Daniel de Carvalho Bittencourt Andrade

Uma técnica para verificação de modelos a partir da execução do código de sistemas concorrentes

Feira de Santana - BA, Brasil

02 de setembro de 2008

Daniel de Carvalho Bittencourt Andrade

Uma técnica para verificação de modelos a partir da execução do código de sistemas concorrentes

Monografia apresentada para obtenção do Grau de Engenheiro de Computação pela Universidade Estadual de Feira de Santana.

Orientador:
José Amancio Macedo Santos

Universidade Estadual de Feira de Santana

Feira de Santana - BA, Brasil

02 de setembro de 2008

Monografia de Projeto Final de Graduação sob o título *"Uma técnica para verificação de modelos a partir da execução do código de sistemas concorrentes"*, defendida por Daniel de Carvalho Bittencourt Andrade e aprovada em 02 de setembro de 2008, em Feira de Santana, Estado da Bahia, pela banca examinadora constituída pelos professores:

Prof. Msc. José Amancio Macedo Santos
Orientador

Prof. Msc. Daniel Gouveia Costa
Universidade Estadual de Feira de Santana

Prof. Dr. Antonio Lopes Apolinário Jr.
Universidade Estadual de Feira de Santana

Resumo

Com a crescente complexidade dos sistemas de software, a construção de modelos se faz cada vez mais necessária em algum momento do processo de desenvolvimento. Quando se trata de sistemas concorrentes, a necessidade da utilização de modelos se torna mais evidente, já que a natureza não determinística de tais sistemas aumenta ainda mais a sua complexidade. Entretanto, o custo para a manutenção desses artefatos é bastante alto, o que faz com que, algumas vezes, a modelagem seja posta em segundo plano, como é o caso dos processos ágeis de desenvolvimento. Esse trabalho apresenta uma técnica que permite ao desenvolvedor verificar a conformidade entre o código de um sistema concorrente e seu modelo, proporcionando uma maior facilidade na manutenção dos modelos e conseqüentemente na análise do comportamento de tais sistemas. Dessa forma, os processos ágeis de desenvolvimento ganham mais um recurso para manter a qualidade dos softwares, já que com a técnica é possível diminuir o custo para a manutenção de modelos.

Agradecimentos

Primeiramente gostaria de agradecer à Vida por todas as oportunidades - aproveitadas e perdidas - sobretudo por ter me dado a oportunidade de fazer este curso, que vem a ser minha maior conquista até o momento. Entretanto, essa não é uma conquista individual. Por isso, agradeço a meus pais! Mãe, Pai; mesmo que eu tivesse o dom da palavra, não seria capaz de agradecer por tudo que vocês são e representam pra mim. A vocês eu devo tudo que sou e tudo que ainda serei. Vocês me deram o que há de melhor em mim e para isso não existem palavras que possam agradecer. Amo vocês! Agradeço também a minhas irmãs; Priscilla e Anna Clara, pelo companheirismo e cumplicidade. Vocês são as melhores irmãs que alguém pode ter. Agradeço a minha namorada, Núbia, por ser tão compreensiva e pelo jeito só dela de me fazer esquecer todos os problemas. Amor, sem você teria sido muito mais difícil.

Gostaria de fazer um agradecimento especial ao professor João Batista por ter acreditado e apostado em mim, num momento em que nem mesmo eu acreditava. João, muito obrigado! Gostaria de agradecer também ao professor José Amancio, meu orientador, pelo incentivo e orientação e por ter continuado nos momentos em que eu poderia ter me empenhado mais e não o fiz. Agradeço a TODOS os meus colegas-amigos de curso, sem os quais eu tenho a plena consciência de que não conseguiria chegar até aqui.

Por fim, agradeço a todos os meus familiares - avós, tios, primos - e amigos por fazerem parte da minha vida e sempre acreditarem que eu seria capaz. Vocês contribuíram bastante nesta minha caminhada.

Muito obrigado a todos e espero que um dia consiga retribuir à altura tudo o que me foi oferecido.

Sumário

Lista de Figuras

Lista de Códigos

1 Introdução

A construção de modelos de software vem sendo cada vez mais aplicada nos processos de desenvolvimento, devido à crescente complexidade dos sistemas [3]. Os sistemas possuem cada vez mais funcionalidades e o acompanhamento de sua evolução durante o processo de desenvolvimento se torna impraticável se esse acompanhamento for feito apenas diretamente pelo código. A modelagem é uma especificação que possibilita a abstração de um determinado sistema com o intuito de simplificar o seu entendimento como um todo [4]. Ao se modelar um sistema, abstraindo-se partes complexas - ou que não são tão importantes num determinado momento do processo de desenvolvimento - melhora-se a comunicação e o entendimento do sistema entre os participantes do projeto e conseqüentemente torna-se mais fácil sua implementação.

Quando o sistema em questão é um sistema concorrente, a necessidade de se construir modelos se torna mais evidente. Softwares concorrentes possuem várias linhas (fluxos) de execução que podem se intercalar, produzindo um comportamento não determinístico no sistema, pois a concorrência entre as linhas de execução pode gerar resultados diferentes ao seu final [5], o que torna ainda maior a dificuldade para se entendê-lo.

Um problema que decorre da construção de modelos é o custo envolvido para a manutenção de mais um artefato. Manter o modelo em conformidade com o código é difícil e custoso, pois a velocidade com que o código evolui é muito alta e é necessário despender recursos para que o modelo acompanhe essa evolução. Isso faz com que em muitos casos o modelo deixe de ser um artefato útil no decorrer do processo de desenvolvimento, já que tal modelo passa a não descrever mais o sistema. De qualquer forma, as exigências de alguns processos de desenvolvimento apontam para uma atenção especial na garantia destes artefatos. E essa garantia pode ser alcançada através do uso de técnicas de verificação.

As técnicas de verificação são utilizadas para se descobrir falhas que não foram encontradas pelo desenvolvedor, evitando que elas cheguem ao usuário final [6]. Como os prejuízos causados pelos defeitos não detectados são muito maiores que o investimento

necessário para se detectar defeitos durante as fases iniciais do projeto [7], o emprego de técnicas de verificação é compensado pela economia.

O objetivo desse trabalho é apresentar uma técnica que permita simplificar o processo de verificação de sistemas concorrentes e minimizar as diferenças existentes entre código e modelo, diferenças essas que surgem durante o próprio desenvolvimento. Com a utilização desta técnica, é permitido ao desenvolvedor escrever comandos, a partir do código, para disparar eventos em um modelo comportamental previamente construído. Assim, será possível confrontar o código do sistema com seu modelo para que o seu comportamento seja verificado, o que é bastante interessante para se utilizar em processos ágeis de desenvolvimento, que passam a contar com mais um recurso para melhorar a qualidade do software a um custo baixo.

1.1 Estrutura da Monografia

O restante deste trabalho está estruturado da seguinte forma:

Capítulo 2 - Verificação, Linguagens Formais e AOP - Neste capítulo será visto um pouco da teoria que fundamenta este trabalho. Serão discutidos tópicos sobre verificação e sobre as linguagens formais estudadas para se modelar sistemas concorrentes. Será apresentada também uma visão geral sobre a programação orientada a aspectos, que é utilizada na captura de eventos ocorridos no código.

Capítulo 3 - O Uso da Técnica em Problemas Clássicos de Concorrência - Aqui serão apresentados alguns problemas clássicos de concorrência, bem como suas implementações e o uso da técnica em cada um deles. Serão discutidos também os problemas que foram encontrados durante o desenvolvimento deste trabalho.

Capítulo 4 - Estudo de Caso - Neste capítulo é discutido um problema real, que já havia sido implementado e é aplicada a técnica.

Capítulo 5 - Considerações Finais - Aqui são discutidas as vantagens do uso da técnica e as possibilidades de desenvolvimento futuro, tomando-se como base este trabalho.

2 *Verificação, Linguagens Formais e AOP*

2.1 Verificação de Sistemas

Verificação é, em linhas gerais, avaliar se o software está sendo desenvolvido de forma adequada, de acordo com alguma especificação (que pode ser um modelo) [6]. É através dela que são obtidas evidências sobre a corretude do sistema. Embora as técnicas de verificação sejam bastante úteis, em muitos casos elas são deixadas de lado e perde-se seus benefícios, em função dos custos de se manter modelos atualizados.

As técnicas de verificação podem ser dinâmicas ou estáticas [6]. Na verificação dinâmica, a execução do código do programa é parte do processo de verificação, já na verificação estática, a verificação é feita sem que o código precise ser executado.

Os testes são exemplos de verificação dinâmica muito utilizados. Um dos tipos de testes mais utilizados são os testes de unidade, já que são bastante simples com relação a outros tipos. O objetivo dos testes de unidade é dividir o sistema em unidades pequenas para que possam ser testadas isoladamente, diminuindo assim a complexidade na aplicação da técnica [8, 9]. Além de sua simplicidade existe todo um conjunto de ferramentas que foi desenvolvido para dar suporte a esse tipo de teste, o que os torna mais atrativos [10]. O problema é que esses testes não são suficientes para fornecer a evidência de corretude de um sistema concorrente, em função do grande número de possibilidades que resultam da execução de um sistema desse tipo.

Um outro tipo de teste são os testes formais [11]. Nestes testes são definidos cenários de execução com base em uma especificação formal do sistema e, a partir dessa especificação são construídos casos de testes. O problema é que esses cenários não são suficientes para a verificação de sistemas concorrentes, já que o comportamento não determinístico de tais sistemas impossibilita a criação de uma quantidade de cenários capaz de cobrir todo o comportamento do sistema.

No campo da verificação estática duas técnicas podem ser utilizadas para verificar sistemas concorrentes: métodos dedutivos e verificação de modelos. Com a utilização de métodos dedutivos, é construído um modelo matemático para cada uma das propriedades do sistema [12]. Com base nesses modelos são buscadas provas matemáticas de que o sistema se comporta corretamente. Pelo fato dos métodos dedutivos serem muito complexos, sua aplicação precisa da presença de profissionais especializados. Aplicando essa técnica, a prova de um sistema simples pode demandar muito tempo, devido à sua complexidade, o que torna sua utilização não condizente com a velocidade com que o desenvolvimento é realizado atualmente [13] (ao menos no que diz respeito aos casos em que a utilização dessa técnica não seja uma necessidade intrínseca, devido à própria natureza do sistema que será desenvolvido).

Com a verificação de modelos, é construído um modelo formal do sistema e a partir dele são derivados os comportamentos possíveis. Em seguida são especificados os comportamentos desejáveis do sistema e, por fim, o modelo e a especificação são submetidos à verificação [14]. A depender das linguagens que estejam sendo utilizadas, essa abordagem pode dispensar a presença de especialistas, já que apenas um treinamento capacitaria pessoas para lidar com as linguagens de modelagem e de especificação.

O problema com os tipos de verificação estática é que o modelo pode não representar corretamente o sistema. Geralmente existe uma falta de conformidade entre código e modelo [15]; o código evolui muito rapidamente enquanto o modelo não é atualizado na mesma velocidade, resultando em um modelo que não descreve o comportamento real do código. É válido ressaltar que existem projetos nos quais a sincronização é muito importante; é necessário um modelo que descreva exatamente o código. Para suprir essa necessidade é feito muito investimento para se manter o modelo atualizado, o que aumenta bastante o custo do projeto. Diferentemente, em projetos que utilizam processos ágeis de desenvolvimento, já que o foco é no produto, o custo-benefício para se manter essa sincronização pode não ser tão atraente.

Pode-se também fazer a verificação de modelos usando a execução do código como especificação para a técnica. Com isso é possível fazer a verificação do modelo - confrontar os comportamentos possíveis com os desejáveis - e ao mesmo tempo fazer a sincronização do código com o modelo, já que a verificação seria feita diretamente a partir do código.

Essa idéia surgiu a partir de um trabalho apresentado por Yan e Garlan [16]. Nesse trabalho é descrita uma técnica que utiliza observações em tempo de execução para construir uma visão arquitetural do sistema. A técnica mostra como eventos de baixo nível -

disparados diretamente do código - podem ser interpretados como operações arquiteturais mais abstratas.

2.2 Linguagens Formais

Existem diversas linguagens formais que podem ser utilizadas na modelagem comportamental de sistemas concorrentes. Dentre as linguagens estudadas estão CSP, MSC e Redes de Petri. Diagramas de estados UML também podem servir para modelar comportamentos de sistemas concorrentes, entretanto o seu uso foi descartado por não possuir o rigor formal que é encontrado em outras linguagens.

2.2.1 CSP

Communicating Sequential Processes (CSP), é um formalismo que permite descrever sistemas como um conjunto de processos seqüenciais e independentes que podem se comunicar através de mensagens síncronas enviadas por canais bem definidos [17]. A descrição do comportamento de um processo é feita através de termos de uma álgebra estendida - álgebra de processos - capaz de representar eventos e comportamentos [18]. Por exemplo, na descrição do comportamento de uma máquina de venda de chocolates (MVC) [17]: ao se colocar uma moeda na máquina, ela libera um chocolate para o consumidor. Nessa máquina é possível a observação dos seguintes eventos:

- *moeda* - a inserção de uma moeda na máquina.
- *chocolate* - a extração de um chocolate da máquina.

O conjunto contendo os nomes dos eventos que são considerados importantes para um modelo constituem o seu alfabeto. Na MVC o alfabeto é α MVC = $\{moeda, chocolate\}$.

Um modelo contém apenas eventos que estejam em seu alfabeto e a ocorrência de um desses eventos é considerada uma ação instantânea. A seguir é mostrado o modelo para representar a máquina que vende chocolates:

MVC = (moeda $\rightarrow$ (chocolate $\rightarrow$ MVC))

A máquina está em um estado inicial. Ao receber uma moeda (primeiro evento) ela libera um chocolate (segundo evento) e volta ao estado inicial.

Outro conceito importante de CSP é o de processo composto. Este é um processo formado a partir de um grupo de outros processos. Vamos imaginar, por exemplo, que existe um consumidor que deseja consumir o chocolate sem pagar, estando disposto a pagar somente se não for possível a extração do doce sem o pagamento. Contudo, a máquina não permite a extração do doce sem o pagamento antecipado. Seu comportamento é representado abaixo:

GULOSO = (chocolate → GULOSO | moeda → chocolate → GULOSO)

A interação entre dois processos corresponde a um processo que possui somente os eventos comuns a ambos os processos. Dessa forma, a interação entre eles é mostrada a seguir:

(GULOSO | MVC) = (moeda → chocolate → (GULOSO | MVC))

Nesse tipo de processo, as interações entre os elementos de cada processo separadamente devem ser escondidas, enquanto que interações entre os processos devem ser visíveis [19].

2.2.2 MSC

Message Sequence Chart (MSC), é uma linguagem que utiliza elementos gráficos e textuais para descrever e especificar interações entre componentes de um sistema [20]. Um MSC é formado por um conjunto de entidades e mensagens, que são trocadas entre as entidades. Cada mensagem é definida pela ação de envio e pela ação de recepção da mensagem. Graficamente, as entidades são representadas por uma linha vertical com um retângulo no topo e o nome da entidade. Cada mensagem é representada por uma seta conectando duas entidades. Na Figura 1 é representado um MSC contendo a troca de duas mensagens, M e N, enviadas da entidade A para a entidade B.

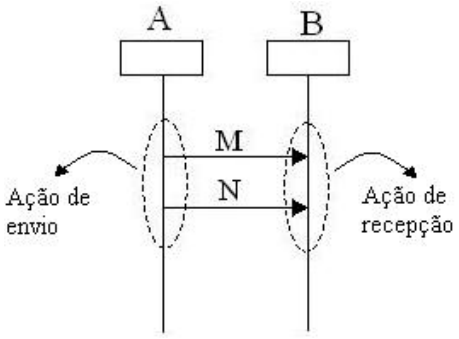

Figura 1: Exemplo de um MSC.

