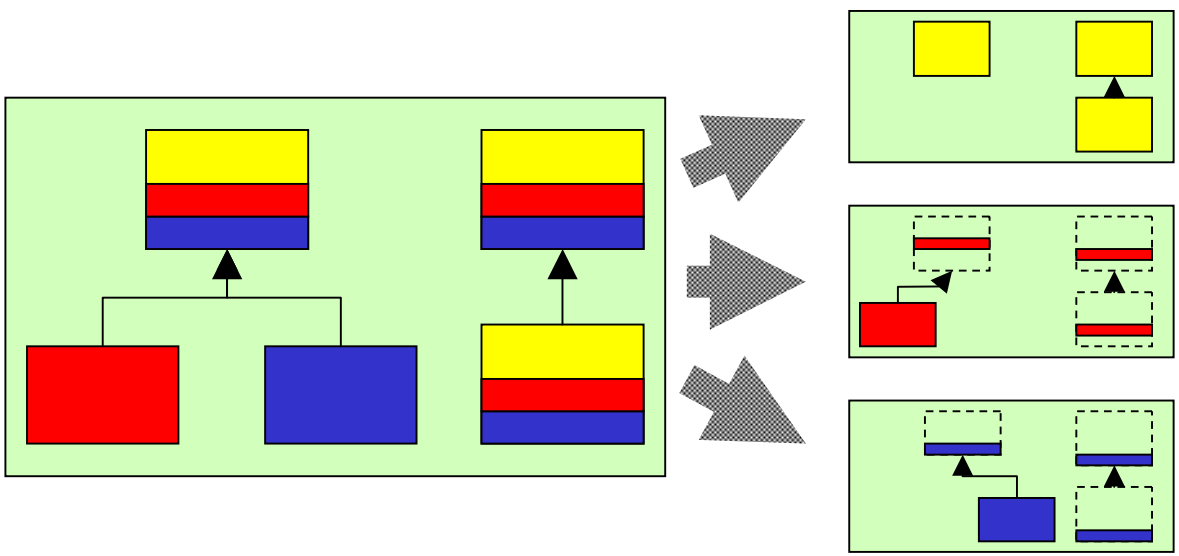
安全に結合可能なアスペクトを提供するためのルール

2002年9月11日

産業技術総合研究所　情報処理研究部門
一杉裕志、田中哲
国立情報学研究所／東京工業大学
渡部 卓雄

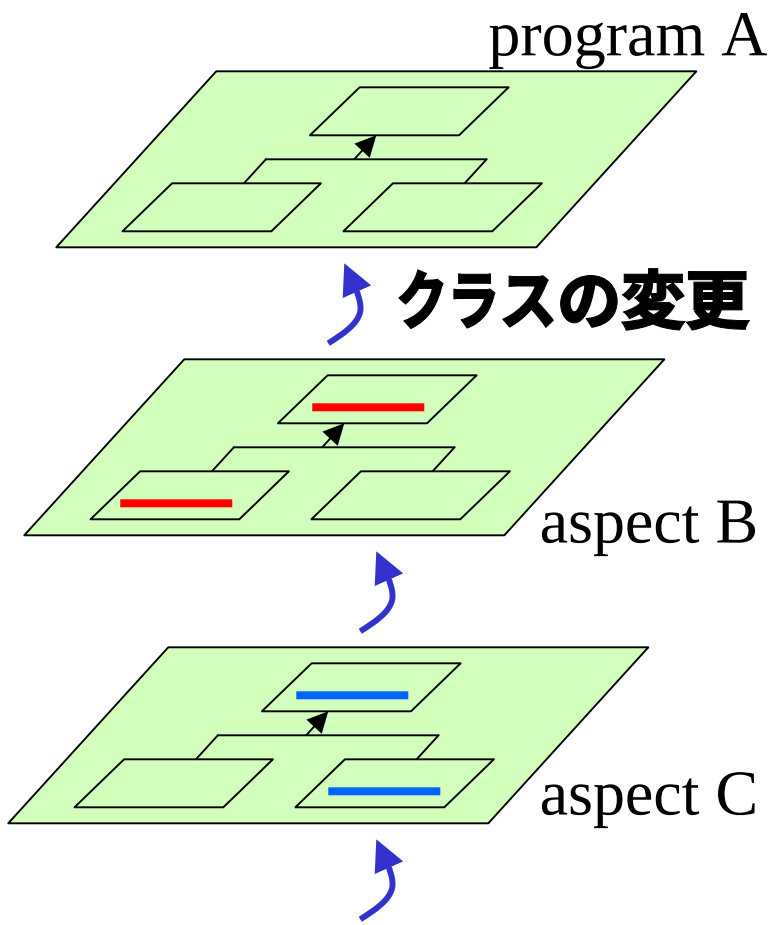
アスペクト指向言語とは？

- 複数のクラスを横断するコードを分離して、再利用可能にする言語
  - MixJuice, AspecJ, Hyper/J, DemeterJ, …
- 現在のオブジェクト指向言語の欠点を補う技術



アスペクト指向言語 に対する もっともな疑問

- 既存のクラスの動作が別のアスペクトによって変更できるのは危険では？
  - （アスペクトの結合の安全性が保証できるか？）
- 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？
  - （modular reasoning 可能か？）
