UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MAYARA MARTINS POIM FERNANDES

ANÁLISE DE FERRAMENTAS DE TESTE NO CONTEXTO DE APRENDIZADO DE PROGRAMAÇÃO CONCORRENTE

RIO DE JANEIRO
2021

MAYARA MARTINS POIM FERNANDES

ANÁLISE DE FERRAMENTAS DE TESTE NO CONTEXTO DE APRENDIZADO DE PROGRAMAÇÃO CONCORRENTE

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Silvana Rossetto

RIO DE JANEIRO

2021

CIP - Catalogação na Publicação

Fernandes, Mayara Martins Poim
F363a Análise de ferramentas de teste no contexto de aprendizado de programação concorrente / Mayara Martins Poim Fernandes. -- Rio de Janeiro, 2021.
94 f.

Orientadora: Silvana Rossetto.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2021.

1. Computação Concorrente. 2. Testes. 3. Java. I. Rossetto, Silvana, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

MAYARA MARTINS POIM FERNANDES

ANÁLISE DE FERRAMENTAS DE TESTE NO CONTEXTO DE APRENDIZADO DE PROGRAMAÇÃO CONCORRENTE

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 10 de dezembro de 2021

BANCA EXAMINADORA:

gov.br Documento assinado digitalmente
SILVANA ROSSETTO
Data: 14/12/2021 06:16:29-0300
Verifique em https://verificador.iti.br

Silvana Rossetto, D.sC.
Instituto de Computação - UFRJ

Noemi de La Rocque Rodriguez, D.sC.
Departamento de Informática - PUC-Rio

Carolina Gil Marcelino, D.sC.
Instituto de Computação - UFRJ

Dedico este trabalho a todos que me auxiliaram nesta jornada acadêmica, em especial a meus professores do Instituto de Computação e à minha orientadora Silvana Rossetto. Também gostaria de dedicar à minha família; a meu parceiro, João Guio, e sua família; e a meus amigos, em especial Aline Rezende, Gilberto Lopes, Larissa Galeno e Victor Corrêa; por seu apoio contínuo durante todo este processo. Deixo também uma dedicatória para todos os meus gatos, em especial Banguela e Sushi, por tornarem os momentos difíceis mais toleráveis e sempre me fazerem companhia.

AGRADECIMENTOS

Agradeço a todo o corpo docente do Instituto de Computação da UFRJ, em especial à professora Silvana Rossetto, por toda a contribuição para o desenvolvimento deste trabalho. Também agradeço à Noemi Rodriguez e à Anna Leticia Alegria por seu apoio contínuo no processo de pesquisa e estudo sobre a área de testes de programas concorrentes.

RESUMO

Programas com múltiplos fluxos de execução são chamados programas concorrentes. Estes programas, devido às inúmeras possibilidades de intercalações destes fluxos e necessidade de sincronização entre eles, estão suscetíveis a diversos problemas não presentes na programação sequencial, como corridas de dados, deadlocks, e violações de atomicidade e ordem. Este não-determinismo traz um grande desafio tanto no desenvolvimento de sistemas concorrentes, quanto no aprendizado de conceitos de concorrência, principalmente para alunos iniciantes na área. Considerando esta dificuldade, este trabalho buscou encontrar ferramentas didáticas para realização de testes em programas concorrentes, analisando diversas abordagens propostas de teste e selecionando as ferramentas mais adequadas para o propósito acadêmico. As ferramentas FastTrack, RVPredict e Atomizer foram estudadas com maior detalhamento e experimentadas com programas clássicos de concorrência, como produtor-consumidor, avaliando problemas de corrida de dados e violação de atomicidade. Na avaliação de cada ferramenta, foram considerados sua usabilidade, facilidade de entendimento da saída apresentada, limitações de uso e acurácia. Dentre as técnicas estudadas, o FastTrack apresentou falsos positivos em seus resultados, mas obteve resultados acurados em programas que utilizam a primitiva de sincronização `synchronized`. O Atomizer apresentou falsos negativos, porém, tem o diferencial de explorar o problema de violação de atomicidade, que é pouco abordado por outras ferramentas. Já o RVPredict obteve resultados acurados, mas não é compatível com programas que possuem muitos laços, ou ramificações. Assim, é recomendado um uso complementar destas ferramentas, avaliando as necessidades do programa a ser testado.

Palavras-chave: concorrência; testes; corridas de dados; deadlock; violação de atomicidade; violação de ordem; ferramentas de teste; java.

ABSTRACT

Programs with multiple execution flows are called concurrent programs. These programs, due to the numerous possibilities of interleavings, and the need for synchronization between these flows, are susceptible to several problems not present in sequential programming, like data races, deadlocks, atomicity violation and order violation. This non-determinism entails in a great challenge both in the development of concurrent systems, and in the learning of concurrency concepts, specially for beginner students in this area of study. Taking this difficulty into account, this work sought to find didactic tools for testing concurrent programs, analyzing several proposed testing approaches and selecting the most suitable tools for an academic purpose. The tools FastTrack, RVPredict and Atomizer were studied in greater detail and tested with classic concurrency programs, such as producer-consumer, evaluating data race and atomicity violation problems. In the evaluation of each tool, its usability, ease of understanding the presented output, limitations, and accuracy were considered. Among the studied techniques, FastTrack presented false positives in its results, but obtained accurate results in programs that use the synchronization primitive `synchronized`. Atomizer presented false negatives, however, it has the advantage of exploiting the atomicity violation problem, which is little addressed by other tools. RVPredict, on the other hand, obtained accurate results, but it is not compatible with programs that have a lot of loops or branches. Therefore, a complementary use of these tools is recommended, evaluating the needs of the program to be tested.

Keywords: concurrency; tests; data race; deadlock; atomicity violation; order violation; testing tools; java.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

O estudo e aplicação de computação concorrente tiveram um crescimento no início dos anos 2000, quando a curva de eficiência de processadores, projetada pela Lei de Moore, encontrou uma barreira, não sendo possível aumentar o desempenho de um único processador, devido às limitações físicas. Assim, acelerou-se a pesquisa em processadores com múltiplos núcleos. Além disso, com o avanço da Internet, surgiu a necessidade de arquitetar sistemas distribuídos mais complexos, com milhares de clientes e servidores, normalmente em localizações geográficas distintas. Desta forma, o desenvolvimento de novas aplicações web e móveis tornaram a computação concorrente ainda mais relevante, e estimularam avanços nesta área.

Um programa concorrente é um programa com mais de um fluxo de execução. Estes fluxos podem ter suas execuções alternadas entre si em um mesmo processador, executar paralelamente em uma máquina com múltiplos processadores, ou em máquinas distintas, sendo este último caso classificado como computação distribuída.

O uso da programação concorrente permite utilizar o tempo que seria gasto aguardando resultados de operações de entrada/saída para realizar outra tarefa, reduzindo a latência do sistema. Também possibilita um melhor aproveitamento de múltiplos núcleos de processamento. Além disso, pode facilitar a modelagem de problemas que são naturalmente concorrentes, como em situações em que eventos físicos podem ser disparados concorrentemente.

No entanto, a existência de múltiplos fluxos de execução dentro de uma mesma aplicação dificulta o processo de coordenação entre todos estes fluxos, como tentar estabelecer uma sequência correta de interações entre eles ou sincronizar acessos a recursos compartilhados. Esta complexidade de controle de concorrência acarreta em desafios não observados na programação sequencial, como corridas de dados, violações de atomicidade e violações de ordem.

Para mitigar estes desafios, existem mecanismos (ou primitivas) de sincronização, como *locks* e semáforos, utilizados para sincronizar os fluxos de execução. Contudo, nem sempre o sincronismo necessário é facilmente identificável e implementável, e o próprio uso destes mecanismos pode levar a outros problemas de concorrência, como *deadlock*. O próprio comportamento de programas concorrentes é inerentemente não-determinístico, e, consequentemente, estes erros apresentados podem não se manifestar em inúmeras execuções, mesmo estando presentes no programa, impondo uma grande dificuldade no processo de depuração e checagem dessas aplicações.

Todos estes aspectos apresentados tornam clara a importância da disponibilidade de ferramentas adequadas que auxiliem nos testes de programas concorrentes, para mitigar estes desafios descritos, permitindo identificar erros de concorrência no processo de

desenvolvimento, e auxiliando na criação de programas confiáveis.

Com o crescente uso da computação concorrente, este paradigma se mostra cada vez mais presente no dia a dia de estudantes, e torna-se ainda mais necessário estabelecer uma compreensão mais completa dos conteúdos relacionados a esta área. Contudo, como já mencionado, a programação concorrente traz consigo diversos complicadores e desafios, que tornam seu desenvolvimento muito suscetível a erros lógicos, dificultando não apenas o desenvolvimento de aplicações concorrentes confiáveis, mas também o próprio aprendizado dos conteúdos relacionados.

O não-determinismo característico da computação concorrente pode facilmente levar estudantes iniciantes na área a considerar que um algoritmo está correto, quando na verdade ele não está. Por isto, ferramentas de teste são fundamentais na fase de aprendizado, com intuito de apontar esses erros que podem não ser notados em um primeiro momento.

Tendo em vista esta necessidade, o objetivo principal deste trabalho é identificar ferramentas didáticas para testes de programas concorrentes. Procurou-se, assim, ferramentas com entradas simples e saídas compreensíveis, voltadas para os paradigmas vistos em sala de aula. Estas ferramentas foram então estudadas e analisadas sob uma perspectiva de viabilidade de aplicação em ambiente acadêmico para auxiliar o processo de ensino, considerando sempre um público estudantil. O foco é investigar as técnicas de teste adotadas, experimentar as ferramentas e analisar seus resultados, discutindo sobre suas limitações, acurácia, facilidade de uso, e, de modo mais geral, se são aptas para servirem de apoio ao aprendizado de concorrência.

Este trabalho utiliza como referência básica o artigo intitulado "A Survey of Recent Trends in Testing Concurrent Software Systems"(BIANCHI; MARGARA; PEZZE, 2017), que realizou um levantamento extenso de técnicas recentes de testes para sistemas concorrentes propostas entre os anos 2000 e 2015, estabelecendo um esquema de classificação que utiliza para categorizar as técnicas. Além disto, também são apresentadas comparações e discussões sobre elas, indicando tendências e limitações nesta área de estudo.

A partir do esquema de classificação apresentado, estabeleceu-se filtros para selecionar abordagens para estudo e avaliação mais aprofundada, buscando selecionar técnicas que se encaixem nos requisitos acadêmicos de ensino de concorrência. Optou-se por abordagens voltadas para programas Java, uma vez que o levantamento de ferramentas de teste da pesquisa base apresentava uma quantidade maior de ferramentas Java, comparativamente com outras linguagens, como C. Além disso, Java é uma linguagem utilizada no ensino básico de concorrência. Portanto, todos os códigos apresentados neste trabalho são escritos em Java.

Estes crivos foram então aplicados na listagem de técnicas levantada na pesquisa, resultando em uma lista final de três ferramentas a serem analisadas: FastTrack (FLANAGAN; FREUND, 2009), RVPredict (HUANG; MEREDITH; ROSU, 2014), e Atomizer (FLANAGAN; FREUND, 2004), sendo as duas primeiras voltadas para identificação de

corridas de dados em programas concorrentes, e a última voltada para detectar violações de atomicidade.

O FastTrack é uma ferramenta voltada para identificar corridas de dados, de fácil instalação e com uma saída extensa. Por ser uma abordagem de dedução, não explora todas as intercalações possíveis para o programa sendo testado, podendo não identificar corridas que estejam dentro de ramos não executados. Não possui compatibilidade com semáforos e *locks* reentrantes, o que acarretou em notificações de alarmes falsos nos programas testados que utilizavam estes mecanismos. Contudo, apresentou bons resultados em códigos que utilizavam a primitiva `synchronized`.

O RVPredict também identifica corridas de dados, mas é uma técnica preditiva e mais completa, possuindo compatibilidade com mais mecanismos de sincronização. Nos testes executados, não reportou nenhum alarme falso. A ferramenta possui uma saída muito direta e clara, facilitando a compreensão do relatório de erro. Diferentemente do FastTrack, o RVPredict não responde bem a programas com laços com quantidade de iterações acima de 25, ou qualquer tipo de recurso que possa ramificar o fluxo de controle do programa, apresentando lentidão da ordem de minutos, e resultando em travamento da máquina.

Já o Atomizer é a única técnica analisada que aborda o problema de violação de atomicidade, apresentando um grande diferencial. Assim como o FastTrack, é uma ferramenta de detecção, possuindo o mesmo complicador de nem sempre identificar violações que estejam dentro de ramos. Foi observado a notificação de falsos negativos nos testes executados. É de fácil uso, e sua saída, apesar de extensa, é de fácil compreensão.

O estudo conclui que cada abordagem possui suas limitações e contextos mais apropriados. É sugerido, assim, um uso agregado de todas as três ferramentas, de modo a aplicar cada uma nas situações mais adequadas e também complementar uma a outra no processo de teste e aprendizado.

O restante deste trabalho está organizado da seguinte forma. O capítulo 2 introduz conceitos básicos sobre sistemas concorrentes, detalha os desafios deste paradigma de programação e explora alguns dos mecanismos existentes para lidar com estes desafios. O capítulo 3 discute as dificuldades de testar sistemas concorrentes, e discorre sobre um levantamento de técnicas propostas para testes para programas concorrentes, falando sobre critérios de classificação, e apresentando diversas etapas de filtro para selecionar as técnicas mais interessantes para estudo. No capítulo 4, as ferramentas selecionadas são apresentadas em maior detalhes, explicando sua metodologia, mostrando como utilizá-las, e apresentando seus resultados. Além disto, o capítulo também discorre sobre outras técnicas existentes em alguns compiladores e *toolkits* mais acessíveis. Finalizando, o capítulo 5 discute os resultados encontrados no estudo, e apresenta sugestões de trabalhos futuros.

2 CONCORRÊNCIA

Neste capítulo, é apresentada uma visão geral de concorrência, passando por sua definição, por tipos de fluxos de execução e paradigmas de comunicação entre eles, por desafios da programação concorrente e recursos existentes para impor sincronizações, assim como exemplos de problemas clássicos da área.

2.1 O CONCEITO DE CONCORRÊNCIA

Um fluxo de execução cuja execução sobrepõe no tempo com um outro fluxo é chamado de fluxo concorrente, e estes dois fluxos são classificados como executando concorrentemente. O caso geral de múltiplos fluxos executando concorrentemente é conhecido como concorrência (BRYANT; RICHARD, 2003).

O conceito de concorrência independe da quantidade de núcleos ou máquinas em que os fluxos estão executando. Ou seja, mesmo que dois fluxos estejam executando no mesmo processador, se houver sobreposição entre os dois, eles são concorrentes. Por exemplo, um processador com somente um núcleo não é capaz de factualmente executar quatro processos ao mesmo tempo, mas ele consegue passar a ilusão de estar executando estes quatro processos simultaneamente, realizando uma troca de contexto entre esses fluxos de execução, e alternando entre eles.

Caso dois ou mais fluxos estejam executando concorrentemente, mas em diferentes núcleos ou máquinas, estes fluxos podem ser classificados de forma mais específica como fluxos paralelos. Paralelismo é executar múltiplas tarefas simultaneamente, e é um conceito que depende da quantidade de processadores envolvidos.

2.2 PROCESSOS

Um processo é a abstração do sistema operacional para um programa em execução (BRYANT; RICHARD, 2003). E é por meio desta abstração que o sistema operacional provê a ilusão de que o programa em execução é o único sendo executado em todo o sistema, aparentando ter uso exclusivo do hardware.

Uma única CPU executa vários processos concorrentemente fazendo o processador efetuar trocas entre eles. Essas intercalações são realizadas pelo Sistema Operacional. Dessa forma, é possível ter centenas de processos rodando em uma única máquina contendo apenas uma CPU.

Utilizando múltiplas CPUs, é possível executar os processos de forma paralela, obtendo uma melhora em desempenho. Também é possível melhorar o desempenho no cenário com um único processador, quando existem muitas operações de I/O (Entrada/Saída), nas

quais os processos ficam bloqueados esperando estas operações completarem, enquanto outros processos recebem uma fatia de tempo do processador, ocupando a CPU.

2.3 THREADS

Um processo pode consistir de múltiplas threads, e elas podem ser executadas concorrentemente, em uma única CPU, ou de forma paralela, em múltiplas CPUs.

Diferentemente de processos, que possuem código e área de memória exclusivos, threads compartilham esses recursos, mas possuem pilha e registradores dedicados, pois estes são recursos necessários para que elas possam ser executadas independentemente.

2.4 COMUNICAÇÃO ENTRE FLUXOS DE EXECUÇÃO

Em sistemas concorrentes, muitas vezes é necessário que os diferentes fluxos de execução interajam entre si, utilizando alguma forma de comunicação. Esta comunicação pode ser feita por meio de memória compartilhada, ou por troca de mensagens.

Na comunicação via **memória compartilhada**, os fluxos interagem entre si por meio de acessos a uma memória em comum. Esta modalidade de interação exige que os fluxos de execução estejam localizados no mesmo nó, pois só assim este espaço de memória consegue ser compartilhado.

Já na comunicação por **troca de mensagem**, os fluxos de execução interagem entre si trocando mensagens por meio de um canal de comunicação. Este modelo permite que os fluxos de execução estejam hospedados tanto no mesmo nó físico, quanto em nós diferentes, sendo este último exemplo um sistema distribuído.

Ambas modalidades podem ser utilizadas por processos e threads. Para interações *inter-threads*, tradicionalmente usa-se comunicação por memória compartilhada, pois as *threads* compartilham uma única área de memória. Contudo, a linguagem de programação pode oferecer mecanismos de troca de mensagem, permitindo que esta forma de comunicação seja construída em cima da memória compartilhada, modelando operações de envio e recebimento de mensagens como operações de escrita e leitura na área de memória.

Interações inter-processos, por sua vez, comportam as duas modalidades. No entanto, a comunicação por memória compartilhada entre processos é mais custosa do que entre *threads*.

2.5 DESAFIOS DA PROGRAMAÇÃO CONCORRENTE

Na programação concorrente, os fluxos de execução intercalam suas execuções no tempo, podendo haver uma variação nestes entrelaçamentos a cada execução do programa. Ou seja, o comportamento do sistema está intrinsecamente ligado à ordem de

execução destes fluxos, e à forma com a qual eles interagem entre si. Esta característica da concorrência traz novos desafios de programação, não presentes na programação sequencial, como tratamento de corridas de dados, violações de atomicidade e violações de ordem. Estes problemas podem levar a falhas ou erros lógicos no programa, e, portanto, é relevante entender suas definições e o que pode causá-los. Nas próximas subseções abordaremos cada um deles.

2.5.1 Corrida de dados

Uma corrida de dados ocorre quando duas operações de fluxos de execução distintos acessam um local de memória compartilhado de forma concorrente, sendo pelo menos uma das operações uma escrita (NETZER; MILLER, 1992).

Código 1 – Exemplo de código Java com corrida de dados

```
class SampleThread extends Thread {
    static int sum = 0;

    public void run() {
        for (int i = 0; i < 100000; i++) {
            sum += 1;
        }
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread threadA = new SampleThread();
        Thread threadB = new SampleThread();

        threadA.start();
        threadB.start();

        threadA.join();
        threadB.join();
        System.out.println(SampleThread.sum);
    }
}
```

O código 1 mostra um exemplo de corrida de dados. Nele, é criada uma classe `SampleThread` que herda da classe `Thread` do Java, por meio da palavra-chave `extends`. Nesta subclasse, o método `run()` é sobrescrito para conter o código que deve ser executado ao iniciar uma instância de `SampleThread`. Na função `main()` são criadas duas instâncias de `SampleThread`, `threadA` e `threadB`, utilizando o comando `new`.

Em seguida, as *threads* recém criadas são iniciadas, por meio da chamada da função

`start()`, e vão executar o código escrito na função `run()`. Após iniciar as *threads* criadas, o programa faz com que o fluxo principal aguarde o término destas threads, chamando o método `join()`, que coloca a *thread* que o invocou em um estado de espera até que as *threads* referenciadas finalizem.

Tanto a `threadA` quanto a `threadB` operam sobre uma mesma variável estática `sum`, que pertence à classe `SampleThread`, somando 1 a seu valor a cada iteração de um *loop* que vai de 0 a 100000. Após ambas as threads finalizarem, o fluxo principal imprime o valor de `sum` na saída padrão.

Ao executar o programa, uma das threads pode ler um valor de `sum` em um loop, e antes de efetuar a escrita deste valor incrementado de volta na memória, a outra thread efetua uma (ou mais) escritas do `sum` incrementado. Isto resulta na primeira thread sobrescrevendo o valor de `sum` de modo a descartar as alterações feitas pela outra thread. Desta forma, o resultado final impresso não seria 200000, como o esperado, pois houve uma perda de incrementos, e seu valor irá depender da ordem em que as operações são executadas.

2.5.2 Violação de atomicidade

Apesar de corrida de dados ser um dos problemas mais comuns em programas concorrentes, nem sempre um código livre desta corrida, cujos acessos estão todos protegidos, está isento de erros lógicos. Muitas vezes, é necessário garantir a **atomicidade** das operações, e não somente a ausência de corridas.

Atomicidade é a propriedade que garante que quando um fluxo modifica o estado da aplicação, um outro fluxo qualquer só consegue observar o estado anterior à operação, ou o estado resultante da operação, não conseguindo ver nenhum estado intermediário (ARPACI-DUSSEAU; ARPACI-DUSSEAU, 2018).

Uma violação de atomicidade é decorrente da violação da seriação desejada entre múltiplas operações de acesso à memória. Ou seja, quando uma região de código deve ser atômica, mas essa atomicidade não é garantida durante a execução (LU et al., 2008).

O programa apresentado nos códigos 2 e 3 apresenta uma situação de violação de atomicidade. Nele, existe uma classe `Account` que possui um atributo `balance` para guardar o saldo de uma conta, e possui métodos para obter este saldo (`getBalance`), configurar o valor do saldo para um novo valor especificado no parâmetro (`setBalance`), e incrementar ou decrementar o saldo por um valor passado (`increaseBalanceBy` e `decreaseBalanceBy`, respectivamente).

No programa principal, são criadas e iniciadas duas *threads* distintas que compartilham uma única instância de `Account`, com um saldo inicial de 500. Uma destas *threads* é uma instância da classe `RunThread`, e na sua execução, irá verificar se o saldo da conta é superior a uma quantia `amount = 100`. Caso seja, ela irá decrementar esta quantia da conta. Esta verificação existe para evitar que a conta fique com o saldo zerado ou negativo,

que seria um resultado errôneo. A outra *thread* é uma instância de `ModifierThread`, e tem como função configurar o saldo da conta para um novo valor, 10. Ao final da execução do programa, a *thread* principal imprime o valor final do saldo na saída padrão.

É possível observar uma violação de atomicidade na operação da instância de `RunThread`, pois o estado do saldo pode ser alterado pela *thread* modificadora entre a chamada de `getBalance()`, feita na verificação de saldo, e a chamada de decremento, podendo, assim, resultar em um saldo negativo de -90. A operação apropriada do programa seria a *thread* efetuar a checagem de saldo e decremento de forma atômica, e as outras *threads* só enxergarem o valor final do saldo após estas operações. É importante ressaltar que também existe uma corrida de dados neste programa, pois os acessos ao atributo de saldo não estão sincronizados.