Existe uma relação de ordem entre as ações de uma entidade. Essa relação é definida graficamente de cima para baixo no eixo vertical de cada entidade. Sendo assim, na Figura 1 a mensagem M é enviada por A antes do envio da mensagem N. Da mesma forma, a mensagem M é recebida por B antes da recepção de N.

Em um MSC o envio de uma mensagem sempre ocorre antes de sua recepção. Disso pode-se concluir que, na Figura 1, o envio da mensagem M por parte de A ocorre antes da recepção dessa mesma mensagem por parte de B, da mesma forma que o envio da mensagem N em A ocorre antes de sua recepção em B. Representando as ações de envio e recepção com *send*(*sender*, *receiver*, *message*) e *receive*(*sender*, *receiver*, *message*) respectivamente e observando as relações de ordem já discutidas, pode-se determinar duas seqüências de ações possíveis para o modelo apresentado na Figura 1:

1. *send*(A, B, M) → *send*(A, B, N) → *receive*(A, B, M) → *receive*(A, B, N)
2. *send*(A, B, M) → *receive*(A, B, M) → *send*(A, B, N) → *receive*(A, B, N)

Esta é apenas uma parte simplificada da formalização de MSC. Sua formalização completa pode ser encontrada em [21] e referências importantes estão em [22] e [23].

2.2.3 Redes de Petri

Redes de Petri (RP) é uma linguagem de modelagem gráfica e matemática aplicável a muitos sistemas [24]. Ela é capaz de modelar sincronização de processos, concorrência, conflitos e compartilhamento de recursos, o que faz com que essa linguagem se torne uma ótima ferramenta para modelar sistemas concorrentes [25]. As redes de Petri oferecem um ambiente único para a modelagem, análise formal e simulação, permitindo uma visualização simultânea da estrutura e comportamento do sistema [24]. Mais especificamente, as redes de Petri modelam dois aspectos desses sistemas; eventos e condições e também as relações entre eles. Em cada estado do sistema verificam-se determinadas condições. Essas condições podem possibilitar a ocorrência de eventos, que por sua vez podem ocasionar a mudança de estado do sistema.

Na RP, eventos são representados pelas chamadas transições e as condições são representadas pelos lugares [25]. Os lugares funcionam como depósitos de recursos (fichas) e as transições são ações que manipulam esses recursos. A disposição dos recursos nos lugares representa o estado da rede e, como a função de uma transição é criar e destruir recursos

nos lugares através do disparo de sua ação, as transições surgem como agentes que fazem a rede evoluir de estado para estado [25].

A representação gráfica de uma Rede de Petri é um grafo orientado [25] que possui círculos e retângulos como nós. Os círculos representam os lugares, retângulos representam transições e os arcos (as setas) representam a idéia de um fluxo que vai de uma transição para um lugar ou de um lugar para uma transição. Quando um arco vai de um lugar para uma transição é dito que esse é um lugar de entrada dessa transição, da mesma forma quando um arco vai de uma transição para um lugar é dito que esse é um lugar de saída dessa transição [26]. Na Figura 2, onde é mostrada a representação gráfica de uma RP, é possível ver que os arcos são rotulados com pesos (o peso 1 é omitido) e no interior dos lugares existem fichas, que são os recursos.

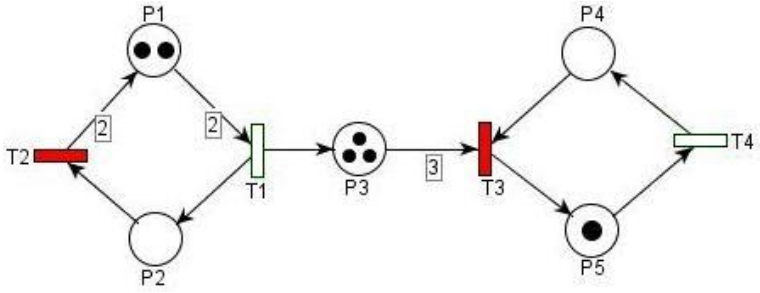

Figura 2: Representação gráfica de uma rede de Petri. [1]

O funcionamento de uma RP se baseia em três regras de disparo de transições [1]:

1. Uma transição está habilitada se cada um dos seus lugares de entrada contém pelo menos um número n de recursos, onde n é o peso do respectivo arco que vai de cada lugar de entrada à transição.

2. Uma transição que está habilitada pode ou não disparar.

3. Se uma transição é disparada, são removidos n recursos de cada um dos lugares de entrada da transição e são adicionados m recursos em cada um dos lugares de saída, onde n é o peso de cada arco de entrada e m é o peso de cada arco de saída da transação.

Com essas regras em mente, fazendo o disparo da transição T4 da Figura 2, a rede passa para o estado que está representado na Figura 3.

Se a partir desse estado for disparada a transição T3, a rede passa para o estado representado na Figura 4.

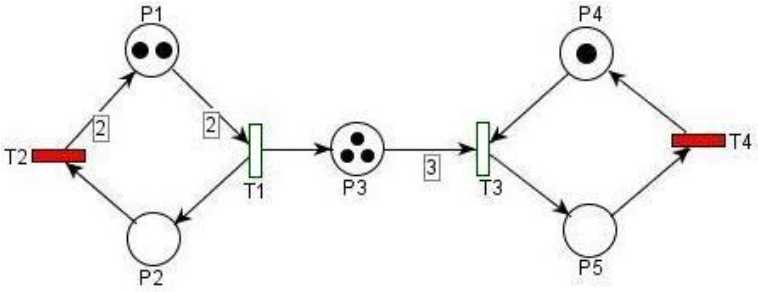

Figura 3: Estado da rede depois do disparo de T4 da Figura 2.

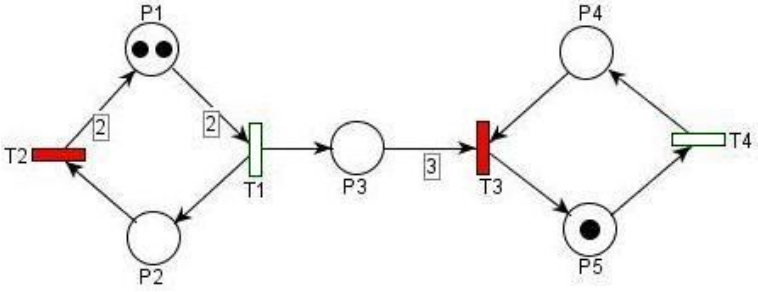

Figura 4: Estado da rede depois do disparo de T3 da Figura 3.

A partir daí, disparando a transição T1, a rede passa para o estado representado na Figura 5 e assim a rede vai "evoluindo"(mudando de estado) - a partir dos disparos de suas transições.

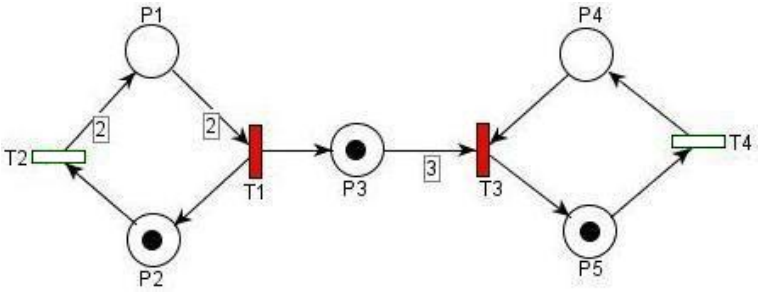

Figura 5: Estado da rede depois do disparo de T2 da Figura 4.

Existem algumas maneiras particulares das transições se relacionarem entre si [1]. É dito que duas transições T1 e T2 formam uma **seqüência** caso T1 possa ocorrer e T2 não possa, mas após a ocorrência de T1, T2 é habilitada. Um exemplo pode ser visto na Figura 6, onde as transições T1 e T2 formam uma seqüência.

Uma situação de **conflito estrutural** ocorre se as transições possuem lugar de entrada em comum como é o caso das transições T1 e T2 da Figura 7. Entretanto, um **conflito efetivo** ocorre quando as transições além de estarem em conflito estrutural estão habilitadas, mas não podem todas serem disparadas, pois o disparo de uma desabilita a(s) outra(s) [26]. Um exemplo de conflito efetivo entre T1 e T2 é o estado da rede que

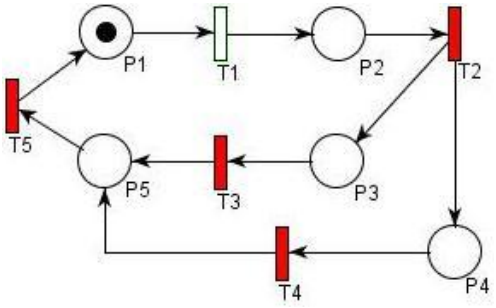

Figura 6: Exemplo de situação de seqüência.

resulta do disparo de T3 na Figura 7. Em uma situação de conflito não se pode prever qual ação irá ocorrer, o que acaba gerando um comportamento não determinístico.

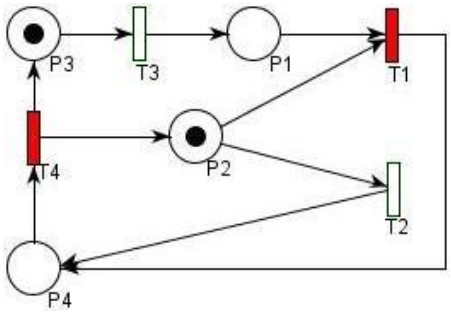

Figura 7: Exemplo de situação de conflito estrutural.

Duas transições podem ocorrer concorrentemente caso elas sejam independentes; não sofram interferência uma da outra. Um exemplo de **concorrência** pode ser visto na Figura 8, onde T1 e T3 ocorrem de forma concorrente (note que é de forma concorrente e não de forma simultânea).

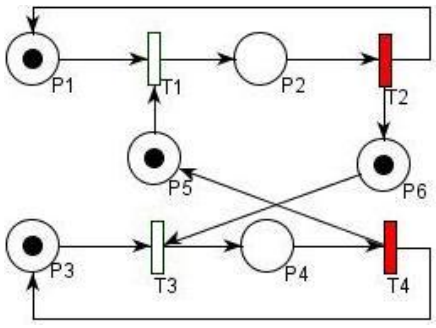

Figura 8: Exemplo de situação de concorrência.

Quando uma situação de conflito e concorrência se misturam em uma única estrutura tem-se uma situação de **confusão**. Observe na Figura 9 que T1 e T3 são concorrentes e que T1 e T2 estão em conflito, assim como T2 e T3. Dessa forma se configura uma situação de confusão.

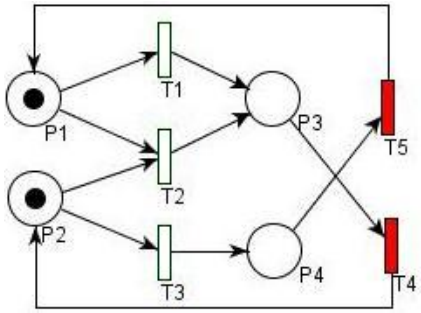

Figura 9: Exemplo de situação de confusão.

A teoria de redes de Petri é muito mais extensa do que o que foi apresentado. Aqui foram passadas apenas uma parte da modelagem gráfica das redes e algumas de suas propriedades mais simples. Existem diversas propriedades estruturais e comportamentais - que se baseiam nos estados que podem ser atingidos durante a execução da rede - além de vários métodos de análise de redes de Petri. Existe ainda toda uma modelagem matemática formal que dá sustentação a essa linguagem, o que faz com que as redes de Petri sejam uma linguagem extremamente poderosa. Além de todo seu poder de representação, ainda existem muitas ferramentas que auxiliam na modelagem e na análise das redes e suas propriedades, abstraindo seus aspectos matemáticos mais densos.

Em [24, 25, 26, 27] pode ser encontrada uma ampla abordagem sobre a teoria, modelagem e análise de redes de Petri, cobrindo inclusive o formalismo matemático que foi omitido aqui. Já em [28, 29] podem ser encontradas ferramentas de modelagem e análise das redes.

Para concluir esta seção, faz-se necessária uma justificativa por se ter optado por Redes de Petri em lugar de CSP ou MSC. Um dos principais motivos para se descartar o uso de CSP como linguagem de modelagem para este trabalho é o fato dela possuir um tipo de notação textual. Isso aumenta bastante o grau de complexidade da linguagem e também a dificuldade para se modelar os sistemas. Um outro motivo é que CSP não dá suporte a eventos assíncronos [19], o que é uma característica peculiar de muitos sistemas concorrentes. Já no caso de MSC, por possuir um tipo de notação mista (tanto gráfico quanto textual) é mais simples de se trabalhar do que CSP. Entretanto, o fato de ser um linguagem para descrever apenas cenários dificulta a análise dos modelos, visto que não se tem uma visão global dos comportamentos modelados [30], apenas as visões de cada cenário separadamente. Outro fator que contribuiu para uma avaliação negativa de MSC é que essa linguagem possui um conjunto muito restrito de ferramentas de software que lhes dêem suporte, o que acaba por dificultar o uso prático da linguagem.

2.3 Programação Orientada a Aspectos

Embora a programação orientada a objetos permita a produção de softwares de qualidade e maiores níveis de reuso [31, 32] e também promova um aumento na produtividade do desenvolvimento e no suporte a mudanças de requisitos, este paradigma possui algumas limitações [33]. Entre estas limitações estão o entrelaçamento e espalhamento de código, que acabam por dificultar a manutenção dos sistemas. O entrelaçamento e o espalhamento surgem da implementação de interesses transversais, que são requisitos específicos cujo código está em objetos ou módulos que não têm a implementação desses requisitos como suas reais funções, afetando diferentes partes do sistema. Para esses objetos (ou módulos) a implementação de tais requisitos é uma função secundária.

A programação orientada a aspectos (AOP) vem para solucionar os problemas causados pela implementação de interesses transversais. Ela aumenta a modularidade do sistema, separando interesses transversais e dessa forma, tornando o código mais legível, aumentando a coesão - pois os interesses transversais não precisam mais ser implementados por vários módulos - e diminuindo o acoplamento - pois diminui a dependência entre os módulos.

Uma abordagem bastante disseminada para trabalhar com o paradigma de orientação a aspectos é o uso de AspectJ [34]. Esta linguagem é uma extensão de Java para AOP e possui o aspecto (*aspect*) como sua principal construção. O aspecto é um tipo abstrato de dados que pode alterar tanto a estrutura estática quanto a estrutura dinâmica de um sistema [35]. A estrutura estática pode ser alterada por meio da adição de atributos, métodos ou construtores a uma classe. Já a estrutura dinâmica é alterada através da adição de comportamentos em determinados pontos que podem ser interceptados no fluxo de execução do programa. Esses pontos são chamados de *join points*. Existem determinadas ações em um programa, suscetíveis a interceptação, que são consideradas *join points*, entre eles estão: chamada e execução de métodos, inicialização de objetos, execução de construtores, tratamento de exceções, acesso a atributos, entre outros [36]. Para cada uma dessas ações podem ser criados *join points* que irão interceptar o fluxo de execução do programa no momento em que essas ações sejam realizadas.

Os *join points* podem ser combinados utilizando os operadores lógicos && (AND), || (OR) e ! (NOT), formando *pointcuts*. Utilizando os *pointcuts* é possível ter acesso a valores de argumentos de métodos, objetos em execução, atributos e exceções dos *join points* [36]. No código 2.1 é declarado um *pointcut* cujo nome é *fireTransition*. Esse *pointcut* recebe

uma String como parâmetro e intercepta o fluxo de execução do programa quando o método *fireTransition()* da classe estática *Transition* é executado. O valor do argumento passado no método *fireTransition()* é atribuído à variável transition do *pointcut* e a partir desse momento o *pointcut* tem acesso a esse valor.

```
1 public pointcut fireTransition(String transition) :
2         execution (public static void Transition.fireTransition(String))
3                 && args(transition);
```

Código 2.1: *Pointcut* que intercepta a execução de um método e recebe um argumento.