- 本研究は、これらの疑問に答えるもの



発表の概要

- 安全性を保障するための基本方針
  - 「拡張ルール」でアスペクトの記述を制限
- 「安全性」について厳密に定義
  - 事前条件・事後条件、behavioral subtyping
- 拡張ルールとその安全性の検証
  - 一階述語論理
- 今後の実用化の方針
  - アスペクト指向プログラミングを簡単かつ安全に！

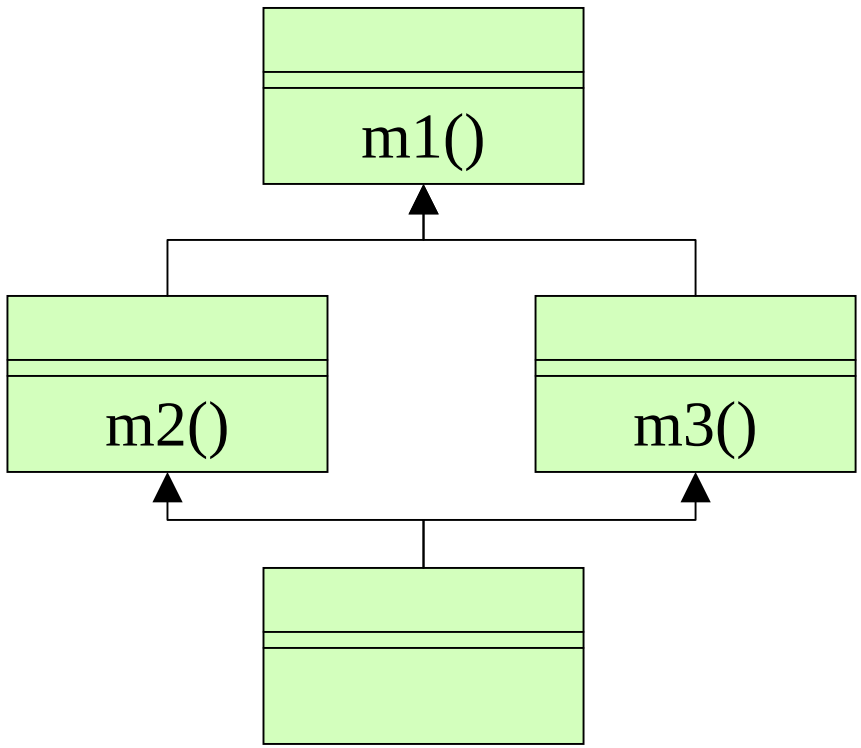
安全性を保障するための基本方針

アスペクト指向と多重継承

- 多くのアスペクト指向言語は、多重継承と機能的に類似
  - アスペクトとは関連する mixin をひとまとめにしたもの
  - アスペクトの結合の安全性の問題は、多重継承の安全性の問題とほとんど同一
- 本論文では「mixin の多重継承における安全性」を説明する
- 本論文の手法はアスペクト指向言語に素直に適用可能
  - すでに MixJuice 言語に応用

