa	
	Service	Store	Service	Store
モジュール (JML の仕様, 表明アスペクト) の数	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

のクラスのインスタンスの状態の一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伴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無いものも含まれる．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) の場合は，修正の波及箇所が，表 6.2 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全てとなっている (表 6.1 参照)．これは，仕様の記述対象であるクラスやインター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 B) 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ーフェースの振舞を，それが持つ側面に基づき幾つかの独立した

表 6.2: 仕様の変更に伴う修正の波及範囲の比較

	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

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を表す表明アスペクト (`StoreSpec_isConnected`) に局所化されている．更に，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，一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．

6.4.4 結論

第 6.4.2 節及び第 6.4.3 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言える．

- 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事ができる．
- 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事ができる．

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．

第 7 章

まとめ

本章では，考察，今後の課題について述べ，更に関連研究との比較について述べる．

7.1 考察

本論文では，アスペクト指向にもとづく表明記述の新しい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 の設計および実現方式について述べた．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もとづいてデザインされたものである．

アスペクト指向仕様記述言語 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伴う表明記述の大規模化と複雑さを押さえることができる．さらに，同じ視点にたって記述された表明の記述を一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よくすることに貢献する．

契約による設計 (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 [16] 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．本研究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．

Moxa を利用し，オブジェクトの持つ異なる側面を複数の表明アスペクトに分割して記述することにより，各表明アスペクトに記述される表明が表現する構造が明確になり，その結果，表明アスペクトをオブジェクトやオブジェクト群の状態をある側面から見た状態遷移の記述であると捉えることができる場合がある．このような場合，以下の手順に従い表明アスペクトから状態遷移モデルを生成する．

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．事前条件の記述は，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．

遷移の抽出: 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．

例えば，図 7.1 の Moxa による表明アスペクトの記述の中の `Folder_state` は，8–36 行目が事前条件を，37–68 行目が事後条件を表すアドバイスとなっており，まず，事前条件を表すアドバイスから，

- `getSize().isConnected() && !exists()`
- `getSize().isConnected() && exists()`
- `getSize().isConnected() && exists() && !isOpen()`
- `getSize().isConnected() && exists() && isOpen()`

という四つの論理式が得られ，これらの重なりを取り除くことにより,(T, F, _), (T, T, F), (T, T, T) という三つの状態が得られる (これらは，それぞれ，式 `getSize().isConnected()`, `exists()`, `isOpen()` の値の三つ組に対応しており，_ は，任意の真偽値を表している)．また，事後条件を表すアドバイスから, 各アドバイス

の持つポイントカットで指定されるメソッドをアクションとする状態遷移を決定できる (図 7.2). これを利用して，表明アスペクトのモデルチェックを行うことを考えている．表明はプログラムの正しさを検査するために記述するのだが，プログラムが大規模になるにつれ，表明の記述も複雑になり，その整合性を確認することが難しくなる．そこで，表明アスペクトから状態遷移モデルを取り出し，その状態遷移モデルに対しモデルチェックを行い，表明の記述の整合性を検査したい．このために，状態遷移モデルへ変換可能な表明の記述の形とその変換方法を考える必要がある．

7.2 今後の課題

今後の課題を以下に示す.

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及びアスペクトを利用してモジュール化が行われる．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対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 では表明アスペクトとして別々に記述できた幾つ化の側面を合成した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事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元に，Moxa の改良を行うことを今後の課題とする．

7.3　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 [9]，Diotalevi の方法 [4] 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．

Pipa[22] は，JML の対象言語を Java から AspectJ へと拡張した言語である．Pipa では，JML と同じく Java のクラスやインターフェイスに対して表明を指定することができる事に加え，AspectJ のアドバイスやイントロダクションに対しても表明を指定することができる．しかし一方，この言語も JML と同様に，表明の記述のモジュール化の単位は対象言語である AspectJ のモジュール構造に依存する形で指定しなければ成らず，表明の大規模化・複雑化に対応できない．これに対し，Pipa が JML を拡張した方法を参考に現在の Moxa を拡張し，AspectJ のアドバイスやイントロダクションとして組み込まれるコードの実行直前・直後をジョインポイントとして選択できるようにすることで，AspectJ のモジュール化の単位から独立した表明の記述のモジュール化を可能とする事ができる．この，Moxa の対象言語と

