PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Programa de Pós-Graduação em Informática

Otmar Martins Pereira Junior

MINERAÇÃO DE SEQUÊNCIAS DE CHAMADAS A APIs DIRIGIDA POR TESTES UNITÁRIOS

Belo Horizonte

2015

Otmar Martins Pereira Junior

MINERAÇÃO DE SEQUÊNCIAS DE CHAMADAS A APIs DIRIGIDA POR TESTES UNITÁRIOS

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Mestre em Informática.

Orientador: Prof. Dr. Mark Alan Junho Song

Belo Horizonte

2015

FICHA CATALOGRÁFICA
Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

Pereira Junior, Otmar Martins
P436m Mineração de sequências de chamadas a APIs dirigida por testes unitários / Otmar Martins Pereira Junior. Belo Horizonte, 2015.
65f. : il.

Orientador: Mark Alan Junho Song
Dissertação (Mestrado) – Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática.

1. Métodos formais (Computação). 2. Interface de programas aplicativos (Software). 3. Especificações. 4. Mineração de dados (Computação). 5. Sequências (Matemática). I. Song, Mark Alan Junho. II. Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática. III. Título.

SIB PUC MINAS

CDU: 681.3.011

Otmar Martins Pereira Junior

MINERAÇÃO DE SEQUÊNCIAS DE CHAMADAS A APIs DIRIGIDA POR TESTES UNITÁRIOS

Dissertação apresentada ao Programa de Pós-Graduação em Informática como requisito parcial para qualificação ao Grau de Mestre em Informática pela Pontifícia Universidade Católica de Minas Gerais.

Prof. Dr. Mark Alan Junho Song – PUC Minas (Orientador)

Prof. Dr. Marcelo de Almeida Maia – UFU (Banca Examinadora)

Prof. Dr. Zenilton Kleber Gonçalves do Patrocínio Júnior – PUC Minas (Banca Examinadora)

Prof. Dr. Wladmir Cardoso Brandão – PUC Minas (Banca Examinadora)

Belo Horizonte, 21 de setembro de 2015.

Dedico este trabalho à minha esposa Amanda, pelo suporte incondicional que me permitiu realizar este trabalho.

AGRADECIMENTOS

Agradeço primeiramente a Deus pelos dons concedidos e à minha esposa Amanda, por ter sido importante patrocinadora deste projeto e também ter me dado o suporte e apoio emocional durante toda a trajetória deste trabalho de pesquisa. Ao seu lado, pude também evoluir academicamente.

Aos meus pais, Otmar e Izabel, agradeço pela criação e educação, exemplos inspiradores para toda minha vida nos mais diversos aspectos. Seus grandes esforços me permitiram ter uma boa educação e suas lições serão para toda a vida.

Aos meus orientadores, Mark e Wladmir, agradeço pelos ensinamentos, cordialidade e inspiração em cada um de nossos encontros. A cada reunião eu me sentia instigado a melhorar a pesquisa e perseguir os melhores caminhos indicados por eles.

Agradeço à PUC Minas pelas parcerias que resultaram em uma bolsa de estudos que contribuiu financeiramente para este trabalho. À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), agradeço pelo apoio financeiro dado aos trabalhos desenvolvidos e pela colaboração para o desenvolvimento do ensino no país. À Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG) e Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), também agradeço pelo suporte financeiro dado à apresentação dos resultados obtidos com as pesquisas conduzidas.

À Giovana agradeço pela disponibilidade e energia positiva na administração da secretaria e facilidade para lidar com os trâmites internos da vida acadêmica.

Aos colegas e amigos de Mestrado agradeço pela troca de ideias que geraram importantes reflexões: Matheus Souza, Sebastião, Matheus Queiroz, Hayala, Paulo, Anselmo, Pedro e a todos os que compartilharam das alegrias e dificuldades durante estes anos de pesquisa.

Finalmente, gostaria de agradecer aos pesquisadores Choongwan Lee, Michale Pradel, Claire Le Goues, Ricardo Terra que sempre foram solícitos para elucidar detalhes técnicos de suas pesquisas.

"É impossível para um homem aprender aquilo que ele acha que já sabe."

Epiteto

RESUMO

Desde o final do século XX, quando sistemas de software começaram a ser desenvolvidos em escalas cada vez maiores, e com nível de complexidade crescente, altas taxas de defeitos vêm sendo um dos principais desafios enfrentados. Este desafio se mostra de grande relevância especialmente no contexto de bibliotecas de software que possibilitam altas taxas de reuso para clientes destas aplicações por meio de uma *Application Programming Interface* (API), que nem sempre é de fácil compreensão. Aprender a utilizar corretamente APIs se torna importante para o aumento da qualidade de sistemas e redução de defeitos.

Dentre as diferentes abordagens propostas para recomendar e educar programadores na utilização de uma biblioteca, se encontra a MINERAÇÃO DE ESPECIFICAÇÃO. O objetivo é a inferência de diferentes propriedades e comportamentos de um software a partir de diferentes artefatos de software, com uma representação baseada em formalismos matemáticos. Este trabalho propõe o emprego de técnicas de mineração de dados para aprendizado de sequências de chamadas válidas à APIs com altas taxas de precisão. Esta dissertação demonstra que os testes unitários de uma biblioteca de software podem ser efetivos para apoiar esta atividade e facilitar a compreensão e utilização correta da API pelos programadores de aplicações clientes.

Palavras-chave: Métodos Formais, Mineração de Especificação, Especificação Formal

ABSTRACT

Since the end of 20th century, when software systems began to be developed in ever larger scales, and with ever increasing complexity, high defect rates have been one of the main challenges faced. This issue is relevant in the context of software libraries which enable high reuse rates for their clients via an Application Programming Interface (API), which is not always easily understood. Learning how to correctly use such APIs can greatly contribute to increase the software quality and reduce defects rate.

Among the different approaches to recommend and educate programmers in library usage, there is SPECIFICATION MINING. The goal is the inference of behaviors and properties of a library based on different software artifacts using a mathematical framework in the specification description. This dissertation proposes the use of data mining techniques to learn valid sequences of calls to an API with high precision rates. This work demonstrates that unit tests bundled with a software library can effectively enable the inference process and make it easier for client developers comprehend correct API usage.

Keywords: Formal Methods, Specification Mining, Formal Specification.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API - Interface de Programação de Aplicação, do inglês *Application Programming Interface*

CORES - Especificações formais colaborativas de referência, do inglês **CO**llaborative **RE**ference **S**pecifications

CUT - Classe-sob-teste, do inglês, *Class-under-Test*

DFA - Autômato Finito Determinístico, do inglês *Deterministic Finite State Automaton*

FSA - Autômato de Estados Finitos, do inglês, *Finite State Automaton*

IPv6 - versão 6 do Protocolo de Internet.

JDK - Conjunto de biblioteca e outras ferramentas utilizadas para a construção de programas Java, do inglês, *Java Development Kit*

LTL - Lógica temporal linear, do inglês *linear temporal logic*.

MUTE - Minerando testes unitários, técnica desenvolvida neste trabalho, do inglês **M**ining **U**nit **TE**sts

PFSA - Autômato Probabilístico de Estados Finitos, do inglês, *Probabilistic Finite State Automaton*

URI - Identificador uniforme de recurso, do inglês *Universal Resource Identifier*, que corresponde a uma cadeia de caracteres utilizada para representar o endereço de um recurso.

SUMÁRIO

1 INTRODUÇÃO

Sommerville afirma em (SOMMERVILLE, 2010) que os defeitos de software vêm infestando a indústria desde a década de 1960, quando os sistemas baseados em tecnologia de informação começaram a ser utilizados em larga escala.

Devido à presença cada vez maior na vida das pessoas, é importante que os softwares sejam confiáveis e não falhem, pois os mesmos são empregados em diversas funções cujas falhas podem ter efeitos catastróficos, que vão desde equipamentos de lançamento de foguetes até o monitoramento de usinas nucleares. Além do uso de softwares para missões críticas, tais desafios de confiabilidade e precisão são agravados com a crescente complexidade dos requisitos impostos para os sistemas, como a obtenção de tempos de respostas reduzidos apesar do número crescente de variáveis que devem ser consideradas nos cálculos.

Uma das principais abordagens propostas para atender ao requisito de maior confiabilidade dos sistemas é a Verificação Formal, que consiste no emprego de técnicas matemáticas para se atestar a conformidade do software com as propriedades desejadas, como por exemplo, a impossibilidade de um estado perigoso em que o forno esteja ligado e com a porta aberta, ou um dispositivo de controle de estabilidade do carro que não é acionado quando a velocidade em curva ultrapassa certo limite de segurança.

Como contraponto da Verificação Formal surge a complexidade do seu emprego: seu uso requer grande habilidade matemática, o que inibe sua adoção por um número maior de pessoas. Com o intuito de aumentar a adoção dos métodos formais, pesquisadores começaram a propor o uso de técnicas que levem à automatização de atividades ligadas aos Métodos Formais, o que hoje é conhecido principalmente pela expressão *mineração de especificação*, cunhada por Ammons, Bodík e Larus (2002).

1.1 Motivação

A utilização de técnicas de mineração de dados para apoiar o time de desenvolvimento na confecção da especificação formal de maneira automática traz consigo os desafios tipicamente presentes nos algoritmos relativos ao aprendizado de máquina, principalmente no que tange à precisão e abrangência[1] do resultado.

Tendo em vista que as técnicas de mineração de especificação se baseiam na análise

[1] Precisão e revocação discutidas no Capítulo 2.

de grande número de artefatos presentes no processo de desenvolvimento, pesquisadores precisam lidar com questões relativas à qualidade do processo de inferência. Dentre o vasto universo de artefatos presentes durante o processo de desenvolvimento de software, é preciso escolher aqueles que melhor contribuam para os processos de inferência. A escolha de fontes de dados com muito ruído resultará em uma especificação com poucas regras úteis para os usuários do software, seja por elas serem redundantes, inválidas ou nunca observadas na prática. Uma métrica frequentemente utilizada para avaliar a precisão dos dados inferidos em uma especificação formal é a precisão, que mede o percentual de regras corretas inferidas, ao comparar em relação ao número total de regras inferidas.

Além do propósito original de uma especificação de software, que é principalmente servir de entrada para verificação do software, a mesma pode servir para diferentes tipos de atividades, como localização automática de defeitos (PRADEL; GROSS, 2012), apoio na compreensão do código-fonte (ROBILLARD et al., 2013; PRADEL; BICHSEL; GROSS, 2010), otimização (LERNER et al., 2005), dentre outros tipos. Para ilustrar o apoio na compreensão de programas, o Código 1 ilustra a utilização da *Application Programming Interface* (API) de comunicação de rede do *Java Development Kit* (JDK), presente na classe `MulticastSender` do projeto livre do servidor de aplicações Apache Tomcat 7. Com uma especificação formal em mãos, é possível verificar se a sequência de chamadas aos métodos da classe `MulticastSocket` está consistente com a especificação.

Código 1 – Sequência de chamadas à classe `MulticastSocket`.

```
1  public class MultiCastSender implements Sender {
2   @Override
3   public int send(String mess) throws Exception {
4      if (s == null) {
5         try {
6            group = InetAddress.getByName(config.getGroup());
7            if (config.getHost() != null) {
8               InetAddress addr = InetAddress.getByName(config.getHost());
9               InetSocketAddress addrs = new InetSocketAddress(addr, config
                    .getMultiport());
10              s = new MulticastSocket(addrs);
11           } else s = new MulticastSocket(config.getMultiport());
12
13           s.setTimeToLive(config.getTtl());
14           s.joinGroup(group);
15        } catch (Exception ex) {
16           log.error("Unable to use multicast: " + ex);
17           s = null;
18           return -1;
19        }
20     }
```

```
21          byte[] buf;
22          buf = mess.getBytes(StandardCharsets.US_ASCII);
23          DatagramPacket data = new DatagramPacket(buf, buf.length, group,
                config.getMultiport());
24          try {
25              s.send(data);
26          } catch (Exception ex) {
27              log.error("Unable to send colllected load information: " + ex);
28              s.close();
29              s = null;
30              return -1;
31          }
32          return 0;
33      }
34
35  }
```

Fonte: Adaptado do código-fonte do projeto *open-source* Tomcat 7 (Apache Software Foundation, 2011).

No exemplo do Código 1, pode-se observar que a seguinte sequência de métodos é executada para a classe `MulticastSocket` ⟨ `<init>()`, `setTimeToLive()`, `joinGroup()`, `send()` ⟩ a partir das chamadas feitas nas linhas 10 ou 11, 13, 14 e 25. Uma especificação formal permite verificar se a sequência de chamadas à API feita pela aplicação cliente está ou não em conformidade com os cenários de uso previstos. Por exemplo, uma especificação baseada em autômatos, permite verificar a conformidade por meio da aceitação da sequência de chamadas pelo autômato correspondente à classe alvo. Idealmente, a linguagem do autômato seria formada por todos os cenários de utilização válidos da API.

1.2 Objetivos

Este trabalho propõe métodos que colaborem para automatização de parte do processo de especificação formal voltada para clientes de uma biblioteca de software, que consiste na extração de sequências válidas de chamadas às APIs disponibilizadas. Este processo é feito a partir dos elementos tipicamente presentes no processo de desenvolvimento da mesma, como por exemplo, o código-fonte. Com isto, espera-se que seja gerada uma especificação formal constituída por autômatos que permita que desenvolvedores que utilizam a biblioteca verifiquem se o código cliente escrito por eles, para utilizar a API disponível corresponde à uma sequência de chamadas aceita pelos autômatos que compõem a especificação.

De maneira similar a outros trabalhos de inferência do campo de mineração de

dados, uma característica importante da inferência de especificação formal é obtenção de maiores taxas de precisão, problema que Weimer e Necula (2005) e Engler et al. (2001) apontam como principais desafios dos mineradores de especificação modernos. Outra métrica importante é a revocação, que busca indicar o quão completa é o modelo inferido, conforme discussão no Capítulo 2.

1.2.1 Objetivos específicos

Tendo em vista o amplo espectro de objetivos para os trabalhos de mineração de especificação, os seguintes objetivos específicos foram definidos para nortear este trabalho:

- Desenvolvimento de uma metodologia que possibilite a geração de uma especificação voltada para apoiar os programadores de aplicações cliente a compreender mais facilmente a utilização correta da API disponibilizada por uma biblioteca de software;
- Implementação de um minerador de especificações formais que exija um conjunto de entrada de mais fácil obtenção, útil para contextos em que o código-fonte de aplicações cliente similares não esteja disponível e
- Investigação do efeito de certas propriedades dos artefatos utilizados no processo de inferência nos resultados obtidos pelo minerador de especificações construído.

1.2.2 Justificativa

Segundo Baier e Katoen (2008), vivemos em uma sociedade cuja dependência de sistemas baseados em hardware está em contínua expansão. O elevado uso de recursos computacionais também aumenta a exigência por sistemas mais robustos e confiáveis, que falhem menos, ainda que os requisitos sejam continuamente mais complexos.

Há situações em que as falhas de computação podem provocar desde grandes prejuízos financeiros até tragédias que custem vidas humanas. Mesmo para sistemas que não sejam de missão crítica, erros no funcionamento do software podem comprometer tanto a experiência do usuário quanto comprometer a produtividade daqueles que utilizam a aplicação. Técnicas que contribuam para a diminuição de erros em software podem auxiliar na redução deste tipo de problema em softwares.