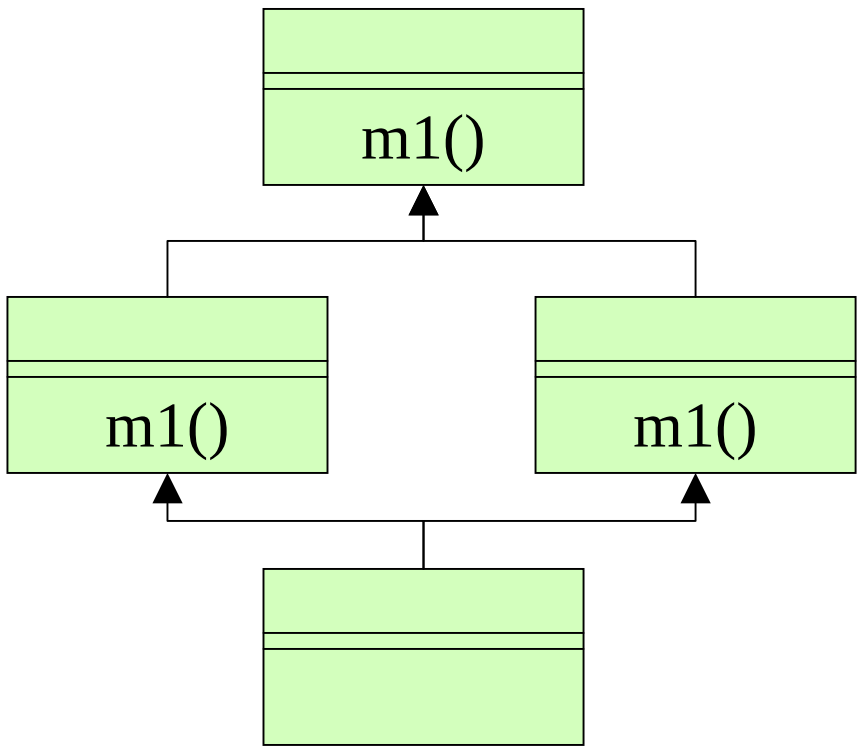
安全な多重継承

- 各クラスが定義するメソッドに重なりがないならば、(普通は) 安全に多重継承できる



危険そうな多重継承

- 同じメソッドを複数のクラスが override する場合は危険
  - アスペクト指向言語では意図的にこれをやることが多い
- この場合の安全性を保障するのが本研究の目的

