Uma Abordagem ao Controlo de Concorrência em Orientação por Objectos

An Approach to Concurrency Control in Object Orientation

Carlos Baquero* Francisco Moura†
DI / INESC
Universidade do Minho
4700 Braga, Portugal
{mescbm,fsm}@di.uminho.pt

Resumo

Este artigo principia com um breve enquadramento na área de programação concorrente em orientação por objectos, sendo tidas em especial atenção questões relacionadas com a anomalia da herança. É apresentado um conjunto de parâmetros a ter em conta no desenho deste tipo de sistemas, após o qual se introduz o sistema de programação concorrente em OO, **CA/C++**. São também analizados, na óptica do uso (ou não) de concorrência interna, alguns tempos obtidos num programa **CA/C++**. É também apresentado um exemplo de expansão por herança de objectos com controlo de concorrência.

Abstract

This paper starts with a brief introduction to concurrent object oriented programming, whilst paying special attention to the inheritance anomaly issues. After presenting a set of parameters for the design of this kind of systems, we introduce the concurrent OO system, **CA/C++**. Some execution times obtained with **CA/C++** programs are analized with respect to internal concurrency issues. An example of expansion by inheritance of objects with concurrency control is also presented.

*Financiado pela bolsa JNICT BM92 / 3556 / IA
†No âmbito do projecto JNICT PMCT 163/90

1 Introdução

O paradigma da Orientação por Objectos (OO) demonstrou, na sua aplicação à engenharia de sistemas, fornecer boas abstracções para a representação de problemas reais, favorecendo o encapsulamento de dados e a partilha de código. Este paradigma apresenta-se aparentemente como um bom candidato à integração de características concorrentes.

A motivação imediata para a utilização de um modelo de programação onde sejam exprimíveis tarefas concorrentes prende-se com a aumento de expressividade assim obtido, bem como com o aproveitamento eficaz do paralelismo existente nas máquinas actuais. Com efeito, a diminuição do custo dos processadores, que no passado conduziu à proliferação de computadores e estações de trabalho pessoais, conduz actualmente à disponibilização de máquinas com mais de um processador. A relativa facilidade de adaptação de sistemas **UNIX** a arquitecturas multiprocessador de memória partilhada, bem como as capacidades multiprocessadoras de sistemas como o **Windows-NT**, são

sintomáticas do interesse em explorar o paralelismo disponibilizado nestas novas plataformas computacionais.

Estes factores propiciaram ao longo dos últimos anos o aparecimento de linguagens baseadas ou orientadas ao objecto que de algum modo permitem concorrência. Cita-se, entre outras, **Ada**, **ABCL/1** [Yonezawa 87], **ConcurrentSmalltalk** [Yokote 87], **Pool-I** [America 90], **ACT++** [Kafura 89], **Mentat** [Grimshaw 93], **Rosette** [Tomlinson 89], **Dragoon** [Reghizzi 91], **CEiffel** [Lohr 93], **Panda** [Assenmacher 93], **Eiffel//** [Caromel 93]. Estas linguagens podem ser agrupadas segundo vários critérios, tendo sido classificadas segundo diversas taxonomias [Papathomas 89, Papathomas 92, Kafura 93, Matsuoka 93, Bergmans 94]. De entre estas linguagens e sistemas é de realçar a existência de várias abordagens assentes na extensão de linguagens OO sequenciais. Este tipo de aproximação apresenta a vantagem de herdar o património existente na linguagem sequencial bem como veicular uma mais fácil adaptação dos programadores à nova linguagem. Por outro lado, surge muitas vezes a necessidade de limitar a expressividade anteriormente permitida, bem como são geralmente impostas restrições ao tipo e grau das extensões introduzidas.

