Instituto de Computação
Universidade Estadual de Campinas

Estados Globais Consistentes em Sistemas Distribuídos

Islene Calciolari Garcia[1]

Agosto de 1998

Banca Examinadora:

- Prof. Dr. Luiz Eduardo Buzato (Orientador)
- Profª Drª Dilma Menezes da Silva
  IME—USP
- Prof. Dr. Cid Carvalho de Souza
  IC—UNICAMP
- Prof. Dr. Ricardo de Oliveira Anido (Suplente)
  IC—UNICAMP

[1]Apoio financeiro da FAPESP, processo número 95/1983-8 para Islene Calciolari Garcia e 96/1532-9 para o Laboratório de Sistemas Distribuídos do Instituto de Computação–UNICAMP.

Estados Globais Consistentes em Sistemas Distribuídos

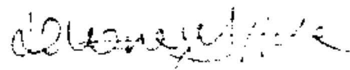
Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Islene Calciolari Garcia e aprovada pela Banca Examinadora.

Campinas, 27 de julho de 1998.

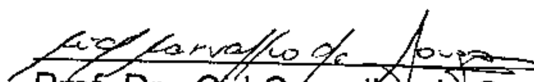
Prof. Dr. Luiz Eduardo Buzato (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Tese de Mestrado defendida e aprovada em 27 de julho de 1998 pela Banca Examinadora composta pelos Professores Doutores



Profa. Dra. Dilma Menezes da Silva



Prof. Dr. Cid Carvalho de Souza

Prof. Dr. Luiz Eduardo Buzato

© Islene Calciolari Garcia, 1998.
Todos os direitos reservados.

Para Tainã, Maíra, Matheus e Ana Elisa,
pela alegria que trouxeram.

Agradecimentos

Ao Prof. Luiz Eduardo Buzato, pela orientação, apoio, amizade e por ter confiado a mim o tema a que várias vezes se referiu como sua "menina dos olhos".

Aos vários professores do Instituto de Computação que colaboraram para a minha formação desde a graduação. Um agradecimento especial para o Prof. Pedro Jussieu de Rezende, pelos ensinamentos que recebi durante minha iniciação científica.

Aos funcionários do Instituto de Computação, por colaborarem para um ambiente de trabalho agradável.

À FAPESP, pelo apoio financeiro.

À minha mãe, pelo carinho e dedicação constantes.

A meu pai (*in memoriam*), por ter me ensinado a ser perseverante.

À minha família e meus amigos, por afastarem de mim o fantasma da solidão.

Ao meu noivo, Alexandre, por ser meu companheiro no trabalho e na vida.

Resumo

Estados globais consistentes são fundamentais para a solução de uma série de problemas em sistemas distribuídos, em especial para a monitorização e reconfiguração de aplicações distribuídas. A construção assíncrona de estados globais consistentes permite maior eficiência e flexibilidade a sistemas de monitorização.

Consideramos aplicações distribuídas construídas sobre o modelo de processos e mensagens (MPM) e sobre o modelo de objetos e ações atômicas (MOA). Enquanto para o primeiro modelo vários algoritmos foram desenvolvidos para a obtenção de estados globais consistentes, para o segundo a oferta é extremamente reduzida. Estabelecemos correlações entre esse dois modelos e conseguimos o mapeamento de algoritmos para a obtenção de estados globais consistentes do MPM para o MOA.

Enquanto grande parte da literatura para a obtenção de estados globais consistentes está voltada para a recuperação de falhas com retrocesso de estado, temos como objetivo a construção de uma *visão progressiva* da aplicação (consideramos que o monitor deve analisar estados globais consistentes mais recentes que os anteriores). Esta abordagem propiciou a criação de algoritmos originais para a construção de estados globais consistentes a partir de estados de interesse (*checkpoints*) selecionados pelos componentes da aplicação.

Abstract

Consistent global states are fundamental to solve a large class of problems in distributed systems, especially for monitoring and reconfiguration of distributed applications. The asynchronous construction of consistent global states allows greater efficiency and flexibility to monitoring systems.

We consider distributed applications built upon the processes and messages model (MPM) and upon the objects and atomic actions model (MOA). Many algorithms have been designed for the construction of consistent global states in the MPM, but for the MOA the number of algorithms is extremely reduced. We have established correlations between these two models and, using them, we have mapped algorithms to construct consistent global states from the MPM to the MOA.

A great part of the literature for the construction of consistent global states is devoted to the backward recovery of failures. Our goal is the construction of a *progressive view* of the application (we consider that the monitor must analyze consistent global states that are as up-to-date as possible). This approach allowed the creation of new algorithms for the construction of consistent global states using checkpoints; states selected by the application components.

Conteúdo

Agradecimentos vi

Resumo vii

Abstract viii

1 Introdução 1

2 Modelos Computacionais 5
2.1 Modelo de Processos e Mensagens (MPM) 5
2.2 Modelo de Objetos e Ações (MOA) 6
2.2.1 Orientação a Objetos 7
2.2.2 Ações Atômicas 7
2.3 Modelo de Processos e Conversações (MPC) 13
2.4 Dualidade entre MOA e MPC 14
2.5 Sumário 15

3 Estados Globais Consistentes 16
3.1 Precedência entre Eventos no MPM 16
3.2 Precedência entre Ações Atômicas no MOA 19
3.3 Propagação de Relógios 22
3.3.1 Propagação de Relógios no MPM 24
3.3.2 Propagação de Relógios no MOA 25
3.4 Relógios Lógicos 28
3.5 Vetores de Relógios 33
3.5.1 Vetores de Relógios no MPM 33
3.5.2 Vetores de Relógios no MOA 34
3.5.3 Implementação de Vetores de Relógios 35
3.5.4 Propriedades de Vetores de Relógios no MPM 36
3.5.5 Propriedades de Vetores de Relógios no MOA 37

3.6 Visão Progressiva 39
3.7 Mapeamento entre MPM e MOA 43
3.8 Sumário 44

4 *Checkpoints* Globais Consistentes **46**
4.1 Estados e *Checkpoints* 46
4.2 Caminhos e Ciclos em Zigzag 49
4.3 Utilidade de um *Checkpoint* 52
4.4 Protocolos Quase-Síncronos 53
4.5 Classificação de Protocolos Quase-Síncronos 58
4.5.1 Protocolos com Ausência de Relações $\mathcal{Z}$ Não-Precedentes (ZPF) . . 59
4.5.2 Protocolos com Ausência de Ciclos-Z (ZCF) 60
4.5.3 Protocolos que Restrigem a Presença de Ciclos-Z (PZCF) 61
4.6 Exemplos de Protocolos Quase-Síncronos 62
4.6.1 Relógios Lógicos e Vetores de Relógios para *Checkpoints* 62
4.6.2 Vetores de Relógios Fixos por Intervalo 62
4.6.3 Relógios Lógicos Fixos por Invertavalo 65
4.6.4 Quebra de Ciclos-Z Quase-Precedentes 67
4.7 Visão Progressiva Utilizando *Checkpoints* 71
4.8 Sumário 78

5 Conclusão **80**
5.1 Mapeamento entre MPM e MOA 81
5.2 Visão Progressiva da Aplicação 81
5.3 Trabalhos Futuros 82

Bibliografia **84**

Lista de Tabelas

2.1 Dualidade entre MOA e MPC 14

3.1 Correlações entre MPM e MOA 43

Lista de Figuras

1.1 Arquitetura de *software* para monitorização 2

2.1 Diagrama espaço–tempo para o MPM 6
2.2 Exemplos de estruturas de ações 8
2.3 Ações atômicas encadeadas 8
2.4 Protocolo *two phase commit* 8
2.5 Gerente de ações 9
2.6 Representação gráfica para ações atômicas 10
2.7 Diagrama espaço–tempo para o MOA 10
2.8 Diagrama espaço–tempo para o MPC 13

3.1 Relações de precedência no MPM 17
3.2 Reticulado referente à Figura 3.1 19
3.3 Relações de precedência no MOA 20
3.4 Diagrama objeto–tempo (Cenário Figura 3.3) 21
3.5 Reticulado referente ao cenário da Figura 3.3 23
3.6 Propagação de relógios no MPM 24
3.7 Fluxo de informação de precedência no MOA 26
3.8 Cenário MPM com relógios lógicos 30
3.9 Cenário MOA com relógios lógicos 31
3.10 Relógios lógicos e detecção de *gaps* no MOA 32
3.11 Cenário MPM com vetores de relógios 35
3.12 Cenário MOA com vetores de relógios 35
3.13 Visão progressiva utilizando relógios lógicos 41
3.14 Visão progressiva utilizando vetores de relógios 43
3.15 Precedência estrutural no MOA 44

4.1 Monitorização utilizando *checkpoints* 47
4.2 *Checkpoints* e intervalos de *checkpoints* no MPM 48
4.3 *Checkpoints* e intervalos de *checkpoints* no MOA 48
4.4 Ciclos–Z 51

4.5 Dificuldade de se projetar protocolos ótimos 60
4.6 Classes de protocolos quase–síncronos 61
4.7 Cenário MPM com protocolo VC_FDI 64
4.8 Cenário MOA com protocolo VC_FDI 64
4.9 Cenário MPM com protocolo LC_FDI 65
4.10 Cenário MOA com protocolo LC_FDI 66
4.11 Ciclos–Z quase–precedentes 68
4.12 Relações de precedência simples e não simples 68
4.13 Cenário MPM com protocolo XN_BHMR 69
4.14 Cenário MOA com protocolo XN_BHMR 69
4.15 Progressão no MPM com protocolo ZCF 75
4.16 Progressão no MPM com protocolo PZCF 77
4.17 Detecção de *checkpoints* inúteis 77

Lista de Classes e Interfaces

2.1 Application.java 6
2.2 Process.java 7
2.3 AtomicObject.java 10
2.4 ActionManager.java 11
2.5 AtomicAction.java 12
2.6 userAtomicClass.java 13
3.1 Clock.java 23
3.2 ClockProcess.java 25
3.3 ClockActionManager.java 28
3.4 ClockAtomicAction.java 29
3.5 userClockClass.java 30
3.6 LogicalClock.java 31
3.7 VectorClock.java 36
3.8 LC_Photographer.java 41
3.9 VC_Photographer.java 42
4.1 QS_Clock.java 54
4.2 QS_Process.java 55
4.3 QS_ActionManager.java 56
4.4 QS_AtomicAction.java 57
4.5 userQS_ClockClass.java 59
4.6 QS_LogicalClock.java 63
4.7 QS_VectorClock.java 63
4.8 VC_FDI.java 64
4.9 LC_FDI.java 66
4.10 XN_BHMR.java 70
4.11 VC_Ckpt.java 73
4.12 ZCF_Photographer.java 74
4.13 PZCF_Photographer.java 76

Lista de Símbolos

Modelo	**Símbolo**	Significado	Páginas
MPM	σ	Estado de um processo ou objeto	6, 9
e	σ_a	Estado do processo ou objeto a*	6, 9
MOA	σ_a^α	α-ésimo† estado do processo ou objeto a	6, 9
	ι_i	Sobrescrito relativo ao processo ou objeto i	17
	n	Número de processos ou objetos na aplicação	5, 6
	C	Corte de eventos ou ações atômicas	17, 21
	Σ	Estado global	18, 21
	$\hat{\sigma}$	*Checkpoint* (estado de interesse)	46
	Δ	Intervalo de *checkpoints**†	46
	$\hat{\Sigma}$	*Checkpoint* global	52
	τ	Relógio (informação de precedência)	22
	LC	Relógio lógico*	28
	VC	Vetor de relógios*	33
	θ	Histórico de eventos, ações atômicas ou estados precedentes	33, 34
	$\prec$	Precedência entre estados ou ações atômicas	17, 20
	$\parallel$	Concorrência entre estados, eventos ou ações atômicas	17, 20
	$\rightsquigarrow$	Relação $\mathcal{Z}$ entre *checkpoints*	50
MPM	p	Processo*	5
	e	Evento*†	6
	m	Mensagem	6
	$\rightarrow$	Precedência entre eventos	16
	h	Histórico de eventos de um processo*†	6
	H	Histórico global da aplicação $H = \{h_0 \cup \ldots \cup h_{n-1}\}$	6
MOA	o	Objeto*	6
	a	Ação atômica	9
	$\prec_s$	Precedência estrutural entre ações atômicas	19
	m	Gerente de ações de um objeto*	26
	$\leftrightarrow$	Simultaneidade entre estados de objetos	21

* Subscrito em letras romanas refere-se ao identificador único do processo ou objeto.

† Sobrescrito em letras gregas refere-se à posição relativa à execução do processo ou objeto.

Capítulo 1

Introdução

A avaliação de predicados sobre estados globais consistentes [7] pode ser utilizada na construção de algoritmos distribuídos para detecção de *deadlocks*, reconstrução de ficha perdida, coleta de lixo, recuperação de falhas com retrocesso de estado, além de monitorização e reconfiguração em geral [8].

Um estado global de uma aplicação distribuída é formado pela união dos estados locais dos componentes dessa aplicação [6]. Informalmente, um estado global é *consistente* se corresponde a um estado global que poderia ter sido obtido instantaneamente por um observador onisciente externo.

Aplicações distribuídas são freqüentemente construídas utilizando-se o modelo de processos e mensagens (MPM). Existem vários algoritmos na literatura para a construção de estados globais consistentes nesse modelo [1, 6, 9, 17, 26, 28], que podem ser classificados segundo três abordagens [17]: (i) síncrona, (ii) assíncrona e (iii) quase–síncrona.

A primeira abordagem apresenta um custo de comunicação extra e possível suspensão das atividades dos componentes durante a sincronização [6]. A abordagem assíncrona permite uma maior autonomia aos componentes, mas há o risco de não se construir um estado global consistente [18].

A abordagem quase–síncrona está baseada em um protocolo obedecido pelos processos para a seleção de estados, de acordo com informações de controle propagadas através das mensagens da aplicação. Prioritariamente, os processos selecionam estados livremente, mas eventualmente podem ser forçados a selecionar estados adicionais a partir de um predicado avaliado sobre a informação de controle. Esta abordagem apresenta um compromisso entre autonomia e garantias quanto à obtenção de um estado global consistente.

Aplicações distribuídas confiávies [14, pp. 1–9] podem ser construídas utilizando-se o modelo de objetos e ações atômicas (MOA) [27]. Estados globais consistentes podem ser obtidos de maneira síncrona e trivial a partir de uma ação at ômica que adquira permissão para leitura sobre todos os objetos componentes da aplicação. Um mecanismo assíncrono simples

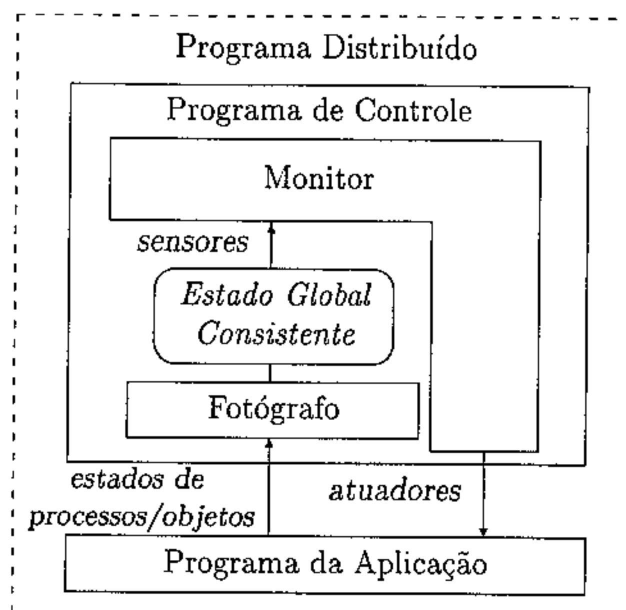


Figura 1.1: Arquitetura de *software* para monitorização

e custoso requer a duplicação da aplicação: uma cópia continua sua execução normalmente enquanto a outra finaliza as ações atômicas abertas e coleta um estado global [10].

Outro modelo sobre o qual é possível a construção de aplicações distribuídas confiáveis é o modelo de processos e conversações (MPC), dual ao MOA [22]. Utilizamos essa dualidade como ponto de partida para o estabelecimento de correlações entre o MPM e o MOA [11]. Através desse estudo, mapeamos protocolos quase–síncronos para construção de estados globais consistentes do MPM para o MOA. Além disso, conseguimos evidências para a verificação da possibilidade de mapeamento de um algoritmo do MPM para o MOA.

Monitorização de Aplicações Distribuídas

Estamos particularmente interessados na obtenção de estados globais consistentes com a finalidade de monitorização e reconfiguração de aplicações distribuídas construídas sobre o MPM ou sobre o MOA. Respeitando uma política de divisão de tarefas, separamos o *programa da aplicação* do *programa de controle* (Figura 1.1). Subdividimos o programa de controle em dois módulos: o *fotógrafo*, responsável pela obtenção dos estados globais consistentes e o *monitor*, responsável pela avaliação de predicados globais e atuação sobre a aplicação.

O fotógrafo constrói estados globais de maneira assíncrona a partir de estados de processos ou objetos fornecidos pela aplicação através de canais FIFO.[1] O monitor avalia os predicados através de *sensores* e reage sobre a aplicação através de *atuadores* (sensores e atuadores são métodos presentes nas interfaces dos processos ou objetos). Consideramos que o monitor deve ter uma *visão progressiva* da aplicação, no sentido de que o fotógrafo deve sempre lhe apresentar um estado global consistente mais recente que o anterior.

A monitorização de aplicações distribuídas levando-se em consideração todos os estados da aplicação é extremamente custosa. Para cada processo ou objeto podemos selecionar estados de interesse, denominados *checkpoints* e construir *checkpoints* globais a partir de *checkpoints* locais. Caso um *checkpoint* participe de pelo menos um *checkpoint* global consistente ele é denominado *útil*; caso contrário, é denominado *inútil*. Protocolos quase–síncronos podem ser utilizados para limitar ou eliminar a ocorrência de *checkpoints* inúteis.

Dentro do nosso conhecimento, obtivemos os primeiros algoritmos para a construção progressiva de *checkpoints* globais consistentes. Exploramos essa abordagem e encontramos uma equivalência entre a possibilidade da construção de uma visão progressiva da aplicação e o fato de um protocolo quase–síncrono permitir ou não *checkpoints* inúteis. Notamos que vários protocolos na literatura provam que não admitem *checkpoints* inúteis através da apresentação de um *checkpoint* global consistente para cada *checkpoint* local [1, 3]. Nosso resultado permite provas mais flexíveis e pode vir a ser interessante no desenvolvimento de novos protocolos.

Estrutura da Dissertação

Nesta Seção, delineamos a estrutura da dissertação e apresentamos algumas decisões acerca da apresentação do texto.

Optamos por manter alguns termos e siglas em inglês. Priorizamos a uniformidade com a literatura da área em detrimento do uso exclusivo da língua portuguesa. As classes e algoritmos são apresentados utilizando a linguagem de programação Java [12, 24],[2] com o texto dos programas em inglês. Escolhemos Java por ser uma linguagem com bom poder de expressão e porque o uso de herança contribui para uma exposição clara e incremental dos algoritmos.

O Capítulo 2 caracteriza os dois modelos computacionais distribuídos com os quais trabalhamos: modelo de processos e mensagens (MPM) e modelo de objetos e ações (MOA). Na medida do possível, estabelecemos notações que fossem adequadas aos dois modelos. Entretanto, como o MPM e o MOA não são duais, necessitamos de algumas convenções que só se aplicam a um dos modelos. O leitor deve ficar atento a estes detalhes.

---

[1] FIFO é a sigla em inglês para *First In First Out* e indica estruturas de dados nas quais o primeiro elemento a ser inserido é o primeiro a ser retirado.

[2] Java is a trademark of Sun Microsystems, Inc.

O Capítulo 3 analisa as relações de precedência e consistência no MPM e no MOA. Propõe algoritmos para a manutenção e propagação de informação de precedência (relógios) e apresenta algoritmos para a construção progressiva de estados globais consistentes. Expõe correlações entre o MPM e o MOA, discutindo limitações para o mapeamento de algoritmos do MPM para o MOA.

O Capítulo 4 considera a seleção de estados de interesse (*checkpoints*) para envio ao fotógrafo e apresenta as condições necessárias e suficientes para a utilidade de um *checkpoint*. Propõe algoritmos genéricos para a manutenção e propagação de informação de controle para a implementação de protocolos quase-síncronos no MPM e MOA. Apresenta uma classificação para protocolos quase-síncronos e exemplos de protocolos que a ilustram. Propõe algoritmos para a construção de uma visão progressiva da aplicação baseada em *checkpoints*.

Finalmente, o Capítulo 5 encerra a dissertação, resumindo as contribuições e apresentando trabalhos futuros.

Capítulo 2

Modelos Computacionais

Iniciamos este Capítulo descrevendo o modelo de processos e mensagens (MPM), que é um dos modelos mais utilizados para estruturação de sistemas distribuídos assíncronos. Em seguida, descrevemos dois modelos adequados para a construção de aplicações distribuídas confiáveis: o modelo de objetos e ações (MOA) e o modelo de processos e conversações (MPC). Apresentamos a dualidade entre o MOA e o MPC [22].

2.1 Modelo de Processos e Mensagens (MPM)

Uma aplicação distribuída no modelo de processos e mensagens (MPM) é composta por um conjunto de n processos $(p_0, \ldots, p_{n-1})$ que executam de maneira estritamente seqüencial e independente. Não existem mecanismos para compartilhamento de memória, acesso a um relógio global, sincronização de relógios locais ou conhecimento a respeito das diferenças de velocidade entre os processadores. A comunicação é feita através de canais unidirecionais entre pares de processos. Consideramos que a rede de comunicação é fortemente conexa (nenhum processo é isolado), mas não necessariamente completa (a comunicação entre um par de processos pode se dar via processos intermediários). Há garantia de entrega de mensagens, mas estas podem sofrer atrasos arbitrários e inclusive chegar aos seus destinos fora de ordem.

Todo processo possui um conjunto de variáveis locais, cujo valor representa o estado deste processo em determinado instante da computação. Subdividimos a execução de um processo em eventos internos e de comunicação. Eventos internos afetam apenas as variáveis locais do processo, enquanto eventos de comunicação ocorrem através do envio e recepção de mensagens.

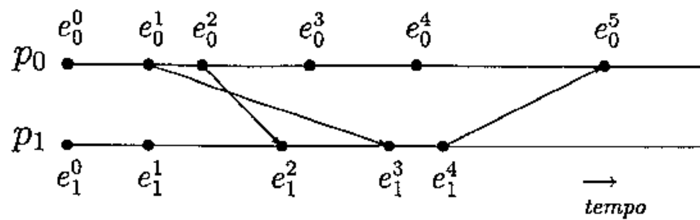


Figura 2.1: Diagrama espaço–tempo para o MPM

Classe 2.1 Application.java

```
public class Application {
    static int N; // Number of components in the application
}
```

Notação auxiliar

Relacionamos eventos a processos utilizando e_i^ι para indicar o ι-ésimo evento executado pelo processo p_i. O estado do processo p_i imediatamente após a execução do evento e_i^ι é denotado por σ_i^ι. Consideramos a existência de um evento de iniciação e_i^0 que coloca o processo p_i em seu estado inicial σ_i^0. Chamamos de histórico local de p_i o conjunto de eventos $h_i = \{e_i^0, e_i^1, \ldots\}$ e prefixos desses históricos são representados por $h_i^\iota = \{e_i^0, e_i^1, \ldots, e_i^\iota\}$. O histórico global H da aplicação é formado pela união dos históricos locais de todos os processos $H = \{h_0 \cup h_1 \cup \ldots \cup h_n\}$. Dada uma mensagem m, denominamos *envio*(m) e *recepção*(m) os seus respectivos eventos de envio e recepção.

Uma maneira usual de se representar graficamente uma computação no MPM é através de um diagrama de espaço–tempo (Figura 2.1). As linhas horizontais representam os eventos executados pelos processos, com o tempo progredindo da esquerda para a direita. Uma aresta ligando dois eventos representa uma mensagem sendo enviada por um processo (base da aresta) e recebida por outro (ponta da aresta).

A classe `Application` (Classe 2.1) define a constante `N` que indica o número total de processos ou objetos na aplicação. A classe `Process` (Classe 2.2) apresenta métodos para envio e recebimento de mensagens, bem como para a execução de eventos internos. Todo processo possui como identificador único um inteiro `pid` no intervalo $0 \leq \texttt{pid} < N$.

2.2 Modelo de Objetos e Ações (MOA)

Uma aplicação distribuída no modelo de objetos e ações (MOA) é composta por um conjunto de n objetos $(o_0, \ldots, o_{n-1})$ que interagem através de invocação de métodos atômicos

Classe 2.2 Process.java

```
public class Process {

    public class Message { /*...*/}

    public int pid;

    public Process(int pid) { this.pid = pid; }

    public void sendMessage(Message m) { /*...*/}

    public void receiveMessage(Message m) { /*...*/}

    public void internalEvent() { /*...*/}
}
```

(implementados através de ações atômicas). Este modelo é considerado adequado para implementação de aplicações distribuídas confiáveis [27].

2.2.1 Orientação a Objetos

O paradigma de orientação a objetos permite a decomposição do sistema em componentes (objetos) com facilidades para modularização, abstração de tipos, encapsulamento, polimorfismo e relacionamentos de associação, agregação e generalização/especialização [5, 20].

Todo objeto é instância de uma classe, que define seus atributos, sua interface e seu comportamento. O estado interno de um objeto reflete o conteúdo de seus atributos e seus relacionamentos com outros objetos da aplicação.

2.2.2 Ações Atômicas

Constituem um mecanismo de tolerância a falhas através de recuperação por retrocesso de estado [14, pp. 151–184] e caracterizam-se por possuir as seguintes propriedades [27]:

- **Equivalência serial**: Todo acesso a estados de objetos feito no decorrer da execução de uma aplicação ocorre sem interferências, produzindo efeitos equivalentes a uma execução seqüencial;

- **Atomicidade**: Uma ação atômica é finalizada, produzindo os efeitos pretendidos ou é abortada, tendo seus efeitos anulados;

- **Permanência de efeito**: Toda mudança de estado produzida por uma ação atômica finalizada é armazenada em memória estável.

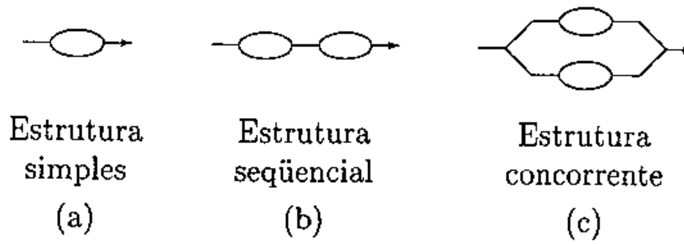


Figura 2.2: Exemplos de estruturas de ações

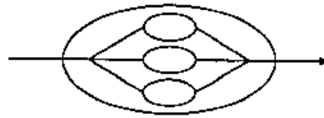


Figura 2.3: Ações atômicas encadeadas

Ações atômicas são organizadas segundo *estruturas de ações* [23] determinadas pela semântica da aplicação. A estrutura de ações básica é formada por uma única ação atômica (Figura 2.2 (a)). Estruturas de ações mais complexas podem ser construídas utilizando-se combinações seqüenciais (Figura 2.2 (b)) e concorrentes (Figura 2.2 (c)) de estruturas de ações. Podemos considerar que uma estrutura de ações é um grafo orientado e acíclico no qual os nós representam ações atômicas. Um par de nós, a e a', é unido por uma aresta se a semântica da aplicação impõe que a deve ocorrer antes de a'.

