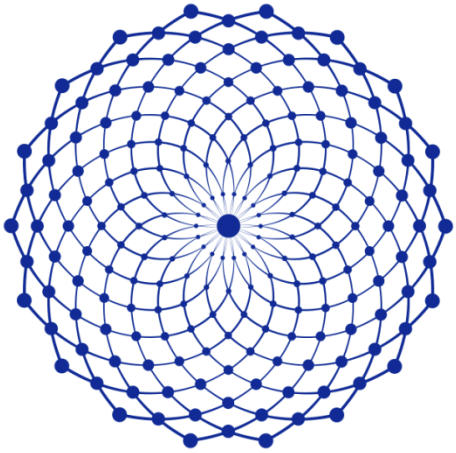

MODELAGEM COMPUTACIONAL DE CONHECIMENTO

Dissertação de Mestrado

Um Método para Verificação Formal e Dinâmica de Sistemas de Software Concorrentes

Bruno Roberto Santos

Orientador:

Leandro Dias da Silva

Coorientador:

Márcio de Medeiros Ribeiro

Maceió, Maio de 2016

Bruno Roberto Santos

Um Método para Verificação Formal e Dinâmica de Sistemas de Software Concorrentes

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Curso de Mestrado em Modelagem Computacional de Conhecimento do Instituto de Computação da Universidade Federal de Alagoas

Orientador:

Leandro Dias da Silva

Coorientador:

Márcio de Medeiros Ribeiro

Maceió, Maio de 2016

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Bibliotecária Responsável: Janaina Xisto de Barros Lima

S237u Santos, Bruno Roberto.
Um método para verificação formal e dinâmica de sistemas de software concorrentes / Bruno Roberto Santos. - 2016.
57 f. : il.

Orientador: Leandro Dias da Silva.
Coorientador: Márcio de Medeiros Ribeiro.
Dissertação (mestrado em Modelagem Computacional de Conhecimento) – Universidade Federal de Alagoas. Instituto de Computação. Maceió, 2015.

Bibliografia: f. 55-57.

1. Verificação formal. 2. Sistemas concorrentes. 3. Análise dinâmica. I. Título.

CDU: 004.032.24

UNIVERSIDADE FEDERAL DE ALAGOAS/UFAL
Programa de Pós-Graduação em Modelagem Computacional de Conhecimento
Avenida Lourival Melo Mota, Km 14, Bloco 09, Cidade Universitária
CEP 57.072-900 – Maceió – AL – Brasil
Telefone: (082) 3214-1364

Membros da Comissão Julgadora da Dissertação de Mestrado de Bruno Roberto Santos, intitulada: "Um Método para Verificação Formal e Dinâmica de Sistemas de Software Concorrentes", apresentada ao Programa de Pós-Graduação em Modelagem Computacional de Conhecimento da Universidade Federal de Alagoas, em 20 de maio de 2016, às 14h00min, auditório do Instituto de Computação da Ufal.

COMISSÃO JULGADORA

Prof. Dr. Leandro Dias da Silva
Ufal – Instituto de Computação
Orientador

Prof. Dr. Márcio de Medeiros Ribeiro
Ufal – Instituto de Computação
Coorientador

Prof. Dr. Patrick Henrique da Silva Brito
Ufal – Instituto de Computação
Examinador

Prof. Dr. Angelo Perkusich
UFCG – Departamento de Engenharia Elétrica
Examinador

Maceió, maio de 2016.

AGRADECIMENTOS

Primeiramente, agradeço a Deus por me conceder as oportunidades e a força necessária para superar todos os desafios. Agradeço à minha família por todo o apoio, confiança e exemplo dado por toda vida. À minha amada esposa, Mayara Jordana, por toda compreensão e incentivo em todos os momentos. Aos meus orientadores, Leandro Dias e Márcio Ribeiro, pela paciência e contribuição em minha vida acadêmica. Agradeço também a todos os amigos que de alguma forma cooperaram para a conclusão desta jornada.

RESUMO

Neste trabalho é apresentado um método para verificação formal e dinâmica de *software* concorrentes. O objetivo é oferecer um método capaz de identificar problemas inerentes a programas cuja execução baseia-se em múltiplas *threads*, além de analisar propriedades comportamentais descritas com base nos preceitos da lógica temporal. Propõe-se um método capaz de detectar problemas e verificar formalmente a adequação da execução de sistemas de software concorrentes com relação ao comportamento desejável a tais sistemas, baseando-se em informações coletadas dinamicamente, ou seja, em tempo de execução. As informações coletadas correspondem às sequências de execução de sistemas de software, bem como dados sobre a maneira como se comunicam seus componentes durante sua execução. Os dados colhidos refletem a execução do sistema de software propriamente dito, o que garante maior confiança às informações coletadas. Tais informações são analisadas de modo a identificar impasses e condições de corrida em um processo denominado **Análise Dinâmica**. Ademais, estas informações também são utilizadas para geração automática de um modelo que descreve o comportamento do sistema de software, o qual é utilizado para verificação de propriedades comportamentais. A este processo de verificação dá-se o nome de **Verificação Formal**. A geração automática do modelo elimina a necessidade de construção manual do mesmo, que requer muito esforço e conhecimento acerca de métodos formais, isso pode aumentar custos e tempo de desenvolvimento do sistema de software. Entretanto, a análise dinâmica é conhecida por apenas realizar cobertura sobre o comportamento atual de sistemas de software concorrentes, sem considerar a análise de todas as outras possíveis sequências de execuções devido ao não determinismo. Em razão do comportamento não determinístico, sistemas de software concorrentes são capazes de produzir resultados diferentes para a mesma entrada a cada nova execução. Deste modo, reproduzir o comportamento que leva sistemas de software concorrente à falha é uma tarefa complexa. O presente trabalho propõe um método para realizar verificação formal e dinâmica de sistemas de software concorrente capaz de capturar o comportamento não determinístico desses sistemas, além de proporcionar a redução de custos de desenvolvimento através da eliminação da necessidade de construção manual de modelos de sistemas de software concorrente. O método é validado através de um estudo de caso composto por testes em três sistemas de software.

ABSTRACT

This work presents a method to perform formal and dynamic verification of concurrent software. The objective is to provide a method capable of identifying problems in programs whose execution is based on multiple threads, and analyze behavioral properties. The method is able to detect problems in concurrent software, as well as check conformity of the concurrent software with desirable behavior, based on information collected dynamically, i.e. at runtime. The information collected consists of the software execution flow as well as data about the way communicate the software components during this run. The data collected reflect the software's execution, which ensures greater confidence to the information collected. This information is analyzed to identify deadlocks and race conditions in a process called ***Dynamic Analysis****. In addition, this information is also used to automatically generate a model that describes the behavior of a software, which is used for verification of behavioral properties. This process is called* ***Formal Verification****. The automatic model generation eliminates the need for manual construction of the model, which requires much effort and knowledge of formal methods, this can increase costs and development time software. However, the dynamic analysis is known to only perform coverage of the current behavior of competing software systems. Current behavior is one that occurs only during an execution of concurrent software systems, without considering all other possible behaviors from the non-determinism. Due to the non-determinism, concurrent software can produce different results for the same input to each execution of software. Therefore reproduce the behavior that leads to competitive software failure is a complex task. This paper proposes a method to perform formal verification and dynamic concurrent software capable of capturing the non-deterministic behavior of these systems and provide reduced development costs by eliminating the need for manual construction of concurrent software system models. The method is validated by a case study consists of three test software systems.*

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A programação concorrente é um paradigma de programação que proporciona aos sistemas de *software* a capacidade de executar múltiplas tarefas de maneira concorrente de forma que seja reduzida a ociosidade do processador (BUSTARD, 1990). Um sistema de software concorrente consiste de dois ou mais programas sequenciais que devem ser executados concorrentemente como processos paralelos (BUSTARD, 1990), deste modo, o processador pode realizar diversas operações praticamente ao mesmo tempo. Entretanto, os recursos para a execução de sistemas de *software* concorrentes podem ser compartilhados entre os diversos programas sequenciais que o compõem, de maneira que há necessidade de um mecanismo que gerencie o acesso a tais recursos. Os programas sequenciais executam de maneira não determinística, desta maneira não há como prever qual será o próximo programa a ser executado. Isso significa que, para uma mesma entrada, várias execuções de um mesmo sistema de *software* concorrente poderão retornar resultados diferentes. Por essa razão, sistemas de *software* construídos sob esse paradigma são substancialmente mais complexos (AVIRAM, WENG, *et al.*, 2012).

As características supracitadas aumentam a complexidade de desenvolvimento de sistemas de software concorrentes (ALGLAVE, KROENING e TAUTSCHNIG, 2013) (SINHA e WANG, 2010). Defeitos como, por exemplo, **impasse** (ISLOOR, 1980) (AGARWAL, 2006) e **condições de corrida** (CHEN, 2002) (BIN LEI, 2008) (POZNIANSKY, 2007) podem provocar, tanto a paralisação de sistemas de software, como ocasionar a entrega de resultados inconsistentes. Entre os anos de 1985 e 1987, um acidente com a máquina de radioterapia Therac-25 resultou em seis mortes (LEVESON, 1993) (LEVESON, 1995). A razão foi a ocorrência de *race conditions* no software que controlava o equipamento. Em 1997, a missão espacial *Mars Pathfinder* tinha o intento de colher informações sobre o planeta Marte com o auxílio do robô explorador *Sojourner*. Entretanto, um erro conhecido como inversão de prioridades (WELC, 2004) (O. BABAOGLU, 1993), característico de sistemas de *software* concorrentes, resultou em perda dos dados coletados.[1]

As consequências provocadas por sistema de *software* defeituoso também são desastrosas para a indústria. Segundo o relatório do *National Institute of Standard & Technology (NIST)* (TASSEY, 2002), empresas americanas arcavam com um prejuízo de

[1] http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html

cerca de US$59,5 bilhões devido ao uso de sistemas de *software* defeituosos. Posteriormente, em 2005, o artigo proposto por (CHARETTE, 2005) revelava um gasto entre US$60 e US$70 bilhões em manutenção e correção de sistemas de *software* problemáticos. No mesmo ano em (HILDRETH, 2005), a revista *Computerworld* confirmou que sistemas de *software* incorretos podem resultar em desperdício de horas de trabalho e de oportunidades de venda, o alto custo de suporte e manutenção, baixa satisfação do cliente, em razão de sistemas de *software* incorretos. Em (KEIL, NEUMANN, *et al.*, 2002), foi listado pela revista *Computerworld* dez grandes prejuízos da indústria americana em virtude de projetos de baixa qualidade nos anos 90, evidenciando o problema do *software* mal feito. Mais recentemente, em 2012, a empresa *Knight Capital Group (KCG)* sofreu um prejuízo de aproximadamente US$ 440 milhões em 30 minutos (BLOOMBERG, 2012). O caso ocorreu devido a uma falha no algoritmo de negociações da empresa, o software realizou um volume de negócios muito acima do permitido. Em 2011, a Honda realizou a notificação de 2,26 milhões de veículos em razão de problemas com o software de transmissão automática (JOURNAL, 2011). O erro poderia ocasionar a quebra da transmissão, colocando a vida dos motoristas em risco. De 2007 a 2009, um erro em um software financeiro causou um prejuízo de US$ 217 milhões a investidores.

Tendo em vista a complexidade adicional quando da construção de sistemas de *software* concorrentes, bem como os problemas a que estão sujeitos, fica evidente a necessidade de métodos que possam reduzir a possibilidade de falhas em tais sistemas. A fim de verificar se programas realmente funcionam de acordo com o especificado, estratégias convencionais de teste, tais como, testes unitários, teste de carga, foram desenvolvidas e amadurecidas ao longo dos anos (BERTOLINO, 2007) (BINDER, 1999). Basicamente, com a realização de testes de software, propõe-se observar o comportamento do programa em teste para determinar se o mesmo funciona de acordo com o especificado e identificar possíveis defeitos (BERTOLINO, 2007). Entretanto, em razão do comportamento não determinístico, as estratégias convencionais de teste podem não ser efetivas quando aplicadas em sistemas de *software* concorrentes. Em sistemas de *software* concorrentes, sequências de execução podem se manifestar em algumas execuções do sistema ou desaparecer em outras execuções. Dessa forma, defeitos podem ser expostos em uma sequência de execução com baixa probabilidade de ocorrência (LEE, 2006). Tal comportamento torna inviável a utilização de estratégias convencionais (EDELSTEIN, FARCHI, *et al.*, 2002).

Para testar sistemas de *software* concorrentes é preciso observar todas as suas possíveis sequências de execução, de modo que proporcione a análise de sequências de execução com alta e baixa probabilidade de ocorrência. Nesse contexto, a técnica de **verificação de modelos** revela-se uma alternativa para verificar sistemas de *software* não determinísticos (BAIER e KATOEN, 2008) (DIMITRA GIANNAKOPOULO, 2002) (SOUSA, 2012). A técnica permite que seja gerado um modelo do sistema de *software*, o qual deve descrever seu comportamento. Assim, todas as possíveis sequências de execução de um sistema de *software* concorrente podem ser representadas no modelo. Este modelo é confrontado com propriedades que prescrevem o que um sistema deve fazer ou não. Um sistema é considerado correto se seu modelo satisfaz às propriedades especificadas. A Verificação de modelos é uma abordagem de verificação formal de sistemas (JIM WOODCOCK, 2009) (JOSÉ BACELAR ALMEIDA, 2011) (SOUSA, 2012). Ou seja, ela fornece a prova algorítmica de conformidade com as propriedades especificadas acerca de algum sistema de software ou *hardware*. Assim, a técnica de verificação de modelos é capaz de realizar cobertura de todo o comportamento de um sistema de software concorrente e, deste modo, possibilitar a identificação de falhas.

A capacidade de representar todas as possíveis sequências de execução de sistemas de *software* concorrente pode auxiliar a identificação de impasses e condições de corrida. Isto é, as sequências de execução exploradas para verificação sobre o comportamento podem ser analisadas de forma que seja possível identificar a presença de impasses e condições de corrida.

No entanto, a utilização de verificação de modelos envolve a construção de um modelo que descreva o comportamento do software em teste. Tal atividade requer muito esforço e conhecimento acerca de métodos formais. Métodos formais consistem na utilização de modelos matemáticos para analisar e verificar qualquer parte de um programa ou hardware (JIM WOODCOCK, 2009). Segundo (GABBAR, 2006), o esforço para a realização da modelagem é o principal impedimento para utilização da abordagem. Ainda de acordo com (GABBAR, 2006), há resistência em se utilizar modelos matemáticos para verificação de sistemas de software em virtude da dificuldade de aprendizado da técnica, e que por essa razão a abordagem não se tornou popular.

Diante deste contexto, um método que possibilite a combinação da técnica de verificação de modelos com algoritmos de detecção de impasses e condições de corrida

proporcionaria maior capacidade de cobertura sobre o comportamento não determinístico de sistemas de software concorrentes. Além disso, um método capaz de gerar o modelo do sistema de software automaticamente poderia reduzir, significativamente, os esforços empreendidos no processo de desenvolvimento de sistemas de software.

1.1 Descrição do problema

Tendo em vista a busca por desempenho, programas concorrentes estão cada vez mais comuns. A evolução do *hardware* é um fator preponderante para utilização da concorrência (GROGONO e SHEARING, 2008). Os processadores *multicore*, os *terabytes* de memória, e todos os outros avanços do mundo do *hardware*, permitem que uma extensa quantidade de operações seja realizada a incríveis velocidades. Tal progresso requer a construção de softwares compatíveis com tal capacidade de desempenho.

Entretanto, é sabido que a programação concorrente introduz complexidade ao processo de desenvolvimento de *software*. Reduzir a possibilidade de erros em programas concorrentes é essencial, mas não é simples. A automação da análise e verificação de sistemas de *software* tem sido a estratégia proposta para testar tais programas. Isto é, métodos capazes de realizar todo o processo de análise e verificação com o mínimo de interação humana. Basicamente, estes métodos consistem em capturar, automaticamente, informações do programa em teste (MUSUVATHI, 2007) (VISSER, 2000) (ARTHO e BIERE, 2001) (BERGHOFER, NIPKOW, *et al.*, 2009) e, posteriormente, fazer uso dessas informações para a análise e verificação do sistema de *software*. Dessa forma, o analista de teste fica livre da tarefa de modelagem, reduzindo o esforço na atividade de verificação.

Existem duas categorias de ferramentas de análise de sistemas de *software* concorrente: as que o fazem de maneira estática (ARTHO e BIERE, 2001); e as que realizam a análise dinâmica (OLIVEIRA, 2006). As ferramentas de análise estática extraem as informações do programa sem a necessidade de execução do mesmo (D'SILVA e WEISSENBACHER, 2008). Tais informações referem-se às sequências de execução de sistemas de *software* concorrentes, bem como dados sobre a maneira como se comunicam seus componentes durante essa execução. Os dados coletados são processados em algoritmos que buscam identificar falhas como *deadlock* e *race conditions*. Em razão de não necessitar da execução do sistema de software, as ferramentas de análise estática são consideradas muito velozes (ARTHO e BIERE, 2001). Entretanto, ao extrair informações estaticamente, a

ferramenta deixa de capturar o comportamento exato do programa em teste, sendo necessários mecanismos para inferir sequências de execução. Dessa forma, a análise estática é propensa a revelar falsos alarmes. A cobertura sobre todas as sequências de execução possíveis é baseada em predições sobre comportamentos futuros do software em teste. Entretanto, predições podem ser imprecisas, o que pode culminar na exposição de defeitos que na verdade não ocorrem, ou mesmo deixar de detectar defeitos.