Código 2 – Exemplo de código Java com violação de atomicidade - classe Account

```
class Account {
    private double balance;

    Account(double balance) {
        this.balance = balance;
    }

    double getBalance() {
        return balance;
    }

    void setBalance(double balance) {
        this.balance = balance;
    }

    void increaseBalanceBy(double amount) {
        balance += amount;
    }

    void decreaseBalanceBy(double amount) {
        balance -= amount;
    }
}
```

Código 3 – Exemplo de código Java com violação de atomicidade - Threads

```
class ModifierThread extends Thread {
    Account account;

    public ModifierThread(Account account) {
        this.account = account;
    }

    public void run() {
        account.setBalance(10);
    }
}

class RunThread extends Thread {
    Account account;

    public RunThread(Account account) {
        this.account = account;
    }

    public void run() {
        double amount = 100;
        if (account.getBalance() > amount) {
            account.decreaseBalanceBy(amount);
        }
    }
}

public class AtomicityViolation {
    public static void main(String[] args) throws InterruptedException {
        Account account = new Account(500);
        Thread a = new RunThread(account);
        Thread b = new ModifierThread(account);
        a.start();
        b.start();

        a.join();
        b.join();
        System.out.println(account.getBalance());
    }
}
```

2.5.3 Violação de Ordem

Corridas de dados e violações de atomicidade são problemas que costumam resultar em violações de ordem de intercalações específicas. No entanto, muitas violações de ordem

de entrelaçamentos não podem ser reduzidas a estas propriedades mais conhecidas.

Essencialmente, uma violação de ordem ocorre quando a ordem de execução desejada entre duas operações de acessos à memória é invertida. Por exemplo, o acesso A deve sempre ser executado antes do acesso B, mas esta ordem não é garantida durante a execução (LU et al., 2008).

Em alguns programas, pode ser necessário que os fluxos observem alguma condição de estado de execução, como um *buffer* estar vazio ou cheio, e só operem quando esta condição for atendida. Ou seja, é estabelecida uma regra de ordem de execução de entrelaçamentos decorrente das especificações do programa.

Já outros programas podem levar a uma sequência de intercalações que acarretam em uma desreferência a um ponteiro nulo, ocasionada quando um fluxo de execução tenta acessar uma referência de objeto após a ele ter sido atribuído um valor nulo, ou antes dele ser inicializado por um outro fluxo. Neste caso, ocorre uma violação de ordem associada a uma falha específica, que no caso é de desreferência a um ponteiro nulo (também conhecida como `NullPointerException`).

No código 4 é exposto um exemplo em que esta situação ocorre. O programa apresenta uma variável `violationTest`, inicialmente com valor nulo, uma `threadA`, que irá inicializar esta variável com uma instância da classe `ViolationTest`, e uma `threadB`, que vai imprimir na saída padrão o valor do atributo `item` desta instância.

O fluxo desejado deste programa é que a `threadB` só execute quando a variável `violationTest` não estiver nula. Ou seja, existe uma ordem de execução esperada em que a `threadA` executa antes da `threadB`. Porém, não existe nenhuma garantia da ordem de execução destes fluxos, então é possível que a `threadB` execute primeiro, e tente acessar um atributo de uma variável nula, causando uma exceção de `null pointer`.

Código 4 – Exemplo de código Java com violação de ordem

```
class ViolationTest {
    int item;

    ViolationTest(int item) {
        this.item = item;
    }
}

public class OrderViolation {
    static ViolationTest violationTest = null;

    public static void main(String[] args) throws InterruptedException {
        Thread threadA = new Thread() {
            public void run() {
                violationTest = new ViolationTest(1);
            }
        };

        Thread threadB = new Thread() {
            public void run() {
                System.out.println(violationTest.item);
            }
        };

        threadA.start();
        threadB.start();
    }
}
```

2.6 MECANISMOS DE SINCRONIZAÇÃO

Como visto anteriormente, o paradigma concorrente adiciona muitos desafios decorrentes das inúmeras possibilidades de intercalações entre fluxos de execução, e da interação entre eles. Para lidar com estas complicações, é necessário efetuar sincronizações entre os fluxos. No restante desta seção, serão descritos dois mecanismos de sincronização comumente utilizados, *locks* e semáforos, passando por suas definições e aplicações.

2.6.1 Lock

Locks são mecanismos utilizados para implementar sincronização por exclusão mútua. Para resolver o problema de corrida de dados, por exemplo, as linhas de código contendo os acessos aos recursos compartilhados devem ser transformadas em uma seção crítica, que consiste em um bloco de código que pode ser executado por apenas uma thread. Esta

propriedade de controle de concorrência é chamada de exclusão mútua (HERLIHY et al., 2020).

O código 5 discrimina as operações feitas sobre a variável compartilhada `sum` da classe `SampleThread`, indicando as possíveis zonas de perigo que devem ser transformadas em seções críticas, a fim de garantir a exclusão mútua entre as operações de leitura (acesso ao valor da variável para efetuar a soma) e escrita (atribuição do resultado da soma), e resolver a condição de corrida presente no código.

Código 5 – Código 1 com acessos a recurso compartilhado explícitos

```
class SampleThread extends Thread {
    static int sum = 0; // recurso compartilhado

    public void run() {
        for (int i = 0; i < 100000; i++) {
            // inicio de acesso de leitura e escrita
            // de recurso compartilhado
            sum += 1;
            // fim de acesso de leitura e escrita
            // de recurso compartilhado
        }
    }
}
```

Locks, também chamados de *mutex* (acrônimo de *mutual exclusion*, exclusão mútua em inglês), funcionam como uma fechadura. Antes de entrar em uma seção crítica, o fluxo de execução deve trancar o *lock*, e ao final da seção, o *lock* deve ser liberado. Enquanto houver um fluxo executando a seção crítica, qualquer outro fluxo que queira executá-la deve aguardar a liberação do *lock*.

No código 6, é apresentado uma versão da classe `SampleThread` que resolve o problema de condição de corrida presente anteriormente no programa do código 1, por meio de uma implementação de exclusão mútua utilizando um *lock*. O mecanismo é utilizado para delimitar a seção crítica discriminada no código 5, por meio de funções `lock()` e `unlock()`, que fazem a aquisição e liberação do *lock*.

O *lock* disponibilizado na biblioteca padrão do Java é um *lock* reentrante `ReentrantLock()`, que é um tipo de *mutex* que pode ser adquirido múltiplas vezes pela mesma *thread* (HERLIHY et al., 2020).

Código 6 – Exclusão mútua usando *lock*

```
class SampleThread extends Thread {
    static int sum = 0;
    static Lock lock = new ReentrantLock();

    public void run() {
        for (int i = 0; i < 100000; i++) {
            lock.lock();
            sum += 1;
            lock.unlock();
        }
    }
}
```

Pode-se afirmar que a exclusão mútua tem como objetivo garantir que operações de acessos a memória sejam atômicas, uma definição mais geral que demonstra que esta modalidade de sincronização também serve para resolver problemas de violação de atomicidade, não somente de corrida de dados.

O código 7 mostra uma solução para o problema de corrida de dados presente no código 2 descrito na seção 2.5.2, utilizando blocos `synchronized` nas assinaturas dos métodos para fazer exclusão mútua, garantindo que somente uma *thread* consiga acessar o atributo de cada vez.

A palavra-chave `synchronized`, também chamada de *lock* intrínseco, pode ser utilizada a nível de métodos, ou de blocos de código em Java, e está vinculada a um objeto. Portanto, todos os blocos `synchronized` de um mesmo objeto podem ter somente uma *thread* os executando a cada instante. Quando a palavra-chave é colocada diretamente na assinatura do método, o objeto de *lock* implícito é o `this` (ORACLE, 2021).

Porém, esta correção da corrida de dados não resolve o problema da violação de atomicidade, pois ainda é possível que outra *thread* modifique o saldo entre a verificação e o decremento. Para solucionar esta violação, adiciona-se um bloco `synchronized` envolvendo a operação de checagem e de decremento, atrelado ao objeto da conta. Desta forma, é garantido que somente uma *thread* irá acessar a instância de `Account`, assegurando a atomicidade do conjunto de operações. Ou seja, a *thread* modificadora não consegue mais alterar o valor do saldo entre as operações de verificação e decremento.

Código 7 – Correção de corrida de dados - classe Account

```
class Account {
    private double balance;

    Account(double balance) {
        this.balance = balance;
    }

    synchronized double getBalance() {
        return balance;
    }

    synchronized void setBalance(double balance) {
        this.balance = balance;
    }

    synchronized void increaseBalanceBy(double amount) {
        balance += amount;
    }

    synchronized void decreaseBalanceBy(double amount) {
        balance -= amount;
    }
}
```

Código 8 – Correção de violação de atomicidade

```
class RunThread extends Thread {
    Account account;

    public RunThread(Account account) {
        this.account = account;
    }

    public void run() {
        double amount = 100;
        synchronized(account) {
            if (account.getBalance() >= amount) {
                account.decreaseBalanceBy(amount);
            }
        }
    }
}
```

2.6.2 Semáforo

Um semáforo é a generalização de um *lock* de exclusão mútua (HERLIHY et al., 2020). Semáforos são contadores com operações atômicas que podem ser usados tanto para efetuar uma sincronização por exclusão mútua, como descrita na seção 2.6.1, quanto para sincronização condicional.

Existem muitos casos em que uso de exclusão mútua não é suficiente para a construção de um programa concorrente. Às vezes, um fluxo precisa verificar se uma condição é verdadeira ou não antes de dar sequência a sua execução, como uma *thread* que precisa verificar se um buffer não está cheio antes de inserir nele (ARPACI-DUSSEAU; ARPACI-DUSSEAU, 2018). Esta sincronização com base em uma condição é chamada de sincronização condicional.

É importante ressaltar que é possível efetuar algumas verificações de condição por meio de exclusão mútua, utilizando uma variável compartilhada que é checada pelo fluxo em *loop* até se tornar verdadeira. No entanto, em geral esta solução é muito ineficiente, e ocupa desnecessariamente a CPU, sendo mais interessante uma abordagem que faça com que o fluxo aguardando a condição fique inativo ou preso até que ele seja notificado da mudança no estado da condição.

Um semáforo é uma variável especial que contém um contador inteiro N, uma fila de tarefas inicialmente vazia associada a ele, e um conjunto de operações atômicas para manipulá-lo. O valor inicial atribuído ao contador é denominado sua capacidade, e depende de como o semáforo será utilizado. Semáforos com capacidade $N = 1$ são chamados de semáforos binários, pois estão restritos aos valores 0 e 1, representando os estados bloqueado e liberado, e são utilizados normalmente para implementar *locks*, e, consequentemente, sincronização por exclusão mútua. Já a capacidade de ter múltiplos sinais costuma ser utilizada para implementar sincronização condicional.

Semáforos possuem duas operações atômicas principais: decremento e incremento de sinais. A operação de decremento verifica se é possível decrementar sinais. Isto é, se o contador não está zerado, uma vez que ele não pode ficar negativo. Caso seja possível, a operação decrementa, e o fluxo segue a execução. Caso contrário, o fluxo será bloqueado e vai para a fila de tarefas bloqueadas do semáforo. Já a operação de incremento vai incrementar o contador, e verificar se existem tarefas na fila do semáforo. Caso haja, irá sinalizar para que uma delas seja desbloqueada.

Código 9 – Exclusão mútua usando semáforo

```
class SampleThread extends Thread {
    static int sum = 0;
    static Semaphore sem = new Semaphore(1);

    public void run() {
        try {
            for (int i = 0; i < 100000; i++) {
                sem.acquire();
                sum += 1;
                sem.release();
            }
        } catch (InterruptedException e) {
            System.out.println("Error " + e);
        }
    }
}
```

O código 9 apresenta uma solução para o problema de corrida de dados descrito no código 1, seguindo a mesma ideia implementada utilizando *locks* no programa 6, descrito na seção 2.6.1, mas utilizando um semáforo binário para implementar a exclusão mútua. Esta abordagem é equivalente ao uso de *lock*, em que um semáforo comum às instâncias da classe `SampleThread` é utilizado para delimitar a seção crítica, por meio das operações de `acquire` e `release`.

É possível também utilizar semáforos para resolver problemas de violação de ordem, por meio de sincronização condicional. No código 10, é apresentada uma solução para o problema exposto no código 4, descrito na seção 2.5.3. Houve a adição de um semáforo `initConditionSem`, para controlar a condição de inicialização da variável `violationTest`. Este semáforo é configurado inicialmente com capacidade zerada, indicando que a variável ainda não foi inicializada.

Para garantir que a `threadB` só execute quando a variável não estiver nula, ela deve tentar adquirir o semáforo antes de operar em cima da variável. Caso a condição de inicialização de `violationTest` ainda não tenha sido atendida, com a sinalização do semáforo, a *thread* ficará bloqueada aguardando a condição.

Após a `threadA` inicializar `violationTest` com uma instância da classe `ViolationTest`, ela irá liberar o semáforo, efetuando um incremento. A partir deste momento, a `threadB` consegue adquiri-lo (saindo da espera, caso tenha ficado aguardando) e imprimir o atributo do objeto. Desta forma, é sempre garantido que a `threadB` só irá operar em cima da variável após ela ter sido inicializada pela `threadA`, assegurando a ordem esperada de execução.

Código 10 – Correção de violação de ordem com semáforo condicional

```
class ViolationTest {
    int item;

    ViolationTest(int item) {
        this.item = item;
    }
}

public class OrderViolationSemaphore {
    static ViolationTest violationTest = null;
    static Semaphore initConditionSem = new Semaphore(0);

    public static void main(String[] args) throws InterruptedException {
        Thread threadA = new Thread() {
            public void run() {
                violationTest = new ViolationTest(1);
                initConditionSem.release();
            }
        };

        Thread threadB = new Thread() {
            public void run() {
                try {
                    initConditionSem.acquire();
                    System.out.println(violationTest.item);
                } catch (InterruptedException e) {
                    System.out.println("Error " + e);
                }
            }
        };

        threadB.start();
        threadA.start();
    }
}
```

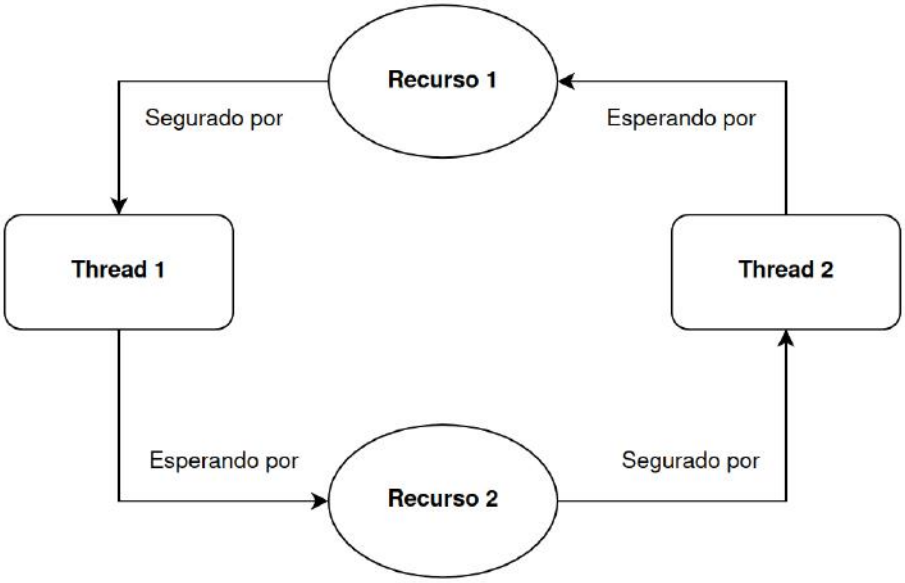
2.7 DEADLOCK

Mecanismos de sincronização podem gerar um erro de execução conhecido como *deadlock*, em que uma coleção de *threads* é bloqueada, esperando por uma condição que jamais será verdadeira (BRYANT; RICHARD, 2003).

O diagrama apresentado na figura 1 descreve uma ocorrência clássica de *deadlock*. A *Thread* 1 adquiriu o Recurso 1, e a *Thread* 2 adquiriu o Recurso 2. Em algum momento posterior, antes de liberar o Recurso 1, a *Thread* 1 pede o Recurso 2, enquanto a *Thread*

2 pede o Recurso 1 antes de liberar o Recurso 2. Desta forma, ambos os fluxos ficam aguardando uma condição que nunca ocorrerá, e o programa entra em *deadlock*.

Figura 1 – Diagrama de um *deadlock*



2.8 PROBLEMA DO PRODUTOR-CONSUMIDOR

Um exemplo clássico de sincronização de múltiplos fluxos de execução é o problema do produtor-consumidor. Em sua versão mais simples, ele possui dois fluxos cíclicos — um produtor e um consumidor — e ambos compartilham um *buffer* de tamanho fixo. O produtor continuamente produz novos itens e os insere no *buffer*, enquanto o consumidor repetidamente lê os dados do *buffer*, consumindo-os de alguma forma. Este fluxo de produção e consumo é apresentado no diagrama da figura 2. Também é possível ter múltiplos produtores e consumidores.

Figura 2 – Diagrama de um problema Produtor-Consumidor

Código 11 – Primeira tentativa de implementação do problema Produtor-Consumidor - Threads

```
class Producer extends Thread {
    int[] buffer;
    int[] bufferPos;
    int id;

    Producer(int[] buffer, int[] bufferPos, int id) {
        this.buffer = buffer;
        this.bufferPos = bufferPos;
        this.id = id;
    }

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            buffer[bufferPos[0]] = id;
            bufferPos[0] = (bufferPos[0] + 1) % buffer.length;
        }
    }
}

class Consumer extends Thread {
    int[] buffer;
    int[] bufferPos;

    Consumer(int[] buffer, int[] bufferPos) {
        this.buffer = buffer;
        this.bufferPos = bufferPos;
    }

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            int item = buffer[bufferPos[1]];
            buffer[bufferPos[1]] = -1;
            bufferPos[1] = (bufferPos[1] + 1) % buffer.length;
            System.out.println(item);
        }
    }
}
```

Código 12 – Primeira tentativa de implementação do problema Produtor-Consumidor - Fluxo Principal

```
public class ProdCons {
    public static void main(String[] args) throws InterruptedException {
        int BUFFER_SIZE = 3;

        Producer[] prods = new Producer[3];
        Consumer[] cons = new Consumer[3];

        int[] buffer = new int[BUFFER_SIZE];
        int[] bufferPos = {0, 0};
        // bufferPos[0] dedicated to producers and bufferPos[1] to
            consumers

        for (int i = 0; i < BUFFER_SIZE; i++) {
            buffer[i] = -1;
        }

        for (int i = 0; i < 3; i++) {
            prods[i] = new Producer(buffer, bufferPos, i);
            cons[i] = new Consumer(buffer, bufferPos);
        }

        for (int i = 0; i < 3; i++) {
            prods[i].start();
            cons[i].start();
        }

        for (int i = 0; i < 3; i++) {
            prods[i].join();
            cons[i].join();
        }
    }
}
```

O código descrito em 11 e 12 apresenta uma primeira tentativa de implementação para o problema, com três produtores e três consumidores sendo criados no fluxo principal, um *buffer* com tamanho igual a três, e uma variável para controlar as posições para inserção e remoção de itens do *buffer*.

O produtor insere seu identificador de *thread* no *buffer* compartilhado, enquanto o consumidor retira um item de cada vez, imprimindo o conteúdo na saída padrão, e reiniciando o valor escrito na posição do buffer à medida que consome. Cada produtor e consumidor executa estas ações cinco vezes.

Na implementação ingênua descrita em 11 e 12, é possível observar inconsistências em

seus resultados, pois o programa apresenta uma corrida de dados. As operações de inserção e remoção de itens do problema envolve a atualização de duas variáveis compartilhadas entre as threads — o `buffer` e o `bufferPos`. Portanto, é necessário que seja implementado algum mecanismo de exclusão mútua nestes acessos, para garantir a consistência dos recursos compartilhados.

Além desta exclusão mútua, é preciso controlar os acessos ao *buffer* observando seu estado atual, com uma sincronização condicional. Caso o *buffer* esteja cheio, o produtor deve aguardar até que haja um espaço disponível. Analogamente, caso o *buffer* esteja vazio, o consumidor deve aguardar até que exista algum item disponível para ser consumido.

A solução mais completa para este problema está descrita nos códigos 13, 14 e 15, e consiste na utilização de dois semáforos contadores: `fillCountSem`, que é inicializado com zero, e `emptyCountSem`, que é inicializado com o tamanho do *buffer*. Esses semáforos são usados para manter a contagem de espaços disponíveis para serem consumidos, e preenchidos, respectivamente, realizando uma sincronização condicional. Além destes semáforos contadores, também é necessário utilizar um semáforo binário para estabelecer a exclusão mútua nos acessos ao `buffer` e ao `bufferCount`.

O Produtor começa tentando adquirir o semáforo que conta quantas posições podem ser preenchidas `emptyCountSem`. Caso haja uma posição disponível, ele irá decrementar este valor, indicando que uma posição foi preenchida. Caso não haja, ele ficará aguardando até que uma posição seja liberada. Em seguida, o produtor tenta adquirir o semáforo de exclusão mútua, e caso consiga, adentra na região crítica, inserindo no *buffer* e atualizando o valor da próxima posição para inserção. Ao finalizar, ele sai da região crítica e notifica que uma das posições foi preenchida e pode ser consumida, incrementanto o semáforo `fillCountSem`.

O Consumidor, por sua vez, inicia seu fluxo verificando se existe algum item para consumir por meio do semáforo `fillCountSem`. Caso haja um item para consumir, ele irá decrementar o valor do semáforo, indicando que uma posição está sendo consumida. Caso contrário, ele aguardará a disponibilidade. Em sequência, ele executa uma lógica análoga à do Produtor para adentrar na seção crítica, na qual ele consome o item do *buffer* e atualiza a próxima posição de consumo. Após liberar a exclusão mútua, ele notifica que uma das posições foi consumida e pode ser preenchida, incrementando o semáforo `emptyCountSem`.

Código 13 – Implementação do problema Produtor-Consumidor usando semáforos - Produtor

```
class Producer extends Thread {
    Semaphore emptyCountSem;
    Semaphore fillCountSem;
    Semaphore bufferSem;
    int[] buffer;
    int[] bufferPos;
    int id;

    Producer(Semaphore emptyCountSem, Semaphore fillCountSem, Semaphore
       bufferSem, int[] buffer, int[] bufferPos, int id) {
        this.emptyCountSem = emptyCountSem;
        this.fillCountSem = fillCountSem;
        this.bufferSem = bufferSem;
        this.buffer = buffer;
        this.bufferPos = bufferPos;
        this.id = id;
    }

    @Override
    public void run() {
        int item;
        for (int i = 0; i < 5; i++) {
            try {
                item = id;
                emptyCountSem.acquire();
                bufferSem.acquire();

                buffer[bufferPos[0]] = item;
                bufferPos[0] = (bufferPos[0] + 1) % buffer.length;

                bufferSem.release();
                fillCountSem.release();
            } catch (InterruptedException e) {
                System.out.println("Error " + e);
            }
        }
    }
}
```

Código 14 – Implementação do problema Produtor-Consumidor usando semáforos - Consumidor

```
class Consumer extends Thread {
    Semaphore emptyCountSem;
    Semaphore fillCountSem;
    Semaphore bufferSem;
    int[] buffer;
    int[] bufferPos;

    Consumer(Semaphore emptyCountSem, Semaphore fillCountSem, Semaphore
       bufferSem, int[] buffer, int[] bufferPos) {
        this.emptyCountSem = emptyCountSem;
        this.fillCountSem = fillCountSem;
        this.bufferSem = bufferSem;
        this.buffer = buffer;
        this.bufferPos = bufferPos;
    }

    @Override
    public void run() {
        int item;
        for (int i = 0; i < 5; i++) {
            try {
                fillCountSem.acquire();
                bufferSem.acquire();

                item = buffer[bufferPos[1]];
                buffer[bufferPos[1]] = -1;
                bufferPos[1] = (bufferPos[1] + 1) % buffer.length;

                bufferSem.release();
                emptyCountSem.release();

                System.out.println(item);
            } catch (InterruptedException e) {
                System.out.println("Error " + e);
            }
        }
    }
}
```

Código 15 – Implementação do problema Produtor-Consumidor usando semáforos - Fluxo Principal

```
public class PCSemaphore {
    public static void main(String[] args) throws InterruptedException {
        int BUFFER_SIZE = 3;

        Producer[] prods = new Producer[3];
        Consumer[] cons = new Consumer[3];

        Semaphore emptyCountSem = new Semaphore(BUFFER_SIZE);
        Semaphore fillCountSem = new Semaphore(0);
        Semaphore bufferSem = new Semaphore(1);
        int[] buffer = new int[BUFFER_SIZE];
        int[] bufferPos = {0, 0};

        for (int i = 0; i < BUFFER_SIZE; i++) {
            buffer[i] = -1;
        }

        for (int i = 0; i < 3; i++) {
            prods[i] = new Producer(emptyCountSem, fillCountSem,
                bufferSem, buffer, bufferPos, i);
            cons[i] = new Consumer(emptyCountSem, fillCountSem,
                bufferSem, buffer, bufferPos);
        }

        for (int i = 0; i < 3; i++) {
            prods[i].start();
            cons[i].start();
        }

        for (int i = 0; i < 3; i++) {
            prods[i].join();
            cons[i].join();
        }
    }
}
```

É relevante ressaltar a importância da ordem correta de aquisição e liberação dos semáforos, uma vez que o uso incorreto destes recursos pode levar a complicações como *deadlocks*. Por exemplo, o código 16 apresenta uma inversão da ordem de chamadas dos semáforos feita pelo Consumidor, agora, ele primeiro tenta adquirir o semáforo de exclusão mútua e somente em seguida verifica a existência ou não de um item para consumir, por meio do semáforo condicional `fillCountSem`. Neste fluxo, é possível que o consumidor adquira o semáforo de exclusão mútua e se bloqueie em seguida porque o *buffer* está

vazio, mantendo presa a exclusão mútua e, consequentemente, impedindo que produtores insiram no *buffer* e causando uma situação de *deadlock*.

