ユニットテストを利用したアスペクトの衝突の検出方法の提案

A Framework for Detecting Aspect Interferences with Unit Tests

平野広樹† 福田浩章‡
Hiroki Hirano Hiroaki Fukuda‡
† 芝浦工業大学大学院　電気電子情報工学専攻
‡ 芝浦工業大学 工学部 情報工学科
† Department of Electrical Engineering and Computer Science. Graduate School of Engineering, Shibaura Institute of Technology
‡ Department of Computer Science. Shibaura Institute of Technology

要旨

現在, 横断的関心事をモジュール化するためにアスペクト指向プログラミングが提案されている. 横断的関心事は特定のプログラムの実行時点にアドバイスとして織り込まれる. この時, ある実行時点に複数のアスペクトが織り込まれると, ソフトウェア開発者の意図に反した実行結果が導出されることがある. これはアスペクトの衝突の問題としてよく知られ, これまでに様々な研究がされてきた. しかし, それらの研究では衝突が起こる可能性のある他のアスペクトについて開発者が知っている必要がある.

そこで本研究では, 衝突が起こる可能性のある他のアスペクトを意識することなく, アスペクトの衝突を検出するために, アドバイスの実行が開発者の意図通りに正しく行われることをテストする枠組みを提案する.

1. はじめに

はじめに本研究の背景としてアスペクト指向プログラミングについて説明する. その後アスペクト指向プログラミングの問題点であるアスペクトの衝突について説明し, 既存研究とその問題点について説明する.

1.1. アスペクト指向プログラミング

ソフトウェアは継続的に開発が行われるため, 保守性や再利用性を高めることが重要になってくる. そのためソフトウェア工学では長年に渡り様々なソフトウェア開発方法論が研究されてきた[1]. その中で, カプセル化や継承の機能を導入することで, 保守性や再利用性を高めることに成功した方法論としてオブジェクト指向プログラミングが存在する. しかし, オブジェクト指向プログラミングでは横断的関心事と呼ばれる要素をモジュール化することが難しい. 横断的関心事とはロギング機能のように 1 つのモジュールに収まりきらず, 複数のモジュールに散在される機能である. アスペクト指向プログラミングは横断的関心事をアスペクトとして分離し, 再利用できるように 1 箇所にまとめてモジュール化する. そして, モジュール化され 1 箇所にまとめられたアスペクトはポイントカットの記述にしたがって, 既存コードの特定の箇所に織り込まれる. 織り込まれる既存コードはアスペクトに対してベースコードと呼ばれ, メソッド呼び出しなどの織り込みが行われる可能性のある箇所はジョインポイントと呼ばれる. また, 織り込み後に処理が行われるコードをアドバイスと呼ばれ, アスペクトはポイントカットとアドバイスから構成される.

1.2. アスペクト指向プログラミングの問題

アスペクトが単独で織り込まれた場合には問題なく動作するにもかかわらず, 複数のアスペクトがベースコードの同じジョインポイントに織り込まれる場合に互いのアスペクトが影響を及ぼし合い, プログラムが開発者の意図しない動作結果になってしまうことがある. このような問題はアスペクトの衝突と呼ばれる.

1.3. アスペクトの衝突の種類

アスペクトの衝突は以下の 4 種類に分類できる[2].

1. 織り込み時に, あるアスペクトのジョインポイントが, 他のアスペクトによって変化させられる場合
2. 織り込み時に, 静的なプログラムの構造を変化させるアスペクトによって織り込みがあいまいになる場合(織り込みの順番に依存する)
3. 実行時に, あるアスペクトがフィールドや変数を変更する可能性があり, 他のアスペクトの振る舞いに影響を与える場合
4. 実行時に, あるアスペクトがシステムのコントロールフローを変更する可能性があり, 他のアスペクトのジョインポイントに辿りつかなくなる場合

これらのうちプログラムの入力によらずに常に起こる可能性のある衝突は 3 の場合のみであるので, 本研究では 3 の衝突について扱う.