Um dos meios de atingir este maior grau de qualidade é a representação matemática do sistema de modo que formalismos possam ser aplicados para se provar matematicamente que a conformidade com os requisitos definidos e que certas propriedades e comportamentos se encontram implementados no sistema.

Pela perspectiva de software, é possível perceber que métodos formais caracterizam valiosa ferramenta para garantir a qualidade durante todo o ciclo de desenvolvimento do sistema (WOODCOCK et al., 2009). Isto permite que pesquisadores se concentrem nas áreas mais críticas da aplicação em desenvolvimento, conforme exemplo relatado por Clarke e Wing (1996): um módulo do sistema de controle aéreo de Londres foi entregue para a agência de aviação do Reino Unido com uma taxa de falhas até dez vezes inferior àquela observada em aplicações do mesmo domínio desenvolvidas sem o uso de métodos formais.

Com isto, espera-se que seja gerada uma especificação formal que permita que desenvolvedores que utilizam a biblioteca verifiquem se o código cliente escrito por eles, para utilizar a API disponível, está correto ou não.

De maneira similar a outros trabalhos de inferência do campo de mineração de dados, uma característica importante da inferência de especificação formal é obtenção de maiores taxas de precisão, problema que Weimer e Necula (2005) e Engler et al. (2001) apontam como principais desafios dos mineradores de especificação modernos. Além disso, é importante que este processo exija pouca intervenção manual, de modo que a maior parte da especificação seja extraída automaticamente e a intervenção humana se faça necessária apenas em poucas situações.

Ademais, o uso de ferramentas que auxiliem os profissionais no processo de especificação se torna de grande valia para tornar o processo de definição da especificação mais eficiente. O emprego da mineração de dados permite o processo reverso de obtenção de partes da especificação, sem que o processo seja totalmente completado para que as etapas seguintes sejam realizadas. Apesar de nem sempre ser possível inferir uma especificação completa, as mesmas ainda assim são úteis, pois há técnicas de verificação formal aplicáveis a especificações parciais (BAIER; KATOEN, 2008). Desta maneira, é possível reconstruir uma especificação correspondente ao software implementado e a partir daí verificar se os comportamentos e propriedades do software atendem às expectativas de seus usuários.

Este trabalho busca se investigar se os testes unitários, que tipicamente são de tamanho reduzido e são voltados para testar componentes básicos de um software podem servir como boa entrada de um minerador de sequências de chamadas à API.

1.3 Contribuições

Os resultados desta dissertação consistem principalmente em um minerador de especificações formais e o repositório utilizado para a avaliação da precisão dos autômatos inferidos com base em oráculos. Estes dois principais produtos de trabalho são discutidos

nas próximas duas subseções.

É importante observar que o minerador de sequências de chamadas a APIs pode apoiar o desenvolvedor por meio de sugestões de chamadas de métodos que a partir da sequência de chamadas já feita, indica aquelas que contribuem para a formação de uma *string* reconhecida pelos autômatos produzidos pelo minerador.

O oráculo, por sua vez, serve para apoiar os processos de cálculo da precisão e revocação de diferentes mineradores de revocação, e por meio da hospedagem em um repositório público na Internet, práticas de Engenharia de Software, como a revisão de código a se incorporar no ramo principal de desenvolvimento permitem o refinamento iterativo de versões aprimoradas dos oráculos construídos pela comunidade.

1.3.1 Abordagem MUTE

Em contraste com as abordagens existentes na literatura, este trabalho se concentrou em utilizar os testes unitários como atores principais no processo de mineração da especificação formal de uma biblioteca de software voltado para clientes da mesma. Foi demonstrada a possibilidade de se produzir esta especificação tendo como entrada apenas apenas a biblioteca em si, e não suas aplicações clientes. Isto certamente contribui para a generalidade da documentação gerada e pode servir como primeira etapa para desenvolvedores que desejam obter automaticamente uma documentação para a API construída por eles.

Pradel e Gross (2012) mostraram que os testes unitários são valiosos para se detectar erros de utilização de uma API. No trabalho corrente, foi possível observar que os testes unitários podem contribuir fortemente para o processo de inferência de especificações formais. A própria natureza típica de organização destes artefatos - o foco em uma unidade do software, em rotinas geralmente pequenas - faz com que técnicas mais avançadas de *clusterização* não sejam necessárias.

Algoritmos voltados para a *clusterização* se encontram presentes em muitos trabalhos de mineração de especificações formais que utilizam uma abordagem dinâmica para analisar o código-fonte (LO, 2008). Uma vez que os *traces* contém registros de chamadas em sequência cronológica, não existem critérios muito claros para extração das partes relacionadas.

Os testes unitários por sua vez, possuem uma estrutura que permite explorar esta organização para se encontrar as partes relacionadas do software. Isto pôde ser percebido na abordagem MUTE, em que não se notou a necessidade de empregar algoritmos para se extrair agrupamentos nos dados utilizados como entrada.

1.3.2 Plataforma CORES

Conforme discutido por diversos autores (SOMMERVILLE, 2010; BAIER; KATOEN, 2008), a construção de especificação formal não é um processo tão simples quanto o de especificação de software tradicional, em que é empregada a prosa em linguagem natural para descrever as propriedades e comportamentos esperados de um software.

Para contornar as dificuldades de se construir e gerar as especificações formais, foi proposto um repositório público para que especificações formais de software sejam desenvolvidas colaborativamente e compartilhadas com um grupo maior de pesquisadores. Espera-se que com isto as dificuldades atuais encontradas para se obter especificações formais de referência para as pesquisas relacionadas ao campo de Métodos Formais sejam reduzidas.

1.4 Limitações

A estratégia de inferência de especificações proposta neste trabalho possui duas limitações: sensibilidade ao conjunto de entrada e poder de expressividade do formalismo utilizado para extração das sequências de chamadas aos métodos da API.

Conforme será observado nos experimentos conduzidos no Capítulo 4, há uma baixa disponibilidade de testes unitários junto às bibliotecas de software para guiarem o processo de aprendizado de autômatos. Esta limitação motiva o emprego de técnicas para a geração automática de testes unitários, que em alguns casos pode levar a bons resultados, conforme é discutido neste mesmo capítulo.

Outro limite de efetividade decorre do formalismo empregado para representar as especificações formais voltadas para clientes da biblioteca de software. Ao se optar pela utilização de autômatos finitos, é possível representar sequências de chamadas de métodos que respeitem restrições de ordenação, todavia este formalismo é incapaz de representações mais poderosas, como indicação dos parâmetros de métodos e representação de regras sensíveis a dados que são armazenados persistentemente.

Com isto, as especificações formais produzidas têm seu escopo delimitado a bibliotecas de software, que em geral não se utilizam de bancos de dados para persistir informações e implementar restrições de domínio e de dados, como é o caso com sistemas de informação tradicionais.

2 REFERENCIAL TEÓRICO

Este trabalho investiga a aplicação da mineração de dados em problemas do campo de pesquisa de Métodos Formais. Para fomentar a discussão e esclarecer o vocabulário utilizado adiante, a Seção 2.1 apresenta os conceitos relativos à Especificação Formal, empregados neste trabalho. Em seguida, a Seção 2.2 apresenta as principais noções presentes em pesquisas voltadas à inferência de especificações formais, além de apresentar definições empregadas nos experimentos conduzidos nesta dissertação.

Ao final, a Seção 2.3, descreve como diferentes autores empregam os testes unitários para extração de outros tipos de informações, de modo complementar à investigação conduzida nesta pesquisa. Ao final, são discutidas diferentes estratégias de geração de testes unitários, técnica empregada, conforme discussão no Capítulo 3, para aumentar o volume de dados disponíveis para o processo de mineração das especificações formais.

2.1 Métodos Formais e Especificação Formal

Sommerville (2010) discute vários aspectos da qualidade de software, dentre eles o predomínio dos processos ligados à manutenção e melhoria da garantia da qualidade como principais fatores para elevação do custo de desenvolvimento de um software. Este mesmo autor descreve a especificação formal como um formato de especificação de sistema mais preciso, com rigor matemático embutido. Este tipo de artefato se faz presente em um contexto mais amplo, tipicamente em um Processo de Desenvolvimento Formal de Software, que faz uso de diferentes Métodos Formais ao longo do ciclo de vida.

Métodos Formais são definidos por Clarke e Wing (1996) como um conjunto de linguagens, técnicas e ferramentas com base matemática para especificar, projetar e verificar sistemas. Não obstante o fato de que o uso de métodos formais não garanta totalmente a correção de um sistema, tais técnicas podem ser muito úteis para melhorar o entendimento do sistema e eliminar as ambiguidades do mesmo.

Estes mesmos autores definem a especificação como o processo de descrição de um sistema e suas propriedades desejadas, sendo que a especificação formal possui uma sintaxe matemática para expressar tais características do sistema. Estas propriedades podem ser de diversos tipos: comportamento funcional, temporal, atributos de desempenho e estrutura interna.

A partir da especificação de um sistema, o mesmo é projetado, implementado e

verificado. Usualmente, o desenho de um software se baseia em modelos, que buscam representar o problema e sua respectiva solução com uma certa sintaxe. Baier e Katoen (2008) explicam que a solução de um software constitui um modelo que pode ser verificado por meio de formalismos matemáticos para desambiguização e conformidade com as propriedades desejadas definidas na especificação. Os autores explica que a verificação pode se valer de técnicas se dividem em exploração exaustiva (verificação de modelos), simulação de cenários e o teste com o produto real. Com o avanço dos computadores desde o final do século XX e o uso inteligente de estruturas de dados e algoritmos, é possível que até 10^{476} estados sejam verificados em certos tipos de problemas (BAIER; KATOEN, 2008).

2.2 Mineração de Especificação Formal

Especificações formais permitem a verificação precisa de sistemas computacionais sem as ambiguidades inerentes à linguagem natural. Contudo, a geração requer sólidos conhecimentos matemáticos para formular adequadamente as expressões que formam a especificação formal. Isto pode ser observado no trabalho de Chen, Wagner e Dean (2002), no qual é feita uma análise formal de uma ferramenta do sistema operacional Unix manualmente, tendo como principal objetivo a localização de vulnerabilidades de segurança. Tal trabalho revela um grande número de imprecisões e até mesmo problemas de segurança na plataforma, mas tal processo de especificação demandou um esforço humano não negligenciável.

Isto confirma a afirmação de Dallmeier et al. (2012), que diz que tanto a escrita da especificação formal quanto a verificação da especificação constituem um processo que torna proibitiva sua aplicação em métodos de desenvolvimento mais avançados ou em sistemas de maior escala. Conforme explicado por Goues e Weimer (2012), tais dificuldades motivam o emprego de técnicas automáticas para geração da especificação formal.

A expressão *mineração de especificação* foi cunhada por Ammons, Bodík e Larus (2002) para descrever os trabalhos que envolvem o emprego de técnicas de mineração de dados para inferência de especificação formal a partir de artefatos de software. Tais técnicas podem fazer análises de dois tipos: estática e dinâmica.

A análise estática se caracteriza por utilizar elementos como o código-fonte, que são empregados sem necessidade da execução do software e têm como principal fragilidade a geração um número muito elevado de cenários, dentre os quais muitos não são observados na prática.

A análise dinâmica consiste na análise de artefatos ligados à execução do software, como *logs* de execução ou pilhas de execução no momento da ocorrência de erros, dentre

outros (GOUES; WEIMER, 2012; DALLMEIER et al., 2012). A principal limitação deste tipo de análise pode decorrer da cobertura dos registros coletados, que podem ter uma abrangência pequena em relação a todo o software analisado. Vale observar que as limitações de cada um destes dois tipos de análise podem ser reduzidas com uma abordagem híbrida, que combina as duas técnicas no processo de inferência da especificação formal.

Geralmente, estas especificações formais não são completas, mas de acordo com Clarke e Wing (1996), há métodos de verificação formal que podem ser aplicados mesmo quando se possui apenas uma especificação formal parcial. Isto possibilita a mineração de especificação formal dos principais componentes de um software, de modo que erros são encontrados mais rapidamente nas partes mais importantes de software. A especificação formal produzida pode ainda ser baseada em diferentes tipos de formalismos matemáticos (LAMSWEERDE, 2000; ALMEIDA et al., 2011; WOODCOCK et al., 2009). Os principais tipos são baseados em linguagens de domínio específico, equações algébricas ou representações baseadas em modelos.

2.2.1 Aplicações de Especificações Formais

Uma especificação de software não serve exclusivamente para a verificação de software. Segundo Alagar e Periyasamy (2011), as especificações são utilizadas durante todo o processo de software e em diferentes momentos podem exercer funções como o aprendizado do domínio da aplicação e controle do ambiente de execução do mesmo. De acordo com Goues e Weimer (2009), especificações formais já foram utilizadas para documentar sistemas, otimizar compiladores, localizar defeitos e melhorar a compreensão do código-fonte.

Por exemplo, a localização de defeitos é objetivo principal nos trabalhos de Pradel e Gross (2012), Dallmeier et al. (2012), Monperrus, Bruch e Mezini (2010). Pradel e Gross (2012) apresentaram um minerador de especificações formais que não identificou falso-positivos para os defeitos reportados em aplicações clientes de APIs do JDK.

As especificações que contemplam cenários voltados para o entendimento de utilização da biblioteca também são de interesse em outras pesquisas. Em (BRUCH; SCHäFER; MEZINI, 2006; BRUCH; MONPERRUS; MEZINI, 2009; PRADEL; BICHSEL; GROSS, 2010), aplicações clientes de diferentes bibliotecas são examinadas a fim de se aprender padrões frequentes que possam ser utilizados para recomendar padrões de utilização da API.

2.2.2 Formalismos empregados

Uma especificação formal se diferencia de uma especificação de software típica pelo emprego de modelos matemáticos para uma representação precisa das propriedades e comportamentos do software. Não obstante o rigor matemático, existe uma grande variedade de diferentes modelos matemáticos que podem ser empregados para representar os formalismos da especificação formal construída.

Sistemas de equações e notações algébricas são empregados por alguns autores, como Guttag e Horning (1978) que representam tipos abstratos de dados com este formalismo, alvo do minerador de especificação proposto por Henkel e Diwan (2003).

A representação de propriedades em formatos como invariantes, pré e pós-condições de classes do sistema é empregado nos trabalhos de Ernst et al. (1999), Ramanathan, Grama e Jagannathan (2007), Flanagan e Leino (2001). Tal tipo de representação tornou-se popular especialmente após o trabalho de Meyer (1992), que propôs a técnica de *desenho-por-contrato*, que envolve a definição de *contratos* dos componentes de um software, de maneira similar ao que se observa em especificações de software.

Não obstante os formalismos discutidos anteriormente, o mais popular deles atualmente é aquele baseado em modelos que utilizam autômatos de estados finitos (FSA) na representação de unidades do software. Há dois tipos de autômatos principais empregados na literatura: máquinas de dois estados ou autômatos de vários estados (GOUES; WEIMER, 2009; LO, 2008; ROBILLARD et al., 2013).

Enquanto os autômatos de dois estados são mais simples, eles não cobrem todos os cenários observados, pois seu foco é a representação de padrões de *abertura-e-fechamento*, que envolvem operações que ocorrem aos pares, como por exemplo, a abertura e fechamento de um arquivo ou liberação da memória alocada para a construção de um certo tipo de dados. Por outro lado, as máquinas de vários estados descrevem uma sequência maior de chamadas e, em geral, têm como objetivo descrever um cenário que envolva um grande número de chamadas.

