Anomalia da Herança em Programação Concorrente Orientada a Objetos: Uma Proposta de Tratamento[1]

Laís do Nascimento Salvador*
Liria Matsumoto Sato†

*LSI- Laboratório de Sistemas Integráveis

†PCS- Departamento de Engenharia de Computação e Sistemas Digitais

Escola Politécnica da Universidade de São Paulo
Av. Prof. Luciano Gualberto, travessa 3, no 158
CEP. 05508-900 - São Paulo, SP
tel: 818-5589
e-mail: lnsalvad@lsi.usp.br e liria@pcs.usp.br

RESUMO

Recentemente, o termo Anomalia de Herança tem sido mencionado em vários trabalhos relacionados à Programação Concorrente Orientada a Objetos. Esta anomalia diz respeito ao fato de que herança e sincronização em sistemas concorrentes com objetos, normalmente, entram em conflito, resultando na necessidade de redefinição de métodos herdados. Este artigo apresenta uma discussão deste problema e uma proposta de como tratá-lo dentro do escopo da linguagens concorrentes baseadas em C++.

ABSTRACT

The term Inheritance Anomaly has been recently used in Object-Oriented Concurrent Programming. This anomaly is related to the fact that inheritance and synchronization constraints in concurrent object systems often conflict with each other, requiring redefinitions of some inherited methods. We present a brief discussion of inheritance anomaly and also propose a scheme of how to treat this problem in the context of C++ based concurrent languages.

[1] Este trabalho foi financiado pelos seguintes órgãos: CNPq, projeto FINEP/PAD no. 5.6.94.0260.00 e projeto FINEP/RECOPE no. 3607/96

1. INTRODUÇÃO

Em um programa orientado a objetos concorrente as mensagens podem chegar a um objeto em qualquer ordem. Isto vem do fato de que muitos processos estão ativos simultaneamente. Estas mensagens podem resultar em alguma modificação no estado de um objeto. Isto pode causar conflitos se as ações não estão sincronizadas umas com as outras.

Uma característica importante da programação orientada a objetos é a herança. A herança permite que uma nova classe de objetos possa ser construída a partir de alguma classe preexistente. A nova classe somente declara as novas características que os seus objetos devem possuir em adição às características declaradas na classe preexistente, chamada de super classe. As características declaradas nas super classes são herdadas pelos objetos da nova classe, chamada subclasse.

Em um ambiente paralelo há a necessidade de construções de sincronização. As construções de sincronização devem ser herdadas através da hierarquia de classes. "Infelizmente, tem sido mostrado que o código de sincronização não pode ser efetivamente herdado sem redefinições não-triviais de classes" [Mat93]. Este conflito é conhecido como Anomalia de Herança na literatura. Um fato notável é que a ocorrência desta anomalia depende do esquema de sincronização da linguagem. O coração do problema são os conflitos semânticos entre as descrições de sincronização nas classes e o mecanismo de herança e não como estas características são implementadas.

Este artigo apresenta a seguinte estrutura: na seção 2 é apresentado o problema da anomalia da herança baseado em [Mat93]; na seção 3 é apresentada a nossa proposta para minorar os efeitos da anomalia da herança de forma teórica; na seção 4 é apresentada uma extensão para linguagens concorrentes baseadas em C++ [Str86], de acordo com a proposta da seção 3; na seção 5 apresentamos uma biblioteca de moldes ("templates") em C++ que implementa as construções da seção anterior; na seção 6 são apresentadas as conclusões finais e perspectivas de trabalhos futuros.

2. ANOMALIA DA HERANÇA

Na Programação Orientada a Objetos Concorrente, um esquema natural de sincronização é adicionar um predicado a cada método, na forma de uma guarda,

transformando cada objeto em uma região crítica condicional. A estrutura apresenta a seguinte sintaxe:

<method name> (<formal arguments>) **when** (<guard>) { <body of method >}

onde "guard" é uma expressão booleana cujos termos são constantes ou atributos da classe limitados a valores primitivos. A semântica da estrutura guardada é a seguinte: o método é apenas invocado quando a avaliação da guarda for verdadeira. Nesta seção usaremos guardas, uma forma bastante conhecida para especificar sincronização.