Uma das possíveis extensões de estruturas de ações atômicas permite que uma ação atômica contenha uma estrutura de ações (Figura 2.3). Neste caso, obtem-se ações atômicas encadeadas em relação a uma ação atômica de nível mais alto, que oferecem maior flexibilida-

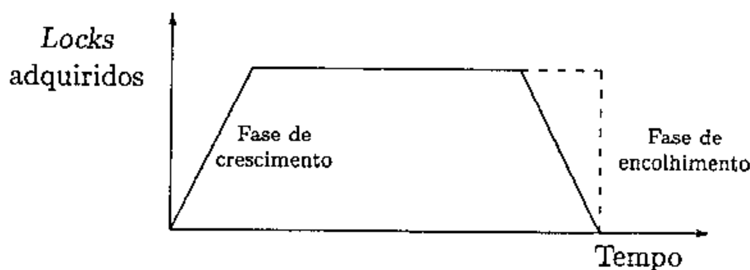


Figura 2.4: Protocolo *two phase commit*

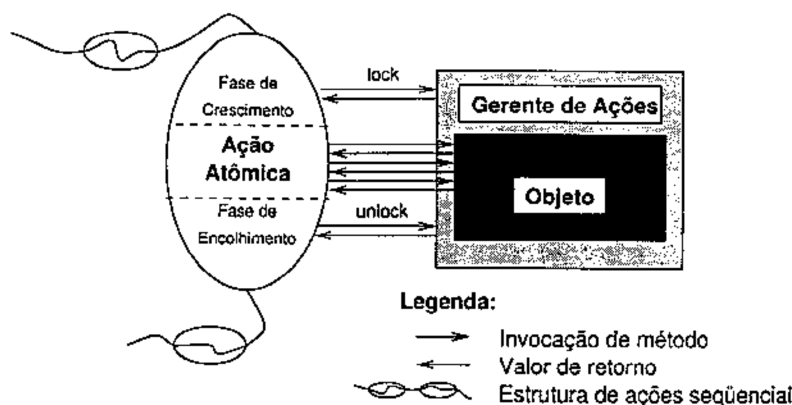


Figura 2.5: Gerente de ações

de para a recuperação de falhas e permitem concorrência dentro de uma ação atômica [22]. Como no contexto desta dissertação estamos interessados nos estados globais formados a partir de estados de objetos presentes na fronteira das ações atômicas de mais alto nível é irrelevante o fato delas possuírem ou não ações encadeadas.

Consideramos que as ações atômicas respeitam o protocolo *two phase commit* [4, pp. 49–53]. Na primeira fase, denominada *fase de crescimento*, todos os *locks* (permissões para acesso) sobre estados de objetos necessários à execução da ação são adquiridos. Na segunda fase, denominada *fase de encolhimento* os *locks* são liberados, de forma *instantânea* para observadores externos (Figura 2.4) [22].

A ausência de interferência entre ações atômicas que fazem acesso a um mesmo objeto é garantida por um gerente de ações que encapsula o objeto e distribui *locks* de leitura e escrita (Figura 2.5). Várias ações atômicas podem adquirir *locks* de leitura sobre um mesmo objeto simultaneamente, enquanto apenas uma pode adquirir *lock* de escrita em determinado momento. Para evitar ambigüidades na descrição de algoritmos, dizemos que uma ação atômica adquiriu um *lock* sobre um estado e não sobre um objeto.

Notação auxiliar

Denominamos σ_i^ι como sendo o estado do objeto o_i imediatamente após a ι-ésima ação atômica que adquiriu *lock* de escrita sobre ele e σ_i^0 como sendo o seu estado inicial. Definimos uma função *Act* que pode ser aplicada a um estado de objeto e retorna a ação em que este foi criado. Na Figura 2.6, por exemplo, $Act(\sigma_0^1) = \mathsf{a}$.

Durante sua execução, uma ação atômica pode obter *locks* de leitura ou escrita sobre vários estados de objetos da aplicação. Definimos as seguintes funções para referenciar subconjuntos desses estados:

- $LockR(\mathsf{a})$—conjunto de estados de objetos para os quais a obteve *lock* de leitura;

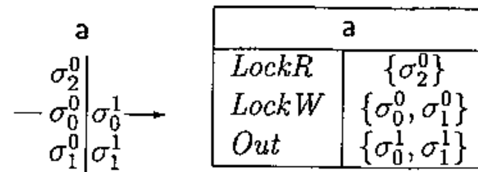


Figura 2.6: Representação gráfica para ações atômicas

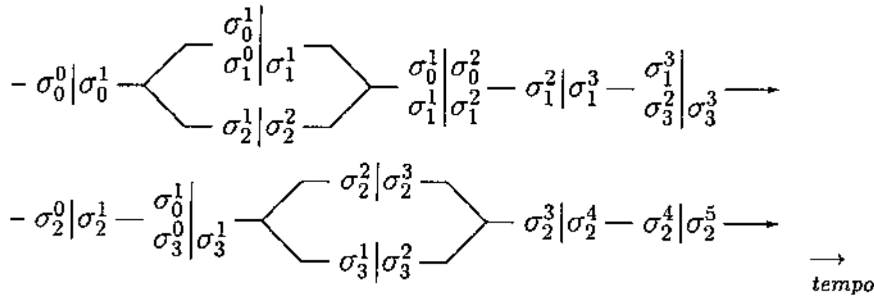


Figura 2.7: Diagrama espaço–tempo para o MOA

- *LockW*(a)—conjunto de estados de objetos para os quais a obteve *lock* de escrita;
- *LockRW*(a)—*LockR*(a) ∪ *LockW*(a);
- *Out*(a)—conjunto de estados de objetos promovidos por a.

Ações atômicas são representadas graficamente por segmentos de reta verticais. À esquerda desse segmento colocamos os estados lidos pela ação, e à direita os estados promovidos (Figura 2.6). Podemos desenhar um diagrama espaço–tempo semelhante ao do MPM, sendo que ações atômicas estão conectadas de acordo com as estruturas de ações impostas pela aplicação (Figura 2.7).

Classe **2.3** AtomicObject.java

```
public class AtomicObject {

    public int oid;
    public ActionManager actionManager;

    public AtomicObject(int oid) { this.oid = oid; }
}
```

Classe 2.4 ActionManager.java

```
public class ActionManager {

    protected AtomicObject obj;

    protected void saveState() {}  // Save object state on stable memory
    protected void restoreState() {}  // Restore object state from stable memory

    public ActionManager(AtomicObject obj) { this.obj = obj; saveState(); }

    public synchronized void readLock() { /*...*/}
    public synchronized void readUnlock() { /*...*/}
    public synchronized void abortReadLock() { /*...*/ }

    public synchronized void writeLock() { /*...*/}
    public synchronized void writeUnlock() { saveState(); /*...*/ }
    public synchronized void abortWriteLock() { restoreState(); /*...*/ }
}
```

A classe `AtomicObject` (Classe 2.3) define uma classe base para *objetos atômicos*—objetos que são manipulados exclusivamente através de ações atômicas. Todo objeto possui como identificador único um inteiro `oid` no intervalo $0 \leq \texttt{oid} < N$ (N definida na Classe 2.1).

A classe `ActionManager` (Classe 2.4) descreve um gerente de ações. Apresenta métodos para a obtenção e liberação de *locks* e para a gravação e recuperação do estado do objeto que gerencia em memória estável. Como várias ações atômicas podem fazer acesso a um mesmo gerente de ações simultaneamente, seus métodos exigem o modificador `synchronized`, que garante controle de concorrência. A associação entre um gerente de *locks* e um objeto é feita por alguma entidade externa (configurador).

A classe `AtomicAction` (Classe 2.5) define uma ação atômica e possui métodos para início (`begin`), término (`commit`) e aborto (`abort`) de execução. Toda ação atômica armazena referências para os gerentes de ações dos objetos para os quais foram adquiridos *locks* de leitura ou escrita. No término ou aborto da execução, todos os *locks* são liberados. Cabe notar que o algoritmo apresentado para a liberação de *locks* apenas ilustra o fato de todos os gerentes de *lock* serem notificados; uma implementação real do protocolo término de uma ação atômica pode ser bem mais complexa [4, pp. 217–260].

A Classe 2.6 traz um exemplo de código utilizando as classes base apresentadas para o MOA. O método `userAtomicMethod` é um método simples, no qual uma ação atômica é iniciada, alguns *locks* são obtidos e a ação é terminada.

Classe 2.5 AtomicAction.java

```
public class AtomicAction {

    public ActionManager[ ] rLocked = new ActionManager[Application.N];
    public ActionManager[ ] wLocked = new ActionManager[Application.N];
    int nRLocked=0, nWLocked=0;

    public void readLock(AtomicObject atomicObject) {
        atomicObject.actionManager.readLock();
        rLocked[nRLocked++] = atomicObject.actionManager;
    }

    public void writeLock(AtomicObject atomicObject) {
        atomicObject.actionManager.writeLock();
        wLocked[nWLocked++] = atomicObject.actionManager;
    }

    public void begin() { /*...*/}

    public void commit() {
        for (int i=0; i < nRLocked; i++) rLocked[i].readUnlock();
        for (int i=0; i < nWLocked; i++) wLocked[i].writeUnlock();
        removeActionManagers();
        /*...*/
    }

    public void abort() {
        for (int i=0; i < nRLocked; i++) rLocked[i].abortReadLock();
        for (int i=0; i < nWLocked; i++) wLocked[i].abortWriteLock();
        removeActionManagers();
        /*...*/
    }

    protected void removeActionManagers() {
        for (int i=0; i < nRLocked; i++) rLocked[i] = null; nRLocked=0;
        for (int i=0; i < nWLocked; i++) wLocked[i] = null; nWLocked=0;
    }
}
```

Classe 2.6 userAtomicClass.java

```
public class userAtomicClass {

    static public void userAtomicMethod(AtomicObject rObj, AtomicObject wObj) {
        AtomicAction a = new AtomicAction();

        a.begin();
        a.readLock(rObj);
        a.writeLock(wObj);
        /*...*/
        a.commit();
    }
}
```

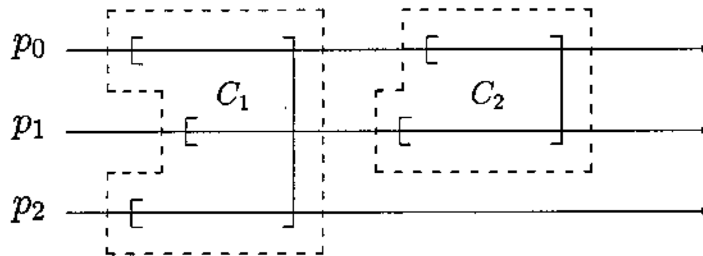


Figura 2.8: Diagrama espaço–tempo para o MPC

2.3 Modelo de Processos e Conversações (MPC)

O modelo de processos e conversações (MPC) [22] é uma extensão do modelo de processos e mensagens (MPM) que permite a construção de aplicações distribuídas confiáveis. Os protocolos para entrega de mensagens devem garantir que estas são enviadas sem serem corrompidas e respeitando a ordem de envio. Além disso, os processos devem ser capazes de salvar seu estado em memória estável.

Processos dentro de uma conversação podem se comunicar livremente uns com os outros, mas não com processos que não pertençam à conversação. A entrada na conversação pode ser feita a qualquer momento, mas a saída deve ser coordenada (todos os processos devem sair da conversação juntos). Os protocolos de entrada e saída impõem a gravação do estado corrente do processo em memória estável. A Figura 2.8 apresenta um diagrama espaço–tempo para o MPC com os estados gravados representados por colchetes.

MOA	MPC
Objetos	Processos
Ações atômicas	Conversações
Invocação de métodos	Troca de mensagens
Controle de concorrência para seriação	Regras contra comunicação externa
Estados estáveis de objetos	Estados estáveis de processos
Fase de crescimento	Entrada dos processos na conversação
Fase de encolhimento	Saída dos processos da conversação

Tabela 2.1: Dualidade entre MOA e MPC

2.4 Dualidade entre MOA e MPC

MPC é tipicamente empregado para a estruturação de sistemas que se preocupam com controle de tempo real, como sistemas de controle de tráfego aéreo. MOA, por outro lado, é largamente utilizado em aplicações baseadas em bancos de dados, como sistemas bancários. Apesar das áreas de aplicação serem distintas, Shrivastava et al. [22] constataram a *dualidade* desses dois modelos (Tabela 2.1).

Como resultado dessa dualidade, soluções desenvolvidas para um modelo podem ser facilmente transpostas para o outro. Por exemplo, técnicas desenvolvidas para replicação de objetos podem ser aplicadas à replicação de processos servidores. Os seguintes fatos foram observados:

- um programa construído utilizando estritamente as primitivas de um modelo pode ser mapeado diretamente em um programa dual que opera no outro modelo;
- os programas duais são logicamente idênticos e podem ser considerados textualmente similares e
- o desempenho de um programa é idêntico ao do seu dual.

Além da transferência de soluções de um modelo para o outro, essa dualidade serviu para dar evidências de que um dos modelos não é *superior* em relação ao outro. Isso contribuiu para diminuir dúvidas em relação ao uso do MOA em aplicações normalmente estruturadas no MPC.

2.5 Sumário

Neste Capítulo, apresentamos três modelos para estruturação de aplicações distribuídas: o modelo de processos e mensagens (MPM), o modelo de objetos e ações (MOA), e o modelo de processos e conversações (MPC).

O MPM é um modelo assíncrono, baseado em processos independentes que se comunicam exclusivamente através do envio e recepção de mensagens.

O MOA é um modelo síncrono, baseado em objetos que interagem através da invocação de métodos atômicos. As ações atômicas conferem um grau de confiabilidade à aplicação, devido às suas propriedades de equivalência serial, atomicidade e permanência de efeito. Consideramos que os objetos são controlados por gerentes de ações que se encarregam da distribuição de *locks* de leitura ou escrita.

O MPC é uma extensão do MPM, no qual processos só podem se comunicar com outros processos dentro de uma conversação. Os processos devem obedecer a um protocolo para a entrada e saída de conversações que exige a gravação do seu estado corrente em memória estável.

O MPC é dual ao MOA: podemos fazer um paralelo entre processos e objetos e entre conversações e ações atômicas. Inspirados nessa dualidade, exploramos, nos próximos capítulos, as correlações entre o MPM e o MOA.

Capítulo 3

Estados Globais Consistentes

Relações de precedência são fundamentais para o entendimento de uma aplicação distribuída e de suas execuções [21]. Devido às características de concorrência e assincronismo intrínsecas a sistemas distribuídos, a ausência de uma forma de se relacionar eventos tornaria a análise de um algoritmo distribuído muito difícil.

Considerando as relações de precedência, podemos construir algoritmos para a obtenção de estados globais consistentes, que são úteis para uma série de aplicações, como recuperação de falhas, depuração (*debugging*), reconfiguração e monitorização em geral [8].

Iniciamos este Capítulo com a exposição das relações de precedência e consistência para o MPM e para o MOA. Em seguida, propomos mecanismos para a manutenção e propagação de informação de precedência (relógios) entre os componentes da aplicação. A partir dos relógios, apresentamos algoritmos para o fotógrafo (Figura 1.1) construir estados globais consistentes progressivamente para enviar ao monitor. Baseados na teoria apresentada, encerramos o Capítulo explorando as correlações entre o MPM e o MOA.

3.1 Precedência entre Eventos no MPM

Apresentamos a relação "aconteceu antes" entre eventos como proposta por Lamport [13]. Outros autores [8, 21], em publicações posteriores, preferiram o termo "precedência causal" devido às conotações temporais impostas pela primeira expressão. Adotamos esta última convenção e dizemos que existe uma relação de precedência causal, ou simplesmente precedência, entre dois eventos, e e e', denotada por $(e \rightarrow e')$ quando e pode ter influenciado, direta ou indiretamente, a ocorrência de e'. Utilizamos $(e \not\rightarrow e')$ para indicar que $\neg(e \rightarrow e')$.

Definição 3.1 Precedência causal entre eventos: $e_a^\alpha \rightarrow e_b^\beta$

O evento e_a^α precede o evento e_b^β se, e somente se:

1. $(a = b) \wedge (\beta = 1 + \alpha)$, *ou*

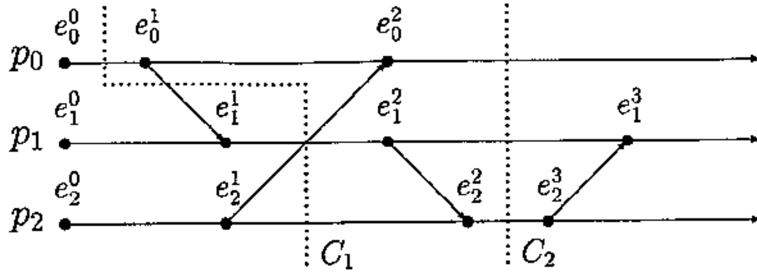


Figura 3.1: Relações de precedência no MPM

2. $\exists m : e_a^\alpha = envio(m) \wedge e_b^\beta = recepção(m)$, *ou*

3. $\exists e_l^\lambda : (e_a^\alpha \rightarrow e_l^\lambda) \wedge (e_l^\lambda \rightarrow e_b^\beta)$.

Dentro de um mesmo processo, um evento tem precedência causal direta sobre o seu evento sucessor. Em processos distintos, o envio de uma mensagem tem precedência causal direta sobre sua recepção. Além disso, caso um evento tenha precedido um segundo que, por sua vez, precedeu um terceiro, então o primeiro precedeu indiretamente o terceiro. Na Figura 3.1 podemos observar, por exemplo, as seguintes precedências $(e_0^1 \rightarrow e_0^2)$, $(e_1^2 \rightarrow e_2^2)$ e $(e_0^1 \rightarrow e_2^2)$.

Utilizando um diagrama espaço–tempo é possível determinar graficamente a precedência causal entre dois eventos. Para isso, basta verificar a existência de um caminho entre eles percorrendo-se linhas horizontais e arestas da esquerda para a direita.

Definição 3.2 Concorrência entre eventos: $e \| e'$
Eventos e e e′ são concorrentes se, e somente se:

$$(e \not\rightarrow e') \wedge (e' \not\rightarrow e)$$

Dois eventos e e e' são ditos concorrentes $(e \| e')$ se nenhuma das relações $(e \rightarrow e')$ ou $(e' \rightarrow e)$ é válida. Cabe notar que essa relação não é transitiva. Na Figura 3.1 podemos observar que $(e_2^3 \| e_0^2)$ e $(e_0^2 \| e_1^3)$, mas e_2^3 e e_1^3 correspondem, respectivamente, ao envio e à recepção de uma mesma mensagem, sendo, portanto, causalmente relacionados.

No decorrer do texto, para simplificar a notação, utilizamos $\sigma_a^\alpha \prec \sigma_b^\beta$ significando que o evento e_a^α, que gerou σ_a^α, precedeu o evento e_b^β, que gerou σ_b^β. Da mesma forma, utilizamos $\sigma_a^\alpha \not\prec \sigma_b^\beta$ se o evento que gerou σ_a^α não precedeu o evento que gerou σ_b^β e $\sigma_a^\alpha \| \sigma_b^\beta$ quando os eventos que geraram σ_a^α e σ_b^β foram concorrentes.

Consistência no MPM

Um corte é um subconjunto C do histórico global H formado por prefixos dos históricos locais de cada um dos processos da aplicação, $C = \{h_0^{c_0} \cup \ldots \cup h_{n-1}^{c_{n-1}}\}$. A ênupla de números naturais

$(\iota_0, \ldots, \iota_{n-1})$ define um conjunto de eventos $(e_0^{\iota_0}, \ldots, e_{n-1}^{\iota_{n-1}})$ que é denominado fronteira do corte.

Um corte é dito *consistente* se para todo par de eventos e e e' a seguinte relação é válida:

$$(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C$$

O fato de um evento e pertencer a um corte consistente C implica que todos os outros eventos da computação que o precedem causalmente também estão em C. Dessa forma, para toda mensagem m, não é permitida a presença do evento *recepção*(m) sem a inclusão do evento *envio*(m). Essa propriedade permite a verificação gráfica da consistência de um corte no diagrama espaço-tempo: uma aresta não pode ter sua ponta à esquerda e sua base à direita de um corte consistente.

Na Figura 3.1, podemos observar que o corte C_1 é inconsistente pois e_1^1 pertence a C_1 enquanto e_0^1 não (e_1^1 está à esquerda e e_0^1 à direita de C_1). C_2 é um exemplo de corte consistente.

Um estado global $\Sigma = (\sigma_0^{\iota_0}, \ldots, \sigma_{n-1}^{\iota_{n-1}})$ da computação é formado pela união dos estados dos processos na fronteira de um corte $C = (h_0^{\iota_0}, \ldots, h_{n-1}^{\iota_{n-1}})$. Salvo menção em contrário, não é feito nenhum tratamento especial para a obtenção do estado dos canais, uma vez que estes podem ser inferidos a partir dos estados dos processos [8]. Um estado global é consistente se foi gerado a partir de um corte consistente. Na Figura 3.1, podemos observar o estado global $\Sigma_1 = (\sigma_0^0, \sigma_1^1, \sigma_2^1)$ associado ao corte inconsistente C_1 e $\Sigma_2 = (\sigma_0^2, \sigma_1^2, \sigma_2^2)$ associado ao corte consistente C_2.

Dizemos que dois estados são inconsistentes entre si se não podem participar de um mesmo estado global consistente. A Definição 3.3 estabelece uma condição, baseada em precedência entre estados, para a inconsitência de um par de estados no MPM.

Definição 3.3 Inconsistência entre estados (MPM)
Dois estados de processos σ_a^α e σ_b^β são inconsistentes se, e somente se:

$$(\sigma_a^{\alpha+1} \prec \sigma_b^\beta) \vee (\sigma_b^{\beta+1} \prec \sigma_a^\alpha)$$

Um estado global consistente contém apenas pares de estados consistentes entre si. A Definição 3.4 utiliza a Definição 3.3 para a determinação da consistência de um estado global.

Definição 3.4 Estado Global Consistente
Um estado global $\Sigma = (\sigma_0^{\iota_0}, \ldots, \sigma_{n-1}^{\iota_{n-1}})$ é consistente se, e somente se:

$$\forall i, j (0 \leq i, j < n) : (\sigma_i^{\iota_i+1} \not\prec \sigma_j^{\iota_j})$$

À primeira vista, pode-se pensar que dois estados são consistentes apenas se são concorrentes. É possível, no entanto, que haja precedência e consistência. Na Figura 3.1, $e_1^2 \rightarrow e_2^2$ e estes eventos estão na fronteira do corte consistente C_2.

O conjunto de todos os estados globais consistentes possíves de uma aplicação forma o seu *reticulado* [8]. O reticulado é formado por n eixos ortogonais, sendo um eixo para

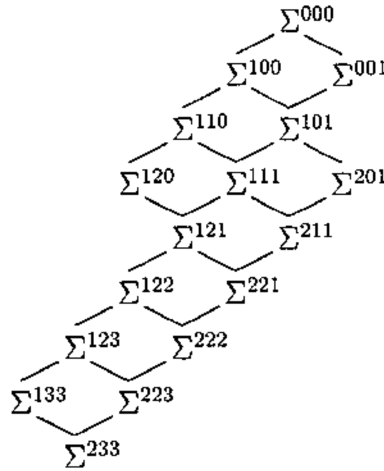


Figura 3.2: Reticulado referente à Figura 3.1

cada processo. Nos diagramas utilizamos a notação $\Sigma^{\iota_0 \cdots \iota_{n-1}}$ para denotar o estado global consistente $\Sigma = (\sigma_0^{\iota_0}, \ldots, \sigma_{n-1}^{\iota_{n-1}})$. A soma $\iota_0 + \iota_1 + \cdots + \iota_{n-1}$ define o *nível* do estado global, ou seja, o número total de eventos executados para se atingir esse estado. Estados globais de mesmo nível são apresentados na mesma linha do reticulado. Um caminho no reticulado representa uma execução válida. A Figura 3.2 mostra o reticulado referente ao cenário apresentado na Figura 3.1.

3.2 Precedência entre Ações Atômicas no MOA

De maneira análoga à relação de precedência causal entre eventos [13], podemos estabelecer uma relação de precedência entre ações atômicas.

O primeiro tipo de precedência está relacionado à semântica da aplicação, representada pelas estruturas de ações (Seção 2.2.2). A *precedência estrutural* (Definição 3.5) é análoga à precedência existente entre eventos executados por um mesmo processo. Uma diferença marcante está no fato das estruturas de ações formarem um grafo, em contraposição às seqüências simples de execução de eventos em processos.

Definição 3.5 Precedência estrutural entre ações atômicas: $\mathsf{a} \prec_s \mathsf{a}'$

Dizemos que uma ação atômica a *precede estruturalmente uma outra ação atômica* a′ *(*$\mathsf{a} \prec_s \mathsf{a}'$*) se, e somente se, existe um caminho de* a *a* a′ *no grafo formado pelas estruturas de ações da aplicação.*

O evento de envio de uma mensagem no MPM precede a sua recepção. No MOA, se uma ação atômica a′ leu um estado escrito por a, a precede a′. Uma outra forma de precedência

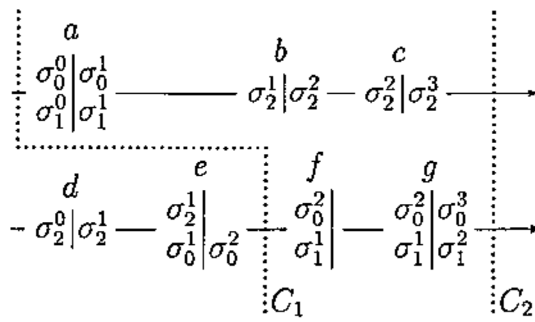


Figura 3.3: Relações de precedência no MOA

no MOA ocorre quando uma ação atômica a leu um estado antes dele ser alterado por uma ação a'. Nesse caso, a precede a' mas pode não ter havido fluxo de informação entre a e a' (Figura 3.3, ações atômicas *e* e *b*). Por esta razão, não denominamos a relação definida a seguir de precedência causal, mas simplemente de *precedência*.

Definição 3.6 Precedência entre ações atômicas: $a \prec a'$
A ação atômica a *precede a ação atômica* a' *se, e somente se:*

1. $a \prec_s a'$, *ou*
2. $Out(a) \cap LockRW(a') \neq \emptyset$, *ou*
3. $LockR(a) \cap LockW(a') \neq \emptyset$, *ou*
4. $\exists a'' : (a \prec a'') \wedge (a'' \prec a')$.

Utilizamos $(a \not\prec a')$ quando $\neg(a \prec a')$. Como no MPM, a relação de precedência entre ações atômicas é transitiva. Na Figura 3.3 podemos observar que $b \prec c$, $a \prec e$, $e \prec b$ e $e \prec c$.

Definição 3.7 Concorrência entre ações atômicas: $a \| a'$
Duas ações atômicas a *e* a' *são concorrentes se, e somente se:*

$$(a \not\prec a') \wedge (a' \not\prec a)$$

Quando não há relação de precedência entre duas ações atômicas a e a', dizemos que elas são *concorrentes*. Na Figura 3.3 podemos observar os seguintes pares de ações concorrentes: $a\|d$, $b\|f$, $b\|g$, $c\|f$ e $c\|g$.

Como no MPM, simplificamos a notação utilizando $\sigma \prec \sigma'$ significando que a ação atômica que promoveu σ precedeu a ação atômica que promoveu σ' $(Act(\sigma) \prec Act(\sigma'))$. Da mesma forma, utilizamos $\sigma \not\prec \sigma'$ quando a ação que promoveu σ não precedeu a que promoveu σ' $(Act(\sigma) \not\prec Act(\sigma'))$ e $\sigma\|\sigma'$ quando as ações que promoveram σ e σ' foram concorrentes

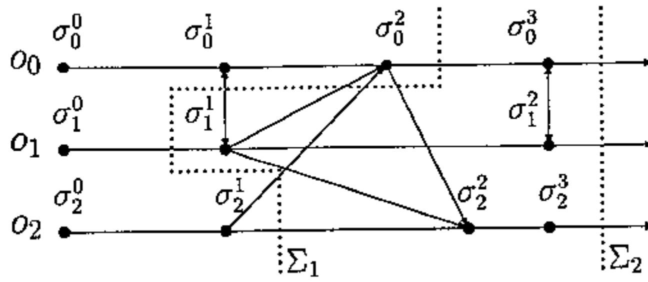


Figura 3.4: Diagrama objeto–tempo (Cenário Figura 3.3)

$(Act(\sigma) \| Act(\sigma'))$. Necessitamos de uma notação adicional para o caso em que dois estados foram promovidos pela mesma ação atômica, ou seja, são estados *simultâneos*.

