ANÁLISE FORMAL DE CÓDIGO JAVA CONCORRENTE

Trabalho de Conclusão de Curso

Engenharia da Computação

Hugo Leonardo C. de O. Lima

Orientador: Prof. M.Sc. Gustavo H. P. Carvalho

Hugo Leonardo C. de O. Lima

ANÁLISE FORMAL DE CÓDIGO JAVA CONCORRENTE

Monografia apresentada como requisito parcial para obtenção do diploma de Bacharel em Engenharia de Computação pela Escola Politécnica de Pernambuco - Universidade de Pernambuco

Orientador:
Prof. M.Sc. Gustavo H. P. Carvalho

UNIVERSIDADE DE PERNAMBUCO
ESCOLA POLITÉCNICA DE PERNAMBUCO
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Recife - PE, Brasil

7 de dezembro de 2011

POLI
ESCOLA POLITÉCNICA
DE PERNAMBUCO

MONOGRAFIA DE FINAL DE CURSO

Avaliação Final (para o presidente da banca)*

No dia 21 de Dezembro de 2011, às 14:00 horas, reuniu-se para deliberar a defesa da monografia de conclusão de curso do discente HUGO LEONARDO CORREIA DE OLIVEIRA LIMA, orientado pelo professor Gustavo Henrique Porto de Carvalho, sob título Análise formal de código java concorrente, a banca composta pelos professores:

Joabe Bezerra de Jesus Júnior

Gustavo Henrique Porto de Carvalho

Após a apresentação da monografia e discussão entre os membros da Banca, a mesma foi considerada:

☒ Aprovada ☐ Aprovada com Restrições* ☐ Reprovada

e foi-lhe atribuída nota: 9,5 (nove e meio)

*(Obrigatório o preenchimento do campo abaixo com comentários para o autor)

O discente terá 7 dias para entrega da versão final da monografia a contar da data deste documento.

____
JOABE BEZERRA DE JESUS JÚNIOR

____
GUSTAVO HENRIQUE PORTO DE CARVALHO

____

* Este documento deverá ser encadernado juntamente com a monografia em versão final.

A todos que nunca desistem.

Agradecimentos

Seria necessário desenvolver um texto do tamanho desta monografia para apontar e agradecer todas as pessoas que estiveram ao meu lado durante todos os anos de minha vida acadêmica. É difícil lembrar de todos. Alguns permaneceram sempre ao meu lado, outros perdi o contato, porém levo as lembranças sempre comigo.

Vou dedicar este parágrafo (trabalho) inteiro somente a uma pessoa. Para a mulher que sempre acreditou em mim, desde os meus primeiros nanosegundos de vida. À minha mãe, eu quero agradecer por ter lutado tanto para que eu não me perdesse durante minha caminhada até o presente momento. Existiram ocasiões onde desviei do caminho, e lá estava dona Djardiere para me colocar de volta nos trilhos corretos da vida. Muito obrigado por ter sido "linha dura"com aquele menino tímido da quarta série, que não gostava de matemática e agora está se formando engenheiro.

Ao meu pai, Gilson, por ser um exemplo de honestidade, luta e perserverança; às minhas queridas avós, Creuza e Iraci, pelo amor elevado ao quadrado oferecido a mim; aos meus dois irmãos, Aline e Victor e a todos os meus familiares.

À minha amada esposa, Conceição, agradeço por ter me apoiado desde os nossos primeiros dias juntos e por me fazer acreditar sempre em mim, independentemente das dificuldades. Obrigado por todo o suporte oferecido ao longo destes nove anos juntos. Este trabalho é meu e seu também, pois sem sua ajuda, talvez não conseguiria.

Aos meus amigos do peito: Breno Maia, Paulo, Bruno Loureiro, Hugo, Stanley, Arthur, Bruno Soares e Breno Gallo, pois sempre me incentivaram.

Ao meu orientador, Gustavo Carvalho, que apesar de nunca ter pago nenhuma disciplina com ele, foi um dos professores mais atenciosos, presentes e pacientes que já tive. Muito obrigado pelo apoio, suporte intelectual e emocional também.

Resumo

Técnicas convencionais como testes e simulações não são as mais indicadas para validar sistemas concorrentes. Vários problemas, como não-determinismo, podem influir nos resultados da computação sem ser detectados. Linguagens de descrição formal para verificação de sistemas oferecem formas mais avançadas para validar sistemas concorrentes. Uma dessas linguagens, CSP (*Communicating Sequential Processes*), é baseada em processos independentes que comunicam enventos entre si, tornando mais fácil a descrição de sistemas concorrentes. Devido ao suporte a multiprocessamento da linguagem Java, este trabalho propõe regras de mapeamento de *bytecodes* Java para CSP#, um dialeto de CSP. Esse mapeamento permite realizar verificações, através da ferramenta PAT (*Process Analysis Toolkit*), que técnicas tradicionais de teste não oferecem. PAT é um *framework* desenvolvido para simular e verificar o comportamento de sistemas concorrentes. Como é uma ferramenta ainda em desenvolvimento, ela possui algumas limitações. Devido a esse fato, foi necessário impor algumas restrições às regras propostas neste trabalho. Apesar das restrições, a ferramenta, aliada ao mapeamento das instruções, conseguiu obter resultados satisfatórios na detecção de problemas fruto da execução concorrente de processos.

Palavras-chave: análise formal, Java, *BtyeCode*, concorrência, CSP.

Abstract

Conventional techniques such as tests and simulations are not the most suitable to validate concurrent systems. Several problems, for instance non-determinism, may influence the results of computations without being detected. Description languages for formal verification of systems offer more advanced ways to validate concurrent systems. One of these languages, CSP (Communicating Sequential Processes), is based on independent processes that communicate events with each other, making easier the description of concurrent systems. Due to the multiprocessing support of Java language, this work proposes mapping rules for translating Java bytecodes to CSP#, a dialect of CSP. This mapping allows to perform checks, using the PAT (Process Analysis Toolkit) tool, which traditional techniques are not able to. PAT is a framework designed to simulate and verify the behavior of concurrent systems. As a work in progress, this tool is still under development and it has some limitations such as poor support for object orientation. Therefore, it was necessary to impose some restrictions on the mapping rules proposed herein. Despite the restrictions, the tool, combined with the instructions mapping, achieved satisfactory results finding problems due to in concurrent processes execution.
Keywords: formal analysis, Java, *BtyeCode*, concurrency, CSP.

Sumário

Lista de Figuras p. x

Lista de Tabelas p. xi

Lista de Abreviaturas e Siglas p. xii

1 Introdução p. 13

1.1 Qualificação do Problema . . . p. 13

1.2 Objetivos . . . p. 14

1.2.1 Objetivos Específicos . . . p. 15

1.3 Resultados e Impactos Esperados . . . p. 15

1.4 Estrutura do Documento . . . p. 15

2 Referencial Teórico p. 17

2.1 Conceitos Chaves . . . p. 17

2.1.1 CSP . . . p. 17

2.1.2 PAT e CSP# . . . p. 20

2.1.3 JVM e o *Bytecode* . . . p. 21

2.2 Trabalhos Relacionados . . . p. 24

2.2.1 Java *Pathfinder* . . . p. 25

2.2.2 JCSP . . . p. 26

3 Método de Pesquisa p. 29

3.1 Qualificação do Método de Pesquisa . . . p. 29

3.2 Etapas do Método de Pesquisa . . . . p. 29
3.2.1 Estudo de CSP . . . . p. 29
3.2.2 Estudo de *Bytecode* Java . . . . p. 30
3.2.3 Estudo de Trabalhos Relacionados . . . . p. 30
3.2.4 Definição do Mapeamento de *Bytecode* Java para CSP# . . . . p. 31
3.2.5 Definição dos Exemplos de Concorrência . . . . p. 31
3.2.6 Transformação dos Exemplos em CSP# . . . . p. 32
3.2.7 Análise dos Resultados . . . . p. 32
3.3 Limitações da Pesquisa . . . . p. 32

4 Resultados p. 35
4.1 Regras de Mapeamento . . . . p. 35
4.1.1 Regras de Mapeamento Geral . . . . p. 35
4.1.2 Regras de Nomenclatura . . . . p. 38
4.1.3 Regras de Mapeamento de Instruções *Bytecode* Java . . . . p. 41
4.2 Aplicação das Regras de Mapeamento . . . . p. 51
4.2.1 Aplicação das Regras para Definição de um Processo . . . . p. 51
4.2.2 Exemplo 1: Produtores e Consumidores . . . . p. 53
4.2.3 Exemplo 2: Lista de Recursos . . . . p. 54

5 Considerações Finais p. 57
5.1 Trabalhos Futuros . . . . p. 57

Referências p. 59

Apêndice A – Tabelas Complementares e Códigos dos Arquivos Fontes p. 61
A.1 Instruções Bytecode . . . . p. 61
A.2 Exemplo 1 . . . . p. 62

A.2.1 Java . . . . p. 62
A.2.2 *Bytecode* . . . . p. 63
A.2.3 CSP# . . . . p. 69
A.3 Exemplo 2 . . . . p. 71
A.3.1 Java . . . . p. 71
A.3.2 *Bytecode* . . . . p. 73
A.3.3 CSP# . . . . p. 81

Lista de Figuras

2.1 Processo CSP#. . . . . p. 20

2.2 Exemplo de código *bytecode*. . . . . p. 23

2.3 Exemplo de frame *bytecode*. . . . . p. 24

2.4 Arquitetura JPF. . . . . p. 26

2.5 Exemplo de Classe JCSP. . . . . p. 27

2.6 Exemplo de canal Muitos-para-muitos. . . . . p. 28

3.1 Exemplo de Falha de Pilhas no PAT . . . . p. 34

4.1 Resultado das Asserções do Exemplo 1. p. 53

4.2 Resultado das Asserções do Exemplo 2. p. 54

4.3 Resultado das Asserções do Exemplo 2 Simulando *Deadlock*. . . . . p. 56

4.4 Alteração do Exemplo 2 para Introduzir *Deadlock*. . . . . p. 56

Lista de Tabelas

2.1 Bytecodes para Manipulação da Pilha de Operandos. . . . . p. 24

4.1 Mapeamento da Instrução de Inserção de Constante na Pilha de Operandos. . p. 42

4.2 Mapeamento das Instruções de Atualização de Atributos. . . . . p. 42

4.3 Mapeamento das Instruções de Finalização de Métodos. . . . . p. 43

4.4 Mapeamento de Empilhamento de Valor de Variável na Pilha. . . . . p. 43

4.5 Mapeamento de Empilhamento de Valor de Atributo de Classe. . . . . p. 44

4.6 Mapeamento de Criação de Nova Lista. . . . . p. 44

4.7 Mapeamento da Instrução de Inserção em uma Lista. . . . . p. 44

4.8 Mapeamento da Instrução de Remoção em uma Lista. . . . . p. 45

4.9 Mapeamento da Instrução de Empilhamento do Tamanho de Lista. . . . . . . p. 45

4.10 Mapeamento da Instrução de Obtenção de Ítem de uma Lista. . . . . p. 45

4.11 Mapeamento de algumas Instruções de Comparação. . . . . p. 47

4.12 Mapeamento de Desvios Incondicionais. . . . . p. 48

4.13 Mapeamento da Chamada de Método com Mapeamento já Definido. . . . . . p. 48

4.14 Mapeamento de Paralelismo entre Processos. . . . . p. 49

4.15 Mapeamento de Instruções não Utilizadas. p. 49

4.16 Mapeamento de Instruções Aritméticas. . . . . p. 50

4.17 Mapeamento de Recebimento de Valores. p. 51

A.1 Tabela com Bytecodes mais Comuns. . . . . p. 61

Lista de Abreviaturas e Siglas

API	*Application Programming Interface)*
CSP	*Communicating Sequential Processes*
CPU	*Central Processing Unit*
JCSP	*Communicating Sequential Processes for Java*
JPF	*PathFinder*
JVM	*Java Virtual Machine*
MJI	*Model Java Interface*
PAT	*Process Analysis Toolkit*

1 *Introdução*

Testes de software possuem um importante papel na criação de ambientes supridos por sistemas de informação seguros, confiáveis, ágeis e estratégicos. A busca pela qualidade dos sistemas se tornou uma atividade indispensável para empresas que fabricam software. Isso visa um melhor desenvolvimento de sistemas, pois evita problemas como retrabalho e insatisfação de clientes, o que acarreta custos indesejados para o fabricante. O teste de software é um importante meio de avaliação do software para determinar sua qualidade. Segundo Miller (1981), o objetivo geral dos testes de software é confirmar a qualidade dos sistemas de software, exercitando-os de maneira sistemática em circunstâncias cuidadosamente controladas.

Técnicas de testes diferem em como é feita a escolha das entradas, e sobre o quanto de conhecimento assume-se sobre o sistema sendo testado em seu ambiente de execução. Isso afeta diretamente como podem ser feitas a definição e verificação correta do comportamento. De acordo com Dijkstra (1970), testes de programas podem ser usados para mostrar a presença de erros, mas nunca mostrar sua ausência. Normalmente, para compensar esse fator, são feitas baterias de testes que verificam exaustivamente um sistema com o maior nível de cobertura possível.

1.1 Qualificação do Problema

À medida que o tamanho e a complexidade dos sitemas atuais tende a crescer, um maior volume de testes também será necessário para verificar se os sistemas apresentam o comportamento desejado. De acordo com Marciniak (1994), normalmente os testes consomem de 40 à 50% dos esforços de desenvolvimento, e demandam mais esforço para sistemas que requerem altos níveis de confiabilidade, tornando-se uma parte significativa do ciclo de desenvolvimento do mesmo. Contudo, este tipo de abordagem nem sempre é adequada para identificar erros em códigos de sistemas concorrentes. Uma vez que erros podem acontecer em função de decisões de outros ambientes, como o escalonamento de processos em uma CPU, é praticamente impossível realizar testes que envolvam todas as situações que o sistema pode abranger.

Portanto, uma dificuldade encontrada na área de testes de software é a análise de processos que executam concorrentemente. Por exemplo, programas na linguagem JAVA, onde *threads* podem ser definidos ou executados em várias entidades ou classes diferentes (OAKS; WONG, 1999).

Por outro lado, a verificação formal de sistemas concorrentes permite analisar de forma exaustiva certas propriedades de um sistema. Propriedades como, por exemplo, a presença de *deadlock*. Principalmente quando o foco está na descrição de processos concorrentes, tem-se a notação CSP (*Communicating Sequential Processes*) (HOARE, 1985). Esta é uma linguagem para descrição formal de componentes constituídos por processos que atuam independentemente e se comunicam entre si por meio de troca de mensagens. Contudo, por ser uma linguagem de especificação formal, não é comum que os desenvolvedores de aplicações a conheçam.

Outra vertente, para verificar o comportamento de sistemas concorrentes, consiste em utilizar outros meios como JCSP (WELCH; MARTIN, 2000) e CSP++ (GARDNER, W.B.; SERRA, M., 2000), que são *frameworks* que permitem a modelagem de sistemas, e ferramentas como Java PathFinder (HAVELUND, K.; PRESSBURGER T., 2000), que permitem implementar modelos para verificar propriedades básicas em sistemas concorrentes de maneira também formal. Contudo, usando linguagens de programação conhecidas, como Java e C++. Dentre estes meios de modelagem, os dois primeiros utilizam conceitos de CSP para realizar as verificações. Apesar desta possibilidade, estas formas não são capazes de realizar análises mais complexas, como prova de propriedades mais complexas através de refinamentos de processos, o que é possível em CSP.

Logo, o problema de pesquisa deste trabalho é: **como gerar automaticamente especificações CSP a partir de código JAVA?** Em particular, a geração deverá considerar o código *bytecode* de um dado código Java, uma vez que o chaveamento entre *threads* se dá neste nível e não no nível de comandos JAVA (LINDHOLM; YELLIN, 1999).

1.2 Objetivos

O objetivo principal deste trabalho é propor um mapeamento entre instruções *bytecode* JAVA para a linguagem CSP# (um dialeto CSP que é processado pela ferramenta *Process Analysis Toolkit* - PAT (LIU; SUN; DONG, 2008)).

1.2.1 Objetivos Específicos

- Definir, a partir do universo de todas as instruções *bytecode* Java, as que se enquadram no grupo de instruções interessantes a este estudo. Não será preciso mapear um conjunto de instruções de um código que imprime um String em uma tela, por exemplo.
- Definir regras de mapeamento entre *bytecodes* e comandos CSP#. Identificando as características e o comportamento inerentes a um subconjunto de instruções, serão definidas as regras para a transformação do trecho de *bytecode* em CSP#. Essas regras serão utilizadas para definir os modelos de transformação entre as linguagens.
- Analisar propriedades básicas, como ausência de *deadlock*, utilizando a ferramenta PAT, em código CSP# obtido a partir de um código escrito em Java.

1.3 Resultados e Impactos Esperados

As regras de mapeamento se caracterizam como o principal objetivo a ser alcançado por esse estudo. É a partir delas que será possível obter os modelos de transformação de um código para o outro. Sem as regras, a tradução é impossível de ser realizada de forma automatizada e isso implicaria na adoção de outro procedimento.

Desta forma, espera-se facilitar a análise de sistemas concorrentes escritos em Java. Em particular, analisar, de maneira formal, mas implícita, se os mesmos possuem a possibilidade de apresentar uma situação de *deadlock*.

1.4 Estrutura do Documento

Este trabalho está dividido em 5 capítulos, incluindo este , que apresentou uma breve introdução ao tema, qualificação do problema, e os objetivos almejados.

- **Capítulo 2 - Referêncial Teórico**: neste capítulo, será abordado todo o embasamento teórico, referente às definições e características de CSP e *bytecode*, necessário para o desenvolvimento deste trabalho. Serão tratados conceitos, trabalhos relacionados e técnicas existentes na área de interesse. O foco deste capítulo será prover uma visão geral sobre estes conceitos, sem apresentar uma definição detalhada dos mesmos, pois esta pode ser encontrada nas referências técnicas referenciadas por este trabalho.

- **Capítulo 3 - Método de Pesquisa**: este capítulo apresentará o processo utilizado no desenvolvimento e obtenção das regras de mapeamento usadas na tradução entre a linguegem fonte (*bytecode*) e a destino (CSP). Este inicia a explanação com a contextualização do problema e do ambiente onde o estudo foi realizado, sendo finalizado com o levantamento dos exemplos que serão considerados.

- **Capítulo 4 - Resultados**: este capítulo apresentará o conjunto de regras de transformação de comandos *bytecode* Java para comandos CSP, bem como exemplos de aplicação prática destas regras em exemplos fictícios.

- **Capítulo 5 - Conclusão**: por fim, este capítulo apresenta as conclusões deste trabalho, analisa os benefícios para a área de interesse, relata as dificuldades encontradas, como também identifica trabalhos futuros.

2 *Referencial Teórico*

O objetivo deste capítulo é apresentar os fundamentos relacionados à comunicação entre processos concorrentes e como podem ser realizadas verificações para garantir que o código desenvolvido funcione sem erros de execução.

2.1 Conceitos Chaves

Os principais conceitos relacionados a este trabalho são: especificação formal de sistemas concorrentes e interpretação do código Java pela *Java Virtual Machine* (JVM). Quanto ao primeiro conceito, será apresentada a notação CSP (HOARE, 1985) que permite especificar de maneira formal processos concorrentes. Quanto ao segundo conceito, será descrita a estrutura da JVM e o formato de alguns dos comandos bytecode Java.

2.1.1 CSP

CSP (HOARE, 1985) ou *Communicating Sequential Processes* é definido como uma linguagem de descrição formal, usada para definir o comportamento e os aspectos de processos que se comunicam entre si. Pode-se estabelecer um paralelo, para um melhor entedimento, com a orientação a objetos. Enquanto linguagens de orientação a objetos se preocupam com definições mais concretas das estruturas de dados e possuem uma visão centrada na execução de comandos imperativos, CSP lida com estruturas de dados de um ponto de vista mais abstrato e possui uma visão centrada na especificação do comportamento dos processos. De acordo com Schneider (1999, p. 1), a estrutura conceitual adotada pelo CSP considera os componentes, ou processos, como entidades independentes e auto-suficientes com interfaces individuais por onde são realizadas as interações com seu meio ambiente.

O elemento central de uma especificação CSP são processos, cujo comportamento é descrito por um conjunto de eventos. Portanto, um processo em CSP é completamente definido pelos eventos que pode comunicar. Eventos são considerados como transições que levam um

determinado processo mudar seu estado ou comportamento. Eles são de natureza atômica, logo, indivisível durante sua ocorrência. Semânticamente, um processo P_1 passa a se comportar com um processo P_2 após a ocorrência de um evento μ como é mostrado na seguinte formulação:

$$P_1 \xrightarrow{\mu} P_2$$

2.1.1.1 Pré-fixo

O operador prefixo permite definir o comportamento de um processo através da realização sequencial de eventos. No exemplo abaixo, tem-se a especificação de um processo que inicialmente comunica o evento a, em seguida o evento b, e, por fim, passa a se comportar como um outro processo (o processo P2).

$$a \rightarrow b \rightarrow P_2$$

2.1.1.2 Alfabeto

A partir da visão anterior, que define o comportamento de um processo como uma sequência ordenada de eventos, pode-se definir o conceito alfabeto de um processo como o conjunto de todos os eventos que os mesmos pode comunicar. Portanto, para o exemplo anterior, o alfabeto do processo seria: {a, b} U $\alpha(P2)$, ou seja, o conjunto dos eventos a e b unido ao alfabeto do processo P_2.

2.1.1.3 Processos Especiais de Finalização

Além dos processos especificados pelo usuário, CSP oferece dois processos básicos que podem ser utilizados pelo usuário: os processos *STOP* e *SKIP*.

O primeiro deles (*STOP*) representa o comportamento mais simples que é a impossibilidade de realizar qualquer evento. Portanto, o alfabeto do processo *STOP* é vazio. Uma das formas de se obter um deadlock em CSP é alcançar um estado no qual o processo passa a se comportar como *STOP*, ou seja, alcança um estado sem eventos de saída.

