2E−6

並列オブジェクト指向言語への安全な継承の導入について

脇田 建　松岡 聡
東京大学　理学部

1 はじめに

並列オブジェクト指向言語では，同期制約の記述を継承することとの困難が指摘されて以来 [1, 2]，さまざまな言語で同期制約の記述の工夫が図られてきた．その多くのものは，同期制約を受理可能なメッセージの集合で表すものであったが，本稿はその方法の問題点を指摘し，それに対する解決として同期制約の論理式による表現法を挙げる．さらに，このように表現されたプログラムをプログラム変換を用いて実現することを提案する．

2 並列オブジェクト指向と同期制約

われわれがとりあげる世界は，互いにメッセージで通信しあうオブジェクトから成りたつ．各オブジェクトは到達したメッセージを一つずつ受理し，クラスの定義に基づいた振舞いをする（メソッドの起動）．この振舞いとしては オブジェクトの状態変化，メッセージの送信，新しいオブジェクトの生成などがある．メッセージはオブジェクトの状態によらずに到達するため，オブジェクトの状態の整合性をとるためには受理するメッセージに制約を設けなくてはならない．この制約を我々は同期制約 (*synchronization constraint*) と呼ぶ．メソッド put と get が定義された有限長バッファを考えてみよう．はじめは，空のために put メッセージのみが受理可能である．したがって，この状態では get メッセージの受理を禁止する制約がある．データが挿入されると get メッセージが受理可能となり，バッファが満杯になると put メッセージは受理できなくなる．このようにオブジェクトの内部状態の変化に伴い同期制約は変化する．

3 Accept Set による同期制約の指定

同期制約は，オブジェクトの内部状態に対する受理可能なメッセージの集合 (*accept set*) によって指定できる．この方法による有限長バッファクラスの定義を示す (図 1)．記法は [3] にしたがい，実際のデータの格納，取り出しをするコードは省略した．in, out は今までに put, get が起動された回数であり，バッファの中のデータ数を与える．behavior 宣言で三つの accept set empty, partial, full を宣言し，各メソッド定義の最後にある条件分岐中の become 文は状態変化に伴い accept set を変化させ，同期制約の変化を表現する．並列オブジェクト指向言語と継承の相性の悪さが指摘されて以来 [1, 2]，継承を扱う多くの並列オブジェクト指向言語がこのような accept set の指定によって同期制約を表現してきた．しかし，この方法には次例が示す問題点がある．

クラス b-buf のサブクラスとしてメソッド get2 を加えたクラス x-buf を定義することを考える．get2 は，バッファに二つ以上のデータが存在する状態で二つのデータを同時に取りだす．そのため，バッファにたかだか一つしかデータがないときには get2 は受理不能である．このような同期制約を表現するために，データがちょうど一つある状態に対応した accept set x-one を導入する [4]．各 accept set には 新しいメソッド get2 の追加に対して適当な再定義が必要となる．しかし，いっそう深刻な問題は親のクラスで定義された二つのメソッド (get と put) が共に再定義されることである．accept set の指定による同期制約の表現では，メソッドの本体の最後に変化した内部状態を調べ accept set を変化させるコードがくるが，新たに accept set が追加される場合には，それに対応した内部状態に関わるメソッドはすべて定義変更が必要となり，その結果，プログラミング言語の継承機構はほとんど役にたたないものとなる [4].

```
Class b-buf: Object {
    int in, out;
  behavior: empty   = {put};
            partial = {put, get};
            full    = {get};
  public: b-buf() { in = out = 0;
                    become empty; }
    put() { in++;  /* insert an item */
            if (in==out) become full;
            else become partial; }
    get() { out++; /* remove an item */
            if (in==out+N) become empty;
            else become partial; } }
```

図 1: The Bounded Buffer Class

4 Guard による同期制約の指定

accept set の指定による同期制約の表現は失敗した．その原因は，メソッドに内部状態を accept set に反映するようなコードがあり，それが subclass の同期制約の影響を受けるためである．各メッセージの受理条件を内部状態に対する論理式 (*guard*) として与えれば，サブクラスのメソッドとは独立に同期制約を記述でき，b-buf で定義されたメソッドを x-buf にそのまま継承できる (図 2)．各メソッドの起動条件は when (条件) で指定され，この条件を満たすときのみメソッドは起動可能となる．

この方法により，表現上の問題は解決されるが，効率的な実装は困難になる．accept set 指定の場合はメッセージキュー中の accept set に含まれるメッセージを探せばよく，各メソッドごとにキューを持たせることで効率的に実装できる．一方，guard 指定による場合，キューの中の各メッセージごとに対応する guard を評価する素朴なやり方では効率が悪い．ここではプログラム変換とその後の最適化による，より効率的な実現法を提案する．

A Safe Inheritance Scheme for Concurrent Object-Oriented Languages
Ken Wakita, Satoshi Matsuoka, and Akinori Yonezawa
Faculty of Science, University of Tokyo

```
Class b-buf: Object {
     int in, out;
  public: b-buf() { in = out = 0; }
    put() when (in < out + N) { in++; }
    get() when (in >= out + 1) { out++; }
}

Class x-buf: b-buf { /* subclass of b-buf */
  public: x-buf()
    get2() when (in >= out + 2) { out += 2; } }
```

