AOP を応用した実用的なソフトウェアモデル検査手法

古賀陽一郎[†1]　田辺良則[†2]

本報告では，ソフトウェアモデル検査に AOP を応用して実現する実用的なソフトウェアモデル検査手法を提案する．提案手法では検査内容に応じて検査対象プログラムを選択的にスタブ化する．スタブコード・検証コード，その他，検査にのみ必要なコードは全てアスペクト側に記述し，検査対象プログラムには一切手を加えない．これにより，ソースコード改変によるリグレッションの発生と検査時の状態爆発問題の軽減を試みる．

A Practical Method for Software Model Checking Using AOP

YOICHIRO KOGA[†1]　YOSHINORI TANABE[†2]

In this paper, we propose a practical method for software model checking using AOP (Aspect Oriented Programming). The proposed method provides selective code insertion/replacement for each test case. We do not make any changes in the original program. Code required only at testing, such as stub or assertion, is implemented on the aspect side. The method contributes to reducing regression and state explosion.

1. はじめに

モデル検査は同時並行に動作するモデルが定義された検査対象に対し，検査対象が取りうる全ての実行経路を網羅的に探索してデッドロック・ライブロックなどの基本的問題が発生しないこと，あるいは与えられた条件を満たすことを確認する手法である．

従来は設計レベルの成果物を検査対象とすることが主体であった[3]が，近年ではソフトウェアそのものを検査対象とする技術が発展している．本報告ではマルチスレッド・プログラミングされたバイトコードを検査対象とし，Java バイトコードを検査対象にしたモデル検査器である JPF（Java PathFinder）を使用する．JPF は JVM 上で動作するプログラム向けのソフトウェアモデル検査器である．デッドロックやライブロック，あるいは通常のテストで検出が難しい同時並行処理に起因する未捕捉の例外を検出することができる．しかし，JPF はネイティブコードを実行することができないという問題を抱えている．検査対象となるプログラムがファイル入出力コード・ネットワークアクセスコード等に含まれるネイティブコード[1]を実行する場合，スタブ化によりネイティブコードの実行を回避しなければ検査を最後まで実行できない．また，スタブ化により JPF を実行可能にしたとしても多くの場合は状態爆発により現実的な時間内に実行完了しない．状態爆発は比較的規模の大きいソフトウェアにモデル検査を適用する際に避けることができない課題である．

本報告では AOP (Aspect Oriented Programming)ツールである AspectJ を導入することで，ソフトウェアモデル検査の実用面での適用を妨げる次の課題の解決・軽減に取り組む．

- 検証のためのスタブ化によるソースコード改変
- 状態爆発による検証空間・検証時間の大幅増加

本報告では，まず 2 節でソースコードモデル検査器，及びソフトウェアモデル検査技術が抱える実用面での課題について整理する．3 節では 2 節で整理した課題を解決するための AOP 技術を応用したアプローチについて説明する．4 節では，提案したアプローチを用いたスタブコード，検証コードの実装例について紹介する，5 節では，実際に開発中のソフトウェアのコードを使った実験により，現状の状態削減効果を示す．6 節では本報告に関連する既存の研究を紹介する．7 節で全体のまとめを行い，今後の課題と方向性について説明する．

2. 従来の課題

本節ではソースコードを対象にしたモデル検査が抱える実用面の課題を整理する．

2.1 スタブ化による設計の複雑化

JPF はネイティブコードを実行することができない．実行経路にファイル入出力やネットワークアクセス，外部コマンドの呼び出しなどのネイティブコードが含まれると，そこで実行を停止してしまう．実行時に起こりうるあらゆる状態の組み合わせを網羅して未発見の例外を検出するモデル検査にとって，これは致命的な問題である．

検査を最後まで続行するためには，ネイティブコードを JPF が実行可能なスタブコードに置き換えなければならない．従来は，検査時はスタブコード，実行時はネイティブ

†1 (株)東芝 ソフトウェア技術センター
Software Engineering Center, Toshiba Corporation
†2 国立情報学研究所
National Institute of Informatics

1 マシン語で記述されたプログラム.ソフトウェアが稼働するハードウェアのプラットフォーム体系に合わせた専用のコードとなる．

ⓒ2012 Information Processing Society of Japan

コードを使用するよう検査元のプログラム変更することでJPF に対応していた．ネイティブコードを実行するオリジナルクラスの上に抽象クラスを作り，そのクラスからスタブを実装するクラスを派生させる等の手法がとられる．

