Anomalia dell’Ereditarietá

Sincronizzazione e Ereditarietà hanno caratteristiche conflittuali che inibiscono il loro uso simultaneo senza dover rompere pesantemente l’incapsulation degli oggetti.

In sostanza: mantenere i vincoli di sincronizzazione dei metodi di una classe durante la definizione di una classe derivata impone la necessità di ridefinire i metodi inficiando le potenzialità dell’ereditarietà.

Costrutti di Sincronizzazione degli OOCPL

Ogni linguaggio per la programmazione ad oggetti in ambiente concorrente fornisce il proprio meccanismo di sincronizzazione.

Il meccanismo più diffuso è quello dei *bodies*: il body un metodo dell’oggetto con un proprio thread di controllo. Il thread del body rimane in attesa di ricevere delle richieste destinate all’oggetto corrispondente. Ricevuto il messaggio sarà il body ad occuparsi di attivare il metodo corrispondente.

Esempio di Body

```
class b−buf : ACTOR {
    int in, out, buf[SIZE];
  public:
    void b−buf() { in = out = 0;}
    void process_put() {  in++; }
    int process_get() {  out++; }
    void body() {
      loop {
         select {
            accept get() when (!(in == out))  start process_get();
         or
            accept put() when (!(in == out+SIZE))  start process_put();
         }
      }
    }
}
```

Behavior Abstractions

Si associa ad un insieme di messaggi accettabili (*accept sets*) un identificatore:

empty = {put}
partial = {get, put}
full = {get}

gli identificatori definiscono lo stato dell'oggetto e quali metodi può accettare in tale stato.

Tramite la primitiva become l'oggetto passa da uno stato all'altro.

Behavior Abstraction (Esempio)

```
class b−buf : ACTOR {
  int in, out, buf[SIZE];
 behavior:
  empty = {put};
  partial = {put, get};
  full = {get};
 public:
  void b−buf() {
    in = out = 0;
    become empty;
  }

  void put() {
    in++;
    if (in == out + size) become full;
    else become partial;
  }

  int get() {
    out++;
    if (in == out) become empty;
    else become partial;
  }
}
```

```
class x−buf: b−buf {
 behavior:
   x−empty = renames empty;
   x−partial = {put, get, last} redefines partial;
   x−full = {get, last} redefines full;
 public:
   void x−buf(){}


   int last() {
     in−−;
     if (in == out) become x−empty;
     else become x−partial;
   }
}
```

Behavior Abstraction (Segue)

```
class x−buf2: b−buf {
 behavior:
   x−empty = renames empty;
   x−one = {put, get};
   x−partial = {put, get, get2} redefines partial;
   x−full = {get, get2} redefines full;
 public:
   void x−buf2(){}

   void get2() {
     out += 2;
     if (in == out) become x−empty;
     else if (in == out+1) become x−one
     else become x−partial;
   }
```

```
 void get() {
    out++;
    if (in == out) become x−empty;
    else if (in == out+1) become x−one
    else become x−partial;
  }

 void put() {
    in++;
    if (in == out) become x−empty;
    else if (in == out+1) become x−one
    else become x−partial;
  }
}
```

Enabled Set

Gli insiemi dei messaggi accettati (accepted set) sono cittadini di prima classe.

- **Enable(<messaggi accettabili>)**
- **Enable<method>() {return Enable(<messaggi accettabili>)}**
- **become(<enabled set>,(<nuovo−stato>))**

La natura degli Enabled Set permette di localizzare le ridefinizioni.

Enabled Set (Esempio)

```
class b−buf : ACTOR {
   int in, out, buf[SIZE];
 private:
   Enable empty() {return enable([put])};
   Enable partial() {return enable([put, get])};
   Enable full() {return enable([get])};
 public:
   void b−buf() {
     in = out = 0;
     become(empty(), (in,out,buf));
   }

  void put() {
    in++;
    if (in == out + size) become(full(), (in,out,buf));
    else become(partial(), (in,out,buf));
  }

  int get() {
    out++;
    if (in == out) become(empty(), (in,out,buf));
    else become(partial(), (in,out,buf));
  }
}
```

Enabled Set (Esempio Segue)

```
class x−buf2: b−buf {
 private:
   Enable empty() {return Enable(empty?)+super empty()};
   Enable one() {return Enable(get,put,empty?)};
   Enable partial() {
      if (in == out+1) return super partial() + Enable(empty?);
      else return super partial() + Enable(empty?, get2);
   };
   Enable full() {return super full() + Enable(empty?)};
 public:
   void x−buf2(){}

   void get2() {
     out += 2;
     if (in == out) become(empty(), (in,out,buf));
     else if (in == out+1) become(one(), (in,out,buf));
     else become(partial(), (in,out,buf));
   }
}
```

Method Guards

Si attacca un predicato ad ogni metodo: il metodo può essere eseguito solo se il predicato è verificato.

<method>(<arguments>) when (<guard>) {method body}

Il predicato trasforma ogni metodo in una regione critica.

