拡張ルール：安全に結合可能なアスペクトの記述ルール

PPL2003 発表資料*

一杉 裕志 †、田中 哲 †、渡部 卓雄 ‡

†産業技術総合研究所

‡国立情報学研究所／東京工業大学

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あらかじめ決められた**拡張ルール**に従って設計されていれば、複数同時に組み合わせて安全に動作させることが可能である。拡張ルールは、アスペクトによる既存のメソッドの振る舞いの変更の仕方を制限する。本論文では mixin および MixJuice 言語のモジュールがダイアモンド継承の形になるケースについて考察し、拡張ルールの安全性を形式的に検証する手法について述べる。安全性の定義および検証には、事前条件、事後条件、 behavioral subtyping の考え方を用いる。具体的な拡張ルールの例として、 After ルール、Plus ルール、Functional Protocol ルール、Disjoint Branch ルールと呼ぶ４つを挙げ、それぞれについて mixin における場合と MixJuice 言語における場合について安全性を検証する。また、本論文の考えに基づいた実用的言語を設計する方針についても述べる。

1 背景

アスペクト指向言語に対して、次のような疑問がよく聞かれる。

- 既存のクラスの動作がアスペクトの追加で変更されるのは危険では？
- プログラムの意味を局所的に理解することが不可能では？

この論文は、これらの疑問に答える第一歩となるものである。

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、厳密に **Design by Contract**[11] 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を仮定して正しく動くプログラムを書く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の表現方法の１つに、事前条件と事後条件という論理式を用いるものがある。Design by Contract がもし厳密に実践されているならば、たとえ個々の抽象データ型の内部実装が変化しても、それらの外部仕様さえ変化しなければ、それらの利用者側のプログラムの正しさには影響を与えない。

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

*この論文の英語版は産総研テクニカルレポート AIST01-J00002-4 として公開されている。

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。このとき、スーパークラスとサブクラスの間に **behavioral subtyping** [9] の関係がある、と言う。このルールが守られていれば、たとえサブクラスがスーパークラスの振る舞いを変更したとしても、その変更の範囲はスーパークラスの外部仕様の範囲に収まり、スーパークラスの型の利用者側のプログラムの正しさに影響を与えない。

多重継承可能な mixin が書けるオブジェクト指向言語では、さらにルールの追加が必要になる。複数の mixin を安全に結合（多重継承）可能にするためのルールについて形式的に論じた過去の研究はないが、経験的な知見は得られている。本論文では、 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と定義する。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。後者の場合の安全性を向上させるために、CLOS[16] およびその前身である Flavors[12] という言語は、method combination という機構を提供している。しかし、method combination の機構の安全性は形式的に証明されているわけではなく、実際、完全に安全ではない。また、組み込みで提供されている以外の安全な method combination を、どう定義すればよいかという指針も明確ではない。

そこで本論文では、 Flavors の method combination のアイデアを一般化した**拡張ルール**によって、mixin の安全な結合を保証する方法を提案する。そして、拡張ルールの安全性を形式的に検証する方法について述べる。拡張ルールは、メソッドのオーバーライドによる振る舞いの変更の仕方を制限する。２つの mixin がスーパークラスのメソッドの振る舞いを拡張ルールを満たすように拡張するならば、２つの mixin は安全に結合可能である。つまり、拡張ルールは、多重継承を行なったクラスのインスタンスの振る舞いが、多重継承前の個々のクラスの外部仕様を満たしていることを保証する。

そして次に、この手法を広い意味での**アスペクト指向言語** [4] の１つである MixJuice[19, 6, 5] に応用する。アスペクト指向言語において、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 [4]。本手法は、 MixJuice 言語に限らず、他のアスペクト指向言語にも適用できると考えている。アスペクト指向言語の機構の一部は mixin と類似しているからである。実際、アスペクト指向言語の１つである AspectJ[8] が有する before/after/around advice という機構は、もともとは Flavors の method combination に関する機構の一部である。我々の研究は、 AspectJ のこの機構の安全性を向上させるために役立つであろう。

本論文では拡張ルールの検証は mixin および MixJuice のモジュールのダイアモンド継承のもとで行なっている。しかし、本論文で述べた全ての拡張ルールはダイアモンド継承に限らず、一般の継承グラフのもとでも安全であると予想している。

本論文では厳密な形式化よりも、わかりやすい非形式的な説明を優先する。本論文の読者はオブジェクト指向言語と１階述語論理の基礎知識を持っていると仮定する。

我々は、安全性の高いアスペクト指向プログラミングを行なうためには、全てのプログラマーが事前条件・事後条件を考えながら厳密なプログラミングをする必要があるとは決して考えていない。むしろ、 Flavors の method combination を発展させることで、事前条件・事後条件を全く知らないプログラマーでも、十分に安全なアスペクト指向プログラミングが可能になると考えている。それを可能にするための今後の研究方針については最後の章で述べる。

本論文は以下のような構成になっている。2章では behavioral subtyping について簡単に説明する。3章では mixin の拡張ルール、4章では MixJuice のモジュールの拡張ルールについて述べる。5章は拡張ルールの安全性の検証支援ツールについて、6章では関連研究について述べる。7章ではまとめと今後について述べる。

2 Behavioral Subtyping

この章では behavioral subtyping[9] という考え方を簡単に説明する。この考え方は、次の章で「安全な結合」を定義する際に用いられる。

behavioral subtyping とは、クラスやメソッドの signature だけでなく、クラスやメソッドの振る舞い（外部仕様）も考慮して定義する subtype 関係である。

C++, Java など多くのオブジェクト指向言語では、クラス定義時におけるスーパークラスの宣言が、sub-

type の関係の宣言をも意味する。例えばクラス C1 を継承するクラス C2 を定義すると、型 C2 は型 C1 の subtype になる。サブクラスはスーパークラスのすべてのインスタンス変数とメソッドを継承するため、スーパークラスに対する全ての操作は、サブクラスに対しても行なうことができる。その意味で、この型システムは安全である。

しかし、これではコンパイラの型チェックを通ったプログラムは “method not found error” を出さないことが保証されるだけで、それ以上のことが保証されるわけではない。言うまでもなく、“method not found error” が出ないにも関わらず期待どおりに動作しないプログラムはいくらでもあり得る。