こうした設計・実装面の変更は検査対象を必要以上に複雑にする．更に変更に起因する不具合（リグレッション）を作り込む可能性もあるため，スタブ化による検査対象の改変はモデル検査の適用を難しくする原因の１つとなっている．

2.2 状態爆発

ネイティブコードを全てスタブに置き換えたとしても，JPF による検査が可能とは限らない．多くの場合，状態数が巨大であるために現実的な時間内に検証が終わらないという問題に直面する．この問題は設計レベルを対象にするモデル検査ツールと比べると，検査対象の元々の規模が大きいため顕著に現れる．

本報告で行った実験では検査対象プログラムの規模は 3000LOC 程度だが，状態数は最大 900,000 以上に達する．この場合の検査時間は，検査を実行する計算機の性能にもよるが， 30 分以上を要する．

現実的な時間内に検査を終えるためには，検証対象となるソースコードをなるべく単純化し，検査内容に関係の無いコードをそぎ落とさなければならない．しかし，こうした対応も検査対象の変更を必要とし，変更に起因した不具合（リグレッション）を埋め込んでしまう．

2.3 モデル検査のためのソースコード改変の限界

2.1 節，2.2 節で説明した課題から，従来の手法では JPF による検査を現実的な時間内に終えるためにソースコード本体に手を加えるプロセスを避けることができない．モデル検査のための主な改修内容はスタブ化と不要なコードのそぎ落としであるが，この改修はソースコード改変によるリグレッションの他にも下記の問題を抱えている．

この方法ではスタブ化を行う階層が固定的になる．一度決定した階層でスタブ化を行うと，スタブ化した階層以下の構造は検査時に考慮されない．本来のプログラムの振る舞いを正確に再現するためには，そぎ落としによる本体コードの改変を最小限に抑え，ネイティブコードの実行直前でスタブ化することが望ましいが，こうした対応は状態爆発を容易に引き起こす．

また改変の結果，検査元プログラムの振る舞いが変化する可能性があり，検査結果の信頼性が低下する．検査結果の正しさを説明するためには，改変の前後で検査内容に関するプログラムの振る舞いに変化が無いことを確認する必要があるが，これは現状人手で確認するしかなく，余分なコストが発生する．

こうした問題から，大規模なソフトウェアに対してそのままモデル検査を適用するのは現実的に難しい状況である．

3. AOP を応用したソフトウェアデル検証

本節では AspectJ を用いて検証対象プログラムを JPF で実行可能にする手法について述べる．従来のスタブ化手法に換えて AspectJ を利用することで，スタブコードや検証用の特殊なコードを，ソースコードを改変することなく簡単に追加・置き換えることができるようになる．

本手法では AspectJ をスタブコードや検査用コードの追加・置き換えに特化した形で利用する．検査対象を直接改変する変わりに，アスペクトに改変内容を記述する．これを AspectJ の提供する処理の差し込み・置き換え機能を用いて検査対象と合成することで，検査対象を JPF で実行可能にする．

アスペクトは検査の観点に応じて複数作成し，検査内容に応じて最適なアスペクトを使用することで実行経路，及び検査時に探索する状態数を削減する．

AspectJ によるスタブコード・検証コードの追加・置き換えによって元々の検査対象の基本構造が壊れることはないため，本手法を使った検査の信頼性は高い．

3.1 利用ツール

AOP ツールとソースコードモデル検査器はそれぞれ以下のツールを使用した．

表 1 利用ツール

AOP ツール	AspectJ 1.6
モデル検査器	Java PathFinder v6.0 (rev 618+)

3.1.1 JPF

JPF は Java バイトコードを直接モデル検査するツールである．その本体は全ての状態を記憶し，バックトレースを可能とする JVM である. この特性により全ての実行経路を網羅的に探索し，複数のスレッドが絡みあって発生するデッドロックや null 参照といった通常のテストでは見つけづらい問題を検出することができる．

3.1.2 AspectJ

AspectJ は Java 用の AOP 拡張ツールである．「.aj」という拡張子のファイルに独自の文法規則を持つアスペクトコードを記述する．アスペクトコードは Java ソースコードと共に，専用のコンパイラ ajc を使ってコンパイルされる．ajc は Java コンパイラによってコンパイルされたバイトコードに.aj ファイルに記述したアスペクトコードを差し込む．出力ファイルは JVM[2]が読み取り可能な Java バイトコードであるため，JPF で実行可能である．

図 ，図 ，図 に AspectJ を用いた典型的なアスペクトコードとその実行結果を示す．

2 Java 仮想マシン. Java バイトコードを実行するソフトウェア.

```
1  package sample;
2
3  public class Sample {
4
5      public static void main(String[] args) {
6          Sample sample = new Sample();
7          System.out.println(sample.hoge());
8      }
9
10     public String hoge() {
11         return "hoge";
12     }
13
14 }
15
```

このメソッドの実行前にhogecut()が挿入される.

