Detecção dinâmica de condições de disputa para programas *multithreaded* em Java

Clovis Seragiotto Júnior

Dissertação submetida em cumprimento parcial
dos requisitos para obtenção de grau de

Mestre em Ciência da Computação

Orientadora: Prof.ª Dr.ª Dilma Menezes da Silva

(O autor recebeu apoio financeiro da FAPESP durante a elaboração deste trabalho)

Instituto de Matemática e Estatística da USP

São Paulo, outubro de 2000.

Detecção dinâmica de condições de disputa para programas *multithreaded* em Java

Este exemplar corresponde à versão final, devidamente corrigida, da dissertação apresentada por Clovis Seragiotto Júnior e aprovada pela comissão julgadora.

São Paulo, novembro de 2000

Banca examinadora:

Profa. Dra. Dilma Menezes da Silva	IME-USP
Prof. Dr. Alan Mitchel Durham	IME-USP
Prof. Dr. Raimundo José de Araújo Macêdo	UFBA

Resumo

Embora a programação concorrente tenha se popularizado, construir um programa concorrente correto é ainda uma tarefa muito difícil: a falha exibida pelo programa pode ser dependente do escalonamento e apenas raramente se repetir. Nesta dissertação é descrita uma nova ferramenta, chamada Ladybug, capaz de detectar dinamicamente a existência de condições de disputa em programas Java.

Ladybug rescreve classes Java já compiladas, inserindo invocações a métodos de monitoramento. O algoritmo utilizado pelos métodos de monitoramento (Ladybug oferece dois), bem como sua implementação (privilegiando velocidade ou economia de memória), são escolhidos pelo usuário no momento da execução do programa rescrito.

Ladybug foi utilizada com problemas clássicos de concorrência, programas de alunos de graduação da disciplina "Programação Concorrente" , um servidor e um cliente HTTP, e pareceu ser efetiva na descoberta de condições de disputa.

Abstract

Even though concurrent programming has become popular, building a correct concurrent program is still a very difficult task: the failure shown by the program may be scheduling dependent and it may only seldom recur. In this dissertation, a new tool called Ladybug is described. It can dynamically detect the presence of race conditions in Java programs by rewriting classes already compiled and inserting invocations to monitoring methods in them. The algorithm used by the monitoring methods (Ladybug offers two) as well as its implementation (one that privileges speed or saves memory) are chosen by the user when the program is started.

Ladybug has been used with classical concurrent problems, programming assignments from undergraduate students of the Concurrent Programming course, an HTTP client and an HTTP server, and it appeared to be effective in finding race conditions.

Agradecimentos

A todos que, direta ou indiretamente, tornaram possível a realização deste trabalho.

A maioria das espécies de joaninhas está entre as espécies mais benéficas de insetos que conhecemos, pois elas consomem um grande número de insetos que se alimentam de plantas. Este fato e a sua aparência atrativa contribuíram para a opinião em geral boa que a maioria das pessoas tem de joaninhas. Por exemplo, os franceses as chamam de *lesbetes du bon Dieu* (criaturas do bom Deus) e *les vaches de la Vierge* (as vacas da Virgem). Os alemães as chamam de *Marienkäfer* (besouros de Maria) e os ingleses de *Ladybugs*, *Lady Beetles* ou *Ladybirds* (insetos, besouros, pássaros de Nossa Senhora).

Entomology notes © Michigan Entomological Society
http://insects.ummz.lsa.umich.edu/MES/notes/entnotes6.html

Merriam Webster OnLine
http://m-w.com

Índice

Índice de figuras

Introdução

Há algum tempo, a programação concorrente (em particular, a programação *multithreaded*) deixou de ser uma técnica exclusiva da comunidade científica e dos desenvolvedores de sistemas operacionais, e passou a ser utilizada também comercialmente. Programas populares como o navegador Netscape e o processador de textos Microsoft Word são *multithreaded*, e linguagens como C (através de bibliotecas), Java e Delphi oferecem *threads* e todo o aparato necessário para construir programas concorrentes livres de erros.

Embora capaz de expressar as necessidades de certos programas de forma simples e elegante, a programação concorrente também introduz diversos tipos de erros que não são encontrados em programas seqüenciais, e cuja causa contraria a intuição da maioria dos programadores. Muitos desses erros são dependentes do escalonamento seguido e podem se repetir apenas ocasionalmente, levando até mesmo anos para serem descobertos e podendo ser encontrados em bibliotecas já maduras [6][36].

Nesta dissertação, descreveremos Ladybug, uma ferramenta que desenvolvemos capaz de detectar em programas Java concorrentes uma classe de erros conhecida por condição de disputa (embora condições de disputa não sejam de fato erros, e sim a causa potencial deles). Ladybug é composta por dois módulos: um deles é um "rescritor", responsável por inserir invocações a métodos de monitoramento no código de classes compiladas para a máquina virtual Java; o outro contém os métodos de monitoramento, cujas invocações foram inseridas pelo rescritor. Esses métodos são invocados durante a execução do programa rescrito, descobrindo dinamicamente a existência de condições de disputa. Optamos por oferecer uma ferramenta para Java porque seus mecanismos de concorrência têm sido utilizados por uma comunidade cada vez maior de programadores, embora os recursos que Java oferece para detectar erros de sincronização, tanto em tempo de compilação quanto de execução, sejam extremamente precários.

Ladybug foi testada com programas de alunos da disciplina Programação Concorrente, com um servidor HTTP e uma biblioteca para clientes HTTP. Além disso, a degradação trazida pela monitoração foi medida em alguns problemas clássicos de concorrência e também em programas preparados por nós especialmente para este fim.

Este trabalho é organizado como segue. O capítulo 1 descreve o modelo de concorrência utilizado em Java e como um programa Java pode ser afetado por condições de disputa, e o capítulo 2 descreve dois algoritmos para detecção dinâmica de condições de disputa, um proposto por Anne Dinning e Edith Schonberg em 1991, e o outro proposto e implementado por Stefan Savage *et al.* em 1997 (ambos algoritmos necessitam de apenas uma execução para detectar se, para uma determinada entrada, o programa contém condições de disputa). No capítulo 3 é descrito o formato binário aceito pela máquina virtual Java. O capítulo 4 descreve Ladybug: como o rescritor funciona (isto é, quais são as modificações que ele faz em um arquivo compilado) e os métodos de monitoramento invocados (que contêm a implementação dos algoritmos vistos no capítulo 2). O capítulo 5 traz os resultados obtidos pela aplicação de Ladybug a programas e bibliotecas escritos em Java. O capítulo 6 trata de outros trabalhos relacionados à concorrência em Java, seguido pela conclusão, onde discutimos a efetividade da ferramenta.

O apêndice A apresenta um conjunto de classes que tivemos de desenvolver para manipulação de *bytecodes* (não encontramos nenhuma ferramenta ou biblioteca que se adequasse às nossas necessidades ou cujo código fonte pudesse ser alterado), e o apêndice B contém a definição de blocos e de ordem parcial entre blocos que é utilizada em Ladybug.

Capítulo 1

Concorrência em Java

Neste capítulo, o modelo de concorrência adotado em Java é brevemente descrito, e é dada a definição de "condição de disputa", um conceito fundamental em programação concorrente e a razão deste trabalho. O modelo de memória utilizado em Java também é visto, como forma de demonstrar que o problema de condições de disputa pode ser pior do que muitos programadores imaginam.

1.1 Processos e threads

Um processo é uma abstração fornecida pelo sistema operacional de um programa em execução, sendo possível e freqüente que diversos processos estejam (ou pareçam estar) executando simultaneamente. Analogamente, em uma máquina virtual Java, uma *thread* representa um trecho de código em execução, sendo também comum que existam diversas *threads* rodando simultaneamente (ou de forma aparentemente simultânea). Diferente do que normalmente ocorre com processos, porém, a área da qual a memória para todos os objetos é alocada é compartilhada por todas as *threads* e, portanto, quaisquer objetos (como arquivos, conexões de rede ou botões em uma janela) podem ser compartilhados entre as *threads* de uma instância de uma máquina virtual (normalmente, uma máquina virtual Java é um processo em um máquina "real").

Os "trechos de código" que a *thread* executa são métodos de classes. A criação, na máquina virtual, da representação interna de uma classe – e conseqüentemente de seus métodos — é em geral precedida pela carga da classe[1], quando a representação binária da classe é encontrada e lida. Comumente, mas não obrigatoriamente, esta representação binária estará armazenada em um arquivo cujo formato é descrito no Capítulo 2. Também diferente de processos, a representação interna de uma classe é compartilhada entre todas as *threads* da máquina virtual.

[1] A representação interna de matrizes é construída sem o processo de carga.

1.2 Condições de disputa

A memória compartilhada só pode ser lida e alterada através de leituras e escritas a elementos de vetores, variáveis de instância ou variáveis estáticas[2]. Caso duas ou mais *threads* possuam uma referência a um mesmo objeto, elas podem usá-la para, através desse objeto, trocar informações entre si (em Java, esta é a forma mais comum de comunicação entre *threads*). Suponha, por exemplo, que exista na memória compartilhada um objeto p pertencente à classe Ponto:

```
class Ponto {
    int x, y;
    Ponto(int x, int y) { this.x = x; this.y = y; }
}
```

e que duas *threads* tenham uma referência a este objeto. Uma delas atualiza as coordenadas do ponto em intervalos regulares:

```
p.x++;
p.y++;
```

e a outra é responsável por desenhar o ponto, com a mesma freqüência com que ele é atualizado:

```
drawPoint(p.x, p.y);
```

É impossível que a freqüência com que o ponto é atualizado ou desenhado seja mantida rigorosamente constante durante a execução; conseqüentemente, algumas vezes o ponto poderá ser desenhado **enquanto** suas coordenadas são atualizadas, produzindo uma saída como a da figura 1. Ainda que o programador possa não se importar com os "pontos anômalos", em muitas situações é inaceitável que uma *thread* "veja" o estado do objeto durante a sua atualização por outra *thread*.

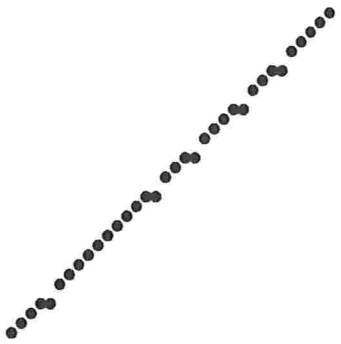


figura 1 – Saída incorreta devido a condições de disputa

Suponha agora que exista uma terceira *thread* que periodicamente reinicializa o ponto, colocando-o em determinada posição:

---

[2] Variáveis estáticas são também chamadas de variáveis de classe; variáveis de instância e estáticas podem ambas ser chamadas apenas de campos.

```
p.x = 0;
p.y = 0;
```

Novamente, caso a reinicialização seja feita **enquanto** as coordenadas são incrementadas pela primeira *thread*, e supondo que o ponto devesse percorrer a trajetória y = x; teríamos que a partir de então todas as posições ocupadas pelo ponto seriam inválidas. A nova trajetória dependeria de qual *thread* "venceu" a disputa pelo acesso aos campos do ponto, o que motiva a seguinte definição:

Uma **condição de disputa** ocorre quando duas (ou mais) *threads* acessam um campo, ao menos um dos acessos é de escrita, e nenhum mecanismo é utilizado para prevenir que os acessos ocorram concorrentemente. [2]

O mecanismo oferecido por Java para impedir que acessos ocorram concorrentemente e, assim, que um programa não apresente condições de disputa, é descrito na próxima seção.

1.3 Locks

Em Java, cada objeto criado tem um *lock* associado (em particular, existe um *lock* associado a cada objeto *Class* representando a classe a que um objeto pertence; neste caso, diremos apenas que o *lock* está associado à classe). A cada instante, um *lock* pode estar sob a posse de no máximo uma *thread*; não estando sob a posse de nenhuma, diremos que o *lock* está **livre**. Um *lock* livre pode ser **adquirido** por uma *thread*, ficando sob sua posse até que a mesma *thread* o **libere**.

Há duas maneiras de uma *thread* adquirir um *lock*: uma é iniciar a execução de um método declarado synchronized (o *lock* é automaticamente liberado quando o método termina); a outra é iniciar a execução de um bloco synchronized (ao término do bloco, o *lock* também é liberado). O código no método ou bloco synchronized é dito **protegido** pelo *lock* que foi adquirido antes da sua execução.

Ao entrar em um método synchronized que tenha sido também declarado static, o *lock* adquirido será aquele associado à classe onde o método foi declarado, caso contrário será o *lock* associado ao objeto para o qual o método foi invocado. Por exemplo, dadas as classes:

```
class C1 {
    static synchronized m1() {
        ...
    }
}
```

```
class C2 {
    synchronized m2() {
        m1();
    }
}
```

e uma instância c da classe C2, então, quando uma *thread* iniciar a execução do método m2 para c, ela irá adquirir o *lock* associado ao objeto c; quando o método m1 iniciar sua execução a partir de m2, a *thread* irá adquirir também a posse do *lock* associado à C1.

No caso de blocos synchronized, o objeto cujo *lock* será adquirido deve ser explicitado. Em

```
synchronized(c) {
    ...
}
```

o *lock* adquirido será aquele associado ao objeto c, enquanto em

```
synchronized(C1.class) {
    ...
}
```

será adquirido o *lock* associado à classe C1.

Enquanto um *lock* está sob a posse de uma *thread*, todas as outras *threads* que tentarem adquirir o mesmo *lock* serão bloqueadas; quando ele for liberado, uma destas *threads* bloqueadas será arbitrariamente escolhida para adquirir o *lock* e prosseguir, e as demais continuarão bloqueadas, à espera de outra chance.

Caso, no exemplo dado anteriormente, os trechos de código que acessavam o ponto p tivessem sido escritos desta forma:

```
synchronized (p) {
    p.x++;
    p.y++;
}
```

```
synchronized (p) {
    drawPoint(p.x, p.y);
}
```

```
synchronized (p) {
    p.x = 0;
    p.y = 0;
}
```

os erros apontados jamais poderiam ter sido exibidos. Por exemplo, o ponto não poderia ser desenhado enquanto suas coordenadas fossem atualizadas, pois para isso o *lock* associado a p deveria estar sob a posse da *thread* responsável por exibir o ponto e sob a posse da *thread*

que atualiza p.x e p.y, o que é proibido. Note, porém, que se um dos trechos não estivesse sob a proteção do mesmo *lock*, ele poderia ser executado concorrentemente com qualquer um dos outros.

Embora do ponto de vista da segurança[3] todo acesso a uma variável compartilhada (isto é, um campo compartilhado ou um elemento de uma vetor compartilhado) devesse ser feito sob a proteção de um mesmo *lock*, há algumas razões que fazem com que o programador queira evitar isso:

- a concorrência diminui, pois as *threads* poderão freqüentemente ser bloqueadas (isto é, não fazer nada) enquanto esperam a chance de adquirir um *lock*;
- o custo de adquirir e liberar um *lock* não é nulo, ainda que na ausência de bloqueios;
- o risco de *deadlock* aumenta. Um conjunto de *threads* está em *deadlock* se cada *thread* no conjunto está bloqueada à espera de um *lock* sob a posse de alguma outra *thread* do conjunto; isto impede definitivamente todas as *threads* desse conjunto de fazerem qualquer progresso.

Há, de fato, casos em que um programa com condições de disputa está realmente correto. Por exemplo, quando uma *thread* estiver executando um trecho de código como o seguinte:

```
while (continua) {...}
```

onde continua é uma variável cujo valor, inicialmente true, será alterado para false por alguma outra *thread* quando o usuário pressionar um botão. O único detalhe nesse caso é a necessidade de declarar o campo continua como volatile; do contrário, a *thread* executando o laço acima poderia manter uma cópia local de continua jamais vendo a alteração feita pela outra *thread*.

1.4 O modelo de memória

Cada *thread* possui uma memória de trabalho na qual pode manter cópias das variáveis compartilhadas, mas certos eventos obrigam-na a carregar um valor mais recente, ou então a enviar o valor da sua cópia para ser escrito na memória compartilhada. Exatamente quais são

[3] Em programação concorrente, segurança significa que as propriedades de correção que se espera de um objeto são alcançadas [18].

esses eventos, a ordem em que pedidos de leitura e escrita devem ser feitos à memória compartilhada e a ordem em que a memória compartilhada atende a esses pedidos é, basicamente, o conteúdo da especificação do modelo de memória Java [21][23][38]. A intenção da especificação era definir um modelo em que a interação entre a memória de trabalho de cada *thread* e a memória compartilhada fosse até certo ponto natural para o programador, mas que ainda assim permitisse otimizações comuns nos processadores (e multiprocessadores) atuais.

Considere, por exemplo, um objeto Ponto compartilhado (referenciado por p), cujas coordenadas iniciais são x = y = 0 e que está prestes a ser acessado sem sincronização por duas *threads*:

Thread 1	*Thread* 2
int a = p.x; int b = p.x	p.x = 1; p.x = 2;

Após a execução dos trechos acima, seria impossível obter os seguintes valores para o par (a, b):

- a = 1, b = 0;
- a = 2, b = 0;
- a = 2, b = 1.

Esses valores são impossíveis porque a especificação proíbe que, em uma mesma *thread*, leituras e escritas de uma mesma variável sejam reordenadas, o que, no exemplo, proíbe que b tenha um valor mais antigo que a[4].

Vamos agora tomar um objeto compartilhado m pertencente à classe Círculo:

```
class Círculo {
    Ponto centro;
    int raio;
}
```

e tal que m.centro.x = m.centro.y = 50. Caso este objeto seja acessado sem sincronização por estas duas *threads*:

[4] A utilidade prática de exigir que duas leituras a uma mesma variável não sejam reordenadas não está clara [39].

Thread 1	*Thread* 2
m.centro = new Ponto(10, 10);	b = m.centro.x;

O valor que b poderá exibir, além de 50 ou 10, poderá também ser 0. Isto porque a especificação permite que variáveis compartilhadas diferentes (no caso, m.centro, m.centro.x e m.centro.y) sejam escritas na memória compartilhada em uma ordem diferente da que foi seguida na memória de trabalho. Portanto, é possível que o novo valor de m.centro seja escrito na memória compartilhada **antes** dos valores atribuídos a m.centro.x e m.centro.y no construtor do Ponto e, no intervalo entre uma escrita e outra, qualquer leitura de m.centro.x e m.centro.y feita por outra *thread* obterá o valor *default* de um campo escalar: 0. (Note que estamos falando apenas de permissões dadas pela especificação; dificilmente, hoje, seria encontrada uma máquina virtual que aproveitasse todas as liberdades oferecidas [13]).

O modelo de memória tem sido vítima de diversas críticas [39], como:

- é difícil de entender;
- os acréscimos feitos ao modelo para corrigir seus *bugs* tornam-no cada vez mais difícil de entender;
- as implicações do modelo no contexto dos multiprocessadores atuais é ainda mais difícil de entender;
- é demasiadamente permissivo para certas coisas (permitindo resultados que poucos programadores julgariam possíveis);
- é demasiadamente restringente em outras (o que dificulta a otimização de código).

Para combater esses males, a utilização de código sincronizado parece ser a melhor solução para o programador:

- quando um *lock* é liberado por uma *thread*, ela é obrigada a enviar à memória compartilhada todos os valores resultados de escritas a variáveis compartilhadas (*flush*);
- quando uma *thread* adquire um *lock*, é garantido que ela verá todas as escritas feitas pelas *threads* que já tenham liberado este ou qualquer outro *lock*.

- caso a especificação mude para permitir algumas otimizações que hoje são proibidas, é provável que o código sincronizado não sofrerá ou quebrará por causa disso.
- é difícil provar que código não sincronizado é correto, já que a especificação é tão complexa.

1.5 Um erro na especificação

Dada a complexidade da especificação, alguns textos foram escritos dando a sua própria "interpretação" do significado do modelo de memória. Infelizmente, as conclusões desses textos não foram as mesmas, e por isso é necessário esclarecer uma propriedade que será utilizada no restante deste texto e que pode ser interpretada diferentemente dependendo da referência utilizada.

Se uma *thread* T inicia uma *thread* P, podemos afirmar que nenhum dos acessos já feitos por T à memória compartilhada pode ocorrer concorrentemente com quaisquer acessos feitos por P (não havendo, portanto, condições de disputa). Analogamente, se T espera o término de P, podemos afirmar que nenhum dos acessos que T fizer à memória compartilhada **após** ter esperado o fim de P pode fazer parte de uma condição de disputa com um acesso feito por P (veja a figura 2).

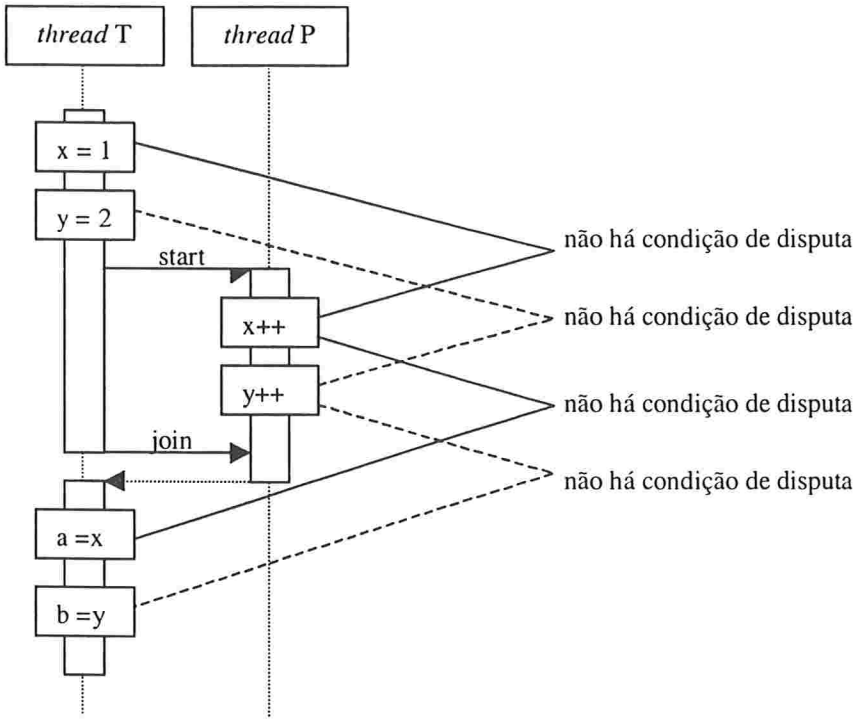


figura 2 – Exemplo sem condições de disputa mas incorreto de acordo com a especificação

Contudo, ainda que não existam condições de disputa, a especificação do modelo de memória original [21] (1996) não obriga que os valores escritos por T antes de iniciar P sejam feitos visíveis para P, nem garante que os valores escritos por P sejam feitos visíveis para T após T ter esperado o término de P. William Pugh (1999) afirma que a comunidade já reconheceu esse comportamento como sendo um erro na especificação [39], e Doug Lea (1999) afirma que essas garantias existem e estariam presentes na próxima especificação do modelo de memória [13]. Contudo, na nova especificação [23] (2000), o problema ainda persiste. Neste trabalho, nós também assumiremos que, de fato, isto é um erro da especificação, e que cedo ou tarde ele será corrigido.

Capítulo 2

Análise dinâmica

Dois algoritmos para detecção dinâmica de condições de disputa são apresentados neste capítulo; suas vantagens e limitações são discutidas. Ambos algoritmos são disponibilizados pela nossa ferramenta de detecção, descrita no Capítulo 4. Propostos em 1991 e 1997, são os algoritmos de análise dinâmica mais recentes e eficientes que pudemos encontrar e que se adaptavam ao modelo de sincronização adotado em Java.

2.1 Motivação

Detectar erros em programas concorrentes pode ser impossível se aplicadas as mesmas técnicas utilizadas na depuração de programas seqüenciais: um erro pode não se repetir ainda que um programa seja executado inúmeras vezes com a mesma entrada que o originou, ou ainda, a inspeção de código pode revelar que o valor exibido por uma variável nunca lhe é atribuído. Isso serviu de motivação ao desenvolvimento de diversas técnicas visando facilitar, ou mesmo automatizar, a busca por propriedades indesejáveis em programas concorrentes, tais como a existência de condições de disputa ou *deadlocks*. Uma delas, conhecida por análise dinâmica, é o tema deste capítulo (outras técnicas são discutidas superficialmente no Capítulo 6). Uma ferramenta de análise dinâmica monitora, **durante a execução do programa**, os acessos a variáveis compartilhadas e outros eventos que julgar úteis, de forma a poder responder se para uma determinada entrada, o programa exibiu condições de disputa (ferramentas de análise estática, por outro lado, tentam detectar condições de disputa sem executar o programa; como será visto no Capítulo 6, a análise estática tende a ser mais lenta, inexata e consumir mais memória do que a análise dinâmica).

Métodos de análise dinâmica são classificados como *on-the-fly* ou *post-mortem*. Os métodos *post-mortem* apenas registram informações do programa durante sua execução; é necessária uma outra fase, após o término da execução, em que os dados obtidos são analisados e os erros encontrados são informados ao usuário. Já métodos *on-the-fly* obtêm dados do programa e informam o usuário sobre condições de disputa durante a execução. Contudo, o monitoramento (e eventual análise) de cada acesso à memória pode prejudicar o desempenho do programa; nesse caso, ferramentas de análise estática (descritas na seção 6.1) podem

auxiliar no processo, eliminando a necessidade de se monitorar todo e qualquer acesso a variáveis compartilhadas.

Em geral, métodos de análise dinâmica podem necessitar, para cada região crítica (isto é, cada região que não deva ser acessada concorrentemente por mais de uma *thread*), de N! execuções para (probabilisticamente) detectar a existência de condições de disputa com uma determinada entrada, onde N é o número de *tasks* simultaneamente em execução [2]. Os métodos descritos a seguir (pertencentes à categoria *on-the-fly*) permitem que, para um grande número de programas o número de execuções caia de N! para 1.

2.2 O algoritmo de Anne Dinning e Edith Schonberg

Em [2], Dinning e Schonberg propuseram um algoritmo capaz de, com apenas uma execução do programa sendo depurado, determinar se, para uma certa entrada, este programa exibe ou não condições de disputa (chamadas por elas de "anomalias de acesso"). O método proposto por elas representava uma melhoria sobre os métodos existentes até então, que se baseavam exclusivamente na relação "aconteceu antes" (*happened before*) definida por Leslie Lamport [30], e que discutiremos antes de descrever o algoritmo de Dinning e Schonberg.

2.2.1 Algoritmos baseados apenas em "aconteceu antes"

Os algoritmos baseados apenas na relação "aconteceu antes" encontram anomalias analisando um grafo dirigido, chamado POEG (*Partial Order Execution Graph*), que modela o programa em execução e reflete a relação "aconteceu antes":

- cada vértice do POEG é ou uma operação concorrente (*lock*, *unlock*, crie *task*[5], termine *task*) ou um bloco de instruções sem qualquer operação concorrente;
- em uma mesma *task*, arestas conectam vértices de maneira a refletir a ordem de execução;
- entre *tasks* concorrentes, há arestas que ligam cada operação *unlock*, que libera um *lock*, com a próxima operação *lock* que efetivamente o adquire (veja a figura 3).

[5] Uma *task* é uma linha de execução independente, seja ela uma *thread*, um processo ou qualquer outra abstração fornecida pelo ambiente de programação.

Num POEG, dizemos que dois vértices estão ordenados se existe um caminho entre eles, e dizemos que existe uma anomalia de acesso se há uma variável que é acessada em dois vértices não ordenados, e ao menos um acesso é de escrita.

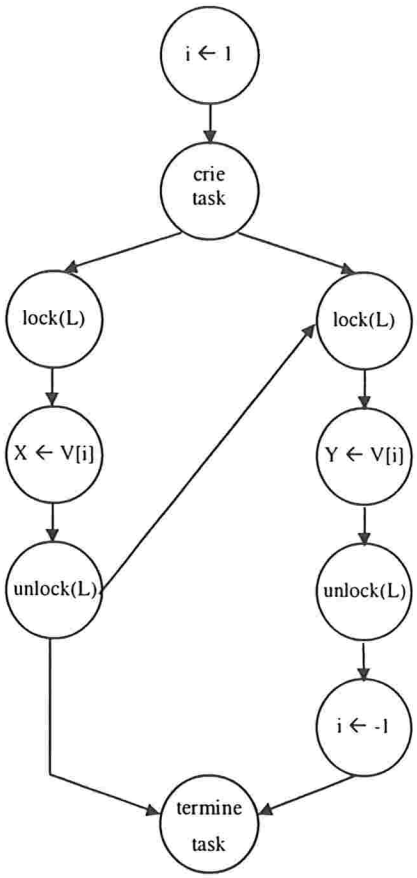


figura 3 – Exemplo de POEG

O grafo nunca é explicitamente construído: sempre que um bloco começa a executar, ele recebe um identificador, gerado de maneira que seja eficiente calcular *Ordenado(a, b)*, onde *a* e *b* são identificadores. Além disso, cada variável *X* guarda dois conjuntos representando seu histórico de acessos: um com os identificadores dos blocos que já a acessaram para leitura — *Readers(X)* — e outro com os que já a acessaram para escrita — *Writers(X)*.

