Pontifícia Universidade Católica
do Rio de Janeiro

Anna Leticia Alegria Pinheiro de Oliveira

EventManager: Uma ferramenta de análise de programas concorrentes

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática da PUC-Rio.

Orientador: Prof. Roberto Ierusalimschy

Rio de Janeiro
Agosto de 2022

Pontifícia Universidade Católica
do Rio de Janeiro

Anna Leticia Alegria Pinheiro de Oliveira

EventManager: Uma ferramenta de análise de programas concorrentes

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática da PUC-Rio . Aprovada pela Comissão Examinadora abaixo:

Prof. Roberto Ierusalimschy
Orientador
Departamento de Informática – PUC-Rio

Profª. Noemi de La Rocque Rodriguez
Pesquisadora autônoma

Prof. Luiz Fernando Bessa Seibel
PUC-Rio

Profª. Silvana Rossetto
UFRJ

Rio de Janeiro, 26 de Agosto de 2022

Anna Leticia Alegria Pinheiro de Oliveira

Graduada em engenharia da computação pela Pontifícia Universidade Católica do Rio de Janeiro.

Ficha Catalográfica

Alegria Pinheiro de Oliveira, Anna Leticia

EventManager: Uma ferramenta de análise de programas concorrentes / Anna Leticia Alegria Pinheiro de Oliveira; orientador: Roberto Ierusalimschy. – 2022.

79 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2022.

Inclui bibliografia

1. Informática – Teses. 2. programação concorrente. 3. testes de programas. 4. linguagens de domínio específico. 5. aprendizagem de concorrência. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

À minha orientadora Professora Noemi Rodriguez por todo o apoio, dedicação e parceria para a realização deste trabalho.

Ao meu orientador Professor Roberto Ierusalimschy, pelos conselhos e contribuições para este trabalho.

À PUC-Rio, pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

À minha família, em especial à minha mãe, pelo carinho, apoio e dedicação pela minha formação.

Ao meu namorado Rafael, pelos conselhos, incentivos e carinho.

Aos meus amigos, que contribuíram para que esta jornada fosse mais leve.

Aos professores que participaram da Comissão examinadora.

A todos os professores e funcionários do Departamento pelos ensinamentos e pela ajuda.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Resumo

Alegria Pinheiro de Oliveira, Anna Leticia; Ierusalimschy, Roberto. **EventManager: Uma ferramenta de análise de programas concorrentes**. Rio de Janeiro, 2022. 79p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Alunos aprendendo programação concorrente muitas vezes têm dificuldades de testar seus programas por conta do não-determinismo presente no escalonamento de *threads*. Em geral, é difícil testar cenários específicos e mais difícil ainda repetir um determinado cenário para testar mudanças do código. Nesta tese, apresentamos a *EventManager*: uma ferramenta que criamos para permitir que um usuário instrumente seu programa, marcando eventos no código e especificando sequências de eventos através de uma linguagem de domínio específico (DSL). Esta linguagem restringe o escalonamento das *threads* para que obedeça as sequências permitidas para estes eventos. Descrevemos a implementação da *EventManager* para aplicações baseadas em *threads POSIX*. Investigamos a aplicação da ferramenta em soluções de problemas clássicos de concorrência para averiguar a expressividade da linguagem que criamos.

Palavras-chave

programação concorrente; testes de programas; linguagens de domínio específico; aprendizagem de concorrência.

Abstract

Alegria Pinheiro de Oliveira, Anna Leticia; Ierusalimschy, Roberto (Advisor). **EventManager: A tool for analysing concurrent programs**. Rio de Janeiro, 2022. 79p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Students learning concurrent programming often struggle with tests due to the non-deterministic nature of thread scheduling. It is in general hard to test specific scenarios and harder yet to repeat a given scenario for further tests after changes to the code. In this thesis, we present *EventManager*: a tool we developed that allows the user to instrument their program, marking events in the code and specifying valid event sequences using a domain-specific language. This language restricts thread scheduling to obey allowed sequences for these events. We describe the implementation of EventManager for applications based on POSIX threads. We investigate our tool applied on solutions of classical concurrency problems to verify the expressiveness of the created language.

Keywords

concurrent programming; program tests; DSLs; concurrency learning.

Sumário

1 Introdução **2**
1.1 Motivação 2
1.2 Objetivos 3
1.3 Metodologia 4
1.4 Estrutura da dissertação 4

2 Trabalhos relacionados **5**
2.1 Discussão 7

3 Ambiente desenvolvido **8**
3.1 Visão geral 8
3.2 Linguagem de *script* 10
3.3 Formato do arquivo de configuração 13

4 Implementação **14**
4.1 Funcionamento geral 14
4.2 Processamento do arquivo de configuração 15
4.3 Processamento do *script* 15
4.4 Análise de eventos 22
4.5 Ocorrência de *deadlock* causado pela ferramenta 23
4.6 Testes de funcionamento 24

5 Problemas de concorrência **26**
5.1 Problema dos leitores escritores 26
5.2 Problema dos produtores e consumidores 35
5.3 Problema da fila de itens 44
5.4 Problema da barreira 51
5.5 Problema do jantar dos filósofos 56
5.6 Discussão Geral 66

6 Conclusão e trabalhos futuros **67**

7 Referências bibliográficas **69**

Lista de figuras

Figura 3.1 Exemplo de corpo de uma *thread* de um programa concorrente 9
Figura 3.2 Exemplo de um corpo de *thread* utilizando a ferramenta desenvolvida. Ao fazer isso, foi possível encontrar erros de concorrência no código. 9
Figura 3.3 *Script* com a ordem dos eventos a ser testada no exemplo da Figura 3.2 10
Figura 3.4 Gramática da linguagem de eventos. 11
Figura 3.5 Exemplo de *script* com sequências, alternativa e repetição. 11
Figura 3.6 Gramática da linguagem de especificação de *threads*. 12
Figura 3.7 Exemplos de especificações de *thread* que um evento pode conter. 12

Figura 4.1 Pseudo código com o funcionamento geral da *EventManager*. 15
Figura 4.2 Exemplo de *script* com sequências, alternativa e repetição, apresentado anteriormente. 16
Figura 4.3 Árvore de sintaxe criada a partir da interpretação do *script* da Figura 3.5 17
Figura 4.4 Grafo criado a partir da árvore de sintaxe da Figura 4.3 17
Figura 4.5 Estrutura de dados que representa o grafo gerado a partir do *script* da Figura 3.5 18
Figura 4.6 Árvore de sintaxe da Figura 4.3 com ligações que exemplificam o nó de destino de cada evento. 19
Figura 4.7 Exemplo de *script* com repetição. 19
Figura 4.8 Grafos criado a partir do *script* da Figura 4.7. (a) Mostra o grafo sem os nós auxiliares, podendo ocorrer ordens não esperadas pelo *script* do usuário. (b) Mostra a correção que fizemos no grafo para corrigir estes erros. 20
Figura 4.9 *Script* com grafo que mostra um exemplo de que podem existir vários possíveis nós correntes. 21
Figura 4.10 Árvores de cada exemplo de especificação de *thread* apresentado na Figura 3.7. 22
Figura 4.11 Estrutura de dados que mostra as *threads* contidas em cada conjunto. 22
Figura 4.12 *Script* utilizado em uma das simulações de programas. 25
Figura 4.13 Simulação de um programa concorrente através de chamadas consecutivas de eventos. 25

Figura 5.1 Corpo das *threads* leitoras. 27
Figura 5.2 Corpo das *threads* escritoras. 28
Figura 5.3 *Script* descrevendo o caso de dois escritores entrarem na região crítica juntos 29
Figura 5.4 Resultado da execução do programa dos leitores e escritores com o *script* da Figura 5.3 29

Figura 5.5 *Script* descrevendo o caso de um escritor e um leitor entrarem na região crítica juntos 30
Figura 5.6 Resultado da execução do programa dos leitores e escritores com o *script* da Figura 5.5 30
Figura 5.7 *Script* descrevendo o caso de três leitores entrarem na região crítica juntos 31
Figura 5.8 Resultado da execução do programa dos leitores e escritores com o *script* da Figura 5.7 31
Figura 5.9 *Script* descrevendo uma sequência que causa *starvation* nos escritores. 32
Figura 5.10 Resultado da execução do programa dos leitores e escritores com o *script* da Figura 5.9 33
Figura 5.11 *Script* descrevendo uma sequência infinita do problema dos leitores e escritores. 34
Figura 5.12 Resultado da execução do programa dos leitores e escritores com o *script* da Figura 5.11 34
Figura 5.13 Corpo das *threads* produtoras. 36
Figura 5.14 Corpo das *threads* consumidoras. 37
Figura 5.15 *Script* com a sequência dos eventos do problema do produtor e consumidor que descreve um programa iniciando a sua execução com um consumidor. 38
Figura 5.16 Resultado da execução do programa de produtores e consumidores com o *script* da Figura 5.15 38
Figura 5.17 *Script* que descreve um produtor e um consumidor tentando acessar a região crítica ao mesmo tempo. 38
Figura 5.18 Resultado da execução do programa de produtores e consumidores com o *script* da Figura 5.17 39
Figura 5.19 *Script* que descreve um produtor executando duas vezes seguidas. 39
Figura 5.20 Resultado da execução do programa de produtores e consumidores com o *script* da Figura 5.19 40
Figura 5.21 *Script* que descreve uma execução completa de um produtor seguida por duas execuções do consumidor. 40
Figura 5.22 Resultado da execução do programa de produtores e consumidores com o *script* da Figura 5.21 41
Figura 5.23 *Script* de um teste com sequências infinitas de eventos do problema do produtor e consumidor, que força a execução alternada entre um produtor e um consumidor. 41
Figura 5.24 Resultado da execução do programa de produtores e consumidores com o *script* da Figura 5.23 42
Figura 5.25 *Script* de um teste com sequências infinitas de eventos do problema do produtor e consumidor. 43
Figura 5.26 Resultado da execução do programa de produtores e consumidores com o *script* da Figura 5.25 44
Figura 5.27 Corpo das *threads* que inserem itens na fila. 45
Figura 5.28 Corpo das *threads* que retiram itens da fila. 46
Figura 5.29 *Script* que descreve uma *thread dequeuer* sendo executada antes de alguma *thread enqueuer*. 47

Figura 5.30 Resultado da execução do programa da fila de itens com o *script* da Figura 5.29 47
Figura 5.31 *Script* com a sequência dos eventos que descreve um teste realizado pelo artigo de Mayer e Madhavan (MAYER; MADHAVAN, 2016). 48
Figura 5.32 Resultado da execução do programa da fila de itens com o *script* da Figura 5.31 49
Figura 5.33 *Script* com a sequência dos eventos do teste do código das Figuras 5.27 e 5.28 para ilustrar problemas que poderiam ocorrer pelo uso da função *pthread_cond_wait* 50
Figura 5.34 Resultado da execução do programa da fila de itens com o *script* da Figura 5.33 50
Figura 5.35 Implementação ingênua do problema da barreira com erros. 52
Figura 5.36 *Script* com a ordem dos eventos do código da Figura 5.35 52
Figura 5.37 Resultado da execução do teste do código da Figura 5.35 com o *script* da Figura 5.36 53
Figura 5.38 Implementação correta da barreira disponibilizada pela professora Silvana Rossetto. 54
Figura 5.39 Resultado da execução do teste do código da Figura 5.38 com o *script* da Figura 5.36 54
Figura 5.40 *Script* descrevendo sequências infinitas do problema da barreira. 55
Figura 5.41 Resultado da execução do teste do código da Figura 5.38 com o *script* da Figura 5.40 56
Figura 5.42 Pseudo código das ações executadas por um filósofo (DOWNEY, 2009). 57
Figura 5.43 Corpo da *thread* correspondente a cada filósofo. 58
Figura 5.44 Funções com as implementações de cada ação executada pelos filósofos. 59
Figura 5.45 *Script* com a sequência dos eventos do problema do jantar dos filósofos que descreve o filósofo canhoto adquirindo um garfo e os outros quatro filósofos adquirindo um garfo 60
Figura 5.46 Resultado da execução do programa do jantar dos filósofos com o *script* da Figura 5.45 61
Figura 5.47 *Script* com a sequência dos eventos do problema jantar dos filósofos que descreve dois filósofos comendo juntos. 62
Figura 5.48 Resultado da execução do programa do jantar dos filósofos com o *script* da Figura 5.47 63
Figura 5.49 *Script* com a sequência dos eventos do problema do jantar dos filósofos que descreve dois filósofos segurando o mesmo garfo ao mesmo tempo. 64
Figura 5.50 Resultado da execução do programa do jantar dos filósofos com o *script* da Figura 5.49 65

1
Introdução

Programas concorrentes são amplamente utilizados em diversas áreas da computação. Entretanto, afirmar sua corretude ou encontrar seus erros é um desafio, em boa parte por conta da dificuldade de reproduzir esses erros ou de encontrar todas as possíveis situações que podem ocorrer no programa. Isso se deve ao fato do escalonamento das *threads* ou dos processos pelo sistema operacional não ser determinístico.

Existem diversas ferramentas que auxiliam o teste de programas concorrentes (BIANCHI; MARGARA; PEZZè, 2018). A maior parte delas captura erros de propriedades específicas do código como corrida de dados e violações de atomicidade. Entretanto, há pouco apoio para quem está desenvolvendo um programa testar a lógica do mesmo, em especial para alunos tendo seus primeiros contatos com concorrência.

Neste projeto, desenvolvemos uma ferramenta que permite o teste de programas concorrentes voltada para os alunos dos cursos de programação. Ela permite que o aluno desenvolva *scripts* de testes ou que o professor utilize um *script* para facilitar a correção dos trabalhos.

Para utilizar a ferramenta, o usuário define eventos ao longo do seu código. No *script*, ele determina a ordem que estes eventos podem ser realizados e analisa o resultado de seu programa observando a execução do mesmo seguindo tal ordem.

A ferramenta foi aplicada em problemas conhecidos da computação concorrente. Tais problemas permitiram uma investigação acerca da expressividade da linguagem de *scripts* desenvolvida, discussão presente neste trabalho.

1.1
Motivação

A concorrência em um programa pode ser implementada de diversas formas, mas neste trabalho vamos focar em implementações com *threads* de sistema operacional (*POSIX threads*).

Um sistema operacional preemptivo intercala as operações realizadas por cada *thread*. Em cada execução, a ordem de execução das *threads* pode ser diferente, dependendo do momento em que determinada *thread* for interrompida para que outra possa ser executada. Desta forma, erros de concorrência podem ocorrer em algumas execuções e não em outras, sendo difíceis de reproduzir e, consequentemente, corrigir. Existem ferramentas que se propõe a detectar erros mais genéricos de concorrência, como *data races* (BIANCHI; MARGARA; PEZZè, 2018). Entretanto, há poucas opções de ferramentas que testam a semântica do programa.

Uma forma de detectar erros de concorrência é controlar o andamento das *threads* para tentar forçar um escalonamento. Dentro dessa abordagem, muitos programadores utilizam tempos de espera, comumente inseridos através de *sleeps*, para esperar que determinado evento ocorra. Porém, utilizar *sleeps* acaba deixando os testes desnecessariamente demorados e pode levar a falsos

positivos e negativos. Desta forma, outra abordagem foi criada. Ela consiste em definir eventos ao longo do programa e forçar o programa a seguir determinada ordem de eventos (JAGANNATH et al., 2011). Com isso, é possível reproduzir os testes para investigar o comportamento do programa.

1.2 Objetivos

A ferramenta deste projeto, projetada para fins didáticos, tem o objetivo de facilitar o teste de programas concorrentes. Ela controla o andamento das *threads* do programa concorrente do usuário tentando forçar determinada ordem de eventos. Os eventos são inseridos pelo usuário em pontos de seu programa. O usuário especifica ordens de eventos permitidas dentro das ordens possíveis de seu programa através de uma linguagem de *script* que desenvolvemos.

Um *script* define a estrutura das sequências de eventos que serão aceitas. Dentre as sequências que o usuário especificar no *script*, permitiremos que uma seja executada, caso possível. O usuário define estas sequências para testar a semântica de seu programa. Uma sequência de eventos é considerada inválida quando ela viola os requisitos semânticos desse programa. Ou seja, é uma sequência que não deveria ser permitida por ele. Caso este programa consiga executar uma sequência inválida (dizemos que o programa *aceitou* uma sequência inválida), podemos garantir que ele está errado. Analogamente, uma sequência válida se encaixa nos requisitos semânticos do programa do usuário. Se ele não aceitar uma sequência válida, podemos garantir que ele está errado. Porém, caso este programa aceite uma sequência válida, não é possível afirmar que o mesmo está correto. Neste caso, o programador pode ainda comparar o resultado obtido com o resultado esperado, utilizando o *script* para reproduzir a mesma sequência de eventos caso seja necessário corrigir o programa.

A linguagem de *script* que estamos explorando é baseada em expressões regulares e permite que o usuário expresse uma ordem de eventos, podendo utilizar repetições de blocos de eventos e alternativas entre esses blocos. Além disso, é possível especificar quais grupos de *threads* podem executar determinado evento. Neste projeto, investigamos até que ponto estas linguagens de eventos e *threads* permitem expressar os testes que o usuário deseja realizar.

A ferramenta permite que o aluno teste seu programa e o ajuda a entender o problema dado, já que elaborar uma sequência de eventos a ser testada exige uma compreensão dos possíveis caminhos que cada *thread* pode ou não seguir.

Nosso objetivo com este projeto é criar uma ferramenta que auxilie no teste de programas concorrentes, na qual o usuário consiga expressar uma ordem de eventos a ser seguida pelo seu programa. Ao verificar se a ordem foi executada por completo ou a partir de qual evento ela não foi aceita, o usuário consegue entender melhor quais erros seu código possui e é capaz de reproduzi-los com maior facilidade. Além disso, queremos investigar a expressividade da abordagem que utilizamos e das linguagens de eventos e de especificação de *threads* que desenvolvemos.

1.3 Metodologia

A metodologia deste trabalho consistiu em criar uma linguagem de domínio específico para expressar os testes e, em cima dela, construir a ferramenta, estudando os exemplos dos problemas clássicos de concorrência.

A abordagem de testes escolhida para a ferramenta foi a de controlar o andamento das *threads* do programa concorrente do usuário.

Para investigar a expressividade da linguagem de *script* que desenvolvemos, visitamos problemas clássicos de concorrência que nos deram retornos para aprimorar nossa ferramenta.

Diferentemente de outras ferramentas existentes que forçam determinada ordem de eventos controlando o escalonamento das *threads* através de algum *framework*, nossa ferramenta utiliza *waits* para controlar o andamento das *threads*. Sendo assim, ela não apresenta uma solução exclusiva para a linguagem de programação C, podendo ser transcrita para outras linguagens. Por outro lado, por não interferirmos diretamente na execução do programa do usuário, encontramos algumas limitações na nossa ferramenta (como apresentamos na discussão da Seção 5.6).

1.4 Estrutura da dissertação

Este trabalho está dividido em seis capítulos. No Capítulo 2, mostramos alguns trabalhos existentes acerca do tema de testes concorrentes relacionados ao nosso. No Capítulo 3, apresentamos a *EventManager* e a linguagem de *script* que desenvolvemos para a mesma. No capítulo seguinte, explicamos a implementação de cada etapa, incluindo os algoritmos e as estruturas de dados que utilizamos. No Capítulo 5, exploramos o uso da *EventManager* para testes de alguns problemas de concorrência bem conhecidos. Além disso, para cada problema de concorrência, discutimos sobre alguns pontos de melhoria da ferramenta que só foram possíveis graças aos testes práticos. Por fim, no Capítulo 6, concluímos nosso trabalho e analisamos a expressividade das linguagens que desenvolvemos, além de mostrarmos alguns pontos de melhorias para a ferramenta.

2
Trabalhos relacionados

Grande parte dos principais trabalhos existentes do tema de testes em sistemas concorrentes estão resumidamente apresentados em um *survey* publicado em 2017 (BIANCHI; MARGARA; PEZZè, 2018). Os autores também classificam estes sistemas quanto aos seus aspectos. A classificação avalia os seguintes sete critérios:

- *Input*: Algumas ferramentas recebem como entrada um conjunto de testes e outras geram testes automaticamente. Além disso, algumas técnicas exigem um modelo de sistema que especifique algumas propriedades ou comportamentos esperados do mesmo.
- Seleção de escalonamentos do fluxo de execução: Os escalonamentos das instruções de diferentes fluxos de execução devem ser levados em conta em testes de programas concorrentes. Os sistemas diferem em como selecioná-los. Podem ter uma abordagem baseada em alguma propriedade ou realizar uma exploração exaustiva de todo o espaço de possíveis escalonamentos.
- Propriedade dos escalonamentos: As ferramentas que selecionam escalonamentos específicos diferem em qual propriedade será testada neles. Estas propriedades podem ser corrida de dados, violações de atomicidade, *deadlocks*, violações de ordem ou várias propriedades combinadas.
- *Output*/Oráculo: Algumas ferramentas geram um *output* que informa se determinado escalonamento satisfaz a propriedade em questão e outras identificam falhas de execução comparando o resultado com um oráculo ou verificando se houve *deadlocks*, falha no sistema ou assertivas violadas.
- Garantias: Cada técnica oferece diferentes garantias de sua precisão e corretude acerca de seus resultados.
- Sistema escolhido: A maioria das técnicas dependem do modelo de comunicação escolhido pelo sistema a ser testado, paradigma de programação utilizado e modelo de consistência escolhido.
- Técnica de testes: As ferramentas diferem no tipo de testes utilizados, granularidade dos testes, escopo e arquitetura dos mesmos.

