安全に結合可能なアスペクトを提供するためのルール

Rules for Safe Composition of Aspects

一杉 裕志 †、田中 哲 †、渡部 卓雄 ‡

Yuuji ICHISUGI, Akira TANAKA, Takuo WATANABE

†産業技術総合研究所

National Institute of Advanced Industrial Science and Technology

‡国立情報学研究所／東京工業大学

National Institute of Informatics / Tokyo Institute of Technology

Abstract

複数のクラスにまたがるコードを分離して記述できる言語として、アスペクト指向言語がある。独立して開発されたアスペクトであっても、個々のアスペクトが何らかのルールに従って設計されていれば、複数同時に組み合わせて動作させることが可能であると我々は考えている。本論文ではそのようなルールを見いだし検証するための第一歩として、安全に結合可能な mixin を提供するためのルールを検証する方法について述べる。ルールの記述および検証には、Design by Contract や behavioral subtyping の考え方を用いる。

1 背景

プログラムのモジュール化の目的の１つは、個々のモジュールを独立したプログラマーによって開発可能にすることである。独立して開発されたモジュールを結合してより大きなプログラムを構築する時、個々のモジュールは開発時と同じように動作することが望まれる。このことを「**安全な結合が可能である**」と呼ぶ。逆に個々のモジュールを開発するときは、結合後のソースコード全体の知識を持たなくても正しく開発できることが望まれる。このことを「**modular reasoning が可能である**」と呼ぶ。この２つは同じことを別の側面から見た表現である。

抽象データ型をサポートする言語において、モジュール（この場合抽象データ型の定義）を安全に結合可能にするためには、**Design by Contract**[4] を行なえばよい。つまり、「抽象データ型を実装する側は外部仕様を満たすように責任を果たし、抽象データ型を利用する側は外部仕様のみに依存するように責任を果たす」というルールを全てのプログラマーに要請すればよい。抽象データ型の外部仕様は通常、事前条件と事後条件という形で表現される。

静的型と継承のあるオブジェクト指向言語においてモジュール（この場合クラス定義）を安全に結合可能にするためには、もう１つルールの追加が必要になる。「サブクラスのインスタンスは、スーパークラスのインスタンスとしても正しく振る舞わなければならない」というルールである。このルールは「サブクラスはメソッドをオーバーライドする時、事前条件を弱く、事後条件を強くできる」とも表現できる [4]。このとき、スーパークラスとサブクラスの間に **behavioral subtyping** [5] の関係がある、と言う。

複数の mixin を安全に結合可能にするためのルールについて形式的に論じた研究はないが、経験的な知見は得られている。mixin とは、多重継承されることを想定して定義されるクラスである。もし個々の mixin が定義するメソッドに重複がなければ、これらの mixin は安全に結合することができる。逆に、１つのメソッドの振る舞いを複数の mixin がそれぞれオーバーライドして拡張することは普通は危険である。しかし、個々の mixin があるルールにしたがってメソッド拡張をしているならば、mixin は安全に結合することができる。CLOS[7] およびその前身である Flavors[6] という言語は、method combination という機構により、複数の mixin による安全なメソッド拡張を支援している。

アスペクト指向言語 [3] において、アスペクトを安全に結合するにはどうすればよいかという問題は、未解決の問題である。

アスペクト指向と mixin は非常に関連が深い。一般にアスペクト指向言語は１つのクラスの実装を複数に分割する機能を有しており、これは機構的に mixin に近い。例えばアスペクト指向言語の１つである AspectJ[3] では、１つのメソッドの振る舞いを、複数

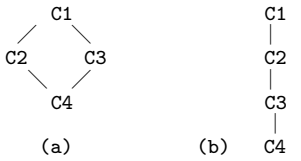

図 1: (a) 4つのクラスの継承関係と、(b) C4 のスーパークラスをリニアライズした結果

のアスペクトから拡張可能である。

本論文では、独立して定義された2つの mixin を安全に結合する（多重継承する）ために必要なルールについて考察する。このルールにより、2つの mixin が安全に1つのメソッドの振る舞いを拡張できるようになる。本論文が提案するルールの記法と検証方法は、多くのアスペクト指向言語に応用できると考えている。