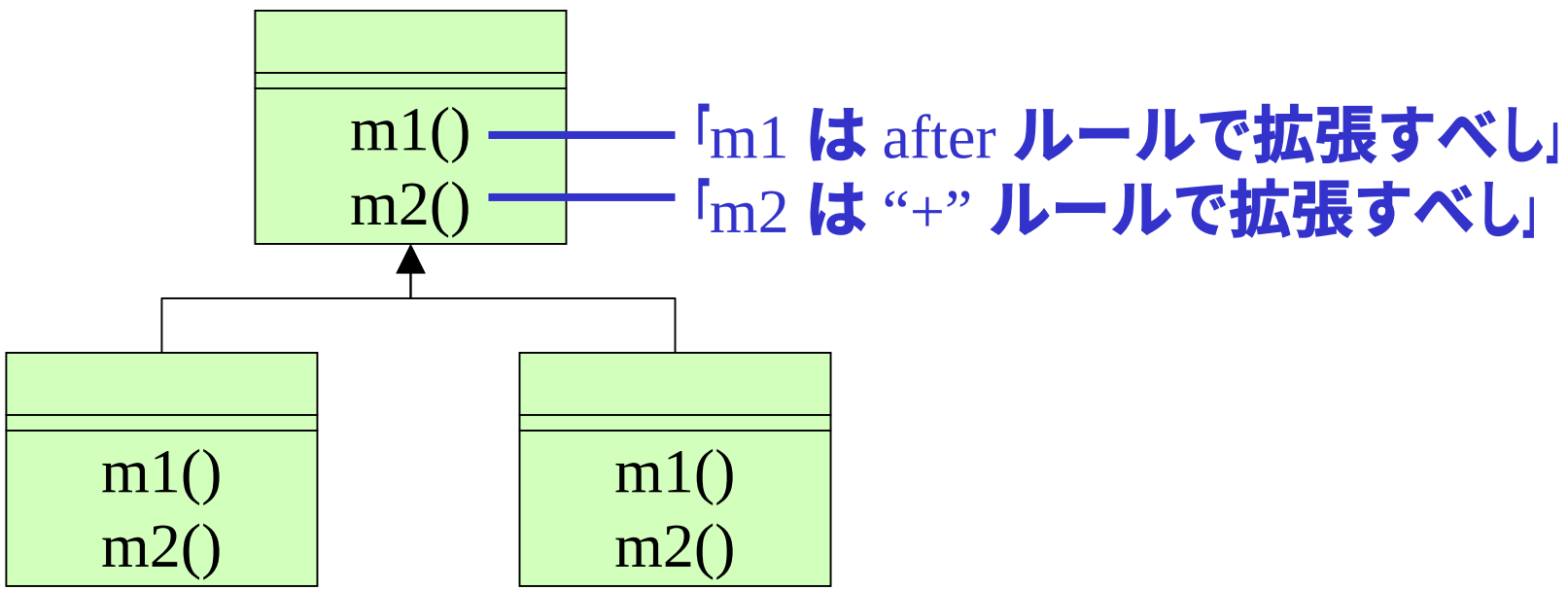


従来の方法

- Flavors[OOPSLA'86],CLOS method combination
  - before/after/around, call-next-method
  - +, list, nconc, …
  - **実用上便利かつ比較的安全**
- **問題点**
  - **安全性に理論的根拠がない**
  - **完全に安全ではない**
  - **組み込みの** method combination **だけでは不十分**
- **本研究：** method combination **を一般化し、かつ形式的に検証可能にする**

拡張ルール

- メソッドを定義すると同時に「拡張ルール」を宣言しておく
- すべての mixin は、決められた拡張ルールに従ってメソッドの振る舞いを拡張する



ここまでのまとめ

- 本論文では mixin の多重継承を使って説明
  - ただし素直にアスペクト指向言語に応用可能
- 同一メソッドを複数の mixin (アスペクト) が同時に override する場合の問題点を解決
- 「拡張ルール」で mixin (アスペクト) の記述を制限することで、安全な結合を保証する
- 拡張ルールは Flavors の method combination の安全性を形式的に検証可能にしたもの

「安全性」の定義

Design by Contract [Meyer'88]

- **事前条件：メソッド呼び出し前に成り立つ条件**
- **事後条件：メソッド呼び出し後に成り立つ条件**
- **メソッドの実装が変わっても、R1,E1が変わらなければ安全**

```
class Client {
  void foo(C1 c1){
      …
      // {R1}
      int r = c1.m(p);
      // {E1}
      …
}
```

```
/** 事前条件 R1 、
    事後条件 E1 */
class C1 {
  int m(int p){
     …
 }
}
```

安全なサブクラス 定義[Liskov'94]

- 「C2 が C1 に置き換え可能」
  ＝「C2 が C1 の behavioral subtype」
  ＝「C2 が事前条件を弱く、事後条件を強くする」
  ＝「(R1→R2)∧(R1→(E2→E1))」 ならば安全

```
class Client {
  void foo(C1 c1){
     …
     // {R1}
     int r = c1.m(p);
     // {E1}
     …
}
```

```
/** 事前条件 R1、
    事後条件 E1 */
class C1 {
  int m(int p){
     …
  }
}
```