Os *pointcuts* são utilizados pelos *advices*, que são as construções que definem o que será adicionado para executar nos *join points*. No código 2.2 é mostrado um *advice* que recebe uma String como argumento passado pelo *pointcut fireTransition* e executa o comando da linha 2 antes (before) do *join point* definido no *pointcut*. Antes da execução do método *Transition.fireTransition(String)* - critérios estabelecidos pelo *pointcut* - será executado o comando *System.out.println(transition)* - comportamento adicionado pelo *advice*.

```
1 before(String transition) : fireTransition(transition){
2    System.out.println(transition);
3 }
```

Código 2.2: *Advice* que insere um comando antes de um *join point*.

Serão mostrados apenas esses pontos sobre AspectJ, já que são os pontos que foram utilizados para a realização deste trabalho. Entretanto, estudos mais completos e aprofundados sobre AspectJ e AOP podem ser feitos em [36], [37] e [35].

3 O Uso da Técnica em Problemas Clássicos de Concorrência

A idéia principal é que o desenvolvedor possa inserir marcações em "pontos-chave" do código do sistema e, dessa forma sejam mapeados no modelo eventos ocorridos no código. A linguagem de modelagem escolhida é redes de Petri [24, 25]. O seu uso é justificado pela facilidade de modelagem em função dos seus recursos gráficos e da existência de um extenso conjunto de ferramentas de software e teorias a respeito desta linguagem.

As marcações inseridas no código pelo desenvolvedor indicam os pontos onde ocorrem eventos que fazem com que o sistema mude seu estado, de forma que essa mudança seja visível no modelo. Assim, tais marcações servirão para indicar quando uma mudança de estado no código irá afetar o estado do modelo. Sendo assim, a cada evento ocorrido no código, que esteja indicado por uma marcação, deve ser sinalizado um evento no modelo em rede de Petri, que represente o evento do código. Isso faz com que o modelo do sistema também mude seu estado.

A dificuldade para fazer a sinalização do evento é saber como capturar esses eventos no código para relacioná-los a seus respectivos "representantes" no modelo. Uma possível abordagem para o mapeamento das marcações na rede de Petri é através da utilização de programação orientada a aspectos (*aspect-oriented programming*, AOP) [36]. A AOP procura solucionar algumas ineficiências da orientação a objetos, como o espalhamento e o entrelaçamento de código [38] resultantes da implementação de interesses transversais (*crosscutting concerns*). Os interesses transversais são funções específicas que o sistema deve executar, mas que estão distribuídas (espalhadas) pelo código [37]. Para se conseguir a modularização desses interesses transversais, foi introduzido pela AOP o conceito de aspecto, que é um tipo abstrato de dado [37]. A implementação dos interesses transversais formam o aspecto, juntamente com diretivas que definem em que lugar do código do programa essa implementação deve ser inserida. Dessa forma, o aspecto isola os interesses transversais, fazendo com que funções que seriam espalhadas por todo o código sejam

isoladas em apenas um local, o que torna o sistema mais modularizado [36].

Como as marcações que devem indicar a sinalização dos eventos no modelo em rede de Petri estarão dispersas pelo código, para evitar que a sinalização de um evento seja implementada no próprio programa - o que é contra os princípios de orientação a objetos - o uso de aspectos é bastante apropriado para a solução dessa questão.

A técnica parte da construção de aspectos para "capturar" as marcações que foram inseridas pelo desenvolvedor nos "pontos-chave" e assim servem para disparar, na rede de Petri, transições que estejam relacionadas com os comandos que foram "marcados". Para tanto, foi necessário utilizar uma ferramenta de simulação de redes de Petri. A ferramenta escolhida foi o Jarp [29]. Ela possui código aberto, é escrita em Java e possui um ambiente para simulação de fácil utilização.

Foram realizados alguns experimentos com o uso de AspectJ com o objetivo de capturar comandos e gerar *traces* (fluxos, linhas ou caminhos) de execução, identificando quais comandos são executados a cada momento. Os experimentos partiram da implementação de alguns problemas clássicos de concorrência com a posterior construção de seus respectivos modelos em rede de Petri.

O programa referente a cada problema foi posto em execução e, depois de gerados os *traces*, foram disparadas manualmente na rede de Petri as transições referentes aos comandos encontrados no *trace*. Dessa forma, a rede de Petri é executada com a mesma linha de execução do programa, o que permite o confronto direto entre o código e o modelo do programa.

Os problemas que foram implementados são: o jantar dos filósofos, produtor-consumidor com *buffer* limitado, leitores e escritores e o problema do barbeiro dorminhoco. Ao longo desta seção serão discutidos cada um desses problemas juntamente com a utilização da técnica em cada um deles.

O código-fonte do aspecto utilizado nos problemas e da classe *Transition*, que é a classe responsável pela marcação no código, pode ser visto no Anexo A.

3.1 O Jantar dos Filósofos

O problema do jantar dos filósofos é um problema clássico de concorrência [2]. Esse problema consiste em uma mesa redonda, rodeada por filósofos, na qual estão uma tigela de arroz e uma determinada quantidade de garfos, de forma que cada garfo fica entre

dois filósofos, como pode ser visto na Figura 10. Enquanto estão meditando, os filósofos não interagem com seus colegas. Entretanto, quando um filósofo fica com fome ele tenta pegar os dois garfos mais próximos - os que estão entre ele e seus colegas da esquerda e da direita. Um filósofo só pode pegar um garfo de cada vez e ele não pode pegar um garfo que já esteja na mão de um colega. Enquanto estiver com os garfos, o filósofo pode comer pelo tempo que quiser, sem soltar os garfos. Ao terminar sua refeição, o filósofo devolve os garfos à mesa e volta a meditar.

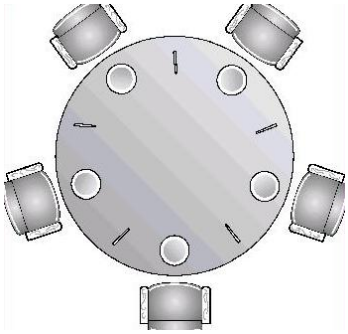

Figura 10: Representação do Jantar dos Filósofos [2].

A importância desse problema não se deve à sua natureza prática, mas porque ele é uma representação simples da necessidade de alocar vários recursos entre vários processos sem incorrer em *deadlock*.

Na Figura 11 pode ser visto o modelo que foi feito em rede de Petri e que representa o problema do jantar dos filósofos para dois filósofos. No modelo podem ser vistos os lugares que, dependendo da forma como estão preenchidos com recursos, representam determinados estados do sistema. Nessa mesma figura é mostrado o estado inicial, no qual os filósofos estão pensando e os garfos estão na mesa - observe que os lugares que representam cada um dos filósofos pensando e cada um dos garfos na mesa estão preenchidos com recursos. Fazendo o disparo da transição *Pega garfo direito* para *Filosofo0*, por exemplo, o sistema evolui para um outro estado, que é o estado em que *Filosofo0* tenta pegar o segundo garfo (*Garfo1*). Nesse momento o recurso que representa *Filosofo0* pensando e o que representa *Garfo0* na mesa serão consumidos e será gerado um recurso no lugar que representa o estado em que *Filosofo0* já pegou *Garfo0* mas ainda não pegou *Garfo1*, como mostra a Figura 12. Os estados possíveis para cada um dos filósofos são:

- Pensando, que é quando o filósofo não tem nenhum garfo;
- Estado intermediário em que o filósofo pegou o primeiro garfo;
- Comendo, que é o estado em que o filósofo pega o segundo garfo;

- Estado intermediário em que o filósofo devolve o primeiro garfo.

Quando o filósofo devolve o segundo garfo, automaticamente ele volta para o estado em que fica pensando.

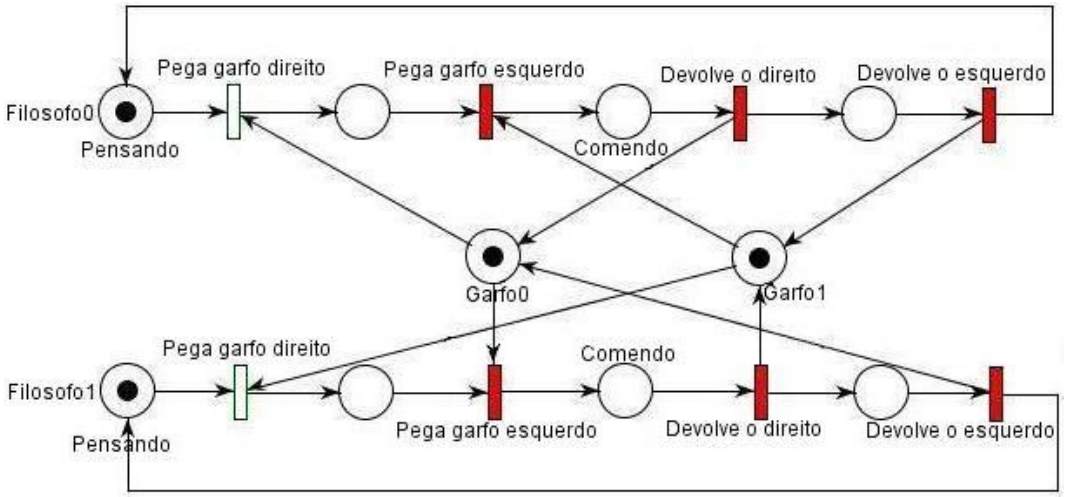

Figura 11: Modelo em Rede de Petri para o Jantar dos Filósofos.

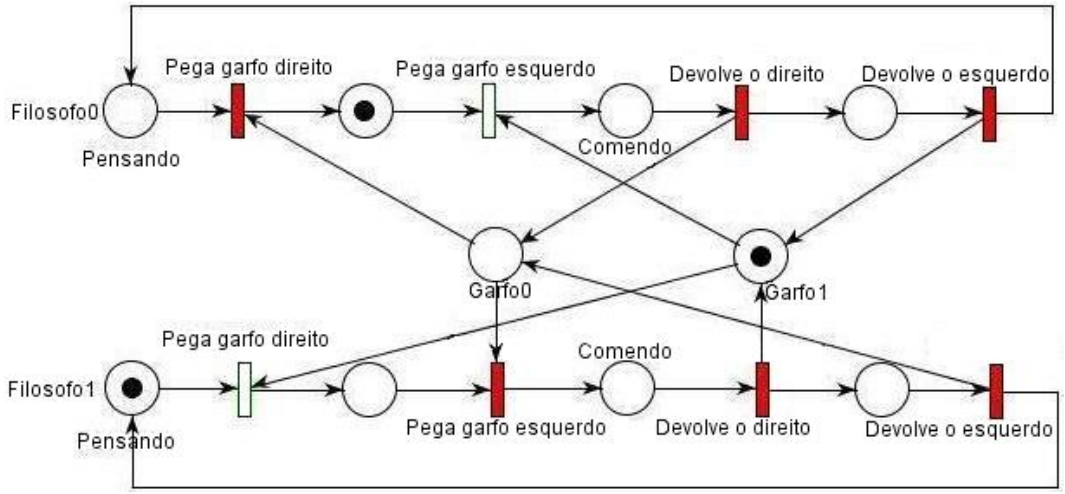

Figura 12: Estado da rede depois do disparo *Pega garfo direito* na Figura 11.

É importante notar que nesse modelo existe a possibilidade da ocorrência de *deadlock*, que irá ocorrer se cada um dos filósofos pegar um garfo. Dessa forma, com cada um dos filósofos disparando sua transição *Pega garfo direito*, não haverá mais transições habilitadas. A Figura 13 retrata como fica o estado do sistema nessa situação.

Foi feita a implementação em Java desse problema dos filósofos, tentando representar o que é descrito no modelo. O código do programa foi marcado em pontos-chave, pontos esses que se localizam logo depois de cada instrução de código responsável por uma

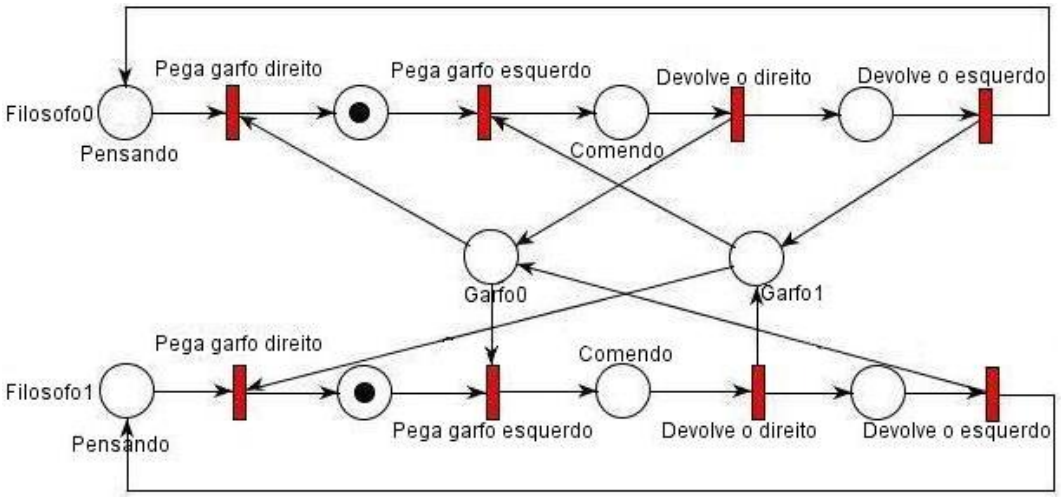

Figura 13: Modelo retratando o estado de *deadlock* do sistema.

mudança de estado no sistema que seja visível no modelo. Dessa forma, tem-se que essas instruções (ou comandos) são os "representantes", no código, das transições presentes na rede, já que as transições são as responsáveis pela mudança de estado do sistema no modelo.

Quando o programa está em execução, essas marcações são "capturadas" por um aspecto que irá imprimir qual transição deve ser disparada na rede de Petri. Depois de um determinado tempo, ou ao final da execução do programa, obtém-se a seqüencia das transições que devem ser disparadas. De posse dessa seqüencia de transições, a atenção é voltada para o modelo. As transições da rede de Petri são disparadas na seqüencia em que foram escritas pelo aspecto, obedecendo assim o fluxo de execução do programa e conseqüentemente a alternância que ocorreu entre os processos durante sua execução.

No Código 3.1 é mostrado o método *run()* da classe *Philosopher*. Durante a execução desse método, são invocados os métodos *get()* e *put()* da classe *Fork*. Já na classe *Fork*, nos métodos citados, foram inseridas marcações, que podem ser vistas nas linhas 3 e 10 do Código 3.2 (método *fireTransition()* da classe *Transition*, que tem sua implementação mostrada no Código A.2 do Anexo A). Na linha 3 essa marcação fica logo após a instrução *taken=false*; que representa uma mudança no estado do sistema visível no modelo, ou seja, o ato do filósofo devolver um dos garfos à mesa. Da mesma forma, a marcação situada na linha 10 do Código 3.2 localiza-se após a instrução *taken=true*; que também representa uma mudança de estado no sistema visível no modelo - o ato do filósofo pegar um dos garfos da mesa.

```
1 public void run() {
2    try {
3       while (true) {
```

```
4            //thinking
5            sleep(controller.sleepTime());
6
7            //hungry
8            right.get(identity);
9
10           //got right chop stick
11           sleep(500);
12
13           left.get(identity);
14
15           //eating
16           sleep(controller.eatTime());
17           sleep(controller.eatTime());
18
19           right.put(identity);
20           left.put(identity);
21       }
22   }catch(InterruptedException e){}
23 }
```

Código 3.1: Método *run()* da classe *Philosopher*.

```
1  public synchronized void put(int phil) {
2      taken=false;
3      Transition.fireTransition("Filósofo "+phil+" devolve garfo "+identity);
4      notify();
5  }
6
7  public synchronized void get(int phil)throws InterruptedException {
8      while (taken) wait();
9      taken=true;
10     Transition.fireTransition("Filósofo "+phil+" pega garfo "+identity);
11 }
```

Código 3.2: Métodos *put()* e *get()* da classe *Fork*.