A maioria das ferramentas analisadas pelo *survey* faz uma busca exaustiva por alguma propriedade específica como corrida de dados e violações de atomicidade, não se preocupando com o aspecto semântico do problema.

Sabendo dos diversos tipos de ferramentas existentes, foi possível buscar trabalhos que mais se relacionassem com os nossos objetivos.

Uma ferramenta interessante encontrada é a *IMUnit*, apresentada no artigo *Improved Multithreaded Unit Testing* (JAGANNATH et al., 2011). É uma ferramenta que utiliza uma linguagem que permite especificar escalonamentos das ordens de eventos que ocorrem na execução do programa. Ela utiliza o *framework* de testes *JUnit* da linguagem de programação *Java* para permitir a execução do escalonamento desejado. Neste trabalho, os autores também

discutem a ineficiência de se utilizar tempos de espera, comumente inseridos através de *sleeps*, no programa para esperar que determinado evento ocorra. Quando se utiliza uma abordagem baseada em *sleeps*, muitas vezes coloca-se o tempo a ser esperado como muito maior do que o tempo real de execução do evento, gerando testes desnecessariamente demorados. Além disso, os autores demonstram como essa abordagem acaba gerando falsos positivos e negativos. Para solucionar este problema, a *IMUnit* não utiliza *sleeps*, ela espera que um evento ocorra para iniciar o seguinte, controlando o escalonamento.

Outra ferramenta com uma abordagem interessante é a *Concurrit*, desenvolvida por pesquisadores da Universidade de Berkeley (ELMAS et al., 2013). O foco é a reprodução de *bugs* em sistemas concorrentes através da escrita de um *script* que especifique os escalonamentos das *threads*. A linguagem utilizada no *script* permite uma grande variedade de testes, tendo como exemplo a possibilidade de especificar uma ordem das funções a serem executadas e quais *threads* podem se intercalar ao executar determinada função, permitindo um controle do escalonamento. Além disso, a linguagem permite a escrita de testes totalmente determinísticos ou com trechos não determinísticos, já que não é preciso controlar todo o escalonamento para exibir o *bug*, basta controlar uma parte crítica. A elaboração de uma linguagem como a desenvolvida na *Concurrit* é essencial para realizar testes mais complexos e mais expressivos.

Em um âmbito didático, a ferramenta do artigo de Mayer e Madhavan apresenta algumas ideias que poderiam ser utilizadas neste projeto, caso não fossem específicos para a linguagem de programação *Scala* (MAYER; MADHAVAN, 2016). Eles desenvolveram uma biblioteca para testar programas em *Scala* que força uma ordem em operações marcadas explicitamente no programa, através de primitivas de sincronização como *synchronized*, *wait*, *notify* e *notifyall*. Usando as facilidades de *Scala*, a biblioteca permite associar essas e outras funções a blocos de código no início dos quais ela realiza o escalonamento entre as *threads*. O objetivo é testar programas escritos por alunos da graduação, detectando *bugs* nos mesmos e gerando *feedbacks* úteis para que eles localizem e consertem os *bugs*. A ferramenta foi testada por 150 alunos, tendo sido passados testes a serem realizados em seus programas. O interessante é que alguns alunos criaram seus próprios testes com base nos que foram passados.

Um trabalho que explorou muito bem os tipos de erros de concorrência mais comumente encontrados em aplicações do mundo real é o artigo de Shan Lu e sua equipe (LU et al., 2008). Eles analisaram 105 *bugs* reportados por usuários de programas como *MySQL*, *Apache*, *Mozilla* e *OpenOffice*. No artigo, a equipe diz que o maior desafio de testes concorrentes é o espaço exponencial dos possíveis escalonamentos de um programa. Para encontrar um *bug*, é preciso encontrar o escalonamento cuja execução leve a esse erro. Assim, uma cobertura de testes completa precisaria cobrir todos os possíveis escalonamentos do programa a ser testado, o que não é possível na prática. Porém, segundo o levantamento realizado por eles, quase todos os *bugs* examinados se manifestam garantidamente se uma certa ordem parcial entre duas *threads* é forçada. Além disso, quase todos os erros que envolviam o surgimento de um *deadlock* envolvem duas *threads* esperando pelos mesmos dois recursos. Eles concluíram que testes em pares de *threads* podem expor a maior parte dos erros e reduzir

a complexidade do teste. Este trabalho nos motivou a não nos preocuparmos em gerar todos os possíveis escalonamentos do programa para ter uma boa ferramenta de análise portanto que ela permita que o programa especifique duas ou mais *threads* a se entrelaçarem para expor os erros de concorrência.

Em 1978, Brinch Hansen criou um método para testar monitores de um programa concorrente (HANSEN, 1978). A ideia proposta consistia em instrumentar o código utilizando funções *await(time)* e utilizando um *clock* global. Assim, cada ponto que contém um *await* é liberado quando o valor do *clock* é igual ao tempo a ser esperado, criando uma ordem entre esses pontos. Pelo que pesquisamos, Brinch Hansen foi o primeiro a propor esta abordagem, mas não a implementou.

2.1 Discussão

Tendo em mente os diversos tipos de sistemas de testes e os trabalhos existentes, começamos a desenvolver a ferramenta deste trabalho com alguns critérios específicos. Quanto ao *input*, ela recebe um *script* com as ordens de execução permitidas. Ela não gera todos os possíveis escalonamentos, mas sim, monitora a ordem de eventos específicos, obrigando o programa a seguir alguma das ordens permitidas. A forma de *output* escolhida foi a impressão na tela da sequência que foi possível de ser executada, além da ferramenta informar caso o programa não chegue a um próximo evento permitido. Analisando a sequência de eventos impressa, o programador pode analisar se o programa executou alguma sequência inválida e, no caso de sequências válidas, pode comparar o resultados com o esperado para essa sequência.

Em comparação com os trabalhos existentes, focamos em desenvolver uma ferramenta simples de ser usada e que permita a realização de testes no âmbito semântico, para que o usuário possa verificar se a lógica de sua aplicação foi corretamente implementada.

3
Ambiente desenvolvido

Neste capítulo descrevemos o ambiente que desenvolvemos para permitir que o usuário defina o escalonamento de seu programa. Vamos descrever a biblioteca que criamos e a linguagem que desenvolvemos para utilizá-la. Questões acerca da implementação de cada parte da ferramenta e detalhes mais técnicos serão mostrados no Capítulo 4.

3.1
Visão geral

A ferramenta *EventManager* funciona como uma biblioteca usada para instrumentar o programa que o usuário deseja testar. Existem três funções que devem ser chamadas ao longo do programa. A primeira, chamada de *initializeManager*, deve ser chamada no início do programa para inicializar as estruturas e realizar a leitura do arquivo que contém o *script* de testes. A segunda é a função *checkCurrentEvent*, que deve ser chamada em cada ponto que o usuário deseja marcar como um evento. E por fim, a *finalizeManager* deve ser chamada no final do programa do usuário para liberar as estruturas criadas pela ferramenta.

Para introduzir o funcionamento da *EventManager*, vamos utilizar um exemplo. Na Figura 3.1, temos o corpo das *threads* de um programa concorrente. As *threads* têm acesso à variável global *count* e salvam seu valor em uma variável local chamada de *aux*. Em seguida, atualizam o valor da variável global *count* como sendo o valor lido anteriormente incrementado de 1. Por último, as threads imprimem o valor da variável *count*. Alunos aprendendo concorrência poderiam esperar que o último resultado impresso seja igual ao número de *threads* criadas pelo programa. Porém, como o sistema operacional poderia interromper uma *thread* logo após a leitura do valor de *count* e liberar alguma outra *thread* que alterasse seu valor, a primeira *thread* poderia sobrescrever alguma operação realizada em *count* e o valor final da variável não ser igual ao número de *threads* do programa. O código divide artificialmente em duas etapas a operação de incremento da variável *count*, que poderia ser feito pela operação *count++*, para facilitar nossa ilustração da ferramenta *EventManager*.

Podemos instrumentar esse código inserindo o evento *ThreadStarts* logo após a declaração e inicialização das variáveis locais, o evento *ReadCount* antes da leitura da variável global *count* e o evento *UpdateCount*, posicionado anteriormente à atualização da variável *count*. Os eventos são bloqueados através de *waits* para controlarmos o andamento das *threads*, por isso é importante que a chamada à função de inserção de evento seja feita antes do ponto que o usuário deseja marcar como evento. Ao longo deste trabalho, usaremos a expressão *"inserir um evento X"* como uma forma resumida de *"inserir uma chamada à função checkCurrentEvent("X")"*. A Figura 3.2 mostra como ficou o código após a inserção destes eventos.

```
1 void * threadFunction (void * id) {
2   int myId = *((int *) id);
3   int aux;
4
5   printf("---Thread %d started\n", myId);
6
7   aux = count;
8
9   count = aux + 1;
10  printf("---Thread %d wrote %d\n", myId, count);
11
12  pthread_exit(NULL);
13 }
```

Figura 3.1: Exemplo de corpo de uma *thread* de um programa concorrente

```
1 void * threadFunction (void * id) {
2   int myId = *((int *) id);
3   int aux;
4
5   checkCurrentEvent("ThreadStarts");
6   printf("---Thread %d started\n", myId);
7
8   checkCurrentEvent("ReadCount");
9   aux = count;
10  checkCurrentEvent("UpdateCount");
11  count = aux + 1;
12  printf("---Thread %d wrote %d\n", myId, count);
13
14  pthread_exit(NULL);
15 }
```

Figura 3.2: Exemplo de um corpo de *thread* utilizando a ferramenta desenvolvida. Ao fazer isso, foi possível encontrar erros de concorrência no código.

Para expor o erro, podemos testar o programa da Figura 3.2 com o *script* mostrado na Figura 3.3. Este *script* descreve uma ordem de execução para o programa do exemplo. O *script* assume que o programa será executado com duas *threads* e ordena os eventos descritos ao longo de seus códigos. Neste *script*, a primeira *thread* que começar a executar é chamada de *thread1* e *executa o evento* chamado *ThreadStarts* (isto é, executa a chamada à função *checkCurrentEvent("ThreadStarts")*. Em seguida, a segunda *thread*, chamada de *thread2*, executa o evento *ThreadStarts*. Cada linha do *script* contém o nome de um evento com uma especificação das *threads* que podem executá-lo entre colchetes. Este *script* descreve uma sequência de eventos que começa na linha 1 e termina na linha 6. A linguagem do *script* contém a linguagem de eventos e a linguagem de *threads* e será explicada na Subseção 3.2.

Testando o programa da Figura 3.2 com duas *threads* e utilizando uma

ordem como a da Figura 3.3, percebemos problemas no programa do usuário. O escalonamento imposto expõe o *bug* do programa, já que permite que o valor na variável *count* seja sobrescrito por ter duas *threads* lendo o valor de *count* antes que alguma delas o atualize.

```
1 ThreadStarts [>>thread1];
2 ThreadStarts [>>thread2];
3 ReadCount [thread1];
4 ReadCount [thread2];
5 UpdateCount [thread1];
6 UpdateCount [thread2];
```

Figura 3.3: *Script* com a ordem dos eventos a ser testada no exemplo da Figura 3.2

Utilizando a EventManager para verificar como o programa anterior permitia que uma ordem inválida de eventos fosse executada, o usuário nota um erro em seu código. Ao não utilizar um mecanismo de exclusão mútua para proteger a manipulação da variável global, não houve garantia de atomicidade nas operações de leitura e escrita da variável. No Capítulo 5 veremos outros exemplos de uso da ferramenta com as devidas discussões.

3.2 Linguagem de *script*

Para descrever as sequências que podem ser aceitas pelo seu programa, o usuário escreve um *script*. A linguagem de *script* que criamos é uma *domain-specific language (DSL)* e envolve a linguagem de eventos e a linguagem de *threads*. Para a de eventos optamos por utilizar expressões regulares. Já para a de *threads*, utilizamos uma linguagem baseada em conjuntos.

3.2.1 Linguagem de eventos

Cada evento é descrito através de um identificador e possui, necessariamente, uma especificação de *threads* que podem executá-lo contida entre os caracteres '[' e ']'. A linguagem de especificação de *threads* será explicada na próxima seção.

Uma sequência de eventos ou de blocos de eventos é separada e encerrada pelo caracter '**;**'. Uma alternativa de eventos é descrita pelo caracter '|' entre os eventos ou blocos de eventos mutuamente exclusivos. Por último, o caracter '+' após o evento ou bloco de eventos indica repetição, ou seja, um bloco a ser executado uma ou mais vezes. A Figura 3.4 apresenta a gramática da linguagem de eventos. A regra gramatical *threadExp* será apresentada na gramática da linguagem de especificação de *threads*.

A Figura 3.5 contém um exemplo de *script*. O primeiro evento é o 'Event1', que contém a especificação de *thread* '[»thread1]'. Em seguida, o *script* descreve que o 'Event2' é o único evento que pode ser executado após

```
1 S <- [ \t\n]*
2 OS <- '[' S
3 CS <- ']' S
4 OP <- '(' S
5 CP <- ')' S
6 OrOp <- '|' S
7 Plus <- '+' S
8 Minus <- '-' S
9 Assign <- '>>' S
10 Sc <- ';' S
11 NotOp <- '~' S
12
13 ID <- [a-zA-Z] [a-zA-Z0-9]* S
14
15 prog <- S exp !.
16 exp <- seq Sc? (OrOp exp)?
17 seq <- iterate (Sc seq)?
18 iterate <- item Plus?
19 item <- ID OS threadExp CS / OP exp CP
```

Figura 3.4: Gramática da linguagem de eventos.

o 'Event1'. Os parênteses descrevem um bloco de eventos contido entre as linhas 3 e 7. Este bloco descreve que ou o evento 'Event3' ou o 'Event4' serão executados em seguida, por conta do caracter '|' na linha 5 que indica uma alternativa. Por último, o caracter '+' indica uma possível repetição do bloco entre as linhas 3 e 7.


```
1 Event1 [>>thread1];
2 Event2 [>>thread2];
3 (
4   Event3 [thread1];
5   |
6   Event4[thread2];
7 )+ ;
```

Figura 3.5: Exemplo de *script* com sequências, alternativa e repetição.

3.2.2 Linguagem de especificação de *threads*

Cada evento possui uma especificação de *threads* associada, contida numa string dentre os caracteres '[' e ']'. A Figura 3.6 contém a gramática da linguagem de especificação de *threads*.

Uma especificação é composta por duas partes opcionais: A primeira descreve um conjunto de *threads* e a segunda uma atualização a algum conjunto. O conjunto de *threads* especifica quais *threads* podem executar determinado evento. Este conjunto pode ser descrito por um nome de conjunto combinado com operações de união ('+'), diferença ('-') e complemento('~') entre conjuntos. Atualizações de conjuntos descrevem alguma atualização a

```
1 S <- [ \t\n]*
2 OS <- '[' S
3 CS <- ']' S
4 OP <- '(' S
5 CP <- ')' S
6 OrOp <- '|' S
7 Plus <- '+' S
8 Minus <- '-' S
9 Assign <- '>>' S
10 Sc <- ';' S
11 NotOp <- '~' S
12
13 ID <- [a-zA-Z] [a-zA-Z0-9]* S
14
15 threadExp <- term? Assign term / term
16 term <- prefixed ((Plus / Minus) prefixed)*
17 prefixed <- (Plus / Minus / Not)? threadItem
18 threadItem <- ID / OP threadExp CP
```

Figura 3.6: Gramática da linguagem de especificação de *threads*.

ser realizada no conjunto especificado caso o evento ocorra. Essas atualizações são descritas por um operador de *atualização* seguido pelo nome do conjunto especificado. O operador '»' descreve a criação de um novo conjunto contendo apenas a *thread* que realizou o evento (a *thread* corrente). O operador '»+' descreve a inserção da *thread* corrente no conjunto especificado. Já o operador '»-' descreve a remoção da *thread* atual do conjunto especificado.

A Figura 3.7 mostra exemplos de especificações de *threads* que um evento pode conter. Os exemplos das linhas 1, 2 e 3 mostram o caso da especificação conter dois termos. Na linha 1, conferimos se a *thread* que está tentando executar o evento pertence ao conjunto *group2*, que deve ter sido preenchido previamente. Caso satisfaça essa condição, a *thread* é inserida ao conjunto *group3*. Na linha 2, temos um exemplo semelhante. Conferimos se a *thread* corrente pertence ao conjunto *group3* e se sim, a retiramos deste mesmo conjunto. Na linha 3, há um exemplo com operações de conjuntos no primeiro termo. A *thread* corrente deve satisfazer a condição de estar presente no conjunto resultante da soma de *group1* com *group2*. Caso pertença, é criado um novo conjunto chamado *group4* contendo a identificação desta *thread*.

```
1 [group2 >>+ group3]
2 [group3 >>- group3]
3 [(group1 + group2) >> group4]
4 [>> group2]
5 [group1]
6 [(~(group1) - (group2))]
```

Figura 3.7: Exemplos de especificações de *thread* que um evento pode conter.

A especificação pode também conter apenas um dos termos - uma expressão com conjunto, indicando ser o termo à esquerda, ou o sinal de *assign*

seguido por um termo à direita.

No caso do termo à direita, não há verificação da *thread* que executar o evento, isto é, qualquer *thread* que o executar será aceita. O sinal de *atualização* fará com que seja criado um novo grupo contendo aquela *thread*. Este caso está explicitado no exemplo da linha 4 da Figura 3.7. Nele, qualquer *thread* pode executar o evento e a que executar será inserida no novo conjunto chamado *group2*.

Caso a especificação contenha somente um termo ou conjunto de termos sem um sinal de *atualização*, contém somente o termo à esquerda. O programa verifica se a *thread* que chamou o evento pertence ao conjunto resultante das operações deste termo mas não o insere a nenhum conjunto. A linhas 5 e 6 do exemplo da figura mostram este caso. No exemplo da linha 5, a *thread* só pode executar o evento que contém essa especificação caso ela tenha sido inserida no grupo *group1* anteriormente. No exemplo da linha 6, verificamos se a *thread* pertence ao conjunto resultante da subtração de *group2* do complementar de *group1*.

3.3 Formato do arquivo de configuração

O arquivo de configuração é opcional e permite que o usuário coloque todos os possíveis eventos que ocorrem em seu programa e eventos a serem ignorados.

O formato do arquivo de configuração é ter um nome de evento por linha. A *string* '//' no início de uma linha indica que o evento a seguir deve ser ignorado.

Desta forma, é possível evitar erros de digitação do nome de um evento no programa do usuário ou no *script*, já que se for encontrado algum evento que não esteja no arquivo de configuração, o programa encerra com erro indicando que este evento não existe.

Quando o programa tentar realizar algum evento marcado como a ser ignorado na execução do programa do usuário, a execução prossegue normalmente. Isto é útil para o usuário inserir vários eventos ao longo de seu programa e ter a possibilidade de testar conjuntos de eventos em *scripts* separados sem ter a necessidade de recompilar o programa para comentar chamadas de eventos que não serão utilizados em determinado teste.

Como o *script* é dependente do teste a ser realizado, ele não pode conter um evento que tenha sido marcado como ignorado. Caso contenha, o programa encerrará com um erro, indicando que este evento não existe nos possíveis eventos do programa do usuário.

4
Implementação

Neste capítulo, descrevemos como implementamos cada passo da *EventManager* e como testamos cada um deles.

4.1
Funcionamento geral

A ferramenta *EventManager* possui uma arquitetura dividida em módulos, que consiste em um módulo principal em C, um módulo principal em Lua e quatro módulos auxiliares, também em Lua. A *EventManager* deve ser utilizada como uma biblioteca a ser chamada pelo usuário.

O módulo principal em C implementa as funções a serem chamadas pelo usuário ao longo de seu programa. Suas funções chamam funções do módulo principal em Lua. Essa comunicação entre o módulo em C e o módulo em Lua é realizada com a *API C* de Lua (IERUSALIMSCHY, 2016) e funciona através de chamadas a funções que manipulam uma pilha virtual compartilhada pelas duas linguagens. Ao acessar as funções através de um mesmo ambiente Lua, as variáveis globais inicializadas em Lua mantêm seus valores mesmo após o controle da execução retornar a C, possibilitando que várias funções em C da *EventManager* chamem diferentes funções em Lua que manipulem uma mesma estrutura de dados nesta linguagem.

Na inicialização da biblioteca realizamos o processamento do arquivo de configuração, que gera uma lista de eventos permitidos e uma lista de eventos a serem ignorados. Em seguida, processamos o *script* passado pelo usuário, convertendo-o em uma árvore de sintaxe contendo os eventos. Esta árvore é convertida em um grafo de eventos, que descreve as possíveis sequências de eventos especificados no *script*. Cada evento possui uma especificação de *thread* associada. Durante a criação do grafo de eventos, criamos para cada evento uma árvore de sintaxe correspondente à especificação de *thread* associada.

Tendo essas estruturas preparadas, é possível começar a processar os eventos. A cada chamada da função *checkCurrentEvent*, fazemos uma consulta ao grafo de eventos verificando se este evento corresponde a alguma das arestas que saem do nó atual. Caso sim, processamos a árvore de sintaxe da especificação de *thread* associada para conferir se a *thread* atual satisfaz a condição de executar este evento. Desta forma, vamos percorrendo o grafo de eventos e controlando o escalonamento para executar uma ordem de eventos que satisfaça o *script* do usuário.

A Figura 4.1 mostra o pseudo código com o funcionamento geral da ferramenta.