Definição 3.8 Simultaneidade entre estados de objetos: *$(\sigma \leftrightarrow \sigma')$*
Dois estados de objetos σ e σ' são simultâneos se, e somente se, foram gerados na mesma ação atômica:

$$(\sigma \leftrightarrow \sigma') \equiv Act(\sigma) = Act(\sigma')$$

O diagrama espaço–tempo para ações atômicas, como apresentado na Figura 3.3, não nos permite uma verificação gráfica simples da precedência entre estados de objetos. Propomos um diagrama *objeto-tempo* similar ao diagrama espaço–tempo para processos (Figura 3.4). As linhas horizontais representam os estados percorridos pelos objetos, com o tempo progredindo da esquerda para a direita. Uma aresta simples ligando dois estados representa uma relação de precedência imediata (não devida à transitividade) entre esses estados. Uma aresta com duas pontas indica estados simultâneos. Vale ressaltar que, ao contrário do MPM, várias arestas podem sair de ou chegar em um mesmo estado.

Para determinar uma relação de precedência entre estados no diagrama objeto–tempo, basta percorrer linhas horizontais e arestas simples da esquerda para a direita e arestas duplas em qualquer sentido. Cabe notar que um caminho restrito a uma única aresta dupla indica simultaneidade e não precedência.

Consistência no MOA

Um corte no MOA é formado por um prefixo das estruturas de ações:

$$(\mathsf{a} \in C) \wedge (\mathsf{a}' \prec_s \mathsf{a}) \Rightarrow (\mathsf{a}' \in C).$$

Um corte é consistente se é fechado à esquerda no que diz respeito às relações de precedência entre as ações atômicas, ou seja

$$(\mathsf{a} \in C) \wedge (\mathsf{a}' \prec \mathsf{a}) \Rightarrow (\mathsf{a}' \in C).$$

Na Figura 3.3, C_1 é inconsistente pois a ação atômica e, pertencente a C_1, leu um estado de objeto σ_0^1 escrito pela ação a, que não pertence a C_1. C_2 é um exemplo de corte consistente.

A cada corte C associamos um estado global $\Sigma = (\sigma_0^{\iota_0}, \ldots, \sigma_{n-1}^{\iota_{n-1}})$ da seguinte maneira:

$$\forall i(0 \leq i < n), \forall \mathsf{a} \in C : \iota_i = max(\alpha : \sigma_i^\alpha \in (Out(\mathsf{a}) \cup LockR(\mathsf{a})))$$

com $\iota_i = 0$ caso esse conjunto seja vazio. Nas Figuras 3.3 e 3.4, podemos observar $\Sigma_1 = (\sigma_0^2, \sigma_1^0, \sigma_2^1)$ associado a C_1 e $\Sigma_2 = (\sigma_0^3, \sigma_1^2, \sigma_2^3)$ associado a C_2.

A cada corte está associado um único estado global, enquanto o contrário pode não ser verdade devido às ações atômicas só de leitura. Um estado global Σ é *garantidamente* consistente se está associado a um corte consistente. Em um diagrama como o representado na Figura 3.4 a verificação gráfica da consistência pode ser feita de maneira semelhante ao MPM: uma aresta simples não pode ter sua ponta à esquerda e sua base à direita de um corte consistente e uma aresta dupla não pode ser atravessada pelo corte.

Como no MPM, dizemos que dois estados são inconsistentes entre si se não podem participar de um mesmo estado global consistente. A Definição 3.9 estabelece uma condição, baseada em precedência e simultaneidade entre estados, para a inconsitência de um par de estados no MOA.

Definição 3.9 Inconsistência entre estados (MOA)
Dois estados de objetos σ_a^α e σ_b^β são inconsistentes se, e somente se:

$$(\sigma_a^{\alpha+1} \prec \sigma_b^\beta) \vee (\sigma_a^{\alpha+1} \leftrightarrow \sigma_b^\beta) \vee (\sigma_b^{\beta+1} \prec \sigma_a^\alpha) \vee (\sigma_b^{\beta+1} \leftrightarrow \sigma_a^\alpha)$$

Um estado global consistente contém apenas pares de estados consistentes entre si. A Definição 3.10 utiliza a Definição 3.9 para a determinação da consistência de um estado global.

Definição 3.10 Estado Global Consistente
Um estado global $\Sigma = (\sigma_0^{\iota_0}, \ldots, \sigma_{n-1}^{\iota_{n-1}})$ é consistente se, e somente se:

$$\forall i, j(0 \leq i, j < n) : (\sigma_i^{\iota_i+1} \not\prec \sigma_j^{\iota_j}) \wedge (\sigma_i^{\iota_i+1} \not\leftrightarrow \sigma_j^{\iota_j})$$

Podemos construir o reticulado de uma aplicação no MOA de maneira análoga ao que foi feito no MPM. Estados globais $\Sigma = (\sigma_0^{\iota_0}, \ldots, \sigma_{n-1}^{\iota_{n-1}})$ são representados por $\Sigma^{\iota_0 \cdots \iota_{n-1}}$. A soma $\iota_0 + \iota_1 + \cdots + \iota_{n-1}$ *não* indica o número de ações atômicas executadas para se chegar a esse estado e sim o número de estados de objetos gerados até esse estado global. Obviamente, esse número não é igual ao número de ações atômicas que obtiveram *lock* de escrita, pois uma mesma ação pode promover o estado de vários objetos. A Figura 3.5 apresenta o reticulado referente à Figura 3.3.

3.3 Propagação de Relógios

Chamamos de *relógio* qualquer informação de precedência associada a eventos ou ações atômicas e o representamos como τ. O valor de relógio de um estado ($\tau(\sigma)$) é idêntico ao valor de relógio do evento ($\tau(e)$) ou ação atômica ($\tau(Act(\sigma))$) que o promoveu.

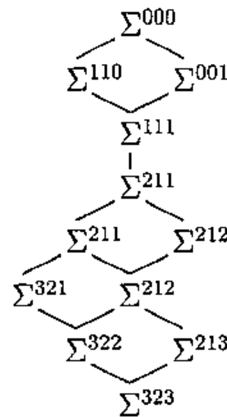


Figura 3.5: Reticulado referente ao cenário da Figura 3.3

Interface 3.1 Clock.java

```
public interface Clock {

    /** Increments the clock information based on the process id */
    void increment(int pid);

    /** Increments the clock information based on the write locked object set */
    void increment(boolean[ ] W);

    /** Keeps the maximum between the stored information and the parameter given */
    void maximum(Clock clock);

    /** Returns clock information to be propagated through the application */
    Clock propagate();
}
```

No MPM, relógios são mantidos pelos processos e propagados através das mensagens da aplicação [8, 21]. A atribuição de um valor de relógio a um evento deve levar em consideração as mensagens recebidas por esse processo, bem como o seu identificador único.

A propagação de relógios no MOA não é tão simples como no MPM, necessitando da colaboração do gerente de ações (Seção 3.3.2). Além disso, a atribuição de um valor de relógio a uma ação atômica deve levar em consideração os relógios vistos pela estrutura de ações à qual essa ação pertence e os identificadores de todos os objetos que participaram com *lock* de escrita nessa ação.

A interface `Clock` (Interface 3.1) define uma interface em Java para representação de relógios, com métodos para incremento, máximo e propagação de relógios. Foram coloca-

Evento interno ou de envio: $\tau(e_i) \leftarrow incremento(\tau)$;
Evento de recepção: $\tau(e_i) \leftarrow max\{\tau(p_i), \tau(m)\}; incremento(\tau(p_i))$;

Figura 3.6: Propagação de relógios no MPM

dos dois métodos para incremento. O primeiro, associado ao MPM, incrementa o relógio baseando-se no identificador único do processo. O segundo, associado ao MOA, incrementa o relógio levando em consideração os identificadores dos objetos sobre os quais foram adquiridos *locks* de escrita.

3.3.1 Propagação de Relógios no MPM

Todo processo p_i mantém um relógio $\tau(p_i)$, que é incrementado em toda execução de evento. Toda mensagem enviada por p_i é carimbada com $\tau(m) = \tau(envio(m))$. Ao receber uma mensagem m', p_i calcula o máximo entre $\tau(p_i)$ e $\tau(m')$, incrementa esse valor e o atribui a $\tau(p_i)$ (algoritmo descrito na Figura 3.6). O Teorema 3.1 garante que os valores de relógio propagados dessa maneira são consistentes com as relações de precedência, ou seja, $e \rightarrow e'$ implica que $\tau(e) < \tau(e')$.

Teorema 3.1 *Sejam e e e' dois eventos e τ o relógio propagado segundo o algoritmo descrito na Figura 3.6. Neste caso, temos:*

$$e \rightarrow e' \Rightarrow \tau(e) < \tau(e')$$

Prova: Dada uma relação de precedência entre dois eventos ($e \rightarrow e'$), essa relação pode ser expandida em uma seqüência de relações:

$$e = e_{i_0}^{\iota_0} \rightarrow e_{i_1}^{\iota_1} \rightarrow \ldots \rightarrow e_{i_p}^{\iota_p} = e' \quad p \geq 1$$

tal que as relações $e_x^{\iota_x} \rightarrow e_{x+1}^{\iota_{x+1}}$, $0 \leq x < p$, são devidas a um dos itens (1) ou (2) da Definição 3.1 (sem considerar transitividade). Provamos por indução no tamanho dessa seqüência que $\tau(e) < \tau(e')$.

Base: $p = 1 \quad e = e_{i_0}^{\iota_0} \rightarrow e_{i_1}^{\iota_1} = e'$

Analisamos os casos (1) e (2) da Definição 3.1 separadamente:

1. $(i_0 = i_1) \wedge (\iota_1 = 1 + \iota_0)$—$\tau(e) < \tau(e')$, pois o processo p_{i_0} deve incrementar $\tau(p_{i_0})$ em toda execução de evento.

2. $\exists m : e_{i_0}^{\iota_0} = envio(m) \wedge e_{i_1}^{\iota_1} = recepção(m)$—Ao receber m, p_{i_1} calcula o máximo entre $\tau(p_{i_1})$ e $\tau(m)$, incrementa esse valor e o atribui a $\tau(recepção(m))$. Logo, $\tau(e) < \tau(e')$.

Passo: Suponha que a hipótese é válida para uma seqüência de tamanho p. Vamos provar que continua válida para uma seqüência de tamanho $p + 1$. Consideramos:

Classe 3.2 ClockProcess.java

```
public class ClockProcess extends Process {

    protected Clock clock;

    public class ClockMessage extends Message {
        public Clock clock;
    }

    public ClockProcess(int pid, Clock clock) {
        super(pid);
        this.clock = clock;
    }

    public void sendMessage(ClockMessage message) {
        clock.increment(pid);
        message.clock = clock.propagate();
        super.sendMessage(message);
    }

    public void receiveMessage(ClockMessage message) {
        clock.maximum(message.clock);
        clock.increment(pid);
        super.receiveMessage(message);
    }

    public void internalEvent() {
        clock.increment(pid);
        super.internalEvent();
    }
}
```

$$e = e_{i_0}^{t_0} \to e_{i_1}^{t_1} \to \ldots \to e_{i_p}^{t_p} \to e_{i_{p+1}}^{t_{p+1}} = e' \quad \tau(e) < \tau(e_{i_p}^{t_p})$$

Pela regra de formação da seqüência, $e_{i_p}^{t_p} \to e_{i_{p+1}}^{t_{p+1}}$ sem considerar transitividade. Podemos aplicar a base da indução e concluir que $\tau(e_{i_p}^{t_p}) < \tau(e_{i_{p+1}}^{t_{p+1}})$. Logo, $\tau(e) < \tau(e')$. □

A classe `ClockProcess` (Classe 3.2) especializa a classe `Process` para a propagação de relógios segundo o algoritmo da Figura 3.6.

3.3.2 Propagação de Relógios no MOA

Os canais de comunicação disponíveis no MOA são os estados de objetos e as estruturas de ações da aplicação. Infelizmente, esses canais não permitem a troca de toda a informação necessária para a propagação de relógios. O terceiro item da Definição 3.6 permite a construção de cenários nos quais uma ação precede outra, mas não há possibilidade de comunicação de

`Begin`	$\mathcal{R} \leftarrow \emptyset$; $\mathcal{W} \leftarrow \emptyset$;
	$\tau(\mathsf{a}) \leftarrow max(\tau(\mathsf{a}' \mid \mathsf{a}' \prec_S \mathsf{a}))$;
`Read_lock`(i)	$\tau(\mathsf{a}) \leftarrow max(\tau(o_i), \tau(\mathsf{a}))$;
	$\mathcal{R} \leftarrow \mathcal{R} \cup \{i\}$;
`Write_lock`(i)	$\tau(\mathsf{a}) \leftarrow max(\tau(\mathsf{m}_i), \tau(\mathsf{a}))$;
	$\mathcal{W} \leftarrow \mathcal{W} \cup \{i\}$;
`Commit`	if $\mathcal{W} \neq \emptyset$
	$\tau(\mathsf{a}) \leftarrow incremento(\tau(\mathsf{a}))$;
	$\forall i \in \mathcal{W}$ do $\tau(o_i) \leftarrow \tau(\mathsf{a})$;
`Unlock`$(i)_{i \in \mathcal{R}}$	$\tau(\mathsf{m}_i) \leftarrow max(\tau(\mathsf{m}_i), \tau(\mathsf{a}))$;
`Unlock`$(i)_{i \in \mathcal{W}}$	$\tau(\mathsf{m}_i) \leftarrow \tau(\mathsf{a})$;

Figura 3.7: Fluxo de informação de precedência no MOA

relógios utilizando os canais citados. Na Figura 3.3 podemos observar que $(\sigma_0^2 \prec \sigma_2^2)$, mas $Act(\sigma_0^2)$ e $Act(\sigma_2^2)$ pertencem a estruturas de ações concorrentes e não houve *lock* sobre o objeto o_0 em $Act(\sigma_2^2)$.

Para preencher essa lacuna utilizamos os gerentes de ações como canais de comunicação. Os gerentes de ações acumulam relógios recebidos das ações atômicas que adquiriram *lock* de leitura sobre o objeto gerenciado e adicionam essa informação ao estado do objeto no próximo *lock* de escrita. A Figura 3.7 descreve a propagação de relógios em uma ação atômica de maneira consistente com as relações de precedência (Teoremas 3.2 e 3.3), sendo que m_i representa o gerente de ações do objeto o_i.

Teorema 3.2 *Sejam* a *e* a' *duas ações atômicas e* τ *o relógio propagado conforme o algoritmo descrito na Figura 3.7. Neste caso, temos:*

$$\mathsf{a} \prec \mathsf{a}' \Rightarrow \tau(\mathsf{a}) \leq \tau(\mathsf{a}')$$

Prova: Quando a relação $(\mathsf{a} \prec \mathsf{a}')$ é válida, ela pode ser expandida em uma seqüência de relações:

$$\mathsf{a} = \mathsf{a}_0 \prec \mathsf{a}_1 \prec \cdots \prec \mathsf{a}_p = \mathsf{a}' \quad p \geq 1$$

onde cada uma das relações $\mathsf{a}_k \prec \mathsf{a}_{k+1}$ $(0 \leq k < p)$ é devida a um dos itens (1), (2) ou (3) da Definição 3.6.

Provamos por indução no tamanho dessa seqüência que $\tau(\mathsf{a}) \leq \tau(\mathsf{a}')$.

Base: $p = 1$.

$$\mathsf{a} = \mathsf{a}_0 \prec \mathsf{a}_1 = \mathsf{a}'$$

Analisamos os casos (1), (2) e (3) da Definição 3.6 separadamente:

1. $a_0 \prec_S a_1$

   No passo `Begin` do algoritmo, $\tau(a_1)$ recebe um valor inicial que garante $\tau(a_0) \leq \tau(a_1)$. Como os outros passos do algoritmo aplicam a $\tau(a_1)$ apenas operações de máximo ou incremento, concluímos que $\tau(a_0) \leq \tau(a_1)$.

2. $Out(a_0) \cap LockRW(a_1) \neq \emptyset$

   Através do `Commit` de a_0 sabemos que todo objeto o_i que teve seu estado promovido por a_0 e seu respectivo gerente de ações receberam $\tau(a_0)$ ($\tau(\sigma_i) \leftarrow \tau(a_0)$ e $\tau(m_i) \leftarrow \tau(a_0)$). Vamos analisar separadamente os casos em que a_1 adquiriu *lock* de leitura ou escrita sobre σ_i.

   - $\sigma_i \in LockR(a_1)$—Pelo passo `Read_lock` sabemos que $\tau(\sigma_i) \leq \tau(a_1)$. Como $\tau(\sigma_i) = \tau(a_0)$, temos $\tau(a_0) \leq \tau(a_1)$.
   - $\sigma_i \in LockW(a_1)$—Pelo passo `Write_lock` sabemos que $\tau(m_i) \leq \tau(a_1)$. Cabe notar que $\tau(m_i)$ pode ter sido alterado pelo passo `Unlock` de alguma outra ação atômica que tenha adquirido *lock* de leitura sobre σ_i, mas, nesse caso, só pode ter recebido valores maiores que $\tau(a_0)$.[1] Então, $\tau(a_0) \leq \tau(a_1)$.

3. $LockR(a_0) \cap LockW(a_1) \neq \emptyset$

   Considere um estado σ_i lido por a_0 e promovido por a_1. Após o `Commit` de a_0, $\tau(a_0) \leq \tau(m_i)$. Além disso, pelo passo `Write_lock` de a_1, o valor $\tau(m_i)$ foi incorporado em $\tau(a_1)$ e concluímos que $\tau(a_0) \leq \tau(a_1)$. Nesse caso, como $LockW(a_1) \neq \emptyset$, sabemos que $\tau(a_1)$ recebeu um incremento após incorporar $\tau(a_0)$ e portanto $\tau(a_0) < \tau(a_1)$.

Passo: Suponha que a hipótese é válida para uma seqüência de tamanho p. Vamos provar que continua válida para uma seqüência de tamanho $p+1$. Consideramos:

$$a = a_0 \prec a_1 \prec \cdots \prec a_p \prec a_{p+1} = a' \text{ e } \tau(a) \leq \tau(a_p)$$

Pela construção da seqüência, $a_p \prec a_{p+1}$ sem transitividade. Podemos aplicar novamente a base da indução e concluir que $\tau(a_p) \leq \tau(a_{p+1})$. Nesse caso, $\tau(a) \leq \tau(a')$. □

Teorema 3.3 *Sejam σ e σ' dois estados de objetos e $Act(\sigma)$ e $Act(\sigma')$ as ações atômicas que os promoveram. Sejam $\tau(\sigma)$ e $\tau(\sigma')$ os relógios associados a σ e σ' e propagados segundo o algoritmo descrito na Figura 3.7. Neste caso, temos:*

$$\sigma \prec \sigma' \Rightarrow \tau(\sigma) < \tau(\sigma')$$

[1] O algoritmo garante que o valor $\tau(m_i)$ permanece monotonicamente crescente mesmo em presença de uma ação atômica a'' que tenha adquirido um *lock* de escrita sobre o_i. No passo `Write_lock` de a'' sabemos que $\tau(m_i) \leq \tau(a'')$ e pelo passo `Unlock` de a'' sabemos que $\tau(m_i)$ recebeu o valor de $\tau(a'')$ que pode ter aumentado, mas não diminuído.

Classe 3.3 ClockActionManager.java

```
public class ClockActionManager extends ActionManager {

    protected Clock managerClock; // Clock associated to the action manager
    protected Clock objectClock; // Clock associated to the object

    public ClockActionManager(AtomicObject obj) { super(obj); }

    public synchronized Clock rClock() { return objectClock; }

    public synchronized Clock wClock() { return managerClock; }

    public synchronized void updateRClock(Clock clock) { managerClock.maximum(clock); }

    public synchronized void updateWClock(Clock clock) {
        objectClock = clock.propagate();
        managerClock = clock.propagate();
    }
}
```

Prova: O Teorema 3.2 garante que $\tau(\sigma) \leq \tau(\sigma')$. Além disso, no passo (3) da base da indução da prova do Teorema 3.2 demonstramos que quando uma ação promove um estado, seu relógio é estritamente maior que o de seus predecessores. Assim, considerando transitividade, concluímos que $\tau(\sigma) = \tau(Act(\sigma)) < \tau(Act(\sigma')) = \tau(\sigma')$. □

As classes `ClockActionManager` (Classe 3.3) e `ClockAtomicAction` (Classe 3.4) especializam, respectivamente, as classes `ActionManager` e `AtomicAction` para armazenamento e propagação de relógios segundo o algoritmo descrito na Figura 3.7. Cabe notar que o relógio relacionado ao objeto foi colocado junto ao gerente de ações, de maneira a não alterar a classe `AtomicObject`.

Antes de uma chamada ao método `begin` da classe `ClockAtomicAction`, deve-se invocar o método `setClockAtBeginning` passando como parâmetro o valor de relógio referente à estrutura de ações à qual essa instância de ação atômica pertence. Ao final da ação atômica, o método `getClockAtCommit` pode ser chamado para dar continuidade à propagação de relógios pela estrutura de ações. A classe `userClockClass` (Classe 3.5) modifica o exemplo apresentado na Classe 2.6 para ilustrar a propagação de relógios em uma aplicação MOA.

3.4 Relógios Lógicos

Os *relógios lógicos* (*LC*) definidos por Lamport [13] utilizam números naturais para a implementação de relógios. A classe `LogicalClock` (Classe 3.6) implementa a interface `Clock`

Classe 3.4 ClockAtomicAction.java

```
public class ClockAtomicAction extends AtomicAction {

    protected Clock clock;
    protected boolean[ ] W;

    public void readLock(AtomicObject obj) {
        super.readLock(obj);
        clock.maximum(((ClockActionManager) obj.actionManager).rClock());
    }

    public void writeLock(AtomicObject obj) {
        super.writeLock(obj);
        W[obj.oid] = true;
        clock.maximum(((ClockActionManager) obj.actionManager).wClock());
    }

    public void setClockAtBeginning(Clock clock) { this.clock = clock.propagate(); }

    public void begin () {
        super.begin();
        for (int i=0; i < Application.N; i++) W[i] = false;
    }

    public void commit() {
        clock.increment(W);
        for (int i=0; i < nRLocked; i++)
            ((ClockActionManager) rLocked[i]).updateRClock(clock);
        for (int i=0; i < nWLocked; i++)
            ((ClockActionManager) wLocked[i]).updateWClock(clock);
        super.commit();
    }

    public Clock getClockAtCommit() { return clock.propagate(); }
}
```

Classe 3.5 userClockClass.java

```
public class userClockClass {

    /** Clock information is propagated through the methods invocations. */
    static public void userClockMethod(Clock clock, AtomicObject rObj, AtomicObject wObj) {
        ClockAtomicAction a = new ClockAtomicAction();
        a.setClockAtBeginning(clock);
        a.begin();
        a.readLock(rObj);
        a.writeLock(wObj);
        /*...*/
        a.commit();
        clock = a.getClockAtCommit();
    }
}
```

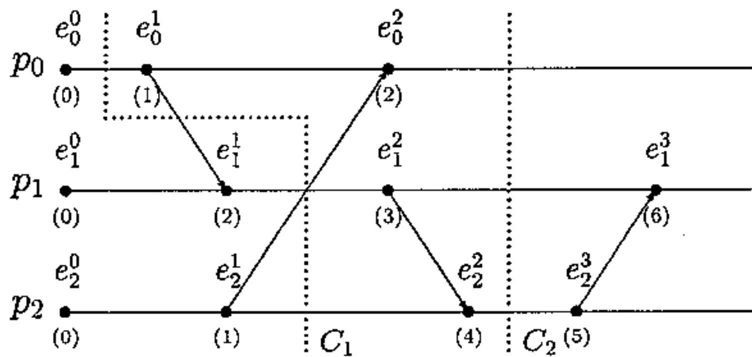


Figura 3.8: Cenário MPM com relógios lógicos

para a propagação de relógios lógicos. As Figuras 3.8 e 3.9 reapresentam os cenários das Figuras 3.1 e 3.3 com os relógios lógicos associados a cada evento ou ação atômica entre parênteses.

Apesar de consistentes com a relação de precedência, os relógios lógicos não a *caracterizam* [8]. Dados dois eventos e e e', com $LC(e) < LC(e')$, não é possível determinar se $e \rightarrow e'$ ou $e \| e'$. Na Figura 3.8 vemos que $LC(\sigma_0^1) = LC(\sigma_2^1) = 1$, $LC(\sigma_1^1) = 2$, $e_0^1 \rightarrow e_1^1$ e $e_2^1 \| e_1^1$. Da mesma forma, dadas duas ações atômicas a e a', com $LC(\mathsf{a}) < LC(\mathsf{a}')$ não podemos determinar se $\mathsf{a} \prec \mathsf{a}'$ ou $\mathsf{a} \| \mathsf{a}'$. Na Figura 3.9 vemos que $b \prec c$ e $g \| c$ embora $LC(b) = LC(g)$.

No MPM, se dois estados de processos têm o mesmo LC, podemos concluir que eles são concorrentes. No MOA, entretanto, eles podem ser concorrentes ou simultâneos. Na Figura 3.9 podemos observar que $LC(\sigma_0^1) = LC(\sigma_1^1) = LC(\sigma_2^1)$, enquanto $(\sigma_0^1 \leftrightarrow \sigma_1^1)$ e $(\sigma_0^1 \| \sigma_2^1)$.

Classe 3.6 LogicalClock.java

```
public class LogicalClock implements Clock {

    public int lc;

    public LogicalClock() { lc = 0; }
    protected LogicalClock(LogicalClock LC) { lc = LC.lc; }

    public void increment(int pid) { lc++; }

    public void increment(boolean[ ] W) {
        for (int i = 0; i < Application.N; i++)
            if (W[i]) { lc++; return; }
    }

    public void maximum(Clock LC) {
        if ( ((LogicalClock) LC).lc > lc )
            lc = ((LogicalClock) LC).lc;
    }

    public Clock propagate() { return new LogicalClock(this); }
}
```

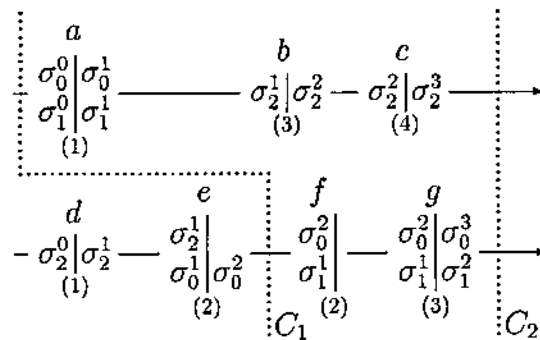


Figura 3.9: Cenário MOA com relógios lógicos

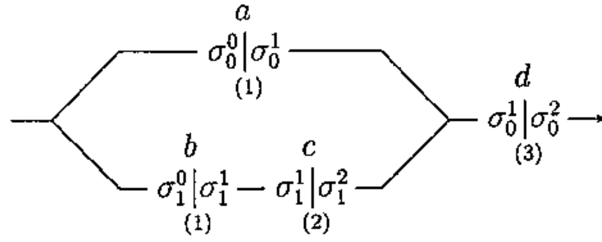


Figura 3.10: Relógios lógicos e detecção de *gaps* no MOA

Dado um par de eventos e e e' tal que $LC(e) < LC(e')$, não é sempre possível garantir a existência de um terceiro evento e'' tal que $LC(e) < LC(e'') < LC(e')$. Na Figura 3.8 podemos observar que $LC(e_2^1) < LC(e_2^2)$ e não existe evento e_i^t tal que $e_2^1 \rightarrow e_i^t$ e $e_i^t \rightarrow e_2^2$. Numa situação semelhante, podemos observar que $LC(e_1^2) < LC(e_1^3)$, sendo que e_2^2 e e_2^3 são tais que $e_1^2 \rightarrow e_2^2$, $e_2^2 \rightarrow e_2^3$ e $e_2^3 \rightarrow e_1^3$.

Esse problema é conhecido como *detecção de gaps* [8] e também ocorre no MOA. Na Figura 3.10 podemos observar que $LC(b) < LC(d)$ e $b \prec c \prec d$. Da mesma forma, $LC(a) < LC(d)$ e não existe uma ação atômica x tal que $a \prec x \prec d$.

Vamos listar algumas propriedades que podem ser derivadas na presença de relógios lógicos, utilizando o termo condição *fraca* para implicações.