図 1 サンプルコード

```
1  package sample;
2
3  public aspect SampleAspect {
4      pointcut hogecut() : execution(public String Sample.hoge(..));
5
6      before() : hogecut() {
7          System.out.println("hello from aspect.");
8      }
9
10 }
```

アスペクトコードの挿入位置の指定.
挿入位置は「実行前」
挿入するコード.

図 2 サンプルアスペクトコード

```
<terminated> Sample (1) [Java Application] /usr/local/jdk1.6.0_29/bin/java

hello from aspect.
hoge
```

図 3 サンプルコードの実行結果

図 に示す通り，アスペクトコードは Java の文法に従って記述する挿し込む処理本体と，AspectJ 独自の文法に従って記述する差し込み先を指定するコードで構成される．

以降，本報告では表 1 に掲載する AOP の用語を用いる．

表 1 AOP 用語集

用語	意味
アスペクト	ポイントカットとアドバイスの組み合わせを指定するモジュール
ジョインポイント	アドバイスの実行を織り込み可能なコード上の位置
ポイントカット	プログラム中の全ジョインポイントからアドバイスを織り込むポイントを特定するための絞り込み条件
アドバイス	スレッドの実行がポイントカットで指定されたジョインポイントに到達したときに実行されるコード
ウィーブ	アドバイスをプログラム中に埋め込むこと

3.2 基本アイデア

図 1 に AOP を使ったソースコードモデル検査の基本アイデアを示す．

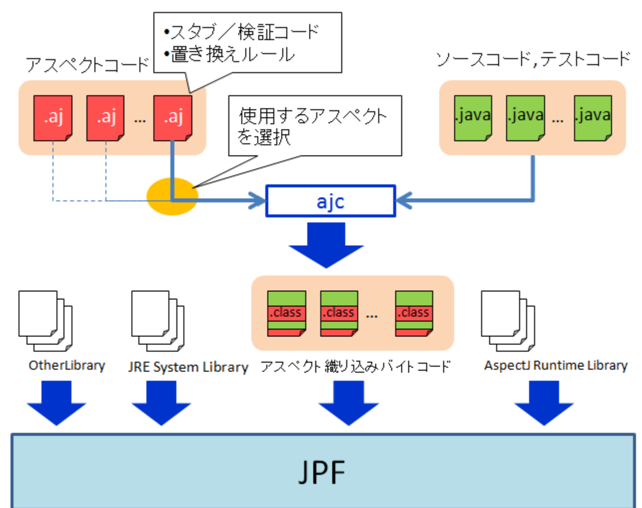

図 1 基本アイデア

本手法ではスタブ・検証コードは全てアスペクト側（.aj ファイル）に記述する．検査対象となるソースコードには手を加えない．

アスペクトは複数作成する．中身はスタブコード・検証用コード（アドバイス）と置き換え先や追加先を指定するルール（ポイントカット）のセットである．

スタブコードは，主にネイティブコードの置き換えと検査内容に関係のない処理をスキップするコードを実装する．

検査内容に関係がないが必須の処理である場合には，スキップする変わりに原子化[3]のコードを処理の実行前後に埋め込む．検査内容によっては，処理の過程を飛ばして最終的に得られる結果だけを非決定的に返却するといった JPF の機能を活用した高度な振る舞いを実装する場合もある．

検証用コードにはアサート文を実装し，条件に合致した時に JPF がそこで実行を停止するようにする．

アスペクトセレクタは ajc を使ったビルドの直前に，複数のアスペクトの中から検査対象に織り込む最適なアスペクトを選択する．

ビルドによって生成されたバイトコードは JRE System Library，AspectJ Runtime Library，及び検査対象となるソースコード･テストコードが参照する外部ライブラリと共に，JPF によって実行される． この時，JPF に渡されたバイトコードはネイティブコードを全てスタブコードに置き換えているため，JPF は途中で止まることなく最後まで検査を続けることができる．