して AspectJ を許すようにするための拡張は，今後の課題である．

```
...
public spec Folder_state {
 Store Folder.getStore();
 boolean Store.isConnected();
 boolean Folder.exists() throws MessagingException;
 boolean Folder.isOpen();

/*@ public behavior
  @ requires this.getStore().isConnected();
  @*/
 String Folder.getName();
 String Folder.getFullName();
 URLName Folder.getURLName();
 ...
/*@ public behavior
  @ requires this.getStore().isConnected() && this.exists();
  @*/
 int Folder.getMessageCount() throws MessagingException;
 ...
/*@ public behavior
  @ requires this.getStore().isConnected() && !this.exists();
  @*/
 boolean Folder.create(int type) throws MessagingException;
 ...
/*@ public behavior
  @ requires this.getStore().isConnected() && this.exists() && !this.isOpen();
  @*/
 boolean Folder.delete(boolean rescue) throws MessagingException;
 void Folder.open(int mode) throws MessagingException;
 ...
/*@ public behavior
  @ requires this.getStore().isConnected() && this.exists() && this.isOpen();
  @*/
 Message Folder.getMessage(int msgnum) throws MessagingException;
 void File.close(boolean expunge) throws MessagingException;
  ...
/*@ public behavior
  @ ensures this.getStore()==\old(this.getStore())
  @  && this.exists()==\old(this.exists())
  @  && this.isOpen()==\old(this.isOpen());
  @*/
 String Folder.getName();
 String Folder.getFullName();
 URLName Folder.getURLName();
 int Folder.getMessageCount() throws MessagingException;
 Message Folder.getMessage(int msgnum) throws MessagingException;
 ...
/*@ public behavior
  @ ensures this.getStore()==\old(this.getStore())
  @  && this.exists() && this.isOpen();
  @*/
 void Folder.open(int mode) throws MessagingException;
 ...
/*@ public behavior
  @ ensures this.getStore()==\old(this.getStore()) && this.exists() && !this.isOpen();
  @*/
 void File.close(boolean expunge) throws MessagingException;
 ...
/*@ public behavior
  @ ensures this.getStore()==\old(this.getStore()) && !this.exists();
  @*/
 boolean Folder.delete(boolean rescue) throws MessagingException;
 ...
/*@ public behavior
  @ ensures this.getStore()==\old(this.getStore()) && this.exists();
  @*/
 boolean Folder.create(int type) throws MessagingException;
 ...
}
```

図 7.1: Moxa による `Folder` クラスの仕様の記述の一部

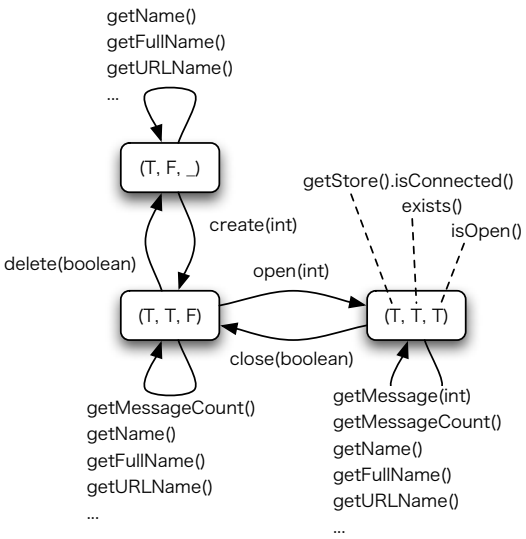

図 7.2: Maildir プロバイダの `Folder` クラスの `Folder_state` 表明アスペクトを状態遷移で表したもの

参考文献

[1] D. J. Bernstein. maildir – directory for incoming mail messages. http://www.qmail.org/man/man5/maildir.html.

[2] D. J. Bernstein. qmail: Second most popular mta on the internet. http://www.qmail.org/top.html.

[3] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report Research Report 159, Compaq Systems Research Center, December 1998.

[4] Filippo Diotalevi. Contract enforcement with aop. http://www-106.ibm.com/developerworks/java/library/j-ceaop/.

[5] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[6] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *IEEE Transactions on Software Engineering*, Vol. 27(2), pp. 1–25, 2001.

[7] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. Conference on Programming Language Design and Implementation (PLDI '2002), pp. 234–245. ACM, 2002.

[8] Erich Gamma and Kent Beck. JUnit, testing resources for extreme programing. http://www.junit.org/.

[9] 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.

[10] jcovrage ltd. jcoverage. http://www.jcoverage.com/.

[11] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP 2001: Object-Oriented Programming*, Vol. 2072 of *Lecture Notes in Computer Science*, pp. 327–355, 2001.