1.4. アスペクトの衝突例

アスペクトの衝突の例として図 1, 2 のアスペクトを考える. 図 1 は User クラスの password フィールドをロギングするアスペクトで, 図 2 は User クラスの password フィールドを暗号化するアスペクトである. この 2 つのアスペクトが同じジョインポイントに織り込まれる時に, ロギングアスペクトが暗号化アスペクトの前に織り込まれる場合, ロギングアスペクトがロギングするフィールドの値は暗号化アスペクトによって暗号化されていない状態の値となる. しかし, ロギングアスペクトが暗号化アスペクトの後に織り込まれる場合, ロギングアスペクトがロギングするフィールドの値は暗号化された値となる.

ロギングアスペクトの開発者がロギングされる値が暗号化された値かどうか判断するには, 暗号化アスペクトがロギングアスペクトの前と後のどちらに織り込まれるのか知っている必要がある. しかし, アスペクトが織り込まれる順序は処理系に依存するため開発者にはわかりづらい. さらに, アスペクト指向プログラミングではアスペクトとベースコードが別々の部分に記述されるため, ロギングアスペクトと暗号化アスペクトの開発者が異なる場合, 開発者が, もう一方のアスペクトの存在に気づきにくい. その結果, アスペクトの衝突が発生する可能性がある.

```
Aspect.new :before, :calls_to => :hello, :for_type
=> :User do |jp, object, *args|
  p "user's password: #{object.password}"
end
```

図 1 ロギングアスペクト

```
Aspect.new :before, :calls_to => :hello, :for_type
=> :User do |jp, object, *args|
  object.password = Digest::SHA1.hexdigest
(object.password)
end
```

図 2 暗号化アスペクト

1.5. 既存研究とその問題点

今までにアスペクトの衝突に関する研究がいくつか行われてきた. [3]では, アスペクトの順番を記述するテストコードを用意する必要がある. そのため, 開発者が織り込まれる全てのアスペクトについて知っている必要がある. アスペクト指向プログラミングでは, アスペクトはポイントカットによって自動的に織り込まれる. そのため, ベースコードやアスペクトの開発者は他のアスペクトの存在を意識する必要がなくなる. しかし, [3]ではその利点が失われてしまう. そこで本研究ではユニットテストの考え方を利用して, 開発するアスペクト以外のアスペクトを意識することなく, アスペクトの衝突を検出する手法を提案する.

2. アプローチ

本研究では, アスペクトの衝突を検出するために, ユニットテストを利用してアスペクトのテストを記述する方法を提案する. 本章ではまずソフトウェアテストについて述べ, それがアスペクトの衝突を検出する方法として適している理由を述べる. その後アスペクトをテストする際の問題点について述べ, アスペクトのテストの実現方法を述べる.

2.1. ソフトウェアテスト

ソフトウェアテストはプログラムが正しく動作することを確認する作業である. プログラムが正しく動作するとはプログラムが開発者の意図通りに動作するということを指す.

アスペクトの衝突とは1.2節で述べたように, プログラムが開発者の意図通りに動作しないことを指す. したがって, アスペクトの衝突を検出するためには, 開発者がプログラムの動作に対して期待する意図を知る必要がある. そこで, その開発者の意図を書きあらわす手段としてソフトウェアテストを利用する.

2.2. ユニットテスト

ソフトウェアテストの 1 つにメソッドなどの小さい単位を対象とし, その単位が仕様を満たしているかどうかをテストするユニットテストと呼ばれる手法が存在する. ユニットテストはメソッドを単位とする場合, メソッドの入力として適当な値を用意し, その入力でメソッドを実行した時の戻り値と, その入力に対して期待する値とを比較し, その 2 つが同値であればテストは通過し, 異なればテストは失敗する.