A integração de concorrência no paradigma OO introduziu diversos problemas, em especial na aplicação de mecanismos de herança [Kafura 89, Tomlinson 89, America 90]. A raiz destes problemas assenta na dificuldade em adaptar código de controlo de concorrência (herdado) aos novos estados e métodos de interacção introduzidos nas classes que o herdam [Matsuoka 93]. A dificuldade em integrar os mecanismos de concorrência e sincronização na estrutura de herança das linguagem é designada por *anomalia da herança.*

Este artigo, descreve uma extensão concorrente a uma linguagem OO, sendo evitada a anomalia da herança. O modelo aqui apresentado encontra-se concretizado numa extensão à linguagem OO (**C++**), sendo designada por **CA/C++**. Neste sistema são geradas actividades concorrentes, mediante o uso de invocações assíncronas. Esta concorrência é regulada pela possibilidade de introduzir controlo de acesso em objectos utilizados concorrentemente. A nossa proposta baseia-se no uso de *anotações*[1] para a descrição do comportamento de objectos potencialmente sujeitos a acessos concorrentes. A concorrência é assim controlada sem se recorrer explicitamente a primitivas de sincronização de baixo nível.

2 Concorrência em Linguagens Orientadas por Objectos

2.1 Primeiras Abordagens

O não-determinismo presente na concorrência entre objectos conduz à necessidade de coordenar a ordem de satisfação dos diversos pedidos de execução (mensagens) recebidos pelos objectos. Uma das primeiras abordagens ao controlo de concorrência entre objectos assenta na introdução de primitivas de recepção de mensagens no seio do código responsável pela funcionalidade do objecto, tal como sucede em **Ada**[2] e noutras linguagens baseadas no modelo **CSP** [Hoare 78]. Outras abordagens seguem o modelo de Actores [Agha 86], e baseiam-se na mudança dos conjuntos de métodos disponibilizados pelo objecto num dado momento [Kafura 89, Tomlinson 89, Matsuoka 93].

Estas abordagens revelaram-se incapazes de acondicionar a herança e definição incremental de código de controlo de concorrência, forçando, por vezes, a própria reescrita do código funcional [Matsuoka 93], o que vem mutilar um dos pontos mais importantes da OO, a reutilização. Tal deve-se, quer ao uso de pontos de recepção explicita de mensagens no seio do código funcional, como no **Ada**, **CSP**, **Pool-I**, **Eiffel//**, quer

[1] O termo *anotações*, bem como o termo *concurrency annotations*, não possui relação com os termos usados na linguagem CEiffel.

[2] Linguagem Baseada em Objectos mas que não suporta herança.

ao uso de formas de recepção indirecta como seja por conjuntos de métodos, como em **Rosette** e **ACT++** [Kafura 89]. A coabitação entre código funcional e código de controlo de concorrência, impossibilita a sua redefinição e partilha separada.

Algumas linguagens limitam-se a enquadrar, na orientação por objectos, mecanismos clássicos de sincronização, como semáforos e monitores [Madsen 93]. Deixam ao cargo do programador a utilização destes objectos como pontos de sincronização, o que fica aquém das possibilidades introduzidas com o paradigma OO.

2.2 Soluções Recentes

Nos anos mais recentes, surgiram várias propostas para a resolução deste problema [Matsuoka 93, lund 92, Neusius 91, Kafura 93], sendo em todas preconizada a separação entre código de sincronização e código funcional, de modo a permitir uma redefinição separada de ambos. Numa das últimas propostas, na linguagem **ABCL/R2** [Matsuoka 93], são criados vários mecanismos alternativos por forma a ultrapassar os problemas de herança. Esta multiplicidade tem o inconveniente de complicar consideravelmente o modelo de programação disponibilizado.

Uma vez que várias destas propostas são baseadas no modelo de Actores, é mantido o uso de conjuntos de métodos, assim como é imposta uma restrição à concorrência interna ao objecto, impedindo o uso simultâneo de métodos. A abordagem subjacente ao sistema **CA/C++** evita a utilização de conjuntos de métodos e disponibiliza mecanismos para o controlo de concorrência interna (intra-objecto).