[12] Joseph R. Kiniry and David R. Cok. Esc/java2: Uniting esc/java and jml: Progress and issues in building and using esc/java2 and a report on a case study involving the use of esc/java2 to verify portions of an internet voting tally system. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, Vol. 3362 of *Lecture Notes in Computer Science*, pp. 108–128, 2005.

[13] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and William Harvey, editors, *Behavioral Specifications for Businesses and Systems*, chapter 12, pp. 175–188. Kluwer Academic Press, 1999.

[14] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Esc/java user ’s manual. Technical Report Technical Note 2000-002, Compaq SRC, October 2000.

[15] Barbara Liskov and Jannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programing Languages and Systems*, Vol. 16, No. 6, pp. 1811–1841, November 1994.

[16] Bertrand Meyer. Applying “Design by contract”. *IEEE Computer*, Vol. 25, No. 10, pp. 40–51, October 1992.

[17] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. Pvs: Combining specification, proof checking, and model checking. In *Computer Aided Verification*, Vol. 1102 of *Lecture Notes in Computer Science*, pp. 411–414, 1996.

[18] Etsuya Shibayama, Shigeki Hagihara, Shinya Nisizaki, Kenjiro Taura, and Takuo Watanabe. AnZenMail: A secure and certified e-mail system. In Mitsuhiro Okada, Benjamin Pierce, Andre Scedrov, Hideyuki Tokuda, and Akinori Yonezawa, editors, *Software Security: Theories and Systems*, Vol. 2609 of *Lecture Notes in Computer Science*, pp. 201–216. Springer-Verlag, February 2003.

[19] Sun Microsystems Inc. JavaMail API. http://java.sun.com/products/javamail/.

[20] Joachim van den Berg and Bart Jacobs. The loop compiler for java and jml. In *TACAS'01*, Vol. 2031 of *Lecture Notes in Computer Science*, pp. 299–312, 2001.

[21] 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).

[22] J. Zhao and M. Rinard. Pipa: A behavioral interface specification language for aspectj. In *Proc. Fundamental Approaches to Software Engineering (FASE'2003)*, No. 2621 in svlncs, pp. 150–165, April 2003.

本研究に関する発表論文

[1] 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, 2B-4, 2003, 9.

[2] 山田聖, 渡部卓雄, “アスペクト指向的なモジュール記述を可能とする仕様記述言語”, 日本ソフトウェア科学会第 21 回大会論文集, 2C-3, 2004, 9.

[3] 山田聖, 渡部卓雄, “契約による設計を支援する表明記述のアスペクト指向的モジュール化方式”, 第 51 回プログラミング研究会 (PRO04-3), 2004, 10.

[4] 山田聖, 渡部卓雄, “アスペクト指向的な表明のモジュール化”, 第 11 回ソフトウェア工学の基礎ワークショップ (FOSE2004), pp 29–39, 2004, 11.

[5] 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”, 第 52 回プログラミング研究会 (PRO04-4), 2005, 1.

[6] Kiyoshi Yamada, Takuo Watanabe, “An Aspect-Oriented Approach to Modular Behavioral Specification of Java Component”, The IASTED International Conference on Software Engineering 2005(SE2005), 2005, 2.

第 A章

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 *);
```

```
  @     signals  (AuthenticationFailedException) !this.isConnected();
  @                                              // subclassof MessagingException
  @     signals  (MessagingException)            !this.isConnected();
  @*/
public void connect(String host, String user, String password)
    throws MessagingException;

/*@ public behavior
  @
  @     requires !this.isConnected();
  @     ensures  this.isConnected();
  @     ensures  (* open ConnectionEvent is delivered *);
  @     signals  (AuthenticationFailedException) !this.isConnected();
  @                                              // subclassof MessagingException
  @     signals  (MessagingException)            !this.isConnected();
  @*/
public void connect(String host, int port, String user, String password)
    throws MessagingException;

/*@ public behavior
  @     requires this.isConnected();
  @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
  @     signals (MessagingException)
  @               URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
  @     ensures !this.isConnected();
  @     ensures (* close ConnectionEvent is delivered *);
  @     signals (MessagingException) !this.isConnected();
  @*/
public synchronized void close() throws MessagingException;

/*@ public behavior
  @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
  @     assignable \nothing;
  @*/
public /*@ pure @*/
boolean isConnected();

//-------------------------------------------------------------------------
/*@ protected behavior
  @     requires !this.isConnected();
  @     ensures this.isConnected() == \old(this.isConnected());
  @     signals (AuthenticationFailedException)
  @               this.isConnected() == \old(this.isConnected());
  @     signals (MessagingException)
  @               this.isConnected() == \old(this.isConnected());
  @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
  @     signals (AuthenticationFailedException)
  @                 URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
  @     signals (MessagingException)
  @                 URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
  @*/
protected boolean protocolConnect(
    String host,
    int port,
    String user,
    String password)
    throws MessagingException; // , AuthenticationFailedException