O usuário é avisado de que há uma anomalia quando uma variável *X* é acessada num bloco de identificador *id-atual* e:

- o acesso é de leitura **e** *Ordenado(a, id-atual)* é falso para ao menos um elemento *a* do conjunto *Writers(X)*; ou
- o acesso é de escrita **e** *Ordenado(a, id-atual)* é falso para ao menos um elemento *a* da união dos conjuntos *Readers(X)* e *Writers(X)*.

É possível apagar identificadores antigos do histórico de acessos de uma variável. No caso de *Writers(X)*, é suficiente manter o identificador do último bloco que escreveu em *X*, e em *Readers(X)*, não é preciso guardar dois identificadores ordenados, basta o mais recente. Com essa técnica garante-se que, se há anomalias, pelo menos uma será relatada.

O problema de se utilizar um POEG é que as arestas "fictícias" que ligam os vértices *unlock* aos vértices *lock* representam uma ordenação arbitrária de uma execução particular, ou seja, podem esconder algumas anomalias que seriam acusadas em outras execuções (por exemplo, o bloco "i ← -1" na figura 3 é uma anomalia de acesso, que está invisível no POEG).

Por outro lado, são essas arestas que garantem que variáveis acessadas dentro de regiões críticas protegidas pelo mesmo *lock* não serão apontadas como anomalias.

2.2.2 A melhoria

Dinning e Schonberg propuseram o uso de coberturas de *locks* para substituir as arestas "fictícias". Coberturas de *locks* garantem que anomalias são relatadas para todas as variáveis acessadas sem uma proteção consistente de *locks* e, ainda, que não são apontadas anomalias quando variáveis protegidas por um mesmo *lock* são acessadas.

A cobertura de *locks* associada a um bloco é o conjunto de *locks* que esse bloco retém ao ser executado, e diz-se que há uma anomalia de acesso se existir uma variável que é acessada em dois blocos não ordenados, ao menos um dos acessos é de escrita e não existe nenhum *lock* que pertença à cobertura de ambos os blocos (essa definição de anomalia **não** é equivalente à primeira). Por exemplo, na figura 3, a cobertura de *locks* dos blocos "X ← V[i]" e "Y ← V[i]" é o conjunto {L}, enquanto a cobertura dos blocos "i ← 1" e "i ← -1" é o conjunto vazio. Os blocos "Y ← V[i]" e "i ← -1" estão ordenados, logo este par não pode ser culpado por nenhuma anomalia de acesso. Já os blocos "X ← V[i]" e "i ← -1" não estão ordenados nem têm nenhum *lock* em comum, sendo, portanto, responsáveis pela existência de uma anomalia de acesso (que será descoberta não importa qual escalonamento seja seguido).

A implementação de um algoritmo *happened before* com esta modificação é semelhante à descrita na seção anterior, mas, além do identificador, também devem ser guardados os *locks* que protegem cada bloco. Porém, excluir identificadores antigos do histórico de acessos de uma variável torna-se mais difícil. Na figura 4 (onde $\leftarrow_K$ representa uma escrita protegida pelo *lock* K), suponha que os blocos com identificadores B_1, B_2, B_3 e B_4 sejam executados nesta ordem. Se, quando B_2 for executado, a entrada (B_2, {L, M}) for acrescentada ao histórico de *X* e a entrada (B_1, {L}) for simplesmente removida, então, quando B_3 for executado, a anomalia existente entre B_1 e B_3 será perdida. Também seria incorreto supor que é possível excluir (B_1, {L}) desde que utilizemos a intersecção das

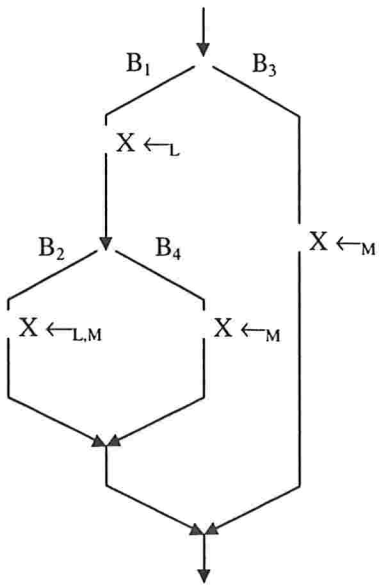


figura 4 – Quando é preciso cuidado ao excluir entradas do histórico de uma variável

coberturas de B_1 e B_2 na nova entrada — isto é, $(B_2, \{L\})$ ao invés de $(B_2, \{L, M\})$: nesse caso seria acusada uma anomalia inexistente quando B_4 fosse executado.

Assim, chamando de *id-atual* o identificador do bloco atual e de *locks-atuais* a cobertura de *locks* que o protege, o par $\langle id, locks \rangle$ associado a um bloco anterior só pode ser excluído do histórico se *id* e *id-atual* estão ordenados **e** $locks\text{-}atuais \subseteq locks$.

2.2.3 Limitações

Embora represente uma melhoria sobre os algoritmos anteriores, a proposta de Dinning e Schonberg tem limitações: para programas contendo indeterminismo interno, isto é, em que há acessos a posições compartilhadas de memória determinados pela ordem de execução das regiões críticas, anomalias existentes podem ser perdidas pelo algoritmo, como ilustra o seguinte exemplo:

```
        int j;
        synchronized (lock) {
◊           A[i] = ...;
            j = global++;
        }
        if (j == 1) {
◊◊          A[1] = ...;
        }
```

Suponha que o vetor A e a variável global (cujo valor inicial é 1) sejam compartilhadas e que duas *threads*, T_1, com i = 1, e T_2, com i = 2, executem esse trecho de código. Se a *thread* T_2 executar a região crítica primeiro, então será acusada uma condição de disputa (linhas ◊ com *thread* T_1 e ◊◊ com *thread* T_2), caso contrário, a condição de disputa será perdida.

Já o seguinte trecho de código, quando executado por diversas *threads*, serve para exemplificar a ocorrência de alarmes falsos (locked e p são variáveis compartilhadas):

```
// protocolo de entrada na região crítica
synchronized(p) {
    while (locked) wait();
    locked = true;
}

// região crítica
p.x++;
p.y++;

// protocolo de saída da região crítica
synchronized(p) {
    locked = false;
    notify();
}
```

Aqui, o programador utilizou seu próprio protocolo de entrada e saída da região crítica; o algoritmo, porém, não é capaz de detectar que este protocolo tem como efeito ordenar todos os acessos a p.x e p.y, e acusará uma condição de disputa inexistente.

2.3 Eraser

Eraser [36] é uma ferramenta para detectar dinamicamente condições de disputa em programas C *multithreaded*. O algoritmo que Eraser utiliza, chamado LockSet, é uma modificação daquele descrito na seção anterior; a diferença é que em Eraser, por questões de eficiência, é ignorada a relação "aconteceu antes" e a distinção entre acessos de leitura e escrita.

Para cada variável compartilhada *v*, Eraser mantém um conjunto *C(v)* de todos os *locks* que têm protegido *v*. O conjunto é (conceitualmente) iniciado com todos os *locks* possíveis e é atualizado sempre que *v* é acessada, fazendo a sua intersecção com o conjunto de *locks* que protegem *v* no momento do acesso. Se *C(v)* torna-se vazio, então foi detectada uma condição de disputa (pois não há nenhum *lock* que consistentemente proteja *v*).

A inexistência de um *lock* que proteja todos acessos a determinada variável não é condição suficiente para a existência de condições de disputa, como mostra o exemplo a seguir (em Java, não C):

Thread 1	*Thread* 2	*Thread* 3

synchronized(a) {
  synchronized(b) {
  draw(p.x, p.y);
  }
}
	synchronized(b) {
	  synchronized(c) {
	  p.x++; p.y++
	  }
	}
		synchronized(a) {
		  synchronized(c) {
		  p.x = p.y = 10;
		  }
		}

Estes trechos são livre de condições de disputa, embora não haja nenhum *lock* que proteja todos os acessos a p.x e p.y. Porém, aninhar regiões críticas é uma prática propensa a erros e que portanto devia ser evitada por programadores [2]. Há, contudo, práticas comuns de programação que são livres de condições de disputa, mas violam a disciplina de proteção consistente que, de acordo com Eraser, deveria ser seguida:

- atribuição do valor inicial às variáveis compartilhadas, o que normalmente é feito sem proteção;
- variáveis compartilhadas apenas para leitura, que podem ser acessadas seguramente sem *locks*;
- variáveis compartilhadas que podem ser lidas por várias *threads* simultaneamente, mas que só podem ter uma única *thread* atualizando-as[6].

Para tratar das duas primeiras práticas (que são um problema devido à decisão de Eraser de ignorar tanto a ordenação entre eventos quanto a separação entre acessos de leitura e de escrita), o algoritmo LockSet também mantém um estado para cada variável compartilhada. Há quatro estados possíveis (veja a figura 5): **virgem** (a variável ainda não foi acessada), **exclusiva** (a variável até o momento só foi acessada por uma *thread*), **compartilhada** (a variável foi acessada por mais de uma *thread*, mas da segunda *thread* em diante todos os acessos foram de leitura) e **compartilhada-modificada** (a variável já foi escrita por mais de uma *thread*). *C(v)* é atualizado apenas nos estados **compartilhada** e **compartilhada-modificada**; os avisos sobre condições de disputa ocorrem apenas no estado **compartilhada-modificada**.

[6] É um problema clássico de comunicação entre processos, conhecido por "*Multiple readers, single writer*" [1].

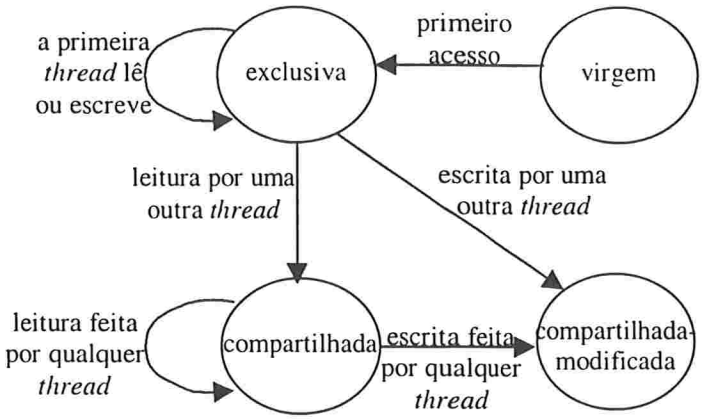


figura 5 – Estados de uma variável em Eraser

Entretanto, essa maneira de tratar a inicialização de variáveis torna o algoritmo dependente do escalonamento, podendo ocasionar a perda de algumas condições de disputa. O trecho seguinte (escrito em Java, não C) exemplifica este fato; a condição de disputa entre as linhas ◊ e ◊◊ não seria relatada pelo Eraser caso a linha ◊◊ fosse executada antes de ◊.

```
    // variáveis compartilhadas:
    double pi;
    Thread t = new Thread() {
        public void run() {
◊           float v = pi;
            ...
        }
    };
    ...
    // início do código de inicialização
    pi = 2.14;
    t.start();
◊◊  pi++;
    // fim do código de inicialização
    ...
```

Para cuidar do último caso, o algoritmo LockSet define dois modos em que um *lock* pode ser adquirido: **escrita** e **leitura**. Quando adquirido em modo **escrita**, o *lock* pode estar sob a posse de no máximo uma *thread*; já quando adquirido em modo leitura, pode estar sob a posse de várias *threads* (desde que nenhuma delas o adquira em modo **escrita**). Note que um *lock* "normal", isto é, que pode estar sob a posse de no máximo uma *thread* em determinado instante, tem a mesma semântica de um *lock* sempre adquirido em modo **escrita**. Sabendo o modo em que um *lock* foi adquirido, a atualização do conjunto *C(v)* é feita de acordo com as seguintes regras:

- se o acesso é de leitura, *C(v)* passa a ser a sua intersecção com o conjunto de *locks* que, em qualquer modo, protegem *v* no momento do acesso;
- se o acesso é de escrita, *C(v)* passa a ser a sua intersecção com o conjunto de *locks* que, em modo de escrita, protegem *v* no momento do acesso (*locks* adquiridos em modo leitura são ignorados pois eles não devem ser utilizados para proteger escritas a uma variável).

A implementação do Eraser é feita com o auxílio de uma ferramenta para modificação de binários para o processador Alpha, inserindo-se código no programa compilado original sempre que:

- um *lock* é adquirido ou liberado;
- há um acesso (*load* ou *store*) a uma variável;
- uma *thread* é iniciada ou finalizada;
- é alocada memória dinamicamente.

Não é mantido um conjunto de *locks* para cada posição compartilhada da memória. Ao invés disso, Eraser associa a cada conjunto distinto de *locks* um inteiro (o número de conjuntos nunca passou de dez mil nos testes realizados pelos autores). Novos conjuntos podem ser criados quando um *lock* é adquirido ou liberado ou como resultado de intersecções, mas nunca são destruídos. Assim, para cada posição de memória (palavra de 32 bits na área de dados ou no *heap*) há um inteiro associado, com 30 bits para indicar o índice do conjunto de *locks* e 2 para indicar o estado da variável (no estado **exclusiva**, os 30 bits contém o ID da única *thread* que acessou a variável).

Alarmes falsos podem ser evitados inserindo-se anotações no código fonte. A experiência mostrou que os alarmes falsos normalmente caíam em uma destas três categorias:

- a memória é reaproveitada sem que o Eraser saiba, pois o programa tem suas próprias rotinas de alocação de memória;
- condições de disputa benignas (existem realmente, mas não afetam a correção[7] do programa);

[7] Esta dissertação foi escrita consultando-se dois dicionários brasileiros (Aurélio 2ª edição e www.uol.com/michaelis), um dicionário português (www.portoeditora.pt) e o Vocabulário Ortográfico da Língua

- implementação particular de *locks* para o problema de "múltiplos leitores, um atualizador" (isto é, não é usada a interface *pthread* [1], a única reconhecida pelo Eraser).[8]

A reutilização da memória pode ser comunicada ao Eraser com *EraserReuse(endereço, tamanho)*. A implementação particular de *locks*, com *EraserReadLock(lock)*, *EraserReadUnlock(lock)*, *EraserWriteLock(lock)* e *EraserWriteUnlock(lock)*. Para impedir que o Eraser avise sobre condições de disputa benignas, *EraseIgnoreOn()* e *EraseIgnoreOff()* podem ser utilizadas.

Eraser foi testado com o serviço de busca na WEB AltaVista, o servidor de cache Vesta e o sistema de discos distribuídos Petal. As condições de disputa encontradas no AltaVista eram todas benignas e puderam ser eliminadas com anotações. Em Vesta, foi encontrada uma condição de disputa que realmente era um erro. Após a sua correção e a inserção de anotações para inibir avisos sobre condições benignas de disputa, o número de avisos dados pelo Eraser (mais de uma centena) caiu para zero. Já no caso de Petal, houve um aviso que não pôde ser eliminado mesmo com anotações. Os autores do Eraser ainda pediram ao autor de uma das componentes do AltaVista que inserisse novamente duas condições de disputa que tinham existido ali por meses até serem manualmente detectadas. Sem qualquer informação sobre onde estavam as condições de disputa ou o que as causava, os erros inseridos foram corrigidos em 30 minutos.

Eraser também foi testado em trabalhos de alunos de graduação da Universidade de Washington. Excluindo aqueles que nem compilavam ou que imediatamente entravam em *deadlock*, 10% dos trabalhos tiveram condições de disputa apontadas pelo Eraser, que eram realmente erros.

Portuguesa (www.academia.org.br/vocabula.htm). A palavra "corretude" não foi encontrada em nenhum deles (diferente, por exemplo, de "inicialização").

[8] Embora Eraser trabalhe com a biblioteca Pthreads, nada foi encontrado nessa biblioteca que permita que um *lock* seja adquirido em modo leitura ou escrita.

Capítulo 3

O formato do arquivo .class

Neste capítulo será descrita a representação binária aceita pela máquina virtual Java de uma classe ou interface [26][38]. Essa representação é independente de *hardware* e sistema operacional, e será chamada aqui de **arquivo .class**, por estar normalmente (embora não obrigatoriamente) armazenada em um arquivo com a extensão .class. É fundamental entender essa representação antes de construir uma ferramenta capaz de manipulá-la.

3.1 Definições

Não apenas a descrição do formato de um arquivo .class, mas também o restante deste trabalho poderá referir-se às seguintes áreas de dados da máquina virtual Java:

- *heap*: área da qual é alocada memória para todos os objetos. É criada quando a máquina virtual é iniciada, sendo compartilhada por todas as *threads*.
- área de métodos: área onde são armazenados dados das classes ou interfaces carregadas. Como o *heap*, é criada quando a máquina virtual é iniciada, e é compartilhada por todas as *threads*.
- tabela de constantes de execução: armazena símbolos de uma classe, aos quais a própria classe ou outras classes podem se referir. É alocada na área de métodos.
- pilha da máquina virtual: área onde os registros de ativação (descritos a seguir) são armazenados. Cada *thread* tem uma pilha, criada quando a *thread* é iniciada.
- registro de ativação: área criada quando um método é invocado, e destruída quando a invocação termina, seja de forma normal ou abrupta (isto é, se uma exceção for lançada). A memória para o registro de ativação é alocada da pilha da máquina virtual da *thread* que invocou o método. O registro de ativação é composto de uma tabela de variáveis locais, uma pilha de operandos (ambas descritas a seguir) e uma referência à tabela de constantes de execução da classe onde o método foi declarado.
- tabela de variáveis locais: armazena as variáveis locais de um método, inclusive os argumentos que o método recebeu. Variáveis do tipo long ou double ocupam duas entradas nessa tabela.

- pilha de operandos: armazena resultados parciais computados por um método, bem como os parâmetros para métodos que serão invocados e o valor de retorno desses métodos. Valores do tipo long ou double ocupam duas posições na pilha.

O formato de todas essas áreas é dependente da implementação da máquina virtual.

3.2 Descrição do formato do arquivo

Um arquivo .class é uma seqüência de bytes. Valores de 2, 4 ou 8 bytes são construídos pela concatenação de bytes consecutivos, que estarão sempre armazenados no formato *big-endian* (bytes de mais alta ordem primeiro). A descrição do formato de um arquivo .class, apresentada a seguir, utilizará uma notação semelhante à de uma estrutura da linguagem C. Além disso, será usada a notação de matrizes para representar as muitas tabelas que um arquivo .class contém (como os elementos dessas tabelas normalmente têm comprimento variável, elas não podem ser vistas exatamente como matrizes da linguagem C).

A notação **u1**, **u2** e **u4** será usada para representar quantidades sem sinal (*unsigned*) de, respectivamente, 1, 2 e 4 bytes.

```
ClassFile {
    u4 número_mágico;
    u2 versão_menor;
    u2 versão_maior;
    u2 número_de_constantes_mais_1;
    Constante tabela_de_constantes[número_de_constantes_mais_1 – 1];
    u2 flags_de_acesso;
    u2 índice_desta_classe;
    u2 índice_da_superclasse;
    u2 número_de_interfaces;
    u2 tabela_de_interfaces[número_de_interfaces];
    u2 número_de_campos;
    Campo tabela_de_campos[número_de_campos];
    u2 número_de_métodos;
    Método tabela_de_métodos[número_de_metodos];
    u2 número_de_atributos;
    Atributo tabela_de_atributos[número_de_atributos];
}
```

- número_mágico:

  Os 4 primeiros bytes de um arquivo .class contêm um "número mágico", que deve ser sempre 0xCAFEBABE.

- versão_menor, versão_maior:

  Representam a versão do formato do arquivo .class. Para a versão 1.2 da plataforma 2 do Java, o número maior da versão deve ser 45 ou 46, e o número menor deve ser 0 — se o número maior for 46 — ou qualquer outro — se o número maior for 45.

- número_de_constantes_mais_1:

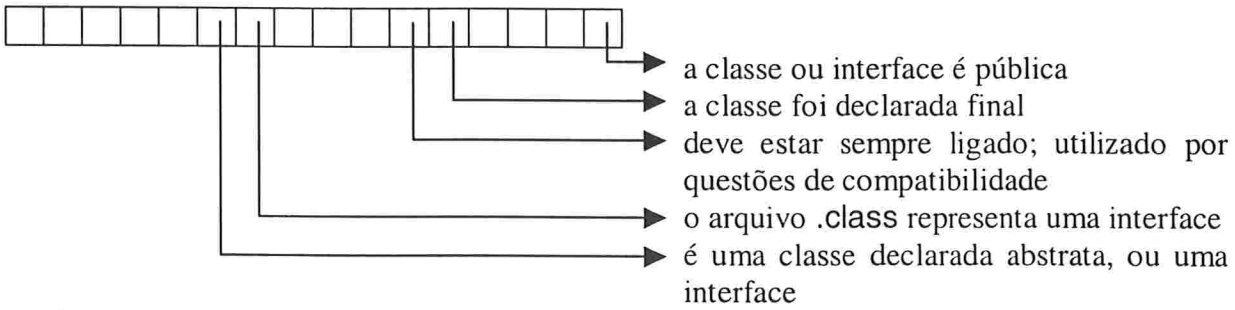
  Contém o número de entradas de tabela_de_constantes, mais 1.

- tabela_de_constantes:

  É uma tabela de estruturas, indexada de 1 a número_de_constantes_mais_1 – 1, contendo constantes e símbolos a que o arquivo .class se refere (como métodos e campos), sendo utilizada para construir a tabela de constantes de execução. O formato dessa tabela está detalhado na seção 3.3.

- flags_de_acesso:

  É uma máscara contendo algumas propriedades da classe ou interface que o arquivo .class representa. A figura seguinte descreve o significado dos bits dessa máscara, quando ligados:



- índice_desta_classe:

  Índice, na tabela de constantes, da constante que simboliza a classe ou interface definida pelo arquivo .class.

- índice_da_superclasse:

  Se o arquivo .class representa a classe java.lang.Object, deve ser 0; caso represente uma outra classe, deve ser o índice, na tabela de constantes, da constante que simboliza a superclasse da classe definida pelo arquivo.

  Se o arquivo .class representa uma interface, então índice_da_superclasse deve ser o índice da constante que simboliza a classe java.lang.Object na tabela de constantes.

- número_de_interfaces:

  Contém o número de entradas da tabela_de_interfaces.

- tabela_de_interfaces:

  É uma tabela que contém índices, na tabela de constantes, das entradas que simbolizam as interfaces implementadas ou estendidas pela classe ou interface que o arquivo .class representa. É indexada de 0 a número_de_interfaces – 1.

- número_de_campos:

  Contém o número de entradas da tabela_de_campos.

- tabela_de_campos:

  É uma tabela, indexada de 0 a número_de_campos – 1, contendo informações sobre os campos da classe ou interface representada pelo arquivo .class. Campos herdados não estão presentes nessa tabela, descrita com mais detalhes na seção 3.4.

- número_de_métodos:

  Contém o número de entradas da tabela_de_métodos.

- tabela_de_métodos:

  É uma tabela, indexada de 0 a número_de_métodos – 1, contendo informações sobre os métodos da classe ou interface representada pelo arquivo .class. Métodos herdados não estão presentes nessa tabela, descrita com mais detalhes na seção 3.4.

- número_de_atributos:

  Contém o número de entradas da tabela_de_atributos.

- tabela_de_atributos:

  É uma tabela, indexada de 0 a número_de_atributos – 1, contendo informações extras sobre a classe ou interface que o arquivo .class representa. Tabelas de atributos estão descritas com mais detalhes na seção 3.5.

3.3 A tabela de constantes

Cada entrada na tabela de constantes representa algum símbolo ou constante a que o arquivo .class se refere em algum momento. Por exemplo, se no corpo do método de alguma classe existe o código

```
System.out.println(3.9);
```

ao compilar esta classe, será necessário gerar entradas na tabela de constantes que simbolizem a classe System, a variável estática out, o método println, a classe em que o método println foi declarado, e a constante 3.9.

Os nomes de classes são representados na tabela de constantes na sua forma **completamente qualificada**, isto é, contendo o nome do pacote ao qual a classe pertence. Além disso, por razões históricas, são utilizadas barras, e não pontos, para separar o nome de pacotes e subpacotes. Assim, o nome da classe java.lang.Object é representado por java/lang/Object.

O tipo de um campo é representado por uma cadeia de caracteres chamada **descritor do campo**, codificado de acordo com a seguinte tabela:

cadeia	**tipo**
B	byte
C	char
D	double
F	float
I	int
J	long
L⟨nome da classe⟩;	referência
S	short
Z	boolean
[	referência a uma dimensão de uma matriz

onde ⟨nome da classe⟩ é o nome de uma classe completamente qualificado e utilizando barras como separadores. Por exemplo, um campo do tipo double tem seu tipo representado pela letra D, um campo do tipo double[][] tem seu tipo representado por [[D, e um campo do tipo java.util.List tem seu tipo representado por Ljava/util/List;.

De maneira semelhante, o **descritor de um método** codifica o tipo dos argumentos e o valor de retorno de um método. Ele é composto de um sinal "abre parênteses", seguido de zero ou mais caracteres representando o tipo dos argumentos do método, um sinal "fecha

parênteses" e uma cadeia de caracteres representando o tipo do valor retornado. A codificação utilizada para descrever os tipos dos argumentos e do valor de retorno é a mesma utilizada para descrever o tipo de um campo, com uma única diferença: V é utilizado para representar o valor de retorno de um método que não retorna valor algum. Como exemplo, o descritor do método

```
void foo(long, java.awt.Component, int);
```

será (JLjava/awt/Component;I)V.

A estrutura de cada entrada na tabela de constantes é diferente, sendo determinada pelo seu primeiro byte. As estruturas básicas são as seguintes:

- Constante_Utf8 {
  u1 tag;
  u1 cadeia_de_caracteres[ ];
  }

  tag, o primeiro byte da estrutura, deve ser 1. Os bytes restantes conterão uma cadeia de caracteres codificada no formato conhecido por Utf8 [38], ligeiramente modificado[9].

- Constante_Integer {
  u1 tag;
  u4 valor;
  }

  tag deve ser 3. Os outros quatro bytes representam uma constante inteira de 32 bits, em formato *big-endian*.

- Constante_Float {
  u1 tag;
  u4 valor;
  }

  tag deve ser 4. Os outros quatro bytes contém o valor de uma constante float, representada no formato de precisão simples IEEE 754 [38].

[9] Diferente do formato Utf8 original, a codificação usada utiliza 2 bytes, e não 1, para representar o byte nulo (0). Além disso, todos caracteres são codificados utilizando no máximo 3 bytes.

- Constante_Long {
    u1 tag;
    u4 bytes_de_alta_ordem;
    u4 bytes_de_baixa_ordem;
  }

tag deve ser 5. Os outros oito bytes representam uma constante long de 64 bits, em formato *big-endian*. Uma constante long é considerada como ocupante de duas posições na tabela de constantes, isto é, a existência de uma constante long na posição n torna obrigatória a existência da entrada ⟨n + 1⟩, embora não possa haver em nenhum ponto do arquivo .class uma referência a esta entrada extra.

- Constante_Double {
    u1 tag;
    u4 bytes_de_alta_ordem;
    u4 bytes_de_baixa_ordem;
  }

tag deve ser 6. Os oito bytes restantes contém o valor de uma constante double, representada no formato de precisão dupla IEEE 754. Como ocorre com a Constante_Long, a Constante_Double também "ocupa" duas entradas na tabela de constantes.

As demais estruturas são compostas apenas de índices de outras entradas na própria tabela de constantes:

- Constante_Classe {
    u1 tag;
    u2 índice_do_nome_da_classe;
  }

Simboliza uma classe ou interface. tag deve ser 7, e o índice deve referir-se a uma Constante_Utf8 contendo o nome, completamente qualificado e utilizando barras como separadores, da classe ou interface sendo simbolizada.

- Constante_String {
    u1 tag;
    u2 índice_do_valor;
  }

Representa uma constante String. tag deve ser 8, e o índice deve referir-se a uma Constante_Utf8 contendo o valor da String.

- Constante_Nome_e_tipo {
    u1 tag;
    u2 índice_do_nome;
    u2 índice_do_descritor;
  }

Representa um campo ou método sem especificar sua classe. tag deve ter o valor 12; índice_do_nome indica a entrada da tabela com a Constante_Utf8 que fornece o nome do campo ou método, e índice_do_descritor indica a entrada com a Constante_Utf8 que contém o descritor do campo ou método.

- Constante_Campo {
    u1 tag;
    u2 índice_da_classe_ou_interface;
    u2 índice_do_nome_e_tipo;
  }

Representa um campo de uma classe ou interface. tag deve ter o valor 9; índice_da_classe_ou_interface deve fornecer a entrada na tabela com a Constante_Classe que simboliza a classe ou interface onde o campo foi declarado[10], e índice_do_nome_e_tipo deve indicar qual entrada contém a Constante_Nome_e_tipo com o nome e o descritor do campo.

- Constante_Método {
    u1 tag;
    u2 índice_da_classe;
    u2 índice_do_nome_e_tipo;
  }

Representa um método de uma classe, de forma semelhante a uma Constante_Campo (exceto pelo fato de que tag deve ter o valor 10).