Há autores que utilizam um tipo específico de FSA para representar os sistemas. É o caso dos FSAs que incorporam probabilidades nas transições entre dois estados de um autômato, a fim de se refletir a probabilidade de execução daquela transição, com a restrição adicional de que as probabilidades de todas transições que partem de um mesmo estado devem totalizar 100%. Este formalismo, chamado de autômatos probabilísticos de estados finitos (PFSA) está presente no trabalho seminal de Ammons, Bodík e Larus (2002) e também foi empregado por Lo (2008).

Devido à equivalência do poder de expressividade entre FSAs determinísticos e não-determinísticos, os autores os utilizam livremente, sem se preocupar em distinguir

claramente o tipo de autômato utilizado. Todavia, nesta pesquisa, será utilizado o formalismo de autômato de estados finitos determinístico (DFA) para representar cada unidade do sistema alvo. Isto se deve à maior facilidade de depuração e percepção visual ao se trabalhar manualmente com a especificação, de forma a superar as dificuldades discutidas por Ammons et al. (2003) para que um humano entenda as especificações ao examinar manualmente um grande número de possibilidades de computação.

2.2.3 Representação de classes orientadas-por-objeto com DFAs

De maneira similar a outros trabalhos, este trabalho utiliza como unidade básica do software, uma classe, sendo que cada uma delas será representada por um DFA, cuja linguagem corresponde aos cenários válidos de utilização de uma instância da classe. O DFA é representado como uma quíntupla da forma $(\Sigma, Q, q_0, \delta, F)$, em que os elementos são definidos como:

- Σ é o alfabeto cujos elementos denotam os métodos públicos da classe correspondente;
- Q é o conjunto de estados deste DFA;
- A função de transição $\delta\colon Q \times \Sigma$ indica o próximo estado da computação após a leitura de uma chamada de método correspondente a um elemento de Σ;
- q_0 corresponde ao estado inicial;
- $F \subseteq Q$ é o conjuno de estados de aceitação do autômato.

Para se verificar que uma sequência de chamadas é válida, é criada uma cadeia de caracteres correspondente à concatenação dos elementos da sequência. A partir daí, a sequência é considerada válida se, e apenas se, o DFA termina a computação desta *string* em um estado de aceitação q_f pertencente ao conjunto F.

O *framework* proposto por Pradel, Bichsel e Gross (2010) discute três diferentes abordagens para representar uma sequência de chamadas: centrada em métodos, centrada em objetos e orientada por uma única instância. A estratégia de construção do DFA apresentada acima corresponde à abordagem orientada por uma única instância, sendo este último o formalismo que apresenta os melhores resultados de precisão, sendo que tal estratégia corresponde ao estado-da-arte para mineradores de especificação.

2.2.4 Inferência de Expressões Regulares

Apesar da abordagem MUTE ser proposta de maneira agnóstica quanto ao tipo de análise de código empregada, os experimentos conduzidos nesta dissertação se valerão

de uma abordagem que utiliza *traces* de execução dos testes unitários para aprendizado de sequências válidas, conforme será discutido nos experimentos apresentados no Capítulo 4.

Uma das questões que precisam ser abordadas com a utilização de *traces* de execução é a presença de laços de repetição, o que cria múltiplas repetições de chamadas internas ao laço. Isto pode ser observado no exemplo do Código 2, ao ser invocado com os argumentos `n`=2 e `array` = [2,3], cujo trace correspondente é a sequência $\langle A, B, C, B, C, D, A, B, C, B, C, D \rangle$.

Código 2 – Exemplo de Código que gera repetições no *trace*.

```
1  void APIClient_ABCD(int n,int[] array){
2      int k=0;
3      API.A();
4      for (int i=0;i<n;i++){
5          k++;
6          do{
7              API.B();
8              API.C();
9          } while(k<array[i]);
10         API.D();
11     }
12 }
```

Fonte: Adaptado de (LO, 2008).

Conforme explicado por Lo (2008), este *trace* é mais útil se representado como uma expressão regular, que indica a possibilidade de repetições de certas chamadas sem que os limites específicos de cada aplicação estejam presentes. Isto pode ser feito com o algoritmo de inferência de gramática regular proposto por Nevill-Manning e Witten (1997). Ao se utilizar o algoritmo SEQUITUR, o *trace* discutido no parágrafo anterior será representado como a expressão regular $(A(BC) + D)+$.

2.2.5 Avaliação da precisão e revocação

A mineração de especificações formais de software utiliza métricas do campo de mineração de dados para analisar a qualidade do modelo inferido por um algoritmo. Por conta disso, as medidas mais populares são precisão e revocação. Segundo Han e Kamber (2006), a precisão é uma medida de exatidão definida pela relação de registros positivos encontrados dentro do universo dos registros inferidos. A revocação é uma avaliação da abrangência dos resultados obtidos, sendo definida pela relação de elementos existentes que foram recuperados.

No contexto dos trabalhos de mineração de especificações formais, a métrica de precisão é a mais frequentemente utilizada para avaliar os trabalhos (ROBILLARD et al., 2013). A avaliação da precisão pode ser conduzida de maneira manual, ou então, com base em uma especificação formal de referência em um *framework*, de maneira semelhante à proposta por Pradel, Bichsel e Gross (2010). A precisão é dada pela expressão:

$$Precisão = \frac{número \quad de \quad registros \quad corretamente \quad inferidos}{total \quad de \quad registros \quad inferidos}$$

A revocação, correspondente à razão entre números de registros corretamente inferidos e o total de registros existentes, não é tão frequentemente reportada nas pesquisas de mineração de especificações de software. Ela pode ser calculada pela seguinte fórmula:

$$Revocação = \frac{número \quad de \quad registros \quad corretamente \quad inferidos}{total \quad de \quad registros \quad existentes}$$

Apesar de ser uma medida útil em diversos contextos de mineração de dados, frequentemente, esta informação é muita vezes omitidas dos trabalhos relacionados (ROBILLARD et al., 2013).

2.3 Testes unitários

Testar unidades de um sistema é uma tarefa de engenharia de software que objetiva verificar a conformidade da implementação de um código-fonte com a especificação correspondente do software (SOMMERVILLE, 2010). Para Tillmann e Schulte (2005), os testes unitários são componentes chave de um processo de desenvolvimento de software. Estes autores afirmam que os testes de sistemas orientados por objeto têm por objetivo verificar se o comportamento apresentado por chamadas feitas à métodos de uma *class-under-test* (CUT) está correto.

Os testes unitários são implementados por programadores que conhecem os detalhes internos de uma aplicação, que os utilizam para verificar o comportamento correto de uma sequência de chamadas feitas para uma das unidades do software. No contexto de aplicações desenvolvidas em linguagens orientadas por objetos, uma unidade corresponde à uma classe do software a ser testado.

Apesar de não serem estritamente necessários para desenvolver e tornar um software disponível para seu público alvo, os testes unitários são úteis para evitar defeitos de regressão de maneira automática, e refletem uma perspectiva, pela ótica de seus programadores, de como o sistema deve se comportar.

2.3.1 Mineração de Testes Unitários

As características dos testes unitários explicam porque muitos trabalhos anteriores concentram esforços em aplicar técnicas de mineração de dados em suítes de testes unitários. Em seu trabalho, Santhiar, Pandita e Kanade (2013) têm como objetivo um melhor entendimento de um conjunto de bibliotecas matemáticas, ao passo que melhorar a produtividade de desenvolvedores é a meta no trabalho de Ghafari et al. (2014). Frequentemente, as suítes de testes unitários são utilizadas para gerar rastros de execução de aplicações clientes no processo de mineração de especificação formal, como ocorre nos trabalhos de Gabel e Su (2008), Lee, Chen e Roşu (2011), Pradel e Gross (2009).

Ainda que os trabalhos de Pradel e Gross (2012), Dallmeier et al. (2012) façam uso de casos de teste no processo de mineração de especificação formal, os mesmos são utilizados para complementar a entrada do programa. Em (PRADEL; GROSS, 2012), a entrada consiste no código-fonte de aplicações clientes de uma API, e as chamadas feitas pelas aplicações clientes são utilizadas para guiar a geração de testes unitários que serão posteriormente utilizados para se gerar uma especificação formal. Por sua vez, Dallmeier et al. (2012) utilizam inicialmente o código-fonte do programa alvo. Além disto, ambos os trabalhos buscam gerar uma especificação formal para localização de defeitos na utilização da API por aplicações cliente.

A especificação formal alvo do minerador proposto neste trabalho é voltada para a verificação das sequências de chamadas aos métodos da API, com base no formalismo de autômatos, mesmo que estas sequências contenham métodos simples como `toString()` e `getters`, que geralmente não fazem parte dos exemplos principais de utilização da API.

2.3.2 Geração aleatória de testes unitários

Conforme discutido no início desta Seção, os testes unitários não são necessários para que uma aplicação seja implementada e possa ser executada. Por conta disto, os mesmos nem sempre estão disponíveis em grande quantidade juntamente às bibliotecas de software, apesar de sua utilidade no processo de verificação de software, especialmente por permitirem a execução automática de testes de regressão em suítes de testes modernas.

Uma nova linha de pesquisa que surge para apoiar equipes de desenvolvimento de software nos esforços de verificação de software é a geração automática de diferentes tipos de teste, inclusive testes unitários. Estes trabalhos, além de ajudar a equipe de desenvolvimento, são de grande valia para outras técnicas que dependam de um maior número de testes para ter um bom desempenho.

Este cenário se aplica a este trabalho, pois o aprendizado da API é favorecido positivamente por um número maior de testes unitários dos quais possam ser extraídos

exemplos de como se utilizar uma biblioteca de software. As ferramentas de geração de teste, portanto, são empregadas para apoiar na criação de testes e contornar eventuais problemas com a falta de testes unitários disponíveis para a execução da abordagem proposta.

Segundo o *survey* Anand et al. (2013), há diferentes abordagens existentes para a geração automática dos testes unitários, as quais variam entre geração aleatória, métodos baseados em busca e a geração de testes baseadas em modelos. Estas abordagens objetivam apoiar a equipe de desenvolvimento a lidar com o trabalhoso desafio de criação de casos de teste para o software em construção por meio da geração automática dos mesmos.

A estratégia de geração randômica consiste em selecionar aleatoriamente de um conjunto de métodos disponíveis, um dos mesmos para compor um teste que está sendo construído pelo algoritmo. Devido à sua simplicidade, esta estratégia permite a geração de um grande número de testes em um curto intervalo de tempo.

A ferramenta Randoop (PACHECO; ERNST, 2007) é um dos principais representantes desta categoria. A mesma embute um mecanismo de retro-alimentação para guiar o processo de geração de testes. Este mecanismo de retro-alimentação se caracteriza por controlar a sequência de chamadas feitas em um teste unitário por meio da análise do resultado da execução de um número menor de chamadas feitas, de maneira similar ao comportamento do algoritmo *APriori* apresentado em (AGRAWAL; SRIKANT, 1994). Uma série de chamadas com n operações deve executar com sucesso para se derivar novos testes com $n+1$ operações. Devido ao espaço de busca fatorial neste tipo de estratégia, os critérios de parada adotados geralmente variam entre tempo máximo de execução ou número de testes unitários produzidos.

3 METODOLOGIA

O desenvolvimento de mineradores de especificações formais é composto de dois desafios importantes: a construção do mecanismo de inferência proposto e a validação dos resultados obtidos a partir de uma base de dados que funciona como um oráculo para determinar a qualidade dos resultados obtidos.

A estratégia de inferência proposta neste trabalho se baseia na hipótese de que os testes unitários podem ser utilizados para o aprendizado de especificações formais de uma API. Para a avaliação desta hipótese, foi desenvolvido um arcabouço que coloca os testes unitários como atores principais do processo de mineração de especificações. Este *framework* foi designado como MUTE, acrônimo para a expressão em inglês **M**ining **U**nit **TE**sts.

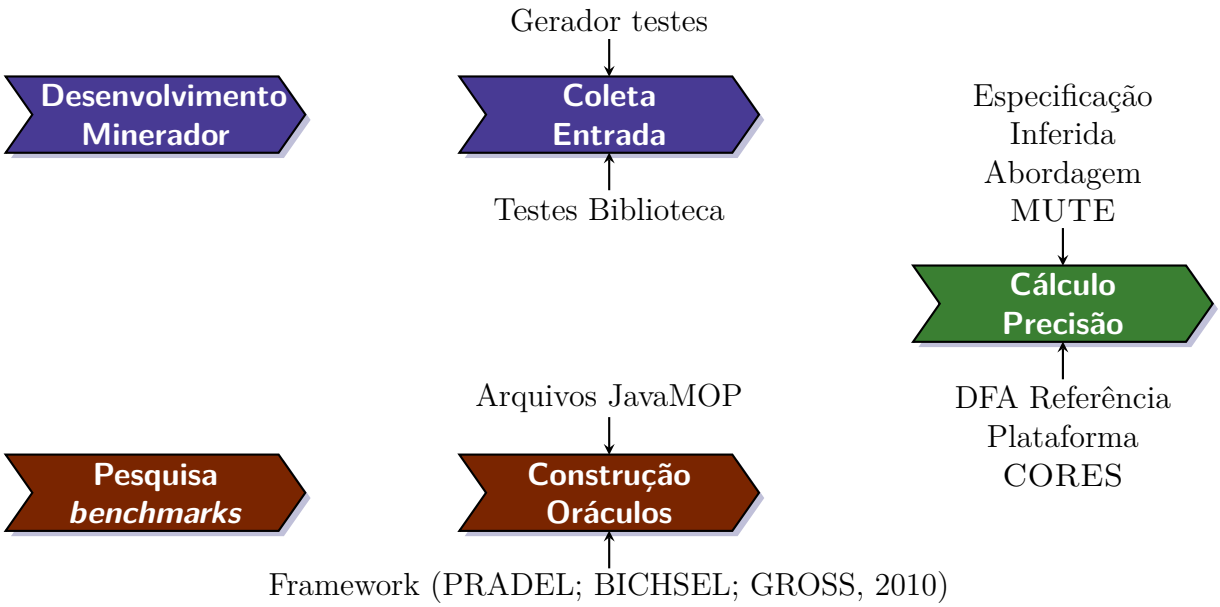
No que concerne a avaliação da precisão de mineradores de especificações formais, são relevantes as observações em (ROBILLARD et al., 2013): há grande variação na metodologia de avaliação de resultados e muitas vezes a mesma é conduzida manualmente. Isto dificulta a reprodução de resultados e ameaça a generalidade dos mesmos. Para contornar tais problemas, esta pesquisa envolveu também o desenvolvimento de especificações formais de referência para que a precisão e revocação das especificações inferidas seja avaliada de forma automática em vez de requerer que os pesquisadores validem manualmente.

A Figura 1 apresenta a metodologia deste trabalho, que envolve o desenvolvimento do minerador de especificação formal MUTE e avaliação de sua precisão com uma referência obtida na plataforma CORES. O desempenho do minerador MUTE foi avaliado tanto com testes unitários originalmente distribuídos juntamente ao código-fonte da biblioteca quanto testes unitários gerados automaticamente.

A especificação formal de referência foi construída a partir da combinação dos resultados de Pradel, Bichsel e Gross (2010) e Lee et al. (2012), o que deu origem à plataforma CORES, discutida na subseção 3.2. O objetivo desta plataforma é superar as dificuldades de geração de especificações formais de referência por meio do trabalho colaborativo entre vários pesquisadores e desenvolvedores.