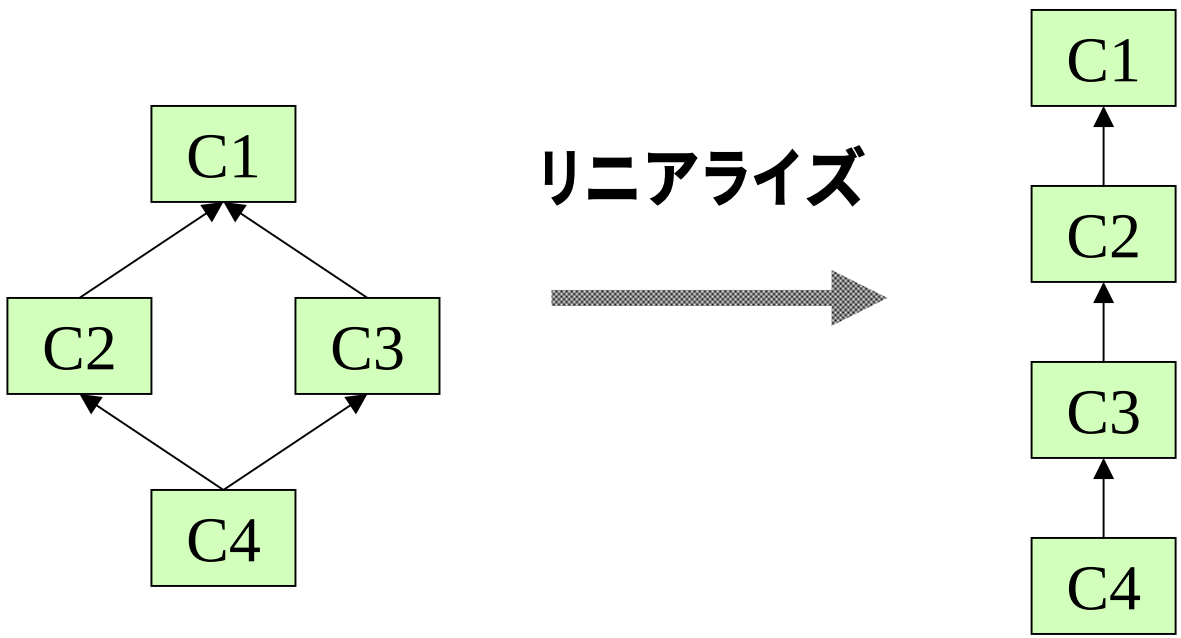
```
/** 事前条件 R2、
    事後条件 E2 */
class C2
    extends C1{
  int m(int p){
     …
  }
}
```

mixinの安全な多重継承

- **すべてのメソッド呼び出しについて、「多重継承後のメソッドの振る舞い」が、「メソッド呼び出しの記述時に想定していた振る舞い」の**
  behavioral subtype **になれば安全**

本論文の考察の対象

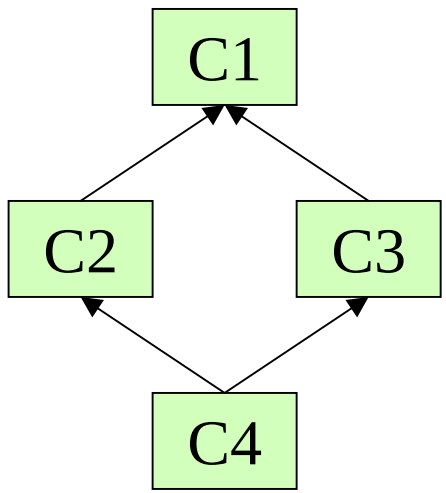
- クラスをリニアライズする仮想的な言語におけるダイアモンド継承　(一般化は容易)
- C4 はメソッド定義を持たないものとする