A abordagem adoptada no sistema **Demeter**, descrita em [Lopes 94], defende a utilização de concorrência interna. Existe também uma clara separação entre o código funcional e código de controlo de concorrência. Ao contrário do sistema **CA/C++**, não é suportada a partilha (por herança) nem a definição incremental do código de controlo de concorrência. Observa-se neste sistema a utilização do conceito de *exclusive regions* para a definição de exclusividades entre métodos num dado objecto, o que constitui um dos mecanismos de controlo de concorrência interna. Carece, porém, na versão apresentada, de mecanismos que protejam a execução do código de controlo de concorrência.

Importa referir que a última versão da linguagem OO **Sina** [Bergmans 94], com suporte a aplicações de tempo real, apresenta um modelo que permite a composição e redefinição de código de controlo de concorrência, permitindo o uso e controlo de concorrência interna.

3 Propriedades Desejáveis num Sistema OO Concorrente

A construção de um sistema OO concorrente, em particular, de uma extensão a um sistema OO série compilável, deve, na nossa perspectiva, enquadrar as seguintes premissas.

1 **Correcta integração no sistema de verificação de tipos da linguagem, mantendo as mesmas garantias na compilação.** Esta premissa, aparentemente óbvia, é desrespeitada em alguns sistemas onde se introduz, no **C++**, concorrência por invocações assíncronas (gerando novos fios de execução). No sistema **Presto** [Faust 90] e **Mercury** [Kontothanassis 93] a utilização de invocações assíncronas implica a perda de verificação de tipos nos argumentos e resultado dos métodos invocados. A origem desta má integração assenta no facto de estes sistemas se restringirem ao uso de bibliotecas, evitando o recurso a uma fase de pré-compilação. São já bem conhecidos os elevados riscos introduzidos com a circunvenção da verificação estática de tipos em linguagens como o **C++**.

2 **Capacidade de adaptar hierarquias de código série, idealmente, mesmo**

quando estas se encontram compiladas separadamente. Mesmo que apenas aplicável a um certo número de casos, esta possibilidade potencia o aproveitamento de código previamente desenvolvido, conduzindo a uma mais fácil migração para ambientes concorrentes. Esta capacidade adequa-se perfeitamente à necessidade de separar código funcional e código de controlo de concorrência.

3 **Forte separação entre código de controlo de concorrência e código funcional.** Esta separação é crucial se se pretende contornar a anomalia da herança, pois só assim se possibilita a partilha e redefinição separada destas duas classes de código. A ausência desta faculdade é a causa principal de anomalias da herança nas linguagens OO concorrentes. O grau de separação aplicado, constitui um ponto de divergência entre esta [Baquero 94] e outras [Lopes 94, Bergmans 94, Matsuoka 93] recentes abordagens à questão da anomalia.

4 **Herança separada de código de controlo de concorrência.** Como corolário da separação entre as duas classes de código, importa facultar a sua herança e redefinição separadas, sem as quais não fica completa a solução da anomalia.

5 **Extensão de hierarquias concorrentes compiladas, sem acesso a detalhes do código.** Esta possibilidade assume um papel especial em linguagens como o **C++**, onde são marcantes as preocupações de eficiência e de capacidade de tratamento de grandes volumes de código.

6 **Evitar limitações desnecessárias à concorrência interna.** Ao não se negar, à partida, a possibilidade de aproveitamento de concorrência intra-objecto, abre-se caminho a soluções mais eficientes. Isto poderá ser observado num exemplo apresentado adiante.

De entre as abordagens aqui referidas, as que se baseiam no modelo de actores entram naturalmente em conflito com a última premissa. Também é imediato observar que as abordagens não compiladas e as que assentam em linguagens não fortemente tipadas, não se adequam às preocupações de eficiência expressas nos pontos 1, 2 e 5.

Ainda, no que respeita aos pontos 3 e 5, importa notar que o uso de conjuntos de métodos, ou de estruturas equivalentes — tal como sucede em quase todas as abordagens — obriga ao conhecimento de detalhes, sobre a política de sincronização utilizada no código herdado, o que entra em conflito com o ponto 5.