O segundo deles (*SKIP*) representa a terminação com sucesso. Este processo comunica o evento especial (e interno) *tick* ($\checkmark$) e em seguida passa a se comportar como *STOP*.

$$SKIP \xrightarrow{\checkmark} STOP$$

2.1.1.4 Recursão

Até este ponto foram mostrados exemplos de processos que realizam ações e eventualmente terminam. CSP permite a especificação de processos que não terminam através do uso de recursão. Tomando por exemplo um processo que executa sempre o mesmo evento a, não é possível declará-lo como uma sequência infinita de operadores de pré-fixo $a \rightarrow a \rightarrow a \rightarrow a \rightarrow ...$ Portanto, neste caso, deve-se definir um processo realiza um evento a e em seguida comporta-se como ele mesmo:

$$P = a \rightarrow P$$

Outra possibilidade do uso de recursão é na definição de processos mutuamente recursivos. Por exemplo:

$$P = a \rightarrow Q$$

$$Q = b \rightarrow P$$

Nestes dois exemplos, têm-se processos que nunca param, e alternam entre si.

2.1.1.5 Escolha

CSP fornece formas para decidir entre vários fluxos de execução de um processo. Este controle em CSP é determinado pelas escolhas externa (determinística) e interna (não determinística). Em sistemas concorrentes o não-determinismo é útil para distinguir entre os casos em que o controle sobre a resolução da escolha reside dentro de um processo (interna), ou está fora dele (externa). A distinção é importante em sistemas concorrentes, já que problemas podem surgir se a dois processos for dado controle sobre uma mesma escolha particular (SCHNEIDER, 1999).

Se P e Q são processos, então $P \square Q$ é a escolha externa entre o comportamento de P e Q, onde esta escolha se dará em função do evento oferecido pelo ambiente. Se é um evento que implica em P, então $P \square Q$ se comporta como P, caso contrário se comporta como Q. As escolhas ineterna e externa são idempotentes, comutativas e associativas (HOARE, 1985).

Dado dois processos P e Q, a escolha interna de P e Q é denotada por $P \sqcap Q$, que é um processo que se comporta como P ou como Q. A escolha entre as duas é tomada imediatamente, ou seja, a escolha é realizada de forma abitrária, sem nenhuma influência do ambiente.

2.1.1.6 *Interleaving*

CSP oferece diferentes operadores para representar a execução paralela de dois processos. Um destes operadores é o de *interleaving*, que se caracteriza por dois processos P e Q que intercalam eventos sem sincronização e independentes entre si, tendo em vista nesse momento que a escolha do momento do chaveamento é aleatória. A repesentação para o *interleaving* é:

$$P \mid\mid\mid Q$$

O operator de *interleaving* é associativo, simétrico e distributivo (HOARE, 1985). É por meio de *interleaving* que são representados os dois exemplos de concorrências vistos nesse trabalho.

2.1.2 PAT e CSP#

PAT (LIU; SUN; DONG, 2008) é um framework que provê a composição, simulação e a lógica de sistemas concorrentes. Ele implementa vários modelos de verificação para diferentes propriedades, tais como: ausência de *deadlocks* e divergências, alcançabilidade, lógicas temporais, refinamento e verificação probabilística.

CSP# é uma das linguagens de modelagem suportadas pelo PAT (LIU; SUN; DONG, 2008). Ela é uma implementação da notação CSP, contudo, oference também algumas possibilidades não presentes em CSP. A linguagem combina operadores de modelagem de alto nível (escolhas, composição paralela e *interleaving*) com estruturas de linguagem de programação (C#, neste caso) de nível mais baixo (variáveis, arrays e controle de fluxo de execução), fornecendo assim uma grande flexibilidade na modelagem de sistemas concorrentes. Outra importante diferença de CSP# é que a mesma permite comunicação entre processos através de memória compartilhada nativamente. A Figura 3.1 descreve um processo em CSP#.

```
var<Stack> pilha;
var<List> lista;
AdicionarValorLista(valor) = empilhar{
                                  pilha.Push(valor)
                             }
                             -> adicionarLista{
                                  var topo = pilha.Pop();
                                  lista.Add(topo)
                             }
                             -> Skip;
```

Figura 2.1: Processo CSP#.
[Fonte: produzido pelo autor]

O código na Figura 3.1 possui um processo chamado *AdicionarValorLista* com um parâmetro de tipo inteiro *valor* e duas declarações de variáveis C#. Esse processo se comporta de acordo com dois eventos: *empilhar* e *adicionarLista*. O evento *empilhar* possui uma estrutura de código em C# dentro de seu corpo, essa estrutura armazena o valor do parâmetro no topo de uma variável C# do tipo *Stack*. Após a ocorrência do evento *empilhar*, a variável *pilha* possuirá seu topo atualizado. O evento *adicionarLista* possui duas linhas de comandos C#, um comando que atribui à variável *topo* o conteúdo armazenado no topo da variável *pilha*, e outro comando que adiciona a variável *topo* no final de uma lista. Por fim, o processo irá finalizar sua execução com sucesso por meio do processo *Skip*.

É importante ressaltar que, como mencionado, eventos são ações atômicas. Qualquer estrutura de linguagem de programação contida no corpo de um evento será executada, independente da quantidade de comandos existentes, de uma só vez na execução deste evento.

2.1.3 JVM e o *Bytecode*

JVM (*Java Virtual Machine*) (LINDHOLM; YELLIN, 1999) é uma plataforma independente de ambiente de execução que converte *bytecodes* de um programa Java em linguagem de máquina e o executa. A maioria das linguagens de programação compilam o código fonte diretamente em código de máquina, que geralmente é projetado para funcionar em uma arquitetura de microprocessador ou sistema operacional específico. A JVM simula um processador Java, permitindo *bytecodes* Java serem executadas como ações ou chamadas de sistema operacional em qualquer processador, independentemente do sistema nativo. Por exemplo, o estabelecimento de uma conexão de *socket* de uma estação de trabalho para uma máquina remota envolve uma chamada de sistema operacional. Uma vez que diferentes sistemas operacionais lidam com *sockets* de diferentes maneiras, a JVM traduz o código para que duas ou mais estações, que podem ser em diferentes plataformas, sejam capazes de se comunicar.

Na especificação da máquina virtual Java, o comportamento de uma instância é descrito em termos de subsistemas, áreas de memória, tipos de dados e instruções (LINDHOLM; YELLIN, 1999). Esses componentes descrevem uma arquitetura abstrata que define a máquina virtual Java. A finalidade destes componentes não têm o objetivo de ditar uma arquitetura interna para implementações. A intenção é fornecer uma maneira de definir rigorosamente o comportamento externo de implementações. Esta especificação define o comportamento exigido de qualquer implementação de máquina virtual Java que contem estes componentes abstratos e suas interações.

Bytecodes são códigos escritos em linguagem de nível inferior que a máquina virtual Java

é capaz de processar (LINDHOLM; YELLIN, 1999). Quando a JVM carrega um arquivo de classe, gera-se um fluxo de *bytecodes* para cada método da classe. Os *bytecodes* de um método são executados quando esse é invocado durante a execução de um programa. Eles podem ser executados por interpretação, compilação *just-in-time*, ou qualquer outra técnica que for escolhida pelo arquiteto de uma JVM (BÖRGER, 2000).

A JVM possui uma pilha que contém *frames* para cada invocação de método feito por um processo ou *thread*. A JVM por si só pode empurrar e remover da pilha de frames. Todas as outras formas de manipulação das pilhas nativas de cada frame são feitas pela *thread* em execução (VENNERS, 1999). Por exemplo, a inserção ou remoção valores na pilha de operandos contida dentro de um frame é tratada pela *thread* corrente.

Todas as constantes relacionadas a uma classe ou interface ficarão armazenados no *pool* de constantes (LINDHOLM; YELLIN, 1999). Compõem as constantes: nomes de classes, nomes de variáveis, nome de interfaces, nomes de métodos e suas assinaturas, valores de variáveis finais e *strings*.

Cada fluxo de execução de um método é determinado pelo conjunto de instruções da JVM. Cada instrução consiste em um *opcode*, que indica a ação a ser tomada, com tamanho de um *byte*, seguido por zero ou mais operandos. Se forem necessárias mais informações antes da JVM executar, estas serão codificadas em um ou mais operandos que seguem após o *opcode*. Cada tipo desse possui um mnemônico similar ao estilo da linguagem assembly. Um exemplo de código pode ser visto na Figura 2.2.

O código referente à Figura 2.2 representa a definição de uma classe *Teste*. A classe é definida por um construtor *Teste()* e pelo método *getNumero()*, que retorna um valor inteiro. O método do construtor *Teste()* possui 3 instruções. A primeira instrução insere no topo da pilha de operandos do construtor a referência *this*, a segunda instrução invoca o construtor da superclasse *Object*, desempilhando a referência inserida previamente, e a terceira instrução finaliza a execução do método não fornecendo nenhum retorno.

O método *getNumero()* apresenta 10 instruções. Inicialmente as 2 primeiras instruções são compostas por uma instrução que empilha um valor inteiro de 16 bits convertidos para 32 bits no topo da pilha e outra instrução que desempilha esse valor e armazanea em um índice do array de variáveis locais correspondente à variável *a*. A terceira e quarta instrução fazem o mesmo que as duas primeiras agora com a variável local *b*. Em seguida, nas instruções 5 e 6, os valores associados às duas variáveis locais são empilhados. Na sétima, instrução são desempilhados estes dois valores e logo após o resultado de sua divisão é empilhado. Para finalizar, o valor é desempilhado e armazenado na variável local *c* e depois empilhado novamente. A última

```
// Compiled from Teste.java (version 1.6 : 50.0, super bit)
public class Teste {

 // Method descriptor #6 ()V
 // Stack: 1, Locals: 1
 public Teste();
  0  aload_0 [this]
  1  invokespecial java.lang.Object() [8]
  4  return
    Line numbers:
      [pc: 0, line: 4]
      [pc: 4, line: 5]
    Local variable table:
      [pc: 0, pc: 5] local: this index: 0 type: Teste

 // Method descriptor #15 ()I
 // Stack: 2, Locals: 4
 int getNumero();
   0  sipush 300
   3  istore_1 [a]
   4  sipush 907
   7  istore_2 [b]
   8  iload_1 [a]
   9  iload_2 [b]
  10  idiv
  11  istore_3 [c]
  12  iload_3 [c]
```

Figura 2.2: Exemplo de código *bytecode*.
[Fonte: produzido pelo autor]

instrução remove o valor do topo da pilha e atribui ao retorno do método.

2.1.3.1 Estrutura

A JVM é uma máquina orientada à pilhas. Para cada fluxo de execução de um método de um programa Java é criado um *frame*. Nesse *frame*, que é criado cada vez que um método é invocado, estão contidos uma matriz de variáveis locais, uma referência ao *pool* de constantes e uma pilha de operandos. A Figura 2.3 representa esse *frame*.

Toda a computação presente na JVM é baseada na pilha de operandos. Já que a JVM não dispõe de registradores para armazenamento de valores, todos esses devem ser inseridos na pilha, assim como o resultado de seus cálculos.

2.1.3.2 *Opcodes*

As instruções a seguir são as mais comumente encontradas na maiora dos programas, apesar de existirem cerca de 200 instruções suportadas (LEROY, 2001). Fundamentalmente, o conjunto de instruções *bytecode* Java é dividido em várias categorias distintas. A mais impor-

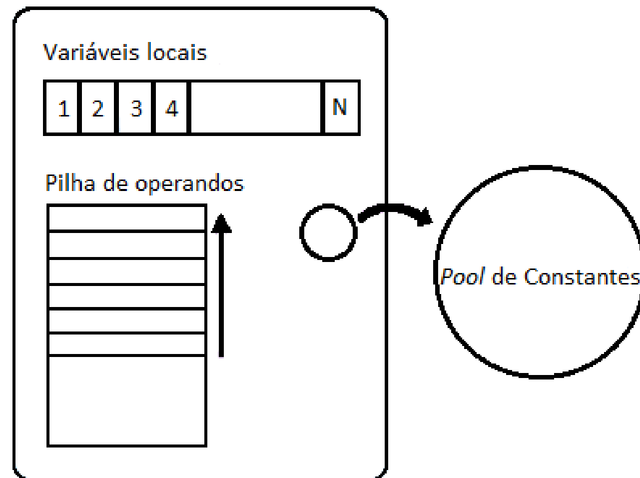

Figura 2.3: Exemplo de frame *bytecode*.
[Fonte: produzido pelo autor]

tante categoria é a de manipulação da pilha. Na Tabela 2.1 são apresentados alguns exemplos de mnemônicos de instruções e seus efeitos.

Tabela 2.1: Bytecodes para Manipulação da Pilha de Operandos.
[Fonte: Elaboração própria]

Mnemônico	**Operação**
pop	Descarta o valor do topo da pilha.
dup	Duplica o valor superior da pilha.
aconst_null	Empilha *null* na pilha.
bipush	Empilha um valor constante do tipo *byte* na pilha.
sipush	Empilha um valor constante do tipo *short* na pilha.
ldc	Empilha um valor constante a partir do *pool* de constantes na pilha.
xload	Empilha um argumento local ou variável na pilha. Onde *x* pode ser: b (*boolean*), c (*char*), d (*double*), f (*float*) i (*integer*), l (*long*), s (*short*).
xstore	Remove o valor do topo da pilha e o insere em um local no *array* de variáveis.
xconst_y	Empilha uma constante de valor y e tipo x na pilha.

São apresentadas mais instruções de exemplos como: controle do fluxo de execução do programa, instruções aritméticas e suporte a orientação a objetos na Tabela A.1, inserida no apêndice deste trabalho.

2.2 Trabalhos Relacionados

Verificação de modelos tornou-se muito popular nas duas últimas décadas (CLARKE, 1997). A verificação de modelos de programas não é uma tarefa simples devido à complexidade do código e, em muitos casos, não é possível analisar completamente o sistema devido a problemas de limite de memória. Por essa razão, alguns dos verificadores de modelo mais conhecidos se

baseiam em abstrações para reduzir o tamanho do espaço ocupado em memória pela árvore de estados possíveis do programa. Estas técnicas não são adequadas para lidar com o códigos que manipulam dados complexos, ou seja, a introdução de muitos predicados torna o processo de abstração ineficiente. Serão mostrados nesta seção alguns trabalhos que obtiveram êxito na área de testes de software utilizando verificação de modelos e que de alguma maneira são parecidos com a iniciativa deste trabalho.

2.2.1 Java *Pathfinder*

Java *PathFinder* (JPF) (HAVELUND, K.; PRESSBURGER T., 2000) é um verificador de modelos, desenvolvido pela *National Aeronautics and Space Administration* (NASA) para programas Java. Ele foi feito em cima de uma Máquina Virtual Java (JVM) customizada. JPF pode lidar com todos os recursos da linguagem Java e também trata escolhas não determinísticas, definidas como anotações no programa que está sendo analisado. As anotações são adicionadas aos programas através de chamadas de método para uma classe especial chamada *Verifier*. JPF pode identificar os pontos de um programa a partir de onde a execução pode sofrer ramificações em função de comandos de desvios, entre outros. Assim, JPF permite explorar sistematicamente todas essas ramificações. Isso quer dizer que JPF percorre, de forma teórica, todos os caminhos possíveis de uma linha de execução de programa. Ao mesmo tempo ele consegue manter o rastro de valores de variáveis locais, pilhas de *frames*, referências a objetos e estados de processos.

JPF suporta *backtracking* de estados e controle sobre escolhas não-determinísticas. Normalmente, um dos principais problemas em verificação de modelos é a explosão de estados possíveis de um programa quando o número de estados do modelo pode crescer de forma ilimitada ou acima dos limites do ambiente de verificação. A primeira precaução adotada pelo JPF está na identificação de estados similares (CLARKE, 1997). Cada vez que JPF atinge um ponto onde é necessária uma escolha para continuar, ele verifica se por acaso já passou por um estado parecido. Nesse caso ele pode abandonar o caminho que o levou a esse estado e retornar ao ponto de escolha anterior que ainda possui escolhas indefinidas, realizando assim uma decisão não determinística. A JVM customizada permite a verificação de propriedades de uma grande quantidade de programas Java sem a necessidade dos *designers* desenvolverem modelos em uma linguagem específica. A Figura 2.4 ilustra a arquitetura do JPF.

Esta arquitetura é composta pela JVM que testa todos os estados possíveis do programa. Junto à JVM estão suas três principais extensões. A interface de modelos Java, ou MJI (*Model Java Interface*), é um mecanismo utilizado para separar e realizar a comunicação entre a execu-

ção de estados já rastreados dentro da JVM do Java *PathFinder* e a execução de estados ainda não rastreados. Os geradores de escolhas, utilizados para implementar as heurísticas específicas de aplicações para aquisições de dados e políticas de escalonamento não determinísticas. Por fim, os *VMListeners* oferecem uma maneira para extender o modelo interno de estados do JPF, criando assim formas de checagem de propriedades mais complexas, buscas mais diretas, ou obter estatísticas da execução.

É possível observar também na Figura 2.4 um exemplo de diagrama de estados, onde é apresentado um fluxo de verificação de estados visitados e um fluxo de erro, e também um relatório de verificações.

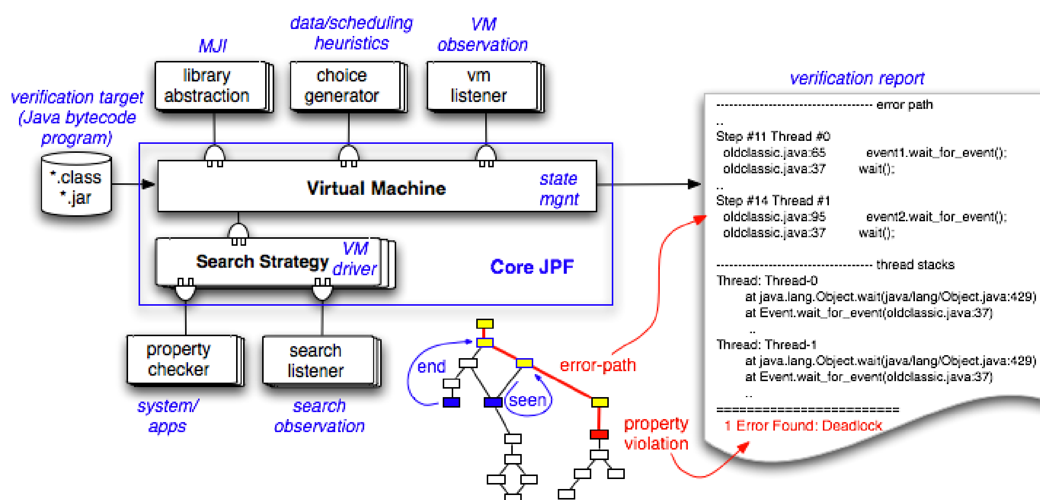

Figura 2.4: Arquitetura JPF.
[Fonte: http://javapathfinder.sourceforge.net/]

Entre as vantagens da ferramenta estão a detecção de *deadlocks* e exceções não tratadas pelo programa. Por outro lado, ao passo que JPF consegue analisar qualquer programa na linguagem Java por meio da JVM, isso o desabilita para a verificação de qualquer outro código nativo específico de plataforma. Outra desvantagem é que JPF não permite verificar se um dado programa Java possui uma dada propriedade especificada utilizando lógica temporal. Como todo verificador de modelos, também é suscetível à explosão de estados, mesmo possuindo redução de ordem parcial (WOLPER, 1995).

2.2.2 JCSP

JCSP (WELCH; MARTIN, 2000) é uma biblioteca que implementa uma API (*Application Programming Interface*) para Java e assim fornece suporte aos operadores de processos definidos em CSP na forma de classes e interfaces. Como uma biblioteca de pacotes, ela permite

a especificação de sistemas multiprocessados utilizando as primitivas de CSP para sincronização, como canais e eventos, entre outras, como escolha e paralelismo. A principal vantagem do JCSP é sua interface que permite utilizar de forma simplificada os operadores de CSP. Desta forma, permite-se a criação de especificações CSP utilizando a sintaxe da linguagem Java.

Um processo em CSP deve ser definido e instanciado como uma classe que implementa a interface *CSProcess*. As ações que cada processo deve tomar devem ser definidas dentro de um método *run* implementado da interface. A interação dos processos deve ser feita apenas por canais, eventos ou por acessos sincronizados a objetos compartilhados. Abaixo é representado um exemplo na Figura 2.5. Neste exemplo é apresentada uma classe que implementa *CSProcess* chamada *Regular*. Essa classe gera um fluxo regular de saída no canal *out*. A taxa de saída é determinada pelo atributo *interval*. O comportamento do processo é definido pelo método *run()*, onde é possível observar que o processo comunica através do seu canal de saída um inteiro *N* e com o auxílio de um *CSTimer*, que é um objeto que permite um processo aguardar a finalização de determinados eventos, aguarda pelo *interval* até a próxima comunicação.

```
import org.jcsp.lang.*;

public class Regular implements CSProcess {

  final private ChannelOutput out;
  final private Integer N;
  final private long interval;

  public Regular (final ChannelOutput out, final int n, final long interval) {
    this.out = out;
    this.N = new Integer (n);
    this.interval = interval;
  }

  public void run () {

    final CSTimer tim = new CSTimer ();
    long timeout = tim.read ();       // read the (absolute) time once only

    while (true) {
      out.write (N);
      timeout += interval;            // set the next (absolute) timeout
      tim.after (timeout);            // wait until that (absolute) timeout
    }
  }

}
```

Figura 2.5: Exemplo de Classe JCSP.
[Fonte: http://www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp-1.1-rc4/jcsp-doc/]

O que torna esse padrão tão simples é que um processo pode ser considerado como um objeto que atua individualmente como um programa, onde toda a relação com o ambiente é realizada através de operações de entrada e saída. Levando em consideração a existência de vários processos, a composição paralela entre eles pode ser considerada como uma rede baseada em camadas que contém processos encapsulados.