結合性判定基準(1/2)

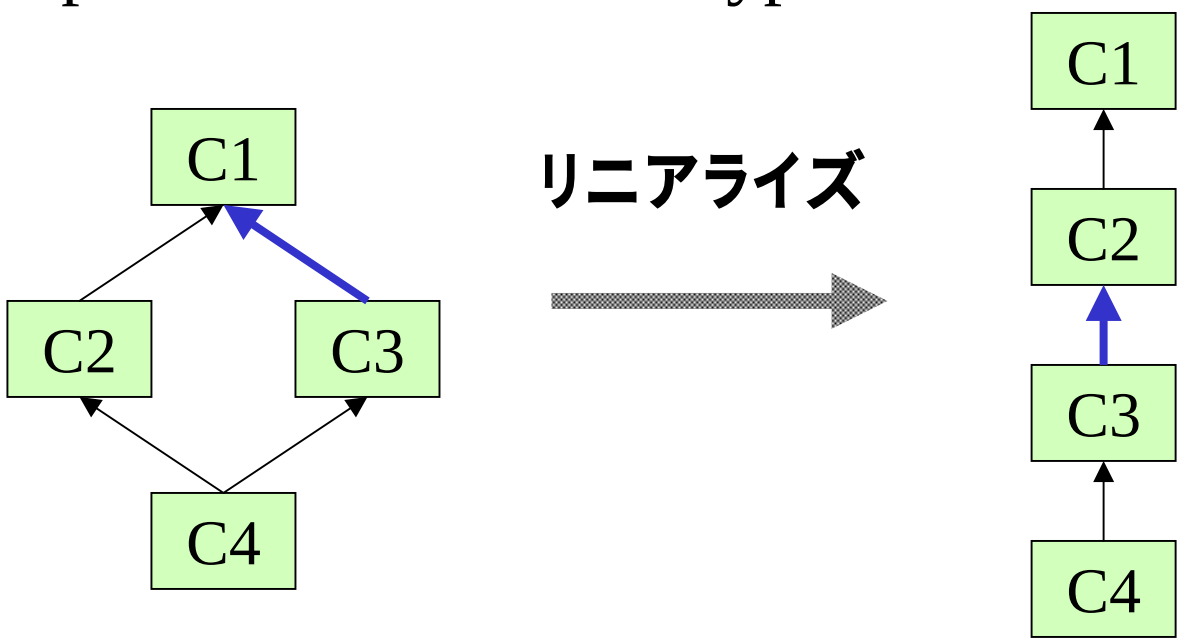
- **下記の6つの条件がすべてなりたてば、安全にダイアモンド継承できる**
  1. C2 **が** C1 **の** behavioral subtype
  2. C3 **が** C1 **の** behavioral subtype
  3. C4 **が** C2 **の** behavioral subtype
  4. C4 **が** C3 **の** behavioral subtype

(次ページにつづく)



結合性判定基準(2/2)

5. **リニアライズ後の** C2 **の** super **が** C2 **の定義時の** super **の** behavioral subtype
6. **リニアライズ後の** C3 **の** super **が** C3 **の定義時の** super **の** behavioral subtype



「拡張ルール」とその安全性の検証方法

拡張ルールの例：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。

（各mixinが扱うデータに依存関係を生じさせないルール）

after ルールの論理式を使った表現

- C1 の事前条件・事後条件を R1, E1、super の事前条件・事後条件を RO, EOとすると、Ci (i = 2, 3) の事前条件・事後条件 Ri,Ei が以下の形で書ける:
- Ri = RO(s1, sO, p)
- Ei = EO(s1, s1', sO, sO', p, r) ∧ Ei'(s1', si, si', p, r)

after ルールの検証

- **結合性判定基準の6つの条件をすべて満たしているかを検証すればよい**
- **例**:「C4 **が** C2 **の** behavioral subtype」**という条件**
- R2 = R1
- E2 = E1 ∧ E2’
- R4 = R1
- E4 = E1 ∧ E2’ ∧ E3’
- (R4 → R2) ∧ (R2 → (E4 → E2))
  = (R1 → R1)
  ∧(R1 → (E1 ∧ E2' ∧ E3' → E1 ∧ E2'))
  = true