Por outro lado, as ferramentas que realizam análise dinamicamente extraem informações durante a execução do programa, característica que as tornam mais lentas que as ferramentas de análise estática (ARTHO e BIERE, 2001). Porém, os resultados revelados por essas ferramentas são confiáveis, uma vez que capturam o comportamento exato do programa testado. Entretanto, apenas revelam erros referentes à sequência de execução corrente, não sendo capazes de capturar todas as possíveis sequências de execução de sistemas de *software* concorrentes. Dessa forma, erros podem passar despercebidos em sequências de execução que raramente se manifestam. A estratégia deve ser capaz de cobrir todo comportamento de um programa concorrente.

Tendo em vista as características das abordagens de análise de programas concorrentes, observa-se certa limitação quanto à cobertura do comportamento não determinístico desses programas. Nota-se também que esta limitação é fator preponderante na atividade de detecção de defeitos, podendo ocasionar a emissão de falsos positivos e falsos negativos. Deste modo, se faz necessário o desenvolvimento de métodos capazes de prover maior confiança às informações coletadas, de modo que elas reflitam ao máximo o comportamento de software concorrente.

1.2 Objetivo

O objetivo neste trabalho é propor um método para identificar impasses e condições de corrida em sistemas de *software* concorrentes, bem como realizar verificação de modelos. A proposta consiste em um método capaz de observar todas as possíveis sequências de execução de um sistema de *software* concorrente. O método permite que o código fonte de um sistema de *software* concorrente seja instrumentado de maneira que durante sua execução, informações sobre o sistema sejam monitoradas, coletadas e utilizadas para análise quanto à existência de sequências de execução alternativas que necessitam ser verificadas. As sequências de execução alternativas representam o comportamento do sistema de *software*

concorrente que não se manifestou durante sua execução. Tais sequências de execução são utilizadas tanto para verificação quanto a ocorrência de impasses, condições de corrida, como para geração e verificação automática do modelo do sistema. Deste modo, com o presente trabalho, propõe-se um método que possibilita a redução do esforço para construção de modelos, bem como proporciona a redução da quantidade de emissões de falsos positivos e falsos negativos.

Um método semelhante foi proposto em (VASCONCELOS, 2012) e (VASCONCELOS, SILVA e PERKUSICH, 2012), e consiste em coletar dados durante a execução de sistemas embarcados e, automaticamente, gerar um modelo que pode ser verificado contra propriedades elaboradas sob os conceitos de lógica temporal. A dinâmica do método proposto neste trabalho é a mesma, a diferença está no fato de que este é voltado para análise e verificação de sistemas de software concorrentes, além de possibilitar a busca por impasses e condições de corrida.

Basicamente, a proposta consiste em combinar as técnicas de análise dinâmica e verificação de modelos. Com a análise dinâmica, pretende-se buscar informações mais confiáveis acerca do comportamento do sistema de *software*, uma vez que, através da técnica, é possível buscá-las durante sua execução, ao passo que, com a verificação de modelos objetiva-se utilizar a capacidade de representação de todas as possíveis sequências de execução de um sistema de *software* concorrente para realizar cobertura sobre seu comportamento.

O método é proposto em forma de um *plug-in* Eclipse[2] e dá suporte a sistemas de *software* concorrentes escritos em Java[3] e aos sistemas voltados para a plataforma Android.[4] O suporte à linguagem Java e à plataforma Android decorre da vasta quantidade de sistemas de *software* construídos para estas plataformas.

1.3 Trabalhos Relacionados

Na presente seção são apresentados alguns trabalhos relacionados. Nestes trabalhos são propostas ferramentas capazes de realizar verificação de sistemas de *softwares* concorrentes a partir da extração automática de informações dos mesmos. Investigar tais

[2] https://www.eclipse.org/
[3] http://www.oracle.com/br/technologies/java/overview/index.html
[4] http://www.android.com

trabalhos é importante para que se entenda a relevância do tema no cenário da engenharia de software. Os projetos aqui encontrados utilizam algum formalismo matemático para garantir a correção dos programas testados, entretanto, não realizam monitoramento sobre todas as possíveis sequências de execução dos sistemas de *software* concorrentes, fator motivador para esta comparação.

1.3.1 Jlint

Jlint (ARTHO e BIERE, 2001) é um verificador de programas escritos na linguagem Java. As informações necessárias para a verificação são extraídas automaticamente em tempo de compilação, ou seja, Jlint verifica sistemas de softwares de forma estática. Os dados coletados referem-se às características das classes e seus relacionamentos. Tais dados são utilizados para gerar um grafo de dependência do sistema de software. Todos os caminhos do grafo são exaustivamente testados a fim de descobrir problemas relacionados à concorrência tais como impasse, condições de corrida, além de detectar violações de herança e de limites de um determinado *array*.

Por ser uma ferramenta que realiza verificação estática, Jlint apresenta bom desempenho. Além disso, a aplicação não requer anotações no código fonte, eliminando o custo de inseri-las em todo o sistema, e principalmente, o esforço de aprender sua sintaxe e semântica. Contudo, o fato de Jlint capturar informações em tempo de compilação o torna menos preciso. Isto é, em (ARTHO e BIERE, 2001) constatou-se que em certos casos, a ferramenta não pôde deduzir suficientemente o contexto de execução do código fonte. Isso significa que os grafos gerados nessas situações não reproduzem fielmente o fluxo de execução dos sistemas, o que reduz a eficácia de Jlint. Em outros casos, o modelo construído é muito simples, incorrendo em dificuldades na verificação de algumas propriedades, como condições de corrida por exemplo.

Por outro lado, apesar de apresentar desempenho inferior, o método proposto neste trabalho realiza verificações com dados mais confiáveis. Isso decorre de sua capacidade de coletar informações em tempo de execução. Tais informações descrevem com maior confiança o comportamento do software, o que proporciona a geração de melhores resultados.

1.3.2 VCC: *Verifying Concurrent* C

O VCC (BERGHOFER, NIPKOW, *et al.*, 2009) (COHEN, MOSKAL, *et al.*, 2010) é um verificador estático de sistemas de softwares escritos na linguagem C. A ferramenta faz uso dos conceitos inerentes à técnica **Design by Contract (DBC)** (MEYER, 1992), e os aplica a sistemas concorrentes. A ideia é introduzir assertivas (**pré-condições**, **pós-condições** e **invariantes**) no código fonte, e através de um formalismo matemático provar que estas são verdadeiras ou falsas para todas as possíveis sequências de execução.

Cada assertiva é verificada em um escopo diferente. Uma pré-condição consiste em uma propriedade que deve ser satisfeita antes da execução de um método. Ao contrário da anterior, uma pós-condição estabelece o que um método deve retornar após sua execução. As pré e pós-condições são utilizadas para verificar métodos individuais. Já as invariantes são empregadas para verificar propriedades globais de uma instância, ou seja, propriedades que devam ser satisfeitas em qualquer ponto da execução do sistema.

Com a aplicação o usuário anota a parte do sistema que deseja testar. O código anotado é compilado pelo próprio VCC, como um compilador C faria. Após a compilação, o código é transformado para uma linguagem intermediária denominada Boogie (K.RUSTANM.LEINO, 2005), onde são adicionadas formulas lógicas. O programa resultante é passado ao provador de teoremas Z3 (DE MOURA, 2008).

VCC mostra-se uma alternativa interessante para verificação de sistemas concorrente desenvolvidos em C. Apesar de exigir o aprendizado de uma nova linguagem de anotação, além de requerer certo esforço para inserir as tais anotações no código, a abordagem Design by Contract é extremamente útil quando se fala em verificação de sistemas, pois o uso desta técnica permite evitar que os problemas inerentes a concorrência ocorram.

Ao passo que VCC objetiva a prevenção contra defeitos em software concorrente, o método proposto no presente trabalho visa detectá-los. Deste modo, pode-se dizer que as duas ferramentas executam atividades complementares.

1.3.3 Java *PathFinder* (JPF)

Esta é uma das aplicações mais conhecidas no campo da verificação de sistemas. JPF (VISSER, 2000) é capaz de verificar programas escritos em Java e os voltados para

plataforma Android (VAN DER MERWE, 2012). Faz uso da verificação estática para checar propriedades invariantes. Para detectar impasses, bem como condições de corrida, JPF utiliza a verificação dinâmica. Trata-se de uma ferramenta preparada para detectar uma quantidade maior de defeitos associados a softwares sequenciais e concorrentes.

O funcionamento básico de JPF se dá de forma análoga às ferramentas citadas anteriormente. Primeiro, o código fonte é traduzido para um *bytecode*, este, gerado pelo compilador específico de JPF. O *bytecode* produzido é fornecido como entrada para o verificador de modelos da ferramenta. Ao fim execução, a aplicação retorna os problemas detectados no software em teste, ou, no melhor caso, não relata defeitos.

O suporte a Android ainda é recente, dessa forma, apenas alguns recursos estão disponíveis testar softwares voltados para esta plataforma (VAN DER MERWE, 2012). Além disso, apesar de JPF fazer uso de verificação dinâmica para identificar impasses e condições de corrida, a busca por esses erros ocorre apenas na sequência de execução atual do programa em teste. Isso significa que somente uma intercalação é verificada.

O método proposto neste trabalho, por sua vez, é capaz de identificar condições de corrida e potenciais ocorrências de impasses em sequências de execução que não são as atuais, promovendo uma cobertura mais abrangente do software em teste.

1.3.4 CHESS

CHESS (MUSUVATHI, 2007) realiza verificações utilizando uma abordagem diferente da que normalmente é empregada. Com a ferramenta, o desenvolvedor possui mecanismos para controlar o escalonamento das *threads* do sistema. Com o domínio do escalonador de threads, CHESS pode controlar as diversas sequências de execução de um sistema e, dessa forma, verificar todas elas a procura de impasses, condições de corrida entre outros problemas que porventura aconteçam no programa.

Essencialmente, CHESS fornece meios para escrever diretamente sobre as *APIs* de concorrência das plataformas de desenvolvimento. Tal metodologia foi desenvolvida com o objetivo de não afetar as funções ou implementação da *API* ou do sistema em teste. Atualmente, as plataformas suportadas por CHESS são WIN32 e .NET.

A possibilidade de controlar as sequências de execução de um sistema concorrente torna CHESS uma ferramenta interessante. Assim, a equipe de testes pode determinar quais sequências de execução devem ser examinadas e verificar aquele cenário. No entanto, possuir este controle pode se tornar um problema, visto que com o domínio do fluxo de execução o testador pode direcionar a verificação para determinado ponto do sistema e deixa passar um erro em outro. Ao contrário de CHESS, a abordagem utilizada pelo método proposto neste trabalho visa reduzir ao mínimo as chances de não detecção de impasses e condições de corrida em intercalações com raríssimas ocorrências.

1.4 Estrutura do Documento

O presente documento é composto por cinco capítulos, organizados da seguinte forma:

- No Capítulo 2 são apresentados os princípios e características de sistemas de softwares concorrentes, e como eles proporcionam o surgimento de falhas como *deadlock* e *race condition*. Além disso, os conceitos de Verificação de Modelos são descritos;
- No Capítulo 3 é apresentado o método proposto neste trabalho. São apresentados os algoritmos utilizados para sua construção, bem como o fluxo de execução da ferramenta;
- No Capítulo 4, os experimentos realizados com o método são apresentados. Com o intuito de avaliar a eficácia do método foram realizados estudos de casos com três sistemas de softwares;
- No Capítulo 5, são apresentadas as Conclusões e Trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados conceitos acerca de sistemas de software concorrentes. Basicamente, a forma como funcionam tais sistemas é abordada, de forma que seja possível compreender como surgem problemas como impasses e condições de corrida. No decorrer do capítulo ainda são mostrados os princípios de Verificação de Modelos.

2.1 Sistemas de *Software* Concorrentes

Segundo o trabalho de Per Brinch Hansen (HANSEN, 1989), a programação concorrente proporciona a construção de sistemas de *software* que são compostos por um número fixo de programas sequenciais, os quais são executados concorrentemente. Tais sistemas de *software* são denominados concorrentes e os programas sequenciais que o compõem, cooperam entre si para produzir um resultado ao final da execução do sistema. Durante a execução de um sistema de *software* concorrente, o processador intercala a execução dos programas sequenciais até que se alcance o resultado final do sistema de *software*.

A programação concorrente permite que seja reduzida a ociosidade do processador, uma vez que, enquanto o sistema de *software* executa, sempre haverá um programa sequencial para ser executado. Por essa razão, os sistemas de *software* concorrentes têm sido utilizados para desenvolver a eficiência de computadores, na medida em que possibilita lidar com ambientes nos quais muitas atividades são requeridas ao mesmo tempo (HANSEN, 1979).

Para utilizar concorrência, boa parte das linguagens de programação oferece suporte a **programação *multithreaded***. A programação *multithreaded* permite a criação de um conjunto de *threads* as quais executam simultaneamente (YANG, 2009). Cada *thread* é responsável por executar uma operação específica e comunica-se com outras threads por meio de compartilhamento de recursos, tais como espaço de memória, dispositivos físicos, objetos, entre outros.

As operações são executadas de acordo com o determinado pelo **escalonador de *threads***. O escalonador intercala o tempo de execução de cada *thread*. Assim, cada *thread* recebe um período de tempo para executar sua atividade, ao fim desse período o escalonador escolhe outra *thread* para realizar seu trabalho.

O funcionamento de sistemas de *software* concorrentes é mais bem compreendido quando se entende o ciclo de vida de uma *thread*. A Figura 1 ilustra esse ciclo:

- **Estado novo e executável**: uma nova *thread* inicia no estado **novo**. Ela fica nesse estado até que o programa inicie a *thread*, que o coloca no estado **executável**;
- **Estado de espera**: é o estado no qual uma *thread* espera enquanto outra *thread* executa uma tarefa. A *thread* em **espera** passa para o estado **executável** quando outra *thread* a notifica para continuar executando;
- **Estado de espera sincronizada**: este é um estado similar ao de **espera**, a diferença é que uma *thread* que entra no estado **espera sincronizada**, aguarda por certo período de tempo especificado. Uma *thread* neste estado torna-se **executável** quando esse período expira ou quando o evento pelo qual ela está esperando ocorre. Quando em **espera sincronizada**, uma *thread* não pode utilizar o processador até que seja notificada por outra *thread* ou quando o intervalo sincronizado termina;
- **Estado bloqueado**: uma *thread* entra no estado **bloqueado** quando tenta executar alguma atividade que não pode ser completada de imediato, nesse caso a *thread* deve esperar até que essa atividade seja concluída;
- **Estado terminado**: uma *thread* entra no estado **terminado** quando completa sua tarefa. Uma *thread* também entra neste estado quando termina sua execução em virtude de um erro.

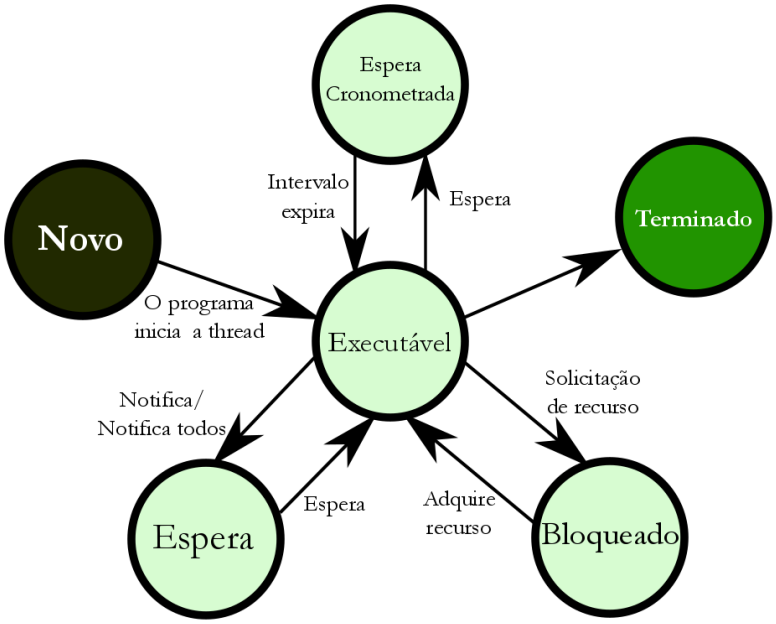

Figura 1 - Ciclo de vida de uma thread

Como citado anteriormente, as transições de um estado para outro são controladas pelo escalonador de *threads*. Durante a execução de um sistema de software ele determina qual *thread* deve executar. Essa *thread* possui um intervalo de tempo para que execute sua tarefa. Ao fim do intervalo de tempo, outra *thread* deve receber a chance de realizar sua tarefa. Essa intercalação ocorre até que a execução do programa chegue ao seu fim.

2.1.1 Não Determinismo

A capacidade de proporcionar a redução da ociosidade de processadores dos sistemas de *software multithread* é o que propicia o ganho de eficiência das aplicações construídas com essa abordagem. Em razão da intercalação de *threads*, é possível manter o processador trabalhando continuamente, desenvolvendo, desta maneira, a eficiência do computador. Diante desse cenário, é exigido que as *threads* cooperem entre si para que o sistema de *software* funcione corretamente. A cooperação entre *threads* é uma das questões mais importantes no campo da concorrência.