/*@ protected behavior
  @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
  @     ensures  this.isConnected() == connected;
  @*/
protected void setConnected(boolean connected);

//=========================================================================
// URLName Management
//=========================================================================

/*@ public behavior
  @     ensures this.isConnected() == \old(this.isConnected());
```

```
  @     ensures  \result == null || \result.getPassword() == null;
  @*/
public /*@ pure @*/
URLName getURLName();

/*@ protected behavior
  @     ensures this.isConnected() == \old(this.isConnected());
  @*/
protected void setURLName(URLName url);

//=========================================================================
// Event Management
//=========================================================================
/*@ public behavior
  @     requires l != null;
  @     ensures this.isConnected() == \old(this.isConnected());
  @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
  @*/
public synchronized void addConnectionListener(ConnectionListener l);

/*@ public behavior
  @     requires l != null;
  @     ensures this.isConnected() == \old(this.isConnected());
  @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
  @*/
public synchronized void removeConnectionListener(ConnectionListener l);

/*@ protected behavior
  @     requires type == ConnectionEvent.CLOSED
  @           || type == ConnectionEvent.DISCONNECTED
  @           || type == ConnectionEvent.OPENED;
  @     ensures this.isConnected() == \old(this.isConnected());
  @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
  @*/
protected void notifyConnectionListeners(int type);

//=========================================================================
// methods inherit from Object
//=========================================================================
/*@ also public behavior
  @     ensures this.isConnected() == \old(this.isConnected());
  @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
  @     ensures  \result != null;
  @*/
public String toString();

//=========================================================================
// Event Management
//=========================================================================
// private EventQueue q;
// private Object qLock;

/*@ protected behavior
  @     requires   event != null;
  @     requires   vector != null;
  @     ensures this.isConnected() == \old(this.isConnected());
  @     ensures URLName.equals2_model(this.getURLName(), \old(this.getURLName()));
  @*/
protected void queueEvent(MailEvent event, Vector vector);

// private void terminateQueue();

//=========================================================================
// methods inherit from Object
//=========================================================================
/*@ also protected behavior
  @     requires true;
  @     ensures  !this.isConnected();
  @*/
protected void finalize() throws Throwable;
```

```
188 }
```

第 B章

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
```

```
    void Service.connect() throws MessagingException;
    void Service.connect(String, String, String) throws MessagingException;
    void Service.connect(String, int, String, String) throws MessagingException;

    /*@ public behavior
      @  ensures isConnected() == connected;
      @*/
    void Service.setConnected(boolean connected);

    // no changes
    /*@ public behavior
      @   ensures isConnected() == \old(isConnected());
      @*/
    URLName Service.getURLName();
    void Service.setURLName(URLName);
    void Service.addConnectionListener(ConnectionListener);
    void Service.removeConnectionListener(ConnectionListener);
    void Service.notifyConnectionListeners(int);
    void Service.queueEvent(MailEvent, Vector);
    void Service.toString();

    // constructor
    /*@ public behavior
      @  ensures !isConnected();
      @*/
    Service.new(Session, URLName);
}

//=======================================
// for public URLName Service.getURLName()
//=======================================
spec ServiceSpec_getURLName {
//---requires

//---ensures
    // changes

    // no changes
    /*@ public behavior
      @  requires MaildirUtility.checkEqualsURLNames(getURLName(), \old(getURLName()));
      @*/
     boolean Service.protocolConnect(String, int, String, String) throws MessagingException;
     boolean Service.isConnected();
     void Service.setConnected(boolean);
     void Service.close() throws MessagingException;
     void Service.addConnectionListener(ConnectionListener);
     void Service.removeConnectionListener(ConnectionListener);
     void Service.notifyConnectionListeners(int);
     String Service.toString();
     void Service.queueEvent(MailEvent, Vector);
}

//=======================================
// extra preconditions
//=======================================
spec ServiceSpec {
//---requires
    /*@ public behavior
      @  requires session != null && urlname != null;
      @*/
    Service.new(Session sesison,URLName urlname);

    /*@ public behavior
      @  requires l != null;
      @*/
    void Service.addConnectionListener(ConnectionListener l);
    void Service.removeConnectionListener(ConnectionListener l);

    /*@ public behavior
      @  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   }
```