“+” ルール

- after ルールと違って、 super の返値に非負の値を足せる。

コード例

```
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(); }
}
```

functional protocol ルール

- super の返値をキャッシュしてもよい

コード例

```
class C1 { String m(String s){ ...; return r; } }
class C2 extends C1 {
  String m(String s){
    String r = (String) cache.get(s);   // ハッシュ表のキャッシュ
    if (r == null){  r = super.m(s); cache.put(s, r); }
    return r;
  } }
class C3 extends C1 {
  String m(String s){
     別のアルゴリズムでキャッシュ
     return r;
   } }
```

disjoint branch ルール

- disjoint な定義域をそれぞれのmixinで処理

コード例

```
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); }
  }
}
```

MixJuice 言語への応用

- **本論文で述べた拡張ルールの定義･検証方法は、素直にアスペクト指向言語に応用可能**

例:

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。

$$R2(s1, sO, s2, p) \equiv RO(s1, sO, p) \vee R2'(s1, s2, p)$$
$$E2(\ldots, r) \equiv$$
$$(RO(s1, sO, p) \Rightarrow \exists s1'', r'' .$$
$$(EO(s1, s1'', sO, sO', p, r'')$$
$$\wedge E2'(s1'', s1', s2, s2', p, r'', r)$$
$$\wedge E1(s1, s1', p, r)$$
$$))$$
$$\wedge (\neg RO(s1, sO, p) \Rightarrow E2''(s1, s1', s2, s2', p, r))$$

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。

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

Common Lisp による簡単な検証支援ツール

- ルールを入力するとS式で論理式を出力
- 論理式が恒真かどうかは人間が確かめる

“+”ルールの記述例

```
(setq plus-c1
  (list #'(lambda (ROs EOs s1 s1d s2 s2d s3 s3d p r)
            `(R1 ,s1 ,p))
        #'(lambda (ROs EOs s1 s1d s2 s2d s3 s3d p r)
            `(and (E1 ,s1 ,s1d ,p r1) (>= ,r r1)))))
(setq plus-c2
  (list #'(lambda (ROs EOs s1 s1d s2 s2d s3 s3d p r)
            `,(funcall (car ROs) (cdr ROs) (cdr EOs) s1 s1d s2 s2d s3 s3d p r))
        #'(lambda (ROs EOs s1 s1d s2 s2d s3 s3d p r)
            `(and
              ,(funcall (car EOs)(cdr ROs)(cdr EOs) s1 s1d s2 s2d s3 s3d p
                        'rO2)
              (E2d ,s1d ,s2 ,s2d ,p r2)
              (>= r2 0)
              (>= ,r (+ rO2 r2))))))
```

関連研究

- アスペクト結合後に衝突を検出
  - データ依存解析する[Assman'97]
  - 継承関係・呼び出し関係を調べる[Snelting'02]
  - モデル検査ツールを適用[Ubayashi'02]
- アスペクト追加方法に制限を加える
  - 追加先にaccept 宣言を書かせる[Clifton'02]
  - 制限が強すぎる
- 本研究の特徴
  - behavioral subtyping を応用
  - 結合する前に安全性が保証できる
  - アスペクトによる拡張の自由度が高い

今後の実用化の方針

- 実用上よく使われるルールのカタログ化
  - プログラマーはカタログの中から選択
- 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
  - かなりの部分をコンパイラがチェック可能
- 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 の表明検査機構をアスペクト指向言語用に拡張
- 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

まとめ

- **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**
- **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複数示し、安全性を証明した**
- **今後の実用化の方針を示した**