Código 16 – Produtor-Consumidor com deadlock - Consumidor

```
class Consumer extends Thread {
    Semaphore emptyCountSem;
    Semaphore fillCountSem;
    Semaphore bufferSem;
    int[] buffer;
    int[] bufferPos;

    Consumer(Semaphore emptyCountSem, Semaphore fillCountSem, Semaphore
       bufferSem, int[] buffer, int[] bufferPos) {
        this.emptyCountSem = emptyCountSem;
        this.fillCountSem = fillCountSem;
        this.bufferSem = bufferSem;
        this.buffer = buffer;
        this.bufferPos = bufferPos;
    }

    @Override
    public void run() {
        int item;
        for (int i = 0; i < 5; i++) {
            try {
                bufferSem.acquire();
                fillCountSem.acquire();

                item = buffer[bufferPos[1]];
                buffer[bufferPos[1]] = -1;
                bufferPos[1] = (bufferPos[1] + 1) % buffer.length;

                bufferSem.release();
                emptyCountSem.release();

                System.out.println(item);
            } catch (InterruptedException e) {
                System.out.println("Error " + e);
            }
        }
    }
}
```

2.9 DISCUSSÃO

Neste capítulo, observou-se que a necessidade de lidar com múltiplos fluxos de execução acarreta em novos problemas e dificuldades que não são observados na programação sequencial. Isso ocorre pois o comportamento de sistemas concorrentes depende não só da sequência de ações executadas em cada fluxo individual, mas também da intercalação das ações nos diferentes fluxos de execução (BIANCHI; MARGARA; PEZZE, 2017). Mesmo que cada fluxo esteja correto individualmente, ainda é possível que haja uma falha na execução do sistema como um todo, por conta das interações dos fluxos entre si, e com o próprio sistema.

Além de complicações como **corrida de dados**, **violação de atomicidade** e **violação de ordem**, foi exemplificado que as próprias abordagens para lidar com estes problemas — os mecanismos de sincronização — se usadas erroneamente, podem levar a outras adversidades, como deadlock. Até mesmo problemas tradicionais descritos neste capítulo, como **Produtor-Consumidor**, que possuem soluções clássicas e conhecidas para suas versões mais simples, não estão imunes de erro em suas implementações. Dificilmente será implementado a versão mais básica do problema, e portanto, é necessário adaptar a abordagem para atender às restrições específicas do sistema, e potencialmente mais falhas de concorrência poderão surgir. Quanto mais complexo o programa concorrente for, mais cuidado será necessário na hora de implementá-lo.

A programação concorrente é muito suscetível a erros lógicos, tornando o desenvolvimento destes programas uma tarefa mais complexa para desenvolvedores, e, principalmente, para alunos que ainda estão aprendendo e se familiarizando com o paradigma concorrente.

Mesmo que alguns destes problemas presentes na computação concorrente tenham definições bem estruturadas e meios existentes para resolvê-los, nem sempre eles são facilmente identificáveis. Pela característica inerente de não-determinismo da programação concorrente, estas falhas podem não se manifestar, mesmo com múltiplas execuções do programa, dificultando o processo de aprendizado e de desenvolvimento de programas concorrentes.

Como é possível, então, testar a corretude de programas concorrentes, a fim de lidar com estas dificuldades apresentadas? No próximo capítulo, serão apresentadas algumas ferramentas que auxiliam na detecção de erros lógicos em programas concorrentes.

3 TESTES EM PROGRAMAS CONCORRENTES

A testagem de um programa consiste no processo de analisar um componente de software para detectar as diferenças entre condições existentes e requisitadas, e avaliar os atributos destes componentes (IEEE, 1990). Testes são realizados para tentar garantir que o programa em questão está funcionando como o esperado.

Este capítulo explora o conceito de testes para programas concorrentes, apresentando sua importância, e suas peculiaridades, comparativamente com as abordagens de testes em sistemas sequenciais. Além disso, apresenta uma análise sobre um levantamento de técnicas propostas para testes de sistemas concorrentes, passando por critérios de classificação destas técnicas, e efetuando uma filtragem das ferramentas levantadas, visando separar ferramentas a serem estudadas com mais profundidade.

3.1 DIFICULDADES PARA TESTAR PROGRAMAS CONCORRENTES

Testar programas concorrentes é uma tarefa muito mais complexa do que testar programas sequenciais. Falhas de concorrência são inerentemente não-determinísticas, dado que elas acontecem somente com a ocorrência de intercalações específicas de fluxos, sendo estas intercalações dependentes das condições da execução que não estão sobre controle direto do programa (BIANCHI; MARGARA; PEZZE, 2017).

Este não-determinismo faz com que as falhas de concorrência possam ser extremamente difíceis de identificar e reproduzir, consequentemente complicando o processo de analisar o programa com base em condições existentes de execução. Para contornar esses complicadores e conseguir expor as falhas, abordagens para testagem de sistemas concorrentes precisam experimentar um espaço de possíveis intercalações que pode crescer substancialmente, dependendo da quantidade de fluxos e instruções do programa.

Contudo, esta complexidade de reprodução e identificação de falhas de concorrência reforça ainda mais a importância de testar programas concorrentes, objetivando desenvolver sistemas concorrentes confiáveis, pois estas falhas podem facilmente passar despercebidas no processo de desenvolvimento, e causarem algum problema significativo no sistema.

Este não-determinismo também dificulta o aprendizado dos conceitos de concorrência, e a experimentação prática destes conceitos no ambiente acadêmico. Esta inconsistência dificulta a compreensão do porquê um programa está sendo executado de determinada maneira e produzindo uma saída específica, assim como a identificação de erros lógicos em um programa que aparenta executar corretamente.

Dispor de ferramentas que auxiliem na realização de testes de programas concorrentes ajuda no ensino do paradigma concorrente, permitindo que os estudantes tenham maior entendimento da interação entre fluxos executando seus programas, e identifiquem mani-

festações de problemas de sincronização, e suas possíveis causas. Deste forma, a barreira do não-determinismo é reduzida, simplicando a compreensão do fluxo do programa.

3.2 CLASSIFICAÇÃO DAS TÉCNICAS DE TESTE DE PROGRAMAS CONCORRENTES

Ao longo dos anos, diversas abordagens foram propostas e desenvolvidas com objetivo de testar sistemas concorrentes. Para melhor entendimento e análise destas técnicas, é interessante estabelecer um **esquema de classificação** para categorizá-las. Esta classificação auxilia no entendimento das vantagens e desvantagens de cada abordagem, além de possibilitar uma melhor análise, agrupamento e comparação das técnicas.

Este agrupamento permite um estudo mais focado nas abordagens relevantes para o contexto de testes com foco em aprendizado, pois torna mais fácil a identificação e filtragem de técnicas.

Existem algumas iniciativas para estabelecer critérios e classificar técnicas de testes, com base em inúmeras publicações relacionadas a esta área de estudo. A pesquisa feita em (BRITO et al., 2010) apresenta um estudo sistemático que explora a literatura existente buscando quais critérios de teste (referente a quais predicados devem ser satisfeitos por um conjunto de casos de teste), taxonomia de falhas de concorrência, e ferramentas foram propostos anteriormente, classificando as abordagens em testes estruturais, baseados em erro ou baseados em modelo.

Já no estudo (ARORA; BHATIA; SINGH, 2016), é apresentado um levantamento mais completo de técnicas propostas, e uma categorização intrinsecamente associada a classificações tradicionais de testes de software, como testes estruturais, aleatórios, baseados em busca, baseados em mutação, acessibilidade (*reachability*), baseados em modelo, baseados em fatias, e de métodos formais. Estas duas classificações estão mais associadas à metodologia que as técnicas utilizam para testar os programas. Ou seja, *como* estes testes são realizados.

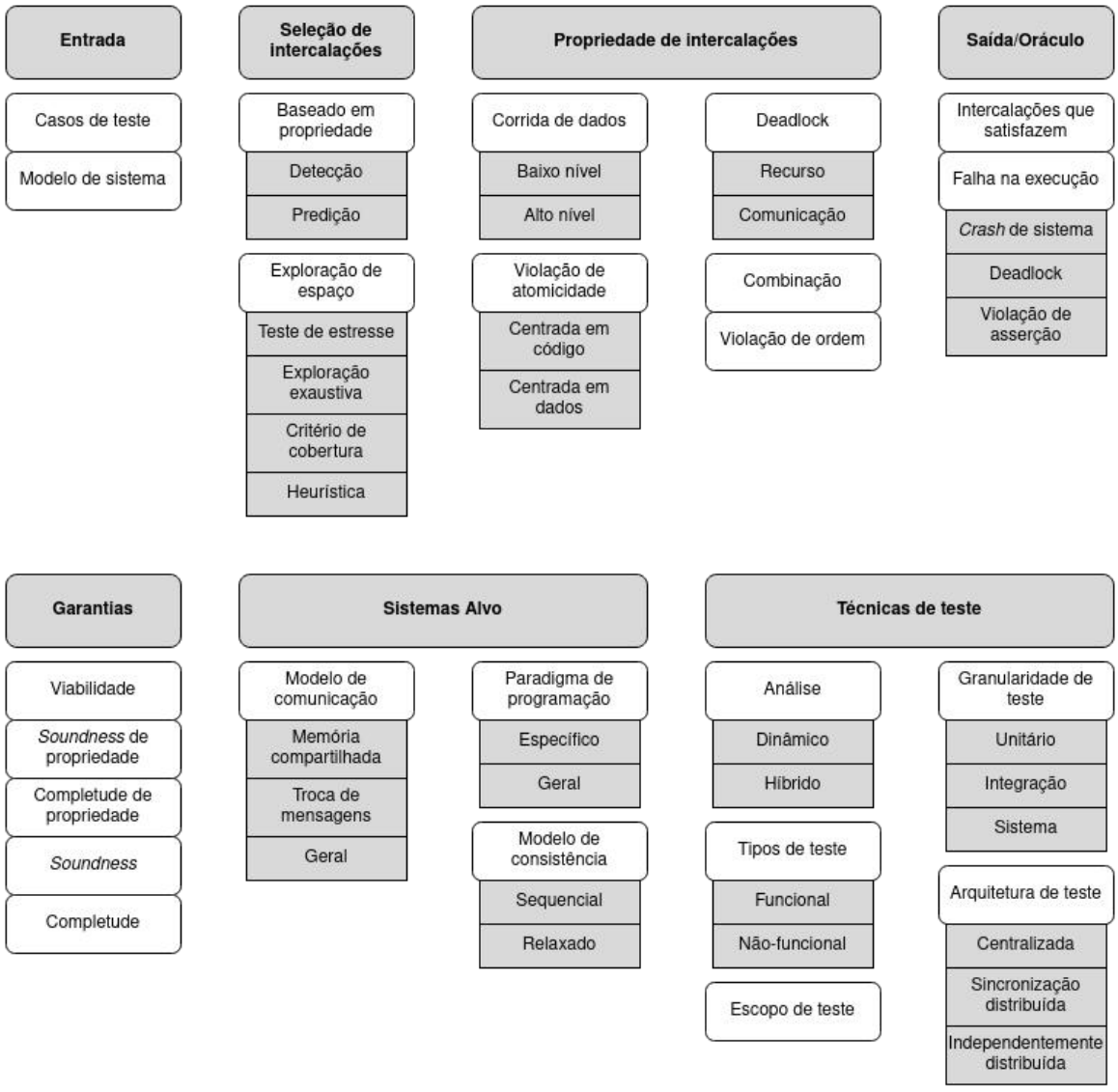
Para os objetivos deste trabalho, a classificação proposta em (BIANCHI; MARGARA; PEZZE, 2017) se mostrou a mais adequada, pois além de fazer um levantamento extremamente amplo e descritivo de abordagens existentes de teste, sua divisão principal é atrelada a possíveis erros lógicos de concorrência sendo avaliados pelas técnicas. Em outras palavras, é proposto uma categorização das técnicas com base no *que* está sendo testado, mas também em *como* aquilo é testado, sendo o primeiro critério a classificação mais relevante. Essa classificação separa as técnicas com base em problemas de concorrência que são estudados em sala de aula, como corrida de dados, *deadlock*, e violação de atomicidade e ordem. Desta forma, é mais fácil identificar ferramentas que facilitem a compreensão e entendimento do aluno sobre o conceito sendo estudado.

A figura 3 apresenta a classificação proposta em (BIANCHI; MARGARA; PEZZE,

2017). Ela consiste em sete critérios de classificação distintos, alguns deles contendo subcritérios, usados para distinguir e agrupar as abordagens:

A **entrada** é relacionada ao que a técnica espera como entrada. A **seleção de intercalações** é associada a qual método a técnica utiliza para selecionar quais intercalações serão exploradas. A **propriedade de intercalações** refere-se a qual erro lógico de concorrência é utilizado por uma técnica para selecionar intercalações que expõem este erro, sendo as opções: corrida de dados, violação de atomicidade, *deadlock*, combinação ou violação de ordem. A **saída** ou **oráculo** é relacionada ao que a técnica oferece como saída. As **garantias** são o que a técnica assegura que irá entregar. O **sistema alvo** refere-se aos aspectos do sistema que a técnica tem como foco. E as **técnicas de teste** são associadas às características dos testes que aquela técnica aborda.

Figura 3 – Diagrama do esquema de classificação das técnicas de teste



No restante desta seção, estes critérios serão descritos em maior detalhe. O critério de técnicas de teste não será abordado, pois não é relevante para o escopo do trabalho.

3.2.1 Entrada

As técnicas de teste podem receber duas categorias de entrada: **casos de teste** e **modelo de sistema**. Essas categorias de entrada não são excludentes, podendo uma complementar a outra. É importante ressaltar que o sistema sendo testado é considerado uma entrada implícita em todas as abordagens (BIANCHI; MARGARA; PEZZE, 2017).

Algumas técnicas precisam receber **casos de teste** como entrada. Um caso de teste é um conjunto de entradas de teste, condições de execução, e resultados esperados, desenvolvido para um objetivo específico, como exercitar um caminho particular do programa, ou verificar conformidade com uma requisição específica (IEEE, 1990).

Outras técnicas requerem também uma entrada que seja um **modelo do sistema**, especificando algum comportamento esperado ou propriedade relevante do sistema, como uso de anotações no código para identificar blocos de código que devem ser atômicos. Para algumas abordagens, um **modelo de sistema** é uma entrada opcional, podendo ajudar a melhorar a acurácia da técnica.

Muitas técnicas geram automaticamente um conjunto finito de casos de teste, a partir do espaço de entrada. Este conjunto gerado é utilizado como entrada para a técnica de teste, e portanto, abordagens que utilizam essa geração automática entram no critério de **casos de teste**.

3.2.2 Seleção de intercalações

As abordagens para selecionar intercalações identificam um subconjunto de entrelaçamentos relevantes de fluxos de execução, a ser explorado durante o processo de testagem. Estas abordagens podem focar no espaço de intercalações como um todo, ou em propriedades de intercalações específicas (BIANCHI; MARGARA; PEZZE, 2017).

Algumas técnicas, chamadas de **baseadas em propriedade**, selecionam determinados entrelaçamentos de fluxos que explorem determinadas propriedades de interesse. Já outras técnicas exploram o espaço de intercalações exaustivamente ou aleatoriamente, e recebem o nome de técnicas de **exploração de espaço**.

3.2.2.1 Baseado em propriedade

Técnicas **baseadas em propriedade** selecionam intercalações de acordo com uma ou mais propriedades de entrelaçamentos. Propriedades de entrelaçamento (ou intercalação) tipicamente identificam padrões de interação entre fluxos de execução que provavelmente irão expor falhas de concorrência (BIANCHI; MARGARA; PEZZE, 2017). Estas abordagens visam identificar as intercalações de fluxos que são mais prováveis de expor estas propriedades de interesse. Algumas técnicas também utilizam a propriedade de interesse para gerar um conjunto de casos de teste que conseguem executar os entrelaçamentos identificados, visando expor a propriedade selecionada.

Normalmente, estas técnicas utilizam alguma abordagem — dinâmica ou híbrida — para analisar um rastreamento de execução, e construir um modelo especulativo do sistema, identificando restrições de sincronização relevantes entre instruções. Uma análise **dinâmica** utiliza somente informações obtidas por meio de execuções do sistema sendo testado. Já a análise **híbrida** utiliza informações dinâmicas, e também estáticas. Informações estáticas são aquelas obtidas sem necessidade de execução do programa, como o próprio código, ou especificações fornecidas.

Algumas abordagens deste tipo utilizam este modelo gerado para detectar se o rastreamento de execução analisado expõe a propriedade de interesse, sendo chamadas de técnicas de **detecção**. Enquanto outras técnicas exploram as informações obtidas no processo de análise para também identificar intercalações de fluxos alternativas (ou seja, que não necessariamente foram executadas) que possam expor a propriedade de interesse, recebendo o nome de técnicas de **predição**.

3.2.2.2 Exploração de espaço

Técnicas de **exploração de espaço** não focam em nenhuma propriedade de intercalação específica, explorando o espaço de intercalações.

Uma das abordagens mais populares é executar o conjunto de testes inúmeras vezes, visando observar diferentes entrelaçamentos. Esta abordagem é chamada de **teste de estresse**. No entanto, mesmo estressando o conjunto de testes, não existe garantia de que uma determinada parcela do espaço de intercalações será observado, pois é possível que apenas alguns entrelaçamentos sejam executados, mesmo que em múltiplas execuções, e a técnica não introduz mecanismos que aumentem as chances de executar entrelaçamentos novos.

É importante ressaltar que a definição de **teste de estresse** desta classificação difere da definição mais tradicional do termo no contexto de engenharia de *software*, que se refere a testes conduzidos para avaliar um sistema ou componente dentro ou além dos limites de seus requisitos especificados (IEEE, 1990).

Algumas técnicas, denominadas **exploração exaustiva**, visam executar todas as intercalações possíveis. Estas técnicas costumam ou limitar o número de instruções e fluxos de execução dos casos de teste de entrada, ou introduzir limites ao espaço de exploração. Estas limitações são necessárias pois, em geral, o espaço a ser explorado pode ser bem extenso. Também é comum elas utilizarem técnicas de redução para evitar execução de entrelaçamentos equivalentes (BIANCHI; MARGARA; PEZZE, 2017).

Já abordagens de **critério de cobertura** identificam as intercalações que devem ser observadas no processo de testagem, em termos de profundidade do espaço explorado. Por exemplo, um critério de cobertura pode determinar que para cada par de instruções i_1 e i_2, pertencentes a diferentes fluxos e que operam na mesma variável, deve existir pela menos um caso de teste que exercite o entrelaçamento em que i_1 ocorre antes de i_2, e

um que exercite o entrelaçamento em que i_2 precede i_1 (BIANCHI; MARGARA; PEZZE, 2017).

Também existem as técnicas baseadas em **heurísticas**, em que determinadas heurísticas estabelecem como o espaço de intercalações deve ser explorado. Algumas técnicas, por exemplo, priorizam entrelaçamentos com base na diversidade deles em relação aos entrelaçamentos que já foram executados (BIANCHI; MARGARA; PEZZE, 2017).

3.2.3 Propriedades

Como explicado em mais detalhes anteriormente na seção 3.2.2.1, técnicas baseadas em propriedade focam em propriedades de intercalações específicas, que têm como alvo erros lógicos mais clássicos, como corrida de dados, violação de atomicidade e deadlocks. Outras abordagens, por sua vez, trabalham em cima de uma combinação destes erros clássicos; ou focam em violações de ordem específicas do programa.

3.2.3.1 Corrida de dados

Como definido na seção 2.5.1, *data race*, ou **corrida de dados**, geralmente ocorre quando duas operações de fluxos de execução distintos acessam um local de memória compartilhado de forma concorrente, sendo pelo menos uma das operações uma escrita. Esse acesso não controlado a um dado pode ser ou não um erro, dependendo da lógica da aplicação. Técnicas de detecção de corrida de dados podem focar em corridas de dados de **baixo** ou **alto nível**, propondo diferentes abordagens para as análises e algoritmos.

Técnicas de detecção de corridas de dados de **baixo nível** trabalham em uma granularidade fina, focando em corridas que ocorrem a nível de locais de memórias individuais. Tais abordagens utilizam alguma metodologia de análise para identificar relações de ordem entre instruções de memória em um determinado rastreamento de dados, ou detectando a ocorrência de uma corrida de dados, ou prevendo a possibilidade de ocorrência de uma corrida em um entrelaçamento alternativo (BIANCHI; MARGARA; PEZZE, 2017). Essas análises podem ser análises *lockset* (ou análise de conjunto de *locks*), análises *happens-before* (ou análise acontece-antes), ou abordagens híbridas.

A análise *lockset* foca em sincronização baseada em *locks*, buscando acessos a dados compartilhados que não estejam protegidos por *locks*. A abordagem descobre estes acessos identificando o conjunto de *locks* de exclusão mútua que são requisitados por *threads* quando elas acessam cada variável compartilhada.

Já a análise *happens-before* captura restrições gerais de ordem entre as operações. A relação de *happened-before* (aconteceu antes) foi formulada por Lamport, visando capturar não só uma relação de ordem temporal (mais facilmente percebida), mas também um relacionamento de causalidade entre pares de eventos (LAMPORT, 2019). Diferentes tipos de de análises *happens-before* se aplicam para diferentes mecanismos de sincronização, e

trazem consigo vários compromissos na relação custo/precisão (BIANCHI; MARGARA; PEZZE, 2017).

Técnicas de detecção de corrida de dados de **alto nível**, por sua vez, envolvem análise de estruturas de dados complexas, como um objeto, checando por comportamentos errôneos causados por diferentes fluxos operando de forma concorrente nestas estruturas, por meio de métodos públicos, por exemplo (BIANCHI; MARGARA; PEZZE, 2017).

3.2.3.2 Violação de atomicidade

Como definido na seção 2.5.2, uma violação de atomicidade é decorrente da violação da seriação desejada entre múltiplas operações de acesso à memória. Ou seja, quando uma região de código deve ser atômica, mas essa atomicidade não é garantida durante a execução (LU et al., 2008).

Violações de atomicidade também podem ocorrer em sistema cuja comunicação é feita por troca de mensagens, quando os blocos de código que processam duas ou mais mensagems que devem ser executados atomicamente são intercalados pelo processamento de alguma outra mensagem (BIANCHI; MARGARA; PEZZE, 2017). Abordagens de detecção de violação de atomicidade podem ser divididas em duas classes principais, baseadas na forma com que definem blocos atômicos: **centradas em código** e **centradas em dados**.