3.3 アスペクトセレクタ

検査対象になるプログラムは大規模で複雑なものが多く，DB への接続・ファイル/ネットワーク入出力・OS コマンドの実行などに含まれる複数のネイティブコードを実行する．こうしたネイティブコードを，AspectJ を用いて呼び

3 まとまった処理の固まりを 1 つの処理とみなし，状態を作らないこと

ⓒ2012 Information Processing Society of Japan

出しの直前にスタブコードに置き換えても状態爆発により現実的な時間内に検査が終わらない場合が多い．

しかし，実際には実行経路上の全てのコードを限なく探索する必要はない．検査内容に応じて実行時の組み合わせを網羅する必要のあるコードとそうでないコードが存在する．検査に必要なコードだけを残し，検査に必要のないコードをスキップまたは原子化すれば，検査時に探索する状態空間を大幅に削減することができる．

アスペクトセレクタは，検査を実行するに当たって必要なコードを残し，それ以外をスキップまたは原子化する最適なアスペクトを複数個用意したアスペクトの中から選択する．

図 2 に使用するアスペクトによって実行コードが変化するイメージを示す．

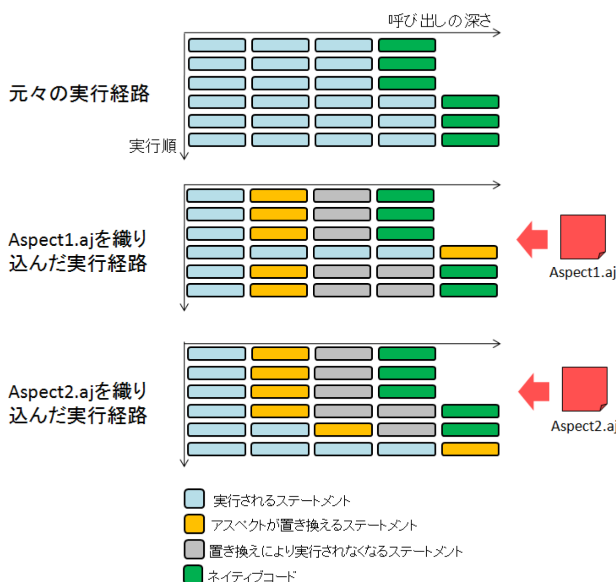

図 2 織り込むアスペクトによる実行経路の変化

アスペクトは検査の観点別に用意する．観点を共有する検査項目は，組み合わせを網羅する必要のあるコードとそうでないコードが共通する場合が多い．こうした検査項目には同じアスペクトを適用し，作成するアスペクトの数を削減する．

ソースコードを直接スタブ化する手法では，スタブ化する部分の実装が複雑になるために，本手法のように検査内容に応じて実行コードを柔軟に切り替えるのは困難であった．AspectJ のポイントカットの記法を使って必要な部分だけをピンポイントで置き換えることができ，かつ置き換え部の実装を複雑にしないこの特性は AOP を利用する大きな利点である．

4. スタブ/検証コードの実装

本節では AspectJ を使ったスタブコード・検証コードの実装例を紹介する．

4.1 スタブコードの実装

4.1.1 空処理への置き換え

ネイティブコードを実行するメソッドを何もしない空のメソッドに置き換えるのは従来のスタブ化手法でもよく行われる．これを AspectJ で置き換えると次のようになる．

```
/**
 * ディレクトリ生成処理を以下の内容に置き換える.
 */
pointcut mkdir() : execution(void *.createDirectory());

void around() : mkdir() {
    // do nothing.
}

public class Workspace implements Serializable {

    private static final long serialVersionUID = 1L;

    private String path;

    public Workspace(String path) {
        this.path = path;
        createDirectory();
    }

    private void createDirectory() {
        FileUtil.mkdirs(path);
    }

    public String getPath() {
        return path;
    }
}
```

図 3 空処理への置き換え

この例では，AspectJ の持つポイントカットの記法を用いて検査対象に存在する全ての createDirectory という名前のメソッドを空処理に置き換えている．

4.1.2 仮の返り値の返却

本来ネイティブコードが返すはずであった返り値をアスペクト側に実装したスタブコードに返させるようにする場合は次のように記述する．この例ではファイルまたはデータベースで ID を管理する代わりに，アスペクト側に static 変数を定義して ID を管理させている．

```
/**
 * ID生成処理を以下の内容に置き換える.
 * @return
 */
private static Integer id = 0;

pointcut idfile() : execution(Integer *.create*Id());

Integer around() : idfile() {
    id = id + 1;
    return id;
}

/**
 * データ収集設定のIDを生成する.
 */
private static synchronized Integer createJobSettingId() {
    return JobSettingIdFile.getInstance().getNextId();
}
```