このことは、特に、個々のクラスが独立したプログラマーによって定義されるときに問題になる。例えばあるクラス A のメソッド m が、引数として、ライブラリ中で定義されている型 Vector を受け取るとする。

```
class A {
  void m(Vector v){
    ...
    int s = v.size();
    ...
    Object x = v.elementAt(i);
    ...
  }
}
```

普通、このクラスの実装者 A は、受け取った引数がクラス Vector のインスタンスであると仮定してプログラミングする。もし別のプログラマー B が、クラス Vector とは全く異なる振る舞いをするサブクラスを定義し、さらにその事情を知らない別のプログラマー C が、そのインスタンスを A が定義したメソッド m に渡したら、メソッド m は正しく動作しなくなる。例えば、 `ArrayIndexOutOfBoundsException` が発生して異常終了する可能性がある。

このような事態を避けるために、「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置き換え可能でなければならない」というルールが、必要になる。つまり、新たなサブクラスを定義するプログラマーは、そのサブクラスのインスタンスが、スーパークラスと同じように振る舞うようにしなければならない[1]。このルールを満たしているとき、「サブクラスの型はスーパークラスの型の behavioral subtype である」と言う。

behavioral subtyping は、クラスの外部仕様を論理式を使って表現することで、より厳密に定義することができる。あるクラスのメソッド m の仕様は形式的には、事前条件と事後条件という 2 つの論理式で表される。事前条件は m が呼ばれる直前に成り立っていなければならない条件、事後条件は m の実行終了直後に成り立っていなければならない条件である。あるメソッド m のスーパークラス C1 における事前条件と事後条件を R1, E1 、サブクラス C2 における事前条件と事後条件を R2, E2 （ R は require 、 E は ensure の意味）とすると、C2 が C1 の behavioral subtype になるためには、次の論理式が成り立つことが必要である[2] 。

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

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してほしい[3] 。

例として、次のようなメソッド m を考える。

```
class C1 {
  double m(double p) { return Math.sqrt(p); }
}
```

このメソッドは、非負の実数を受け取って、その平方根を返すメソッドである。このメソッドの事前条件と事後条件は、次のように書ける。ただし p はメソッド m の引数、 r は返値を表すものとする。

$$R1(p) \equiv p \geq 0$$
$$E1(p,r) \equiv r = \sqrt{p}$$

一方、次のようなメソッドをもつクラス C2 があるとする。

```
class C2 {
  double m(double p) {
    if (p >= 0){
      return Math.sqrt(p);
    } else {
      return -1;
    }
  }
}
```

[1] この条件はプログラマーの間で the Liskov Substitution Principle [10] とも呼ばれる。

[2] [9] で述べられているように、他にクラスの invariant と constraint に関する条件が必要であるが、この扱いは future work とする。

[3] [3] で指摘されているように、[11] や [9] で述べられている $(R1 \Rightarrow R2) \wedge (E2 \Rightarrow E1)$ は厳しすぎる条件である。なお [3] では、$(R1 \Rightarrow R2) \wedge ((R2 \Rightarrow E2) \Rightarrow (R1 \Rightarrow E1))$ という論理式を用いているが、これは本論文で用いる論理式と等価である。

クラス C2 のメソッド m の仕様は、以下のようなものであるとする。

$$R2(p) \equiv true$$
$$E2(p,r) \equiv (p \geq 0 \Rightarrow r = \sqrt{p}) \wedge (p < 0 \Rightarrow r = -1)$$

ここで、$(R1 \Rightarrow R2) \wedge (R1 \Rightarrow (E2 \Rightarrow E1))$ すなわち、

$$\begin{aligned}&(p \geq 0 \Rightarrow true)\\ &\wedge (p \geq 0 \Rightarrow ((p \geq 0 \Rightarrow r = \sqrt{p}) \wedge (p < 0 \Rightarrow r = -1)\\ &\qquad \Rightarrow r = \sqrt{p}))\end{aligned}$$

という論理式は恒真なので、 C2 は C1 の behavioral subtype である。

実際、 C2 のインスタンスが型 C1 の変数に代入されたとしても、呼び出されるメソッド m の引数は常に非負であり、その返値は C1 のインスタンスが返すはずの値と必ず一致する。

逆に言えば、型 C1 の利用者は、その型の変数に実際には C2 のインスタンスが入り得ることを意識する必要がなく、それがあたかも C1 のインスタンスであると仮定してプログラミングを行なうことができる。型 C1 の利用者は、型 C1 の変数に入っているインスタンスの振る舞いに関する様々な性質を、正しく推論することができる。つまり、 modular reasoning を行なうことができる。

ただし、ここで型 C1 の利用者が Design by Contract を行なうという前提が必要である。つまり、型 C1 の利用者は、 C1 の外部仕様に書かれている以上の仮定を置いてはいけない。例えばメソッド m に負の値を渡し、その返値として NaN (Not a Number) が帰って来ることを期待してはいけない。たまたまクラス C1 の実装はそのような振る舞いをするが、外部仕様には負の値を引数に渡したときの動作については何も書かれていないからである。

Design by Contract および behavioral subtyping を厳密に実践することは現実のプログラミングでは難しいが、近似的に実践するだけでも、プログラムの信頼性向上に大きな効果があることはよく知られている [11]。

3 安全に結合可能な mixin を提供するための拡張ルール

この章ではまず手始めとして、 mixin のダイアモンド継承について考察する。(次の章では、この章で定義する記法と検証方法を用いて、MixJuice のモジュールのダイアモンド継承について考察する。)

3.1 考察するクラス階層

本章では、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オブジェクト指向言語を仮定する。(CLOS に静的型を持たせたような言語である。)

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 (直列化) される。C4 のインスタンスの振る舞いは、図 1(b) のように定義されるクラスのインスタンスの振る舞いと同一である。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。なお、メソッド m 内で、 super の m 以外のメソッドを呼ぶ場合については、本論文では扱わない。

本論文では、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メソッドが再帰的に呼び出されることはないものと仮定する。(つまり、本論文では downcall problem[14] は扱わない。)