Nas subseções seguintes, mostraremos as etapas do funcionamento da ferramenta com detalhes da implementação.

```
1 Inicialização (script, arq_config) {
2     eventos_permitidos, eventos_ignorados <-
3               Leitura_arquivo_configuracao (arq_config)
4     arvore <- Converte_gramatica_em_arvore (script)
5     grafo  <- Converte_arvore_em_grafo (arvore,
6               eventos_permitidos)
7 }
8
9 Processa_evento (evento, id_thread) {
10    pode_prosseguir <- Consulta_grafo_eventos (evento,
11                       eventos_ignorados)
12    SE (pode_prosseguir == TRUE) ENTAO {
13        pode_prosseguir <- Consulta_arvore_thread (id_thread)
14    }
15    RETORNA pode_prosseguir
16 }
```

Figura 4.1: Pseudo código com o funcionamento geral da *EventManager*.

4.2 Processamento do arquivo de configuração

O processamento do arquivo de configuração é feito na função de inicialização da ferramenta, a função *initializeManager*. Ela chama uma função auxiliar em Lua que lê o arquivo passado e insere em uma lista os possíveis eventos do programa. Caso o evento contenha a *string* '//', é um evento a ser ignorado. Neste caso, ele não é inserido na lista dos possíveis eventos, mas sim na lista dos eventos a serem ignorados.

4.3 Processamento do *script*

A linguagem de *script* que desenvolvemos para esta ferramenta é interpretada utilizando a biblioteca *LPeg* de Lua. Esta biblioteca faz reconhecimento de padrões se baseando em *Parsing Expression Grammars*, que permite uma maior descrição de padrões do que utilizar expressões regulares puras (IERUSALIMSCHY, 2009).

Usamos a *LPeg* para definir uma gramática e converter o *script* em uma árvore de sintaxe abstrata (AST) de eventos a partir da qual, posteriormente, criamos o grafo de eventos. As próximas seções descrevem as funções de criação da árvore e de sua transformação em grafo.

4.3.1 Criação da árvore de sintaxe da linguagem de *script*

A função de criação da árvore de sintaxe cria uma gramática utilizando *LPeg* e analisa o *script* de eventos. Caso haja algum erro de escrita no *script* que não permita sua interpretação, o programa encerra mostrando em qual ponto do *script* este erro está.

A árvore resultante do processamento do arquivo é uma árvore em que cada nó que não é uma folha contém um campo chamado *tag*. As *tags* possíveis são: *seq*, *or*, *plus* e *item*. O único nó com *tag* com filhos que são folhas são os *item*. Neste caso, o primeiro filho é o nome do evento e o segundo filho é a especificação de *thread* associada a este evento. Os nós com *tag seq* têm como primeiro filho o primeiro *item* da sequência (isto é, o primeiro evento) e como segundo filho um nó com *tag* que representa o restante da sequência. De forma análoga, os nós com *tag or* são criados a partir de alternativas e seus dois filhos são os eventos ou blocos de eventos desta alternativa. Já os nós *plus* são criados a partir da identificação da repetição de um evento ou de um bloco de eventos. Sendo assim, eles têm somente um filho que é, necessariamente, um nó com *tag*.

A Figura 4.3 contém a árvore resultante do processamento do *script* descrito anteriormente (A Figura 4.2 relembra este *script*). Note que a sequência de dois eventos seguida por um bloco de eventos foi separada em um nó *seq* tendo o primeiro evento da sequência como primeiro filho e o restante da sequência, representado pelo segundo nó *seq*, como segundo filho. Além disso, perceba que o nó *plus* engloba todo o bloco de eventos indicado pelo caracter '+' como repetição.


```
1 Event1 [>>thread1];
2 Event2 [>>thread2];
3 (
4   Event3 [thread1];
5   |
6   Event4[thread2];
7 )+ ;
```

Figura 4.2: Exemplo de *script* com sequências, alternativa e repetição, apresentado anteriormente.

4.3.2 Transformação da árvore de sintaxe no grafo de eventos

O grafo gerado a partir da AST de eventos é um autômato finito não determinístico (NFA), cujas arestas correspondem a eventos e os nós a estados do programa: em cada estado apenas determinados eventos (arestas com origem naquele nó) podem ser aceitos. A Figura 4.4 mostra o grafo criado a partir da árvore de sintaxe da Figura 4.3. Perceba que cada aresta possui um nome de um evento associado assim como sua especificação de *thread*. Do nó de número 3, é possível ocorrer uma repetição causada tanto pelo evento *Event3* como pelo *Event4* ou o programa encerrar, também através destes mesmos eventos.

A estrutura de dados usada para representar o grafo é uma tabela Lua na qual cada entrada representa um nó do grafo e contém outra tabela. Dentro destas tabelas internas, cada entrada representa uma aresta que sai deste nó e tem como chave o nome do evento associado a esta aresta. O valor de cada

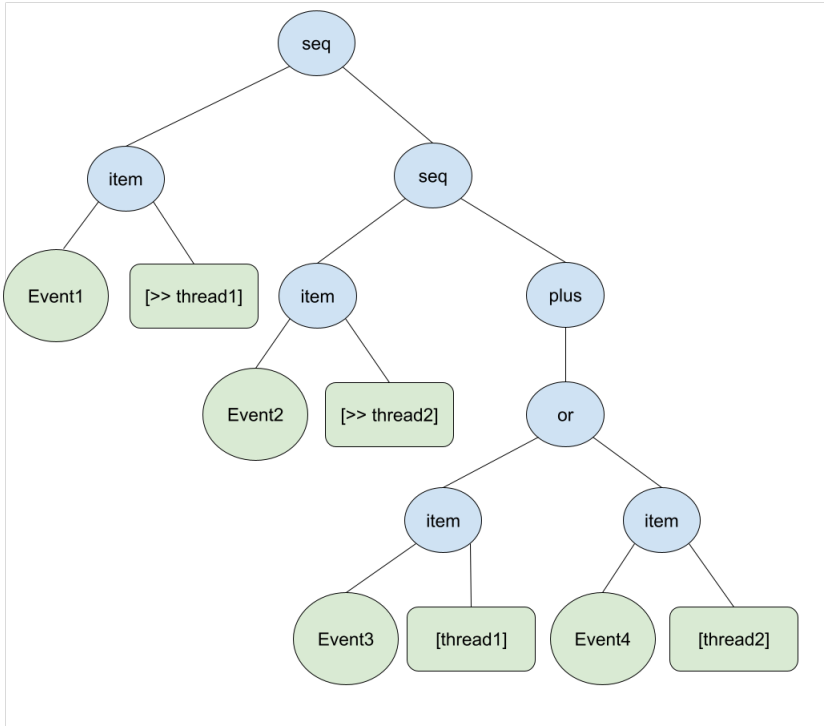

Figura 4.3: Árvore de sintaxe criada a partir da interpretação do *script* da Figura 3.5

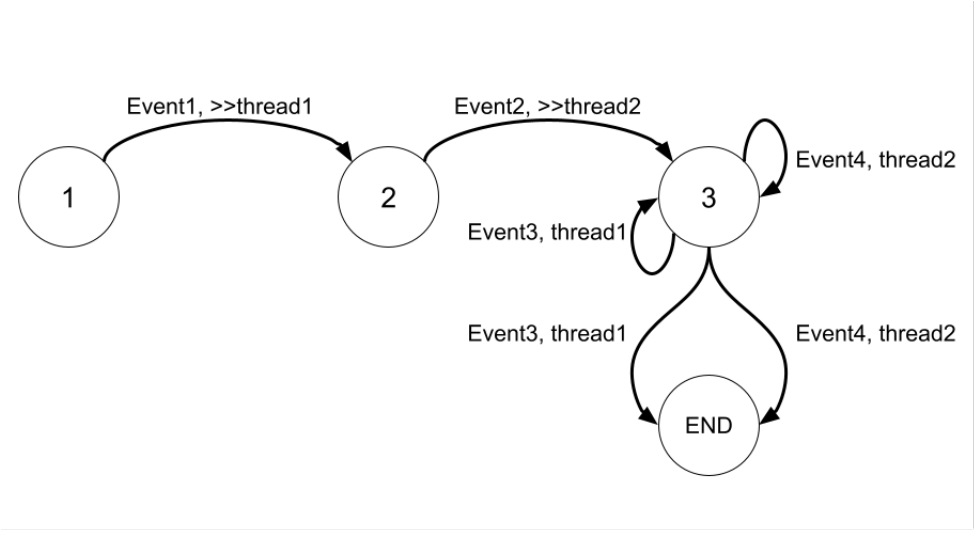

Figura 4.4: Grafo criado a partir da árvore de sintaxe da Figura 4.3

entrada contém uma tabela Lua com os nós destinos da aresta, assim como uma tabela que representa a especificação da *thread* associada a este evento. Caso uma aresta leve ao estado final do programa, um dos seus nós de destino possui o valor -1. Na Figura 4.5, vemos a estrutura da variável *graph*, que representa o grafo do exemplo da Figura 4.4. Note que o nó denominado "*END*" na representação visual do grafo não é explicitamente criado na estrutura de dados do grafo.

Para criar este grafo, fizemos o processamento da árvore de forma recursiva. Como os eventos representam as arestas do grafo e não os nós,

```
graph = {
    1 = {
        Event1 = { {2}, table: 0x7fbe74405ac0 }
    },
    2 = {
        Event2 = { {3}, table: 0x7fbe745072c0 }
    },
    3 = {
        Event3 = { {3, -1}, thread1 },
        Event4 = { {3, -1}, thread2 }
    }
}
```

Figura 4.5: Estrutura de dados que representa o grafo gerado a partir do *script* da Figura 3.5

precisamos inferir quando criar um novo nó e quando não criar. Por exemplo, alternativas nunca geram novos nós, somente criam duas arestas que chegam a um mesmo nó. Já sequências sempre geram um novo nó no grafo. Repetições podem gerar novos nós ou não, como explicaremos adiante.

Além de entender quando criar novos nós ou não, o grande desafio desta etapa foi capturar o nó de destino de cada aresta. A cada passo da recursão, chamamos imediatamente a função recursiva para os filhos de cada nó. Somente na volta da recursão conseguimos fazer a conexão dos nós através das arestas, pois é quando identificamos os nós de destino de cada aresta. A Figura 4.6 mostra uma representação visual de quando é possível identificar o nó de destino de cada evento.

Depois de desenvolver uma primeira versão, percebemos que não implementamos corretamente a operação de repetição de um algoritmo de construção de um NFA. Os grafos criados admitiam que a repetição fosse ignorada, como se ela representasse que um bloco de eventos deveria ocorrer zero ou mais vezes, quando na verdade ela indica que este bloco deve ocorrer uma ou mais vezes. Além disso, ocorriam erros quando havia uma alternativa dentro de uma repetição, permitindo ordens incoerentes com o *script* criado pelo usuário. Solucionamos este erro criando nós auxiliares de forma similar aos criados pelo algoritmo de construção de Thompson, que constrói um NFA a partir de uma expressão regular (THOMPSON, 1968). A Figura 4.7 mostra um exemplo de *script* que contém uma alternativa dentro de uma repetição. O usuário deseja que o bloco de eventos *Event1* e *Event2* seja executado de forma sequencial, podendo o evento *Event1* ocorrer uma ou mais vezes. Este bloco está em uma alternativa com o bloco de eventos *Event3* e *Event4*. De forma análoga, o evento *Event3* pode ocorrer uma ou mais vezes. Antes de criarmos os nós auxiliares, o grafo criado era como o grafo *(a)* da Figura 4.8. Note que o grafo permitia uma ordem de eventos como: *Event1 -> Event3 -> Event4*, já que a repetição do evento *Event1* fazia com que ele voltasse ao nó 1, podendo ter como próximo evento o *Event3*. Esta ordem não condiz com o esperado pelo *script* do usuário . Após corrigirmos o erro, criamos um grafo como o *(b)* dessa mesma figura. Como é possível perceber, o caminho que descrevemos

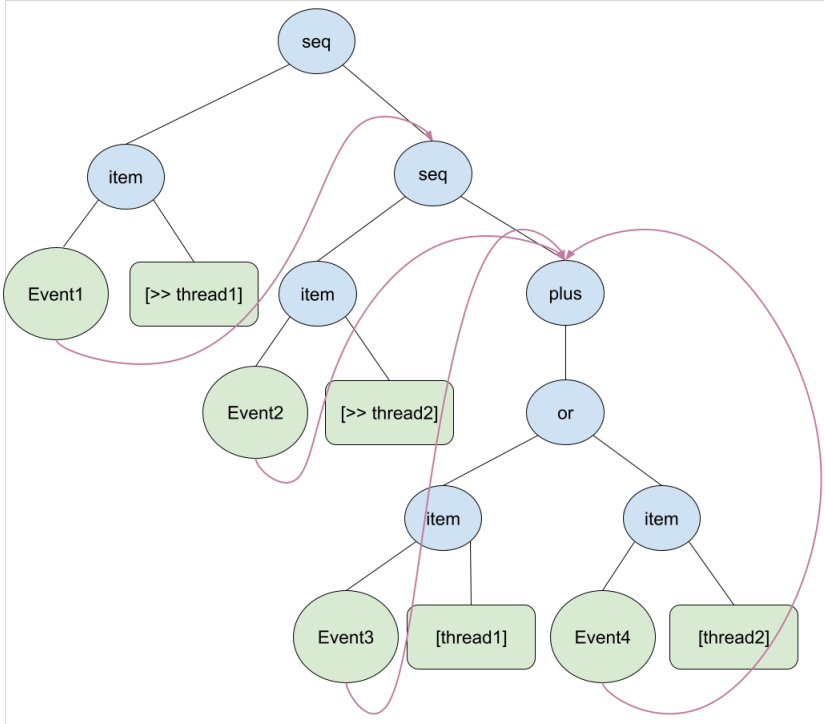

Figura 4.6: Árvore de sintaxe da Figura 4.3 com ligações que exemplificam o nó de destino de cada evento.

anteriormente não existe mais. Neste grafo, o primeiro nó representa o estado do programa antes da repetição, onde o *script* exige uma ocorrência de algum dos blocos *plus*. O segundo e quinto nó representam os estados seguintes, onde o *script* admite mais uma ocorrência deste bloco mas não mais a exige.

```
1 (
2   (
3     (Event1 [>> thread1])+;
4     Event2 [thread1];
5   )
6   |
7   (
8     (Event3 [>> thread2])+ ;
9     Event4 [thread2];
10  )
11 ) +;
```

Figura 4.7: Exemplo de *script* com repetição.

Para caminhar no grafo ao longo da execução do programa, é preciso armazenar qual o nó corrente a cada evento executado. Este nó começa sempre com o valor 1, sendo o primeiro estado do programa. Como o grafo é um NFA, é possível que exista mais de um nó corrente. Por isso, o nó corrente é uma variável que armazena o conjunto de possíveis nós correntes do programa. Um desses casos ocorre quando temos um mesmo evento podendo levar a diferentes nós do grafo. Quando esse evento é executado, o próximo evento pode ser uma

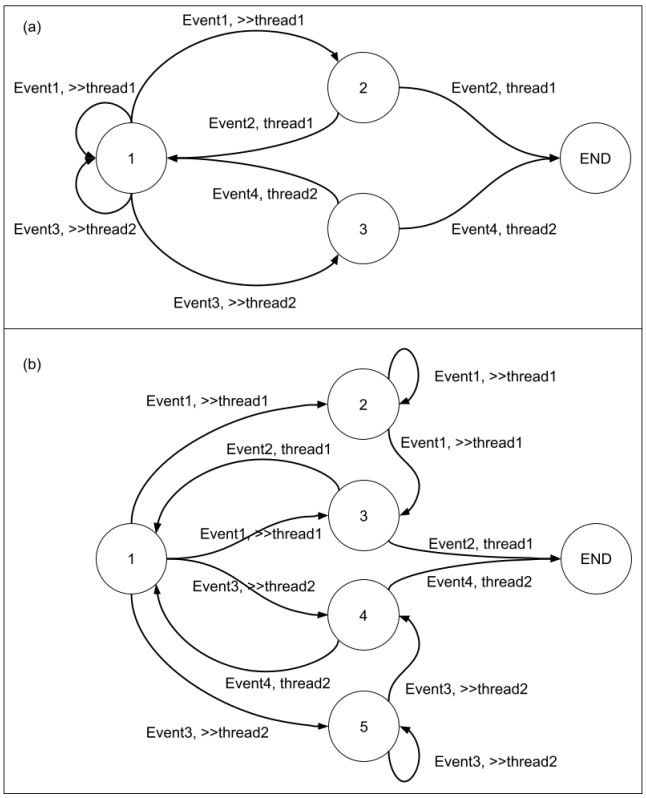

Figura 4.8: Grafos criado a partir do *script* da Figura 4.7. (a) Mostra o grafo sem os nós auxiliares, podendo ocorrer ordens não esperadas pelo *script* do usuário. (b) Mostra a correção que fizemos no grafo para corrigir estes erros.

aresta saindo de qualquer um dos nós aos quais o evento anterior levava. A Figura 4.9 contém um grafo que mostra um exemplo disso. Quando o evento *Event1* é executado, não é possível afirmar se o nó atual do programa é o 1 ou o 2, já que este evento leva a estes dois nós. Por isso, ambos os nós são adicionados à variável que armazena o possível nó corrente. Assim, é permitido que o evento *Event1* seja executado novamente ou que o evento *Event2* seja executado.

Por se tratar de um NFA, existe a possibilidade de um mesmo nó conter várias arestas com o mesmo nome de evento mas com especificações de *thread* diferentes. Isso se torna mais preocupante quando essas especificações permitem que o mesmo grupo de *threads* execute o mesmo evento mas possuem atualizações diferentes nos conjuntos de *threads*. Neste caso, não é possível afirmar qual das arestas o programa irá seguir, e nem determinar qual conjunto deve ser atualizado.

Fizemos um algoritmo para conferir se o *script* do usuário gera um grafo com algum conjunto de arestas incoerentes, ou seja, que possuem o mesmo evento mas atualizações de conjuntos de *threads* diferentes. Percorremos o grafo montando conjuntos de nós que poderiam ser correntes juntos em algum estado do programa do usuário (ou seja, eles são nós de destino de arestas com os mesmos nomes de eventos). Conferimos se as arestas que saem de cada um dos nós de um mesmo conjunto são incoerentes, tomando o cuidado de não conferir um mesmo conjunto mais de uma vez. Este algoritmo é executado após a construção do grafo e, caso exista algum conjunto de arestas incoerentes,

encerramos o programa do usuário reportando um erro.

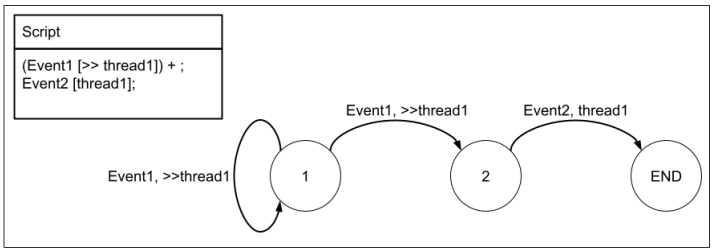

Figura 4.9: *Script* com grafo que mostra um exemplo de que podem existir vários possíveis nós correntes.

Quando a função recursiva processa um nó com *tag item*, temos acesso aos seus dois filhos: o primeiro contém o nome do evento e o segundo a especificação da *thread* associada a este evento. Neste momento, chamamos a função que converte a *string* que contém essa especificação em uma estrutura de dados para ser manipulada ao longo do programa. Este passo será descrito na próxima Subseção.

4.3.3 Criação da árvore de sintaxe da sublinguagem de especificação de *threads*

A função que cria a árvore de sintaxe da linguagem de especificação de *threads* é chamada durante a criação do grafo de eventos para cada linha do *script* (exceto as que indicam eventos ignorados).

Esta árvore é construída a partir de uma gramática criada também com a biblioteca *LPeg*. Assim como no caso da árvore de sintaxe dos eventos, os nós que são folhas desta árvore não possuem *tags*. Os outros possuem uma *tag* dentre: *assign*, *plus*, *minus*, *not* e *item*.

Os nós com *tag item* contêm apenas um filho, que é um nó com o nome do conjunto de *threads* especificado. Os nós com *tag assign* podem conter um ou dois filhos. Se o nó do *assign* tiver dois filhos, significa que é um caso no qual a *thread* que executar o evento precisa satisfazer as especificações das ramificações do primeiro filho do *assign* para não ser bloqueada e ser inserida ou removida do conjunto especificado pelo segundo filho, como é o caso dos exemplos das letras *(a)*, *(b)* e *(c)* da Figura 4.10. Se tem somente um filho, é um caso como os das letras *(d)*, *(e)* e *(f)*. No primeiro, ocorre um *assign* no conjunto *group2* da *thread* que executar o evento associado, sem restrições sobre qual *thread* é aceita. Já os dois últimos não possuem atualizações de nenhum conjunto, somente especificam um conjunto de *threads* aceitas: na letra *(e)*, qualquer *thread* que pertença ao conjunto *group1* é aceita, enquanto que na letra *(f)*, qualquer *thread* que pertença ao conjunto resultante da diferença entre o complemento de *group1* e o *group2* é aceita.