図3, 4 にユニットテストの例をあげる. 図3 がテストの対象となる User クラスが定義されたコードで, 図4 が User

クラスをテストするテストメソッドが定義された TC_User クラスのコードである．この例では，TC_User クラスの test_get_info メソッドの assert_equal メソッドで'tarou 20 password'という文字列と User クラスの get_info メソッドの戻り値を比較している．

```
class User
 def initialize(name, age, password)
   @name = name
   @age = age
   @password = password
 end
 def get _info
   "#{@name} #{@age} #{@password}"
 end
end
```

図 3 ユーザクラス

```
require 'test/unit'
require 'user.rb'
class TC_User < Test::Unit::TestCase
 def setup
   @tarou = User.new('tarou', 20, 'password')
 end
 def test_get_info
   assert_equal('tarou 20 password', @tarou.get_info)
 end
end
```

図 4 ユーザクラスのテストコード

2.3. アスペクトをテストする際の問題点

アスペクトの衝突を検出するためのテストをしようとする場合，既存のテストフレームワークでは実現が難しい．アスペクトの衝突を検出するためには，アドバイスが実行される時点で，そのアドバイスが利用しているフィールドの値を確認する必要がある．しかし，既存のテストフレームワークの場合，ユニットテストで比較する値はメソッドの戻り値である．そのため，メソッド本体の実行時に，フィールドの値を確認する手段が提供されていない．

2.4. アスペクトのテストの実現方法

本研究ではアスペクトの衝突を検出するために，テスト対象のアスペクトと同じジョインポイントに織り込まれる別のアスペクトを実行し，その後にテスト対象のアスペクトで利用するフィールドの状態を確認する．

2.4.1. アドバイスの実行

テストしたいアスペクトのアドバイスが利用しているフィールドが，そのアドバイスが実行されるまでにどのような値になるかを確かめるために，テストしたいアスペクトと同じジョインポイントに織り込まれる別のアスペクトを実行する．図 5 はアスペクトのテストがどのように行われるかを示している．はじめに，テスト対象以外のアスペクトを取り出す．次にアスペクトのアドバイスを実行し，その後にフィールドの値を確認する．本研究では，テスト対象以外のアスペクトを 1 つずつ切り替えて実行することで，衝突が起こったかどうかだけでなく，衝突が起こった場合にテスト対象のアスペクトと衝突しているアスペクトを識別する．

```
non_test_aspects.each |aspect|
   aspect.call
   assert_equal(field, expected_value)
end
```

図 5 アスペクトのテスト方法

```
def get _info
   "#{@name} #{@age} #{@password}"
   object.password = Digest::SHA1.hexdigest
(object.password)
   p "user's password: #{object.password}"
end
```

図 6 アスペクトの織り込み例

2.4.2. フィールドの状態のテスト

テスト対象以外のアスペクトのアドバイスが実行し終わった箇所に，図 5 の assert_equal メソッドのようにフィールドの値を確認するためのコードを挿入する．この時フィールド名とその時点でのフィールドの期待する値はテストメソッドの記述者が指定する．そのため，織り込まれるアスペクトがどのフィールドを利用しているか知っているアスペクトの開発者が，主にテストメソッドを記述することになる．

2.5. アスペクトのテストの動作例

例えば 1.4 節で示したアスペクトが図 1 の User クラスの get_info メソッドに図 6 のように織り込まれたとする．ここでロギングアスペクトがロギングする値と暗号化アスペクトが暗号化する値は User クラスの password フィールドとする．この時ロギングアスペクトがロギングする password フィールドの値をテストしたい場合は，暗号化アスペクトのアドバイスを実行した時点で password フィールドの値を確認する．この時ロギングアスペクトの開発者が暗号化アスペクトの存在に気づいていない場合でも，暗号化アスペクトが password の値を変更すれば，ロギングアスペクトのテストは失敗するので，アスペクトの衝突を検出することができる．