図 4 仮の返り値の返却

4.1.3 コードの原子化

ある実行経路上の一部のコードが検査内容に関係が無い場合，その部分については JPF に状態を作らないよう指示することで，余計な状態を削減できる．

ⓒ2012 Information Processing Society of Japan

図 5の例では，updateContainer という複雑で多くの状態を作るメソッドを呼び出す前後に JPF に状態を作らないよう指示するコードを追加挿入している．

```
pointcut createJob() : call(List<Job> *..ScheduledJobContainer.updateContainer());

before() : createJob() {
    Verify.beginAtomic();
}

after() : createJob() {
    Verify.endAtomic();
}
```

検査内容に関係が無いが実行ステートメント数が多く複雑なメソッドの呼び出し前後に原子化するコードを挿入．

アスペクトコード

```
public class ScheduledJobHandler {

    public ScheduledJobContainer jobContainer;

    public ScheduledJobHandler(ScheduledJobContainer jobContainer) {
        this.jobContainer = jobContainer;
    }

    public synchronized void hundleSchduledJob() {
        List<Job> jobs = null;
        while(true) {
            try {
                jobs = jobContainer.updateContainer();
            } catch (Exception e1) {
                // 例外が発生したときは空のジョブリストを生成する.
                jobs = new ArrayList<Job>();
            }

            for(Job job : jobs) {
                ScheduledJobExecutor executor = new ScheduledJobExecutor(job, this);
                SystemResource.getInstance().getThreadpool().submit(executor);
            }
```

ソースコード

図 5 原子化による状態数の削減例

4.2 検証コードの実装例

4.2.1 検証コードへの置き換え

プログラムが特定条件を満たしていることを確認する検証コードを追加した例を図 6に示す．

別スレッドが実行，エラー発生のフラグを立てる．

```
private static int n = 0;

private boolean error = false;

pointcut openlog() :
    execution(public void jp.co.toshiba.provisio.job.framework
            .log.JobLogger.open(..));

void around() : openlog() {
    if(n++ > 1) {
        error = true;
    }
}
```

```
pointcut checkerror() :
    execution(private void jp.co.toshiba.provisio.job.framework
            .control.ScheduledJobHandler.checkError());

/**
 * ログファイルへの書き込みに同時に２つ以上のスレッドが入ったかどうかを確認.
 */
void around() : checkerror() {
    assert !error;
}
```

```
public synchronized void hundleSchduledJob() {
    List<Job> jobs = null;
    while(true) {
        try {
            jobs = jobContainer.updateContainer();
        } catch (Exception e1) {
            // 例外が発生したときは空のジョブリストを生成する.
            jobs = new ArrayList<Job>();
        }

        for(Job job : jobs) {
            ScheduledJobExecutor executor = new ScheduledJobExecutor(job, this);
            SystemResource.getInstance().getThreadpool().submit(executor);
        }
        try {
            wait();
            checkError();
        } catch (InterruptedException e) {
            break;
        }
    }
}
```

図 6 検証コードへの置き換え

この例では，本来は何もしない checkError メソッドを「エラー発生のフラグが立てられていないかを確認する」コードに置き換えている．エラー発生のフラグは別スレッドが実行するアドバイス openlog 内で立てられる．

スレッドプールなど，エラーが発生してもメインスレッド通知せずに処理を進行させせるプログラムに対して有効な方法である．

4.2.2 メソッド呼び出し前後の追加処理挿入

この方法は主にメソッド実行前後にシステムが特定の条件をクリアしているかを確認するために使用するもので，[1]でも言及されている．

```
/**
 * 同じIDが生成されることは無いことを検証する.
 */
pointcut verifyIdIsNotSame(DataCollectSetting dcSetting) :
    execution(public jp.co.toshiba.provisio.job
            .framework.DataCollectSetting.new(..)) && this(dcSetting);

after(DataCollectSetting dcSetting) returning() : verifyIdIsNotSame(dcSetting) {
    assert SystemResource.getInstance()
        .getDataCollectSettings().get(dcSetting.getId()) == null;
}
```