As anomalias de herança, decorrentes do uso de construções de sincronização, podem ser classificadas em três categorias principais:

1- Partição do conjunto de estados aceitáveis;
2- Sensibilidade ao histórico das chamadas;
3- Modificação do conjunto de estados aceitáveis

Para explicar melhor estas anomalias, será usado o exemplo clássico do "buffer", com os métodos put() e get(). Nesta seção o código dos métodos está simplificado, para manter uma coerência com o texto descrito em[Mat93], nas próximas seções o código é melhor especificado.

Uma classe de "buffers" limitados é mostrado na figura 2.1. No exemplo da classe b-buf, a guarda (in < out+SIZE) foi adicionada ao método put(), para garantir que o método put() não será invocado quando o buffer estiver cheio e a guarda (in>=out+1) para garantir que o método get() não será invocado quando o buffer estiver vazio.

```
Class  b-buf: ACTOR {
       int in, out, buf[SIZE];

public:
       void b-buf() {in=out=0;}
       void put(int x) when(in < out+SIZE){buf[in%SIZE]=x; in++;}
       int get() when(in >= out+1) {return buf[out++%SIZE];}
};
```

Figura 2.1 - Classe b-buf

Segue a descrição das anomalias segundo [Mat93]. Os tipos de anomalias serão ilustrados através de exemplos baseados na classe b-buf, ilustrada na figura 2.1.

2.1Partição do Conjunto de Estados Aceitáveis

Esta anomalia é comum em modelos computacionais baseados em atores[Agh86], onde o próximo estado do objeto é derivado e atribuído explicitamente

após a execução de cada método. Um exemplo para esta anomalia é a classe x-buf2 de buffers limitados, subclasse de b-buf, com dois métodos a mais: get2() e empty(). Os objetos da classe x-buf2 podem aceitar a mensagem get2() para remover dois elementos do buffer de forma atômica.

O conjunto de estados aceitáveis dos objetos é particionado em: não-cheio (put()), não-vazio (get()), no-mínimo-2 (get2()). A partição dos estados aceitáveis não é um problema para guardas, porque elas estão capacitadas para julgar diretamente quando uma mensagem é aceitável ou não, baseadas no estado corrente, isto é, nos atributos da classe. Logo, mesmo que novos métodos sejam adicionados, as guardas não necessitariam ser redefinidas.

```
Class  x-buf2: public b-buf {
public:
      void x-buf2()
      int get2() when(in >= out+2){//código p/ remover 2 element.
                                        out+=2;}
      int empty() when (true) {return in == out;}
};
```

Figura 2.2 - Classe x-buf2

2.2 Sensibilidade ao Histórico das Chamadas

Esta anomalia é exemplificada através de uma variante da classe b-buf, chamada gb-buf. Esta classe é uma subclasse de b-buf adicionando-se um único método: gget(). O comportamento de gget() é quase idêntico ao método get(), com a única restrição que ele não pode ser aceito imediatamente após a invocação do método put(). Esta classe é ilustrada abaixo:

```
Class gb-buf: public b-buf {
      int after-put;
public:
      void gb-buf(){after-put = false;}
      int gget() when (!after-put && in >= out+1)
            {after-put = false; return buf[out++%SIZE];}
      void put(int x) when (in < out+SIZE)
            {buf[in%SIZE]=x; in++; after-put = true;}
      int get() when (in >= out+1)
            {after-put = false; return buf[out++%SIZE];}
};
```

Figura 2.3 - Classe gb-buf

Observa-se que é necessário redefinir os métodos put() e get() para levar em conta o novo atributo after-put. A razão desta anomalia é que o estado imediatamente após a execução do método put(), não pode ser extraído do conjunto de atributos disponíveis em

b-buf, requerendo a adição de um novo atributo after-put. Como é necessária uma avaliação deste novo atributo em todos os métodos, há a necessidade de se redefinir, modificar os métodos previamente definidos. Esta anomalia se aplica para os casos onde o estado do objeto não registra "naturalmente" o estado necessário pelo requerimento de sincronização, sendo necessária a adição de novos atributos.