Durante a execução do programa, essas marcações são capturadas pelo aspecto, cuja implementação é mostrada no Código A.1 do Anexo A, e é gerado um *trace* de execução, como o da Figura 14, por exemplo. Confrontando esse *trace* com a rede de Petri da Figura 11, observa-se as correspondências entre as linhas do *trace* e as transições da rede, pois o que o aspecto escreve são exatamente as transições na ordem em que devem ser disparadas. As linhas em destaque na Figura 14, por exemplo, representam as transições *Pega garfo direito* e *Devolve o direito* de *Filosofo1* da Figura 11, respectivamente.

O método *Transition.fireTransition()* é a marcação responsável por indicar quando um determinado comando é executado. Assim, quando o aspecto "captura" a instrução *Transition.fireTransition()* e escreve a linha do trace relativa a essa marcação, ele na verdade está mapeando o comando que se deseja marcar no código, para a transição na rede e a identificação dessa transição é feita pelo parâmetro que é passado para o método *fireTransition()*. Dessa forma, com o código em execução é possível simular a execução da rede e assim saber se o código está em conformidade com o modelo, pois se para algum

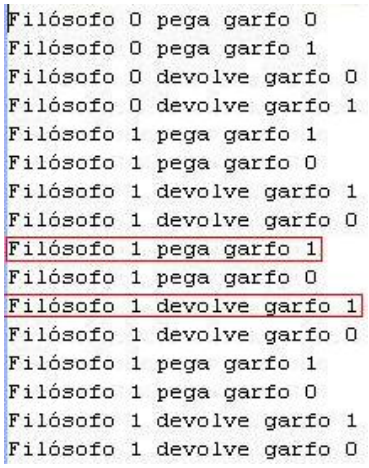

Figura 14: *Trace* gerado pela execução do programa dos filósofos.

dos eventos registrados no *trace* não há transições habilitadas, então a conformidade não foi atendida em algum ponto. O próprio uso do programa é um teste que está sendo executado. Se o programa for executado n vezes, serão n testes realizados para verificar a conformidade do código com o modelo.

3.2 O *Buffer* Limitado

O *buffer* limitado é um outro problema clássico bastante citado na literatura. O problema consiste de dois processos, um produtor e um consumidor, que compartilham um *buffer* de n elementos. O processo produtor é responsável por produzir recursos e disponibilizá-los no *buffer* para serem consumidos pelo processo consumidor. Deve-se ter o cuidado para que o produtor não produza nenhum recurso quando o *buffer* já estiver cheio, para evitar a perda dos recursos produzidos anteriormente e que ainda não foram consumidos. É necessário evitar também que o consumidor tente consumir algum recurso enquanto o *buffer* estiver vazio.

Na Figura 15 pode ser visto o modelo em rede de Petri construído para representar o problema do buffer limitado para duas posições. Nesse modelo é usada uma analogia a um estacionamento com apenas duas vagas. Os carros chegam através do disparo da transição *arrive()* e saem do estacionamento através do disparo da transição *depart()*. Quando um carro chega, uma ficha é retirada do lugar *Para chegar* e é criada uma ficha em *Para partir*. Esses lugares funcionam para indicar, aos processos produtor e consumidor, quantos carros ainda podem entrar no estacionamento e quantos podem sair, respectivamente.

Fazendo um disparo em *arrive()*, por exemplo, o modelo evolui para o estado repre-

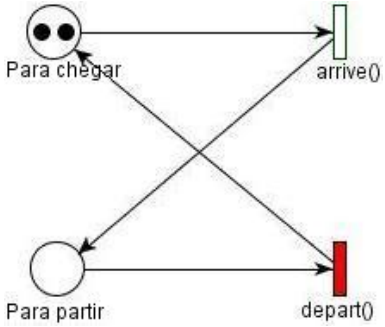

Figura 15: Modelo em Rede de Petri para o *Buffer* Limitado.

sentado na Figura 16. Observe que foi consumida uma ficha de *Para chegar* e criada uma ficha em *Para sair*. Como só resta uma ficha em *Para chegar* isso significa que apenas mais um carro poderá entrar no estacionamento, pois isso acontecendo a transição *arrive()* será desabilitada. Da mesma forma, como em *Para sair* possui apenas uma ficha - representando que existe apenas um carro no estacionamento - apenas um carro pode sair, pois logo depois *depart()* será desabilitada e o modelo voltará ao estado representado na Figura 15.

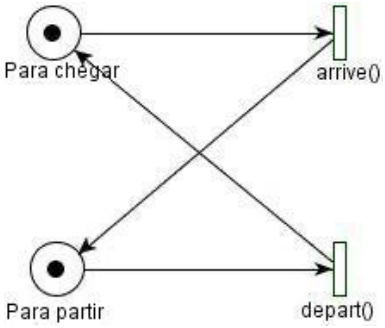

Figura 16: Estado da rede depois do disparo de *arrive()* na Figura 15.

Foi feita a implementação em Java do problema e foram inseridas as marcações nos pontos-chave. Observando o modelo, os instantes em que ocorre uma mudança de estado são quando chega um carro e quando sai um carro. Dessa forma as marcações foram inseridas nos métodos *arrive()* e *depart()* da classe *CarParkTwo*, logo após as instruções que indicam que um carro saiu ou chegou. Os métodos com as marcações podem ser vistos no Código 3.3.

```
1 public synchronized void arrive() {
2     while (spaces==capacity){
3         try {
4             wait();
5         } catch (InterruptedException e) {
6             e.printStackTrace();
7         }
8     }
9     ++spaces;
```

```
10      Transition.fireTransition("Um carro chegou!");
11      notifyAll();
12  }
13
14  public synchronized void depart() {
15      while (spaces==0){
16          try {
17              wait();
18          } catch (InterruptedException e) {
19              e.printStackTrace();
20          }
21      }
22      --spaces;
23      Transition.fireTransition("Um carro partiu!");
24      notifyAll();
25  }
```

Código 3.3: Métodos *arrive()* e *depart()* da classe *CarParkTwo*.

Como é mostrado na linha 10, foi inserida uma marcação logo após a instrução *++spaces*, que representa uma mudança de estado visível no modelo; indica a chegada de um carro ao estacionamento. Da mesma forma, na linha 23 foi inserida uma marcação logo após a instrução *–spaces*, que indica a saída de um carro do estacionamento; uma outra mudança visível no modelo.

Nos Códigos 3.4 e 3.5 são mostrados os métodos *run()* das classes produtora *Arrivals* e consumidora *Departures*, respectivamente. É possível notar que a única diferença entre elas é que *Arrivals* invoca o método *arrive()* de *CarParkTwo*, enquanto *Departures* invoca o método *depart()*.

```
1   public void run() {
2       while(true) {
3           try {
4               sleep(sleepTime());
5               this.carpark.arrive();
6               sleep(sleepTime());
7           } catch (InterruptedException e) {
8               e.printStackTrace();
9           }
10      }
11  }
```

Código 3.4: Método *run()* da classe *Arrivals*.

```
1   public void run() {
2       while(true) {
3           try {
4               sleep(sleepTime());
5               this.carpark.depart();
6               sleep(sleepTime());
7           } catch (InterruptedException e) {
8               e.printStackTrace();
9           }
10      }
11  }
```

Código 3.5: Método *run()* da classe *Departures*.

Colocando o programa em execução, foi gerado um *trace* como o da Figura 17, a partir da captura das marcações por parte do aspecto. A transição "*Um carro chegou!*", que é escrita pelo trace, equivale à transição *arrive()* do modelo, enquanto que "*Um carro partiu!*" equivale a *depart()*.

```
Um carro chegou!
Um carro partiu!
Um carro chegou!
Um carro chegou!
Um carro partiu!
Um carro chegou!
Um carro partiu!
Um carro chegou!
Um carro partiu!
Um carro chegou!
```

Figura 17: *Trace* gerado pela execução do programa do *buffer* limitado.

Confrontando o *trace* gerado e o modelo do sistema é possível notar a conformidade existente, pois fazendo o disparo, no modelo, das transições referentes ao *trace*, na mesma ordem em que foram escritas, observa-se que a execução do código e do modelo se encaixam perfeitamente, o que não poderia ser notado, por exemplo, caso a transição "*Um carro chegou!*" tivesse sido escrita no *trace* mais de duas vezes seguidas. Tomando como base o modelo da Figura 15 e fazendo o disparo das transições na ordem descrita pelo *trace* da Figura 17, ao final do *trace* o estado do modelo passa para o que é mostrado na Figura 18.

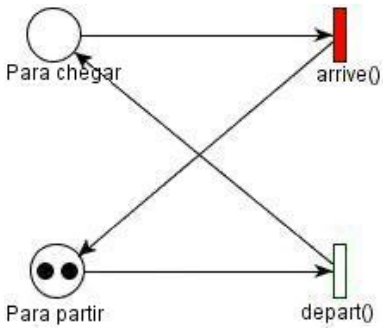

Figura 18: Estado da rede após o disparo das transições do *trace* da Figura 17.

3.3 Leitores e Escritores

Este é um outro problema já bastante conhecido entre os problemas concorrentes e é utilizado para modelar o acesso concorrente de processos leitores e escritores a uma mesma base de dados. O acesso à base pode ser feito simultaneamente por vários leitores. Entretanto, se um escritor estiver acessando - escritores realizam alterações - nenhum outro processo poderá ter acesso à base de dados, seja ele leitor ou escritor, para não gerar inconsistência de dados.

Na Figura 19 é mostrado o modelo em rede de Petri para um problema com três leitores e dois escritores. É importante observar que a estrutura desse modelo serve para representar o problema para quaisquer número de leitores ou escritores, bastando para isso acrescentar os recursos necessários nos devidos lugares.

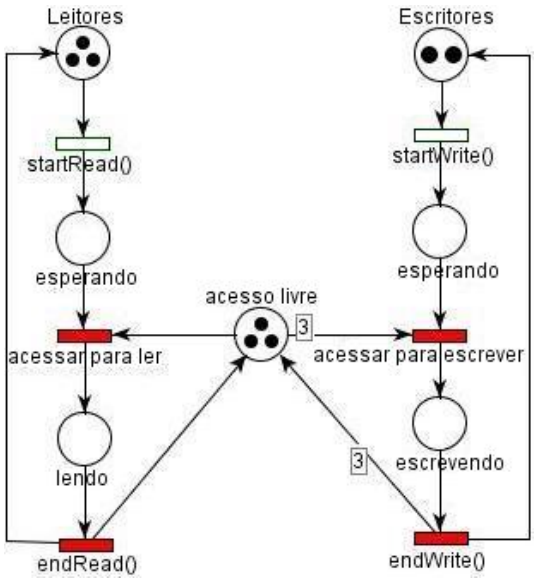

Figura 19: Modelo em rede de Petri para o problema dos leitores e escritores.

Observe que quando a transição *startRead()* é disparada, o leitor vai para um estado de espera, no qual ele aguarda para que a base de dados esteja livre para a leitura. Somente depois que foi constatado que não existem escritores acessando - o lugar *acesso livre* contém um ou mais recursos - é que o leitor pode ir para o estado no qual irá fazer a leitura - através do disparo de *acessar para ler*. Da mesma forma, quando a transição *startWrite()* é disparada, o escritor também vai para um estado de espera, no qual ele aguarda para que a base esteja livre para a escrita. Observe que o arco que vai de *acesso livre* para *acessar para escrever* possui peso 3 (três), isso quer dizer que nenhum leitor poderá estar em *lendo* para que *acesso livre* esteja habilitada e o escritor poder ir para *escrevendo*. Assim, *acesso livre* funciona como um lugar que controla quem está acessando

a base de dados, pois seus recursos são retirados antes do acesso à base e são repostos depois do acesso.

A seguir são vistas as implementações dos métodos *run()* do leitor e do escritor, nos Códigos 3.6 e 3.7, respectivamente.

```
1  public void run(){
2      while(true){
3          try {
4              sleep(base.sleepTime());
5              base.startRead();
6              sleep(base.sleepTime());
7              base.endRead();
8          } catch (InterruptedException e) {
9              e.printStackTrace();
10         }
11     }
12 }
```

Código 3.6: Método *run()* da classe *Reader*.

```
1  public void run(){
2      while(true){
3          try {
4              sleep(base.sleepTime());
5              base.startWrite();
6              sleep(base.sleepTime());
7              base.endWrite();
8          } catch (InterruptedException e) {
9              e.printStackTrace();
10         }
11     }
12 }
```

Código 3.7: Método *run()* da classe *Writer*.

A classe *Reader* invoca o método o método *startRead()* de *Database* para pedir acesso de leitura à base. Quando o acesso é liberado, o leitor faz a leitura - que é "simulada" pela instrução *sleep(base.sleepTime())* - e depois encerra seu acesso invocando o método *endRead()*. A classe *Writer* invoca o método *startWrite()* e espera até que o acesso a escrita seja liberado. Depois que a escrita é feita ele encerra seu acesso invocando o método *endWrite()* de *Database*. Os métodos da classe *Database* são mostrados nos Códigos 3.8 e 3.9.

```
1  public synchronized void startRead() {
2      Transition.fireTransition("Um leitor pediu acesso");
3      while(dbWriting == true){
4          try {
5              wait();
6          } catch (InterruptedException e) {
7              e.printStackTrace();
8          }
9      }
10     ++readerCount;
11     Transition.fireTransition("Leitor acessando");
```

```
12      if(readerCount == 1){
13          dbReading = true;
14      }
15  }
16
17  public synchronized void endRead() {
18      --readerCount;
19      Transition.fireTransition("Um leitor terminou!");
20      if(readerCount == 0){
21          dbReading = false;
22      }
23      notifyAll();
24  }
```

Código 3.8: Métodos da classe *Database* invocados pelos leitores.

No método *startRead()* foi inserida uma marcação logo na primeira linha. Essa marcação indica a execução da instrução equivale à transição *startRead()* do modelo. Logo depois da marcação, o leitor que invocou o método fica bloqueado enquanto existir algum escritor acessando a base de dados. Depois que passa pelo *while*, o leitor incrementa um contador (que guarda a quantidade de leitores que estão acessando a base) e, caso seja o primeiro leitor a obter acesso, ele faz *dbReading = true*, para informar que agora existem leitores acessando os dados. Logo após o *if* foi inserida uma outra marcação, que equivale à transição *acessar para ler* do modelo, que é disparada para que o leitor obtenha o acesso que foi solicitado, passando para *lendo*, que é onde de fato ocorre a leitura dos dados.

No método *endRead()* - método invocado quando um leitor termina sua leitura - o contador de leitores é decrementado e, caso seja o ultimo leitor a acessar o banco, é feito *dbReading = false* (isso para indicar que o ultimo leitor que estava acessando acaba de sair). Após essa instrução foi inserida uma marcação, que é referente à transição *endRead()* do modelo.

```
1   public synchronized void startWrite() {
2       Transition.fireTransition("Um escritor pediu acesso!");
3       while(dbReading == true || dbWriting == true){
4           try {
5               wait();
6           } catch (InterruptedException e) {
7               e.printStackTrace();
8           }
9       }
10      dbWriting = true;
11      Transition.fireTransition("Escritor acessando!");
12  }
13
14  public synchronized void endWrite() {
15      dbWriting = false;
16      Transition.fireTransition("O escritor terminou!");
17      notifyAll();
18  }
```

Código 3.9: Métodos da classe *Database* invocados pelos escritores.