[10] Na verdade, em algumas situações é permitido que ⟨índice_da_classe_ou_interface⟩ refira-se a uma subclasse (ou subinterface) da classe (ou interface) na qual o campo foi realmente declarado.

- Constante_Método_de_interface {
    u1 tag;
    u2 índice_da_interface;
    u2 índice_do_nome_e_tipo;
  }

Representa um método de uma interface, de forma semelhante a uma Constante_Campo ou a uma Constante_Método (exceto pelo fato de que tag deve ter o valor 11).

3.4 Campos e métodos da classe

Um arquivo .class contém uma tabela dos campos e outra dos métodos declarados pela classe ou interface que ele representa (eventualmente, qualquer uma dessas tabelas pode ter tamanho zero). As estruturas

```
Campo {
    u2 flags_de_acesso;
    u2 índice_do_nome;
    u2 índice_do_descritor;
    u2 número_de_atributos;
    Atributo tabela_de_atributos[número_de_atributos];
}
```

e

```
Método {
    u2 flags_de_acesso;
    u2 índice_do_nome;
    u2 índice_do_descritor;
    u2 número_de_atributos;
    Atributo tabela_de_atributos[número_de_atributos];
}
```

compõem, respectivamente, a tabela de campos e a tabela de métodos. O significado de cada elemento da estrutura é semelhante para campos e métodos:

- flags_de_acesso é uma máscara cujos bits indicam quais modificadores foram utilizados na declaração do campo ou método. A figura 6 mostra o significado dos bits da máscara, quando ligados[11].

[11] A semântica de cada um dos modificadores pode ser encontrada em [23] e [38].

- índice_do_nome é o índice, na tabela de constantes, da Constante_Utf8 contendo o nome do campo ou método.
- índice_do_descritor é o índice, na tabela de constantes, da Constante_Utf8 contendo o descritor do campo ou método.
- número_de_atributos contém o número de entradas na tabela_de_atributos.
- tabela_de_atributos, indexada de 0 a número_de_atributos – 1, contém informações extras sobre o campo ou método. A tabela de atributos é descrita na próxima seção.

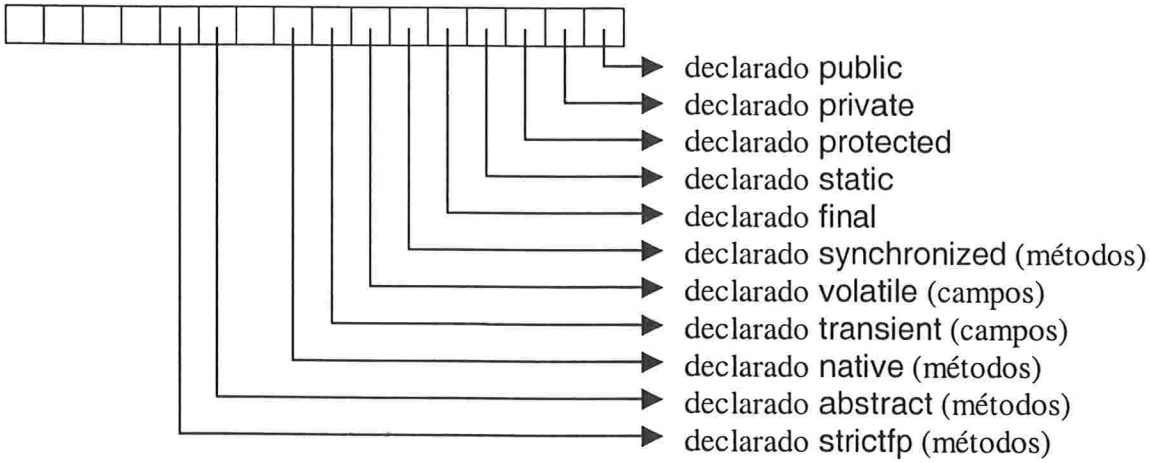


figura 6 – Máscara dos modificadores de um campo ou método

3.5 Atributos

Atributos fornecem informações extras sobre uma classe, um campo, um método e, eventualmente, sobre um outro atributo. A estrutura genérica de cada atributo presente em uma tabela de atributos é esta:

```
Atributo {
    u2 índice_do_nome;
    u4 tamanho;
    u1 info[tamanho];
}
```

onde

- índice_do_nome é um índice, na tabela de constantes, de uma Constante_Utf8 contendo o nome do atributo.
- tamanho fornece o tamanho total do atributo, excluindo os seis bytes iniciais.

- info contém o restante do atributo.

Atualmente, nove atributos estão definidos juntamente com a especificação do formato do arquivo .class: Code, ConstantValue, Deprecated, Exceptions, InnerClasses, LineNumberTable, LocalVariableTable, Synthetic e SourceFile. Destes, cinco devem ser reconhecidos por qualquer implementação da máquina virtual Java: Code, ConstantValue, Exceptions, InnerClasses e Synthetic. Nada impede, porém, que um fabricante crie um atributo que adicione funcionalidade a uma classe, e que uma implementação de máquina virtual Java reconheça e utilize esse atributo. Por outro lado, não é permitido que a classe ou interface representada pelo arquivo .class tenha sua semântica alterada quando utilizada por outra máquina virtual que não reconhece o atributo, nem é permitido a uma máquina virtual rejeitar um arquivo .class que não contenha algum atributo não definido pela especificação.

Um atributo pode não ser aplicável em todas as situações. É inválido para a tabela de atributos de um campo, por exemplo, conter o atributo Code. Os atributos definidos pela especificação e sua aplicabilidade estão descritos a seguir.

- Code {
  u2 índice_do_nome;
  u4 tamanho;
  u2 tamanho_máximo_da_pilha;
  u2 número_de_variáveis_locais;
  u4 tamanho_do_código;
  u1 código[tamanho_do_código];
  u2 tamanho_da_tabela_de_exceções;
  Exceção tabela_de_exceções[tamanho_da_tabela_de_exceções];
  u2 número_de_atributos;
  Atributo tabela_de_atributos[número_de_atributos];
  }

  Contém o código de um método que não seja nativo nem abstrato.

  ⇒ Na posição índice_do_nome da tabela de constantes deve haver uma Constante_Utf8 contendo o palavra "Code".

  ⇒ tamanho deve conter o tamanho do atributo excluindo os 6 bytes iniciais.

  ⇒ tamanho_máximo_da_pilha contém o tamanho máximo que a pilha de operandos pode atingir durante a execução do método; número_de_variáveis_locais fornece o tamanho da tabela de variáveis locais.

⇒ tamanho_do_código informa o tamanho do vetor código, que contém as instruções da máquina virtual Java relativas ao método ao qual o atributo Code está associado. A posição da instrução no vetor é chamada *offset* da instrução mas, como nem todas as instruções tem o tamanho de um byte, dificilmente ocorrerá que o *offset* da n-ésima instrução seja também n. Assim, o *offset* da primeira instrução é 0, o da segunda será *offset* da primeira instrução + tamanho da primeira instrução, e assim por diante.

⇒ tamanho_da_tabela_de_exceções contém o número de entradas da tabela_de_exceções. Cada entrada da tabela_de_exceções contém informações sobre como a máquina virtual deve proceder se uma certa região do código lançar uma determinada exceção. A estrutura de cada entrada é esta:

```
Exceção {
    u2 offset_inicial;
    u2 offset_final;
    u2 offset_tratamento;
    u2 índice_da_classe_da_exceção;
}
```

Suponha que uma das entradas da tabela de exceções tenha uma entrada em que offset_inicial valha 10, offset_final valha 32, offset_tratamento valha 96 e índice_da_classe_da_exceção contenha o índice da entrada na tabela de constantes que simboliza a classe java.io.IOException. Essa entrada indica que, se uma exceção que seja instância da classe java.io.IOException for lançada por uma instrução com *offset* entre 10 e 32 (10 inclusive e 32 exclusive), então a execução deve continuar a partir da instrução com *offset* 96.

⇒ número_de_atributos contém o número de entradas na tabela_de_atributos. Cada entrada da tabela_de_atributos contém um atributo que fornece informações extras sobre o código.

- ConstantValue {
    u2 índice_do_nome;
    u4 tamanho;
    u2 índice_da_constante;
  }

  Representa o valor de uma campo constante, isto é, declarado estático e final.

⇒ Na posição índice_do_nome da tabela de constantes deve haver uma Constante_Utf8 contendo o palavra "ConstantValue".

⇒ tamanho deve ser 2.

⇒ índice_da_constante deve conter o índice, na tabela de constantes, de uma entrada do tipo Constant_Integer, Constant_Float, Constant_Long, Constant_Double ou Constant_String, compatível com o tipo do campo constante.

- Deprecated {
  u2 índice_do_nome;
  u4 tamanho;
  }

  Indica que um método, campo, classe ou interface é obsoleto.

  ⇒ Na posição índice_do_nome da tabela de constantes deve haver uma Constante_Utf8 contendo o palavra "Deprecated".

  ⇒ tamanho deve ser 0.

- Exceptions {
  u2 índice_do_nome;
  u4 tamanho;
  u2 número_de_exceções;
  u2 tabela_de_exceções[número_de_exceções];
  }

  Descreve as exceções que um método pode lançar.

  ⇒ Na posição índice_do_nome da tabela de constantes deve haver uma Constante_Utf8 contendo o palavra "Exceptions".

  ⇒ tamanho deve conter o tamanho do atributo excluindo os 6 bytes iniciais.

  ⇒ número_de_exceções contém o número de entradas da tabela_de_exceções. Cada elemento da tabela_de_exceções é um índice, na tabela de constantes, de uma Constante_Classe simbolizando a classe de uma exceção que o método pode lançar.

- InnerClasses {
  u2 índice_do_nome;
  u4 tamanho;
  u2 número_de_classes;
  Classe_aninhada tabela_de_classes[número_de_classes];
  }

Contém informações sobre classes (ou interfaces) aninhadas. Deve estar presente na tabela de atributos das classes (ou interfaces) aninhadas e na das classes (ou interfaces) que contém classes (ou interfaces) aninhadas.

⇒ Na posição índice_do_nome da tabela de constantes deve haver uma Constante_Utf8 contendo o palavra "InnerClasses".

⇒ tamanho deve conter o tamanho do atributo excluindo os 6 bytes iniciais.

⇒ número_de_classes contém o número de entradas da tabela_de_classes. Cada entrada desta tabela descreve com detalhes uma classe aninhada C. A estrutura de cada entrada é esta:

```
Classe_aninhada {
    u2 índice_da_classe_aninhada;
    u2 índice_da_classe_externa;
    u2 índice_do_nome_da_classe;
    u2 flags_de_acesso;
}
```

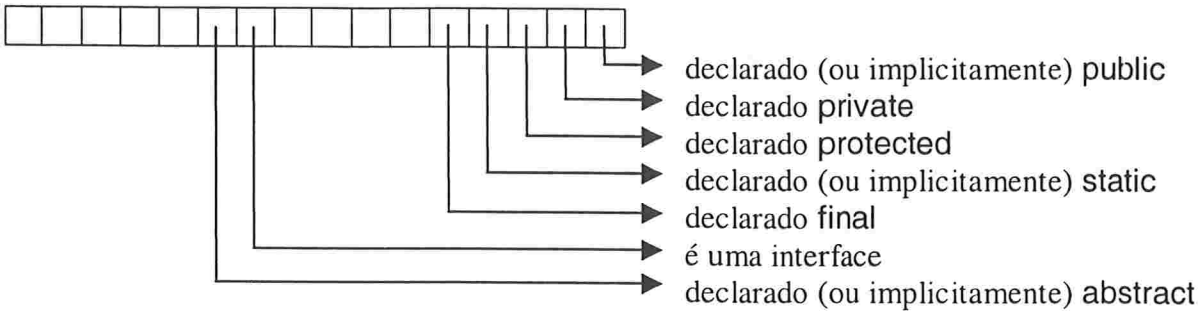
onde:

• índice_da_classe_aninhada contém o índice, na tabela de constantes, da Constante_Classe que simboliza a classe aninhada C;

• índice_da_classe_externa contém o índice, na tabela de constantes, da Constante_Classe simbolizando a classe na qual C foi declarada (ou 0, se a classe C foi declarada no corpo de um **método** da classe, e não no corpo da classe);

• índice_do_nome_da_classe contém o índice, na tabela de constantes, da Constante_Utf8 contendo o nome com o qual C foi declarada (ou 0, se C é uma classe anônima);

• flags_de_acesso é uma máscara contendo as permissões de acesso e outras propriedades da classe C. A figura a seguir representa o significado dos bits desta máscara, quando ligados:



- LineNumberTable {
    u2 índice_do_nome;
    u4 tamanho;
    u2 tamanho_da_tabela;
    {
        u2 offset_inicial;
        u2 número_de_linha;
    } tabela_de_números_de_linhas[tamanho_da_tabela];
  }

Contém a correspondência entre os *offsets* das instruções no código compilado e números de linhas no código fonte.

⇒ Na posição índice_do_nome da tabela de constantes deve haver uma Constante_Utf8 contendo o palavra "LineNumberTable".

⇒ tamanho deve conter o tamanho do atributo excluindo os 6 bytes iniciais.

⇒ tamanho_da_tabela contém o número de entradas da tabela_de_números_de_linhas. Cada entrada desta tabela indica a primeira instrução do código compilado (offset_inicial) que está associada a determinada linha do código fonte (número_de_linha).

- LocalVariableTable {
    u2 índice_do_nome;
    u4 tamanho;
    u2 tamanho_da_tabela;
    {
        u2 offset_inicial;
        u2 extensão;
        u2 índice_do_nome;
        u2 índice_do_descritor;
        u2 índice;
    } tabela_de_variáveis_locais[tamanho_da_tabela];
  }

Contém informações sobre variáveis locais do código.

⇒ Na posição índice_do_nome da tabela de constantes deve haver uma Constante_Utf8 contendo o palavra "LocalVariableTable".

⇒ tamanho deve conter o tamanho do atributo excluindo os 6 bytes iniciais.

⇒ tamanho_da_tabela contém o número de entradas da tabela_de_variáveis_locais. Cada entrada desta tabela fornece informações sobre uma das variáveis locais do código. O nome e o descritor da variável são dados, respectivamente, pelos elementos da tabela de constantes nas posições índice_do_nome e índice_do_descritor. A região do código onde a variável local está ativa é o trecho [offset_inicial, offset_inicial + extensão], e o índice da variável local na tabela de variáveis locais é dado por índice.

- Synthetic {
  u2 índice_do_nome;
  u4 tamanho;
  }

Utilizado para indicar que um campo não aparece no código fonte (por exemplo, foi gerado pelo compilador).

⇒ Na posição índice_do_nome da tabela de constantes deve haver uma Constante_Utf8 contendo o palavra "Synthetic".

⇒ tamanho deve ser 0.

- SourceFile {
  u2 índice_do_nome;
  u4 tamanho;
  u2 índice_do_nome_do_arquivo;
  }

Fornece o nome do arquivo fonte que gerou o arquivo .class.

⇒ Na posição índice_do_nome da tabela de constantes deve haver uma Constante_Utf8 contendo o palavra "SourceFile".

⇒ tamanho deve ser 2.

⇒ índice_do_nome_do_arquivo é o índice, na tabela de constantes, da Constante_Utf8 contendo o nome do arquivo fonte.

3.6 Restrições em um arquivo .class

Há inúmeras restrições impostas a um arquivo .class para que ele possa ser considerado válido. Essas restrições são verificadas pela máquina virtual Java em três momentos:

- quando a classe ou interface é carregada, isto é, quando sua representação binária é encontrada e, a partir dela, é construído um objeto java.lang.Class que a representa;
- quando a classe ou interface é ligada (*linked*), isto é, quando sua representação binária, já carregada, é combinada com a máquina virtual de forma a tornar-se algo executável;
- quando um tipo, método ou instrução é referenciado pela primeira vez.

As restrições verificadas a cada momento são distintas e não são feitas mais de uma vez. Por exemplo, a verificação de que o nome de um campo é válido ou de que as variáveis locais não são lidas antes de serem inicializadas é feita apenas uma vez, quando a classe é ligada, e a verificação de que um método X tem permissão de executar um método Y é feita apenas quando o método X invocar o método Y pela primeira vez.

Um erro (isto é, um objeto que é instância da classe java.lang.Error) é lançado sempre que a verificação falha.

Capítulo 4

Ladybug

Neste capítulo descrevemos Ladybug, a ferramenta que desenvolvemos capaz de detectar dinamicamente condições de disputa em programas Java. Ladybug é composta por um programa que rescreve classes compiladas (chamado **Hatchery**) e por uma biblioteca que implementa os algoritmos de detecção de condições de disputa descritos no Capítulo 2 (o algoritmo escolhido para monitorar a execução é escolhido pelo usuário, no instante em que ele executar a classe rescrita).

Este capítulo explica a necessidade do programa rescritor, como ele funciona e quando se mostra insuficiente para a análise dinâmica; também discute o funcionamento dos algoritmos de detecção implementados e as modificações que foram necessárias para que funcionassem de acordo com o modelo de concorrência adotado em Java.

4.1 A rescrita de uma classe

Se as invocações aos métodos de detecção de condições de disputa fossem inseridas manualmente pelo programador no código a ser verificado, a eficácia da ferramenta certamente diminuiria, por dois motivos: primeiro, a inserção manual é lenta e tediosa; segundo, é sujeita a erros. Tome, por exemplo, o seguinte método:

```
synchronized int foo() {
    return i/j;
}
```

onde i e j são variáveis de instância. Se o programador tivesse que rescrever o código fonte para que ele pudesse ser monitorado, a versão rescrita seria extremamente difícil de ler, parecendo-se com isto:

```
synchronized int foo() {
    if (DEBUG) Ladybug.monitorEnter(this);
    try {
        if (DEBUG) StaticLadybug.readField(this, "i");
        if (DEBUG) StaticLadybug.readField(this, "j");
        int temp = i/j;
        if (DEBUG) StaticLadybug.monitorExit(this);
        return temp;
    }
    catch (Throwable e) {
        if (DEBUG) StaticLadybug.monitorExit(this);
        throw e;
    }
}
```

É por isso que Ladybug possui um módulo que rescreve classes já compiladas (chamado Hatchery). Utilizando-se do pacote classfile, descrito na seção 4.2, Hatchery examina o código dos métodos das classes e, sempre que apropriado, insere instruções para invocação dos métodos de monitoramento. A definição de "apropriado", no entanto, está fortemente ligada aos métodos de monitoramento disponíveis. Por exemplo, não existe nenhum método para monitorar acessos a variáveis locais (porque isso não é necessário para os algoritmos implementados), e portanto, ao rescrever uma classe, tais acessos são ignorados.

As próximas seções descrevem os eventos que necessitam ser monitorados para que condições de disputa possam ser descobertas com os algoritmos descritos no Capítulo 2, bem como os respectivos métodos que devem ser invocados para monitorá-los. Em alguns casos, o método de monitoramento deve ser invocado **antes** da ocorrência do evento; em outros, **depois**:

- Se o método deve ser invocado antes da ocorrência do evento, sua descrição dirá que a invocação deve ser feita quando o evento está prestes a ocorrer, significando que, entre a instrução que invoca o método de monitoramento e o evento monitorado não deve haver nenhuma outra instrução.
- Se o método deve ser invocado depois da ocorrência do evento, sua descrição dirá que a invocação deve ser feita imediatamente após a ocorrência do evento ou quando o evento acabou de ocorrer, isto é, entre o evento monitorado e a instrução que invoca o método que o monitora não deve haver nenhuma outra instrução.

A decisão de colocar um método de monitoramento antes ou depois do evento monitorado é normalmente arbitrária, a menos que a descrição do evento afirme o contrário. Mas é importante frisar que os métodos de monitoramento que ocorrem **antes** do evento monitorado assumem que esse evento será totalmente executado sem que nenhuma exceção seja lançada (por exemplo, o programa não deve tentar ler um campo de um objeto null).

A descrição dos eventos cujo monitoramento é necessário irá se referir diversas vezes a certas instruções da máquina virtual Java. Todas as instruções relevantes estão listadas a seguir:

aload0	Empilha o valor da variável local de índice 0, que deve ser uma referência a um objeto; em métodos de instância, tal variável contém sempre uma referência a this.
athrow	Lança uma exceção; a referência à exceção que será lançada deve estar no topo da pilha de operandos.
dup	Duplica a palavra[12] do topo da pilha de operandos, empilhando-a.
dup_x2	Duplica a palavra do topo da pilha de operandos, inserindo-a três posições abaixo do topo.
dup2	Duplica as duas palavras do topo da pilha de operandos, empilhando-as na mesma ordem.
dup2_x1	Duplica as duas palavra do topo da pilha de operandos, inserindo-as, em ordem, três posições abaixo do topo.
dup2_x2	Duplica as duas palavras do topo da pilha de operandos, inserindo-as, em ordem, quatro posições abaixo do topo.
pop	Desempilha a palavra do topo da pilha de operandos.
pop2	Desempilha as duas palavras do topo da pilha de operandos.
iconst0, iconst1	Empilha o valor 0 (ou 1).
iload, lload	Empilha o valor de uma variável local inteira (iload) ou long (lload). O argumento da instrução é o índice da variável local.

[12] Uma palavra na máquina virtual Java tem 32 bits.

istore, lstore — Salva o valor do topo da pilha em uma variável local inteira (istore) ou long (lstore), desempilhando-o em seguida. O argumento da instrução é o índice da variável local.

invokeinterface — Invoca métodos de interfaces. O argumento dessa instrução é um índice, na tabela de constantes, do método a ser invocado; a pilha de operandos deve conter os argumentos que o método receberá e também uma referência ao objeto para o qual o método foi invocado (o último argumento deve estar no topo da pilha, e a referência abaixo do primeiro argumento). Tanto a referência quanto os argumentos são desempilhados antes do método ser executado. Por clareza, quando o texto se referir a essa instrução (ou a qualquer outra que invoque um método), não será utilizado o índice do método, mas o próprio valor da tabela de constantes (com os nomes de classes e descritores devidamente interpretados), como em:

```
invokeinterface pacote.Interface.método(int, byte[])
```

A instrução invokeinterface possui, na verdade, mais dois argumentos, que são mantidos por razões de compatibilidade. Esses argumentos podem simplesmente ser ignorados no contexto deste trabalho.

invokespecial — Invoca métodos de instância privados, construtores ou métodos de instância definidos em superclasses. O argumento dessa instrução é um índice, na tabela de constantes, do método a ser invocado. A pilha de operandos deve conter os argumentos que o método receberá e também uma referência ao objeto para o qual o método foi invocado (como com invokeinterface); tanto os argumentos quanto a referência são desempilhados antes do método ser executado.

invokestatic — Invoca métodos estáticos. O argumento dessa instrução é um índice, na tabela de constantes, do método a ser invocado; a pilha de operandos deve conter os argumentos que o método receberá. Antes do método ser executado, os argumentos são desempilhados.

invokevirtual	Invoca métodos de instância. O argumento dessa instrução é um índice, na tabela de constantes, do método a ser invocado; a pilha de operandos deve conter os argumentos que o método receberá e uma referência ao objeto para o qual o método foi invocado (como em invokespecial e invokeinterface), que serão desempilhados antes da execução do método.
ldc	Empilha uma constante da tabela de constantes; o argumento dessa instrução é o índice da constante a ser empilhada, mas, por clareza, quando o texto se referir a esta instrução será utilizado diretamente o valor da constante que será carregada.
return	Retorna de um método.
areturn, dreturn, freturn, ireturn, lreturn	Retorna de um método, utilizando o valor no topo da pilha de operandos como valor de retorno (a = referência; d = double; f = float; i = int, byte, short, boolean, char; l = long).

Ladybug não fornece nenhuma forma de inserir automaticamente código que monitore qualquer um dos eventos descritos nas seções a seguir caso eles ocorram no interior de métodos nativos (escritos numa linguagem que não seja Java) ou sejam executados utilizando reflexão (pacote java.lang.reflect). Além disso, embora nem a máquina virtual Java nem as classes descritas na seção 4.2 assumam que as instruções sendo executadas foram geradas pela compilação de código fonte escrito em Java, Ladybug assume.

4.2 O pacote classfile

Desenvolvemos um conjunto de classes — pacote br.ime.usp.classfile — que modela arquivos .class e permite a sua manipulação conveniente, embora não forneça um nível muito alto de abstração (pois sua utilização exige o conhecimento prévio da estrutura de um arquivo .class). As tarefas que essas classes facilitam incluem:

- leitura/escrita de arquivos .class;
- inserção, remoção e substituição de campos, métodos e instruções no código de um método, inclusive a atualização automática de *offsets* em todos os pontos da classe onde

isso se fizer necessário caso uma instrução seja inserida, removida ou substituída em um método;

- verificação da validade de um arquivo .class.

Uma descrição mais detalhada desta biblioteca encontra-se no Apêndice A. Por ora, é suficiente saber que este pacote é utilizado por Hatchery para rescrever uma classe, de acordo com as necessidades de monitoramento descritas nas próximas seções.

4.3 Detectando o início de uma thread

Há uma única maneira de se iniciar uma nova *thread*: invocar o método void start() declarado na classe java.lang.Thread. É importante que o início de uma nova *thread* seja registrado, pois se uma *thread* t1 cria uma *thread* t2 então não existem condições de disputa entre os acessos à memória feitos por t2 e aqueles feitos por t1 **antes** de iniciar t2 (veja a seção 1.5).

A invocação do método start() pode ser reconhecida tanto no código fonte quanto no código compilado pela ocorrência de um destes três padrões:

1. Código fonte: `t.start();`

   Código compilado:
   ```
   ⟨código para empilhar referência a t⟩
   invokevirtual void ⟨classe⟩.start()
       ou
   invokeinterface void ⟨interface⟩.start()
   ```

   Restrições: t, cuja referência está no topo da pilha, deve pertencer à classe java.lang.Thread ou então a uma classe C tal que C é subclasse de java.lang.Thread e nem C nem qualquer uma de suas superclasses — exceto java.lang.Thread — define o método void start(). Além disso, t deve ser uma instância de ⟨classe⟩ ou implementar ⟨interface⟩.

2. Código fonte: `this.start(); // ou apenas start();`

   Código compilado:
   ```
   ⟨código para empilhar referência a this⟩
   invokevirtual void ⟨classe⟩.start()
   ```

   Restrições: this nas mesmas condições do parâmetro t descrito acima.

3. Código fonte: `super.start();`

   Código compilado: ⟨código para empilhar referência a **this**⟩
   `invokespecial void ⟨classe⟩.start()`

   Restrições: this, cuja referência deve estar no topo da pilha, deve ser uma instância de uma subclasse C da classe java.lang.Thread, e nenhuma superclasse de C — exceto java.lang.Thread — define o métod void start(); além disso, ⟨classe⟩ é uma superclasse da classe onde o código compilado se encontra.

Ladybug tem apenas um método responsável por registrar o início de uma nova *thread*:

```
void StaticLadybug.startThread(Object t, boolean invokeSpecial)
```

onde t é o objeto para o qual o método start será invocado (isto é, a *thread* que será iniciada), e invokeSpecial indica se a invocação será feita utilizando a instrução da máquina virtual invokespecial (a necessidade do segundo argumento será explicada adiante). Este método de monitoramento deve ser chamado quando a *thread* t está prestes a ser iniciada, e assume ainda que a *thread* que o invocou é a *thread* iniciadora de t (é importante que a invocação ao método de monitoramento preceda o início de uma *thread*; do contrário, haveria uma nova *thread* em execução sem o conhecimento do módulo de monitoramento). Ao rescrever uma classe, é necessário, portanto, fazer com que as invocações:

```
t.start();
this.start();
super.start();
```

comportem-se, respectivamente, como se tivessem sido escritas desta forma:

```
StaticLadybug.startThread(t, false); t.start();
StaticLadybug.startThread(this, false); this.start();
StaticLadybug.startThread(this, true); super.start();
```

Como nem sempre é possível verificar, durante a fase de rescrita, se as restrições dadas acima estão satisfeitas (por exemplo, quando a invocação do método start é feita utilizando a instrução invokeinterface), a abordagem adotada foi acrescentar código de monitoramento sempre que é encontrada uma invocação a **qualquer** método de instância de nome start que

não recebe nenhum argumento e não retorna valor algum, e deixar que o próprio método de monitoramento decida se esse start deve ou não ser monitorado. Essa é a razão do método StaticLadybug.startThread ter, como primeiro argumento, uma referência a um objeto qualquer, e não a uma *thread*, e é também a razão da existência do argumento invokeSpecial (note que as restrições quanto ao primeiro argumento são diferentes no caso de uma invocação feita com invokespecial). Assim,

```
invokevirtual void ⟨classe⟩.start() ; ou invokeinterface void ⟨interface⟩.start()
```

é rescrito como:

```
dup       ; empilha outra referência ao objeto para o qual start é invocado
iconst0   ; empilha o valor false
invokestatic StaticLadybug.startThread(java.lang.Object, boolean)
invokevirtual void ⟨classe⟩.start(); ou invokeinterface void ⟨interface⟩.start()
```

e

```
invokespecial void ⟨classe⟩.start()
```

é rescrito como:

```
dup       ; empilha outra referência ao objeto para o qual start é invocado
iconst1   ; empilha o valor true
invokestatic StaticLadybug.startThread(java.lang.Object, boolean)
invokespecial void ⟨classe⟩.start()
```

Como *threads* não são iniciadas com freqüência (em relação às outras operações monitoradas), deixar que o próprio método StaticLadybug.startThread decida se uma invocação ao método start deve ou não ser monitorada não trará degradação perceptível ao desempenho do programa sendo monitorado — a menos que ele contenha muitas invocações a métodos start que não iniciem, de fato, novas *threads*. Note que StaticLadybug.startThread não precisa verificar se o objeto recebido (que é o objeto para o qual o método start será invocado) é uma instância de ⟨classe⟩ ou de uma classe que implementa ⟨interface⟩ (se não fosse, o código seria inválido e teria sido recusado pela máquina virtual).