A estrutura de dados que armazena os *ids* das *threads* contidas em cada conjunto de *threads* é uma tabela Lua na qual cada entrada possui uma chave com o nome do conjunto e o valor associado sendo uma tabela contendo os *ids* das *threads* deste conjunto. A Figura 4.11 mostra um exemplo do estado da variável global *threadIdTable* em um ponto do programa. Perceba que um

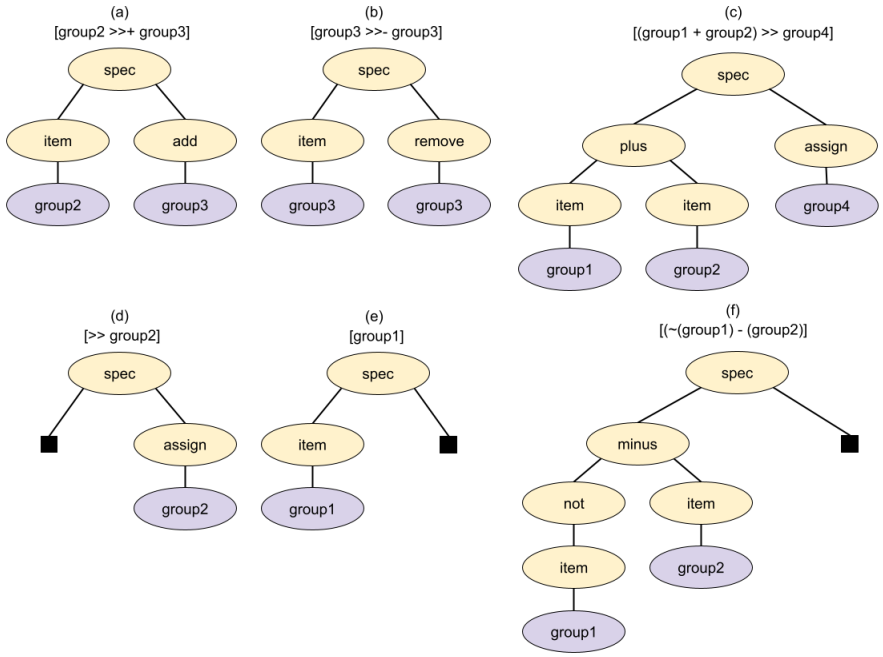

Figura 4.10: Árvores de cada exemplo de especificação de *thread* apresentado na Figura 3.7.

mesmo *id* pode estar em diferentes grupos e que podem existir grupos vazios após a remoção de alguma *thread* de um grupo.

```
1 threadIdTable = {
2     group1 = {32220, 43330},
3     group2 = {32220},
4     group3 = {}
5 }
```

Figura 4.11: Estrutura de dados que mostra as *threads* contidas em cada conjunto.

Como cada conjunto de *threads* descrito na linguagem de especificação de *threads* depende e muda conforme os eventos que ocorrem ao longo do programa, não é possível processar a árvore nesta etapa de pré-processamento do *script*, somente no momento da análise de cada evento.

4.4 Análise de eventos

Os eventos são analisados a cada chamada da função *checkCurrentEvent*/*checkCurrentEventWithId*. Estas duas funções verificam se o evento passado como parâmetro pode ser o próximo evento a ser executado, de acordo com o *script* passado como teste. Caso este evento possa ser executado, a função retorna para o programa do usuário. Caso contrário, a função coloca a *thread* que a chamou em espera através da chamada da função *pthread_cond_wait* da biblioteca *pthread*. A espera acaba quando uma *thread* consegue

executar um evento e manda, através de um *broadcast*, um sinal para acordar todas as outras *threads* para que elas possam verificar novamente se o evento que elas estão executando pode ser o próximo evento. Isto é feito através da função *pthread_cond_broadcast*, também da biblioteca *pthread*.

A decisão se o evento pode ser executado ou não depende de dois fatores: verificação no grafo de eventos e verificação da *thread* corrente. Ambos serão detalhados nas subseções a seguir.

4.4.1 Verificação de próximos eventos no grafo de eventos

O grafo de eventos criado na etapa descrita na Subseção 4.3.2 mostra os possíveis caminhos que o programa pode seguir, de acordo com o *script* do usuário. O programa inicia no nó de número 1 do grafo. A partir daí, cada vez que a função *checkCurrentEvent* é chamada com um evento, conferimos se há uma aresta saindo do nó corrente do grafo (conjunto de possíveis nós) com o nome do evento passado. Caso haja, é feita uma verificação da *thread* que está tentando executar este evento levando em consideração as especificações de *threads* descritas pela aresta do grafo correspondente ao evento. Caso este evento não esteja em nenhuma aresta que sai do nó corrente, o evento passado ainda não pode ser executado, então a *thread* que o chamou executa um *wait*, se colocando em espera.

4.4.2 Verificação da *thread* responsável pelo evento

Nesta etapa, verificamos se a *thread* que está tentando executar este evento satisfaz as regras de especificação de *thread* associadas ao mesmo. Para isso, tratamos a árvore de sintaxe da sublinguagem de especificação de *threads*, descrita anteriormente na Subseção 4.3.3.

A função responsável por esta etapa processa a árvore de forma recursiva, resolvendo as operações de conjunto e realiza inserções e remoções nas cópias das listas contidas na variável global *threadIdTable* para obter um conjunto final de *ids* de *threads*. O programa verifica se o *id* da *thread* que está tentando executar aquele evento pertence a esse conjunto final. Caso satisfaça, a operação de *assign* pode ser tratada, se existir. Ou seja, o *id* desta *thread* é inserido ou removido do conjunto do lado direito do *assign*.

Se a verificação da *thread* que está executando o evento aprová-la, o nó atual do grafo de eventos é modificado para o nó de destino da aresta correspondente a este evento.

4.5 Ocorrência de *deadlock* causado pela ferramenta

Como as *threads* se colocam em espera caso não seja a vez delas executarem, é possível que não seja a vez de nenhuma *thread* que tentou executar algum evento e que as outras *threads* do programa estejam bloqueadas pelo próprio programa do usuário e não tenham chegado em nenhum evento. Neste caso, é possível que tenha ocorrido um bloqueio causado pela ferramenta. Se-

guindo à risca o conceito de *deadlock*, este bloqueio não se caracteriza como tal. Entretanto, vamos chamá-lo de *deadlock* informalmente no resto do texto para facilitar o entendimento.

A ocorrência ou não deste bloqueio permite que o usuário realize uma análise de corretude do seu programa. Caso o *script* que ele tenha passado como parâmetro represente uma ordem viável de acordo com a semântica do problema, é esperado que não haja bloqueios. Desta forma, se ocorrer um bloqueio, o usuário tem como verificar em que ponto ele aconteceu e saber qual parte do seu programa pode estar causando erros. De forma análoga, de um *script* com uma ordem que não deveria poder ser executada até o final é esperada a ocorrência de um bloqueio causado pela ferramenta. Caso este bloqueio não ocorra ou ocorra em outro ponto sem ser o esperado pelo usuário, existe a possibilidade do programa conter erros.

4.5.1 Detectando possível bloqueio

Para detectar possíveis bloqueios causados pela ferramenta, cada vez que um evento é executado, um alarme de 5 segundos é criado através da função *alarm* da biblioteca em C *unistd.h*. Esta função gera um sinal do tipo SIGALRM após o tempo especificado ter terminado. Um mesmo programa só pode ter um alarme ativo de cada vez. Desta forma, chamadas à função *alarm* sobrescrevem um possível alarme anterior com o novo valor passado.

Assim, é possível capturar este sinal através de um *handler* para este tipo de sinal. Quando nenhum evento é executado dentro de 5 segundos após o último evento, consideramos como sendo um possível bloqueio causado pela ferramenta e o *handler* do sinal de alarme é chamado. Este *handler* chama uma função que retorna uma lista com o nome de todos os possíveis próximos eventos que poderiam ser executados após o útimo evento executado. Ou seja, retorna os eventos de cada aresta dos nós correspondentes ao possível estado atual.

Como o tempo entre cada evento depende também do programa do usuário, caso ele tenha alguma função muito demorada entre os eventos ou algum tipo de bloqueio entre os mesmos, é possível que o tempo de 5 segundos entre eventos seja pouco para afirmar a possível ocorrência de um bloqueio causado pela ferramenta. Por isso, o usuário pode modificar este tempo.

Como não estamos contabilizando se o total de *threads* que estão em espera é igual ao total de *threads* do programa, não podemos afirmar que ocorreu um bloqueio. Por isso, informamos ao usuário que ocorreu um *possível* bloqueio após o tempo estipulado ter passado e nenhuma *thread* ter voltado a executar.

4.6 Testes de funcionamento

A *EventManager* foi testada a cada etapa desenvolvida. Para testar as gramáticas, diversos casos foram utilizados e a árvore de sintaxe resultante de cada um deles foi comparada com o resultado esperado.

Para os testes da conversão da árvore para o grafo, testamos diferentes combinações entre sequências, alternativas e repetições e comparamos o resultado com o que desejávamos.

Como a verificação da ordem dos eventos dependeria do programa concorrente e do escalonamento desses eventos, criamos simulações de programas. Para um determinado *script*, criamos sequências de chamadas à função *checkEvent* em Lua que simulam as chamadas realizadas por cada *thread*. A Figura 4.12 contém um exemplo de *script* utilizado em um de nossos testes da sublinguagem de especificação de *threads*. Nas chamadas da função *checkEvent*, passamos um número inteiro que simula o valor do *id* de uma *thread*. Ao executar o código mostrado na Figura 4.13, podemos testar se a ferramenta está controlando da forma correta quais *threads* podem executar determinado evento.

```
1 (Event1 [>> id1]);
2 (Event2 [id1]);
3 (Event3 [id1 >> group1]);
4 (Event4 [ >>+ group1]);
5 (Event5 [ >>- group1]);
6
7 (Event6 [~id1 >> notid1]);
8 (Event7 [id1 + notid1 >> union]);
9 (Event8 [>>+ union]);
10 (Event9 [(union - id1)]);
```

Figura 4.12: *Script* utilizado em uma das simulações de programas.

```
1 readEventsFile("../GraphTests/threadIdTest1.txt")
2
3 checkEvent ("Event1", 1)
4 checkEvent ("Event2", 1)
5 checkEvent ("Event3", 1)
6 checkEvent ("Event4", 2)
7 checkEvent ("Event5", 2)
8 checkEvent ("Event6", 3)
9 checkEvent ("Event7", 1)
10 checkEvent ("Event8", 2)
11 checkEvent ("Event9", 1)
12 checkEvent ("Event9", 2)
```

Figura 4.13: Simulação de um programa concorrente através de chamadas consecutivas de eventos.

Após verificar o funcionamento de cada parte da ferramenta, criamos programas que implementam problemas clássicos de concorrência para realizar testes reais da aplicação da *EventManager*. Estes programas e seus testes serão apresentados no Capítulo 5.

5
Problemas de concorrência

Para testar a ferramenta e entender as limitações das linguagens de eventos e da sublinguagem de especificação de *threads*, utilizamos a *EventManager* em programas que implementam soluções para os seguintes problemas clássicos de concorrência: problema dos leitores e escritores, problema dos produtores e consumidores, problema da barreira e problema dos filósofos.

Em alguns problemas, utilizamos soluções clássicas apresentadas por diversos autores como (ANDREWS, 1999; TANENBAUM; BOS, 2014; DOWNEY, 2009). Em outros, apresentamos soluções corretas e incorretas que criamos.

Cada problema será apresentado em uma subseção dividida em três partes: apresentação do problema; soluções instrumentadas com os testes que criamos e resultados destes testes; discussão de aprendizados que o problema gerou.

No final desse capítulo, apresentamos uma discussão crítica das linguagens que criamos assim como possíveis melhorias para a *EventManager*.

5.1
Problema dos leitores escritores

O problema dos leitores e dos escritores é um problema clássico de sincronização que ilustra o caso de *threads* concorrentes lerem e modificarem um mesmo recurso. Este recurso pode ser uma estrutura de dados, uma tabela de um banco de dados, um arquivo, etc (ANDREWS, 1999).

As operações de leitura e escrita devem manter a consistência do estado do recurso compartilhado. Como os leitores não modificam o recurso compartilhado, pode haver mais de um leitor na região crítica simultaneamente. Já no caso dos escritores, o programa precisa garantir que cada escritor tenha acesso exclusivo à região crítica (JAGANNATH et al., 2011).

O padrão de exclusão deste problema é chamado de exclusão mútua categórica, na qual a presença de uma *thread* na regiao crítica não necessariamente exclui outras *threads* (como é o caso das *threads* leitoras). Mas se uma *thread* de uma categoria específica estiver na região crítica, nenhuma outra pode estar (como é o caso das *threads* escritoras) (JAGANNATH et al., 2011).

5.1.1
Testes

A Figura 5.1 mostra o corpo das *threads* leitoras. Implementamos um código com um *buffer* de tamanho 1. O semáforo *mutexR* protege a manipulação da variável *activeReaders*, que conta quantos leitores estão na região crítica juntos. O primeiro leitor a chegar adquire o semáforo *rw*, que protege a região crítica. Os próximos leitores não precisam fazer a aquisição desse semáforo, e escritores são impedidos de entrar por não conseguirem adquiri-lo. Após ler o

elemento do *buffer*, os leitores saem da região crítica. Quando o último leitor sai, ele libera o semáforo *rw*.

Esse padrão presente no código dos leitores é conhecido como *Lightswitch*: a primeira *thread* a entrar em uma região adquire um semáforo e a última a sair o libera. O nome foi dado como uma analogia a um interruptor de luz: a primeira pessoa a entrar em um quarto acende a luz e a última a sair a desliga (DOWNEY, 2009).

Inserimos o evento *ReaderWantsToStart* na linha 7 do código para indicar que um leitor deseja iniciar uma iteração. O evento *FirstReader* na linha 13 só é executado pelo primeiro leitor. Em seguida, cada leitor que entra executa o evento *ReaderStarts* (linha 18). O último leitor a sair da região crítica executa o evento *LastReader* (linha 27). Por último, cada leitor executa o evento *ReaderEnds* para indicar o término de uma iteração (linha 31).

```
1  void* readTask (void * num)
2  {
3    int id = *((int *) num);
4    int element;
5    while(1)
6    {
7          checkCurrentEvent("ReaderWantsToStart");
8
9          sem_wait(mutexR); //P(mutexR)
10
11         activeReaders ++;
12         if (activeReaders == 1) { //if first, get lock
13             checkCurrentEvent("FirstReader");
14             sem_wait(rw); //P(rw)
15         }
16         sem_post(mutexR); //V(mutexR)
17
18         checkCurrentEvent("ReaderStarts");
19
20         /* read the database */
21         element = buffer;
22         printf("----Reader %d reads %d\n", id, element);
23
24         sem_wait(mutexR);
25         activeReaders --;
26         if (activeReaders == 0) { //if last, release lock
27             checkCurrentEvent("LastReader");
28             sem_post(rw); //V(rw)
29         }
30
31         checkCurrentEvent("ReaderEnds");
32
33         sem_post(mutexR); //V(mutexR)
34   }
35   pthread_exit(NULL);
36 }
```

Figura 5.1: Corpo das *threads* leitoras.

A Figura 5.2 mostra o corpo das *threads* escritoras. Como só pode haver uma *thread* escritora por vez na região crítica, não há um *Lightswitch*. Cada escritor precisa adquirir o semáforo *rw* para poder escrever no *buffer*. Por último, o escritor libera o semáforo para encerrar essa iteração.

Inserimos os eventos *WriterWantsToStart* e *WriterStarts* englobando a aquisição do semáforo *rw*. Na linha 17, inserimos o evento *WriterEnds* para indicar que um escritor encerrou uma iteração.

```
1  void* writeTask (void * num)
2  {
3    int id = *((int *) num);
4    int element;
5
6    while(1)
7    {
8          checkCurrentEvent("WriterWantsToStart");
9          sem_wait(rw); //P(rw)
10         checkCurrentEvent("WriterStarts");
11
12         /* write the database */
13         element = rand()%200;
14         buffer = element;
15         printf("----Writer %d writes %d\n", id, element);
16
17         checkCurrentEvent("WriterEnds");
18
19         sem_post(rw); //V(rw)
20   }
21   pthread_exit(NULL);
22 }
```

Figura 5.2: Corpo das *threads* escritoras.

Para verificarmos se a semântica do problema está correta, podemos fazer três testes. O primeiro e o segundo testam se os escritores têm acesso exclusivo do *buffer*, tentando colocar o programa em um estado errado. Já o terceiro testa se vários leitores conseguem entrar na região crítica juntos, verificando se o programa aceita uma sequência válida.

Para o primeiro teste, fizemos o *script* mostrado na Figura 5.3. Utilizamos um leitor e dois escritores. A sequência do *script* mostra que um escritor deve executar o evento *WriterWantsToStart* e um escritor diferente (garantido pela operação de *not* na frente do grupo de *threads tWriter*) deve executar este mesmo evento. Em seguida, ambos devem executar o evento *WriterStarts*, indicando que os dois entraram na região crítica. Por último, os dois executam o evento de *WriterEnds*.

Na Figura 5.4 vemos o resultado da execução do programa dos leitores e escritores com o *script* da Figura 5.3. Como era esperado de um programa correto, o segundo escritor não conseguiu executar o evento *WriterStarts* e, portanto, não conseguiu entrar na região crítica junto com o primeiro escritor.

```
1 (WriterWantsToStart [>>tWriter]);
2 (WriterWantsToStart [~tWriter]);
3
4 (WriterStarts [tWriter]);
5 (WriterStarts [~tWriter]);
6
7 (WriterEnds [~tWriter]);
8 (WriterEnds [tWriter]);
```

Figura 5.3: *Script* descrevendo o caso de dois escritores entrarem na região crítica juntos

```
1 [110739456] executed WriterWantsToStart
2 [111276032] executed WriterWantsToStart
3 [110739456] executed WriterStarts
4 ----Writer 0 writes 103
5
6 END
7
8 Possible blocking detected!!
9
10 The input event sequence wasn't accepted by your program. Please note
       that:
11 -- If the input sequence was supposed to be invalid, this means that
       your program might be ok.
12 -- If the input sequence was supposed to be valid, this means that your
        program might have some issues.
13
14 Possible Events Expected:
15    -> WriterStarts
```

Figura 5.4: Resultado da execução do programa dos leitores e escritores com o *script* da Figura 5.3

O segundo teste verifica o caso de um leitor entrar na região crítica enquanto um escritor estiver acessando o *buffer*. A Figura 5.5 mostra o *script* que desenvolvemos para este teste. Um escritor começa executando os eventos *WriterWantsToStart* e *WriterStarts*, indicando que entrou na região crítica. Em seguida, um leitor executa os eventos *ReaderWantsToStart*, *FirstReader* e *ReaderStarts*, para acessar a região crítica por ser o primeiro leitor. Por último, o escritor executa o evento *WriterEnds*, saindo da região crítica e o leitor executa o evento análogo *ReaderEnds*.

A Figura 5.6 mostra o resultado deste teste. Como o esperado de um programa correto, o leitor conseguiu executar os eventos *ReaderWantsToStart* e *FirstReader*, que vem antes da aquisição do semáforo que protege a região crítica. Ele não conseguiu executar o evento *ReaderStarts*, indicando que não conseguiu entrar na região crítica.

Para o terceiro teste, fizemos um *script* que descreve o caso de três leitores entrarem na região crítica juntos após um escritor ter escrito algo. A Figura 5.7 mostra este *script*. As linhas 1-4 descrevem os eventos de um escritor

```
1 (WriterWantsToStart [>>tWriter]);
2 (WriterStarts [tWriter]);
3
4 (ReaderWantsToStart [>>tReader]);
5 (FirstReader [tReader]);
6 (ReaderStarts [tReader]);
7
8
9 (WriterEnds [tWriter]);
10 (ReaderEnds [tReader]);
```

Figura 5.5: *Script* descrevendo o caso de um escritor e um leitor entrarem na região crítica juntos

```
1 [265162752] executed WriterWantsToStart
2 [265162752] executed WriterStarts
3 ----Writer 0 writes 103
4 [264626176] executed ReaderWantsToStart
5 [264626176] executed FirstReader
6
7 END
8
9 Possible blocking detected!!
10
11 The input event sequence wasn't accepted by your program. Please note
       that:
12 -- If the input sequence was supposed to be invalid, this means that
       your program might be ok.
13 -- If the input sequence was supposed to be valid, this means that your
        program might have some issues.
14
15 Possible Events Expected:
16    -> ReaderStarts
```

Figura 5.6: Resultado da execução do programa dos leitores e escritores com o *script* da Figura 5.5

escrevendo no *buffer*. Em seguida, temos três eventos *ReaderWantsToStart*, indicando que temos três leitores tentando acessar o *buffer* e sendo todos adicionados ao grupo de *threads* chamado *tReaders*. O primeiro que conseguir entrar e que fizer parte de *tReader* executa o evento *FirstReader*. Em seguida, os três executam o evento *ReaderStarts*. Depois, temos uma ou mais execuções do evento *ReaderEnds*. O último leitor a sair da região crítica executa o evento *LastReader* seguido pelo evento *ReaderEnds*. Vale ressaltar que somente o último leitor a sair da região crítica executa o evento *LastReader*. Sendo assim, quando este evento for executado, todos os outros leitores já encerraram a execução de uma iteração.