No método *startWrite()*, a marcação inserida logo na primeira linha é referente à transição *startWrite()*, que é quando o escritor pede acesso. Logo após a marcação, o escritor fica bloqueado até que não existam leitores nem escritores acessando a base de dados. Passando pelo *while* é feito $dbWriting = true$ - para indicar que passou a ter um escritor acessando a base - e logo depois foi inserida uma marcação, equivalente à transição *acessar para escrever* no modelo, que é através da qual se obtém de fato o acesso à escrita.

No método *endWrite()* é feito $dbWriting = false$, para indicar que o escritor liberou o acesso aos dados e logo depois foi inserida uma marcação que se refere à transição *endWrite()* do modelo.

Colocando o programa em execução, foi gerado um *trace* como o da Figura 20, por exemplo. Fazendo o disparo, no modelo, das transições indicadas pelo *trace*, observa-se que o comportamento demonstrado pelo código está em conformidade com o modelo. Depois do disparo dessas transições o modelo irá se encontrar no estado mostrado na Figura 21.

```
Um leitor pediu acesso
Leitor acessando
Um leitor pediu acesso
Leitor acessando
Um leitor pediu acesso
Leitor acessando
Um escritor pediu acesso!
Um leitor terminou!
Um escritor pediu acesso!
Um leitor terminou!
Um leitor pediu acesso
Leitor acessando
Um leitor terminou!
Um leitor pediu acesso
Leitor acessando
Um leitor terminou!
Um leitor terminou!
Escritor acessando!
Um leitor pediu acesso
Um leitor pediu acesso
O escritor terminou!
```

Figura 20: *Trace* gerado pela execução do programa dos leitores e escritores.

3.4 Barbeiro Dorminhoco

Este é um problema típico, utilizado para modelar leitores de disco. Em uma barbearia trabalha um barbeiro, que fica a espera de clientes. Quando um cliente chega, se o barbeiro estiver ocupado, ele espera até que o barbeiro o chame para se sentar e cortar o cabelo, por outro lado, se o barbeiro estiver livre o cliente é chamado para sentar-se a fim de

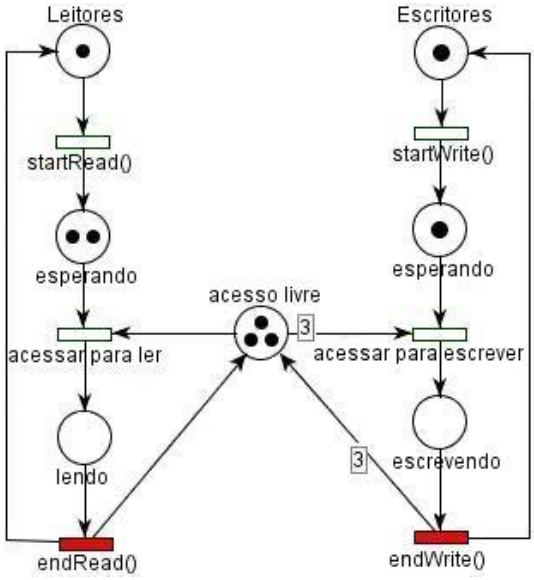

Figura 21: Estado da rede da Figura 19 depois da execução do *trace* da Figura 20.

realizar o corte. Quando o barbeiro termina o corte ele tem que abrir a porta para o cliente sair e então chamar o próximo cliente. O cliente só pode sair, ao terminar o corte, depois que o barbeiro abre a porta.

Na Figura 22 é mostrado o modelo em rede de Petri para o problema do barbeiro com três clientes. Observe que o barbeiro e o cliente se sincronizam logo no início da execução; o barbeiro espera o cliente chegar e o cliente espera o barbeiro chamá-lo para sentar-se. Caso o barbeiro não chame nenhum cliente ou não haja nenhum cliente esperando, a transição *sentar na cadeira* não estará habilitada. No momento em que o cliente senta-se, enquanto o barbeiro corta seu cabelo, ele fica apenas aguardando o barbeiro abrir a porta para que ele possa sair. O barbeiro só irá abrir a porta para o cliente sair depois que terminar de cortar o cabelo do cliente. O momento da saída é um outro ponto de sincronização, pois o barbeiro só irá chamar o próximo depois que o cliente sair.

Os métodos *run()* das classes *Barber* e *Client* são mostrados nos Códigos 3.10 e 3.11, respectivamente. O barbeiro invoca o método *nextClient()* de *BarberShop* para sincronizar com o cliente até o momento em que este senta na cadeira, invoca o método privado *cutHair()* para cortar o cabelo e, por fim, invoca o método *endHairCut()*, também de *BarberShop*, para sincronizar com o cliente até o momento em que este sai da barbearia. Já o cliente, o único método de *BarberShop* invocado por ele é *cutHair()*, que é a requisição do serviço que será realizado pelo barbeiro.

```
1 public void run() {
2    while( true ) {
3       try {
4          sleep((long) (100*Math.random()));
```

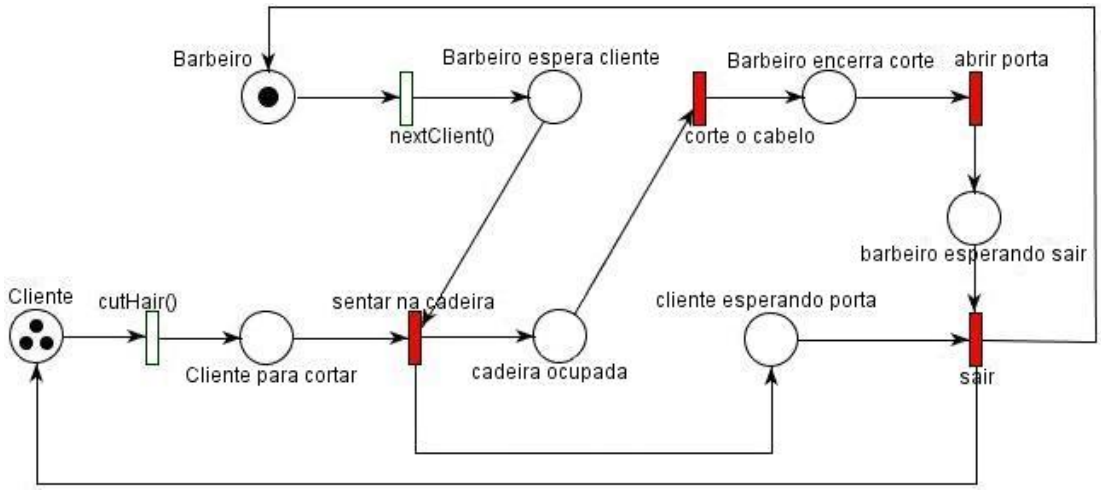

Figura 22: Modelo em rede de Petri para o problema do barbeiro dorminhoco.

```
5             barberShop.nextClient();
6             this.cutHair();
7             barberShop.endHairCut();
8         } catch (InterruptedException e) {
9             e.printStackTrace();
10        }
11    }
12 }
```

Código 3.10: Método *run()* da classe *Barber*.

```
1  public void run() {
2      while(true) {
3          try {
4              sleep(1 + (int)(Math.random()*4000));
5              barberShop.cutHair(id);
6          } catch (InterruptedException e) {
7              e.printStackTrace();
8          }
9      }
10 }
```

Código 3.11: Método *run()* da classe *Client*.

Nos Códigos 3.12 e 3.13 são mostrados os métodos da classe *BarberShop* que são utilizados pelo barbeiro e pelos clientes, respectivamente.

```
1  public synchronized void nextClient() {
2      barber++;
3      Transition.fireTransition("BARBEIRO: Próximo cliente!");
4      notifyAll();
5      while( chair == 0 ) {
6          try {
7              wait();
8          } catch (InterruptedException e) {
9              e.printStackTrace();
10         }
11     }
12     chair--;
13     Transition.fireTransition("BARBEIRO: Cortando cabelo!");
```

```
14 }
15
16 public synchronized void endHairCut() {
17     open++;
18     Transition.fireTransition("BARBEIRO: Porta aberta!");
19     notifyAll();
20     while(open > 0){
21         try {
22             wait();
23         } catch (InterruptedException e) {
24             e.printStackTrace();
25         }
26     }
27 }
```

Código 3.12: Métodos da classe *BarberShop* invocados pelo barbeiro.

Logo na primeira linha do método *nextClient()* - que é invocado pelo barbeiro - é feito *barber++* para indicar que o barbeiro está livre. Como essa instrução provoca uma mudança de estado visível no modelo - o disparo de *nextClient()* na rede de Petri - foi inserida uma marcação na linha logo abaixo, para indicar tal mudança de estado. Logo após, é executada a instrução *notifyAll()*, para notificar que o barbeiro está livre a possíveis clientes que estejam esperando e então o barbeiro passa a esperar que o próximo cliente sente-se na cadeira. Depois que o barbeiro encerra sua espera - que é quando o cliente se senta - é feito *chair–*, para indicar que a cadeira agora está ocupada - esse é o momento que o barbeiro começa a realizar o corte de cabelo. Nesse ponto foi inserida uma marcação, para representar a transição *corte o cabelo* da rede.

Depois que realiza o corte, o barbeiro invoca o método *endHairCut()*, que executa a instrução *open++* para indicar que a porta está aberta. Nesse ponto é feita uma marcação pois a instrução supramencionada representa a transição *abrir porta*. Logo após é feito *notifyAll()* para notificar ao cliente que a porta está aberta e em seguida o barbeiro espera até que o cliente saia da barbearia.

```
1  public synchronized void cutHair(int id) {
2      Transition.fireTransition("CLIENTE: Vou esperar a vez!");
3      while (barber == 0){
4          try {
5              wait();
6          } catch (InterruptedException e) {
7              e.printStackTrace();
8          }
9      }
10     barber--;
11     chair++;
12     Transition.fireTransition("CLIENTE: Sentei na cadeira!");
13     notifyAll();
14     while(open == 0){
15         try {
16             wait();
17         } catch (InterruptedException e) {
```

```
18              e.printStackTrace();
19          }
20      }
21      open--;
22      Transition.fireTransition("CLIENTE: Vou sair!");
23      notifyAll();
24  }
```

Código 3.13: Método da classe *BarberShop* invocado pelos clientes.

Já no método *cutHair()* - invocado pelo cliente - logo na primeira linha foi inserida uma marcação. Isso porque o simples fato de invocar o método quer dizer que o cliente chegou à barbearia e isso leva ao disparo da transição *cutHair()* no modelo. Logo depois o cliente espera até que o barbeiro esteja livre. Assim que termina a espera, é feito *barber–*, para indicar que o barbeiro agora está ocupado, e *chair++*, para indicar que a cadeira agora está ocupada. Nesse ponto foi inserida uma marcação, pois ele representa a transição *sentar na cadeira*. O barbeiro é notificado de que o cliente já se sentou e então o cliente aguarda até que o barbeiro abra a porta - o que irá acontecer depois que o corte terminar. Continuando a execução - depois que a porta já foi aberta pelo barbeiro - é feito *open–*, para indicar que o cliente saiu (aqui foi inserida uma marcação, já que esse ponto representa a transição *sair*) e o barbeiro é notificado da saída do cliente.

A Figura 23 apresenta um *trace* gerado a partir da execução desta implementação do barbeiro dorminhoco. Fazendo o disparo, no modelo, das devidas transições na mesma seqüência em que foram registradas no *trace*, é possível notar a conformidade entre o código e o modelo propostos para o problema.

```
BARBEIRO: Próximo cliente!
CLIENTE: Vou esperar a vez!
CLIENTE: Sentei na cadeira!
BARBEIRO: Cortando cabelo!
BARBEIRO: Porta aberta!
CLIENTE: Vou sair!
BARBEIRO: Próximo cliente!
CLIENTE: Vou esperar a vez!
CLIENTE: Sentei na cadeira!
BARBEIRO: Cortando cabelo!
CLIENTE: Vou esperar a vez!
BARBEIRO: Porta aberta!
CLIENTE: Vou sair!
BARBEIRO: Próximo cliente!
CLIENTE: Sentei na cadeira!
BARBEIRO: Cortando cabelo!
CLIENTE: Vou esperar a vez!
CLIENTE: Vou esperar a vez!
BARBEIRO: Porta aberta!
```

Figura 23: *Trace* gerado pela execução do programa do barbeiro dorminhoco.

O estado da rede da Figura 22, depois da execução do *trace* mostrado na Figura 23 é

apresentado na Figura 24.

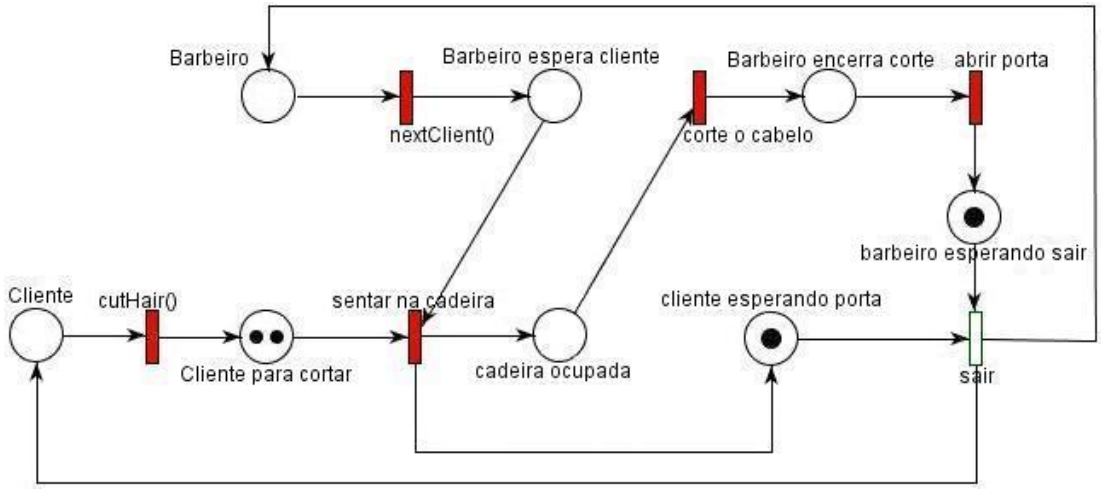

Figura 24: Estado da rede da Figura 22 depois da execução do *trace* da Figura 23.

3.5 Problemas Encontrados

Durante as fases iniciais do trabalho, a idéia da técnica era basicamente a mesma: inserir marcações em pontos-chave do código, para que essas marcações pudessem ser capturadas e, a partir daí, fosse possível disparar eventos em um modelo comportamental, através da relação entre as marcações capturadas e os eventos do modelo. Entretanto, o que mudava era o local onde eram inseridas essas marcações.

Inicialmente as marcações eram inseridas depois da chamada de um método que modificasse o estado do sistema de forma visível no modelo. Ou seja, a marcação não era inserida dentro de um método, depois da instrução que realmente modifica o estado do sistema; ela era inserida depois da chamada do método que contém a instrução que faz essa modificação. Dessa forma a marcação era capturada apenas depois da execução de todo o método, diferente de como acontece atualmente, que a captura ocorre durante a execução do método.

A marcação sendo inserida depois da chamada de um método, ao invés de ser inserida dentro do próprio método, permite um alcance muito maior à técnica. Seria possível, por exemplo, inserir marcações depois de métodos que fizessem parte de bibliotecas externas, cujo código o desenvolvedor não tem acesso. Entretanto, essa forma de inserção das marcações ocasionou um problema de **atomicidade**.

Para que a técnica funcione corretamente é necessário que a execução da instrução marcada e a execução do aspecto ao capturar a marcação sejam feitas de forma atômica.