Um fator negativo em relação ao JCSP é que a comunicação nativa entre processos é feita, exclusivamente, na forma de canais ou *links* (canais entre nódulos ou rede de processos). Não foi disponibilizada nenhuma outra forma de comunicação que não seja dessa natureza o que exclui, por exemplo, comunicação através de memória compartilhada. Contudo, esta restrição é coerente com CSP que também só permite este tipo de comunicação diretamente. Os canais de comunicação são classes que conectam um ou mais processos e podem ser:

- Um-para-um, um-para-muitos, muitos-para-um ou muitos-para-muitos
- Barreiras
- *Optional buffering*
- *Connections* (transações de 2 vias)

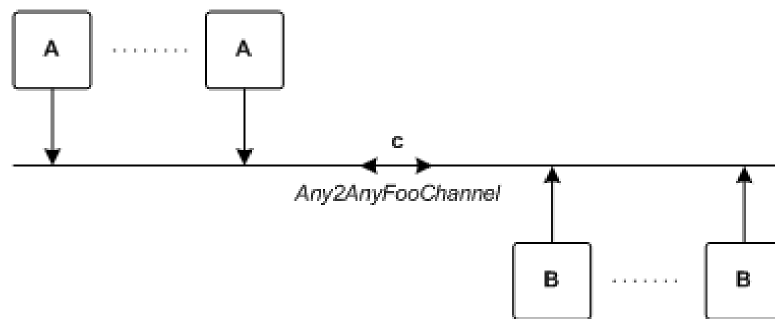

Figura 2.6: Exemplo de canal Muitos-para-muitos.
[Fonte: http://www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp-1.1-rc4/jcsp-doc/]

Na figura Figura 2.6 é apresentado um exemplo de canal de comunicação entre muitos processos. A sincronização também pode ser definida por meio de canais. Um exemplo simples seria a utilização dos canais *Barriers* ou barreiras. Neste tipo de canal a execução dos processos comunicantes só segue mediante a finalização de algum evento que permita a liberação através da barreira.

3 *Método de Pesquisa*

Esse trabalho é executado em várias etapas. Inicialmente, para entender os detalhes dos comandos *bytecode* é necessário, compreender a arquitetura e o funcionamento da JVM (*Java Virtual Machine*). Em seguida, será estudada a notação CSP. Neste trabalho é adotada a linguagem CSP# que é uma implementação de CSP para a ferramenta PAT (*Process Analysis Toolkit*).

3.1 Qualificação do Método de Pesquisa

Como dito anteriormente no capítulo 2, o framework PAT é capaz de verificar diferentes propriedades, tais como: ausência de deadlocks, divergências, não-determinismo. Contudo, este trabalho vai focar apenas na análise de ausência de deadlocks e a não finalização de programas.

Com PAT é possível analisar de forma exaustiva o LTS (*Labeled Transition System*) que representa o comportamento de um código escrito em CSP#. A partir deste fato é possível verificar se um programa obtido de um código Java pode apresentar *deadlock* em algum momento ou se o mesmo termina sempre sem erros.

3.2 Etapas do Método de Pesquisa

Em seguida serão apresentadas as atividades definidas para a execução deste trabalho. Foram incluídas desde a coleta de referencial teórico, para o estudo do problema, até a elaboração das regras de mapeamento que serão definidas para a solução do problema proposto.

3.2.1 Estudo de CSP

Nesta estapa, buscou-se identificar quais as propriedades da linguagem, para que serve, e porque é útil para a verificação de propriedades de sistemas compostos por processos concorrentes. Nesta etapa, também se fez o estudo de duas implementações de CSP: CSP_M e CSP#.

A partir da comparação das semelhanças e diferenças entre ambas, optou-se, como dito anteriormente, pelo uso de CSP#, pois esta oferece diretamente o modelo de comunicação entre processos que precisa ser adotado neste estudo: comunicação através de memória compartilhada.

3.2.2 Estudo de *Bytecode* Java

Para que o mapeamento fosse possível, foi necessário um estudo sobre o que ocorre internamente na JVM. Sabe-se que a mesma é uma estrutura orientada a pilhas, por isso a forma de contrução das estruturas que compõem os processos também precisam estar de acordo com o comportamento de uma pilha. Foi preciso identificar os *bytecodes* mais importantes, e entender como os mesmos poderiam ser traduzidos para o CSP#, levando em consideração o comportamento de cada um (parâmetros, retorno, ações, etc.) utilizando estruturas de pilhas como base.

3.2.3 Estudo de Trabalhos Relacionados

Com o estudo de trabalhos relacionados, em particular, JCSP, e Java PathFinder, foi realizado um comparativo, entre métodos mais conhecidos existentes de análise formal de código concorrente, e o método proposto por esse trabalho. Foram observados os benefícios e desvantagens de cada método. Desde a facilidade para definição de um sistema utilizando JCSP quanto a possibilidades de verificação oferecidas por Java *PathFinder*.

Apesar de abstrair alguns detalhes de CSP e de permitir o uso da linguagem Java, JCSP ainda requer que o desenvolvedor tenha conhecimento mínimo da semântica de CSP. Já Java PathFinder tem uma curva de aprendizagem menor. Contudo, esta não permite realizar algumas análises possíveis em CSP, nem fazer uso, por exemplo, de refinamentos. Além desses aspectos, nenhuma destas duas ferramentas permitem verificar propriedades do sistema através da formulação de proposições em lógica temporal.

Portanto, este trabalho possui seu diferencial, uma vez que:

1. Não exige que o desenvolvedor tenha conhecimento de CSP.
2. Ao gerar código CSP#, permite a verificação de proposições temporais, o que é possível fazer utilizando PAT. Apesar deste não ser o foco deste trabalho.

3.2.4 Definição do Mapeamento de *Bytecode* Java para CSP#

Inicialmente, foi feito um levantamento de quais as instruções seriam mais importantes para a definição do comportamento de um processo. Foram identificadas como mais importantes as instruções de manipulação de pilha, atribuição de valores e constantes, e, instruções que realizam cálculos matemáticos e lógicos. Como quase todo o apanhado de instruções trabalha, em função da aquitetura da JVM, com manipulação de pilhas, estas foram consideradas as instruções mais importantes para alvo de um primeiro esforço de mapeamento.

A partir das instruções selecionadas, foi necessário traçar uma relação entre estas e CSP#. Por exemplo, como mapear o comportamento da instrução iadd que é tira os dois últimos elementos da pilha, soma-os, e em seguida empilha o resultado da soma.

Na JVM tudo é calculado empilhando e desempilhado valores, realizando a operação definida pelo *opcode* e depois empilhando, se houver, o retorno da operação. Devido a este fato, a utilização de, pelo menos, uma variável que simula uma pilha no código CSP# torna-se imprescindível.

3.2.5 Definição dos Exemplos de Concorrência

Foram selecionados dois exemplos de programas concorrentes para extrair e demonstrar que as regras de transformação, juntamente com as asserções no PAT, funcionam de forma satisfatória. Os critérios para a seleção dos exemplos foram:

1. Os dois exemplos não devem possuir erros de compilação, uma vez que o *bytecode* só é gerado para códigos sintaticamente corretos.

2. Um exemplo constitui um programa que se espera que nunca termine, e de fato que nunca termina, e outro que se espera que sempre termina, mas que eventualmente pode não terminar.

Logo, uma vez que o primeiro exemplo for executado, ele nunca terminará. Já o outro exemplo, poderá terminar ou não. Isso vai depender da forma que as *threads* serão tratadas pelo escalonador de processos responsável pelo chaveamento dessas. O exemplo com a possibilidade indesejada de não terminar foi escolhido pois, casos de teste, em algumas vezes, não conseguem identificar este tipo de falha. Caso um ambiente favoreça a não ocorrência de um erro de concorrência, o caso de teste pode fornecer uma asserção positiva em um determinado método

que mesmo assim pode executar um programa de forma errada. Portanto, nos testes propostos pelos dois exemplos definidos, deseja-se chegar aos seguintes resultados:

- A verificação de um código no PAT, obtido a partir da aplicação das regras de transformação, deve informar que o código é *non-terminating* quando de fato ele nunca termina.
- A verificação de um código no PAT, obtido a partir da aplicação das regras de transformação, deve mostrar que um código pode não terminar, quando se espera que o mesmo sempre termine.

3.2.6 Transformação dos Exemplos em CSP#

Os exemplos foram então transformados em códigos CSP# de acordo com o mapeamento que será apresentado no capítulo 4 através de tabelas de relacionamento. Para cada instrução, ou conjunto de instruções, serão definidos processos que os representam.

O fluxo de transformação das linhas do programa para CSP# ocorrerá de forma sequencial. Qualquer desvio, seja ele condicional ou incondicional, será tratado por regras específicas que alteram o fluxo de leitura e transformação das instruções. Esse fato ocorre, principalmente, na execução de laços ou rotinas recursivas. Quando uma instrução de desvio for localizada, as regras de mapeamento irão definir um novo processo. Este novo processo vai possuir em seu corpo todas as instruções compreendidas entre a posição da instrução de desvio e a posição desviada. Este procedimento será apresentado em detalhes no próximo capítulo.

3.2.7 Análise dos Resultados

Os resultados serão coletados através das asserções que podem ser feitas no PAT. Após a conversão para CSP# de cada exemplo, será utilizada uma verificação para cada propriedade interessante ao estudo deste trabalho. Essas propriedades são: *deadlock-free* e *nonterminating*.

3.3 Limitações da Pesquisa

Estimar a quantidade necessária de exemplos para definir um mapeamento uniforme esntre as instruções foi uma das dificuldades encontradas neste trabalho. Para evitar que a quantidade de transformações não interferissem no andamento do trabalho, foram escolhidos apenas dois exemplos para as transformações. A pouca quantidade de informações gerada por apenas dois

exemplos também influiu negativamente na definição das regras, pois isso facilita uma interpretação errada da lógica de tranformação entre o *bytecode* e o CSP#, ocasionando em uma falsa impressão de que um mapeamento de um caso particular possa ser utilizador de forma geral.

Foi necessária também, a utilização de um exemplo fictício que só manipula listas de inteiros, uma vez que a API padrão de PAT só fornece implementação para listas de inteiros. Contudo, esta pode ser modificada para suportar outros tipos.

Ao longo dos teste realizados com os códigos obtidos a partir do mapeamento, foram encontrados alguns erros relacionados à ferramenta PAT. Tais erros inlfuíram na alteração de algumas regras de mapeamentos.

Por ser uma ferramenta ainda em desenvolvimento a ocorrência de erros já era esperada. Porém algumas falhas apresentaram uma criticidade acima da média. Abaixo estão listados os *bugs*:

1. **Desordem de elementos empilhados em sequencia**. O PAT embaralhou os elementos inseridos em uma variável do tipo <*Stack*>. Isso gerou uma mudança muito grande no mapeamento, pois antes uma pilha do operandos de um método era tipada como <*Stack*>. Após a descoberta desse erro foi necessária a utilização da biblioteca <*List*> para simular uma pilha. Para isso foram criados métodos *Pop()* e *Push(v)* dentro da biblioteca <*List*>. A Figura 3.1 mostra os estado da pilha durante a execução do processo.

2. **Objetos perdem seu estados após chaveamento entre processo diferentes**. Quando a execução era passada de um processo para outro em paralelo as variávies correspondentes às pilhas de operandos (tanto <*Stack*> quanto para <*List*>) estavem sendo reinicializadas. Este erro ocorreu dentro das estruturas de escollha de fluxo de execução após o empilhamento do retorno de um *bytecode* de comparação.

3. **Perda da atomicidade de eventos se colocados dentro de um *ifa*. Ao criar um bloco atômico com um *ifa*, o mesmo não se comportava de forma atômica. Ao mudar para *if*, obteve-se o comportamento atômico esperado.**

4. **A geração do grafo completo de um modelo contendo uma pilha do tipo <*Stack*> não apresenta as informações corretas**. O PAT possui a capacidade de vizualiar todos os estados das variáveis envolvidas em um ambiente de execução durante cada evento ocorrido. Ao executar uma simulação evento por evento, é possível visualizar os valor das variáveis. Porém se a simulação for feita de uma só vez todos os valores de listas e pilhas são reiniciados.

```
var<Stack> pilha;
P() = a{pilha.Push(4)}
»       -> a2{pilha.Push(2)}
»       -> b{pilha.Push(2)}
»       -> c{pilha.Push(6)}
»       -> d{var x = pilha.Pop(); pilha.Push(x*x)}
»       -> Skip;
»
»
The process is:
P()
Variables:
pilha=[];
-------------------------------
The process is:
a2{pilha.Push(2)}->b{pilha.Push(2)}->c{pilha.Push(6)}->d{var x = pilha.Pop(); pilha.Push((x * x))}->Skip
pilha=[4];
-------------------------------
The process is:
b{pilha.Push(2)}->c{pilha.Push(6)}->d{var x = pilha.Pop(); pilha.Push((x * x))}->Skip
Variables:
pilha=[2,4];
-------------------------------
The process is:
c{pilha.Push(6)}->d{var x = pilha.Pop(); pilha.Push((x * x))}->Skip
Variables:
pilha=[2,4,2];
-------------------------------
The process is:
d{var x = pilha.Pop(); pilha.Push((x * x))}->Skip
Variables:
pilha=[6,2,4,2];
-------------------------------
The process is:
SkipVariables:
pilha=[4,4,2,6];
```

Figura 3.1: Exemplo de Falha de Pilhas no PAT .
[Fonte: produzido pelo autor]

4 *Resultados*

Neste capítulo serão apresentadas as regras de transformação entre o código *bytecode* e CSP#. Cada regra será apresentada como uma associação entre uma ou um conjunto de instruções e seus respectivos comandos em CSP#.

4.1 Regras de Mapeamento

As regras de mapeamento serão apresentadas em três subseções: as regras de mapeamento gerais, as regras de nomenclatura e as regras de mapeamento de instruções *bytecode* Java.

4.1.1 Regras de Mapeamento Geral

Em primeiro lugar serão apresentadas as regras gerais que definem o mapeamento das instruções. Isto irá caracterizar como será modelado o comportamento do ambiente de transformação.

É necessário definir primeiramente um conjunto de leis estruturais que definam o ambiente onde as instruções serão mapeadas. As regras gerais vão definir como será feita a modelagem da JVM.

4.1.1.1 RG1: As instruções geradas por chamadas de construtor sem código não serão mapeadas

As chamadas de construtores onde não são realizados nenhum comando não precisam de mapeamento e podem ser abstraídas. As instruções que correspondem a estas chamadas apenas carregam a referência para a classe na pilha de operandos e não implica em nada relevante no nível de processos.

Como nenhum código é executado não foi detectada a necessidade de traduzir isso para o PAT. Construtores que executam código serão mapeados normalmente e sua tradução será

similar à tradução de métodos de classe onde seu mapeamento será explicado mais adiante nesta seção.

4.1.1.2 RG2: Cada método irá definir, pelo menos, um processo que descreverá seu comportamento

Para cada método de classe é definido um escopo. A JVM especifica que cada escopo deve possuir uma pilha de operandos onde será realizada toda a computação dos cálculos. Serão considerados neste estudo apenas escopos definidos por chamadas de métodos, construtores de classe e blocos de repetição.

4.1.1.3 RG3: Cada método deve possuir sua própria pilha de operandos que será representada por uma variável do tipo List de C#

Como visto nos capítulos anteriores, cada método na JVM necessita de sua própria pilha de operandos. Para representar essa estrutura será adotada a biblioteca *List*, nativa da linguagem C# e suportada pelo PAT. A opção pela biblioteca *List* e não a *Stack* ocorreu devido a erros que a ferramenta PAT apresenta. Foram incluídos dois métodos na biblioteca de *List* para que um objeto de sua classe se comporte como uma pilha. O método *Pop* remove um elemento do fim da lista e um método *Push*, insere um elemento no fim da lista. Desta forma, temos uma lista que se comporta de acordo com o padrão LIFO (*Last In*, *First Out*), que, por sua vez, é a idéia de uma pilha.

Quando existir uma chamada de método, construtor de classe (que possua código mapeável) ou a presença de um bloco de repetição, será imediatamente definida uma variável do tipo *List*, representando uma pilha, para o processo definido por uma dessas três estruturas. Essa defenição ocorrerá antes do mapeamento do próprio método para garantir que a variável seja acessada pelos procedimentos que serão definidos dentro do processo.

As regras de nomenclatura de cada variável irão depender do seus escopos e serão vistas mais adiante.

4.1.1.4 RG4: Para cada variável instanciada será criada uma variável similar em C#

Para definição e chamadas de variáveis locais, o mapeamento também deve criar a devida representação desta em C#. A declaração das variáveis devem ser feitas antes da definição do processo que irá utilizá-la em sua representação.

Inicialmente, só será possível manipular variáveis que sejam dos tipos primitivos inteiro

(*int*) e booleano (*boolean*), como também objetos compostos por estes tipos primitivos, além de objetos do tipo *ArrayList<int>*.

4.1.1.5 RG5: O suporte à *ArrayList<int>* será em função da biblioteca *List*

Nas instruções onde existirem instâncias de objetos do tipo *ArrayList<int>*, será oferecido suporte através da biblioteca *List* de C# (serão considerados apenas inteiros inicialmente).

O método *void obj.add(v)* de Java será mapeado no método *void obj.Add(v)* de C#. O método *int obj.remove(pos)* de Java será mapeado no metodo *void obj.RemoveAt(pos)* de C#, com uma chamada *int obj.Get(pos)* antes. Por fim, o método *int obj.size()* de Java será mapeado no método *int obj.Count()* de C#.

O suporte a outros tipos de dados no PAT é limitado para alguns tipo de bibliotecas em C#. É possível definir novos tipos de dados porém essa funcionalidade não será explorada nesse trabalho.

4.1.1.6 RG6: A parametrização de processos será possível, porém limitada

Em processos parametrizados, somente parâmetros inteiros e booleanos são suportados devido a uma restrição do PAT. Parâmetros só precisam ser definidos na assinatura dos processos e sua nomenclatura segue a a nomenclatura normal de variáveis. Por simplificação, os parâmetros booleanos serão tratados como valores inteiros.

4.1.1.7 RG7: O retorno de processos será realizado por variáveis de retorno

Como processos em CSP não retornam valores, o retorno de métodos mapeados será por meio de variáveis exclusivas para cada processo. Isso traz uma limitação, pois não será possível, no momento, mapear métodos que retornam valores e podem ser chamados simultaneamente por *threads* diferentes. Logo, nesta proposta inicial de mapeamento, assume-se que os métodos que retornam valores são *synchronized*.

As variáreis de retorno serão declaradas antes dos processos que a utilizam. Assim como na regra de parametrização de processos, também será simplificado o retorno de métodos do tipo *boolean* como valores inteiros.

4.1.1.8 RG8: Chamada de métodos *synchronized*

Na ocorrência de métodos *synchronized*, os mesmos serão chamados dentro de blocos *atomic* necessários para manter a uniformidade da execução dos eventos. A JVM insere o *bytecode monitorenter* e o *bytecode monitorexit* na definição de um bloco de comandos *synchronized*. Isso poderia resultar em uma regra de mapeamento nova, incluindo os comandos compreendidos entre o monitor em um bloco *atomic*. Porém ao definir um método como *synchronized*, a JVM realiza uma abstração na chamada desse monitor. Devido a essa abstração é que o bloco *atomic* será inserido na chamada de um processo que define um método *synchronized*, juntamente com seu retorno caso exista.

4.1.2 Regras de Nomenclatura

Aqui serão apresentadas as regras que definem a nomenclatura de todos os elementos que irão compor o código CSP#. Será apresentada a forma e os padrões de como serão nomeadas todas as variáveis do ambiente. As regras de nomeação servirão para identificar as estruturas do mapeamento. Estarão envolvidas nessa seção as formas para definir nomes de variáveis, processos e eventos a partir de suas classes, métodos e escopos.

4.1.2.1 RN1: Padrão para nomeação de métodos

%NomeObjeto%_Object_%NomeDoEscopo%_Scope_%nomeDoMetodo%_method()

Cada processo será modelado baseado no funcionamento de um determinado método do código a ser analizado. Para isso é necessário identificar qual o objeto que invoca esse método e também em qual escopo ele está sendo executado. Podem existir várias instancias de uma mesma classe executando seus métodos, devido a esse fato é muito importante definir os escopos de cada processo.

No mapeamento, ao identificar que dois objetos *obj1* e *obj2* do tipo *Obj* executam um método *m*, só podemos diferenciar os processos definidos pela chamada desse método pelo seu escopo. Isso resultaria na definição de processos como por exemplo:

Obj_Object_S1_Scope_m_method() e Obj_Object_S2_Scope_m_method().

4.1.2.2 RN2: Padrão para nomeação de operações

%NomeObjeto%_Object_%NomeDoEscopo%_Scope

_%nomeDoMetodo%_method%_%nomeDoEvento%

Processos são definidos por eventos. Cada evento de um processo será representado por uma operação. Uma vez que algumas instruções podem realizar uma ou mais operações, será comum a ocorrência de várias operações associadas a apenas um evento.

Exemplo: Obj1_Object_Esc1_Scope_run_method_PushConstant{}

4.1.2.3 RN3: Padrão para nomeação de variáveis locais

%NomeObjeto%_Object_%NomeDoEscopo%_Scope

_%nomeDoMetodo%_method%_nomeDaVariavel%_var

Toda variável local vai possuir uma semelhante no mapeamento CSP#. Seu padrão de nomeação será semelhante ao padrão de definição de processos, com um diferencial, a informação ao fim da declaração informando que se trata de uma variável utilizando o sufixo *var*

Exemplo: Exemplo1_Object_Main_Scope_m_method_f_var

4.1.2.4 RN4: Padrão para nomeação de variáveis de retorno de métodos

%NomeObjeto%_Object_%NomeDoEscopo%_Scope

_%nomeDoMetodo%_method%_nomeDaVariavel%_varReturn

Processos no PAT não possuem a capacidade de simular retorno de métodos. Para suprir essa necessidade serão criadas, para cada processo que defina o comportamento de um método que retorna valores, variáveis locais que terão a função de representar o retorno de processos. Elas serão definidas antes dos processos e, assim como todos os valores adotados, possuirão valores inteiros ou inteiros que representem valores booleanos.

Exemplo: Exemplo1_Object_main_Scope_flag_varReturn

4.1.2.5 RN5: Padrão para nomeação de atributos de classe

%NomeObjeto%_Object_%NomeDoEscopo%_Scope_nomeDaVariavel%_var_attribute

Assim como as variáveis locais, os atributos também possuirão representação em CSP#. Para cada escopo de classe detectado serão criadas novas variáveis que representem seus atributos, caso existam.