Nas próximas três seções, serão discutidos os detalhes específicos das etapas envolvidas no desenvolvimento de um minerador de especificação baseado no *framework* MUTE, além da construção da especificação formal utilizada como referência para avaliar a acurácia de uma especificação formal inferida.

Figura 1 – Metodologia empregada nesta dissertação.



Fonte: Elaborado pelo autor.

3.1 A abordagem MUTE

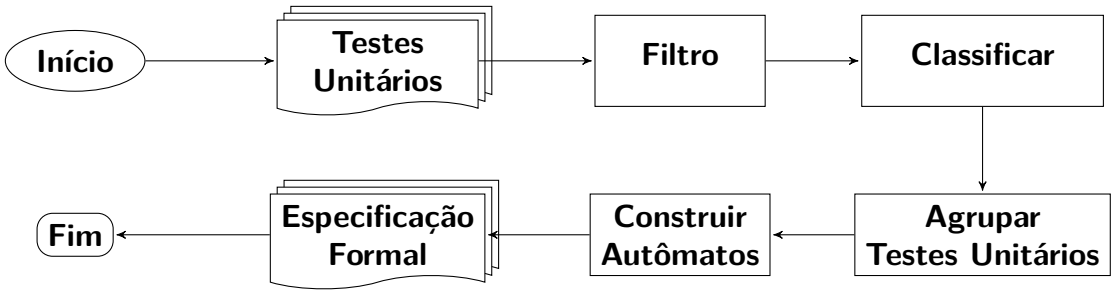
A partir da investigação dos trabalhos relacionados discutidos no Capítulo 2 e também dos resultados presentes nos trabalhos de Pradel e Gross (2012), Dallmeier et al. (2012), optou-se por uma investigação mais profunda sobre a possibilidade de se utilizar unicamente os testes unitários presentes no código-fonte de uma biblioteca de software para se obter sua especificação formal, útil para contextos em que o código-fonte de aplicações cliente similares à utilizada não estejam disponível. Portanto, a investigação da abordagem MUTE tem como objetivo analisar a utilização de testes unitários como insumo primário ao processo de aprendizado dos autômatos que constituem uma especificação formal de uma biblioteca de software voltada para o entendimento da API desta biblioteca.

O formalismo escolhido para representação dos DFAs na especificação formal é aquele baseado em autômatos voltados para uma instância da classe, conforme discussão na Subseção 2.2.3. Isto se deve aos melhores resultados obtidos por Pradel, Bichsel e Gross (2010), dentre as três abordagens apresentadas. Isto que torna mais fácil a compreensão pelos programadores que procuram entender qual o próximo método a se chamar para uma instância já construída anteriormente em seus programas.

A Figura 2 mostra a arquitetura do *framework MUTE*, que é executada uma vez para cada classe alvo e ao final produz um DFA que é adicionado à especificação formal constituída de todos os DFAs da biblioteca de Software. A linguagem de cada DFA

corresponde aos cenários válidos de uma instância a partir da instanciação de uma das classes da biblioteca.

Figura 2 – Componentes da arquitetura do *framework* MUTE.



Fonte: Elaborado pelo autor.

A Figura 2 indica a organização de componentes válida para a geração da especificação de cada uma das classes de uma biblioteca. Este fluxo é executado uma vez para cada das classes para as quais se deseja obter um DFA para compor a especificação formal da API.

O *framework* MUTE recebe como entrada todos os testes unitários disponíveis do software em estudo. A metodologia proposta não faz restrições quanto ao tipo de análise feita sob os testes unitários. Portanto, pode-se optar tanto por fazer análise estática a partir do código-fonte dos testes unitários ou se valer de uma análise dinâmica que reconhece registros de execução dos testes unitários. É possível ainda que seja empregada uma abordagem híbrida, que combine os dois tipos de análise. É importante apenas fornecer testes unitários como entrada para o método e definir uma estratégia de coleta das sequências de chamadas contidas nos mesmos.

3.1.1 Filtro

O *pipeline* de execução da abordagem MUTE inicia seu processamento efetivo da entrada com o componente de filtro, que possui o papel de reduzir o ruído nos artefatos processados. O ruído é caracterizado principalmente por testes unitários que não contribuem para o aprendizado de sequências válidas de chamadas à API. Para cada uma das classes, todos os testes unitários são inspecionados para escolher aqueles que podem contribuir positivamente para o aprendizado do autômato correspondente.

A primeira restrição imposta pelo filtro é a inicialização da unidade com elementos padrão da linguagem, como o operador `new` em *Java*, seguida de pelo menos uma chamada a um método público da classe. A utilização incidental de uma classe em um teste é evitada por meio da restrição que exige a invocação de pelo menos um método público da mesma naquele teste. Um exemplo de presença fortuita de uma classe em um teste unitário é, por

exemplo, a presença de instâncias da classe *java.util.Date*, muitas vezes criadas apenas para a obtenção da data e hora corrente.

Além desta restrição, não são considerados testes que possuem dependência para pacotes externos ao da unidade, a não ser pelo pacote padrão da linguagem de programação, de maneira similar ao trabalho de Goues e Weimer (2009). Desta maneira, busca-se cenários mais específicos, voltados para os principais casos de uso de cada classe, sem envolver padrões de cooperação de classes mais complexos.

Para um software construído na linguagem em *Java*, isto implica em considerar apenas os testes do pacote da classe e classes do pacote *java.lang*, que contém tipos básicos como *String* ou *Integer*.

De forma sucinta, o componente de filtro, para cada uma das classes do software, manterá o vínculo da unidade com os testes que atendem aos seguintes critérios simultaneamente:

- Chamada a pelo menos um método público da unidade instanciada que é alvo do teste em análise pelo filtro;
- Invocação de operações de classes dentro do mesmo pacote da unidade testada ou do pacote com os tipos básicos da linguagem alvo. No caso da linguagem Java, o pacote `java.lang`.

3.1.2 Classificação

Os testes unitários são valiosos não apenas por ajudar com asserções sobre o comportamento correto das operações de uma *CUT*. Eles também são utilizados para verificar a resiliência de uma classe contra parâmetros incorretos, chamadas fora de ordem e outras violações que poderiam comprometer a consistência de seu estado interno. Para proteger as unidades contra estes tipos de problemas, desenvolvedores se valem de verificações acionadas pelos métodos públicos para levantar exceções em caso de problemas que previnam a execução bem-sucedida do método.

Um exemplo deste tipo de teste unitário é mostrado no Código 3, no qual o critério de aceitação é a sinalização de um erro de *timeout* pela classe `DatagramSocket`. Após a criação de uma instância desta classe, a mesma é configurada para bloquear nas operações de recebimento de dados por no máximo 2 milissegundos por meio da chamada ao método `setSoTimeout()`. No exemplo mostrado, o recebimento de dados bloqueia o processo em execução ao invocar a operação `receive()` da classe durante o intervalo configurado previamente. Caso a classe sinalize o erro por meio da exceção `SocketTimeoutException`, será considerado que a classe passou com sucesso por este teste.

Código 3 – Teste unitário `DatagramTimeout`, para testar o lançamento de exceção de temporização pela classe `DatagramSocket`.

```
1  public class DatagramTimeout {
2
3      public static ServerSocket sock;
4
5      public static void main(String[] args) throws Exception {
6          boolean success = false;
7          try {
8              DatagramSocket sock;
9              DatagramPacket p;
10             byte[] buffer = new byte[50];
11             p = new DatagramPacket(buffer, buffer.length);
12             sock = new DatagramSocket(2333);
13             sock.setSoTimeout(2);
14             sock.receive(p);
15         } catch (SocketTimeoutException e) {
16             success = true;
17         }
18         if (!success)
19             throw new RuntimeException("Socket timeout failure.");
20     }
21 }
```

Fonte: Adptado do código-fonte do Open JDK 6(Oracle Corporation, 2011)

É importante observar que para fins de geração de uma especificação formal para clientes de uma API, o tipo de teste descrito anteriormente não é útil, pois a sequência de chamadas levaria à ocorrência de exceções em uma aplicação cliente. Apesar da importância para auxiliar os desenvolvedores de uma biblioteca de software a evitar defeitos de regressão, os mesmo não constituem bons exemplos de como consumir a API.

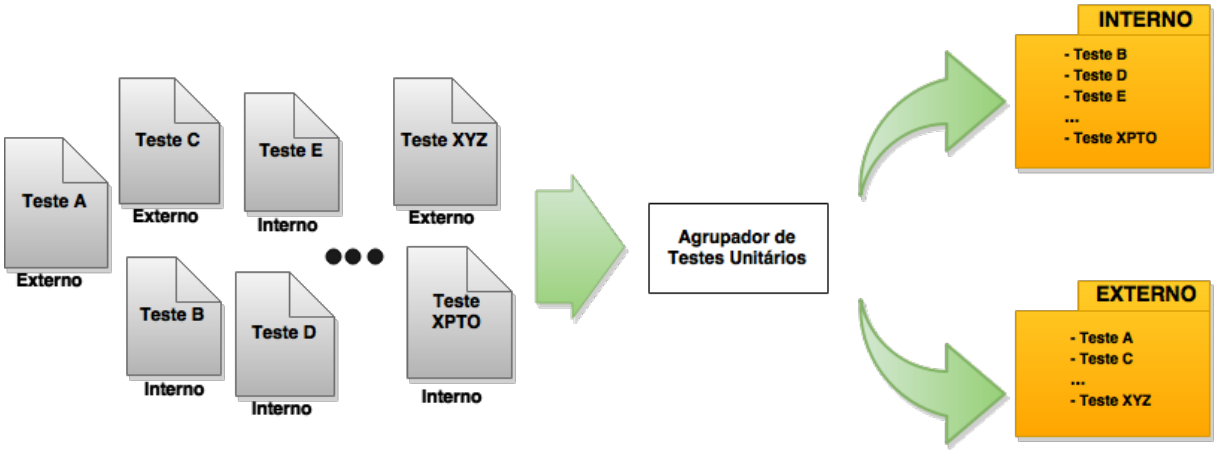
Portanto, o componente *Classificador* busca verificar a intenção original daquele teste: preservação do estado interno da classe ou verificação das sequências de chamadas públicas da classe. Com isto, cada teste unitário recebe um rótulo - *interno* ou *externo*. O rótulo *interno* corresponde aos testes unitários cujo critério de sucesso é ocorrência de uma exceção e o rótulo *externo* é atribuído aos testes unitários que não apresentam esta característica.

3.1.3 Agrupamento de Testes Unitários

Tendo em vista o objetivo de se definir um *framework* para bibliotecas de software em geral, não é possível assumir uma estratégia específica de organização dos arquivos

dos testes unitários. Entretanto, os dois componentes que participaram anteriormente no processamento fizeram a identificação das unidades e seus respectivos testes unitários aplicáveis. Com isto, é possível criar um agrupador por unidade de software para que posteriormente sejam examinados diferentes cenários de utilização válidos para o agrupamento criado.

Figura 3 – Funcionamento do componente criador de agrupamentos dos testes unitários.



Fonte: Elaborado pelo autor.

Conforme ilustrado pela Figura 3, este componente cria um agrupamento por rótulo definido pelo componente *Classificador* para que cada *cluster* seja processado independentemente mais à frente no *pipeline*. Em seguida, este componente organiza todos os testes que podem ser utilizados para se extrair diferentes exemplos da utilização da API em uma única pasta correspondente aos testes unitários classificados com o rótulo *externo*. Ao final, os testes que receberam o rótulo *interno* são descartados por não serem úteis para o processo de aprendizado da API.

3.1.4 Construção de Autômatos

O componente *Construtor de Autômatos* é responsável por construir um DFA para cada agrupamento criado anteriormente. Neste agrupamento, cada teste unitário contém uma sequência de k chamadas $S =< c_1, c_2, ..., c_k >$ onde cada c_i: $1 \leq i \leq k$ corresponde a uma operação da unidade que deu origem ao agrupamento.

Uma vez que a abordagem MUTE é agnóstica quanto a estratégia de análise de artefatos, é importante observar que em uma implementação do método MUTE, este é o componente mais afetado pela escolha da estratégia de análise de código.

Por um lado, a análise estática de código precisa lidar com problemas como análise de *alias* e controle de fluxo inter-procedimentos (SRIDHARAN et al., 2013), ao passo

que a análise dinâmica devem idealmente representar os registros de execução sem levar em consideração diferentes valores para laços *for* e *while*, cujo número de repetições é específico por aplicação (LO, 2008).

Ao final, cada agrupamento com rótulo externo contém um multi-conjunto com n sequências de chamadas. Para cada classe C do software, existe um multi-conjunto $M_c = \{S_1, S_2, ..., S_n\}$ que é transformado em um DFA. Para isto, cada sequência de chamadas $S_j \in M_c$ é convertida em uma expressão regular R_j (NEVILL-MANNING; WITTEN, 1997), conforme discutido na Subseção 2.2.4. Em seguida, o DFA A_c, correspondente à classe que originou o *cluster*, é construído ao se calcular o DFA que reconhece a linguagem $L_c = \bigcup_{R_j} \forall R_j \in M_c \wedge 1 \leq j \leq n$. Esta operação pode ser feita com bibliotecas como aquela escrita por Møller (2010), que utiliza a minimização para gerar o DFA correspondente.

Para exemplificar, pode-se pensar em uma classe C_i, que possui os métodos públicos *a,b,c,d,d,e*. Para ela, foram obtidas as sequências $S_1 = \langle a, b, c, d \rangle$ e $S_2 = \langle b, d, e \rangle$. Ao final, o DFA desta classe será dado por $R_1 \bigcup R_2$, em que $R_1 = abcd$ e $R_2 = bde$.

3.1.5 Coleta de testes unitários

Os testes unitários que são disponibilizados juntamente ao código-fonte são os candidatos naturais para servir como entrada para uma implementação da abordagem MUTE. Todavia, haverá situações em que eventualmente nenhum teste unitário dentre aqueles distribuídos junto à biblioteca estará disponível após a execução de todo o processo proposto.

Uma das opções para contornar a escassez de testes unitários é a geração automática de testes unitários para a biblioteca. Ainda que estes testes unitários não representem a perspectiva dos desenvolvedores que programaram o software, há diferentes abordagens que permitem a geração automática de testes em grande número (ANAND et al., 2013). Um volume maior de testes unitários auxilia a equipe de desenvolvimento a obter melhores taxas de cobertura de testes e, no contexto deste trabalho, servem para gerar um maior número de exemplos para o aprendizado de DFAs que descrevem as classes da biblioteca de software.