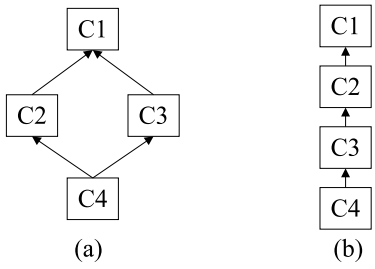

図 1: (a) ４つのクラスの継承関係と、(b) C4 のスーパークラスをリニアライズした結果

3.2 インスタンスの振る舞いの記法

本論文では、あるクラスのインスタンスの振る舞いについて述べるとき、インスタンスをそのクラス自身とスーパークラスをリニアライズしたリストで表記することがある。インスタンスの正確な振る舞いは、その内部における super 呼び出しの振る舞いに依存し、 super 呼び出しの振る舞いはリニアライズすることで初めて確定するからである。例えばクラス C1, C2, C3, C4 のインスタンスの振る舞いはそれぞれ、$(C1)$ 、 $(C1\ C2)$ 、 $(C1\ C3)$ 、 $(C1\ C2\ C3\ C4)$ と表記する。

また、インスタンス o2 の振る舞いがインスタンス o1 の振る舞いの behavioral subtype であるという関係を、 $o2 \hookrightarrow o1$ と表記する。例えばクラス C2 のインスタンスがクラス C1 のインスタンスの behavioral subtype であるという関係は、次のように表記する。

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

上記の関係が成り立つためには、前章で述べたように、C1 が持つ各メソッドについて、インスタンス $(C1)$ 、 $(C1\ C2)$ における事前条件をそれぞれ $R_{(C1)}$,$R_{(C1C2)}$、事後条件をそれぞれ $E_{(C1)}$,$E_{(C1C2)}$ とすると、以下の論理式が成り立つことが必要である。

$$(R_{(C1)} \Rightarrow R_{(C1C2)}) \wedge (R_{(C1)} \Rightarrow (E_{(C1C2)} \Rightarrow E_{(C1)}))$$

3.3 結合性判定基準

この節では「 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種類の条件群からなる。１種類目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種類はクラス定義時の 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。

１種類目の条件の例を１つ挙げる。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についても 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 の定義時に想定する super は図 1(a) で示されているように $(C1)$ であり、C4 のインスタンス内から見た C3 の super は図 1(b) で示されているように $(C1\ C2)$ 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 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、クラス C2, C3 に 3.5節で述べる拡張ルールを守らせることで、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。

3.4 メソッドの外部仕様の表現方法

この節では、メソッドの外部仕様の表現方法について説明する。(次の節では、この表現方法に基づいて、クラス C1, C2, C3 の外部仕様が満たすべき制約、すなわち拡張ルールを表現し、その正当性を検証する。)

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。)

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

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; }
}
```

クラス 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。

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

ただし sO は、リニアライズ後に自分自身と C1 との間に入る他の mixin によって導入された状態である。例えばインスタンス $(C1\ C2)$ 内の C2 から見た sO は空であり、インスタンス $(C1\ C2\ C3)$ 内の C3 から見た sO は $s2$ である。

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、インスタンス $(C_1\ \cdots\ C_n\ C_{n+1})$ 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、インスタンス (C) 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、クラス C2 の事後条件の中の 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７つの条件のうち、$(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C2)$ が満たされていない。$E_{(C1C2C3C4)} \Rightarrow E_{(C1C2)}$ という論理式が恒真にならないからである。

3.5 拡張ルールとその検証

この節では、 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（拡張ルー

ル）について述べ、その拡張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。

3.5.1 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拡張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、各メソッドごとに拡張ルールを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拡張ルールに従って、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。

3.5.2 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拡張ルールを１つ挙げ、その安全性を検証する。

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。この拡張ルールを「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拡張ルールである。

このルールは、リニアライズされたクラスの上の方から順にメソッドを実行していき、しかも上のクラスによって満たされた事後条件が、下のクラスのメソッドの実行後も保存されるように設計されている。

After ルール:
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。

この拡張ルールは、形式的には次のように表現できる。

After ルール:
C1 を拡張する mixin Ci (i = 2, 3) のメソッドの事前条件・事後条件が以下の形で書ける。

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

E_i' は、パラメタ $s1', si, si', p, r$ を含む任意の論理式である。

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・事後条件は以下のようになる。

$$R_{(C1\ C2\ C3\ C4)} \equiv R1$$
$$E_{(C1\ C2\ C3\ C4)} \equiv E1 \wedge E2' \wedge E3'$$

また $(C1\ C2)$ のメソッドの事前条件・事後条件は以下のようになる。

$$R_{(C1\ C2)} \equiv R1$$
$$E_{(C1\ C2)} \equiv E1 \wedge E2'$$

このとき、$(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C2)$ 、すなわち下記の論理式は確かに恒真である。

$$\begin{aligned}&(R_{(C1\ C2)} \Rightarrow R_{(C1\ C2\ C3\ C4)})\\ &\quad \wedge (R_{(C1\ C2)} \Rightarrow (E_{(C1\ C2\ C3\ C4)} \Rightarrow E_{(C1\ C2)}))\\ &= (R1 \Rightarrow R1) \wedge (R1 \Rightarrow (E1 \wedge E2' \wedge E3' \Rightarrow E1 \wedge E2'))\\ &= true\end{aligned}$$

3.5.3 Plus ルール

結合性判定基準を満たすためのもう１つの拡張ルールの例として、 Plus ルールについて説明する。Plus 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拡張ルールである。

Plus ルールは After ルールとほとんど同じで、リニアライズされたクラスの上の方から順にメソッドを実行させるものだが、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、 Plus 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、各クラスの事後条件は、正確な返値は保証せず、返値の最低値のみを保証する。

下記のプログラムは、この拡張ルールにしたがったプログラムの例である。