4.4 Detectando sincronizações feitas utilizando o método join

Uma *thread* pode ser notificada que outra *thread* morreu através de algum dos métodos join definidos na classe java.lang.Thread. Embora essa não seja a única maneira de uma

thread tomar conhecimento do término de outra, é a única que pode ser automaticamente reconhecida, no código fonte ou no código compilado, pela ocorrência de um destes padrões:

1. Código fonte: t.join([long [, int]]);[13]

   Código compilado: ⟨código para empilhar referência a **t**⟩
   [⟨código para empilhar os argumentos⟩]
   invokevirtual void java.lang.Thread.join([long [, int]])
   ou
   invokeinterface void ⟨interface⟩.join([long [, int]])

   Restrições: t deve ser uma instância da classe java.lang.Thread ou de uma classe que implementa ⟨interface⟩. Como os métodos join são finais, é desnecessário colocar a restrição de que as superclasses da classe a qual t pertence não devem declarar um método join com a mesma assinatura de um dos métodos join declarados em java.lang.Thread.

2. Código fonte: this.join([long [, int]]); *// ou apenas join(...)*

   Código compilado: ⟨código para empilhar referência a **this**⟩
   [⟨código para empilhar os argumentos⟩]
   invokevirtual void ⟨classe⟩.join([long [, int]])

   Restrições: this deve ser uma instância da classe java.lang.Thread.

3. Código fonte: super.join([long, [int]]);

   Código compilado: ⟨código para empilhar referência a **this**⟩
   [⟨código para empilhar os argumentos⟩]
   invokespecial void java.lang.Thread.join([long [, int]])

   Restrições: this deve ser uma instância da classe java.lang.Thread (como java.lang.Object não declara nenhum método join e é a única superclasse de java.lang.Thread, é impossível que this pertença à classe java.lang.Thread).

[13] A notação [ ] significa "opcional".

A importância de saber que uma *thread* t1 esperou o término da *thread* t2 está no fato de que não há condições de disputa entre nenhum dos acessos feitos pela *thread* t2 e aqueles feitos pela *thread* t1 após t1 ter invocado t2.join() (veja a seção 1.5). O método de Ladybug responsável por registrar que uma *thread* esperou o término de outra é:

```
void StaticLadybug.joinThread(Object t);
```

onde t é o objeto para o qual um dos métodos join foi invocado. StaticLadybug.joinThread deve ser invocado imediatamente após a invocação ao método join e, além disso, assume que a *thread* que o invocou é também a que esperou o término da *thread* t (a invocação ao método de monitoramento não poderia vir antes da invocação do método join, pois após o término da execução de qualquer um dos métodos join disponíveis é possível que a *thread* para a qual o método foi invocado ainda esteja viva, caso o método join tenha lançado uma exceção InterruptedException ou terminado devido a um *timeout*). Assim, ao se rescrever uma classe, um dos objetivos será fazer com que as invocações:

```
t.join(...);
this.join(...);
super.join(...);
```

comportem-se, respectivamente, como se tivessem sido escritas desta forma:

```
t.join(...); StaticLadybug.joinThread(t);
this.join(...); StaticLadybug.joinThread(this);
super.join(...); StaticLadybug.joinThread(this);
```

Qualquer invocação a um método de instância join com a mesma assinatura de um dos métodos join declarados na classe java.lang.Thread será monitorada. A responsabilidade de verificar se a invocação foi feita para um objeto que seja realmente instância da classe Thread é deixada para o método de monitoramento. E como é possível que, após a invocação a t.join(...), a *thread* representada por t não tenha ainda terminado, StaticLadybug.joinThread deve também verificar se a *thread* recebida como argumento está viva ou não.

O código compilado é rescrito da maneira descrita a seguir. Uma invocação ao método join sem argumentos:

```
invokevirtual void java.lang.Thread.join() ; ou invokespecial ou invokeinterface
```

é rescrita como:

```
dup ; empilha outra referência ao objeto para o qual join é invocado
invokevirtual void java.lang.Thread.join() ; ou invokespecial, ou invokeinterface
invokestatic StaticLadybug.joinThread(java.lang.Object)
```

Uma invocação ao método join com um argumento do tipo long:

```
invokevirtual void java.lang.Thread.join(long) ; ou invokespecial ou invokeinterface
```

é rescrita como:

```
lstore ⟨n⟩          ; salva em uma variável local o valor do topo da pilha, que é o
                    ; argumento para o método join, e desempilha-o; ⟨n⟩ é o índice da
                    ; variável local
dup                 ; empilha outra referência ao objeto para o qual join é invocado
lload ⟨n⟩           ; restaura o valor salvo
invokevirtual void java.lang.Thread.join(long) ; ou invokespecial, invokeinterface
invokestatic StaticLadybug.joinThread(java.lang.Object)
```

E uma invocação ao método join com dois argumentos, um do tipo long e outro do tipo int:

```
invokevirtual void java.lang.Thread.join(long, int) ; invokespecial, invokeinterface
```

é rescrita como:

```
istore ⟨n⟩          ; salva em uma variável local o valor do topo da pilha, que é o
                    ; argumento int para o método join, e desempilha-o; ⟨n⟩ é o índice da
                    ; variável local
lstore ⟨n+1⟩        ; salva em uma variável local o valor do topo da pilha, que agora é o
                    ; argumento long, e desempilha-o
dup                 ; empilha outra referência ao objeto para o qual join é invocado
lload ⟨n+1⟩         ; restaura o valor long salvo
iload ⟨n⟩           ; restaura o valor int salvo
invokevirtual void java.lang.Thread.join(long, int) ; invokespecial, invokeinterface
invokestatic StaticLadybug.joinThread(java.lang.Object)
```

4.5 Descobrindo quando um lock *é adquirido ou liberado*

Há duas maneiras de uma *thread* adquirir um *lock*: a primeira é executar uma sentença synchronized; a outra é executar um método declarado como synchronized. O *lock* será liberado quando o corpo da sentença synchronized ou o método terminar, seja de forma normal ou abrupta (seção 1.3).

A máquina virtual possui apenas duas instruções para manipulação de *locks*: monitorenter, que faz com que a *thread* que executou a instrução adquira o *lock* associado a

um objeto, e monitorexit, que faz com que a *thread* que executou a instrução libere o *lock* associado a um objeto. Essas instruções assumem que no topo da pilha de execução há uma referência ao objeto cujo *lock* será adquirido ou liberado, referência esta que será desempilhada após a execução da instrução. Assim, quando compilando uma sentença como esta:

```
synchronized (objeto) {
    ...corpo da sentença synchronized
}
```

o compilador deve gerar um código como este:

```
    ⟨instruções para empilhar uma referência a objeto⟩
    monitorenter
    ⟨instruções referentes ao corpo da sentença synchronized⟩
    ⟨instruções para empilhar uma referência a objeto⟩
    monitorexit
    goto ♦
•:  ⟨instruções para empilhar uma referência a objeto⟩
    monitorexit
    athrow
♦:  ⟨instruções seguintes à sentença synchronized⟩
```

e, além disso, deve inserir na tabela de exceções do código uma entrada informando que uma exceção pode ser lançada quando executando qualquer uma das "instruções referentes ao corpo da sentença synchronized", e que, se isso ocorrer, a execução deve continuar a partir de • (onde o *lock* é liberado e a exceção é lançada novamente).

A compilação de um método synchronized, por outro lado, é mais simples, pois nenhuma instrução monitorenter ou monitorexit é gerada — o compilador deve apenas ligar um bit específico do método, e a máquina virtual garantirá que, ao executar a primeira instrução deste método, a *thread* terá a posse do *lock* associado ao objeto para o qual o método foi invocado (no caso de métodos de instância) ou do *lock* associado à classe em que o método foi declarado (no caso de métodos estáticos). A máquina virtual também garante que o *lock* será liberado quando a execução do método terminar.

Ladybug possui seis métodos, todos estáticos e declarados na classe StaticLadybug, para ser informado de que um *lock* foi adquirido ou liberado:

- void monitorEnter(Object o): informa que a *thread* que invocou este método acabou de executar uma instrução monitorenter, adquirindo o *lock* associado ao objeto o.

- void monitorExit(Object o): informa que a *thread* que invocou este método está prestes a executar uma instrução monitorexit, liberando o *lock* associado ao objeto o.
- void enterMethod(Object instância, String nomeDoMétodo): informa que a *thread* que executou este método acabou de entrar no método sincronizado de nome nomeDoMétodo relativo ao objeto instância.
- void leaveMethod(Object instância, String nomeDoMétodo): informa que a *thread* que executou este método está prestes a deixar o método sincronizado de nome nomeDoMétodo relativo ao objeto instância.
- void enterStaticMethod(String nomeDaClasse, String nomeDoMétodo): informa que a *thread* que executou este método acabou de entrar num método estático e sincronizado, de nome nomeDoMétodo e declarado na classe cujo nome, completamente qualificado e utilizando barras como separadores, é nomeDaClasse.
- void leaveStaticMethod(String nomeDaClasse, String nomeDoMétodo): informa que a *thread* que executou este método está prestes a deixar um método estático e sincronizado de nome nomeDoMétodo, declarado na classe cujo nome, completamente qualificado e utilizando barras como separadores, é nomeDaClasse.

Assim, há mais três objetivos ao se rescrever uma classe:

1. Uma sentença *synchronized* como esta:

```
synchronized (foo) {
    ...
    return; // ou qualquer outra instrução que provoque a saída deste bloco
    ...
}
```

deve funcionar como se, na verdade, tivesse sido escrita desta forma:

```
synchronized (foo) {
    try {
        StaticLaydbug.monitorEnter(foo);
        ...
        StaticLadybug.monitorExit(foo);
        return;
        ...
        StaticLadybug.monitorExit(foo);
    }
    catch (Throwable e) {
        StaticLadybug.monitorExit(foo);
        throw e;
    }
}
```

2. Um método *synchronized* como este:

```
synchronized int foo() {
    ...
    return 1;
    ...
    return cálculo();
}
```

deve funcionar como se, na verdade, tivesse sido escrito desta forma:

```
synchronized int foo() {
    StaticLadybug.enterMethod(this, "foo");
    try {
        ...
        StaticLadybug.leaveMethod(this, "foo");
        return 1;
        ...
        int temp = cálculo();
        StaticLadybug.leaveMethod(this, "foo");
        return temp;
    }
    catch (Throwable e) {
        StaticLadybug.leaveMethod(this, "foo");
        throw e;
    }
}
```

3. E um método *synchronized* como este (declarado na classe C do pacote p)

```
synchronized static String foo() {
    ...
    return "Marco Antônio";
    ...
    return "Cleópatra";
}
```

precisa funcionar como se, na verdade, tivesse sido escrito desta forma:

```
synchronized static String foo() {
    StaticLadybug.enterStaticMethod("p/C", "foo");
    try {
        ...
        StaticLadybug.leaveStaticMethod("p/C", "foo");
        return "Marco Antônio";
        ...
        StaticLadybug.leaveStaticMethod("p/C", "foo");
        return "Cleópatra";
    }
    catch (Throwable e) {
        Ladybug.leaveStaticMethod("p/C", "foo");
        throw e;
    }
}
```

Para atingir estes objetivos, o código compilado é rescrito da maneira descrita a seguir:

1. As instruções monitorenter e monitorexit encontradas no código compilado são substituídas, respectivamente pelas seguintes seqüências de instruções:

```
dup      ; duplica a referência ao objeto cujo lock será adquirido
monitorenter
invokestatic void StaticLadybug.monitorEnter(java.lang.Object)
```

e

```
dup      ; duplica a referência ao objeto cujo lock será liberado
invokestatic void StaticLadybug.monitorExit(java.lang.Object)
monitorexit
```

2. No início dos métodos de instância que sejam sincronizados, são inseridas as instruções:

```
aload0                    ; empilha referência ao objeto this
ldc ⟨nome do método⟩      ; empilha referência à String contendo o nome do método
invokestatic void StaticLadybug.enterMethod(java.lang.Object,
                                            java.lang.String)
```

e, antes de cada instrução return (ou ireturn, freturn, dreturn, lreturn, areturn), é inserida a seqüência:

```
aload0
ldc ⟨nome do método⟩
invokestatic void StaticLadybug.leaveMethod(java.lang.Object,
                                            java.lang.String)
```

É necessário ainda acrescentar, no final do código do método, instruções para o tratamento de exceções:

```
♦: aload0
   ldc ⟨nome do método⟩
   invokestatic void StaticLadybug.leaveMethod(java.lang.Object,
                                               java.lang.String)
   athrow          ; lança novamente a exceção
```

e inserir, na tabela de exceções do código do método, uma entrada informando que, caso seja lançada qualquer exceção durante a execução do método e caso essa exceção não seja capturada, a execução deve continuar a partir de ♦.

3. Nos métodos estáticos que também sejam sincronizados, são inseridas seqüências de instruções semelhantes àquelas inseridas para métodos de instância, substituindo as instruções aload0, que empilham a referência ao objeto this, por ldc ⟨nomeDaClasse⟩, que empilharão uma referência ao nome da classe. Neste caso, porém, a escolha de inserir o método de monitoramento **após** o método já ter sido iniciado não foi arbitrária; existe um motivo importante para isso, explicado ao final da seção 4.7.

4.6 Descobrindo quando uma variável compartilhada é acessada

As únicas variáveis que **podem** ser compartilhadas por diversas *threads* são as variáveis de classe, variáveis de instância e elementos de vetores (variáveis locais, parâmetros e elementos da pilha de operandos de uma *thread*, por outro lado, não podem ser acessados por outras *threads*, não precisando, portanto, ter seus acessos monitorados). Descobrir quando uma dessas variáveis "potencialmente compartilhadas" é acessada é, novamente, uma questão de reconhecimento de padrões no código compilado, e todas as possíveis formas de acesso estão exemplificadas a seguir:

Código compilado (exemplo)	Descrição
getstatic Foo.campo	Empilha o valor da variável estática campo definida na classe Foo ou em uma das suas superclasses. Ocorre sempre que a variável é lida no código fonte, como em: ... = Foo.campo
putstatic Foo.campo	Atribui o valor no topo da pilha de operandos à variável estática campo definida na classe Foo ou em uma de suas superclasses. Ocorre sempre que a variável é escrita no código fonte, como em: Foo.campo = ...
getfield Foo.campo	Empilha o valor da variável de instância campo (declarada na classe Foo ou em uma das suas superclasses) relativa ao objeto cuja referência está no topo da pilha. Ocorre sempre que a variável é lida no código fonte, como em: ... = instânciaDeFoo.campo
putfield Foo.campo	Atribui o valor no topo da pilha à variável de instância campo (declarada na classe Foo ou em uma das suas superclasses) relativa ao objeto cuja referência está na segunda ou terceira posição da pilha. Ocorre sempre que a variável é escrita no código fonte, como em: instânciaDeFoo.campo = ...
iaload	Empilha o valor de um dos elementos de um vetor de inteiros. O índice do elemento deve estar no topo da pilha, e uma referência ao vetor deve vir logo em seguida. Ocorre sempre que um elemento de um vetor de inteiros é lido no código fonte, como em: ... = m[i]

As instruções aaload, baload, caload, daload, faload, laload e saload são utilizadas, respectivamente para empilhar o valor de um elemento de um vetor de referências, *bytes/booleans*, caracteres, *doubles*, *floats*, *longs* e *shorts*.

iastore

Atribui o valor no topo da pilha a um dos elementos de um vetor de inteiros. O índice do elemento deve vir em seguida ao valor, e uma referência ao vetor deve seguir o índice. Ocorre sempre que, no código fonte, um elemento de um vetor é escrito, como em:

m[i] = ...

As instruções aastore, bastore, castore, dastore, fastore, lastore e sastore são utilizadas, respectivamente para atribuir um valor a um elemento de um vetor de referências, *bytes/booleans*, caracteres, *doubles*, *floats*, *longs* e *shorts*.

Os métodos de Ladybug responsáveis por registrar que uma variável foi lida ou escrita, todos eles estáticos e declarados na classe StaticLadybug, são:

- void readStaticField(String nomeDaClasse, String nomeDoCampo, int n): informa que a *thread* que invocou este método acabou de ler o conteúdo da variável estática de nome ⟨nomeDoCampo⟩ definida na classe cujo nome, completamente qualificado e utilizando barras como separadores, é ⟨nomeDaClasse⟩. ⟨n⟩ deve conter o número da linha no código fonte onde o acesso à variável ocorre, ou -1 caso esta informação seja desconhecida.

  Se o campo estiver definido em uma superclasse de ⟨nomeDaClasse⟩, o monitoramento poderá falhar (este caso é discutido com detalhes na seção 4.8). A necessidade deste método (e também do método seguinte) tratar de um acesso que **já** ocorreu, e não de um que está **prestes** a ocorrer, está justificada no final da seção 4.7.

- void writeStaticField(String nomeDaClasse, String nomeDoCampo, int n): informa que a *thread* que invocou este método acabou de escrever na variável estática de nome

⟨nomeDoCampo⟩ definida na classe cujo nome, completamente qualificado e utilizando barras como separadores, é ⟨nomeDaClasse⟩. ⟨n⟩ deve conter o número da linha no código fonte onde o acesso à variável ocorre, ou -1 se esta informação for desconhecida.

Como ocorre com o método anterior, se o campo estiver definido em uma superclasse de ⟨nomeDaClasse⟩, o monitoramento poderá falhar (veja a seção 4.8).

- void readField(Object instância, String nomeDoCampo, int n): informa que a *thread* que invocou este método está prestes a ler o conteúdo da variável de instância ⟨nomeDoCampo⟩ do objeto ⟨instância⟩. ⟨n⟩ deve conter o número da linha no código fonte onde o acesso à variável ocorre, ou -1 se esta informação for desconhecida.
- void writeField(Object instância, String nomeDoCampo, int n): informa que a *thread* que invocou este método está prestes a escrever na variável de instância ⟨nomeDoCampo⟩ do objeto ⟨instância⟩. ⟨n⟩ deve conter o número da linha no código fonte onde o acesso à variável ocorre, ou -1 se esta informação for desconhecida.
- void readArray(Object vetor, int índice, int n): informa que a *thread* que invocou este método está prestes a ler o conteúdo do elemento ⟨índice⟩ do vetor ⟨vetor⟩. ⟨n⟩ deve conter o número da linha no código fonte onde o acesso ao elemento ocorre, ou -1 se esta informação for desconhecida.
- void writeArray(Object vetor, int índice, int n): informa que a *thread* que invocou este método está prestes a escrever no elemento ⟨índice⟩ do vetor ⟨vetor⟩. ⟨n⟩ deve conter o número da linha no código fonte onde o acesso ao elemento ocorre, ou -1 se esta informação for desconhecida.

Nem todas as variáveis de instância, estáticas ou elementos de vetores são, contudo, compartilhados. No exemplo a seguir:

```
class Foo {
    int campo = 1;

    static int m() {
        Foo f1 = new Foo();
        f1.campo += 41;
        return f1.campo;
    }
}
```

é impossível que qualquer um dos acessos à variável de instância campo feitos dentro do método m estejam envolvidos em condições de disputa (já que as variáveis locais de uma *thread* não podem ser acessadas por outra *thread*). Reconhecer quando ocorre um acesso a uma variável que seja realmente compartilhada apenas examinando o código (fonte ou compilado) está além do escopo deste trabalho[14], e é por isso que, quando uma classe for rescrita, **todas** as atribuições da forma:

```
Classe.x = ⟨expr⟩; // linha ⟨n1⟩
instância.x = ⟨expr⟩; // linha ⟨n2⟩
vetor[i] = ⟨expr⟩; // linha ⟨n3⟩
```

serão rescritas para que se comportem como se tivessem sido escritas, respectivamente, desta forma:

```
Classe.x = ⟨expr⟩; StaticLadybug.writeStaticField("Classe", "x", ⟨n1⟩);
tmp = ⟨expr⟩; StaticLadybug.writeField(instância, "x", ⟨n2⟩); instância.x = tmp;
tmp = ⟨expr⟩; StaticLadybug.writeArray(vetor, i, ⟨n3⟩); vetor[i] = tmp;
```

sem levar em conta se Classe, instância ou vetor são ou não realmente compartilhadas entre diversas *threads*. Da mesma forma, toda leitura a uma variável de instância ou estática, ou a um elemento de um vetor será rescrita para que se comporte como se fosse imediatamente precedida ou sucedida pela invocação ao método de monitoramento conveniente. Por exemplo:

```
System.out.println(m[i] + obj.v); // número desta linha: ⟨n⟩
```

deve se comportar como se tivesse sido escrito assim:

[14] Este reconhecimento pode ser feito através de **análise de escape** [29][32].

```
StaticLadybug.readArray(m, i, ⟨n⟩);
tmp1 = m[i];
StaticLadybug.readField(obj, "v", ⟨n⟩);
tmp2 = obj.v;
System.out.println(tmp1 + tmp2);
StaticLadybug.readStaticField("java.lang.System", "out", ⟨n⟩);
```

Esse efeito é conseguido através de inserções, no código compilado, das seqüências descritas a seguir:

- Imediatamente após cada instrução da forma getstatic ⟨nomeDaClasse⟩ ⟨nomeDoCampo⟩ é inserida a seqüência:

```
ldc ⟨nomeDaClasse⟩
ldc ⟨nomeDoCampo⟩
ldc ⟨número da linha no código fonte⟩[15]
invokestatic StaticLadybug.readStaticField( java.lang.String,
                                            java.lang.String, int)
```

- Imediatamente após cada instrução da forma putstatic ⟨nomeDaClasse⟩ ⟨nomeDoCampo⟩ é inserida a seqüência:

```
ldc ⟨nomeDaClasse⟩
ldc ⟨nomeDoCampo⟩
ldc ⟨número da linha no código fonte⟩
invokestatic StaticLadybug.writeStaticField( java.lang.String,
                                             java.lang.String, int)
```

- Imediatamente antes de cada instrução da forma getfield ⟨nomeDaClasse⟩ ⟨nomeDoCampo⟩ é inserida a seqüência:

```
dup ; duplica a referência na pilha ao objeto cujo campo será lido
ldc ⟨nomeDoCampo⟩
ldc ⟨número da linha no código fonte⟩
invokestatic StaticLadybug.readField(java.lang.Object, java.lang.String, int)
```

- Imediatamente antes de cada instrução da forma putfield ⟨nomeDaClasse⟩ ⟨nomeDoCampo⟩

[15] A informação sobre o número da linha no código fonte que gerou a instrução sendo monitorada está disponível no próprio arquivo .class. Caso o compilador não tenha gerado esta informação, será utilizado o valor -1.

⇒ se o valor que será escrito ocupar apenas uma palavra na pilha de operandos (isto é, seu tipo não é nem long nem double), é inserida a seqüência:

```
dup2     ; a combinação ⟨dup2—pop⟩ terá como efeito inserir no topo da pilha uma cópia
pop      ; da referência ao objeto cujo campo será escrito
ldc ⟨nomeDoCampo⟩
ldc ⟨número da linha no código fonte⟩
invokestatic StaticLadybug.writeField(java.lang.Object, java.lang.String, int)
```

⇒ caso contrário, é inserida a seqüência:

```
dup2_x1          ; a combinação ⟨dup2_x1—pop2—dup_x2⟩ terá como efeito inserir
pop2             ; no topo da pilha uma cópia da referência ao objeto cujo campo
dup_x2           ; será escrito
ldc ⟨nomeDoCampo⟩
ldc ⟨número da linha no código fonte⟩
invokestatic StaticLadybug.writeField(java.lang.Object, java.lang.String, int)
```

- Imediatamente antes de cada instrução aaload, baload, caload, daload, faload, iaload laload ou saload é inserida a seqüência:

```
dup2             ; faz uma cópia da referência ao vetor e do índice do elemento
ldc ⟨número da linha no código fonte⟩
invokestatic StaticLadybug.readArray(java.lang.Object, int, int)
```

- Imediatamente antes de cada instrução aastore, bastore, castore, fastore, iastore ou sastore é inserida a seqüência:

```
dup_x2           ; a combinação ⟨dup_x2—pop—dup2_x1⟩ terá como efeito inserir
pop              ; no topo da pilha uma referência ao vetor e também o índice do
dup2_x1          ; elemento acessado, nesta ordem
ldc ⟨número da linha no código fonte⟩
invokestatic StaticLadybug.writeArray(java.lang.Object, int, int)
```

- Imediatamente antes de cada instrução lastore ou dastore é inserida a seqüência:

```
dup2_x2          ; a combinação ⟨dup_x2—pop2—dup2_x2⟩ terá como efeito inserir
pop2             ; no topo da pilha uma referência ao vetor e também o índice do
dup2_x2          ; elemento acessado, nesta ordem
ldc ⟨número da linha no código fonte⟩
invokestatic StaticLadybug.writeArray(java.lang.Object, int, int)
```

4.7 O método <clinit>

Uma classe ou interface pode conter um (e no máximo um) método de inicialização. Esse método, chamado <clinit>, é gerado automaticamente pelo compilador, sendo proibido ao

código compilado conter qualquer instrução que o invoque. É um método estático, não aceita argumentos e é executado **apenas** quando a classe que o contém é inicializada (o que pode ocorrer no máximo uma vez).

<clinit> contém o código de inicialização das variáveis de classe, bem como todo o código dos inicializadores estáticos. Por exemplo, para a classe seguinte:

```
class C {
    static String s1 = "Isolda";
    static String s2 = "Tristão";
    static { System.out.println(s1); }
    static String s3;
    static { s3 = s1 + ", " + s2; }
}
```

o método <clinit> conterá, nesta ordem, instruções para:

1. inicializar a variável s1 com a cadeia "Isolda";
2. inicializar a variável s2 com a cadeia "Tristão";
3. imprimir o valor de s1;
4. inicializar a variável s3 com s1 + ", " + s2.

Uma classe é inicializada (e seu método <clinit>, caso exista, é executado) imediatamente antes da primeira ocorrência dos seguintes eventos:

- invocação de um método estático declarado na classe;
- criação de uma instância da classe;
- atribuição a uma variável estática declarada na classe;
- leitura de uma variável estática declarada na classe, desde que essa variável não seja uma constante de tempo de compilação[16];
- invocação de certos métodos da classe java.lang.Class ou do pacote java.lang.reflect.

Além disso, uma classe não pode ser inicializada enquanto a sua superclasse também não tiver sido.

---

[16] Uma constante de tempo de compilação é uma variável estática e final da classe String ou de um dos tipos primitivos (boolean, int, etc.), inicializada com um valor que pode ser decidido em tempo de compilação.

A *thread* que primeiro executar um desses eventos com uma classe não inicializada deverá, portanto, inicializar a classe antes de prosseguir com a execução normal. Para garantir que duas *threads* não inicializem a mesma classe simultaneamente ou que a inicialização seja executada mais de uma vez, o processo de inicialização deve ser todo executado sob a proteção do *lock* associado à classe:

1. ao tentar inicializar uma classe, a *thread* deve adquirir o *lock* associado à classe;
2. após adquiri-lo, deve inicializar a classe **caso isso ainda não tenha sido feito**;
3. por fim, deve liberar o *lock* associado à classe.[17]

Uma conseqüência do processo de inicialização é o fato de que todo acesso a uma variável estática feito dentro do método <clinit> da classe onde a variável foi declarada irá com certeza preceder os acesso feitos dentro dos métodos "normais" e, portanto, não pode fazer parte de uma condição de disputa. É por isso que, quando o método <clinit> de uma classe é rescrito, **não** é inserido código para monitorar as variáveis estáticas declaradas na classe.

Como o método <clinit> é implicitamente sincronizado, é inserida uma invocação ao método

```
void StaticLadybug.enterClinit(String nomeDaClasse);
```

no início do método <clinit>, e uma invocação a

```
void StaticLadybug.leaveClinit(String nomeDaClasse);
```

nos possíveis pontos de saída do método <clinit>, de forma semelhante a que é feita com o monitoramento de métodos estáticos sincronizados. O método enterClinit serve para informar que a thread que o invocou acabou de entrar no método <clinit> da classe cujo nome (completamente qualificado, usando barras como separadores) é dado como parâmetro; o método leaveClinit serve para informar que a *thread* que o invocou está prestes a deixar o método <clinit>. Além de registrar o fato de que a *thread* que os invocou está adquirindo ou liberando o *lock* associado à classe sendo inicializada, esses métodos podem ter outras atribuições, que serão vistas na seção 4.12.3.

[17] O processo, na verdade, é bem mais complexo, pois deve também tratar da ocorrência de erros durante a inicialização. Esse tratamento, porém, é irrelevante aqui.