図 7 メソッド実行後の検証コード挿入

図 7の例では，インスタンス生成直後に，生成したインスタンスのもつ ID がシステムに既に登録されているかどうかをチェックするアサーションコードが挿入される．

4.3 アスペクトセレクタの実装

今回は単純にテストコードのパッケージを検査内容別に分割することでアスペクトセレクタを実装した．

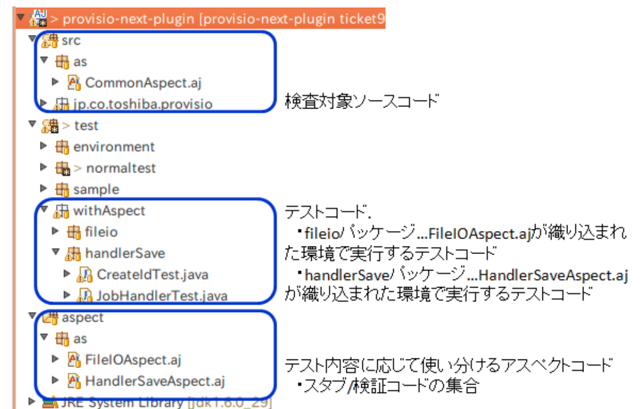

図 8 実装したアスペクトセレクタ

JPF を実行する Ant[4]ターゲットを検査の観点別に用意している．各ターゲットは実行されると，まずテスト実行に必要なアスペクトを aspect パッケージから，ビルドする src パッケージにコピーする．その後 ajc によるビルドを実行，生成されたバイトコードを JPF に与えて検査を実行する．

4.4 アスペクトによる置き換えパターンの変化

使用するアスペクトによって検査対象のソースコードの置き換えが変化する事例を紹介する．織り込むアスペクトは 2 種類（A，B）用意した．

図 9にアスペクト織り込み前のソースコードを示す．

4 Java 用のメイクツール．AspectJ，JPF は ant 用のターゲットを提供しており，AspectJ によるビルドから JPF による検査実行までを自動化できる．

```
public static class Saver {

    /**
     * データ収集設定をファイル保存する
     * @param jobSettingId
     * @param saveDirPath
     * @throws Exception
     */
    public void save(Integer jobSettingId, String saveDirPath)
            throws Exception {
        DataCollectSetting jobSetting =
                SystemResource.getInstance().findJobSetting(jobSettingId);
        final String configFilePath =
                FileUtil.combinePaths("saveDirPath", "job", jobSettingId.toString());
        ObjectSaverResource.getObjectSerializer(configFilePath).save(jobSetting);
    }
}

public class ObjectSerializer implements IObjectSaver {

    private final String filePath;

    public ObjectSerializer(String filePath) {
        this.filePath = filePath;
    }

    @Override
    public synchronized void save(Object object)
            throws Exception {
        if(filePath.length() == 0) {
            return;
        }
        FileUtil.saveObject(new File(filePath), object);
    }
```

call

図 9 オリジナルソースコード

アスペクト A を図 10 に示す

```
public aspect JobHandlerAspect {

    pointcut save(Integer jobSettingId, String saveDirPath) :
        execution(void *..*Handler.Saver.save(..))
            && args(jobSettingId, saveDirPath);

    /**
     * HandlerのレベルでI保存処理を置き換える.
     * <li> 保存処理を無効にする.
     * <li> jobSettingが存在することを確認する.
     * <br><br>
     * <p>
     * ファイル保存の排他制御を検証するときは除外してください.
     * </p>
     */
    void around(Integer jobSettingId, String saveDirPath) :
        save(jobSettingId, saveDirPath) {
        DataCollectSetting jobSetting =
                SystemResource.getInstance().findJobSetting(jobSettingId);
        assert jobSetting != null;
    }
```

図 10 アスペクト A

アスペクト A を織り込んだ時に JPF によって実行されるコードを図 11 に示す．Saver#save メソッドで実行するコードが「保存されるオブジェクトはいかなる実行経路でも必ず存在する」ことを確認するコードに置き換わる．

```
public static class Saver {

    /**
     * データ収集設定をファイル保存する
     * @param jobSettingId
     * @param saveDirPath
     * @throws Exception
     */
    public void save(Integer jobSettingId, String saveDirPath)
            throws Exception {
        DataCollectSetting jobSetting =
                SystemResource.getInstance().findJobSetting(jobSettingId);
        assert jobSetting != null;
    }
}
```