```
class C1 {
  Vector v1;
  int m(){ return v1.size(); }
}
class C2 extends C1 {
  Vector v2;
  int m(){ return super.m() + v2.size(); }
}
class C3 extends C1 {
  Vector v3;
  int m(){ return super.m() + v3.size(); }
}
```

この拡張ルールは、形式的には次のように表現できる。

> Plus ルール:
> C1 のメソッドの事後条件と、それを拡張する mixin Ci (i = 2, 3) のメソッドの事前条件・事後条件が以下の形で書ける。
>
> $E1(\cdots) \equiv \exists r1\ .\ E1'(s1, s1', p, r1) \wedge (r \geq r1)$
>
> $R_i(\cdots) \equiv RO(s1, sO, p)$
>
> $E_i(\cdots) \equiv \exists rO, ri\ .\ EO(\cdots, rO)$
> $\quad \wedge Ei'(s1', si, si', p, ri) \wedge (ri \geq 0) \wedge (r \geq rO + ri)$

この拡張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・事後条件は以下のようになる。

$$R_{(C1\ C2\ C3\ C4)} \equiv R1$$
$$E_{(C1\ C2\ C3\ C4)} \equiv$$
$$\exists r1, r2, r3\ .\ E1' \wedge E2' \wedge E3'$$
$$\wedge (r2 \geq 0) \wedge (r3 \geq 0) \wedge (r \geq r1 + r2 + r3)$$

また $(C1\ C2)$ のメソッドの事前条件・事後条件は以下のようになる。

$$R_{(C1\ C2)} \equiv R1$$
$$E_{(C1\ C2)} \equiv \exists r1, r2\ .\ E1' \wedge E2' \wedge (r2 \geq 0) \wedge (r \geq r1 + r2)$$

このとき、$(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C2)$ すなわち下記の論理式は確かに恒真である。

$$\begin{aligned}
&(R_{(C1\ C2)} \Rightarrow R_{(C1\ C2\ C3\ C4)}) \\
&\quad \wedge (R_{(C1\ C2)} \Rightarrow (E_{(C1\ C2\ C3\ C4)} \Rightarrow E_{(C1\ C2)})) \\
&= (R1 \Rightarrow R1) \wedge \\
&\quad (R1 \Rightarrow (\exists r1, r2, r3\ .\ E1' \wedge E2' \wedge E3' \wedge (r2 \geq 0) \wedge (r3 \geq 0) \\
&\qquad \wedge (r \geq r1 + r2 + r3) \\
&\qquad \Rightarrow \exists r1, r2\ .\ E1' \wedge E2' \wedge (r2 \geq 0) \wedge (r \geq r1 + r2))) \\
&= true
\end{aligned}$$

Plus ルールの様々なバリエーションを、容易に考えることができる。一般に、 super の返値を mixin が何らかの意味で単調に増加（または減少）させていくことを許す拡張ルールは、結合可能性判定基準を満たすであろう。例えば、集合に要素を追加していく拡張ルールなどが考えられる。より現実的な例で言えば、ハッシュ表に要素を追加するメソッドや GUI のメニューにエントリを追加するメソッドも、このルールのバリエーションである。

Plus ルールは、 domain specific な拡張ルールの例である。このルールは整数の値に関する論理式を用いて記述されており、その正当性の証明には、

$$(a \geq b + c) \wedge (c \geq 0) \Rightarrow (a \geq b)$$

という整数論の定理を用いている。このように、実際のプログラミングでは、アプリケーションの対象領域に関する記述を含む拡張ルールが必要になる場合がある。例えば、コンパイラなどの言語処理系を mixin を使って拡張可能にする場合は、言語の意味論を扱った拡張ルールを提供する必要がある。例えば [18] では、そのような拡張ルールについて非形式的に述べている。

3.5.4 Functional Protocol ルール

メソッドの返値を求めるアルゴリズムの変更を許すような拡張ルールを考えることができる。[7] では、このような拡張を許すメソッドを functional protocol と分類している[4] 。そこで、そのような拡張ルールを Functional Protocol ルールと呼ぶことにする。

これまで述べた拡張ルールでは mixin は必ず super 呼び出しをしなければならなかったが、Functional Protocol ルールでは、mixin は必ずしも super 呼び出

[4] より正確に言うと、 [7] でいう functional protocol は、アルゴリズムだけでなく、返すべき値そのものを変えるようなオーバーライドも想定している。サブクラスによるオーバーライドならそれでも問題ないが、mixin の場合はそのようなオーバーライドを行なっていると安全に結合できない。そこで、この節で述べる拡張ルールでは返値の変更は禁止している。

しをする必要はない。その代わり、メソッドの呼び出しによって副作用（内部状態の更新）が必ず起きることは保証できず、「副作用は起きるかもしれないし起きないかもしれない」ということしか保証できない。

下記のプログラムは、この拡張ルールにしたがったプログラムの例である。2つの mixin が独立した方法で、メソッドの返値をキャッシュしている。

```
class C1 {
  String m(String s){ ...; return r; }
}
class C2 extends C1 {
  Hashtable cache = new Hashtable();
  String m(String s){
    String r = (String)cache.get(s);
    if (r == null){
      r = super.m(s);
      cache.put(s, r);
    }
    return r;
  }
}
class C3 extends C1 {
  String lastS = null;
  String lastR = null;
  String m(String s){
    if (! (lastS != null && s.equals(lastS))){
      lastS = s; lastR = super.m(s);
    }
    return lastR;
  }
}
```

Functional Protocol ルールは形式的には以下のように表現できる。

Functional Protocol ルール:

クラス C1 のメソッドの事後条件と、C1 を拡張する mixin $C_i(i = 2,3)$ のメソッドの事前条件・事後条件が以下のように書ける：

$$E1(s1, s1', p, r) \equiv$$
$$(s1 = s1' \vee E1''(s1, s1', p)) \wedge E1'(s1, p, r)$$
$$R_i(s1, sO, si, p) \equiv RO(s1, sO, p)$$
$$E_i(s1, s1', sO, sO', si, si', p, r) \equiv$$
$$(EO(s1, s1', sO, sO', p, r)$$
$$\vee (s1 = s1' \wedge sO = sO' \wedge E1'(s1, p, r)))$$
$$\wedge (si = si' \vee E_i''(s1', si, si', p))$$

ただし、 $E1'$ は返値を決める論理式、$E1'', E_i''$ は副作用を決める論理式である。論理式 $s1 = s1' \vee E1''(s1, s1', p)$ は、 C1 が定義するメソッド m による副作用が起きるかもしれないし起きないかもしれないことを示している。論理式 $EO(s1, s1', sO, sO', p, r) \vee (s1 = s1' \wedge sO = sO' \wedge E1'(s1, p, r))$ はクラス C2, C3 のメソッド m が super を呼び出すかもしれないし呼び出さないかも知れないことを示している。また、論理式 $si = si' \vee E_i''(s1', si, si', p)$ は、クラス C2, C3 で定義するメソッド m による副作用が起きるかもしれないし起きないかもしれないことを示している。いずれの場合も、返値としては論理式 $E1'$ で決まる値が返される。

3.5.5 Disjoint Branch ルール

メソッドの定義域を、 mixin が拡張することを許すような拡張ルールも考えられる。そのようなルールの1つを Disjoint Branch ルールと呼ぶことにする。このルールは、個々の mixin が処理する定義域の間に重なりがないことを前提とする。

下記のプログラムは、この拡張ルールにしたがったプログラムの例である。