Para que cooperem entre si as *threads* comunicam-se umas com as outras. O escalonador escolhe uma *thread* para executar sua tarefa, esta por sua vez, ao concluir sua operação, escreve o resultado na memória. Outra *thread* é escolhida para processamento, para isso ela pode utilizar o valor armazenado anteriormente na memória. Em um ambiente em que

a memória é compartilhada, tal comportamento pode ocasionar resultados inconsistentes, uma vez que as *threads* são livres para realizar qualquer tipo de operação com o conteúdo da memória. Resultados inconsistentes são produto do **não determinismo**, característica essa inerente a sistemas de software concorrentes. Um sistema de software **não determinístico** é aquele que pode produzir vários resultados diferentes para uma mesma entrada.

O exemplo apresentado na (b) Figura 2 é utilizado para ilustrar como comportamento não determinístico afeta o funcionamento dos sistemas de software concorrentes. Note que existem duas sequências de execução possíveis ilustradas pela Figura 2. Observe que os valores iniciais de x e y são os mesmos para as sequências da Figura 2(a) e da Figura 2(b). Entretanto, os resultados retornados são diferentes para as duas execuções.

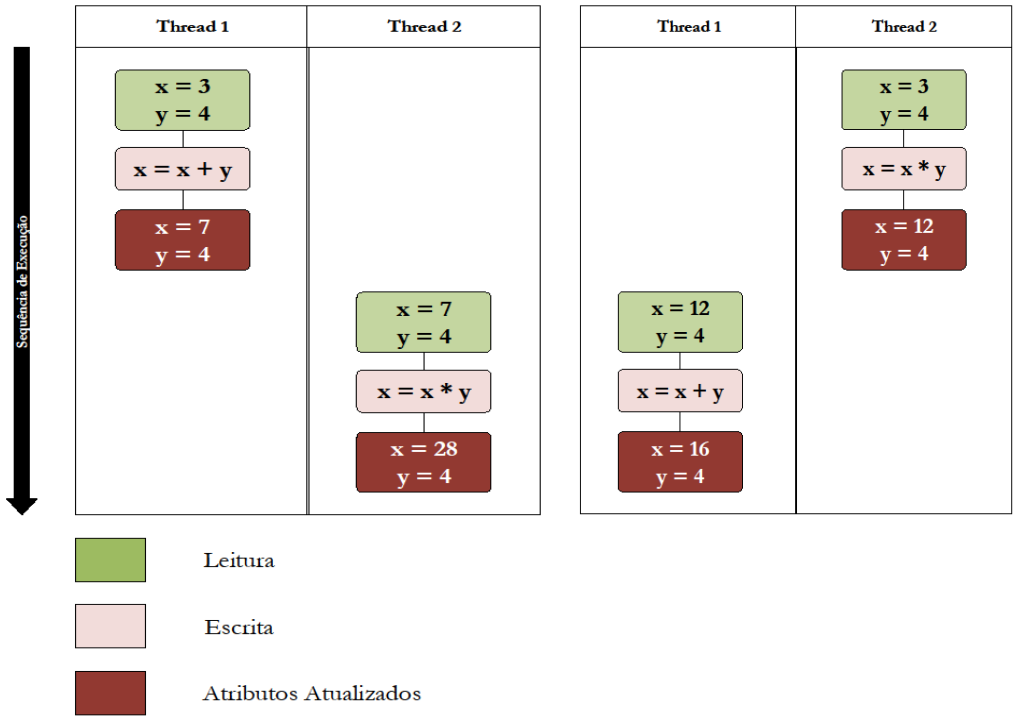

Figura 2 – (a): sequência de execução 1; (b): sequência de execução 2.

2.1.2 Condições de Corrida e Sincronização de *Threads*

O comportamento não determinístico dos sistemas de *software* concorrentes pode se tornar bastante prejudicial. Além de retornar resultados diferentes para uma mesma entrada,

estes programas estão suscetíveis à **interferência** de *threads*. A interferência é popularmente denominada **condição de corrida**. Acontece quando uma *thread* acessa um objeto compartilhado e antes que ela possa manipulá-lo, uma ou mais *threads* alteram o valor de tal objeto fazendo com que a primeira *thread* utilize um valor obsoleto. O problema de condição de corrida é ilustrado na Figura 3.

Note que a *thread* 1 obtém acesso à variável x = 3, no entanto, enquanto executa sua tarefa o valor de x é alterado pela *thread* 2 para 28. A *thread* 1 não percebe tal alteração e realiza seu trabalho utilizando valores obsoletos. Por essa razão, se faz necessário que o acesso à memória seja controlado. No trabalho de Dijkstra (DIJKSTRA, 2002) é proposto que os acessos a objetos compartilhados ocorram de maneira exclusiva. Trata-se da **exclusão mútua**, um dos principais fundamentos de sistemas de software concorrentes. A exclusão mútua impossibilita que *threads* efetuem acesso simultâneo à memória compartilhada, assim, cada *thread* recebe um período de tempo para manipular a memória. Durante esse período, outras *threads* que desejam obter acesso a essa parte da memória devem ser mantidas em espera.

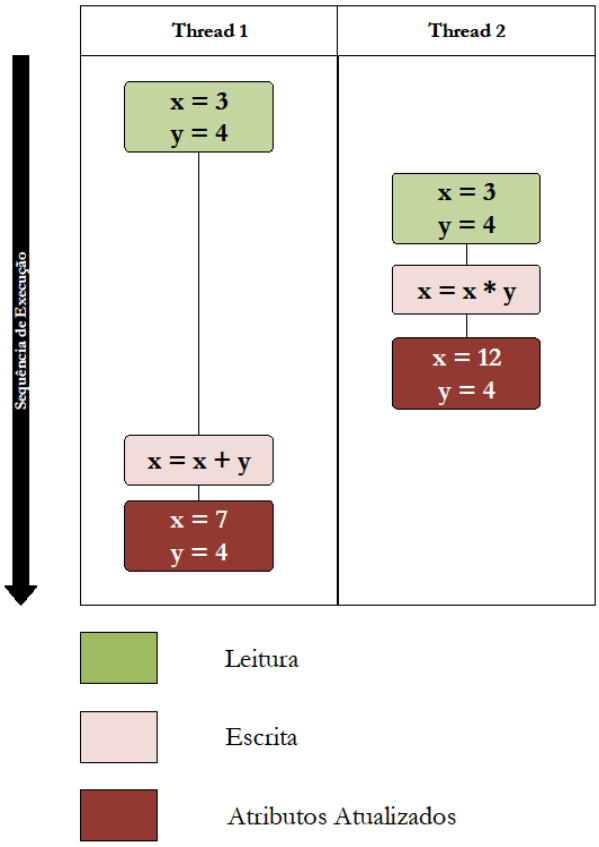

Figura 3 - Condição de corrida

Na Figura 4 ilustrado o conceito de exclusão mútua. A *thread* 1 obtém acesso aos atributos x e y, e os manipula de maneira exclusiva. Apenas ao concluir sua tarefa, a *thread* 1 libera os objetos para que a *thread* 2 os utilize. Dessa forma, os acessos aos dados compartilhados são ordenados, evitando condições de corrida e, consequentemente, resultados inconsistentes.

Entretanto, em muitos casos o mecanismo de sincronizar *threads* é utilizado incorretamente, provocando a construção de sistemas de softwares problemáticos. Por essa razão, recursos para detectar problemas como condições de corrida são necessários.

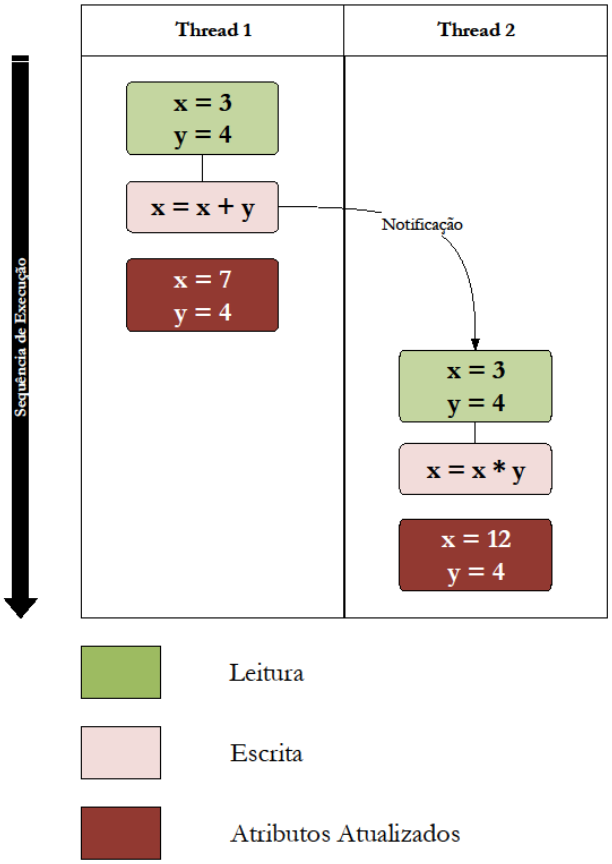

Figura 4 - Sincronização

2.1.3 Impasse

O acesso exclusivo introduz um problema bastante comum a sistemas de software concorrentes, o **impasse**. Um conjunto de *threads* está em impasse quando, uma ou mais *threads* desse conjunto está aguardando a liberação de um objeto que está sendo utilizado por outra *thread*, esta, por sua vez, está a espera da liberação de um recurso alocado pela primeira

thread. Quando isso acontece, as *threads* em questão permanecem à espera do recurso por tempo indeterminado, tal espera é denominada **espera circular**.

A ocorrência de um impasse é ilustrada na Figura 5. Observe que a *thread* 1 obtém acesso ao **recurso** 2, mas para prosseguir necessita do **recurso** 1. No entanto, o **recurso** 1 já foi alocado pela *thread* 2, que por sua vez, precisa obter acesso ao **recurso** 2, garantido pela *thread* 1. Está criado um impasse, que permanece até que ocorra alguma interferência externa.

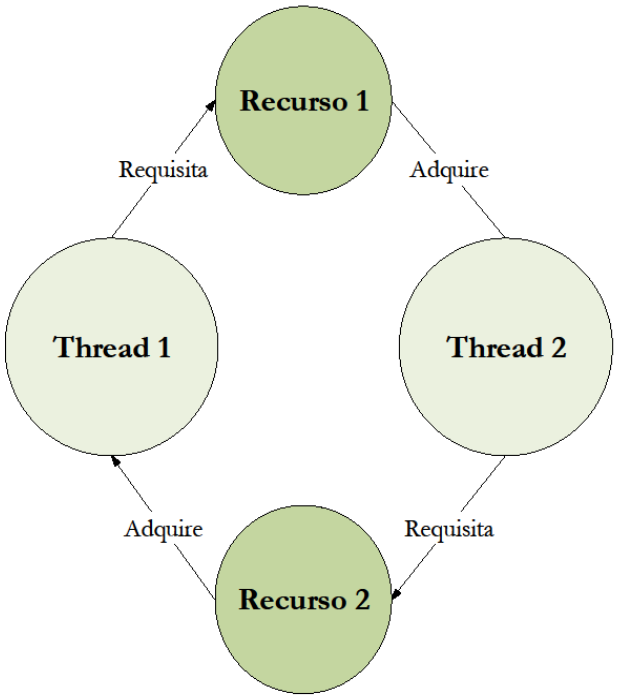

Figura 5 - Impasse

2.2 Detecção Dinâmica

A detecção dinâmica é uma abordagem que consiste em observar a execução concreta de determinado sistema de software e identificar possíveis falhas. A possibilidade de monitorar a execução de sistemas de software dá à detecção dinâmica o caráter de abordagem confiável, uma vez que se baseia em informações reais sobre a execução de sistemas de software. Neste trabalho, as informações acerca de sistemas de software concorrentes são coletadas de maneira dinâmica de forma a conferir maior confiança aos resultados do processo de identificação de falhas. Isso é feito através do uso das ferramentas ASM (BRUNETON, 2011)[5] e ASMDEX (NEVON, 2011). Tratam-se de ferramentas construídas

[5] http://asm.ow2.org

para manipular e analisar o *bytecode* gerado pelas máquinas virtuais de Java (JVM) e Android (Dalvik), respectivamente.

As ferramentas escolhidas oferecem uma API menos complexa que outros produtos similares, possibilitando assim, uma compreensão mais rápida do que deveria ser realizado para se atingir os objetivos deste trabalho. Sua documentação clara e concisa também foi fator preponderante para a escolha dos frameworks. As duas ferramentas são usadas para monitorar e extrair as informações listadas na Tabela 1. Tais informações são importantes para a constatação do modo como se comunicam as *threads* de um sistema de software, como estas *threads* afetam a execução umas das outras.

Tabela 1 - Informações extraídas por ASM e ASMDEX.

Informação	Propósito
Nome dos objetos	Utilizado para relacionar método/atributo, necessário para descobrir quais métodos acessam quais atributos.
Objetos sincronizados	Informação necessária para o processo de detecção de impasses.
Acesso de leitura/escrita	Indica se os atributos foram alterados ou apenas lidos por determinado método.

2.3 Verificação de Modelos

Segundo o trabalho de Christel Baier e Joost-Pieter Katoen (BAIER e KATOEN, 2008), verificação de modelos é *"um mecanismo automatizado que, dado um modelo de estados finito de um sistema e uma propriedade formal, sistematicamente verifica se esta propriedade é garantida para qualquer dado estado naquele modelo"*. Isto é, a técnica fornece a capacidade de examinar todos os possíveis cenários de execução de um sistema e verificar se uma propriedade é verdadeira para esse sistema.

A verificação de modelos requer que algumas fases sejam seguidas, elas são listadas abaixo:

- **Fase de modelagem**: este estágio consiste em modelar o sistema em teste utilizando uma determinada linguagem de modelagem. É nesta etapa que as propriedades as quais se deseja verificar são especificadas. As propriedades são escritas em Lógica Temporal.

- **Fase de execução**: nesta fase ocorre a execução do verificador de modelos com o objetivo de verificar a validade da propriedade no modelo do sistema.
- **Fase de análise**: esta etapa consiste em analisar os resultados gerados pela fase de **execução**. Após a análise dos resultados existem três resultados possíveis: a **propriedade é satisfeita**, nesse caso verifica-se a próxima propriedade; a **propriedade é violada**, quando isso acontece um contraexemplo é gerado; a **memória foi excedida**, aqui, uma redução do modelo se faz necessária.

Um pré-requisito da verificação de modelos é a construção do modelo que descreva o comportamento do sistema em teste. Para elaboração desse modelo são utilizados os **Sistemas de Transições** (BAIER e KATOEN, 2008). Basicamente, os sistemas de transições são grafos direcionados os quais um nó representa um **estado**, e arestas representam **transições**, estas por sua vez, significam mudanças de estado.

Na Figura 6 é ilustrado um sistema de transições que representa o comportamento de um semáforo. Os estados descrevem alguma informação do sistema em um dado momento de sua execução. As transições entre os estados especificam como o sistema pode evoluir com o tempo. Na Figura em questão, observam-se os estados verde, amarelo e vermelho, e as transições entre eles. Os estados representam as cores dos faróis de um semáforo enquanto as transições caracterizam o fluxo de funcionamento do mesmo.

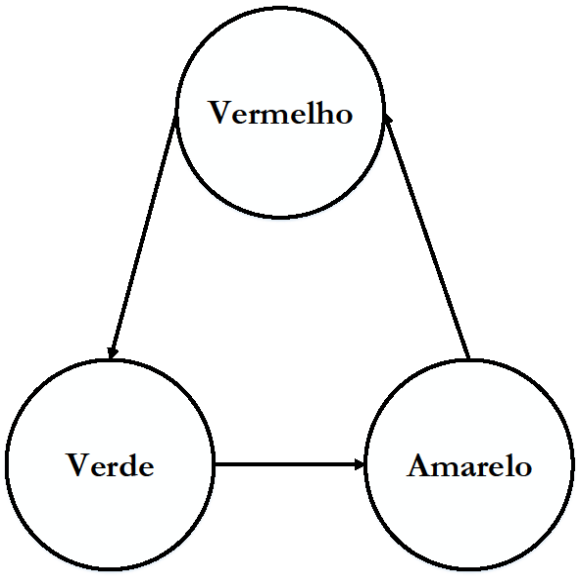

Figura 6 - Sistema de transições

2.3.1.1 Sistema de transição

Um sistema de transição ***ST*** é um uma tupla (*S*, *Act*, →, *I*, *AP*, *L*):

- *S*: é um conjunto de estados;
- *Act*: é um conjunto de ações;
- $\rightarrow \subseteq S\ X\ Act\ X\ S$: é um conjunto de transições;
- $I \rightarrow S$: é o conjunto de estados iniciais;
- *AP*: é um conjunto de proposições atômicas;
- $L : S \rightarrow 2^{AP}$: função que rotula cada estado s $\in S$.

Um sistema de transição inicia a partir de um estado inicial $s_0 \in I$ e evolui de acordo com a relação de transição →. O sistema de transição chega ao fim quando alcança um estado onde não há transições de saída.

2.3.2 Lógica Temporal

A lógica temporal (BAIER e KATOEN, 2008) consiste em um formalismo que permite descrever e analisar o comportamento de um sistema de transição, isso ocorre confrontando cada passo da evolução do sistema com uma ou mais proposições atômicas. A ideia é que as possíveis sequências de execução de um sistema de transição satisfaçam estas

proposições. Para verificação de modelos, as lógicas LTL e CTL são as mais utilizadas. A seguir, uma breve introdução é apresentada acerca das técnicas.