図 11 アスペクト A を織り込んだソースコード

続いてアスペクト B を図 12 に示す．

```
public aspect FileIOAspect {

    pointcut verifySerialize(Object obj) :
        execution(void *..ObjectSerializer.save*(..))
            && args(obj);

    private static Integer n = 0;

    void around(Object obj) : verifySerialize(obj) {
        n = n + 1;
        assert n <= 1;
        n = n - 1;
    }

}
```

図 12 アスペクト B

アスペクト B を織り込んだ時に JPF によって実行されるコードを図 13 に示す．ObjectSerializer#save メソッドで実行するコードが「ファイル保存へのクリティカルセクションへは一度に 1 つのスレッドしか入れない」ことを確認するコードに置き換わる．

```
public static class Saver {

    /**
     * データ収集設定をファイル保存する
     * @param jobSettingId
     * @param saveDirPath
     * @throws Exception
     */
    public void save(Integer jobSettingId, String saveDirPath)
            throws Exception {
        DataCollectSetting jobSetting =
                SystemResource.getInstance().findJobSetting(jobSettingId);
        final String configFilePath =
                FileUtil.combinePaths("saveDirPath", "job", jobSettingId.toString());
        ObjectSaverResource.getObjectSerializer(configFilePath).save(jobSetting);
    }
}

public class ObjectSerializer implements IObjectSaver {

    private final String filePath;

    public ObjectSerializer(String filePath) {
        this.filePath = filePath;
    }

    private static Integer n = 0;

    @Override
    public synchronized void save(Object object)
            throws Exception {
        n = n + 1;
        assert n <= 1;
        n = n - 1;
    }
```

call

図 13 アスペクト B を織り込んだソースコード

本例での置き換えの階層は 1 つしか違わないが，生成される状態数及び深度は表 2 に示すだけの違いが出る．

表 2 置き換え階層による状態数と深度の変化

アスペクト	状態数			深度
	new	visited	backtracked	
A	6295	7499	13793	33
B	49719	50446	100164	49

尚，本節で紹介した置き換え後のコードは簡潔さのため，ajc が挿入するコードを省略している．実際に検証されるコードには，アドバイスで実装したコードの呼び出し前後に AspectJ Runtime Libarary のコードが存在する．

4.5 テストコードの実装

検証コードは図 14 に示すように JUnit4 フレームワークを利用して記述する．

ⓒ2012 Information Processing Society of Japan

```
@Test
public void 下流ジョブを2つ並列に動かす() throws Exception {
    if(verifyNoPropertyViolation("")) {
        final Project project = new Project();
        final DataCollectSetting setting =
                JobSettingHandler.getCreater().create(project,
                SampleParalelRoutineJob.class);

        MockSession threadExecuter = new MockSession() {
            @Override
            public void doSomething() throws Exception {
                JobSettingHandler.startJobExecution();
            }
        };
        threadExecuter.start();

        final ScheduleRequest request = new ScheduleRequest();
        request.setStartDate(DateUtil.createDate(2012, 1, 10));
        request.setEndDate(DateUtil.createDate(2012, 1, 11));
        JobSettingHandler.getScheduler().reserve(setting.getId(), request);

        threadExecuter.interrupt();

        SystemResource.getInstance().getThreadpool().shutdown();
    }
}
```

図 14　テストコードサンプル

検証コードは JUnit で書かれたテストコードと認識されるため，Eclipse などの統合開発ツールを利用していれば JUnit テストとして実行できる．

5. 状態数の比較

本節では本手法の適用前後において JPF が生成した状態数の比較結果を示す．

5.1 検査対象

Java で記述した Web アプリケーションプログラムからタスクの並列実行モジュールを抜き出して検査対象のベースとした．このモジュールはファイル入出力，DB アクセス，ネットワークアクセスなどに数種類のネイティブコードを含む．

このモジュールに対し，従来のスタブ化手法を用いて JPF による検査を可能にしたプログラムを検査対象 1 とする．全ての検査に 1 つのプログラムで対応するため，スタブ化の位置はネイティブコードを呼び出す直前に固定した．

本報告で提案した手法を用いて JPF による検査を可能にしたプログラムを検査対象 2 とする．

尚，検査内容は同一でありテストコードは同じ物を使用している．

5.2 検査内容

実施した検査の概要を表 3 に示す．

表 3 検査の概要

検査 1	設定ファイルの保存は排他制御される．
検査 2	タスクの作成時に割り当てられる ID が重複しない．
検査 3	タスクを並列に実行できる．
検査 4	サブタスクの同一ログファイルへの書き込みは排他制御される．
検査 5	サブタスクを並列に実行できる．（非決定的に正常終了/エラー終了を発生させる）