Exemplo: Exemplo1_Object_Main_Scope_tamanhoMaximo_var_attribute

4.1.2.6 RN6: Padrão para nomeação da variável correspondente à pilha de operandos

%NomeObjeto%_Object_%NomeDoEscopo%_Scope_m_method_Stack

%NomeObjeto%_Object_%NomeDoEscopo%_Scope_m_method_L<n>_Stack

Como todo método ou laço precisa de sua própria pilha de operandos, para cada necessidade detectada também será criada uma variável do tipo *List* de C# que a represente. Semelhante à nomeação de atributos, sua difereça está na terminação informando o tipo de dado que define a variável e a ausência do indentificador do nome da variável, uma vez que as pilhas dependem exclusivamente do escopo que às definem. Para os laços, as pilhas instanciadas possuirão o mesmo padrão de nomenclatura, adicionando apenas a letra "L"e um número que é determinado pela ordem de aparição do laço dentro do método.

4.1.2.7 RN7: Padrão para nomeação de variáveis que representam tipos de dados

%NomeObjeto%_Object_%NomeDoEscopo%_Scope_%nomeDaVariavel%

var%TipoDado%

Em alguns casos, é necessário criar variáveis CSP# para representar tipos de dados não primitivos, seja para facilitar no desenvolvimento do algoritmo ou para aumentar o nível de abstração, facilitando assim o entendimento do código. Para isso será adotado o mesmo padrão de nomeação de variáveis locais, diferenciando apenas na concatenação contendo a informação do tipo de dado representado ao final da variável.

Exemplo: Exemplo1_Object_Main_Scope_colecao_var_List

4.1.2.8 RN8: Padrão para nomeação de método *synchronized*

synchronized_%NomeObjeto%_Object_%NomeDoEscopo%_Scope_m_method

Em definições de métodos *synchronized* essa definição será concatenada ao início do nome do processo.

Exemplo: synchornized_ListaRecursos_Class_S1_Scope_updateRecurso_method

4.1.2.9 RN9: Padrão para nomeação de variáveis de retorno de comparações

%NomeObjeto%_Object_%NomeDoEscopo%_Scope_m_method_comparison_result

São as variáveis que representam o retornos dos mapeametos de instruções que realizam

comparações. Se houver mais de uma comperação dentro do mesmo método, será adicionado um contador após o texto *result*.

Exemplo: O_Object_S_Scope_m_method_comparison_result

4.1.3 Regras de Mapeamento de Instruções *Bytecode* Java

Serão relacionadas todas as transformações das instruções *bytecode* selecionadas por este estudo. Como existem atualmente 207 instruções definidas de 256 possíveis (cada *opcode* ocupa um *byte*) e muitas instruções não são utilizadas no contexto do nosso estudo, que é a concorrência entre processos, vamos apresentar o mapeamento das intruções mais importantes relacionadas ao tema.

As regras foram obtidas através da inferência do comportamento da JVM, cujo foco principal é na execução de cálculos através da pilha de operandos.

Todo *bytecode* foi analisado de forma sequencial, passando instrução por instrução, formando assim um mapeamento que oferecesse significado a uma ou várias instruções consecutivas. Para a grande maioria dos *opcodes*, tem-se um mapeamento por instrução, porém em alguns casos, como em comparação de valores empilhados e mudança de fluxo de execução, não é possível realizar o mapeamento utilizando apenas o primeiro *opcode*. Foi necessário analisar os *opcodes* seguintes para poder formar uma regra que definisse uma operação representando uma comparação.

Nem todas as instruções foram mapeadas. O conjunto é muito extenso e existem instruções que não são interessantes ao estudo (como funções de impressão de valores em tela). As instruções não mapeadas foram descartadas e na presença de algum conjunto de instrução que não possua regra correspondente, esta não deve ser levado em consideraçãor.

4.1.3.1 RB1: Inserção de constante na pilha

Este mapeamento representa a operação de inserir uma constante na pilha de operandos. Onde i é o dado do tipo inteiro a ser inserido e N o valor, m1 para "-1" ou variando de 0 à 5. Neste caso específico vamos utilizar o *opcode* iconst. Ele possui a habilidade de empilhar apenas constantes do tipo *integer*.

O mapeamento é definido declarando um evento *PushConstant*, que no momento de sua ocorrência vai executar o empilhamento do valor "N" no topo da fila de operandos. A Tabela 4.1 apresenta o mapeamento.

Tabela 4.1: Mapeamento da Instrução de Inserção de Constante na Pilha de Operandos.
[Fonte: Elaboração própria]

Instrução	Mapeamento CSP#

iconst_N	O_Object_S_Scope_m_method_PushConstant{
	O_Object_S_Scope_m_method_Stack.Push(N)
	}

4.1.3.2 RB2: Atualização de valor de um atributo de classe

Através do evento *UpdateAttribute* é possível atribuir um valor a um atributo de classe. Ao identifcar o *opcode putfield* o evento realiza uma operação atômica. Primeiro é atribuído o valor contido no topo da pilha a uma variável auxiliar definida dentro do próprio corpo do evento. Logo após, uma variável, que representa um atributo de classe, recebe esse valor, antes pertencente ao topo da pilha, por meio da variável auxiliar. Esses passos estão contidos dentro do corpo do evento, caracterizando assim a ocorrência do mesmo após terem sido realizadas essas duas operações.

A Tabela 4.2 mostra o mapeamento da operação para atualizar valores (previamente empilhado e no topo da pilha) a um atributo de classe

Tabela 4.2: Mapeamento das Instruções de Atualização de Atributos.
[Fonte: Elaboração própria]

Instrução	Mapeamento CSP#

putfield Objeto.Atributo	O_Object_S_Scope_m_method_UpdateAttribute{
	var top = O_Object_S_Scope_m_method_Stack.Pop();
	O_Object_S_Scope_v_var_attribute = top
	}

4.1.3.3 RB3: Finalização de método

Ao final de cada método Java existe um comando em seu *bytecode* que corresponde à finalização do mesmo. Mesmo que o método não possua retorno, a instrução *return* sempre estará presente. Para diferenciar métodos com e sem retorno, a instrução possui um indicador que informa o tipo de retorno do método. Caso não possua nada é uma função que retorna *void*, se por acaso possuir a letra *i* antecedendo a instrução *return* significa que o método retorna um valor inteiro.

O mapeamento que definirá o retorno de métodos será constituído por um ou mais eventos. Se por acaso o método não possuir retorno somente o evento *SKIP* será utilizado. Informando, assim, a finalização do processo que representa o método. Existindo um retorno, o evento *SKIP* será antecedido pelo evento *SetReturn*. O evento *SetReturn* possuirá em seu corpo uma operação

onde uma variável de retorno irá receber um resultado calculado previamente pelo processo e inserido no topo da pilha. A Tabela 4.3 apresenta o mapeamento de finalização de método.

Tabela 4.3: Mapeamento das Instruções de Finalização de Métodos.
[Fonte: Elaboração própria]

Instrução	**Mapeamento CSP#**
return	Skip;

ireturn	O_Object_S_Scope_m_method_SetReturn{
	    O_Object_S_Scope_m_method_v_varReturn =
	    O_Object_S_Scope_m_method_Stack.Pop()
	}
	-> Skip;

4.1.3.4 RB4: Inserção do valor de uma variável na pilha

Sempre que for necessário carregar o valor de uma variável na pilha de operandos o evento *PushValue* será chamado. Ele representa o mapeamento da intrução *iload_N*, onde i é um dado do tipo inteiro a ser inserido, e N um valor variando de 0 à 3 (existe a possibilidade ainda de escolher qualquer índice na tabela utilizando *iload índice* porém não precisaremos utilizar) correspondente ao campo dentro da tabela de variáveis. A Tabela 4.4 apresenta seu mapeamento.

Tabela 4.4: Mapeamento de Empilhamento de Valor de Variável na Pilha.
[Fonte: Elaboração própria]

Instrução	**Mapeamento CSP#**

iload_N	O_Object_S_Scope_m_method_PushValue{
	    O_Object_S_Scope_m_method_Stack.Push(
	        O_Object_S_Scope_m_method_v_var)
	}

4.1.3.5 RB5: Empilhar valor de atributo do objeto

Similar ao empilhamento de variáveis, tem-se também o de valores de atributos. Utilizando o mesmo evento *PushValue*, porém com uma diferença, no código dentro evento será empilhado o valor de um atributo do objeto. O mapeamento é semelhante, porém o conjunto de instruções é totalmente diferente. a instrução *getfield* antecedida por um *aload* resgata o valor de atributo contido numa classe obtida de um determinado escopo.

Novamente, para esse tipo de instrução, será definido que, por causa das restrições do modelo adotado neste estudo, só será gerado código se o atributo for do tipo inteiro. A Tabela 4.5 exemplifica o mapeamento.

Tabela 4.5: Mapeamento de Empilhamento de Valor de Atributo de Classe.
[Fonte: Elaboração própria]

Instrução	Mapeamento CSP#

aload_0 [this]
getfield Objeto.Atributo
	O_Object_S_Scope_m_method_PushValue{
	O_Object_S_Scope_m_method_Stack.Push(
	O_Object_S_Scope_v_var_attribute)
	}

4.1.3.6 RB6: Mapeamento de listas

Para estruturas mais complexas, é necessário um mapeamento mais específico das operações inerentes destas. Será utilizado aqui o mapeamento de listas por diversos motivos. São estruturas largamente utilizadas em qualquer tipo de sistema, além de possuir métodos já definidos que permitem sua manipulação. O mapeamento de listas vai focar nos métodos mais importantes.

Antes de qualquer método, é necessário definir o mapeamento do momento em que uma lista é criada. Uma vez que o evento que definirá a criação de uma lista vai conter o comando *new* em seu corpo para instanciar uma lista local, será adotada uma nomeação diferente para este evento: concatena-se *new* no começo da palavra que define o evento. Isso é visto na tabela 4.6 e será identificado como a regra **RB6.1**.

Tabela 4.6: Mapeamento de Criação de Nova Lista.
[Fonte: Elaboração própria]

Instrução	Mapeamento CSP#
new java.util.ArrayList invokespecial java.util.ArrayList()	new_O_Object_S_Scope_v_var_List{O_Object_S_Scope_v_var_List = new List()}

Na Tabela 4.7 pode-se observar como funciona o mapeamento da adição de um elemento na lista. Quando a instrução *invokevirtual java.util.ArrayList.add(N)* é detectada o mapeamento vai considerar que um atributo ou variável do tipo lista indicada em uma instrução anterior referenciada por um *getfield* está adicionando um valor. Esta regra será conhecida como **RB6.2**. A operação *Event* pode representar tanto um *UpdateAttribute* quanto um *UpdateLocalVar*.

Tabela 4.7: Mapeamento da Instrução de Inserção em uma Lista.
[Fonte: Elaboração própria]

Instrução	Mapeamento CSP#

invokevirtual
java.util.ArrayList.add(N)
	O_Object_S_Scope_m_method_Event{
	var top = O_Object_S_Scope_m_method_Stack.Pop();
	O_Object_S_Scope_v_var_List.Add(top)
	}

De forma semelhante à inserção, funciona a detecção da instrução de remoção de um valor

(previamente empilhado no topo da pilha) em um objeto do tipo *List* ou, como será conhecida, regra **RB6.3**. A Tabela 4.8 mostra que seu mapeamento é bastante diferente da inserção.

Tabela 4.8: Mapeamento da Instrução de Remoção em uma Lista.
[Fonte: Elaboração própria]

Instrução	Mapeamento CSP#

invokevirtual
java.util.ArrayList.
remove(N)
	O_Object_S_Scope_m_method_Event{
	var top = O_Object_S_Scope_m_method_Stack.Pop();
	var value = O_Object_S_Scope_v_var_List.Get(top)
	O_Object_S_Scope_m_method_Stack.Push(value);
	O_Object_S_Scope_v_var_List.RemoveAt(top)
	}

Quando o método *size()* de uma lista é chamado, o resultado do cálculo do tamanho da lista é empilhado no topo da pilha de operandos. Por isso a regra **RB6.4** vai representar o empilhamento da quantidade de elementos de uma *List* na pilha . A Tabela 4.9 apresenta essa operação.

Tabela 4.9: Mapeamento da Instrução de Empilhamento do Tamanho de Lista.
[Fonte: Elaboração própria]

Instrução	Mapeamento CSP#

invokevirtual
java.util.ArrayList.size()
	O_Object_S_Scope_m_method_PushValue{
	var quant = O_Object_S_Scope_v_var_List.Count();
	O_Object_S_Scope_m_method_Stack.Push(quant)
	}

É possível resgatar um valor em qualquer posição de uma lista. Através do método *get(int)* da biblioteca *List* é passados um parâmentro que representa o índice onde stá guardado o valor na lista. A Tabela 4.10 apresenta essa operação e sua regra será conhecida como **RB6.5**.

Tabela 4.10: Mapeamento da Instrução de Obtenção de Ítem de uma Lista.
[Fonte: Elaboração própria]

Instrução	Mapeamento CSP#

invokevirtual
java.util.ArrayList.get(I)
	O_Object_S_Scope_m_method_Event{
	var top = O_Object_S_Scope_m_method_Stack.Pop();
	var value = O_Object_S_Scope_v_var_List.Get(top)
	O_Object_S_Scope_m_method_Stack.Push(value)
	}

4.1.3.7 RB7: Mapeamento de comparação de valores

Comparações são utilizadas pela JVM para definir o fluxo de execução de um programa a partir de desvios condicionais. Instruções de comparação desempilham, dependendo do *opcode*, um ou dois valores do topo da pilha de execução para realizar suas operações. Instruções que

desempilham apenas um valor realizam a comparação do mesmo com o valor zero, enquanto que instruções que desempilham dois valores realizam a comparação entre eles.

No mapeamento podem existir duas maneiras de definir as instruções que serão executadas após uma comparação. O sentido do *jump* de uma comparação vai determinar o tipo de operação a que a comparação pertence. Dado um *bytecode*: *l1 compara l2*, onde *l1* é o *label* atual do fluxo de execução, *compara* é o *opcode* da operação e *l2* é o *label* para onde o fluxo de execução será desviado se os parâmetros satisfizerem a condição proposta. Se *l2* for menor que *l1* significa que o fluxo está sendo desviado para um ponto anterior ao de execução, causando uma repetição das instruções anteriores e definindo assim, um laço. Já se *l2* for maior que *l1* quer dizer que a execução está partindo para um ponto à frente do fluxo de execução, isso ocorre em estruturas como *if* e *switch*. A Tabela 4.11 contêm quatro exemplos de *bytecode* de comparações.

Uma vez que laços necessitam de um tratamento especial, já que em algum momento precisa existir repetição de código, será adotado um modelo onde toda a ocorrência de comando de repetição vai definir um processo de comportamento recursivo.

Durante o desenvolvimento desse mapeamento foi encontrada uma dificuldade com relação à tomanda de decisão do desvio. Já que o resultado da comparação precisa ser computado, inicialmente ele era empilhado e depois utilizado em uma comparação no código CSP# que definiria quais instruções seriam utilizadas. Como dito anteriormente a ferramenta apresentou uma falha ao chavear entre processos.. Tal falha fazia com que o valor da pilha fosse perdido caso acontecesse um chaveamento. Para contornar esse problema, foram implementadas variáveis especiais que guardam o retorno da computação das comparações em cada processo.

A definição de quais os código serão executados caso a condição seja satisfeita vai depender do retorno da comparação e do tipo de estrutura a qual ela pertence. Se a estrutura for um laço, todo o código que o compreende será executado caso a condição seja satisfeita, onde a variável de retorno da comparação possui valor “1". Ao final da execução de todo o código do laço, precisa existir uma chamada recursiva do processo que o define. Caso a condição não seja satisfeita, geralemente um evento *SKIP* estará presente indicando a finalização da operação.

Para operações que simulam o comportamento do comando *if*, os códigos *bytecode* normalmente realizam o desvio para um *label* com valor posterior ao do *opcode*. Para essa situação (*l2>l1*) todo o código contido entre *l1* e *l2* só será executado caso a condição não seja satisfeita. Caso o contrário a execução irá realizar *jump* para *l2*.

Tabela 4.11: Mapeamento de algumas Instruções de Comparação.
[Fonte: Elaboração própria]

Instrução	Mapeamento CSP#

l1 if_eq l2	O_Object_S_Scope_m_method_CompEQ{
	var v1 = O_Object_S_Scope_m_method_Stack.Pop();
	if (v1 == 0) {
	O_Object_S_Scope_m_method_comparison_result = 1;
	} else {
	O_Object_S_Scope_m_method_comparison_result = 0;
	}
	} -> if (O_Object_S_Scope_m_method_comparison_result != 0) {
	[se l2 > l1 código CSP# após l2 senão se l2 < l1 código CSP# de l2 à l1]
	} else {
	[código CSP# após l1]
	}

l1 if_ne l2	O_Object_S_Scope_m_method_CompNE{
	var v1 = O_Object_S_Scope_m_method_Stack.Pop();
	if (v1 != 0) {
	O_Object_S_Scope_m_method_comparison_result = 1;
	} else {
	O_Object_S_Scope_m_method_comparison_result = 0;
	}
	} -> if (O_Object_S_Scope_m_method_comparison_result != 0) {
	[se l2 > l1 código CSP# após l2 senão se l2 < l1 código CSP# de l2 à l1]
	} else {
	[código CSP# após l1]
	}

l1 if_icmplt l2	O_Object_S_Scope_m_method_CompEQ2{
	var v2 = O_Object_S_Scope_m_method_Stack.Pop();
	var v1 = O_Object_S_Scope_m_method_Stack.Pop();
	if (v1 < v2) {
	O_Object_S_Scope_m_method_comparison_result = 1;
	} else {
	O_Object_S_Scope_m_method_comparison_result = 0;
	}
	} -> if (O_Object_S_Scope_m_method_comparison_result != 0) {
	[se l2 > l1 código CSP# após l2 senão se l2 < l1 código CSP# de l2 à l1]
	} else {
	[código CSP# após l1]
	}

l1 if_icmpne l2	O_Object_S_Scope_m_method_CompNE2{
	var v2 = O_Object_S_Scope_m_method_Stack.Pop();
	var v1 = O_Object_S_Scope_m_method_Stack.Pop();
	if (v1 != v2) {
	O_Object_S_Scope_m_method_comparison_result = 1;
	} else {
	O_Object_S_Scope_m_method_comparison_result = 0;
	}
	} -> if (O_Object_S_Scope_m_method_comparison_result != 0) {
	[se l2 > l1 código CSP# após l2 senão se l2 < l1 código CSP# de l2 à l1]
	} else {
	[código CSP# após l1]
	}

4.1.3.8 RB8: Mapeamento de desvios incondicionais

Os desvios incondicionais possuirão mapeamento semelhante ao mapeamento de comparações com relação aos *labels*. O desvio não gera novos processos, ele apenas aponta a aplicação das regras para o ponto do *label l2*, como pode-se observar na Tabela 4.12

Tabela 4.12: Mapeamento de Desvios Incondicionais.
[Fonte: Elaboração própria]

Instrução	**Mapeamento CSP#**
l1 goto l2	*[código CSP# a partir de l2]*
l1 goto 0	O_Object_S_Scope_m_method() *[geralemente chamada recursiva]*

4.1.3.9 RB9: Chamada de método já mapeado

Métodos previamente mapeados podem ser chamados dentro de outros processos. Uma vez que estes métodos já foram mapeados como processos, sua chamada dentro do mapeamento do novo método deve ser realizado por meio de composição sequencal entre processos. A Tabela 4.13 mostra um exemplo de mapeamento de invocação de métodos já mapeados.

A JVM desempilha os valores dos parâmetros antes da invocação do método e os utiliza para executar o método. Porém como o PAT possui algumas falhas, já explicadas no capítulo 3, não foi possível realizar uma chamada de desempilhamento diretamente no local do parâmetro do processo. Foi necessária a criação de uma variável temporária para armazenar o valor do parâmetro desempilhado. Serão criadas tantas variáveis temporárias para guardar os valores dos parâmetros quanto o número máximo de parâmetros chamados em métodos mapeados.

Tabela 4.13: Mapeamento da Chamada de Método com Mapeamento já Definido.
[Fonte: Elaboração própria]

Instrução	**Mapeamento CSP#**
invokevirtual Object. method()	O_Object_S_Scope_m_method;

invokevirtual Object.
method(param)
	atomic{
	    O_Object_S_Scope_m_method_arg{
	        O_Object_S_Scope_m_method_v_varArg =
	        O_Object_S_Scope_m_method_Stack.Pop()
	    }
	    -> O_Object_S_Scope_m_method(O_Object_S_Scope_m_method_v_varArg)
	}

4.1.3.10 RB10: Paralelismo entre Threads

O paralelismo entre *threads* será representado pelo *interleaving* entre processos já mapeados. A Tabela 4.14 mostra um exemplo de paralelismo entre *threads*. As duas instruções

apresentadas na tabela, quando chamam diferentes *threads* dentro de um método qualquer, vão representar dois processos que atuam intercaladamente.

Tabela 4.14: Mapeamento de Paralelismo entre Processos.
[Fonte: Elaboração própria]

Instrução	Mapeamento CSP#
invokevirtual java.lang.Thread.start() . . invokevirtual java.lang.Thread.start()	C1_Object_S1_Scope_m_method() ||| C2_Object_S2_Scope_m_method()

4.1.3.11 RB11: Mapeamento de códigos não considerados

Os *bytecodes* que realizam operações que serão abstraidas serão desconsiderados do mapeamento. Esses comandos possuem desde verificação de tipo até empilhamento de variáveis de tipo diferente de *int*. Na Tabela 4.15 serão apresentados os comandos *bytecode*, de maior ocorrência em código, e o motivo explicando porque ele não será mapeado.