2.3 Modificação do Conjunto de Estados Aceitáveis

Esta anomalia se refere aos casos em que uma subclasse herda variáveis adicionais de uma superclasse e as variáveis herdadas devem ser detectadas nas construções de sincronização herdadas. O exemplo introduzido é a classe abstrata Lock. Uma classe abstrata sempre é usada juntamente com outra classe para criação de objetos. O propósito é combinar o uso de classes abstratas com outras classes para oferecer algumas características específicas aos objetos. Neste exemplo a classe Lock pode ser usada juntamente com a classe b-buf para definir a classe lb-buf com características de regiões críticas, através do mecanismo de herança múltipla.

```
Class Lock: public Actor {
      int locked;
public:
      void Lock(){locked = 0;}
      void lock() when (!locked) {locked = 1;}
      void unlock() when (locked) {locked = 0;}

}
Class lb-buf: public b-buf, Lock{
public:
      void lb-buf();
      void put(int x)
            when (!locked && (in<out+SIZE))
            { buf[in%SIZE]=x; in++;}
      int get() when (!locked && (in >= out+1))
            { return buf[out++%SIZE];}
};
```

Figura 2.4 - Classes Lock e lb-buf

Os métodos da classe Lock são também sensíveis ao histórico de chamadas de forma similar ao método gget() do exemplo anterior. A diferença é que a execução dos métodos da classe Lock modifica o conjunto de estados aceitáveis sobre os quais os métodos herdados podem ser invocados. Isto é, a classe Lock introduz uma distinção pequena para o conjunto de estados sobre os quais os métodos get() e put() em lb-buf devem ser chamados. Isto requer a modificação das guardas dos métodos para considerar esta nova restrição de sincronização. Neste exemplo, as guardas devem ser totalmente rescritas para levar em conta a variável locked.

3.PROPOSTA

O objetivo deste trabalho é apresentar uma proposta de como especificar aspectos de sincronização, de forma a minorar os efeitos da anomalia de herança. Nesse trabalho, os aspectos de sincronização são classificados em: sincronização condicional e ação de sincronização.

3.1 Sincronização Condicional

A sincronização condicional é referente às condições nas quais um objeto deve ou não aceitar as mensagens. Esta especificação é realizada através de duas primitivas: **enable** e **disable**.

A primitiva **enable** possui a mesma semântica de uma guarda. De forma mais específica:

c **enable** *m* ↔ **when** *c m*, isto é, quando *c* for verdadeiro *m* está habilitado a ser invocado (executado).

onde *c*: condição e *m*: declaração de um método

Uma condição é uma expressão booleana cujos termos são constantes ou atributos da classe. Dentro desse contexto, *m* se refere à declaração do método, isto é, seu nome e argumentos formais.

A primitiva **disable** funciona como uma guarda negativa. Esta primitiva é baseada no trabalho de Frølund [Frø92]. Com esta primitiva, ao invés de se especificar as condições de habilitação de execução de métodos, especificam-se as restrições à execução do método. Mais formalmente:

c **disable** *m* ↔ **when not** *c m* ou
when *c* **not** *m*, isto é, quando *c* for verdadeiro *m* não está habilitado a ser executado.

Às primitivas **enable** e **disable**, podemos adicionar a cláusula **all-except**[Frø92]. Dessa forma:

c **enable-all-except** *m*
↔
when *c M-m*
onde *M*: conjunto de todos os métodos da classe *C* e das subclasses de *C*.
C: classe associada ao método *m*.

De forma similar:

c **disable-all-except** *m*

$\leftrightarrow$

when not *c M-m* ou

when *c* **not** *M-m*

3.2 Ação de Sincronização

Além das primitivas para especificar condições de habilitação ou restrição à execução dos métodos, usaremos também primitivas para especificar ações de sincronização [Neu91]. Através destas primitivas é possível especificar uma pré-ação ou uma pós-ação, isto é, ações que devem ser executadas antes ou após a execução de um método. As primitivas são **after** e **before**. Mais formalmente:

a **before** *m*: *a* deve ser executado antes de *m*

a **after** *m*: *a* deve ser executado após *m*.

onde *a*: ação de sincronização.

Uma questão a ser levantada é a diferença entre método e ação de sincronização. Neste trabalho, método se refere ao código de uma determinada função do objeto, sem considerar aspectos de sincronização. Por outro lado, uma ação é referente ao código de sincronização, isto é, comandos que são especificados na linguagem fonte que estão associados a aspectos de sincronização.