Técnicas **centradas em código** focam em blocos de código que deveriam ser executados de forma atômica de acordo com alguma especificação do sistema, podendo estes blocos serem especificados de forma explícita (com o uso de anotações no código), ou deduzidos implicitamente, com base em hipóteses ou heurísticas. Estas técnicas, então, verificam se a implementação do programa garante estes requisitos de atomicidade.

Técnicas **centradas em dados** utilizam o conceito de seriação (*serializability*) de conjunto atômico, que, por sua vez, é construído em cima da ideia de conjuntos atômicos e unidades de trabalho. Conjuntos atômicos são conjuntos de dados correlacionados por algumas restrições de consistência, e unidades de trabalho são blocos de código utilizados para atualizar variáveis em um conjunto atômico (BIANCHI; MARGARA; PEZZE, 2017). Algumas técnicas de detecção de violação de atomicidade selecionam entrelaçamentos perigosos por meio de padrões que violam a seriação de conjuntos atômicos. Como este tipo de seriação precisa de conjuntos atômicos definidos, essas técnicas dependem de anotações no código, ou de inferências sobre estes conjuntos atômicos por meio de heurísticas ou premissas sobre o paradigma em uso.

3.2.3.3 Deadlock

Como definido anteriormente na seção 2.7, mecanismos de sincronização podem gerar um erro de execução conhecido como *deadlock*, em que uma coleção de fluxos de exe-

cução é bloqueada, esperando por uma condição que jamais será verdadeira (BRYANT; RICHARD, 2003).

Deadlocks podem ocorrer tanto por uma ordem incorreta de acessos a recursos compartilhados, chamado de **deadlock de recurso**, ou por um protocolo de comunicação incorreto entre fluxos de execução, chamado de **deadlock de comunicação** (SINGHAL, 1989).

Em deadlocks de **recurso**, um conjunto de fluxos de execução tentam acessar um recurso em comum, e cada fluxo do conjunto solicita um recurso já alocado por outro fluxo no conjunto. Já em deadlocks de **comunicação**, que ocorrem em sistemas que utilizam troca de mensagens como comunicação, um conjunto de fluxos de execução trocam mensagens, e cada um deles espera por uma mensagem de outro fluxo no mesmo conjunto.

3.2.3.4 Combinação

Algumas técnicas focam em uma combinação das propriedades clássicas de intercalações. Ou seja, abordam mais de uma das propriedades (ou erros lógicos) definidas anteriormente.

3.2.3.5 Violação de ordem

Algumas técnicas de teste abordam violações de ordem de entrelaçamentos específicas do programa ou domínio. Estas técnicas são chamadas de abordagens de **violação de ordem**. Como definido na seção 2.5.3, uma violação de ordem ocorre quando a ordem de execução desejada entre duas operações de acesso à memória é invertida. Por exemplo, o acesso A deve sempre ser executado antes do acesso B, mas esta ordem não é garantida durante a execução (LU et al., 2008).

Muitas abordagens, por exemplo, têm como alvo o domínio de programas concorrentes orientados a objeto, e focam em investigar intercalações que levam a um desreferenciamento de ponteiro nulo, como o ocorrido no código exemplo 4.

3.2.4 Saída/Oráculo

As técnicas de teste podem produzir duas categorias de saídas: **intercalações que satisfazem** e **falhas na execução**.

Abordagens que produzem **intercalações que satisfazem** geram uma saída contendo entrelaçamentos que expõem uma propriedade de interesse, chamados de intercalações que satisfazem uma propriedade. Como nem todos os entrelaçamentos satisfatórios necessariamente levam a uma falha, este tipo de técnica pode resultar em uma saída com falsos positivos. Ou seja, intercalações que expõem a propriedade de interesse, mas não levam a uma falha no programa, como corridas de dados que são consideradas benignas por serem

aceitas pelas regras do programa, não violando nenhum resultado esperado. Já abordagens que produzem **falhas na execução**, efetuam comparações dos resultados produzidos pela execução de uma intercalação com um oráculo, identificando execuções falhas.

Oráculos definem critérios para discriminar entre execuções aceitáveis e execuções falhas. Oŕaculos de ***crash* de sistema** e ***deadlock*** são conhecidos como oráculos implícitos e identificam execuções que levam a *crashes* de sistema e *deadlocks*, respectivamente. Já oráculos de **violação de asserção** exploram afirmações sobre o comportamento apropriado do sistema, baseado em especificações explícitas, ou premissas implícitas (BIANCHI; MARGARA; PEZZE, 2017).

3.2.5 Garantias dos resultados

Técnicas variadas para seleção de intercalações oferecem diferentes níveis de garantia em relação à corretude e precisão de seus resultados.

Uma técnica dá garantia de **viabilidade** dos resultados se ela somente produz intercalações que podem ser observadas em uma execução concreta. Nem todas as abordagens de teste oferecem essa garantia. Algumas técnicas de predição, por exemplo, que produzem entrelaçamentos alternativos aos observados no rastreamento de execuções analisado, podem não ter levado em conta alguma restrição do programa, retornando, assim, intercalações que não correspondem a uma execução viável, e, consequentemente, não garantindo a **viabilidade** de seus resultados (BIANCHI; MARGARA; PEZZE, 2017).

Uma técnica garante ***soundness*** (solidez ou correção, em tradução direta) dos resultados se ela somente produz intercalações que levam a uma violação de oráculo, e garante **completude** dos resultados se ela produz todas as intercalações que são viáveis (que podem ser observadas em uma execução concreta) e levam a uma violação de oráculo.

Existem também dois conceitos que somente se aplicam a técnicas **baseadas em propriedade**: ***soundness* de propriedade** e **completude de propriedade**. Uma técnica dá garantia de ***soundness* de propriedade** se ela identifica somente intercalações viáveis que exibem a propriedade de interesse, para os casos de testes considerados. A garantia de **completude de propriedade**, por sua vez, se dá quando a técnica identifica todas as intercalações viáveis que exibem a propriedade de interesse, para os casos de testes considerados.

Os conceitos de ***soundness*** e **completude** estão diretamente relacionadas à precisão de uma técnica em detectar falhas de concorrência, de modo geral. Enquanto que ***soundness* de propriedade** e **completude de propriedade** descrevem a precisão de uma técnica em detectar entrelaçamentos que expõem uma determinada propriedade.

De modo geral, o tipo de relação de ordem e de análise utilizada em muitas técnicas **baseadas em propriedade** — como análise *happens-before* — pode incorporar aproximações que afetam tanto *soundness* como completude.

3.2.6 Sistema Alvo

A maioria das técnicas tem como alvo alguns tipos específicos de sistemas concorrentes. Esta especificação é caracterizada por três elementos do sistema a ser testado: o **modelo de comunicação**, o **paradigma de programação** e o **modelo de consistência**.

3.2.6.1 Modelo de comunicação

O modelo de comunicação descreve a forma com que os fluxos de execução interagem entre si naquele sistema, e inclui modelos de **memória compartilhada** e modelos de **troca de mensagens**, descritos em maior detalhe na seção 2.4. Existem também técnicas classificadas como **gerais**, que independem do paradigma de comunicação.

Em sistemas que utilizam comunicação por memória compartilhada, os fluxos de execução interagem entre si por meio de acessos a uma memória em comum. Em sistemas que usam comunicação por **troca de mensagem**, os fluxos de execução interagem entre si trocando mensagens por meio de um canal de comunicação.

3.2.6.2 Paradigma de programação

Paradigmas de programação são formas de classificar linguagens de programação com base em seus recursos. Esta classificação não é necessariamente exclusiva. Ou seja, linguagens podem ser classificadas como sendo multiparadigmas. Alguns paradigmas se preocupam com a organização do código, como agrupá-lo em unidades juntamente com seu estado (como o padrão de orientação a objeto). Enquanto outras focam nas implicações do modelo de execução da linguagem, como a sequência de operações ser algo determinado pelo modelo (como ocorre no padrão imperativo). Cada paradigma comporta um conjunto de conceitos que faz com que ele se adeque melhor a um tipo de problema (ROY et al., 2009).

Algumas técnicas não fazem qualquer suposição sobre o paradigma de programação adotado, trabalhando com sistemas **gerais**. Outras técnicas, por sua vez, focam em paradigmas de programação **específicos**, algumas vezes identificados pelos mecanismos de sincronização alvo. Algumas abordagens, por exemplo, exploram propriedades específicas do paradigma de orientação a objeto, como encapsulamento de estado. Já outras técnicas podem apenas considerar falhas decorrentes do uso de algum mecanismo específico de sincronização, como *deadlocks* causados por uso incorreto de uma sincronização baseada em *locks* (BIANCHI; MARGARA; PEZZE, 2017).

3.2.6.3 Modelo de consistência

O modelo de consistência (ou modelo de memória) determina uma convenção entre o sistema e o desenvolvedor, na qual o sistema assegura que funcionará de forma apropriada,

mantendo um grau acordado de consistência de memória, caso o programa desenvolvido siga determinadas regras. Algumas técnicas assumem um modelo de consistência **sequencial**, enquanto outras presumem um modelo de consistência **relaxada**.

A consistência **sequencial** assegura que todos os fluxos de execução do sistema observem a mesma ordem de instruções, e que esta ordem conserva a ordem de instruções definida nos fluxos individuais. Ou seja, se dentro de um determinado fluxo de execução uma operação precede outra, na sequência completa de operações do programa, esta ordem de precedência se mantém.

É possível, assim, modelar com um histórico as intercalações de instruções de múltiplos fluxos de execução em uma execução concreta do programa. Este histórico seria uma sequência ordenada de instruções destes diferentes fluxos (BIANCHI; MARGARA; PEZZE, 2017). Consistência **sequencial** ainda pode produzir resultados não-determinísticos, dado que a sucessão de operações sequenciais entre os fluxos de execução podem ser diferentes durante diferentes execuções do programa.

Modelos de consistência **relaxada**, por sua vez, são simplesmente modelos definidos a partir de um relaxamento das restrições da consistência sequencial (MANKIN, 2007). Os históricos de sistemas **relaxados** podem violar as propriedades que caracterizam o histórico de sistemas sequencialmente consistentes, e, consequentemente, levar a novas possíveis falhas de concorrência que não ocorrem sob a hipótese de consistência sequencial (BIANCHI; MARGARA; PEZZE, 2017).

3.3 FILTROS

Com a definição de uma classificação de técnicas de testes para programas concorrentes, é possível agora definir critérios de inclusão e exclusão para filtrar a listagem de ferramentas levantadas na pesquisa (BIANCHI; MARGARA; PEZZE, 2017), selecionando aquelas mais relevantes para serem estudadas e testadas em maior detalhamento, considerando sempre o propósito de aplicação das técnicas em ambiente acadêmico.

3.3.1 Primeiro crivo - Finalidade acadêmica

Para um crivo inicial, determinou-se como principal critério de inclusão as propriedades de intercalação sendo abordadas, uma vez que o objetivo do trabalho é auxiliar a identificação de problemas comuns de concorrência, e suas possíveis causas. Este processo de detecção de falhas e reconhecimento de suas causas tem intuito de ajudar no aprendizado destes conceitos e no entendimento das interações entre os fluxos do programa, fazendo uma exploração mais proposital do espaço de entrelaçamentos. Portanto, somente técnicas baseadas em propriedade são incluídas no crivo, sendo exploração de espaço um critério excludente.

Além disso, considera-se somente abordagens voltadas para modelos com comunicação por memória compartilhada, pois este é o paradigma normalmente abordado em disciplinas de introdução ao estudo de concorrência. Consequentemente, abordagens para modelos de troca de mensagem ou geral são excluídas.

Dentre as propriedades de intercalação sendo testadas pelas técnicas, foram selecionadas como critérios de inclusão aquelas que estão presentes no contexto de uma disciplina introdutória de concorrência, como melhor descritas nas seções 2.5 e 2.7, sendo elas: **corrida de dados de baixo nível**, **violação de atomicidade centrada em código**, **deadlock de recurso**, **combinação** e **violação de ordem**.

Deadlock de comunicação não foi considerado pois a seleção está restrita a modelos de comunicação por memória compartilhada. Já **corrida de dados de alto nível** e **violação de atomicidade centrada em dados** não foram consideradas pois estrapolam o escopo de uma disciplina introdutória de concorrência.

Além disto, foram excluídas técnicas com paradigmas de programação específicos, como abordagens para aplicações *Android* ou plataformas *web*, ou para programas baseados em eventos, ou que utilizam *fork-join*. Também foram excluídas técnicas voltadas para paradigma de sincronização por barreira, pois este mecanismo não faz parte do escopo do trabalho.

Foram estabelecidos alguns critérios desejáveis, que tornam uma abordagem mais interessante para filtros posteriores, como garantia de viabilidade e *soundness* de propriedade (para evitar falsos positivos), e um sistema alvo com modelo de consistêcnia sequencial. Ademais, é desejável que técnicas que possuem **casos de teste** como entrada trabalhem em um modelo de geração destes casos de teste, com intuito de facilitar o processo de aplicação das abordagens no contexto acadêmico, evitando que o aluno tenha que escrever os próprios casos de teste, muitas vezes um conhecimento que ele ainda não obteve no momento do aprendizado de concorrência. O restante dos critérios são considerados indiferentes.

Esta primeira filtragem permitiu uma redução das ferramentas a serem analisadas, de 92 para 64, produzindo um levantamento que engloba apenas técnicas que abordam os desafios e paradigmas de interesse.

3.3.2 Segundo crivo - Adequação das técnicas

Uma vez que a saída resultante do primeiro filtro é muito extensa e ainda contém abordagens que não se enquadram no objetivos de uso didático, é necessário realizar filtragens adicionais.

Trabalhando com a listagem resultante do primeiro crivo, foi realizada uma primeira leitura da visão geral destas técnicas, descrita em (BIANCHI; MARGARA; PEZZE, 2017), a fim de identificar abordagens pioneiras no estudo da área, ou que propunham abordagens diferentes (como geração de testes) ou com melhoria de precisão nos resultados, e não

somente visando ganho de tempo de execução de abordagens já existentes. Este crivo foi feito para estudar mais a fundo as propostas mais centrais à pesquisa de testes para aquele tipo de propriedade. Com base nos resultados desta primeira análise, avaliou-se a disponibilidade dos artigos de referência das técnicas levantadas, eliminando do escopo as abordagens cuja descrição não estava disponível para acesso público.

Quadro 1 – Técnicas resultantes do primeiro e segundo crivos

TÉCNICA	**PROPRIEDADE**	**REFERÊNCIA**
Shacham et al.	Corrida de Dados - *Lockset*	(SHACHAM; SAGIV; SCHUSTER, 2007)
FastTrack	Corrida de Dados - *Happens-before*	(FLANAGAN; FREUND, 2009)
DrFinder	Corrida de Dados - *Happens-before*	(CAI; CAO, 2015)
RVPredict	Corrida de Dados - *Happens-before*	(HUANG; MEREDITH; ROSU, 2014)
Choi et al.	Corrida de Dados - Híbrido	(CHOI et al., 2002)
RaceFuzzer	Corrida de Dados - Híbrido	(SEN, 2008)
Narada	Corrida de Dados - Híbrido	(SAMAK; RAMANATHAN; JAGANNATHAN, 2015)
Atomizer	Violação de Atomicidade	(FLANAGAN; FREUND, 2004)
AtomFuzzer	Violação de Atomicidade	(PARK; SEN, 2008)
Best	Violação de Atomicidade	(GANAI, 2011)
Intruder	Violação de Atomicidade	(SAMAK; RAMANATHAN, 2015)
Chen and MacDonald	Combinação	(CHEN; MACDONALD, 2007)
PECAN	Combinação	(HUANG; ZHANG, 2011)
GoodLock	Deadlock	(HAVELUND, 2000)
DeadlockFuzzer	Deadlock	(JOSHI et al., 2009)
WOLF	Deadlock	(SAMAK; RAMANATHAN, 2014c)
Omen	Deadlock	(SAMAK; RAMANATHAN, 2014a)
JPredictor	Violação de Ordem	(CHEN; SERBANUTA; ROSU, 2008)
GPredictor	Violação de Ordem	(HUANG; LUO; ROSU, 2015)

Em uma análise preliminar das ferramentas apontadas nos artigos de referência, observou-se uma maior ocorrência de ferramentas voltadas para programas Java. Por conta disto, optou-se por focar apenas em implementações compatíveis com a linguagem Java, para que fosse possível usar os mesmos códigos em todos os testes com as ferramentas.

Neste segundo processo de filtragem, houve uma redução para um total de 19 ferramentas, resultando na listagem apresentada no quadro 1.

3.3.3 Terceiro crivo - Disponibilidade

Uma vez selecionadas as abordagens que se enquadram melhor para a proposta do trabalho, foi necessário identificar quais das técnicas resultantes do segundo filtro possuem implementações disponíveis para *download* e uso. Este passo é essencial, pois o foco deste trabalho não é implementar as abordagens estudadas, e sim analisar o material já existente, no escopo teórico e prático, possibilitando o entendimento e utilização em sala de aula destas ferramentas de teste, para que os próprios alunos consigam testar seus programas, auxiliando no aprendizado dos conceitos de concorrência.

Para encontrar as implementações foi realizada uma análise dos artigos, buscando *links* para repositórios ou *websites* referentes à implementação descrita, e pesquisas em serviços de busca utilizando as palavras-chave *git*, *github*, *gitlab*, *bitbucket*, *repository*, *download*, *concurrency tool*, acompanhadas do nome da técnica ou de seus pesquisadores, a fim de encontrar repositórios *online* ou páginas relacionadas à abordagem buscada. Foram encontradas 10 ferramentas com implementação disponível, listadas no quadro 2.

Quadro 2 – Técnicas com implementação disponível

TÉCNICA	**PROPRIEDADE**
FastTrack	Corrida de Dados
RVPredict	Corrida de Dados
RaceFuzzer	Corrida de Dados
Narada	Corrida de Dados
Atomizer	Violação de Atomicidade
AtomFuzzer	Violação de Atomicidade
Intruder	Violação de Atomicidade
PECAN	Combinação
DeadlockFuzzer	Deadlock
Omen	Deadlock

3.3.4 Quarto crivo - Usabilidade

Trabalhando com a lista resultante de ferramentas disponíveis, foi realizada uma avaliação de cada uma das ferramentas, em um contexto prático, efetuando sua instalação e executando testes básicos, para averiguar sua facilidade e praticidade de uso, objetivando selecionar as abordagens mais apropriadas para serem estudadas e testadas em maior detalhamento.

Além desta fase de estudo servir para determinar quais ferramentas são simples para serem instaladas e executadas por alunos, serve para confirmar qual tipo de entrada o pro-

grama recebe. Como já mencionado na seção 3.3.1, é importante que o aluno não precise escrever casos de teste, para que o uso das ferramentas seja acessível para estudantes que ainda não aprenderam sobre testes de *software*. Portanto, é decisivo que a entrada das ferramentas seja algo simplificado, como o arquivo compilado do programa, ou o código fonte.

Também é importante avaliar a saída das execuções da ferramenta, para averiguar se seus resultados são compreensíveis, e analisar o quanto o aluno precisa entender sobre a ferramenta para identificar qual erro está sendo apontado, e onde este erro se encontra. No restante desta seção, é descrito o resultado deste estudo de usabilidade das técnicas.

O RaceFuzzer (SEN, 2008) é uma abordagem preditiva para identificar **corrida de dados**, por meio de análise híbrida. A técnica utiliza uma ferramenta de análise dinâmica híbrida já existente para obter potenciais corridas de dados, consumindo estas informações para controlar um escalonador aleatório de *threads*. Por meio deste escalonador, o RaceFuzzer busca criar corridas reais com alta probabilidade, usando *crashes* ou asserções para diferenciar entre corridas benignas ou problemáticas.

DeadlockFuzzer (JOSHI et al., 2009) é uma técnica preditiva para identificar potenciais ***deadlocks*** em programas *multithread*, por meio de uma análise de duas etapas. Na primeira etapa, a ferramenta utiliza a análise da abordagem Goodlock para identificar *deadlocks* potenciais a partir da execução do programa, capturando o padrão de *locking* dos fluxos de execução (HAVELUND, 2000). Na sua segunda etapa, similarmente ao RaceFuzzer, ela controla um escalonador aleatório para expor estes *deadlocks* com alta probabilidade. Esta abordagem é limitada a considerar somente falhas causadas por sincronização por *locks*.

O AtomFuzzer (PARK; SEN, 2008) é uma abordagem preditiva para encontrar **violações de atomicidade** por meio de análise dinâmica aleatória. Ela gera intercalações aleatoriamente, explorando as relações de ordem determinadas na análise *happens-before*, feita a partir de pares de fluxos que utilizam um único *lock* para garantir a atomicidade de um bloco. O AtomFuzzer, assim como o RaceFuzzer e DeadlockFuzzer, modifica o escalonador de *threads* Java para criar violações com alta probabilidade, e permite facilmente dar um *replay* na execução que gera a violação.

A implementação das técnicas RaceFuzzer, DeadlockFuzzer e AtomFuzzer se encontram em uma mesma ferramenta, chamada CalFuzzer (SEN et al., 2014). A última atualização do repositório foi feita há sete anos, e sua instalação exige uma JDK mais antiga. Na documentação, é descrito como requisito a JDK 5, mas é possível executar os testes utilizando JDK 7. Ao tentar utilizar versões mais atualizadas da JDK, a ferramenta apresentou problemas. Além da JDK, é necessária a instalação de uma versão compatível do Ant (FOUNDATION, 2021a). Esta instalação do Ant se mostrou comum a maioria das ferramentas testadas.

O CalFuzzer tem como entrada o código fonte do programa. Para testar um pro-

grama, é necessária a criação de *targets* no arquivo `run.xml` que apontem para o código a ser testado, e indiquem qual modalidade de análise deve ser realizada (`race-analysis`, `deadlock-analysis` ou `atomfuzzer-analysis`), além de outras propriedades. Isto dificulta o processo de testagem, pois exige algum nível de entendimento da organização e estrutura do projeto fonte do CalFuzzer. Além disso, a saída correspondente à testagem de violação de atomicidade não é muito legível, sendo difícil identificar a causa desta violação. Por estes motivos, estas três técnicas foram cortadas.

A ferramenta Narada (SAMAK; RAMANATHAN; JAGANNATHAN, 2015) utiliza análise híbrida (*lockset* combinado com *happens-before*) para automatizar a geração de testes *multithread* para detectar **corridas de dados**. Em seguida utiliza o RaceFuzzer para executar e analisar estes testes.

Esta ferramenta tem como entrada a implementação de uma biblioteca, e um conjunto de testes sequenciais. Esta necessidade de casos de teste como entrada foge da proposta deste trabalho, pois o objetivo é que o aluno não precise elaborar testes que utilizem seu programa. Além disso, o foco desta ferramenta é testagem de bibliotecas, e, portanto, exige que o programa sendo testado esteja estruturado como uma biblioteca, com métodos invocáveis, limitando a organização do código do aluno.

O Intruder (SAMAK; RAMANATHAN, 2015) é uma ferramenta similar ao Narada. Sua proposta é a geração automática de casos de teste para detectar **violações de atomicidade**, por meio de mapeamentos de acquisições e liberações de locks, e de acessos à memória, analisando estas informações para inferir possíveis violações. A ferramenta trabalha somente com a geração de testes, dependendo de outras abordagens para receber estes testes como entrada e detectar as violações. Portanto, por si só, o Intruder não oferece nenhum critério de garantia ou uma saída indicando as violações. Esta dependência de uma segunda camada para executar os testes faz com que esta ferramenta não seja desejável para os objetivos deste trabalho. Além disto, analogamento ao Narada, a ferramenta tem como entrada a implementação da biblioteca, e casos de testes sequenciais, implicando nas mesmas limitações inconvenientes já descritas anteriormente.