A Figura 5.8 mostra o resultado da execução do *script* apresentado na Figura 5.7. Vemos que a sequência foi executada por completo, como era o esperado de um programa que implementa corretamente o padrão de *Lightswitch*.

```
1 (WriterWantsToStart [>>tWriter]);
2 (WriterStarts [tWriter]);
3 (WriterEnds [tWriter]);
4
5 (ReaderWantsToStart [>>  tReaders]);
6 (ReaderWantsToStart [>>+ tReaders]);
7 (ReaderWantsToStart [>>+ tReaders]);
8 (FirstReader [tReaders]);
9
10 (ReaderStarts [tReaders]);
11 (ReaderStarts [tReaders]);
12 (ReaderStarts [tReaders]);
13
14 (ReaderEnds [tReaders >>- tReaders])+ ;
15
16 (LastReader [tReaders]);
17 (ReaderEnds [tReaders >>- tReaders]);
```

Figura 5.7: *Script* descrevendo o caso de três leitores entrarem na região crítica juntos

```
1 [165851136] executed WriterWantsToStart
2 [165851136] executed WriterStarts
3 ----Writer 0 writes 103
4 [165851136] executed WriterEnds
5 [164241408] executed ReaderWantsToStart
6 [165314560] executed ReaderWantsToStart
7 [164777984] executed ReaderWantsToStart
8 [164241408] executed FirstReader
9 [164241408] executed ReaderStarts
10 ----Reader 0 reads 103
11 [165314560] executed ReaderStarts
12 ----Reader 2 reads 103
13 [164777984] executed ReaderStarts
14 ----Reader 1 reads 103
15 [164241408] executed ReaderEnds
16 [165314560] executed ReaderEnds
17 [164777984] executed LastReader
18 [164777984] executed ReaderEnds
19
20 END
21
22 Finished executing the script!!
23
24 The input event sequence was accepted by your program. Please note that
       :
25 -- If the input sequence was supposed to be invalid, this means that
       your program might have some issues.
26 -- If the input sequence was supposed to be valid, this means that your
        program might be ok.
```

Figura 5.8: Resultado da execução do programa dos leitores e escritores com o *script* da Figura 5.7

Esta solução do problema de leitores e escritores funciona para sistemas em que a carga do sistema é baixa. Entretanto, se a carga aumentar, aumentam as chances de ocorrer *starvation* nos escritores (DOWNEY, 2009). Essa situação ocorre quando leitores não param de chegar enquanto um leitor está na região crítica, demorando a liberarem o semáforo *rw* para que algum escritor possa entrar.

A Figura 5.9 mostra um *script* que criamos para mostrar essa situação. Nas linhas 1-2 temos um leitor começando sua execução, sendo inserido ao conjunto de *threads FR*, indicando ser o primeiro leitor a entrar na região crítica. Antes que ele possa ler o *buffer*, criamos um bloco de eventos (linhas 3-7) que descreve um novo leitor entrando na região crítica, sendo inserido ao conjunto *TR* e executando uma iteração completa de seu código. O primeiro leitor aguarda que outras *threads* executem suas iterações para executar o evento *ReaderStarts* (linha 8). Como o bloco de repetição descreve que os outros leitores entrem e saiam da região crítica, o primeiro leitor a chegar também é o último a sair. Por isso, ele executa os eventos *LastReader* e *ReaderEnds* (linhas 9-10), encerrando o *script*.

```
1 (ReaderWantsToStart [>> fr]);
2 (FirstReader [fr]);
3 (
4   (ReaderWantsToStart [>>+ tr]);
5   (ReaderStarts [tr]);
6   (ReaderEnds [>>- tr]);
7 )+;
8 (ReaderStarts [fr]);
9 (LastReader [fr]);
10 (ReaderEnds [fr]);
```

Figura 5.9: *Script* descrevendo uma sequência que causa *starvation* nos escritores.

Executamos dez vezes o programa com vinte leitores, três escritores e tendo o *script* anterior como entrada, para simular um sistema com uma carga de leitores bem maior que a de escritores. A Figura 5.10 mostra um dos resultados que obtivemos. A média de execuções completas consecutivas dos leitores foi de 97 vezes, mostrando como um escritor precisaria esperar antes de conseguir executar em alguns casos. Caso tirássemos as últimas três linhas do *script*, o último leitor nunca tentaria executar e a sequência seria executada um número arbitrariamente grande de vezes, mostrando como o programa permite *starvation* dos escritores. Uma possível solução para impedir a *starvation* seria implementar o problema utilizando passagem de bastão (ANDREWS, 1999).

As funções das *threads* de leitores e escritores que apresentamos nas Figuras 5.1 e 5.2 contêm *loops* infinitos. Esse *loop* pode ser executado um número arbitrariamente grande de vezes caso o programa esteja correto. Podemos fazer um *script* que descreve uma sequência infinita para acompanhar essas execuções. Discutiremos mais sobre as vantagens de *scripts* com sequências infinitas na Subseção 5.2.2.

```
1 [116858880] executed ReaderWantsToStart
2 [116858880] executed FirstReader
3 [122761216] executed ReaderWantsToStart
4 [122761216] executed ReaderStarts
5 ----Reader 14 reads 0
6 [122761216] executed ReaderEnds
7 ......
8 (linha 2205)
9 ----Reader 7 reads 0
10 [119005184] executed ReaderEnds
11 [116858880] executed ReaderStarts
12 ----Reader 3 reads 0
13 [116858880] executed LastReader
14 [116858880] executed ReaderEnds
15
16 END
17
18 Finished executing the script!!
19
20 The input event sequence was accepted by your program. Please note that
       :
21 -- If the input sequence was supposed to be invalid, this means that
      your program might have some issues.
22 -- If the input sequence was supposed to be valid, this means that your
       program might be ok.
```

Figura 5.10: Resultado da execução do programa dos leitores e escritores com o *script* da Figura 5.9

A Figura 5.11 mostra um *script* que descreve uma sequência infinita do programa dos leitores e escritores. O *script* descreve uma repetição de uma ou mais vezes do bloco contido entre as linhas 1-20. Dentro desse bloco, tem uma alternativa entre o bloco de eventos que descreve a execução dos leitores (linhas 2-13) e o bloco de eventos que descreve a execução dos escritores (linhas 15-19). Como os escritores têm acesso exclusivo ao *buffer*, sua iteração completa deve passar por todos os seus eventos: *WriterWantsToStart*, *WriterStarts* e *WriterEnds*. Já o bloco dos leitores descreve o padrão do *Lightswitch*. O primeiro leitor executa os eventos *ReaderWantsToStart* e *FirstReader*, sendo inserido no conjunto de *threads* chamado *fr* (*first reader*). Em seguida, um ou mais leitores executam o bloco de eventos entre as linhas 5-9. Este bloco descreve um leitor que não é o primeiro a entrar e nem o último a sair da região crítica realizando uma iteração completa: *ReaderWantsToStart*, *ReaderStarts* e *ReaderEnds*. Depois, forçamos que o primeiro leitor a entrar seja o último a sair, realizando os eventos *ReaderStarts*, *LastReader* e *ReaderEnds*, sendo retirado do grupo de *threads fr*.

Na Figura 5.12 vemos o resultado parcial da execução do *script* da Figura 5.11 com 30 leitores e 30 escritores. Como era esperado de um *script* com sequências infinitas correto, os leitores e escritores executaram um número grande de iterações e produziram mais de 20 mil linhas de saída até encerrarmos o programa.

```
1 (
2   (
3     ReaderWantsToStart[>>fr];
4     FirstReader[fr];
5     (
6       (ReaderWantsToStart[>>+ tr]);
7       (ReaderStarts[tr]);
8       (ReaderEnds[>>- tr]);
9     )+;
10    ReaderStarts[fr];
11    LastReader[fr];
12    ReaderEnds[>>- fr];
13  );
14  |
15  (
16    WriterWantsToStart[>>tw];
17    WriterStarts[tw];
18    WriterEnds[tw];
19  );
20 )+
```

Figura 5.11: *Script* descrevendo uma sequência infinita do problema dos leitores e escritores.

```
1 [226238464] executed ReaderWantsToStart
2 [226238464] executed FirstReader
3 [232140800] executed ReaderWantsToStart
4 [232140800] executed ReaderStarts
5 ----Reader 11 reads 0
6 [232140800] executed ReaderEnds
7 [232140800] executed ReaderWantsToStart
8 [232140800] executed ReaderStarts
9 ----Reader 11 reads 0
10 ..........
11 (linha 23256)
12 ----Reader 18 reads 21
13 [235896832] executed ReaderEnds
14 [235896832] executed ReaderWantsToStart
15 [235896832] executed ReaderStarts
```

Figura 5.12: Resultado da execução do programa dos leitores e escritores com o *script* da Figura 5.11

5.1.2 Discussão: Pares de eventos

Ao longo dos testes, percebemos a necessidade de alguns pares de eventos englobando a aquisição de um semáforo. Chamamos de pares de evento pois precisamos colocá-los sempre em conjunto para conseguir um controle maior do programa do usuário.

No exemplo do problema dos leitores e escritores, os eventos *WriterWantsToStart* e *WriterStarts* são pares de eventos, assim como os eventos

ReaderWantsToStart e *ReaderStarts* também, no caso de *scripts* que ignorem o evento *FirstReader*.

Vamos mostrar a importância deles descrevendo o que ocorreria caso eles fossem substituídos por um evento só.

Caso só existissem os eventos *WriterStarts* e *ReaderStarts*, sendo inseridos após a aquisição do semáforo *rw*, a *EventManager* não conseguiria controlar estes eventos. Poderia ocorrer o caso em que temos um *script* que descreve que um escritor deve começar. Entretanto, o não determinismo do programa do usuário pode gerar casos em que o leitor adquira o semáforo primeiro, já que não há eventos antes da aquisição deste. Sendo assim, o escritor não conseguiria executar o evento *WriterStarts* e o leitor não conseguiria executar *ReaderStarts*, por conta da sequência descrita no *script*, gerando um bloqueio pela falta de controle do programa.

Caso só existissem os eventos *WriterWantsToStart* e *ReaderWantsToStart*, inseridos antes da aquisição do semáforo *rw*, também não haveria controle desta operação. Caso existisse um *script* descrevendo a sequência de eventos *WriterWantsToStart -> ReaderWantsToStart*, poderia ocorrer a seguinte situação: o escritor executa o primeiro evento mas não adquire o semáforo *rw* ainda; o leitor executa o segundo evento e adquire o semáforo *rw*. Assim, o leitor começaria a execução antes do escritor e o que estaria acontecendo no programa do usuário não corresponderia ao que o seu *script* descreve, fazendo com que o programa parecesse admitir sequências inválidas.

Como nossa ferramenta não interfere no escalonamento nem utiliza elementos específicos da linguagem para controlar o fluxo do programa, a necessidade de utilizarmos pares de eventos surgiu. Entretanto, por conta dessas mesmas características, a *EventManager* oferece portabilidade para outras linguagens.

5.2 Problema dos produtores e consumidores

O problema dos produtores e consumidores apresenta um padrão de divisão de trabalho entre as *threads*. Generalizando, o produtor cria itens de um tipo enquanto o consumidor os processa.

Um exemplo de produtor-consumidor são programas orientados a eventos como movimentos do mouse ou teclas do teclado serem apertadas (DOWNEY, 2009). Quando um evento desses ocorre (note que não estão relacionados aos eventos da ferramenta *EventManager*), um produtor cria um objeto de evento e o adiciona a uma fila. Em cenários de memória compartilhada, o produtor coloca o item em um *buffer* de onde o consumidor os retira. Quando este evento pode ser tratado, um consumidor retira o evento desta fila e o processa.

Problemas deste tipo precisam cumprir os seguintes requisitos de sincronização: Enquanto um item está sendo inserido ou retirado do *buffer*, este está em um estado inconsistente, por isso, as *threads* devem ter acesso exclusivo ao *buffer* (DOWNEY, 2009). Para garantir que não haja sobrescrita de dados e que eles sejam recebidos uma única vez, a inserção e a retirada de elementos do *buffer* deve ocorrer de forma coordenada com uma inserção sendo executada primeiro (ANDREWS, 1999).

5.2.1
Testes

Mostramos uma solução clássica do problema dos produtores e consumidores nas Figuras 5.13 e 5.14 (TANENBAUM; BOS, 2014; ANDREWS, 1999; MACHADO; MAIA, 2017; DOWNEY, 2009). Esta solução utiliza um *buffer* circular limitado no qual os produtores depositam itens e do qual os consumidores retiram itens. A primeira figura apresenta o código com o corpo das *threads* produtoras e a segunda mostra o corpo das *threads* consumidoras. Neste programa, o usuário define, por linha de comando, o número de produtores e o número de consumidores criados, além do tamanho do *buffer* utilizado. As variáveis *empty* e *full* são semáforos contadores que contabilizam, respectivamente, o número de posições vazias e o número de posições ocupadas. O semáforo *exc* controla o acesso ao *buffer*, garantindo que haja somente um consumidor ou um produtor por vez na região crítica.

```
1  sem_t * empty; //inicializado com o valor de bufferSize
2  sem_t * full; //inicializado com 0
3  sem_t * exc; //inicializado com 1
4
5  int nxtfree = 0;
6
7  void* producer (void * num) {
8    int id = *((int *) num);
9    int a;
10
11   while(1) {
12     a = rand() % 100;
13
14     checkCurrentEvent("ProducerWantsToStart");
15     sem_wait(empty);
16
17     sem_wait(exc);
18     checkCurrentEvent("ProducerStarts");
19
20     buffer[nxtfree] = a;
21
22     nxtfree = (nxtfree + 1) % bufferSize;
23
24     checkCurrentEvent("ProducerEnds");
25     sem_post(exc);
26
27     sem_post(full);
28
29     printf("----Producer %d produced item %d----\n", id, a);
30   }
31   pthread_exit(NULL);
32 }
```

Figura 5.13: Corpo das *threads* produtoras.

Inserimos a dupla de eventos *ProducerWantsToStart* e *ProducerStarts* nas linhas 14 e 18, envolvendo a aquisição dos dois semáforos necessários para que o

produtor acesse o *buffer*. Após inserir um item neste *buffer*, inserimos o evento *ProducerEnds*, na linha 24, antes que o produtor libere algum consumidor para consumir um item. Os eventos *ConsumerWantsToStart*, *ConsumerStarts* e *ConsumerEnds* funcionam de forma análoga para os consumidores.

```
1  int nxtdata = 0;
2
3  void* consumer (void * num) {
4    int id = *((int *) num);
5    int b;
6
7    while(1) {
8
9      checkCurrentEvent("ConsumerWantsToStart");
10     sem_wait(full);
11
12
13     sem_wait(exc);
14     checkCurrentEvent("ConsumerStarts");
15
16     b = buffer[nxtdata];
17
18     nxtdata = (nxtdata + 1) % bufferSize;
19
20     checkCurrentEvent("ConsumerEnds");
21     sem_post(exc);
22
23     sem_post(empty);
24
25     printf("--------Consumer %d consumed item %d--------\n", id,
         b);
26   }
27   pthread_exit(NULL);
28 }
```

Figura 5.14: Corpo das *threads* consumidoras.

Criamos cinco *scripts* para testar este problema. Em todos os testes que relatamos, utilizamos um *buffer* de tamanho 1, para facilitar o acompanhamento dos testes. Queremos testar se o programa está correto conferindo se ele cumpre os requisitos de sincronização que comentamos anteriormente: as *threads* devem ter acesso exclusivo ao *buffer* e a inserção e a retirada de elementos do *buffer* devem ocorrer de forma alternada para uma mesma entrada do *buffer*, com uma inserção sendo executada primeiro.

A Figura 5.15 mostra um *script* para testar um programa que inicie a execução com um consumidor. Nele, vemos se a sequência de eventos *ConsumerWantsToStart*, *ConsumerStarts* e *ConsumerEnds* é aceita pelo programa.

Na Figura 5.16, temos o resultado da execução do programa utilizando o *script* da Figura 5.15. Podemos observar que a sequência é inválida para o programa, já que ele não conseguiu executá-la por completo. Como era esperado, o consumidor não conseguiu adquirir o semáforo *full* e não conseguiu entrar na região crítica para executar o evento *ConsumerStarts*.

```
1 (ConsumerWantsToStart [>> consumer1]);
2 (ConsumerStarts [consumer1]);
3 (ConsumerEnds [consumer1]);
```

Figura 5.15: *Script* com a sequência dos eventos do problema do produtor e consumidor que descreve um programa iniciando a sua execução com um consumidor.

```
1 [163160064] executed ConsumerWantsToStart
2
3 END
4
5 Possible blocking detected!!
6
7 The input event sequence wasn't accepted by your program. Please note
      that:
8 -- If the input sequence was supposed to be invalid, this meas that
      your program might be ok.
9 -- If the input sequence was supposed to be valid, this means that your
       program might have some issues.
10
11 Possible Events Expected:
12    -> ConsumerStarts
```

Figura 5.16: Resultado da execução do programa de produtores e consumidores com o *script* da Figura 5.15

O segundo teste que realizamos está apresentado no *script* da Figura 5.17. O objetivo dele é testar se duas *threads* podem estar na região crítica ao mesmo tempo - no caso, se um consumidor pode acessar um *buffer* enquanto um produtor também o acessa. Para isso, a sequência exige que um consumidor execute a dupla de eventos *ConsumerWantsToStart* e *ConsumerStarts* antes que um produtor execute o evento *ProducerEnds*. Esta é um sequência que esperamos que seja inválida para o programa.

```
1 (ProducerWantsToStart [>> producer1]);
2 (ProducerStarts [producer1]);
3
4 (ConsumerWantsToStart [>> consumer1]);
5 (ConsumerStarts [consumer1]);
6
7 (ProducerEnds [producer1]);
8 (ConsumerEnds [consumer1]);
```

Figura 5.17: *Script* que descreve um produtor e um consumidor tentando acessar a região crítica ao mesmo tempo.

A Figura 5.18 mostra a execução do programa tendo o *script* da Figura

5.17 como entrada. Como esperado, o programa não conseguiu executar o evento *ConsumerStarts*, já que o consumidor não consegue entrar na região crítica enquanto o produtor não libera o semáforo *exc*.

```
1 [103534592] executed ProducerWantsToStart
2 [103534592] executed ProducerStarts
3 [104071168] executed ConsumerWantsToStart
4
5 END
6
7 Possible blocking detected!!
8
9 The input event sequence wasn't accepted by your program. Please note
      that:
10 -- If the input sequence was supposed to be invalid, this means that
      your program might be ok.
11 -- If the input sequence was supposed to be valid, this means that your
       program might have some issues.
12
13 Possible Events Expected:
14    -> ConsumerStarts
```

Figura 5.18: Resultado da execução do programa de produtores e consumidores com o *script* da Figura 5.17

Nos testes 3 e 4 queremos testar se a execução dos produtores e dos consumidores ocorre de forma alternada. Para isso, utilizamos 1 produtor e 1 consumidor e mantemos o *buffer* de tamanho 1 na execução dos testes.

Na Figura 5.19, temos o *script* com o terceiro teste. Nele, tentamos forçar duas execuções seguidas de um produtor.

```
1 (ProducerWantsToStart [>> producer1]);
2 (ProducerStarts [producer1]);
3 (ProducerEnds [producer1]);
4
5 (ProducerWantsToStart [>> producer2]);
6 (ProducerStarts [producer2]);
7 (ProducerEnds [producer2]);
```

Figura 5.19: *Script* que descreve um produtor executando duas vezes seguidas.

Como vemos na figura que mostra o resultado deste teste (Figura 5.20), o produtor não conseguiu executar o evento *ProducerStarts* em sua segunda execução, isto é, não conseguiu adquirir o semáforo *empty* e produzir um item. Como esperado, a sequência descrita no *script* da Figura 5.19 é inválida.

O quarto teste está mostrado no *script* da Figura 5.21. Nele, tentamos forçar a execução completa de um produtor seguida por duas execuções do consumidor. De forma análoga ao teste anterior, esta sequência também deve ser inválida.

A Figura 5.22 mostra o resultado da execução do programa com o *script* da Figura 5.21. Percebemos que a sequência é inválida, como desejado, pois

```
[123924480] executed ProducerWantsToStart
[123924480] executed ProducerStarts
[123924480] executed ProducerEnds
[124461056] executed ProducerWantsToStart
--------Producer 0 produced item 49--------

END

Possible blocking detected!!

The input event sequence wasn't accepted by your program. Please note that:
-- If the input sequence was supposed to be invalid, this means that your program might be ok.
-- If the input sequence was supposed to be valid, this means that your program might have some issues.

Possible Events Expected:
   -> ProducerStarts
```

Figura 5.20: Resultado da execução do programa de produtores e consumidores com o *script* da Figura 5.19

```
(ProducerWantsToStart [>> producer1]);
(ProducerStarts [producer1]);
(ProducerEnds [producer1]);

(ConsumerWantsToStart [>> consumer1]);
(ConsumerStarts [consumer1]);
(ConsumerEnds [consumer1]);

(ConsumerWantsToStart [>> consumer2]);
(ConsumerStarts [consumer2]);
(ConsumerEnds [consumer2]);
```

Figura 5.21: *Script* que descreve uma execução completa de um produtor seguida por duas execuções do consumidor.

o consumidor não conseguiu executar o segundo evento *ConsumerStarts* do *script*.

O quinto e último teste mostra um *script* contendo definições de todas as sequências possíveis no programa. Como vemos na Figura 5.23, ele força que um produtor execute primeiro (definido nas três primeiras linhas do *script*). Em seguida, força que um consumidor seja o próximo, forçando a execução de um evento *ConsumerWantsToStart*. A seguir, temos um bloco de repetição que contém uma alternativa: a primeira opção é a continuação da execução de um produtor, forçando em seguida o começo da execução de um consumidor; analogamente, a segunda opção é a continuação da execução de um consumidor, forçando em seguida o começo da execução de um produtor.