Tabela 4.15: Mapeamento de Instruções não Utilizadas.
[Fonte: Elaboração própria]

Instruções	Motivo
aload_0 [this]	Carrega *this* na pilha de operandos. Não será considerado pois objetos são referênciados pelos seus nomes.
invokespecial java.package.Class()	Em CSP# o construtor padrão é automaticamente chamado, por isso não precisa de mapeamento.
putfield Class.ins : java.package.Class	Insere um objeto na pilha de operandos. Não precisa ser mapeado pois já existe uma variável CSP# que o representa.
getfield Class.ins : java.package.Class	Resgata um objeto da pilha de operandos. Não precisa ser mapeado pois já existe uma variável CSP# que o representa.
invokestatic java.package.Class.method() : java.package.Class	Invocações de métodos estáticos de classe não serão mapeadas pois já existem nos objetos instanciados através das bibliotecas C#. Isso não se aplica a listas.
invokevirtual java.package.Class.method() : java.package.Class	Invocações de métodos virtuais de classe não serão mapeadas pois já existem nos objetos instanciados através das bibliotecas C#. Isso não se aplica a listas.
checkcast java.lang.Integer	*Casts* não serão mapeados pois somente serão utilizadas variáveis inteiras.
ldc <String> ou ldc <float>	Não serão mapeadas pois somente serão utilizadas variáveis inteiras.
dup	Duplica o valor do topo da pilha. Não é necessário o mapeamento pois já é tratado dentro das invocações de métodos

4.1.3.12 RB12: Mapeamento de operações aritméticas

Cálculos matemáticos também estão presente em comando *bytecode*. Nos casos mais comuns: adição, subtração, multiplicação e divisão, serão desempilhados dois valores armazenados na pilha de operandos. Após o desempilhamento será realizada a operação selecionada entre

eles e seu resultado sera inserido novamente na pilha. A JVM realiza a otimização de algumas operações, como incremento de valores inteiros por exemplo. Nestes casos será desempilhado somente o valor correspondido pela variável a ser incrementada.

A instrução de incremento (*iinc*) possui dois parâmetros adotados: N e i. O primeiro é o valor do índice do array de variáveis locais, onde está armazenado o valor da variável (isso é abstraido neste trabalho). O segundo é o valor do incremento que pode receber valores tanto positivos quanto negativos. A Tabela 4.16 apresenta as quatro operações básicas e o incremento.

Tabela 4.16: Mapeamento de Instruções Aritméticas.
[Fonte: Elaboração própria]

Instrução	**Mapeamento CSP#**

iadd	O_Object_S_Scope_m_method_SumValues{
	var v2 = O_Object_S_Scope_m_method_Stack.Pop();
	var v1 = O_Object_S_Scope_m_method_Stack.Pop();
	var v3 = v1 + v2;
	O_Object_S_Scope_m_method_Stack.Push(v3)
	}

isub	O_Object_S_Scope_m_method_SubValues{
	var v2 = O_Object_S_Scope_m_method_Stack.Pop();
	var v1 = O_Object_S_Scope_m_method_Stack.Pop();
	var v3 = v1 - v2;
	O_Object_S_Scope_m_method_Stack.Push(v3)
	}

imul	O_Object_S_Scope_m_method_MulValues{
	var v2 = O_Object_S_Scope_m_method_Stack.Pop();
	var v1 = O_Object_S_Scope_m_method_Stack.Pop();
	var v3 = v1 * v2;
	O_Object_S_Scope_m_method_Stack.Push(v3)
	}

idiv	O_Object_S_Scope_m_method_DivValues{
	var v2 = O_Object_S_Scope_m_method_Stack.Pop();
	var v1 = O_Object_S_Scope_m_method_Stack.Pop();
	var v3 = v1 / v2;
	O_Object_S_Scope_m_method_Stack.Push(v3)
	}

iinc N i	O_Object_S_Scope_m_method_IncVar{
	O_Object_S_Scope_m_method_v_var = O_Object_S_Scope_m_method_v_var + i;
	}

4.1.3.13 RB13: Mapeamento de armazenamento de valores empilhados em variáveis locais

Variáveis que recebem valores possuem o seu mapeamento de acordo com a Tabela 4.17. Nela o valor obtido através da remoção do topo da pilha é armazenado em uma variável local.

Tabela 4.17: Mapeamento de Recebimento de Valores.
[Fonte: Elaboração própria]

Instrução	Mapeamento CSP#

istore_N	O_Object_S_Scope_m_method_UpdateLocalVar{
	O_Object_S_Scope_m_method_v_var =
	O_Object_S_Scope_m_method_Stack.Pop();
	}

4.2 Aplicação das Regras de Mapeamento

4.2.1 Aplicação das Regras para Definição de um Processo

Após a apresentação das regras de mapeamento é necessário realizar a conversão de um código *bytecode* que representa um método Java para um processo em CSP#. Isto servirá para que se tenha uma idéia de como foi feita a conversão de todos os códigos contidos nos dois exemplos selecionados. Abaixo é apresentado uma implementação de um método *main()*.

```
public static void main(String[] args) {
 int a = 3;
 int b = 2;
 int c = a + b;
 if ((c / 5) == 1)
  a++;
 else
  b++;
}
```

Após a compilação é possível obter o seguinte *bytecode*, correspondente ao método *main()*. Ao lado de cada instrução foram aplicadas uma ou mais regras de mapeamento.

```
public static void main(java.lang.String[] args); RG2, RG3, RG4
     0  iconst_3       RB1
     1  istore_1 [a]       RB13
     2  iconst_2       RB1
     3  istore_2 [b]       RB13
     4  iload_1 [a]      RB4
     5  iload_2 [b]      RB4
     6  iadd        RB12
     7  istore_3 [c]       RB13
     8  iload_3 [c]      RB4
     9  iconst_5       RB1
    10  idiv        RB12
    11  iconst_1       RB1
    12  if_icmpne 21       RB7
    15  iinc 1 1 [a]       RB12
    18  goto 24       RB8 depois RB3
    21  iinc 2 1 [b]       RB12
    24  return        RB3
```

Em Seguida é obtido o código CSP# através das regras:

```
var<List> Teste_Object_S_Scope_main_method_Stack;        RG3
var Teste_Object_S_Scope_main_method_a_var;        RG4, RN3
var Teste_Object_S_Scope_main_method_b_var;        RG4, RN3
var Teste_Object_S_Scope_main_method_c_var;        RG4, RN3
var Teste_Object_S_Scope_main_method_comparison_result;        RN9
Teste_Object_S_Scope_main_method() =        RG2, RN1
  Teste_Object_S_Scope_main_method_PushConstant{Teste_Object_S_Scope_main_method_Stack.Push(3)
      }  RB1
  -> Teste_Object_S_Scope_main_method_UpdateLocalVar{        RB13
    Teste_Object_S_Scope_main_method_a_var = Teste_Object_S_Scope_main_method_Stack.Pop();
  } -> Teste_Object_S_Scope_main_method_PushConstant{Teste_Object_S_Scope_main_method_Stack.
      Push(2)}  RB1
  -> Teste_Object_S_Scope_main_method_UpdateLocalVar{        RB13
    Teste_Object_S_Scope_main_method_b_var = Teste_Object_S_Scope_main_method_Stack.Pop();
  } -> Teste_Object_S_Scope_main_method_PushValue{        RB4
    Teste_Object_S_Scope_main_method_Stack.Push(Teste_Object_S_Scope_main_method_a_var)
  } -> Teste_Object_S_Scope_main_method_PushValue{        RB4
    Teste_Object_S_Scope_main_method_Stack.Push(Teste_Object_S_Scope_main_method_b_var)
  } -> Teste_Object_S_Scope_main_method_SumValues{        RB12
    var v2 = Teste_Object_S_Scope_main_method_Stack.Pop();
    var v1 = Teste_Object_S_Scope_main_method_Stack.Pop();
    var v3 = v1 + v2;
    Teste_Object_S_Scope_main_method_Stack.Push(v3)
  } -> Teste_Object_S_Scope_main_method_UpdateLocalVar{        RB13
    Teste_Object_S_Scope_main_method_c_var = Teste_Object_S_Scope_main_method_Stack.Pop();
  } -> Teste_Object_S_Scope_main_method_PushValue{        RB4
    Teste_Object_S_Scope_main_method_Stack.Push(Teste_Object_S_Scope_main_method_c_var)
  } -> Teste_Object_S_Scope_main_method_PushConstant{Teste_Object_S_Scope_main_method_Stack.
      Push(5)}  RB1
  -> Teste_Object_S_Scope_main_method_DivValues{        RB12
    var v2 = Teste_Object_S_Scope_main_method_Stack.Pop();
    var v1 = Teste_Object_S_Scope_main_method_Stack.Pop();
    var v3 = v1 / v2;
    Teste_Object_S_Scope_main_method_Stack.Push(v3)
  } -> Teste_Object_S_Scope_main_method_PushConstant{Teste_Object_S_Scope_main_method_Stack.
      Push(1)}  RB1
  -> Teste_Object_S_Scope_main_method_CompNE2{        RB7
    var v2 = Teste_Object_S_Scope_main_method_Stack.Pop();
    var v1 = Teste_Object_S_Scope_main_method_Stack.Pop();
    if (v1 != v2) {
      Teste_Object_S_Scope_main_method_comparison_result = 1;
    } else {
      Teste_Object_S_Scope_main_method_comparison_result = 0;
    }
  } -> if (Teste_Object_S_Scope_main_method_comparison_result != 0) {
    Teste_Object_S_Scope_main_method_IncVar{        RB12
      Teste_Object_S_Scope_main_method_a_var = Teste_Object_S_Scope_main_method_a_var + i;
    } -> Skip        RB8, RB3
  } else {
    Teste_Object_S_Scope_main_method_IncVar{        RB12
      Teste_Object_S_Scope_main_method_a_var = Teste_Object_S_Scope_main_method_b_var + i;
    } -> Skip };        RB3
```

4.2.2 Exemplo 1: Produtores e Consumidores

O primeiro exemplo escolhido é o do problema clássico dos produtores e consumidores. O código desenvolvido não possui erros de implementação que possam favorecer falhas de execução, ou seja, para todas as vezes que o programa rodar não existirá ocorrência de deadlock. Isto se deve ao fato que só serão considerados um produtor e um consumidor. Portanto, ao verificar o código no PAT, o mesmo deverá informar que o programa escrito em CSP# é *deadlock free* (livre de *deadlocks*).

Ao realizar as asserções pode-se verificar na Figura 4.1 que todas as suas propriedades são válidas. Portanto, pode-se dizer que as verificações em PAT corroboram o comportamento esperado do código Java.

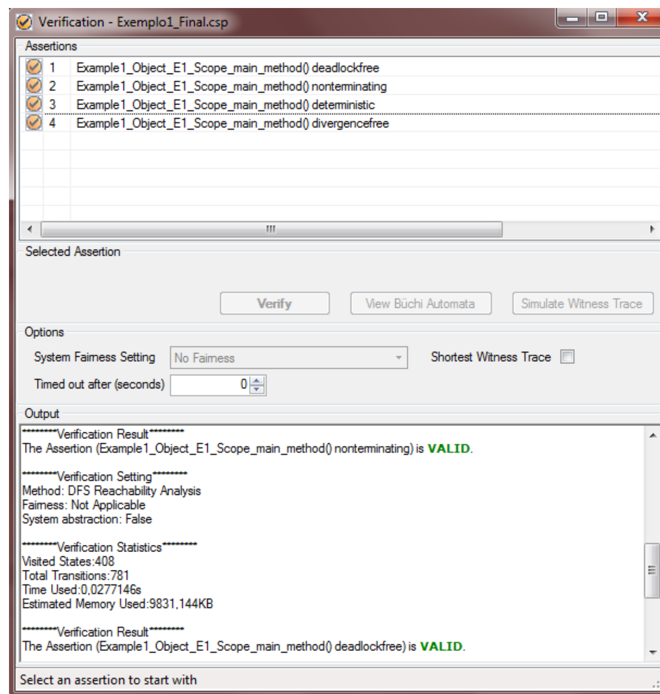

Figura 4.1: Resultado das Asserções do Exemplo 1.
[Fonte: produzido pelo autor]

1. O programa é *deadlock free* (não possui *deadlock*), pois não existe nenhum *trace* onde um processo não finalize sua execução com sucesso.

2. O programa é *nonterminating* (não termina), pois não existe nenhuma situação em que ele termine.

3. O programa é *deterministic*, pois não existem estados onde dois eventos de mesmo nome possam ser comunicados.

4. O programa é *divergence free*, pois os processos gerados pelo mapeamento proposto neste trabalho não divergem. Ou seja, não existe uma chamada recursiva, direta ou indireta, ao mesmo processo sem a comunicação de um evento visível.

4.2.3 Exemplo 2: Lista de Recursos

O segundo exemplo foi implementado, propositalmente, com falhas no código. Tais falhas permitem erros de execução do programa em ocasiões específicas. Essa indefinição se o programa terminará ou não com sucesso está relacionada com o chaveamento entre *threads*. Neste exemplo, dependendo da ordem que os recursos sejam bloqueados pelos processos, o programa pode eventualmente nunca terminal.

É interessante informar que o programa Java termina com sucesso em quase todas as vezes, o que torna o erro bastante perigoso, pois passaria despercebido pela maioria dos testes convencionais de software. A Figura 4.2 apresenta as propriedades desse exemplo.

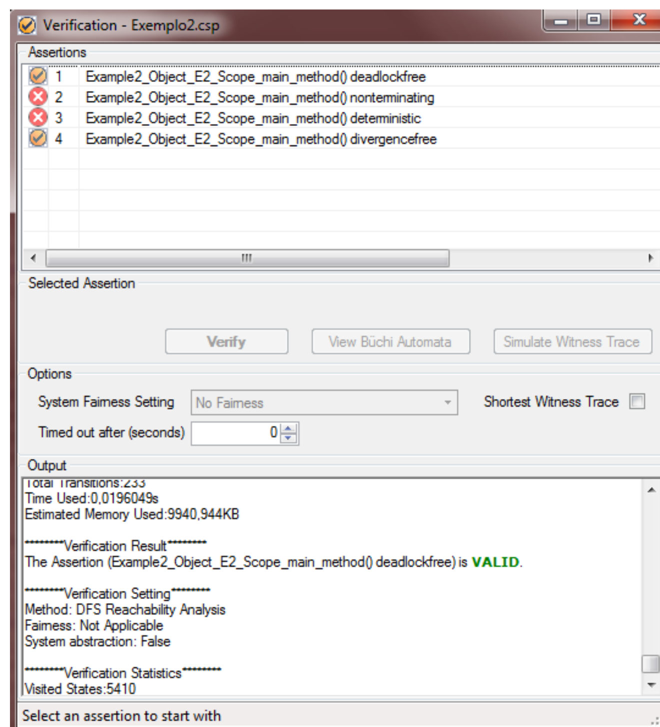

Figura 4.2: Resultado das Asserções do Exemplo 2.
[Fonte: produzido pelo autor]

1. O programa é *deadlock free* (não possui *deadlock*) pois não existe nenhum *trace* de estados onde um processo não finalize sua execução com sucesso.

2. O programa não é *nonterminating* (ou seja, ele termina) pois existe pelo menos um *trace* em que ele termina.

3. O programa não é *deterministic* pois existem processos que comunicam o mesmo evento em um mesmo estado.

4. O programa é *divergence free*, pois os processos gerados pelo mapeamento proposto neste trabalho não divergem. Ou seja, não existe uma chamada recursiva, direta ou indireta, ao mesmo processo sem a comunicação de um evento visível.

É importante ressaltar que apesar de existir uma situação em que o programa Java nunca termina, isto não implica em um *deadlock*, pois ambos os processos continuam realizando infinitamente eventos que verificam se o *lock* já foi liberado (similar a uma espera ocupada). Este comportamento não caracteriza a noção de *deadlock* utilizada por CSP: processos que em um dado momento não são capazes de comunicar nenhum evento.

As verificações padrões oferecidas por PAT não permitem verificar se existe um cenário onde o programa nunca termina. O que é possível verificar é se existe pelo menos uma situação que ele nunca termina. Para verificar esta propriedade, codificou-se, independente da aplicação das regras, o seguinte comportamento: duas variáveis foram criadas com o intuito de armazenar se um processo está bloqueado esperando que um recurso seja liberado. Quando um processo eventualmente é bloqueado, ele verifica se o outro processo já está bloqueado. Se esta condição for verdadeira, todos os processos do sistema estarão bloqueados e continuarão rodando indefinidamente para este exemplo. Neste cenário, se insere um *deadlock* artificial utilizando o processo *STOP*. Como a especificação sem esse novo comportamento era livre de *deadlock*, se aparecer um *deadlock* nessa nova implementação, é porque existe um cenário no qual os dois processos podem ficar bloqueados. Veja na Figura 4.3 o contra-exemplo gerado por PAT que demonstra esse cenário. A Figura 4.4 mostra o código que foi inserido no código gerado pelo mapeamento.

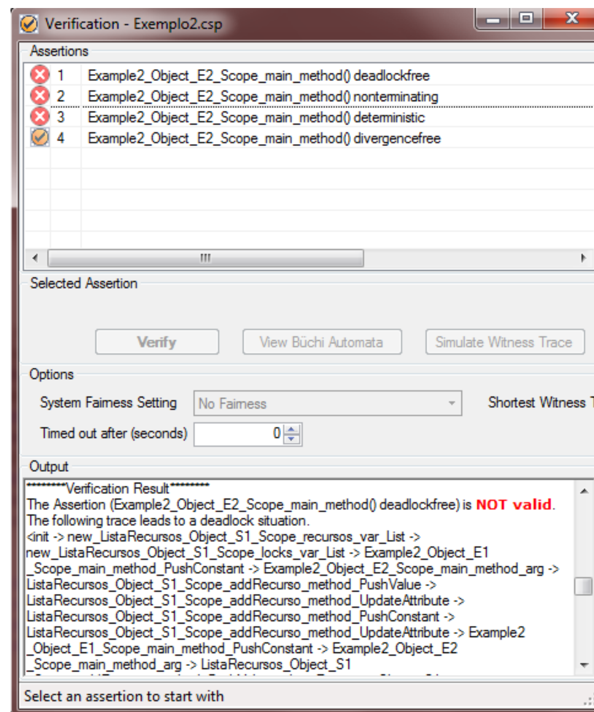

Figura 4.3: Resultado das Asserções do Exemplo 2 Simulando *Deadlock*.
[Fonte: produzido pelo autor]

```
3    // Declaração variáveis para simular deadlock
4    var P1Blocked = false;
5    var P2Blocked = false;
```

```
174  } -> ifa (ProcessoTipo1_Object_P1_Scope_run_method_L2_comparison1 != 0) {
175      // Simulação de Deadlock do Processo Tipo 1
176      atomic{
177          updateP1_state{P1Blocked = true} -> if ( P2Blocked == true ) { Stop } else { Skip }
178      } ;
179      ProcessoTipo1_Object_P1_Scope_run_method_L2()
180  } else {
181      updateP1_state{P1Blocked = false} ->
182      Skip
183  };
```

```
249  } -> ifa (ProcessoTipo2_Object_P2_Scope_run_method_L2_comparison1 != 0) {
250      // Simulação de Deadlock do Processo Tipo 2
251      atomic{
252          updateP2_state{P2Blocked = true} -> if ( P1Blocked == true ) { Stop } else { Skip }
253      } ;
254      ProcessoTipo2_Object_P2_Scope_run_method_L2()
255  } else {
256      updateP2_state{P2Blocked = false} ->
257      Skip
258  };
```

Figura 4.4: Alteração do Exemplo 2 para Introduzir *Deadlock*.
[Fonte: produzido pelo autor]

5 *Considerações Finais*

Obter uma forma de tornar o processo de análise de um código concorrente mais efetivo é a principal motivação do desenvolvimento de estudo desse trabalho. A análise manual e humana de um trecho, ou de todo, código associado a uma rotina feita na linguagem Java não fornece resultados precisos. Essa forma de análise torna-se ainda menos confiável quando associada a programas concorrentes.

Com o produto de estudo desenvolvido por este projeto foi possível estabelecer uma forma de verificar formalmente software concorrente através da união entre *bytecode* Java e verificação formal de modelos. Esses modelos, quando implementados em uma linguagem como CSP#, permitem que sejam feitas verificações mais eficientes de problemas que geralmente não são visíveis para outros tipos de verificações.

A verificação formal de um modelo que simulou o funcionamento de uma JVM a partir de seus *bytecodes* conseguiu obter resultados satisfatórios nos problemas de concorrência estudados. Portanto, pode-se afirmar que desta forma este estudo alcançou os seus objetivos e respondeu o problema de pesquisa proposto inicialmente.

5.1 Trabalhos Futuros

Apesar dos resultados obtidos por este estudo, alguns trabalhos futuros são pertinentes:

1. O mapeamento obtido neste trabalho apresentou apenas uma gama básica de instruções. É necessário realizar um mapeamento de todas as instruções possíveis para que se possa traduzir qualquer conjunto de comandos *bytecode* Java para a linguagem CSP#.
2. Também é sugerida a criação de uma rotina automatizada para a captura de comandos *bytecode* de arquivos .class para posterior produção de um arquivo .csp que possa ser utilizado na ferramenta PAT.
3. Suporte a outros tipos primitivos além do que inteiro.

4. Suporte a tipos compostos de Java mais complexos do que *ArrayList*, como: *HashMap*.

5. Métodos que retornam valores sem serem *synchronized*.

6. Métodos que recebem tipos compostos.

7. Mapeamento de blocos *sychronized*.

Referências

BÖRGER, E. *Architecture design and validation methods*. Springer, 2000. ISBN 9783540649762. Disponível em: <http://books.google.com.br/books?id=wu_WHeT7Hr4C>.

CLARKE, E. Model checking. In: RAMESH, S.; SIVAKUMAR, G. (Ed.). *Foundations of Software Technology and Theoretical Computer Science*. Springer Berlin / Heidelberg, 1997, (Lecture Notes in Computer Science, v. 1346). p. 54–56. ISBN 978-3-540-63876-6. 10.1007/BFb0058022. Disponível em: <http://dx.doi.org/10.1007/BFb0058022>.

DIJKSTRA, E. W. Notes on Structured Programming. April 1970.

GARDNER, W.B.; SERRA, M. *CSP++: A Framework for Executable Specifications*. Tese (Doutorado), 2000.

HAVELUND, K.; PRESSBURGER T. Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, Springer, v. 2, p. 366–381, 2000.