Uma conseqüência da maneira como a inicialização de classes é feita em Java é o fato de que a invocação aos métodos que monitoram acessos a variáveis estáticas devem vir imediatamente **após** a instrução de acesso (putstatic ou getstatic), o mesmo valendo para as invocações que monitoram a entrada em métodos estáticos e sincronizados. O problema nesse caso é o fato de que a *thread* executando esses eventos pode "interromper" sua execução normal para executar a inicialização de uma ou mais classes. Tome, por exemplo, as classes abaixo:

```
class C1 {
    public static void main(String[] args) {
•       C2.val = 3.14;
        ...
    }
}
```

```
class C2 {
    static double val;
    static {
        new Thread() {
            public void run() {
◊             val = 2.718;
◊◊            for(int i = 0; i < 10; i++) val++;
            }
        }.start();
    }
}
```

Quando a linha • estiver sendo executada (mas antes da atribuição ser efetivamente realizada), a *thread* que a estiver executando será obrigada a interromper a execução para inicializar a classe C2, e a *thread* criada nesta inicialização executará atribuições (em ◊ e ◊◊) potencialmente concorrentes com aquela feita em •. Estas condições de disputa pela variável C2.val poderiam não ser percebidas caso o método de monitoramento viesse antes do evento que acessa a variável estática: para Ladybug, • teria certamente sido executada antes de ◊ e ◊◊, não sendo considerada como causadora de uma condição de disputa.

4.8 Compatibilidade binária

Suponha que as três classes seguintes sejam compiladas:

```
class A {
    static int v;
}

class B extends A {
    static int v;
}
```

```
class C {
    static void foo() { B.v = 1; }
}
```

As classes A e B não conterão código algum. Já o método foo da classe C terá um código como este:

```
iconst_1
putstatic B.v
```

Se a classe C for rescrita para que possa ser monitorada por Ladybug, seu método foo ficará assim:

```
iconst_1
putstatic B.v
ldc "B"
ldc "v"
invokestatic Ladybug.writeStaticField(java.lang.String, java.lang.String)
```

Suponha agora que a declaração do campo v da classe B seja removida:

```
class B extends A { }
```

e que a classe B seja recompilada, mas a classe C não. Quando o método C.foo for executado, a máquina virtual utilizará o campo v definido na classe A, já que aquele definido na classe B foi excluído[18]. Por outro lado, para Ladybug o campo sendo acessado ainda será B.v, e o monitoramento poderá falhar.

Quando a questão da compatibilidade binária é uma preocupação, isto é, quando as classes sendo rescritas não foram todas compiladas juntas, o módulo de Ladybug responsável por rescrevê-las deve ser informado que os métodos StaticLadybug.bcReadStaticField (String nomeDaClasse, String nomeDoCampo) e StaticLadybug.bcWriteStaticField (String nomeDaClasse, String nomeDoCampo) precisam ser utilizados no lugar de StaticLadybug.readStaticField e StaticLadybug.writeStaticField[19]. Os métodos bcReadStaticField e bcWriteStaticField sempre procuram a classe em que o campo estático foi declarado, da mesma maneira que a máquina virtual faz: se o campo ⟨nomeDoCampo⟩

[18] A especificação da linguagem Java [21][23] manda que uma exceção NoSuchFieldError seja lançada, mas isso contradiz a especificação da máquina virtual [38] e as implementações da máquina virtual da SUN (http://java.sun.com/j2se).

[19] Isso é feito através de uma opção na linha de comando ou selecionando uma opção na interface do programa Hatchery.

não é encontrado em ⟨nomeDaClasse⟩, a busca pelo campo prossegue recursivamente nas superclasses de ⟨nomeDaClasse⟩. Isso garantirá que o monitoramento será sempre condizente com o comportamento do programa, mas pode tornar a execução do código monitorado muito mais lenta, mesmo que o campo ⟨nomeDoCampo⟩ seja sempre encontrado entre os campos da classe ⟨nomeDaClasse⟩.

4.9 Quando a rescrita automática não é suficiente

Garantir que todos os acessos a uma variável compartilhada ocorrem sempre protegidos por um mesmo *lock* não é a única maneira de construir um programa sem condições de disputa. Outros mecanismos de sincronização, como barreiras, semáforos ou *bounded buffers* podem ser construídos em Java [14], e a sua utilização potencialmente fará com que Ladybug emita uma série de avisos sobre condições de disputa de fato inexistentes. Além disso, um programa pode realmente conter condições de disputa cuja presença não afeta sua correção. Para tratar desses casos, Ladybug permite que o programador guie o processo de análise de duas formas:

1. fornecendo, na linha de comando do programa que rescreverá o código compilado, nomes de variáveis cujos acessos não devem ser monitorados;
2. inserindo invocações a métodos de monitoramento diretamente no código fonte (anotações).

Ao fornecer nomes de variáveis, o programador fará que, durante a rescrita do código compilado, não sejam inseridas invocações a métodos de monitoramento para as instruções getField, putField, getStatic e putStatic que se refiram a uma variável cujo nome esteja entre aqueles fornecidos. No entanto, essa forma de guiar o monitoramento pode não ter utilidade quando:

- deseja-se ignorar os acessos às variáveis de instância de alguns, e não todos objetos pertencentes a uma classe;
- deseja-se ignorar os acessos a elementos de vetores;
- o código compilado refere-se a um campo utilizando o nome de uma subclasse da classe onde o campo está realmente declarado (o que só deve ocorrer quando a classe é recompilada após ter um de seus campos excluídos, sem que as outras classes que se referem ao campo sejam recompiladas também).

Há oito métodos de monitoramento, todos estáticos e declarados na classe StaticLadybug, cujas invocações podem ser inseridas no código fonte:

- void ignoreOn()

  Desabilita o monitoramento da *thread* que o invocou.

- void ignoreOff()

  Habilita o monitoramento da *thread* que o invocou. Chamadas aos métodos ignoreOn e ignoreOff devem estar balanceadas, isto é, após **n** invocações a ignoreOn, são necessárias **n** invocações a ignoreOff para que o monitoramento seja novamente habilitado para uma determinada *thread*.

- Object getCoordinationPoint()

  Retorna um objeto representando o "ponto" em que uma *thread* está.

- void coordinate(Object o)

  Acrescenta informações sobre a ordem dos acessos à memória feitos por duas *threads*: a que invocou este método e alguma outra que tenha previamente invocado o método getCoordinationPoint (veja a figura a seguir).

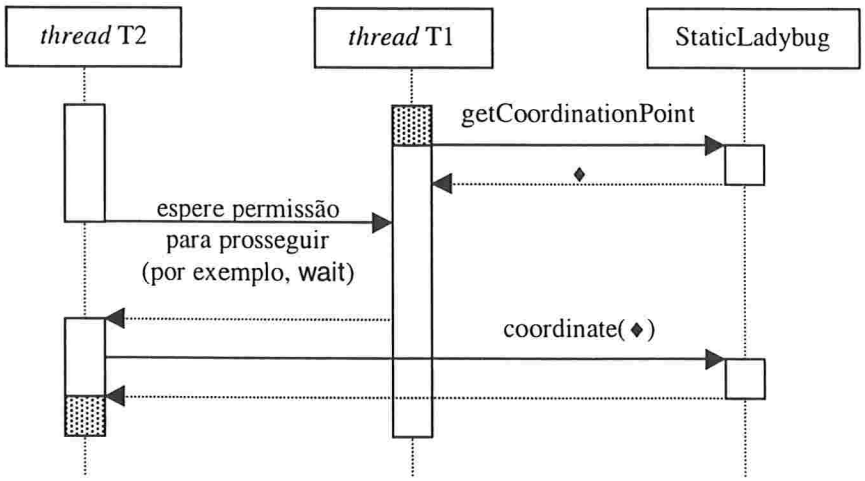


figura 7 – Como informar a Ladybug sobre a ordem dos acessos

No exemplo da figura 7, a *thread* T1 invoca o método getCoordinationPoint e o valor retornado é utilizado pela *thread* T2 (após coordenar-se com T1) na invocação do método coordinate. Ladybug pode assumir, então, que todos os acessos à memória feitos por T1 **antes** de invocar getCoordinationPoint aconteceram antes dos acessos que T2 fizer a

memória **após** invocar coordinate (áreas hachuradas). Este método e o anterior podem ser utilizados, por exemplo, para modelar barreiras ou *rendezvous* em Ladybug.

Os quatro métodos restantes são utilizados para dar suporte a implementações privadas de *locks*, indicando que uma certa região do código pode ser vista por Ladybug como estando de fato protegida. Para utilizar estes métodos, é necessário um objeto que "represente" o *lock* perante Ladybug.

- acquireWriteLock(Object wlock)
  Indica que a *thread* que invocou este método adquiriu, em modo escrita, o *lock* representado pelo objeto wlock.
- releaseWriteLock(Object wlock)
  Indica que a *thread* que invocou este método liberou o *lock* representado pelo objeto wlock, previamente adquirido em modo escrita.
- acquireReadLock(Object rlock)
  Indica que a *thread* que invocou este método adquiriu, em modo leitura, o *lock* representado pelo objeto rlock.
- releaseReadLock(Object rlock)
  Indica que a *thread* que invocou este método liberou o *lock* representado pelo objeto rlock, previamente adquirido em modo leitura.

Os métodos que adquirem e liberam *locks* assumem que o mesmo *lock* pode ser adquirido mais de uma vez por uma mesma *thread* e, portanto, se um *lock* é adquirido **n** vezes, é necessário invocar **n** vezes o respectivo método de liberação para conseguir que o *lock* seja realmente liberado.

Suponha, por exemplo, que exista uma classe que implemente semáforos em Java. Com os métodos acima, seria possível modelar a semântica dessa classe em Ladybug, como no exemplo a seguir (adaptado de [14]):

```
public class Semaforo {
    private int n;

    public Semaforo(int n) { this.n = n; }

    public synchronized void down() throws InterruptedException {
        while (n <= 0) wait();
        --n;
        if (n == 0) br.ime.usp.ladybug.StaticLadybug.acquireWriteLock(this);
    }

    public synchronized void up() {
        if (n == 0) br.ime.usp.ladybug.StaticLadybug.releaseWriteLock(this);
        ++n;
        notify();
    }
}
```

4.10 A execução de um programa rescrito

Nas seções anteriores, foi visto que os métodos de monitoramento invocados durante a execução de um programa pertencem todos à classe StaticLadybug. De fato, esta classe é composta apenas de métodos estáticos, alguns feitos para ter suas invocações inseridas automaticamente pelo programa que rescreve classes e outros cujas invocações precisam ser inseridas manualmente pelo usuário no código fonte. Esses métodos, porém, não realizam nenhuma atividade de monitoramento. StaticLadybug mantém uma referência (estática) a um objeto Ladybug ao qual esse trabalho é delegado (optamos por inserir invocações a métodos estáticos pois isso é mais simples do que invocar métodos de instância). O relacionamento entre StaticLadybug e as outras classes que são efetivamente capazes de realizar monitoração é mostrado no diagrama UML [17] na figura 8.

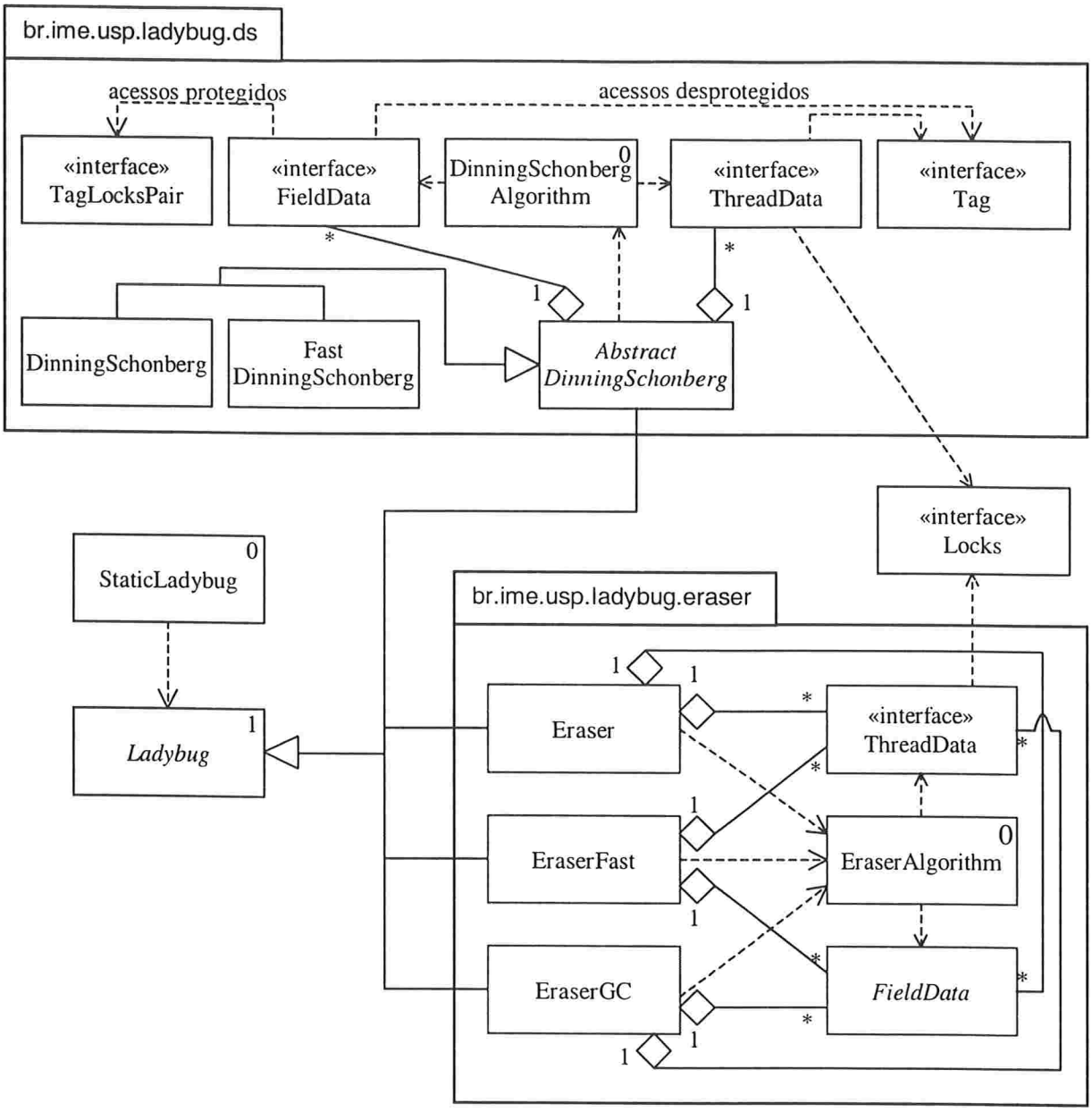


figura 8 – Os pacotes br.ime.usp.ladybug.*

As cinco classes concretas de Ladybug diferem quanto ao algoritmo que implementam e à forma de implementação:

- Eraser, EraserGC e EraserFast contém implementações do algoritmo LockSet, o mesmo utilizado pela ferramenta Eraser descrita na seção 2.3. A descrição das implementações encontra-se na seção 4.11., e os resultados obtidos no Capítulo 5.

- DinningSchonberg e DinningSchonbergFast contém implementações do algoritmo proposto por Anne Dinning e Edith Schonberg (seção 2.2). A descrição da implementações encontra-se na seção 4.12, e os resultados obtidos no Capítulo 5.

A decisão de qual dessas cinco subclasses deve ser instanciada é tomada quando a classe StaticLadybug é inicializada, com base na propriedade de sistema ladybug.species: para os valores eraser, eraser_gc, eraser_fast, ds e ds_fast, a classe escolhida será, respectivamente, Eraser, EraserGC, EraserFast, DinningSchonberg e DinningSchonbergFast. Por exemplo, com

```
java –Dladybug.species=ds Principal
```

a monitoração será feita utilizando a classe DinningSchonberg.

Há ainda mais três propriedades do sistema que afetam o comportamento dos métodos de monitoramento:

1. Para que mensagens detalhadas sobre o monitoramento sejam impressas (quando uma *thread* adquire ou libera um *lock*, quando acessa uma variável, etc.), a propriedade ladybug.verbose deve ser especificada (o valor da propriedade não é levado em conta, basta especificá-la).
2. Para que o monitoramento seja "desligado" automaticamente enquanto uma *thread* estiver executando o método <clinit> de uma classe (seção 4.7), a propriedade ladybug.ignoreclinit deve ser especificada (o valor da propriedade não é levado em conta, basta especificá-la). A necessidade desta opção é explicada na seção 4.12.3.
3. Para definir o que fazer quando uma condição de disputa é encontrada, é utilizada a propriedade ladybug.warning:

   ⇒ se ladybug.warning é halt, o programa será encerrado na primeira condição de disputa encontrada (mas antes será impressa uma mensagem com informações sobre esta condição de disputa);

   ⇒ se ladybug.warning é throw ou throw1, cada condição de disputa encontrada fará com que uma exceção RCException seja lançada (throw1 indica que uma exceção não deve ser lançada duas vezes devido à mesma variável);

⇒ se ladybug.warning é go_on ou go_on1, serão apenas impressas mensagens com informações sobre as condições de disputas encontradas (go_on1 indica que a mesma mensagem não deve ser impressa duas vezes para uma mesma variável).

As informações sobre um acesso que provocou uma condição de disputa contêm o nome da *thread* em que o acesso ocorreu (toda *thread* tem um nome), dados sobre a variável envolvida (nome da variável ou índice de um elemento de um vetor) e, se disponível, a linha do código fonte onde o acesso foi feito. Note que Ladybug é capaz de dizer que algum trecho do código **precisa** ser protegido por algum *lock*, mas não é capaz de dizer **qual** *lock* deve protegê-lo; por isso, não pudemos construir uma ferramenta capaz de inserir automaticamente sincronizações onde quer que fosse necessário.

4.11 O algoritmo LockSet e as implementações Eraser, EraserFast e EraserGC

Como foi visto na seção 2.3, para o algoritmo LockSet original há quatro estados em que uma variável pode se encontrar. Esses estados, porém, não são suficientes para impedir que alarmes falsos sejam emitidos quando o estado de um objeto é constantemente lido **e** escrito por uma única *thread* que seja diferente daquela que inicializou o objeto, como no exemplo a seguir.

```
class Ponto implements Runnable {
    boolean continua = true;
    private int x = 1, y = 2;

    public void run() {
        while (true) {
◊           x++;   y++;
            synchronized (this) {
                if (!continua) break;
            }
        }
◊◊      System.out.println(x + " " + y);
    }
}
```

```
class Exemplo {
    public static void main(String[] args) {
        Ponto p = new Ponto();
        new Thread(p).start();
        ...
        synchronized (p) {
            continua = false;
        }
        ...
    }
}
```

Nesse exemplo, livre de condições de disputa, uma implementação que utilizasse os mesmos estados definidos pelo algoritmo original iria acusar a existência de condições de

disputa envolvendo as variáveis x e y, nas linhas ◊ e ◊◊. Nossa sugestão para resolver o problema foi a adição de mais dois estados (inicializada-e-lida e inicializada-e-escrita), produzindo o seguinte diagrama de estados:

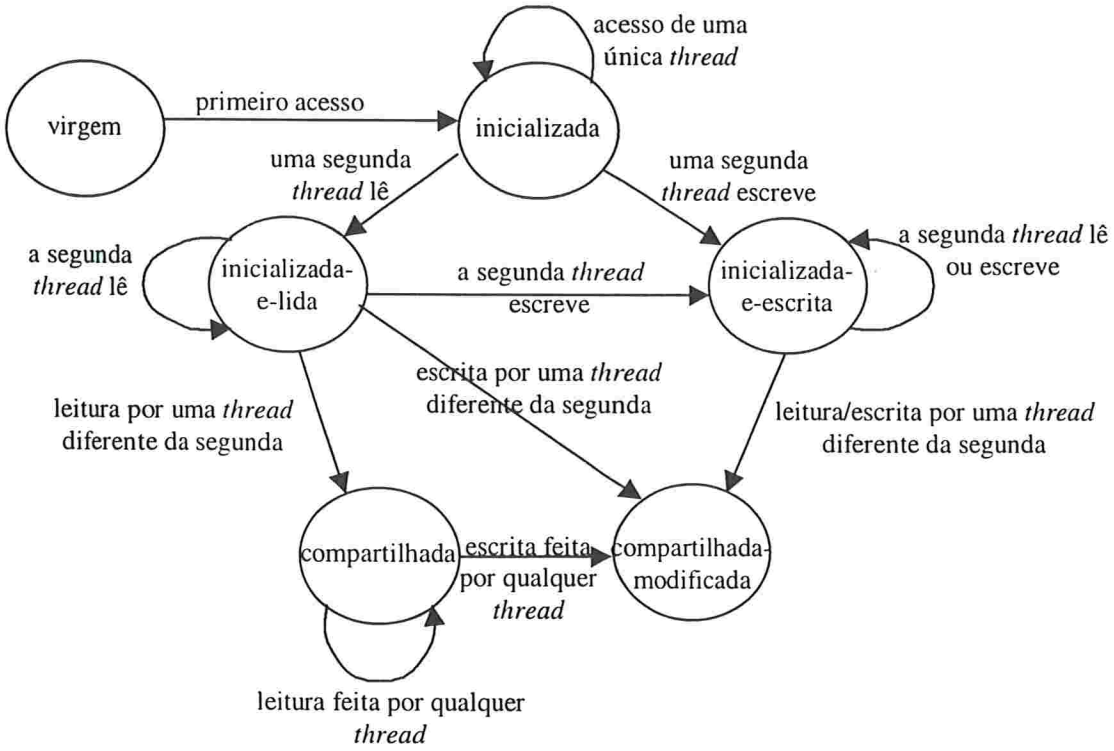


figura 9 – Modificação no diagrama de estados original de Eraser

O estado virgem é apenas conceitual: uma variável só existe para Ladybug após o primeiro acesso e, portanto, o seu estado inicial será, na verdade, inicializada. Enquanto a variável estiver neste estado (isto é, enquanto só uma *thread* acessá-la) o seu conjunto de *locks* não é atualizado. Nos demais estados, esse conjunto é atualizado de acordo com o seguinte procedimento:

- quando a variável sai do estado inicializada, os *locks* sob a posse da *thread* responsável pelo acesso que causou a mudança de estado são utilizados para inicializar o conjunto de *locks* da variável;
- nos demais estados, é feita, a cada acesso, a intersecção do conjunto de *locks* atual com o conjunto de *locks* sob a posse da *thread* que acessou a variável;

- caso o conjunto de *locks* da variável fique vazio, um aviso será emitido apenas se o estado for compartilhada-modificada (ou quando este estado for atingido).

Note que, ainda que o diagrama proposto resolva o problema apontado anteriormente, ele ainda sofre do mesmo problema do original, isto é, incapacidade de determinar o fim do processo de inicialização de uma variável.

Como o algoritmo LockSet ignora completamente as informações de ordem existente no programa sendo monitorado, as classes Eraser, EraserFast e EraserGC só implementam os métodos que manipulam *locks* e os que informam quando um campo foi lido ou escrito (respectivamente, getLocksHeld, readField e writeField). Os demais métodos — threadStart, threadJoin, getCoordPoint e coordinateThreads — têm sua implementação (que não faz nada) simplesmente herdada da classe Ladybug.

4.11.1 Detalhes da implementação

O algoritmo LockSet recebeu três implementações em Ladybug, que diferem entre si pela maneira como representam os *locks* que uma *thread* possui num determinado instante e os *locks* que vêm protegendo uma variável até o momento.

Duas dessas implementações, EraserGC e EraserFast, mantém uma lista de *locks* para cada *thread* e outra para cada variável. Quando um *lock* é adquirido, o objeto ao qual o *lock* estiver associado é acrescentado ao final da lista da *thread* que o adquiriu; quando é liberado, a busca pelo objeto iniciará no fim da lista e, ao ser encontrado, o objeto é removido. Já quando uma variável é acessada, é necessário fazer a intersecção da lista de *locks* da variável com a lista de *locks* da *thread* que a acessa.

Uma lista é uma boa estrutura para representar *locks* em Java porque a linguagem garante que o último *lock* adquirido por uma *thread* será também o primeiro a ser liberado[20]. Mesmo assim, não é possível assumir que é sempre o último *lock* da lista o que deve ser removido, pois Ladybug permite que o usuário insira manualmente invocações aos métodos de monitoramento no código fonte (seção 4.9), tornando impossível utilizar a lista de *locks* de uma

---

[20] O fato de que o método wait da classe java.lang.Object tenha o poder de liberar um *lock* que não foi o último adquirido é irrelevante neste trabalho, pois o método wait não é monitorado por Ladybug (o que não afeta a detecção de condições de disputa).

thread simplesmente como uma pilha. Na ausência de invocações manuais, porém, o custo de adquirir ou liberar um *lock* será constante.

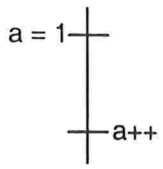
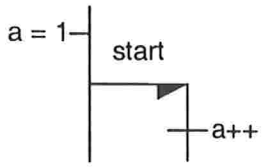
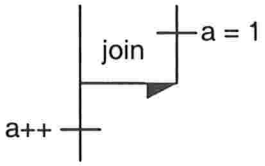
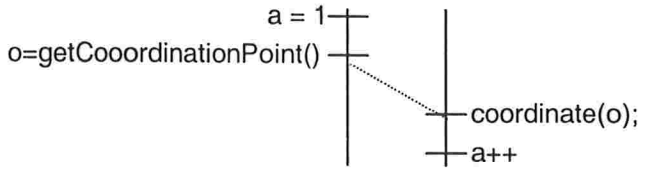
Já a operação de intersecção dos *locks* em posse de uma *thread* com aqueles que vêm protegendo uma variável é potencialmente bem mais lenta: para cada *lock* da lista da variável, é necessário verificar se ele também está presente na lista da *thread* e, caso não esteja, deve ser removido. Este não é de fato um problema grave, pois o número de *locks* na posse de uma *thread* é normalmente pequeno, bem como o número de *locks* utilizado para proteger os acessos à determinada variável [2].

A diferença entre EraserGC e EraserFast está na maneira como cada um armazena referências a outros objetos. EraserGC utiliza referências fracas (isto é, referências que o coletor de lixo não leva em conta) para referir-se às *threads* sendo monitoradas, aos objetos cujos *locks* são adquiridos e aos objetos cujos campos foram acessados; conseqüentemente, quando não houver nenhuma referência a um objeto exceto aquelas mantidas por EraserGC, o objeto poderá ser descartado pelo coletor de lixo. Já EraserFast sempre utiliza referências normais, o que o deixa mais rápido que EraserGC, mas por outro lado pode consumir muita memória, dependendo do programa sendo monitorado.

A classe Eraser, por sua vez, utiliza a idéia da implementação original para manter os conjuntos de *locks*: se dois objetos referem-se a conjuntos de *locks* com os mesmos elementos, então a referência a estes conjuntos deve ser a mesma. Além disso, outras otimizações sugeridas pelos autores de Eraser também foram implementadas, como manter uma tabela *hash* de todos os conjuntos já criados e um *cache* com resultados de intersecções entre conjuntos.

4.12 O algoritmo Dinning-Schonberg e suas implementações

O algoritmo Dinning-Schonberg implementado segue exatamente a proposta original (seção 2.2), exceto pelo fato de que incorpora o suporte a *locks* adquiridos em modo escrita ou leitura (seção 2.3). As duas implementações oferecidas — classes DinningSchonberg e FastDinningSchonberg — diferem apenas na maneira como decidem se dois acessos a uma mesma variável estão ou não ordenados. O princípio aplicado para essa decisão, porém, é sempre o mesmo:

1) dois acessos na mesma *thread* estão sempre ordenados 	2) se uma *thread* inicia outra, os acessos que ocorreram antes do método start certamente precedem os acessos na *thread* iniciada 	3) se uma *thread* espera o término de outra, os acessos na *thread* esperada certamente precedem os acessos posteriores à invocação do método join 
4) acessos antes da invocação a getCoordinationPoint precedem os acessos feitos após a invocação a coordinate (desde que coordinate use o valor retornado por getCoordinationPoint) 		5) a propriedade transitiva vale, isto é, se o acesso A ocorreu antes do acesso B e o acesso B ocorreu antes do acesso C, então o acesso A ocorreu antes do acesso C.

Como descrito na seção 2.2.1, uma *thread* é dividida em blocos, cada um dos quais com um identificador associado; além disso, dados dois identificadores, deve ser possível decidir se os blocos aos quais eles estão associados estão ordenados ou não. Porém, como as definições de bloco e POEG dadas na seção 2.2.1 não se adaptam exatamente ao modelo de concorrência Java, tivemos que redefini-los neste novo contexto. As novas definições encontram-se no Apêndice B; pare este capítulo, os exemplos dados deverão deixar clara a nossa abordagem.

4.12.1 Identificadores na implementação DinningSchonberg

Nesta implementação (que não foi proposta por Dinning e Schonberg, mas por nós), cada identificador é composto por um vetor de bytes e uma lista com outros identificadores, chamada lista de coordenação. (Como será visto adiante, decidir se dois identificadores estão ordenados depende do tamanho do vetor de bytes e do tamanho da lista de coordenação).