2.3.2.1 Lógica Temporal LTL

Na Lógica Temporal Linear, do inglês *Linear Temporal Logic (LTL)*, são utilizados operadores lógicos e temporais para elaboração de fórmulas que descrevem propriedades para cada caminho de execução num sistema de transições. Os operadores temporais mais comuns são listados a seguir. Na Figura 7 as fórmulas são ilustradas graficamente.

- *F* **p**: indica que, em um sistema de transições, certa propriedade **p** deve ser garantida ao menos uma vez no futuro;
- *G* **p**: indica que, em um sistema de transições, certa propriedade **p** deve ser garantida em todos os momentos de execução;
- **p** *U* **q**: indica que, em um sistema de transições, certa propriedade **p** é garantida até que seja atingido um estado em que uma propriedade **q** é garantida;
- *X* **p**: indica que, em um sistema de transições, certa propriedade **p** é verdadeira no próximo estado.

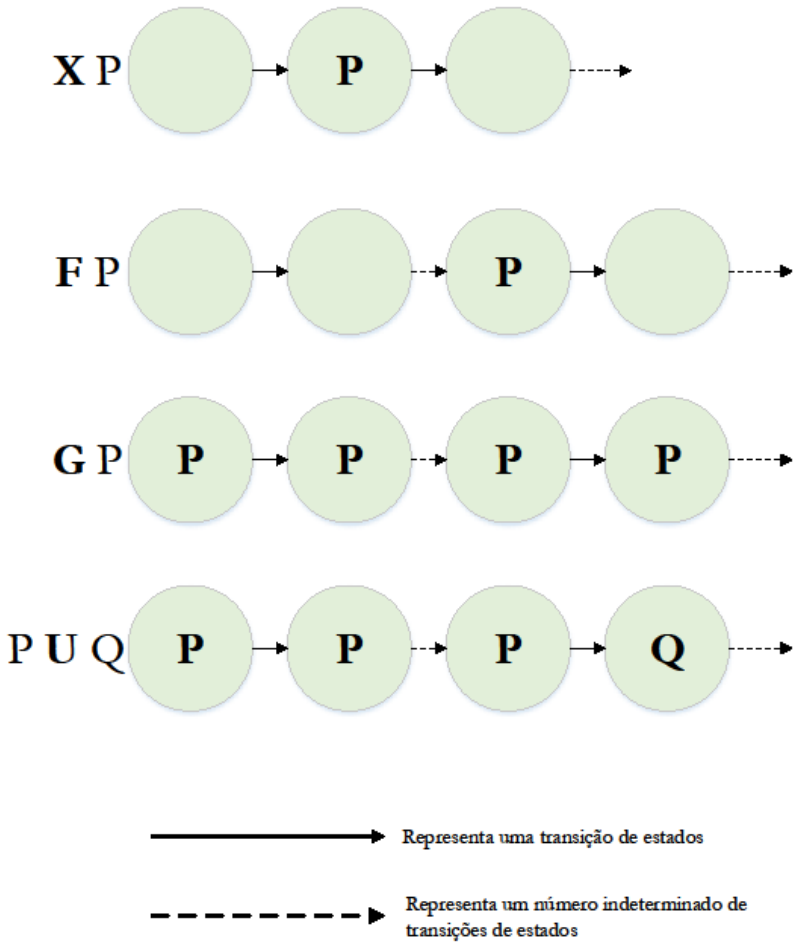

Figura 7 - Fórmulas LTL

As fórmulas são aplicáveis para cada caminho de execução individual de um dado sistema de transições. Deste modo, as fórmulas LTL devem ser utilizadas para representar o comportamento de uma sequência de execução à medida que ela progride.

2.3.2.2 Lógica Temporal CTL

A Computação da Árvore Lógica, do inglês *Computing Tree Logic (CTL)* permite a elaboração de fórmulas que especificam propriedades para os estados de um sistema de transições. Assim, as fórmulas construídas com CTL são utilizadas para verificar se o sistema de transições atinge uma determinada situação, descrita pela própria fórmula. A seguir, as principais fórmulas CTL são apresentadas:

- *AF* **p**: indica que, para todos os caminhos (*A*) a partir de um estado, eventualmente no futuro (*F*) uma propriedade **p** é garantida;
- *EF* **p**: indica que, existe pelo menos um caminho (*E*) a partir de um estado, o qual eventualmente no futuro uma propriedade **p** é satisfeita;
- *AG* **p**: indica que, para todos os caminhos a partir de um estado, uma propriedade **p** é sempre garantida;
- *EG* **p**: indica que, existe pelo menos um caminho a partir de um estado, o qual uma propriedade **p** é continuamente verdadeira;
- *A* **p** *U* **q**: requer que para todos os caminhos de um sistema de transições a propriedade **p** seja satisfeita até que **q** seja verdadeira;
- *E* **p** *U* **q**: requer que, num sistema de transições, exista ao menos um caminho o qual a propriedade **p** seja satisfeita até que **q** seja verdadeira;
- *AX* **p**: requer que uma condição **p** seja verdadeira em todos os próximos estados atingíveis a partir do estado atual;
- *EX* **p**: requer que uma condição **p** seja verdadeira em pelo menos um estado atingível a partir do estado atual;

As fórmulas são ilustradas de maneira gráfica na Figura 8:

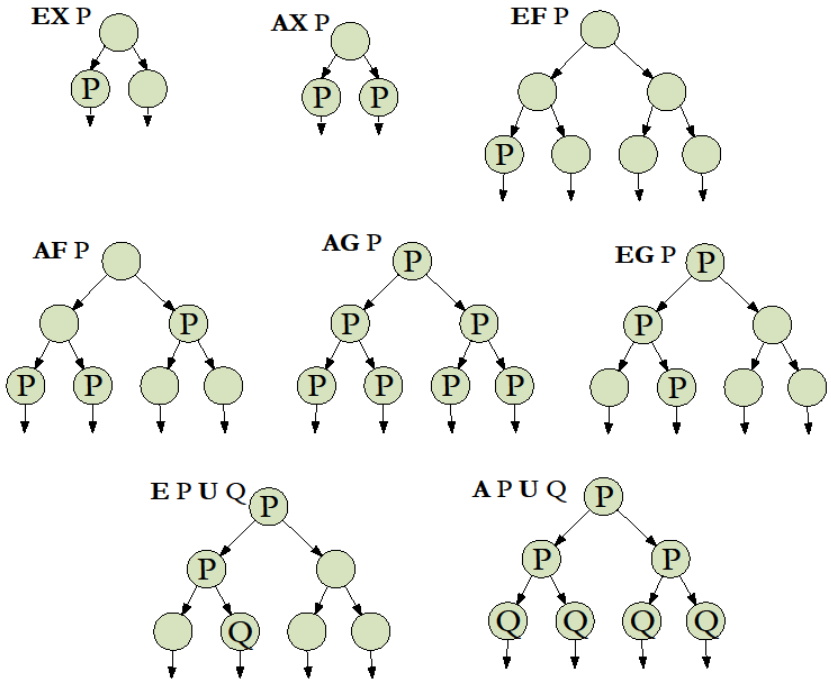

Figura 8 - Fórmulas CTL

As fórmulas CTL são aplicadas para expressar propriedades que devem ser verdadeiras em um determinado conjunto de estados do sistema de transições e leva em conta todos os possíveis comportamentos a partir desses estados.

2.3.3 Redução da Ordem Parcial

Como observado anteriormente, a verificação de modelos é uma técnica efetiva na tarefa de revelar problemas em sistemas. Contudo, a abordagem sofre de um problema extremamente prejudicial, a **explosão de espaço de estados**. Trata-se de uma situação na qual o número de estados do modelo cresce de maneira que a quantidade de memória disponível no computador é excedida (BAIER e KATOEN, 2008).

Uma das soluções para o problema é a técnica de **Redução da Ordem Parcial**, do inglês *Partial Order Reduction (POR)*. O objetivo de **POR** é reduzir o número das possíveis intercalações que necessitam serem analisadas, para verificação de formulas LTL e CTL (BAIER e KATOEN, 2008). Essencialmente, a ideia é reduzir o espaço de estados a ser analisado, substituindo um sistema de transições por apenas um fragmento dele.

A abordagem baseia-se na ideia de independência entre transições de um sistema concorrente, isto é, transições cujas execuções não interferem umas nas outras. A Definição 1, elaborada em (FLANAGAN, 2005), formaliza a noção de independência.

DEFINIÇÃO 1: *Tomando* $\boldsymbol{T}$ *como o conjunto de transições de um sistema concorrente e* $\boldsymbol{D} \subseteq \boldsymbol{T}\ X\ \boldsymbol{T}$ *como sendo uma relação binária, reflexiva e simétrica. A relação* $\boldsymbol{D}$ *é uma relação de dependência válida para o sistema se, e somente se, toda transição* $t_1, t_2 \in \boldsymbol{T}$, $(t_1, t_2) \notin \boldsymbol{D}$ (t_1 *e* t_2 *são independentes) implica que as seguintes propriedades são garantidas para todos os estados s em um espaço de estados do sistema:*

- *Se* t_1 *está habilitado em s e* $s \rightarrow^{t_1} s'$, *então* t_2 *está habilitado em s se, e somente se,* t_2 *estiver habilitado em s'; e*
- *Se* t_1 *e* t_2 *estão habilitados em s, então existe um único estado s' o qual* $s \Rightarrow^{t_1,t_2} s'$ *e* $s \Rightarrow^{t_2,t_1} s'$.

Dessa forma, as transições independentes não têm o poder de habilitar ou desabilitar umas as outras. Ainda, elas são comutativas, o que significa que a ordem na qual executam não altera o resultado final de suas execuções. Na prática, o que POR faz é realizar uma pesquisa através do espaço de estados, e a cada estado atingido durante a busca, computa um subconjunto de transições independentes, reduzindo assim o tamanho do espaço de estados a ser analisado. Tal redução é ilustrada na Figura 9. Na Figura 9(a), está representado o espaço de estados completo, no qual existem várias transições levando a um mesmo estado. Na Figura 9(b), é representado o espaço de estados reduzido.

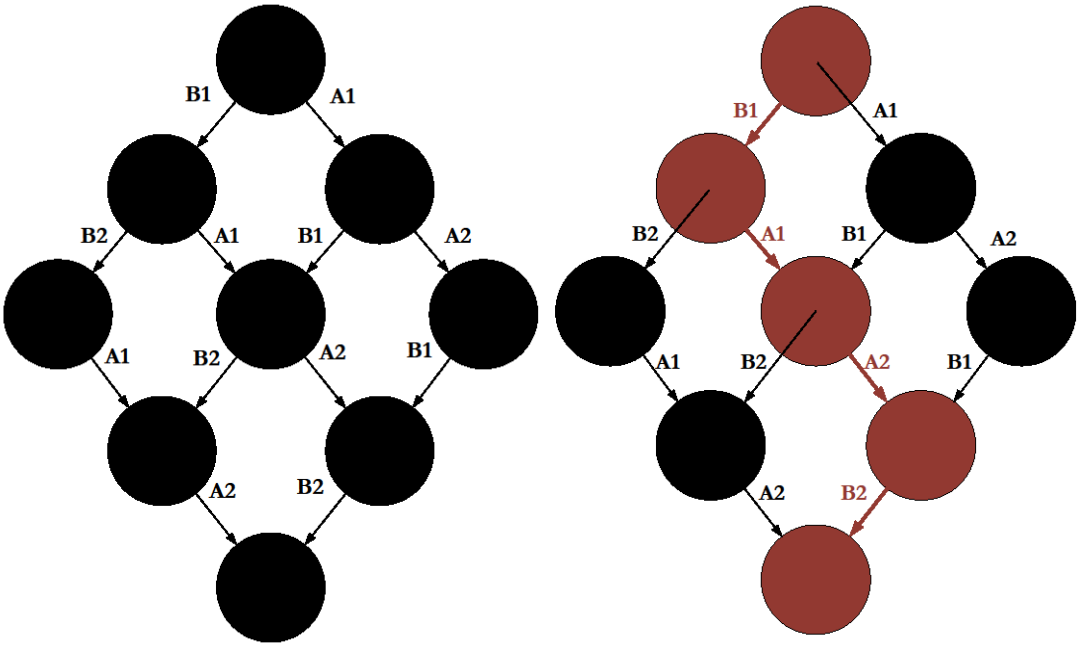

Figura 9 – (a): Espaço de estados completo; (b): espaço de estados reduzido.

2.3.4 New Symbolic Model Verifier (NuSMV)

O verificador de modelos adotado para este trabalho foi o NuSMV (CAVADA, CIMATTI, *et al.*, 2005), (CAVADA, CIMATTI, *et al.*, 2005).[6] A utilização de NuSMV se deu, principalmente, em razão da possibilidade de se utilizar um arquivo simples contendo a descrição do modelo, escrito em smv, como entrada. Além disso, sua documentação clara e concisa contribuiu de forma efetiva para o desenvolvimento deste trabalho.

NuSMV é uma reimplementação e extensão do *Symbolic Model Verifier (SMV)*. Ele fornece mecanismos para que sistemas de estados finitos sejam representados como modelos SMV. O principal objetivo da ferramenta é, dado o modelo de um sistema qualquer, verificar se o mesmo satisfaz um conjunto de propriedades especificadas pelo usuário. As propriedades podem ser expressas em CTL ou LTL. Ao confrontar o modelo com as propriedades descritas pelo usuário, NuSMV investiga se as mesmas são satisfeitas pelo modelo. Nos casos em que as propriedades forem violadas, um contraexemplo é retornado, ele indica a parte da máquina de estados que não satisfez a propriedade.

[6] http://nusmv.fbk.eu

A linguagem de descrição de modelos em NuSMV apresenta uma vasta quantidade de declarações. Para os fins deste trabalho apenas cinco foram utilizadas:

MODULE: um módulo é utilizado para encapsular declarações, restrições e especificações. Isto é, nele está contida a descrição de um modelo, ou mesmo parte de um modelo, uma vez que um módulo pode instanciar outros;

VAR: é utilizada para criação das variáveis que devem ser utilizadas no modelo;

INIT: consiste no conjunto de estados iniciais do modelo. Nesse conjunto ficam definidos os estados a partir dos quais se inicia um sistema de transições;

TRANS: é o conjunto das relações de transição do modelo. Essa declaração define os pares **estado atual** e **próximo estado** do sistema de transições;

LTLSPEC e **CTLSPEC**: são utilizadas para a especificação de propriedades LTL e CTL, respectivamente.

No Código 1 é apresentado o modelo que descreve o comportamento do semáforo ilustrado na Figura 6. Em sua última linha observa-se uma propriedade especificada de acordo com LTL. Nela está indicado que o farol amarelo deve, invariavelmente, acender após o farol verde.

```
1.  MODULE main
2.    VAR
3.       estado: {vermelho, verde, amarelo} ;
4.    INIT
5.       (estado = verde)
6.    TRANS
7.       (estado=verde & next(estado )=vermelho ) |
8.       (estado=vermelho & next(estado )=amarelo ) |
9.       (estado=amarelo & next(estado )=verde )
10. LTLSPEC G ( (estado=verde )→X(estado=amarelo))
```

Código 1 - Exemplo código NuSMV

Na Figura 10 é ilustrado o resultado retornado por NuSMV. Observe que a propriedade especificada é falsa e que a ferramenta informa contraexemplos que invalidam tal propriedade.

```
-- specification G (estado = verde -> X estado = amarelo) is false
-- as demonstrated by following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
  estado = verde
-> State: 1.2 <-
  estado = vermelho
-> State: 1.3 <-
  estado = amarelo
-> State: 1.4 <-
  estado = verde
```

Figura 10 - Resultado retornado por NuSMV

3 VERIFICAÇÃO FORMAL E DINÂMICA

Neste capítulo são apresentados os componentes do método proposto, bem como os passos que devem ser percorridos para o seu funcionamento.

3.1 Visão Geral

Esta seção apresenta uma visão geral sobre o método de verificação proposto neste trabalho. Na Figura 11 são ilustrados os passos para realização da análise e verificação.