Method Guards
(Esempio)

```
class b−buf : ACTOR {
  int in, out, buf[SIZE];
 public:
  void b−buf() { in = out = 0; }
  void put()  when (in < out + size) { in++; }
  int get() when (in >= out+1) { out++; }
}

class x−buf2: b−buf {
 public:
  void x−buf2(){}
  void get2() when (in >= out+2) { out += 2; }
  void empty?() when (true) {return in == out}
}
```

Method Guards
(Esempio Anomalia)

```
class gb−buf: b−buf {
  bool after−put;
 public:
  void gb−buf() {after−put = false;}
  void gget() when (!after−put && (in>=out+1)) {out++; after−put=false;}
  void put() when (in < out+size) {in++; after−put = true;}
  void get() when (in >= out+1) {out++; after−put = false;}
}
```

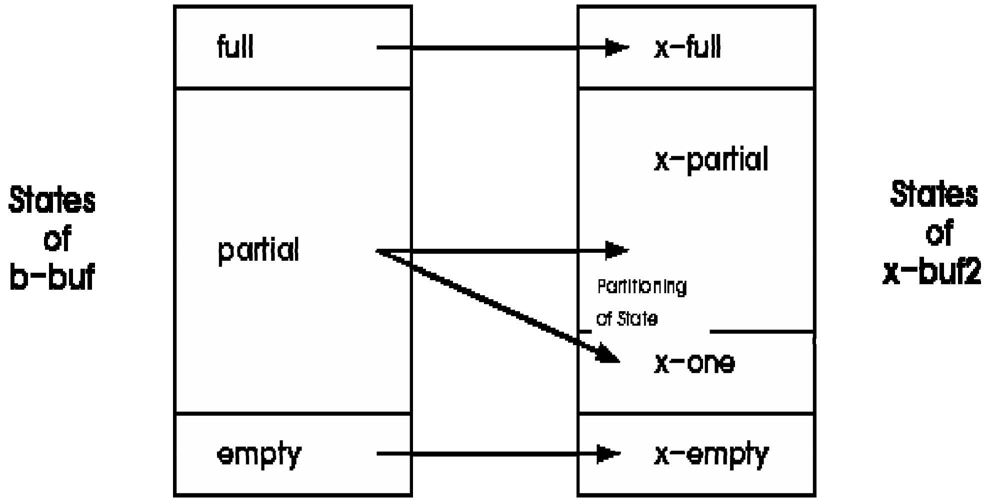
```
class Lock: ACTOR {
  bool locked;
 public:
  void Lock() {locked = false;}
  void lock() when (!locked) {locked=true;}
  void unlock() when (locked) {locked=false;}
}

class lb−buf: b−buf, Lock {
 public:
  void lb−buf() {}
  void put() when (!locked && (in < out+size)) {in++;}
  void get() when (!locked && (in >= out+1)) {out++;}
}
```

Tipi di Anomalie

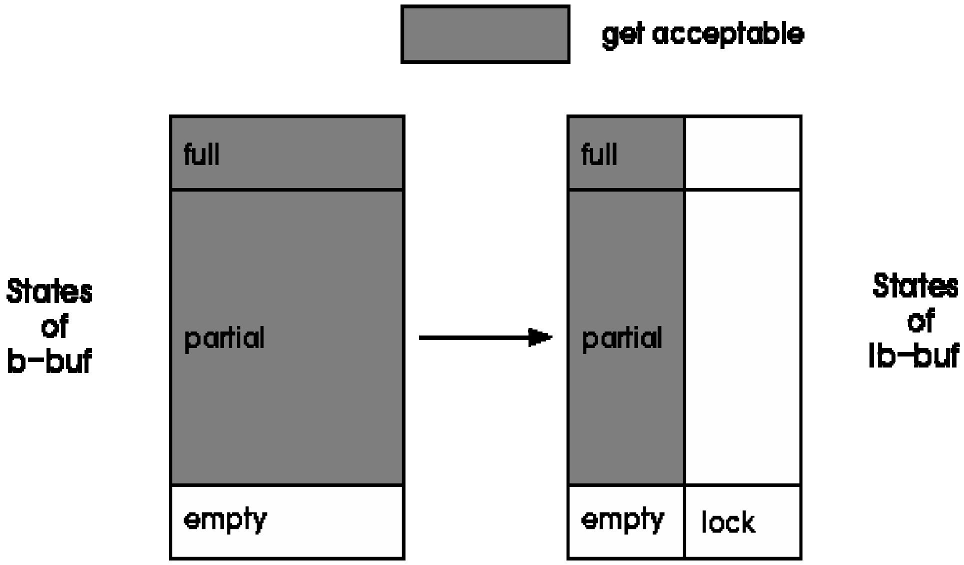
Tre tipologie di Anomalie dovute ai conflitti tra meccanismi di sincronizzazione ed ereditarietà:

- **Partizionamento degli stati accettabili**
  es. get2, gget
- **Suscettibilità alla storia degli stati accettabili**
  es. gget, lock
- **Modifica degli stati accettabili**
  es. lock

Partizionamento degli Stati



Modifica degli Stati



Soluzioni Proposte

<u>La proposta di Shibayama</u>:

Riduzione del codice da dover ridefinire tramite uno schema di ereditarietà a grana fine. Tre tipi di metodi: primary, constraint e transition.

<u>La proposta di Frølund</u>:

Guardie al negativo. I predicati definiscono quando un metodo non può essere attivato. Metodi ridefiniti possono essere attivati solo se tutte le guardie degli antenati sono valutate in falso.

<u>La proposta di Matsuoka e Yonezawa</u>:

Uso della riflessione per incapsulare i vincoli di sincronizzazione nel meta-livello.

Riferimenti Bibliografici

- **Distributed Object–Based Programming Systems. Roger Chin and Samuel Chanson. In ACM Computer Surveys, Vol. 23(1), March 1991.**
- **Analysis of Inheritance Anomaly in Object–Oriented Concurrent Programming Language. Satoshi Matsuoka, and Akinori Yonezawa. In G. Agha, P. Wegner, and A. Yonezawa Editors Research Directions in OO Concurrent Programming. MIT Press. 1993.**