```
class C1 {
  void m(String s){}
}
class C2 extends C1 {
  void m(String s){
    if (s.equals("A")){...}
    else { super.m(s); }
  }
}
class C3 extends C1 {
  void m(String s){
    if (s.equals("B")){...}
    else { super.m(s); }
  }
}
```

この拡張ルールの具体的なアプリケーションとしては、XML を処理するプログラムがある。各 mixin が処理する要素のタグ名を、その mixin の実装者が管理する名前空間に属するようにすることで、各 mixin の処理の対象が disjoint であることを保証することができる。

この拡張ルールは、以下のように表現できる。

Disjoint Branch ルール:

クラス C1 のメソッドとそれを拡張する mixin $C_i(i = 2,3)$ のメソッドの事前条件・事後条件が以下のように書ける。

ただし、e_i は定数で、 $\forall i, j(i \neq j) \,.\, e_i \neq e_j$ とする。

$$R1(s1, p) \equiv false$$
$$E1(s1, s1', p, r) \equiv true$$
$$R_i(s1, sO, si, p) \equiv RO(s1, sO, p) \vee p = e_i$$
$$E_i(s1, s1', sO, sO', si, si', p, r) \equiv$$
$$((p = e_i) \Rightarrow E_i'(s1, s1', si, si', p, r))$$
$$\wedge ((p \neq e_i) \Rightarrow EO(s1, s1', sO, sO', p, r))$$

4 安全に結合可能な MixJuice のモジュールを提供するための拡張ルール

本章では、前章で述べた記法と検証方法を用いて、MixJuice での拡張ルールについて検証する。

4.1 MixJuice の概要

MixJuice[19, 6, 5] は Java 言語が持つクラスベースのモジュール機構を取り除き、代わりに差分ベースモジュールと呼ぶモジュール機構を採用したオブジェクト指向言語である。

個々のモジュール同士は継承関係を持ち得る。MixJuice には、モジュールの継承機構と従来のクラス継承の機構の両方が独立に存在する。クラス継承とモジュール継承は、次のように違う。まず、クラス継承は既存のクラスに対する変更部分を記述する機構だが、モジュール継承では、既存のプログラム全体に対する変更部分を記述する機構である。また、クラス継承では、すでに存在するクラスとは違う名前を持った、新しいサブクラスを定義することしかできないが、モジュール継承では、すでに存在するクラスそのものに対し、その名前を変えることなしに、変更を加えることができる。クラス継承は、subtyping を行なう機構であり、それによりメソッドの動的結合を行なうための機構でもある。一方、モジュール継承は、静的な再利用のための機構であり、また情報隠蔽のための機構である。

個々のモジュールは分割コンパイルすることができる。その際、モジュール内のコードは型チェックされる。あるモジュールをコンパイルするには、そのモジュールの先祖モジュールのソースまたはバイナリ[5]があればよい。したがって、例えば兄弟関係にあるモジュールは、それぞれ独立に開発することができる。

エンドユーザは、独立して開発されたモジュールを、その実装の詳細を知らなくても、組み合わせて使うことができる。本研究の我々の目的の１つは、プログラマーによる結合の安全性だけでなく、このようなエンドユーザによる結合の安全性も保証することである。

[5]Java 言語と同様、バイナリ内にはモジュールの外部インターフェースの情報が入れられている。

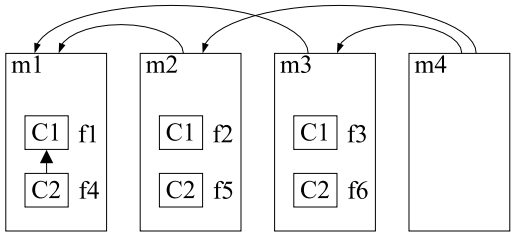

図 3: MixJuice におけるモジュールのダイアモンド継承

```
module m1 {
 define class C1 { define int m(int p){...}}    // f1
 define class C2 extends C1 { int m(int p){...}}// f4
}
module m2 extends m1 {
 class C1 { int m(int p){...}}  // f2
 class C2 { int m(int p){...}}  // f5
}
module m3 extends m1 {
 class C1 { int m(int p){...}}  // f3
 class C2 { int m(int p){...}}  // f6
}
module m4 extends m2, m3 {}
```

図 4: ダイアモンド継承を行なうモジュール定義

4.2 考察する MixJuice プログラム

ここでも一般的なケースについては扱わず、モジュールのダイアモンド継承のケースを扱う。具体的には、図 3のように、２つのクラス定義を含むモジュール m1 を２つのモジュール m2, m3 が継承し、さらに m2, m3 をモジュール m4 が多重継承する場合を考える。図 3は MixJuice のコードで書くと図 4のようになる。

MixJuice のモジュールは、リンクすることで、実行可能なプログラムになる。リンクする際には、モジュールを何らかのアルゴリズムでリニアライズする。本論文では、リニアライズ結果のプログラムを、モジュール名のリストで表すことにする。上記４つのモジュールの可能な組み合わせによってできるプログラムには、下記の４つがある。

$(m1)$

$(m1\ m2)$

$(m1\ m3)$

$(m1\ m2\ m3\ m4)$