O PECAN (HUANG; ZHANG, 2011) é uma abordagem preditiva para identificar anomalias gerais de acesso em programas concorrentes, como **corrida de dados e violação de atomicidade**. Ou seja, é uma abordagem voltada para detectar uma **combinação** de erros lógicos, utilizando análise *happens-before* para determinar relações de ordem entre operações, e criando um *schedule* para os entrelaçamentos. Seu objetivo, além de apontar a falha, e seu local, é reduzir a quantidade de falsos positivos, e apresentar um *bug hatching clip* instruindo o programa a executar as anomalias previstas. A ferramenta possui instalação fácil, mas também exige um cuidado em relação à compatibilidade da versão da JDK 7 com a versão do `Ant`. o PECAN trabalha com o código compilado como entrada do programa, sendo bem adaptável para testar programas novos. Porém, nas execuções realizadas, ele não identificou algumas violações de atomicidade que foram identificadas

por outras ferramentas exploradas. Por conta disto, considerou-se que seria mais interessante focar em outras técnicas que apresentaram resultados mais precisos para o contexto sendo abordado.

OMEN (SAMAK; RAMANATHAN, 2014a) é uma abordagem semelhante ao Intruder, gerando casos de testes concorrentes a partir de análises que geram relações de dependências de *lock*. A ferramenta encontrada foi a OMEN+, que é uma versão melhorada da técnica OMEN, que além de gerar os casos de testes, utiliza a técnica de detecção WOLF (SAMAK; RAMANATHAN, 2014c) para detectar automaticamente *deadlocks* reais por meio de análises dos rastreamentos de execução destes testes (SAMAK; RAMANATHAN, 2014b).

O OMEN+ recebe como entrada uma biblioteca *multithread*, e, opcionalmente, casos de testes sequenciais. Sua instalação é simples, também dependendo da JDK 7, e é possível adicionar novos casos para teste adaptando os *shell scripts* providos no repositório, ou criando seus próprios seguindo o mesmo padrão. Apesar de não ser uma adaptação tão complexa quanto a necessária para o CalFuzzer, ainda é um aspecto indesejável, pois é necessário um entendimento de como a ferramenta está estruturada. Além disso, analogamente ao Narada e ao Intruder, o foco desta ferramenta é testagem de bibliotecas, e, portanto, exige que o programa sendo testado esteja estruturado como uma biblioteca, implicando nas mesmas limitações de estruturação do programa já mencionadas.

FastTrack (FLANAGAN; FREUND, 2009) é uma técnica de detecção que visa identificar **corrida de dados** por meio de análise *happens-before*. É proposto uma abordagem mais flexível e leve para representação de *clocks* vetoriais para expressar as informações de relação de ordem. A ferramenta encontrada é uma implementação de uma atualização do FastTrack, o FastTrack2 (FLANAGAN; FREUND, 2017a).

O Atomizer (FLANAGAN; FREUND, 2004) é o trabalho base para técnicas que visam identificar **violações de atomicidade**, apresentando uma análise dinâmica que explora ideias de análises *lockset*, e teoria de redução, para detectar entrelaçamentos não-serializáveis.

As implementações da versão melhorada do FastTrack, e do Atomizer estão presentes na ferramenta RoadRunner (FLANAGAN; FREUND, 2010), cuja última atualização tem 5 anos. O RoadRunner possui fácil instalação, sendo compatível com uma versão de JDK mais atual (JDK 8) e a versão correspondente de Ant, facilitando a configuração da ferramenta. A execução do processo de testagem é extremamente simples, indicando por uma flag qual a ferramenta específica que deve ser utilizada (Atomizer ou FastTrack), e o programa recebe como entrada o arquivo compilado.

Neste primeiro processo de testagem, foi averiguado que a implementação do FastTrack lida bem com *loops*, e possui uma saída que permite a identificação do local da corrida, mas com uma descrição um pouco visualmente poluída. Já a implementação do Atomizer possui uma saída mais clara, sinalizando o tipo de violação ocorrido. Além disso, apesar de

não ter detectado uma violação específica em um programa teste, obteve resultados mais completos do que o PECAN. Com base nestes resultados, determinou-se que o FastTrack e Atomizer são ferramentas desejáveis para serem estudadas com maior detalhamento.

RVPredict (HUANG; MEREDITH; ROSU, 2014) é uma abordagem para detectar **corridas de dados** que utiliza informações de fluxo de controle para melhorar a acurácia de análise *happens-before*, formulando detecção de corrida como um problema de resolução de restrições. A implementação do RVPredict é uma das mais atuais dentre todas as testadas, exigindo JDK 8 ou superior. Sua instalação é muito simples, precisando apenas executar uma linha de comando. Além de uma instalação simples, ela também permite integração com IDEs, como Eclipse ou IntelliJ, facilitando ainda mais o processo de uso. O RVPredict também possui uma variante voltada para códigos escritos em C, mostrando-se muito adaptável e útil para utilização em sala de aula. Sua entrada pode ser tanto um `jar` quando o arquivo compilado do programa. Possui uma boa documentação, descrevendo problemas comuns encontrados e apresentando muitos exemplos de uso. Quanto a sua execução, não lida bem com *loops* grandes, demorando muito para executar em programas que apresentavam laços com mais de 25 iterações (possivelmente por ser uma ferramenta de predição), mas possui uma saída bem legível e compreensível. Com base nesta análise, determinou-se que o RVPredict é uma ferramenta desejável para ser melhor analisada.

Após estas análises da usabilidade das técnicas, determinou-se, então, uma seleção final de 3 abordagens/ferramentas: **FastTrack**, **RVPredict** e **Atomizer**, eliminando-se um total de 7 técnicas. No próximo capítulo, estas ferramentas serão descritas e testadas de forma mais elaborada, apresentando suas características e análises, método de instalação e execução, e os resultados obtidos em seu estudo.

4 FERRAMENTAS ANALISADAS

Este capítulo descreve em maior detalhe as ferramentas selecionadas no processo de filtragem — FastTrack, RVPredict e Atomizer — realizando um estudo mais profundo destas abordagens. É feita uma análise de suas metodologias para detectar corridas de dados ou violação de atomicidade, assim como uma demonstração do processo de instalação e uso destas ferramentas. Também são realizados testes com programas exemplo para estudar os resultados obtidos, averiguando a acurácia das implementações, identificando suas limitações e dificuldades, e analisando sua saída com intuito de deixá-la mais compreensível.

Além disso, para passar uma melhor visão do estado atual do estudo desta área, também são apresentadas ferramentas de testes concorrentes mais acessíveis, já existentes em alguns compiladores e *toolkits*, listando suas propostas, linguagens alvo e qual propriedade analisam, de forma resumida.

4.1 FASTTRACK

O FastTrack é uma ferramenta de teste que tem como objetivo encontrar **corrida de dados** — definidas em maior detalhamento na seção 2.5.1 — por meio de análise dinâmica *happens-before*. A técnica proposta é voltada para sistemas com paradigma de programação geral, e com consistência sequencial. A abordagem garante a viabilidade dos resultados, dado que é uma técnica de dedução, apenas explorando entrelaçamentos de execuções reais do programa. A ferramenta FastTrack2 é uma atualização do FastTrack, com alguns ajustes nas definições das regras que determinam operações válidas, e operações que causam corrida de dados, mas a abordagem geral se mantém a mesma (FLANAGAN; FREUND, 2017a).

4.1.1 Funcionamento da ferramenta

O FastTrack observa uma execução real do programa, não escolhendo as intercalações executadas, apenas analisa o rastreamento dessa execução. Isso significa que, dada uma determinada execução, o FastTrack é capaz de deduzir se nesta execução ocorreu algum evento de corrida de dados. Essa dedução é realizada usando comparações de relógio para verificar as relações de precedência (happens-before).

A técnica propõe uma abordagem leve e adaptável para representar essas relações de precedência *happens-before*, se baseando na ideia de que na maioria dos casos, a generalidade completa de relógios vetoriais não é realmente necessária, e provendo uma alternativa com caminhos resolvidos em tempo e espaço constante, sem perda de precisão (FLANAGAN; FREUND, 2009). Um relógio vetorial é uma estrutura de dados utilizada para

analisar uma ordenação parcial de eventos em fluxos concorrentes e que permite detectar relações de causalidade entre eventos destes fluxos. Uma das características deste relógio lógico é que utiliza um vetor de inteiros para representar o *timestamp* de um evento (MATTERN et al., 1988). O tamanho deste vetor será a quantidade de processos ou *threads* do programa concorrente.

O FastTrack prioriza precisão, uma vez que alarmes falsos podem ser muito nocivas ao processo de depuração, pois novas falhas, como *deadlocks*, podem ser introduzidas ao tentar consertar uma corrida de dados inexistente, mas que tenha sido reportada. Além disto, para estudantes iniciantes na área de concorrência, o uso de ferramentas precisas é mais apropriado, para evitar confusões relacionadas a falsos positivos e resultados errôneos.

4.1.1.1 Happens-before

O conceito de *happens-before*, apresentado na seção 3.2.3.1, é adaptado para incluir as relações de ordem de aquisição e liberação de *locks* e entre operações de *fork* e *join* de threads. Portanto, as relações *happens-before* podem ser definidas como:

- Se os eventos a e b ocorrem na mesma *thread*, então $a \rightarrow b$, se a ocorrência do evento a preceder a ocorrência do evento b.
- Se o evento a é a aquisição de um *lock*, e o evento b é a liberação do mesmo *lock* pela mesma *thread*, $a \rightarrow b$
- Se o evento a é a liberação de um *lock*, e o evento b é a aquisição do mesmo *lock*, $a \rightarrow b$
- Se o evento a é um *fork* (criação) de uma *thread* t, e o evento b é uma operação realizada pela *thread* t, $a \rightarrow b$
- Se o evento a é uma operação de uma *thread* t, e o evento b é um *join* da *thread* t, $a \rightarrow b$

Se dois eventos em um rastreamento de execução não estão relacionados pela relação de precedência *happens-before*, então os dois eventos são ditos como concorrentes. Ou seja, nem $a \rightarrow b$ nem $b \rightarrow a$.

A partir desta definição, é possível definir que uma corrida de dados ocorre quando um rastreamento de execução possui dois eventos de acesso concorrente, e pelo menos um deles é uma operação de escrita. Ou seja, cada condição de corrida pode ser classificada como: **corrida leitura-escrita**, em que uma operação de leitura é concorrente com uma operação de escrita na mesma variável; **corrida escrita-leitura**, em que uma escrita é concorrente com uma leitura; ou **corrida escrita-escrita**, que envolve duas escritas concorrentes na mesma variável (FLANAGAN; FREUND, 2009).

4.1.1.2 Identificando corridas

Tradicionalmente, a relação *happens-before* é representada por meio de relógios vetoriais. Analogamente às abordagens que o precedem, no algoritmo do FastTrack, cada *thread* t mantém um relógio vetorial C_t de modo que, para qualquer *thread* a, a entrada $C_t(a)$ armazene o relógio da última operação da *thread* a que ocorreu antes da operação atual da *thread* t. Também existem relógios vetoriais L_m para armazenar informações sobre cada *lock* m, que são atualizados em momentos de aquisição e liberação, de modo a refletir as relações de precedência de operações.

A inovação do FastTrack é a introdução do conceito de *epoch* (período, em tradução direta) para substituir os relógios vetoriais em situações nas quais não é necessário armazenar um vetor contendo os relógios de cada uma das *threads* do programa. Uma *epoch* é um par formado por uma *thread* e um relógio, denotado $t@c$, cujo intuito é guardar somente informações sobre o último evento realizado. Por ser uma estrutura mais simples do que um relógio vetorial, a *epoch* é mais facilmente armazenada, copiada e comparada.

O FastTrack armazena *epochs* de escrita e leitura para as variáveis compartilhadas do programa, salvando a *thread* e o relógio da última operação de escrita e leitura nestas variáveis. Para fins de otimização, a técnica propõe uma abordagem adaptativa que observa o histórico de leitura de cada variável, e utiliza *epochs* o máximo possível, alternando para uma representação por relógio vetorial quando necessário (FLANAGAN; FREUND, 2009).

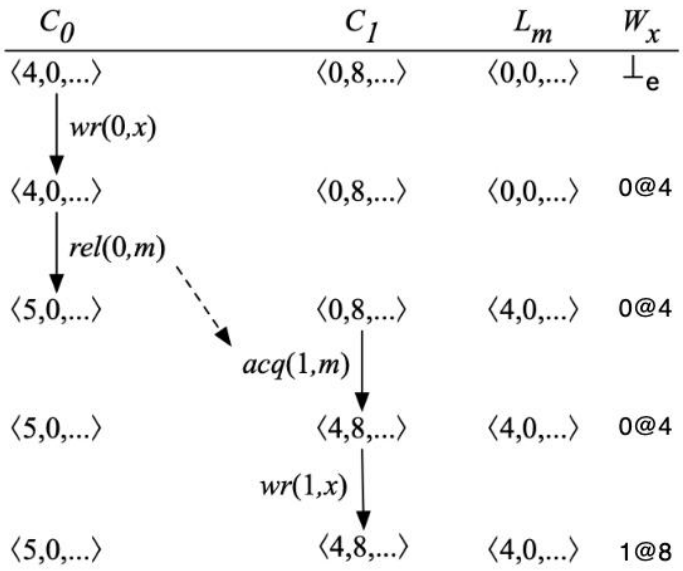
Os relógios vetoriais e as *epochs* são atualizados quando ocorrem operações de aquisição e liberação de *lock*, operações de escrita e leitura, e operações de *fork* ou *join*. Na figura 4 é possível observar estas atualizações em um fluxo de um programa. Por exemplo, a *thread* 0 efetua uma escrita na variável `x`, e a *epoch* de escrita desta variável W_x é atualizada com o identificador da *thread* e seu relógio no momento da operação, 0@4. Nos momentos de liberação e aquisição dos *locks*, o relógio da *thread* operando, e do *lock* correspondente são atualizados, indicando a sincronização.

Com estas representações bem definidas, é possível definir como a ferramenta identifica corridas de dados de forma mais clara.

Para detectar corridas **escrita-escrita** — que envolve duas escritas concorrentes — no momento em que uma *thread* t vai realizar uma escrita em um variável x, o algoritmo compara a *epoch* atual de escrita com o relógio vetorial atual da *thread* t, a fim de verificar que a nova escrita está ocorrendo depois da escrita anterior, respeitando as relações de precedência *happens-before*. Caso o relógio na *epoch* seja superior ao relógio correspondente no vetor, a tentativa de escrita feita por t é concorrente com a escrita anterior, pois implica em uma violação da relação de precedência, e existe uma corrida **escrita-escrita**.

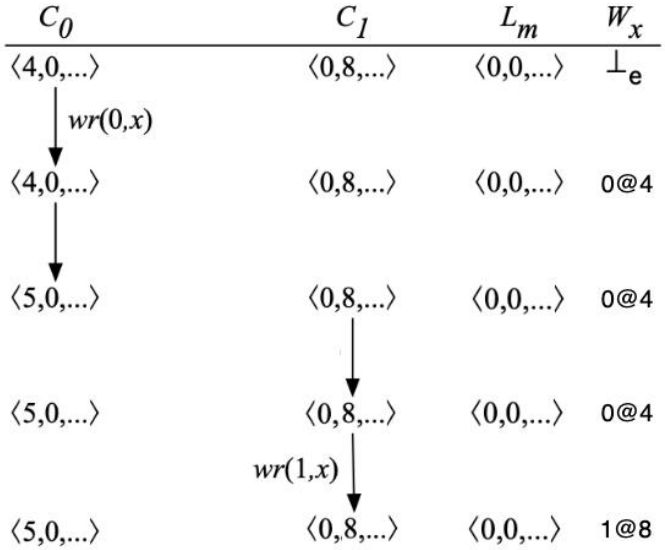
Este fluxo está exemplificado na figura 5. Neste fluxo, a *thread* 0 efetua uma escrita na variável `x`, atualizando a *epoch* de escrita com seu identificador e relógio do momento da

Figura 4 – Relógios Vetoriais e Epochs no Algoritmo FastTrack (FLANAGAN; FREUND, 2009)



operação, 0@4. Uma *thread* 1, então, vai efetuar uma escrita nesta mesma variável. Neste momento, o FastTrack irá comparar a *epoch* de escrita 0@4 com o relógio correspondente no relógio vetorial da *thread* 1. Ou seja, compara o valor de relógio 4 com o campo de C_1 correspondente ao relógio da *thread* 0, que no caso é $C_1[0] = 0$. O algoritmo então verifica que o relógio na *epoch* é superior, e esta tentativa de escrita pela *thread* 1 é concorrente com a escrita realizada pela *thread* 0. Isto ocorre pois não houve nenhuma sincronização entre estas duas *threads*, não permitindo uma atualização local dos relógios referentes aos outros fluxos.

Figura 5 – Relógios Vetoriais e Epochs quando existe uma corrida de dados (FLANAGAN; FREUND, 2009)



Já corridas **escrita-leitura**, em que uma escrita anterior é concorrente com uma leitura, o algoritmo faz uma checagem análoga à apresentada acima, para a leitura em uma variável x realizada por uma *thread* t, é verificado que a leitura acontece depois da última escrita, por meio da mesma comparação da *epoch* atual de escrita da variável x com o relógio vetorial da *thread* t. Caso o relógio na *epoch* seja superior ao correspondente do vetor, esta leitura viola as relações de happens-before e é concorrente à escrita, e existe uma corrida **escrita-leitura**.

Para detectar corridas **leitura-escrita**, em que uma leitura anterior é concorrente com uma escrita, o algoritmo pode trabalhar tanto com relógios vetoriais quanto *epoch* de leitura, dependendo da ordenação ou não de operações de leitura. Analogamente às checagens anteriores, o FastTrack compara o relógio vetorial (ou a *epoch*) de leitura com o relógio vetorial da *thread* realizando a escrita. Caso a comparação falhe, existe uma corrida **leitura-escrita**.

O FastTrack 2 otimiza estas checagens, criando regras mais específicas para facilitar o processo do algoritmo. Contudo, a ideia geral da abordagem se mantém a mesma.

4.1.2 Instalação e uso da ferramenta

A implementação encontrada da ferramenta FastTrack está presente no framework RoadRunner. RoadRunner é um framework de análise dinâmica escrito em Java, feito para facilitar prototipagem e experimentação com análises dinâmicas para programas concorrentes desenvolvidos em Java, adicionando código de instrumentação no *bytecode* do programa em tempo de carregamento (FLANAGAN; FREUND, 2010).

O RoadRunner tem dois propósitos básicos: facilitar a implementação de outras ferramentas de análise, abstraindo grande parte da lógica de baixo nível para a instrumentação; e possibilitar o uso de diversas ferramentas de teste já implementadas, como FastTrack e Atomizer. Para os objetivos deste trabalho, apenas a segunda proposta é explorada.

Para utilizar o RoadRunner, é preciso preparar a máquina, instalar e configurar o framework, e, finalmente, executar o teste (FLANAGAN; FREUND, 2017b). Para preparar a máquina para rodar o framework, é necessário instalar JDK 7 ou 8 (recomenda-se usar a 8), um compilador C++, e Apache Ant. E para instalar e configurar, é preciso seguir o passo a passo descrito no arquivo `INSTALL.txt` presente no repositório do RoadRunner (FLANAGAN; FREUND, 2017b).

É relevante observar alguns detalhes no processo de configuração, como a variável de ambiente `JAVA_HOME`, que precisa apontar para a distribuição correta do Java, pois ela é utilizada no processo de configuração do framework (FLANAGAN; FREUND, 2017b).

Além disso, outro ponto de atenção é o arquivo de configuração `msetup`, que contém todas as montagens de variáveis de ambiente e configurações específicas para uma máquina. Caso seja necessária alguma adaptação, ela deve ser feita neste arquivo antes da chamada de `source msetup`.

Para testar um programa, ele precisa ser adicionado dentro do projeto do RoadRunner, no pacote já existente `test`, ou em um novo pacote. É importante que o código Java sendo adicionado esteja atribuído ao pacote correspondente, com sua primeira declaração sendo o nome do pacote, como `package test;`. Feito isto, o próximo passo é compilar o código, por meio de `javac`, e o programa está preparado para ser testado.

Para testar o programa, o comando `rrrun -tool=FT2 package.CompiledFile` deve ser executado, sendo `rrrun` o comando para rodar o RoadRunner, a flag `-tool=String` para indicar qual ferramenta será usada para teste, neste caso, o FastTrack2, representado pela string `FT2`, e o argumento `package.CompiledFile` para indicar qual programa está sendo testado, sendo `CompiledFile` o nome do arquivo compilado, e o conteúdo precedendo o `.`, o nome do pacote daquele programa.

4.1.3 Resultados

Uma vez instalada a ferramenta, foram efetuados alguns testes, para melhor análise de resultados e da saída apresentada. Começou-se testando o código 1, exemplo de corrida de dados, explicado na seção 2.5.1. Para rodar estes testes, adicionou-se a flag `-stacks`, a fim de entregar uma saída mais completa, mostrando a *stack* dos erros encontrados.

A saída do RoadRunner é bem extensa, e grande parte é em formato XML. Esta seção em XML também fica salva em um arquivo `log.xml`, que pode ser encontrado no pacote `log` do diretório do projeto. Este XML pode ser traduzido para uma *tree view* ou um JSON, para ajudar na legibilidade. A seguir, destacamos alguns pontos centrais para interpretar a saída do programa.

No XML, as partes mais importantes estão ilustradas nos códigos 17 e 18. No código 17, a seção com *tag* `errorTotal` mostra a quantidade total de erros encontrados no programa; a seção `distinctErrorTotal` informa quantos erros distintos ocorreram; e, por fim, a seção `errorCountPerErrorType` lista a contagem de ocorrências de cada tipo de erro. No caso exemplo, os erros são corridas de dados, detectadas pela análise feita pela ferramenta FastTrack, e existem 3 erros. Já em 18, é apresentado o campo (*field*) que contém o erro. No caso, é a variável `sum`, que é um `int`, representado por `_I`.

Código 17 – Informações geradas pelo FastTrack sobre os erros encontrados no código 1

```
<errorTotal> 3 </errorTotal>
<distinctErrorTotal> 1 </distinctErrorTotal>

<errorCountPerErrorType>
    <errorType> <name> FastTrack </name> <count> 3 </count> </errorType>
</errorCountPerErrorType>
```

Código 18 – Informações geradas pelo FastTrack sobre a variável associada aos erros do código 1

```
<fields>
    <field>
        <name> test/SampleThread.sum\_I </name>
        <error> <name> FastTrack </name> <count> 3 </count> </error>
    </field>
</fields>
```

O restante da saída também possui uma seção que descreve melhor o erro encontrado. Como três erros foram encontrados, todos apontados na mesma variável, apenas um será apresentado para exemplificar e explicar a saída, mostrada no código 19.

Código 19 – Informações geradas pelo FastTrack sobre a corrida de dados no código 1

```
## ============================================================================
## FastTrack Error
##
##           Thread: 2
##            Blame: test/SampleThread.sum_I
##            Count: 1    (max: 100)
##     Shadow State: [W=(1:3) R=(1:3) V=[]]
##   Current Thread: [tid=2    C=[(0:4) (1:0) (2:3) (3:0)]   E=(2:3)]
##            Class: test/SampleThread
##            Field: null.test/SampleThread.sum_I
##          Message: Write-Read Race
##     Previous Op: Write by  Thread-0[tid = 1]
##     Currrent Op: Read by  Thread-1[tid = 2]
##            Stack: tools.fasttrack.FastTrackTool.fieldError(
    FastTrackTool.java:752)
##                   tools.fasttrack.FastTrackTool.error(FastTrackTool.
    java:706)
##                   tools.fasttrack.FastTrackTool.read(FastTrackTool.
    java:370)
##                   tools.fasttrack.FastTrackTool.access(FastTrackTool.
    java:307)
##                   rr.tool.RREventGenerator.readAccess(RREventGenerator
    .java:146)
##                   test.SampleThread.__$rr_get_sum(BasicLockProblem.
    java)
##                   test.SampleThread.__$rr_run__$rr__Original_(
    BasicLockProblem.java:8)
##                   test.SampleThread.run(BasicLockProblem.java:6)
##
## ============================================================================
```

No log de saída, é apresentado o tipo do erro, informando que é um erro apontado pelo

FastTrack (*FastTrack Error*), e qual *thread* estava executando no momento da corrida. Os identificadores são atribuídos de forma sequencial, portanto, a *thread* principal tem identificador 0, a primeira *thread* a ser criada a partir da principal tem identificador 1, e assim sucessivamente. Esta seção também aponta a variável na qual ocorreu a corrida, e o tipo de erro (*Message*): se foi uma corrida escrita-escrita (*Write-Write Race*), escrita-leitura (*Write-Read Race*), ou leitura-escrita (*Read-Write Race*). Esta seção também aponta a variável violada (*blame* e *field*), e apresenta a quantidade de violações que ocorreram (*count*). Também são descritas quais são as operações envolvidas na corrida de dado, e por quais *threads* elas foram realizadas (*Previous Op*, indicando a operação anterior, e *Currrent Op*, indicando a operação corrente, que apontou a corrida), já tornando possível identificar o fluxo que apresenta a corrida. No caso, uma leitura pela *thread* 2 concorrendo com uma escrita feita pela *thread* 1.