A Figura 5.24 mostra um trecho do resultado da execução do programa dos produtores e consumidores com o *script* da Figura 5.23. Neste teste,

```
[137478144] executed ProducerWantsToStart
[137478144] executed ProducerStarts
[137478144] executed ProducerEnds
--------Producer 1 produced item 49--------
[138014720] executed ConsumerWantsToStart
[138014720] executed ConsumerStarts
[138014720] executed ConsumerEnds
--------Consumer 0 consumed item 49--------
[138551296] executed ConsumerWantsToStart

END

Possible blocking detected!!

The input event sequence wasn't accepted by your program. Please note that:
-- If the input sequence was supposed to be invalid, this means that your program might be ok.
-- If the input sequence was supposed to be valid, this means that your program might have some issues.

Possible Events Expected:
   -> ConsumerStarts
```

Figura 5.22: Resultado da execução do programa de produtores e consumidores com o *script* da Figura 5.21

```
(ProducerWantsToStart [>> producers]);
(ProducerStarts [producers]);
(ProducerEnds [producers]);
(ConsumerWantsToStart [>> consumers]);
(
  (
    (ProducerStarts [producers]);
    (ProducerEnds [producers >>- producers]);
    (ConsumerWantsToStart [>>+ consumers]);
  )
  |
  (
    (ConsumerStarts [consumers]);
    (ConsumerEnds [consumers >>- consumers]);
    (ProducerWantsToStart [>>+ producers]);
  )
)+
```

Figura 5.23: *Script* de um teste com sequências infinitas de eventos do problema do produtor e consumidor, que força a execução alternada entre um produtor e um consumidor.

utilizamos 5 produtores, 5 consumidores e um *buffer* de tamanho 3. Conforme esperado, a sequência é válida e, por conter uma repetição de um bloco de eventos que englobam a execução completa das *threads* e o código dos produtores e dos consumidores estar dentro de um *while* infinito, é esperado

que a execução deste *script* também seja infinita. Em nosso teste, encerramos o programa após alguns segundos de execução, produzindo uma saída com mais de 70 mil linhas. Na próxima subseção, discutiremos as vantagens de realizar testes como esse.

```
1 [267628544] executed ProducerWantsToStart
2 [267628544] executed ProducerStarts
3 [267628544] executed ProducerEnds
4 [270311424] executed ConsumerWantsToStart
5 --------Producer 0 produced item 7--------
6 [270311424] executed ConsumerStarts
7 [270311424] executed ConsumerEnds
8 --------Consumer 0 consumed item 7--------
9 [269774848] executed ProducerWantsToStart
10 [269774848] executed ProducerStarts
11 [269774848] executed ProducerEnds
12 --------Producer 4 produced item 30--------
13 .........
14 (linha 39892)
15 --------Producer 3 produced item 18--------
16 [271384576] executed ConsumerWantsToStart
17 [271384576] executed ConsumerStarts
18 [271384576] executed ConsumerEnds
19 --------Consumer 2 consumed item 18--------
20 [269774848] executed ProducerWantsToStart
21 [26977484 ^C [encerramos o programa]
```

Figura 5.24: Resultado da execução do programa de produtores e consumidores com o *script* da Figura 5.23

Note que este teste força sempre a execução alternada de um produtor e um consumidor, mas que para um *buffer* com tamanhos maiores, a execução dos mesmos não precisa ser alternada para diferentes entradas do *buffer*. Este *script* está restringindo demais a execução das *threads* para garantir que continuem executando diversas vezes. Isso se deve a algumas limitações da linguagem de eventos e de especificação de *threads*. Caso não obrigássemos que um consumidor só possa executar após um produtor ter executado uma iteração completa, poderia acontecer o caso do consumidor executar o evento *ConsumerWantsToStart* mas não conseguir adquirir o semáforo *full*, por não ter nada para consumir no *buffer*. Assim, este consumidor não conseguiria executar o evento *ConsumerStarts* e ocorreria um bloqueio causado pela ferramenta, mesmo que o programa do usuário estivesse correto. Esse caso poderia ser corrigido caso o *script* oferecesse alguma forma de contabilizar o número de produtores/consumidores que já executaram.

Criamos um *script* menos restrito, mostrado na Figura 5.25. Este *script* resolve a questão do consumidor causar um bloqueio esperando o evento *ConsumerStarts* através do uso de alternativas. Existem quatro alternativas de blocos dentro de uma repetição (linhas 1-9). A primeira opção (linha 2) é um produtor executar o evento *ProducerWantsToStart*. Este produtor não pode ser algum presente no conjunto de *threads* chamado *prod* e será inserido neste conjunto após executar o evento. Analogamente, temos o evento *ConsumerWantsToStart* (linha 4) para um consumidor. Na linha 6, temos a continuação da execução de um produtor: ele executa a sequência de

eventos *ProducerStarts* e *ProducerEnds*, sendo retirado do conjunto *prod*. Analogamente, um consumidor continua sua execução na linha 8, executando a sequência de eventos *ConsumerStarts* e *ConsumerEnds*. Por conta das alternativas, caso um produtor fique esperando pelo semáforo *empty* ou um consumidor pelo semáforo *full*, não acontecerá um bloqueio. Entretanto, nessse *script* não conseguimos contar o número de produtores e de consumidores para coordenar as suas atividades.

```
1 (
2   (ProducerWantsToStart[~prod >>+ prod]);
3   |
4   (ConsumerWantsToStart[~cons >>+ cons]);
5   |
6   (ProducerStarts[prod]; ProducerEnds[prod >>- prod];)
7   |
8   (ConsumerStarts[cons]; ConsumerEnds[cons >>- cons];)
9 )+
```

Figura 5.25: *Script* de um teste com sequências infinitas de eventos do problema do produtor e consumidor.

Na Figura 5.26, apresentamos o resultado da execução do programa utilizando o *script* da Figura 5.25 com 100 produtores, 100 consumidores e um *buffer* de tamanho 5. Encerramos o programa após alguns segundos de execução. Podemos observar que, diferentemente do resultado da Figura 5.24, os produtores e consumidores não executam necessariamente de forma alternada, conforme o desejado ao escrevermos um *script* com sequências infinitas menos restrito.

5.2.2 Discussão: *Scripts* com sequências infinitas

Neste problema e no anterior utilizamos um *script* que definia uma sequência infinita para o programa do usuário. Percebemos a utilidade de *scripts* assim para o usuário testar se seu programa aceita uma sequência válida executando um número arbitrariamente grande de vezes.

Além disso, a saída de testes como estes pode ser utilizada como entrada para um programa que analise os resultados do programa do usuário de forma automática. No exemplo do problema dos produtores e consumidores, nossa ferramenta facilita que o usuário teste alguns aspectos semânticos de seu programa verificando as sequências aceitas ou não pelo mesmo. Entretanto, não verificamos se de fato não ocorre sobrescrita dos itens, itens inconsistentes sendo produzidos ou leituras duplicadas de itens. O usuário poderia fazer um programa que analisa a saída de nossa ferramenta e avalia se, de fato, todos os itens produzidos estão sendo consumidos na ordem correta (mesmo com diferentes sequências de produtores e consumidores, no caso de *buffers* de tamanho maior do que 1).

Criar *scripts* com sequências corretas é mais difícil do que criar *scripts* com sequências incorretas. Especialmente se for um *script* com sequências

```
[52129792] executed ProducerWantsToStart
[54812672] executed ProducerWantsToStart
[54812672] executed ProducerStarts
[54812672] executed ProducerEnds
--------Producer 5 produced item 72--------
[54812672] executed ProducerWantsToStart
[54812672] executed ProducerStarts
[54812672] executed ProducerEnds
--------Producer 5 produced item 65--------
.........
(linha 3928)
--------Consumer 48 consumed item 11--------
[131543040] executed ConsumerWantsToStart
[151396352] executed ConsumerStarts
[151396352] executed ConsumerEnds
--------Consumer 85 consumed item 89--------
[158371840] executed ConsumerWantsToStart
[75202560] executed ProducerStarts
[75202560] executed ProducerEnds
--------Producer 43 produced item 71--------
[89690112] executed ProducerStarts
[89690112] executed ProducerEnds
^C [encerramos o programa]
```

Figura 5.26: Resultado da execução do programa de produtores e consumidores com o *script* da Figura 5.25

infinitas, no qual o usuário deve considerar os vários caminhos possíveis que seu programa pode seguir: A dificuldade está no desafio de descrever uma execução contínua e cobrir em um único *script* todas as possibilidades de execuções corretas. Essa tarefa exige um conhecimento mais profundo acerca da semântica do problema.

5.3 Problema da fila de itens

O problema da fila de itens que implementamos consiste em dois tipos de *threads* acessando uma estrutura de dados que representa uma fila. Algumas *threads* são responsáveis por colocar elementos nesta fila e, por isso, são chamadas de *enqueuers*. As outras *threads* são responsáveis por retirar os itens desta fila, chamadas de *dequeuers*.

Este problema é uma outra implementação do problema dos produtores e consumidores usando variáveis de condições. Escolhemos implementar este problema após encontrá-lo sendo utilizado como exemplo da biblioteca em *Scala* do trabalho relacionado que mencionamos anteriormente (MAYER; MADHAVAN, 2016). Implementamos em C o código descrito em Scala no artigo. Como o artigo utiliza algumas primitivas de sincronização de *Scala* como *waits* e *notifyAll*, decidimos implementar nossa solução utilizando variáveis de condições da biblioteca *pthreads*, ao invés de semáforos. Em Scala, as *threads* que esperam nos *waits* não esperam por uma condição específica. Por isso, para mantermos nosso código em C compatível com o do artigo, deixamos uma condição única de fila cheia e de fica vazia.

5.3.1 Testes

As Figuras 5.27 e 5.28 apresentam nosso código para este problema: as *threads* que inserem itens na fila são chamadas de *enqueuers* enquanto que as que retiram itens da fila são as *dequeuers*. Tanto as *threads enqueuers* quanto as *dequeuers* esperam pelo *lock* chamado *waitLock* para entrar na região crítica. No caso das *enqueuers*, verificamos se a fila está cheia. Se estiver, essa *thread* espera pela condição *waitCondition* para verificar novamente se a fila está cheia. Quando existir um espaço na fila, a *thread* insere o elemento na fila e envia um sinal via *broadcast* para todas as *threads* que estiverem esperando pela condição *waitCondition*. De forma análoga, as *threads dequeuers* esperam pela condição *waitCondition* até que haja algum elemento na fila a ser retirado. Após conseguirem retirar um elemento da fila, a *thread* também envia um sinal *broadcast* para todas as que estiverem esperando por *waitCondition*.

```
1  void enqueue (int element) {
2
3    checkCurrentEvent("EnqueueWantsToStart");
4    pthread_mutex_lock(&waitLock);
5    checkCurrentEvent("EnqueueStarts");
6
7    while (isFull(mainQueue)) {
8      checkCurrentEvent("EnqueueWaits");
9      pthread_cond_wait(&waitCondition, &waitLock);
10
11     pthread_mutex_unlock(&waitLock);
12     checkCurrentEvent("EnqueueContinues");
13     pthread_mutex_lock(&waitLock);
14   }
15
16   checkCurrentEvent("EnqueueEnqueues");
17   mainQueue->buffer[tail(mainQueue)] = element;
18   (mainQueue->count) ++;
19
20   pthread_cond_broadcast(&waitCondition);
21
22   checkCurrentEvent("EnqueueEnds");
23   pthread_mutex_unlock(&waitLock);
24 }
```

Figura 5.27: Corpo das *threads* que inserem itens na fila.

Neste problema, inserimos eventos antes de cada etapa. O par de eventos *EnqueueWantsToStart* (linha 3 – 5.27) e *EnqueueStarts* (linha 5 – 5.27), assim como o par *DequeueWantsToStart* (linha 4 – 5.28) e *DequeueStarts* (linha 6 – 5.28) foram inseridas englobando a aquisição do *lock waitLock*. Inserimos tanto o evento *EnqueueWaits* (linha 8 – 5.27) quanto o *DequeueWaits* (linha 9– 5.28) antes da espera das *threads* caso a fila esteja cheia ou caso a fila esteja vazia, respectivamente. Após a espera por *waitCondition*, inserimos os eventos *EnqueueContinues* (linha 12 – 5.27) e *DequeueContinues* (linha 13 – 5.28).

```
1  int dequeue (void) {
2    int element;
3
4    checkCurrentEvent("DequeueWantsToStart");
5    pthread_mutex_lock(&waitLock);
6    checkCurrentEvent("DequeueStarts");
7
8    while (isEmpty(mainQueue)) {
9      checkCurrentEvent("DequeueWaits");
10     pthread_cond_wait(&waitCondition, &waitLock);
11
12     pthread_mutex_unlock(&waitLock);
13     checkCurrentEvent("DequeueContinues");
14     pthread_mutex_lock(&waitLock);
15   }
16
17   checkCurrentEvent("DequeueDequeues");
18   element = mainQueue->buffer[mainQueue->head];
19   mainQueue->head = (mainQueue->head + 1) % bufferSize;
20   (mainQueue->count) --;
21
22   pthread_cond_broadcast(&waitCondition);
23
24   checkCurrentEvent("DequeueEnds");
25   pthread_mutex_unlock(&waitLock);
26
27   return element;
28 }
```

Figura 5.28: Corpo das *threads* que retiram itens da fila.

Precisamos envolver esses eventos com a liberação e reaquisição de *waitLock*, algo que discutiremos na próxima subseção (Subseção 5.3.2). Inserimos o evento *EnqueueDequeues* (linha 16 – 5.27) e *DequeueEnqueues* (linha 17 – 5.28) antes das operações de retirar da fila e inserir, respectivamente. Os eventos *EnqueueEnds* (linha 22 – 5.27) e *DequeueEnds* (linha 24 – 5.28) indicam o término do corpo da *thread*.

Realizamos um teste análogo ao primeiro teste da implementação do problema dos produtores e consumidores: testar se uma *thread dequeuer* consegue retirar algo da fila sem que nenhum item tenha sido inserido. Para isso, fizemos o *script* da Figura 5.29. Nele, temos a sequência de eventos que caracterizam uma execução completa de uma *thread dequeuer* quando consegue inserir um item na fila: *DequeuerWantsToStart*, *DequeuerStarts*, *DequeuerDequeues* e *DequeuerEnds*.

A Figura 5.30 mostra o resultado da execução do nosso programa com o *script* da Figura 5.29, utilizando uma *thread enqueuer* e uma *thread dequeuer*. Como esperado para um programa correto, o *script* não foi executado por completo: o evento *DequeuerDequeues* não foi executado porque a fila estava vazia.

Como este problema é uma variante do problema dos produtores e

```
1 DequeueWantsToStart [>>thread1];
2 DequeueStarts [thread1];
3 DequeueDequeues [thread1];
4 DequeueEnds [thread1];
```

Figura 5.29: *Script* que descreve uma *thread dequeuer* sendo executada antes de alguma *thread enqueuer*.

```
1 [85446656] executed DequeueWantsToStart
2 [85446656] executed DequeueStarts
3
4 END
5
6 Possible blocking detected!!
7
8 The input event sequence wasn't accepted by your program. Please note
      that:
9 -- If the input sequence was supposed to be invalid, this means that
      your program might be ok.
10 -- If the input sequence was supposed to be valid, this means that your
       program might have some issues.
11
12 Possible Events Expected:
13    -> DequeueDequeues
```

Figura 5.30: Resultado da execução do programa da fila de itens com o *script* da Figura 5.29

consumidores, os testes que realizamos na Seção 5.2 podem ser feitos aqui de forma análoga. Por esta razão, decidimos não incluí-los no texto.

O próximo teste que realizamos tinha o objetivo de testar a semântica do problema. Este teste foi realizado pelo artigo da biblioteca em *Scala* para testá-la (MAYER; MADHAVAN, 2016). O *script* da Figura 5.31 mostra a sequência testada. Ele descreve as sequências de eventos de duas *threads enqueuers* inserindo itens (linhas 1-9), uma *thread dequeuer* retirando um item (linhas 11-14) e outras duas *threads enqueuers* inserindo mais dois itens (linhas 16-24). Assim como no artigo citado, utilizamos uma fila de tamanho três. Como a *thread dequeuer* retira um dos dois elementos inseridos primeiramente, os próximos dois elementos devem ser inseridos sem problemas. A biblioteca do artigo possibilita que o usuário teste se o item que foi retirado é igual ao primeiro item inserido. Nossa ferramenta não faz testes deste tipo mas permite que o usuário verifique isso por conta própria no *log* de teste.

A Figura 5.32 mostra o resultado da execução do programa da fila de itens com o *script* da Figura 5.31. Como podemos observar, o *script* foi executado por completo, mostrando que o programa está possivelmente correto. Além disso, o resultado mostra os *prints* do programa indicando quais itens foram inseridos e quais foram retirados. Conferindo estes *prints*, podemos ver que o primeiro item inserido foi o número 49. Em seguida, o número 7 foi inserido. O elemento retirado pela *thread dequeuer* foi o número 49, o que é esperado

```
1 EnqueueWantsToStart [>>enqThreads];
2 EnqueueStarts [enqThreads];
3 EnqueueEnqueues [enqThreads];
4 EnqueueEnds [enqThreads >>- enqThreads];
5
6 EnqueueWantsToStart [>>+ enqThreads];
7 EnqueueStarts [enqThreads];
8 EnqueueEnqueues [enqThreads];
9 EnqueueEnds [enqThreads >>- enqThreads];
10
11 DequeueWantsToStart [>>deqThreads];
12 DequeueStarts [deqThreads];
13 DequeueDequeues [deqThreads];
14 DequeueEnds [deqThreads >>- deqThreads];
15
16 EnqueueWantsToStart [>>+enqThreads];
17 EnqueueStarts [enqThreads];
18 EnqueueEnqueues [enqThreads];
19 EnqueueEnds [enqThreads >>- enqThreads];
20
21 EnqueueWantsToStart [>>+enqThreads];
22 EnqueueStarts [enqThreads];
23 EnqueueEnqueues [enqThreads];
24 EnqueueEnds [enqThreads >>- enqThreads];
```

Figura 5.31: *Script* com a sequência dos eventos que descreve um teste realizado pelo artigo de Mayer e Madhavan (MAYER; MADHAVAN, 2016).

de uma fila implementada de forma correta.

5.3.2 Discussão: Programas que utilizam condições da *pthread*

Nas Figuras 5.27 e 5.28 vemos que os eventos *EnqueuerContinues* e *DequeuerContinues* estão envolvidos pela liberação e reaquisição de um *lock*, algo que não existiria em um programa que não estivesse sendo testado pela nossa ferramenta. Precisamos inserir estas operações manualmente para garantir que os eventos funcionem. Isto se deve ao fato da *thread* estar de posse do *lock* quando a função *phthread_cond_wait* é chamada. Dentro desta função, a *thread* adquire novamente o *lock* quando volta a ser executada.

Para conseguirmos controlar o andamento das *threads* deste programa, precisamos inserir o evento *DequeueContinues* após a função *phthread_cond_wait* (e, analogamente, o evento *EnqueueContinues*), para forçar que a *thread* espere este evento antes de continuar sua execução. Caso não existisse este evento, a *thread* que estivesse em espera seria liberada após o sinal de *broadcast* de uma outra *thread* e aguardaria a aquisição do *lock waitLock*. Não teríamos como controlar a próxima *thread* a adquirir o *lock* por conta desta aquisição ser feita dentro da função *pthread_cond_wait*, criando um ponto de não determinismo no teste do usuário. Por isso, o evento *DequeueContinues* é importante para realizar um controle maior do andamento das *threads*.

```
1 [214851584] executed EnqueueWantsToStart
2 [214851584] executed EnqueueStarts
3 [214851584] executed EnqueueEnqueues
4 [214851584] executed EnqueueEnds
5 ----------Thread 1 enqueued element 49
6 [214315008] executed EnqueueWantsToStart
7 [214315008] executed EnqueueStarts
8 [214315008] executed EnqueueEnqueues
9 [214315008] executed EnqueueEnds
10 ----------Thread 0 enqueued element 7
11 [216461312] executed DequeueWantsToStart
12 [216461312] executed DequeueStarts
13 [216461312] executed DequeueDequeues
14 [216461312] executed DequeueEnds
15 ----------Thread 4 dequeued element 49
16 [215388160] executed EnqueueWantsToStart
17 [215388160] executed EnqueueStarts
18 [215388160] executed EnqueueEnqueues
19 [215388160] executed EnqueueEnds
20 ----------Thread 2 enqueued element 73
21 [215924736] executed EnqueueWantsToStart
22 [215924736] executed EnqueueStarts
23 [215924736] executed EnqueueEnqueues
24 [215924736] executed EnqueueEnds
25 ----------Thread 3 enqueued element 58
```

Figura 5.32: Resultado da execução do programa da fila de itens com o *script* da Figura 5.31

Podemos utilizar um exemplo para ilustrar isso. A Figura 5.33 mostra um *script* que força uma *thread dequeuer* a executar primeiro e se colocar em espera. Em seguida, duas *threads enqueuers* executam seus códigos, inserindo itens na fila. Por último, a *thread dequeuer* volta a executar e retira um elemento da fila. Esta é uma sequência parecida com os testes anteriores que realizamos e que deveria ser válida, ou seja, deveria ser aceita pelo programa. Testamos com uma fila de tamanho dois, duas *threads enqueuers* e uma *thread enqueuer*. O resultado foi como esperado, o programa executou o *script* por completo.

Entretanto, ao realizarmos este mesmo teste comentando as linhas de liberação e reaquisição do *lock* das *threads dequeuers* (linhas 12 e 14 da Figura 5.28), o resultado não é o mesmo. A Figura 5.34 mostra o resultado que obtivemos. Ocorreu um bloqueio causado pela ferramenta, já que não foi possível executar o evento *EnqueueStarts*. Isto se deve ao fato da *thread dequeuer* ter sido liberada de sua espera e ter readquirido o *waitLock* antes que a segunda *thread enqueuer* pudesse adquiri-lo. Assim, como a *thread dequeuer* espera pelo evento *DequeuerContinues* segurando o *waitLock*, ocorreu um bloqueio causado pela ferramenta, já que sem as esperas pelos eventos, a *thread dequeuer* poderia ser a próxima a continuar sua execução.