```
class t1 { int m(int p){...}}             // f1
class t2 extends t1 { int m(int p){...}}  // f2
class C1 extends t2 { int m(int p){...}}  // f3
class t4 extends C1 { int m(int p){...}}  // f4
class t5 extends t4 { int m(int p){...}}  // f5
class C2 extends t5 { int m(int p){...}}  // f6
```

図 5: プログラム ($m1$ $m2$ $m3$ $m4$) のリンク結果に相当する Java プログラム

(モジュールのリニアライズのアルゴリズムによっては違う結果も起こり得るが、ここでは話を簡単にするためにこれら以外は起きないと仮定する。)

図 3および図 4における $f1, f2, \ldots, f6$ は、「クラスの断片」を表している。$f1, f4$ はそれぞれモジュール $m1$ におけるクラス $C1, C2$ の定義である。$f2, f3$ はクラス $C1$ に対する拡張、$f5, f6$ はクラス $C2$ に対する拡張である。例えばプログラム ($m1$ $m2$) においては、$C1$ と $C2$ はそれぞれ ($f1$ $f2$), ($f1$ $f2$ $f4$ $f5$) とリニアライズされたクラスとして振る舞う。また、プログラム ($m1$ $m2$ $m3$ $m4$) においては、$C1$ と $C2$ はそれぞれ ($f1$ $f2$ $f3$), ($f1$ $f2$ $f3$ $f4$ $f5$ $f6$) とリニアライズされたクラスとして振る舞う。

モジュールのリンク結果は、普通の Java のプログラムになる。例えばプログラム ($m1$ $m2$ $m3$ $m4$) のリンク結果は、図 5の Java プログラムと等価になる。

4.3 結合性判定基準

この節では、4.2節のケースにおける結合性判定基準について考える。

基本的な考え方は前章の mixin の場合と同じで、結合性判定基準は 2 種類の条件群からなる。1 種類目の条件は、スーパークラスの変数にサブクラスのインスタンスが代入される可能性があっても modular reasoning ができるための条件、もう 1 種類の条件は、super 呼び出しの振る舞いが modular reasoning ができるための条件である。

しかし、1 種類目の条件は、MixJuice 言語の場合、クラス継承の他にモジュール継承があるため、より複雑になる。例えば、モジュール $m2$ を実装するプログラマーは、型 $C2$ の変数に代入されているオブジェクトの振る舞いは、($f1$ $f2$ $f4$ $f5$) だと仮定してプログラミングを行なう。しかし、プログラム ($m1$ $m2$ $m3$ $m4$) の実行時には、C2 のインスタンスの振る舞いは ($f1$ $f2$ $f3$ $f4$ $f5$ $f6$) である。したがって、下記の条件が成り立っていなければならない。

$$(f1\ f2\ f3\ f4\ f5\ f6) \hookrightarrow (f1\ f2\ f4\ f5)$$

もう 1 種類の条件、super 呼び出しについても、注意が必要である。MixJuice の言語仕様では、super 呼び出しは、スーパークラスのメソッドを呼び出す場合と、super-module のメソッドを呼び出す場合の 2 種類がある。(実際の MixJuice 言語では super 呼び出しの構文ではなく、どちらも original 呼び出しという構文を用いている。) いずれもリニアライズされたクラスの断片のリスト中における 1 つ上の断片を呼び出すという点では同じである。例えばプログラム ($m1$ $m2$) における $f5$ からの super 呼び出しは、$f2$ ではなく $f4$ を呼び出す。

このことを考慮に入れて、結合性判定基準を列挙したものが図 6である。ただし、$C_j@m_i$ はモジュール m_i が表すプログラムにおけるクラス C_j のインスタンスの振る舞い 、$s_j@m_i$ はモジュール m_i が表すプログラムおけるクラス断片 f_j から見た super の振る舞いの略記である。

ここで、次の性質が成り立っている点に注意してほしい。

- f2 か f3 がリスト中にある時は、それらの左側に必ず f1 がある。

- f5 か f6 がリスト中にある時は、それらの左側に必ず f4 がある。さらに f4 の左側には必ず f1 がある。

この性質は、 MixJuice の言語仕様により、必然的に生じる。MixJuice では、すべてのクラスはまずどこかのモジュールで定義され、その sub-module において拡張される。また、あるクラスのスーパークラスは、そのクラスの定義時に宣言され、他のモジュールによって変更されることはない。

この性質から、各クラス断片の間に非対称性が生じる。例えばクラスの断片 f4 はクラス C2 を定義する断片であり、f5 と f6 は f4 を拡張する断片である。f5 と f6 は対称であるが、 f4 は f5 や f6 とは非対称である。この非対称性は、以下に述べる拡張ルールに反映される。

プログラム $(m1)$ が正しく動く条件：

$C2@m1 \hookrightarrow C1@m1$ すなわち $(f1\ f4) \hookrightarrow (f1)$

プログラム $(m1\ m2)$ が正しく動く条件：

$C1@m2 \hookrightarrow C1@m1$ すなわち $(f1\ f2) \hookrightarrow (f1)$

$C2@m2 \hookrightarrow C1@m1$ すなわち $(f1\ f2\ f4\ f5) \hookrightarrow (f1)$

$C2@m2 \hookrightarrow C1@m2$ すなわち $(f1\ f2\ f4\ f5) \hookrightarrow (f1\ f2)$

$C2@m2 \hookrightarrow C2@m1$ すなわち $(f1\ f2\ f4\ f5) \hookrightarrow (f1\ f2\ f4)$

$s4@m2 \hookrightarrow s4@m1$ すなわち $(f1\ f2\ f4) \hookrightarrow (f1)$

プログラム $(m1\ m3)$ が正しく動く条件：

(m1 m2) の場合と同様

プログラム $(m1\ m2\ m3\ m4)$ が正しく動く条件：

$C1@m4 \hookrightarrow C1@m1$ すなわち $(f1\ f2\ f3) \hookrightarrow (f1)$

$C1@m4 \hookrightarrow C1@m2$ すなわち $(f1\ f2\ f3) \hookrightarrow (f1\ f2)$

$C1@m4 \hookrightarrow C1@m3$ すなわち $(f1\ f2\ f3) \hookrightarrow (f1\ f3)$

$C2@m4 \hookrightarrow C1@m1$ すなわち $(f1\ f2\ f3\ f4\ f5\ f6) \hookrightarrow (f1)$

$C2@m4 \hookrightarrow C1@m2$ すなわち $(f1\ f2\ f3\ f4\ f5\ f6) \hookrightarrow (f1\ f2)$

$C2@m4 \hookrightarrow C1@m3$ すなわち $(f1\ f2\ f3\ f4\ f5\ f6) \hookrightarrow (f1\ f3)$

$C2@m4 \hookrightarrow C2@m1$ すなわち $(f1\ f2\ f3\ f4\ f5\ f6) \hookrightarrow (f1\ f4)$

$C2@m4 \hookrightarrow C2@m2$ すなわち $(f1\ f2\ f3\ f4\ f5\ f6) \hookrightarrow (f1\ f2\ f4\ f5)$

$C2@m4 \hookrightarrow C2@m3$ すなわち $(f1\ f2\ f3\ f4\ f5\ f6) \hookrightarrow (f1\ f3\ f4\ f6)$

$s2@m4 \hookrightarrow s2@m2$ すなわち $(f1) \hookrightarrow (f1)$

$s3@m4 \hookrightarrow s3@m3$ すなわち $(f1\ f2) \hookrightarrow (f1)$

$s4@m4 \hookrightarrow s4@m1$ すなわち $(f1\ f2\ f3) \hookrightarrow (f1)$