2 考察するクラス階層

本論文では、静的型を持ち、クラスをリニアライズする多重継承機構を持つ仮想的な言語を仮定する。

考察の対象とするのは図 1(a) のような最も単純な、ダイアモンド継承のケースである。クラス C2, C3 は C1 を継承するクラス、C4 は C2, C3 を多重継承するクラスである。C4 自身はメソッドの定義を持たないものとする。

この言語ではクラスはリニアライズされる。C4 のインスタンスの動作は、図 1(b) のように定義されるクラスのインスタンスの動作と同一である。リニアライズのアルゴリズムにはいろいろあるが、今回は、図 1(b) 以外のリニアライズ結果は起き得ないものと仮定する。

この言語では、サブクラスのインスタンスをスーパークラスの型の変数に代入できるものとする。

この言語では、サブクラスはスーパークラスのメソッドをオーバーライドできる。その際、スーパークラスのメソッドを super 呼び出しで呼び出すことができる。super 呼び出しによって実行時にどのクラスのメソッドが呼び出されるかは、リニアライズ結果によって決まる。例えば図 1(b) においては、クラス C3 中の super 呼び出しはクラス C2 のメソッドを呼び出す。

本論文では、説明を簡単にするために、あるオブジェクトのメソッド実行中に、そのオブジェクトの別のメソッドが再帰的に呼び出されることはないものとする。

3 結合性判定基準

この章では「 C2, C3 が安全に結合可能か」を判定する、判定基準を定義する。

判定基準は、2つの条件群からなる。1つはスーパークラスとサブクラスの間に behavioral subtyping が成り立つための条件、もう1つはクラス定義時の super とリニアライズ後の super の振る舞いの間に behavioral subtyping が成り立つための条件である。

ここで、判定基準の定義に用いる記法について説明する。本論文では、クラス C1 とクラス C2 の間に behavioral subtyping の関係が成り立っているという条件を次のように表記する。

$$(C1\ C2) \hookrightarrow (C1)$$

矢印の両辺には、それぞれのクラスのスーパークラス (自分自身を含む) をリニアライズしたリストを書く。上記の条件は、C1 が持つ各メソッドについて、クラス C1, C2 における事前条件をそれぞれ $R1$, $R2$、事後条件をそれぞれ $E1$, $E2$ とすると、以下の論理式が成り立つことと等価である[1] [2]。

$$(R1 \Rightarrow R2) \wedge (R1 \Rightarrow (E2 \Rightarrow E1))$$

この記法を用いて、判定基準を以下に定義する。

まず、スーパークラスとサブクラスの外部仕様の間に behavioral subtyping の関係が成り立っている必要がある。例えば、 C4 は C3 のサブクラスなので、この2つのクラスの間には behavioral subtyping の関係が成り立っている必要がある。この条件は、次のように書ける。

$$(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C3)$$

もう1種類の条件として、クラス定義時の super とリニアライズ後の super の間の behavioral subtyping の関係がある。この条件は super 呼び出しの振る舞いついても modular reasoning をするために必要である。例えば C3 の定義時にプログラマーが super 呼び出しの振る舞いに対して行なう仮定が、C4 のインスタンス内でも成り立っていて欲しい。C3 の定義時の super は C1 であり、 C4 のインスタンス内から見た C3 の super は C2 である。従って、この条件は以下のように書ける。