3.3 Notação

Segue a notação destas primitivas para especificar os aspectos de sincronização:

<constraint-synchronization>	::=	<conditional-synchronization> | <synchronization-action>
<conditional-synchronization>	::=	condition <enable-constraint> method | condition <restriction-constraint> method
<enable-condition>	::=	**enable | enable-all-except**
<restriction-condition>	::=	**disable | disable-all-except**
<synchronization-action>	::=	action <order-constraint> method
<order-constraint>	::=	**before | after**

3.4Herança de Aspectos de Sincronização

Com posse dessas primitivas, é possível expressar vários aspectos de sincronização. Não é obrigatória a utilização de todas estas primitivas para cada método.

Porém um fator primordial para minorar os efeitos da anomalia é declarar os aspectos de sincronização separadamente da especificação do código dos métodos. Dessa forma, a herança das construções de sincronização torna-se mais fácil, não havendo necessidade de se redefinir aspectos já existentes. Por isso nas definições acima, os aspectos de sincronização estão associados à declaração do método e não à sua definição (código).

Com a possibilidade da herança dos aspectos de sincronização, deve-se definir como é a semântica da composição dos aspectos de sincronização, nos métodos herdados.

No caso dos aspectos de sincronização condicional, temos dois tipos básicos: habilitação (primitiva **enable**) e restrição (primitiva **disable**). Assumindo que:

- C_2 é subclasse de C_1
- m é um método de C_1, herdado por C_2
- S_1 é uma condição de sincronização de m em C_1 e S_2 é uma condição de sincronização de m em C_2.

então a sincronização condicional para o método m na classe C_2, apresenta a seguinte semântica:

when (S_1 **and** S_2) m, onde S_1 e S_2: habilitação
when (S_1 **or** S_2) **not** m, onde S_1 e S_2: restrição
when (S_1 **and not** S_2) m, onde S_1: habilitação e S_2: restrição
when (not S_1 **and** S_2) m, onde S_1: restrição e S_2: habilitação

Em outras palavras, a herança de aspectos de sincronização usando a condição de habilitação é resolvida através da conjunção das condições. Enquanto que a herança de aspectos de sincronização usando a condição de restrição é resolvida através da disjunção de condições. Outro ponto é que a negativa de uma condição de restrição é equivalente a uma condição de habilitação.

No caso da herança de ações de sincronização, a semântica é bem mais simples. Ela é feita através da composição das ações, isto é, a concatenação do código das ações na ordem da hierarquia de classes.

4. PROPOSTA DE CONSTRUÇÕES PARA LINGUAGENS CONCORRENTES BASEADAS EM C++

A maioria das propostas assume o modelo computacional como sendo o de atores, que permite um nível maior de declaratividade. Como a pesquisa em programação paralela com C++ tem se desenvolvido bastante, vide [Wil96], resolvemos propor uma solução usando C++.

Como já foi visto anteriormente, para minorar os efeitos da anomalia da herança, um caminho importante é separar o código de sincronização do código dos métodos. Em função disso, propomos a adição de uma nova seção dentro da declaração de classes em C++ chamada **synchronizers** onde serão especificados os aspectos de sincronização usando a notação apresentada no tópico anterior. O formato geral dessa nova seção está ilustrado na fig. 4.1.

```
synchronizers:
      (condition) <disable| disable_all_except method(args<<);
      (condition) <enable  | enable_all_except method(args<<);
      { action }  <before  | after  method(args<<);
```

Figura 4.1 - Formato da seção synchronizers

Para manter uma coerência com a linguagem C++, as condições aparecem entre parênteses e as ações entre chaves. Uma ação é um conjunto de comandos separados por ponto e vírgula, isto é, um código descrito em C++.

Uma observação importante é que a seção **synchronizers** é herdada pelas subclasses. Como será visto no próximo tópico, ela pode ser implementada como uma seção protegida em C++.