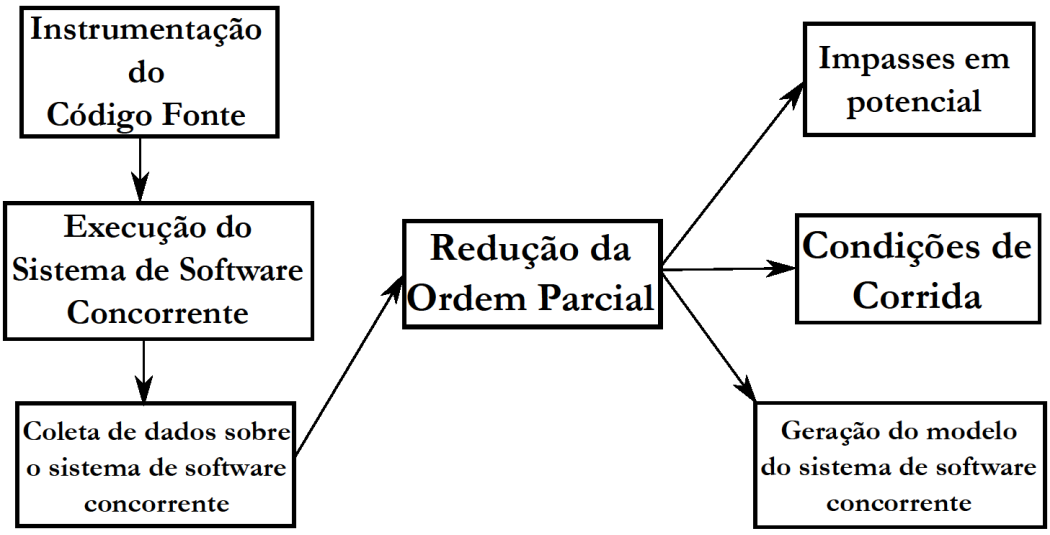

Figura 11 - Procedimentos para Verificação Formal e Dinâmica

Primeiramente, é exigida a instrumentação do código fonte de determinado sistema de software, conforme é ilustrado no Código 2. Em seguida o sistema de software deve ser executado. Conforme o sistema de software executa, as informações necessárias para verificação são coletadas. As informações consistem em:

- Objetos acessados;
- Quais *threads* acessaram que objetos;
- Tipo de acesso aos objetos: escrita ou leitura;
- Acesso sincronizado, ou não; e,
- Fluxo de execução.

```
1. public Class Test {
2.       private static final Traces traces = new Traces("Test");
3.       public void getAccess() {
4.             trace.getOperation();
5.       }
6. }
```

Código 2 - Exemplo de instrumentação do código fonte

Utilizando-se as ferramentas ASM e ASMDEX é possível analisar o *bytecode* e descobrir se objetos estão sincronizados ou não através do monitoramento das instruções *bytecode* MONITORENTER e MONITOREXIT. Estas instruções implementam o bloqueio e desbloqueio de operações, elas são utilizadas para coordenar o acesso a objetos compartilhados entre múltiplas *threads*. Assim, ao analisá-las é possível determinar se objetos estão ou não sincronizados. Contudo, MONITORENTER e MONITOREXIT não são utilizadas para implementar métodos sincronizados. Isto significa que as instruções não podem ser utilizadas para verificar se métodos estão ou não sincronizados. Para tanto, é necessário monitorar a *flag* ACC_SYNCHRONIZED, invocada quando um método sincronizado é acessado por uma *thread*.

As informações sobre o tipo de acesso (leitura/escrita) aos atributos são extraídas monitorando os *bytecodes* GETFIELD, GETSTATIC, PUTFIELD e PUTSTATIC, também conseguidas através das ferramentas ASM e ASMDEX. GETFIELD é uma instrução que coleta o valor de um determinado objeto. A captura desse valor pode ser interpretada como sua leitura. Dessa forma, ele pode ser monitorado a fim de verificar se determinado objeto foi lido. O GETSTATIC coleta valores de atributos estáticos. Analogamente ao GETFIELD, a instrução GETSTATIC é monitorada para descobrir se determinado atributo foi lido. A instrução PUTFIELD é responsável por inserir valores aos atributos, podendo, portanto, ser utilizada para determinar se algum valor foi escrito em um atributo. Já o PUTSTATIC escreve valores em atributos estáticos, sendo, portanto, utilizado para aferir se atributos estáticos foram modificados.

Além das informações sobre objetos, são necessários dados que indiquem quais *threads* realizaram acesso aos mesmos. Ainda, deve ser verificada a ordem em que tais acessos ocorrem. Neste trabalho, essas informações foram capturadas utilizando a API de *threads* fornecida pelo Java.

As informações coletadas são organizadas e compõem uma sequência de execução, onde ficam conhecidas as formas de comunicação entre as *threads*, bem como a ordem na qual elas executam suas tarefas. Tal sequência de execução passa pelo algoritmo de Redução da Ordem Parcial Dinâmico, do inglês, **Dynamic Partial Reduction (DPOR)** (FLANAGAN, 2005). DPOR analisa a sequência de execução resultante e retorna uma série de sequências de execução alternativas, as quais são consideradas sujeitas a ocorrências de falhas.

Esse conjunto de sequências de execução resultante pode ser analisado quanto a potencial existência de impasses, analisado quanto à ocorrência de condições de corrida e utilizado para geração de um modelo SMV do sistema de software em teste. Nesse último caso, o modelo construído é verificado utilizando a ferramenta de verificação de modelos NuSMV.

3.2 Redução do Espaço de Estados

As informações contidas no arquivo gerado durante a execução do sistema, são utilizadas por um algoritmo de redução da ordem parcial. O algoritmo utilizado neste trabalho é o Dynamic Partial Order Reduction (DPOR) desenvolvido em (FLANAGAN, 2005) e otimizado em (SAARIKIVI, 2012). A escolha do algoritmo se deu em razão da natureza dinâmica do método proposto. A maioria das técnicas que realizam a redução, o fazem utilizando dados que apenas podem ser obtidos estaticamente. DPOR, por sua vez, é um algoritmo voltado para verificação dinâmica de modelos, isto é, as informações necessárias para seu funcionamento são coletadas em tempo de execução.

Ao analisar os dados contidos no arquivo, DPOR captura o não determinismo adicionando pontos de retrocesso ao longo da sequência de execução. Esses pontos identificam transições alternativas que precisam ser exploradas posteriormente. Tais transições são identificadas de acordo com o conceito de classes de equivalência, tema abordado mais adiante. O processo se repete até que todas as transições alternativas sejam exploradas e que não mais existam pontos de retrocesso. Além de reduzir o espaço de estados, DPOR ainda detecta todos os impasses do sistema.

O algoritmo faz uma busca em profundidade na sequência de transições a partir do estado inicial. Uma sequência de transições é representada por $t_{1,}t_2 \dots t_n$ onde existem estados

$s_n ... s_{n+1}$ tal que s_0 é o estado inicial e $s_0 \rightarrow^{t_1} s_1 ... \rightarrow^{t_n} s_n$. Dada uma sequência de transições *S*, a seguinte notação é utilizada:

- S_i refere-se à transição t_i;
- S_t denota *S* com uma transição adicional *t*;
- *dom(S)* significa o conjunto {1,...,n};
- *pre(S,i)* para $i \in dom(S)$ refere-se ao estado s_i; e
- *last(S)* refere-se a s_{n+1}.

3.2.1 Definições

A compreensão do Algoritmo DPOR exige o entendimento dos conceitos apresentados nas seções a seguir.

3.2.1.1 Conceitos Fundamentais

Abaixo são apresentados alguns conceitos importantes para a compreensão do modo como funciona o algoritmo em questão:

- **Estado**: compreende o estado local de uma *thread*, combinado ao estado compartilhado por várias *threads*. Isto é, um estado é a junção de objetos que são acessados localmente com objetos compartilhados.
- **Operações visíveis**: operações realizadas em objetos compartilhadas.
- **Transições**: move o sistema de um estado para outro, através de uma operação visível executada por uma *thread* ou processo. O estado o qual leva uma transição é denominado *next(s,p)*, onde *s* é o estado de origem e *p* é o processo.
- **Transição habilitada**: uma transição é dita habilitada em um estado, se os estados locais e compartilhados estão definidos.

Uma transição move um sistema de transições de um estado para outro através da execução de uma operação visível realizada por processo ou *thread*.

3.2.1.2 Eliminando transições equivalentes

Em uma sequência de transições, a ordem de execução de quaisquer transições pode ser trocada sem que se altere o resultado final. Nesse caso, a sequência de transições representa uma classe de equivalência entre sequências similares, que podem ser obtidas a partir da troca de transições independentes (FLANAGAN, 2005). Tal cenário corresponde ao ordenamento parcial, ele deve ser identificado para evitar a análise desnecessária de transições que levam ao mesmo resultado. A ordem parcial é capturada por meio do relacionamento **acontece-antes**. Essa relação ocorre entre duas transições e indica que uma deve ocorrer após a outra. Em uma sequência de transições, $S=t_1 \ldots t_n$ é a menor relação em *1...n* tal que:

- $i \leq$ j e S_j é dependente de S_i então $i \rightarrow_s j$;
- $\rightarrow_s j$ é transitivo.

Para identificação dos pontos de retrocesso, uma variação da relação acontece-antes é utilizada. A variação relaciona o índice $i \in dom(S)$ de uma sequência de transições e um processo *p*. A relação garante que para $i \rightarrow_s p$ se, e somente se, existe um *k* tal que $i \rightarrow_s k$ e $p = proc(S_s)$. Isso significa que uma transição ocorre antes de qualquer operação executada por um processo.

No algoritmo DPOR, a relação acontece-antes é implementada utilizando vetores de relógios (MATTERN, 1989). Um vetor de relógios mapeia processos e índices em uma sequência de transições *S:* $CV = P \rightarrow \mathbb{N}$. Cada processo *P* possui um vetor de relógios $C(p) \in CV$. Agora $i \rightarrow_s p$ se, e somente se $i \leq C(p)(proc(S_i))$.

Além dos processos, são mantidos vetores de relógios para todos os objetos acessados *o*. Para os objetos que foram alterados, são mantidos relógios de escrita *CW(o)*. Os que apenas foram acessados, possuem relógios de leitura *CA(o)*. Os vetores de relógios são utilizados para manter a ordem a qual tais objetos são acessados ou alterados, e são atualizados da seguinte forma:

- Se ocorrer uma operação de escrita sobre um objeto *o*, então o vetor de relógios do processo que realizou o acesso *C(p)* é atualizado a partir do vetor de acesso *CA(o)*. Para operações de leitura sobre o mesmo objeto, a atualização ocorre a partir do vetor de escrita *CW(o)*.

- Caso ocorra uma operação de escrita sobre um objeto *o*, *CW(o)* é atualizado para ser igual a *C(p)*.
- *CA(o)* é sempre atualizado para ser igual a *C(p)*.

Os últimos acessos ou alterações a objetos são capturados por *LR* e *LW*, respectivamente. *LW* armazena os índices das últimas leituras desde a última escrita.

3.2.1.3 Detectando transições dependentes

Como anteriormente citado, DPOR baseia-se na busca por transições que sejam dependentes entre si. Duas transições v_1 e v_2 são ditas dependentes se, e somente se, eles acessam o mesmo objeto, isto é $\alpha(v_1) = \alpha(v_2)$. Dessa forma, para identificar a dependência entre diferentes transições, basta verificar se as mesmas obtêm acesso a pelo menos um objeto em comum.

É importante notar que, caso as operações realizadas sobre um objeto compartilhado sejam apenas de leitura, estas podem ser excluídas da relação de dependência. Isto ocorre porque leituras concorrentes não interferem no resultado final da operação.

3.2.2 O Algoritmo

Para o início da execução o algoritmo recebe as seguintes entradas:

- A sequência de transições *S*;
- O vetor de relógios de um determinado processo *CP*;
- O vetor de relógios de um objeto que teve seu conteúdo alterado *CW*;
- O vetor de relógios de um objeto que teve seu conteúdo lido *CA*;
- O índice da última escrita num objeto *LW*; e,
- E os índices das últimas leituras em um objeto, desde a última ocorrência de uma operação de escrita nesse objeto *LW*;

Basicamente, DPOR é formado por duas partes. Uma delas procura por estados habilitados em um determinado objeto. A segunda parte do algoritmo realiza a pesquisa por pontos de retrocesso a partir dos estados habilitados.

A sequência de execução do programa em teste é colocada em uma pilha. Cada estado analisado é retirado da pilha. O processo se repete até que a pilha esteja vazia. A Linha 2 do Algoritmo 1 marca o início da busca por estados habilitados. Nela o último estado da sequência de execução é armazenado em *s*. Na Linha 18, *s* é analisado de forma a verificar a existência de uma transição habilitada a partir desse estado. Caso ocorra, o estado para o qual *s* está habilitado é adicionado ao fim da pilha, na Linha 23. A partir da linha 26 a 40 ocorrem as atualizações nos vetores de relógios conforme visto anteriormente.

1. **Função Explore**(S, CP, CW, CA, LW, LR)
2. *s* ← *last(S)*
3. **para todo** processes p **faça**
4. ***v*** ← *next(s, p)*
5. **se v** é de escrita e ∃*k* = *max*({*k* | $LR(\alpha(v))_k \not\leq$ CP(p)(proc(S_1))) então
6. *i*←$LR(\alpha(v))_k$
7. **senão**
8. *i*←*LR*($\alpha(v)$)
9. **fim se**
10. **se** i ≠ 0 e i ≰ CP(p)(proc(S_i))
11. **se *v*** ∈ *habilitado(pre(S,i))*
12. adicione *v* ao *backtrack(pre(S,i))*
13. **senão**
14. adicione *habilitado(pre(S,i))* ao *backtrack(pre(S,i))*
15. **fim se**
16. **fim se**
17. **fim para**
18. **se** ∃v_0 ∈ *habilitado(s)*
19. *backtrack(s)* ← v_0
20. *done* ← ∅
21. **enquanto** ∃v_{next} ∈ (*backtrack(s)*\ *done*)
22. adiciona v_{next} ao *done*
23. *S'* ← *S*. v_{next}
24. p ← *proc*(v_{next})
25. *o* ← *α(v)*
26. **se** v é de escrita **então**
27. *cv* ← *vmax(CP(p), CA(o))[p:=S']*
28. *CP'* ← *CP[p:=cv]*
29. *CW'* ← *CW[o:=cv]*
30. *CA'* ← *CA[o:=cv]*
31. *LW'* ← *LW[o:=[S']]*
32. *LR'* ← *LR[o:=ϵ]*
33. **senão**
34. *cv* ← *vmax(CP(p), CW(o))[p:=S']*
35. *CP'* ← *CP[p:=cv]*
36. *cv* ← *vmax(CP(p), CW(o))*
37. *CA'* ← *CA[o:=cv]*
38. *LW'* ← *LW*

39. *LR' ← LR[o:=LR(o).|S'|]*
40. fim se
41. ***Explore****(S', CP', CW', CA', LW', LR')*
42. fim enquanto
43. fim se
44. **fim função**

Algoritmo 1 – Dynamic Partial Order Reduction (DPOR)

Nas Linhas 4-17 ocorrem a identificação de transições dependentes e a construção do conjunto *backtrack(Pre(S,i))*, o conjunto das sequências de execuções que devem ser verificadas posteriormente. Na Linha 4 é coletado o estado *v* que ocorre logo em seguida a *s*. A partir daí verifica-se a existência de uma transição que possua relação de dependência com *v* e que seja concorrente em relação à *v*, isto é, a transição deve ter sido executada por um processo diferente do que executou *v*. Caso exista uma transição que satisfaça tais condições, ela é incluída no conjunto *backtrack(Pre(S,i))*. Esse processo ocorre até que todos os estados da pilha tenham sido analisados.

Muitas vezes as transições são concorrentes e dependentes, mas executam apenas operações de leitura sobre os objetos compartilhados. Nessa situação, a ordem de execução não altera o resultado final do programa. Por essa razão foram adicionadas as Linhas 5-9. Nelas é verificado se *v* é uma operação de escrita ou de leitura. Em caso de escrita, são pesquisadas todas as últimas leituras $LR(\alpha(v))_k$ sobre o objeto compartilhado desde a última operação de escrita. As últimas leituras são consideradas porque, ao contrário das operações de escrita, as leituras são operações desordenadas, então não se sabe exatamente qual foi a última operação de leitura ser executada. No entanto, se *v* for operação de leitura, é pesquisada a última operação de escrita sobre o objeto $LW(\alpha(v))$. $LR(\alpha(v))_k$ ou $LW(\alpha(v))$ que sejam dependentes e concorrentes em relação à *v* são denominados pontos de retrocesso.

As transições adicionadas no ponto de retrocesso compõem o espaço de estados reduzido. Ademais, o algoritmo detecta todos os eventuais impasses ou estados terminais, a cada estado sem transições habilitadas.

3.3 Geração Automática do Modelo

O sistema de transições proveniente da execução de DPOR é traduzido em um modelo escrito na linguagem SMV. Para realizar tal atividade, neste trabalho foi adotado o algoritmo

de **Time-Preserving Merger** desenvolvido em (PAULO SALEM DA SILVA, 2009). O algoritmo é capaz de, a partir de uma coleção de informações sobre as sequências de transições de um dado sistema, unir todas essas informações em um modelo SMV. O modelo resultante é utilizado como entrada para verificação com a ferramenta NuSMV.

O modelo é gerado baseando-se no conceito de **estados estendidos**, o qual permite a captura do histórico dos estados do sistema de transições. Segundo (PAULO SALEM DA SILVA, 2009), um estado estendido é o par *(s, t)*, em que *s* é um estado e *t* é o instante onde $t \in \mathbb{N}$, e $t \geq 1$. Armazenar o instante em que cada estado ocorre é útil para inferir possíveis comportamentos futuros, que de outra forma não seriam identificados.

Basicamente, o algoritmo define os conjuntos de estados estendidos para cada instante, Linhas 6 a 8 do Algoritmo 2. Posteriormente, os conjuntos de estados estendidos são vinculados de acordo com as transições presentes no sistema de transições, Linhas 10 a 16.