O modelo expresso no sistema **CA/C++**, irá de encontro a estes vários pontos.

4 *Concurrency Annotations* e sua Aplicação ao C++

4.1 Apresentação

O sistema *Concurrency Annotations* (**CA**) apresenta-se como uma extensão a uma linguagem OO série, em particular o **C++**. Pelo uso desta extensão passa a ser permitida a criação de actividades concorrentes, por meio de chamadas assíncronas a métodos com devolução de **futuros** explícitos. Estes futuros constituem elementos de primeira categoria na linguagem e, como tal, podem ser usados como argumentos de mensagens. Sendo a concorrência assim introduzida ortogonal à linguagem, torna-se necessária a sua coordenação, neste caso pela introdução de uma camada de controlo de concorrência. Esta camada envolve determinados objectos permitindo o controlo das condições de invocação dos seus métodos, o que pode ser entendido como a aplicação de reflectividade no mecanismo de invocação de métodos da linguagem [Chiba 93]. A introdução de código de sincronização segue uma estrutura quase declarativa.

São disponibilizados dois mecanismos de

controlo de concorrência, sendo um mais apropriado à coordenação de acessos simultâneos ao objecto (concorrência interna), pela capacidade de ligar e desligar métodos (conceptualmente representada pela abertura e fecho de *portões*[3] associados a cada método). Existe um outro mecanismo, expresso pela associação de predicados (guardas) a cada método, este é mais adaptado à coordenação da ordem de aceitação de invocações, prendendo-se pois com a coordenação de concorrência externa. O código de controlo de concorrência é executado antes e depois do código funcional, sendo cada um dos seus elementos expansível nas anotações de classes derivadas. A aplicação das **CA** aos casos de anomalia de herança encontra-se descrita em [Baquero 94].

Apesar dos **CA** serem potencialmente aplicáveis a outras linguagens OO, o protótipo desenvolvido encontra-se concretizado num tradutor de **CA/C++** para **C++**, que permite regular classes **C++** série. As anotações são identificadas pela palavra `annotation` seguida do nome da classe a regular. Estas anotações devem definir hierarquias correspondentes às das classes que regulam. Em cada anotação pode ser indicado para cada método — entre os métodos definidos nessa classe ou nas classe herdada —, a sua guarda, pré e pós-acções, respectivamente denotadas pelos identificadores `cond:`, `pre:`e `post:`. Pode ser definido um estado privado ao código de sincronização, denotado por `state:`, bem como construtores próprios, usando `start:`. Ambos estes elementos são extensíveis em anotações mais derivadas. Pode igualmente ser utilizado o interface público do código funcional, o que propicia uma forte separação entre as duas classes de código. Encontra-se em estudo a possibilidade de permitir o acesso aos argumentos dos métodos, no seio das guardas e acções.

O tradutor gera as hierarquias de classes apropriadas mediante a análise das anotações, sendo também responsável pela

[3]A abertura e fecho são respectivamente denotados por `@MethodName+` e `@MethodName-`.

```
#define CAPACITY 5
class Store {
  private:
    int shelf[CAPACITY];
    int pprt, gptr
  public:
    Store() : pptr(0), gptr(0) {}
    void put(int e)
    {
      shelf[pptr++]=e;
      if (pptr==CAPACITY) pptr=0;
    }
    int get() {
       int e;
       e=shelf[gptr++];
       if (gptr==CAPACITY) gptr=0;
       return(e);
    }
}

annotation Store {
  state: { int shelf_load; }
  start: { @put+; @get+; shelf_load=0; }
  void put(int) {
    cond: { return(shelf_load<CAPACITY); }
    pre:  { @put-; }
    post: { @put+; shelf_load++; }
  }
  int get() {
    cond: { return(shelf_load>0); }
    pre:  { @get-; }
    post: { @get+; shelf_load--; }
  }
}
```

Figura 1: Classe `Store` com as suas anotações.

mudança dos clientes de objectos anotados, por forma a estes usarem os objectos regulados. As invocações assíncronas são igualmente traduzidas, sendo criados fios de execução suportados pelo núcleo do sistema operativo **Solaris** [SunSoft 93], o que permite paralelismo real entre actividades concorrentes.