3.2 CORES: Uma plataforma colaborativa de especificações de referência

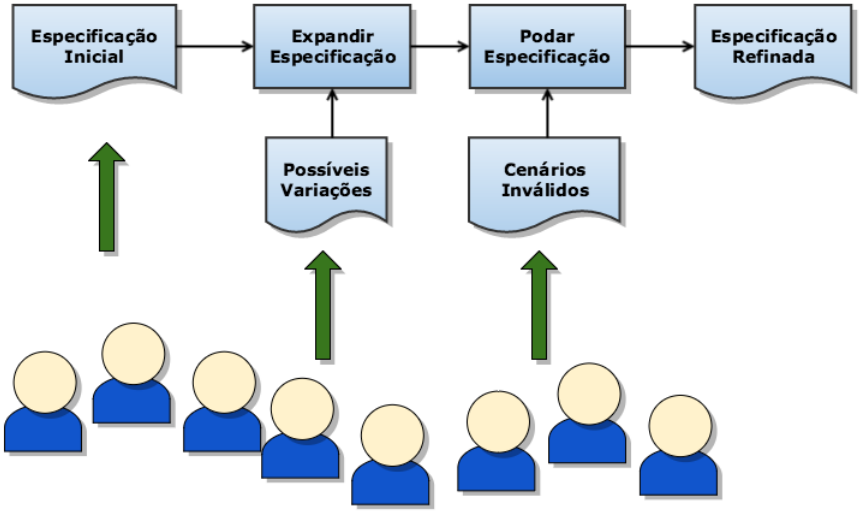
Muitos campos de pesquisa relacionados à inferência de modelos se valem de um oráculo para determinar a acurácia dos resultados, como é o caso, por exemplo, das áreas de segmentação de imagens, recuperação de informação ou até testes de software. Frequentemente, este oráculo precisa ser construído manualmente por especialistas humanos

para se estabelecer um *benchmark* que serve de referência de avaliação do desempenho. Especificamente para o campo de mineração de especificação formal de software, existe ainda a necessidade de se incorporar rigor matemático nos modelos produzidos.

Estes desafios no processo construção de especificações formais se manifestam também em um grande número de trabalhos: conforme o levantamento feito por Robillard et al. (2013) sobre diferentes técnicas de inferência automática de propriedades de API, predomina na maior parte dos trabalhos um esforço manual de validação de resultados. Isto compromete tanto a reprodução de resultados como também ameaça a validade e generalidade dos resultados obtidos por pesquisadores.

Para contornar as dificuldades de avaliação de um minerador de especificação foi criada a plataforma CORES, acrônimo para **CO**llaborative **RE**ference **S**pecifications, que se busca funcionar como um repositório público com especificações formais desenvolvidas colaborativamente[1]. A Figura 4 exibe o processo colaborativo proposto pela plataforma.

Figura 4 – Processo de refinamento colaborativo de especificações formais.



Fonte: Elaborado pelo autor.

A Figura 4 busca mostrar o processo de refinamento de especificações na plataforma, cujo trabalho colaborativo é empregado para superar as dificuldades de construção de uma especificação formal por um único indivíduo. Acredita-se que a escassez de especificações formais de referência se deve em parte aos esforços necessários para se especificar formalmente softwares maiores e mais populares, conforme discutido por Sommerville (2010, cap. 27) sobre os aspectos inibidores da adoção ampla de Especificações Formais.

A partir de especificações iniciais obtidas pela comunidade, a mesma pode ser expandida para incluir outras situações e casos que foram registrados por outros membros. Após esta fase de expansão da documentação, a especificação pode ser reduzida para

[1]Disponível online em <https://github.com/otmarjr/cores>

evitar que outros casos inválidos que surgiram na fase anterior ou na versão inicial da especificação formal sejam incluídos. Ao final, uma nova versão da especificação formal está disponível e pronta para ser usada para os mais diferentes fins pela comunidade. Caso evoluções sejam necessárias, o mesmo processo pode ser executado de maneira iterativa, tendo agora como entrada de especificação formal inicial, aquela produzida em uma iteração anterior do fluxo de trabalho na plataforma.

3.3 Refinamento iterativo de especificações formais

Dada a natureza das atividades de construção de uma especificação formal, propõe-se uma abordagem de refinamento progressivo das especificações formais: a partir de uma especificação formal de referência simples, a mesma passa por um ciclo de refinamento, conforme mostrado na Figura 4. Este refinamento pode ser para incluir novos casos válidos que são conhecidos ou adicionar restrições de certas situações em uma especificação. Tal processo tem semelhança com o ciclo de aperfeiçoamento de um software, em que novas funcionalidades são acrescidas a partir dos requisitos descobertos e em seguida, defeitos são removidos, de maneira que o software é iterativamente evoluído.

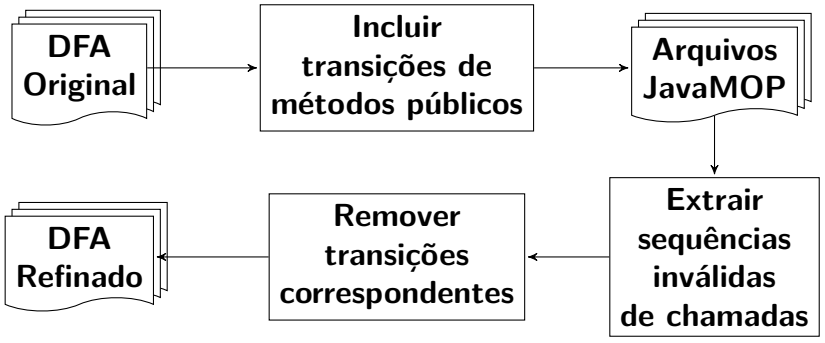
Para estudar este processo, neste trabalho, o conjunto de dados de referência escolhido foi o JDK, de maneira a se gerar uma especificação formal de referência para a mesma biblioteca de software alvo da implementação do *framework* MUTE. Os arquivos de apoio que acompanham o material dos trabalhos de investigação do JDK de Pradel, Bichsel e Gross (2010), Lee et al. (2012) foram utilizados como especificação inicial do fluxo de refinamento iterativo da especificação formal. Um exemplo de especificação inicial que pode ser utilizado neste processo é mostrado na Figura 6.

Para entender melhor este processo de refinamento iterativo das especificações formais, pode-se observar a Figura 5, que detalha as etapas de aprimoramento da documentação de uma única classe do sistema. Para cada classe que compõe a API, o DFA correspondente é submetido a um método que se baseia na estratégia de expandir a linguagem do DFA para cobrir todos os cenários possíveis. Em seguida, é feita uma poda a fim de retirar todos os padrões previamente conhecidos como inválidos. No contexto exemplificado pela figura, estas sequências inválidas são extraídas de arquivos no formato da ferramenta JavaMOP (CHEN; ROşU, 2005).

Assume-se que ao se iniciar o processo de refinamento de uma especificação formal, ela contenha alguns fluxos básicos e que possíveis cenários que se deseja incorporar ou remover da mesma sejam de conhecimento de pesquisadores e desenvolvedores interessados em aprimorar a especificação formal.

Um exemplo disso é a especificação formal inicialmente construída por Pradel, Bi-

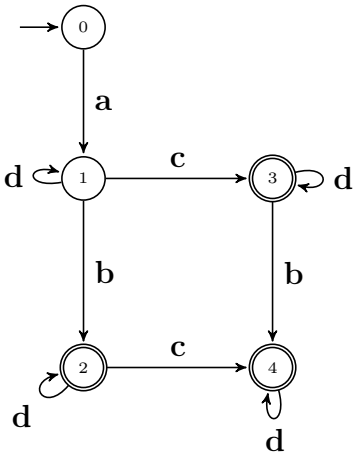
Figura 5 – Processo de extensão da especificação formal seguido pela poda de sequências inválidas.



Fonte: Elaborado pelo autor.

chsel e Gross (2010). Cada DFA presente nesta especificação formal foi projetado para que sua linguagem corresponda aos principais cenários de uso da classe, como por exemplo, uma sequência que envolva os métodos `connect()` e `close()` da classe `java.net.URL`. Um desenvolvedor que queira verificar se seu programa está em conformidade com esta especificação pode enfrentar problemas, porque esta especificação inicial não suporta chamadas de chamadas como `toString()`, `hashCode()` e métodos *getters*. A especificação formal parcial da classe `java.net.URL` é mostrada na Figura 6, cujos métodos correspondentes podem ser identificados na Tabela 1.

Figura 6 – Adaptação do FSM que descreve classe java.net.URL.



Fonte: Adaptado de (PRADEL; BICHSEL; GROSS, 2010).

Tais métodos são secundários na execução dos fluxos básicos de utilização da classe, mas acabam ocorrendo em implementações reais que envolvem aquela API. Além disso, estes métodos estão presentes em programas reais sem nenhum efeito colateral negativo. Muitas vezes eles podem se encontrar em um código-fonte para outros propósitos, como o de *log* de informações.

Tabela 1 – Métodos correspondentes aos símbolos do autômato da Figura 6

Símbolo	Sequência de métodos
a	`<init>()`
b	`openStream()`
c	`getContent()`
d	`sameFile()`, `getAuthority()`, `getHost()`, `getDefaultPort()`, `getQuery()`, `getPath()`, `getFile()`, `getPort()`, `getRef()`, `toExternalForm()`, `getProtocol()`, `toURI()`, `getUserInfo()`
e	`setURLStreamHandlerFactory()`
f	`java.net.Url.*` $-$ **a** $-$ **b** $-$ **c** $-$ **d**
g	**f** $-$ **e**

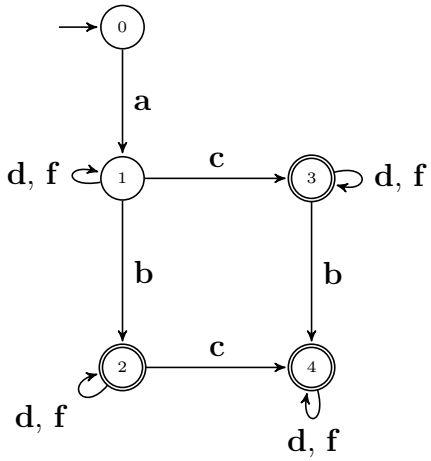
A observação atenta da Figura 6 indica que o DFA correspondente à classe `URL` do pacote `java.net` não reconhece a chamada a nenhum destes métodos *redundantes*. Para se incluir estas variações de chamadas possíveis na especificação inicial, podem ser utilizadas as bibliotecas de *Reflection* do Java, que permitem a consulta por todos os métodos públicos disponíveis em um pacote Java. Neste trabalho, esta foi a técnica utilizada. Uma vez que estes métodos são secundários - pois não faziam parte dos cenários básicos da especificação inicial, os mesmos são adicionados como auto-laços nos estados do DFA modificado, de maneira a não alterar o estado do DFA original em relação ao método adicionado.

A Figura 7 apresenta o mesmo autômato após a inclusão destes auto-laços correspondentes aos métodos públicos da classe que estavam ausentes anteriormente. O determinismo de cada autômato é preservado ao se incluir as auto-transições nos estados apenas quando aquele estado não possui outra transição correspondente a este método público.

Após a execução da primeira etapa de inclusão de métodos que converte o DFA da Figura 6 naquele mostrado na Figura 7, é preciso trabalhar no DFA para evitar que ele fique excessivamente permissivo e aceite sequências de chamadas inválidas. Para isto, o viés colaborativo da plataforma mostrado na Figura 4 entra novamente em cena por meio da leitura e avaliação de sequências inválidas informadas na plataforma CORES.

Na versão desenvolvida neste trabalho, são utilizados os arquivos de formalização do JDK produzidos por Lee et al. (2012), cujo formato é da ferramenta JavaMOP. Esta biblioteca implementa um mecanismo de verificação formal em tempo de execução de programas Java para verificar violações cometidas por programas em monitoramento. A verificação pode se basear diferentes tipos de formalismos, como *linear temporal logic* (LTL), expressões regulares estendidas, autômatos, dentre outros.

Figura 7 – DFA da Figura 6 após a fase de expansão.



Fonte: Elaborado pelo autor.

No momento da escrita deste trabalho, o repositório CORES foi alimentado com todas as sequências cujo arquivo JavaMOP correspondente representa a violação com uma expressão regular estendida. Espera-se que, no futuro, a comunidade de pesquisadores interessados contribua com extração dos outros formalismos existentes no JavaMOP.

O Código 4 busca ilustrar a etapa de poda baseada nos arquivos do JavaMOP, ao mostrar o conteúdo do arquivo URL_SetURLStreamHandlerFactory.mop, utilizado para se extrair uma sequência de chamadas inválida que não deve fazer parte da linguagem de um DFA da classe `java.net.Url`.

Código 4 – Arquivo JavaMOP com a formalização para monitorar a quantidade de chamadas ao método *setURLStreamHandlerFactory* da classe *Url*.

```
1  package mop;
2  import java.net.Url.*;
3  import javamoprt.MOPLogging;
4  import javamoprt.MOPLogging.Level;
5  /* Warns if URL.setURLStreamHandlerFactory() is called multiple times.
6   * This method can be called at most once in a given Java Virtual Machine.
7   * http://docs.oracle.com/javase/6/docs/api/java/net/URL.html#
        setURLStreamHandlerFactory%28java.net.URLStreamHandlerFactory%29
8   * @severity error
9   */
10 URL_SetURLStreamHandlerFactory() {
11    event set before() :
12       call(* URL.setURLStreamHandlerFactory(..)) {}
13
14    ere : set set+
15
16    @match {
```

```
17          MOPLogging.out.println(Level.CRITICAL, __DEFAULT_MESSAGE);
18          MOPLogging.out.println(Level.CRITICAL, "URL.
                setURLStreamHandlerFactory() can be called at most once in a given
                 Java Virtual Machine.");
19      }
20  }
```

Fonte: Adaptado de Lee et al. (2012).

As Linhas 11 e 12 do Código 4 definem o símbolo *set* como uma chamada para o método `setURLStreamHandlerFactory()` da classe `java.net.URL`. A linha 14 referencia este símbolo e define uma expressão regular estendida a ser monitorada pelo JavaMOP. A linha 16 indica que caso seja observada um sequência de chamadas compatível com a expressão regular, um erro deve ser registrado. Os leitores interessados no JavaMOP são aconselhados a se referir aos trabalhos de Chen e Roşu (2005), Meredith et al. (2012) para maiores detalhes do funcionamento do JavaMOP. Conforme é possível observar nos comentários do arquivo mostrado no Código 4, o objetivo é para verificar se o método `setURLStreamHandlerFactory()` não é chamado mais de uma vez.

Após a extração desta sequência inválida, definida como $ere = set \quad set+$, é utilizada a operação de complemento relativo de linguagens regulares para se obter um DFA cuja linguagem não aceite nenhuma *string* que possua *ere* como subsequência. Para isto, um novo DFA é gerado para corresponder à linguagem $L_{refinada} = L_{estendida} - ere$, onde $L_{estendida}$ corresponde à linguagem do DFA mostrado na Figura 7. Esta operação pode ser calculada por meio da interseção de $L_{estendida}$ com o complemento de *ere*, operação que está disponível, por exemplo, na biblioteca de manipulação de autômatos construída por (MØLLER, 2010).

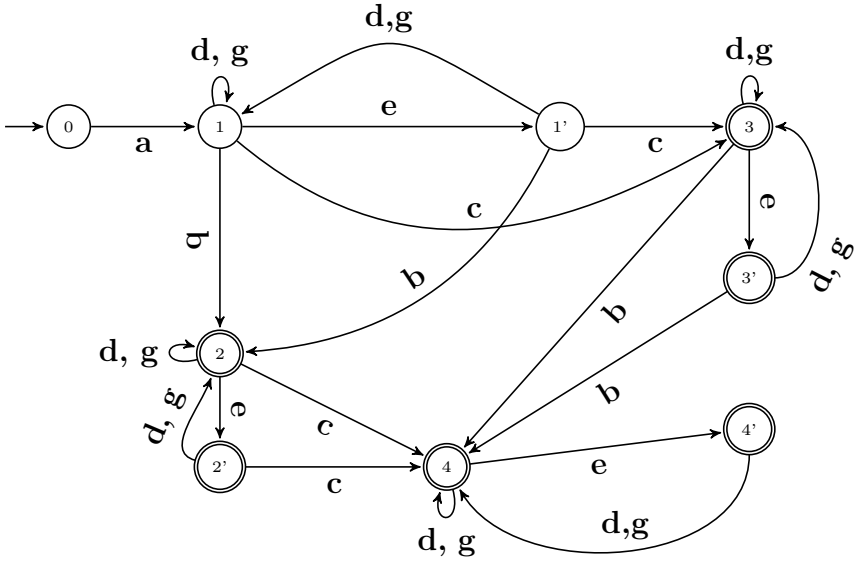
A Figura 8 ilustra o DFA resultante que possui esta linguagem e não reconhece a subsequência correspondente à expressão regular *ere*.