Por esta saída, também é possível ilustrar melhor o processo de verificar a existência de uma corrida. É indicado que o tipo de erro foi uma *Write-Read Race*, ou seja, uma corrida escrita-leitura. Como visto anteriormente, quando explicado como identificar cada tipo de corrida, no momento em que a *thread* 2 tenta fazer uma leitura, o FastTrack tenta determinar se esta leitura é concorrente com alguma escrita anterior. Para isto, ele realiza uma comparação da epoch atual de escrita (representada no *Shadow State*) `W=(1:3` — indicando que a última escrita foi realizada pela *thread* 1, com relógio 3 — com o relógio vetorial da *thread* 2, apresentado em *Current Thread* como um vetor de *epochs*, `C=[(0:4) (1:0) (2:3) (3:0)]`. A comparação resulta em $3 \leqslant 0$, que é falso, e portanto, a tentativa de leitura é concorrente com uma escrita realizada pela *thread* 1. Esta diferença no relógio da *thread* 1 presente no vetor da *thread* 2 ocorre porque não há uma sincronização entre estas duas *threads*, e consequentemente, não há uma atualização dos *relógios* locais.

Pela *stack*, é possível perceber que o fluxo da *thread* 2 (cuja execução apontou a corrida) foi iniciar a execução do seu método run, e tentar executar a leitura da variável `sum` para depois incrementá-la — leitura presente na linha `sum += 1`, como é indicado por `test.SampleThread.__$rr_get_sum(BasicLockProblem.java)`. E então iniciam-se chamadas internas do FastTrack que levam ao apontamento do erro. É importante ressaltar que a *stack* só foi impressa neste exemplo pois a flag `-stacks` foi adicionada ao rodar o teste.

Em alguns testes, o campo *Previous Op*, não está indicando a *thread* que a realizou, apresentando `null`. Isto pode ocorrer porque no momento em que a mensagem é impressa, a instrumentação já terminou com esta *thread*, e portanto, sua representação está nula. Contudo, ainda é possível identificar as *threads* envolvidas na corrida por meio das *epochs*.

Rodando o teste na versão corrigida deste programa, presente no código 6 explicado na seção 2.6.1, no qual não existe uma corrida de dados, a ferramenta ainda indica que existe um erro. Isto ocorre pois o RoadRunner ignora as operações de aquisição e de liberação de *locks* reentrantes, e portanto, os relógios vetoriais das *threads* não são atualizados

para refletir a sincronização implementada (FLANAGAN; FREUND, 2010). Utilizando um bloco `synchronized`, passando ou o objeto de *lock*, ou efetuando uma adaptação na classe para passar outro objeto como referência, como exemplificado no código 20, a ferramenta corretamente aponta que não existem corridas de dados no programa, como apresentado no código 21.

Código 20 – Adaptação do código 6 usando `synchronized`

```
class SampleThread extends Thread {
    static int sum = 0;
    static Lock lock = new ReentrantLock();

    public void run() {
        for (int i = 0; i < 100000; i++) {
            synchronized (lock) {
                sum += 1;
            }
        }
    }
}
```

Código 21 – Saída sem erro gerada pelo FastTrack para o código 20

```
<errorTotal> 0 </errorTotal>
<distinctErrorTotal> 0 </distinctErrorTotal>
```

Ao tentar testar a solução utilizando semáforos, encontrou-se o mesmo problema, indicando que o RoadRunner também não é compatível com a primitiva de semáforos.

Também testou-se a implementação do problema do produtor-consumidor, apresentada na seção 2.8, começando por sua versão ingênua, mostrada nos códigos 11 e 12.

Como neste código as variáveis compartilhadas são vetores, na saída do RoadRunner, ao invés da informação `<fields>` estar preenchida, o dado vem apresentado em um campo `arrays`, indicando a operação sendo feita no vetor e sua localização no programa, como apresentado no código 22. Por exemplo, é possível ver que uma corrida ocorreu em um vetor em uma operação de escrita (`wr_array`), e a localização desta operação no código é na linha 39, correspondente a uma operação de escrita em uma posição do vetor `buffer`. Nesta saída, também são apresentados outras operações em vetores que ocasionaram corridas, como o vetor sendo lido (`rd_array`) na linha 38, correspondente a leitura de uma posição do vetor `buffer`.

É importante lembrar que esta listagem de `arrays` pode variar de execução para execução do FastTrack, dependendo das intercalações que ocorrem naquela execução específica, podendo haver mais ou menos operações em vetores na saída.

Código 22 – Informações geradas pelo FastTrack sobre o vetor responsável pela corrida de dados do programa Produtor-Consumidor

```
<arrays>
    <target>
        <location> wr_array@test/ProdCons.java:39:30 </location>
        <error> <name> FastTrack </name> <count> 9 </count> </error>
    </target>
    <target>
        <location> rd_array@test/ProdCons.java:38:17 </location>
        <error> <name> FastTrack </name> <count> 9 </count> </error>
    </target>
    <target>
        <location> wr_array@test/ProdCons.java:19:23 </location>
        <error> <name> FastTrack </name> <count> 11 </count> </error>
    </target>
</arrays>
```

O fluxo de checagem é análogo ao apresentado no teste anterior. Usando como exemplo uma corrida identificada na escrita de um vetor, quando a *thread* 2 tenta uma escrita, o FastTrack efetua uma comparação para verificar se esta escrita é concorrente com a última escrita feita no mesmo vetor. No código 23 é apresentado um exemplo de saída de uma corrida identificada nos acessos a um vetor. De modo geral, seus campos são os mesmos do *log* anterior, mostrado em 19. É possível ver, pelo campo *Array*, que a posição sendo acessada no vetor na operação que aponta corrida de dados é a posição 0, apresentada entre colchetes.

Código 23 – Informações geradas pelo FastTrack sobre a corrida de dados do programa Produtor-Consumidor

```
## ========================================================================
## FastTrack Error
##
##           Thread: 2
##            Blame: wr_array@test/PCWithoutSemaphore.java:39:30
##            Count: 1    (max: 100)
##       Alloc Site: null
##     Shadow State: [W=(1:3) R=(0:0) V=[]]
##  Current Thread: [tid=2    C=[(0:3) (1:0) (2:3) (3:0)]   E=(2:3)]
##            Array: @02[0]
##          Message: Write-Write Race
##     Previous Op: Write by  null
##     Currrent Op: Write by  Thread-1[tid = 2]
##            Stack: tools.fasttrack.FastTrackTool.arrayError(
    FastTrackTool.java:726)
##                   tools.fasttrack.FastTrackTool.error(FastTrackTool.
    java:708)
##                   tools.fasttrack.FastTrackTool.write(FastTrackTool.
    java:470)
##                   tools.fasttrack.FastTrackTool.access(FastTrackTool.
    java:305)
##                   rr.tool.RREventGenerator.arrayWrite(RREventGenerator
    .java:584)
##                   test.Consumer.__$rr_run__$rr__Original_(
    PCWithoutSemaphore.java:39)
##                   test.Consumer.run(PCWithoutSemaphore.java:36)
##
## ========================================================================
```

Rodando o teste na versão completa deste programa, presente nos códigos 13, 14 e 15, explicados na seção 2.8, na qual as corridas de dados e violações de ordem foram corrigidas, a ferramenta corretamente indica que não existe erro, produzindo uma saída de sucesso com seção de erros igual à apresentada anteriormente no código 21.

4.1.4 Discussão

Após estudo e testes da ferramenta, conclui-se que o FastTrack é uma implementação com instalação e uso fáceis e acessíveis, e com um tempo de execução da ordem de segundos, principalmente por ser uma abordagem baseada em *happens-before*. O RoadRunner possui poucas dependências, e o aluno consegue facilmente instalá-lo e testar seus próprias programas, sem necessidade de casos de teste. Sua saída possui um nível de complexidade

um pouco mais alto, mas ainda é compreensível, e bem completa, indicando a causa de cada tipo de erro.

Contudo, também é relevante levantar as limitações da técnica. Por não apresentar compatibilidade com semáforos, e ignorar as aquisições e liberações do *lock* reentrante padrão do Java, a ferramenta pode apresentar falsos positivos, potencialmente confundindo o aluno no processo de aprendizado. Além disto, há uma limitação que não apareceu nos casos de teste avaliados, mas que existe por definição na técnica. Por ser uma técnica de detecção, ela não explora intercalações alternativas viáveis, trabalhando somente com o rastreamento de execução analisado, é possível que ela não identifique algumas corridas de dado, caso a execução não entre dentro de blocos condicionais que possam conter uma corrida.

4.2 RVPREDICT

O RVPredict é uma ferramenta de teste para identificar **corridas de dados** — definidas em maior detalhes na seção 2.5.1 — por meio de análise dinâmica *happens-before*, analogamente ao FastTrack. É uma técnica de predição, ou seja, trabalha com intercalações alternativas à resultante da execução, e tem como alvo sistemas com paradigma de programação geral e com consistência sequencial.

4.2.1 Funcionamento da ferramenta

O RVPredict inclui informações de fluxo de controle, a fim de melhorar a acurácia de análises *happens-before*, aprimorando a exploração de entrelaçamentos alternativos para identificar corridas, por meio de um encapsulamento das relações de ordem em um conjunto de restrições. A técnica, então, utiliza uma ferramenta de resolução para solucionar o problema de restrições, identificando as corridas.

O RVPredict possui duas fases de análise. Na primeira fase, ele somente coleta o rastreamente corrente, registrando os acessos a dados compartilhados, sincronizações de *thread* e eventos de *branch*. Na segunda fase, é realizada análise preditiva de corridas de dados, identificando os pares conflitantes, e montando as restrições correspondentes.

4.2.1.1 Modelagem

Para modelar o problema de predição de corridas de dados, o RVPredict estabelece algumas definições de conceitos a serem utilizados no processo de estruturação de restrições relativas às relações de ordem.

Eventos são operações realizadas por *threads* em objetos concorrentes. O RVPredict adiciona um novo evento, denominado *branch*, além dos eventos tradicionais, comumente estudados:

- *begin* e *end*: primeiro e último evento de uma *thread*
- *read* e *write*: leitura e escrita de uma variável
- *acquire* e *release*: aquisição e liberação de um *lock*
- *fork*: criação de uma nova *thread*
- *join*: bloqueio até que a *thread* finalize
- *branch*: salto para uma nova operação (ramificação)

Eventos de *branch* servem para indicar uma possível mudança de fluxo de controle, que determina a próxima operação a ser executada em uma *thread*. A ocorrência ou não desta mudança depende de uma computação local não visível para outras *threads*, como o resultado de uma condicional. Por conta desta característica, o RVPredict considera que a escolha do evento *branch* depende de todas as operações de leitura anteriores feitas nesta mesma *thread* (HUANG; MEREDITH; ROSU, 2014).

A partir destas definições, o RVPredict abstrai um rastreamento de execução como uma sequência de eventos. A consistência de um rastreamento determina que todas as especificações seriais que descrevem o comportamento válido dos objetos concorrentes envolvidos no rastreamento estão sendo cumpridas (HUANG; MEREDITH; ROSU, 2014). Ou seja, todas as operações envolvendo objetos concorrentes (como *locks*, *threads* e dados compartilhados) devem estar respeitando as relações de ordem destes objetos. Estas relações são similares às definidas na seção 4.1.1.1 de *happens-before* do FastTrack, com algumas especificações a mais:

- Um evento de leitura de um dado deve conter o valor escrito pela escrita mais recente neste mesmo dado;
- Todo evento de liberação de *lock* é precedido por um evento de aquisição no mesmo *lock*, pela mesma *thread*;
- Cada par de liberação e aquisição de um *lock* não pode ser intercalado por nenhum outro evento de aquisição ou liberação do mesmo *lock*;
- O primeiro evento de uma *thread* (*begin*) só pode ocorrer após o evento de *fork* que criou esta *thread*;
- O evento de *join* de uma *thread* só pode ocorrer após o último evento dela (*end*);

Eventos de *branch* podem ocorrer em qualquer momento em um rastreamento de execução, sem afetar sua consistência. É importante ressaltar que escritas e leituras a variáveis locais não são incluídas em rastreamentos, pois são irrelevantes para detecção de corridas de dados.

4.2.1.2 Rastreamentos alternativos viáveis

Qualquer rastreamento produzido por um programa em execução deveria ser consistente. É necessário, então, estabelecer uma forma de identificar rastreamentos alternativos — não necessariamente já executados — que respeitem a esta regra, e são, consequentemente, viáveis.

Para identificar outras possibilidades de rastreamento de execuções viáveis (diferentes do que foi executado) que possam apresentar corridas de dados, técnicas de predição anteriores permutavam um rastreamento de execução por meio de uma definição conservadora de viabilidade, muitas vezes exigindo que toda leitura de uma intercalação alternativa retorne o mesmo valor que no fluxo original (BIANCHI; MARGARA; PEZZE, 2017). Esta abordagem, contudo, é muito limitante, e pode ignorar intercalações alternativas válidas.

O RVPredict utiliza uma definição mais precisa para rastreamentos alternativos viáveis — para aumentar a cobertura da técnica — determinando que para um rastreamento ser válido ele deve respeitar determinismo local e fechamento em prefixo (*prefix closedness*). Fechamento em prefixo garante que eventos são gerados em ordem de execução, com possibilidade de intercalação entre qualquer um deles. Já pelo determinismo local, o próximo evento de uma *thread* é determinado somente pelos eventos anteriores daquela mesma *thread*. Este próximo evento pode ocorrer em qualquer momento consistente após os anteriores, sempre respeitando as regras de consistências.

A partir desta definição, o RVPredict consegue modelar o problema de identificação de corridas estruturando restrições de viabilidade e, a partir delas, descobrindo as intercalações viáveis, independente delas terem sido executadas ou não.

4.2.1.3 Identificando corridas de dados

A ideia central do RVPredict é identificar corridas de dados encontrando rastreamentos de execução viáveis nos quais existam pares de operações conflitantes cujos eventos sejam seguidos um do outro no rastreamento.

Os eventos a e b formam um par de operações conflitantes — COP (*Conflicting Operation Pair*) — se e somente se, ambas forem executadas por *threads* distintas, estiverem acessando o mesmo dado, e pelo menos uma das operações é uma escrita (a outra podendo ser uma escrita ou leitura) (HUANG; MEREDITH; ROSU, 2014). Esta definição está alinhada com as definições de **corrida de dados** descritas anteriormente na seção 2.5.1.

Para detectar estes rastreamentos viáveis, a abordagem formula um problema de resolução de restrições. As restrições consistem de três partes: restrições MHB (*must happen-before*, ou deve acontecer antes), restrições de *lock*, e restrições de corrida de dados, sendo as primeiras duas comuns a todas as corridas, e a última específica para cada corrida.

Restrições MHB se assemelham com as regras de consistência relacionadas às operações de *thread*, descritas na modelagem do RVPredict, na seção 4.2.1.1:

- Se o evento a é um *fork* (criação) de uma *thread* t, e o evento b é uma operação realizada pela *thread* t, a deve ocorrer antes de b
- Se o evento a é uma operação de uma *thread* t, e o evento b é um *join* da *thread* t, a deve ocorrer antes de b

Restrições de *lock* definem as restrições de ordem entre eventos de aquisição de liberação de *locks* específicos daquele programa, sempre obedecendo a definição de que se o evento a é a liberação de um *lock*, e o evento b é a aquisição do mesmo *lock*, a deve acontecer antes de b. O intuito é restringir que duas sequências de eventos protegidas pelo mesmo *lock* não se entrelacem.

Restrições de corrida encapsulam a corrida (a definição de que os eventos de um par conflitante sejam seguidos um do outro no rastreamento) e as condições de fluxo de controle específicas para cada par sendo considerado. A condição de fluxo de controle só é considerada nos casos em que existe um evento de *branch* precedendo algum dos eventos do par (HUANG; MEREDITH; ROSU, 2014). Para os COPs em que isto ocorre, o fluxo de controle é necessário pois o evento de *branch* pode indicar ou não uma mudança no fluxo, e esta escolha depende das leituras anteriores. Portanto, é necessário garantir que todas as leituras desta *thread* anteriores à *branch* lêem os mesmos valores que os registrados no rastreamento original. Esta garantia é dada por meio de restrições de fluxo de controle, e asseguram que o segundo evento do par é viável (averiguando que é possível o fluxo entrar naquele ramo).

A partir destas restrições, para cada par conflitante COP, o RVPredict gera uma fórmula, de modo que todas as restrições MHB, de *lock*, e as restrições correspondente àquele par devem ser cumpridas. Ou seja, encapsula todas as regras de reordenação do rastreamento definidas no modelo apresentado em restrições lógicas de primeira ordem. Esta fórmula é então resolvida por meio de um mecanismo de solução de restrições (um *SMT solver*, ou solucionador de SMT - *Satisfiability Modulo Theories*, teorias de módulo de satisfabilidade em tradução direta), e, desta forma, a técnica determina se aquele par de eventos corresponde a uma corrida ou não. Essencialmente, a existência de uma solução para o problema implica na existência de um rastreamento viável em que aquela corrida ocorra, e, consequentemente, existe uma corrida de dados no programa (HUANG; MEREDITH; ROSU, 2014).

4.2.2 Instalação e uso da ferramenta

A implementação da técnica apresentada se encontra em uma ferramenta com o mesmo nome, RVPredict (RV representa *Runtime Verification*). Trata-se de uma ferramenta de

análise dinâmica escrita em Java, voltada para testar programas concorrentes desenvolvidos em Java, com foco em detectar corridas de dados (INC, 2021a). Ela possui uma variante para programas feitos em C, porém, neste trabalho, apenas a ferramenta para Java será analisada, com intuito de manter a consistência com as outras técnicas estudadas.

Para instalar e rodar a ferramenta, é necessário instalar JDK 8, ou superior, e seguir o passo a passo da seção *Quickstart* da documentação do RVPredict (INC, 2021b). Se, ao tentar executar o comando `RVPredict`, ocorrer algum problema, mesmo após a configuração de variável de ambiente, é possível substituir este comando pela sua versão mais verbosa `java -jar $rvPath/lib/rv-predict.jar`.

Para testar um programa, é necessário compilá-lo, por meio de `javac`. Não existe a necessidade de adicioná-lo dentro do projeto do RVPredict, desde que os *paths* utilizados na linha de comando, ou na variável de ambiente, estejam corretos. Por exemplo, `java -jar $rvPath/lib/rv-predict.jar package.CompiledFile`, para programas dentro de um pacote, ou `java -jar $rvPath/lib/rv-predict.jar CompiledFile` para programas sem pacote.

4.2.3 Resultados

Uma vez instalada a ferramenta, foram efetuados alguns testes para melhor análise de resultados e da saída apresentada.

Começou-se testando um exemplo de corrida de dados (código 1, explicado na seção 2.5.1). Ao tentar executar esse teste 10 vezes, em todas as tentativas o RVPredict demorou um tempo elevado rodando — na ordem de minutos —, travando o computador depois de 10 minutos, em média. Isto possivelmente ocorreu pois o programa sendo testado possui um laço `for` de 100000 iterações, executado por cada uma das duas *threads*. É possível que um laço extenso seja muito pesado computacionalmente para esta ferramenta, por ser uma abordagem de predição, que monta as restrições dinamicamente a partir do rastreamento executado e tenta resolver todos os rastreamentos alternativos. Esta incompatibilidade com programas que possuem uma quantidade de ramos de uma ordem superior a 25 é uma limitação significativa da ferramenta, umas vez que restringe os programas que podem ser testados de forma bem sucedida. Na documentação do RVPredict, é indicado que isto pode ocorrer devido a um sobrecarga da análise, ou por conta de alguma condição de deadlock no programa. A sugestão para lidar com isto é interromper a execução a qualquer momento, e rodar a etapa de predição no *log* produzido até aquele momento, mas não obteve-se sucesso ao tentar esta abordagem, mesmo aumentando o tempo de *timeout* (parâmetro interno da ferramenta) do RVPredict.

Para fins de comparação da saída com o FastTrack, tentou-se, então, modificar o programa, para reduzir o custo computacional e possibilitar uma análise de seus resultados. Reduzindo o laço para 1000 iterações, ainda manteve-se o comportamento de lentidão e

travamento, e reduzindo ainda mais, para 100, obteve-se o resultado da execução, mas o processo de detecção de corridas deu erro de *timeout*. Diminuindo para 25, foi possível obter uma saída.

A saída pura do RVPredict é menos extensa do que a do RoadRunner, apresentando um resultado mais direto, contendo somente informações dos eventos envolvidos em cada corrida identificadas, e a saída do programa executado. Começa indicando o nome do diretório criado para armazenar o *log* da execução. Neste diretório, existe um arquivo `result.txt`, contendo a saída do programa, indicando os erros. A saída referente a este primeiro teste é apresentada no código 24.

Código 24 – Informações geradas pelo RVPredict sobre a corrida de dados no código 1

```
----------------Instrumented execution to record the trace
   -----------------
[RVPredict] Log directory: /tmp/RVPredict3488692360090783480
[RVPredict] Finished retransforming preloaded classes.
50
Data race on field SampleThread.sum:
    Write in thread 12
      > at SampleThread.run(BasicLockProblem.java:6)
    Thread 12 created by thread 1
        at BasicLockProblem.main(BasicLockProblem.java:16)

    Read in thread 13
      > at SampleThread.run(BasicLockProblem.java:6)
    Thread 13 created by thread 1
        at BasicLockProblem.main(BasicLockProblem.java:17)
```

Além dos erros, o resultado também apresenta a própria saída do programa. No caso da variante do programa testada, com laço de 25 iterações, esta saída foi `50`. A parte mais relevante do resultado apresentado, contudo, é a seção final, que aponta a corrida. O RVPredict indica qual o campo que apresenta a corrida `Data race on field SampleThread.sum`, descrevendo qual o par de eventos envolvido nela. É possível observar que o primeiro evento do par conflitante é uma escrita, `Write in thread 12`, na linha 6 (`BasicLockProblem.java:6`) do método `run` (referente ao incremento da variável `sum`), realizada pela *thread* que foi iniciada pela thread 1 (a *thread* principal) na linha 16 do método `main`, ou seja, a `threadA`. A outra operação do par conflitante é uma leitura, `Read in thread 13`, sendo feita também na linha 6 do método `run` (referente à leitura da variável `sum` antes do incremento), realizada pela *thread* que foi iniciada pelo fluxo principal na linha 17 — a `threadB`.

Rodando o teste na versão consertada deste programa, presente no código 6 explicado na seção 2.6.1, no qual não existe uma corrida de dados, a ferramenta indica que de fato não existe mais uma corrida, diferentemente do RoadRunner. Sua saída de sucesso

também apresenta a saída do programa, e um *log* indicando que nenhuma corrida foi detectada (*No races found*). Isto ocorre pois o RVPredict leva em conta operações de aquisição e de liberação de *locks* reentrantes, fazendo somente uma adaptação na execução para considerar somente as aquisições e liberações mais externas do mesmo *lock*, a fim de simplificar as restrições (HUANG; MEREDITH; ROSU, 2014).