No caso da implementação do problema que utilizamos, os eventos *DequeueContinues* e *EnqueueContinues* poderiam ser descartados sem alterar o funcionamento dos testes. Entretanto, escolhemos mantê-los para deixar mais explícito o momento da continuação da execução de uma *thread* após a espera pela variável de condição. Além disso, mantendo este evento podemos expor a

```
1 DequeueWantsToStart [>>thread1];
2 DequeueStarts [thread1];
3 DequeueWaits [thread1];
4
5 EnqueueWantsToStart [>>thread2];
6 EnqueueStarts [thread2];
7 EnqueueEnqueues [thread2];
8 EnqueueEnds [thread2];
9
10 EnqueueWantsToStart [>>thread3];
11 EnqueueStarts [thread3];
12 EnqueueEnqueues [thread3];
13 EnqueueEnds [thread3];
14
15 DequeueContinues [thread1];
16 DequeueDequeues [thread1];
17 DequeueEnds [thread1];
```

Figura 5.33: *Script* com a sequência dos eventos do teste do código das Figuras 5.27 e 5.28 para ilustrar problemas que poderiam ocorrer pelo uso da função *pthread_cond_wait*

necessidade de liberar e readquirir o *lock* em alguns casos.

```
1 [104222720] executed DequeueWantsToStart
2 [104222720] executed DequeueStarts
3 [104222720] executed DequeueWaits
4 [103149568] executed EnqueueWantsToStart
5 [103149568] executed EnqueueStarts
6 [103149568] executed EnqueueEnqueues
7 [103149568] executed EnqueueEnds
8 [103686144] executed EnqueueWantsToStart
9 ----------Thread 0 enqueued element 7
10
11 END
12
13 Possible blocking detected!!
14
15 The input event sequence wasn't accepted by your program. Please note
       that:
16 -- If the input sequence was supposed to be invalid, this means that
       your program might be ok.
17 -- If the input sequence was supposed to be valid, this means that your
        program might have some issues.
18
19 Possible Events Expected:
20    -> EnqueueStarts
```

Figura 5.34: Resultado da execução do programa da fila de itens com o *script* da Figura 5.33

5.4 Problema da barreira

O problema da barreira descreve um padrão de sincronização básico. Ele consiste em um ponto de sincronização para as *threads* do programa. Quando uma *thread* chega neste ponto denominado barreira, ela deve esperar até que todas as outras *threads* cheguem neste ponto. Quando a última *thread* alcança a barreira, todas as *threads* são liberadas para prosseguir.

A barreira pode ser útil, por exemplo, em problemas que utilizam múltiplas *threads* para computar partes separadas de uma solução em paralelo e que precisam de uma sincronização a cada iteração (ANDREWS, 1999).

A professora Silvana Rossetto disponibilizou um código de uma implementação correta da barreira que ela utiliza em suas aulas na disciplina Computação Concorrente na UFRJ. Vamos apresentar um código com erros e o correto para mostrar como a ferramenta *EventManager* seria útil em ambos os casos.

5.4.1 Testes

O código da Figura 5.35 apresenta uma solução ingênua e errada para o problema da barreira. Uma das *threads* do programa executa a função *coordinator* e age como coordenadora das demais. As *threads* restantes executam a função *worker*. A cada chamada à função *naive_barrier* (linha 11), as *threads* trabalhadoras incrementam o semáforo *arrived* (linha 3) e, em seguida, esperam pelo semáforo *youMayGo* (linha 5). A *thread* coordenadora executa *NUMTHREADS* vezes um *sem_wait(arrived)*, para esperar que todas as *threads* trabalhadoras cheguem na barreira. Depois, executa *NUMTHREADS* vezes um *sem_post(youMayGo)*, para liberar as *threads* da barreira a cada volta.

Marcamos três eventos neste código. O evento *WorkerArrives* (linha 2) indica que uma *thread* trabalhadora chegou na barreira. Marcamos o evento *WorkerWaits* na linha 4, anterior à espera pelo semáforo *youMayGo*. Já o evento *CoordinatorPosted* indica o fim de uma iteração do coordenador.

A Figura 5.36 mostra um *script* que criamos para expor o erro do código. A sequência descrita começa com o evento *WorkerArrives* e a *thread* que o executar será adicionada ao grupo de *threads* chamado *firstWorker*. Em seguida, temos novamente o evento *WorkerArrives* mas a *thread* que executá-lo será adicionada a *secondworker*. O próximo evento é o *CoordinatorPosted*, indicando que o coordenador terminou uma iteração. Os próximos eventos descritos são executados pela primeira *thread* que chegou na barreira: *WorkerWaits*, *WorkerArrives*, *WorkerWaits* e *WorkerArrives*. Esta sequência de eventos indica que a *thread* pertencente a *firstWorker* passou pela barreira duas vezes seguidas. Portanto, é esperado que um programa correto não execute este *script* por completo, já que ele permite que uma das *threads* fique na frente das outras por uma volta.

A Figura 5.37 mostra o resultado da execução do *script* da Figura 5.36 com o código da Figura 5.35, utilizando duas *threads* trabalhadoras e uma *thread* coordenadora. O resultado mostra que o *script* foi executado por

```
1 void naive_barrier (void) {
2   checkCurrentEvent("WorkerArrives");
3   sem_post(arrived);
4   checkCurrentEvent("WorkerWaits");
5   sem_wait(youMayGo);
6 }
7
8 void* worker (void* id) {
9   int i;
10  for (i=0;i<ITER;i++) {
11    naive_barrier();
12    printf("-------------%s finished iteration %i \n", (char *)
      id, i);
13  }
14  return NULL;
15 }
16
17 void* coordinator (void* id) {
18  int i, j;
19  for (i=0;i<ITER;i++) {
20    for (j=0;j<NUMTHREADS;j++)
21      sem_wait(arrived);
22    for (j=0;j<NUMTHREADS;j++)
23      sem_post(youMayGo);
24    checkCurrentEvent("CoordinatorPosted");
25  }
26  return NULL;
27 }
```

Figura 5.35: Implementação ingênua do problema da barreira com erros.

```
1 (WorkerArrives [>> firstWorker]);
2 (WorkerArrives [>> secondWorker]);
3 (CoordinatorPosted [coordinator]);
4 (WorkerWaits [firstWorker]);
5 (WorkerArrives [firstWorker]);
6 (WorkerWaits [firstWorker]);
7 (WorkerArrives [firstWorker]);
```

Figura 5.36: *Script* com a ordem dos eventos do código da Figura 5.35

completo, isto é, o programa aceitou uma sequência inválida. Vemos nas linhas 5 e 8 que a *thread* 2 executou duas iterações seguidas, sem que a *thread* 1 executasse alguma. Este resultado expõe o erro da implementação ingênua do problema da barreira: a *thread* coordenadora incrementa em dois o semáforo *youMayGo* (porque são duas *threads* trabalhadoras). Uma *thread* passa pela barreira decrementando o semáforo *youMayGo* e em sua próxima iteração passa novamente pela barreira antes que a outra *thread* consiga decrementar o semáforo.

A Figura 5.38 mostra o código de uma implementação correta da barreira.

```
1 [101060608] executed WorkerArrives
2 [100524032] executed WorkerArrives
3 -------------coordinator
4 [101597184] executed CoordinatorPosted
5 [101060608] executed WorkerWaits
6 -------------two finished iteration 0
7 [101060608] executed WorkerArrives
8 [101060608] executed WorkerWaits
9 -------------two finished iteration 1
10 [101060608] executed WorkerArrives
11 -------------coordinator
12
13 END
14
15 Finished executing the script!!
16
17 The input event sequence was accepted by your program. Please note that
       :
18 -- If the input sequence was supposed to be invalid, this means that
       your program might have some issues.
19 -- If the input sequence was supposed to be valid, this means that your
        program might be ok.
```

Figura 5.37: Resultado da execução do teste do código da Figura 5.35 com o *script* da Figura 5.36

Nesta implementação, todas as *threads* do programa executam a função *barrier*. A última que chega na barreira a cada iteração funciona como a *thread* coordenadora. A variável *mutex* é um semáforo para proteger a região crítica de acesso à variável *arrived*, que armazena o número de *threads* que chegaram na barreira. A *thread* entra na condição da linha 4 do código caso não tenha sido a última a chegar. Ali, ela libera o semáforo *mutex* e espera pela liberação do semáforo *allArrived*, que representa a barreira. A última *thread* a chegar não cumpre a condição da linha 4, então entra no bloco das linhas 16-20. Ela libera uma *thread* que esteja esperando por *allArrived* e decrementa 1 da variável *arrived*. Isto é, ela libera alguma *thread* para passar pela barreira. Note que ela não liberou o semáforo *mutex*. Alguma das *threads* que estivesse esperando por *allArrived* na linha 8 é liberada e, caso não seja a última a ser liberada, executa um *sem_post* para liberar a próxima *thread*. A última não precisa liberar nenhuma *thread*, somente o semáforo *mutex*. Por conta da liberação de uma *thread* de cada vez, não existe o risco de corrida de dados na variável *arrived*, ocorrendo uma passagem de bastão entre as *threads*. Utilizamos os mesmos eventos do código anterior para conseguir utilizar o mesmo *script* e conseguir comparar os testes.

Executando o código da implementação correta (Figura 5.38) com o mesmo *script* (Figura 5.36) que utilizamos na implementação errada, obtemos o resultado mostrado na Figura 5.39. Como é possível observar, o *script* não foi executado de forma completa, como esperávamos de um programa correto. O programa não conseguiu que a *thread* pertencente a *firstWorker* executasse o evento *WorkerArrives* antes que a segunda *thread* passasse pela barreira.

Criamos um *script* descrevendo sequências infinitas para o problema da barreira, conforme apresentado na Figura 5.40. O bloco entre as linhas 1-3

```
1 void barrier (int numThreads) {
2   sem_wait(mutex);
3   arrived++;
4   if (arrived < numThreads) {
5     checkCurrentEvent("WorkerArrives");
6     sem_post(mutex);
7     checkCurrentEvent("WorkerWaits");
8     sem_wait(allArrived);
9     arrived--;
10    if (arrived==0) {
11      sem_post(mutex);
12    }
13    else {
14      sem_post(allArrived);
15    }
16  } else {
17    arrived--;
18    sem_post(allArrived);
19    checkCurrentEvent("CoordinatorPosted");
20  }
21 }
```

Figura 5.38: Implementação correta da barreira disponibilizada pela professora Silvana Rossetto.

```
1 [255627264] executed WorkerArrives
2 [256163840] executed WorkerArrives
3 [256700416] executed CoordinatorPosted
4 [255627264] executed WorkerWaits
5
6 END
7
8 Possible blocking detected!!
9
10 The input event sequence wasn't accepted by your program. Please note
       that:
11 -- If the input sequence was supposed to be invalid, this means that
       your program might be ok.
12 -- If the input sequence was supposed to be valid, this means that your
        program might have some issues.
13
14 Possible Events Expected:
15    -> WorkerArrives
```

Figura 5.39: Resultado da execução do teste do código da Figura 5.38 com o *script* da Figura 5.36

descreve as *threads* que não são a coordenadora entrando na barreira e sendo adicionadas ao grupo de *threads workers1*. A seguir, temos a repetição do bloco das linhas 4-17. Este bloco começa com a *thread* coordenadora da rodada (a última que chegou na barreira) indicando pelo evento *CoordinatorPosted* que liberou as outras a passarem pela barreira (linha 5). As *threads* que chegam na barreira executam o evento *WorkerWaits* (linha 9) e são retiradas do

conjunto de *threads workers1*. Depois, as *threads* que chegarem na próxima volta e não pertencerem ao conjunto *workers1* (ou seja, devem ter executado anteriormente o evento *WorkerWaits* ou ser o coordenador da volta anterior) executam o evento *WorkerArrives*, indicando que estão entrando em outra volta (linha 7). As *threads* que executarem este evento serão adicionadas ao conjunto *workers2*. O bloco contido nas linhas 11-16 é análogo ao bloco das linhas 5-10, sendo as *threads* retiradas do conjunto *workers2* e inseridas em *workers1*. Usamos os conjuntos *workers1* e *workers2* em *round robin* para monitorar quais *threads* estão em cada volta.

Usar uma alternativa entre as linhas 7 e 9 (e, analogamente, linhas 13 e 15) ao invés de escrever algo como:

```
WorkerWaits[workers1 >>- workers1]+;
WorkerArrives[~workers1 >>+ workers2]+;
```

foi necessário para evitar que alguma *thread* chegasse no evento *WorkerArrives* antes das outras terminarem a etapa anterior e impedisse que *threads* mais lentas conseguissem executar o evento *WorkerWaits*.

```
1  (
2    WorkerArrives[>>+ workers1];
3  )+;
4  (
5    (CoordinatorPosted[>>coordinator]);
6    (
7      (WorkerArrives[((~workers1)) >>+ workers2]);
8      |
9      (WorkerWaits[workers1 >>- workers1]);
10   )+;
11   (CoordinatorPosted[>>coordinator]);
12   (
13     (WorkerArrives[((~workers2)) >>+ workers1]);
14     |
15     (WorkerWaits[workers2 >>- workers2]);
16   )+;
17 )+
```

Figura 5.40: *Script* descrevendo sequências infinitas do problema da barreira.

A Figura 5.41 mostra o resultado da execução do *script* da Figura 5.40 com o código da Figura 5.38. Podemos observar que o programa correto executou o *script* com sequências infinitas sem problemas. Com alguns segundos de execução, o programa produziu uma saída com mais de 11 mil linhas e as *threads* realizaram 1316 voltas.

5.4.2 Discussão: Mesmo *script* de teste

Neste problema, utilizamos um mesmo *script* de teste em dois programas diferentes. Para isso, inserimos os mesmos eventos em cada um dos programas

```
1 [194338816] executed WorkerArrives
2 [193802240] executed WorkerArrives
3 [194875392] executed WorkerArrives
4 [195411968] executed WorkerArrives
5 [195948544] executed CoordinatorPosted
6 [195411968] executed WorkerWaits
7 [194338816] executed WorkerWaits
8 [194875392] executed WorkerWaits
9 [193802240] executed WorkerWaits
10 [195948544] executed WorkerArrives
11 [195411968] executed WorkerArrives
12 .........
13 (linha 11842)
14 [195948544] executed WorkerWaits
15 [193802240] executed WorkerWaits
16 ^C [encerramos o programa]
```

Figura 5.41: Resultado da execução do teste do código da Figura 5.38 com o *script* da Figura 5.40

para que eles se adequassem a um mesmo *script*. Isto é muito útil para comparar diferentes programas entre si e mostrar suas diferenças e seus erros.

Didaticamente, um professor poderia utilizar um mesmo *script* para testar todos os programas de seus alunos, facilitando a correção e o entendimento do aluno acerca do seu programa.

5.5 Problema do jantar dos filósofos

O problema do jantar dos filósofos foi proposto por Dijkstra em 1965. Ele representa problemas nos quais cada processo requer acesso simultâneo a mais de um recurso (ANDREWS, 1999), com sobreposições entre os conjuntos de recursos requisitados.

O problema descreve a situação de cinco filósofos sentados ao redor de uma mesa circular com um garfo entre cada um deles. No centro da mesa, existe uma tigela de espaguete que os filósofos compartilham. Cada filósofo só pode realizar uma ação de cada vez: comer ou pensar. O tempo de cada uma dessas ações varia. Para comer o espaguete, cada um dos filósofos precisa dos dois garfos adjacentes a ele. Estes garfos representam os recursos que as *threads* precisam manter em exclusividade.

Neste problema precisamos garantir quatro requisitos de sincronização: Somente um filósofo pode segurar determinado garfo a cada vez; *Deadlocks* não podem ser possíveis de ocorrer; Não pode acontecer *starvation* de nenhum filósofo; Deve ser possível que mais de um filósofo consiga comer de cada vez (DOWNEY, 2009).

A Figura 5.42 mostra o pseudo código das ações executadas por cada filósofo (DOWNEY, 2009). Nas ações de adquirir e liberar os garfos, cada filósofo pega e solta um garfo de cada vez.

Programadores iniciantes poderiam pensar em uma solução simples como fazer cada filósofo pegar/soltar sempre o garfo da direita primeiro e em seguida o da esquerda. Porém, poderia ocorrer a situação na qual todos os filósofos

```
1 while True:
2     think()
3     get_forks()
4     eat()
5     put_forks()
```

Figura 5.42: Pseudo código das ações executadas por um filósofo (DOWNEY, 2009).

pegam o garfo da direita e nenhum consegue pegar o garfo da esquerda por conta da mesa ser circular. Como nenhum filósofo conseguiria comer, nenhum garfo seria liberado, causando um *deadlock*.

5.5.1 Testes

Uma das possíveis soluções para este problema é torná-lo assimétrico: podemos fazer com que um dos filósofos seja canhoto, ou seja, adquira primeiro o garfo da esquerda, enquanto que os outros filósofos adquirem primeiro o garfo da direita. A Figura 5.43 mostra o corpo da *thread* correspondente a cada filósofo e a Figura 5.44 contém as implementações de cada ação executada por um filósofo.

Na nossa implementação, os garfos são representados por um vetor de *mutex*. Sendo assim, adquirir um garfo seria igual a adquirir o *mutex*. De forma análoga, uma *thread* que estiver esperando determinado garfo para prosseguir está esperando que o *mutex* correspondente a este garfo seja liberado. O filósofo canhoto é o que possui o *id* igual a zero. Sendo assim, ele adquire e libera primeiro o garfo correspondente à posição anterior da sua no vetor de *mutex*. Enquanto isso, os filósofos destros adquirem primeiro o garfo correspondente a sua posição no vetor de *mutex*.

Neste problema, marcamos diversos eventos da *EventManager*. Na linha 5 da Figura 5.43, temos uma chamada à função *checkCurrentEventWithId*. Esta função marca um evento com o nome passado concatenado ao *id* da *thread* passado como parâmetro. Assim, cada *thread* terá um evento diferente. Ao longo dos testes, explicaremos a necessidade de utilizar essa função.

Na linha 10, o filósofo canhoto adquire o primeiro garfo. Por isso, englobamos a chamada a essa função com o par de eventos *LeftieWantsFork1* e *LeftieGotFork1*. Na linha 14, o filósofo canhoto adquire o segundo garfo. Essa função está entre o par de eventos *LeftieWantsFork2* e *LeftieGotFork2*. De forma análoga, os filósofos destros possuem os eventos *RightieWantsFork1*, *RightieGotFork1*, *RightieWantsFork2* e *RightieGotFork2*.

Antes da chamada à função *eat*, que representa quando um filósofo está comendo, marcamos o evento *Eating*, na linha 27.

Para soltar os garfos, marcamos eventos semelhantes aos que inserimos na aquisição. Para soltar o primeiro garfo, o filósofo canhoto deve executar o par de eventos *LeftieGivesFork1* e *LeftiePutFork1*, enquanto que os filósofos destros executam o par *RightieGivesFork1* e *RightiePutFork1*. De forma aná-

```
void* philosopherFunction (void * num) {
  int id = *((int *) num);

  while(1) {
    checkCurrentEventWithId ("Starting", id + 1);
    think(id);

    if(id == 0) {
      checkCurrentEvent("LeftieWantsFork1");
      getFork(id, nThreads - 1);
      checkCurrentEvent("LeftieGotFork1");

      checkCurrentEvent("LeftieWantsFork2");
      getFork(id, 0);
      checkCurrentEvent("LeftieGotFork2");
    }
    else {
      checkCurrentEvent("RightieWantsFork1");
      getFork(id, id);
      checkCurrentEvent("RightieGotFork1");

      checkCurrentEvent("RightieWantsFork2");
      getFork(id, id - 1);
      checkCurrentEvent("RightieGotFork2");
    }

    checkCurrentEvent("Eating");
    eat(id);

    if(id == 0) {
      checkCurrentEvent("LeftieGivesFork1");
      putFork(id, nThreads - 1);
      checkCurrentEvent("LeftiePutFork1");

      checkCurrentEvent("LeftieGivesFork2");
      putFork(id, 0);
      checkCurrentEvent("LeftiePutFork2");
    }
    else {
      checkCurrentEvent("RightieGivesFork1");
      putFork(id, id);
      checkCurrentEvent("RightiePutFork1");

      checkCurrentEvent("RightieGivesFork2");
      putFork(id, id - 1);
      checkCurrentEvent("RightiePutFork2");
    }
  }
  pthread_exit(NULL);
}
```

Figura 5.43: Corpo da *thread* correspondente a cada filósofo.

```
1 void eat (int philosopher) {
2   printf("--------------Philosopher %d eating--------------\n",
      philosopher + 1);
3   sleep(1);
4 }
5
6 void think (int philosopher) {
7   printf("--------------Philosopher %d thinking--------------\n"
      , philosopher + 1);
8   sleep(1);
9 }
10
11 void getFork (int philosopher, int fork) {
12
13   sem_wait(mutexArray[fork]);
14
15   printf("--------------Philosopher %d got fork %d
      --------------\n", philosopher + 1, fork);
16 }
17
18 void putFork (int philosopher, int fork) {
19
20   sem_post(mutexArray[fork]);
21
22   printf("--------------Philosopher %d put fork %d
      --------------\n", philosopher + 1, fork);
23 }
```

Figura 5.44: Funções com as implementações de cada ação executada pelos filósofos.

loga, temos os eventos *LeftieGivesFork2*, *LeftiePutFork2*, *RightieGivesFork2* e *RightiePutFork2*.

Para testar se esta solução cumpre os requisitos de sincronização que apresentamos, fizemos três testes.