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

LEROY, X. Java bytecode verification: An overview. In: BERRY, G.; COMON, H.; FINKEL, A. (Ed.). *Computer Aided Verification*. Springer Berlin / Heidelberg, 2001, (Lecture Notes in Computer Science, v. 2102). p. 265–285. ISBN 978-3-540-42345-4. 10.1007/3-540-44585-4_26. Disponível em: <http://dx.doi.org/10.1007/3-540-44585-4_26>.

LINDHOLM, T.; YELLIN, F. *Java Virtual Machine Specification*. 2nd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0201432943.

LIU, Y.; SUN, J.; DONG, J. S. Model checking csp revisited: Introducing a process analysis toolkit. In: *In ISoLA 2008*. [S.l.]: Springer, 2008. p. 307–322.

MARCINIAK, J. J. *Encyclopedia of software engineering*. [S.l.]: Wiley, 1994. 1327-1358 p.

MILLER, E. F. *Tutorial: Software Testing & Validation Techniques*. [S.l.]: IEEE Computer Society Press, 1981. 4–16 p.

OAKS, S.; WONG, H. *Java Threads, Second Edition*. 2nd. ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1999. ISBN 1565924185.

SCHNEIDER, S. *Concurrent and Real Time Systems: The CSP Approach*. 1st. ed. New York, NY, USA: John Wiley & Sons, Inc., 1999. ISBN 0471623733.

VENNERS, B. *Inside the Java Virtual Machine*. 1st. ed. [S.l.]: McGraw-Hill Professional, 1999. ISBN 0071350934.

WELCH, P.; MARTIN, J. A csp model for java multithreading. In: *Software Engineering for Parallel and Distributed Systems, 2000. Proceedings. International Symposium on*. [S.l.: s.n.], 2000. p. 114 –122.

WOLPER, P. An introduction to model checking. In: *Proc. of the Software Quality Week (SQW'95)*. San Francisco: [s.n.], 1995.

APÊNDICE A – *Tabelas Complementares e Códigos dos Arquivos Fontes*

Aqui serão apresentados os fontes .java dos exemplos selecionados, juntamente aos seus respectivos mapeamentos em CSP# na forma de arquivos .csp. Também serão apresentadas tabelas com exemplos de instruções *bytecode* mais conhecidas.

A.1 Instruções Bytecode

Tabela A.1: Tabela com Bytecodes mais Comuns.
[Fonte: Elaboração própria]

Mnemônico	Operação
nop	Não realiza nenhuma operação.
if (_null, _notnull, _eq, _ne, _gt, _lt, _icmpeq, _icmpne)	Se uma condição for verdade realiza um *jump* para a *tag* passada como parâmetro.
goto	Realiza um *jump* para a *tag* passada como parâmetro.
return e *xreturn*	Retorna o valor do topo da lista (de tipo *x*) para o evento chamador.
add	Desempilha dois elementos e empilha o resultado da soma entre eles.
sub	Desempilha dois elementos e empilha o resultado da subtração entre eles.
mult	Desempilha dois elementos e empilha o resultado da multiplicação entre eles.
div	Desempilha dois elementos e empilha o resultado da divisão entre eles.
rem	Desempilha dois elementos e empilha o valor correspondente ao resto da divisão entre eles.
new	Cria um novo objeto e o insere no topo da pilha.
getfield, setfield, getstatic, setstatic	Resgata ou seta o valor do campo de objeto referenciado pelo valor do topo da pilha.
invokevirtual, invokestatic, invokespecial, invokeinterface	São invocadores de métodos. O primeiro é para a chamada de métodos que estão sujeitos à dynamic binding, o segundo para métodos estáticos, o terceiro para construtores e o último para métodos implementados de uma interface.
castclass, instanceof	Realizam *cast*. Caso a correspondência seja confirmada, um valor booleano *true* é empilhado na pilha. Caso contrário é lançada uma exceção de *cast* (na instrução *castclass*) ou é empurrado um valor *false* na pilha (na instrução *instanceof*).

A.2 Exemplo 1

A.2.1 Java

Example1.java

```
import java.util.ArrayList;

// Este exemplo nÃ£o apresenta problemas de deadlock e a fila se comporta ok

class Fila {

 public int tamanho_maximo = 0;
 public ArrayList<Integer> fila = null;

 public Fila(int tamanho) {
  this.tamanho_maximo = tamanho;
  this.fila = new ArrayList<Integer>();
 }

 public void inserir(int valor) {
  this.fila.add(valor);
 }

 public synchronized int remover() {
  return this.fila.remove(0);
 }

 public synchronized void filaCheia() {
  if ( this.tamanho_maximo == this.fila.size() ) {
   System.out.println("Produtor esperando");
  } else {
   System.out.println("Produtor produzindo");
   this.inserir(1);
  }
 }

 public synchronized void filaVazia() {
  if ( this.fila.size() == 0 ) {
   System.out.println("Consumidor esperando");
  } else {
   System.out.println("Consumidor consumindo");
   this.remover();
  }
 }

}

class Produtor implements Runnable {

 public Fila fila = null;

 public Produtor(Fila fila) {
```

```
    this.fila = fila;
  }

  @Override
  public void run() {
    while ( true ) {
      this.fila.filaCheia();
    }
  }

}

class Consumidor implements Runnable {

  public Fila fila = null;

  public Consumidor(Fila fila) {
    this.fila = fila;
  }

  @Override
  public void run() {
    while ( true ) {
      this.fila.filaVazia();
    }
  }

}

public class Example1 {

  public static void main(String[] args) {
    Fila f = new Fila(2);
    Produtor p1 = new Produtor(f);
    Consumidor c1 = new Consumidor(f);

    Thread t1 = new Thread(p1);
    Thread t2 = new Thread(c1);

    t1.start();
    t2.start();
  }

}
```

A.2.2 *Bytecode*

```
Bytecode Exemplo 1

// Compiled from Example1.java (version 1.6 : 50.0, super bit)
public class Example1 {

  // Method descriptor #6 ()V
  // Stack: 1, Locals: 1
```

```
public Example1();
  0  aload_0 [this]
  1  invokespecial java.lang.Object() [8]
  4  return
    Line numbers:
      [pc: 0, line: 75]
    Local variable table:
      [pc: 0, pc: 5] local: this index: 0 type: Example1

// Method descriptor #15 ([Ljava/lang/String;)V
// Stack: 3, Locals: 6
public static void main(java.lang.String[] args);
   0  new Fila [16]
   3  dup
   4  iconst_2
   5  invokespecial Fila(int) [18]
   8  astore_1 [f]
   9  new Produtor [21]
  12  dup
  13  aload_1 [f]
  14  invokespecial Produtor(Fila) [23]
  17  astore_2 [p1]
  18  new Consumidor [26]
  21  dup
  22  aload_1 [f]
  23  invokespecial Consumidor(Fila) [28]
  26  astore_3 [c1]
  27  new java.lang.Thread [29]
  30  dup
  31  aload_2 [p1]
  32  invokespecial java.lang.Thread(java.lang.Runnable) [31]
  35  astore 4 [t1]
  37  new java.lang.Thread [29]
  40  dup
  41  aload_3 [c1]
  42  invokespecial java.lang.Thread(java.lang.Runnable) [31]
  45  astore 5 [t2]
  47  aload 4 [t1]
  49  invokevirtual java.lang.Thread.start() : void [34]
  52  aload 5 [t2]
  54  invokevirtual java.lang.Thread.start() : void [34]
  57  return
    Line numbers:
      [pc: 0, line: 78]
      [pc: 9, line: 79]
      [pc: 18, line: 80]
      [pc: 27, line: 82]
      [pc: 37, line: 83]
      [pc: 47, line: 85]
      [pc: 52, line: 86]
      [pc: 57, line: 87]
    Local variable table:
      [pc: 0, pc: 58] local: args index: 0 type: java.lang.String[]
      [pc: 9, pc: 58] local: f index: 1 type: Fila
```

```
        [pc: 18, pc: 58] local: p1 index: 2 type: Produtor
        [pc: 27, pc: 58] local: c1 index: 3 type: Consumidor
        [pc: 37, pc: 58] local: t1 index: 4 type: java.lang.Thread
        [pc: 47, pc: 58] local: t2 index: 5 type: java.lang.Thread
}

// Compiled from Example1.java (version 1.6 : 50.0, super bit)
class Fila {

  // Field descriptor #6 I
  public int tamanho_maximo;

  // Field descriptor #8 Ljava/util/ArrayList;
  // Signature: Ljava/util/ArrayList<Ljava/lang/Integer;>;
  public java.util.ArrayList fila;

  // Method descriptor #12 (I)V
  // Stack: 3, Locals: 2
  public Fila(int tamanho);
     0  aload_0 [this]
     1  invokespecial java.lang.Object() [14]
     4  aload_0 [this]
     5  iconst_0
     6  putfield Fila.tamanho_maximo : int [17]
     9  aload_0 [this]
    10  aconst_null
    11  putfield Fila.fila : java.util.ArrayList [19]
    14  aload_0 [this]
    15  iload_1 [tamanho]
    16  putfield Fila.tamanho_maximo : int [17]
    19  aload_0 [this]
    20  new java.util.ArrayList [21]
    23  dup
    24  invokespecial java.util.ArrayList() [23]
    27  putfield Fila.fila : java.util.ArrayList [19]
    30  return
      Line numbers:
        [pc: 0, line: 8]
        [pc: 4, line: 5]
        [pc: 9, line: 6]
        [pc: 14, line: 9]
        [pc: 19, line: 10]
        [pc: 30, line: 11]
      Local variable table:
        [pc: 0, pc: 31] local: this index: 0 type: Fila
        [pc: 0, pc: 31] local: tamanho index: 1 type: int

  // Method descriptor #12 (I)V
  // Stack: 2, Locals: 2
  public void inserir(int valor);
     0  aload_0 [this]
     1  getfield Fila.fila : java.util.ArrayList [19]
     4  iload_1 [valor]
     5  invokestatic java.lang.Integer.valueOf(int) : java.lang.Integer [30]
```

```
    8  invokevirtual java.util.ArrayList.add(java.lang.Object) : boolean [36]
   11  pop
   12  return
     Line numbers:
       [pc: 0, line: 14]
       [pc: 12, line: 15]
     Local variable table:
       [pc: 0, pc: 13] local: this index: 0 type: Fila
       [pc: 0, pc: 13] local: valor index: 1 type: int

// Method descriptor #42 ()I
// Stack: 2, Locals: 1
public synchronized int remover();
    0  aload_0 [this]
    1  getfield Fila.fila : java.util.ArrayList [19]
    4  iconst_0
    5  invokevirtual java.util.ArrayList.remove(int) : java.lang.Object [43]
    8  checkcast java.lang.Integer [31]
   11  invokevirtual java.lang.Integer.intValue() : int [47]
   14  ireturn
     Line numbers:
       [pc: 0, line: 18]
     Local variable table:
       [pc: 0, pc: 15] local: this index: 0 type: Fila

// Method descriptor #16 ()V
// Stack: 2, Locals: 1
public synchronized void filaCheia();
    0  aload_0 [this]
    1  getfield Fila.tamanho_maximo : int [17]
    4  aload_0 [this]
    5  getfield Fila.fila : java.util.ArrayList [19]
    8  invokevirtual java.util.ArrayList.size() : int [51]
   11  if_icmpne 25
   14  getstatic java.lang.System.out : java.io.PrintStream [54]
   17  ldc <String "Produtor esperando"> [60]
   19  invokevirtual java.io.PrintStream.println(java.lang.String) : void [62]
   22  goto 38
   25  getstatic java.lang.System.out : java.io.PrintStream [54]
   28  ldc <String "Produtor produzindo"> [68]
   30  invokevirtual java.io.PrintStream.println(java.lang.String) : void [62]
   33  aload_0 [this]
   34  iconst_1
   35  invokevirtual Fila.inserir(int) : void [70]
   38  return
     Line numbers:
       [pc: 0, line: 22]
       [pc: 14, line: 23]
       [pc: 25, line: 25]
       [pc: 33, line: 26]
       [pc: 38, line: 28]
     Local variable table:
       [pc: 0, pc: 39] local: this index: 0 type: Fila
     Stack map table: number of frames 2
```

```
        [pc: 25, same]
        [pc: 38, same]

  // Method descriptor #16 ()V
  // Stack: 2, Locals: 1
  public synchronized void filaVazia();
     0  aload_0 [this]
     1  getfield Fila.fila : java.util.ArrayList [19]
     4  invokevirtual java.util.ArrayList.size() : int [51]
     7  ifne 21
    10  getstatic java.lang.System.out : java.io.PrintStream [54]
    13  ldc <String "Consumidor esperando"> [74]
    15  invokevirtual java.io.PrintStream.println(java.lang.String) : void [62]
    18  goto 34
    21  getstatic java.lang.System.out : java.io.PrintStream [54]
    24  ldc <String "Consumidor consumindo"> [76]
    26  invokevirtual java.io.PrintStream.println(java.lang.String) : void [62]
    29  aload_0 [this]
    30  invokevirtual Fila.remover() : int [78]
    33  pop
    34  return
      Line numbers:
        [pc: 0, line: 31]
        [pc: 10, line: 32]
        [pc: 21, line: 34]
        [pc: 29, line: 35]
        [pc: 34, line: 37]
      Local variable table:
        [pc: 0, pc: 35] local: this index: 0 type: Fila
      Stack map table: number of frames 2
        [pc: 21, same]
        [pc: 34, same]
}

// Compiled from Example1.java (version 1.6 : 50.0, super bit)
class Produtor implements java.lang.Runnable {

  // Field descriptor #8 LFila;
  public Fila fila;

  // Method descriptor #10 (LFila;)V
  // Stack: 2, Locals: 2
  public Produtor(Fila fila);
     0  aload_0 [this]
     1  invokespecial java.lang.Object() [12]
     4  aload_0 [this]
     5  aconst_null
     6  putfield Produtor.fila : Fila [15]
     9  aload_0 [this]
    10  aload_1 [fila]
    11  putfield Produtor.fila : Fila [15]
    14  return
      Line numbers:
        [pc: 0, line: 45]
```

```
        [pc: 4, line: 43]
        [pc: 9, line: 46]
        [pc: 14, line: 47]
      Local variable table:
        [pc: 0, pc: 15] local: this index: 0 type: Produtor
        [pc: 0, pc: 15] local: fila index: 1 type: Fila

  // Method descriptor #14 ()V
  // Stack: 1, Locals: 1
  public void run();
     0  aload_0 [this]
     1  getfield Produtor.fila : Fila [15]
     4  invokevirtual Fila.filaCheia() : void [22]
     7  goto 0
      Line numbers:
        [pc: 0, line: 52]
        [pc: 7, line: 51]
      Local variable table:
        [pc: 0, pc: 10] local: this index: 0 type: Produtor
      Stack map table: number of frames 1
        [pc: 0, same]
}

// Compiled from Example1.java (version 1.6 : 50.0, super bit)
class Consumidor implements java.lang.Runnable {

  // Field descriptor #8 LFila;
  public Fila fila;

  // Method descriptor #10 (LFila;)V
  // Stack: 2, Locals: 2
  public Consumidor(Fila fila);
     0  aload_0 [this]
     1  invokespecial java.lang.Object() [12]
     4  aload_0 [this]
     5  aconst_null
     6  putfield Consumidor.fila : Fila [15]
     9  aload_0 [this]
    10  aload_1 [fila]
    11  putfield Consumidor.fila : Fila [15]
    14  return
      Line numbers:
        [pc: 0, line: 62]
        [pc: 4, line: 60]
        [pc: 9, line: 63]
        [pc: 14, line: 64]
      Local variable table:
        [pc: 0, pc: 15] local: this index: 0 type: Consumidor
        [pc: 0, pc: 15] local: fila index: 1 type: Fila

  // Method descriptor #14 ()V
  // Stack: 1, Locals: 1
  public void run();
     0  aload_0 [this]
```

```
      1  getfield Consumidor.fila : Fila [15]
      4  invokevirtual Fila.filaVazia() : void [22]
      7  goto 0
        Line numbers:
          [pc: 0, line: 69]
          [pc: 7, line: 68]
        Local variable table:
          [pc: 0, pc: 10] local: this index: 0 type: Consumidor
        Stack map table: number of frames 1
          [pc: 0, same]
}
```

A.2.3 CSP#

```
ï»¿Exemplo1.csp

#import "Pat.Lib.List";

var<List> Fila_Object_S1_Scope_Fila_method_Stack;
var Fila_Object_S1_Scope_tamanho_maximo_var_attribute = 0;
var<List> Fila_Object_S1_Scope_fila_var_List;
Fila_Object_S1_Scope_Fila_method(Fila_Object_S1_Scope_Fila_method_tamanho_var) =
    Fila_Object_S1_Scope_Fila_method_PushConstant{Fila_Object_S1_Scope_Fila_method_Stack.Push
    (0)}
         -> Fila_Object_S1_Scope_Fila_method_UpdateAttribute{var top =
              Fila_Object_S1_Scope_Fila_method_Stack.Pop();
              Fila_Object_S1_Scope_tamanho_maximo_var_attribute = top}
         -> Fila_Object_S1_Scope_Fila_method_PushValue{Fila_Object_S1_Scope_Fila_method_Stack.
              Push(Fila_Object_S1_Scope_Fila_method_tamanho_var)}
         -> Fila_Object_S1_Scope_Fila_method_UpdateAttribute{var top =
              Fila_Object_S1_Scope_Fila_method_Stack.Pop();
              Fila_Object_S1_Scope_tamanho_maximo_var_attribute = top}
         -> new_Fila_Object_S1_Scope_fila_var_List{Fila_Object_S1_Scope_fila_var_List = new
              List()}
         -> Skip;


var<List> Fila_Object_S1_Scope_inserir_method_Stack;
synchronized_Fila_Object_S1_Scope_inserir_method(Fila_Object_S1_Scope_inserir_method_valor_var
    ) = Fila_Object_S1_Scope_inserir_method_PushValue{
              Fila_Object_S1_Scope_inserir_method_Stack.Push(
                  Fila_Object_S1_Scope_inserir_method_valor_var)}
        -> Fila_Object_S1_Scope_inserir_method_UpdateAttribute{var top =
             Fila_Object_S1_Scope_inserir_method_Stack.Pop();
             Fila_Object_S1_Scope_fila_var_List.Add(top)}
        -> Skip;


var<List> Fila_Object_S1_Scope_remover_method_Stack;
var Fila_Object_S1_Scope_remover_method_retorno_varReturn;
synchronized_Fila_Object_S1_Scope_remover_method() =
    Fila_Object_S1_Scope_remover_method_PushConstant{Fila_Object_S1_Scope_remover_method_Stack
    .Push(0)}
```

```
     -> Fila_Object_S1_Scope_remover_method_UpdateAttribute{var top =
         Fila_Object_S1_Scope_remover_method_Stack.Pop(); var value =
         Fila_Object_S1_Scope_fila_var_List.Get(top);
         Fila_Object_S1_Scope_remover_method_Stack.Push(value);
         Fila_Object_S1_Scope_fila_var_List.RemoveAt(top)}
     -> Fila_Object_S1_Scope_remover_method_SetReturn{
         Fila_Object_S1_Scope_remover_method_retorno_varReturn =
         Fila_Object_S1_Scope_remover_method_Stack.Pop()}
     -> Skip;


var<List> Fila_Object_S1_Scope_filaCheia_method_Stack;
var synchronized_Fila_Object_S1_Scope_filaCheia_method_parametro_varArg;
var Fila_Object_S1_Scope_filaCheia_method_comparison_result;
synchronized_Fila_Object_S1_Scope_filaCheia_method() =
    Fila_Object_S1_Scope_filaCheia_method_PushValue{
    Fila_Object_S1_Scope_filaCheia_method_Stack.Push(
    Fila_Object_S1_Scope_tamanho_maximo_var_attribute)}
       -> Fila_Object_S1_Scope_filaCheia_method_PushValue{var quant =
           Fila_Object_S1_Scope_fila_var_List.Count();
           Fila_Object_S1_Scope_filaCheia_method_Stack.Push(quant)}
      -> Fila_Object_S1_Scope_filaCheia_method_CompNE2{var v2 =
          Fila_Object_S1_Scope_filaCheia_method_Stack.Pop(); var v1 =
          Fila_Object_S1_Scope_filaCheia_method_Stack.Pop();
      if (v1 != v2) {
        Fila_Object_S1_Scope_filaCheia_method_comparison_result = 1
      }else{
        Fila_Object_S1_Scope_filaCheia_method_comparison_result = 0
         }
     } -> if(Fila_Object_S1_Scope_filaCheia_method_comparison_result != 0) {
       Produtor_Object_P1_Scope_run_method_PushConstant{
           Produtor_Object_P1_Scope_run_method_Stack.Push(1)}
      -> atomic{synchronized_Fila_Object_S1_Scope_filaCheia_method_arg{
          synchronized_Fila_Object_S1_Scope_filaCheia_method_parametro_varArg =
          Produtor_Object_P1_Scope_run_method_Stack.Pop()}
        -> synchronized_Fila_Object_S1_Scope_inserir_method(
            synchronized_Fila_Object_S1_Scope_filaCheia_method_parametro_varArg)
      } ;  Skip
        } else {
      Skip
        };


var<List> Fila_Object_S1_Scope_filaVazia_method_Stack;
var Fila_Object_S1_Scope_filaVazia_method_comparison_result;
synchronized_Fila_Object_S1_Scope_filaVazia_method() =
    Fila_Object_S1_Scope_filaVazia_method_PushValue{var quant =
    Fila_Object_S1_Scope_fila_var_List.Count(); Fila_Object_S1_Scope_filaVazia_method_Stack.
    Push(quant)}
      -> Fila_Object_S1_Scope_filaVazia_method_CompNE{
      var v1 = Fila_Object_S1_Scope_filaVazia_method_Stack.Pop();
      if (v1 != 0) {
        Fila_Object_S1_Scope_filaVazia_method_comparison_result = 1
      } else {
```

```
        Fila_Object_S1_Scope_filaVazia_method_comparison_result = 0
        }
      } -> if(Fila_Object_S1_Scope_filaVazia_method_comparison_result != 0) {
     atomic{synchronized_Fila_Object_S1_Scope_remover_method()} ; Skip
      } else {
        Skip
      };


var<List> Produtor_Object_P1_Scope_run_method_Stack;
Produtor_Object_P1_Scope_run_method() = atomic{
    synchronized_Fila_Object_S1_Scope_filaCheia_method() } ;
    Produtor_Object_P1_Scope_run_method();

var<List> Consumidor_Object_C1_Scope_run_method_Stack;
Consumidor_Object_C1_Scope_run_method() = atomic{
    synchronized_Fila_Object_S1_Scope_filaVazia_method() } ;
    Consumidor_Object_C1_Scope_run_method();

var<List> Example1_Object_E1_Scope_main_method_Stack;
var Example1_Object_E1_Scope_main_method_parametro_varArg;
Example1_Object_E1_Scope_main_method() = Example1_Object_E1_Scope_main_method_PushConstant{
    Example1_Object_E1_Scope_main_method_Stack.Push(2)}
        -> atomic{Example1_Object_E1_Scope_main_method_arg{
            Example1_Object_E1_Scope_main_method_parametro_varArg =
            Example1_Object_E1_Scope_main_method_Stack.Pop()}
        -> Fila_Object_S1_Scope_Fila_method(
            Example1_Object_E1_Scope_main_method_parametro_varArg)}
        ;
        (Produtor_Object_P1_Scope_run_method() ||| Consumidor_Object_C1_Scope_run_method());


#assert Example1_Object_E1_Scope_main_method() deadlockfree;
#assert Example1_Object_E1_Scope_main_method() nonterminating;
#assert Example1_Object_E1_Scope_main_method() deterministic;
#assert Example1_Object_E1_Scope_main_method() divergencefree;
```