Propriedade 3.1 Condição fraca para precedência (MPM e MOA)
Dados dois estados de processos ou objetos σ e σ':

$$\sigma \prec \sigma' \Rightarrow LC(\sigma) < LC(\sigma')$$

Propriedade 3.2 Condição fraca para concorrência (MPM)
Dados dois estados de processos σ e σ, $\sigma \neq \sigma'$:

$$LC(\sigma) = LC(\sigma') \Rightarrow (\sigma \| \sigma')$$

Propriedade 3.3 Condição fraca para concorrência ou simultaneidade (MOA)
Dados dois estados de objetos σ e σ, $\sigma \neq \sigma'$:

$$LC(\sigma) = LC(\sigma') \Rightarrow (\sigma \| \sigma') \vee (\sigma \leftrightarrow \sigma')$$

As propriedades para a inconsistência entre pares de estados são obtidas através da aplicação das Definições 3.3 e 3.9 e da Propriedade 3.1.

Propriedade 3.4 Condição fraca para inconsistência (MPM)
Caso dois estados de processos σ_a^α e σ_b^β sejam inconsistentes, a seguinte condição é válida:

$$LC(\sigma_a^\alpha) > LC(\sigma_b^{\beta+1}) \vee LC(\sigma_b^\beta) > LC(\sigma_a^{\alpha+1})$$

Propriedade 3.5 Condição fraca para inconsistência (MOA)
Caso dois estados de objetos σ_a^α *e* σ_b^β *sejam inconsistentes, a seguinte condição é válida:*

$$LC(\sigma_a^\alpha) \geq LC(\sigma_b^{\beta+1}) \vee LC(\sigma_b^\beta) \geq LC(\sigma_a^{\alpha+1})$$

As propriedades para a consistência de estados globais são obtidas através da aplicação das Definições 3.4 e 3.10 e da Propriedade 3.1.

Propriedade 3.6 Condição fraca para estado global consistente (MPM)
Um estado global $\Sigma = (\sigma_0^{\iota_0}, \ldots, \sigma_{n-1}^{\iota_{n-1}})$ *é consistente se:*

$$\forall i,j (0 \leq i,j < n) : LC(\sigma_i^{\iota_i+1}) \geq LC(\sigma_j^{\iota_j})$$

Propriedade 3.7 Condição fraca para estado global consistente (MOA)
Um estado global $\Sigma = (\sigma_0^{\iota_0}, \ldots, \sigma_{n-1}^{\iota_{n-1}})$ *é consistente se:*

$$\forall i,j (0 \leq i,j < n) : LC(\sigma_i^{\iota_i+1}) > LC(\sigma_j^{\iota_j})$$

3.5 Vetores de Relógios

Relógios lógicos são consistentes com a causalidade, mas não a caracterizam [8]. Vamos considerar abordagens *força-bruta* para a caracterização da precedência no MPM e MOA, partindo depois para uma abordagem mais eficiente, utilizando vetores de tamanho n.

3.5.1 Vetores de Relógios no MPM

Podemos acrescentar a cada evento informação a respeito de todos os eventos da computação que o precedem causalmente [8, 21]. Definimos o histórico causal de um evento, $\theta(e)$, como o conjunto:

$$\theta(e) = \{e' : e' \to e\} \cup \{e\}$$

A verificação da precedência causal entre dois eventos pode ser calculada da seguinte maneira:

$$e \to e' \equiv \theta(e) \subset \theta(e')$$

podendo ser reduzida a $e \in \theta(e')$ quando $e' \neq e$. A relação de precedência entre estados de processos pode ser calculada da mesma forma.

A projeção do histórico causal de um evento $\theta(e)$ sobre um processo p_i é definida pelo conjunto $\theta_i(e) = \theta(e) \cap h_i$. O conjunto $\theta_i(e)$ corresponde a um prefixo h_i^k, possivelmente vazio, do histórico local de p_i. Como o inteiro k é capaz de identificar unicamente o conjunto de eventos $\{e_i^0 \ldots e_i^k\}$, um vetor com n inteiros é suficiente para a representação do histórico causal de um evento. Tal vetor é definido da seguinte maneira:

$$VC(e)[i] = \begin{cases} 0 & \text{se, e somente se, } \theta_i(e) = \emptyset \\ k > 0 & \text{se, e somente se, } \theta_i(e) = \{e_i^0 \dots e_i^k\} \end{cases} \tag{3.1}$$

Como os eventos iniciais não são precedidos por nenhum evento da computação, seus vetores de relógios têm todas as entradas iguais a 0.

3.5.2 Vetores de Relógios no MOA

A cada ação atômica podemos acrescentar informação a respeito de todas as ações atômicas que a precederam. O histórico de ações precedentes de uma ação, $\theta(\mathsf{a})$, pode ser definido como:

$$\theta(\mathsf{a}) = \{\mathsf{a}' : \mathsf{a}' \prec \mathsf{a}\} \cup \{\mathsf{a}\}$$

A precedência entre duas ações atômicas pode ser verificada da seguinte maneira:

$$\mathsf{a} \prec \mathsf{a}' \equiv \theta(\mathsf{a}) \subset \theta(\mathsf{a}')$$

podendo ser reduzida a $\mathsf{a} \in \theta(\mathsf{a}')$ quando $\mathsf{a}' \neq \mathsf{a}$.

Uma outra alternativa força–bruta para o MOA seria a propagação de informação a respeito de todos os estados de objetos que precederam um determinado estado. O histórico de estados precedentes de um estado, $\theta(\sigma)$, pode ser definido como:

$$\theta(\sigma) = \{\sigma' : Act(\sigma') \prec Act(\sigma) \vee Act(\sigma) = Act(\sigma')\}$$

A precedência entre estados pode ser calculada como:

$$\sigma \prec \sigma' \equiv \theta(\sigma) \subset \theta(\sigma')$$

ou $\sigma \in \theta(\sigma')$ quando $Act(\sigma) \neq Act(\sigma')$.

A projeção do histórico de estados precedentes de um estado $\theta(\sigma)$ sobre um objeto o_i corresponde a um prefixo, possivelmente vazio, dos estados percorridos por este objeto ($\theta_i(\sigma) = \emptyset$ ou $\theta_i(\sigma) = \{\sigma_i^0 \dots \sigma_i^k\}$). Podemos definir um vetor de relógios para o MOA da seguinte maneira:

$$VC(\sigma)[i] = \begin{cases} 0 & \text{se, e somente se, } \theta_i(\sigma) = \emptyset \\ k \geq 0 & \text{se, e somente se, } \theta_i(\sigma) = \{\sigma_i^0 \dots \sigma_i^k\} \end{cases} \tag{3.2}$$

Como os estados inicias não são precedidos por nenhuma ação atômica e conseqüentemente por nenhum estado, os seus respectivos vetores de relógios possuem todas as entradas iguais a 0. O vetor de relógios associado a uma ação atômica é igual ao máximo por componente dos vetores de relógios dos estados lidos e dos estados promovidos por esta ação atômica.

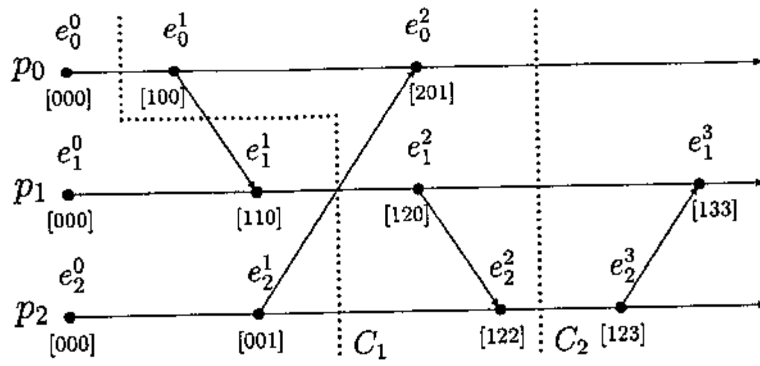


Figura 3.11: Cenário MPM com vetores de relógios

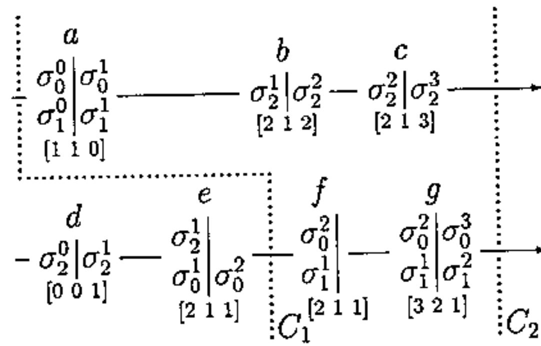


Figura 3.12: Cenário MOA com vetores de relógios

3.5.3 Implementação de Vetores de Relógios

Dado que os vetores de relógio no MPM e MOA foram definidos praticamente da mesma maneira, vamos apresentar as seguintes operações sobre vetores de relógios:

$$\begin{array}{rcl} VC \leftarrow max\{VC_i, VC_j\} & \equiv \forall k : 0 \leq k < n & : \quad VC \leftarrow max\{VC_i[k], VC_j[k]\} \\ VC_i \neq VC_j & \equiv \exists k : 0 \leq k < n & : \quad VC_i[k] \neq VC_j[k] \\ VC_i < VC_j & \equiv \forall k : 0 \leq k < n & : \quad VC_i[k] \leq VC_j[k] \wedge VC_i \neq VC_j \end{array}$$

A classe `VectorClock` (Classe 3.7) implementa a interface `Clock` para a propagação de vetores de relógios. As Figuras 3.11 e 3.12 reapresentam os cenários das Figuras 3.1 e 3.3 com os vetores de relógios associados a cada evento ou ação atômica entre colchetes.

Nas próximas subseções, vamos explorar as propriedades de vetores de relógios para o MPM e o MOA. Como foi feito para os relógios lógicos, dizemos que uma condição é *fraca* quando temos implicação e *forte* quando temos uma equivalência. Caso uma condição leve em consideração apenas as entradas dos vetores referentes aos processos ou objetos citados na propriedade, dizemos que é uma condição *simples*.

Classe 3.7 VectorClock.java

```
public class VectorClock implements Clock {

    public int[ ] v;

    public VectorClock() { for (int i=0; i < Application.N; i++) v[i] = 0; }
    protected VectorClock(VectorClock V) { this.v = (int[ ]) V.v.clone(); }

    public void increment(int pid) { v[pid]++; }

    public void increment(boolean[ ] W) {
        for (int i=0; i < Application.N; i++) if (W[i]) v[i]++;
    }

    public void maximum(Clock V) {
        for (int i=0; i < Application.N; i++)
            if ( ((VectorClock) V).v[i] > v[i]) v[i] = ((VectorClock) V).v[i];
    }

    public Clock propagate() { return new VectorClock(this); }
}
```

3.5.4 Propriedades de Vetores de Relógios no MPM

Várias informações podem ser extraídas de maneira simples dos vetores de relógios para o MPM, como podemos verificar através das propriedades listadas a seguir [8]. Cabe notar, que apesar de escritas considerando eventos, todas as propriedades são válidas para estados de processos.

Propriedade 3.8 Condição forte para precedência

Dados dois eventos e e e':

$$e \rightarrow e' \equiv VC(e) < VC(e')$$

A primeira propriedade garante que dados dois eventos e seus respectivos vetores de relógios é possível determinar se um deles precede causalmente o outro ou não. Caso se tenha conhecimento a respeito dos processos que executaram os eventos, essa detecção pode ser obtida com a comparação de dois inteiros (Propriedade 3.9).

Propriedade 3.9 Condição simples e forte para precedência

Dado o evento e_i do processo p_i e o evento e_j do processo p_j, com $i \neq j$

$$e_i \rightarrow e_j \equiv VC(e_i)[i] \leq VC(e_j)[i]$$

A partir dessas propriedades torna-se possível detectar se dois eventos são concorrentes, bastando para isso verificar a ausência de relações de precedência causal entre eles.

Propriedade 3.10 Condição forte para concorrência

Dado o evento e_i do processo p_i e o evento e_j do processo p_j

$$e_i \| e_j \equiv (VC(e_i)[i] > VC(e_j)[i]) \wedge (VC(e_j)[j] > VC(e_i)[j])$$

Propriedade 3.11 Condição forte para inconsistência

Evento e_i do processo p_i é inconsistente com um evento e_j do processo p_j, $i \neq j$, se, e somente se:

$$(VC(e_i)[i] < VC(e_j)[i]) \vee (VC(e_j)[j] < VC(e_i)[j])$$

Propriedade 3.12 Condição forte para corte consistente

Um corte definido por $(\iota_0, \ldots, \iota_{n-1})$ é consistente se, e somente se:

$$\forall i, j (0 \leq i, j < n) : VC(e_i^{\iota_i})[i] \geq VC(e_j^{\iota_j})[i]$$

Para todo $j \neq i$, o componente $VC(e_i)[j]$ indica o número de eventos de p_j que precedem causalmente e_i. $VC(e_i)[i]$ indica o número de eventos executados por p_i até e incluindo e_i. Seja $\#(e_i) = (\sum_{j=0}^{n-1} VC(e_i)[j]) - 1$ o número exato de eventos que precederam causalmente e_i na computação.

Propriedade 3.13 Contagem de eventos precedentes

Dado evento e_i do processo p_i e seu vetor de relógios $VC(e_i)$, o números de eventos e tais que $e \rightarrow e_i$ (equivalente a $VC(e) < VC(e_i)$) é dado por $\#(e_i)$.

Vetores de relógios fornecem uma forma fraca de detecção de *gaps* que pode ser utilizada para verificar se o *gap* causal de dois eventos admite um terceiro evento. Essa propriedade é fraca no sentido que para três processos p_i, p_j e p_k não se pode afirmar se três eventos desses processos formam uma cadeia causal $e_i \rightarrow e_k \rightarrow e_j$. Para o caso especial em que $i = k$ podemos detectar tal cadeia.

Propriedade 3.14 Detecção fraca de *gaps*

Dado o evento e_i do processo p_i e o evento e_j do processo p_j, se $VC(e_i)[k] < VC(e_j)[k]$ para algum $k \neq j$, então existe um evento e_k tal que

$$\neg(e_k \rightarrow e_i) \wedge (e_k \rightarrow e_j)$$

3.5.5 Propriedades de Vetores de Relógios no MOA

No MPM, estados distintos recebem vetores de relógios distintos. No MOA, entretanto, estados simultâneos recebem vetores de relógios iguais. Tal diferença se reflete sobre as propriedades de vetores de relógios no MOA.

Propriedade 3.15 Condição forte para simultaneidade

Dados dois estados de objetos σ e σ'

$$(\sigma \leftrightarrow \sigma') \equiv VC(\sigma) = VC(\sigma')$$

Propriedade 3.16 Condição simples e forte para simultaneidade
Dados o estado σ_i do objeto o_i e o estado σ_j do objeto o_j:

$$(\sigma_i \leftrightarrow \sigma_j) \equiv (VC(\sigma_i)[j] = VC(\sigma_j)[j]) \wedge (VC(\sigma_j)[i] = VC(\sigma_i)[i])$$

Se o vetor de relógios de uma ação atômica é menor que o de outra, é possível concluir que a primeira precedeu a última. O contrário não é sempre verdade. Na Figura 3.12 vemos que $e \prec f$ apesar delas possuírem vetores de relógios iguais.

Propriedade 3.17 Condição fraca para precedência (ações atômicas)
Dadas duas ações atômicas a *e* a'*:*

$$VC(\mathsf{a}) < VC(\mathsf{a}') \Rightarrow \mathsf{a} \prec \mathsf{a}'$$

Dados dois estados de objetos e seus respectivos vetores de relógios é possível determinar se um deles precede o outro ou não.

Propriedade 3.18 Condição forte para precedência (estados de objetos)
Dados dois estados de objetos σ e σ':

$$(\sigma \prec \sigma') \equiv VC(\sigma) < VC(\sigma')$$

Caso se tenha conhecimento acerca dos objetos aos quais os estados pertencem, o teste de causalidade pode ser simplificado.

Propriedade 3.19 Condição simples e forte para precedência (estados de objetos)
Considere o estado σ_i do objeto o_i e o estado σ_j do objeto o_j:

$$(\sigma_i \prec \sigma_j) \equiv (VC(\sigma_i)[i] \leq VC(\sigma_j)[i]) \wedge (VC(\sigma_i)[j] \neq VC(\sigma_j)[j])$$

A detecção da concorrência entre ações atômicas é fraca, como conseqüência da Propriedade 3.17. Duas ações atômicas podem possuir o mesmo vetor de relógios e serem concorrentes. Basta que elas tenham visto o mesmo conjunto de estados de objetos e não pertençam a uma mesma estrutura de ações seqüencial.

Propriedade 3.20 Condição fraca para concorrência (ações atômicas)
Dadas duas ações atômicas distintas a *e* a'*:*

$$\neg(VC(\mathsf{a}) < VC(\mathsf{a}')) \wedge \neg(VC(\mathsf{a}') < VC(\mathsf{a})) \Rightarrow \mathsf{a} \,||\, \mathsf{a}'$$

Propriedade 3.21 Condição forte para concorrência (estados de objetos)
Dados dois estados de objetos σ e σ':

$$(\sigma || \sigma') \equiv \neg(VC(\sigma) < VC(\sigma')) \wedge \neg(VC(\sigma') < VC(\sigma))$$

Propriedade 3.22 Condição simples e forte para concorrência (estados de objeto)
Dados dois estados de objetos σ_i e σ_j:

$$(\sigma_i || \sigma_j) \equiv (VC(\sigma_i)[j] < VC(\sigma_j)[j]) \wedge (VC(\sigma_j)[i] < VC(\sigma_i)[i])$$

Propriedade 3.23 Condição simples para inconsistência
Dois estados de objeto σ_i e σ_j, com $i \neq j$, são inconsistentes se, e somente se:

$$(VC(\sigma_i)[i] < VC(\sigma_j)[i]) \vee (VC(\sigma_j)[j] < VC(\sigma_i)[j])$$

Propriedade 3.24 Estado Global Consistente
Um estado global definido por $\Sigma = (\sigma_0^{t_0}, \sigma_1^{t_1}, \ldots, \sigma_{n-1}^{t_{n-1}})$ é consistente se, e somente se:

$$\forall i, j (0 \leq i, j < n) : VC(\sigma_i^{t_i})[i] \geq VC(\sigma_j^{t_j})[i]$$

Para todo $j \neq i$, o componente $VC(\sigma_i)[j]$ indica o número de estados de o_j que precedem causalmente σ_i. O componente $VC(\sigma_i)[i]$ indica o número de estados percorridos por o_i até e incluindo σ_i. Seja $\#(\sigma_i) = (\sum_{j=0}^{n-1} VC(\sigma_i)[j]) - 1$ o número exato de estados que precederam σ_i na computação. Não é possível, através do uso de vetores de relógios, a contagem do número de ações atômicas que precederam um determinado estado.

Propriedade 3.25 Contagem de estados precedentes
Dado um estado de objeto σ_i, o número de estados σ tais que $\sigma \prec \sigma_i$ (equivalente a $VC(\sigma) < VC(\sigma_i)$) é dado por $\#(\sigma_i)$.

Propriedade 3.26 Detecção fraca de *gaps*
Dado o estado de objeto σ_i e o estado de objeto σ_j, se $VC(\sigma_i)[k] < VC(\sigma_j)[k]$ para algum $k \neq j$, então existe um estado de objeto σ_k tal que

$$\neg(\sigma_k \prec \sigma_i) \wedge (\sigma_k \prec \sigma_j)$$

Como no MPM, essa forma de detecção de *gaps* é fraca pois dados três estados σ_i, σ_j, σ_k de três objetos quaisquer, não é possível determinar se eles formam uma cadeia de precedência: $\sigma_i \prec \sigma_j \prec \sigma_k$. Se $i = k$, essa detecção é possível.

3.6 Visão Progressiva

Dado que o fotógrafo possui um estado global consistente Σ da aplicação e os estados sucessores a Σ estão presentes nos canais FIFO (Figura 1.1, página 2) que ligam o fotógrafo aos componentes da aplicação, gostaríamos de montar um novo estado global consistente. Para facilitar a apresentação dos algoritmos e provas de correção, vamos introduzir uma notação auxiliar (Definição 3.11).

Definição 3.11 *Seja $\Sigma = \{\sigma_0^{t_0}, \ldots, \sigma_{n-1}^{t_{n-1}}\}$ um estado global consistente. Definimos os seguintes conjuntos de estados relacionados a Σ.*

- *$Channel(\Sigma) = \{\sigma_0^{t_0+1}, \ldots, \sigma_{n-1}^{t_{n-1}+1}\}$—O conjunto de estados sucessores aos estados em Σ. Alguns estados de processos ou objetos podem não estar presentes em $Channel(\Sigma)$ caso seu respectivo estado em Σ seja um estado final.*

- *Step*(Σ)—*Um subconjunto de estados em Channel*(Σ) *que pode ser substituído em* Σ *de maneira a formar um novo estado global consistente.*
- *Next*(Σ)—*Estado global consistente formado pela substituição de Step*(Σ) *em* Σ.

A construção progressiva de estados globais consistentes requer conhecimento acerca da precedência entre estados. O Teorema 3.4 sugere uma regra para a escolha de *Step*(Σ).

Teorema 3.4 *Seja* Σ *um estado global consistente e Channel*(Σ) *o conjunto de estados sucessores a* Σ. *Podemos definir Step*(Σ) *da seguinte maneira:*

$$Step(\Sigma) = \{\sigma_i^{t_i+1} \in Channel(\Sigma) : \sigma_j^{t_j+1} \in Channel(\Sigma) \Rightarrow \sigma_j^{t_j+1} \not\prec \sigma_i^{t_i+1}\}$$

Next(Σ), *formado pela substituição de Step*(Σ) *em* Σ, *é um estado global consistente.*

Prova: Suponha que *Next*(Σ) é inconsistente e existe um par de estados σ_a^α e σ_b^β pertencentes a *Next*(Σ) tal que

$$(\sigma_a^{\alpha+1} \prec \sigma_b^\beta) \vee (\sigma_a^{\alpha+1} \leftrightarrow \sigma_b^\beta)_{\text{MOA}}$$

Existem quatro alternativas para a origem de σ_a^α e σ_b^β:

- $\sigma_a^\alpha \in \Sigma, \sigma_b^\beta \in \Sigma$—Contraria a hipótese de Σ ser um estado global consistente.
- $\sigma_a^\alpha \in \Sigma, \sigma_b^\beta \in Step(\Sigma)$—Temos duas possibilidades:
  - $(\sigma_a^{\alpha+1} \prec \sigma_b^\beta)_{\text{MPM e MOA}}$: Contraria a regra de formação de *Step*(Σ);
  - $(\sigma_a^{\alpha+1} \leftrightarrow \sigma_b^\beta)_{\text{MOA}}$: O conjunto de estados precedentes de $\sigma_a^{\alpha+1}$ e σ_b^β é exatamente o mesmo, e não faz sentido que σ_b^β esteja em *Step*(Σ) e $\sigma_a^{\alpha+1}$ não.
- $\sigma_a^\alpha \in Step(\Sigma), \sigma_b^\beta \in \Sigma$—$(\sigma_a^\alpha \prec \sigma_b^\beta)$ e $\sigma_a^{\alpha-1}$ seria inconsistente com σ_b^β em Σ.
- $\sigma_a^\alpha \in Step(\Sigma), \sigma_b^\beta \in Step(\Sigma)$—$(\sigma_a^\alpha \prec \sigma_b^\beta)$, o que contraria a regra de formação de *Step*(Σ). □

Um subconjunto de estados que respeita a regra expressa no Teorema pode ser utilizado para a substituição em Σ. Não pode haver, no entanto, a separação de estados simultâneos.

Vamos explorar a construção progressiva de estados globais consistentes utilizando relógios lógicos e vetores de relógios.

Classe 3.8 LC_Photographer.java

```
public class LC_Photographer {

    public class LC_State {
        public int lc;
        // Process or object state
    }

    static protected void newConsistentGlobalState(LC_State[ ] C, // Consistent global state
                                                   LC_State[ ] S) // Succeeding states of C
    {
        int min = S[0].lc;
        for (int i=1; i < Application.N; i++)
            if (S[i].lc < min) min = S[i].lc;

        for (int i=0; i < Application.N; i++)
            if (S[i].lc == min) { C[i] = S[i]; S[i] = null; }
    }
}
```

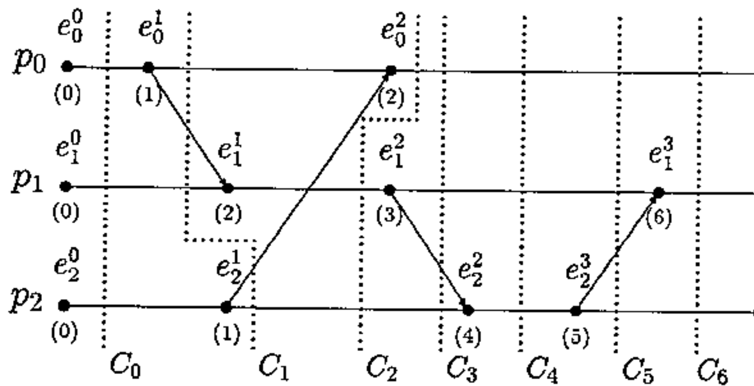


Figura 3.13: Visão progressiva utilizando relógios lógicos

Construção Progressiva Utilizando Relógios Lógicos

Os estados que possuem valor mínimo de relógio lógico em *Channel*(Σ) respeitam a regra expressa no Teorema 3.4. Esta propriedade é a base do algoritmo apresentado na classe `LC_Photographer` (Classe 3.8) para a construção progressiva de estados globais consistentes utilizando relógios lógicos. Para simplificar o código, neste e nos próximos algoritmos desenvolvidos para o fotógrafo, consideramos que o canal está sempre completo e não verificamos se um estado em Σ é um estado final.

Classe **3.9** VC_Photographer.java

```
public class VC_Photographer {

    public class VC_State {
        public VectorClock VC;
        public int id;
        // Process or object state

        public boolean precedes(VC_State state)
            { return (state.VC.v[id] ≤ VC.v[id] && state.VC.v[state.id] ≠ VC.v[state.id]); }
    }

    static protected void newConsistentGlobalState(VC_State[ ] C, // Consistent global snapshot
                                                   VC_State[ ] S) // Succeeding states of C
    {
        boolean[ ] M = new boolean[Application.N];

        for (int i=0; i < Application.N; i++) {
            M[i] = false;
            for (int j = 0; !M[i] && j < Application.N; j++)
                M[i] = (i ≠ j) && S[j].precedes(S[i]);
        }

        for (int i=0; i < Application.N; i++)
            if (!M[i]) { C[i] = S[i]; S[i] = null; }
    }
}
```

A Figura 3.13 reapresenta o cenário da Figura 3.8 (página 30) mostrando quais seriam os cortes construídos pelo fotógrafo utilizando o algoritmo proposto.

Construção Progressiva Utilizando Vetores de Relógios

Vetores de relógios caracterizam a precedência entre estados, o que permite a utilização da regra expressa no Teorema 3.4 para a escolha de $Step(\Sigma)$. A classe `VC_Photographer` (Classe 3.9) apresenta um algoritmo para a construção progressiva de estados globais consistentes utilizando vetores de relógios. Marcamos todos os estados em S que são precedidos por outros estados em S e substituímos em C os estados não marcados. A condição expressa para a verificação da precedência entre estados na classe `VC_State` é válida para o MPM e MOA.