$s5@m4 \hookrightarrow s5@m2$ すなわち $(f1\ f2\ f3\ f4) \hookrightarrow (f1\ f2\ f4)$

$s6@m4 \hookrightarrow s6@m3$ すなわち $(f1\ f2\ f3\ f4\ f5) \hookrightarrow (f1\ f3\ f4)$

図 6: MixJuice モジュールのダイアモンド継承における結合性判定基準

4.4 MixJuice After ルール

我々は mixin における After ルールを修正し、図 7 のような「MixJuice After ルール」を考えた。なお、図 7における「サブクラスによるメソッド拡張」とは f4 が行なうメソッドのオーバーライド、「 sub-module によるメソッド拡張」とは f2, f3, f5, f6 が行なうメソッドのオーバーライドを意味する。

ただし、以下の点に注意して欲しい。

- 注１：
  普通のオブジェクト指向言語ではサブクラスは「事前条件を弱く、事後条件を強く」というルールさえ守れば、必ずしも super 呼び出しをする必要がないが、MixJuice After ルールでは super 呼び出しは必須である。これは、 sub-module がスーパークラスの振る舞いを拡張する可能性があるからである。例えば、もし f4 が super 呼び出しを行なわないと、f2 や f3 で拡張された事後条件が満たされないという問題が起きる。f4 を実装するプログラマーは f2 および f3 の存在を知らないので、一般にこれらの事後条件を満たし得ない。

- 注２：
  mixin における After ルールではスーパークラスから継承した状態を更新はできず、参照は許されたが、 MixJuice After ルールでは、サブクラスは継承した状態を参照・更新可、sub-module は継承した状態を参照・更新不可という組み合せを、我々は選択した。サブクラスによる拡張の自由度を、普通のオブジェクト指向言語にできるだけ近づけたかったからである。

  なお、もし「 sub-module は継承した状態を参照可」とすると、次のような状況で問題が起きる。モジュール $m1$ と $m2$ において、各 f_i のあるメソッド m の仕様が次のようなものだとする。

  – $f1$ での事後条件が $(s1' > 0)$
  – $f2$ での事後条件が $(s2' = s1')$
    （`s2 = s1;` という代入文の実行に相当）
  – $f4$ での事後条件が $(s1' = 1)$
    （`s1 = 1;` という代入文の実行に相当）

  この時、たまたま $f1$ で `s1 = 2;` と実装されていたとすると、プログラム $(m1\ m2)$ が正しく動かなくなる。なぜなら、$C2$ のインスタンスのメソッド呼び出しの結果 $s1' = 1$ かつ $s2' = 2$ となるが、モジュール $m2$ 内のコードは $s2' = s1'$ となることを前提として実装されているからである。

  ただし、 $m1$ の仕様から $s1'$ が取るべき値が厳密に予測できる場合は、その予測結果を使っても構わない。つまり、その場合は事実上 $s1'$ を参照可能であると思って良い。例えば f1 での事後条件が $s1' = 1$ であるならば、f2, f3 で $s1'$ を参照することは、実質的には定数 1 を使うことと変わらず、何ら問題はない。

- 注３：
  もし $f2$ が `s2 = super.m();` というふうに返値を使ったとすると、$f4$ が返値を変えたときに問題が起きる。ただし注２と同様に、仕様から正確な返値が予測できる場合は、実質的にその返値を使っても構わない。

この MixJuice After ルールは、形式的には以下のように書くことができ、安全性を検証することができる。

MixJuice After ルール:

１． クラス C1 のメソッドをサブクラス C2 が拡張するとき、サブクラス C2 におけるメソッドの事前条件・事後条件が以下のように書ける。

$$
\begin{aligned}
&R2(s1, sO, s2, p) \equiv RO(s1, sO, p) \vee R2'(s1, s2, p)\\
&E2(\ldots, r) \equiv\\
&\quad (RO(s1, sO, p) \Rightarrow \exists s1'', r'' .\\
&\qquad\qquad (EO(s1, s1'', sO, sO', p, r'')\\
&\qquad\qquad \wedge E2'(s1'', s1', s2, s2', p, r'', r)\\
&\qquad\qquad \wedge E1(s1, s1', p, r)\\
&\qquad\qquad))\\
&\quad \wedge (\neg RO(s1, sO, p) \Rightarrow E2''(s1, s1', s2, s2', p, r))
\end{aligned}
$$

２． super-module m1 のメソッドを sub-module $m_i (i = 2, 3)$ で拡張するとき、m_i におけるメソッドの事前条件・事後条件が以下のように書ける。

$$
\begin{aligned}
&R_i(s1, sO, s_i, p) \equiv RO(s1, sO, p)\\
&E_i(s1, s1', sO, sO', si, si', p, r) \equiv\\
&\quad EO(s1, s1', sO, sO', p, r) \wedge E_i'(s_i, s_i', p)
\end{aligned}
$$

4.5 MixJuice における他の拡張ルール

mixin における Plus ルール、 Functional Protocol ルール、Disjoint Branch ルールは、素朴な方法で MixJuice 向けに修正することができる。具体的には、mixin におけるルールにおいて「mixin がスーパーク

MixJuice After ルール:

1. サブクラスによるメソッド拡張は、
   - メソッドの事前条件を弱くしてもよい。
   - 受け取った引数がスーパークラスの事前条件を満たしているなら、
     - スーパークラスの事後条件を強くしてもよい。
     - super を最初にちょうど１回呼び出さなければならない。（注１）
     - 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と違う返値を返してもよい。
     - スーパークラスから継承した状態を参照してもよい。
     - スーパークラスから継承した状態を更新してもよい。（注２）
     - サブクラス自身が追加した状態を参照・更新してもよい。
   - 受け取った引数がスーパークラスでの事前条件を満たしていないならば、
     - 事後条件に関する制約はない。
     - super を呼び出してはいけない。
2. sub-module によるメソッド拡張は、
   - super-module のメソッドの事前条件を変えてはいけない。
   - 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。
   - super から受け取った返値を参照してはいけない。（注３）
   - super-module から継承した状態を参照してはいけない。（注２）
   - super-module から継承した状態を更新してはいけない。
   - sub-module 自身が追加した状態を参照・更新してもよい。

図 7: MixJuice After ルール

ラスのメソッドをオーバーライドするとき」と書かれている部分を、「 subclass あるいは sub-module がメソッドをオーバーライドするとき」と修正すればよい。

5 検証支援ツール

我々は本論文で述べた全ての拡張ルール（mixin における拡張ルール４つと MixJuice における拡張ルール４つ）について、Common Lisp で書かれた簡単な検証支援ツールを使って、ダイアモンド継承における安全性を検証した。検証支援ツールは、各 mixin（または MixJuice module ）におけるメソッドの仕様と結合性判定基準を与えると、各仕様をリニアライズし、満たされるべき論理式を生成し、それをある程度簡略化して出力する。出力された論理式が恒真かどうかは、人間が確かめる。

メソッドの仕様は、リニアライズされた super の仕様のリストと変数名を引数として受け取り、S式で表現された論理式を返す lambda 式として定義する。

6 関連研究

本論文のように 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。