Ao final, após executar este processo para cada uma das sequências proibidas disponibilizadas, é obtido um DFA refinado, que pode ser utilizado para apoiar um desenvolvedor no processo de verificação da sequência de chamadas encontrada em sua aplicação cliente da API, mesmo com a presença de outras chamadas redundantes presentes no código da aplicação cliente.

Além de ser um DFA mais abrangente, o mesmo explicita sequências de chamadas reconhecidamente inválidas. A Figura 8 possui transições correspondentes ao método `setURLStreamHandlerFactory`, todavia, sem suportar duas ou mais chamadas consecutivas a este método.

Por outro lado, o DFA original, mostrado na Figura 6, negligencia este método,

Figura 8 – Autômato da Figura 7 após a operação de diferença da linguagem correspondente à expressão regular do Código 4.



Fonte: Elaborado pelo autor.

assim como os métodos triviais como `toString()` e similares. Uma vez que a linguagem deste DFA não contém nem os métodos redundantes, nem métodos que participam de sequências proibidas, caberá ao desenvolvedor discernir o motivo da não aceitação da sequência de chamadas, o que prejudica a compreensão em comparação com o DFA da Figura 8 e disponibilizado na plataforma CORES.

4 EXPERIMENTOS

Foram realizados experimentos para se avaliar a implementação da abordagem MUTE, discutida na Seção 3.1, tendo como alvo do processo de mineração de especificação um subconjunto de classes do *Java Development Kit* (JDK). O JDK contém as classes básicas da linguagem de programação Java e também é alvo outros estudos relacionados (WU et al., 2011; LEE; CHEN; ROsU, 2011), inclusive o minerador de especificações formais correspondente ao estado-da-arte (PRADEL; BICHSEL; GROSS, 2010).

Os experimentos foram conduzidos tendo como referência uma especificação formal composta por 21 classes concretas dos pacotes `java.net` e `java.util`, cujos DFAs foram recuperados da plataforma CORES, discutida na Seção 3.2 . Estas classes estão listadas na Tabela 2 juntamente com a quantidade de métodos públicos existentes em cada uma.

Tabela 2 – Classes analisadas

Pacote	**Classe**	**N.º de métodos públicos**
java.net	DatagramSocket	39
java.net	MultiCastSocket	54
java.net	Socket	50
java.net	URL	28
java.util	ArrayDeque	51
java.util	ArrayList	42
java.util	EnumMap	35
java.util	EnumSet	43
java.util	Formatter	16
java.util	HashMap	33
java.util	HashSet	28
java.util	Hashtable	36
java.util	IdentityHashMap	33
java.util	LinkedHashMap	33
java.util	LinkedHashSet	28
java.util	LinkedList	62
java.util	PriorityQueue	33
java.util	StringTokenizer	15
java.util	TreeMap	57
java.util	TreeSet	57
java.util	WeakHashMap	32

Fonte: Dados de pesquisa.

Testes unitários foram extraídos a partir do código-fonte do Open JDK[1]. Por terem sido criados por desenvolvedores do Open JDK, os testes são considerados manuais. Já os testes unitários gerados automaticamente foram produzidos pela ferramenta Randoop

[1]Versão 6-b33, recuperado em 29 de dezembro de 2014

(PACHECO; ERNST, 2007).[2].

Devido à natureza exponencial da quantidade de possibilidades de casos de testes a serem gerados, o critério de parada utilizado foi tempo: para cada classe, a ferramenta foi configurada para executar durante 10 minutos e produzir testes com no máximo 20 chamadas de métodos, além de reter na saída apenas os casos de teste que não levantaram exceção durante o processo de geração e execução pelo Randoop. A partir desta massa de

[2]Versão 1.3.4

dados, os testes unitários foram instrumentados com o AspectJ[3], para produzir os registros de execução dos testes unitários e gerar a entrada utilizada pelo *framework* MUTE.

4.1 Resultados

As Tabelas 3 e 4 apresentam os resultados dos estágios de execução da implementação da abordagem MUTE e os valores de precisão obtidos tanto de testes unitários manuais, disponibilizados junto ao código-fonte do Open JDK, quanto dos testes unitários gerados automaticamente pelo Randoop. A primeira linha, cuja coluna **Tipo** indica *Manual*, corresponde aos testes distribuídos com o código-fonte do Open JDK, ao passo que os casos de teste criados pelo Randoop correspondem à linha cujo valor da coluna **Tipo** é *Gerado*.

A terceira coluna destas tabelas indica o número de testes unitários disponíveis como entrada para o primeiro componente no *pipeline* da arquitetura MUTE. A coluna seguinte indica o número de itens que foram aceitos pelo componente *Filtro*. O número de testes unitários considerados úteis para aprender a utilizar a API, classificados como de propósito *externo* pelo componente *Classificador*, são indicados pela coluna **Externos**.

Finalmente, a coluna **Aceitos** indica quantos dos testes unitários estão em conformidade com a especificação formal de referência. Esta verificação é feita a partir do DFA produzido pelo componente *Construtor de Autômatos* e a proporção entre o número de testes unitários classificados como *externos* e o total de testes unitários que produzem DFAs *aceitos*. Este índice determina a precisão do método, indicada na penúltima coluna, com o rótulo **Precisão**. A última coluna, **Revocação**, indica quantos dos *traces* corretos foram descobertos, com base na discussão feita na Subseção 2.2.5.

4.1.1 Precisão e revocação das classes no pacote `java.util`

A Tabela 3 revela características diferentes do funcionamento do método MUTE, tanto para testes automáticos quanto para testes manuais. Apesar do pacote `java.util` conter 17 das 21 classes analisadas, há um menor número de testes unitários manuais, classificados como *externos*, para guiar o processo de aprendizado dos DFAs: apenas seis, face aos 15 testes unitários manuais *externos* extraídos do pacote `java.net`, a despeito deste último pacote possuir apenas quatro classes analisadas.

As classes `EnumSet` e `EnumMap` são omitidas da Tabela 3 devido à ausência de dados disponíveis, a partir tanto do Randoop quanto dos testes unitários manuais, conforme discussão feita ao final desta seção.

[3]Versão 1.8 do AspectJ

Tabela 3 – MUTE aplicado às classes do pacote `java.util`

Classe	Tipo	N.º de testes	Filtrados	Externos	Aceitos	Precisão
ArrayDeque	Manual	6	0	0	0	-
	Gerado	133526	29741	13750	13750	1.0
ArrayList	Manual	52	1	0	0	-
	Gerado	131702	16306	10005	10005	1.0
Formatter	Manual	26	0	0	0	-
	Gerado	169887	13608	7793	6550	0.8405
HashMap	Manual	39	2	1	1	1.0
	Gerado	112054	24356	13311	13311	1.0
HashSet	Manual	23	2	2	2	1.0
	Gerado	100035	19536	11492	11492	1.0
Hashtable	Manual	18	0	0	0	-
	Gerado	103204	28886	15556	15556	1.0
IdentityHashMap	Manual	8	0	0	0	-
	Gerado	103366	23800	12575	12575	1.0
LinkedHashMap	Manual	8	0	0	0	-
	Gerado	107402	26002	14562	14562	1.0
LinkedHashSet	Manual	5	0	0	0	-
	Gerado	103648	19945	11701	11701	1.0
LinkedList	Manual	15	1	1	1	1.0
	Gerado	152098	34436	12122	12122	1.0
PriorityQueue	Manual	6	0	0	0	-
	Gerado	113167	16936	9874	9874	1.0
StringTokenizer	Manual	4	2	2	2	1.0
	Gerado	190404	24720	13588	13588	1.0
TreeMap	Manual	19	0	0	0	-
	Gerado	116415	25148	12446	12446	1.0
TreeSet	Manual	12	0	0	0	-
	Gerado	139567	23211	12726	12726	1.0
WeakHashMap	Manual	8	0	0	0	-
	Gerado	115606	22937	12397	12397	1.0

Fonte: Dados de pesquisa.

A análise da Tabela 3 revela que os testes unitários das classes utilitárias e de coleções encontradas no pacote `java.util`, presentes no código-fonte do Open JDK, são mais voltados à verificação da proteção das classes contra situações limite. Por exemplo, uma das verificações feitas pelo teste unitário `MOAT`, mostrado no Código 5, é checar se uma instância da classe `HashMap` sem entradas lança exceções para chamadas inválidas. A instância é criada na linha 13 e é recebida pela rotina `testEmptyMap` sem que nada

tenha sido adicionado à esta instância.

Entre as linhas 19 e 24, são feitas verificações para checar se os métodos de leitura da classe `HashMap` reportam que a instância recém-criada não possui entradas. Já as situações de erro são testadas entre as linhas 26 e 29, nas quais é esperado que a classe lance exceções do tipo `NullPointerException` após as chamadas nas linhas 26 e 28.

Código 5 – Teste unitário `MOAT`.

```
1
2  public class MOAT {
3    public static void main(String[] args) throws Throwable {
4      try { realMain(args); }
5      catch (Throwable t) { unexpected(t); }
6
7      System.out.printf("%nPassed=%d, failed = %d%n%n", passed, failed);
8      if (failed > 0) throw new Exception("Some tests failed");
9    }
10
11   public static void realMain(String[] args) {
12     // (...)
13     Map emptyMap = new HashMap();
14     testEmptyMap(emptyMap);
15     // (...)
16    }
17
18   private static <K,V> void testEmptyMap(final Map<K,V> m) {
19     check(m.isEmpty());
20     equal(m.size(), 0);
21     equal(m.toString(),"{}");
22     testEmptySet(m.keySet());
23     testEmptySet(m.entrySet());
24     testEmptyCollection(m.values());
25
26     try { check(! m.containsValue(null)); }
27     catch (NullPointerException _) { /* OK */ }
28     try { check(! m.containsKey(null)); }
29     catch (NullPointerException _) { /* OK */ }
30     check(! m.containsValue(1));
31     check(! m.containsKey(1));
32   }
33
34   static void check(boolean cond) {
35     if (cond) pass();
36     else fail();
37   }
38
```

```
39    static void equal(Object x, Object y) {
40      if (x == null ? y == null : x.equals(y)) pass();
41      else {
42        System.out.println(x + " not equal to " + y);
43        fail();
44      }
45    }
46  }
```

Fonte: Adptado do código-fonte do Open JDK 6(Oracle Corporation, 2011).

Além deste teste, observa-se também que juntamente ao código-fonte do Open JDK é distribuído um grande número de testes unitários dedicados à verificação de controles de sincronização internos das classes, empregados no contexto de aplicações *multi-thread*, o que muitas vezes é feito com exceções específicas para reportar a ocorrência de erros de utilização concorrente de uma estrutura de dados.

Uma vez que os testes unitários distribuídos juntamente com o Open JDK são predominantemente voltados para a verificação de situações excepcionais de controle interno das classes, a implementação do MUTE não pôde utilizar como exemplos para aprendizado de DFA mais do que seis testes no total. Contudo, todos os testes unitários coletados levaram à produção de sequências de chamadas em conformidade com a especificação formal de referência utilizada.

Efetivamente, o principal problema ao se utilizar os testes unitários construídos originalmente pelos desenvolvedores da biblioteca é sua a escassez: para nenhuma classe foi possível extrair mais do que dois exemplos classificados com o rótulo *externo*, próprios para extração de exemplos de utilização da API. Apenas as classes `HashSet` e `StringTokenizer`, dentre as 17 analisadas no pacote `java.util`, tiveram ao final dois testes unitários do tipo *externo* utilizados no processo de aprendizado de DFA.

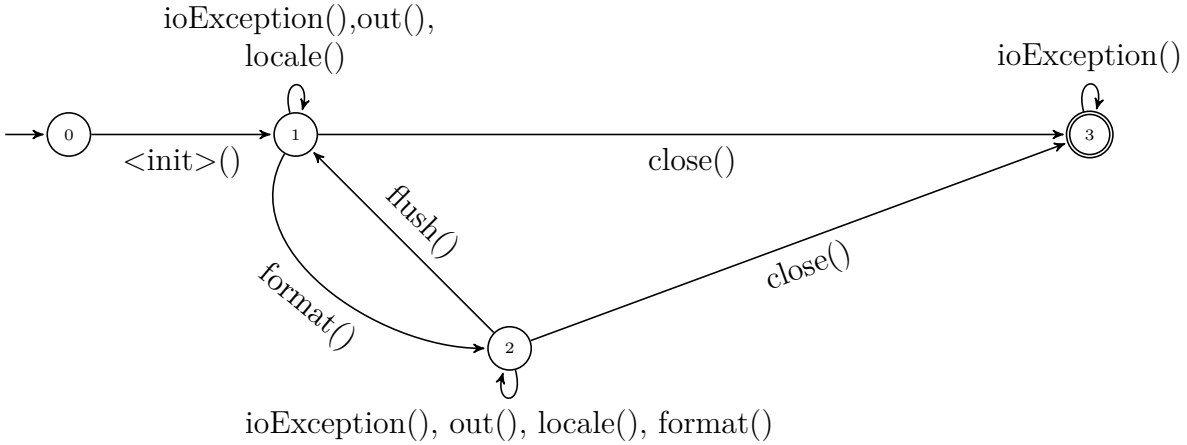
O Randoop, por sua vez, tem uma alta taxa de sucesso na geração de testes unitários para classes do pacote `java.util`. Como resultado do grande número de operações que trabalham com tipos primitivos, ele é capaz de gerar mais de 100 mil testes unitários para todas as classes no pacote, sem a presença de falsos positivos.

A única exceção para a afirmação anterior é a classe `Formatter`, para a qual a abordagem MUTE não é capaz de inferir uma especificação formal sem falsos positivos. Para entender melhor este caso, pode-se observar a Figura 9, que contém a especificação formal de referência original do trabalho de Pradel, Bichsel e Gross (2010). É importante observar que apesar deste ser apenas o DFA utilizado para se produzir um equivalente refinado para a plataforma CORES, sua linguagem já permite entender a razão por trás dos falsos positivos para a classe `Formatter`.

Este DFA mostra que, simplificadamente, os cenários válidos para esta classe envolvem uma chamada para o método `close()` precedida, opcionalmente, de chamadas aos métodos `ioException(), out(), locale()` e `format()`.

Neste contexto, o método `format()` acaba sendo o principal responsável pelos falsos positivos. Esta operação recebe como parâmetro uma cadeia de caracteres que segue um certo padrão. Por exemplo[1], a chamada `format(Locale.FRANCE, "e = %+10.4f", Math.E)`, utilizada para produzir a *string* `"e = +2,7183"`.

Figura 9 – DFA que especifica cenários válidos para a classe `java.util.Formatter`.



Fonte: Adaptado de (PRADEL; BICHSEL; GROSS, 2010).