Vamos denotar um identificador da seguinte maneira:

$v_1.v_2.....v_n \bullet (e_1; e_2; ...; e_m)$

onde $v_1.v_2.....v_n$ é o vetor e $(e_1; e_2; ...; e_m)$ a lista de coordenação. Uma lista cujo conteúdo não seja importante será denotada por *, e a lista vazia será denotada por ∅, como em 1.5.8 • (3.2 • ∅; 4.6 • ∅).

Em blocos de uma mesma *thread*, apenas a última posição do vetor varia (sempre de forma crescente no decorrer do tempo); além disso, a lista de coordenação em uma mesma *thread* não pode nunca diminuir. Assim, dados dois identificadores cujo vetor tem n posições, se as $n - 1$ posições iniciais são iguais, então os blocos associados estão ordenados. Por exemplo, os identificadores:

2.3.4 • ∅

2.3.7 • (7.8 • ∅)

2.3.9 • (7.8 • ∅; 3.1.9 • ∅)

estão todos associados a blocos de uma mesma *thread* e, portanto, ordenados.

Quando uma *thread* T1 inicia uma *thread* T2, o primeiro bloco da *thread* T2 terá um identificador semelhante ao do bloco atual de T1; a única diferença é que o vetor no identificador do primeiro bloco de T2 conterá um byte a mais, cujo valor inicial será o menor byte possível em Java (que por clareza na explicação assumiremos ser 0, embora na verdade seja –128). Além disso, um novo bloco é iniciado em T1 imediatamente após a invocação do método start para T2. Por exemplo, se o bloco que cria uma nova *thread* tiver o identificador:

3.5.2 • *

então o primeiro bloco da *thread* criada será:

3.5.2.0 • *

e a *thread* iniciadora começará o bloco:

3.5.3 • *

Dessa forma, é possível decidir quais blocos precedem o primeiro bloco de qualquer *thread*. O bloco 3.5.2.0 • *, por exemplo, é precedido pelos blocos

3.5.2 • *	3.5.1 • *	3.5.0 • *			
3.5 • *	3.4 • *	3.3 • *	3.2 • *	3.1 • *	3.0 • *
3 • *	2 • *	1 • *	0 • *		

isto é:

a) ou o vetor é um prefixo de 3.5.2.0 (primeira coluna);

b) ou o identificador precede algum dos identificadores encontrados no item a).

A lista de coordenação contida em cada identificador contém informações sobre precedência obtidas através de outras formas de coordenação entre *threads* (join, coordinate):

- se, no bloco B, a *thread* T invoca o método cordinate, um identificador (retornado por getCoordinationPoint) passa a fazer parte de todos os blocos de T seguintes a B;
- se, no bloco B, a *thread* T invoca o método join, o identificador do último bloco da *thread* esperada passa a fazer parte de todos os blocos de T seguintes a B.

Por exemplo, o identificador

5.6 • (6.9 • (7.8.2 • ∅); 10.8.8.0 • ∅; 14.3 • (16.2.2 • ∅))

identifica um bloco que é precedido pelos blocos 6.9, 7.8.2, 10.8.8.0, 14.3 e 16.2.2 (e por todos os blocos que precedem estes blocos). Note que, para decidir se dois identificadores estão ordenados, pode ser preciso percorrer toda a lista de coordenação.

A figura 10 exemplifica os valores que os identificadores assumem conforme novos blocos são criados em diversas *threads*.

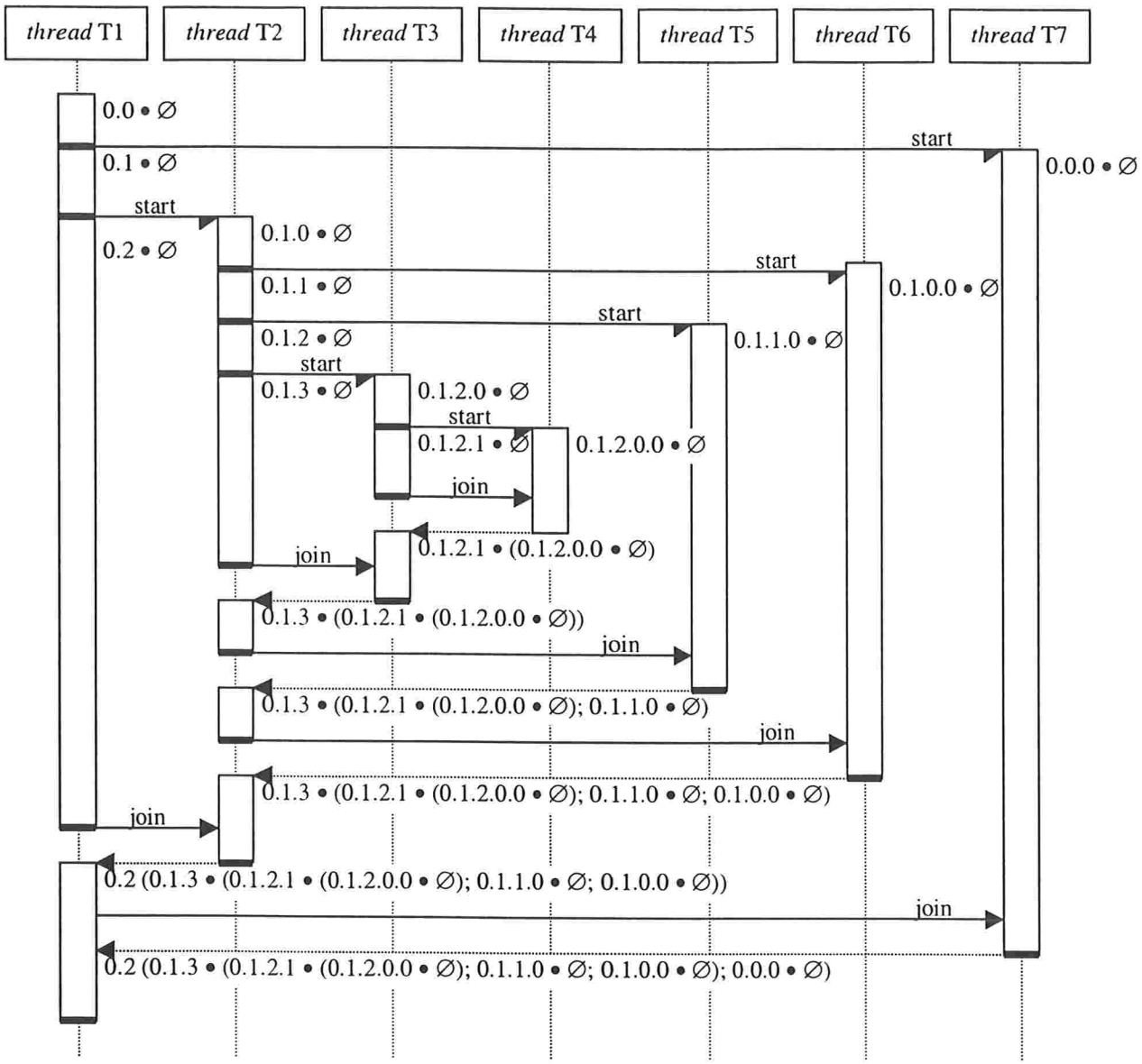


figura 10 – Exemplo de funcionamento de DinningSchonberg

4.12.2 Identificadores na implementação DinningSchonbergFast

Nesta implementação, baseada no algoritmo *Task Recycling* [3], cada identificador é composto por um inteiro, um byte e um vetor de bytes, chamado vetor *parent*. Este vetor pode tornar-se bem grande, mas garante que decidir se dois blocos estão ordenados seja feito em tempo constante.

Vamos denotar um identificador da seguinte maneira:

a.b [$v_1.v_2.....v_n$] (ou a.b.V se o vetor não for importante no contexto)

onde a é o inteiro, b é o byte, e $v_1.v_2.....v_n$ é o vetor *parent*, como, por exemplo, 3.5 [0.1.0.5].

O inteiro a é único para uma mesma *thread* e é o mesmo para todos os blocos desta *thread*. O byte b é diferente para cada bloco da *thread*, mas varia sempre de forma crescente com o decorrer do tempo. Por fim, o vetor $v_1.v_2.....v_n$ registra informações sobre identificadores precedentes. Por exemplo, o valor 3 na posição 9 do vetor de um identificador significa que os blocos 9.3.V, 9.2.V, 9.1.V e 9.0.V precedem este identificador. Por uniformidade, mantemos a invariante $v_a = b$, refletindo o fato de que os blocos a.(b-1).V, a.(b-2).V,..., a.0.V precedem o identificador a.b.V.

Quando uma *thread* T1 inicia uma *thread* T2, o primeiro bloco da *thread* T2 terá seu vetor *parent* herdado do bloco atual de T1, indicando que tudo que precedeu T1 até aquele momento também precede T2 (note que o vetor herdado precisará ser modificado para obedecer a invariante $v_a = b$). Já quando uma *thread* T1 invoca, em um bloco B, o método join para a *thread* T2 (cujo último bloco será chamado de C), então o próximo bloco de T1 (seguinte ao bloco B) deve incorporar as informações sobre os blocos que precedem C. Isso é feito comparando-se cada elemento dos vetores dos blocos B e C e tomando o maior valor. Por exemplo, se no bloco de T1 identificado por 3.5 [4.2.0.5] é invocado o método join para uma *thread* cujo último bloco é identificado por 2.9 [1.3.8], então o próximo bloco de T1 será 3.5 [4.3.8.5].

A atualização do vetor *parent* causada pela invocação a coordinate em um bloco B é semelhante: o valor retornado por getCoordinationPoint (que é um identificador de bloco) é combinado com o vetor *parent* do identificador do bloco B (sempre tomando-se o maior valor), produzindo o vetor para os blocos seguintes a B.

A figura 11 exemplifica os valores que os identificadores assumem conforme novos blocos são criados em diversas *threads*.

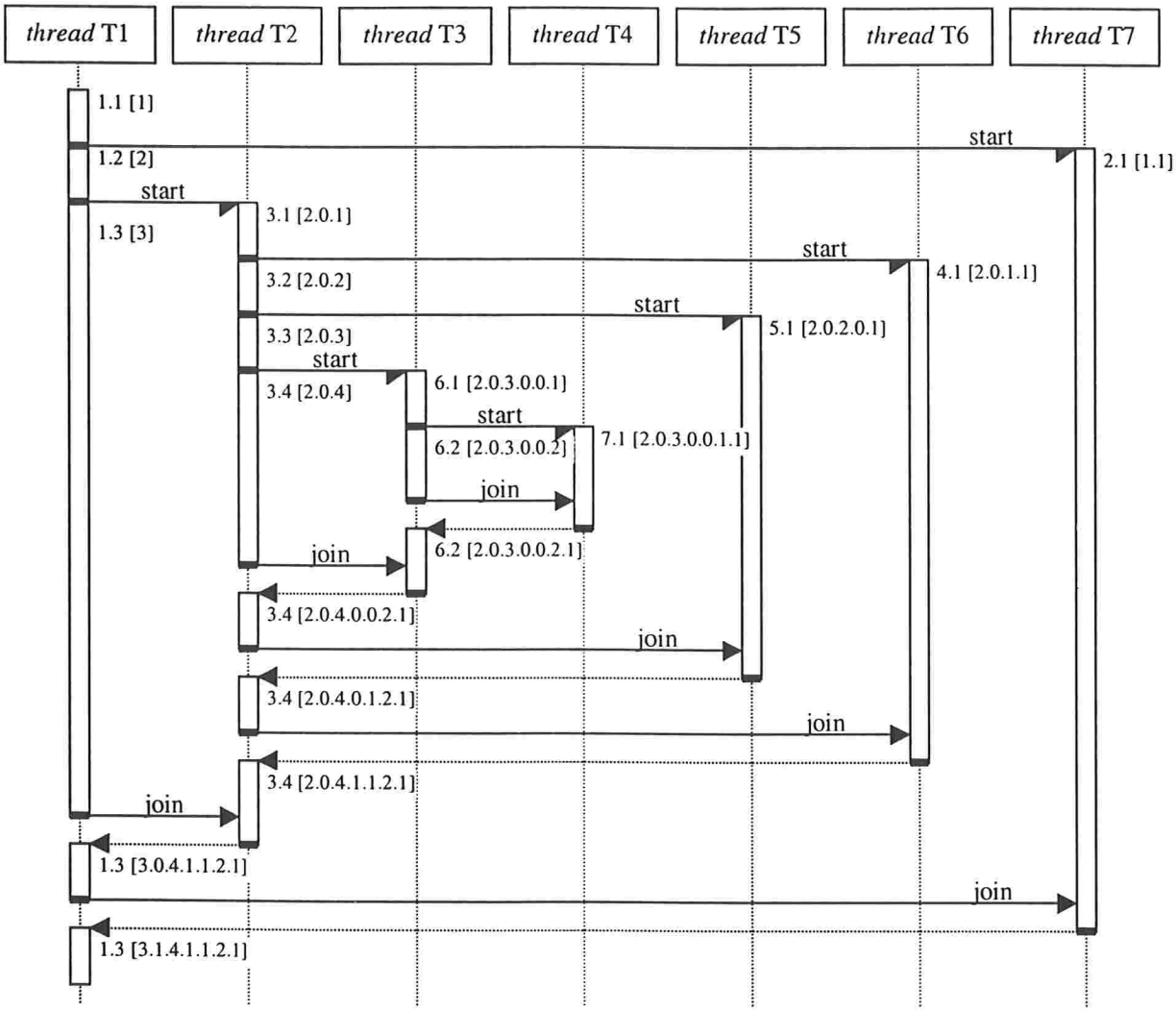


figura 11 – Exemplo de funcionamento de DinningSchonbergFast

4.12.3 Novamente o método <clinit>

O método <clinit> (visto em detalhes na seção 4.7) pode ser a causa de falsos alarmes emitidos por Ladybug se o algoritmo escolhido para o monitoramento é Dinning-Schonberg. Tome, por exemplo, o seguinte programa:

```
interface X {
    int[] vetor = { 1, 2 };
}

class Principal implements Runnable {
    public static void main(String[] args) {
        Principal p = new Principal();
        new Thread(p).start();
        new Thread(p).start();
    }

    public void run() { System.out.println(X.vetor[0]); }
}
```

A primeira das *threads* criadas que acessar X.vetor inicializará também a interface X e, conseqüentemente, executará seu método <clinit> (que, neste caso, conterá código para criar um vetor com dois inteiros, inicializar seus elementos e atribuí-lo à variável vetor). Note que, independente de qual *thread* acessar X.vetor primeiro, a inicialização do vetor **precederá** este acesso, não havendo, portanto, condições de disputa. Mas para o algoritmo Dinning-Schonberg, o que ocorre é simplesmente a escrita da posição 0 do vetor por uma *thread* e a leitura da mesma posição por outra *thread*, não ordenada com a primeira. E, como não há um *lock* que proteja ambos os acessos, será emitido um aviso.

Especificar a propriedade ladybug.ignoreclinit ao executar o programa monitorado faz com que o monitoramento seja desligado para as *threads* enquanto ela estiverem executando um método <clinit>. Como efeito, acessos realizados durante a inicialização da classe ou interface não entrarão em conflito com nenhum acesso posterior, já que, para Ladybug, eles jamais ocorreram.

A opção de desligar o monitoramento durante a inicialização das classes é deixada ao usuário porque os acessos a variáveis dentro do método <clinit> podem de fato fazer parte de condições de disputa, como neste exemplo:

```
interface Y {
◊◊  float PI = Principal.pi;
}

class Principal {
    static float pi = 2.14;

    public static void main(String[] args) {
        new Thread() {
            public void run() { System.out.println(Y.PI); }
        }.start();

◊       Principal.pi++;
    }
}
```

Como a inicialização da interface Y, que lê o valor de Principal.pi, pode ocorrer concorrentemente com ◊, onde o valor de Principal.pi é escrito, existe uma condição de disputa (envolvendo ◊ e ◊◊).

Quando a instância da classe Ladybug utilizada no monitoramento é Eraser, EraserGC ou EraserFast, a propriedade ladybug.ignoreclinit é ignorada, pois os estados das variáveis utilizados pelo algoritmo LockSet são suficientes para abranger, também, a inicialização de classes e interfaces (por outro lado, o algoritmo LockSet falharia no último exemplo).

Capítulo 5

Resultados experimentais

Para verificar a efetividade de Ladybug na detecção de condições de disputa, nós a testamos com um servidor HTTP [25], uma biblioteca para clientes HTTP [35] e doze programas de alunos da disciplina "Programação Concorrente". Ladybug foi ainda testada com alguns problemas clássicos de concorrência e com pequenos trechos de código preparados por nós, para que pudéssemos ter uma medida da degradação do desempenho trazida pela invocação de métodos de monitoramento. Os testes foram realizados em um Pentium II MMX com 128 Mb de memória, Windows 98 e SDK 1.3.

5.1 Medidas de desempenho

Ladybug foi aplicada a implementações concorrentes do problema das N rainhas[21] e de multiplicação de matrizes. Nestas implementações (retiradas de [14]) não são utilizados *locks* para sincronização, mas apenas a ordenação imposta pelo início e término de *threads* (chamada sincronização *fork/join*). Por isso, não utilizamos o algoritmo Eraser nestes testes.

A tabela 1 mostra os resultados médios obtidos utilizando-se o algoritmo DinningSchonbergFast (4.12.2) com diferentes números de *threads* para o problema das N rainhas com N = 11:

Número de *threads*	Tempo (s)		Degradação
	Sem monitoração	Com monitoração	
2	0,16	1,01	6,3
4	0,16	1,62	10,3
8	0,16	3,58	22,4
16	0,17	5,50	32,6

tabela 1 – Tempo necessário para encontrar a solução para o problema das 11 rainhas

Já para o algoritmo DinningSchonberg (4.12.1), o desempenho foi extremamente ruim (tanto que sequer esperamos o término da execução). A razão do fraco desempenho encontra-se na rapidez com que o tamanho dos identificadores cresce num problema como este, e o

[21] Este problema consiste em encontrar uma disposição de N rainhas em um tabuleiro NxN, de forma que nenhuma rainha possa atacar outra segundo as regras do xadrez.

desempenho do algoritmo DinningSchonberg, como vimos, depende do tamanho dos identificadores.

A tabela 2 contém os resultados para diferentes instâncias do problema de multiplicação de matrizes (novamente, apenas para o algoritmo DinningSchonbergFast). As duas últimas linhas devem ser interpretadas com cuidado, pois podem dar a impressão de que a degradação aumenta devido ao aumento do número de *threads*. O que ocorre, na verdade, é que um número maior de *threads* prejudica a execução não monitorada do programa (num ambiente com um único processador), enquanto que, para a execução monitorada, a degradação trazida pelo aumento do número de *threads* é desprezível frente àquela trazida pela monitoração.

Número de *threads* / dimensões	**Tempo (s)**		**Degradação**
	Sem monitoração	**Com monitoração**	
2 / 16x16	0,37	1,58	4,27
2 / 32x32	0,38	12,12	32,3
4 / 32x32	0,38	11,75	31,3
2 / 64x64	0,47	102,10	217,1
4 / 64x64	0,6	102,13	170

tabela 2 – Tempo necessário para realizar uma multiplicação de matrizes

Implementamos ainda as soluções propostas em [1] para o problema dos filósofos e para o problema de vários leitores e um atualizador. Os resultados para diferentes instâncias destes problemas e para todos os algoritmos de detecção oferecidos por Ladybug encontram-se respectivamente nas tabelas 3 e 4.

Número de filósofos	**Total de vezes que os filósofos conseguiram ambos os garfos no período de 5 segundos**					
	Sem monitoração	**Eraser**	**EraserFast**	**EraserGC**	**Dinning-Schonberg**	**Dinning-Schonberg Fast**
3	180	178	178	165	142	144
30	1797	1579	1621	1426	290	294
250	13649	1574	1558	1106	167	176

tabela 3 – Comparação de métodos de monitoramento no problema dos filósofos

Número de leitores / escritores	Total de vezes que leitores e escritores consegue acesso a um *buffer* no período de 5 Segundos					
	Sem monitoração	Eraser	EraserFast	EraserGC	Dinning-Schonberg	Dinning-Schonberg Fast
20 / 4	980/390	937/356	950/360	933/37	590/68	600/77
200 / 40	9865/3732	2322/0	2165/0	1640/26	157/0	162/0
10 / 50	492/4857	10/2127	182/1839	150/1621	33/244	38/259

tabela 4 - Comparação de métodos de monitoramento no problema de múltiplos leitores/um atualizador

Tanto no problema dos leitores/atualizadores quanto no problema dos filósofos, as *threads* passam algum tempo dormindo (nas atividades comer/pensar ou simulando o processamento do *buffer*). Por isso, criamos dois exemplos em que as *threads* passam todo o seu tempo apenas acessando variáveis compartilhadas — o que, embora soe irreal, dá uma idéia mais precisa da degradação causada pelo monitoramento.

No primeiro teste, diversas *threads* acessam regiões disjuntas de um vetor (isto é, não há disputa), incrementando o valor de seus elementos; já no segundo teste, as áreas podem se sobrepor, sendo utilizados *locks* para garantir que não haja condições de disputa. Cada um destes testes está dividido em dois subcasos:

- é utilizado um vetor de inteiros
- é utilizado um vetor de objetos pertencentes à classe Integer (que, por serem imutáveis, não podem ser incrementados, precisando sempre ser substituídos por novos objetos).

Os dados do primeiro e segundo testes encontram-se respectivamente nas tabelas 5 e 6.

Tipo do vetor	Número médio de incrementos realizados por uma *thread* no período del milissegundo (número entre parênteses: degradação)					
	Sem monitoração	Eraser	EraserFast	EraserGC	Dinning-Schonberg	Dinning-Schonberg Fast
inteiros	26003	5,1 (5098)	3,9 (6667)	3,0 (8668)	2,2 (11820)	2,2 (11820)
objetos	603	4,0 (151)	3,0 (201)	2,1 (287)	1,6 (179)	1,6 (179)

tabela 5 – Teste de desempenho em que *threads* não disputam a mesma posição do vetor

Tipo do vetor	Número médio de incrementos realizados por uma *thread* no período de 1 milissegundo (número entre parênteses: degradação)					
	Sem monitoração	Eraser	EraserFast	EraserGC	Dinning-Schonberg	Dinning-Schonberg Fast
inteiros	743	3,0 (248)	2,5 (297)	1,6 (464)	0,3 (2476)	0,3 (2476)
objetos	285	2,7 (106)	2,1 (136)	1,2 (238)	0,2 (1425)	0,2 (1425)

tabela 6 – Teste de desempenho em que *threads* podem entrar em disputa pelo acesso à mesma posição do vetor

5.2 Efetividade da ferramenta

Ladybug foi testada com quatorze programas de alunos de Programação Concorrente cujo tamanho variava entre 400 e 2000 linhas, tendo descoberto condições de disputa em todos. Em doze deles, foi possível verificar que as condições de disputa apontadas por Ladybug poderia realmente acarretar uma falha no programa; os outros dois programas, porém, eram um tanto confusos, e não puderam ser analisados.

Ladybug também foi aplicado na biblioteca para clientes HTTP "HTTPClient" [35] e no servidor HTTP "Jigsaw" [25], e apontou condições de disputa nos dois. As informações dadas por Ladybug foram suficientes para determinar que os acessos envolvidos nas condições de disputa apontadas não eram, de fato, protegidos por um *lock* em comum. Mas tanto Jigsaw quanto HTTPClient são programas **muito** grandes, e não foi possível determinar se as condições de disputa apontadas poderiam causar uma falha no programa ou não.

Tanto com os programas dos alunos quanto com o cliente e o servidor HTTP não foi possível perceber nenhuma degradação no desempenho.

5.3 Disponibilidade da ferramenta

Ladybug está disponível em http://www.ime.usp.brdcc/posgrad/teses/junior.

Capítulo 6

Trabalhos relacionados

Neste capítulo são abordadas brevemente outras técnicas para a análise de programas concorrentes, em especial a análise estática. Também são descritos alguns trabalhos sobre geração de código otimizado para programas concorrentes, e outras críticas ao modelo de concorrência utilizado em Java são levantadas.

6.1 Outras técnicas para análise de programas concorrentes

Técnicas de **análise estática** consistem em construir um modelo do programa concorrente a ser analisado e então verificar se neste modelo ocorrem certas propriedades indesejáveis, sejam elas específicas do sistema ou genéricas (como existência de *deadlocks* e condições de disputa). Como o tamanho do modelo cresce muito rápido e torna intratável a análise mesmo para programas pequenos, os artigos sobre análise estática abordam maneiras de gerenciar esse problema. Porém, a detecção de propriedades como *deadlock* ou condições de disputa é um problema NP-completo [31]; conseqüentemente, qualquer técnica que se proponha a verificar a existência dessas propriedades tem um pior caso exponencial, ou não se aplica a alguns casos, ou pode acusar erros inexistentes, ou necessita da interferência do usuário de forma a guiar a análise.

O sistema pode ser modelado como um grafo representando o conjunto de estados do programa e as transições possíveis entre esses estados. Mas gerar esse grafo de maneira direta, isto é, com uma instrução em cada nó, pode levar a um grafo muito grande (é o chamado "problema da explosão de estados"), o que, além de deixar a análise intratável, pode ser por si só impraticável dadas as limitações de espaço. Assim, técnicas para reduzir o tamanho do modelo também são pesquisadas.

Flavers [15] é uma ferramenta de análise estática adaptada para Java (pois foi originalmente escrita para a análise de programas Ada), capaz de verificar erros como:

- uma *thread* invocando o método *join* de outra *thread* ainda não iniciada;
- uma *thread* invocando o método *start* de outra *thread* já iniciada;
- uma *thread* tentando interagir com outra que já terminou;

- uma *thread* invocando o método *notify*, sem que haja nenhuma *thread* em estado *wait*;
- uma *thread* que não pode sair do estado *wait*, pois nenhuma outra *thread* invoca o método *notify* ou *notifyAll*.

A técnica utilizada por Flavers, embora polinomial, apresenta algumas desvantagens: não modela situações que necessitam de contagem (para obter, por exemplo, o fato de que o número de invocações a *wait* e *notify* deve estar balanceado) nem é completamente automática, dependendo em parte da habilidade do analista para que não detecte erros espúrios [15]. Além disso, os resultados empíricos obtidos pelos autores referem-se apenas ao protótipo de Flavers, testado apenas com instâncias pequenas de problemas clássicos de concorrência, como o problema dos filósofos (com três filósofos).

Os mesmos autores de Flavers propuseram ainda um algoritmo polinomial capaz de fornecer uma estimativa conservadora das sentenças que podem ser executados em paralelo [16]. Além da utilidade desse algoritmo para a análise estática (não pode haver condições de disputa entre duas instruções que não podem executar em paralelo), este algoritmo também pode ser útil na otimização de código (uma vez determinado que duas *threads* nunca tentam simultaneamente entrar na mesma região crítica, a aquisição e liberação de *locks* para entrada nesta região pode ser removida).

Em [21], são propostas algumas técnicas para redução do modelo analisado específicas para a linguagem Java: é possível agrupar ações que atualizam variáveis locais a uma *thread* ou as variáveis protegidas por um *lock* sem afetar o resultado que seria obtido num modelo sem o agrupamento. Alguns tipos de variáveis podem ser trivialmente detectados como locais (aquelas declaradas no corpo de um método ou como parâmetro formal), mas, para outros casos, são necessárias heurísticas ou uma estimativa (conservadora) do estado do *heap* durante a execução, para, por exemplo, determinar se uma variável no *heap*, isto é, um objeto, só é acessada por uma *thread* (essas idéias estão relacionadas também à otimização; veja a seção 6.3). Verificar se um mesmo *lock* protege todos os acessos a uma variável também requer heurísticas, já que os mecanismos de proteção podem ser muitos sofisticados para serem detectados estaticamente.

Java2Spin [6][9] é uma ferramenta capaz de utilizar diretamente o código Java para gerar um modelo para análise de *deadlock*. O código Java é compilado para uma linguagem de

descrição formal chamada **Promela**, para que possa ser então analisado por outra ferramenta, chamada Spin. Um dos autores também já construiu uma ferramenta que gera uma rede de Petri a partir de código Java. Redes de Petri foram propostas em 1962 para analisar sistemas concorrentes em geral, e talvez sua maior vantagem seja a quantidade de teoria e ferramentas já construídas para analisá-las [27][31].

A ferramenta rccjava [10] utiliza um sistema de verificação formal de tipos para detectar condições de disputa em programas Java, embora obrigue o usuário a inserir anotações no código fonte, na forma de comentários. Mesmo assim, a ferramenta mostrou-se efetiva: ela foi utilizada na verificação de mais de 40 mil linhas de código (em média não exigindo mais que 20 anotações para cada 1000 linhas), tendo encontrado erros em programas teoricamente maduros, como a classe java.util.Vector, que faz parte da biblioteca de classes Java.