1. **Input** *Um conjunto S de possíveis estados e o conjunto T de traces em S.*
2. **Output** *Um espaço de estados.*
3. *S' é um conjunto vazio*
4. *R é uma relação binária vazia*
5. *I é um conjunto vazio*
6. **para todo** $t \in T$
7. s_1 *é o estado inicial de t*
8. $S' \leftarrow S' \cup \{(s_1, 1)\}$
9. $I \leftarrow I \cup \{(s_1, 1)\}$
10. **fim para**
11. *n é o tamanho do maior trace T*
12. **para** i ← 1 até *n-1* ***faça***
13. **para todo** t $\in$ *T faça*
14. ***se*** $|t| \geq i+1$*faça*
15. s_1 *é o iésimo estado em t*
16. s_{i+1} *é o estado (i + 1) em t*
17. *x é uma tupla* (s_i, i)
18. *y é a tupla* $(s_i, i+1)$
19. $S' \leftarrow S' \cup \{y\}$
20. $R \leftarrow R \cup \{y\}$
21. **fim se**
22. **fim para**
23. **fim para**
24. **Devolve** *(S', R, I)*

Algoritmo 2 - Time-Preserving Merger

3.4 Detectando condições de corrida

Como observado no Capítulo 2, condições de corrida ocorrem quando existem acessos desordenados a um determinado objeto compartilhado, e que pelo menos um dos acessos é de escrita. Em sistemas de bancos de dados, condições de corrida podem surgir quando transações concorrentes ocorrem fora de ordem. Para detectar tais problemas, são utilizados algoritmos que analisam a ordem de execução dessas transações, ao identificar qualquer transação desordenada, uma condição de corrida é revelada (CHU, 2004). Do mesmo modo, a abordagem de monitoramento da ordem de acessos concorrentes a objetos compartilhados é utilizada para detectar condições de corrida em sistemas de *software* concorrentes. Neste trabalho, foi adotado o algoritmo de detecção de condições de corrida denominado ***FastTrack*** (FLANAGAN, 2009). O algoritmo implementa uma variação de vetores de relógios para monitorar a ordem na qual as operações em objetos compartilhados acontecem. Dessa forma, qualquer inconsistência na ordem de execução é detectada.

Algoritmos baseados em vetores de relógios são considerados muito custosos. Isso porque há a necessidade de que se mantenham relógios para todas as *threads* e objetos compartilhados do sistema, um para operações de escrita e outro para operações de leitura. A manutenção de todos esses relógios significa que, no momento em que se verifica a ordem de execução das operações (utilizando a abordagem **acontece-antes**), uma comparação entre os vetores de relógios de cada *thread* e de cada objeto em analise é realizada. A necessidade de um grande número de comparações ocasiona uma perda no desempenho desses algoritmos.

FastTrack ataca o problema do desempenho introduzindo o conceito de épocas. Uma época corresponde ao par *thread t* e um relógio *c*, *c@t*. Essencialmente, o que acontece é uma substituição do vetor de relógios mantido para operações de escrita, por uma época. Agora, cada objeto compartilhado possui uma época indicando o momento *c* em que determinada *thread t* realizou uma operação de escrita nela. Assim, durante a verificação, o relógio de uma *thread* é comparado apenas com a época do objeto.

A noção de época baseia-se na ideia de que, assumindo que não haja condições de corrida em uma sequência de execução, todas as operações de escrita em um determinado objeto *x* estão ordenadas (FLANAGAN, 2009). Dessa forma, apenas informações sobre a última operação de escrita sobre *x* são suficientes para identificar condições de corrida.

3.4.1 Detectando condições de corrida entre operações de escrita

Para detectar condições de corrida entre operações de escrita num dado objeto *x*, basta comparar o relógio *CV* de uma *thread* com a época *c@t* de *x*. Caso *c@t > CV*, existe uma condição de corrida no objeto *x*.

3.4.2 Detectando Condições de Corrida entre operações de escrita e de leitura

Vetores de relógios são mantidos para operações de leitura em um objeto. Para identificar condições de corrida entre operações de escrita e leitura, é suficiente verificar se a leitura acontece depois da operação de escrita. Isso é feito comparando-se o relógio de leitura com a época na qual ocorreu a última operação de escrita sobre o objeto em análise.

3.4.3 Detectando condições de corrida entre operações de leitura e de escrita

Revelar condições de corrida entre uma operação de leitura e outra de escrita é um pouco mais complicado. As operações de leitura, ao contrário das de escrita, podem ocorrer desordenadamente. Por se tratarem de operações que apenas acessam o objeto, a ordem na qual elas ocorrem não afeta o resultado final. Por essa razão, não é suficiente analisar apenas o último acesso, e sim o vetor de relógios mantido para operações de leitura. Os casos em que ocorrem leituras desordenadas são denominados **leitura compartilhada**.

Entretanto, em alguns casos as operações de leitura são ordenadas. Isso ocorre quando um objeto é protegido por bloqueio, devido à sincronização entre *threads*. Outro cenário em que é possível que operações de leitura aconteçam ordenadamente é quando um dado é *thread* local, onde somente uma *thread* tem acesso ao objeto. Para estas situações é suficiente verificar se a última leitura acontece depois da operação de escrita.

3.4.4 O Algoritmo

O Código 3 apresenta um esboço do que seria a implementação do algoritmo. Para a execução do *FastTrack* são buscadas informações como:

- Estado da *Thread* (*ThreadState*): essa informação compreende a identificação de cada *thread* e um vetor de relógios. A época de cada *thread* pode ser expressada por *t.C[t.tid]*.

- Estado do objeto (*VarState*): contém as épocas de escrita e leitura *W* e *R*, além do vetor de relógios para leitura compartilhada, denominada *READ_SHARED*, *Rvc*.
- Monitor de bloqueio (*LockState*): mantém um vetor de relógios que informa quais foram as últimas *threads* a realizarem operações nos objetos.

Nas Linhas 13 a 31 do Código 3 pode ser observado o procedimento para detectar condições de corrida entre operações de escrita. Na Linha 16 é descrita a comparação entre a época em que ocorreu a última escrita sobre um objeto é o relógio de determinada *thread*, o qual indica o momento da última operação de escrita realizada pela referida *thread* sobre este objeto. Caso o valor correspondente ao relógio da *thread* em questão seja menor que a época na qual ocorreu a última escrita, significa que ocorreram operações desordenadas sobre o objeto em análise, ou seja, houve condição de corrida. O restante do procedimento, das Linhas 18 a 31, é dedicado à atualização da época em que ocorreu a operação de leitura sobre o objeto em análise, ou do vetor *Rvc*, que armazena o tempo em que ocorreram os últimos acessos de leitura sobre esse objeto. Neste último caso, apenas quando da ocorrência de sucessivas operações de leitura sobre o objeto.

Nas Linhas 33 a 44 está implementado o procedimento para identificar condições de corrida entre operações de escrita e entre operações de leitura e escrita. Na Linha 36 é apresentada a implementação do mesmo procedimento descrito na Linha 16. Nas Linhas 39 e 41 são mostradas as implementações dos procedimentos que realizam a análise quanto à existência de condições de corrida entre operações de leitura e escrita. Na Linha 39 é descrita a análise quanto à existência de condições de corrida entre a última operação de leitura sobre um objeto e a última operação de escrita executada sobre esse objeto por determinada *thread*. Enquanto na Linha 41 é apresentada a implementação do procedimento de análise quanto à ocorrência de condições de corrida entre as últimas operações sucessivas de leitura sobre um objeto e a última operação de escrita sobre esse objeto executada por determinada *thread*.

```
1.  class ThreadState {
2.      int tid ;
3.      int C[ ] ;
4.      int epoch ;
5.  }
6.  class VarState {
7.      int W, R;
8.      int Rvc[ ] ;
9.  }
10. class LockState {
```

```
11.     int L[ ] ;
12.  }
13.  void read (VarState x, ThreadState t ) {
14.     if (x.R == t.epoch) return;
15.     // Race entre operações de escrita e leitura?
16.     if (x.W > t.C[TID(x.W)]) error ;
17.     // Atualiza estado de leitura
18.     if (x.R == READ_SHARED) {
19.           x.Rvc[t.tid ] = t.epoch ;
20.     } else {
21.           if (x.R <= t.C[TID(x.R)] ) {
22.                 x.R = t.epoch ;
23.           } else { if(x.R <= t.C[TID(x.R)]) {
24.                 if (x.Rvc == null )
25.                       x.Rvc = newClockVector ( ) ;
26.                 x.Rvc[TID(x.R)] = x.R;
27.                 x.Rvc[t.tid] = t.epoch ;
28.                 x.R = READ_SHARED;
29.                 }
30.           }
31.  }
32.
33.  void write(VarState x, ThreadState t) {
34.     if (x.W == t.epoch ) return ;
35.     // Race entre operações de escrita? write-write race?
36.     if (x.W > t.C[TID(x.W)]) error;
37.     // Race entre operações de leitura e escrita?
38.     if (x .R != READ_SHARED) {
39.                 if (x.R > t .C[TID(x.R)] ) error ;
40.           } else {
41.                 if (x.Rvc[u] > t .C[u] for any u) error ;
42.           }
43.     x.W = t.epoch ; // Atualiza estado de escrita
44.  }
```

Código 3 - FastTrack

É importante salientar que existe outra categoria de algoritmos para detectar condições de corrida em tempo de execução, são aqueles baseados em conjunto de bloqueios. Tais algoritmos mantêm um conjunto de bloqueios candidatos para cada objeto. Na medida em que são acessados, um bloqueio é armazenado no conjunto. Posteriormente, os conjuntos são comparados e, caso a interseção entre eles seja vazia, uma condição de corrida aconteceu. O algoritmo **Eraser** é o exemplo mais famoso (SAVAGE, 1997).

A abordagem baseada em conjunto de bloqueios apresenta um desempenho melhor que a estratégia que faz uso de vetores de relógios. Contudo, aquela depende dos idiomas dos

bloqueios (*synchronized*, *fork-join*, *lock-unlock*) utilizados em um sistema. Em muitos casos essa características podem resultar em falsos alarmes (FLANAGAN, 2009). Por outro lado, apesar do desempenho inferior, os algoritmos baseados em relógios são precisos, por acompanhar a ordem exata de execução do sistema e não dependerem de idiomas de bloqueios.

3.5 Detectando Impasses em potencial

Cada sequência de execução capturada pelo DPOR é analisada com o intuito de detectar os pontos onde é possível haver impasse. A tarefa de detectar impasses em potencial consiste em identificar **ciclos** de requisições de bloqueios aos objetos do sistema (SADDEK BENSALEM, 2006). Tomando o conjunto de estados de um sistema de transições S = $x_1, x_2, \ldots, x_k$ e o conjunto de transições R= $R = (x_1, x_2), (x_2, x_3),\ldots,(x_i, x_k),$ onde x_i são todos distintos, exceto que x_k deve ser igual a x_1, nesse caso o caminho forma um ciclo.

O algoritmo 3 foi desenvolvido em (SADDEK BENSALEM, 2006) para detectar ciclos. Essencialmente, o algoritmo percorre cada sequência de transições em um sistema de transições e verifica se existe ou não algum ciclo. Caso ocorra algum, um alerta de impasse em potencial é emitido.

Contudo, alguns cuidados devem ser tomados. Primeiramente, um ciclo sobre um objeto só ocorre quando a operação é realizada por *threads* diferentes, pois um mesma *thread* realizando duas operações sobre um mesmo objeto não configura um impasse. Em segundo lugar, deve-se levar em conta o cenário onde uma *thread* é iniciada a partir de outra, nesse caso também é impossível a ocorrência de impasse. O primeiro caso é tratado através de um mapeamento entre os objetos e as *threads* que o acessaram. A segunda situação é tratada por meio de um monitoramento das operações *start()* e *join()*, elas marcam o início de um *segmento de código*. Segmentos que não podem executar em paralelo são eliminados da verificação, uma vez que não podem resultar em impasse.

Os segmentos são identificados por um número natural (a partir de 0) a cada operação *start()* ou *join()*. Por exemplo, quando uma *thread* T_1 inicia outra *thread* T_2, trata-se de dois segmentos de código que não podem ser executados concorrentemente. Para descobrir se os segmentos executam paralelamente ou não, novamente o conceito de acontece-antes é

utilizado. Dados dois segmentos s_1 e s_2, eles são ditos paralelos se $s_1 > s_2$ e $s_2 > s_1$. O algoritmo é apresentado a seguir.

1. **Input** *Uma sequência de execução S.*
2. **para todo** *S* **faça**
3. *Define os segmentos em S.*
4. **se** existe um ciclo em S **então**
5. **se** as threads que executam as operações são diferentes **então**
6. *Impasse em Potencial.*
7. **fim se**
8. **fim se**
9. **fim para**

Algoritmo 3 – Impasses em potencial

4 ESTUDO DE CASO

Neste capítulo são apresentados experimentos realizados com o fim de verificar a eficácia do método proposto. Os experimentos foram realizados em 3 sistemas de *software* concorrentes, um desenvolvido para plataforma Android e dois escritos em Java. Para fins de comparação os sistemas também foram testados utilizando Java Pathfinder (VISSER, 2000) (VAN DER MERWE, 2012), considerada a principal ferramenta de verificação de sistemas de software Java.

4.1 Alterações no Código Fonte

Para produzir comportamentos incorretos, bem como provocar a ocorrência de impasses e condições de corrida nos sistemas de *software* testados, foi utilizada a técnica de injeção de falhas (D'SILVA e WEISSENBACHER, 2008) (MEI-CHEN HSUEH, 1997). Com a injeção de falhas é possível introduzir erros em determinado sistema com o intuito de analisar seu comportamento na presença de falhas (D'SILVA e WEISSENBACHER, 2008). A técnica permite que sejam injetadas falhas em nível de *hardware*, de *software* e de modelo (D'SILVA e WEISSENBACHER, 2008).

Neste trabalho, foi utilizada a injeção de falhas em nível de *software*. A técnica consiste em realizar alterações no código fonte de determinado sistema de *software* de modo a inserir defeitos no mesmo. A seguir, a essência das modificações do código fonte realizadas no presente trabalho é apresentada.

4.1.1 Alterações no Código Fonte: Condição de Corrida

A ocorrência de condições de corrida está condicionada ao acesso desordenado de diferentes *threads* a objetos compartilhados. A estratégia utilizada para combater acessos dessa natureza é a exclusão mútua, conseguida através da implementação dos diversos mecanismos de sincronização. Deste modo, para proporcionar o surgimento de condições de corrida, tais mecanismos de sincronização foram extraídos do código fonte, como é mostrado o exemplo dos Códigos 4 e 5.

```
1. ...
2. public Class Test{
3.       ...
4.       public synchronized void getAccess(){
```

```
5.              ...
6.        }
7.        ...
8.  }
```

Código 4 – Código fonte com sincronização

No Código 4 o método *getAccess()* está sincronizado. Para realização dos testes o mecanismo *synchronized* é extraído, como mostra o Código 5. O mesmo acontece com os outros métodos de sincronização.

```
9.  ...
10. public Class Test{
11.       ...
12.       public void getAccess(){
13.            ...
14.       }
15.       ...
16. }
```

Código 5 – Código fonte sem sincronização

4.1.2 Alteração no Código Fonte: Impasses

Com relação aos impasses, situações nas quais ciclos de dependência ocorrem foram inseridas no código fonte. Um exemplo da natureza dessas alterações é mostrado no Código 6.

```
1.  public Class SimpleDeadLock {
2.        public static Object resource1 = new Object();
3.        public static Object resource2 = new Object();
4.        ...
5.        private static class Thread1 implements Runnable {
6.              public void run() {
7.                   synchronized (resource1) {
8.                        ...
9.                        try { Thread.sleep(10);
10.                            } catch (InterruptedException e) {}
11.                                ...
12.                            synchronized (resource2) {
13.                                ...
14.                            }
15.                  }
16.             }
17.       }
18.       private static class Thread2 implements Runnable {
19.                 public void run() {
```

```
20.                     synchronized (resource2) {
21.                         ...
22.                         try { Thread.sleep(10);
23.                         } catch (InterruptedException e) {}
24.                             ...
25.                         synchronized (resource1) {
26.                             ...
27.                         }
28.                     }
29.                 }
30.         }
31.  }
```

Código 6 – Código alterado para simular impasses

Situações como a mostrada no Código 6 foram implementadas no código fonte dos programas em teste para simular a ocorrência de impasses.

4.1.3 Alterações no Código Fonte: Comportamentos errôneos

Como comportamentos incorretos pode-se entender a execução de métodos fora da ordem esperada, de modo a alterar o resultado final do programa e prejudicar seu funcionamento. A simulação desse tipo de comportamento se deu através de modificações no próprio fluxo de execução do software, alterando a ordem de chamada dos métodos do programa em teste.

As seções que se seguem são dedicadas à apresentação dos sistemas de software que foram objetos do estudo de caso. Posteriormente são expostos os resultados dos testes realizados.

4.2 MultCare

O software utilizado como primeiro estudo de caso foi o **MultCare** (SOBRINHO, 2013). Trata-se de um aplicativo Android que auxilia o diagnóstico precoce da Doença Renal Crônica (DRC). Basicamente, o aplicativo permite que o usuário insira dados referentes à sua saúde, obtidos por meio de exames laboratoriais, e obtenha diagnóstico acerca da Doença Renal Crônica, maiores detalhes podem ser encontrados no trabalho "Estudo e Desenvolvimento de uma Solução para o Auxílio ao Cuidado à Saúde" (SOBRINHO, 2013).

Os testes foram concentrados no módulo de **análise de risco para DRC**. Nos outros módulos as threads existentes realizam atividades independentes, não havendo, assim, a possibilidade de ocorrência de impasses ou condições de corrida. O comportamento do módulo em questão é apresentado a seguir, sua compreensão é necessária para que as propriedades sejam especificadas de maneira correta.