Para ilustrar o uso da seção **synchonizers**, serão apresentados os exemplos do "buffer", discutidos no segundo tópico. A figura 4.2 apresenta a classe "b-buf" com os métodos put() e get(). Convém ressaltar que a especificação dos aspectos de sincronização encontram-se dentro da classe, porém de forma independente do código dos métodos. Neste exemplo foi usada a primitiva **disable**, como também poderia ter sido usada a primitiva **enable**.

```
class b-buf{
protected
      int in,out,buf[SIZE];
public
      b-buf(){in=out=0;}
      void put(int x){buf[in%SIZE]=x; in++;}
      int get {return buf[out++%SIZE];}

synchronizers
      (in >= out+SIZE) disableput(int x);
      (in == out) disableget();
};
```

Figura 4.2 - Classe "b-buf"

O exemplo da classe x-buf2 (fig. 4.3) apresenta uma questão interessante. O método empty() pode ser sempre chamado, sem nenhuma restrição. Nesse caso não é feita nenhuma referência ao método empty() na seção **synchronizers**.

```
class x-buf2: public b-buf{
public
      x-buf2(){b-buf();}
      void get2(int& a, int& b){//código para remover
                                // 2 elementos
                                   out+=2;}
      int empty() {return in==out;}

synchronizers
      (in >= out+2) enable get2(int& a, int& b);
};
```

Figura 4.3 - Classe "x-buf2"

O exemplo da classe gb-buf (fig.4.4) apresenta o problema de sensibilidade ao histórico de chamadas. Essa anomalia ocorre em função do novo atributo after-put, que foi adicionado para indicar uma característica do histórico de invocações do método put(). Nesse caso há a necessidade de testar esta variável antes do método gget() e prever esta variável no código dos métodos put() e get(). O método gget() é similar ao método get() com a restrição que ele não pode ser invocado após um put(). Assim, o código do get() é herdado pelo método gget() e é imposta a seguinte restrição:

(after-put) **disable** gget().

Os métodos put() e get() não mudam de forma essencial, porém devem levar em conta um novo atributo resultante de um problema de sincronização. Então, resolvemos usar as ações de sincronização definidas no tópico anterior. Nesse caso específico foram especificadas pós-ações, através da primitiva **after**.

```
class gb-buf: public b-buf{
      int after-put;
public
      gb-buf(){b-buf(); after-put = false;}
      int gget {b-buf::get();}
synchronizers
      (after-put) disable gget();
      {after-put=true;} afterput();
      {after-put=false;}afterget();
};
```

Figura 4.4 - Classe "gb-buf"

Na figura 4.5 é ilustrada a classe Lock e a classe lb-buf, subclasse das classes: b-buf e Lock. Os aspectos de sincronização da classe Lock apresentam a seguinte sincronização condicional:

(locked) **disable_all_except** unlock();

Esta cláusula indica que o estado "locked" restringe a execução de todos os métodos da classe Lock e de suas subclasses, com exceção do método unlock(). Na classe lb-buf não houve necessidade de se definir aspectos de sincronização, pois estes foram herdados das superclasses através da herança múltipla.

```
class Lock{
      int locked;
public
      Lock(){locked = 0;}
      void lock() {locked = 1;}
      void unlock(){locked = 0;}
synchronizers
      (!locked) disableunlock();
      (locked) disable_all_exceptunlock();
};

class lb-buf: publicb-buf, Lock{
public
      lb-buf(){b-buf(); Lock();};
};
```

Figura 4.5 - Classes "Lock" e "lb-buf"

5.SUPORTE EM C++ PARA A IMPLEMENTAÇÃO DAS PRIMITIVAS

No tópico anterior nós propomos construções que podem ser adicionadas a linguagens concorrentes baseadas em C++ para especificação de aspectos de sincronização. Porém a tarefa de estender uma linguagem não é muito fácil, ainda mais quando se tem uma linguagem complexa como é a C++.

Por isso, neste trabalho, é também apresentado um protótipo de uma biblioteca de "templates" que oferece suporte para se especificar ações de sincronização e sincronização condicional. É também proposta uma técnica de como usar esta biblioteca de forma que os efeitos da anomalia de herança sejam minimizados.

A motivação ao projetarmos esta biblioteca não foi apenas oferecer formas para tratar a sincronização, definindo pura e simplesmente uma biblioteca de rotinas (funções em C++). Porém, principalmente, tratar o problema de forma "elegante", mantendo-se o mais fiel possível ao paradigma de orientação a objetos.