DejaVu [28] é uma ferramenta de **repetição determinística**: ela obriga um escalonamento a ser sempre seguindo (isto é, torna a execução determinística), possibilitando, assim, que o usuário depure o programa de forma "tradicional". DejaVu estende a máquina virtual Java da seguinte forma:

- grava, durante a execução do programa, informações sobre o escalonamento das *threads*;
- nas execuções posteriores do programa obriga que esse mesmo escalonamento seja seguido.

6.2 Críticas ao modelo de concorrência adotado em Java

Não só o modelo de memória em Java (seção 1.4) é vítima de críticas; há quem condene **todo** o modelo de concorrência utilizado em Java [4]. Alguns exemplos de "falhas" nesse modelo são:

- a ativação de *threads* é explicita, o que é propenso a erros (como ativar uma *thread* mais de uma vez ou esquecer de ativá-la);
- a comunicação de dados entre *threads* não é explícita;

- os efeitos do término de uma *daemon thread*[22] não estão especificados;
- invocar um método sincronizado não garante que os acessos serão seguros, já que ele pode chamar métodos não sincronizados;
- há o risco de *deadlock* devido ao problema dos "monitores aninhados", isto é, é possível que uma *thread* retenha um *lock* enquanto espera ser notificada por outra *thread*, e a *thread* capaz de fazer a notificação esteja impedida de prosseguir pois está tentando adquirir esse *lock*;
- as restrições de sincronização ficam embutidas no código, o que as torna mais difíceis de identificar;
- a política de escalonamento é dependente da implementação da máquina virtual;
- não há garantias sobre qual *thread* é acordada por um notify();
- falta um mecanismo de atraso absoluto (como "durma até 15h03min");
- a granularidade de sleep() não é garantida.
- a linguagem é particularmente sensível à "anomalia de herança" [20], um termo cunhado para descrever a incapacidade de se herdar código sincronizado em certas linguagens concorrentes e orientadas a objetos (como Java).

A razão de algumas destas deficiências (e de todo este trabalho) é o fato de que Java não implementa o conceito original de monitores [5][7], com o qual só seria possível acessar uma variável do monitor através de um método do próprio monitor. Por este motivo, em [10] é apresentado um dialeto da linguagem Java — Guava — cuja sintaxe impede que uma variável compartilhada possa ser acessada sem sincronização (isto é, fora de um monitor). Este dialeto permite ainda certas otimizações automáticas, impossíveis em Java.

6.3 Otimização de código Java concorrente

A implementação de sincronização na máquina virtual deve ser ao mesmo tempo eficiente — pois programas Java sincronizam com muita freqüência[23] — e econômica quanto

[22] Há dois tipos de *threads* em Java: *user threads* e *daemon threads*. Quando não há mais *user threads* rodando, a execução da máquina virtual Java termina; já as *daemon threads* não impedem o fim da execução da máquina virtual.

ao espaço utilizado — pois, embora todos os objetos possam servir para sincronização, apenas uma pequena parte deles realmente é utilizada para esse fim.

A implementação original da máquina virtual Java utilizava um esquema eficiente apenas em termos de espaço: era mantida uma tabela global de monitores, onde cada entrada era associada a um objeto. Quando ocorria uma operação de sincronização em um objeto, a *thread* verificava se o monitor associado já existia nessa tabela, sendo criado e instalado se necessário. Embora o espaço utilizado fosse proporcional ao número de objetos em que efetivamente houvesse sincronização, qualquer operação de sincronização envolvia ao menos uma busca na tabela para localizar o monitor. Além disso, o acesso à tabela precisava ser protegido, diminuindo a concorrência.

Thin locks [11] foram utilizados em diversas versões do JDK da IBM. Eles exigem 24 bits no cabeçalho do objeto, um dos quais para indicar se o *lock* está em estado "magro" ou "gordo". Enquanto não houver disputa, o *lock* permanece em seu estado magro, sendo necessárias poucas instruções de máquina para cada operação *lock* ou *unlock*, incluindo uma operação atômica, *compare and swap*, apenas no caso de *lock*. Quando a instrução *compare and swap* falha, o que indica que o *lock* já está em posse de outra *thread*, o *lock* deve ser colocado em seu estado gordo:

- a *thread* que não conseguiu a posse do *lock* fica em *busy-waiting* até que o *lock* seja liberado;
- cria, então, um monitor e insere-o em uma tabela, guardando a posição da inserção nos outros 23 bits exigidos. Esse monitor passa a ser usado em todas as operações de sincronização subseqüentes, isto é, o *lock* nunca mais volta ao seu estado magro.

Embora *busy-waiting* não seja uma característica desejada num algoritmo, os autores argumentam que esse custo é diluído no tempo de vida do objeto, já que pode ocorrer no máximo uma vez (para cada objeto). Além disso, eles afirmam que, uma vez que houve disputa pelo *lock*, ela provavelmente ocorrerá novamente, e por isso o *lock* deve permanecer em seu estado gordo. Onodera e Kawachiya [33] viram que isso não é, contudo, verificado na prática, o que mostra que a decisão de deixar o *lock* gordo para sempre pode não ser a melhor em

[23] Em [32], foi medida a quantidade de operações de sincronização do um compilador Java. O resultado encontrado foi 765.000 operações de sincronização por segundo.

termos de desempenho. Motivado por isso, desenvolveram o *tazuki lock*, uma espécie de *thin lock* sem *busy-waiting* e capaz de voltar ao estado magro.

Para Agesen *et al.* [32], os 24 bits no cabeçalho (também presentes no *tazuki lock*) representam uma sobrecarga significativa, e por isso propõem o *meta-lock*, que utiliza um algoritmo mais complexo, mas apenas dois bits no cabeçalho do objeto (caso não haja disputa). Neste algoritmo também não há *busy-waiting*.

Mesmo reduzindo-se o custo de aquisição de *locks*, esse custo nunca será nulo. Muitas vezes, porém, pode-se descobrir que um certo objeto nunca é acessado por mais de uma *thread* (isto é, o objeto **não escapa** de uma *thread*), e que as operações de sincronização feitas com ele podem seguramente ser eliminadas. Bogda e Hölzle [24] propõem um algoritmo que detecta quais objetos são acessados por apenas uma *thread*, e criam automaticamente versões não sincronizadas desses objetos — nos testes realizados, o algoritmo foi capaz, em alguns casos, de retirar mais de 70% das sincronizações; em outros, porém, retirou menos de 20%. Jong *et al.* [29] vão um pouco além, e aproveitam a análise de escape para também otimizar a alocação de objetos. A idéia é que objetos que não escapam de um método (ou seja, que não podem ser acessados após o término do método que os criou) podem ser alocados no *stack*, e não necessariamente no *heap*. Objetos alocados no *stack*, além de não requererem ocasionalmente a sincronização com outras *threads* (seção 1.4), diminuem a coleta de lixo, já que são automaticamente destruídos no final do método.

Capítulo 7

Conclusões

Embora Java utilize um "mecanismo de alto nível para garantir que apenas uma *thread* por vez execute a região do código protegida" [23], isso não significa que o programador possa se sentir seguro com este mecanismo. Pelo contrário, toda a segurança prometida pelos livros de Java (e pela Sun) serve apenas para dar ao programador a sensação ilusória de que monitores, existentes há 25 anos, são o que de melhor se pode esperar para proteger código concorrente. Porém, é muito fácil esquecer de sincronizar um bloco de código ou utilizar o objeto errado na sincronização, e em Java você não será alertado sobre isso.

Este trabalho descreveu o funcionamento de uma ferramenta, Ladybug, capaz de apontar condições de disputa em programas Java. Como a ferramenta que lhe serviu de base, Eraser, Ladybug também rescreve o código compilado inserindo automaticamente invocações a métodos de monitoramento. Outra abordagem teria sido alterar uma máquina virtual para torná-la capaz de monitorar o programa em execução, o que, embora possivelmente causasse uma degradação menor no desempenho dos programas sendo monitorados, resultaria em uma ferramenta que funcionaria em apenas uma plataforma.

Desenvolvemos nossa própria biblioteca para manipulação de classes compiladas porque não encontramos nenhuma ferramenta para alteração de *bytecodes* que se adequasse às nossas necessidades (inicialmente consideramos BIT [19], mas o código dessa ferramenta, embora disponível, não pode ser modificado, e como BIT não atendia exatamente às nossas necessidades de monitoramento — e ainda continha alguns erros — a idéia teve de ser abandonada). Contudo, optamos por construir uma biblioteca genérica, que pudesse ser utilizada no contexto de outros trabalhos e não apenas por Ladybug.

A natureza dinâmica de Java e seus requisitos de compatibilidade, aliados à sua relativa imaturidade e a uma definição informal que tenta ser ao mesmo tempo precisa, foram obstáculos extras à implementação de Ladybug. Deparamo-nos com definições e interpretações contraditórias sobre o comportamento que a máquina virtual devia exibir em certas situações, e tivemos que nos preocupar com situações das quais um programador dificilmente tiraria proveito. Tivemos que ser conservadores na nossa abordagem de detecção, o que algumas

vezes penalizou a detecção de condições de disputa em programas normais, seja pela degradação do desempenho, seja por avisos sobre condições de disputa inexistentes.

Optamos por oferecer Ladybug não apenas com o algoritmo LockSet, utilizado em Eraser, mas também com aquele proposto por Anne Dinning e Edith Schonberg, para, assim, dar ao usuário uma alternativa de detecção mais precisa e genérica (ainda que também mais lenta), já que o algoritmo LockSet não é capaz de detectar que os métodos start e join podem ser utilizados como formas de sincronização (ele é uma simplificação do algoritmo de Dinning e Schonberg). Além disso, é possível escolher implementações que privilegiam a economia de memória (pois interagem com o coletor de lixo) ou a velocidade de execução. Os testes mostraram que diversas vezes uma implementação "inocente" de conjuntos de *locks* pode ser tão ou mais efetiva que a implementação otimizada proposta pelos autores de Eraser.

É possível construir programas que, quando monitorados, apresentem uma degradação de quase doze mil vezes em relação àqueles sem o código para monitoração (tabela 5, Capítulo 5). Nos testes realizados com programas "reais", porém, não conseguimos detectar qualquer degradação na saída do programa, exceto quando solicitávamos que todas as atividades monitoradas fossem também impressas.

Analisando os erros apontados por Ladybug nos programas testados, verificamos que muitas vezes a disciplina de proteção de variáveis por um mesmo *lock*, recomendada pela especificação da linguagem Java [23] e considerada pelos autores de Eraser como "a forma de programas concorrentes modernos se sincronizarem", simplesmente não é respeitada. Sempre que pode, o programador tenta eliminar a palavra synchronized de algum trecho do código ou criar esquemas de combinação de *locks* para aumentar a concorrência no programa (e, da mesma maneira que ocorre com programas seqüenciais, pode ser um pesadelo entender um programa com estas "otimizações manuais", ou mesmo convencer-se de que ele funciona). Como resultado, toda vez que programador pensar "acho que este trecho não precisa estar protegido por este *lock*", ele provavelmente estará fazendo com que Ladybug emita mais avisos sobre condições de disputa — e a experiência, ao menos com os exercícios dos alunos de graduação, mostrou que esses avisos não podem ser apenas ignorados.

Em resumo, as contribuições que trouxemos foram:

- melhoria do Eraser, eliminando alguns alarmes falsos;

- incorporação das otimizações de implementação sugeridas pelos autores de Eraser;
- oferecimento de outras alternativas de implementação para manter conjuntos de *locks* que lidam também com o coletor de lixo;
- implementação do algoritmo de Dinning e Schonberg com suporte a *locks* adquiridos em modo leitura e escrita;
- novas definições de blocos e POEG que se adaptam ao modelo de concorrência utilizado em Java;
- um pacote de classes para manipulação de arquivos .class.

Apêndice A

O pacote classfile

Este apêndice descreve uma biblioteca que desenvolvemos para examinar e alterar um arquivo .class. Esse pacote foi utilizado para modificar arquivos .class de acordo com as necessidades enumeradas no Capítulo 4.

A biblioteca aqui descrita constitui um produto incidental mas independente do restante do trabalho. Os exemplos de utilização dados mostram o poder dessas classes, mas não são fundamentais para o entendimento de Ladybug.

A organização deste pacote, bem como sua forma de utilização, é descrita a seguir, através de figuras UML e exemplos.

A.1 A representação de constantes

As categorias de constantes (seção 3.3) que podem ser encontradas em um arquivo .class estão representadas pelas classes do pacote br.ime.usp.classfile.constants (veja a figura 12).

Para criar uma constante inteira, por exemplo, é suficiente fazer

```
new ConstantInteger(15);
```

O relacionamento entre constantes é feito utilizando-se índices da tabela de constantes à qual as constantes pertencem. Assim, para criar uma constante que simbolize a classe java.lang.System em uma certa tabela t, é necessário um trecho de código como o seguinte:

```
int índice = t.add(new ConstantUtf8("java/lang/System"));
// t.add insere a constante no final da tabela,
// retornando o índice em que a inserção foi feita
t.add(new ConstantClass(índice));
```

Constantes podem ainda ser removidas, substituídas ou inseridas em posições arbitrárias da tabela. Os índices das constantes seguintes a uma constante removida ou inserida, porém, serão alterados, e os relacionamentos utilizando os valores antigos precisarão ser refeitos.

Como constantes têm um papel fundamental em uma classe, outros exemplos serão vistos no restante deste apêndice.

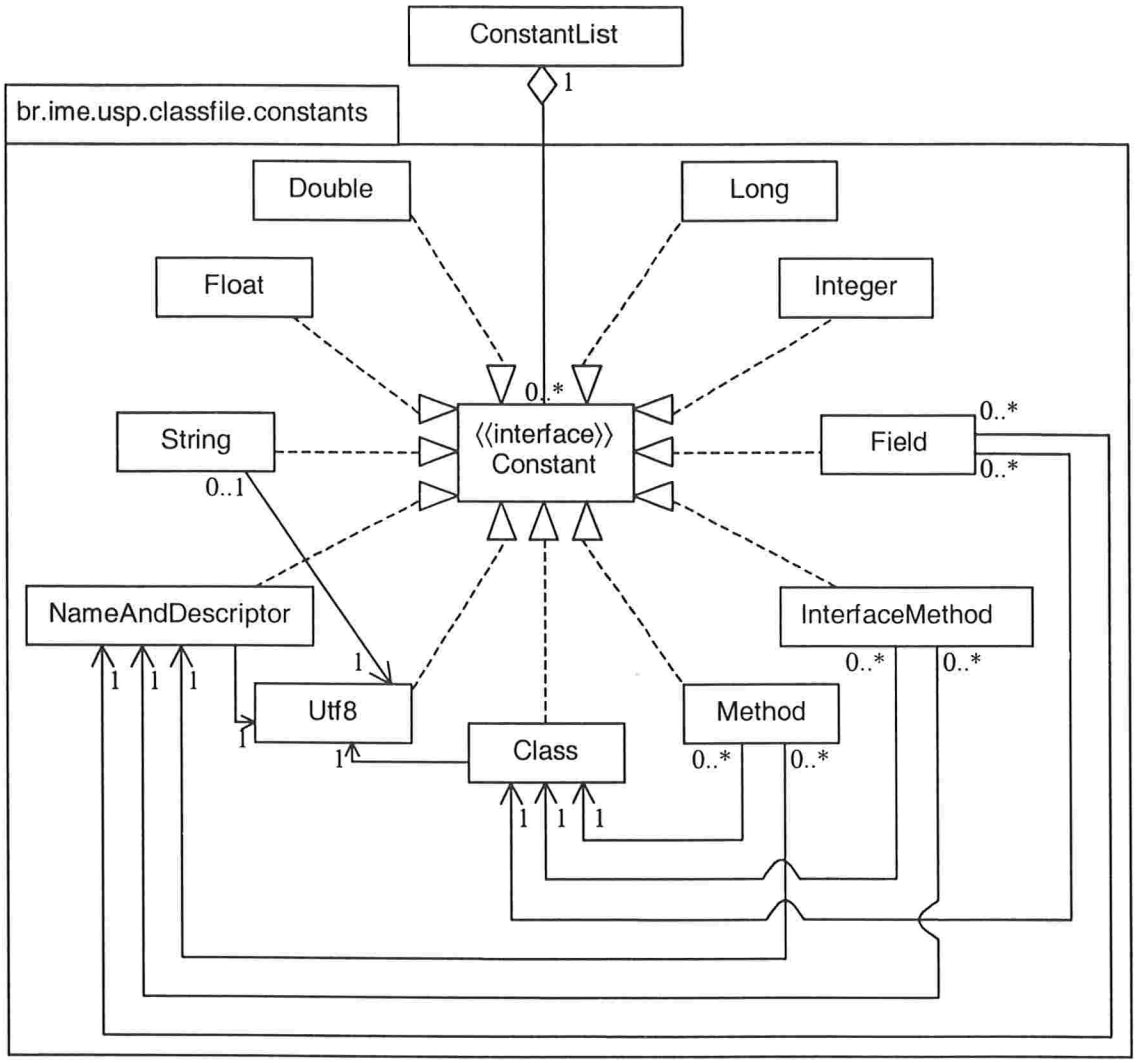


figura 12 – O pacote br.ime.usp.classfile.constants

A.2 A representação de uma classe

A classe ClassFile representa arquivos .class (veja a figura 13), e as classes ReaderImpl e WriterImpl fornecem a "ponte" entre um arquivo .class real e a sua representação como uma instância de um ClassFile. O pacote oferece ainda duas classes verificadoras: Ostrich, que não faz verificação nenhuma (simplesmente retorna verdadeiro sempre que questionada sobre a validade de um arquivo .class) e Standard, que verifica uma grande parte das restrições impostas sobre um arquivo .class [38].

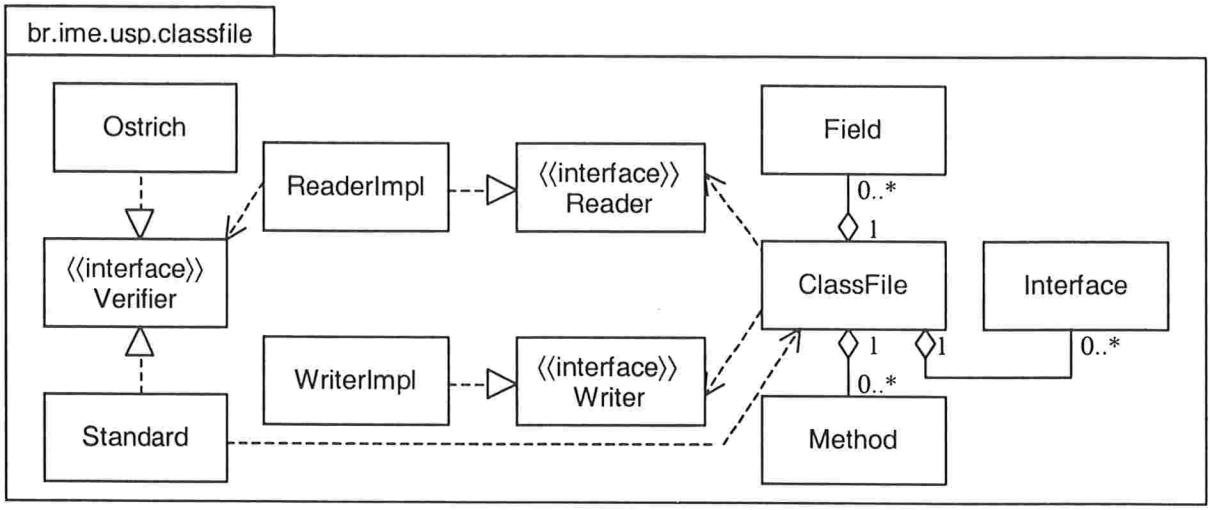


figura 13 – O pacote br.ime.usp.classfile

O seguinte trecho de código lê um arquivo .class e grava-o de volta, sem nenhuma modificação:

```
DataInputStream in = new DataInputStream(new FileInputStream(...));
DataOutputStream out = new DataOutputStream(new FileOutputStream(...));

ClassFile classFile = new ClassFile();   // cria um arquivo .class vazio

StandardVerifier verifier = new StandardVerifier(classFile); // cria um verificador

// cria um leitor utilizando o verificador acima e a tabela de
// constantes do arquivo .class que será preenchido com os dados lidos
ReaderImpl r = new ReaderImpl(in, verifier, classFile.getConstants());

try {
    classFile.read(r);
}
catch (InvalidClassFileException e) {
    System.out.println("Arquivo .class inválido: " + e);
}

classFile.write(new WriterImpl(out));
in.close(); out.close();
```

Já o exemplo seguinte cria e grava no disco uma classe pública e final chamada Foo, que estende java.lang.Object e implementa a interface java.lang.Cloneable, sem definir nenhum campo novo.

```
int i, j, k;
ClassFile classFile = new ClassFile();
ConstantList constants = classFile.getConstants();

// cria as constantes simbolizando a própria classe Foo...
i = constants.add(new ConstantClass(
    constants.add(new ConstantUtf8("Foo"))
));

// ...a classe java.lang.Object...
j = constants.add(new ConstantClass(
    constants.add(new ConstantUtf8("java/lang/Object"))
));

// ...e a interface java.lang.Cloneable
k = constants.add(new ConstantClass(
    constants.add(new ConstantUtf8("java/lang/Cloneable"))
));

classFile.setMagicNumber(0xCAFEBABE);
classFile.setVersionNumber(0, 45);
classFile.setMaskOfFlags(
    AccessFlags.PUBLIC | AccessFlags.SUPER | AccessFlags.FINAL
); // o bit SUPER deve sempre estar ligado (por questões de compatibilidade)
classFile.setThisClassIndex(i);
classFile.setSuperClassIndex(j);
classFile.getInterfaces().add(k);

// agora só precisa gravar a classe
DataOutputStream out = new DataOutputStream(
    new FileOutputStream("Foo.class")
);
classFile.write(new WriterImpl(out));
out.close();
```

O próximo exemplo cria uma variável de instância inteira e privada para cada campo inteiro já existente (de agora em diante, assuma que classFile é uma referência a um objeto ClassFile já lido):

```
int i = -1;
List list = new LinkedList();
ConstantList constants = classFile.getConstants();
FieldList fields = classFile.getFields();

// Acrescenta à lista ⟨list⟩ o nome de todas os campos inteiros
for(FieldList.Iterator it = fields.iterator(); it.hasNext(); ) {
    Field f = it.next();
    ConstantUtf8 descriptor =
        (ConstantUtf8)constants.get(f.getDescriptorIndex());
    if (descriptor.value.equals("I")) {
        i = f.getDescriptorIndex();
        list.add(constants.get(f.getNameIndex()));
    }
}

// Para cada um dos nomes na lista, cria uma nova variável de instância;
// o nome da nova variável será ⟨nome na lista⟩$overflow
for(Iterator it = list.iterator(); it.hasNext(); ) {
    String name = ((ConstantUtf8)it.next()).value;
    Field f = new Field();

    f.setNameIndex( constants.add(new ConstantUtf8(name+"$overflow")) );
    f.setDescriptorIndex(i);
    f.setMaskOfFlags(AccessFlags.PRIVATE);

    fields.add(f);
}
```

Por fim, o trecho seguinte acrescenta a uma classe um método público e abstrato boolean foo(int, char):

```
ConstantList constants = classFile.getConstants();
Method m = new Method();

m.setNameIndex(constants.add(new ConstantUtf8("foo")));
m.setDescriptorIndex( constants.add(new ConstantUtf8("(IC)Z")) );
m.setMaskOfFlags(AccessFlags.PUBLIC | AccessFlags.ABSTRACT);
classFile.getMethods().add(m);

classFile.setMaskOfFlags(
    AccessFlags.ABSTRACT | classFile.getMaskOfFlags()
); // como foi acrescentado um método abstrato, marca a classe como abstrata também
```

A.3 A representação de atributos

Atributos (seção 3.5) são representados pelas classes na figura 14.

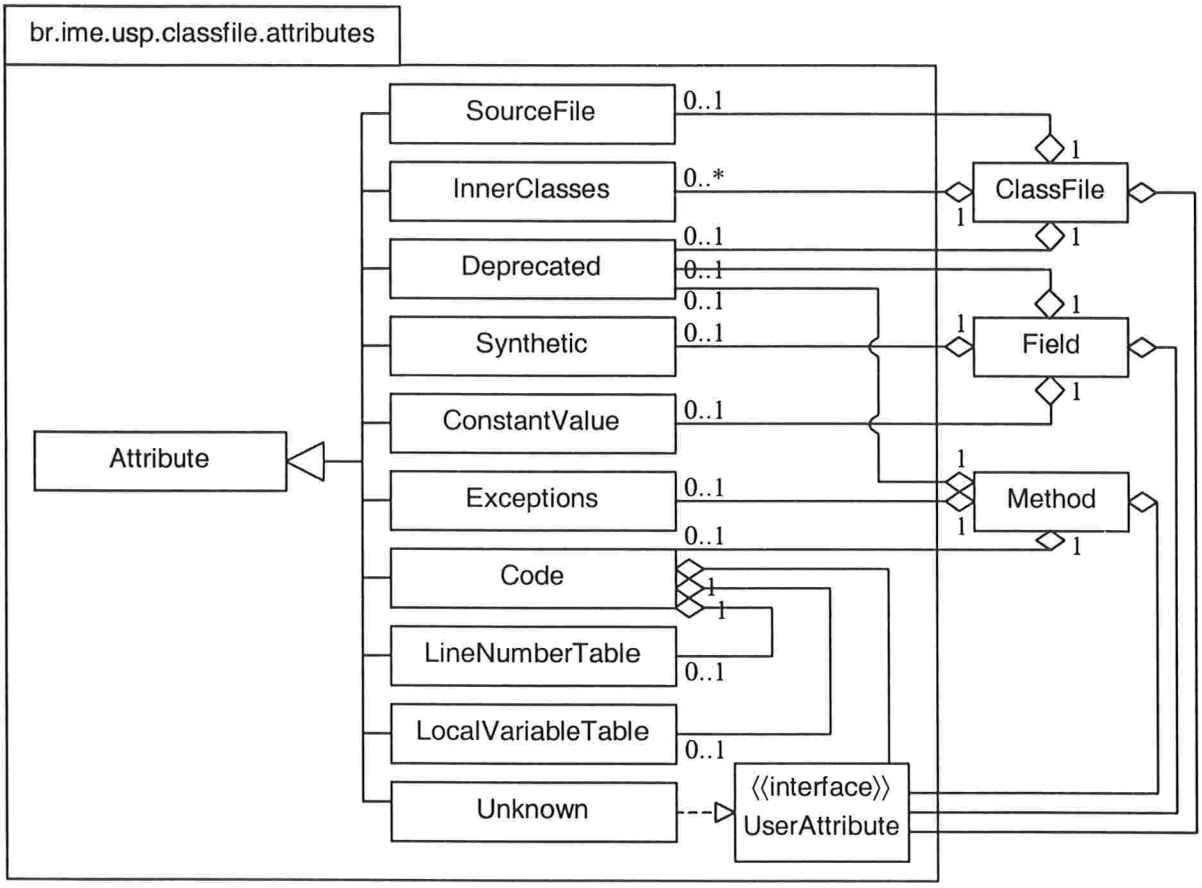


figura 14 – O pacote br.ime.usp.classfile.attributes

Alguns exemplos de utilização de cada um destes atributos encontram-se a seguir.