A Figura 3.14 reapresenta o cenário da Figura 3.11 (página 35) mostrando quais seriam os cortes construídos pelo fotógrafo utilizando o algoritmo proposto. Podemos observar que os cortes apresentados são os mesmos que foram construídos utilizando relógios lógicos. Isso

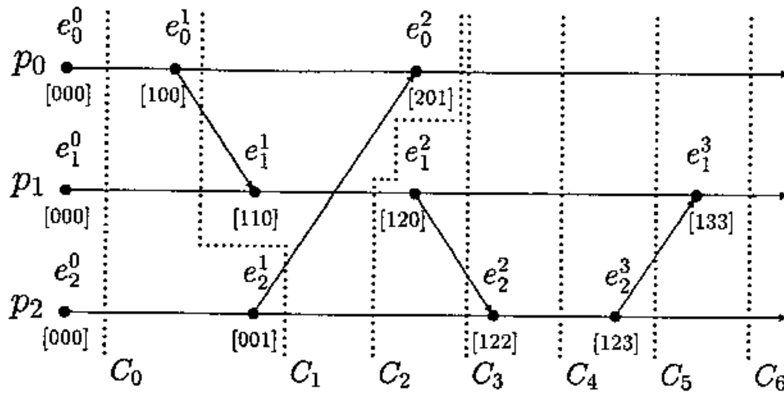


Figura 3.14: Visão progressiva utilizando vetores de relógios

MPM	MOA
execução de processos	estrutura de ações
estados de processos	estados de objetos
eventos	ações atômicas
precedência entre eventos	precedência entre ações

Tabela 3.1: Correlações entre MPM e MOA

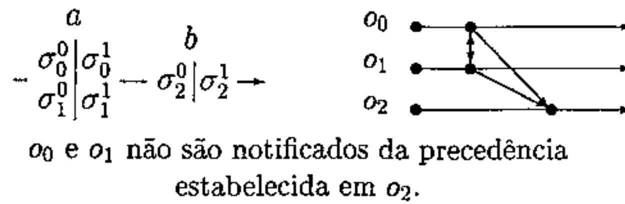
sempre acontece, desde que o fotógrafo inicie sua visão com o estado global inicial e utilize os algoritmos descritos sempre que o canal estiver completo. Com vetores de relógios é possível, no entanto, verificar se um estado é ou não precedido por outros do canal, mesmo que o canal não esteja completo.

3.7 Mapeamento entre MPM e MOA

A Tabela 3.1 mostra as correlações que encontramos entre o MPM e o MOA baseando-nos nas relações de precedência e consistência apresentadas. O mapeamento de algoritmos, no entanto, requer uma análise cuidadosa das diferenças entre os dois modelos.

Uma mensagem m estabelece uma relação de precedência imediata (sem considerar transitividade) entre exatamente dois eventos de processos distintos. Em contrapartida, uma ação atômica a pode estabelecer relações de precedência imediatas entre vários pares de estados de objetos. Tais estados não estão restritos aos participantes de a—podem ser estados que participaram em ações atômicas que precederam a estruturalmente.

Dessa forma, o conjunto de objetos para os quais um estado σ_i estabeleceu uma relação de precedência imediata não pode ser determinado em $Act(\sigma_i)$. Na Figura 3.15 podemos



o_0 e o_1 não são notificados da precedência estabelecida em o_2.

Figura 3.15: Precedência estrutural no MOA

observar o estabelecimento das relações de precedência imediatas ($\sigma_0^1 \prec \sigma_2^1$) e ($\sigma_1^1 \prec \sigma_2^1$), sem que o_0 ou o_1 tenham como prever o estabelecimento dessas relações.

Como conseqüência desse fato, se um algoritmo precisa saber qual é o destinatário de uma mensagem, então ele não pode ser mapeado. Como exemplos de protocolos não mapeáveis, temos os protocolos para seleção de *checkpoints* propostos por Xu e Netzer [28] e Baldoni et al. [1].

Temos um problema análogo, que é a determinação do remetente de uma mensagem. Um único estado no MOA pode ser precedido imediatamente por vários outros estados (Figura 3.15, estado σ_2^1), o que indica que o paralelo ao remetente de uma mensagem pode ser um conjunto de objetos. Apesar de poder alterar a complexidade de alguns passos de algoritmos de $O(1)$ para $O(n)$, acreditamos que esse problema é contornável na totalidade dos casos. Um exemplo de protocolo para seleção de *checkpoints* que precisa saber o destinatário de uma mensagem é outro protocolo proposto por Baldoni et al. [3].

3.8 Sumário

Neste Capítulo, estabelecemos um mapeamento entre o MPM e o MOA através da exploração dos conceitos de precedência e consistência em aplicações distribuídas construídas sobre estes dois modelos.

Iniciamos com a apresentação da relação de precedência entre eventos proposta por Lamport [13]. Baseados nesta relação, vimos como verificar de maneira teórica e gráfica a consistência de um estado global no MPM.

Definimos uma relação para precedência entre ações atômicas análoga à relação de precedência entre eventos e mostramos como verificar teoricamente a consistência de um estado global no MOA. Para a verificação gráfica desta consistência, propusemos um diagrama, denominado objeto–tempo, similar ao diagrama espaço–tempo para o MPM.

Chamamos de relógio qualquer informação de precedência mantida e propagada pelos componentes da aplicação. A relação de precedência no MPM coincide com o fluxo de informação entre os processos, o que permite a superposição da propagação de relógios sobre

as mensagens da aplicação [8]. No MOA, é a relação de precedência entre estados de objetos é possível mesmo na ausência de um fluxo de informação direto entre os objetos. Para preencher esta lacuna é utilizada a colaboração do gerente de ações do objeto [11]. O gerente acumula informações das ações que obtiveram *lock* de leitura sobre o objeto gerenciado e as incorpora ao estado do objeto no próximo *lock* de escrita.

Estudamos dois tipos de relógios: relógios lógicos e vetores de relógios. Relógios lógicos são consistentes com a precedência, mas não a caracterizam; vetores de relógios são suficientes para caracterizar a relação de precedência. Avaliamos as propriedades apresentadas por estes relógios para os dois modelos.

Com a incorporação dos relógios aos estados dos componentes da aplicação, apresentamos algoritmos para o fotógrafo montar estados globais consistentes para enviar ao monitor. O grande inconveniente desta abordagem é o grande número de estados que são transmitidos. Gostaríamos de poder ser seletivos durante a transmissão de estados dos componentes da aplicação para o fotógrafo, de maneira a economizar a construção de estados globais que não são de interesse para o monitor. Este é o tema do próximo Capítulo.

Concluímos este Capítulo com a apresentação das correlações encontradas entre o MPM e o MOA. De maneira geral, podemos fazer um paralelo entre processos e objetos e entre eventos e ações atômicas, mas o mapeamento de algoritmos exige uma análise cuidadosa dos dois modelos. O fato de uma mensagem conectar exatamente dois processos, enquanto uma ação atômica envolve um conjunto de objetos, pode alterar a complexidade de algoritmos que precisam saber quais são os rementententes das mensagens recebidas. Além disso, a semântica de uma aplicação no MPM está representada pela execução seqüencial dos processos, enquanto as estruturas de ações atômicas formam um grafo que pode tocar um conjunto imprevisível de objetos. Este último fato faz com que algoritmos que precisam saber os destinatários das mensagens enviadas não possam ser mapeados para o MOA. No próximo Capítulo, continuaremos explorando o mapeamento entre o MPM e o MOA para algoritmos que utilizam estados selecionados (*checkpoints*) de processos ou objetos.

Capítulo 4

Checkpoints Globais Consistentes

A utilização de todos os estados de processos ou objetos da aplicação para monitorização pode ter custo proibitivo. Uma solução alternativa consiste na seleção de estados de interesse (*checkpoints*) para envio ao fotógrafo (Figura 4.1). A escolha desses estados é ortogonal à computação da aplicação. Por exemplo, uma escolha seletiva pode filtrar estados em que um predicado local se tornou válido.

Iniciamos este Capítulo com a definição de *checkpoint* para o MPM e o MOA. Analisamos as condições necessárias e suficientes para a participação de *checkpoints* em *checkpoints* globais consistentes [18], ou seja, as condições para se estabelecer a *utilidade* de um *checkpoint*.

Protocolos quase–síncronos [17] permitem a seleção livre de *checkpoints* pela aplicação, mas podem requisitar *checkpoints* adicionais para diminuir ou eliminar a ocorrência de *checkpoints* inúteis. Apresentamos um mecanimo genérico para a manutenção e propagação de informação de controle para protocolos quase–síncronos no MPM e no MOA. Apresentamos uma classificação para protocolos quase–síncronos [17] e protocolos que a ilustram para o MPM e para o MOA.

Propomos algoritmos para a construção progressiva de *checkpoints* globais consistentes, de acordo com a classe de protocolo considerada. Apresentamos uma equivalência entre a possibilidade de construção de uma visão progressiva da aplicação e a ausência de *checkpoints* inúteis.

4.1 Estados e *Checkpoints*

Checkpoints são estados dos componentes da aplicação que formam uma *abstração* da computação feita por estes componentes. Devem ser selecionados em eventos ou ações atômicas independentes da computação da aplicação e implicam em um incremento do índice correspondente à enumeração de estados percorridos pelo componente ao qual pertence o *checkpoint*. Um *checkpoint* é representado por $\hat{\sigma}_a^\alpha$ e está relacionado a algum estado ordinário $\sigma_a^{\alpha'}$,

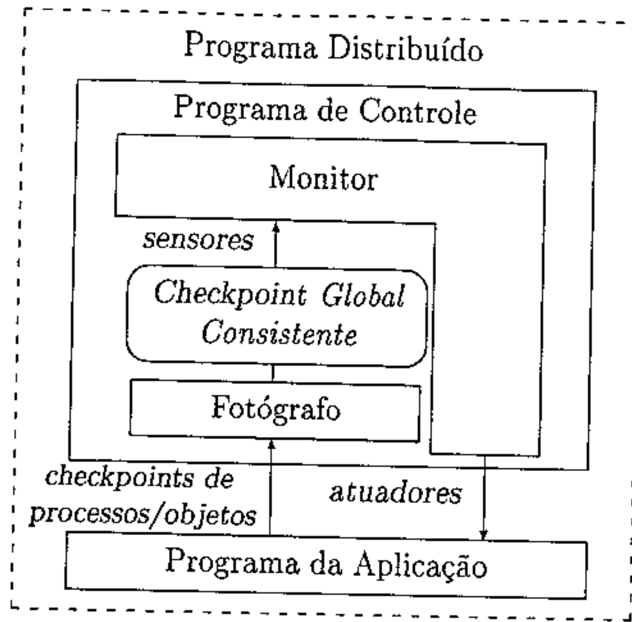


Figura 4.1: Monitorização utilizando *checkpoints*

com $\alpha \leq \alpha'$. Consideramos que os processos ou objetos selecionam um *checkpoint* inicial $\hat{\sigma}_a^0$ relativo ao estado σ_a^0 e um *checkpoint* final, ao terminarem sua execução.

Um par de *checkpoints* sucessivos $\hat{\sigma}_a^\alpha$ e $\hat{\sigma}_a^{\alpha+1}$ define um intervalo de *checkpoints* Δ_a^α que contém todos os estados percorridos por p_a ou o_a entre $\hat{\sigma}_a^\alpha$ e $\hat{\sigma}_a^{\alpha+1}$, incluindo $\hat{\sigma}_a^\alpha$ e excluindo $\hat{\sigma}_a^{\alpha+1}$. As Figuras 4.2 e 4.3 apresentam cenários no MPM e no MOA com *checkpoints* e intervalos de *checkpoints* demarcados (*checkpoints* são representados por quadrados preenchidos).

No MPM, os *checkpoints* são tradicionalmente executados pelos processos em eventos especiais [1, 17, 18, 26]. No MOA, consideramos que o gerente de ações se encarrega de executar as tarefas devidas para que um estado ordinário venha a ser um *checkpoint* de maneira completamente ortogonal à computação da aplicação. A promoção de um estado para um *checkpoint* ocorre como se tivesse havido uma ação atômica independente da estrutura de ações da aplicação que obteve um *lock* de escrita sobre este estado. Utilizamos $Act(\hat{\sigma})$ para referenciar a *pseudo-ação* que gerou o *checkpoint* $\hat{\sigma}$.

As relações de precedência e concorrência entre *checkpoints* estão atreladas à relações entre os eventos ou as ações atômicas que os geraram. Dizemos que $\hat{\sigma} \prec \hat{\sigma}$ ou $\hat{\sigma} || \hat{\sigma}'$ se o evento ou a ação atômica que originou $\hat{\sigma}$ for, respectivamente, precedente ou concorrente ao evento ou à ação atômica que originou $\hat{\sigma}'$. Como as pseudo-ações atômicas que geram os *checkpoints* operam sobre apenas um objeto, não existem *checkpoints* simultâneos.

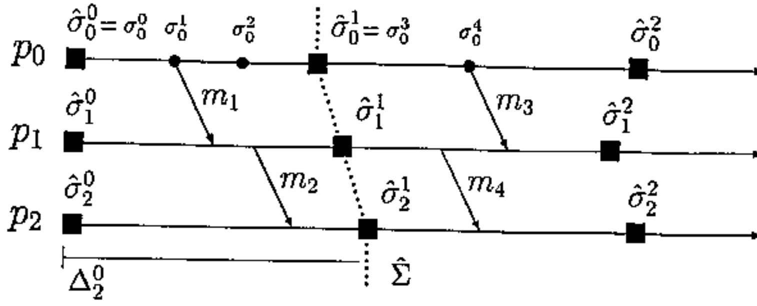


Figura 4.2: *Checkpoints* e intervalos de *checkpoints* no MPM

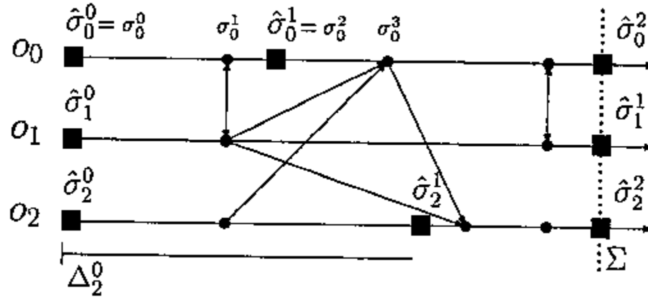


Figura 4.3: *Checkpoints* e intervalos de *checkpoints* no MOA

Dois *checkpoints* são consistentes entre si se, e somente se, são concorrentes (Definição 4.1). No MPM, a presença de uma relação de precedência entre dois *checkpoints* de processos distintos $\hat{\sigma}_a^\alpha$ e $\hat{\sigma}_b^\beta$ ($\hat{\sigma}_a^\alpha \prec \hat{\sigma}_b^\beta$) indica que houve uma caminho causal de mensagens ligando $\hat{\sigma}_a^\alpha$ a $\hat{\sigma}_b^\beta$ e que o evento de envio da primeira mensagem deste caminho ocorreu após $\hat{\sigma}_a^\alpha$. Na Figura 4.2, podemos observar que $\hat{\sigma}_0^0 \prec \hat{\sigma}_2^1$ devido ao caminho causal (m_1, m_2) e estes *checkpoints* são inconsistentes entre si. Considere uma relação de precedência entre dois *checkpoints* de objetos distintos $\hat{\sigma}_a^\alpha$ e $\hat{\sigma}_b^\beta$ ($\hat{\sigma}_a^\alpha \prec \hat{\sigma}_b^\beta$) no MOA. Neste caso, deve existir uma ação atômica a que promoveu o estado referente ao *checkpoint* $\hat{\sigma}_a^\alpha$ e que precede $\hat{\sigma}_b^\beta$. Os *checkpoints* $\hat{\sigma}_a^\alpha$ e $\hat{\sigma}_b^\beta$ são, portanto, inconsistentes. Na Figura 4.3, $\hat{\sigma}_0^1 \prec \hat{\sigma}_2^2$ e a ação atômica que gerou o estado σ_0^3 precede a pseudo-ação atômica que gerou o *checkpoint* $\hat{\sigma}_2^2$.

Definição 4.1 Inconsistência entre *checkpoints* (MPM e MOA)
Dois checkpoints $\hat{\sigma}$ e $\hat{\sigma}'$ são inconsistentes entre si se, e somente se:

$$(\hat{\sigma} \prec \hat{\sigma}') \vee (\hat{\sigma}' \prec \hat{\sigma})$$

Um *checkpoint* global consistente não possui nenhum par de *checkpoints* inconsistentes entre si. A Definição 4.2 utiliza a Definição 4.1 para determinar a consistência de um *checkpoint* global.

Definição 4.2 *Checkpoint* **Global Consistente**
Um checkpoint global $\hat{\Sigma} = (\hat{\sigma}_0^{\iota_0}, \ldots, \hat{\sigma}_{n-1}^{\iota_{n-1}})$ *é consistente se, e somente se:*
$$\forall i, j (0 \leq i, j < n) : (\hat{\sigma}_i^{\iota_i} \not\prec \hat{\sigma}_j^{\iota_j})$$

Apesar de necessária, a concorrência entre dois *checkpoints* não é condição suficiente para que eles possam participar de um mesmo *checkpoint* global consistente [18]. Na Figura 4.2, observamos que $\hat{\sigma}_0^1 \| \hat{\sigma}_2^1$ e eles participam do *checkpoint* global $\hat{\Sigma}$. Da mesma forma, $\hat{\sigma}_0^1 \| \hat{\sigma}_2^2$, mas não há nenhum *checkpoint* global que contenha ambos, pois $\hat{\sigma}_0^1 \prec \hat{\sigma}_1^2$ e $\hat{\sigma}_1^1 \prec \hat{\sigma}_2^2$. Na Figura 4.3, $\hat{\sigma}_2^1$ e $\hat{\sigma}_0^1$ são concorrentes, mas não existe *checkpoint* em o_1 que forme um *checkpoint* global consistente com $\hat{\sigma}_0^1$.

4.2 Caminhos e Ciclos em Zigzag

A especificação de uma condição necessária e suficiente para a participação de um conjunto de *checkpoints* em um *checkpoint* global consistente no MPM é atribuída a Netzer e Xu [18]. Esta condição é descrita a partir de *caminhos em zigzag* ou *caminhos-Z.*

Definição 4.3 Caminho-Z
Existe um caminho em zigzag (caminho-Z) de $\hat{\sigma}_a^\alpha$ *para* $\hat{\sigma}_b^\beta$ *se, e somente se,*

1. $a = b$ *e* $\alpha < \beta$ *ou*
2. *existe uma seqüência de mensagens* $m_1, m_2, \ldots, m_p$ $(p \geq 1)$ *tal que*
   (a) m_1 *é enviada por* p_a *após* $\hat{\sigma}_a^\alpha$*,*
   (b) *se* $m_k (1 \leq k < p)$ *é recebida por* p_r*, então* m_{k+1} *é enviada por* p_r *nesse mesmo intervalo de checkpoints ou em algum posterior a este (embora* m_{k+1} *possa ter sido enviada antes ou depois de* m_k *ser recebida) e*
   (c) m_p *é recebida por* p_b *antes de* $\hat{\sigma}_b^\beta$*.*

A definição original de Netzer e Xu [18] não contempla o item (1). Com o acréscimo dessa condição [16, 17], a definição torna-se uma generalização da relação de precedência causal proposta por Lamport [13]. Na Figura 4.2 podemos observar vários exemplos de caminhos-Z, entre eles: (m_1, m_2) ligando $\hat{\sigma}_0^0$ a $\hat{\sigma}_2^1$ e (m_3, m_4) ligando $\hat{\sigma}_0^1$ a $\hat{\sigma}_2^2$.

A Definição 4.3 se apóia sobre o conceito de mensagens, o que dificulta seu mapeamento para o MOA. Definimos uma relação equivalente entre *checkpoints* (Teorema 4.1) que pode ser aplicada diretamente ao MOA. Denominamos essa relação de *relação* $\mathcal{Z}$ e a representamos por $(\hat{\sigma} \rightsquigarrow \hat{\sigma}')$.

Definição 4.4 Relação $\mathcal{Z}$ *(MPM e MOA)*
$(\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta)$ se, e somente se,

1. $(\hat{\sigma}_a^\alpha \prec \hat{\sigma}_b^\beta)$, *ou*

2. $\exists \hat{\sigma}_c^\gamma : (\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_c^\gamma) \wedge (\hat{\sigma}_c^{\gamma-1} \rightsquigarrow \hat{\sigma}_b^\beta)$.

Teorema 4.1 *Existe um caminho–Z entre $\hat{\sigma}_a^\alpha$ e $\hat{\sigma}_b^\beta$ se, e somente se, $(\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta)$.*

Prova: ($\Rightarrow$) O item 1 da Definição 4.3 está contido no item 1 da Definição 4.4. Provamos, utilizando indução no número p de mensagens do caminho–Z, que o item 2 da Definição 4.3 implica na validade da relação $(\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta)$.

Base: $(p = 1)$ m_1 é enviada pelo processo a após $\hat{\sigma}_a^\alpha$ e recebida pelo processo b antes de $\hat{\sigma}_b^\beta$, o que implica que $(\hat{\sigma}_a^\alpha \prec \hat{\sigma}_b^\beta)$ e, portanto, $(\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta)$.

Passo: Suponha que um caminho–Z de tamanho p $(m_1, \ldots, m_p)$ ligando $\hat{\sigma}_a^\alpha$ a $\hat{\sigma}_c^\gamma$ implica que $(\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_c^\gamma)$. Considere, sem perda de generalidade, que o evento *recepção*(m_p) ocorreu durante o intervalo $\Delta_c^{\gamma-1}$. O evento *envio*(m_{p+1}) deve ter ocorrido durante o intervalo $\Delta_c^{\gamma-1}$ ou algum intervalo posterior a este, o que implica que $(\hat{\sigma}_c^{\gamma-1} \prec \hat{\sigma}_b^\beta)$ e, portanto,$(\hat{\sigma}_c^{\gamma-1} \rightsquigarrow \hat{\sigma}_b^\beta)$. De acordo a Definição 4.3 as relações $(\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_c^\gamma)$ e $(\hat{\sigma}_c^{\gamma-1} \rightsquigarrow \hat{\sigma}_b^\beta)$ implicam que $(\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta)$.

($\Leftarrow$) Dada a relação $(\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta)$, podemos quebrá-la em uma série de relações causais entre *checkpoints* (Definição 4.4, item (2)):

$$\hat{\sigma}_a^\alpha = \hat{\sigma}_{i_0}^{\gamma_0-1} \quad (\hat{\sigma}_{i_0}^{\gamma_0-1} \prec \hat{\sigma}_{i_1}^{\gamma_1}), (\hat{\sigma}_{i_1}^{\gamma_1-1} \prec \hat{\sigma}_{i_2}^{\gamma_2}), \ldots, (\hat{\sigma}_{i_{p-1}}^{\gamma_{p-1}-1} \prec \hat{\sigma}_{i_p}^{\gamma_p}) \quad \hat{\sigma}_{i_p}^{\gamma_p} = \hat{\sigma}_b^\beta$$

Utilizando indução no número p de relações causais, vamos provar que existe um caminho–Z entre $\hat{\sigma}_a^\alpha$ e $\hat{\sigma}_b^\beta$.

Base: $(p = 1)$ Para todo caminho causal existe um caminho–Z, o que nos permite concluir que existe um caminho–Z entre $\hat{\sigma}_{i_0}^{\gamma_0-1}$ e $\hat{\sigma}_{i_1}^{\gamma_1}$.

Passo: $(p \geq 1)$ Suponha que existe um caminho–Z ligando $\hat{\sigma}_{i_0}^{\gamma_0-1}$ a $\hat{\sigma}_{i_p}^{\gamma_p}$ e que a última mensagem deste caminho foi recebida no intervalo $\Delta_{i_p}^{\gamma_p-1}$. Suponha que existe um caminho causal entre $\hat{\sigma}_{i_p}^{\gamma_p-1}$ e $\hat{\sigma}_{i_{p+1}}^{\gamma_{p+1}}$, o que implica que o processo p_p deve enviar a primeira mensagem deste caminho no intervalo de *checkpoints* $\Delta_{i_p}^{\gamma_p-1}$ ou em algum posterior a este. De acordo com a Definição 4.3, esses caminhos podem ser concatenados para formar um caminho–Z entre $\hat{\sigma}_{i_0}^{\gamma_0-1}$ e $\hat{\sigma}_{i_{p+1}}^{\gamma_{p+1}}$. □

A definição da relação $\mathcal{Z}$ entre *checkpoints* traz várias vantagens sobre a definição de caminhos–Z como expressa por Netzer e Xu [18] . A generalização da relação de precedência torna-se clara, bem como a maneira como é feita a composição da relação. Além disso, utilizamos *checkpoints* para obter uma abstração de nível mais alto da computação feita pela aplicação e consideramos que a observação das mensagens e os seus respectivos eventos

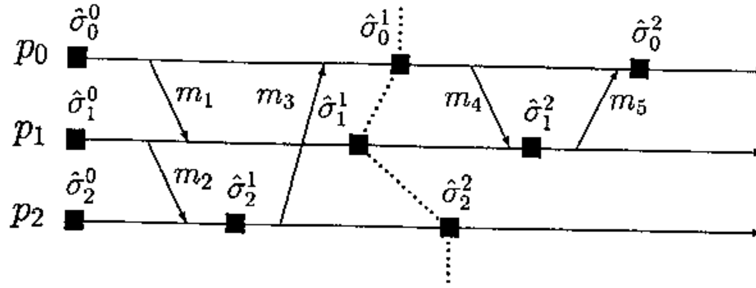


Figura 4.4: Ciclos–Z

de envio e recepção corresponde a uma abstração de nível mais baixo. Preferimos considerar apenas *checkpoints* e precedência entre *checkpoints*, sem nos preocuparmos como essa precedência foi estabelecida. Esta abordagem de nível mais alto permite a utilização desta definição, bem como dos resultados derivados a partir dela, em mais de um modelo computacional.

Definição 4.5 Relação $\mathcal{Z}$ não–precedente—*Dizemos que existe uma relação $\mathcal{Z}$ não–precedente entre $\hat{\sigma}$ e $\hat{\sigma}'$ se, e somente se, $(\hat{\sigma} \rightsquigarrow \hat{\sigma}') \wedge (\hat{\sigma} \not\prec \hat{\sigma}')$.*

Definição 4.6 Ciclo–Z—*Dizemos que um checkpoint $\hat{\sigma}$ está envolvido em um* ciclo em zigzag *(ciclo–Z) se, e somente se, $(\hat{\sigma} \rightsquigarrow \hat{\sigma})$.*

Obviamente, todo ciclo–Z é uma relação $\mathcal{Z}$ não–precedente. Na Figura 4.4 podemos observar que existe uma relação $\mathcal{Z}$ não–precedente entre $\hat{\sigma}_0^0$ e $\hat{\sigma}_2^1$. Além disso, temos dois exemplos de *checkpoints* envolvidos em um ciclo–Z: $\hat{\sigma}_2^1$ devido às mensagens (m_3, m_1, m_2) e $\hat{\sigma}_1^2$, devido às mensagens (m_5, m_4). Na Figura 4.3 podemos observar que $\hat{\sigma}_0^1$ está envolvido em um ciclo–Z.

Definição 4.7 *Seja $\hat{\sigma}$ um checkpoint e sejam R e S dois conjuntos de checkpoints. Definimos as seguintes relações:*

1. *$\hat{\sigma} \rightsquigarrow S$ se existe $\hat{\sigma}' \in S$ tal que $\hat{\sigma} \rightsquigarrow \hat{\sigma}'$;*
2. *$S \rightsquigarrow \hat{\sigma}$ se existe $\hat{\sigma}' \in S$ tal que $\hat{\sigma}' \rightsquigarrow \hat{\sigma}$;*
3. *$S \rightsquigarrow R$ se existe um par de checkpoints $\hat{\sigma}$ e $\hat{\sigma}'$, $\hat{\sigma} \in S$ e $\hat{\sigma}' \in R$, tal que $\hat{\sigma} \rightsquigarrow \hat{\sigma}'$;*
4. *$\hat{\sigma} \not\rightsquigarrow S$, $S \not\rightsquigarrow \hat{\sigma}$ e $S \not\rightsquigarrow R$ para os casos em que, respectivamente, $\neg(\hat{\sigma} \rightsquigarrow S)$, $\neg(S \rightsquigarrow \hat{\sigma})$ ou $\neg(S \rightsquigarrow R)$.*

Consideramos definições análogas para as relações $\prec$ e $\not\prec$.

4.3 Utilidade de um *Checkpoint*

Esta Seção coloca as condições necessárias e suficientes para se estender um conjunto de *checkpoints* S de maneira a formar um *checkpoint* global consistente $\hat{\Sigma}$ (Teorema 4.2). Esta extensão ocorre com a adição de *checkpoints* de processos ou objetos que não estejam em S. Caso S contenha um único *checkpoint* $\hat{\sigma}$ temos as condições necessárias e suficientes para a participação de $\hat{\sigma}$ em algum *checkpoint* global consistente (Colorário 4.3).