4.2 Anotação de Uma Classe

Considere-se a programação de um repositório FIFO de tamanho fixo. Na figura 1 observa-se a sua concretização com recurso a uma fila circular assente num vector `shelf`. A colocação e remoção de elementos é feita por dois índices distintos sobre o vector.

Estas operações podem ser feitas em concorrência (interna), uma vez garantido que o ponto de remoção não ultrapasse o ponto de colocação, pois assim não chegam a actuar sobre os mesmos elementos do estado. O código funcional de `Store` limita-se a colocar e retirar elementos da fila circular, apenas actuando correctamente quando regulado pelas anotações que o completam. Nas anotações, as guardas garantem (em `get`) que não são retirados elementos quando a fila está vazia — nesta concretização do código funcional isto implica que `pptr==gptr` — e que não se excede (em `put`) a capacidade da fila — o que neste caso impede que `pptr` ultrapasse `gptr`[4].

O controlo de portões impede a execução simultânea de mais de um `put` e/ou `get`, visto a execução paralela de dois ou mais de um destes métodos potencialmente interferir nas manipulações dos índices devido à quebra de atomicidade no seu incremento. Note-se que o sistema garante atomicidade na execução de código de controlo de concorrência, assim não ocorre nenhum problema aquando da manipulação — aparentemente concorrente — de `shelf_load`.

4.3 Consequências de Restrições à Concorrência Interna

Por forma a comparar esta codificação com uma outra onde se restringe a concorrência interna, procedeu-se ao seguinte teste. Introduziu-se uma espera activa de 0.05 segundos no código funcional dos métodos `put` e `get`. Animou-se com dois fios de execução autónomos, um objecto responsável pela colocação de 100 elementos numa dada instância de `Store`, e um outro responsável pela remoção desses 100 elementos. A instância da classe regulada `Store` passa assim a ser animada por dois fios de execução distintos.

Foram obtidos os seguintes tempos, com 1 e 2 processadores activos:

[4]Dando-lhe uma volta de avanço.

	Tempo	Ganho	Eficiência
1 P.	12.1s	1.00	100%
2 P.	6.1s	1.98	99%

Os ganhos apresentados nesta tabela, justificam-se pelo facto de ser permitida a invocação concorrente dos métodos de remoção e colocação de elementos. Este é um exemplo típico das vantagens obtidas pela permissão de concorrência interna aos objectos. Abordagens mais conservadoras, como as baseadas no modelo de Actores, onde se limita a concorrência interna demonstram um comportamento menos eficiente.

Modificando o controlo de portões expresso nas pré e pós-acções da anotação da classe `Store` de modo a desligar ambos os métodos `put` e `get` durante a execução de cada um deles, possibilita-se simular uma abordagem mais restritiva no tratamento da concorrência interna. Observa-se na seguinte tabela os resultados obtidos:

	Tempo	Ganho	Eficiência
1 P.	12.2s	1.00	100%
2 P.	10.8s	1.13	57%

Fica assim patente a penalização acarretada pela desnecessária proibição de concorrência interna.

4.4 Herança

Na figura 2 apresenta-se um pequeno exemplo da interacção das CA com a herança. Na classe derivada `SubStore` são herdados os métodos de `Store` e é definido um novo método, `unput`, responsável pela remoção de elementos em ordem LIFO. O código funcional deste método utiliza o índice `pptr` também usado pelo método `put` herdado. Torna-se assim necessário impedir a animação simultânea dos métodos `put` e `unput`. Como este método remove elementos torna-se também necessário impedir um possível conflito com o método `get`. Recorrendo ao controlo de portões, basta indicar nas pré e pós-acções do método `unput`, o fecho e abertura dos portões dos três

```
class SubStore : public Store {
  public:
    int unput()
    {
      int e;
      if (--pptr==-1) pptr=CAPACITY-1;
      return(e=shelf[pptr]);
    }
}

annotation SubStore : public Store {
  start: { @unput+; }
  int unput() {
    cond: { return(shelf_load>0); }
    pre:  { @put-; @unput-; @get-; }
    post: { @put+; @unput+; @get+;
            shelf_load--; }
  }
  void put(int) {
    pre:  { @unput-; }
    post: { @unput+; }
  }
  int get() {
    pre:  { @unput-; }
    post: { @unput+; }
  }
}
```