Foram definidos três moldes ("templates" em C++) para modelar métodos como objetos. Estes moldes são: **method, condition** e **action.** Os moldes **condition** e **action** apresentam métodos equivalentes às primitivas definidas na seção 3. A definição destes moldes encontra-se na Figura 5.1.

```
template <class type> class method{
protected:
      type (* m)();
public:
      method(type (* met)());
};
template <class type> class condition: public method{
public:
      condition(type (* met)());
      void enable(method);
      void enable_all_except(method);
      void disable(method);
      void disable_all_except(method);
};
template <class type> class action: public method {
public:
      action(type (* met)());
      void before(method);
      void after(method);
};
```

Figura 5.1 - Definição dos moldes method, condition e action.

A metodologia para declaração de classes, usando estes moldes, apresenta os seguintes itens:

1- Ao declarar uma classe em C++, definem-se os atributos e métodos de forma usual, sem a especificação de restrições de sincronização;
2- As condições e ações usadas para especificar aspectos de sincronização devem ser descritas através de métodos, que não se encontram no problema original. Para manter um certo encapsulamento esses métodos devem ser declarados dentro da seção protegida da classe;
3- Na seção protegida da classe, o usuário declara os métodos necessários (alguns métodos do item 1 e todos do item 2) como instâncias dos moldes **method, condition** e **action**, passando como parâmetro o tipo do método;
4- Depois define-se um método especial chamado "synchronization", onde são "declaradas" as restrições de sincronização, através de chamadas aos métodos definidos nos moldes **action** e **condition**. O método **synchronization** deve ser chamado pelo método construtor da classe.

O modelo de uma classe usando esta biblioteca de moldes é ilustrado na figura 5.2.

```
class nome-classe {
      // atributos, métodos privados
            . . .
public:
      // Método construtor da classe
      nome-classe(. . .) { . .. .; synchronization();}
      // Métodos públicos especificados de forma usual
      // sem descrever aspectos de sincronização
            . . .
protected:
      // Métodos e atributos protegidos, especificados
      // de forma usual
            . . .
      // Declaração de métodos para as condições e ações
            . . .
      // Declaração de métodos como objetos dos moldes:
      // method, condition e action.

            condition <tipo-do-método> METi(meti), . . .;
            method <tipo-do-método> METj(metj), . . .;
            action <tipo-do-método> METk(metk), . . .;
            . . .

      //Especificação dos aspectos de sincronização
      void synchronization() {

            METi.primitiva-sincronização(METj);

      }
};
 onde:
 met - nome de um método qualquer da classe <nome-classe> e
 MET - nome do objeto associado ao método (normalmente o
       nome do método em letra maiúscula).
 primitiva-sincronização ∈ { enable, enable-all-except,
                             disable, disable-all-except,
                             after, before }
```

Figura 5.2 - Modelo de Especificação de Classes usando os moldes: method, condition e action.

Segue a especificação das classes b-buf, g-buf, usando a biblioteca apresentada.

```
class b-buf{
      int in,out,buf[SIZE];
public:
      b-buf(){in=out=0; synchronization();}
      void put(int x){buf[in%SIZE]=x; in++;}
      int get() {return buf[out++%SIZE];}
protected:
      int full(){return (in>=out+SIZE);}
      int empty() {return in==out;}
      condition <int> FULL(full), EMPTY(empty);
      method <void> PUT(put), <int> GET (get);
      void synchronization() {
            FULL.disable(PUT);
            EMPTY.disable(GET);
      }
};
```

Figura 5.3 - Definição da classe b-buf

```
class gb-buf: public b-buf{
      int after-put;
public:
      gb-buf(){   b-buf(); after-put = false;
                  synchronization();}
      int gget(){b-buf::get();}
protected:
      int value-after-put(){return after-put;}
      void after-put-true() {after-put=true;}
      void after-put-false() {after-put=false;}

      condition <int> AFTER-PUT(value-after-put);
      action <void> AFTER-PUT-TRUE(after-put-true),
      AFTER-PUT-FALSE(after-put-false);
      method <int> GGET(gget);
      void synchronization() {
            AFTER-PUT.disable(GGET);
            AFTER-PUT-TRUE.after(PUT);
            AFTER-PUT-FALSE.after(GET);
      }
};
```