Teorema 4.2 *Um conjunto de checkpoints S pode ser estendido para formar um checkpoint global consistente se, e somente se, $S \not\rightsquigarrow S$.*

Prova: ($\Leftarrow$) *Suficiência:* Construímos um *checkpoint* global $\hat{\Sigma}$, considerando para todo processo ou objeto j que não tem *checkpoint* em S o *checkpoint* $\hat{\sigma}_j^{\iota_j}$ com

$$\iota_j = min\{\gamma : \hat{\sigma}_j^{\gamma} \not\rightsquigarrow S\}$$

ou $\iota_j = 0$ se esse conjunto for vazio. Cabe notar que caso $\iota_j > 0$, existe um checkpoint $\hat{\sigma}_c^{\gamma} \in S$ tal que $(\hat{\sigma}_j^{\iota_j - 1} \rightsquigarrow \hat{\sigma}_c^{\gamma})$.

Os *checkpoints* em $\hat{\Sigma}$ formam um estado global consistente. Para estabelecer uma contradição, vamos supor a existência de uma relação de precedência entre dois *checkpoints* $\hat{\sigma}_a^{\alpha}$ e $\hat{\sigma}_b^{\beta}$ em $\hat{\Sigma}$:

$$(\hat{\sigma}_a^{\alpha} \prec \hat{\sigma}_b^{\beta}) \Rightarrow (\hat{\sigma}_a^{\alpha} \rightsquigarrow \hat{\sigma}_b^{\beta})$$

Devemos notar que nesse caso temos $\beta > 0$, pois nenhum estado precede um estado inicial. Existem quatro possibilidades para a origem desses *checkpoints*:

1. $\hat{\sigma}_a^{\alpha} \in S$, $\hat{\sigma}_b^{\beta} \in S$: O fato de $(\hat{\sigma}_a^{\alpha} \rightsquigarrow \hat{\sigma}_b^{\beta})$ contradiz a hipótese $S \not\rightsquigarrow S$.

2. $\hat{\sigma}_a^{\alpha} \in S$, $\hat{\sigma}_b^{\beta} \notin S$: $\beta > 0$ e existe $\hat{\sigma}_c^{\gamma} \in S$ tal que $(\hat{\sigma}_b^{\beta-1} \rightsquigarrow \hat{\sigma}_c^{\gamma})$. Como $(\hat{\sigma}_a^{\alpha} \rightsquigarrow \hat{\sigma}_b^{\beta}) \wedge (\hat{\sigma}_b^{\beta-1} \rightsquigarrow \hat{\sigma}_c^{\gamma})$ implica que $(\hat{\sigma}_a^{\alpha} \rightsquigarrow \hat{\sigma}_c^{\gamma})$ e $\hat{\sigma}_a^{\alpha}, \hat{\sigma}_c^{\gamma} \in S$, temos uma contradição da hipótese $S \not\rightsquigarrow S$.

3. $\hat{\sigma}_a^{\alpha} \notin S$, $\hat{\sigma}_b^{\beta} \in S$: $(\hat{\sigma}_a^{\alpha} \rightsquigarrow \hat{\sigma}_b^{\beta})$ contraria as regras de formação de $\hat{\Sigma}$.

4. $\hat{\sigma}_a^{\alpha} \notin S$, $\hat{\sigma}_b^{\beta} \notin S$: Como no caso (2), $\beta > 0$ e existe $\hat{\sigma}_c^{\gamma} \in S$ tal que $(\hat{\sigma}_b^{\beta-1} \rightsquigarrow \hat{\sigma}_c^{\gamma})$. Como $(\hat{\sigma}_a^{\alpha} \rightsquigarrow \hat{\sigma}_b^{\beta}) \wedge (\hat{\sigma}_b^{\beta-1} \rightsquigarrow \hat{\sigma}_c^{\gamma})$ implica que $(\hat{\sigma}_a^{\alpha} \rightsquigarrow \hat{\sigma}_c^{\gamma})$, temos uma contradição em relação às regras de formação de $\hat{\Sigma}$.

($\Rightarrow$) *Necessidade:* Suponha que existe uma relação $(\hat{\sigma}_a^{\alpha} \rightsquigarrow \hat{\sigma}_b^{\beta})$ entre dois *checkpoints* distintos em S. Tal relação pode ser quebrada em uma série de relações de precedências entre pares de *checkpoints*:

$$\hat{\sigma}_a^{\alpha} = \hat{\sigma}_{i_0}^{\gamma_0 - 1} \quad (\hat{\sigma}_{i_0}^{\gamma_0 - 1} \prec \hat{\sigma}_{i_1}^{\gamma_1}), (\hat{\sigma}_{i_1}^{\gamma_1 - 1} \prec \hat{\sigma}_{i_2}^{\gamma_2}), \ldots, (\hat{\sigma}_{i_{p-1}}^{\gamma_{p-1} - 1} \prec \hat{\sigma}_{i_p}^{\gamma_p}) \quad \hat{\sigma}_{i_p}^{\gamma_p} = \hat{\sigma}_b^{\beta}$$

Utilizando indução no número p de precedências, vamos provar que $\hat{\sigma}_a^{\alpha}$ e $\hat{\sigma}_b^{\beta}$ não podem participar de um mesmo *checkpoint* global consistente.

Base: ($p = 1$) Como ($\hat{\sigma}_a^\alpha \prec \hat{\sigma}_b^\beta$), eles não podem participar de *nenhum checkpoint* global consistente.

Passo: ($p \geq 1$) Suponha que ($\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_{i_p}^{\gamma_p}$) implica que $\hat{\sigma}_a^\alpha$ não pode participar de um mesmo *checkpoint* global consistente com $\hat{\sigma}_{i_p}^{\gamma_p}$ ou qualquer *checkpoint* posterior a esse em p_p ou o_p. Da mesma forma, $\hat{\sigma}_b^\beta$ não pode participar de um mesmo *checkpoint* global consistente com nenhum *checkpoint* anterior ou correspondente a $\hat{\sigma}_{i_p}^{\gamma_p - 1}$. Portanto, não há *checkpoint* em p_p ou o_p que possa participar simultaneamente com $\hat{\sigma}_a^\alpha$ e $\hat{\sigma}_b^\beta$ em um mesmo *checkpoint* global consistente.

Suponha a existência de um ciclo–Z ($\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_a^\alpha$). Como um ciclo–Z é uma relação $\mathcal{Z}$ não–precedente, existem dois *checkpoints* $\hat{\sigma}_c^\gamma$ e $\hat{\sigma}_c^{\gamma+1}$ tais que ($\hat{\sigma}_c^\gamma \rightsquigarrow \hat{\sigma}_a^\alpha$) e ($\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_c^{\gamma+1}$). Pelo resultados obtidos nesta prova, concluímos que não há *checkpoint* em p_c ou o_c que possa participar com $\hat{\sigma}_a^\alpha$ em um *checkpoint* global consistente. □

Aplicando o Teorema 4.2 com $S = \{\hat{\sigma}\}$ obtemos um colorário bastante interessante, que indica a *utilidade* de um *checkpoint*, ou seja, a possibilidade dele poder ou não participar de algum *checkpoint* global consistente.

Colorário 4.3 *Checkpoint $\hat{\sigma}$ pode pertencer a algum checkpoint global consistente se, e somente se, não estiver envolvido em nenhum ciclo–Z.*

4.4 Protocolos Quase–Síncronos

Protocolos quase–síncronos permitem a seleção arbitrária de *checkpoints* pela aplicação (*checkpoints básicos*), mas podem induzir a seleção de *checkpoints* adicionais (*checkpoints forçados*) através da manutenção e propagação de informação de controle. Protocolos quase–síncronos podem reduzir ou eliminar a ocorrência de *checkpoints* inúteis.

A interface `QS_Clock` (Classe 4.1) descreve os métodos necessários para a implementação de protocolos quase–síncronos: incremento, máximo, comparação e propagação de informação de controle, além de uma condição para a indução de *checkpoints* forçados. A operação de incremento deve ser invocada toda vez que um *checkpoint* é selecionado e a informação de controle deve ser propagada de maneira semelhante aos relógios estudados no Capítulo 3.

A classe `QS_Process` (Classe 4.2) especializa a classe `Process` para implementação de protocolos quase–síncronos. Foram adicionados métodos para a seleção de *checkpoints* no início e no final da execução do processo. Além disso, a aplicação pode invocar o método `takeBasicCheckpoint` a qualquer momento. *Checkpoints* forçados podem ser induzidos no momento da recepção de uma mensagem, quando o método `mustTakeForcedCkpt` é invocado.

Interface 4.1 QS_Clock.java

```
public interface QS_Clock {

    /** Increments the clock based on the process or object id */
    void increment(int id);

    /** Keeps the maximum between the stored clock and the parameter given */
    void maximum(QS_Clock qs_clock);

    /** Compares two QS_Clock objects for equality. */
    boolean equals(QS_Clock qs_clock);

    /** Returns the clock to be propagated through the application */
    QS_Clock propagate();

    /** Returns if a forced checkpoint must be induced */
    boolean mustTakeForcedCkpt(int id, QS_Clock qs_clock);
}
```

As classes `QS_ActionManager` (Classe 4.3) e `QS_AtomicAction` (Classe 4.4) descrevem, respectivamente, extensões das classes `ActionManager` e `AtomicAction` para a implementação de protocolos quase–síncronos no MOA.

A seleção de *checkpoints* básicos no MOA é mais complexa que no MPM. Estados de objetos podem ser lidos concorrentemente por várias ações atômicas, que podem solicitar um *checkpoint* básico sobre o mesmo estado. Como solicitações feitas por ações atômicas abortadas não devem ser consideradas, a ação atômica acumula os pedidos de *checkpoints* básicos e os repassa aos gerentes de ações no momento do `commit`.

A solicitação de um *checkpoint* básico para um estado sobre o qual foi adquirido *lock* de escrita poderia ter interpretação múltipla: gravação do estado referente ao início da ação, ao momento da solicitação (embora possivelmente inconsistente) ou ao término da ação atômica. Para esse caso, colocamos dois métodos na classe `QS_AtomicAction`: `takeRBasicCkpt` e `takeWBasicCkpt`. O primeiro seleciona o estado lido e o segundo o estado alterado.

O gerente de ações armazena o pedido de um *checkpoint* básico em memória estável e só cuida do envio do estado ao fotógrafo quando um *lock* de escrita é obtido sobre o estado selecionado. Embora seja possível enviar o *checkpoint* ao fotógrafo no momento da requisição, optamos por adiar esse envio de maneira a apresentar uma implementação mais simples.

O protocolo de `commit` pode ser dividido em três fases:

1. Obtenção da informação de controle dos objetos participantes da ação:

    - Eventuais pedidos de *checkpoints* básicos feitos nessa ação são repassados aos gerentes de ações.

Classe 4.2 QS_Process.java

```
public class QS_Process extends Process {

    public class QS_Message extends Message {
        public QS_Clock qs_clock;
    }

    protected QS_Clock qs_clock;

    public QS_Process(int pid) { super(pid); takeCheckpoint(); }

    public void finalize() { takeCheckpoint(); }

    protected void takeCheckpoint() {
        qs_clock.increment(pid);
        // Send the current state to the photographer.
    }

    public void takeBasicCheckpoint() { takeCheckpoint(); }

    public void sendMessage(QS_Message m) {
        m.qs_clock = qs_clock.propagate();
        super.sendMessage(m);
    }

    public void receiveMessage(QS_Message m) {
        if (qs_clock.mustTakeForcedCkpt(pid, m.qs_clock))
            takeCheckpoint();
        qs_clock.maximum(m.qs_clock);
        super.receiveMessage(m);
    }
}
```

Classe 4.3 QS_ActionManager.java

```
public class QS_ActionManager extends ActionManager {

    protected QS_Clock objectClock;
    protected QS_Clock managerClock;
    public boolean ckptScheduled;

    public QS_ActionManager(AtomicObject obj) { super(obj); ckptScheduled = true; }

    public void finalize() {
        if (!ckptScheduled) objectClock.increment(obj.oid);
        takeCheckpoint();
    }

    protected void setCkptScheduled(boolean ckptScheduled) {
        this.ckptScheduled = ckptScheduled;
        // Save ckptScheduled on stable memory
    }

    public synchronized void takeBasicCheckpoint() { setCkptScheduled(true); }

    protected void takeCheckpoint() {
        // Send the object stable state to the photographer.
        setCkptScheduled (false);
    }

    public synchronized QS_Clock rClock() { return objectClock; }
    public synchronized QS_Clock wClock() {
        if (ckptScheduled) { objectClock.increment(obj.oid); managerClock.maximum(objectClock); }
        return managerClock;
    }

    public synchronized void ckptInduction(QS_Clock qs_clock) {
        if (!ckptScheduled && managerClock.mustTakeForcedCkpt(obj.oid,qs_clock)) {
            ckptScheduled = true;
            objectClock.increment(obj.oid);
            qs_clock.maximum(objectClock);
        }
    }

    public synchronized void updateRClock(QS_Clock qs_clock) { managerClock.maximum(qs_clock); }
    public synchronized void updateWClock(QS_Clock qs_clock) {
        if (ckptScheduled) takeCheckpoint();
        managerClock = qs_clock.propagate();
        objectClock = qs_clock.propagate();
    }
}
```

Classe 4.4 QS_AtomicAction.java

```
public class QS_AtomicAction extends AtomicAction {

    public QS_Clock qs_clock;
    protected boolean[ ] rBasicCkpt = new boolean [Application.N],
        wBasicCkpt = new boolean [Application.N];

    public void setClockAtBeginning(QS_Clock qs_clock) { this.qs_clock = qs_clock.propagate(); }

    public void begin() {
        for (int i=0; i < Application.N; i++) rBasicCkpt[i] = wBasicCkpt[i] = false;
        super.begin();
    }

    public void takeRBasicCkpt(AtomicObject obj) { rBasicCkpt[obj.oid] = true; }
    public void takeWBasicCkpt(AtomicObject obj) { wBasicCkpt[obj.oid] = true; }

    public void commit() {
        for (int i=0; i < nRLocked; i++) {
            if (rBasicCkpt[i]) ((QS_ActionManager) rLocked[i]).takeBasicCheckpoint();
            qs_clock.maximum(((QS_ActionManager) rLocked[i]).rClock());
        }

        for (int i=0; i < nWLocked; i++) {
            if (rBasicCkpt[i]) ((QS_ActionManager) wLocked[i]).takeBasicCheckpoint();
            qs_clock.maximum(((QS_ActionManager) wLocked[i]).wClock());
        }

        QS_Clock saveClock;
        do {
            saveClock = qs_clock.propagate();
            for (int i=0; i < nWLocked; i++)
                ((QS_ActionManager) wLocked[i]).ckptInduction(qs_clock);
        } while (! saveClock.equals(qs_clock));

        for (int i=0; i < nRLocked; i++)
            ((QS_ActionManager) rLocked[i]).updateRClock(qs_clock);
        for (int i=0; i < nWLocked; i++)
            ((QS_ActionManager) wLocked[i]).updateWClock(qs_clock);

        for (int i=0; i < nWLocked; i++) {
            if (wBasicCkpt[i]) ((QS_ActionManager) wLocked[i]).takeBasicCheckpoint();
        }

        super.commit();
    }

    public QS_Clock getClockAtCommit() { return qs_clock.propagate(); }
}
```

- Para os objetos sobre os quais foram adquiridos *locks* de leitura, os gerentes de ações retornam a informação de controle relativa ao estado lido.
- Os gerentes de ações referentes aos objetos participantes com *lock* de escrita enviam informação acumulada, com possível incremento devido à solicitação de um *checkpoint* básico.

2. Indução de *checkpoints* forçados:

   Dada a informação de controle coletada até o momento, deve-se verificar a necessidade da indução de um *checkpoint* forçado em algum objeto participante com *lock* de escrita na ação. Caso haja indução e alteração na informação de controle, todos os gerentes de ações referentes a esses objetos devem ser percorridos novamente. Apesar de ter complexidade quadrática, características inerentes a alguns protocolos podem tornar esse passo linear. Por exemplo, protocolos que garantem que um *checkpoint* forçado nunca induz outros *checkpoints* forçados necessitam que os gerentes de ações sejam percorridos apenas uma vez.

3. Propagação da informação de controle aos objetos:

   Uma vez estabelecida a informação de controle relativa a essa ação atômica, esta é propagada para todos os gerentes de ações dos objetos participantes da ação. Os gerentes de ações dos objetos sobre os quais foram adquiridos *locks* de escrita e que têm *checkpoints* selecionados executam o envio ao fotógrafo.

   Pedidos de *checkpoints* básicos sobre estados promovidos são propagados.

A classe `userQS_ClockClass` (Classe 4.5) apresenta um exemplo de utilização das classes apresentadas.

4.5 Classificação de Protocolos Quase–Síncronos

Um protocolo quase–síncrono define um *padrão de checkpoints* que pode ou não permitir o estabelecimento de relações $\mathcal{Z}$ não–precedentes ou ciclos–Z entre *checkpoints*. A classificação proposta por Manivannan e Singhal [17] para protocolos quase–síncronos no MPM baseia-se nestas características dos padrões. Considerando o mapeamento entre MPM e MOA (Seção 3.7) introduzimos um subconjunto desta classificação que é adequado ao dois modelos.

Classe 4.5 userQS_ClockClass.java

```
public class userQS_ClockClass {

    /** QS_Clock information is propagated through the methods invocations. */
    static public void userClockMethod(QS_Clock qs_clock, AtomicObject rObj, AtomicObject wObj) {
        QS_AtomicAction a = new QS_AtomicAction();
        a.setClockAtBeginning(qs_clock);
        a.begin();
        a.readLock(rObj);
        a.writeLock(wObj);
        /*...*/
        a.takeRBasicCkpt(rObj);
        a.takeRBasicCkpt(wObj);
        a.takeWBasicCkpt(wObj);
        a.commit();
        qs_clock = a.getClockAtCommit();
    }
}
```

4.5.1 Protocolos com Ausência de Relações $\mathcal{Z}$ Não–Precedentes (ZPF)

Definição 4.8 *Um padrão de checkpoints é dito* livre de relações $\mathcal{Z}$ não–precedentes (ZPF) *se, e somente se, para todo par de checkpoints $\hat{\sigma}$ e $\hat{\sigma}'$ neste padrão*

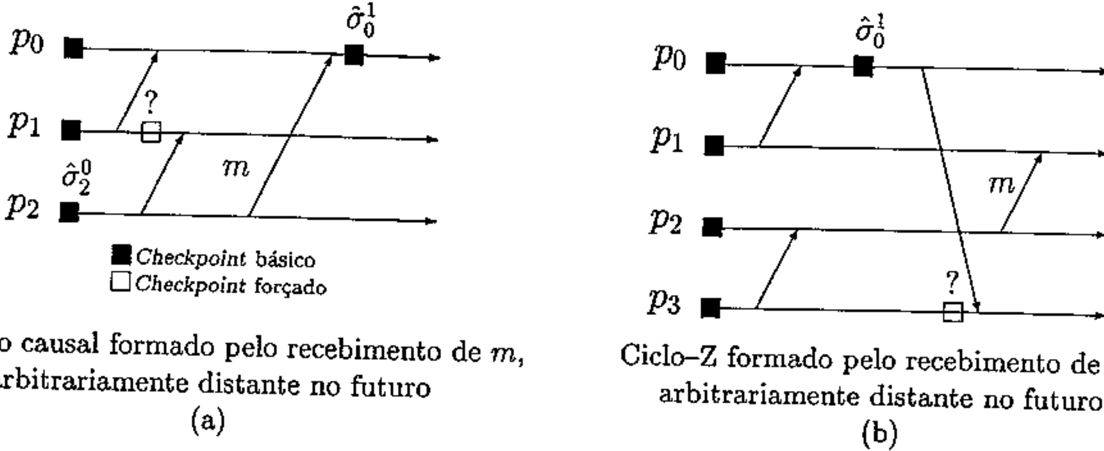
$$(\hat{\sigma} \rightsquigarrow \hat{\sigma}') \equiv (\hat{\sigma} \prec \hat{\sigma}')$$

Teorema 4.4 *Em um sistema ZPF, um conjunto S de checkpoints pode ser estendido para formar um checkpoint global consistente se, e somente se, $S \not\prec S$.*

Prova: Em um sistema ZPF, dados dois *checkpoints* $\hat{\sigma}$ e $\hat{\sigma}'$, $(\hat{\sigma} \prec \hat{\sigma}')$ se, e somente se, $(\hat{\sigma} \rightsquigarrow \hat{\sigma}')$. Dessa forma, para qualquer conjunto S de *checkpoints*, $S \not\prec S$ se, e somente se, $S \not\rightsquigarrow S$. Pelo Teorema 4.2 concluímos que S pode ser estendido para formar um *checkpoint* global consistente. □

Lema 4.1 *Em um sistema ZPF, todos os checkpoints são úteis para a construção de estados globais consistentes.*

Prova: Em um sistema ZPF, dados dois *checkpoints* $\hat{\sigma}$ e $\hat{\sigma}'$, $(\hat{\sigma} \prec \hat{\sigma}')$ se, e somente se, $(\hat{\sigma} \rightsquigarrow \hat{\sigma}')$. Como a relação de precedência não admite ciclos, a relação $(\hat{\sigma} \rightsquigarrow \hat{\sigma})$ não é permitida. Pelo Colorário 4.3 concluímos que todos os *checkpoints* são úteis. □



Caminho causal formado pelo recebimento de m, arbitrariamente distante no futuro
(a)

Ciclo–Z formado pelo recebimento de m, arbitrariamente distante no futuro
(b)

Figura 4.5: Dificuldade de se projetar protocolos ótimos

Um protocolo ZPF é *ótimo* se a retirada de qualquer *checkpoint* induzido resulta em uma relação $\mathcal{Z}$ não–precedente. Projetar um protocolo ZPF ótimo parece impossível, pois deveria levar em consideração informação futura [17]. Na Figura 4.5(a) podemos observar que p_1 não tem como saber que não há necessidade de um *checkpoint* forçado para prevenir a formação de uma relação $\mathcal{Z}$ não–precedente ($\hat{\sigma}_2^0 \prec \hat{\sigma}_0^1$), visto que a mensagem m causará a relação de precedência ($\hat{\sigma}_2^0 \prec \hat{\sigma}_0^1$).

4.5.2 Protocolos com Ausência de Ciclos–Z (ZCF)

Definição 4.9 *Um padrão de checkpoints é dito* livre de ciclos-Z (ZCF) *se, e somente se, nenhum checkpoint deste padrão participa de um ciclo-Z.*

Lema 4.2 *Em um sistema ZCF, todos os checkpoints são úteis para a construção de estados globais consistentes.*

Prova: Esse resultado é garantido pelo Colorário 4.3. □

Um protocolo ZCF tende a induzir um número menor de *checkpoints* que protocolos ZPF, pois permite relações $\mathcal{Z}$ não–precedentes.

Um protocolo ZCF é *ótimo* se a retirada de qualquer *checkpoint* induzido resulta em um ciclo-Z. Projetar um protocolo ZCF ótimo parece impossível, pois deveria levar em consideração informação futura [17]. Na Figura 4.5(b) podemos observar que p_3 não tem como saber que há necessidade de um *checkpoint* forçado, visto que a mensagem m formará um ciclo-Z ($\hat{\sigma}_1^0 \rightsquigarrow \hat{\sigma}_1^0$).

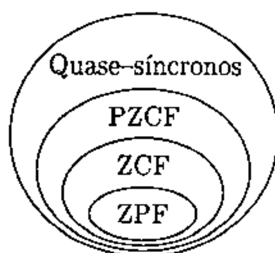


Figura 4.6: Classes de protocolos quase-síncronos

4.5.3 Protocolos que Restrigem a Presença de Ciclos-Z (PZCF)

Definição 4.10 *Um padrão de checkpoints é dito* parcialmente livre de ciclos-Z (PZCF) *se for permitida a presença de ciclos-Z nesse padrão.*

Protocolos PZCF tendem a induzir um número menor de *checkpoints* que os protocolos ZCF, pois permitem ciclos-Z.

Protocolos PZCF são úteis para recuperação com retrocesso de estado, quando as falhas são raras e a aplicação pode pagar o preço de um retrocesso maior. Protocolos PZCF podem não ser adequados para monitorização. Considere um monitor que confia na detecção de um predicado global baseado em *checkpoints* básicos tirados a partir de predicados locais avaliados pelos componentes da aplicação. Caso um desses *checkpoints* locais seja inútil e descartado pelo fotógrafo, há risco de que o monitor não seja capaz de detectar o predicado global. Quando se deseja uma simples visualização da execução da aplicação, protocolos PZCF podem ser utilizados, desde que a velocidade de progressão da visão global não seja crítica.

Protocolos PZCF não são equivalentes à abordagem assíncrona, pois espera-se que os protocolos façam algum tipo de esforço para evitar *checkpoints* inúteis. O protocolo proposto por Wang e Funchs [26] limita o retrocesso. O protocolo proposto por Xu e Netzer [28] garante a quebra de ciclos com determinada característica (Seção 4.6.4) e eles fizeram estudos estatísticos para demonstrar que o número de *checkpoints* inúteis gerados não tornava o protocolo inviável.

A Figura 4.6 mostra o relacionamento entre as classes de protocolos apresentadas.

4.6 Exemplos de Protocolos Quase–Síncronos

Como os principais protocolos quase–síncronos que temos conhecimento carregam informação de controle baseada em informação de precedência entre *checkpoints*, iniciamos esta Seção apresentando relógios lógicos e vetores de relógios para *checkpoints*. Em seguida, apresentamos um protocolo para cada umas das classes de protocolos quase–síncronos citadas.

4.6.1 Relógios Lógicos e Vetores de Relógios para *Checkpoints*

A informação de controle mantida e propagada por protocolos quase–síncronos é similar aos relógios estudados no Capítulo 3. Podemos definir relógios lógicos ou vetores de relógios para *checkpoints*, considerando que o incremento do relógio deve estar atrelado à ocorrência de um *checkpoint*.

As classes abstratas `QS_LogicalClock` (Classe 4.6) e `QS_VectorClock` (Classe 4.7) implementam parte da interface `QS_Clock` para a propagação de relógios lógicos e vetores de relógios para *checkpoints* em protocolos quase–síncronos. Especializações dessas classes serão estudadas nas próximas seções.

A iniciação dos relógios é um pouco diferente da feita no Capítulo 3. Como a implementação apresentada implica que todo *checkpoint* seja precedido por um incremento, iniciamos os relógios lógicos e as entradas dos vetores com -1 (Classes 4.6 e 4.7). No caso dos relógios lógicos, os *checkpoints* iniciais têm valor de relógio 0. No caso de vetores de relógios o componente i da aplicação terá apenas a sua respectiva entrada com 0 no vetor de relógios de seu estado inicial. Isso faz com que os estados iniciais formem um *checkpoint* global consistente (não há relações de precedência entre os *checkpoints* iniciais).

4.6.2 Vetores de Relógios Fixos por Intervalo

O protocolo VC_FDI [17] propaga vetores de relógios para *checkpoints* e induz *checkpoints* forçados sempre que um processo ou objeto recebe informação a respeito de algum *checkpoint* sobre o qual não tinha conhecimento. Desta forma, as incorporações de precedências novas ocorrem apenas imediatamente após os *checkpoints* e os valores dos vetores de relógios propagados são fixos por intervalo.

As Figuras 4.7 e 4.8 apresentam cenários no MPM e MOA com protocolo VC_FDI. O diagrama objeto–tempo apresentado na Figura 4.8 corresponde ao cenário da Figura 2.7 (página 10). A classe `VC_FDI` (Classe 4.8) especializa a classe `QS_VectorClock` para a implementação do protocolo VC_FDI. O Teorema 4.5 garante que o protocolo VC_FDI é ZPF e apresenta uma prova informal deste fato [17].