Apesar da concepção original deste método, o conjunto de *strings* não-nulas que o Randoop utiliza não leva à ocorrência de exceções. Uma vez que o método receba cadeias de caracteres que não correspondem a nenhum padrão, o texto recebido gera uma cadeia cujo formato é igual ao parâmetro recebido. Com isto, o gerador produz um teste válido e algumas das sequências produzidas incorretamente têm o método *format()* como o último chamado.

Nem todas as sequências geradas pelo Randoop que possuem o método `format()` são seguidas pelo método `close()`, devido à natureza aleatória do processo de geração de testes. Uma vez que todos os cenários válidos exigem pelo menos uma chamada ao método `close()`, as sequências produzidas que não possuem esta chamada acabam por gerar falsos positivos.

Um exemplo de teste gerado correspondente ao exemplo do parágrafo anterior pode ser verificado no Código 6. Ao observar a sequência de chamadas para a instância `var0` criada na linha 4, é possível observar que as operações invocadas nas linhas 5 a 9, seguidas ao final pela chamada da linha 19 formam a sequência ⟨`init ()`, `out()`, `locale()`, `out()`, `out()`, `ioException()`, `format()`⟩. Esta série de chamadas, que não

[1]Extraído de <http://docs.oracle.com/javase/6/docs/api/java/util/Formatter.html>

levanta nenhuma exceção, devido ao funcionamento do método `format()`, também não faz parte dos cenários de utilização da classe. Isto pode ser verificado ao observar que o DFA mostrado Figura 9 não aceita esta sequência de chamadas, o que leva esta sequência coletada pelo minerador MUTE a ser considerada um falso positivo.

Código 6 – Teste unitário automático que gerou falso positivo.

```
1   public void test35() throws Throwable {
2     if (debug) { System.out.println(); System.out.print("RandoopTest0.
          test35"); }
3
4     java.util.Formatter var0 = new java.util.Formatter();
5     java.lang.Appendable var1 = var0.out();
6     java.util.Locale var2 = var0.locale();
7     java.lang.Appendable var3 = var0.out();
8     java.lang.Appendable var4 = var0.out();
9     java.io.IOException var5 = var0.ioException();
10    java.util.Formatter var7 = new java.util.Formatter();
11    java.lang.Appendable var8 = var7.out();
12    java.util.Formatter var9 = new java.util.Formatter();
13    java.util.Locale var10 = var9.locale();
14    java.util.Formatter var11 = new java.util.Formatter(var10);
15    java.util.Formatter var13 = new java.util.Formatter();
16    java.lang.Object[] var16 = new java.lang.Object[] { (-1L)};
17    java.util.Formatter var17 = var13.format("hi!", var16);
18    java.util.Formatter var18 = var7.format(var10, "hi!", var16);
19    java.util.Formatter var19 = var0.format("", var16);
20
21    // Regression assertion (captures the current behavior of the code)
22    assertNotNull(var1);
23
24    // Regression assertion (captures the current behavior of the code)
25    assertNotNull(var2);
26
27    // (...) assertNotNull para var3 a var19
28  }
```

Fonte: Dados de pesquisa.

As classes `EnumMap` e `EnumSet` são as duas únicas, de um total de 21, para as quais não foi possível aprender nenhum DFA, seja por meio de testes unitários manuais, seja por meio de testes unitários gerados.

A análise da classe `EnumMap` indica que o OpenJDK 6 possui apenas dois testes unitários para ela, ambos não sendo passados à frente no *pipeline* pelo componente *Filtro*. Os testes unitários nos arquivos `EnumMapBash.java` e `ToArray.java` são descartados

devido à interação com classes em pacotes diferentes. O Randoop por sua vez, não provê suporte à construção de parâmetros que sejam tipos enumerados - construção `enum`. Com isto, não são gerados testes unitários que exercitem a classe `EnumMap`.

Investigação semelhante foi conduzida nos testes unitários para a classe `EnumSet`, o que revelou uma limitação atual na implementação do coletor de *traces*: todos os nove testes unitários da classe envolvem a criação de instâncias a partir de métodos estáticos como `allOf()` ou `range()`, dentre outros, da classe `EnumSet`. Uma vez que a instrumentação implementada com o AspectJ se aplica apenas aos testes unitários em si, e não às bibliotecas que eles consomem, não foi possível detectar a construção de instâncias feitas no interior desta classe, presente no JDK.

A implementação deste rastreio da criação de instâncias inicializadas em métodos fabricadores no JDK exigiria uma configuração avançada do AspectJ e não foi encontrada nenhuma implementação que permitisse a instrumentação do JDK simultânea à execução de programas na plataforma. Por conta disto, decidiu-se que tal evolução seja implementada em versões futuras, com técnicas mais avançadas que suportem tais cenários. Quanto ao Randoop, a classe `EnumSet` trabalha com enumerações, e portanto, as mesmas limitações discutidas para a classe `EnumMap` impediram a geração automática de testes também para esta classe.

4.1.2 Precisão e revocação para as classes no pacote `java.net`

A Tabela 4 revela que, apesar do pequeno número de testes unitários disponíveis para o aprendizado da API, a precisão do mesmo é muito alta: com exceção da classe `MulticastSocket`, os testes unitários de todas as outras classes levaram à inferência de uma especificação formal sem falsos positivos. Ainda assim, para esta classe, há um único teste unitário manual que produz um falso positivo, o que preserva um resultado geral de precisão elevado.

O único falso positivo, correspondente à classe `MulticastSocket`, tem origem no teste unitário `TestDefaults`, mostrado no Código 7. A *string* produzida pelas chamadas feitas nas linhas 5, 7, 8 e 9 não pertence à linguagem do DFA de referência, o que a caracteriza como um exemplo incorretamente extraído pelo minerador MUTE. Apesar da sequência de chamadas ⟨`<init>()`, `getTimeToLive()`, `getInterface()`, `getLoopbackMode()`⟩ ser incorretamente detectada como parte da linguagem da classe `MulticastSocket`, é importante observar que ela não gera nenhuma exceção ao ser invocada.

O falso positivo decorre do fato de a sequência não ser destinada a exercitar nenhum cenário de uso típico da classe, os quais, em geral, envolvem chamadas a métodos voltados

Tabela 4 – MUTE aplicado às classes do pacote `java.net`

Classe	Tipo	N.º de testes	Filtrados	Externos	Aceitos	Precisão
DatagramSocket	Manual	12	8	3	3	1.0
	Gerado	35133	10918	6122	2777	0.4536
MulticastSocket	Manual	16	10	6	5	0.8333
	Gerado	115208	60635	40976	40877	0.9976
Socket	Manual	31	6	2	2	1.0
	Gerado	14554	6997	4419	1396	0.3159
URL	Manual	26	7	4	4	1.0
	Gerado	300	0	0	0	-

Fonte: Dados de pesquisa.

para a comunicação de rede, como por exemplo `send()`, `joinGroup()` e `close()`. É possível observar que nenhum destes métodos está presente no Código 7.

Código 7 – Teste unitário `TestDefaults`, para testar valores padrão coletados pela classe `MulticastSocket`.

```
1  import java.net.*;
2  public class TestDefaults {
3
4      public static void main(String args[]) throws Exception {
5          MulticastSocket mc = new MulticastSocket();
6
7          int ttl = mc.getTimeToLive();
8          InetAddress ia = mc.getInterface();
9          boolean mode = mc.getLoopbackMode();
10
11         System.out.println("Default multicast settings:");
12         System.out.println("      ttl: " + ttl);
13         System.out.println("interface: " + ia);
14         System.out.println(" loopback: " + mode);
15
16         if (ttl != 1) {
17             throw new Exception("Default ttl != 1  -- test failed!!!");
18         }
19     }
20 }
```

Fonte: Adptado do código-fonte do Open JDK 6(Oracle Corporation, 2011).

Este teste, pelo contrário, verifica se os valores como *time-to-live* dentre outros são corretamente lidos do sistema operacional pelo construtor da classe. De acordo com os

comentários explicativos no código-fonte deste teste unitário, o objetivo é checar se os métodos `getInterface()`, `getTimeToLive()` and `getLoopbackMode()` funcionam corretamente no *kernel* 2.2 do Linux quando o IPv6 está habilitado. Devido à verificação de correlação entre valores de campos internos em relação aos valores padrão fornecidos pelo sistema operacional, acredita-se que este seja um tipo de teste que deveria ter sido classificado como *interno*, que, ao contrário da maioria dos outros testes unitários, não utiliza exceções para verificar a consistência interna da classe. Este teste portanto revela uma limitação do componente *Classificador* da arquitetura MUTE e indica que a precisão e obtenção de regras de qualidade depende de um bom componente *Classificador*.

Já os resultados para os testes gerados aleatoriamente pelo Randoop apresentam um comportamento diferente, à exceção da classe `MulticastSocket`. Esta classe possui altos valores de precisão devido aos seus cenários de uso mais simples, diferentemente das outras classes no mesmo pacote. Enquanto os cenários das classes `DatagramSocket` e `Socket` envolvem chamadas para métodos como `send()`, `receive()` ou `connect()`, que possuem parâmetros de tipos complexos, tais como `InetAddress` ou `DatagramPacket`, existem cenários válidos da classe `MulticastSocket`, que possuem uma chamada para o método `close()` sem uma chamada prévia para nenhum outro método que possua parâmetros de tipos complexos.

Uma vez que é difícil gerar aleatoriamente uma sequência válida para os métodos citados, para a qual o Randoop tenha sido capaz de construir um parâmetro de um tipo complexo como `InetAddress` ou `DatagramPacket`, para ser utilizado juntamente com métodos como `send()` ou `bind()` das classes `DatagramSocket` e `Socket`, há uma taxa de falhas maior no processo de geração automática de testes unitários. Além disso, há a influência do mecanismo de retroalimentação do Randoop, que gera combinações progressivamente maiores a partir de sequências menores que foram testadas sem que uma exceção fosse levantada. A influência deste mecanismo é percebida pelo menor número de testes unitários gerados, pois parte do processamento foi desperdiçada com sequências cuja execução falhou no teste do gerador.

Esta solução de compromisso do Randoop, que opta por uma estratégia de geração simples, é mais claramente revelada pela classe `URL`. Uma vez que este não possui nenhuma inteligência embutida para gerar padrões especiais, não é possível então construir instâncias da classe `URL`, porque para todos os valores presentes no catálogo, são levantadas exceções do tipo `java.net.MalformedURLException`.

Esta exceção acontece porque após escolher um método para compor um teste unitário em construção, o Randoop busca seu catálogo de valores pré-cadastrados para os parâmetros do método. Para parâmetros do tipo *String*, o catálogo de opções para seleção aleatória contém opções como cadeias de caracteres vazias, valores nulos ou literais

simples, como o valor `"hi!"`. Nenhum destas opções pré-definidas pelo gerador segue o padrão de um URI válido, o que previne a criação de sequências não vazias para a classe `URL`.

4.1.3 Discussão dos resultados

Para 17 das 21 classes analisadas, havia uma alternativa dentre os testes unitários manuais ou gerados que levavam à inferência de uma especificação formal sem falsos positivos. Os testes unitários dos desenvolvedores da biblioteca não estão disponíveis em quantidades abundantes. Os resultados indicam que as classes que dependem predominantemente de operações com parâmetros de tipos primitivos são aquelas que mais se beneficiam dos testes unitários gerados pelo Randoop: as especificações inferidas a partir dos mesmos não continham falsos positivos para 16 das 17 classes do pacote `java.util`. Mesmo para a única classe do pacote para a qual não foi possível evitar falsos positivos a precisão ainda foi alta, acima de 84%.

Altos valores de precisão foram alcançados para classes dos pacotes `java.net` e `java.util`, especialmente quando testes unitários escritos manualmente estão disponíveis: 21 de 22 testes unitários classificados como *externos* pelo componente *Classificador* estavam em conformidade com a especificação formal de referência disponível na plataforma CORES.

Enquanto que classes que trabalham com parâmetros de tipos complexos, ou padrões específicos em cadeias de caracteres, são as que mais se beneficiam dos testes unitários distribuídos juntamente ao código-fonte das bibliotecas de software, nota-se que um gerador de testes unitários automático pode levar a bons resultados. Este é o caso das classes no pacote `java.util`, para as quais o Randoop foi capaz de produzir um grande número de testes unitários, o que permitiu contornar a escassez de testes unitários manuais que pudessem servir de exemplo de utilização da API de classes neste pacote.

Estes resultados encorajam o desenvolvimento de novas funcionalidades em ferramentas de geração de teste, como a incorporação de padrões específicos nas *strings* geradas e criação em tempo de execução do Randoop de tipos enumerados. Caso a ferramenta tivesse em seu conjunto de opções os padrões compatíveis com *Urls* e outros como nomes de pessoas, códigos-postais e outros recursos presentes em ferramentas de geração de dados, a implementação da abordagem MUTE provavelmente teria alcançado uma taxa de sucesso maior para as classes `java.net.URL` e `java.util.Formatter`. Além disso, o suporte a tipos enumerados também contribuiria para aumentar o alcance do gerador de testes.

5 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho, foi conduzida uma investigação acerca do papel dos testes unitários no processo de inferência de especificações formais. Para isto foi proposta a abordagem MUTE, um inovador *framework* de mineração de testes unitários que pode ser utilizado tanto para testes unitários construídos por programadores, quanto para testes unitários gerados automaticamente por ferramentas. Em particular, a abordagem se baseia na extração de sequências de chamadas de métodos, presentes nos testes unitários, para convertê-las em FSAs de múltiplos estados que representam as diferentes classes de um sistema orientado por objetos.

5.1 Conclusões

Os experimentos conduzidos nesta pesquisa permitiram a validação de trabalhos semelhantes como (PRADEL; BICHSEL; GROSS, 2010; DALLMEIER et al., 2012), e permitiu observar que os testes unitários podem servir como uma boa alternativa inicial para produção de uma especificação formal para uma biblioteca de software que possa auxiliar na verificação de sequências de chamadas de métodos a uma API por parte de programadores interessados em utilizá-la. A escassez de testes unitários escritos por programadores pode ser contornada com ferramentas de geração automática de testes unitários para classes cujos métodos predominantemente dependam de tipos de dados simples.

Tal investigação foi feita por meio da implementação do *framework* MUTE e discussão dos principais fatores que governam seu desempenho face a tipos de testes unitários de diferentes origens utilizados como entrada para o método. Os resultados desta avaliação atestam a efetividade da abordagem proposta: nenhum falso positivo para 95% dos testes unitários manuais, o que levou a uma taxa de precisão superior a 83% para todas as classes para as quais haviam disponíveis testes unitários feitos por programadores do Open JDK.

Foi possível observar também a necessidade de melhores *benchmarks* para avaliação de diferentes mineradores de especificações formais, de maneira semelhante à discutida por Robillard et al. (2013). Acredita-se que a dificuldade na construção dessas referências possa ser contornada com uma plataforma aberta, que permita o refinamento progressivo do *benchmark* por meio da comunidade de desenvolvedores e pesquisadores interessados no refinamento de especificações formais.

Diferentemente do estado-da-arte (PRADEL; BICHSEL; GROSS, 2010), não foi utilizado o código-fonte de aplicações clientes no processo de aprendizado da especificação da API. Pelo contrário, foi proposta uma alternativa viável para os desenvolvedores interessados em gerar uma primeira versão da especificação formal de sua biblioteca de software no momento em que nenhuma aplicação cliente ainda esteja disponível.