Figura 5.4 - Definição da classe gb-buf

Não é preocupação, nessa proposta inicial, se ater em detalhes da linguagem C++. Alguns pontos não foram descritos nesta proposta, como a necessidade de passar o nome da classe junto com o método no construtor do método. Também não se leva em conta métodos com parâmetros, nem o conceito de polimorfismo, onde o mesmo método pode apresentar comportamento diferente a depender dos parâmetros. O objetivo nesse trabalho é apresentar a idéia geral de funcionamento dos moldes e como eles podem ser usados para minorar os problemas de anomalia de herança.

6.CONCLUSÃO

As propostas apresentadas nas seções 3 e 4, são baseadas em alguns trabalhos importantes na área de herança em linguagens orientadas a objetos concorrentes. Podemos citar três trabalhos principais:

- O trabalho descrito em [Mat93] que é uma referência obrigatória na área, ele apresenta de forma elaborada o problema da anomalia de herança e apresenta propostas de como solucionar o problema. No nosso trabalho aproveitamos alguns pontos do trabalho de Matsuoka, entre os mais importantes: o uso de guardas para especificação de sincronização condicional; a sugestão de separar as construções de sincronização da especificação dos métodos;
- Outro trabalho importante na área é o artigo de Frϕlund [Frϕ92]. Neste artigo ele introduz o uso de condições de restrições para execução de métodos, que funciona como uma guarda negativa. Esse tipo de condição é interessante em alguns problemas e nós resolvemos adotá-la na nossa proposta;
- Um dos trabalhos que abordam as ações de sincronização encontra-se em [Neu91]. Nesse trabalho ele utiliza a estrutura de guardas e apresenta formas para especificação de pré e pós ações. A especificação dos aspectos de sincronização é realizada juntamente com a especificação do corpo do método, podendo gerar alguns problemas no mecanismo de herança. Já na nossa proposta a especificação das ações é feita separada do código dos métodos, facilitando a herança destes aspectos de sincronização.

Uma contribuição do nosso trabalho é tentar unificar numa única proposta características interessantes de trabalhos anteriores. Além de oferecer todas estas características de uma forma fácil de ser usada, dentro do escopo de uma linguagem concorrente baseada em C++.

A principal característica da nossa proposta é englobar tanto condições de habilitação como de restrição. Esse fator gera mais flexibilidade para o usuário. Esta flexibilidade por sua vez pode gerar mais complexidade, pois deve-se prever tratamento para aspectos de sincronização conflitantes.

Outra contribuição deste trabalho encontra-se na seção 5, onde é apresentada uma biblioteca de moldes em C++, para a especificação de aspectos de sincronização. A principal característica desta biblioteca é o tratamento de métodos como objetos provendo um nível de reflexividade ao sistema.

Como trabalhos futuros temos: (i) estudar melhor a anomalia de herança, pois segundo Matsuoka não se pode afirmar que não existam outra anomalias além das listadas em [Mat93]; (ii) prover uma implementação eficiente para as primitivas propostas.

7.BIBLIOGRAFIA

[Agh86] AGHA, G. **ACTORS: A Model of Concurrent Computation in Distributed Systems**. The MIT Press, 1986.

[FrØ92] FRØLUND, S. Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. In O. Lehrmann Medsen, editors, Sixth European Conference on Object-Oriented Programming (ECOOP'92), LNCS 615, Springer-Verlag, p.185-196, 1992.

[Mat93] MATSUOKA, S.; YONEZAWA A. Analysis of Inheritance Anomaly in Concurrent Object Oriented Programming. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, **Research Directions in Concurrent Object-Oriented Programming.** The MIT Press, Cambridge, MA, p.107-150, 1993.

[Neu91] NEUSIUS, C. Synchronization Actions. In Proceedings of ECOOP'91, volume 512 of Lecture Notes in Computer Science, p.118-132. Springer-Verlag, 1991.

[Str86] STROUSTRUP, B. **The C++ Programming Language**. Addison-Wesley, Reading, MA, 1986.

[Wil96] WILSON, G.V.; LU, P. **Parallel Programming Using C++.** The MIT Press, Cambridge, MA, 1996.