Código 25 – Saída sem erro gerada pelo RVPredict para o código 6

```
----------------Instrumented execution to record the trace
   -----------------
[RVPredict] Log directory: /tmp/RVPredict925607665039204062
[RVPredict] Finished retransforming preloaded classes.
50
No races found.
```

Ao tentar testar a solução utilizando semáforos, apresentada no código 9, encontrou-se o mesmo resultado de sucesso, indicando que o RVPredict é compatível com esta primitiva. Contudo, no artigo de referência, não há menção desta compatibilidade, podendo ter sido uma adição posterior à ferramenta.

É relevante observar que, para executar o teste com esta solução, foi necessário aumentar a janela de *timeout*, por meio da opção `-window-timeout`. Esta necessidade surge porque o bloco `try/catch` pode modificar o fluxo de controle, adicionando eventos de *branch*, e, consequentemente, tornando a computação de restrições mais complexa.

Também testou-se a implementação do problema do produtor-consumidor, apresentada na seção 2.8, começando por sua versão ingênua, mostrada nos códigos 11 e 12. Como neste exemplo as variáveis compartilhadas são vetores, o *log* do RVPredict, apresentado nos códigos 26 e 27, indica que a corrida de dados se encontra em um *array*, no elemento `#1`, ou seja, em seu segundo elemento. A saída é dividida pelos pares conflitantes, e é possível verificar que foram identificadas cinco corridas, sendo duas delas entre uma *thread* produtora e uma *thread* consumidora, e as restantes entre pares de *threads* consumidoras ou produtoras.

Código 26 – Informações geradas pelo RVPredict sobre a corrida de dados do programa Produtor-Consumidor

```
----------------Instrumented execution to record the trace
   -----------------
[RV-Predict] Log directory: /tmp/rv-predict9194449060596766118
[RV-Predict] Finished retransforming preloaded classes.
0
0
0
1
1
1
-1
-1
-1
-1
-1
-1
2
2
2

Data race on array element #1:
    Write in thread 12
      > at Producer.run(ProdCons.java:17)
    Thread 12 created by thread 1
        at ProdCons.main(ProdCons.java:65)

    Write in thread 16
      > at Producer.run(ProdCons.java:17)
    Thread 16 created by thread 1
        at ProdCons.main(ProdCons.java:65)


Data race on array element #1:
    Write in thread 13
      > at Consumer.run(ProdCons.java:37)
    Thread 13 created by thread 1
        at ProdCons.main(ProdCons.java:66)

    Write in thread 17
      > at Consumer.run(ProdCons.java:37)
    Thread 17 created by thread 1
        at ProdCons.main(ProdCons.java:66)
```

Código 27 – Informações geradas pelo RVPredict sobre a corrida de dados do programa Produtor-Consumidor - Continuação

```
Data race on array element #1:
    Write in thread 17
      > at Consumer.run(ProdCons.java:37)
    Thread 17 created by thread 1
        at ProdCons.main(ProdCons.java:66)

    Write in thread 12
      > at Producer.run(ProdCons.java:17)
    Thread 12 created by thread 1
        at ProdCons.main(ProdCons.java:65)


Data race on array element #1:
    Read in thread 17
      > at Consumer.run(ProdCons.java:36)
    Thread 17 created by thread 1
        at ProdCons.main(ProdCons.java:66)

    Write in thread 12
      > at Producer.run(ProdCons.java:17)
    Thread 12 created by thread 1
        at ProdCons.main(ProdCons.java:65)


Data race on array element #1:
    Read in thread 13
      > at Consumer.run(ProdCons.java:36)
    Thread 13 created by thread 1
        at ProdCons.main(ProdCons.java:66)

    Write in thread 17
      > at Consumer.run(ProdCons.java:37)
    Thread 17 created by thread 1
        at ProdCons.main(ProdCons.java:66)
```

Por simplicidade, somente a primeira corrida apresentada no *log* será mais detalhada, mas a mesma abordagem pode ser aplicada para todos os outros resultados. Esta corrida foi causada por uma concorrência de escrita e escrita, como é possível observar pelos dois eventos da corrida: `Write in thread 12` e `Write in thread 16`. Um evento de escrita realizado no método `run`, na linha 17 (correspondente a uma escrita na variável compartilhada `buffer`, `buffer[bufferPos[0]] = id`), feita pela *thread* produtora, iniciada na linha 65 da `main`. O outro evento do par foi uma escrita feita por uma outra *thread* produtora, também iniciada na linha 65 — dado que é um laço de inicializações na `main`

— em seu método `run`, quando esta *thread* também tenta escrever seu identificador no mesmo `buffer`, na linha 17.

Rodando o teste na versão completa deste programa, presente nos códigos 13, 14 e 15, explicados na seção 2.8, na qual as corridas de dados e violações de ordem foram consertadas, a ferramenta corretamente indica que não existe erro, como é possível observar no código 28.

Código 28 – Saída sem erro gerada pelo RVPredict para o programa Produtor-Consumidor

```
----------------Instrumented execution to record the trace
   -----------------
[RVPredict] Log directory: /tmp/RVPredict7208840192240788542
[RVPredict] Finished retransforming preloaded classes.
0
0
1
0
0
1
1
1
0
1
2
2
2
2
2
No races found.
```

4.2.4 Discussão

Após as análises e testes apresentados ao longo desta seção, é possível concluir que o RVPredict é uma ferramenta muito flexível e de instalação e uso fáceis e acessíveis, exigindo pouquíssimos pré-requisitos, facilitando o processo de testagem para alunos. Além disto, é muito fácil testar programas sem a necessidade de adicioná-los no projeto do RVPredict, e sem precisar da estrutura de pacotes. Possui bastante compatibilidade com primitivas, conseguindo reconhecer *locks* reentrantes, *synchronized* e semáforos, sem apresentar falsos alarmes, que são muito nocivos para o ambiente de aprendizado. A ferramenta também é mais completa do que uma técnica de dedução, pois leva em consideração fluxos alternativos. Esta alta compatibilidade e baixo número de resultados falsos faz da ferramenta muito vantajosa e completa. O RVPredict também possui uma saída facilmente compreensível, e com informações mais diretas sobre as corridas, ajudando na

compreensão do erro.

Sua maior desvantagem é a maior complexidade em suas computações, que não lidam bem com laços com uma quantidade de iterações superior a 25, ou programas que apresentem pontos de alteração de fluxo de controle (como condicionais ou blocos `try/catch`), sendo necessário aumentar a janela de *timeout* para permitir a conclusão da análise da ferramenta. Em alguns casos, porém, nem mesmo o incremento do *timeout* possibilitou a obtenção de resultados, pois a máquina travou ao tentar executar a análise.

Conhecendo suas limitações, é possível aplicar o RVPredict em casos mais compatíveis com seu uso, obtendo resultados muito positivos e completos.

4.3 ATOMIZER

O Atomizer é uma ferramenta de teste que visa identificar violações de atomicidade — definidas em maior detalhamento na seção 2.5.2 — por meio de análises dinâmicas baseadas nos algoritmos de *lockset* e de redução. A técnica proposta pelo Atomizer é voltada para um paradigma de programação geral, e considera que está trabalhando em sistemas com consistência sequencial. Sua análise é dinâmica, e garante viabilidade dos resultados, uma vez que explora os entrelaçamentos de execuções reais do sistema, por ser uma técnica de dedução, e não de predição.

É relevante observar que o artigo do Atomizer usa o termo *race condition* (condição de corrida) para se referenciar ao que é mais conhecido como *data race*, ou corrida de dados.

4.3.1 Funcionamento da ferramenta

O Atomizer propõe a utilização da teoria de redução de Lipton (LIPTON, 1975) para efetuar a checagem de atomicidade, e de uma variação do algoritmo de *lockset* proposto pelo Eraser (SAVAGE et al., 1997) para inferir *locks*.

A técnica propõe que blocos atômicos podem ter seu caminho (ou rastreamento) de execução reduzidos para uma execução serial equivalente, ou seja, em que o caminho é executado sem intercalações de outras *threads*, com o mesmo estado resultante. Esta redução de caminho é feita a partir dos conceitos definidos pela teoria de redução de Lipton.

A teoria de redução, proposta por Lipton, tem como base os conceitos de ações *right-mover* e *left-mover*, traduzidas como deslocamento para direita e deslocamento para esquerda, respectivamente. Uma ação a é *right-mover* se, para qualquer execução em que a ação a, realizada por um fluxo de execução, é imediatamente seguida por uma ação b de um outro fluxo, as ações a e b podem ter a ordem trocada sem afetar o estado resultante. Estado aqui se referenciando às variáveis do programa. Em contraste, uma ação b é *left-mover* se sempre que b imediatamente seguir uma ação a de outra *thread*, elas possam ter a ordem trocada sem modificar o estado resultante (LIPTON, 1975).

De modo geral, aquisições de *lock* são consideradas ações *right-mover*, dado que se uma ação a é uma aquisição de *lock*, então a ação subsequente b da outra *thread* não adquire nem libera aquele *lock*, uma vez que neste instante ele "pertence" à primeira *thread*, não afetando seu estado. Portanto, a operação de aquisição pode ser movida para a direita sem afetar o resultado do estado. Analogamente, liberações de *lock*, em geral, são *left-mover*, uma vez que, caso uma operação b seja uma liberação de *lock* que segue uma ação a, é impossível que outra *thread* esteja segurando o *lock* no instante da ação a, e, portanto, a não pode ser nem uma aquisição nem uma liberação de *lock*, permitindo que a operação b possa ser movida para a esquerda sem modificar o estado resultante (FLANAGAN; FREUND, 2004).

Já acessos a variáveis compartilhadas podem ser consideradas ações *both-mover* ou *non-mover*. Caso a variável seja protegida por um mecanismos de sincronização que faça com que duas *threads* nunca consigam acessar a variável simultaneamente, as operações de acesso a esta variável são classificadas como *both-mover*, o que significa que a operação é tanto *right-mover* quanto *left-mover*. Se a variável não for consistentemente protegida por algum mecanismo e permitir acessos simultâneos, o acesso é considerado uma ação *non-mover*.

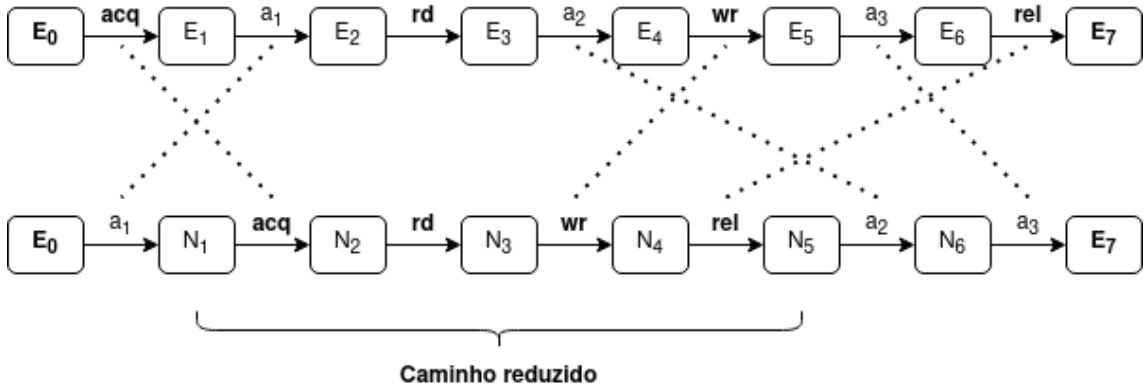
Nem todos *locks* tem suas aquisições e liberações classificadas como *right-mover* e *left-mover*, respectivamente, podendo introduzir alarmes falsos. Portanto, a implementação desta técnica é adaptada para classificar apropriadamente estes *locks*, como por exemplo, *locks* reentrantes, cuja aquisição e liberação é *both-mover*, uma vez que estas ações não podem interagir com outros fluxos (semelhante a um acesso protegido).

Com os conceitos da teoria de redução descritos acima, é possível agora detalhar o processo de redução de um rastreamento de execução de um bloco de código.

Este processo é exemplificado na figura 6 que contém, no primeiro caminho apresentado, uma execução de uma *thread* que adquire um *lock*, lê uma variável protegida por este mesmo *lock*, escreve nesta variável protegida, e libera o *lock*. Essas ações são intercaladas com operações de outras *threads*, representadas por a_n. Dado que a operação de aquisição é *right-mover*, ela pode ter sua ordem no caminho trocada com a operação de outra *thread* que a sucede. Analogamente, pela operação de escrita ser *both-mover*, ela pode ter sua execução mais à esquerda no caminho, postergando a ação de outra *thread* que a precedia. Da mesma forma, a operação de liberação de *lock* é *left-mover*, então pode ser movida para a esquerda no caminho. É possível, assim, construir uma execução serial equivalente (uma vez que os estados iniciais e finais se mantém), sem intercalações com ações de outras *threads*. Esta execução alternativa, apresentada no segundo caminho da figura 6 é o caminho reduzido.

Em suma, um caminho de execução de um bloco de código pode ser reduzido para uma execução serial se ele tiver uma sequência de operações *right-mover*, seguidas de no máximo UMA ação *non-mover*, seguidas por uma sequência de operações *left-mover*. É

Figura 6 – Exemplo de execução reduzida (FLANAGAN; FREUND, 2004)



importante ressaltar que ações *both-mover*, como *locks* reentrantes, e operações de acesso a variáveis protegidas, são tanto *right-mover* quanto *left-mover*, podendo estar presentes nestas respectivas sequências do caminho reduzido.

Este bloco redutível, portanto, é atômico, uma vez que para toda execução do programa, existe uma execução equivalente, com o mesmo resultado, em que este bloco é executado serialmente, ou seja, sequencialmente, sem intercalações com ações de outros fluxos de execução.

O ponto de comprometimento (ou *commit point*) de um caminho redutível é a ação que divide os estados do caminho em pré-comprometimento (*pre-commit*), em que todas as ações anteriores são *right-movers*. e pós-comprometimento (*post-commit*), em que todas as ações seguintes são *left-movers*. O ponto de comprometimento é a ação *non-mover* daquele caminho, caso ele possua alguma. Do contrário, a primeira operação *left-mover* será o *commit point*.

A ideia central para identificar blocos "atômicos" de código que estejam atendendo às restrições de atomicidade é admitir apenas caminhos que sejam redutíveis, uma vez que estes caminhos são atômicos. Para isto, observa-se o estado de cada *thread*, denotando se sua execução se encontra fora de um bloco "atômico", ou na fase pré-comprometimento ou pós-comprometimento daquele bloco.

A partir desta informação de estado, o Atomizer estabelece regras para determinar se as operações a serem realizadas são permitidas, ou violam a atomicidade daquele bloco. Por exemplo, acessos a variáveis não protegidas são ações *non-mover*, e, portanto, podem acontecer quando a *thread* está fora de um bloco atômico. Operações de aquisição, por sua vez, são *right-movers*, então podem ocorrer tanto fora de um bloco atômico, ou na fração pré-comprometimento dele. **Já uma ação diferente de *left-mover*, como um acesso a variável desprotegida ou uma aquisição de *lock*, ocorrendo na parte pós-comprometimento do bloco é apontada como uma ação que viola a atomicidade**.

A técnica também utiliza a ideia de algoritmo de *lockset* para inferir *locks* que estão

protegendo as variáveis. De forma simplificada, neste algoritmo, cada variável compartilhada é associada aos *locks* disponíveis. Quando uma *thread* acessa uma variável compartilhada `x`, a análise faz uma interseção do conjunto atual de *locks* candidatos para a variável `x` com os *locks* alocados por aquela *thread*. Todas as *threads* que acessam uma mesma variável compartilhada `x` devem ter ao menos um *lock* em comum. O conjunto de *locks* candidatos de cada variável após a execução do algoritmo indica os *locks* que protegem cada uma dessas variáveis. Um conjunto vazio indica, então, que nenhum *lock* consistentemente protege a variável compartilhada.

Em resumo, a ferramenta computa dinamicamente o conjunto de *locks* em comum alocados quando as variáveis compartilhadas são acessadas, e identifica os fluxos que acessam as mesmas variáveis, mas não compartilham nenhum *lock*. No contexto da técnica proposta no Atomizer, este algoritmo é majoritariamente utilizado para classificação das operações de acesso como *movers* ou *non-movers*, averiguando as condições de sincronização das variáveis.

Caso o conjunto de *locks* candidatos de uma variável fique vazio, ou seja, aquele dado não é consistentemente protegido por nenhum *lock*, as operações de acesso a esta variável são classificadas como *non-mover*. Caso isto aconteça, acessos subsequentes serão considerados *non-mover*, mas acessos anteriores — que podem ter sido erroneamente classificados como *movers* anteriormente — não serão reclassificados, pois isto seria uma operação custosa. Este relaxamento diminui o grau de *soundness* da técnica, como melhor descrito na seção 3.2.5, mas também evita alarmes falsos sobre corridas de dados não nocivas (FLANAGAN; FREUND, 2004).

A ferramenta também estabelece uma definição mais relaxada de graus de compartilhamento de dados, como estados específicos para campos somente lidos — mas não modificados — por múltiplas *threads*, ou campos cuja propriedade foi transferida para uma *thread* diferente de sua original, que não o acessa mais. Estas extensões são usadas para adaptar a análise *lockset*, reduzindo a quantidade de alarmes falsos, e também diminuindo o grau de *soundness*.

O Atomizer consegue deduzir quais blocos devem ser checados para garantir atomicidade, sem precisar de anotações no código. Ele faz isso por meio de duas opções de heurísticas. A primeira delas considera que todos os métodos exportados por classes (públicos ou protegidos), e todos blocos e métodos sincronizados devem ser atômicos, sem levar em conta o método `main` e os métodos `run` de objetos `Runnable`, dado que estes métodos não costumam ter intenção de serem atômicos. Já a segunda apenas assume que todos blocos e métodos sincronizados devem ser atômicos (FLANAGAN; FREUND, 2004). A implementação do Atomizer encontrada utiliza a segunda heurística.

É importante ressaltar que métodos ou blocos sincronizados não necessariamente garantem atomicidade, uma vez que quando múltiplas classes acessam o mesmo dado compartilhado, o objeto usado para sincronização também deve ser compartilhado, como

melhor exemplificado nas seções 2.5.2 e 2.6.1. Além disto, esta heurística de dedução limita a capacidade de detecção do erro, dado que nos casos em que a violação está presente em um código não sincronizado, a ferramente não é capaz de deduzir que aquele código deveria ser atômico.

4.3.2 Instalação e uso da ferramenta

A implementação encontrada da ferramenta Atomizer está presente no framework RoadRunner, explicado em maior detalhamento na seção 4.1.2. Para testar um programa, é só seguir as instruções apresentadas anteriormente, e executar o comando `rrrun -tool=A package.CompiledFile`, sendo `rrrun` o comando para rodar o RoadRunner, a flag `-tool=String` para indicar qual ferramenta será usada para teste, neste caso, o Atomizer, representado pela string `A`, e o argumento `package.CompiledFile` para indicar qual programa está sendo testado, sendo `CompiledFile` o nome do programa compilado, e o conteúdo precedendo o `.`, o nome do pacote daquele programa.

RoadRunner permite ainda uma composição de ferramentas, resultando em uma cadeia que despacha eventos dinamicamente até que eles sejam testados pela ferramenta apropriada. Portanto, é possível encadear as execuções do FastTrack e Atomizer, identificando tanto corridas de dados quanto violações de atomicidade, por meio da chamada `rrrun -tool=FT2:A package.CompiledFile`.

4.3.3 Resultados

Uma vez instalada a ferramenta, foram efetuados alguns testes, para melhor análise de resultados e da saída apresentada. Começou-se testando o programa descrito nos códigos 2 e 3, exemplo de violação de atomicidade, explicado na seção 2.5.2.

Como mencionado na explicação da ferramenta, o Atomizer por padrão deduz quais blocos deveriam ser atômicos, considerando métodos públicos ou protegidos de classes, e blocos e métodos `synchronized`. Mas ele não leva em conta os métodos `main` e `run`. Portanto, no exemplo de atomicidade testado, a ferramenta não aponta nenhum erro, dado que a violação se encontra no método `run` da classe `RunThread`. Para lidar com esta situação, existem duas saídas: adicionar um método `synchronized` na classe `RunThread` que encapsule a operação atômica e chamar este método no `run`; ou adicionar uma nova classe `Bank` para lidar com esta lógica de débito, que é instanciada em `run`, e tem um método com esta operação atômica. Para fins de simplicidade, a alternativa apresentada é a primeira, presente no código 29. Além disto, os métodos de acesso ao atributo de saldo também foram colocados como `synchronized`, para que não haja interferência da corrida de dados nos resultados, como apresentado anteriormente no código 7 na seção 2.6.1.

Código 29 – Classe `RunThread` dos códigos 2 e 3 modificada para execução do Atomizer

```
class RunThread extends Thread {
    Account account;

    public RunThread(Account account) {
        this.account = account;
    }

    synchronized void debit(double amount) {
        if (account.getBalance() > amount) {
            account.decreaseBalanceBy(amount);
        }
    }

    public void run() {
        double amount = 100;
        this.debit(amount);
    }
}
```

Após estas alterações, o Atomizer ainda não apontou a violação. Isto ocorreu porque ele é uma ferramenta de **detecção**, e nas execuções do programa executadas por ele, ao chegar no instante da checagem do saldo da conta, a *thread* modificadora já havia alterado o saldo da conta. Portanto, neste cenário, a checagem sempre falhava, e o fluxo não entrava no `if`, consequentemente, não chegando no ponto em que existia uma violação. Em programas nos quais estas violações não estão em situações condicionais, este problema não existe. Mas ainda assim é uma limitação considerável da ferramenta.

Para que o Atomizer conseguisse detectar a violação, foi necessário introduzir mecanismos para fazer com que a checagem ocorresse antes da modificação, garantindo que o fluxo entrasse no bloco condicional. Com intuito de facilitar esta adaptação para fins de teste, utilizou-se `sleep`, para postergar a alteração no saldo feita pela *thread* modificadora e tentar garantir que a checagem ocorresse antes. Foi adicionado um `sleep(500)` antes da chamada de `account.setBalance(10)`, com intuito de fazer com que esta operação ocorresse após a checagem do saldo. Com esse ajuste, o Atomizer finalmente identificou a violação existente naquele bloco. Como mencionado nos resultados do FastTrack, a saída do RoadRunner é bem extensa, e grande parte é em formato XML, que fica salvo em um arquivo `log.xml` e pode ser traduzido para uma *tree view* ou um JSON, para ajudar na legibilidade.

Existem alguns pontos centrais para interpretar a saída do programa. No XML, as partes mais importantes estão ilustradas nos códigos 30 e 31. A seção mostrada no código 31 demonstra a quantidade de erros e o tipo deles, de forma análoga àquela vista no FastTrack. Já na seção mostrada no código 30, é apresentado o método que contém a

violação. No caso, é o método `debit` — que tem um `double` de parâmetro, representado por `D`, e retorno do tipo `void`, representado por `V` — da classe `RunThread`, do pacote `test`.

Código 30 – Informações geradas pelo Atomizer sobre o método violado no código 29

```
<methods>
    <method>
        <name> test/RunThread.debit(D)V(NullLoc) </name>
        <error>
            <name> Atomicity </name>
            <count> 1 </count>
        </error>
    </method>
</methods>
```

Código 31 – Informações geradas pelo Atomizer sobre os erros encontrados no código 29

```
<errorCountPerErrorType>
    <errorType>
        <name> Atomicity </name>
        <count> 1 </count>
    </errorType>
</errorCountPerErrorType>
```

O restante da saída também possui uma seção que descreve melhor o erro encontrado, mostrada no código 32. É apresentado o tipo do erro, informando que é um erro de atomicidade, e qual *thread* estava executando o bloco intencionalmente atômico, que no caso é a *thread* com identificador 1. Os identificadores são atribuídos de forma sequencial, portanto, a *thread* principal tem identificador 0, a primeira *thread* a ser criada a partir da principal tem identificador 1, e assim sucessivamente. Logo, neste caso a *thread* apresentada no *log* possui identificador 1, e é referente à *thread* `a` do programa, instância de `RunThread`, pois ela é a primeira criada no fluxo principal. Esta seção também aponta o método que teve sua atomicidade violada (*blame*), e apresenta a quantidade de violações que ocorreram (*count*).