図 2: b-buf and x-buf redefined

4.1 プログラム変換

ここで述べるプログラム変換の目的は，メッセージキューを走査する際の guard 式の評価をなくすことである．このため，各 guard に対応して，真理値をもつ条件変数を導入する．この変数の値は対応した guard の評価値となるよう設定される．メソッドの定義は条件変数を用いて次のように変換する．

```
m(...) when(<guard>) { <body of method m> }
=> m(...) on c { <body of method m> check(); }
```

ここで，c は <guard> に対応した条件変数であり，m が起動されるのは c の値が真の時のみであることを保証する．各メソッドの最後に手続き check が呼びだされ，オブジェクトの内部状態が条件変数に反映される．この変換を x-buf に適用すると 図3のようになる．上の変換の正当性は変換前後のメッセージ取りだし機構を形式化し，それらの振舞いの bisimilarity を導くことで示せる．

変換後のプログラムの実行にあたって各メソッドに独立なメッセージキューを持たせる (図4)．オブジェクトが次の実行のためのメッセージを選ぶときには，真の値を持った条件変数に対応したメソッドのキューから選べばよい．例えば，長さ 8 のバッファに一つのデータが入っているとしよう．この時，c1, c2, c3 の値はそれぞれ，真，真，偽となる．したがって，次に起動可能なメソッドは put, get のいずれかとなり，それらのキューの中に受理可能なメッセージが存在する．get が受理されると，check の呼びだしによって，条件変数の値はそれぞれ真，偽，偽となって，put のみが起動可能となる．この実現法により各メソッドの起動後の guard の評価は一回に限られ，条件変数から受理すべきメッセージの集合が定まるため，キューの走査も必要なくなる．

4.2 最適化

前述のプログラム変換だけでは効率化のために十分ではない．実際，メソッドの数が増えた場合には手続き check の実行が大きな負担となる．しかし，プログラム変換によって得られたプログラムに対して，以下に示すような方針で最適化をすることで，効率的なコードが得られる (図5)．

- 依存性の検証による評価の省略．
- 条件変数の評価順の変更．
- 状態を変えないメソッドからの check 呼びだしの削除．
- 同じ同期制約をもつメソッドに対する条件変数の融合．

これらの手法及び従来の言語に対する最適化技法によって，accept set の実装と同程度の性能のメッセージ選択と同期の実現が可能になることを期待している．

```
Class x-buf {
     int in, out;
     condition c1, c2, c3 ;
  public: b-buf() { in = out = 0; check(); }
         put() on c1 { in++; check();  }
         get() on c2 { out++; check(); }
         get2() on c3 { out += 2; check(); }
  local: check() {
    c1 = (in < out + N);  /* full? */
    c2 = (in >= out + 1); /* more than one? */
    c3 = (in > out + 2);  /* more than two? */ } }
```

図 3: Transformed code

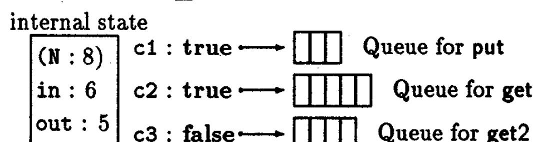

図 4: Implementation of condition variables

```
Class x-buf {
     int in, out;
     condition c1, c2, c3;
  public x-buf() { in = out = 0; check2(); }
        put() on c1 { in++; check1(); }
        get() on c2 { out++; check2(); }
        get2() on c3 { out += 2; check2(); }
  local: check1() { c1 = (in < out + N);
             c2 = true;
             c3 = (!c1 && (in >= out + 2)); }
        check2() { c1 = true;
             c3 = (in >= out + 2);
             c2 = (!c3 && (in >= out + 1)); } }
```

図 5: Optimization on transformed code

5 おわりに

accept set による同期制約の問題点を指摘した．Guard を使うことにより，同期制約が宣言的に記述できることを示し，その実装法につき考察した．今後，この言語を実際に実装する予定である．本稿を書くにあたって，益田 隆教授，米澤 明憲教授，柴山 悦哉氏，渡辺 卓雄氏，武山 誠氏には適切な助言をいただいた．

参考文献

[1] P. America. “Inheritance and subtyping in a parallel object-oriented language.” In Lecture Notes in Computer Science vol. 276, Springer Verlag, 1987.

[2] J. P. Briot and A. Yonezawa. “Inheritance and Synchronization in Concurrent OOP.” In Lecture Notes in Computer Science vol. 276, Springer Verlag, 1987.

[3] D. G. Kafura and K. H. Lee. “Inheritance in actor based concurrent object-oriented languages.” In ECOOP ’89, pages 131–145. Cambridge Univ. Press, 1989.

[4] S. Matsuoka and A. Yonezawa. “並列オブジェクト指向言語における Synchronization Constraints と継承について.” 情報処理学会第 41 回全国大会講演論文集, 1990.