Figura 2: Classe `SubStore` com as suas anotações.

métodos. Extendendo nas anotações de `unput` as acções associadas aos métodos `put` e `get`, fácilmente se associa à execução destes métodos, a abertura e fecho do método `unput`. Quanto à coordenação de concorrência externa, basta indicar a guarda apropriada ao método `unput`. Pode-se observar que houve um total aproveitamento do código funcional e de sincronização herdado, sendo apenas introduzidas as extensões necessária ao controlo da nova situação.

Acrescenta-se que, visto o conflito entre os métodos `unput` e `get` só ocorrer quando apenas existe um elemento na fila, a solução apresentada pode facilmente ser melhorada, fazendo depender o fecho dos portões em questão da condição `(shelf_load==1)`.

5 Conclusões

Para além dos exemplos aqui apresentados, foram testados vários outros em **CA/C++**, como seja a multiplicação de matrizes, o cálculo do conjunto fractal *Mandelbrot*, e a anomalia de herança na refinação sucessiva de uma pilha. Em todos estes casos foi observada a aplicabilidade do sistema, obtendo-se código que exprime claramente a coordenação entre objectos. No caso particular da multiplicação de matrizes, conseguiram-se ainda excelentes resultados na diminuição de tempos de execução [5].

Ao contrário de outros sistemas, em **CA/C++** não é introduzido um novo paradigma de programação, sendo apenas criada uma forma de iniciar concorrência e uma camada adicional de controlo de concorrência. Esta camada deve ser utilizada onde existirem objectos sujeitos a invocações concorrentes. A separação entre anotações e classes permite a reutilização de código série e favorece a clareza na distinção entre código funcional e código de sincronização.

Um dos aspectos mais salientes desta abordagem é o facto da introdução de concorrência não acarretar condicionalismos na herança. Isto permite a criação de hierarquias de componentes utilizáveis em concorrência.

Referências

[Agha 86] Agha, G. (1986). *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press.

[America 90] America, P. (1990). A parallel object-oriented language with inheritance and subtyping. In *ECOOP/OOPSLA'90*, pages 161–168. Philips Research Laboratories, ACM.

[Assenmacher 93] Assenmacher, H., Breitbach, T., Buhler, P., Hubsch, V., and Schwarz, R. (1993). Panda - supporting distributed programming in C++. In *ECOOP '93 Proceedings*, pages 361–383, P.O. Box 3049, W - 6750 Kaiserslautern, Germany. Department of Computer Science, University of Kaiserslautern, Springer Verlag.

[5] *speedup* de 7.5 com 8 CPUs

[Baquero 94] Baquero, C. and Moura, F. (1994). Concurrency annotations in C++. *ACM SigPlan Notices*, 29(7):61–67.

[Bergmans 94] Bergmans, L., Aksit, M., Wakita, K., and Yonezawa, A. (1994). An object-oriented model for extensible concurrent systems: The composition-filters approach. Unpublished.

[Caromel 93] Caromel, D. (1993). Toward a method of object-oriented concurrent programming. *Comunications of the ACM*, 36(9):90–102.

[Chiba 93] Chiba, S. and Masuda, T. (1993). Designing an extensible distributed language with a meta-level architecture. In *Proceedings of ECOOP'93*, pages 483–501. Springer-Verlag.

[Faust 90] Faust, J. E. and Levy, H. M. (1990). The performance of an object-oriented threads package. In *ECOOP/OOPSLA '90 Proceedings*, pages 278–288, Seattle, WA 98195 USA. Department of Computer Science and Engineering, University of Washington, ACM.