Código 32 – Informações geradas pelo Atomizer sobre a violação de atomicidade detectada no código 29

```
## ============================================================================
## Atomicity Error
##           Thread: 1
##            Blame: test/RunThread.debit(D)V
##            Count: 1    (max: 100)
##      Enter Stack: <no stack>
##     Commit Cause: rel\_lock@test/Atom.java:11:-1
##     Commit Stack: <no stack>
##      Error Cause: acq\_lock@test/Atom.java:22:-1
##      Error Stack: <no stack>
## ============================================================================
```

No entanto, a parte mais importante desta seção do *log* é a causa do erro. Neste caso, é possível ver que o motivo do erro (*Error Cause*) foi uma aquisição de *lock*, na linha 22, que contém a declaração do método `synchronized void decreaseBalanceBy`. Mas esta aquisição só é um erro porque no instante em que ela ocorre, a *thread* 1 já está em um estado pós-comprometimento, como é possível averiguar pela informação de *Commit Cause*. Uma primeira liberação de *lock*, presente na linha 11, que contém o retorno do método `getBalance`, faz com que o fluxo se comprometa, entrando no estado pós-comprometimento, e, consequentemente, fazendo com que ações *non-movers* ou *right-movers* (como a aquisição de *lock* da linha 22) sejam ilegais, indicando uma violação.

Existem outros tipos de erros de violação que o Atomizer pode apresentar, mas a maioria está mais relacionado a corridas de dados, como acessar uma variável não protegida (uma ação *non-mover*) pós-comprometimento, ou acessar uma variável protegida sem alocar o *lock* correspondente. Todos os casos de violação de atomicidade testados caíram no mesmo erro de aquisição de *lock* na seção pós-comprometimento do bloco atômico.

Caso seja do interesse do aluno visualizar em maior detalhe o fluxo das *threads*, com a listagem de operações sendo executadas, é possível imprimir essa informação juntamente com o restante da saída do RoadRunner, utilizando a ferramenta de impressão `PrintTool` do framework por meio de uma composição com o Atomizer: `rrrun -tool=A:P package.CompiledFile`.

Rodando o teste na versão consertada deste programa, presente no código 7 explicado na seção 2.6.1, no qual não existe uma violação de atomicidade, a ferramenta corretamente indica que não existe erro:

Código 33 – Saída sem erro gerada pelo Atomizer para o código 7

```
<errorTotal> 0 </errorTotal>
<distinctErrorTotal> 0 </distinctErrorTotal>

<errorCountPerErrorType>
</errorCountPerErrorType>
```

4.3.4 Discussão

Após o estudo e testes da ferramenta, é possível averiguar que o Atomizer é uma implementação com fácil instalação e uso, além de ser bem eficiente no quesito tempo. Ela também se diferencia de muitas outras ferramentas de teste exatamente por propor uma solução para o problema mais complexo de identificar violações de atomicidade. O aluno consegue facilmente instalar os pré-requisitos e testar seus próprios programas, sem precisar de casos de teste ou anotações, inclusive também conseguindo usufruir do FastTrack no mesmo framework, o RoadRunner. Sua saída, apesar de verbosa, também é de fácil compreensão, uma vez entendida a estrutura geral das violações. Todos estes fatores tornam ela uma ótima opção para aplicação no ambiente acadêmico.

Apesar de todas as vantagens, também foi possível analisar as limitações da abordagem. Por não explorar intercalações alternativas, como uma técnica de predição faria, é possível que ela não identifique violações, caso a execução não entre dentro de blocos atômicos condicionais. Além disso, ela também não reconhece métodos `run` (ou `main`) como atômicos, forçando uma estruturação mais engessada aos programas dos alunos para uma verificação de atomicidade.

4.4 FERRAMENTAS EMBUTIDAS EM COMPILADORES E *TOOLKITS*

É interessante observar que já existem algumas ferramentas de detecção de falhas de concorrência disponíveis em alguns compiladores, *toolkits*, ou até mesmo embutidas na própria definição da linguagem. Estas ferramentas mais acessíveis simplicam bastante a etapa de teste de sistemas com um maior nível de garantias contra falhas de concorrência.

O Intel Inspector, por exemplo, é uma ferramenta disponível gratuitamente no *toolkit* Intel® oneAPI HPC voltada para aplicações C/C++ e Fortran, que detecta e localiza falhas de concorrência como **corridas de dados** e ***deadlocks***, por meio de análises de acessos a memória (INTEL, 2021).

Já os compiladores Clang e GCC possuem uma ferramenta chamada ThreadSanitizer (também conhecida como TSan), disponível a partir da versão 3.2 do Clang, e da versão 4.8 do GCC, que pode pode ser usada compilando e ligando o programa com a flag `-fsanitize=thread` (VYUKOV; SAMSONOV; POTAPENKO, 2020). O Tsan detecta corridas de dados e oferece suporte a primitivas de sincronização da biblioteca *pthread*

(KERRISK, 2021), validando a existência de corridas de dados considerando estas sincronizações.

Também existe o Helgrind (DEVELOPERS, 2021a), uma ferramenta Valgrind (DEVELOPERS, 2021b) para detecção de falhas de sincronização em programas C/C++ e Fortran, que é utilizada por meio da especificação da *flag* `-tool=helgrind` na linha de comando Valgrind. O Helgrind verifica estas falhas levando em conta as primitivas de sincronização da biblioteca *pthread*, sendo, portanto, compatível com programas que utilizem os mecanismos disponibilizados por esta biblioteca. Valgrind, por sua vez, é um framework de instrumentação para construção de ferramentas de análise dinâmica, que comumente realizam algum tipo de depuração, perfilamento (*profiling*) ou detecção de *leaks* de memória. Helgrind identifica três classes de falhas: **uso errôneo da API POSIX de pthread**, como liberação de um *mutex* inválido ou alocado por outra *thread*; ***deadlocks*** causados por problemas de ordenação de *locks*; e **corridas de dados**. Para identificar corridas de dados, a ferramenta utiliza análise *happens-before* para determinar relações de ordem, monitorando acessos à memória.

A linguagem de programação Go, por sua vez, possui um detector de corrida de dados embutido, chamado Data Race Detector, que só precisa da adição da *flag* `-race` ao comando Go para ser usado (PIKE et al., 2021).

Já a linguagem de programação Rust, em seu código padrão seguro (*Safe Rust*), garante a ausência de corrida de dados, por meio das funcionalidades básicas da linguagem, que não permitem a existência de múltiplas referências mutáveis a uma mesma variável. Em um código Rust concorrente, a única forma de múltiplas *threads* conseguirem acessar uma mesma variável mutável é encapsulando estes acessos em alguma estrutura segura para concorrência, como *mutex*, gerando um erro de compilação caso este encapsulamento não seja feito. Se um trecho do código for marcado como não-seguro (*Unsafe Rust*), esta garantia de ausência de corrida de dados é perdida (FOUNDATION, 2021b).

É possível perceber que todas estas ferramentas apresentadas possuem foco em identificar ou **corrida de dados**, ou ***deadlocks***, demonstrando que existe uma carência de técnicas voltadas para violações de atomicidade ou de ordem. Além deste fator, também é notável que ferramentas voltadas para testes concorrentes ainda não estão popularizadas e amplamente presentes em *frameworks*, compiladores ou *toolkits* de outras linguagens, como Java, consequentemente deixando este tipo de testagem menos acessível para desenvolvedores.

4.5 DISCUSSÃO

Ao longo deste capítulo, foram analisadas diversas ferramentas voltadas para testes de programas concorrentes. Foram apresentados seus conceitos, usabilidades, limitações e resultados. Para avaliar as respostas e resultados das ferramentas, utilizou-se programas

descritos no capítulo 2, com cenários contendo e não contendo erros lógicos. O projeto e funcionamento destas ferramentas foram descritos de forma breve, pois este não é o objetivo central deste trabalho. É possível encontrar um detalhamento mais completo nas referências de cada ferramenta abordada.

Foi possível observar que FastTrack possui diversas limitações de compatibilidade, não sendo responsivo com *locks* reentrantes ou semáforos, consequentemente reportando alarmes falsos, o que é extremamente prejudicial para o processo de aprendizado. Contudo, não apresenta problemas em programas com laços de muitas iterações, e, por estar no mesmo framework que o Atomizer, possui facilidade de encadeamento de uso.

Já o Atomizer é a única técnica explorada que é voltada para detectar violações de atomicidade, um problema pouco explorado por outras abordagens, sendo assim, interessante para ser utilizada. Sua maior limitação é o fato de ser uma abordagem dedutiva, então nem sempre aponta erro quando existe uma violação em um bloco condicional.

O RVPredict se mostrou a abordagem mais flexível, com saída mais clara e com menos resultados falsos, uma vez que é compatível com as primitivas de sincronização comuns no ambiente acadêmico, sendo, assim, a mais vantajosa para uso. Porém, apresentou problemas em programas com laços superiores a 25 iterações.

É relevante ressaltar que utilizar estas ferramentas e não obter nenhum resultado de erro não garante que o código está livre de problemas concorrentes. Ferramentas dedutivas podem simplesmente não ter entrado nos blocos condicionais com problema, ou podem existir outros erros de concorrência, como violação de ordem, que não são detectados por estar técnicas. Estas abordagens também podem indicar falsos positivos, muitas vezes por problemas de compatibilidade com os mecanismos de sincronização utilizados, sendo importante também se atentar a isto ao utilizar estas ferramentas no contexto de aprendizado. Contudo, apesar de não ofertarem esta garantia de corretude completa, seu uso ainda é muito importante, pois ajudam a identificar alguns erros não-determinísticos significativos e difíceis de detectar no desenvolvimento.

5 CONCLUSÃO

Computação concorrente é uma área de crescente relevância nos estudos modernos de computação, sendo utilizada em conjunto com muitas áreas, como inteligência artificial, computação científica e redes de computadores. Este crescimento da presença de concorrência no mundo computacional torna seu aprendizado cada vez mais importante no âmbito acadêmico.

A proposta deste trabalho foi identificar ferramentas didáticas para testes de programas concorrentes, investigando suas metodologias, usabilidade, limitações e resultados, a fim de determinar sua aptidão para apoiar o processo de ensino de concorrência.

Inicialmente, uma visão geral de concorrência foi apresentada, destacando seus desafios e sua suscetibilidade a erros. Então, foi apresentada a motivação de testes para programas concorrentes, visando detectar estes erros de concorrência e auxiliar no aprendizado dos conceitos da área . É notável que testes para programas concorrentes apresentam um nível muito maior de complexidade quando comparados com testes de programas sequenciais, devido ao não-determinismo do paradigma concorrente.

Foi utilizado como referência inicial o levantamento de abordagens de testes de programas concorrentes apresentado no artigo (BIANCHI; MARGARA; PEZZE, 2017). A classificação proposta pelos autores deste artigo base foi, então, usada para selecionar as ferramentas mais adequadas para o contexto de uma cadeira introdutória de programação concorrente.

Assim, no primeiro passo deste processo de seleção foram selecionadas apenas as abordagens voltadas para o uso de memória compartilhada. No segundo passo, foram selecionadas as abordagens que introduziam um conceito novo ou melhoria de acurácia (em relação a abordagens similares que as precederam), tinham artigos disponíveis e eram voltadas para programas concorrentes escritos em Java. No terceiro passo, o escopo foi reduzido para técnicas que possuem implementação disponível, uma vez que a proposta deste trabalho é exatamente identificar ferramentas para serem utilizadas por alunos. Foi encontrado um total de onze ferramentas disponíveis, um resultado muito positivo, dado que é uma quantidade bem superior à esperada inicialmente. O último passo de filtragem serviu para avaliar a usabilidade destas implementações, e selecionar técnicas com maior potencial para aplicação no ensino de concorrência. Ao final, restaram três ferramentas que foram analisadas e testadas mais detalhadamente: FastTrack, RVPredict e Atomizer, sendo as duas primeiras voltadas para identificar corridas de dados, e a última, violação de atomicidade.

O FastTrack apresentou a maior quantidade de limitações, mas foi a única técnica de detecção de corridas — dentre as duas analisadas — que lidou bem com programas contendo laços com muitas iterações. Contudo, sua saída é a mais complicada para en-

tendimento, e possui o agravante de reportar muitos erros falsos, devido a sua falta de compatibilidade com *locks* reentrantes e semáforos. Apesar desta desvantagem, a ferramenta ainda pode ser utilizada quando outros mecanismos de sincronização — como blocos ou métodos `synchronized` — estão sendo ensinados. Por fim, por ser uma ferramenta de dedução, pode não identificar falhas em blocos ramificados (como condicionais), se a execução corrente não entrar no bloco.

O RVPredict foi a ferramente mais flexível, e com menos limitações, se mostrando compatível com todas as primitivas de sincronização testadas, mas apresentando dificuldades para lidar com laços de muitas iterações, por ser preditiva. Sua saída também é a mais enxuta e facilmente legível dentre todas, e sua usabilidade é a mais simples.

O Atomizer, por sua vez, tem o diferencial e vantagem de ser a única técnica avaliada voltada para detectar violações de atomicidade, um problema não muito explorado. Este aspecto de tratar de um problema diferente faz dela muito interessante para uso em ensino, mesmo com suas limitações. Apresentou bons resultados, e uma saída compreensível, porém, analogamente ao FastTrack, por não ser preditiva, pode não identificar erros em blocos ramificados (como condicionais).

Dentre as ferramentas para identificar corridas de dados, o RVPredict se mostrou a mais apta para uso no ensino de concorrência, devido a todas as limitações apresentadas do FastTrack. Contudo, como cada uma destas técnicas possuem suas restrições e focos, é interessante utilizá-las de forma complementar, sempre com uma visão crítica para avaliar as características do programa sendo testado, determinando quais ferramentas fazem sentido para cada contexto.

Este estudo também demonstra que esta área de testes para programas concorrentes, principalmente com foco em ambiente acadêmico, ainda tem muito a evoluir, e possui muitas outras vertentes relevantes que podem ser exploradas de forma a complementar e estender os resultados aqui apresentados.

Foi perceptível que as saídas de alguns programas não eram muito objetivas e de fácil compreensão. Esta característica também se repete para muitas das ferramentas de teste que não foram aprofundadas neste estudo. Existe, assim, a oportunidade de se desenvolver programas auxiliares que processem as saídas de uma ou mais destas ferramentas, a fim de deixá-las mais compreensíveis e claras. Também seria interessante explorar mais a fundo recursos de linguagens voltadas para prevenção de erros de concorrência, como aqueles presentes em Rust e Go, e analisar suas garantias.

É possível ainda complementar este estudo analisando outras ferramentas que foram listadas, mas não selecionadas para estudo, principalmente abordagens voltadas para C, que é uma linguagem de extrema importância no ensino de concorrência — como Maple (YU et al., 2012), e o RVPredict para C. Ou até mesmo implementar as técnicas exploradas — como FastTrack e Atomizer — para trabalhar com programas feitos em C.

REFERÊNCIAS

ARORA, V.; BHATIA, R.; SINGH, M. A systematic review of approaches for testing concurrent programs. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 28, n. 5, p. 1572–1611, 2016.

ARPACI-DUSSEAU, R. H.; ARPACI-DUSSEAU, A. C. **Operating systems: Three easy pieces**. [S.l.]: Arpaci-Dusseau Books LLC, 2018.

BIANCHI, F. A.; MARGARA, A.; PEZZE, M. A survey of recent trends in testing concurrent software systems. **IEEE Transactions on Software Engineering**, IEEE, v. 44, n. 8, 2017.

BRITO, M. A. et al. Concurrent software testing: a systematic review. São Carlos, SP, Brasil., 2010.

BRYANT, R. E.; RICHARD, O. D. **Computer systems: a programmer's perspective**. [S.l.]: Prentice Hall Upper Saddle River, 2003. v. 2.

CAI, Y.; CAO, L. Effective and precise dynamic detection of hidden races for java programs. In: **Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering**. [S.l.: s.n.], 2015. p. 450–461.

CHEN, F.; SERBANUTA, T.; ROSU, G. Jpredictor. In: IEEE. **2008 ACM/IEEE 30th International Conference on Software Engineering**. [S.l.], 2008. p. 221–230.

CHEN, J.; MACDONALD, S. Testing concurrent programs using value schedules. In: **Proceedings of the twenty-second IEEE/ACM international conference on automated software engineering**. [S.l.: s.n.], 2007. p. 313–322.

CHOI, J.-D. et al. Efficient and precise datarace detection for multithreaded object-oriented programs. In: **Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation**. [S.l.: s.n.], 2002. p. 258–269.

DEVELOPERS, V. **Helgrind: a thread error detector - Valgrind**. 2021. https://valgrind.org/docs/manual/hg-manual.html. [Online; acesso em 15 de Outubro de 2021].

DEVELOPERS, V. **Valgrind**. 2021. https://valgrind.org/docs/manual/manual.html. Online; acesso em 15 de Outubro de 2021.

FLANAGAN, C.; FREUND, S. N. Atomizer: a dynamic atomicity checker for multithreaded programs. **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 39, n. 1, p. 256–267, 2004.

FLANAGAN, C.; FREUND, S. N. Fasttrack: efficient and precise dynamic race detection. **ACM Sigplan Notices**, ACM New York, NY, USA, v. 44, n. 6, p. 121–133, 2009.

FLANAGAN, C.; FREUND, S. N. The roadrunner dynamic analysis framework for concurrent programs. In: **Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering**. [S.l.: s.n.], 2010. p. 1–8.

FLANAGAN, C.; FREUND, S. N. **The fasttrack2 race detector**. [S.l.], 2017.

FLANAGAN, C.; FREUND, S. N. **RoadRunner - Github**. 2017. https://github.com/stephenfreund/RoadRunner. [Online; acesso em 20 de Outubro de 2021].

FOUNDATION, A. S. **Apache Ant**. 2021. https://ant.apache.org/. Online; acesso em 12 de Novembro de 2021.

FOUNDATION, T. R. **Races - The Rustonomicon**. 2021. https://doc.rust-lang.org/nomicon/races.html. [Online; acesso em 15 de Outubro de 2021].

GANAI, M. K. Scalable and precise symbolic analysis for atomicity violations. In: IEEE. **2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)**. [S.l.], 2011. p. 123–132.

HAVELUND, K. Using runtime analysis to guide model checking of java programs. In: SPRINGER. **International SPIN Workshop on Model Checking of Software**. [S.l.], 2000. p. 245–264.

HERLIHY, M. et al. **The art of multiprocessor programming**. [S.l.]: Newnes, 2020.

HUANG, J.; LUO, Q.; ROSU, G. Gpredict: Generic predictive concurrency analysis. In: IEEE. **2015 IEEE/ACM 37th IEEE International Conference on Software Engineering**. [S.l.], 2015. v. 1, p. 847–857.

HUANG, J.; MEREDITH, P. O.; ROSU, G. Maximal sound predictive race detection with control flow abstraction. In: **Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation**. [S.l.: s.n.], 2014. p. 337–348.

HUANG, J.; ZHANG, C. Persuasive prediction of concurrency access anomalies. In: **Proceedings of the 2011 International Symposium on Software Testing and Analysis**. [S.l.: s.n.], 2011. p. 144–154.

IEEE. Ieee standard glossary of software engineering terminology. **IEEE Std 610.12-1990**, 1990.

INC, R. V. **RV-Predict**. 2021. https://runtimeverification.com/predict/. [Online; acesso em 20 de Outubro de 2021].

INC, R. V. **RV-Toolkit Java**. 2021. https://runtimeverification.github.io/rv-toolkit-docs/predict/java/. [Online; acesso em 20 de Outubro de 2021].

INTEL. **Intel® Inspector**. 2021. https://www.intel.com/content/www/us/en/developer/tools/oneapi/inspector.html#gs.dq13rz. [Online; acesso em 15 de Outubro de 2021].

JOSHI, P. et al. A randomized dynamic program analysis technique for detecting real deadlocks. **ACM Sigplan Notices**, ACM New York, NY, USA, v. 44, n. 6, p. 110–120, 2009.

KERRISK, M. **Linux manual page - pthreads**. 2021. https://man7.org/linux/man-pages/man7/pthreads.7.html. Online; acesso em 14 de Novembro de 2021.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. In: **Concurrency: the Works of Leslie Lamport**. [S.l.: s.n.], 2019. p. 179–196.

LIPTON, R. J. Reduction: A method of proving properties of parallel programs. **Communications of the ACM**, ACM New York, NY, USA, v. 18, n. 12, p. 717–721, 1975.

LU, S. et al. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: **Proceedings of the 13th international conference on Architectural support for programming languages and operating systems**. [S.l.: s.n.], 2008. p. 329–339.

MANKIN, J. **CSG280: Parallel Computing Memory Consistency Models: A Survey in Past and Present Research**. [S.l.]: Citeseer, 2007.

MATTERN, F. et al. **Virtual time and global states of distributed systems**. [S.l.]: Univ., Department of Computer Science, 1988.

NETZER, R. H.; MILLER, B. P. What are race conditions? some issues and formalizations. **ACM Letters on Programming Languages and Systems (LOPLAS)**, ACM New York, NY, USA, v. 1, n. 1, p. 74–88, 1992.

ORACLE. **Lesson: Concurrency (The Java™ Tutorials > Essential Java Classes)**. 2021. https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html. [Online; acesso em 30 de Setembro de 2021].

PARK, C.-S.; SEN, K. Randomized active atomicity violation detection in concurrent programs. In: **Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering**. [S.l.: s.n.], 2008. p. 135–145.

PIKE, R. et al. **Data Race Detector - The Go Programming Language**. 2021. https://golang.org/doc/articles/race_detector. [Online; acesso em 15 de Outubro de 2021].

ROY, P. V. et al. Programming paradigms for dummies: What every programmer should know. **New computational paradigms for computer music**, IRCAM/Delatour France, v. 104, p. 616–621, 2009.

SAMAK, M.; RAMANATHAN, M. K. Multithreaded test synthesis for deadlock detection. In: **Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications**. [S.l.: s.n.], 2014. p. 473–489.

SAMAK, M.; RAMANATHAN, M. K. Omen+: A precise dynamic deadlock detector for multithreaded java libraries. In: **Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering**. [S.l.: s.n.], 2014. p. 735–738.

SAMAK, M.; RAMANATHAN, M. K. Trace driven dynamic deadlock detection and reproduction. **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 49, n. 8, p. 29–42, 2014.

SAMAK, M.; RAMANATHAN, M. K. Synthesizing tests for detecting atomicity violations. In: **Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering**. [S.l.: s.n.], 2015. p. 131–142.

SAMAK, M.; RAMANATHAN, M. K.; JAGANNATHAN, S. Synthesizing racy tests. **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 50, n. 6, p. 175–185, 2015.

SAVAGE, S. et al. Eraser: A dynamic data race detector for multithreaded programs. **ACM Transactions on Computer Systems (TOCS)**, ACM New York, NY, USA, v. 15, n. 4, p. 391–411, 1997.

SEN, K. Race directed random testing of concurrent programs. In: **Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation**. [S.l.: s.n.], 2008. p. 11–21.

SEN, K. et al. **CalFuzzer - Github**. 2014. https://github.com/ksen007/calfuzzer. Online; acesso em 10 de Outubro de 2021.

SHACHAM, O.; SAGIV, M.; SCHUSTER, A. Scaling model checking of dataraces using dynamic information. **Journal of Parallel and Distributed Computing**, Elsevier, v. 67, n. 5, p. 536–550, 2007.

SINGHAL, M. Deadlock detection in distributed systems. **Computer**, IEEE, v. 22, n. 11, p. 37–48, 1989.

VYUKOV, D.; SAMSONOV, A.; POTAPENKO, A. **ThreadSanitizer Cpp Manual - Github**. 2020. https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual. [Online; acesso em 15 de Outubro de 2021].

YU, J. et al. Maple: A coverage-driven testing tool for multithreaded programs. In: **Proceedings of the ACM international conference on Object oriented programming systems languages and applications**. [S.l.: s.n.], 2012. p. 485–502.