3. 実装

アスペクト指向プログラミングの実装として代表的な物に Java に対してアスペクト指向プログラミングのための拡張を行った AspectJ というプログラミング言語が存在する．しかし，近年 Web アプリケーション開発などを中心にオブジェクト指向スクリプト言語である Ruby[4]が使用されるようになってきた．そこで，本研究では提案するフレームワークの実装にあたり，アスペクト指向フレームワークとして Ruby の Aquarium[5]を使用し，Ruby の標準ユニットテストフレームワークである Test::Unit ライブラリ[6]を拡張し，アスペクトのテストができるテストメソッドを追加する．

アスペクトのテストメソッドにはテストしたいアスペクトと，そのアスペクトが利用していて状態をテストしたいフィールド名と，そのフィールドがアスペクトのアドバイス実行前にどのような値になっているか期待する値を入力する．Aquarium では織り込みが行われるアドバイスは，ジョインポイントメソッドが属するクラスのクラス変数にリスト形式で保持される．この時，アドバイスは関数として実行可能な Proc 型で格納される．Ruby では Proc 型のオブジェクトは call メソッドの呼び出しで実行することができる．したがって，本研究の実装では，織り込みが行われるアドバイス中の特定のアドバイスを実行する時に，ジョインポイントメソッドが属するクラスのクラス変数からアドバイスを取り出し，call メソッドを呼び出すことで実行する．そして，そのアドバイスの実行後にテスト対象のアスペクトが利用するフィールド値をテストメソッドで入力された期待する値と比較し，値が異なっていれば衝突が発生したと見なす．

4. 関連研究

本研究以外にも，アスペクトの衝突を検出する研究は行われ，様々なアプローチが提案されている．

[7]はシステムの属性がリソースと呼ばれる共通のライブラリに記述される．アスペクトはそのリソースに対して決められた動作を行うことができる．本研究はこの研究のリソースのような全てのアスペクトに関係するようなものは定義しない．

[3]ではアスペクトが織り込まれる順序をベースコードと別に用意し，織り込みが行われた後のベースコードとその順序が記述されたコードを比較することで，アスペクトの衝突を検出する．しかし，この方法だと織り込みが行われる全てのアスペクトを開発者が把握する必要性が生じる．

5. まとめ

アスペクト指向プログラミングでは，アスペクトを同じ部分に織り込むと開発者の意図に反した結果が出力されることがある．これはアスペクトの衝突と呼ばれ，解決するための様々な研究が行われている．しかし，それらの研究では織り込まれる全てのアスペクトを意識する必要があったり，開発者の意図しない順序で織り込みが行われたりすることがある．本研究では 1 つのアスペクトに着目したテストメソッドを書けるようにするフレームワークを提供することにより，開発するアスペクトだけを意識して，他のアスペクトとの衝突を検出できるようにした．

参考文献

[1] P. Naur and B. Randell, (Eds.). Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO (1969) 231pp.

[2] Mehmet Aksit, Arend Rensink, and Tom Staijen. “A Graph-Transformation-Based Simulation Approach for Analysing Aspect Interference on Shared Join Points,” Proceedings of the 8th International Conference on Aspect-Oriented Software Development, Charlottesville, Virginia, USA, March 2009.

[3] 赤堀文隆，櫻井孝平，古宮誠一，アスペクト指向プログラミングのためのアドバイス適用順序をテストする枠組みの提案，電子情報通信学会 知能ソフトウェア工学研究会, 2008.

[4] Ruby: https://www.ruby-lang.org/

[5] library test/unit: http://doc.ruby-lang.org/ja/1.9.2/library/test=2funit.html

[6] Aquarium: http://aquarium.rubyforge.org/

[7] P. E. A. Durr, T. Staijen, L. M. J. Bergmans, and M. Aksit. “Reasoning about semantic conflicts between aspects,” 2nd European Interactive Workshop on Aspects in Software, 2005.