1. Marca como *deprecated* todos os campos públicos de uma classe:

```
ConstantList constants = classFile.getConstants();
FieldList fields = classFile.getFields();
int i = –1; // índice da palavra "Deprecated" na tabela de constantes

for(FieldList.Iterator it = fields.iterator(); it.hasNext(); ) {
    Field f = it.next();
    if ((f.getMaskOfFlags() & AccessFlags.PUBLIC) != 0) {
        if (i == –1) {
            i = constants.add(new ConstantUtf8(Deprecated.NAME));
        }
        f.getAttributes().add(new Deprecated(i));
    }
}
```

2. Acrescenta a uma classe uma variável estática, final e pública chamada DEBUG$, cujo valor inicial será true:

```
ConstantList constants = classFile.getConstants();
Synthetic sattr;
ConstantValue cattr;
Field f = new Field();

sattr = new Synthetic(
    constants.add(new ConstantUtf8(Synthetic.NAME))
);
cattr = new ConstantValue(
    constants.add(new ConstantUtf8(ConstantValue.NAME))
);
cattr.setValueIndex(constants.add(new ConstantInteger(1))); // 1 = true

f.setMaskOfFlags(
    AccessFlags.PUBLIC | AccessFlags.STATIC | AccessFlags.FINAL
);
f.setNameIndex(constants.add(new ConstantUtf8("DEBUG$")));
f.setDescriptorIndex(constants.add(new ConstantUtf8("Z"))); // Z = boolean
f.getAttributes().add(sattr); // marca o campo como sintético
f.getAttributes().add(cattr); // marca o campo como constante

classFile.getFields().add(f);
```

3. Altera o nome do arquivo fonte que originou o arquivo .class (caso exista um arquivo fonte):

```
SourceFile src = null;
AttributeList.Iterator it = classFile.getAttributes().iterator();

// procura o atributo "SourceFile"
while (it.hasNext() && src == null) {
    Attribute a = it.next();
    if (a instanceof SourceFile) {
        src = (SourceFile)a;
    }
}

// se achou, muda o nome do arquivo fonte para "Doe.java"
if (src != null) {
    ConstantList constants = classFile.getConstants();
    src.setSourceFileIndex(constants.add(new ConstantUtf8("Doe.java")));
}
```

4. Muda a assinatura de um método (o primeiro da lista de métodos), informando que ele pode lança uma exceção da classe ExcException:

```
ConstantList constants = classFile.getConstants();
Exceptions exc = null;
Method m = classFile.getMethods().get(0);
AttributeList attrList = m.getAttributes();

// primeiro verifica se o método já tem o atributo Exceptions
for(AttributeList.Iterator it = attrList.iterator(); it.hasNext() && exc == null; ) {
    Attribute a = it.next();
    if (a instanceof Exceptions)
        exc = (Exceptions)a;
}

// caso não tenha, cria um
if (exc == null) {
    exc = new Exceptions(
        constants.add(new ConstantUtf8(Exceptions.NAME))
    );
    m.getAttributes().add(exc);
}

// cria a constante simbolizando a classe "ExcException" e
// insere o seu índice no atributo Exceptions
int index = constants.add(new ConstantUtf8("ExcException"));
exc.getExceptions().add(constants.add(new ConstantClass(index)));
```

5. Acrescenta ao código de um método uma seqüência de instruções que incrementa uma variável estática antes de cada instrução que invoque um método. Assuma que code é o atributo code deste método, e index é o índice, na tabela de constantes, da constante simbolizando o campo que será incrementado. É importante ressaltar que, quando novas instruções são inseridas, excluídas ou substituídas por outras de tamanho diferente, o programador não precisa se preocupar em recalcular novos *offsets*, alterar o destino das instruções de desvio ou os trechos (*offsets* inicial e final) que podem lançar exceções — a classe Code já cuida isso. A tabela de número de linhas e a de variáveis locais (atributos LineNumberTable e LocalVariableTable) também são automaticamente atualizadas (uma ferramenta de depuração que utilize estas informações, por exemplo, não será afetada após o código ter sido alterado).

```
static void addCode(Code code, int index) {
    InstructionList instrs = code.getInstructions();
    for(InstructionList.Iterator it = instrs.iterator(); it.hasNext(); ) {
        int opcode = it.next().getOpcode();
        if (  opcode == Opcodes.INVOKESTATIC ||
              opcode == Opcodes.INVOKEVIRTUAL ||
              opcode == Opcodes.INVOKESPECIAL ||
              opcode == Opcodes.INVOKEINTERFACE ) {
              // acrescenta, antes da instrução atual, a seqüência de instruções
              // responsável pela adição. Todas as instruções de desvio que tenham como
              // destino a instrução atual terão o destino automaticamente alterado para a
              // primeira instrução da seqüência inserida
              it.addBefore(
                   new Instruction[] {
                        new DoubleOperandInstruction(Opcodes.GETSTATIC,
                                                     index/256, index%256),
                        new Instruction(Opcodes.ICONST_1),
                        new Instruction(Opcodes.IADD),
                        new DoubleOperandInstruction(Opcodes.PUTSTATIC,
                                                     index/256, index%256)
                   }
              );
              // GETSTATIC m, n   ; empilha o valor da variável estática simbolizada pela
              //                  ; constante cujo índice é m * 256 + n
              // ICONST_1         ; empilha o valor 1
              // IADD             ; desempilha os dois valores no topo da pilha, soma-os
              //                  ; e empilha o resultado
              // PUTSTATIC m, n   ; desempilha o valor no topo da pilha e armazena-o na
              //                  ; variável estática cujo índice é m * 256 + n
        }
    }
    // como as instruções inseridas empilham dois valores
    // (o valor do campo estático e a constante 1), aumenta em duas
    // unidades o tamanho máximo da pilha de operandos
    code.setMaxStack(code.getMaxStack()+2);
}
```

6. Transforma a classe "InnFoo" em uma classe aninhada da classe "OutFoo":

```
int index = classFile.getThisClassIndex(); // ⟨classFile⟩ deve ser a classe "OutFoo"
ConstantList constants = classFile.getConstants();

InnerClasses.Record record = new InnerClasses.Record(
    constants.add(new ConstantClass(
        constants.add(new ConstantUtf8("OutFoo$InnFoo"))
    )),
    index,
    constants.add(new ConstantUtf8("InnFoo")),
    AccessFlags.STATIC
);

// procura o atributo "InnerClasses"
InnerClasses inc = null;
AttributeList.Iterator it;

for(it = classFile.getAttributes().iterator(); it.hasNext() && inc == null; ) {
    Attribute atributo = it.next();
    if (atributo instanceof InnerClasses) {
        inc = (InnerClasses)atributo;
    }
}

// caso não exista, cria um
if (inc == null) {
    inc = new InnerClasses(
        constants.add(new ConstantUtf8(InnerClasses.NAME))
    );
    classFile.getAttributes().add(inc);
}

inc.getNestedClasses().add(record);
```

7. Imprime, caso existam, tabelas de variáveis locais e de números de linhas do método m.

```
AttributeList atList = m.getAttributes();
for(AttributeList.Iterator it = atList.iterator(); it.hasNext(); ) {
        Attribute a = it.next();
        if (a instanceof Code) print((Code)a, classfile.getConstants());
    }
}
...
static void print(Code code, ConstantList cl) {
    AttributeList.Iterator it = code.getAttributes().iterator();
    while (it.hasNext()) {
        Attribute a = it.next();
        if (a instanceof LineNumberTable) {
            LineNumberTable t = ((LineNumberTable)a).getLineNumbers();
            for(int i = 0; i < t.size(); i++) {
                LineNumberTable.Record r = t.get(i);
                // cada registro ⟨r⟩ da tabela de número de linhas tem duas informações:
                // ⟨r⟩.getLineNumber() retorna o número de linha no código fonte
                // ⟨r⟩.getInstruction() retorna a primeira instrução gerada pelo compilador
                // para essa linha
                System.out.println( r.getLineNumber() + " ⇒ " +
                                    r.getInstruction());
            }
        }
        else if (a instanceof LocalVariableTable) {
            LocalVariableTable.Table t =
                ((LocalVariableTable)a).getLocalVariables();
            for(int i = 0; i < t.size(); i++) {
                LocalVariableTable.Record r = t.get(i);
                // cada registro ⟨r⟩ da tabela de variáveis locais tem cinco informações:
                // ⟨r⟩.first contém a primeira instrução em que a variável local está ativa
            // ⟨r⟩.last contém a última instrução em que a variável local está ativa
                //          (ou null, se a variável estará ativa até o fim do código)
                // ⟨r⟩.nameIndex: índice do nome da variável, na tabela de constantes
                // ⟨r⟩.descriptorIndex: índice do descritor, na tabela de constantes
                // ⟨r⟩.index: índice na tabela de variáveis locais
                String name = ((ConstantUtf8)cl.get(r.nameIndex)).value;
                String type = ((ConstantUtf8)cl.get(r.descriptorIndex)).value;
                int start = r.first.getOffset();
                int end = r.last != null ?  r.last.getOffset() :
                                            code.getInstructions().sizeInBytes();
                System.out.println("var(" + r.index + "): " + type + " " + name
                  + ", [" + start + ".." + end + "]");
            }
        }
    }
}
```

8. Define um novo atributo, chamado "NovoAtributo", e acrescenta-o a uma classe:

```
static class NovoAtributo extends Unknown {
    static final String NOME = "GeneticamenteModificado";
    static final byte[] INFO = new byte[0];
    public NovoAtributo(int index) { super(index, INFO, NOME); }
}

/* Alternativamente, o atributo poderia ser definido como:
static class NovoAtributo extends Attribute implements UserAttribute {
    static final String NOME = "GeneticamenteModificado";
    static final int TAMANHO = 0;
    public NovoAtributo(int index)    { super(index, TAMANHO); }
    public int getCurrentSize()       { return TAMANHO; }
    public String getName()           { return NOME; }
}
*/
...
int i = classFile.getConstants().add(new ConstantUtf8(NovoAtributo.NOME));
classFile.getAttributes().add(new NovoAtributo(i));
```

Apêndice B

Detalhes da implementação de identificadores

B.1 Blocos e tags

Os eventos start, join, getCoordinationPoint e coordinate podem ser vistos como divisores de uma *thread*. Por exemplo, se a *thread* T1 invoca o método start, iniciando a *thread* T2, então T1 fica dividida em dois trechos: um deles anterior e o outro posterior à invocação. Todos os acessos feitos por T1 a variáveis compartilhadas no trecho anterior à invocação precedem os acessos feitos por T2 (e por quaisquer *threads* que T2 eventualmente criar). Essas considerações levam à definição de um **bloco**:

Um bloco é uma região da *thread* em que ao menos um destes eventos ocorre:

- o fim da própria *thread*;
- a *thread* inicia uma outra *thread* (start);
- a *thread* aguarda o término de outra (join);
- a *thread* cria um ponto de coordenação (StaticLadybug.getCoordinationPoint) ou coordena-se com outra *thread* (StaticLadybug.coordinate).

Além disso, em um mesmo bloco, nenhum acesso monitorado a variáveis de instância, estáticas ou elementos de vetores vem **após** um dos eventos acima.

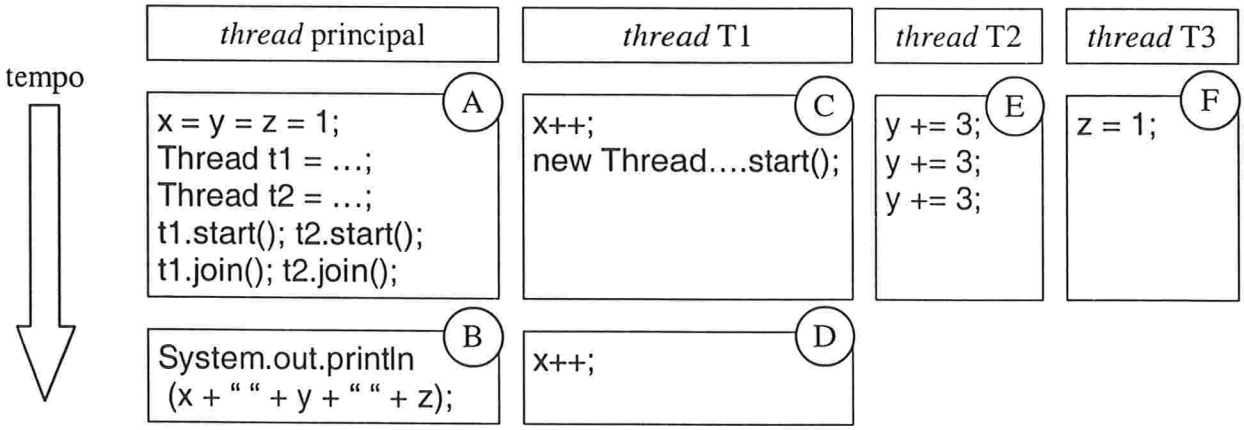
O exemplo a seguir mostra uma possível divisão em blocos das *threads* de um programa (os blocos serão chamados A, B, C, D, E e F):

```
class Exemplo {
    static int x, y, z;

    public static void main(String[] args) throws InterruptedException {
        x = y = z = 1;
        Thread t1 = new Thread() {
            public void run() { m1(); }
        };
        Thread t2 = new Thread() {
            public void run() { m2(); }
        };
        t1.start(); t2.start(); // inicia threads T1 e T2
        t1.join(); t2.join();
        System.out.println(x+" "+y+" "+z);
    }

    static void m1() {
        x++;
        new Thread() {
            public void run() { z = 1; }
        }.start(); // inicia thread T3
        x++;
    }

    static void m2() {
        while (y < 10) y += 3;
    }
}
```



Também será definida uma relação de ordem parcial entre os blocos: diremos que o bloco X precede o bloco Y se, e somente se, X e Y são blocos distintos e ao menos uma destas condições for verdadeira:

1. X e Y são blocos da mesma *thread*, e a execução do bloco X precede no tempo a execução do bloco Y;
2. o bloco Y pertence à *thread* T, e o bloco X contém a invocação do método start que iniciou a *thread* T;
3. o bloco X pertence à *thread* T, existe um bloco Z que contém a invocação do método join que esperou o término da *thread* T, e o bloco Z precede o bloco Y de acordo com a condição 1;
4. o bloco X contém uma invocação ao método getCoordinationPoint, existe um bloco Z que contém a instrução coordinate que usa como argumento o valor retornado por um dos métodos getCoordinationPoint do bloco X, e Z precede o bloco Y de acordo com a condição 1;
5. existe uma seqüência de blocos B_1, B_2, ..., B_n, $n \geq 1$, tal que, por alguma das condições anteriores, X precede B_1, B_i precede B_{i+1} para $1 \leq i < n$, e B_n precede Y.

No exemplo anterior, o bloco C precede os blocos B (regra 3), D (regra 1) e F (regra 2); já o bloco A precede os blocos B (regra 1), C, D, E (regra 2) e F (regra 5: A precede C e C precede F).

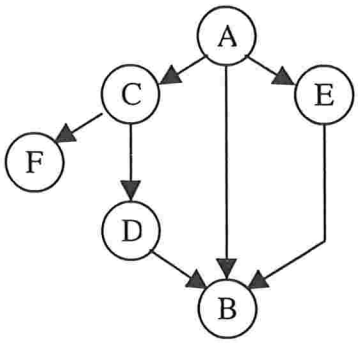


figura 15 – Precedência com a nova definição de blocos

Com as definições de bloco e precedência dadas, é possível construir um grafo similar ao POEG (seção 2.2) onde cada bloco é representado por um vértice (veja a figura 15). Diferente do POEG original, porém, as definições acima **não** são fortes o suficiente para impedir que, dados dois blocos X e Y tais que X preceda Y, existam dois eventos quaisquer, um no bloco X e outro no bloco Y, que sejam executados concorrentemente (no exemplo anterior, as três últimas instruções do bloco A poderiam ser executadas concorrentemente com quaisquer instruções dos blocos C, D, E e F, embora o bloco A preceda todos eles). No entanto, quando ambos eventos referem-se a acessos a variáveis compartilhadas, podemos com certeza dizer que eles **não** são executados concorrentemente.

O conceito de bloco é representado em Ladybug por *tags*. A cada *thread* está associado um objeto *Tag*, cujo valor num determinado instante **identifica** o bloco sendo executado (quando o bloco muda, o valor da *tag* deve mudar também). Além disso, sempre que uma variável compartilhada é acessada, uma **cópia parcial** da *tag* da *thread* que fez o acesso é acrescentada ao histórico de acessos da variável, e essa cópia não é alterada jamais.

Denotando por bloco(t, n) o bloco identificado pelo valor da *tag* t no instante n, diremos que a *tag* t1 precede a *tag* t2 no instante n se, e somente se, bloco(t1, n) precede bloco(t2, n).

Uma *tag* contém os seguintes métodos:

- ordered(Tag outraTag)

  Verdadeiro se, no instante da invocação, a *tag* precede outraTag ou ambas têm o mesmo valor (outraTag deve ser a **cópia parcial** (definida abaixo) de uma *tag* representando o bloco em que foi feito um acesso a uma variável). Note que, embora o método retorne verdadeiro para duas *tags* com o mesmo valor, uma *tag* não pode preceder a si mesma. Isso é apenas uma conveniência: dados dois acessos à memória, ambos realizados no mesmo bloco, então um deles com certeza precede o outro.

- clone()

  Retorna uma cópia da *tag* (a mesma cópia pode ser retornada diversas vezes). O valor da cópia não deve mudar caso o valor da *tag* original seja alterado.

- copy()

  Retorna uma cópia "parcial" da *tag*, para ser armazenada no histórico de uma variável (a mesma cópia pode ser retornada diversas vezes). O valor da cópia não deve mudar caso o valor da *tag* seja alterado. A cópia é dita parcial porque a única condição dela exigida é:

  clone().ordered(t) $\Leftrightarrow$ copy().ordered(t) para qualquer *tag* t.

  Não é requerido que t.ordered(clone()) $\Rightarrow$ t.ordered(copy()). Em outras palavras, uma cópia parcial pode ter seu método ordered invocado, mas não deve nunca ser usada como argumento do método ordered invocado para outra *tag*.

  Essa forma de copiar uma *tag* existe apenas por razões de eficiência; no histórico de uma variável, as informações presentes em uma cópia parcial são suficientes.

A existência da cópia parcial traz outro problema: se uma *tag* é uma cópia parcial, seu valor pode não identificar um bloco, já que, após a cópia, dados sobre esse valor podem ter sido perdidos. Isso torna a definição de precedência entre *tags* dada acima imprecisa; para corrigi-la, é necessário acrescentar que, caso t seja a cópia parcial de uma *tag* u, então bloco(t, n) denota o bloco identificado pelo valor de u no momento da cópia.

- endBlock([Tag outraTag])

  Informa que o fim de um bloco foi atingido, isto é, nenhum outro acesso a variáveis compartilhadas deve estar presente no bloco representado pelo valor atual da *tag*. Note que o valor da *tag* pode não mudar imediatamente, mas a próxima *tag* adicionada ao histórico de uma variável (retornada pelo método copy) com certeza terá um valor diferente. Se outraTag (cujo valor não deve mudar no decorrer do tempo) também for fornecida, então ordered(outraTag) deverá retornar verdadeiro a partir do próximo bloco.

  endBlock sem argumentos é utilizado por Dinning-Schonberg quando uma *thread* é iniciada (threadStart) ou um ponto de coordenação é criado (getCoordPoint). Com argumentos, endBlock é utilizado quando uma *thread* espera o término de outra (joinThread) ou quando coordena-se com outra (coordinate). No primeiro caso, o argumento é a *tag* associada à *thread* cujo término foi esperado, e no segundo é um valor retornado por getCoordPoint.

- createChildTag()

  Retorna uma nova *tag* t tal que ordered(t) é verdadeiro. Além disso, dadas duas *tags* t1 e t2 retornadas por createChildTag da mesma *tag*, tanto t1.ordered(t2) quanto t2.ordered(t1) devem ser falsos. Este método é utilizado para criar *tags* para as *threads* cujo início (start) é monitorado.

Mesmo que uma *thread* **não** tenha seu início monitorado, ela também terá uma *tag* associada. Quando, pela primeira vez, essa *thread* acessar um campo, adquirir um *lock* ou executar qualquer outro evento monitorado, uma *tag* t será criada e associada à *thread*, de forma que u.ordered(t) retorne falso para qualquer *tag* u, durante toda a execução do programa.

B.2 A implementação mais simples da interface Tag

Nesta implementação (utilizada na classe DinningSchonberg), o valor de uma *tag* associada a uma *thread* T é um par (b, l), onde b é um vetor de bytes de tamanho fixo e l é uma lista de *tags* cujos valores não mudam no decorrer do tempo (o tamanho da lista é variável, eventualmente zero). O vetor b contém informações sobre as *threads* ancestrais de T; a lista l contém as *tags* associadas às *threads* cujo término T esperou e/ou clones das *tags* associadas às *threads* com as quais T se coordenou.

Dada uma *tag* (b, l):

- o método createChildTag() cria uma nova *tag* cujo valor inicial é (b', l'), onde b' é a concatenação de uma cópia de b com o byte de menor valor possível em Java, e l' é uma cópia da lista l (portanto, se b for alterado ou se novos elementos forem adicionados a l, isso não afetará b' nem l');
- o método endBlock() incrementa o último byte de b em 1;
- o método endBlock(Tag t) acrescenta a *tag* t à lista l;
- o método clone cria uma cópia (b', l'), onde b' é uma cópia de b e l' é uma cópia de l;
- o método copy cria uma cópia parcial (b', l'), onde b' é uma cópia de b e l' é null;
- dadas duas *tags* $t1 = (b, l)$ e $t2 = (c, p)$, onde $b = b_1 b_2 \ldots b_m$[24], $c = c_1 c_2 \ldots c_n$ e $p = p_1, p_2, \ldots, p_k$, t1.ordered(t2) retorna verdadeiro se, e somente se:

  $m \leq n$, $b_i = c_i$ para $i = 1, 2, \ldots, m - 1$ e $b_m \leq c_m$ (isto é, $t1$ precede $t2$ ou $t1 = t2$), **ou**
  existe i, $1 \leq i \leq k$, tal que t1.ordered(p_i) é verdadeiro

  (note que l nunca é usado).

B.3 A implementação mais eficiente da interface Tag

Nesta implementação (utilizada na classe DinningSchonbergFast) o valor de uma *tag* é uma terna $(t, u, v_{(m)})$, onde t é um inteiro cujo valor é fixo, u é um byte de valor variável, e $v_{(m)}$, é um vetor de bytes de tamanho m. Embora esta implementação possa ocupar mais espaço que a anterior (o tamanho do vetor $v_{(m)}$ é limitado pelo número de *threads* criadas até o momento,

[24] Os vetores aqui aparecem indexados de 1 a n, e não de 0 a n – 1.

como veremos) ele decide em tempo constante se dois acessos estão ordenados (a implementação anterior depende do tamanho da *tag*).

Dadas duas *tags* $g1 = (t_1, u_1, v_{(m)})$ e $g2 = (t_2, u_2, s_{(n)})$, g1.ordered(g2) retorna verdadeiro se, e somente se, $t_1 \leq n$ e $u_2 \leq s[t_1]$ (note que $v_{(m)}$ nunca é usado).

Dada uma *tag* $(t, u, v_{(m)})$:

- o método createChildTag() cria uma nova *tag* $(t', u', v'_{(k)})$, onde t' é um valor seqüencial incrementado a cada nova *tag* criada, u' é o byte de menor valor possível em Java mais 1, $k = t'$ e $v'_{(k)}$ é construído da seguinte maneira:

  $v[i]' \leftarrow v[i]$ para $1 \leq i \leq m$

  $v[i]' \leftarrow \langle$menor valor possível de um byte$\rangle$ para $m < i < t'$

  $v[i]' \leftarrow u'$ para $i = t'$

  (existe uma invariante neste algoritmo: $v[t'] = u'$)

- o método clone cria uma cópia (t', u', v'), onde $t' = t$, $u' = u$, e v' é uma cópia de $v_{(m)}$;

- o método copy cria uma cópia (t', u', v'), onde $t' = t$, $u' = u$, e v' é null;

- o método endBlock() incrementa u e $v[t']$ em 1;

- o método endBlock(Tag tag), onde tag = $(t', u', v'_{(k)})$, executa uma destas ações:

  $\Rightarrow$ se $m \geq k$, faz $v[i] \leftarrow$ máx$\{v[i], v'[i]\}$ para $1 \leq i \leq k$

  $\Rightarrow$ se $m < k$, cria um vetor $w_{(k)}$ tal que:

  $w[i] \leftarrow$ máx$\{v[i], v'[i]\}$ para $1 \leq i \leq m$

  $w[i] \leftarrow v'[i]$ para $m < i \leq k$

  A seguir, faz $v_{(m)} \leftarrow w_{(k)}$.

Bibliografia

[1] Andrew S. Tanembaum. Modern Operating Systems. Prentice-Hall, 1992.

[2] Anne Dinning e Edith Schonberg. Detecting Anomalies in Programs with Critical Sections. Conference Proceedings on ACM/ONR Workshop on Parallel and Distributed Debugging, 1991, páginas 85-96.

[3] Anne Dinning e Edith Schonberg. An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, 1990, páginas 1-10.

[4] Benjamin M. Brosgol. A Comparison of the Concurrency Features of Ada 95 and Java. Proceedings of the ACM SIGAda Annual International Conference on Ada Technology, 1998, página 175.

[5] Brinch Hansen, The Programming Language Concurrent Pascal, IEEE Transactions of Sofware Engineering, 1975, páginas 199-207.

[6] Bug Parade. http://developer.java.sun.com/developer/bugParade/index.jshtml.

[7] C. A. R. Hoare. Monitors, an Operating System Structuring Concept. Communications of the ACM, vol. 17, páginas 549 – 597, 1974.

[8] Claudio Demartini, Riccardo Sisto, Radu Iosif. A Concurrency Analysis Tool for Java Programs. http://www.dai.arc.polito.it/dai-arc/manual/tools/jcat/main.ps.

[9] Claudio Demartini, Radu Iosif, Riccardo Sisto. Modeling and Validation of Java Multithreading Applications Using SPIN.
http://netlib.bell-labs.com/netlib/spin/ws98/p23ps.gz (SPIN98 Workshop site).

[10] Cormac Flanagan e Stephen N. Freund. Type-Based Race Detection for Java. Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation. páginas. 219-232.

[11] David F. Bacon, Ravi Konuru, Chet Murthy, Mauricio Serrano. Thin Locks: Featherweight Synchronization for Java. Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation. páginas 258-268.

[12] David F. Bacon, Robert E. Strom e Ashis Tarafdar. Guava: A Dialect of Java without Data Races. A ser apresentado, Conference on Object-oriented Programming, Systems, Languages, and Applications, 2000.
http://www.research.ibm.com/people/d/dbf/papers.html

[13] Doug Lea. Concurrent Programming in Java, Second Edition. Addison-Wesley, 1999.

[14] Doug Lea. Concurrent Programming in Java – Online Supplement.
http://gee.cs.oswego.edu/dl/cpj/index.html.

[15] Gleb Naumovich e George S. Avrunin, Lori A. Clarke. Data Flow Analysis for Checking Properties of Concurrent Java Programs. Proceedings of the 1999 International Conference on Software Engineering, 1999, páginas 399 - 410.

[16] Gleb Naumovich e George S. Avrunin, Lori A. Clarke. An Efficient Algorithm for computing MHP Information for Concurrent Java Programs.
http://icee.cs.umass.edu/abstracts/98-044.html.

[17] Grady Booch, James Rumbaugh e Ivar Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, 1999.

[18] Gregory Andrews. Concurrent Programming and Practice. Benjamin/Cummings Pub. Co., 1991.

[19] Han Bok Lee e Benjamin G. Zorn. BIT: a tool for instrumenting Java bytecodes. http://www.cs.colorado.edu/~hanlee/BIT/index.html.

[20] Il-Hyung Cho. Separate Inheritance Hierarchy to Solve Inheritance Anomaly Problem Under Java Programming Language Platform. http://www.cs.clemson.edu/~ihcho/java-paper/coots.html
[21] James C. Corbett. Constructing Compact Models of Concurrent Java Programs. Proceeding of ACM SIGSOFT International Symposium on Software Testing and Analysis, 1998, páginas 1-10.
[22] James Gosling, Bill Joy e Guy Steele. The Java Language Specification. Addison-Wesley, 1996.
[23] James Gosling, Bill Joy e Guy Steele. The Java Language Specification, Second Edition. http://java.sun.com/docs/books/jls, 2000.
[24] Jeff Bogda e Urs Hölze. Removing Unnecessary Synchronization in Java. Proceedings of the 1999 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, 1999, páginas 35-46.
[25] Jigsaw. http://www.w3.org/Jigsaw
[26] John Meyer e Troy Downing. Java Virtual Machine. O'Reilly, 1997.
[27] John M. Jeffrey. Using Petri Nets to Introduce Operating System Concepts. Papers of the Twenty-second SIGCSE Technical Symposium on Computer Science Education, 1991, páginas 324 – 329
[28] Jong-Deok Choi e Harini Srinivasan. Deterministic Replay of Java Multithreaded Applications. Proceedings of the SIGMETRICS Symposium on Parallel and Distributed tools, 1998, páginas 48-59.
[29] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Gugranam C. Sreedhar e Sam Midkiff. Escape Analysis for Java. Proceedings of the 1999 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, 1999, páginas 1-19.
[30] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, vol. 21, 1978, páginas 558-564.
[31] Mauro Pezzè, Richard N. Taylor e Michal Young. Graph Models for Reachability Analysis of Concurrent Programs. ACM Transactions on Software Engineering and Methodology, vol. 4, nº 2, abril 1995, páginas 171-213.
[32] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna e Derek White. An Efficient Meta-lock for Implementing Ubiquitous Synchronization. Proceedings of the 1999 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, 1999, páginas 207-222.
[33] Tamiya Onodera e Kiyokuni Kawachiya. A Study of Locking Objects with Bimodal Fields. Proceedings of the 1999 ACM SIGPLAN Conference on Object-oriented Programming, systems, Languages, and Applications, 1999, páginas 223-237.
[34] Rajeshwari SusaiMicheal. Inheritance anomaly in OOCP Languages. http://www.engr.csufresno.edu/Personal/CSci/Students/Grad/Rajeshwari_SusaiMicheal/ia.html.
[35] Ronald Tschalär. HTTPClient V0.3-2. http://www.innovation.ch/java/HTTPClient/
[36] Stefan Savage, Michel Burrows, Greg Nelson, Patrick Sobalvarro e Thomas Anderson. Eraser: a Dynamic Data Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems, vol. 15, nº 4, novembro de 1997, páginas 391-411.
[37] Szabolcs Ferenczi. Guarded Methods vs. Inheritance Anomaly. http://www.kfki.hu/~ferenczi/InhAnom.html.
[38] Tim Lindholm e Frank Yellin. The Java™ Virtual Machine Specification, Second Edition. Addison-Wesley, 1999.

[39] William Pugh. Fixing the Java Memory Model. Proceedings of the ACM 1999 Conference on Java Grande, 1999, páginas 89-98.