$$(C1\ C2) \hookrightarrow (C1)$$

以上の2種類の条件をまとめると、図 2 のようになる。これら全ての条件を**結合性判定基準**と呼ぶ。もし

[1] クラス C2 は、 C1 の事前条件 $R1$ が成り立つ場合にだけ事後条件を強くすればよいという点に注意。

[2] [5] で述べられている invariant および constraint については本論文では述べないが、同様の考え方で扱うことができる。

C1 と C2 の関係: $(C1\ C2) \hookrightarrow (C1)$
C1 と C3 の関係: $(C1\ C3) \hookrightarrow (C1)$
C1 と C4 の関係: $(C1\ C2\ C3\ C4) \hookrightarrow (C1)$
C2 と C4 の関係: $(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C2)$
C3 と C4 の関係: $(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C3)$
C2 の定義時の super とリニアライズ後の super の関係:

$$(C1) \hookrightarrow (C1)$$

C3 の定義時の super とリニアライズ後の super の関係:

$$(C1\ C2) \hookrightarrow (C1)$$

図 2: ダイアモンド継承における結合性判定基準

C2 と C3 が別々のプログラマーによって何の制約もなしに実装されたクラスだとしたら、これらの条件が成り立つかどうかは偶然に頼る他はない。しかし、これらのクラスに 5章で述べる制約を守らせることで、結合性判定基準を成り立たせることができる。結合性判定基準が成り立っていれば、 C4 は C2 と C3 を安全に多重継承することができる。言い替えれば、個々のメソッドの実装時にプログラマーが行なった、全てのメソッド呼び出しの振る舞いに関する仮定が、リニアライズ後も成り立つことが保証される。

4 メソッドの外部仕様の記法

この章では、メソッドの外部仕様の記法について説明する。(次の章では、この記法を用いて C1, C2, C3 の外部仕様が満たすべき制約と、その正当性について述べる。)

クラス C1, C2, C3 が導入した内部状態を $s1$、 $s2$、$s3$ とする。(C2 のインスタンスの内部状態は $s1$ と $s2$ の直積である。)

クラス C1 の各メソッドの事前条件はメソッド実行前の状態 $s1$ と引数 p を含む論理式で表される。メソッドの事後条件は、実行前の状態 $s1$ 、実行後の状態 $s1'$ 、引数 p 、返値 r を含む論理式で表される。例としてメソッド m の仕様 $R1, E1$ とそれを満たす 1つの実装を以下に示す。

$$R1(s1, p) \equiv true$$
$$E1(s1, s1', p, r) \equiv (s1' = p) \wedge (r = p)$$

```
class C1 {
  int s1;
  int m(int p) { s1 = p; return p; }
}
```

C2, C3 の外部仕様の記述には、リニアライズを考慮する必要がある。リニアライズの結果によってこれらのクラスの振る舞いは変化するので、その変化のしかたもあらかじめ仕様に含めておかなければならない。リニアライズによって変化するのは super の振る舞いである。そこで、 C2, C3 の外部仕様記述の中に super の振る舞いを、以下のような事前条件 RO、事後条件 EO、状態 sO という記号を使って表現するものとする。ただし sO は、リニアライズによって自分自身と C1 との間に入る他の mixin によって導入された状態である。

$$RO(s1, sO, p)$$
$$EO(s1, s1', sO, sO', p, r)$$

例として C2, C3 におけるメソッド m の仕様とそれを満たす実装を以下に示す。C2 は super 呼び出しをする例、 C3 はしない例である。

$$R2(s1, sO, s2, p) \equiv RO(s1, sO, p)$$
$$E2(s1, s1', sO, sO', s2, s2', p, r)$$
$$\equiv EO(s1, s1', sO, sO', p, r) \wedge (s2' = p)$$

```
class C2 extends C1 {
  int s2;
  int m(int p){ s2 = p; return super.m(p); }
}
```

$$R3(s1, sO, s3, p) \equiv true$$
$$E3(s1, s1', sO, sO', s3, s3', p, r)$$
$$\equiv (s1' = p) \wedge (sO' = sO) \wedge (s3' = s3) \wedge (r = p)$$

```
class C3 extends C1 {
  int m(int p){ s1 = p; return p; }
}
```