[Grimshaw 93] Grimshaw, A. S. (1993). Easy-to-use object-oriented parallel processing with mentat. *IEEE Computer*, pages 39–48.

[Hoare 78] Hoare, C. (1978). Communicating sequencial processes. *Communications of the ACM*, 21(8):666–477.

[Kafura 93] Kafura, D. G. and Lavender, R. G. (1993). Concurrent object-oriented languages and the inheritance anomaly. In *ISIPCALA'93*, pages 183–213.

[Kafura 89] Kafura, D. G. and Lee, K. H. (1989). Inheritance in actor based concurrent object-oriented languages. In *ECOOP'89 Proceedings*, pages 131–145. Cambridge University Press.

[Kontothanassis 93] Kontothanassis, L. (1993). The mercury user's manual. Technical Report TR465, University of Rochester, Computer Science Department, Rochester, New York 14627.

[Lohr 93] Lohr, K.-P. (1993). Concurrency annotations for reusable software. *Comunications of the ACM*, 36(9):81–89.

[Lopes 94] Lopes, C. and Lieberherr, K. (1994). Abstracting process-to-function relations in concurrent object-oriented applications. In Tokoro, M. and Pareschi, R., editors, *Proceedings of ECOOP'94*, pages 81–99. Springer-Verlag.

[lund 92] lund, S. F. (1992). Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *ECOOP '92 Proceedings*, pages 185–196, 1304 W. Springfield Avenue, Urbana, IL 61801, USA. Department of Computer Science, University of Illinois at Urbana-Champaign, Springer-Verlag.

[Madsen 93] Madsen, O. L. (1993). Building abstractions for concurrent object-oriented programming. Research, Computer Science Department, Aarhus University.

[Matsuoka 93] Matsuoka, S. (1993). *Language Features for Re-use and Extensibility in Concurrent Object-Oriented Programming.* PhD thesis, Department of Information Science, The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan. Thesis draft.

[Matsuoka 93] Matsuoka, S., Taura, K., and Yonezawa, A. (1993). Highly efficient and encapsulated re-use of synchronization code in concurrent object-oriented languages. In *OOSPLA '93 Proceedings, ACM SIGPLAN Notices*, volume 28, pages 109–126.

[Matsuoka 93] Matsuoka, S. and Yonezawa, A. (1993). Analysis of inheritance anomaly in object-oriented concurrent programming languages. Research Directions in Concurrent Object Oriented Programming, MIT Press.

[Neusius 91] Neusius, C. (1991). Synchronizing actions. In *ECOOP '91 Proceedings*, pages 118–132. Springer Verlag.

[Papathomas 89] Papathomas, M. (1989). Concurrency issues in object-oriented programming languages. Technical report, Dept. of Computer Science, University of Geneva.

[Papathomas 92] Papathomas, M. (1992). *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming.* PhD thesis, Dept. of Computer Science, University of Geneva.

[Reghizzi 91] Reghizzi, S. C., de Paratesi, G. G., and Genolini, S. (1991). Definition of reusable concurrent software components. In *ECOOP '91 Proceedings*, pages 148–166. Springer Verlag.

[SunSoft 93] SunSoft (1993). *SunOS 5.2, Guide to Multi-Thread Programming.* SunSoft, A Sun Microsystems, Inc. Business.

[Tomlinson 89] Tomlinson, C. and Singh, V. (1989). Inheritance and synchronization with enabled-sets. In *OOPSLA '89 Proceedings*, pages 103–112, 3500 West Balcones Center Drive, Austin, Texa 78759. MCC, ACM.

[Yokote 87] Yokote, Y. and Tokoro, M. (1987). Concurrent programming in smalltalk. *Object Oriented Concurrent Programming.*

[Yonezawa 87] Yonezawa, A. (1987). Modeling and programming in an object-oriented concurrent language abcl/1. *Object Oriented Concurrent Programming*, pages 55–89.