4.2.1 Análise de Risco para DRC

O módulo de análise de risco para Doença Renal Crônica permite que o usuário insira informações acerca de sua saúde no aplicativo. Após o processamento dessas informações, o aplicativo emite um alerta sobre o risco de desenvolver uma doença renal. Existem três possibilidades de alerta: alto risco, baixo risco, e sem risco.

Parte dos dados fornecidos pelo usuário consiste nas taxas de: creatinina, ureia, potássio e albumina. As taxas são verificadas e classificadas como normal ou anormal. Outro dado informado pelo paciente é a taxa de microalbuminiúria. Seu valor é utilizado para verificar a taxa de proteinúria, que indica a perda de proteína através da urina. O usuário ainda deve inserir a Taxa de Filtração Glomerular (TFG). Ao fim das verificações as taxas de proteinúria TFG são relacionadas, a partir daí é realizada a classificação do estágio da DRC, que pode ser ausente ou existente. Os resultados obtidos são utilizados para realização da análise do risco que o paciente tem de desenvolver DRC. Na Figura 12 é ilustrado o fluxo de execução do componente de análise de risco.

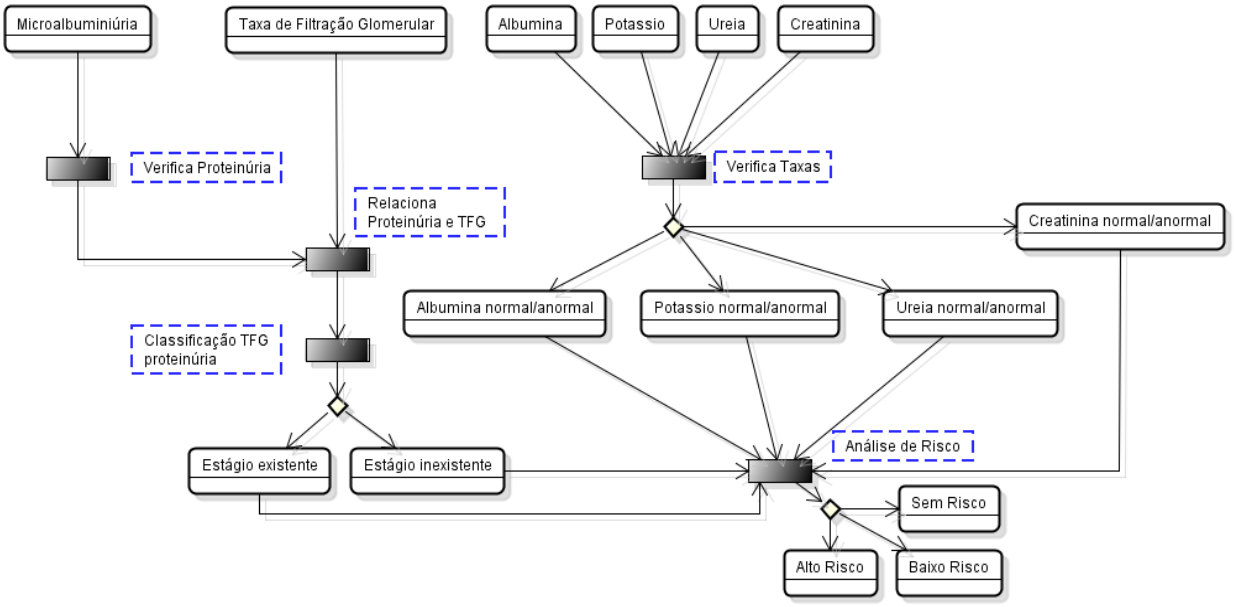

Figura 12 - Fluxo de execução do módulo de análise de risco de DRC.

4.2.2 Instrumentação do Código Fonte

Para a realização da verificação se faz necessária a instrumentação do código fonte do sistema, ou mesmo de uma parte do sistema a qual se deseja testar. No Código 7 é apresentada uma classe do aplicativo MultCare instrumentada:

```
1.  package com.ufal.compe.analise;
2.  import br.ufal.ic.bruno.traces.Traces;
3.  public Class ProteinuriaInfo {
4.          private String proteinuria;
5.          private String resultadoProteinuria;
6.          private boolean ocupado = false;
7.          private static final Traces traces = new Traces(
8.          "C:/Users/Bruno/Documents/Dissertação/danielsanfr-multcare-
            f791bd67a543 (1)/danielsanfr-multcare-f791bd67a543/bin/classes.dex",
9.          "Lcom/ufal/compe/analise/ProteinuriaInfo;");
10.         public synchronized String getProteinuria() {
11.                 while (!ocupado)
12.                     try {
13.                         wait();
14.                     } catch (InterruptedException e) {
15.                         // TODO Auto-generated catch block
16.                         e.printStackTrace();
17.                     }
18.                 traces.getOperation();
19.                 ocupado = false;
20.                 notifyAll();
21.                 return proteinuria;
```

```
22.         }
23.         public synchronized void setProteinuria(String proteinuria) {
24.             while (ocupado)
25.                 try {
26.                     wait();
27.                 } catch (InterruptedException e) {
28.                     // TODO Auto-generated catch block
29.                     e.printStackTrace();
30.                 }
31.             ocupado = true;
32.             this.proteinuria = proteinuria;
33.             this.resultadoProteinuria = proteinuria;
34.             traces.getOperation();
35.             notifyAll();
36.         }
37.         public String getResultadoProteinuria() {
38.             traces.getOperation();
39.             return resultadoProteinuria;
40.         }
41.  }
```

Código 7 – Classe ProteinúriaInfo

O mesmo acontece com todas as outras classes testadas. As informações são extraídas na medida em que o programa é executado.

4.2.3 Resultados Obtidos

Num primeiro momento, a ferramenta MultCare executou normalmente, com ausência de erros. Nessa etapa, nenhum impasse ou condição de disputa foi encontrado. Deste modo, foram necessárias alterações no código fonte do sistema de software em teste, de modo que situações propensas à ocorrência de impasses e condições de corrida fossem criadas. Após nova execução, o método proposto detectou 2 impasses e 4 condições de corrida. A título de comparação, o mesmo procedimento foi adotado com a ferramenta JPF. A aplicação mostrou-se eficiente para detectar impasses. Entretanto, para condições de corrida a eficiência não foi a mesma. Os resultados são ilustrados através da Tabela 2.

Tabela 2 - Resultados MultCare.

Ferramentas/Defeitos	Java Pathfinder	Método proposto
Impasses	2	2
Condições de corrida	2	4

O número de condições de corrida identificado pelo JPF foi menor. Isso se deve ao fato de a ferramenta analisar apenas a sequência de execução atual do programa em teste. Dessa forma, outras possíveis intercalações sequer foram analisadas.

4.2.3.1 Especificação e Análise de Propriedades

Anteriormente, vimos uma breve explicação sobre o comportamento da aplicação MultCare. Seu correto funcionamento depende de um conjunto de funcionalidades. Para testar a efetividade do método proposto neste trabalho foram especificadas e analisadas duas propriedades. O mesmo será aplicado aos estudos de casos subsequentes. A seguir são listados alguns requisitos funcionais que deve ser satisfeitos por MultCare:

1. Para realizar a análise de risco de DRC, deve ocorrer a definição das taxas de creatinina, ureia, potássio e albumina. Se isso não acontecer, não será possível retornar algum resultado.
2. Antes de definir o estágio da DRC, devem ser definidas as taxas de proteinúria e de filtração glomerular. Sem isso, não há como definir em qual estágio de doença renal está o paciente.

A fim de realizar a especificação de propriedades, é necessário mapear os comportamentos enumerados acima em métodos do sistema de *software* MultCare. No modelo, as funcionalidades estão descritas como métodos, deste modo, as propriedades devem ser escritas utilizando-se os nomes dos métodos de MultCare. O estágio da DRC é definido pelo método *classificacaoDRC*. Já a verificação das taxas de creatinina, ureia, potássio e albumina é realizada através do método: *classificaDados()*. A análise de risco de DRC é executada pelo método *analiseDeRisco()*. As taxas de proteinúria e filtração glomerular são realizadas pelos métodos *setProteinuria()*, *setTfg()*.

De acordo com os comportamentos esperados listados, propriedades foram especificadas e apresentadas na Tabela 3.

Tabela 3 - Propriedades MultCare.

Comportamento	Propriedade
1	*CTLSPEC AG* ((state = classificaDados) → *AG* (state = analiseDeRisco))
2	*CTLSPEC AG* ((state=setProteinuria) →*AF*(state=classificacaoTFGProteinuria))

Modificações no código fonte de MultCare foram necessárias para a verificação das propriedades especificadas. Na Figura 13 é apresentado o resultado da verificação da propriedade 1. O contraexemplo retornado indica a sequência de execução que invalidou a propriedade.

```
-- specification AG (state = classificaDados -> AX state = analiseDeRisco) is false
-- as demonstrated by following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  state = setProteinuria
  time = 11
-> State: 1.2 <-
  state = analiseDeRisco
  time = 12
-> State: 1.3 <-
  state = classificaDados
  time = 13
-> State: 1.4 <-
  state = classificacaoDRC
  time = 14
-- Loop starts here
-> State: 1.5 <-
  state = setTfg
  time = 15
```

Figura 13 - Resultado da verificação da propriedade 1.

Na Figura 14 é ilustrado o contraexemplo retornado após a verificação da propriedade 2. Note que o método *classificacaoDRC* foi executado antes do método *setTfg*, configurando inconformidade com a funcionalidade desejada.

```
-- specification AG (state = setProteinuria -> AF state = classificacaoDRC)
& (state = setTfg -> AF state = classificacaoDRC)) is false
-- as demonstrated by following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  state = setProteinuria
  time = 11
-> State: 1.2 <-
  state = classificacaoDRC
  time = 12
-> State: 1.3 <-
  state = classificaDados
  time = 13
-> State: 1.4 <-
  state = setTfg
  time = 14
```

Figura 14 - Resultado da verificação da propriedade 2.

Os erros indicados foram corrigidos e as propriedades novamente verificadas. Na Figura 15 é apresentado resultado de uma nova verificação, desta vez com ausência de erros.

```
-- specification AG (state = classificaDados -> AF state = analiseDeRisco) is true
-- specification AG ((state = setProteinuria -> AF state = classificacaoDRC)
& (state = setTfg -> AF state = classificacaoDRC)) is true
```

Figura 15 - Verificação do modelo do MultCare após correção de código fonte.

4.3 JavaChat

JavaChat[7] é um programa de bate-papo escrito em Java. Ele permite que usuários conectados a um servidor conversem entre si. Basicamente, os usuários clientes realizam autenticação no servidor, que por sua vez intermedeia toda a comunicação subsequente. Analogamente a seção anterior, os testes ocorreram com base em simulações de situações que não deveriam acontecer.

[7] http://javachat.sourceforge.net

4.3.1 Resultados Obtidos

Ao executar JavaChat não foi detectado nenhum impasse ou condições de corrida. Através de alterações no código fonte do JavaChat, foram criadas condições que proporcionaram o surgimento desses defeitos. A partir daí simulações foram realizadas e os resultados são mostrados na Tabela 4. Os testes também foram feitos utilizando JPF.

Tabela 4 - Resultados JavaChat.

Ferramentas/Defeitos	Java Pathfinder	Método proposto
Impasses	3	5
Condições de corrida	2	3

Pelos mesmos motivos levantados no estudo de caso anterior, o método proposto foi capaz de detectar mais impasses e condições de corrida que JPF.

4.3.1.1 Especificação e Análise de Propriedades

Alguns comportamentos são requeridos para que o JavaChat funcione corretamente. Alguns comportamentos esperados são expressados logo abaixo e, posteriormente, propriedades comportamentais são especificadas.

1. Antes de iniciar qualquer comunicação com outros usuários, o cliente deve fazer o login no servidor. A operação de login no JavaChat envolve verificar se o usuário já está autenticado e em seguida averiguar se o *password* digitado pelo usuário é válido. Essas operações são executadas através dos métodos *login()* e *checkUserNamePassword()*, respectivamente.
2. Além do login, todas as vezes que uma instância do servidor recebe uma mensagem ela é recebida e logo após processada. Tais tarefas são realizadas pelos métodos *receiveObject()* e *processReceiveObject()*, respectivamente.

A partir da especificação dos comportamentos desejados, as propriedades podem ser devidamente escritas. Tais propriedades são apresentadas a seguir:

Tabela 5 - Propriedades JavaChat.

Comportamento	Propriedade
1	*CTLSPEC AG* (state = login →*AX* state = checkUserNamePassword)
2	*CTLSPEC AG* (state = receiveObject → *AX* state = processReceiveObject)

A propriedade 1 indica que o método *checkUserNamePassword()* deve operar sempre que o método *login()* é executado. A propriedade 2 indica que é obrigatória a execução do método *processReceiveObject()* logo após a execução de *receiveObject()*.

Alterações no código fonte de JavaChat foram realizadas para que comportamentos incorretos pudessem ocorrer. Para melhor visualização dos resultados, cada propriedade foi testada em separado. Na Figura 16 é apresentado o resultado da verificação da propriedade 1 sobre o modelo de JavaChat. De acordo com o contraexemplo retornado, o programa apresentou comportamento indesejado. O método *checkUserNamePassword()* não está sendo executado logo após a operação *login()*, o que denota não conformidade com a funcionalidade desejada.

```
-- specification AG (state = login -> AX state = checkUserNamePassword) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  state = someoneConnected
  time = 1
-> State: 1.2 <-
  state = receiveObject
  time = 2
-> State: 1.3 <-
  state = processReceivedObject
  time = 3
-> State: 1.4 <-
  state = checkUserNamePassword
  time = 4
-> State: 1.5 <-
  state = login
  time = 5
-> State: 1.6 <-
  state = receiveObject
  time = 6
```

Figura 16 - Resultado da verificação da propriedade 1.

Na Figura 18 é apresentado o resultado da verificação da propriedade 2. O contraexemplo demonstra em qual situação o modelo de JavaChat não satisfez à propriedade 2.

```
-- specification AG (state = receiveObject -> AX state = processReceiveObject) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  state = someoneConnected
  time = 1
-> State: 1.2 <-
  state = processReceivedObject
  time = 2
-> State: 1.3 <-
  state = receiveObject
  time = 3
-> State: 1.4 <-
  state = login
  time = 4
```

Figura 17 - Resultado da verificação da propriedade 2.

O código fonte foi corrigido e a análise e verificação realizadas novamente. O resultado é apresentado na Figura 18. De acordo com a Figura 18 as propriedades foram satisfeitas.

```
-- specification (AF (state = compareFiles -> AF state = getFileList) &
AF (state = getFileList -> !(AF state = compareFiles))) is true
-- specification (AF (state = compareFiles -> AF state = synchronize) &
AF (state = synchronize -> !(AF state = compareFiles))) is true
```

Figura 18 - Verificação do modelo do JavaChat após correção de código fonte.

4.4 JFileSync

Outro sistema de software testado foi JFileSync.[8] O programa é usado para sincronizar pares de diretórios. Ele é útil quando se deseja que um conjunto de arquivos existente em uma estação de trabalho corresponda a um conjunto contido no computador pessoal do usuário, ou mesmo em uma unidade de dados externa. Para sincronizar diretórios, três passos devem ser seguidos:

[8] http://jfilesync.sourceforge.net

- Definem-se quais diretórios se deseja sincronizar;
- Executa uma comparação de todos os diretórios especificados;
- Inicia a sincronização e os passos de confirmação das partes dos diretórios que serão copiados ou deletados.

Para sincronizar diretórios, é necessário que se crie um perfil de sincronização. Nesse perfil ficam armazenadas as configurações desejadas para o processo de sincronização, tais como, quais diretórios serão envolvidos e o modo de sincronização utilizado.

4.4.1 Resultados Obtidos

Alterações no código fonte criaram as condições suficientes para ocorrência dos defeitos relacionados à concorrência. Na Tabela 6Tabela 6 - Resultados obtidos testando JFileSync. são comparados os resultados fornecidos pelo método proposto com os da ferramenta JPF. Com relação a impasses, mais uma vez o JPF revelou uma quantidade inferior de falhas. Já os testes correspondentes a condições de corrida novamente mostraram que mais condições de corrida puderam ser detectadas com o método proposto.

Tabela 6 - Resultados obtidos testando JFileSync.

Ferramentas/Defeitos	Java Pathfinder	Método proposto
Impasses	2	3
Condições de corrida	1	2

4.4.1.1 Especificação e Análise das Propriedades

O correto funcionamento de JFileSync depende de algumas funcionalidades fundamentais, tais como:

1. A sincronização de diretórios está condicionada à comparação dos mesmos;
2. Para que os diretórios sejam comparados, é necessário coletar os arquivos dos mesmos.

JFileSync realiza a comparação entre diretórios é executada pelo método *compareFiles()*. Já a operação de sincronização é realizada pelo método *synchronize()*. A coleta de arquivos é realizada por *getFileList()*.

Deste modo, as propriedades podem ser especificadas. Elas são apresentadas na Tabela 7.

Tabela 7 - Propriedades especificadas para JFileSync.