リニアライズによって super の振る舞いが確定するので、個々のインスタンスの振る舞いは RO, EO, sO を含まない論理式で表すことができる。一般に、$(C_1\ \cdots\ C_n\ C_{n+1})$ のインスタンスの仕様はクラス C_{n+1} の仕様の RO, EO を $(C_1\ \cdots\ C_n)$ の仕様で置き換えたものである。また、(C) のインスタンスの仕様はクラス C の仕様そのものである。

例えば C2 のインスタンスにおけるメソッド m の事後条件 $E_{(C1C2)}$ は、EO を C1 の事後条件に置き換えることによって以下のようになる。

$$E_{(C1C2)}(s1, s1', s2, s2', p, r)$$
$$\equiv (s1' = p) \wedge (r = p) \wedge (s2' = p)$$

また、 C4 のインスタンスにおけるメソッド m の事後条件 $E_{(C1C2C3C4)}$ は以下のようになる。(C4 自体はメソッドの定義を持たないため、 C3 のメソッドがそのまま継承される。状態 sO は、$s2$ に置き換える。)

$$E_{(C1C2C3C4)}(s1, s1', s2, s2', s3, s3', p, r)$$
$$\equiv (s1' = p) \wedge (s2' = s2) \wedge (s3' = s3) \wedge (r = p)$$

なお、ここで挙げた C1, C2, C3, C4 は、多重継承により不都合を起こす例である。この例では C3 が super 呼び出しを行なっていないため、C2 のメソッドが実行されず、インスタンス変数 s2 の値が更新されないという問題が起きる。実際、これらのクラスの仕様は結合性判定基準を満たしていない。図 2に挙げ

た6つの条件のうち、$(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C2)$ が満たされていない。

5 ルールとその検証

この章では、 C2, C3 が安全に結合可能になるために、それぞれのクラスが満たすべき制約（ルール）について述べ、そのルールを満たすクラスが実際に結合性判定基準を満たすことを述べる。

5.1 フレームワーク側と拡張モジュール側の役割

ここで、 C1 は拡張可能なフレームワークとしての役割、C2, C3 はそれを拡張する拡張モジュールとしての役割を果たすと考えることにする。

結合性判定基準を満たすためのルールはおそらく無限に存在するが、C1 を定義するプログラマーが、各メソッドごとにルールを1つ選んで、宣言するものとする。（宣言は、プログラミング言語が提供する何らかの宣言構文か、自然言語によるドキュメントで行なう。）C2, C3 は、 C1 によって定められたメソッドごとのルールに従って、メソッドをオーバーライドするものとする。

これは、Flavors においてメソッド定義時に method combination の種類が宣言され、他のすべての mixin はそれに従うというスタイルに一致している。

5.2 after ルール

この節では、結合性判定基準を満たすための具体的なルールを1つ挙げ、その正当性を検証する。

我々は現在までのプログラミング経験および非形式的な考察から、個々の mixin のメソッドの仕様が以下のような条件を満たしているならば、mixin は安全に結合できると予想した。このルールを「after ルール」と呼ぶことにする。これは、 Flavors の after daemon という機構にヒントを得て見いだしたルールである。

mixin がスーパークラスのメソッドをオーバーライドするとき、

- メソッドの事前条件は変えてはいけない。
- super を最初にちょうど1回呼び出さなければならない。
- super には自分が受け取った引数をそのまま渡さなければならない。
- super から受け取った返値はそのまま返さなければならない。
- スーパークラスから継承した状態を super 呼び出し後に参照してもよいが、更新してはいけない。
- mixin 自身が定義した状態を参照・更新してもよい。

after ルールは、前章の記法を使うと次のように表現できる。

C1 を拡張する mixin Ci (i = 2, 3) のメソッドの仕様が以下の形で書ける。

$$R_i(\cdots) \equiv RO(s1, sO, p)$$
$$E_i(\cdots) \equiv$$
$$EO(s1, s1', sO, sO', p, r) \wedge Ei'(s1', si, si', p, r)$$