Estes desenvolvedores de APIs de software possuem na abordagem MUTE uma alternativa viável para documentar a API e distribuí-la juntamente com a biblioteca, o que pode contribuir para aumentar a produtividade dos desenvolvedores que utilizam ferramentas de auxílio à programação.

5.2 Trabalhos Futuros

5.2.1 Melhorias no framework MUTE

Neste trabalho foi possível observar algumas limitações da implementação atual da abordagem MUTE. Houve um teste unitário que não foi corretamente classificado pelo componente *Classificador*, o que levou a um falso positivo. Além disso, a coleta de testes unitários pode ser aprimoradas com outras técnicas que contornem limitações que se baseiem no rastreio de instâncias criadas, pois neste caso não é possível capturar instâncias construídas por métodos fabricadores dentro da própria API a ter a documentação inferida. Isto pode ser alcançado com rastreadores de código mais robustos, capazes de identificar a construção de classes a partir do rastreamento de instruções em *bytecode Java*.

Além do trabalho na implementação do MUTE, trabalhos futuros podem se dedicar ao desenvolvimento de melhores ferramentas de geração de testes unitários. Foram discutidas algumas limitações no Randoop, e as mesmas podem ser objeto de trabalho futuro que vise incorporar novas funcionalidades e mais recursos para alimentação de parâmetros de tipos complexos e padrões específicos em *strings*.

5.2.2 Utilização de formalismos mais expressivos

O formalismo de autômatos utilizado neste trabalho é capaz de representar as sequências de chamadas dos métodos, entretanto, restrições em argumentos dos métodos e invariantes e pré-condições ligadas às classes não são suportadas por este formalismo.

Os resultados apresentados por Reger, Barringer e Rydeheard (2013), capaz de trabalhar com múltiplas instâncias, além de considerar os parâmetros de um método, indicam que o desenvolvimento de novos algoritmos e métodos baseados neste formalismo podem contribuir para o desenvolvimento de abordagens que produzam especificações

com poder de representação e expressividade ainda maiores.

REFERÊNCIAS

AGRAWAL, R.; SRIKANT, R. Fast Algorithms for Mining Association Rules. In: Proceedings of the 20th International Conference on Very Large Data Bases. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994. (VLDB '94), p. 487–499. ISBN 1-55860-153-8. Disponível em: <http://dl.acm.org/citation.cfm?id=645920.672836>.

ALAGAR, V.; PERIYASAMY, K. The Role of Specification. Springer London, 2011. 3-22 p. (Texts in Computer Science). Disponível em: <http://dx.doi.org/10.1007/978-0-85729-277-3_1>. ISBN 978-0-85729-276-6.

ALMEIDA, J. et al. An Overview of Formal Methods Tools andTechniques. In: Rigorous Software Development. Springer London, 2011, (Undergraduate Topics in Computer Science). p. 15–44. ISBN 978-0-85729-017-5. Disponível em: <http://dx.doi.org/10.1007/978-0-85729-018-2_2>.

AMMONS, G.; BODÍK, R.; LARUS, J. R. Mining Specifications. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA: ACM, 2002. (POPL '02), p. 4–16. ISBN 1-58113-450-9. Disponível em: <http://doi.acm.org/10.1145/503272.503275>.

AMMONS, G. et al. Debugging Temporal Specifications with Concept Analysis. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. New York, NY, USA: ACM, 2003. (PLDI '03), p. 182–195. ISBN 1-58113-662-5. Disponível em: <http://doi.acm.org/10.1145/781131.781152>.

ANAND, S. et al. An Orchestrated Survey on Automated Software Test Case Generation. Journal of Systems and Software, Elsevier, v. 86, n. 8, p. 1978–2001, 2013.

Apache Software Foundation. Tomcat 7. 2011. Disponível em: <http://svn.apache.org/viewvc/tomcat/trunk>. Acesso em: 12/05/2015.

BAIER, C.; KATOEN, J.-P. Principles of Model Checking (Representation and Mind Series). Cambridge, Massachusetts: The MIT Press, 2008. ISBN 026202649X, 9780262026499.

BRUCH, M.; MONPERRUS, M.; MEZINI, M. Learning from Examples to Improve Code Completion Systems. In: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. New York, NY, USA: ACM, 2009. (ESEC/FSE '09), p. 213–222. ISBN 978-1-60558-001-2. Disponível em: <http://doi.acm.org/10.1145/1595696.1595728>.

BRUCH, M.; SCHäFER, T.; MEZINI, M. FrUiT: IDE Support for Framework Understanding. In: Proceedings of the 2006 OOPSLA Workshop on Eclipse

Technology eXchange. New York, NY, USA: ACM, 2006. (eclipse '06), p. 55–59. ISBN 1-59593-621-1. Disponível em: <http://doi.acm.org/10.1145/1188835.1188847>.

CHEN, F.; ROşU, G. Java-MOP: A Monitoring Oriented Programming Environment for Java. In: HALBWACHS, N.; ZUCK, L. (Ed.). Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2005, (Lecture Notes in Computer Science, v. 3440). p. 546–550. ISBN 978-3-540-25333-4. Disponível em: <http://dx.doi.org/10.1007/978-3-540-31980-1_36>.

CHEN, H.; WAGNER, D.; DEAN, D. Setuid Demystified. In: Proceedings of the 11th USENIX Security Symposium. Berkeley, CA, USA: USENIX Association, 2002. p. 171–190. ISBN 1-931971-00-5. Disponível em: <http://dl.acm.org/citation.cfm?id=647253.720278>.

CLARKE, E. M.; WING, J. M. Formal Methods: State of the Art and Future Directions. ACM Comput. Surv., ACM, New York, NY, USA, v. 28, n. 4, p. 626–643, dez. 1996. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/242223.242257>.

DALLMEIER, V. et al. Automatically Generating Test Cases for Specification Mining. Software Engineering, IEEE Transactions on, IEEE, v. 38, n. 2, p. 243–257, 2012.

ENGLER, D. et al. Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles. New York, NY, USA: ACM, 2001. (SOSP '01), p. 57–72. ISBN 1-58113-389-8. Disponível em: <http://doi.acm.org/10.1145/502034.502041>.

ERNST, M. D. et al. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In: Proceedings of the 21st International Conference on Software Engineering. New York, NY, USA: ACM, 1999. (ICSE '99), p. 213–224. ISBN 1-58113-074-0. Disponível em: <http://doi.acm.org/10.1145/302405.302467>.

FLANAGAN, C.; LEINO, K. R. M. Houdini, an Annotation Assistant for ESC/Java. In: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity. London, UK, UK: Springer-Verlag, 2001. (FME '01), p. 500–517. ISBN 3-540-41791-5. Disponível em: <http://dl.acm.org/citation.cfm?id=647540.730008>.

GABEL, M.; SU, Z. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York, NY, USA: ACM, 2008. (SIGSOFT '08/FSE-16), p. 339–349. ISBN 978-1-59593-995-1. Disponível em: <http://doi.acm.org/10.1145/1453101.1453150>.

GHAFARI, M. et al. Mining Unit Tests for Code Recommendation. In: Proceedings of the 22Nd International Conference on Program Comprehension. New York, NY, USA: ACM, 2014. (ICPC 2014), p. 142–145. ISBN 978-1-4503-2879-1. Disponível em: <http://doi.acm.org/10.1145/2597008.2597789>.

GOUES, C. L.; WEIMER, W. Specification Mining with Few False Positives. In: KOWALEWSKI, S.; PHILIPPOU, A. (Ed.). Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2009,

(Lecture Notes in Computer Science, v. 5505). p. 292–306. ISBN 978-3-642-00767-5. Disponível em: <http://dx.doi.org/10.1007/978-3-642-00768-2_26>.

GOUES, C. L.; WEIMER, W. Measuring Code Quality to Improve Specification Mining. IEEE Trans. Softw. Eng., IEEE Press, Piscataway, NJ, USA, v. 38, n. 1, p. 175–190, jan. 2012. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/TSE.2011.5>.

GUTTAG, J.; HORNING, J. The algebraic specification of abstract data types. Acta Informatica, Springer-Verlag, v. 10, n. 1, p. 27–52, 1978. ISSN 0001-5903. Disponível em: <http://dx.doi.org/10.1007/BF00260922>.

HAN, J.; KAMBER, M. Data Mining, Southeast Asia Edition: Concepts and Techniques. San Francisco, CA, USA: Morgan Kaufmann, 2006.

HENKEL, J.; DIWAN, A. Discovering Algebraic Specifications from Java Classes. In: CARDELLI, L. (Ed.). ECOOP 2003 – Object-Oriented Programming. Springer Berlin Heidelberg, 2003, (Lecture Notes in Computer Science, v. 2743). p. 431–456. ISBN 978-3-540-40531-3. Disponível em: <http://dx.doi.org/10.1007/978-3-540-45070-2_19>.

LAMSWEERDE, A. v. Formal Specification: A Roadmap. In: Proceedings of the Conference on The Future of Software Engineering. New York, NY, USA: ACM, 2000. (ICSE '00), p. 147–159. ISBN 1-58113-253-0. Disponível em: <http://doi.acm.org/10.1145/336512.336546>.

LEE, C.; CHEN, F.; ROsU, G. Mining Parametric Specifications. In: Proceedings of the 33rd International Conference on Software Engineering. New York, NY, USA: ACM, 2011. (ICSE '11), p. 591–600. ISBN 978-1-4503-0445-0. Disponível em: <http://doi.acm.org/10.1145/1985793.1985874>.

LEE, C. et al. Towards Categorizing and Formalizing the JDK API. [S.l.], March 2012. Disponível em: <https://www.ideals.illinois.edu/handle/2142/30006>.

LERNER, S. et al. Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules. In: Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA: ACM, 2005. (POPL '05), p. 364–377. ISBN 1-58113-830-X. Disponível em: <http://doi.acm.org/10.1145/1040305.1040335>.

LO, D. Specification mining: Methodologies, theories and applications. 2008. Tese (Doutorado) — National University of Singapore, 2008.

MEREDITH, P. O. et al. An overview of the MOP runtime verification framework. International Journal on Software Tools for Technology Transfer, Springer, v. 14, n. 3, p. 249–289, 2012.

MEYER, B. Applying "Design by Contract". Computer, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 25, n. 10, p. 40–51, 1992. ISSN 0018-9162.

MØLLER, A. dk.brics.automaton – Finite-State Automata and Regular Expressions for Java. 2010. http://www.brics.dk/automaton/.

MONPERRUS, M.; BRUCH, M.; MEZINI, M. Detecting Missing Method Calls in Object-Oriented Software. In: D'HONDT, T. (Ed.). ECOOP 2010 – Object-Oriented Programming. Springer Berlin Heidelberg, 2010, (Lecture Notes in Computer Science, v. 6183). p. 2–25. ISBN 978-3-642-14106-5. Disponível em: <http://dx.doi.org/10.1007/978-3-642-14107-2_2>.

NEVILL-MANNING, C. G.; WITTEN, I. H. Identifying Hierarchical Structure in Sequences: A Linear-time Algorithm. J. Artif. Int. Res., AI Access Foundation, USA, v. 7, n. 1, p. 67–82, set. 1997. ISSN 1076-9757. Disponível em: <http://dl.acm.org/citation.cfm?id=1622776.1622780>.

Oracle Corporation. Open JDK 6 b33. 2011. Disponível em: <http://hg.openjdk.java.net/jdk6/jdk6>. Acesso em: 29/12/2014.

PACHECO, C.; ERNST, M. D. Randoop: Feedback-directed Random Testing for Java. In: Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion. New York, NY, USA: ACM, 2007. (OOPSLA '07), p. 815–816. ISBN 978-1-59593-865-7. Disponível em: <http://doi.acm.org/10.1145/1297846.1297902>.

PRADEL, M.; BICHSEL, P.; GROSS, T. R. A framework for the evaluation of specification miners based on finite state machines. In: 26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania. IEEE Computer Society, 2010. p. 1–10. Disponível em: <http://dx.doi.org/10.1109/ICSM.2010.5609576>.

PRADEL, M.; GROSS, T. R. Automatic Generation of Object Usage Specifications from Large Method Traces. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. Washington, DC, USA: IEEE Computer Society, 2009. (ASE '09), p. 371–382. ISBN 978-0-7695-3891-4. Disponível em: <http://dx.doi.org/10.1109/ASE.2009.60>.

PRADEL, M.; GROSS, T. R. Leveraging Test Generation and Specification Mining for Automated Bug Detection Without False Positives. In: Proceedings of the 34th International Conference on Software Engineering. Piscataway, NJ, USA: IEEE Press, 2012. (ICSE '12), p. 288–298. ISBN 978-1-4673-1067-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2337223.2337258>.

RAMANATHAN, M. K.; GRAMA, A.; JAGANNATHAN, S. Static Specification Inference Using Predicate Mining. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA: ACM, 2007. (PLDI '07), p. 123–134. ISBN 978-1-59593-633-2. Disponível em: <http://doi.acm.org/10.1145/1250734.1250749>.

REGER, G.; BARRINGER, H.; RYDEHEARD, D. E. A pattern-based approach to parametric specification mining. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013. IEEE, 2013. p. 658–663. Disponível em: <http://dx.doi.org/10.1109/ASE.2013.6693129>.

ROBILLARD, M. P. et al. Automated API Property Inference Techniques. IEEE Trans. Softw. Eng., IEEE Press, Piscataway, NJ, USA, v. 39, n. 5, p. 613–637, maio 2013. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/TSE.2012.63>.

SANTHIAR, A.; PANDITA, O.; KANADE, A. Discovering Math APIs by Mining Unit Tests. In: CORTELLESSA, V.; VARRó, D. (Ed.). Fundamental Approaches to Software Engineering. Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 7793). p. 327–342. ISBN 978-3-642-37056-4. Disponível em: <http://dx.doi.org/10.1007/978-3-642-37057-1_24>.

SOMMERVILLE, I. Software Engineering. 9. ed. Harlow, England: Addison-Wesley, 2010. ISBN 978-0-13-703515-1.

SRIDHARAN, M. et al. Alias Analysis for Object-Oriented Programs. In: CLARKE, D.; NOBLE, J.; WRIGSTAD, T. (Ed.). Aliasing in Object-Oriented Programming. Types, Analysis and Verification. Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 7850). p. 196–232. ISBN 978-3-642-36945-2. Disponível em: <http://dx.doi.org/10.1007/978-3-642-36946-9_8>.

TILLMANN, N.; SCHULTE, W. Parameterized Unit Tests. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York, NY, USA: ACM, 2005. (ESEC/FSE-13), p. 253–262. ISBN 1-59593-014-0. Disponível em: <http://doi.acm.org/10.1145/1081706.1081749>.

WEIMER, W.; NECULA, G. C. Mining Temporal Specifications for Error Detection. In: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer-Verlag, 2005, (TACAS'05). p. 461–476. ISBN 3-540-25333-5, 978-3-540-25333-4. Disponível em: <http://dx.doi.org/10.1007/978-3-540-31980-1_30>.

WOODCOCK, J. et al. Formal Methods: Practice and Experience. ACM Comput. Surv., ACM, New York, NY, USA, v. 41, n. 4, p. 19:1–19:36, out. 2009. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/1592434.1592436>.

WU, Q. et al. Iterative Mining of Resource-releasing Specifications. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. Washington, DC, USA: IEEE Computer Society, 2011. (ASE '11), p. 233–242. ISBN 978-1-4577-1638-6. Disponível em: <http://dx.doi.org/10.1109/ASE.2011.6100058>.