Isso quer dizer que entre a execução da instrução e a execução do aspecto não pode haver a execução de uma outra instrução, por parte de um outro processo, que modifique o estado do sistema de forma visível no modelo. Esse é o problema de atomicidade: a instrução marcada e o aspecto não são executados de forma atômica - não são executados inteiramente sem interrupção - devido à alternância de contexto entre os processos em execução. Quando isso acontece o processo de verificação perde a sua consistência, porque se ocorre uma mudança de estado e essa ocorrência só é informada pelo aspecto depois da ocorrência de uma outra mudança, a seqüência correta das mudanças é alterada e, logicamente, quando essa seqüência for confrontada com o modelo, não haverá conformidade. Tomando como exemplo o problema do jantar dos filósofos, pode acontecer do aspecto informar - incorretamente - que um filósofo pegou um garfo que ainda não foi posto na mesa, como pode ser visto no *trace* da Figura 25.

```
Filósofo 0 pega garfo 0
Filósofo 0 pega garfo 1
Filósofo 0 devolve garfo 0
Filósofo 0 devolve garfo 1
Filósofo 0 pega garfo 0
Filósofo 0 pega garfo 1
Filósofo 0 devolve garfo 0
Filósofo 1 pega garfo 1
Filósofo 0 devolve garfo 1
Filósofo 1 pega garfo 0
Filósofo 1 devolve garfo 1
Filósofo 1 devolve garfo 0
```

Figura 25: *Trace* gerado incorretamente devido ao problema de atomicidade.

Para tentar resolver esse problema foi proposto o uso de *Transactional Memory* [39]. Essa é uma técnica utilizada para coordenar processos concorrentes em um nível de abstração muito mais alto que a tradicional sincronização através da combinação de *locks* e condicionais [40]. *Transactional memory* se baseia no mecanismo de transações, comumente utilizado nos bancos de dados. A idéia é que um bloco de comandos pode ser encapsulado por um bloco atômico, que garante que o código irá executar de forma atômica em relação a todos os outros blocos atômicos [41]. Um bloco atômico executa realizando um *log* de transações que registra toda leitura e escrita que é feita na memória. Quando um bloco termina, ele valida seu *log*, para checar se possui uma visão consistente da memória, e então submete suas mudanças para a memória. Se a validação falhar - porque a memória foi alterada por outro processo durante a execução do bloco - o bloco é executado novamente do início.

A técnica de *transactional memory* seria utilizada no trabalho para fazer com que a execução do aspecto e das instruções marcadas fossem feitas de forma atômica. Entre-

tanto, devido à grande dificuldade encontrada para implementar essa técnica, seu uso foi descartado. Por conta também da complexidade que envolve esse assunto, sua abordagem foi feita de forma superficial neste trabalho, mas abordagens mais completas podem ser encontradas em [41, 40, 42, 39] e em [43] é encontrada uma implementação em Java utilizada para a replicação de objetos distribuídos.

A solução mais simples encontrada foi a inclusão de uma regra para a inserção das marcações: as marcações devem ser inseridas logo após a instrução que realmente modifica o estado do sistema e essa instrução deve estar dentro de um método *synchronized*. Dessa forma, as execuções da instrução e do aspecto são feitas de forma atômica - já que o *synchronized* garante isso - o que resolve o problema, mas a técnica perde em versatilidade.

Essa perda de versatilidade se deve ao fato de que a partir do momento que as marcações devem ser inseridas dentro de métodos *synchronized*, a técnica não poderá ser utilizada em sistemas cujas mudanças de estado do modelo sejam ocasionadas por bibliotecas externas. Isso quer dizer que os métodos que receberão as marcações devem ser implementados pelo desenvolvedor, já que não se teria acesso para inserir marcações em métodos de bibliotecas externas ao sistema.

Um outro fator que reduz a versatilidade da técnica é que passa a ser necessário o compartilhamento de um mesmo objeto entre os processos participantes, para que a técnica possa ser aplicada. É necessário, por exemplo, que filósofos compartilhem os mesmos objetos garfo ou que carros compartilhem o mesmo objeto estacionamento. Isso ocorre porque o monitor inserido pela palavra reservada *synchronized* pertence ao objeto, e portanto um método *synchronized* de outro objeto pode executar ao mesmo tempo que algum método deste objeto, mas métodos *synchronized* deste objeto não podem executar ao mesmo tempo, o que faz com que exista atomicidade.

Para se poder avaliar melhor o impacto da adaptação que a técnica sofreu com o uso de *synchronized* sobre a classe de problemas concorrentes, foi proposto que se trabalhasse com a análise de um número maior de problemas. Isso pode ser notado nos problemas que foram discutidos nas seções anteriores, nos quais não foram encontrados empecilhos para o uso da técnica.

4 Estudo de Caso

O problema escolhido para este estudo de caso refere-se a divisão de tarefas entre processos. As entidades principais envolvidas nesse problema são *Supervisor*, *Worker* e *TupleSpace*. O objetivo é que essas entidades trabalhem em conjunto para calcular a área aproximada, no gráfico, abaixo da curva de uma função específica, sendo que essa curva está definida entre os pontos 0 (zero) e 1 (um), como mostra a Figura 26.

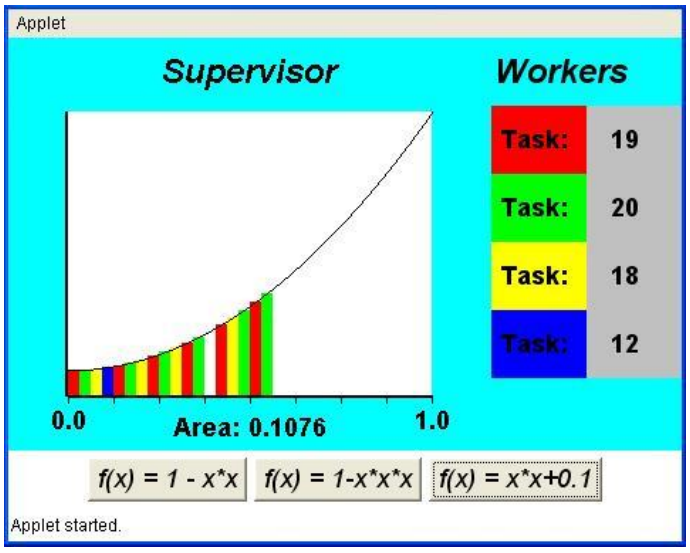

Figura 26: Representação gráfica do cálculo da integral definida de uma função.

Existem quatro trabalhadores e um supervisor. O supervisor divide a tarefa principal - cálculo da área total - em 32 (trinta e duas) partes e as disponibiliza para os trabalhadores. Estes, por sua vez, ficam responsáveis por fazer o cálculo da área dessas partes menores e disponibilizar o resultado para que seja possível somar cada uma das partes e obter o valor da área total. A classe *TupleSpace* funciona como uma espécie de *buffer*, que é compartilhado entre o supervisor e os trabalhadores, e é onde são disponibilizadas as tarefas e os resultados.

Visto que os *Workers* estão em *loop* infinito, sempre pegando a próxima tarefa, é necessário se utilizar um mecanismo para que eles saiam do *loop* e terminem sua execução.

Por isso, depois que o supervisor já tem o resultado de cada uma das tarefas, ele submete uma outra tarefa com o valor *stop*, para que os trabalhadores parem de trabalhar. Quando um trabalhador recebe uma tarefa com o valor *stop*, ele submete a tarefa de volta - para que o próximo trabalhador também possa acessá-la - e pára sua execução. Depois que todos os trabalhadores receberam a tarefa *stop*, o programa termina.

A estrutura geral do sistema é mostrada na Figura 27. A classe principal é *SupervisorWorker*, que possui um supervisor, quatro trabalhadores e um *buffer* - *TupleSpace*. O *buffer* é acessado tanto pelo supervisor quanto pelos trabalhadores, e para que não haja inconsistência dos dados que são colocados e retirados a todo momento, é preciso que haja algum tipo de sincronização. Essa sincronização é feita pelo próprio *TupleSpace*, evitando que mais de um processo tenha acesso a seus dados ao mesmo tempo.

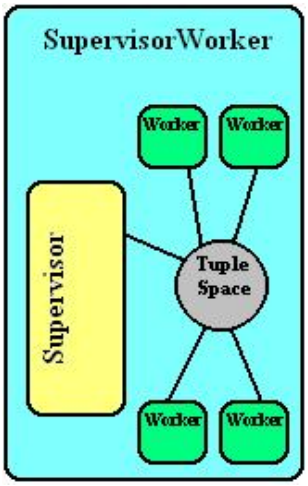

Figura 27: Estrutura geral do sistema.

Existem alguns pontos importantes sobre o comportamento de cada processo:

- Um *Worker* só pode pegar tarefas (*task*) que já tenham sido disponibilizadas;
- Um *Worker* só pode pegar uma *task* por vez;
- Um *Worker* não pode pegar uma *task* que já tenha sido pega por outro *Worker* (exceto a *task stop*);
- O *Supervisor* não pode pegar o resultado (*result*) de uma *task* sem que todas as *tasks* já tenham sido disponibilizadas;
- O *Supervisor* não pode disponibilizar a *task stop* sem que antes tenha pego o *result* de todas as *tasks*.

A Figura 28 mostra o modelo em rede de Petri para este problema da divisão de tarefas. Observe que este modelo é simples, o que facilita bastante o acompanhamento e estudo do comportamento do sistema e suas propriedades, diferentemente do que ocorreria se esse acompanhamento fosse feito diretamente no código. Para se ter uma idéia da complexidade de um acompanhamento do sistema feito diretamente a partir do código, foi inserido no Anexo B o código completo do sistema.

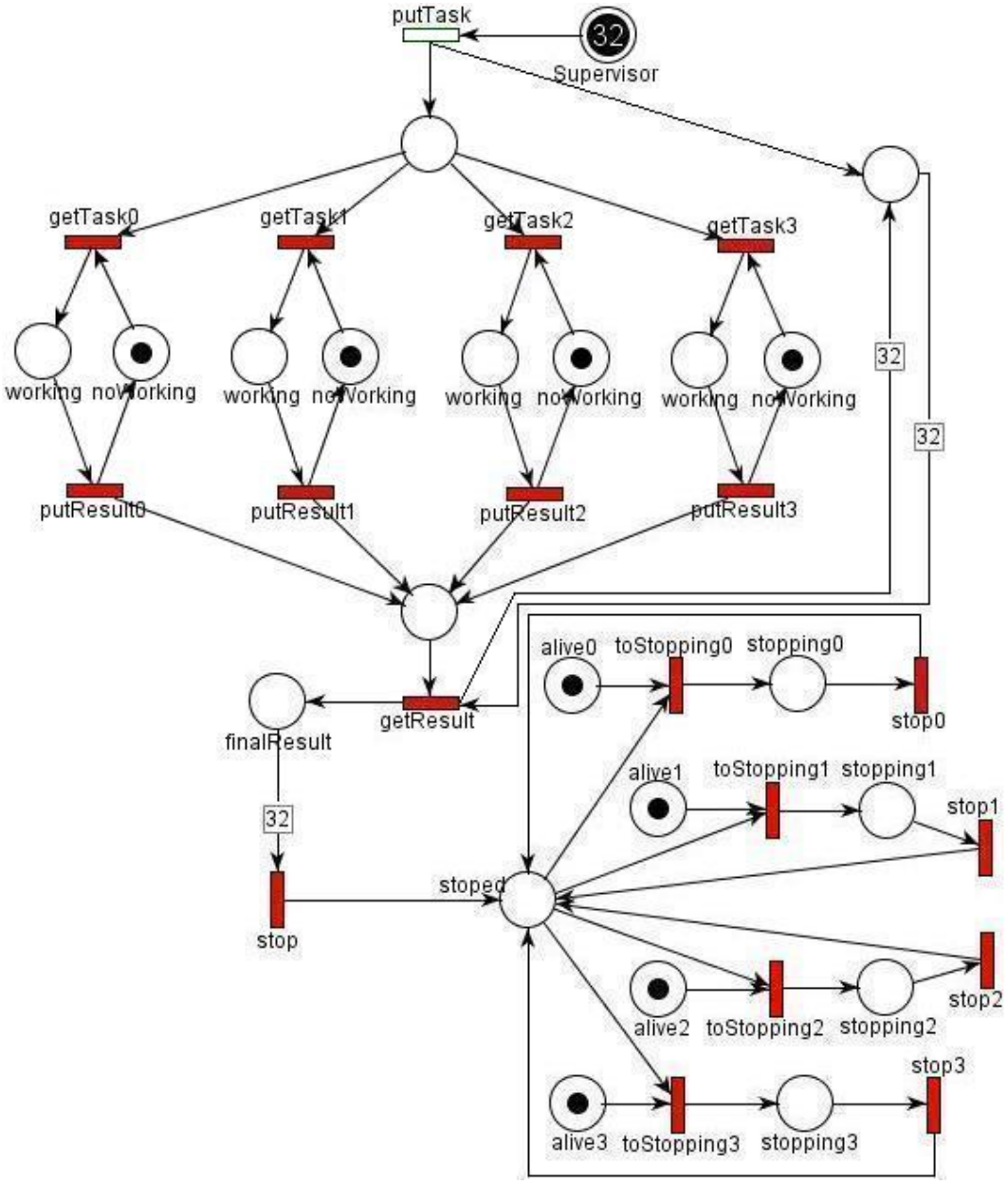

Figura 28: Modelo em rede de Petri para o problema da divisão de tarefas.

Na figura é possível observar que em *Supervisor* existem 32 (trinta e duas) fichas, que é exatamente o número de partes em que foi dividida a tarefa-objetivo para serem distribuídas entre os trabalhadores. Nota-se também que em cada lugar *noWorking* existe

uma ficha, para representar que nenhum *Worker* está trabalhando nesse momento e em cada lugar *alive* também existe uma ficha, indicando que embora os trabalhadores não estejam realizando nenhuma tarefa eles estão em funcionamento.

Ao disparar *putTask*, as transições *getTask* são habilitadas, possibilitando que os trabalhadores peguem uma tarefa para executá-la. Nesse momento, a ficha que estava em *noWorking* é destruída e é criada uma em *working*, indicando que agora o trabalhador está desenvolvendo uma atividade. Ao terminar de executar a tarefa, a transição *putResult* pode ser disparada, mas *getResult* só estará habilitada caso o supervisor já tenha disponibilizado todas as 32 *tasks*. Em *finalResult* são acumuladas as fichas que representam os resultados que já estão de posse do supervisor e a transição *stop* só será habilitada depois que forem reunidos os resultados de todas as 32 *tasks*.

Depois que *stop* é disparada - momento em que o supervisor submete a *task stop* - qualquer um dos *Workers* estará habilitado a disparar *toStopping*. Entretanto, depois que o primeiro *Worker* pega essa tarefa o próximo *Worker* só poderá pegá-la depois que o primeiro disparar sua própria transição *stop* - que é quando ele submete novamente a task *stop* e pára sua execução. Depois que todos os trabalhadores dispararem suas respectivas transições *stop*, não haverá mais fichas nos lugares *alive* - indicando que todos os processos *Worker* foram interrompidos - e não haverá transições habilitadas, indicando que o programa chegou ao fim de sua execução.

O Código 4.1 mostra o método *run()* da classe *Supervisor*. Ele inicia submetendo as 32 *tasks* ao *TupleSpace* (variável *bag*) através do método *bag.out()* e depois coleta os 32 resultados das *tasks* através do método *bag.in()*. Depois de terminada a coleta dos *results*, ele submete a *task* com valor *stop* e termina sua execução.

```
1  public void run () {
2     try {
3        // output tasks to tuplespace
4        for (int i=0; i<SupervisorCanvas.Nslice; ++i)
5           bag.out("task",new Integer(i), id);
6
7        // collect results
8        for (int i=0; i<display.Nslice; ++i) {
9           Result r = (Result)bag.in("result", id);
10          display.setSlice(r.task,r.area,r.worker);
11       }
12       // output stop tuple
13       bag.out("task",stop, id);
14    } catch (InterruptedException e){ }
15 }
```

Código 4.1: Método *run()* da classe *Supervisor*.