C1, C2, C3 の外部仕様が after ルールを満たしているとき、結合性判定基準を満たすことは、容易に確かめることができる。例えば条件 $(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C2)$ は、以下のように確かめられる。(C1 C2 C3 C4) のメソッドの事前条件・事後条件はそれぞれ $R1$, $E1 \wedge E2' \wedge E3'$ である。また (C1 C2) のメソッドの事前条件・事後条件はそれぞれ $R1$, $E1 \wedge E2'$ である。このとき、$(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C2)$ 、すなわち $(R1 \Rightarrow R1) \wedge (R1 \Rightarrow (E1 \wedge E2' \wedge E3' \Rightarrow E1 \wedge E2'))$ は、確かに成り立っている。

5.3 “+” ルール

結合性判定基準を満たすためのもう1つのルールの例として、“+” ルールについて説明する。“+” ルールは、 Flavors にある “+” という method combination にヒントを得て見いだしたルールである。

“+” ルールは after ルールとほとんど同じだが、以下の点が違う。after ルールでは mixin が super 呼び出しの返値をそのまま返すことしか許されなかったが、 “+” ルールでは、非負の値を足して返すことが許される。その代わり、各クラスの利用者は、正確な返値を仕様から知ることはできず、返値の最低値しか知ることができない。

“+” ルールは、前章の記法を使うと次のように表現できる。

C1 のメソッドの事後条件と、それを拡張する mixin Ci (i = 2, 3) のメソッドの仕様が以下の形で書ける。

$$E1(\cdots) \equiv \exists r1\ .\ E1'(s1, s1', p, r1) \wedge (r \geq r1)$$
$$R_i(\cdots) \equiv RO(s1, sO, p)$$
$$E_i(\cdots) \equiv \exists rO, ri\ .\ EO(\cdots, rO)$$
$$\wedge\ Ei'(s1', si, si', p, ri) \wedge (ri \geq 0) \wedge (r \geq rO + ri)$$

このルールも結合性判定基準を満たすことは、容易に確かめることができる。例えば条件 $(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C2)$ は、以下のように確かめられる。(C1 C2 C3 C4) のメソッドの事前条件・事後条件はそれぞれ $R1$, $E1' \wedge E2' \wedge E3' \wedge (r2 \geq 0) \wedge (r3 \geq 0) \wedge (r \geq r1 + r2 + r3)$ である。また (C1 C2) のメソッドの事前条件・事後条件はそれぞれ $R1$, $E1' \wedge E2' \wedge (r2 \geq 0) \wedge (r \geq r1 + r2)$ である。このとき、$(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C2)$ すなわち $(R1 \Rightarrow R1) \wedge (R1 \Rightarrow (E1' \wedge E2' \wedge E3' \wedge (r2 \geq 0) \wedge (r3 \geq 0) \wedge (r \geq r1 + r2 + r3) \Rightarrow E1' \wedge E2' \wedge (r2 \geq 0) \wedge (r \geq r1 + r2)))$ は、確かに成り立っている。

6 関連研究

本論文では「個々のモジュール開発時に仮定したクラスの振る舞いが、モジュール結合後の振る舞いに置き換え可能である」という条件に着目している。このように behavioral subtyping の考えを mixin やアスペクトに応用した研究は過去にない。

[9] では、アスペクト指向言語において、個々のアスペクトが扱うデータが、結合後のプログラムにおいて依存関係を持たなければ、アスペクトの干渉は起きない、と指摘している。しかしこの判定基準は保守的であり、例えば “+” ルールに従う mixin の結合は、干渉を起こし得ると判断されてしまう。“+” ルールでは返値が全ての mixin に依存しているためである。