O primeiro teste, cujo *script* é mostrado na Figura 5.45, descreve cada filósofo adquirindo o primeiro garfo. Caso todos os filósofos fossem destros, iriam adquirir primeiramente o garfo da direita e nenhum filósofo conseguiria adquirir o garfo da esquerda, ocasionando em um *deadlock*. Como na solução que apresentamos o primeiro filósofo é canhoto, o *script* descreve nas linhas 1 e 3 que ele deve adquirir seu primeiro garfo. Em seguida, o *script* descreve nas linhas 5-8 que todos os outros filósofos devem executar o evento *RightieWantsFork1*. Por último, todos os filósofos destros devem adquirir seu primeiro garfo.

A Figura 5.46 mostra o resultado da execução do programa com o *script* da Figura 5.45. Nesta execução utilizamos um arquivo de configuração que ignora os eventos *Starting1*, *Starting2*, *Starting3*, *Starting4* e *Starting5*. Como podemos ver no resultado, o último filósofo não conseguiu adquirir seu primeiro garfo. Isso é o esperado de um programa correto, já que isso indica que a situação na qual todos seguram um garfo causando um *deadlock* não pode acontecer. Como a mesa é circular, o filósofo canhoto adquiriu primeiro o garfo

```
1 (LeftieWantsFork1 [>> philosopherLeftie]);
2
3 (LeftieGotFork1 [philosopherLeftie]);
4
5 (RightieWantsFork1 [>> philosopherRightie]);
6 (RightieWantsFork1 [>>+ philosopherRightie]);
7 (RightieWantsFork1 [>>+ philosopherRightie]);
8 (RightieWantsFork1 [>>+ philosopherRightie]);
9
10 (RightieGotFork1 [philosopherRightie >>- philosopherRightie]);
11 (RightieGotFork1 [philosopherRightie >>- philosopherRightie]);
12 (RightieGotFork1 [philosopherRightie >>- philosopherRightie]);
13 (RightieGotFork1 [philosopherRightie >>- philosopherRightie]);
```

Figura 5.45: *Script* com a sequência dos eventos do problema do jantar dos filósofos que descreve o filósofo canhoto adquirindo um garfo e os outros quatro filósofos adquirindo um garfo

a sua esquerda. O último filósofo não consegue segurar seu primeiro garfo pois é o mesmo que o filósofo canhoto está segurando. Assim, sempre existe pelo menos um filósofo que conseguirá realizar a ação de comer, liberando seus garfos para os outros filósofos.

O segundo *script* descreve uma sequência na qual dois filósofos comem juntos. Este *script* é apresentado na Figura 5.47. Filósofos adjacentes não conseguem comer ao mesmo tempo pois compartilham o mesmo garfo. Sendo assim, precisamos testar este requisito com filósofos não adjacentes. Usamos os eventos iniciados com o nome *Starting* para distinguir as *threads* enumerando suas posições: o filósofo que executar o evento *Starting1* é o que senta na posição 1 da mesa e tem sua *thread* adicionada ao grupo de *threads* chamado *philosopher1*. A função *checkCurrentEventWithId* permite que cada *thread* gere o evento correspondente. No *script* em questão, descrevemos o caso do filósofo sentado na posição 1 e o na posição 3 comendo juntos. Para isso, fizemos o primeiro filósofo adquirir seus dois garfos (linhas 7-11) e, analogamente, o terceiro adquirir seus dois garfos (linhas 13-17). Em seguida, os dois devem executar o evento *Eating* antes de liberarem seus garfos.

Na Figura 5.48 vemos o resultado da execução do programa com o *script* da Figura 5.45. Como era esperado de um programa possivelmente correto, o *script* foi executado por completo.

O teste descrito pelo *script* apresentado na Figura 5.49 contém o caso (inválido) de dois filósofos segurarem o mesmo garfo ao mesmo tempo. Para mostrar de forma mais clara esta situação, descrevemos dois filósofos adjacentes adquirindo o mesmo garfo. Assim, utilizamos os eventos começados com *Starting* da mesma forma que no teste anterior. Nas linhas 7-8, temos o filósofo destro sentado na posição 2 adquirindo seu primeiro garfo. Analogamente, nas linhas 10-11 temos o filósofo destro da posição 3 adquirindo seu primeiro garfo. Depois, o filósofo da posição 3 adquire seu segundo garfo seguido pelo filósofo da posição 2, também adquirindo seu segundo garfo. Os dois, então, executam o evento *Eating*.

```
1 Starting2 --> Ignored
2 --------------Philosopher 2 thinking--------------
3 Starting3 --> Ignored
4 --------------Philosopher 3 thinking--------------
5 Starting1 --> Ignored
6 Starting4 --> Ignored
7 --------------Philosopher 1 thinking--------------
8 Starting5 --> Ignored
9 --------------Philosopher 5 thinking--------------
10 --------------Philosopher 4 thinking--------------
11 [32198656] executed LeftieWantsFork1
12 --------------Philosopher 1 got fork 4--------------
13 [32198656] executed LeftieGotFork1
14 [33271808] executed RightieWantsFork1
15 --------------Philosopher 3 got fork 2--------------
16 [32735232] executed RightieWantsFork1
17 --------------Philosopher 2 got fork 1--------------
18 [34344960] executed RightieWantsFork1
19 [33808384] executed RightieWantsFork1
20 --------------Philosopher 4 got fork 3--------------
21 [32735232] executed RightieGotFork1
22 [33808384] executed RightieGotFork1
23 [33271808] executed RightieGotFork1
24
25 END
26
27 Possible blocking detected!!
28
29 The input event sequence wasn't accepted by your program. Please note
       that:
30 -- If the input sequence was supposed to be invalid, this means that
       your program might be ok.
31 -- If the input sequence was supposed to be valid, this means that your
        program might have some issues.
32
33 Possible Events Expected:
34    -> RightieGotFork1
```

Figura 5.46: Resultado da execução do programa do jantar dos filósofos com o *script* da Figura 5.45

A Figura 5.50 mostra o resultado da execução do programa com o *script* da Figura 5.49. Como podemos observar, o primeiro evento *RightieGotFork2* não foi executado. Como esta era uma sequência inválida, um programa possivelmente correto não deveria aceitá-la. Sendo assim, o resultado foi de acordo com o esperado.

O último requisito de sincronização deste problema é garantir que nenhum filósofo sofra de *starvation*. Isto é algo que não conseguimos mostrar com a *EventManager*. Mesmo que façamos um *script* que descreve uma sequência infinita e analisemos seu *output* para verificar se algum filósofo executa menos vezes que os demais, não conseguimos garantir que isto caracteriza ou descarta a ocorrência de *starvation*. Para ter essa garantia, seria necessário analisar todos os possíveis escalonamentos do programa, algo inviável para um grande número de execuções de cada filósofo.

Entretanto, podemos argumentar que esta solução não permite *deadlocks* e, como a ação de comer ocorre em um tempo finito, o filósofo que estivesse

```
1 (Starting1 [>> philosopher1]);
2 (Starting2 [>> philosopher2]);
3 (Starting3 [>> philosopher3]);
4 (Starting4 [>> philosopher4]);
5 (Starting5 [>> philosopher5]);
6
7 (LeftieWantsFork1 [philosopher1]);
8 (LeftieGotFork1 [philosopher1]);
9
10 (LeftieWantsFork2 [philosopher1]);
11 (LeftieGotFork2 [philosopher1]);
12
13 (RightieWantsFork1 [philosopher3]);
14 (RightieGotFork1 [philosopher3]);
15
16 (RightieWantsFork2 [philosopher3]);
17 (RightieGotFork2 [philosopher3]);
18
19 (Eating [philosopher1]);
20 (Eating [philosopher3]);
```

Figura 5.47: *Script* com a sequência dos eventos do problema jantar dos filósofos que descreve dois filósofos comendo juntos.

esperando um garfo conseguiria adquirí-lo assim que um vizinho o liberasse após comer (DOWNEY, 2009).

5.5.2 Discussão: Testes com *threads* específicas

Para testar os requisitos semânticos do problema do jantar dos filósofos, precisamos testar *threads* específicas. Em alguns casos, queríamos a garantia de testar *threads* correspondentes a filósofos adjacentes. Em outros, queríamos que os filósofos não fossem adjacentes.

Caso todos os filósofos possuíssem o mesmo evento *Starting*, o primeiro filósofo a chegar neste evento teria sua *thread* adicionada ao grupo *philosopher1*, sendo chamado de filósofo 1. Entretanto, o primeiro filósofo a chegar poderia não ser o filósofo sentado na posição 1 da mesa circular. Assim, não teríamos como garantir que os testes estavam sendo realizados com filósofos adjacentes ou não adjacentes.

Desta forma, criamos a função *checkCurrentEventWithId*, que recebe dois parâmetros: um nome e um número. Este número é adicionado como sufixo ao nome passado. Sendo assim, cada *thread* terá um evento único caso o número passado seja único.

```
1 [69517312] executed Starting1
2 --------------Philosopher 1 thinking--------------
3 [70053888] executed Starting2
4 --------------Philosopher 2 thinking--------------
5 [70590464] executed Starting3
6 --------------Philosopher 3 thinking--------------
7 [71127040] executed Starting4
8 --------------Philosopher 4 thinking--------------
9 [71663616] executed Starting5
10 --------------Philosopher 5 thinking--------------
11 [69517312] executed LeftieWantsFork1
12 --------------Philosopher 1 got fork 4--------------
13 [69517312] executed LeftieGotFork1
14 [69517312] executed LeftieWantsFork2
15 --------------Philosopher 1 got fork 0--------------
16 [69517312] executed LeftieGotFork2
17 [70590464] executed RightieWantsFork1
18 --------------Philosopher 3 got fork 2--------------
19 [70590464] executed RightieGotFork1
20 [70590464] executed RightieWantsFork2
21 --------------Philosopher 3 got fork 1--------------
22 [70590464] executed RightieGotFork2
23 [69517312] executed Eating
24 --------------Philosopher 1 eating--------------
25 [70590464] executed Eating
26 --------------Philosopher 3 eating--------------
27
28
29 END
30
31 Finished executing the script!!
32
33 The input event sequence was accepted by your program. Please note that
       :
34 -- If the input sequence was supposed to be invalid, this means that
       your program might have some issues.
35 -- If the input sequence was supposed to be valid, this means that your
        program might be ok.
```

Figura 5.48: Resultado da execução do programa do jantar dos filósofos com o *script* da Figura 5.47

```
(Starting1 [>> philosopher1]);
(Starting2 [>> philosopher2]);
(Starting3 [>> philosopher3]);
(Starting4 [>> philosopher4]);
(Starting5 [>> philosopher5]);

(RightieWantsFork1 [philosopher2]);
(RightieGotFork1 [philosopher2]);

(RightieWantsFork1 [philosopher3]);
(RightieGotFork1 [philosopher3]);

(RightieWantsFork2 [philosopher3]);
(RightieGotFork2 [philosopher3]);

(RightieWantsFork2 [philosopher2]);
(RightieGotFork2 [philosopher2]);

(Eating [philosopher2]);
(Eating [philosopher3]);
```

Figura 5.49: *Script* com a sequência dos eventos do problema do jantar dos filósofos que descreve dois filósofos segurando o mesmo garfo ao mesmo tempo.

```
1 [114720768] executed Starting1
2 --------------Philosopher 1 thinking--------------
3 [115257344] executed Starting2
4 --------------Philosopher 2 thinking--------------
5 [115793920] executed Starting3
6 --------------Philosopher 3 thinking--------------
7 [116330496] executed Starting4
8 --------------Philosopher 4 thinking--------------
9 [116867072] executed Starting5
10 --------------Philosopher 5 thinking--------------
11 [115257344] executed RightieWantsFork1
12 --------------Philosopher 2 got fork 1--------------
13 [115257344] executed RightieGotFork1
14 [115793920] executed RightieWantsFork1
15 --------------Philosopher 3 got fork 2--------------
16 [115793920] executed RightieGotFork1
17 [115793920] executed RightieWantsFork2
18
19 END
20
21 Possible blocking detected!!
22
23 The input event sequence wasn't accepted by your program. Please note
       that:
24 -- If the input sequence was supposed to be invalid, this means that
       your program might be ok.
25 -- If the input sequence was supposed to be valid, this means that your
        program might have some issues.
26
27 Possible Events Expected:
28    -> RightieGotFork2
```

Figura 5.50: Resultado da execução do programa do jantar dos filósofos com o *script* da Figura 5.49

5.6 Discussão Geral

Os testes com estes diversos problemas de concorrência permitiram a realização de melhorias na *EventManager*.

Percebemos as diversas vantagens de seu uso no ensino da programação concorrente, mostrando como o aluno pode instrumentar seu código e entender mais acerca da semântica do problema passado. Além disso, a ferramenta também pode ser usada para auxiliar o professor em suas correções de trabalhos e demonstrações de caminhos que um programa concorrente pode seguir.

Como mostramos nos exemplos, *scripts* com sequências válidas são mais difíceis de elaborar, principalmente se ele descrever todas as possíveis sequências corretas daqueles eventos. Desta maneira, nos *scripts* corretos usamos mais características das linguagens para expressar essas sequências do que nos incorretos.

As limitações da ferramenta se devem, principalmente, ao fato de que não interferimos no funcionamento interno da linguagem de programação C e nos escalonamentos feitos pelo sistema operacional. Por isso, só conseguimos testar até certa granularidade, não podendo interferir nas etapas da operação de incremento de uma variável, por exemplo. Além disso, precisamos criar alguns artifícios como os pares de eventos (para garantir que uma operação seja executada por completo na ordem dos eventos criados pelo usuário) e a necessidade de liberar *locks* e readquirí-los por conta de algum evento (por conta do funcionamento da função *pthread_cond_wait*, por exemplo).

As limitações que encontramos na linguagem de eventos podem ser exploradas em trabalhos futuros. Entre extensões possíveis, temos a implementação de *loops* finitos, nos quais o usuário especifica a quantidade de vezes que ele deve ser repetido utilizando uma variável ou um número estático. Também podemos implementar uma notação para informar que um evento pode ou não acontecer. Outra melhoria seria permitir que um evento ou bloco de eventos possa acontecer 0 ou mais vezes.

A limitações da sublinguagem de *threads* são as que mais nos afetaram. Portanto, seriam as mais interessantes de serem estudadas em trabalhos futuros. Ao longo dos testes, sentimos a necessidade de poder fazer comparações com um conjunto vazio; poder criar variáveis para checar quantas *threads* entraram e quantas saíram de um *loop* de eventos; fazer mais de uma operação de *update* dos conjuntos e poder ter especificações de *threads* sem um evento associado.

Ao longo de nossos testes não sentimos necessidade em escrever especificações de *threads* contendo os operadores de união, diferença e complemento dos conjuntos. Portanto, precisamos explorar se existem exemplos nos quais sua utilização é essencial.

Todas essas limitações podem ser estudadas em trabalhos futuros e, se implementadas, levariam a mais questões e outras possíveis melhorias da ferramenta *EventManager*.

6
Conclusão e trabalhos futuros

Programas concorrentes são difíceis de serem testados por serem não determinísticos. Vimos algumas ferramentas que se propõe a testá-los mas que possuem soluções específicas para algum ambiente de programação. Pensando em criar uma ferramenta simples de ser usada e que apresente portabilidade para outras linguagens de programação, desenvolvemos a *EventManager*.

A *EventManager* permite que o usuário verifique a semântica de seu programa, sendo, por isso, útil para ambientes de aprendizado de programação concorrente. Os alunos precisam entender bem a lógica do problema para poderem instrumentar seus códigos corretamente para o uso da *EventManager*, criando eventos nos pontos certos que necessitam de sincronização. O professor pode fazer uso da ferramenta em suas explicações para demonstrar que algumas sequências inválidas cujo acontecimento é raro podem de fato acontecer. Ele também pode utilizar a ferramenta para auxiliar correções de trabalho, submentendo diversos programas a um mesmo conjunto de *scripts* de testes.

A *EventManager* funciona ordenando os eventos inseridos ao longo do programa seguindo as possíveis sequências descritas pelo *script* passado a cada teste. No início do trabalho, em uma primeira versão da *EventManager*, criamos testes baseados em sequências simples de eventos. Entretanto, percebemos que precisávamos de descrições mais expressivas com alternativas e repetições para possibilitar a criação de alguns testes. Nos inspiramos em alguns trabalhos anteriores para criar uma *DSL*. Porém, ao invés de definir uma linguagem de forma *ad hoc*, optamos por investigar o uso de conceitos bem conhecidos: as expressões regulares e linguagem de conjuntos. Explorando as expressões regulares, criamos a linguagem de eventos. Com ela, conseguimos especificar as possíveis ordens dos eventos contendo sequências, alternativas e repetições. Usando linguagem de conjuntos, criamos a linguagem de especificação de *threads*, que nos permite restringir as *threads* que podem executar cada evento.

Para testar a expressividade das linguagens que criamos, utilizamos a *EventManager* em soluções de problemas clássicos de concorrência. Estes problemas podem aparecer em diversas aplicações reais, além de serem utilizados em sala de aula para o ensino da programação concorrente. Os testes que criamos mostraram a necessidade de algumas funcionalidades a mais do que as presentes no escopo original da ferramenta. Implementamos grande parte destas funcionalidades e descobrimos novas limitações a partir daí. Algumas limitações são resultado de características intrínsecas da ferramenta, como a granularidade dos testes e a necessidade da utilização de pares de eventos.

Nossa discussão acerca da expressividade da *EventManager* abre espaço para diversos trabalhos futuros. Grande parte envolve estudar a viabilidade e necessidade de algumas funcionalidades adicionais que resolveriam as limitações que descrevemos. Também precisamos explorar se existem exemplos que justifiquem a utilização dos operadores dos conjuntos da linguagem de especificação de *threads*. Vimos vários exemplos do uso da ferramenta em programas que utilizam semáforos e um exemplo em um programa que utiliza condições. Outro trabalho futuro seria utilizar a ferramenta em um programa que utiliza

semáforos e em uma outra implementação deste mesmo programa que utiliza variáveis de condições para investigar a relação entre eles.

A portabilidade da *EventManager* permite sua aplicação em outros ambientes. Isso se deve ao fato da maior parte do funcionamento da ferramenta ser feito em Lua. A linguagem C faz a manipulação das *threads* do programa, como acordar e colocá-las em espera, além de chamar as funções em Lua. Sendo assim, é possível substituir o papel da linguagem C por outra linguagem que possibilite uma manipulação de *threads* parecida e que consiga se comunicar com as funções em Lua.

A *EventManager* está nas etapas finais de seu desenvolvimento e ainda não conseguimos testá-la na prática, incentivando seu uso por alunos. Entretanto, uma primeira versão da ferramenta foi testada por alguns alunos da disciplina de Programação Distribuída e Concorrente em 2021.2.

A ferramenta está disponível no *link*: <https://github.com/AnnaLeticiaAlegria/Testes_Concorrentes>.

7
Referências bibliográficas

ANDREWS, G. **Foundations of Multithreaded, Parallel, and Distributed Programming**. [S.l.]: Addison-Wesley, 1999. Citado 6 vezes nas páginas 26, 32, 35, 36, 51 e 56.

BIANCHI, F. A.; MARGARA, A.; PEZZè, M. A survey of recent trends in testing concurrent software systems. **IEEE Transactions on Software Engineering**, 2018. Citado 2 vezes nas páginas 2 e 5.

DOWNEY, A. B. **The Little Book of Semaphores**. 2009. Citado 9 vezes nas páginas 1, 26, 27, 32, 35, 36, 56, 57 e 62.

ELMAS, T. et al. Concurrit: A domain specific language for reproducing concurrency bugs. In: **Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation**. [s.n.], 2013. ISBN 9781450320146. Disponível em: <https://doi.org/10.1145/2491956.2462162>. Citado na página 6.

HANSEN, P. B. Reproducible testing of monitors. **Software: Practice and Experience**, v. 8, n. 6, p. 721–729, 1978. Citado na página 7.

IERUSALIMSCHY, R. A text pattern-matching tool based on parsing expression grammars. **Software - Practice and Experience**, John Wiley amp; Sons, Inc., mar. 2009. ISSN 0038-0644. Citado na página 15.

IERUSALIMSCHY, R. **Programming in Lua, Fourth Edition**. [S.l.]: Lua.Org, 2016. Citado na página 14.

JAGANNATH, V. et al. Improved multithreaded unit testing. In: **Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering**. [s.n.], 2011. Disponível em: <https://doi.org/10.1145/2025113.2025145>. Citado 3 vezes nas páginas 3, 5 e 26.

LU, S. et al. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. **SIGOPS Oper. Syst. Rev.**, 2008. Disponível em: <https://doi.org/10.1145/1353535.1346323>. Citado na página 6.

MACHADO, F. B.; MAIA, L. P. **Arquitetura de Sistemas Operacionais**. 5th. ed. [S.l.]: LTC - Livros Técnicos e Científicos Editora, 2017. Citado na página 36.

MAYER, M.; MADHAVAN, R. A scala library for testing student assignments on concurrent programming. In: **Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala**. [s.n.], 2016. ISBN 9781450346481. Disponível em: <https://doi.org/10.1145/2998392.2998395>. Citado 5 vezes nas páginas 1, 6, 44, 47 e 48.

TANENBAUM, A. S.; BOS, H. **Modern Operating Systems**. 4th. ed. [S.l.]: Prentice Hall Press, 2014. Citado 2 vezes nas páginas 26 e 36.

THOMPSON, K. Programming techniques: Regular expression search algorithm. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 11, n. 6, p. 419–422, 1968. Disponível em: <https://doi.org/10.1145/363347.363387>. Citado na página 18.