Classe 4.6 QS_LogicalClock.java

```
abstract public class QS_LogicalClock implements QS_Clock {

    public int lc;

    public QS_LogicalClock() { lc = -1; }
    protected QS_LogicalClock(QS_LogicalClock LC) { lc = LC.lc; }

    public void maximum(QS_Clock LC) {
        if ( ((QS_LogicalClock) LC).lc > lc ) lc = ((QS_LogicalClock) LC).lc;
    }

    public boolean equals(QS_Clock LC) { return lc == ((QS_LogicalClock) LC).lc; }

    public void increment(int id) { lc++; }
}
```

Classe 4.7 QS_VectorClock.java

```
abstract public class QS_VectorClock implements QS_Clock {

    public int[ ] v;

    public QS_VectorClock() { for (int i=0; i < Application.N; i++) v[i] = -1; }
    protected QS_VectorClock(QS_VectorClock V) { this.v = (int[ ]) V.v.clone(); }

    public void maximum(QS_Clock V) {
        for (int i=0; i < Application.N; i++)
            if ( ((QS_VectorClock) V).v[i] > v[i]) v[i] = ((QS_VectorClock) V).v[i];
    }

    public boolean equals(QS_Clock V) {
        for (int i=0; i < Application.N; i++)
            if ( ((QS_VectorClock) V).v[i] ≠ v[i]) return false;
        return true;
    }

    public void increment(int id) { v[id]++; }
}
```

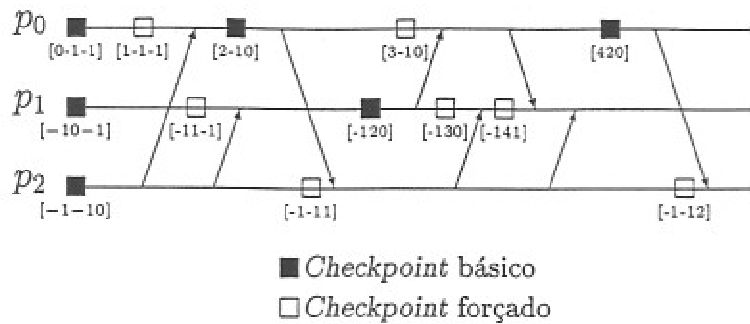


Figura 4.7: Cenário MPM com protocolo VC_FDI

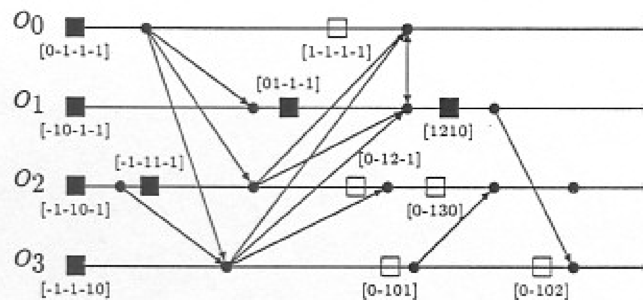


Figura 4.8: Cenário MOA com protocolo VC_FDI

Classe 4.8 VC_FDI.java

```
public class VC_FDI extends QS_VectorClock implements QS_Clock {

    protected VC_FDI(VC_FDI fdi) { super(fdi); }

    public QS_Clock propagate() { return new VC_FDI(this); }

    public boolean mustTakeForcedCkpt(int id, QS_Clock qs_clock) {
        for (int i=0; i < Application.N; i++)
            if ( ((VC_FDI) qs_clock).v[i] > v[i]) return true;
        return false;
    }
}
```

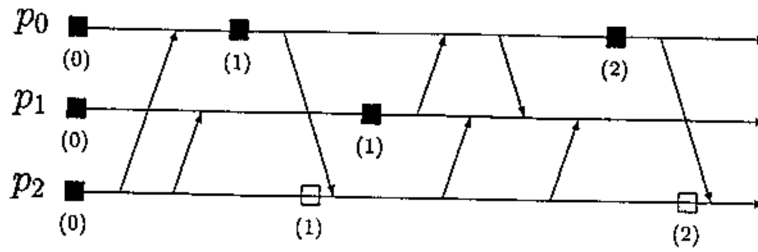


Figura 4.9: Cenário MPM com protocolo LC_FDI

Teorema 4.5 *O protocolo VC_FDI é ZPF.*

Prova: Devemos provar que para todo par de *checkpoints* $\hat{\sigma}_a^\alpha$ e $\hat{\sigma}_b^\beta$ pertencente a este padrão:

$$(\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta) \Rightarrow (\hat{\sigma}_a^\alpha \prec \hat{\sigma}_b^\beta)$$

Podemos quebrar a relação $(\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta)$ em uma série de relações de precedência entre pares de *checkpoints* (Definição 4.4, item (2)):

$$\hat{\sigma}_a^\alpha = \hat{\sigma}_{i_0}^{\gamma_0-1} \quad (\hat{\sigma}_{i_0}^{\gamma_0-1} \prec \hat{\sigma}_{i_1}^{\gamma_1}), (\hat{\sigma}_{i_1}^{\gamma_1-1} \prec \hat{\sigma}_{i_2}^{\gamma_2}), \ldots, (\hat{\sigma}_{i_{p-1}}^{\gamma_{p-1}-1} \prec \hat{\sigma}_{i_p}^{\gamma_p}) \quad \hat{\sigma}_{i_p}^{\gamma_p} = \hat{\sigma}_b^\beta$$

Utilizando indução no número p de precedências, vamos provar que o protocolo garante que $(\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta) \Rightarrow (\hat{\sigma}_a^\alpha \prec \hat{\sigma}_b^\beta)$.

Base: $(p = 1)$ $\hat{\sigma}_a^\alpha = \hat{\sigma}_{i_0}^{\gamma_0-1} \quad (\hat{\sigma}_{i_0}^{\gamma_0-1} \prec \hat{\sigma}_{i_1}^{\gamma_1}) \quad \hat{\sigma}_{i_1}^{\gamma_1} = \hat{\sigma}_b^\beta$

Neste caso, não há dúvidas de que $(\hat{\sigma}_a^\alpha \prec \hat{\sigma}_b^\beta)$.

Passo: $(p \geq 1)$ Suponha que $(\hat{\sigma}_a^\alpha \prec \hat{\sigma}_p^{\gamma_p})$ e $(\hat{\sigma}_p^{\gamma_p-1} \prec \hat{\sigma}_{p+1}^{\gamma_{p+1}})$. Como as incorporações de novas precedências ocorrem apenas imediatamente após os *checkpoints*, a informação a respeito de $\hat{\sigma}_a^\alpha$ deve ter sido propagada durante todo o intervalo $\Delta_p^{\gamma_p-1}$, pois em caso contrário estaríamos considerando que p_p ou o_p violou as regras do protocolo ao incorporar esta informação durante este intervalo. Conseqüentemente, $(\hat{\sigma}_a^\alpha \prec \hat{\sigma}_b^\beta)$. □

4.6.3 Relógios Lógicos Fixos por Invertavalo

O protocolo LC_FDI propaga relógios lógicos para *checkpoints* e induz *checkpoints* forçados sempre que um processo ou objeto recebe informação com valor de relógio maior que o corrente. Como no protocolo VC_FDI, os valores de relógios propagados são fixos por intervalo.

O protocolo LC_FDI induz um número menor ou igual de *checkpoints* em relação ao protocolo VC_FDI, pois relógios lógicos são consistentes com a relação de precedência mas não a caracterizam. Além disso, o protocolo LC_FDI induz no máximo $n-1$ *checkpoints* para cada *checkpoint* básico, enquanto no protocolo VC_FDI o número de *checkpoints* forçados por *checkpoint* básico não é limitado.

O protocolo LC_FDI produz um padrão de *checkpoints* idêntico ao protocolo proposto por Briatico, Ciuffoletti e Simoncini [9, 17]. A numeração dos *checkpoints* sugerida por esse protocolo é um pouco diferente e mais adequada à recuperação com retrocesso.

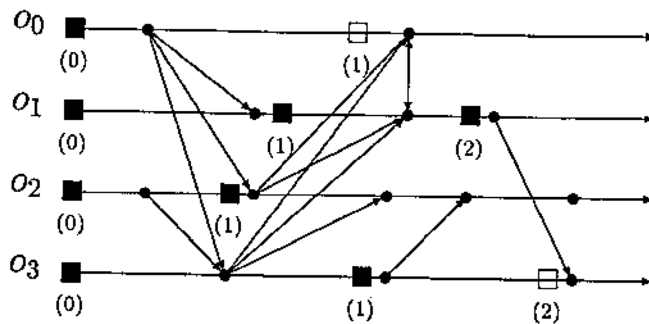


Figura 4.10: Cenário MOA com protocolo LC_FDI

Classe 4.9 LC_FDI.java

```
public class LC_FDI extends QS_LogicalClock implements QS_Clock {

    protected LC_FDI(LC_FDI fdi) { super(fdi); }

    public QS_Clock propagate() { return new LC_FDI(this); }

    public boolean mustTakeForcedCkpt(int id, QS_Clock qs_clock) {
        return (((LC_FDI) qs_clock).lc > lc );
    }
}
```

As Figuras 4.9 e 4.10 apresentam cenários no MPM e MOA com protocolo LC_FDI. A classe `LC_FDI` (Classe 4.9) especializa a classe `QS_LogicalClock` para a implementação do protocolo LC_FDI. O Teorema 4.6 garante que o protocolo LC_FDI é ZCF e apresenta uma prova informal deste fato [17].

Teorema 4.6 *O protocolo LC_FDI é ZCF.*

Prova: Vamos provar que o protocolo LC_FDI garante a seguinte condição:

$$(\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta) \Rightarrow LC(\hat{\sigma}_a^\alpha) < LC(\hat{\sigma}_b^\beta)$$

Como $LC(\hat{\sigma}) < LC(\hat{\sigma})$ não faz sentido, essa condição garante a ausência de ciclos–Z $(\hat{\sigma} \rightsquigarrow \hat{\sigma})$ no padrão. Vamos considerar uma relação $\mathcal{Z}$ $(\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta)$ e quebrá-la em uma série de relações de precedência entre pares de *checkpoints* (Definição 4.4, item (2)):

$$\hat{\sigma}_a^\alpha = \hat{\sigma}_{i_0}^{\gamma_0 - 1} \quad (\hat{\sigma}_{i_0}^{\gamma_0-1} \prec \hat{\sigma}_{i_1}^{\gamma_1}), (\hat{\sigma}_{i_1}^{\gamma_1-1} \prec \hat{\sigma}_{i_2}^{\gamma_2}), \ldots, (\hat{\sigma}_{i_{p-1}}^{\gamma_{p-1}-1} \prec \hat{\sigma}_{i_p}^{\gamma_p}) \quad \hat{\sigma}_{i_p}^{\gamma_p} = \hat{\sigma}_b^\beta$$

Utilizando indução no número p de precedências, vamos provar que o protocolo garante que $(\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta) \Rightarrow LC(\hat{\sigma}_a^\alpha) < LC(\hat{\sigma}_b^\beta)$.

Base: $(p = 1)$ $\hat{\sigma}_a^\alpha = \hat{\sigma}_{i_0}^{\iota_0} \quad (\hat{\sigma}_{i_0}^{\iota_0} \prec \hat{\sigma}_{i_1}^{\gamma_1}) \quad \hat{\sigma}_{i_1}^{\gamma_1} = \hat{\sigma}_b^\beta$

Nesse caso, $(\hat{\sigma}_a^\alpha \prec \hat{\sigma}_b^\beta)$ e, portanto, $LC(\hat{\sigma}_a^\alpha) < LC(\hat{\sigma}_b^\beta)$.

Passo: $(p \geq 1)$ Suponha que $LC(\hat{\sigma}_a^\alpha) < LC(\hat{\sigma}_p^{\gamma_p})$ e $(\hat{\sigma}_p^{\gamma_p - 1} \prec \hat{\sigma}_{p+1}^{\gamma_{p+1}})$. O valor de relógio lógico propagado no intervalo $\Delta_p^{\gamma_p - 1}$ é igual a $LC(\hat{\sigma}_p^{\gamma_p}) - 1$ e como p_{p+1} ou o_{p+1} deve incrementar seu relógio antes do *checkpoint* $\hat{\sigma}_{p+1}^{\gamma_{p+1}}$, temos que $LC(\hat{\sigma}_p^{\gamma_p}) \leq LC(\hat{\sigma}_{p+1}^{\gamma_{p+1}})$. Desta forma, $LC(\hat{\sigma}_a^\alpha) < LC(\hat{\sigma}_b^\beta)$.

□

4.6.4 Quebra de Ciclos-Z Quase–Precedentes

Como foi comentado na Seção 4.5.2 implementar um protocolo ótimo para a quebra de ciclos–Z parece impossível, pois necessitaria de informação do futuro. Existe, no entanto, uma categoria de ciclos–Z que podem ser detectados.

Definição 4.11 Ciclos–Z Quase–Precedentes—*Denominamos ciclos–Z quase–precedentes os ciclos $(\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_a^\alpha)$ tais que*

$$\exists \hat{\sigma}_c^\gamma, \hat{\sigma}_c^{\gamma+1} : (\hat{\sigma}_c^\gamma \prec \hat{\sigma}_a^\alpha) \wedge (\hat{\sigma}_a^\alpha \prec \hat{\sigma}_c^{\gamma+1})$$

A Figura 4.11 apresenta exemplos de ciclos–Z quase–precedentes. Xu e Netzer [28] propuseram um protocolo (XN) que se dispunha a impedir a formação de ciclos quase–precedentes nos quais a relação $(\hat{\sigma}_a^\alpha \prec \hat{\sigma}_c^{\gamma+1})$ é representada por uma única mensagem. Os ciclos representados nos itens (a) e (b) da Figura 4.11 não são permitidos pelo protocolo XN; o ciclo representado no item (c) pode ocorrer. Além disso, o protocolo XN é não mapeável para o MOA, pois necessita saber qual é o destinatário de uma mensagem (Seção 3.7).

Baldoni et al. [1, 2] propuseram um protocolo ZPF (BHMR), no qual a indução de *checkpoints* forçados ocorre por duas razões:

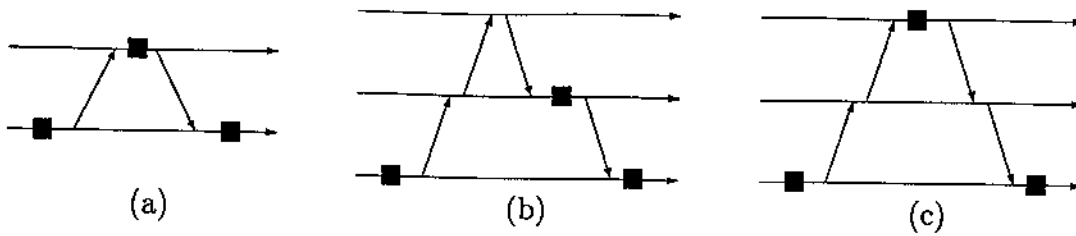


Figura 4.11: Ciclos–Z quase–precedentes

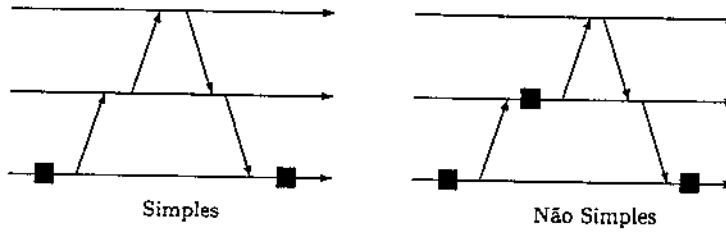


Figura 4.12: Relações de precedência simples e não simples

1. Detecção da formação de um ciclo–Z quase–precedente;
2. Prevenção da formação de ciclos–Z que não sejam quase–precedentes.

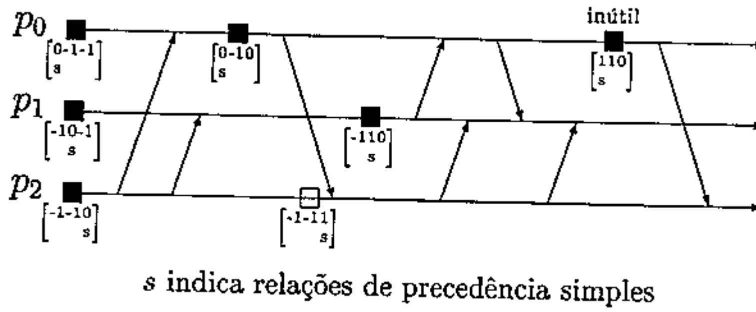
O protocolo BHMR não é mapeável para o MOA, pois a computação relacionada ao item (2) exige o conhecimento do destinatário de uma mensagem (Seção 3.7). O item (1) é mapeável e produz um protocolo PZCF mais eficiente, no que se refere a quebra de ciclos quase–precedentes, que o XN. Chamamos esse protocolo de XN_BHMR.

O protocolo XN_BHMR propaga vetores de relógios e pode induzir um *checkpoint* forçado em um componente da aplicação caso ele receba informação contendo o número do seu intervalo de *checkpoints* corrente. A indução do *checkpoint* está atrelada ao fato da informação a respeito do seu intervalo de *checkpoints* ter percorrido um caminho simples ou não. Em um caminho simples, nenhum componente da aplicação seleciona um *checkpoint* durante o fluxo da informação (Figura 4.12). Precedências simples (Definição 4.12) não podem produzir ciclos quase–precedentes.

Definição 4.12 Precedência Simples—*Dizemos que uma relação de precedência entre checkpoints* ($\hat{\sigma}^\alpha \prec \hat{\sigma}^\beta_b$) *é simples se, e somente se,*

$$\nexists \hat{\sigma}^\gamma_c : (\hat{\sigma}^\alpha \prec \hat{\sigma}^\gamma_c) \wedge (\hat{\sigma}^\gamma_c \prec \hat{\sigma}^\beta_b)$$

As Figuras 4.13 e 4.14 apresentam cenários no MPM e MOA com protocolo XN_BHMR. A classe `XN_BHMR` (Classe 4.10) especializa a classe `QS_VectorClock` para a implementação do



s indica relações de precedência simples

Figura 4.13: Cenário MPM com protocolo XN_BHMR

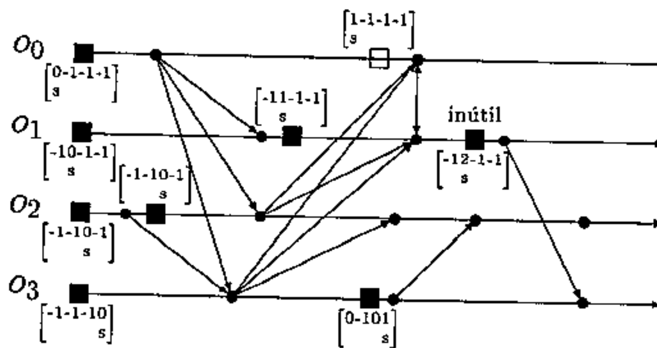


Figura 4.14: Cenário MOA com protocolo XN_BHMR

Classe 4.10 XN_BHMR.java

```
public class XN_BHMR extends QS_VectorClock implements QS_Clock {

    public boolean[ ] simple = new boolean[Application.N];

    protected XN_BHMR(XN_BHMR x) { super(x); simple = (boolean[ ]) x.simple.clone(); }

    public void maximum(QS_Clock qs_clock) {
        XN_BHMR xn_bhmr = (XN_BHMR) qs_clock;
        for (int i=0; i < Application.N; i++)
            if (xn_bhmr.v[i] > v[i]) simple[i] = xn_bhmr.simple[i];
            else if (xn_bhmr.v[i] == v[i]) simple[i] = simple[i] && xn_bhmr.simple[i];
        super.maximum(qs_clock);
    }

    public boolean equals(QS_Clock qs_clock) {
        XN_BHMR xn_bhmr = (XN_BHMR) qs_clock;
        for (int i=0; i < Application.N; i++)
            if (xn_bhmr.v[i] ≠ v[i] || (xn_bhmr.v[i] == v[i] && xn_bhmr.simple[i] ≠ simple[i]))
                return false;
        return true;
    }

    public void increment(int id) {
        super.increment(id);
        for (int i=0; i < Application.N; i++) simple[i] = false;
        simple[id] = true;
    }

    public QS_Clock propagate() { return new XN_BHMR(this); }

    public boolean mustTakeForcedCkpt(int id, QS_Clock qs_clock) {
        return ((((XN_BHMR) qs_clock).v[id] == v[id]) && !((XN_BHMR) qs_clock).simple[id]);
    }
}
```

protocolo XN_BHMR. Todo componente da aplicação mantém um vetor de booleanos cujas entradas recebem valor falso a cada seleção de *checkpoint*. Apenas a entrada correspondente ao componente da aplicação recebe valor verdadeiro. Dessa forma, caso um componente i da aplicação receba informação contendo $VC[i]$ igual ao intervalo corrente e $simple(i)$ com valor falso, um *checkpoint* forçado deve ser induzido.

4.7 Visão Progressiva Utilizando *Checkpoints*

A construção progressiva de *checkpoints* globais consistentes é mais complexa que a de estados globais consistentes. Esta complexidade extra se deve ao fato de estados permitirem precedência e consistência, enquanto *checkpoints* exigem concorrência para consistência.

Como fizemos na Seção 3.6, vamos utilizar uma notação auxiliar para a apresentação e prova de correção dos algoritmos. Seja $\hat{\Sigma}$ um *checkpoint* global consistente. Consideramos definições análogas às apresentadas na Definição 3.11 (página 39) para $Channel(\hat{\Sigma})$, $Step(\hat{\Sigma})$ e $Next(\hat{\Sigma})$.

Como na construção progressiva de estados globais consistentes, é necessário o conhecimento a respeito da precedência entre *checkpoints*. O Teorema 4.7 apresenta uma regra para a escolha de $Step(\hat{\Sigma})$.

Teorema 4.7 *Seja $\hat{\Sigma}$ um checkpoint global consistente. Definimos,*

$$Step(\hat{\Sigma}) = \{\hat{\sigma}_i^{\iota_i+1} \in Channel(\hat{\Sigma}) : \hat{\sigma}_j^{\iota_j+1} \in Channel(\hat{\Sigma}) \Rightarrow \hat{\sigma}_j^{\iota_j+1} \not\rightsquigarrow \hat{\sigma}_i^{\iota_i+1}\}$$

$Next(\hat{\Sigma})$, formado pela substituição de $Step(\hat{\Sigma})$ em $\hat{\Sigma}$, é um checkpoint global consistente.

Prova: Suponha que $Next(\hat{\Sigma})$ é inconsistente e existe um par de *checkpoints* $\hat{\sigma}_a^\alpha$ e $\hat{\sigma}_b^\beta$ pertencente a $Next(\hat{\Sigma})$ tal que $(\hat{\sigma}_a^\alpha \prec \hat{\sigma}_b^\beta)$. Existem quatro alternativas para a origem de $\hat{\sigma}_a^\alpha$ e $\hat{\sigma}_b^\beta$:

- $\hat{\sigma}_a^\alpha \in \hat{\Sigma}$, $\hat{\sigma}_b^\beta \in \hat{\Sigma}$—Contraria a hipótese de $\hat{\Sigma}$ ser um *checkpoint* global consistente.
- $\hat{\sigma}_a^\alpha \in \hat{\Sigma}$, $\hat{\sigma}_b^\beta \in Step(\hat{\Sigma})$—Como $\hat{\sigma}_a^{\alpha+1}$ não está em $Step(\hat{\Sigma})$, deve existir um *checkpoint* $\hat{\sigma}_i^{\iota_i+1}$ em $Channel(\hat{\Sigma})$ tal que $(\hat{\sigma}_i^{\iota_i+1} \rightsquigarrow \hat{\sigma}_a^{\alpha+1})$. A relação $(\hat{\sigma}_a^\alpha \prec \hat{\sigma}_b^\beta)$ concatenada à relação $(\hat{\sigma}_i^{\iota_i+1} \rightsquigarrow \hat{\sigma}_a^{\alpha+1})$ forma uma relação $\mathcal{Z}$ $(\hat{\sigma}_i^{\iota_i+1} \rightsquigarrow \hat{\sigma}_b^\beta)$, que impede $\hat{\sigma}_b^\beta$ de estar em $Step(\hat{\Sigma})$.
- $\hat{\sigma}_a^\alpha \in Step(\hat{\Sigma})$, $\hat{\sigma}_b^\beta \in \hat{\Sigma}$—Contraria a hipótese de $\hat{\Sigma}$ ser um *checkpoint* global consistente.
- $\hat{\sigma}_a^\alpha \in Step(\hat{\Sigma})$, $\hat{\sigma}_b^\beta \in Step(\hat{\Sigma})$—Contraria a regra de formação de $Step(\hat{\Sigma})$. □

Um resultado bastante interessante pode ser derivado da construção da visão progressiva de uma aplicação.

Definição 4.13 *Dizemos que um padrão de checkpoints permite a construção de uma* visão progressiva *da aplicação se para todo checkpoint global consistente $\hat{\Sigma}$, com exceção do checkpoint global final, $Step(\hat{\Sigma})$ é não vazio.*

Teorema 4.8 *Temos visão progressiva se, e somente se, não há ciclos em zigzag.*

Prova: *Ausência de ciclos-Z $\Rightarrow$ Visão progressiva*

Suponha que $Step(\hat{\Sigma})$ é vazio. Então, para todo *checkpoint* $\hat{\sigma}_i^{l_i+1}$ pertencente a $Channel(\hat{\Sigma})$ podemos escolher um *checkpoint* $\hat{\sigma}_j^{l_j+1}$ tal que $(\hat{\sigma}_j^{l_j+1} \rightsquigarrow \hat{\sigma}_i^{l_i+1})$. Como o número de *checkpoints* em $Channel(\hat{\Sigma})$ é finito, teremos um ciclo Z.

Visão progressiva $\Rightarrow$ Ausência de ciclos-Z

Provamos que todos os *checkpoints* são utéis. Considere um *checkpoint* $\hat{\sigma}_i^{l_i}$ pertencente a $Channel(\hat{\Sigma})$ e que $\hat{\Sigma}$ é o *checkpoint* global máximo [25] contendo $\hat{\sigma}_i^{l_i}$. Informalmente, um *checkpoint* global máximo de um *checkpoint* é o *checkpoint* global consistente mais à direita em um diagrama espaço–tempo ou objeto–tempo. Isto implica que nenhum *checkpoint* em $Channel(\hat{\Sigma})$ pode ser substituído em $\hat{\Sigma}$ para formar um novo corte consistente na ausência de $\hat{\sigma}_i^{l_i+1}$. Como a aplicação garante a visão progressiva, $\hat{\sigma}_i^{l_i+1}$ deve pertencer a $Step(\hat{\Sigma})$ e é, portanto, útil. Pelo Colorário 4.3 temos a ausência de ciclos–Z. □

Vamos analisar a construção progressiva de *checkpoints* globais consistentes utilizando relógios lógicos e vetores de relógios para *checkpoints.*

Construção Progressiva Utilizando Relógios Lógicos

Toda implementação de relógios lógicos para *checkpoints* deve ser consistente com a relação de precedência entre *checkpoints*, ou seja,

$$(\hat{\sigma} \prec \hat{\sigma}') \Rightarrow LC(\hat{\sigma}) < LC(\hat{\sigma}').$$

Para a utilização da regra expressa no Teorema 4.7 para escolha de $Step(\hat{\Sigma})$ precisamos de padrões de *checkpoints* que garantam a consistência com a relação $\mathcal{Z}$

$$(\hat{\sigma} \rightsquigarrow \hat{\sigma}') \Rightarrow LC(\hat{\sigma}) < LC(\hat{\sigma}').$$

Todo protocolo que garante esta condição é ZCF, pois $(\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_a^\alpha)$ implicaria $LC(\hat{\sigma}_a^\alpha) < LC(\hat{\sigma}_a^\alpha)$. O protocolo LC_FDI (Seção 4.6.3) é um exemplo de protocolo ZCF consistente com a relação $\mathcal{Z}$. Além disso, todos os protocolos ZPF garantem esta condição, visto que $(\hat{\sigma} \rightsquigarrow \hat{\sigma}') \Rightarrow (\hat{\sigma} \prec \hat{\sigma}') \Rightarrow LC(\hat{\sigma}) < LC(\hat{\sigma}')$.

Os *checkpoints* que possuem o valor mínimo de LC em $Channel(\hat{\Sigma})$ podem ser utilizados como $Step(\hat{\Sigma})$. Dessa forma, o algoritmo apresentado na classe `LC_Photographer` (Classe 3.8)

Classe 4.11 VC_Ckpt.java

```
public class VC_Ckpt {
    public QS_VectorClock VC;
    public int id;
    // Process or object checkpoint

    boolean precedes(VC_Ckpt ckpt) { return (ckpt.VC.v[id] ≤ VC.v[id]); }
}
```

pode ser utilizado para a construção progressiva de *checkpoints* globais consistentes desde que o protocolo quase–síncrono seja consistente com a relação $\mathcal{Z}$.