[11] では、 Hyper/J[8] のようなシステムを用いて複数のクラス階層を結合するときに起きる干渉の静的な検出方法について述べている。この論文では、あるメソッド呼び出しによってディスパッチされるメソッドが、結合の前後で異なる場合を干渉であると定義している。この判定基準では本論文が扱っているようなメソッドのオーバーライドが起こる組み合わせはすべて干渉であると判断されるため、その点では 3章で定義した判定基準より保守的である。一方 [11] の判定基準では、静的解析によって決して呼び出されないと分かっているメソッドの振る舞いは、変化しても許容される。この点では、呼び出されないメソッドに対しても互換性を要求する本論文の判定基準の方が保守的である。

[10] は、アスペクトが追加され得る場所に accept 宣言を書かせることで、 AspectJ における modular reasoning を可能にする提案である。しかしこの方法は後からアスペクトを追加するときに、既存のソースコードの修正が必要になってしまい、アスペクト指向言語のメリットが半減する。

[12] は、アスペクト結合後のプログラムに対して Java 用のモデル検査ツールを適用することで、デッドロックなどの予期しない動作が起きるかどうかを検査する手法について述べている。

7 まとめと今後

本論文では１つのメソッドを２つの mixin が同時に拡張する場合であっても、個々の mixin が同一のルールに従っているならば安全に結合できることを示した。つまり、多くの研究者やプログラマーが考える以上に、mixin に高い安全性と自由度を持たせることができることを示した。

本手法は MixJuice 言語 [1, 2] のような新しいモジュール機構のもとでの「安全な結合のためのルール」の検証に応用できる。実際、我々はそれに着手しており、いくつか結果を得ている。MixJuice 言語は広い意味でのアスペクト指向言語の１つだが、他のアスペクト指向言語にも本手法が適用できると考えている。

本研究は他にも以下の発展の方向を持っている。

- ダイアモンド継承に限らない、一般の継承グラフでの安全性の証明。
- 実用上よく使われるルールのカタログ化。
- ルールを宣言・強制する機構のアスペクト指向言語への追加。
- アスペクトの干渉を実行時に検出する表明検査機構の設計。

参考文献

[1] 一杉裕志: “シンプルかつ強力なモジュール機構を有するオブジェクト指向言語 MixJuice の提案”, コンピュータソフトウエア, Vol.18,No.6(2001),pp.54–58.

[2] Yuuji Ichisugi and Akira Tanaka: “Difference-Based Modules: A Class-Independent Module Mechanism” In Proc. of the ECOOP2002, 2002.

[3] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.M. and Irwin, J.: ”Aspect-Oriented Programming.” Invited Talk. In Proc. of E-COOP’97, LNCS 1241, pp.220-242, 1997.

[4] Bertrand Meyer: “Object-Oriented Software Construction, 2nd Ed”, Prentice-Hall, Inc., 1997.

[5] Barbara H. Liskov and Jeannette M. Wing: “A behavioral notion of subtyping”, ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 16, No. 6, 1994.

[6] David A. Moon: “Object-Oriented Programming with Flavors”, In Proc. of OOPSLA’86.

[7] Steele, G.L.: “Common Lisp the Language, 2nd edition”, Digital Press, 1990.

[8] H. Ossher and P. Tarr: “Multi-Dimensional Separation of Concerns and The Hyperspace Approach”, In Proc. of “the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, Kluwer”, 2000.

[9] Uwe Assmann: “AOP with design patterns as metaprogramming operators”, Universität Karlsruhe Technical Report No. 28, Oct. 1997.

[10] Curtis Cilfton and Gary T. Leavens: “Observer and Assistants: A proposal for Modular Aspect-Oriented Reasoning,” In Proc. of Foundations of Aspect-Oriented Languages Workshop at AOSD 2002.

[11] Gregor Snelting and Frank Tip: “Semantic-based composition of class hierarchies”, In Proc. of ECOOP2002.

[12] 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, June 2002.