A.3 Exemplo 2

A.3.1 Java

```
Example2.java

import java.util.ArrayList;

// Esse caso tem um deadlock 'raro' (normalmente não acontece)

class ListaRecursos {

 public ArrayList<Integer> recursos = null;
 public ArrayList<Integer> locks = null;
```

```
 public ListaRecursos() {
  this.recursos = new ArrayList<Integer>();
  this.locks = new ArrayList<Integer>();
 }

 public void addRecurso(int valor) {
  this.recursos.add(valor);
  this.locks.add(0);
 }

 public synchronized int getSize() {
  return this.locks.size();
 }

 public synchronized boolean updateRecurso(int i) {
  if ( this.locks.get(i) == 0 ) {
   this.locks.set(i, 1);
   Integer r = this.recursos.get(i);
   this.recursos.set(i, r*r);
   return true;
  } else {
   return false;
  }
 }

 public synchronized void freeRecursos() {
  int tamanho = this.getSize();
  for (int i=0; i<tamanho; i++) {
   this.locks.set(i, 0);
  }
 }

}

class ProcessoTipo1 implements Runnable {

 private ListaRecursos recursos = null;

 public ProcessoTipo1(ListaRecursos recursos) {
  this.recursos = recursos;
 }

 @Override
 public void run() {
  int tamanho = this.recursos.getSize();
  for (int i=0; i<tamanho; i++) {
   while ( true ) {
    boolean retorno = this.recursos.updateRecurso(i);
    if ( retorno == true ) {
     break;
    }
   }
  }
```

```
    this.recursos.freeRecursos();
  }
}

class ProcessoTipo2 implements Runnable {

  private ListaRecursos recursos = null;

  public ProcessoTipo2(ListaRecursos recursos) {
    this.recursos = recursos;
  }

  @Override
  public void run() {
    int tamanho = this.recursos.getSize();
    for (int i=tamanho-1; i>=0; i--) {
      while ( true ) {
        boolean retorno = this.recursos.updateRecurso(i);
        if ( retorno == true ) {
          break;
        }
      }
    }

    this.recursos.freeRecursos();
  }
}

public class Example2 {

  public static void main(String[] args) {
    System.out.println("--");

    ListaRecursos listaRecursos = new ListaRecursos();
    listaRecursos.addRecurso(2); listaRecursos.addRecurso(3);

    ProcessoTipo1 p1 = new ProcessoTipo1(listaRecursos);
    ProcessoTipo2 p2 = new ProcessoTipo2(listaRecursos);

    Thread t1 = new Thread(p1);
    Thread t2 = new Thread(p2);

    t1.start();
    t2.start();
  }

}
```

A.3.2 *Bytecode*

```
Bytecode Exemplo 2

// Compiled from Example2.java (version 1.6 : 50.0, super bit)
public class Example2 {
```

```
// Method descriptor #6 ()V
// Stack: 1, Locals: 1
public Example2();
  0  aload_0 [this]
  1  invokespecial java.lang.Object() [8]
  4  return
    Line numbers:
      [pc: 0, line: 92]
    Local variable table:
      [pc: 0, pc: 5] local: this index: 0 type: Example2

// Method descriptor #15 ([Ljava/lang/String;)V
// Stack: 3, Locals: 6
public static void main(java.lang.String[] args);
   0  getstatic java.lang.System.out : java.io.PrintStream [16]
   3  ldc <String "-->"> [22]
   5  invokevirtual java.io.PrintStream.println(java.lang.String) : void [24]
   8  new ListaRecursos [30]
  11  dup
  12  invokespecial ListaRecursos() [32]
  15  astore_1 [listaRecursos]
  16  aload_1 [listaRecursos]
  17  iconst_2
  18  invokevirtual ListaRecursos.addRecurso(int) : void [33]
  21  aload_1 [listaRecursos]
  22  iconst_3
  23  invokevirtual ListaRecursos.addRecurso(int) : void [33]
  26  new ProcessoTipo1 [37]
  29  dup
  30  aload_1 [listaRecursos]
  31  invokespecial ProcessoTipo1(ListaRecursos) [39]
  34  astore_2 [p1]
  35  new ProcessoTipo2 [42]
  38  dup
  39  aload_1 [listaRecursos]
  40  invokespecial ProcessoTipo2(ListaRecursos) [44]
  43  astore_3 [p2]
  44  new java.lang.Thread [45]
  47  dup
  48  aload_2 [p1]
  49  invokespecial java.lang.Thread(java.lang.Runnable) [47]
  52  astore 4 [t1]
  54  new java.lang.Thread [45]
  57  dup
  58  aload_3 [p2]
  59  invokespecial java.lang.Thread(java.lang.Runnable) [47]
  62  astore 5 [t2]
  64  aload 4 [t1]
  66  invokevirtual java.lang.Thread.start() : void [50]
  69  aload 5 [t2]
  71  invokevirtual java.lang.Thread.start() : void [50]
  74  return
    Line numbers:
```

```
        [pc: 0, line: 95]
        [pc: 8, line: 97]
        [pc: 16, line: 98]
        [pc: 26, line: 100]
        [pc: 35, line: 101]
        [pc: 44, line: 103]
        [pc: 54, line: 104]
        [pc: 64, line: 106]
        [pc: 69, line: 107]
        [pc: 74, line: 108]
      Local variable table:
        [pc: 0, pc: 75] local: args index: 0 type: java.lang.String[]
        [pc: 16, pc: 75] local: listaRecursos index: 1 type: ListaRecursos
        [pc: 35, pc: 75] local: p1 index: 2 type: ProcessoTipo1
        [pc: 44, pc: 75] local: p2 index: 3 type: ProcessoTipo2
        [pc: 54, pc: 75] local: t1 index: 4 type: java.lang.Thread
        [pc: 64, pc: 75] local: t2 index: 5 type: java.lang.Thread
}

// Compiled from Example2.java (version 1.6 : 50.0, super bit)
class ListaRecursos {

  // Field descriptor #6 Ljava/util/ArrayList;
  // Signature: Ljava/util/ArrayList<Ljava/lang/Integer;>;
  public java.util.ArrayList recursos;

  // Field descriptor #6 Ljava/util/ArrayList;
  // Signature: Ljava/util/ArrayList<Ljava/lang/Integer;>;
  public java.util.ArrayList locks;

  // Method descriptor #11 ()V
  // Stack: 3, Locals: 1
  public ListaRecursos();
     0  aload_0 [this]
     1  invokespecial java.lang.Object() [13]
     4  aload_0 [this]
     5  aconst_null
     6  putfield ListaRecursos.recursos : java.util.ArrayList [15]
     9  aload_0 [this]
    10  aconst_null
    11  putfield ListaRecursos.locks : java.util.ArrayList [17]
    14  aload_0 [this]
    15  new java.util.ArrayList [19]
    18  dup
    19  invokespecial java.util.ArrayList() [21]
    22  putfield ListaRecursos.recursos : java.util.ArrayList [15]
    25  aload_0 [this]
    26  new java.util.ArrayList [19]
    29  dup
    30  invokespecial java.util.ArrayList() [21]
    33  putfield ListaRecursos.locks : java.util.ArrayList [17]
    36  return
      Line numbers:
        [pc: 0, line: 10]
```

```
      [pc: 4, line: 7]
      [pc: 9, line: 8]
      [pc: 14, line: 11]
      [pc: 25, line: 12]
      [pc: 36, line: 13]
    Local variable table:
      [pc: 0, pc: 37] local: this index: 0 type: ListaRecursos

// Method descriptor #27 (I)V
// Stack: 2, Locals: 2
public void addRecurso(int valor);
   0  aload_0 [this]
   1  getfield ListaRecursos.recursos : java.util.ArrayList [15]
   4  iload_1 [valor]
   5  invokestatic java.lang.Integer.valueOf(int) : java.lang.Integer [28]
   8  invokevirtual java.util.ArrayList.add(java.lang.Object) : boolean [34]
  11  pop
  12  aload_0 [this]
  13  getfield ListaRecursos.locks : java.util.ArrayList [17]
  16  iconst_0
  17  invokestatic java.lang.Integer.valueOf(int) : java.lang.Integer [28]
  20  invokevirtual java.util.ArrayList.add(java.lang.Object) : boolean [34]
  23  pop
  24  return
    Line numbers:
      [pc: 0, line: 16]
      [pc: 12, line: 17]
      [pc: 24, line: 18]
    Local variable table:
      [pc: 0, pc: 25] local: this index: 0 type: ListaRecursos
      [pc: 0, pc: 25] local: valor index: 1 type: int

// Method descriptor #41 ()I
// Stack: 1, Locals: 1
public synchronized int getSize();
  0  aload_0 [this]
  1  getfield ListaRecursos.locks : java.util.ArrayList [17]
  4  invokevirtual java.util.ArrayList.size() : int [42]
  7  ireturn
    Line numbers:
      [pc: 0, line: 21]
    Local variable table:
      [pc: 0, pc: 8] local: this index: 0 type: ListaRecursos

// Method descriptor #46 (I)Z
// Stack: 4, Locals: 3
public synchronized boolean updateRecurso(int i);
   0  aload_0 [this]
   1  getfield ListaRecursos.locks : java.util.ArrayList [17]
   4  iload_1 [i]
   5  invokevirtual java.util.ArrayList.get(int) : java.lang.Object [47]
   8  checkcast java.lang.Integer [29]
  11  invokevirtual java.lang.Integer.intValue() : int [51]
  14  ifne 65
```

```
    17  aload_0 [this]
    18  getfield ListaRecursos.locks : java.util.ArrayList [17]
    21  iload_1 [i]
    22  iconst_1
    23  invokestatic java.lang.Integer.valueOf(int) : java.lang.Integer [28]
    26  invokevirtual java.util.ArrayList.set(int, java.lang.Object) : java.lang.Object [54]
    29  pop
    30  aload_0 [this]
    31  getfield ListaRecursos.recursos : java.util.ArrayList [15]
    34  iload_1 [i]
    35  invokevirtual java.util.ArrayList.get(int) : java.lang.Object [47]
    38  checkcast java.lang.Integer [29]
    41  astore_2 [r]
    42  aload_0 [this]
    43  getfield ListaRecursos.recursos : java.util.ArrayList [15]
    46  iload_1 [i]
    47  aload_2 [r]
    48  invokevirtual java.lang.Integer.intValue() : int [51]
    51  aload_2 [r]
    52  invokevirtual java.lang.Integer.intValue() : int [51]
    55  imul
    56  invokestatic java.lang.Integer.valueOf(int) : java.lang.Integer [28]
    59  invokevirtual java.util.ArrayList.set(int, java.lang.Object) : java.lang.Object [54]
    62  pop
    63  iconst_1
    64  ireturn
    65  iconst_0
    66  ireturn
      Line numbers:
        [pc: 0, line: 25]
        [pc: 17, line: 26]
        [pc: 30, line: 27]
        [pc: 42, line: 28]
        [pc: 63, line: 29]
        [pc: 65, line: 31]
      Local variable table:
        [pc: 0, pc: 67] local: this index: 0 type: ListaRecursos
        [pc: 0, pc: 67] local: i index: 1 type: int
        [pc: 42, pc: 65] local: r index: 2 type: java.lang.Integer
      Stack map table: number of frames 1
        [pc: 65, same_extended]

  // Method descriptor #11 ()V
  // Stack: 3, Locals: 3
  public synchronized void freeRecursos();
     0  aload_0 [this]
     1  invokevirtual ListaRecursos.getSize() : int [63]
     4  istore_1 [tamanho]
     5  iconst_0
     6  istore_2 [i]
     7  goto 26
    10  aload_0 [this]
    11  getfield ListaRecursos.locks : java.util.ArrayList [17]
    14  iload_2 [i]
```

```
    15  iconst_0
    16  invokestatic java.lang.Integer.valueOf(int) : java.lang.Integer [28]
    19  invokevirtual java.util.ArrayList.set(int, java.lang.Object) : java.lang.Object [54]
    22  pop
    23  iinc 2 1 [i]
    26  iload_2 [i]
    27  iload_1 [tamanho]
    28  if_icmplt 10
    31  return
      Line numbers:
        [pc: 0, line: 36]
        [pc: 5, line: 37]
        [pc: 10, line: 38]
        [pc: 23, line: 37]
        [pc: 31, line: 40]
      Local variable table:
        [pc: 0, pc: 32] local: this index: 0 type: ListaRecursos
        [pc: 5, pc: 32] local: tamanho index: 1 type: int
        [pc: 7, pc: 31] local: i index: 2 type: int
      Stack map table: number of frames 2
        [pc: 10, append: {int, int}]
        [pc: 26, same]
}

// Compiled from Example2.java (version 1.6 : 50.0, super bit)
class ProcessoTipo1 implements java.lang.Runnable {

  // Field descriptor #8 LListaRecursos;
  private ListaRecursos recursos;

  // Method descriptor #10 (LListaRecursos;)V
  // Stack: 2, Locals: 2
  public ProcessoTipo1(ListaRecursos recursos);
     0  aload_0 [this]
     1  invokespecial java.lang.Object() [12]
     4  aload_0 [this]
     5  aconst_null
     6  putfield ProcessoTipo1.recursos : ListaRecursos [15]
     9  aload_0 [this]
    10  aload_1 [recursos]
    11  putfield ProcessoTipo1.recursos : ListaRecursos [15]
    14  return
      Line numbers:
        [pc: 0, line: 48]
        [pc: 4, line: 46]
        [pc: 9, line: 49]
        [pc: 14, line: 50]
      Local variable table:
        [pc: 0, pc: 15] local: this index: 0 type: ProcessoTipo1
        [pc: 0, pc: 15] local: recursos index: 1 type: ListaRecursos

  // Method descriptor #14 ()V
  // Stack: 2, Locals: 4
  public void run();
```

```
     0  aload_0 [this]
     1  getfield ProcessoTipo1.recursos : ListaRecursos [15]
     4  invokevirtual ListaRecursos.getSize() : int [22]
     7  istore_1 [tamanho]
     8  iconst_0
     9  istore_2 [i]
    10  goto 29
    13  aload_0 [this]
    14  getfield ProcessoTipo1.recursos : ListaRecursos [15]
    17  iload_2 [i]
    18  invokevirtual ListaRecursos.updateRecurso(int) : boolean [28]
    21  istore_3 [retorno]
    22  iload_3 [retorno]
    23  ifeq 13
    26  iinc 2 1 [i]
    29  iload_2 [i]
    30  iload_1 [tamanho]
    31  if_icmplt 13
    34  aload_0 [this]
    35  getfield ProcessoTipo1.recursos : ListaRecursos [15]
    38  invokevirtual ListaRecursos.freeRecursos() : void [32]
    41  return
      Line numbers:
        [pc: 0, line: 54]
        [pc: 8, line: 55]
        [pc: 13, line: 57]
        [pc: 22, line: 58]
        [pc: 26, line: 55]
        [pc: 34, line: 64]
        [pc: 41, line: 65]
      Local variable table:
        [pc: 0, pc: 42] local: this index: 0 type: ProcessoTipo1
        [pc: 8, pc: 42] local: tamanho index: 1 type: int
        [pc: 10, pc: 34] local: i index: 2 type: int
        [pc: 22, pc: 26] local: retorno index: 3 type: boolean
      Stack map table: number of frames 2
        [pc: 13, append: {int, int}]
        [pc: 29, same]
}

// Compiled from Example2.java (version 1.6 : 50.0, super bit)
class ProcessoTipo2 implements java.lang.Runnable {

  // Field descriptor #8 LListaRecursos;
  private ListaRecursos recursos;

  // Method descriptor #10 (LListaRecursos;)V
  // Stack: 2, Locals: 2
  public ProcessoTipo2(ListaRecursos recursos);
     0  aload_0 [this]
     1  invokespecial java.lang.Object() [12]
     4  aload_0 [this]
     5  aconst_null
     6  putfield ProcessoTipo2.recursos : ListaRecursos [15]
```

```
      9  aload_0 [this]
     10  aload_1 [recursos]
     11  putfield ProcessoTipo2.recursos : ListaRecursos [15]
     14  return
       Line numbers:
         [pc: 0, line: 72]
         [pc: 4, line: 70]
         [pc: 9, line: 73]
         [pc: 14, line: 74]
       Local variable table:
         [pc: 0, pc: 15] local: this index: 0 type: ProcessoTipo2
         [pc: 0, pc: 15] local: recursos index: 1 type: ListaRecursos

  // Method descriptor #14 ()V
  // Stack: 2, Locals: 4
  public void run();
      0  aload_0 [this]
      1  getfield ProcessoTipo2.recursos : ListaRecursos [15]
      4  invokevirtual ListaRecursos.getSize() : int [22]
      7  istore_1 [tamanho]
      8  iload_1 [tamanho]
      9  iconst_1
     10  isub
     11  istore_2 [i]
     12  goto 31
     15  aload_0 [this]
     16  getfield ProcessoTipo2.recursos : ListaRecursos [15]
     19  iload_2 [i]
     20  invokevirtual ListaRecursos.updateRecurso(int) : boolean [28]
     23  istore_3 [retorno]
     24  iload_3 [retorno]
     25  ifeq 15
     28  iinc 2 -1 [i]
     31  iload_2 [i]
     32  ifge 15
     35  aload_0 [this]
     36  getfield ProcessoTipo2.recursos : ListaRecursos [15]
     39  invokevirtual ListaRecursos.freeRecursos() : void [32]
     42  return
       Line numbers:
         [pc: 0, line: 78]
         [pc: 8, line: 79]
         [pc: 15, line: 81]
         [pc: 24, line: 82]
         [pc: 28, line: 79]
         [pc: 35, line: 88]
         [pc: 42, line: 89]
       Local variable table:
         [pc: 0, pc: 43] local: this index: 0 type: ProcessoTipo2
         [pc: 8, pc: 43] local: tamanho index: 1 type: int
         [pc: 12, pc: 35] local: i index: 2 type: int
         [pc: 24, pc: 28] local: retorno index: 3 type: boolean
       Stack map table: number of frames 2
         [pc: 15, append: {int, int}]
```

```
        [pc: 31, same]
}
```