Construção Progressiva Utilizando Vetores de Relógios

O algoritmo a ser utilizado depende da classe a que o protocolo quase–síncrono pertence: ZPF, ZCF ou PZCF. A classe `VC_Ckpt` (Classe 4.11) descreve a informação recebida pelo fotógrafo, contendo um *checkpoint*, o identificador do objeto ou processo e o vetor de relógios relativo a este *checkpoint*. Além disso, apresenta um método para verificação da precedência entre *checkpoints*.

Construção Progressiva para protocolos ZPF

O algoritmo apresentado na classe `VC_Photographer` (Classe 3.9) pode ser utilizado para a construção progressiva de *checkpoints* globais consistentes desde que o protocolo quase–síncrono seja ZPF. Basta fazer a substituição da classe `VC_State` pela classe `VC_Ckpt` (Classe 4.11).

Construção Progressiva para protocolos ZCF

Lema 4.3 *Seja $\mathcal{M}$ um subconjunto de processos ou objetos tal que seus checkpoints em $Channel(\hat{\Sigma})$ respeitam a seguinte condição:*

$$C: \quad i \in \mathcal{M} \Rightarrow \exists \hat{\sigma}_j^{\iota_j+1} \in Channel(\hat{\Sigma}) : (\hat{\sigma}_j^{\iota_j+1} \rightsquigarrow \hat{\sigma}_i^{\iota_i+1})$$

Seja $\mathcal{M}'$ o conjunto formado pelos processos ou objetos que possuem checkpoints em $Channel(\hat{\Sigma})$ precedidos por checkpoints em $\hat{\Sigma}$ correspondentes aos elementos em $\mathcal{M}$:

$$\exists i \in \mathcal{M} : (\hat{\sigma}_i^{\iota_i} \prec \hat{\sigma}_k^{\iota_k+1}) \Rightarrow k \in \mathcal{M}'$$

O conjunto formado pela união de $\mathcal{M}$ e $\mathcal{M}'$ respeita a condição C.

Prova: Devemos provar que os elementos em $\mathcal{M}'$ que não estão presentes em $\mathcal{M}$ respeitam a condição C. Para todo elemento k em $\mathcal{M}'$ existe um elemento i em $\mathcal{M}$ tal que $(\hat{\sigma}_i^{\iota_i} \prec \hat{\sigma}_k^{\iota_k+1})$. Para todo elemento i em $\mathcal{M}$ podemos escolher um *checkpoint* $\hat{\sigma}_j^{\iota_j+1}$ em $Channel(\hat{\Sigma})$ tal que

Classe 4.12 ZCF_Photographer.java

```
public class ZCF_Photographer {

    VC_Ckpt[ ] C; // Consistent global checkpoint
    VC_Ckpt[ ] S; // Succeeding checkpoints of C

    protected boolean[ ] M = new boolean[Application.N];

    public void ZCF_newGlobalCheckpoint() {

        for (int i=0; i < Application.N; i++) M[i] = false;

        for (int i=0; i < Application.N; i++)
            for (int j=0; j < Application.N; j++) {
                if (S[i].precedes(S[j]))
                    mark(j);
            }

        for (int i=0; i < Application.N; i++)
            if (!M[i]) { C[i] = S[i]; S[i] = null; }
    }

    protected void mark(int j) {
        if (!M[j]) {
            M[j] = true;
            for (int k=0; k < Application.N; k++)
                if (C[j].precedes(S[k])) mark (k);
        }
    }
}
```

$(\hat{\sigma}_j^{\iota_j+1} \rightsquigarrow \hat{\sigma}_i^{\iota_i+1})$. Estas duas relações concatenadas formam uma relação $\mathcal{Z}$ $(\hat{\sigma}_j^{\iota_j+1} \rightsquigarrow \hat{\sigma}_k^{\iota_k+1})$, o que indica que o elemento k respeita a condição C. $\square$

O algoritmo apresentado na classe `ZCF_Photographer` (Classe 4.12) coloca em $\mathcal{M}$ os processos ou objetos que têm *checkpoints* em $Channel(\hat{\Sigma})$ precedidos por outros *checkpoints* em $Channel(\hat{\Sigma})$ e aplica recursivamente o resultado apresentado no Lema 4.3 até que nenhum processo ou objeto possa ser adicionado. Vamos provar que esta abordagem é suficiente para escolher o conjunto $Step(\hat{\Sigma})$.

Teorema 4.9 *O algoritmo apresentado na classe 4.12 escolhe devidamente o conjunto* $Step(\hat{\Sigma})$.

Prova: Esta prova é praticamente idêntica à prova apresentada para o Teorema 4.7. Vamos supor que $Next(\hat{\Sigma})$ é inconsistente e existe um par de *checkpoints* $\hat{\sigma}_a^\alpha$ e $\hat{\sigma}_b^\beta$ pertencente a $Next(\hat{\Sigma})$ tal que $(\hat{\sigma}_a^\alpha \prec \hat{\sigma}_b^\beta)$. Existem quatro alternativas para a origem de $\hat{\sigma}_a^\alpha$ e $\hat{\sigma}_b^\beta$:

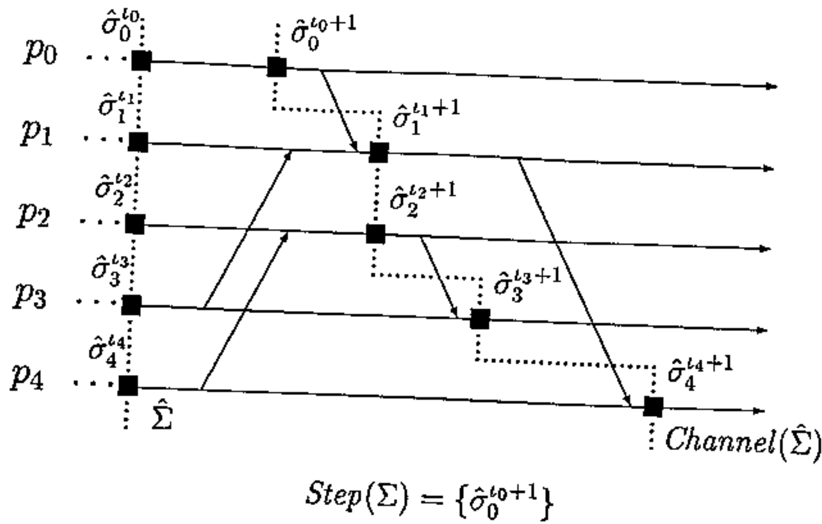


Figura 4.15: Progressão no MPM com protocolo ZCF

- $\hat{\sigma}_a^\alpha \in \hat{\Sigma}$, $\hat{\sigma}_b^\beta \in \hat{\Sigma}$—Contraria a hipótese de $\hat{\Sigma}$ ser um *checkpoint* global consistente.
- $\hat{\sigma}_a^\alpha \in \hat{\Sigma}$, $\hat{\sigma}_b^\beta \in Step(\hat{\Sigma})$—$a$ foi marcado e devido à relação de precedência, b também deveria ser marcado.
- $\hat{\sigma}_a^\alpha \in Step(\hat{\Sigma})$, $\hat{\sigma}_b^\beta \in \hat{\Sigma}$—Contraria a hipótese de $\hat{\Sigma}$ ser um *checkpoint* global consistente.
- $\hat{\sigma}_a^\alpha \in Step(\hat{\Sigma})$, $\hat{\sigma}_b^\beta \in Step(\hat{\Sigma})$—$b$ seria marcado e não poderia estar em $Step(\hat{\Sigma})$.

Como o padrão ZCF garante a ausência de ciclos–Z, pelo menos um processo ou objeto não será marcado e $Step(\hat{\Sigma})$ não é vazio. □

A Figura 4.15 ilustra um passo do fotógrafo no MPM, com protocolo ZCF.

Construção Progressiva para protocolos PZCF

Os protocolos PZCF não garantem a ausência de *checkpoints* inúteis. O objetivo do algoritmo apresentado na classe PZCF_Photographer (Classe 4.13) é poder utilizar os *checkpoints* úteis do padrão e descartar os inúteis. A implementação para os protocolos ZCF se preocupava somente em detectar os *checkpoints* $\hat{\sigma}_i^{\iota_i+1}$ em $Channel(\hat{\Sigma})$ tais que $Channel(\hat{\Sigma}) \rightsquigarrow \hat{\sigma}_i^{\iota_i}$. No caso dos protocolos PZCF é necessário que se saiba qual é a identidade do processo ou objeto. Para isso, utilizamos uma matriz Z tal que se a entrada $Z[i][j]$ tiver valor verdadeiro, temos uma relação $\mathcal{Z}$ $(\hat{\sigma}_i^{\iota_i+1} \rightsquigarrow \hat{\sigma}_j^{\iota_j+1})$.

A Figura 4.16 ilustra um passo do fotógrafo no MPM, com protocolo PZCF. Nem todos os *checkpoints* inúteis presentes em $Channel(\hat{\Sigma})$ podem ser detectados. A Figura 4.17 traz um exemplo de *checkpoint* inútil, $\hat{\sigma}_1^{\iota_1+1}$, que não pode ser detectado no passo atual.

Classe 4.13 PZCF_Photographer.java

```
public class PZCF_Photographer {

    VC_Ckpt[ ] C; // Consistent global checkpoint
    VC_Ckpt[ ] S; // Succeeding checkpoints of C

    protected boolean[ ] M = new boolean[Application.N];
    protected boolean[ ][ ] Z = new boolean [Application.N][Application.N];

    public void PZCF_newGlobalCheckpoint() {
        for (int i=0; i < Application.N; i++) {
            M[i] = false;
            for (int j=0; j < Application.N; j++) Z [i][j] = false;
        }

        for (int i=0; i < Application.N; i++)
            for (int j=0; j < Application.N; j++) {
                if (S[i].precedes(S[j]))
                    mark(i,j);
            }

        for (int i=0; i < Application.N; i++)
            if (Z[i][i]) { S[i] = null; }  // useless checkpoint
            else if (!M[i]) { C[i] = S[i]; S[i] = null; }
    }

    protected void mark(int i, int j) {
        if (!Z[i][j]) {
            Z[i][j] = M[j] = true;
            for (int k=0; k < Application.N; k++)
                if (C[j].precedes(S[k])) mark (i,k);
        }
    }
}
```

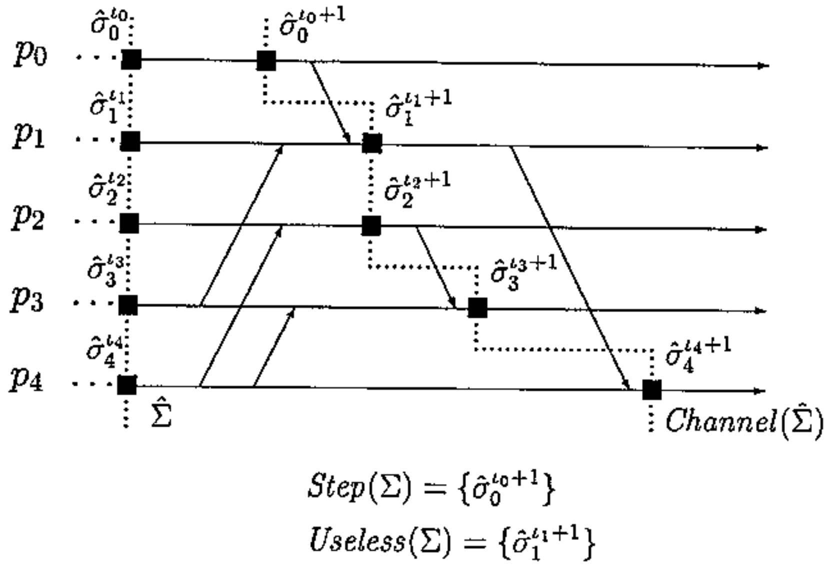
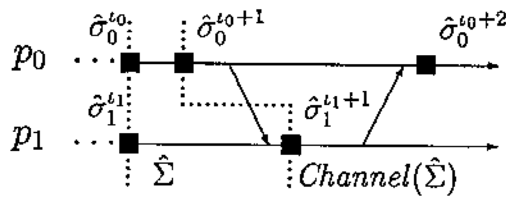


Figura 4.16: Progressão no MPM com protocolo PZCF



A inutilidade de $\hat{\sigma}_1^{\iota_1+1}$ não pode ser detectada neste passo.

Figura 4.17: Detecção de *checkpoints* inúteis

Um fotógrafo que se baseie neste algoritmo sempre progride, pois ou incorpora novos *checkpoints* a $\hat{\Sigma}$ ou existe um ciclo–Z e ele é capaz de descartar um *checkpoint*.

4.8 Sumário

Motivados pelo fato de que o uso de todos os estados de uma aplicação para monitorização pode ter custo excessivamente alto, consideramos a seleção de *checkpoints*.

No MPM, *checkpoints* são selecionados em eventos especiais. No MOA, consideramos a execução de uma pseudo-ação atômica independente da estrutura de ações da aplicação. Em ambos os modelos, dois *checkpoints* são consistentes entre si se, e somente se, não há relação de precedência entre eles. Apesar de necessária, essa condição não é suficiente para que eles participem de um mesmo *checkpoint* global consistente [18].

O estabelecimento de uma condição necessária e suficiente para a participação de um *checkpoint* em um *checkpoint* global consistente no MPM é atribuída a Netzer e Xu [18] e é descrita a partir de caminhos em zigzag (caminho–Z). Um caminho–Z entre dois *checkpoints* impede-os de participar de um mesmo *checkpoint* global consistente. A existência de um caminho–Z que leva um *checkpoint* a ele mesmo (ciclo–Z) impede este *checkpoint* de participar de *qualquer checkpoint* global consistente (*checkpoint* inútil).

A definição de caminhos em zigzag como apresentada por Netzer e Xu [18] baseia-se em mensagens, dificultando o seu mapeamento para o MOA. Propusemos uma relação equivalente entre *checkpoints*, denominada relação $\mathcal{Z}$, que pode ser aplicada diretamente no MOA. A nossa definição está baseada unicamente no conceito de precedência entre *checkpoints* e deixa claro como é feita a composição da relação.

Protocolos quase–síncronos permitem que os componentes da aplicação selecionem *checkpoints* básicos livremente, mas podem ser induzidos pelo protocolo a selecionar *checkpoints* forçados, adicionais, de maneira a eliminar ou reduzir a presença de *checkpoints* inúteis. Apresentamos um mecanismo genérico para a manutenção e propagação de informação de controle para protocolos quase–síncronos no MPM e no MOA, semelhante à manutenção e propagação de relógios apresentada no Capítulo 3.

Protocolos quase–síncronos definem um padrão de *checkpoints* que pode permitir ou não o estabelecimento de relações $\mathcal{Z}$ não–precedentes ou ciclos–Z. Apresentamos a classificação proposta por Manivannan e Singhal [17]: (i) protocolos ZPF (livres de relações $\mathcal{Z}$ não–precedentes), (ii) protocolos ZCF (livres de ciclos–Z) e (iii) protocolos PZCF (permitem ciclos–Z).

Apresentamos três protocolos quase–síncronos que podem ser utilizados no MPM e no MOA e que ilustram essa classificação. O protocolo VC_FDI é ZPF e propaga vetores de relógios. O protocolo LC_FDI é ZCF e propaga relógios lógicos. O protocolo XN_BHMR é PZCF, propaga vetores de relógios e se dispõe a quebrar ciclos–Z quase–precedentes.

Propomos algoritmos para a construção progressiva de estados globais consistentes utilizando relógios lógicos e vetores de relógios para *checkpoints*. Utilizando-se vetores de relógios, o algoritmo utilizado varia conforme a classe de protocolos quase–síncronos considerada.

Demonstramos que o fato de uma aplicação permitir a construção de uma visão progressiva equivale ao fato do protocolo utilizado não permitir *checkpoints* inúteis. A análise da construção progressiva de *checkpoints* globais consistentes trouxe dois resultados interessantes: (i) qualquer protocolo que não permite *checkpoints* inúteis pode ser utilizado para a monitorização e (ii) a visão progressiva é equivalente à ausência de *checkpoints* inúteis. Consideramos que este último resultado pode ser valioso para a elaboração de novos protocolos.

Capítulo 5

Conclusão

A obtenção de estados globais consistentes [6] é fundamental para a solução de uma série de problemas em sistemas distribuídos, em especial para *monitorização* e *reconfiguração* de aplicações distribuídas [8].

Consideramos que deve haver uma separação clara entre a camada de aplicação—que trata do domínio do problema—e a camada de controle—responsável pelos aspectos de gerência da aplicação. A camada de controle, por sua vez, pode ser subdividida em um módulo responsável pela obtenção de estados globais consistentes (fotógrafo) e um módulo responsável pela avaliação de predicados e atuação sobre a aplicação (monitor) [11].

Nesta dissertação, consideramos aplicações distribuídas construídas sobre o modelo de processos e mensagens (MPM) e sobre o modelo de objetos e ações atômicas (MOA). Existem vários algoritmos na literatura para a construção de estados globais consistentes no MPM [1, 3, 6, 9, 17], enquanto a oferta é extremamente pequena no MOA. Uma explicação para este fato pode ser a existência de um mecanismo imediato para a obtenção de estados globais neste modelo: basta abrir uma ação atômica que adquira *locks* de leitura sobre todos os objetos da computação. O único algoritmo assíncrono que temos conhecimento, proposto por Fischer et al. [10], requer a duplicação da aplicação: uma cópia continua a execução normalmente, enquanto a outra termina as ações abertas e coleta o estado global consistente. Não consideramos nenhuma destas abordagens viável para a monitorização e buscamos o mapeamento de algoritmos para a obtenção de estados globais do MPM para o MOA.

A monitorização de aplicações distribuídas utilizando todos os estados de processos ou objetos componentes da aplicação pode ser muito custosa. Gostaríamos de escolher estados de interesse (*checkpoints*) para envio ao fotógrafo. Um requisito da monitorização pode ser a participação destes *checkpoints* em *checkpoints* globais consistentes. *Checkpoints* que participam de pelo menos um *checkpoint* global consistente são denominados *úteis*; caso contrário são denominados *inúteis*.

Consideramos que o monitor deve ter uma *visão progressiva* da aplicação, no sentido de que o fotógrafo deve lhe apresentar um estado global consistente mais recente que os apresentados anteriormente. Propusemos algoritmos para a construção de uma visão progressiva baseado em estados e *checkpoints*. Dentro do nosso conhecimento, esta abordagem, quando aplicada a *checkpoints*, é original e trouxe resultados bastante interessantes.

5.1 Mapeamento entre MPM e MOA

O MPM é um modelo assíncrono, no qual os processos são entidades independentes que se comunicam exclusivamente através da troca de mensagens. O MOA é um modelo síncrono, composto por objetos que interagem através da invocação de métodos atômicos. Possui um grau de confiabilidade devido às propriedades de equivalência serial, atomicidade e permanência de efeito conferidas pelas ações atômicas [27].

Definimos precedência entre ações atômicas de maneira análoga à precedência entre eventos proposta por Lamport [13]. Baseados no conceito de precedência, pudemos definir estados globais consistentes no MOA. Mais do que isso, pudemos definir o reticulado da aplicação, formado pelo conjunto de todos os estados globais consistentes, que é útil para a detecção de predicados instáveis (predicados que podem alterar seu valor de maneira imprevisível) [8, 7].

A inclusão de relógios (informação de precedência) aos estados dos componentes da aplicação permite a verificação da consistência de estados globais por um observador externo (fotógrafo). Analisamos as propriedades de relógios lógicos e vetores de relógios no MPM [8] e MOA [11].

Após um estudo detalhado das relações de precedência nos dois modelos, pudemos estabelecer limitações para o mapeamento de algoritmos. No MPM, relações de precedência imediata (sem considerar transitividade) conectam exatamente dois estados. No MOA, temos vários estados precedendo e sendo precedidos imediatamente por único estado, o que pode vir a aumentar complexidade de alguns algoritmos mapeados. Além disso, como as precedências imediatas estruturais são imprevisíveis, algoritmos que necessitam saber o destinatário de uma mensagem não são mapeáveis.

5.2 Visão Progressiva da Aplicação

Grande parte da teoria e protocolos desenvolvidos para a obtenção de *checkpoints* globais consistentes foi projetada tendo a recuperação de falhas com retrocesso de estado como alvo [9, 17]. Como o nosso objetivo era a monitorização de aplicações distribuídas, tivemos de considerar outra abordagem, que deveria permitir a construção progressiva de estados ou *checkpoints* globais consistentes.

Consideramos que o fotógrafo é o responsável pela construção da visão progressiva e recebe estados ou *checkpoints* dos componentes da aplicação através de canais FIFO. A construção progressiva de estados ou *checkpoints* globais consistentes pode ser feita desde que tenha havido a propagação de relógios lógicos ou vetores de relógios. A construção progressiva de *checkpoints* globais consistentes é um pouco mais complexa que a de estados globais consistentes, pois *checkpoints* exigem concorrência para consistência, enquanto estados podem ser precedentes e consistentes.

Consideramos que os *checkpoints* são selecionados pelos componentes da aplicação de acordo com protocolos quase–síncronos [17]. Esta abordagem permite a seleção independente de estados pelos componentes, mas, eventualmente, pode induzir a seleção de estados adicionais de acordo com informações de controle mantidas e propagadas pelos componentes. Graças ao mapeamento obtido entre o MPM e MOA, pudemos verificar os mecanismos necessários para a manutenção e propagação de informação de controle para protocolos quase–síncronos no MOA e apresentamos protocolos adequados ao MPM e MOA.

As condições necessárias e suficientes para a utilidade de um *checkpoint* foram descritas por Netzer e Xu [18] a partir de caminhos–Z. Como caminhos–Z são baseados em mensagens, não tínhamos um mapeamento direto dessa relação para o MOA. Propusemos uma relação equivalente entre *checkpoints*, denominada relação $\mathcal{Z}$, baseada unicamente na precedência entre *checkpoints* e que pode ser utilizada diretamente no MOA.

Os protocolos quase–síncronos definem um padrão de *checkpoints* e podem ser classificados de acordo com o fato deste padrão permitir ou não o estabelecimento de relações $\mathcal{Z}$ não causais ou *checkpoints* inúteis. Apresentamos algoritmos para a construção progressiva de *checkpoints* globais consistentes de acordo com a classe de protocolo quase–síncrono utilizada. Encontramos uma equivalência entre a construção da visão progressiva e a ausência de *checkpoints* inúteis.

5.3 Trabalhos Futuros

Com a visão progressiva da aplicação, acreditamos ser possível a construção de um reticulado parcial da aplicação, formado apenas por *checkpoints* globais consistentes. A exploração deste reticulado pode ser útil para a detecção de predicados instáveis [8, 7].

Vários protocolos quase–síncronos presentes na literatura provam que não permitem a presença de *checkpoints* inúteis através da apresentação de um *checkpoint* global para cada *checkpoint* local [1, 3]. Acreditamos que o resultado obtido com a visão progressiva possa permitir provas mais flexíveis e, portanto, ser útil para a proposta de novos protocolos.

Na descrição feita, utilizamos a transmissão explícita de relógios e informação de controle. Esta abordagem é intrusiva e sujeita a erros cometidos pelos programadores das aplicações. A utilização de reflexão computacional [15] permite uma maneira elegante e não intrusiva

de se modificar o comportamento de uma aplicação. Consideramos a implementação de um ambiente para monitorização utilizando os algoritmos apresentados sobre Guaraná [19], uma arquitetura de *software* reflexiva bastante flexível, projetada para facilitar a reconfiguração e a reutilização de objetos de meta-nível.

No início, o nosso objetivo era transpor soluções conhecidas do MPM para o MOA. A surpresa foi termos encontrado resultados originais também para o MPM. Concluímos que o estudo feito nos proporcionou um melhor entendimento de ambos os modelos. A definição da relação $\mathcal{Z}$ baseada em precedência em vez de mensagens é um exemplo; a construção da visão progressiva da aplicação é outro. Esperamos que a exploração destes conceitos continue dando bons resultados.

Bibliografia

[1] R. Baldoni, J. M. Helary, A. Mostefaoui, and M. Raynal. Consistent Checkpointing in Message Passing Distributed Systems. Technical Report 2564, INRIA, June 1995.

[2] R. Baldoni, J. M. Helary, A. Mostefaoui, and M. Raynal. A Communication-Induced Checkpoint Protocol that Ensures Rollback Dependency Trackability. In *IEEE Symposium on Fault Tolerant Computing (FTCS97)*, 1997.

[3] R. Baldoni, F. Quaglia, and P. Fornara. An Index-Based Checkpoint Algorithm for Autonomous Distributed Systems. Technical Report 07-97, Universitá "La Sapienza", Roma, Italy, Mar. 1997.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[5] G. Booch. *Object Oriented Analysis & Design.* Benjamin Cummings, second edition, 1994.

[6] M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computing Systems*, 3(1):63–75, Feb. 1985.

[7] R. Cooper and K. Marzullo. Consistent Detection of Global Predicates. *SIGPLAN Notices*, 26(12):167–174, Dec. 1991.

[8] Ö. Babaoğlu and K. Marzullo. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.

[9] E. N. Elnozahy, D. Johnson, and Y.M.Yang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, 1996.

[10] M. J. Fischer, N. D. Griffeth, and N. A. Lynch. Global States of a Distributed System. *IEEE Transactions on Software Engineering*, SE-8(3):198–202, may 1982.

[11] I. C. Garcia and L. E. Buzato. Asynchronous Construction of Consistent Global Snapshots in the Object and Action Model. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, Annapolis, Maryland, EUA, May 1998. IEEE. Disponível como Relatório Técnico IC–98–16.

[12] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification.* Java Series. Addison–Wesley, Sept. 1996. Version 1.0.

[13] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[14] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice.* Second, Revised Edition, Springer-Verlag, New York, 1990.

[15] P. Maes. Concepts and Experiments in Computational Reflection. *ACM SIGPLAN Notices*, 22(12):147–155, Dec. 1987.

[16] D. Manivannan, R. H. B. Netzer, and M. Singhal. Finding consistent global checkpoints in a distributed computation. In *IEEE Transactions on Parallel and Distributed Systems*, pages 623–627, June 1997.

[17] D. Manivannan and M. Singhal. Quasi–Synchronous Checkpointing: Models, Characterization, and Classification. Technical Report OH 43210, Department of Computer and Information Science, The Ohio State University, 1997.

[18] R. H. B. Netzer and J. Xu. Necessary and Sufficient Conditions for Consistent Global Snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, 1995.

[19] A. Oliva, I. C. Garcia, and L. E. Buzato. The Reflexive Architecture of Guaraná. Technical Report IC–98–14, Instituto de Computação, Universidade Estadual de Campinas, Abril 1998.

[20] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[21] R. Schwarz and F. Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, 7(3):149–174, Mar. 1994.

[22] S. K. Shrivastava, L. V. Mancini, and B. Randell. The Duality of Fault-tolerant System Structures. *Software—Practice and Experience*, 23(7):773–798, July 1993.

[23] S. K. Shrivastava and S. M. Wheater. Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-Coloured Actions. In *Proceedings of Tenth International Conference on Distributed Computing Systems*, pages 203–210, Paris, France, may 1990.

[24] Sun Microsystems Computer Corporation, Mountain View, CA, USA. *Java API Documentation*, Dec. 1996. Version 1.1.

[25] Y. M. Wang. Maximum and Minimum Consistent Global Checkpoints and Their Applications. In *IEEE Symposium on Reliable Distributed Systems*, pages 86–95, Bad Neuenahr, Germany, Sept. 1995.

[26] Y. M. Wang and W. K. Fuchs. Lazy Checkpoint Coordination for Bounding Rollback Propagation. In *IEEE Symp. on Reliable Distributed Systems (SRDS-12)*, pages 78–85, Oct. 1993.

[27] S. M. Wheater and D. L. McCue. Configuring Distributed Applications using Object Decomposition in an Atomic Action Environment. In J. Kramer, editor, *Proceedings of the International Workshop on Configurable Distributed Systems*, pages 33–44. IEE (UK), Imperial College of Science, Technology and Medicine, UK, March 1992. ISBN 0-85296-544-3.

[28] J. Xu and R. H. B. Netzer. Adaptive independent checkpointing for reducing rollback propagation. In *IEEE Symp. on Parallel and Distributed Processing*, pages 754–761, Dec. 1993.