O Código 4.2 mostra o método *run()* da classe *Worker*. Ele inicia pegando uma tarefa

através do método *bag.in()*. Em seguida é verificado se o valor da *task* recebida é *stop* (*stop* tem valor inteiro igual a -1). Caso seja, a *task stop* será submetida pelo *Worker* e ele irá parar sua execução, senão a tarefa é realizada (a área é calculada), o resultado é submetido e uma nova tarefa é pega no início da próxima iteração.

```
1  public void run () {
2     double deltaX = 1.0/SupervisorCanvas.Nslice;
3     try {
4        while(true){
5           // get new task from tuple space
6           Integer task = (Integer)bag.in("task", id);
7           int slice = task.intValue();
8           if (slice <0) {  // stop if negative
9              bag.out("task",task, id);
10             break;
11          }
12          display.setTask(slice);
13          sleep(processingTime);
14          double area
15                = deltaX*func.fn(deltaX*((double)slice)+deltaX/2.0);
16          // output result to tuple space
17          bag.out( "result",
18                new Result(slice,area,display.worker), id);
19       }
20    } catch (InterruptedException e){}
21 }
```

Código 4.2: Método *run()* da classe *Worker*.

O Código 4.3 mostra os métodos da classe *TupleSpace*, métodos esses que são invocados por *Supervisor* e por *Worker* para terem acesso aos dados compartilhados entre eles.

A classe *TupleSpace* possui um *HashMap* como atributo (*tuples*), que tem *String* como chave e *ArrayList* como valor. Passando a chave *"task"* como parâmetro para o método *tuples.get()* é retornado o *ArrayList* que guarda as tarefas que estão pendentes. Passando a chave *"result"* para o método *tuples.get()* é retornado o *ArrayList* que guarda os resultados que o supervisor ainda não pegou.

```
1  public synchronized void out (String tag, Object data, int id) {
2     ArrayList v = tuples.get(tag);
3     if (v==null) {
4        v=new ArrayList();
5        tuples.put(tag,v);
6     }
7     v.add(data);
8     if( tag.equals("task") && id==4 && ((Integer)data).intValue()==-1 ){
9        Transition.fireTransition("SUP: stop");
10    } else if( tag.equals("task") && id==4 && ((Integer)data).intValue()!=-1
          ){
11       Transition.fireTransition("SUP: putTask");
12    } else if( tag.equals("task") && id!=4){
13       Transition.fireTransition("WORK: stop" + id);
14    } else if( tag.equals("result") ){
15       Transition.fireTransition("WORK: putResult" + id);
16    }
```

```
17      notifyAll();
18  }
19
20  private Object get(String tag) {
21      ArrayList v = tuples.get(tag);
22      if (v==null) return null;
23      if (v.size()==0) return null;
24      Object o = v.get(0);
25      v.remove(0);
26      return o;
27  }
28
29  // extracts object with tag from tuple space, blocks if not available
30  public synchronized Object in (String tag, int id) throws
        InterruptedException {
31      Object o;
32      while ( ( o = get(tag) ) == null ) wait();
33      if( tag.equals("task") && ((Integer)o).intValue()==-1 ){
34          Transition.fireTransition("WORK: toStopping" + id);
35      } else if( tag.equals("task") && ((Integer)o).intValue()!=-1 ){
36          Transition.fireTransition("WORK: getTask" + id);
37      } else if( tag.equals("result") ){
38          Transition.fireTransition("SUP: getResult");
39      }
40      return o;
41  }
```

Código 4.3: Métodos da classe *TupleSpace*.

No método *out()* é pego o *ArrayList* cuja chave é *tag* e *data* é adicionado à lista. O fato de adicionar um dado a uma lista de *TupleSpace* pode ocasionar o disparo de diferentes transições no modelo; a transição que será disparada vai depender de quem está adicionando o dado (*Supervisor* ou *Worker*), do tipo de dado (*task* ou *result*) e se a *task* é do tipo *stop*. Caso o tipo de dado seja uma *task* do tipo *stop* enviada pelo supervisor, a transição a ser disparada será a transição *stop* do modelo em rede de Petri da Figura 28. Se a *task* enviada pelo supervisor não for *stop*, a transição a ser disparada será *putTask*. No caso de um *Worker* enviar uma *task*, ela necessariamente será *stop*; nesse caso, a transição a ser disparada será o *stop* desse Worker específico (uma das transições de *stop0* a *stop3*, a depender de qual trabalhador submeteu a tarefa). No caso do dado enviado ser do tipo *result* (*tag = "result"*), a transição a ser disparada é o *putResult* do Worker específico que submeteu o dado (de *putResult0* a *putResult3*).

No método *in()* é verificado se existe algum elemento na lista solicitada. Caso não haja nenhum elemento, o processo que invocou o método fica bloqueado (*wait()*) até que esteja disponível algum elemento, caso contrário o primeiro elemento da lista solicitada é removido da lista e retornado. Ao se remover e retornar um elemento da lista é gerada uma mudança de estado no sistema que é visível no modelo e isso implica em disparo de transição no modelo. A depender da lista que está sendo solicitada (*task* ou *result*) e

do tipo de *task* (*task* comum ou *stop*), diferentes transições podem ser disparadas. Se o elemento que será retornado pertence à lista de *task* e é do tipo *stop*, a transição que será disparada é a toStopping do *Worker* que invocou o método (*toStopping0* a *toStopping3*). Caso o elemento seja uma *task* comum, será disparada a transição *getTask* do Worker que invocou o método (*getTask0* a *getTask3*). Caso a lista solicitada seja de *result*, a transição a ser disparada é *getResult*.

Com as marcações já inseridas em seus locais próprios - levando em consideração o que foi descrito a respeito dos métodos *out()* e *in()* - o programa foi posto em execução e o *trace* foi gerado. A conformidade entre o código do programa e seu modelo pode ser verificada confrontando o *trace* (ver Figura 29 do Anexo C) e a rede da Figura 28.

Para observações mais detalhadas no código-fonte do sistema, ele pode ser visto completo no Anexo B em sua forma original, sem a inclusão das marcações que são necessárias à aplicação da técnica.

5 Considerações Finais

O desenvolvimento de uma técnica que confronte o código de um sistema em execução com seu modelo em rede de Petri é bastante útil para se estudar o seu comportamento. Através da utilização dessa técnica é possível verificar se o sistema está em conformidade com seu modelo e, a partir daí, qualquer avaliação comportamental do sistema poderá ser feita com base no modelo, já que todo o seu comportamento pode ser descrito por ele. Isso possibilita um controle direto do sistema em um nível de abstração mais alto, o que facilita a análise de sistemas muito complexos, como é o caso dos sistemas concorrentes. Além disso, existem as vantagens de poder contar com o auxílio de ferramentas de análise de redes de Petri e poder fazer análises formais acerca do modelo, tendo uma visão muito mais completa - e matematicamente fundamentada - do comportamento do sistema.

A utilização dessa técnica também pode contribuir para a solução do problema de sincronização. Como o código está sempre sendo confrontado com o modelo e o desenvolvedor tem participação ativa nesse processo de verificação, a atualização de um modelo se torna mais rápida e fácil de ser realizada.

Por fim, a técnica também permite minimizar o custo de manutenção dos modelos, o que torna viável a adoção da técnica em processos ágeis de desenvolvimento. Assim, tais processos passam a contar com um recurso que utiliza análise formal para estudar o comportamento dos sistemas, o que agrega mais qualidade ao software desenvolvido.

Para trabalhos futuros existe a possibilidade de se automatizar o processo de disparo das transições. É possível construir uma ferramenta ou um *plugin* para que a partir do código as transições possam ser disparadas diretamente, sem a necessidade do processo intermediário de se gerar os *traces* e fazer o disparo das transições de forma manual. Dessa forma o aspecto teria o papel de disparar as transições, ao invés de informar quais transições devem ser disparadas e em que ordem.

Uma outra possibilidade de otimização da técnica é fazer uma adaptação do processo para se utilizar redes de Petri de alto nível. Esse tipo de rede é uma extensão das redes

de Petri clássicas, que introduz o conceito de fichas individuais, permitindo que as fichas (recursos) carreguem diferentes tipos de informação [26]. Nas redes de Petri de alto nível, além do número de recursos em cada lugar, as informações carregadas por esses recursos também determinam o estado da rede. Sendo assim, ao se disparar uma transição, não apenas o número de fichas nos lugares de entrada e saída devem ser levados em conta, mas também o tipo de informação que elas levam [26].

A adoção das redes de alto nível daria uma maior flexibilidade à técnica, já que elas aumentam o poder de modelagem no momento em que fornecem uma maneira mais simples e compreensiva de se construir modelos [26], fazendo com que se trabalhe com os modelos em um nível mais abstrato do que com as redes clássicas.

Referências

[1] MARRANGHELLO, N. *Redes de Petri: Conceitos e Aplicações*. [S.l.], 2005.

[2] SILBERSCHATZ, A.; GAGNE, G.; GALVIN, P. B. *Operating System Concepts*. [S.l.]: Wiley, 2002.

[3] BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. [S.l.]: Addison-Wesley, 1998.

[4] GARLAN, D. Software architecture: a roadmap. In: *International Conference on Software Engineering*. [S.l.: s.n.], 2000.

[5] NETZER, R. H. B.; MILLER, B. P. What are race conditions?: Some issues and formalizations. v. 1, n. 1, p. 74–88, mar. 1992. ISSN 1057-4514.

[6] COLLOFELLO, J. S. *Introduction to Software Verification and Validation*. Dezembro 1988.

[7] WAGNER, S. Towards software quality economics for defect-detection techniques. In: *SEW '05: Proceedings of the 29th Annual IEEE/NASA on Software Engineering Workshop*. Washington, DC, USA: IEEE Computer Society, 2005. p. 265–274. ISBN 0-7695-2306-4.

[8] SCHMITT, W. *Automated Unit Testing of Embedded ARM Applications*. [S.l.], 2004.

[9] KETTENIS, J.; BLOK, R. de. Getting started with unit-testing. December 2004.

[10] WELCOME to JUnit.org! | JUnit.org. Http://www.junit.org. Acesso em: 20 out. 2007.

[11] BEIZER, B. *Software Testing Techniques*. Second. [S.l.]: Van Nostrand Reinhold Company, 1990.

[12] JR., E. M. C.; GRUMBERG, O.; PELED, D. *Model Checking*. [S.l.]: The MIT Press, 2000.

[13] MANIFESTO for Agile Software Development. Http://agilemanifesto.org/. Acesso em: 12 ago. 2007.

[14] PELED, D. Combining partial order reductions with on-the-fly model-checking. In: *Proceedings of the 6th International Conference on Computer Aided Verification (CAV'94)*. [S.l.]: Springer-Verlag, 1994. (Lecture Notes in Computer Science, v. 818), p. 377–390.

[15] GUÉHÉNEUC, Y.-G. et al. *Bridging the Gap Between Modeling and Programming Languages*. [S.l.], July 2002. 56 p.

[16] YAN, H. et al. Discotect: A system for discovering architectures from running systems. In: *ICSE'04*. USA: IEEE, 2004. p. 470–479.

[17] HOARE, C. A. R. *Communicating Sequential Processes*. [S.l.]: Prentice Hall, 1985.

[18] MáRIO, M. F. *Métodos formais na concepção e desenvolvimento de sistemas interactivos*. Tese (Doutorado) — Universidade do Minho, 1995.

[19] ASSIS, A. M. L. de. *Modelagem Executável de Sistemas Distribuídos em Java*. Dissertação (Mestrado) — UNIVERSIDADE FEDERAL DE CAMPINA GRANDE, Março 2006.

[20] GADHWALA, M. B. A. G. S.; SHYAMASUNDAR, R. K. *MSC+: A generalized hierarchical message sequence charts*. 2000.

[21] Z.120, I.-T. recommendation. Message sequence charts (msc'96). In: ITU-T. *ITU Telecommunication Standardization Sector*. [S.l.], 1996.

[22] MAUW, S. The formalization of message sequence charts. In: *Computer Networks and ISDN Systems*. [S.l.: s.n.], 1996.

[23] ETESSAMI, R. A. K.; YANNAKAKIS, M. Inference of message sequence charts. In: *International Conference on Software Engineering*. [S.l.: s.n.], 2000.

[24] MURATA, T. Petri nets: Properties, analysis and applications. In: *Proceedings of the IEEE*. [S.l.: s.n.], 1989.

[25] J.L.PETERSON. *Petri Net Theory and Modeling of Systems*. [S.l.]: Prentice-Hall, N.J., 1981.

[26] BRAGA, J. D. M.; PERKUSICH, A.; FIGUEIREDO, J. C. A. de. *Redes de Petri*. [S.l.], 1999.

[27] PETERSON, J. L. Petri nets. In: *Computing Surveys*. [S.l.: s.n.], 1977. v. 9.

[28] RENEW - The Reference Net Workshop. Http://www.renew.de/. Acesso em: 08 jul. 2007.

[29] JARP Project. Http://jarp.sourceforge.net/. Acesso em: 15 ago. 2007.

[30] HAUGEN, O. Comparing uml 2.0 interactions and msc-2000. In: *System Analysis and Modeling: 4th International SDL and MSC Workshop, SAM 2004*. [S.l.: s.n.], 2004.

[31] MEYER, B. *Object-Oriented Software Construction*. Second. [S.l.]: Prentice-Hall, 1997.

[32] BOOCH, G. *Object-Oriented Analysis and Design With Applications*. 2nd. ed. Menlo Park, CA: Benjamin/Cummings, 1994.

[33] OSSHER, H.; TARR, P. Using subject-oriented programming to overcome common problems in object-oriented software development/evolution. In: *International Conference on Software Engineering*. [S.l.]: ACM, 1999.

[34] THE AspectJ Project. Http://www.eclipse.org/aspectj/. Acesso em: 09 ago. 2007.

[35] SOARES, S.; BORBA, P. *AspectJ Programação orientada a aspectos em Java.* [S.l.], 2004.

[36] KISELEV, I. *Aspect-Oriented Programming with AspectJ.* Indianapolis: Sams Publishing, 2003.

[37] KICZALES, G. et al. Aspect-oriented programming. In: *Proceedings European Conference on Object-Oriented Programming.* Berlin, Heidelberg, and New York: Springer-Verlag, 1997. v. 1241, p. 220–242.

[38] KICZALES, G. et al. An overview of aspectj. In: *ECOOP '01 Conference Proceedings.* [S.l.: s.n.], 2001.

[39] HAMMOND, L. et al. *Transactional Memory Coherence and Consistency.* [S.l.], 2004.

[40] DISCOLO, A. et al. Lock free data structures using stm in haskell. In: . [S.l.: s.n.], 2006.

[41] HARRIS, T. et al. Composable memory transactions. In: *Principles and Practice of Parallel Programming (PPOPP).* [S.l.: s.n.], 2005.

[42] HARRIS, T.; JONES, S. P. *Transactional memory with data invariants.* [S.l.], 2006.

[43] XSTM - Object Replication for Java, .NET and GWT. Http://www.xstm.net/. Acesso em: 19 fev. 2008.

ANEXO A – Código-Fonte do aspecto e da classe Transition

```
1  package concurrency.aspect;
2
3  import concurrency.petriNet.Transition;
4
5  public aspect AspectTransition {
6
7      public pointcut fireTransition(String transition) :
8          execution (public static void Transition.fireTransition(String)) &&
               args(transition);
9
10     before(String transition) : fireTransition(transition){
11         System.out.println(transition);
12     }
13 }
```

Código A.1: Código do aspecto utilizado nos problemas.

```
1  package concurrency.petriNet;
2
3  public class Transition {
4      public static void fireTransition(String transitionName){
5
6      }
7
8  }
```

Código A.2: Código da classe *Transition*.