[1] では、アスペクト指向言語におけるアスペクトの干渉を、データフロー解析によって判定する方法を述べている。しかしこの判定方法は保守的であり、例えば Plus 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。Plus ルールでは返値が全ての mixin の返値に依存しているためである。

[15] では、 Hyper/J[13] 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 3.3節で定義した判定基準より保守的である。一方 [15] 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。

[2] 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る場合がある。

[17] 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、事前条件・事後条件、 behavioral subtyping の概念を使って、「安全な結合とは何か」を定義した。一言で言えば、結合後の個々のクラスの振る舞いが、結合前の振る舞いの behavioral subtype になるならば、その結合は安全である。

そして、 mixin および MixJuice モジュールの安全なダイアモンド継承を可能にする拡張ルールの具体例をそれぞれ４つ示し、その安全性を証明した。拡張ルールは、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、２つの mixin を安全に結合できることを保証する。

mixin に高い安全性と高い拡張の自由度の両方を持たせることができることを示した。

本研究を以下のように発展させることで、アスペクト指向言語による安全なプログラミングを強力に支援できると考えている。

- 実用上よく使われる拡張ルールのカタログ化。プログラマーが自分で拡張ルールを考え、その安全性を検証するのは難しい。しかし、検証ずみの拡張ルールカタログの中から１つを選んで、名前を宣言するだけならば容易である。拡張ルールカタログは、本論文で述べた４つの拡張ルールを一般化したものになるだろう。

- 拡張ルールを宣言・強制する機構のアスペクト指向言語への追加。これにより、メソッド実装が拡張ルールを守っているかどうかを、コンパイラによってある程度検査可能になる。本論文で述べた４つの拡張ルールの場合、かなりの部分がコンパイラによって検査可能である。例えば継承した状

態の参照・更新に関する制約は簡単に検査可能である。

- 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。Eiffel[11] の表明検査機構を拡張ルール用に拡張しアスペクト指向言語に導入することで、コンパイル時に検査できなかった拡張ルール違反を、実行時に検出することが可能になる。

- リニアライズに関する制約がルールに与える影響の形式的な考察。例えば local precedence order や final method などの導入は、多重継承が出来ないケースが発生する一方で、メソッド拡張の自由度を高くすると考えられる。

- ダイアモンド継承に限らない、一般の継承グラフでの、各拡張ルールの安全性の証明。これはクラス数およびモジュール数による帰納法で証明可能であろう。

参考文献

[1] Uwe Aßmann. AOP with design patterns as metaprogramming operators. Technical Report 28, Universität Karlsruhe, October 1997.

[2] Curtis Clifton and Gary T. Leavens. Observers and Assistants: A proposal for modular aspect-oriented reasoning. In Gary T. Leavens and Ron Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Langauges Workshop at AOSD 2002*, number 02-06 in Technical Report, pages 33–44. Department of Computer Science, Iowa State University, April 2002.

[3] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th international conference on Software engineering*, pages 258–267. IEEE Computer Society Press, 1996.

[4] Tzilla Elrad, Mehmet Aksits, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing aspects of AOP. *Communications of the ACM*, 44(10):33–38, 2001.

[5] Yuuji Ichisugi. MixJuice home page. `http://staff.aist.go.jp/y-ichisugi/mj/`.

[6] Yuuji Ichisugi and Akira Tanaka. Difference-based modules: A class-independent module mechanism. In *Proc. of the ECOOP2002*, LNCS 2374, pages 62–88, 2002.

[7] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of Metaobject Protocol*. MIT Press, 1991.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of the ECOOP'97*, LNCS 1241, pages 220–242, 1997. Invited Talk.

[9] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[10] R. C. Martin. The Liskov Substitution Principle. *C++ Report*, Mar 1996.

[11] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Ed.* Prentice-Hall, Inc., 1997.

[12] David A. Moon. Object-oriented programming with Flavors. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–8. ACM Press, 1986.

[13] 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.

[14] Clyde Ruby and Gary T. Leavens. Safely creating correct subclasses without seeing superclass code. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 208–228. ACM Press, 2000.

[15] Gregor Snelting and Frank Tip. Semantic-based composition of class hierarchies. In *Proc. of the ECOOP2002*, LNCS 2374, pages 562–584, 2002.

[16] G.L. Steele. *Common Lisp the Language, 2nd edition*. Digital Press, 1990.

[17] Naoyasu Ubayashi and Tetsuo Tamai. Aspect oriented programming with model checking. In *Proc. of AOSD 2002(1st International Conference on Aspect-Oriented Software Development)*, pages 148–154, April 2002.

[18] 一杉裕志. 拡張モジュール間の衝突が検出可能な言語拡張フレームワーク. 情報処理学会プログラミング研究会 発表資料, October 1999.

[19] 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, 18(6):54–58, 2001.