A.3.3 CSP#

```
ï»¿Exemplo2.csp

#import "Pat.Lib.List";

// DeclaraÃ§Ã£o variÃ¡veis para simular deadlock
var P1Blocked = false;
var P2Blocked = false;

var<List> ListaRecursos_Object_S1_Scope_recursos_var_List;
var<List> ListaRecursos_Object_S1_Scope_locks_var_List;

var<List> ListaRecursos_Object_S1_Scope_ListaRecursos_method_Stack;
ListaRecursos_Object_S1_Scope_ListaRecursos_method() =
    new_ListaRecursos_Object_S1_Scope_recursos_var_List{
    ListaRecursos_Object_S1_Scope_recursos_var_List = new List()}
        -> new_ListaRecursos_Object_S1_Scope_locks_var_List{
            ListaRecursos_Object_S1_Scope_locks_var_List = new List()}
      -> Skip;


var<List> ListaRecursos_Object_S1_Scope_addRecurso_method_Stack;
ListaRecursos_Object_S1_Scope_addRecurso_method(
    ListaRecursos_Object_S1_Scope_addRecurso_method_valor_var) =
    ListaRecursos_Object_S1_Scope_addRecurso_method_PushValue{
    ListaRecursos_Object_S1_Scope_addRecurso_method_Stack.Push(
    ListaRecursos_Object_S1_Scope_addRecurso_method_valor_var)}
        -> ListaRecursos_Object_S1_Scope_addRecurso_method_UpdateAttribute{var top =
            ListaRecursos_Object_S1_Scope_addRecurso_method_Stack.Pop();
            ListaRecursos_Object_S1_Scope_recursos_var_List.Add(top)}
        -> ListaRecursos_Object_S1_Scope_addRecurso_method_PushConstant{
            ListaRecursos_Object_S1_Scope_addRecurso_method_Stack.Push(0)}
        -> ListaRecursos_Object_S1_Scope_addRecurso_method_UpdateAttribute{var top =
            ListaRecursos_Object_S1_Scope_addRecurso_method_Stack.Pop();
            ListaRecursos_Object_S1_Scope_locks_var_List.Add(top)}
        -> Skip;


var<List> ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack;
var ListaRecursos_Object_S1_Scope_updateRecurso_method_r_var;
var ListaRecursos_Object_S1_Scope_updateRecurso_method_retorno_varReturn;
var ListaRecursos_Object_S1_Scope_updateRecurso_method_comparison_result1;
synchronized_ListaRecursos_Object_S1_Scope_updateRecurso_method(
    ListaRecursos_Object_S1_Scope_updateRecurso_method_i_var) =
 ListaRecursos_Object_S1_Scope_updateRecurso_method_PushValue{
     ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Push(
     ListaRecursos_Object_S1_Scope_updateRecurso_method_i_var)}
    -> ListaRecursos_Object_S1_Scope_updateRecurso_method_GetItemList{var top =
        ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Pop(); var value =
        ListaRecursos_Object_S1_Scope_locks_var_List.Get(top);
```

```
        ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Push(value);}
 -> ListaRecursos_Object_S1_Scope_updateRecurso_method_L2_CompNE{
   var v1 = ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Pop();
   if (v1 != 0) {
    ListaRecursos_Object_S1_Scope_updateRecurso_method_comparison_result1 = 1;
   } else {
    ListaRecursos_Object_S1_Scope_updateRecurso_method_comparison_result1 = 0;
   }
 } -> if (ListaRecursos_Object_S1_Scope_updateRecurso_method_comparison_result1 != 0) {
  ListaRecursos_Object_S1_Scope_updateRecurso_PushConstant{
      ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Push(0)}
  -> ListaRecursos_Object_S1_Scope_updateRecurso_SetReturn{
      ListaRecursos_Object_S1_Scope_updateRecurso_method_retorno_varReturn =
      ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Pop()}
  -> Skip
 } else {
  ListaRecursos_Object_S1_Scope_updateRecurso_method_PushValue{
      ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Push(
      ListaRecursos_Object_S1_Scope_updateRecurso_method_i_var)}
  -> ListaRecursos_Object_S1_Scope_updateRecurso_method_PushValue{
      ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Push(1)}
  -> ListaRecursos_Object_S1_Scope_updateRecurso_method_UpdateAttribute{
    var p2 = ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Pop();
    var p1 = ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Pop();
    ListaRecursos_Object_S1_Scope_locks_var_List.Set(p1,p2);
  } -> ListaRecursos_Object_S1_Scope_updateRecurso_method_PushValue{
      ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Push(
      ListaRecursos_Object_S1_Scope_updateRecurso_method_i_var)}
  -> ListaRecursos_Object_S1_Scope_updateRecurso_method_GetItemList{var top =
      ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Pop(); var value =
      ListaRecursos_Object_S1_Scope_recursos_var_List.Get(top);
      ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Push(value)}
  -> ListaRecursos_Object_S1_Scope_updateRecurso_method_UpdateLocalVar{
    ListaRecursos_Object_S1_Scope_updateRecurso_method_r_var =
        ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Pop();
  } -> ListaRecursos_Object_S1_Scope_updateRecurso_method_PushValue{
      ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Push(
      ListaRecursos_Object_S1_Scope_updateRecurso_method_i_var)}
  -> ListaRecursos_Object_S1_Scope_updateRecurso_method_PushValue{
      ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Push(
      ListaRecursos_Object_S1_Scope_updateRecurso_method_r_var)}
  -> ListaRecursos_Object_S1_Scope_updateRecurso_method_PushValue{
      ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Push(
      ListaRecursos_Object_S1_Scope_updateRecurso_method_r_var)}
  -> ListaRecursos_Object_S1_Scope_updateRecurso_method_MulValues{
   var v2 = ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Pop();
   var v1 = ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Pop();
   var v3 = v1 * v2;
   ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Push(v3);
  } -> ListaRecursos_Object_S1_Scope_updateRecurso_method_UpdateAttribute {
   var p2 = ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Pop();
   var p1 = ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Pop();
   ListaRecursos_Object_S1_Scope_recursos_var_List.Set(p1,p2);
```

```
  } -> ListaRecursos_Object_S1_Scope_updateRecurso_PushConstant{
      ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Push(1)}
  -> ListaRecursos_Object_S1_Scope_updateRecurso_SetReturn{
      ListaRecursos_Object_S1_Scope_updateRecurso_method_retorno_varReturn =
      ListaRecursos_Object_S1_Scope_updateRecurso_method_Stack.Pop()}
  -> Skip
 };

var<List> ListaRecursos_Object_S1_Scope_freeRecursos_method_Stack;
var ListaRecursos_Object_S1_Scope_freeRecursos_method_tamanho_var;
var ListaRecursos_Object_S1_Scope_freeRecursos_method_i_var;
synchronized_ListaRecursos_Object_S1_Scope_freeRecursos_method() =
 atomic {synchronized_ListaRecursos_Object_S1_Scope_getSize_method()
  ; ListaRecursos_Object_S1_Scope_freeRecursos_method_PushReturn{
   ListaRecursos_Object_S1_Scope_freeRecursos_method_Stack.Push(
       ListaRecursos_Object_S1_Scope_getSize_method_retorno_varReturn)
  } -> Skip
 } ; ListaRecursos_Object_S1_Scope_freeRecursos_method_UpdateLocalVar{
     ListaRecursos_Object_S1_Scope_freeRecursos_method_tamanho_var =
     ListaRecursos_Object_S1_Scope_freeRecursos_method_Stack.Pop()}
 -> ListaRecursos_Object_S1_Scope_freeRecursos_method_PushConstant{
     ListaRecursos_Object_S1_Scope_freeRecursos_method_Stack.Push(0)}
 -> ListaRecursos_Object_S1_Scope_freeRecursos_method_UpdateLocalVar{
     ListaRecursos_Object_S1_Scope_freeRecursos_method_i_var =
     ListaRecursos_Object_S1_Scope_freeRecursos_method_Stack.Pop()}
 -> ListaRecursos_Object_S1_Scope_freeRecursos_method_L1();

var<List> ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_Stack;
var ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_comparison1;
ListaRecursos_Object_S1_Scope_freeRecursos_method_L1() =
 ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_PushValue{
     ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_Stack.Push(
     ListaRecursos_Object_S1_Scope_freeRecursos_method_i_var)}
 -> ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_PushValue{
     ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_Stack.Push(
     ListaRecursos_Object_S1_Scope_freeRecursos_method_tamanho_var)}
 -> ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_CompLT{
   var v2 = ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_Stack.Pop();
   var v1 = ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_Stack.Pop();
   if ( v1 < v2 ) {
    ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_comparison1 = 1;
   } else {
    ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_comparison1 = 0;
   }
 } -> if (ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_comparison1 != 0) {
  ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_PushValue{
      ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_Stack.Push(
      ListaRecursos_Object_S1_Scope_freeRecursos_method_i_var)}
  -> ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_PushConstant{
      ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_Stack.Push(0)}
  -> ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_UpdateAttribute {
   var p2 = ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_Stack.Pop();
   var p1 = ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_Stack.Pop();
   ListaRecursos_Object_S1_Scope_locks_var_List.Set(p1,p2);
```

```
  } -> ListaRecursos_Object_S1_Scope_freeRecursos_method_L1_IncVar{
      ListaRecursos_Object_S1_Scope_freeRecursos_method_i_var =
      ListaRecursos_Object_S1_Scope_freeRecursos_method_i_var + 1}
  -> ListaRecursos_Object_S1_Scope_freeRecursos_method_L1()
 } else {
  Skip
 };

var<List> ListaRecursos_Object_S1_Scope_getSize_method_Stack;
var ListaRecursos_Object_S1_Scope_getSize_method_retorno_varReturn;
synchronized_ListaRecursos_Object_S1_Scope_getSize_method() =
 ListaRecursos_Object_S1_Scope_getSize_method_PushValue{var quant =
     ListaRecursos_Object_S1_Scope_locks_var_List.Count();
     ListaRecursos_Object_S1_Scope_getSize_method_Stack.Push(quant)}
 -> ListaRecursos_Object_S1_Scope_getSize_method_SetReturn{
     ListaRecursos_Object_S1_Scope_getSize_method_retorno_varReturn =
     ListaRecursos_Object_S1_Scope_getSize_method_Stack.Pop()}
 -> Skip;


var<List> ProcessoTipo1_Object_P1_Scope_run_method_Stack;
var ProcessoTipo1_Object_P1_Scope_run_method_i_var;
var ProcessoTipo1_Object_P1_Scope_run_method_tamanho_var;
var ProcessoTipo1_Object_P1_Scope_run_method_retorno_var;
ProcessoTipo1_Object_P1_Scope_run_method() = atomic {
    synchronized_ListaRecursos_Object_S1_Scope_getSize_method()
            ; ProcessoTipo1_Object_P1_Scope_run_method_PushReturn{
             ProcessoTipo1_Object_P1_Scope_run_method_Stack.Push(
                 ListaRecursos_Object_S1_Scope_getSize_method_retorno_varReturn)}
            -> Skip
           } ; ProcessoTipo1_Object_P1_Scope_run_method_UpdateLocalVar{
               ProcessoTipo1_Object_P1_Scope_run_method_tamanho_var =
               ProcessoTipo1_Object_P1_Scope_run_method_Stack.Pop()}
           -> ProcessoTipo1_Object_P1_Scope_run_method_PushConstant{
               ProcessoTipo1_Object_P1_Scope_run_method_Stack.Push(0)}
           -> ProcessoTipo1_Object_P1_Scope_run_method_UpdateLocalVar{
               ProcessoTipo1_Object_P1_Scope_run_method_i_var =
               ProcessoTipo1_Object_P1_Scope_run_method_Stack.Pop()}
           -> ProcessoTipo1_Object_P1_Scope_run_method_L1()
           ; atomic {synchronized_ListaRecursos_Object_S1_Scope_freeRecursos_method()
           };

var<List> ProcessoTipo1_Object_P1_Scope_run_method_L1_Stack;
var ProcessoTipo1_Object_P1_Scope_run_method_L1_comparison1;
ProcessoTipo1_Object_P1_Scope_run_method_L1() =
 ProcessoTipo1_Object_P1_Scope_run_method_PushValue{
     ProcessoTipo1_Object_P1_Scope_run_method_L1_Stack.Push(
     ProcessoTipo1_Object_P1_Scope_run_method_i_var)} //pula pro 194 devido ao goto no 110
 -> ProcessoTipo1_Object_P1_Scope_run_method_PushValue{
     ProcessoTipo1_Object_P1_Scope_run_method_L1_Stack.Push(
     ProcessoTipo1_Object_P1_Scope_run_method_tamanho_var)}
 -> ProcessoTipo1_Object_P1_Scope_run_method_L1_CompLT{
  var v2 = ProcessoTipo1_Object_P1_Scope_run_method_L1_Stack.Pop();
  var v1 = ProcessoTipo1_Object_P1_Scope_run_method_L1_Stack.Pop();
```

```
  if ( v1 < v2 ) {
   ProcessoTipo1_Object_P1_Scope_run_method_L1_comparison1 = 1;
  } else {
   ProcessoTipo1_Object_P1_Scope_run_method_L1_comparison1 = 0;
  }
 } -> ifa (ProcessoTipo1_Object_P1_Scope_run_method_L1_comparison1 != 0) {
  ProcessoTipo1_Object_P1_Scope_run_method_L2()
  ; ProcessoTipo1_Object_P1_Scope_run_method_L1_IncVar{
      ProcessoTipo1_Object_P1_Scope_run_method_i_var =
      ProcessoTipo1_Object_P1_Scope_run_method_i_var + 1}
  -> ProcessoTipo1_Object_P1_Scope_run_method_L1()
 } else {
  Skip
 };

var ProcessoTipo1_Object_P1_Scope_run_method_L2_p1_varArg;
var<List> ProcessoTipo1_Object_P1_Scope_run_method_L2_Stack;
var ProcessoTipo1_Object_P1_Scope_run_method_L2_comparison1;
ProcessoTipo1_Object_P1_Scope_run_method_L2() =
 ProcessoTipo1_Object_P1_Scope_run_method_L2_PushValue{
     ProcessoTipo1_Object_P1_Scope_run_method_L2_Stack.Push(
     ProcessoTipo1_Object_P1_Scope_run_method_i_var)}
 -> atomic{
   ProcessoTipo1_Object_P1_Scope_run_method_L2_arg{
       ProcessoTipo1_Object_P1_Scope_run_method_L2_p1_varArg =
       ProcessoTipo1_Object_P1_Scope_run_method_L2_Stack.Pop()}
   -> synchronized_ListaRecursos_Object_S1_Scope_updateRecurso_method(
       ProcessoTipo1_Object_P1_Scope_run_method_L2_p1_varArg)
   ; ProcessoTipo1_Object_P1_Scope_run_method_L2_PushReturn{
       ProcessoTipo1_Object_P1_Scope_run_method_L2_Stack.Push(
       ListaRecursos_Object_S1_Scope_updateRecurso_method_retorno_varReturn)}
   -> Skip
 } ; ProcessoTipo1_Object_P1_Scope_run_method_L2_UpdateLocalVar{
     ProcessoTipo1_Object_P1_Scope_run_method_retorno_var =
     ProcessoTipo1_Object_P1_Scope_run_method_L2_Stack.Pop()}
 -> ProcessoTipo1_Object_P1_Scope_run_method_L2_PushValue{
     ProcessoTipo1_Object_P1_Scope_run_method_L2_Stack.Push(
     ProcessoTipo1_Object_P1_Scope_run_method_retorno_var)}
 -> ProcessoTipo1_Object_P1_Scope_run_method_L2_PushConstant{
     ProcessoTipo1_Object_P1_Scope_run_method_L2_Stack.Push(1)}
 -> ProcessoTipo1_Object_P1_Scope_run_method_L2_CompNE2{
  var v2 = ProcessoTipo1_Object_P1_Scope_run_method_L2_Stack.Pop();
  var v1 = ProcessoTipo1_Object_P1_Scope_run_method_L2_Stack.Pop();
  if (v1 != v2) {
   ProcessoTipo1_Object_P1_Scope_run_method_L2_comparison1 = 1;
  } else {
   ProcessoTipo1_Object_P1_Scope_run_method_L2_comparison1 = 0;
  }
 } -> ifa (ProcessoTipo1_Object_P1_Scope_run_method_L2_comparison1 != 0) {
  // SimulaÃ§Ã£o de Deadlock do Processo Tipo 1
  atomic{
   updateP1_state{P1Blocked = true} -> if ( P2Blocked == true ) { Stop } else { Skip }
  } ;
  ProcessoTipo1_Object_P1_Scope_run_method_L2()
```

```
 } else {
  updateP1_state{P1Blocked = false} ->
  Skip
 };


var<List> ProcessoTipo2_Object_P2_Scope_run_method_Stack;
var ProcessoTipo2_Object_P2_Scope_run_method_i_var;
var ProcessoTipo2_Object_P2_Scope_run_method_tamanho_var;
var ProcessoTipo2_Object_P2_Scope_run_method_retorno_var;
ProcessoTipo2_Object_P2_Scope_run_method() = atomic {
            synchronized_ListaRecursos_Object_S1_Scope_getSize_method()
            ; ProcessoTipo2_Object_P2_Scope_run_method_PushReturn{
             ProcessoTipo2_Object_P2_Scope_run_method_Stack.Push(
                 ListaRecursos_Object_S1_Scope_getSize_method_retorno_varReturn)}
            -> Skip
           } ; ProcessoTipo2_Object_P2_Scope_run_method_UpdateLocalVar{
               ProcessoTipo2_Object_P2_Scope_run_method_tamanho_var =
               ProcessoTipo2_Object_P2_Scope_run_method_Stack.Pop()}
           -> ProcessoTipo2_Object_P2_Scope_run_method_PushValue{
               ProcessoTipo2_Object_P2_Scope_run_method_Stack.Push(
               ProcessoTipo2_Object_P2_Scope_run_method_tamanho_var)}
           -> ProcessoTipo2_Object_P2_Scope_run_method_PushConstant{
               ProcessoTipo2_Object_P2_Scope_run_method_Stack.Push(1)}
           -> ProcessoTipo2_Object_P2_Scope_run_method_SubValues{
            var v2 = ProcessoTipo2_Object_P2_Scope_run_method_Stack.Pop();
            var v1 = ProcessoTipo2_Object_P2_Scope_run_method_Stack.Pop();
            var v3 = v1 - v2;
            ProcessoTipo2_Object_P2_Scope_run_method_Stack.Push(v3);
           } -> ProcessoTipo2_Object_P2_Scope_run_method_UpdateLocalVar{
               ProcessoTipo2_Object_P2_Scope_run_method_i_var =
               ProcessoTipo2_Object_P2_Scope_run_method_Stack.Pop()}
           -> ProcessoTipo2_Object_P2_Scope_run_method_L1()
           ; atomic {
            synchronized_ListaRecursos_Object_S1_Scope_freeRecursos_method()
           };

var<List> ProcessoTipo2_Object_P2_Scope_run_method_L1_Stack;
var ProcessoTipo2_Object_P2_Scope_run_method_L1_comparison;
ProcessoTipo2_Object_P2_Scope_run_method_L1() =
 ProcessoTipo2_Object_P2_Scope_run_method_PushValue{
     ProcessoTipo2_Object_P2_Scope_run_method_L1_Stack.Push(
     ProcessoTipo2_Object_P2_Scope_run_method_i_var)}
 -> ProcessoTipo2_Object_P2_Scope_run_method_L1_CompGE{
  var v1 = ProcessoTipo2_Object_P2_Scope_run_method_L1_Stack.Pop();
  if ( v1 >= 0 ) {
   ProcessoTipo2_Object_P2_Scope_run_method_L1_comparison = 1;
  } else {
   ProcessoTipo2_Object_P2_Scope_run_method_L1_comparison = 0;
  }
 } -> ifa (ProcessoTipo2_Object_P2_Scope_run_method_L1_comparison != 0) {
  ProcessoTipo2_Object_P2_Scope_run_method_L2()
  ; ProcessoTipo2_Object_P2_Scope_run_method_L1_IncVar{
      ProcessoTipo2_Object_P2_Scope_run_method_i_var =
```

```
      ProcessoTipo2_Object_P2_Scope_run_method_i_var - 1}
   -> ProcessoTipo2_Object_P2_Scope_run_method_L1()
  } else {
   Skip
  };

var ProcessoTipo2_Object_P2_Scope_run_method_L2_p1_varArg;
var<List> ProcessoTipo2_Object_P2_Scope_run_method_L2_Stack;
var ProcessoTipo2_Object_P2_Scope_run_method_L2_comparison1;
ProcessoTipo2_Object_P2_Scope_run_method_L2() =
 ProcessoTipo2_Object_P2_Scope_run_method_L2_PushValue{
      ProcessoTipo2_Object_P2_Scope_run_method_L2_Stack.Push(
      ProcessoTipo2_Object_P2_Scope_run_method_i_var)}
 -> atomic{
    ProcessoTipo2_Object_P2_Scope_run_method_L2_arg{
         ProcessoTipo2_Object_P2_Scope_run_method_L2_p1_varArg =
         ProcessoTipo2_Object_P2_Scope_run_method_L2_Stack.Pop()}
    -> synchronized_ListaRecursos_Object_S1_Scope_updateRecurso_method(
         ProcessoTipo2_Object_P2_Scope_run_method_L2_p1_varArg)
    ; ProcessoTipo2_Object_P2_Scope_run_method_L2_PushReturn{
         ProcessoTipo2_Object_P2_Scope_run_method_L2_Stack.Push(
         ListaRecursos_Object_S1_Scope_updateRecurso_method_retorno_varReturn)}
    -> Skip
  } ; ProcessoTipo2_Object_P2_Scope_run_method_L2_UpdateLocalVar{
      ProcessoTipo2_Object_P2_Scope_run_method_retorno_var =
      ProcessoTipo2_Object_P2_Scope_run_method_L2_Stack.Pop()}
 -> ProcessoTipo2_Object_P2_Scope_run_method_L2_PushValue{
      ProcessoTipo2_Object_P2_Scope_run_method_L2_Stack.Push(
      ProcessoTipo2_Object_P2_Scope_run_method_retorno_var)}
 -> ProcessoTipo2_Object_P2_Scope_run_method_L2_PushConstant{
      ProcessoTipo2_Object_P2_Scope_run_method_L2_Stack.Push(1)}
 -> ProcessoTipo2_Object_P2_Scope_run_method_L2_CompNE2{
   var v2 = ProcessoTipo2_Object_P2_Scope_run_method_L2_Stack.Pop();
   var v1 = ProcessoTipo2_Object_P2_Scope_run_method_L2_Stack.Pop();
   if (v1 != v2) {
    ProcessoTipo2_Object_P2_Scope_run_method_L2_comparison1 = 1;
   } else {
    ProcessoTipo2_Object_P2_Scope_run_method_L2_comparison1 = 0;
   }
  } -> ifa (ProcessoTipo2_Object_P2_Scope_run_method_L2_comparison1 != 0) {
   // SimulaÃ§Ã£o de Deadlock do Processo Tipo 2
   atomic{
    updateP2_state{P2Blocked = true} -> if ( P1Blocked == true ) { Stop } else { Skip }
   } ;
   ProcessoTipo2_Object_P2_Scope_run_method_L2()
  } else {
   updateP2_state{P2Blocked = false} ->
   Skip
  };


var<List> Example2_Object_E2_Scope_main_method_Stack;
var Example2_Object_E2_Scope_main_method_p1_varArg;
Example2_Object_E2_Scope_main_method() = ListaRecursos_Object_S1_Scope_ListaRecursos_method()
```

```
        ;  Example2_Object_E1_Scope_main_method_PushConstant{
             Example2_Object_E2_Scope_main_method_Stack.Push(2)}
        -> atomic{Example2_Object_E2_Scope_main_method_arg{
             Example2_Object_E2_Scope_main_method_p1_varArg =
             Example2_Object_E2_Scope_main_method_Stack.Pop()}
        -> ListaRecursos_Object_S1_Scope_addRecurso_method(
             Example2_Object_E2_Scope_main_method_p1_varArg)}
        ; Example2_Object_E1_Scope_main_method_PushConstant{
             Example2_Object_E2_Scope_main_method_Stack.Push(3)}
        -> atomic{Example2_Object_E2_Scope_main_method_arg{
             Example2_Object_E2_Scope_main_method_p1_varArg =
             Example2_Object_E2_Scope_main_method_Stack.Pop()}
        -> ListaRecursos_Object_S1_Scope_addRecurso_method(
             Example2_Object_E2_Scope_main_method_p1_varArg)}
        ; (ProcessoTipo1_Object_P1_Scope_run_method() |||
             ProcessoTipo2_Object_P2_Scope_run_method()) ;


#assert Example2_Object_E2_Scope_main_method() deadlockfree;
#assert Example2_Object_E2_Scope_main_method() nonterminating;
#assert Example2_Object_E2_Scope_main_method() deterministic;
#assert Example2_Object_E2_Scope_main_method() divergencefree;
```