ANEXO B – Código-Fonte do Programa da Divisão de Tarefas

```
1  package concurrency.connector;
2
3  public interface TupleSpace {
4
5    // deposits data in tuple space
6    public void out (String tag, Object data, int id);
7
8    // extracts object with tag from tuple space, blocks if not available
9    public Object in (String tag, int id) throws InterruptedException;
10
11 }
```

Código B.1: Código da interface *TupleSpace*.

```
1  package concurrency.connector;
2
3  import java.util.ArrayList;
4  import java.util.HashMap;
5  import concurrency.petriNet.Transition;
6
7  public class TupleSpaceImpl implements TupleSpace {
8    private HashMap<String,ArrayList> tuples = new HashMap<String, ArrayList
       >();
9
10   // deposits data in tuple space
11   public synchronized void out (String tag, Object data) {
12     ArrayList v = tuples.get(tag);
13     if (v==null) {
14         v=new ArrayList();
15         tuples.put(tag,v);
16     }
17     v.add(data);
18     notifyAll();
19   }
20
21   private Object get(String tag) {
22     ArrayList v = tuples.get(tag);
23     if (v==null) return null;
24     if (v.size()==0) return null;
25     Object o = v.get(0);
26     v.remove(0);
27     return o;
28   }
29
30   // extracts object with tag from tuple space, blocks if not available
```

```
31    public synchronized Object in (String tag) throws InterruptedException {
32      Object o;
33      while ( ( o = get(tag) ) == null ) wait();
34      return o;
35    }
36
37  }
```

Código B.2: Código da classe *TupleSpaceImpl*.

```
1  package concurrency.supervisor;
2
3  interface Function {
4    double fn(double x);
5  }
6
7  class OneMinusXsquared implements Function {
8    public double fn (double x) {return 1-x*x;}
9  }
10
11 class OneMinusXcubed implements Function {
12   public double fn (double x) {return 1-x*x*x;}
13 }
14
15 class XsquaredPlusPoint1 implements Function {
16   public double fn (double x) {return x*x+0.1;}
17 }
```

Código B.3: Código da classe *Function*.

```
1  package concurrency.supervisor;
2
3  import concurrency.connector.*;
4
5  import java.awt.*;
6
7  class Result {
8    int task;
9    Color worker;
10   double area;
11   Result(int s, double a, Color c)
12     {task =s; worker=c; area=a;}
13 }
14
15 class Supervisor extends Thread {
16   SupervisorCanvas display;
17   TupleSpace bag;
18   Integer stop = new Integer(-1);
19   int id;
20
21   Supervisor(SupervisorCanvas d, TupleSpace b, int id)
22     { display = d; bag = b; this.id = id;}
23
24   public void run () {
25     try {
26       // output tasks to tuplespace
27       for (int i=0; i<SupervisorCanvas.Nslice; ++i)
28         bag.out("task",new Integer(i), id);
29
30       // collect results
31       for (int i=0; i<display.Nslice; ++i) {
32         Result r = (Result)bag.in("result", id);
```

```
33          display.setSlice(r.task,r.area,r.worker);
34        }
35        // output stop tuple
36        bag.out("task",stop, id);
37      } catch (InterruptedException e){}
38    }
39  }
```

Código B.4: Código da classe *Supervisor*.

```
1  package concurrency.supervisor;
2
3  import java.awt.*;
4  import java.applet.*;
5
6   class SupervisorCanvas extends Canvas {
7      String title_;
8      Font f1 = new Font("Helvetica",Font.ITALIC+Font.BOLD,24);
9      Font f2 = new Font("Helvetica",Font.BOLD,18);
10     final static int Nslice = 32; // number of curve slices
11     boolean drawSlice[] = new boolean[Nslice];
12     Color colorSlice [] = new Color[Nslice];
13     double area = 0.0;
14     Function func = null;
15     private final int Xmax = 256;
16     private final int Ymax = 200;
17     private final int delta = Xmax/Nslice;
18
19     public SupervisorCanvas(String title, Color c) {
20         super();
21         setSize(Xmax+50,Ymax+100);
22         title_=title;
23         setBackground(c);
24         for (int i=0; i<Nslice; i++) drawSlice[i]=false;
25         }
26
27     //display rectangle i with color c, add a to area field
28     synchronized void setSlice(int i,double a, Color c) {
29         drawSlice[i]=true;
30         colorSlice[i]=c;
31         area+=a;
32         repaint();
33     }
34
35     //rest display to clear rectangles and draw curve for f
36     synchronized void reset(Function f) {
37         func = f;
38         for (int i=0; i<Nslice; i++) drawSlice[i]=false;
39         area=0.0;
40         repaint();
41     }
42
43     public void paint(Graphics g) {
44         update(g);
45     }
46
47     Image offscreen;
48     Dimension offscreensize;
49     Graphics offgraphics;
50
51     public synchronized void update(Graphics g){
52         Dimension d = getSize();
```

```
53              if ((offscreen == null) || (d.width != offscreensize.width)
54                                      || (d.height != offscreensize.height))
                                        {
55                  offscreen = createImage(d.width, d.height);
56                  offscreensize = d;
57                  offgraphics = offscreen.getGraphics();
58                  offgraphics.setFont(getFont());
59              }
60
61              offgraphics.setColor(getBackground());
62              offgraphics.fillRect(0, 0, d.width, d.height);
63
64           // Display the title
65          offgraphics.setColor(Color.black);
66          offgraphics.setFont(f1);
67          FontMetrics fm = offgraphics.getFontMetrics();
68          int w = fm.stringWidth(title_);
69          int h = fm.getHeight();
70          int x = (getSize().width - w)/2;
71          int y = h;
72          offgraphics.drawString(title_, x, y);
73          x = (getSize().width - Xmax)/2;
74          drawFunction(offgraphics,x,y+20);
75          offgraphics.setColor(Color.black);
76          offgraphics.setFont(f2);
77          fm = offgraphics.getFontMetrics();
78          String as = "Area: "+area;
79          if (as.length()>12)
80              as = as.substring(0,12);
81          w = fm.stringWidth(as);
82          h = fm.getHeight();
83          int x1 = (getSize().width - w)/2;
84          int y1 = y+Ymax+h;
85          offgraphics.drawString(as, x1, y1+25);
86          offgraphics.drawString("0.0",x-fm.stringWidth("0.0")/2,y1+20);
87          offgraphics.drawString("1.0",x+Xmax-fm.stringWidth("1.0")/2,y1+20);
88          g.drawImage(offscreen, 0, 0, null);
89      }
90
91      private void drawFunction(Graphics g, int X, int Y) {
92          g.setColor(Color.white);
93          g.fillRect(X,Y,Xmax,Ymax);
94
95          //draw ruler
96          g.setColor(Color.black);
97          g.drawLine(X-1,Y,X-1,Y+Ymax);
98          g.drawLine(X-2,Y,X-2,Y+Ymax);
99          g.drawLine(X,Y+Ymax,X+Xmax,Y+Ymax);
100         g.drawLine(X,Y+Ymax+1,X+Xmax,Y+Ymax+1);
101         for (int i=X;i<=Xmax+X;i+=32)
102             g.drawLine(i,Y+Ymax,i,Y+Ymax+5);
103
104         if (func==null) return;
105         double scaleY = Ymax/Math.max(func.fn(1.0),func.fn(0.0));
106         double deltaX = 1.0/Xmax;
107
108         //draw computed areas
109         for (int i = 0; i<Nslice ; ++i) {
110           if (drawSlice[i]) {
111               g.setColor(colorSlice[i]);
112               int x = i*delta;
```

```
113                 int Ylen = (int)(func.fn(deltaX*(x+delta/2))*scaleY);
114                 g.fillRect(X+x,Y+Ymax-Ylen,delta,Ylen);
115               }
116             }
117
118             //draw curve
119             g.setColor(Color.black);
120             for (int x = 0; x < Xmax ; x++) {
121                 double dx = deltaX*x;
122                 g.drawLine(X+x, Y+Ymax-(int)(func.fn(dx)*scaleY), X+x + 1, Y+
                        Ymax-(int)(func.fn(dx+deltaX)*scaleY));
123             }
124
125
126         }
127
128 }
```

Código B.5: Código da classe *SupervisorCanvas*.

```
1  package concurrency.supervisor;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.applet.*;
6  import concurrency.connector.*;
7
8  public class SupervisorWorker extends Applet {
9      SupervisorCanvas display;
10     WorkerCanvas red,green,yellow,blue;
11     Button fn1,fn2,fn3;
12     Font buttonFont = new Font("TimesRoman",Font.ITALIC,18);
13     Font titleFont  = new Font("SanSerif",Font.ITALIC+Font.BOLD,24);
14     Thread supervisor, redWork,greenWork, yellowWork,blueWork;
15
16     public void init() {
17         setLayout(new BorderLayout());
18         display = new SupervisorCanvas("Supervisor", Color.cyan);
19         add("Center",display);
20         Panel p0= new Panel();
21
22         p0.add(fn1 = new Button("f(x) = 1 - x*x"));
23         fn1.addActionListener(new ActionListener() {
24             public void actionPerformed(ActionEvent e) {
25               if (ended()) go(new OneMinusXsquared());
26             }
27          });
28
29         p0.add(fn2 = new Button("f(x) = 1-x*x*x"));
30         fn2.addActionListener(new ActionListener() {
31                 public void actionPerformed(ActionEvent e) {
32                   if (ended()) go(new OneMinusXcubed());
33                 }
34              });
35
36         p0.add(fn3 = new Button("f(x) = x*x+0.1"));
37         fn3.addActionListener(new ActionListener() {
38                 public void actionPerformed(ActionEvent e) {
39                   if (ended()) go(new XsquaredPlusPoint1());
40                 }
41              });
42
```

```
43          fn1.setFont(buttonFont);
44          fn2.setFont(buttonFont);
45          fn3.setFont(buttonFont);
46          add("South",p0);
47          Panel p1 = new Panel();
48          p1.setBackground(Color.cyan);
49          p1.setLayout(new GridLayout(6,1));
50          Label w = new Label("Workers");
51          w.setFont(titleFont);
52          p1. add(w);
53          p1. add(red = new WorkerCanvas(Color.red));
54          p1. add(green = new WorkerCanvas(Color.green));
55          p1. add(yellow = new WorkerCanvas(Color.yellow));
56          p1. add(blue = new WorkerCanvas(Color.blue));
57          add("East",p1);
58      }
59
60      int slice=0;
61
62      private void go(Function fn) {
63          display.reset(fn);
64          TupleSpace bag = new TupleSpaceImpl();
65          redWork =  new Worker(red,bag,fn, 0);
66          greenWork = new Worker(green,bag,fn, 1);
67          yellowWork = new Worker(yellow,bag,fn, 2);
68          blueWork = new Worker(blue,bag,fn, 3);
69          supervisor = new Supervisor(display,bag, 4);
70          redWork.start();
71          greenWork.start();
72          yellowWork.start();
73          blueWork.start();
74          supervisor.start();
75       }
76
77      private boolean ended() {
78        if (redWork!=null && redWork.isAlive()) return false;
79        if (greenWork!=null && greenWork.isAlive()) return false;
80        if (yellowWork!=null && yellowWork.isAlive()) return false;
81        if (blueWork!=null && blueWork.isAlive()) return false;
82        if (supervisor!=null && supervisor.isAlive()) return false;
83        return true;
84      }
85
86     public void stop() {
87        if (redWork!=null && redWork.isAlive())
88          {redWork.interrupt();}
89        if (greenWork!=null && greenWork.isAlive())
90          {greenWork.interrupt();}
91        if (yellowWork!=null && yellowWork.isAlive())
92          {yellowWork.interrupt();}
93        if (blueWork!=null && blueWork.isAlive())
94          {blueWork.interrupt();}
95        if (supervisor!=null && supervisor.isAlive())
96          {supervisor.interrupt();}
97      }
98
99  }
```

Código B.6: Código da classe *SupervisorWorker*.

```
1  package concurrency.supervisor;
2
```

```
3  import java.awt.*;
4  import concurrency.connector.*;
5
6  class Worker extends Thread {
7    WorkerCanvas display;
8    Function func;
9    TupleSpace bag;
10   int processingTime = (int)(6000*Math.random());
11   int id;
12
13   Worker(WorkerCanvas d, TupleSpace b, Function f, int id)
14     { display = d; bag = b; func = f; this.id = id;}
15
16   public void run () {
17     double deltaX = 1.0/SupervisorCanvas.Nslice;
18     try {
19       while(true){
20         // get new task from tuple space
21         Integer task = (Integer)bag.in("task", id);
22         int slice = task.intValue();
23         if (slice <0) {  // stop if negative
24             bag.out("task",task, id);
25             break;
26         }
27         display.setTask(slice);
28         sleep(processingTime);
29         double area
30           = deltaX*func.fn(deltaX*((double)slice)+deltaX/2.0);
31         // output result to tuple space
32         bag.out( "result",
33            new Result(slice,area,display.worker), id);
34       }
35     } catch (InterruptedException e){}
36   }
37 }
```

Código B.7: Código da classe *Worker*.

```
1  package concurrency.supervisor;
2
3  import java.awt.*;
4  import java.applet.*;
5
6  class WorkerCanvas extends Panel {
7
8      Label title = new Label(" Task:");
9      Label value = new Label("       ");
10     Font f1 = new Font("Helvetica",Font.BOLD,18);
11     Color worker;
12
13     WorkerCanvas(Color c) {
14         worker = c;
15         setBackground(c);
16         setLayout(new GridLayout(1,2));
17         add(title); title.setFont(f1);
18         add(value); value.setFont(f1); value.setBackground(Color.lightGray)
               ;
19     }
20
21     // display current task number val
22     synchronized void setTask(int val) {
23         value.setText("   "+val+"   ");
```

```
24      }
25
26  }
```

Código B.8: Código da classe *WorkerCanvas*.

ANEXO C – Trace *gerado com a execução do problema da divisão de tarefas*

```
SUP: putTask
WORK: getTask0
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
SUP: putTask
WORK: getTask1
WORK: getTask2
WORK: getTask3
WORK: putResult3
WORK: getTask3
SUP: getResult
WORK: putResult0
WORK: getTask0
SUP: getResult
WORK: putResult3
WORK: getTask3
SUP: getResult
WORK: putResult3
WORK: getTask3
SUP: getResult
WORK: putResult0
WORK: getTask0
SUP: getResult
WORK: putResult3
WORK: getTask3
SUP: getResult
WORK: putResult1
WORK: getTask1
SUP: getResult
WORK: putResult3
WORK: getTask3
SUP: getResult
WORK: putResult0
WORK: getTask0
SUP: getResult
WORK: putResult3
WORK: getTask3
SUP: getResult
WORK: putResult2
WORK: getTask2
SUP: getResult
WORK: putResult3
WORK: getTask3
SUP: getResult
WORK: putResult0
WORK: getTask0
SUP: getResult
WORK: putResult3
WORK: getTask3
SUP: getResult
WORK: putResult0
WORK: getTask0
SUP: getResult
WORK: putResult3
WORK: getTask3
SUP: getResult
WORK: putResult1
SUP: getResult
WORK: getTask1
WORK: putResult3
SUP: getResult
WORK: getTask3
WORK: putResult0
SUP: getResult
WORK: getTask0
WORK: putResult3
SUP: getResult
WORK: getTask3
WORK: putResult3
SUP: getResult
WORK: getTask3
WORK: putResult0
SUP: getResult
WORK: getTask0
WORK: putResult3
SUP: getResult
WORK: getTask3
WORK: putResult2
SUP: getResult
WORK: getTask2
WORK: putResult3
SUP: getResult
WORK: getTask3
WORK: putResult0
SUP: getResult
WORK: getTask0
WORK: putResult1
SUP: getResult
WORK: getTask1
WORK: putResult3
SUP: getResult
WORK: getTask3
WORK: putResult3
SUP: getResult
WORK: putResult0
SUP: getResult
WORK: putResult1
SUP: getResult
WORK: putResult2
SUP: getResult
SUP: stop
WORK: toStopping3
WORK: stop3
WORK: toStopping2
WORK: stop2
WORK: toStopping1
WORK: stop1
WORK: toStopping0
WORK: stop0
```

Figura 29: *Trace* gerado no problema da divisão de tarefas.