5.3 手法適用前後の状態数比較

表 4 に本手法適用前後の状態数と検査時間の比較結果を示す．不具合が検出されず検査が正常終了した時の状態数であり，JPF が出力する 3 種類の状態数のうち new（新規に作成された状態）の数を掲載している．

表 4 手法適用前後の状態数の変化

	検査対象 1		検査対象 2	
	状態数	検査時間（分:秒）	状態数	検査時間（分:秒）
検査 1	85	00:01	25	00:01
検査 2	5868	00:06	6295	00:06
検査 3	43458	01:47	28450	01:04
検査 4	234254	08:22	37692	01:21
検査 5	906136	33:58	157724	05:30

検査 3,4,5 については検査対象 1 と検査対象 2 の生成する状態数に大きな変化が見られた．検査に関係ない一連の処理の原子化を AspectJ により実行の前後に埋め込んだことで状態数が大幅に減少している．

原子化を行わない場合，検査対象 2 の状態数は検査対象 1 の状態数を若干上回る．これは ajc が埋め込むコードを加えた検査対象 2 のコード量が検査対象 1 のコード量を上回るためと考えられる．尚，同様の要因で検査 2 については検査対象 1 のほうが生成される状態数が少ない結果となった．

6. 関連研究

AOP とモデル検査技術を組み合わせて使用する研究は AOP が一般的に知られるようになった 2000 年代から何例か試みられている．

[1]の研究では AspectJ を使って記述されたプログラムを，モデル検査技術を使って検証する手法を提案している．AOP のメカニズムを応用したモデル検査のフレームワークについても提案しており，検査内容をアスペクトとして検査対象から分離することで，検査対象を変更することなくモデル検査を実行できる長所に触れている．

[2]の研究では，promela の検査モデルに AOP の概念を組み込むため，AspectJ の持つジョインポイントの文法をモデル検査用に拡張した文法を提案している．

これらの研究に対し，本研究ではモデル検査の実用性を高める目的で AspectJ を利用する．JPF によるモデル検査の実用性を損なっている要因を，検査のためとソースコード改変と状態数の爆発であると捉え，AspectJ を用いた検査のための改変の本体からの分離と，検査内容に応じた選択的置換えによりこれらの問題を解消または軽減することに重

ⓒ2012 Information Processing Society of Japan

点を置く.

7. まとめと今後の課題

本報告では AOP を応用したソフトウェアモデル検査手法について紹介した.

スタブコード・検証コードをアスペクトコードとして記述することで，ソースコード改変を行わずとも JPF による検査実行が可能であることを示した．AOP の特性である柔軟なコードの置換え・挿入の機能を活用することで，検査内容別にスタブの適用位置を調整，検査に関係の無い処理を原子化した．これにより検査時の状態数を削減可能であることを確認した.

今回の手法ではスタブコード・検証コードの置き換え時に AspectJ が追加するコードに対して JPF が状態を作る問題を解消していない．よって，スタブコードへの置き換えによって削減したコード数が, AspectJ が追加したコード数より少ない場合は状態数の削減効果を得られない.

本手法では AspectJ をモデル検査のためのスタブコードへの置き換えと検証コードの追加に特化して利用している．従って今後の本手法の発展の方向性として， AspectJ が生成するコードについては状態を作らないよう JPF を改修することで，置き換えによって削減できるコード量が少ない場合でも状態数を削減可能にすることが考えられる.

謝辞

本研究の一部は，JSPS 科研費 23240003 の助成を受けたものです．本研究の一部は，国立情報学研究所トップエスイープロジェクトの活動として実施されました.

参考文献

[1]鵜林尚靖, 玉井哲雄：アスペクト指向プログラミングへのモデル検査手法の適用，情報処理学会論文誌，Vol.43, No.6, pp.1598-1609, 2002
[2]大野真一郎，岸知二：モデル検査のためのアスペクト指向でのモデル記述支援環境,情報処理学会研究報告, Vol.2008-SE-159, pp.41-48, 2008
[3]吉岡 信和，青木 利晃，田原 康：SPIN による設計モデル検証，近代科学社, 2008
[4] E. Clarke, O. Grumberg, and D. Peled：*Model Checking,* MIT Press, 1999
[5]W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda：*Model checking programs*, Automated Software Engineering Journal*,* Vol.10, No.2, pp.203-232, 2003