Comportamento	Propriedade
1	*CTLSPEC AF*(((state=compareFiles) → *AF*(state=getFileList))&*AF* !((state=compareFiles) → *AF*(state=getFileList)))
2	*CTLSPEC AF* ((state=compareFiles) → *AF*(state=synchronize)) & *AF* ((state=synchronize) → *AF*(state=compareFiles))

Analogamente às seções anteriores, mudanças no código fonte do programa em teste foram necessárias para realização da verificação. As propriedades também são provadas individualmente para melhor visualização dos resultados. Na Figura 19 é apresentado o resultado da verificação da primeira propriedade. São mostrados contraexemplos para expressar a inconformidade do comportamento de JFileSync com o comportamento esperado.

```
-> State: 1.56 <-
  state = setRootUriTgt
  time = 56
-> State: 1.57 <-
  state = computeSynchronizationLists
  time = 57
-> State: 1.58 <-
  state = invert
  time = 58
-> State: 1.59 <-
  state = synchronize
  time = 59
-> State: 1.60 <-
  state = compareFiles
  time = 60
-> State: 1.61 <-
  state = getFileList
  time = 61
```

Figura 19 - Resultado da verificação da propriedade 1.

A mesma situação é ilustrada na Figura 20 com relação à propriedade 2.

```
-> State: 1.55 <-
  state = setRootUriSrc
  time = 55
-> State: 1.56 <-
  state = setRootUriTgt
  time = 56
-> State: 1.57 <-
  state = computeSynchronizationLists
  time = 57
-> State: 1.58 <-
  state = invert
  time = 58
-> State: 1.59 <-
  state = synchronize
  time = 59
```

Figura 20 - Resultado da verificação da propriedade 2.

Na Figura 21 é apresentado o resultado retornado por NuSMV após a correção do código fonte.

```
-- specification AG (state = getFileList -> AF state = compareFiles) is true
-- specification AG (state = compareFiles -> AF state = synchronize) is true
```

Figura 21 - Verificação do modelo do JFileSync após correção de código fonte.

5 CONCLUSÕES E TRABALHOS FUTUROS

Com o avanço do *hardware*, sistemas de *software* que proporcionem a otimização do uso de recursos computacionais tornaram-se essenciais. Sistemas de *software* com tal capacidade podem ser construídos utilizando-se o paradigma da programação concorrente. Devido ao comportamento não determinístico dos programas construídos com a estratégia, estes estão sujeitos a defeitos peculiares como impasses e condições de corrida. Por essa razão, mecanismos específicos para testes se fazem necessários.

O método proposto neste trabalho se mostra como uma possibilidade para identificação desses defeitos. Mais especificamente, a proposta consiste em um método capaz de buscar automaticamente os defeitos intrínsecos a sistemas de softwares concorrentes, impasses e condições de corrida. Além disso, o método também possibilita a analise de propriedades comportamentais, úteis para verificar se um programa funciona ou não como o desejado. Tudo isso se apoiando na técnica de verificação de modelos.

A verificação de modelos permite que seja gerado um modelo o qual descreve o comportamento de sistemas de *software*. Tal modelo é utilizado em um processo de verificação contra propriedades que expressam o comportamento que um sistema de *software* deveria, ou não, apresentar. Contudo, a construção de deste modelo é uma atividade bastante dispendiosa. O método proposto neste trabalho automatiza a geração desse modelo, eliminando assim, o custo de sua elaboração. Isso é possível em razão da extração das informações requeridas para a modelagem durante a execução do sistema de software em teste. As informações também são usadas para identificação automática de impasses e condições de corrida.

O diferencial do método está em sua capacidade de capturar sequências de execução que podem influenciar o resultado retornado por um sistema de *software* concorrente dinamicamente. Essa característica é importante, pois permite que todos os caminhos de execução sejam capturados sem que haja necessidade de nenhum tipo de previsão de comportamentos futuros, como acontece em ferramentas de verificação estática. Dessa forma, é possível analisar o comportamento real de um sistema de *software* concorrente e, por conseguinte, prover maior confiança aos resultados retornados pelas verificações. Por essa razão, o método proposto neste trabalho pode contribuir para construção de sistemas de *software* mais confiáveis, na medida em que proporciona a capacidade de identificação de

defeitos em sequências de execução com baixa probabilidade de ocorrência e que, em algum momento futuro, podem se manifestar.

5.1 Trabalhos Futuros

No contexto do presente trabalho, podem ser sugeridos alguns temas como possíveis trabalhos futuros:

- Oferecer métodos de instrumentação que não necessitem ser inseridos em todas as operações do programa em teste. Deste modo, pretende-se reduzir o esforço do desenvolvedor;
- Estender as funcionalidades do método proposto adicionando a capacidade de verificar e implementar pré-condições, pós-condições e invariantes em programas *multithreaded*;
- Estender as funcionalidades do método para verificação de tipos;
- Aumentar o desempenho de método proposto incorporando características de verificação estática à ferramenta;
- Realizar avaliação da ferramenta em vários sistemas de diferentes domínios.

5.2 Bibliografia

AGARWAL, R. A. S. S. D. **Run-time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables**. Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging. [S.l.]: ACM. 2006. p. 51--60.

ALGLAVE, J.; KROENING, D.; TAUTSCHNIG, M. **Partial Orders for Efficient Bounded Model Checking of Concurrent Software**. Proceedings of the 25th International Conference on Computer Aided Verification. [S.l.]: Springer-Verlag. 2013. p. 141--157.

ARTHO, C. A. B. A. Applying static analysis to large-scale, multi-threaded Java programs. **Software Engineering Conference**, 21 Fevereiro 2001. 68--75.

ARTHO, C.; BIERE, A. **Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs**. Proceedings of the 13th Australian Conference on Software Engineering. [S.l.]: IEEE Computer Society. 2001.

AVIRAM, A. et al. Efficient System-enforced Deterministic Parallelism. **Commun. ACM**, Maio 2012. 111--119.

BAIER, C.; KATOEN, J.-P. **Principles of Model Checking (Representation and Mind Series)**. [S.l.]: The MIT Press, 2008.

BERGHOFER, S. et al. VCC: A Practical System for Verifying Concurrent C. In: BERGHOFER, S., et al. **Theorem Proving in Higher Order Logics:** 22nd International Conference. [S.l.]: Springer Berlin Heidelberg, 2009. p. 23--42.

BERTOLINO, A. Software Testing Research: Achievements, Challenges, Dreams. **2007 Future of Software Engineering**, 2007. 85--103.

BIN LEI, L. W. A. X. L. **UML Activity Diagram Based Testing of Java Concurrent Programs for Data Race Condition and Inconsistency**. Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation. [S.l.]: IEEE Computer Society. 2008. p. 200--209.

BINDER, R. V. **Testing Object-oriented Systems:** Models, Patterns, and Tools. [S.l.]: Addison-Wesley Professional, 1999.

BLOOMBERG. Knight Shows How to Lose $440 Million in 30 Minutes., 2012. Disponivel em: <http://www.bloomberg.com/bw/articles/2012-08-02/knight-shows-how-to-lose-440-million-in-30-minutes>. Acesso em: 15 Janeiro 2013.

BRUNETON, E. **ASM 4.0: A Java Bytecode Engineering Library**. [S.l.]. 2011.

BUSTARD, D. W. **Concepts of Concurrent Programming**. University of Ulster. [S.l.]. 1990.

CAVADA, R. et al. **NuSMV 2.2 User Manual**. [S.l.]. 2005.

CAVADA, R. et al. **NuSMV Tutorial**. [S.l.]. 2005.

CHARETTE, R. N. Why Software Fails [Software Failure]. **IEEE Spectr.**, Setembro 2005. 42--49.

CHEN, H. Y. **Race condition and concurrency safety of multithreaded object-oriented programming in Java**. Systems, Man and Cybernetics, 2002 IEEE International Conference on. [S.l.]: IEEE. 2002. p. 6.

CHU, G.-H. H. E. S.-J. C. E. H.-D. Technology for Testing Nondeterministic Client/Server Database Applications. **IEEE Transactions on Software Engineering**, Janeiro 2004. 59--77.

COHEN, E. et al. **Local Verification of Global Invariants in Concurrent Programs**. Proceedings of the 22Nd International Conference on Computer Aided Verification. [S.l.]: Springer-Verlag. 2010. p. 480--494.

DE MOURA, L. A. B. N. **Z3:** An Efficient SMT Solver. Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. [S.l.]: Springer-Verlag. 2008. p. 337--340.

DIJKSTRA, E. W. Cooperating Sequential Processes. In: HANSEN, P. B. **The Origin of Concurrent Programming**. [S.l.]: Springer-Verlag New York, Inc., 2002. p. 65--138.

DIMITRA GIANNAKOPOULO, C. S. P. H. B. Component Verification with Automatically Generated Assumptions. **Automated Software Engineering**, Julho 2002. 297--320.

D'SILVA, V. A. K. D.; WEISSENBACHER, G. A Survey of Automated Techniques for Formal Software Verification. **Trans. Comp.-Aided Des. Integ. Cir. Sys.**, Julho 2008. 1165--1178.

EDELSTEIN, O. et al. Multithreaded Java Program Test Generation. **IBM Syst. J.**, Janeiro 2002. 111--125.

FLANAGAN, C. A. F. S. N. **FastTrack:** Efficient and Precise Dynamic Race Detection. Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. [S.l.]: ACM. 2009. p. 121--133.

FLANAGAN, C. A. G. P. **Dynamic Partial-order Reduction for Model Checking Software**. Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. [S.l.]: ACM. 2005. p. 110--121.

GABBAR, H. A. **Modern Formal Methods and Applications**. [S.l.]: Springer, 2006.

GARVIN, D. A. Competing on the eitgh dimensions of quality. **Harvard Business Review**, v. 65, n. 6, Dezembro 1987.

GROGONO, P.; SHEARING, B. **Concurrent Software Engineering:** Preparing for Paradigm Shift. ASWEC '01 Proceedings of the 13th Australian Conference on Software Engineering. [S.l.]: ACM. 2008. p. 99--108.

HANSEN, P. B. A Keynote Address on Concurrent Programming. **Computer Software and Applications Conference, 1978. COMPSAC '78. The IEEE Computer Society's Second International**, 1979.

HANSEN, P. B. **The Nature of Parallel Programming**. Syracuse University. [S.l.]. 1989.

HANSEN, P. B. Distributed Processes: A Concurrent Programming Concept. In: HANSEN, P. B. **The Origin of Concurrent Programming**. [S.l.]: Springer-Verlag New York, Inc., 2002. p. 444--463.

HANSEN, P. B. The Invention of Concurrent Programming. In: HANSEN, P. B. **The Origin of Concurrent Programming**. [S.l.]: Springer-Verlag New York, Inc., 2002. p. 3--61.

HILDRETH, S. **Site da ComputerWorld**, 2005. Disponivel em: <http://www.computerworld.com/article/2557403/app-development/buggy-software--up-from-a-low-quality-quagmire.html>. Acesso em: 2013.

ISLOOR, S. S. A. M. T. A. The Deadlock Problem: An Overview. **Computer**, Setembro 1980. 58--78.

JIM WOODCOCK, P. G. L. J. B. J. F. Formal Methods: Practice and Experience. **ACM Computing Surveys (CSUR)**, Outubro 2009. 19:1--19:36.

JOSÉ BACELAR ALMEIDA, M. J. F. J. S. P. S. M. D. S. An Overview of Formal Methods Tools and Techniques. In: _____ **Rigorous Software Development:** An Introduction to Program Verification. [S.l.]: Springer London, 2011. p. 15--44.

JOURNAL, W. S. Honda to Recall 2.26 Million Vehicles in U.S., 2011. Disponivel em: <http://www.wsj.com/articles/SB10001424053111903454504576489941069922556>. Acesso em: 22 Janeiro 2013.

K.RUSTANM.LEINO, M. B. A. B.-Y. E. C. A. R. D. A. B. J. A. **Boogie:** A Modular Reusable Verifier for Object-Oriented Programs. FMCO'05 Proceedings of the 4th international conference on Formal Methods for Components and Objects. [S.l.]: Springer-Verlag. 2005.

KEIL, M. et al. **Top 10 Corporate Information Technology Failures**. ComputerWorld. [S.l.]. 2002.

LEE, E. A. The Problem with Threads. **Computer**, Maio 2006. 33--42.

LEMAY, E.; SCARFONE, K. A.; MELL, P. M. **The Common Misuse Scoring System (CMS): Metrics for Software Feature Misuse Vulnerabilities**. The National Institute of Standards and Technology (NIST). [S.l.]. 2012.

LEVESON, N. **Medical Devices: The Therac-25**. University of Washington. [S.l.]. 1995.

LEVESON, N. G. A. T. C. S. An Investigation of the Therac-25 Accidents. **Computer**, Julho 1993. 18--41.

MATTERN, F. **Virtual Time and Global States of Distributed Systems**. Proc. Workshop on Parallel and Distributed Algorithms. [S.l.]: Cosnard M. et al. 1989. p. 215--226.

MCCALL, J. A.; RICHARDS, P. K.; WALTERS, G. F. **Factors in Software Quality: Concepts and Definitions of Software Quality**. United States Air Force. [S.l.]. 1977.

MEI-CHEN HSUEH, T. K. T. R. K. I. Fault Injection Techniques and Tools. **Computer**, Los Alamitos, CA, USA, Abril 1997. 75--82.

MEYER, B. Applying "Design by Contract". **Computer**, Outubro 1992. 40--51.

MUSUVATHI, M. CHESS: A Systematic Testing Tool for Concurrent Software. In: PUEBLA, G. **Logic-Based Program Synthesis and Transformation:** 16th International Symposium, LOPSTR. [S.l.]: Springer Berlin Heidelberg, 2007. p. 15--16.

NBR ISO-IEC 9126-1. Associação Brasileira de Normas Técnicas. [S.l.]. 2003.

NEVON, J. **Adaptation de la bibliotheque ASM pour Dalvik**. [S.l.]. 2011.

O. BABAOGLU, K. M. F. S. A Formalization of Priority Inversion. **Real-Time Systems**, 1993. 285--303.

OLIVEIRA, E. A. D. S. **Uma Técnica para Modelagem e Verificação de Programas Java Concorrentes Auxiliada por Anotações de Código**. UNIVERSIDADE FEDERAL DE CAMPINA GRANDE. Campina Grande. 2006.

PAULO SALEM DA SILVA, A. C. V. D. M. Model Checking Merged Program Traces. **Electronic Notes in Theoretical Computer Science (ENTCS)**, 06 Julho 2009. 97--112.

POZNIANSKY, E. A. S. A. MultiRace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs: Research Articles. **Concurr. Comput. : Pract. Exper.**, Março 2007. 327--340.

PRESSMAN, R. S. **Engenharia de Software - Uma Abordagem Profissional**. 7. ed. [S.l.]: McGrawHill, 2011.

SAARIKIVI, O. A. K. K. A. H. K. Improving Dynamic Partial Order Reductions for Concolic Testing. **Proceedings of the 2012 12th International Conference on Application of Concurrency to System Design**, 2012. 132--141.

SADDEK BENSALEM, K. H. Dynamic Deadlock Analysis of Multi-Threaded Programs. [S.l.]: Springer Berlin Heidelberg, 2006. Cap. 15.

SAVAGE, S. A. B. M. A. N. G. A. S. P. A. A. T. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. **ACM Trans. Comput. Syst.**, Novembro 1997. 391--411.

SINHA, N.; WANG, C. **Staged Concurrent Program Analysis**. Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering. [S.l.]: ACM. 2010. p. 47--56.

SOBRINHO, Á. Á. D. C. C. Uma metodologia de desenvolvimento centrado no usuário para um sistema de prevenção e diagnóstico precoce da Doença Renal Crônica. **CSBC 2013**, 2012.

SOBRINHO, Á. Á. D. C. C. **Estudo e Desenvolvimento de uma Solução para o Auxílio ao Cuidado à Saúde**. Universidade Federal de Alagoas. [S.l.]. 2013.

SOMMERVILLE, I. **Engenharia de Software**. 8. ed. [S.l.]: PEARSON EDUCATION, 2007.

SOUSA, M. A. B. D. **A Framework for Formal Verification of Concurrent**. Utrecht Universiteit. [S.l.]. 2012.

TASSEY, G. **The economic impacts of inadequate Infraestructure for Software Testing**. National Institute of Standards and Technology. [S.l.]. 2002.

VAN DER MERWE, H. A. V. D. M. B. A. V. W. Verifying Android Applications Using Java PathFinder. **SIGSOFT Softw. Eng. Notes**, Novembro 2012. 1--5.

VASCONCELOS, G. D. M. **Verificação automática de programas a partir da monitorando de múltiplas execuções de código**. Universidade Federal de Campina Grande. [S.l.]. 2012.

VASCONCELOS, G.; SILVA, L. D.; PERKUSICH, A. **Verificação automática de programas a partir da monitorando de múltiplas execuções de código**. XIX Congresso Brasileiro de Automática. Campinas, 2012, Campina Grande. Anais do XIX Congresso Brasileiro de Automática. [S.l.]: [s.n.]. 2012.

VISSER, G. B. A. K. H. A. S. P. A. W. **Java PathFinder - Second Generation of a Java Model Checker**. In Proceedings of the Workshop on Advances in Verification. [S.l.]: [s.n.]. 2000.

WELC, A. A. H. A. L. A. J. S. **Preemption-Based Avoidance of Priority Inversion for Java**. Proceedings of the 2004 International Conference on Parallel Processing. [S.l.]: IEEE Computer Society. 2004. p. 529--538.

YANG, Y. **Efficient Dynamic Verification of Concurrent Programs**. University of Utah. [S.l.]. 2009.

ZIADE, H. E. A. R. A. E. V. R. A Survey on Fault Injection Techniques. **Int. Arab J. Inf. Technol.**, 29 Setembro 2004. 171-186.