Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Uma Abordagem para Classificar Bugs sob a Perspectiva de Máquina de Estados

Melquisedec Albert Einstein de Andrade Lima

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Desenvolvimento de Software Dirigido à Modelo

Franklin de Souza Ramalho
(Orientador)

Campina Grande, Paraíba, Brasil

UMA ABORDAGEM PARA CLASSIFICAR BUGS SOB A PERSPECTIVA DE MÁQUINA DE ESTADOS

MELQUISEDEC ALBERT EINSTEIN DE ANDRADE LIMA

DISSERTAÇÃO APROVADA EM 09/12/2019

FRANKLIN DE SOUZA RAMALHO, Dr., UFCG
Orientador(a)

EVERTON LEANDRO GALDINO ALVES, Dr., UFCG
Examinador(a)

PAULO EDUARDO E SILVA BARBOSA, Dr., UEPB
Examinador(a)

CAMPINA GRANDE - PB

L732a Lima, Melquisedec Albert Einstein de Andrade.
Uma abordagem para classificar bugs sob a perspectiva de máquina de estados / Melquisedec Albert Einstein de Andrade Lima. – Campina Grande, 2020.
103 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2020.
"Orientação: Prof. Dr. Franklin de Souza Ramalho".
Referências.

1. Linguagem de Programação. 2. Máquina de Estados. 3. BUG. 4. Classificação. I. Ramalho, Franklin de Souza. II. Título.

CDU 004.42(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA SEVERINA SUELI DA SILVA OLIVEIRA CRB-15/225

Resumo

Dentre os modelos abordados pela linguagem UML, existe o modelo de máquina de estados, que permite modelar o comportamento de um objeto de software isolado, mostrando de forma explícita como o objeto responde a estímulos externos a ele. Quando se utiliza o modelo de máquina de estados em um projeto de software, o código-fonte é seu reflexo, de tal forma que um erro do software pode estar diretamente relacionado com a sua máquina de estados. Identificar a causa de uma falha relacionada ao modelo de máquina de estados nem sempre é uma tarefa trivial, podendo muitas vezes passar pelo processo de avaliação de falha, identificar elementos do software que são impactados pelo erro, determinar a sua causa e corrigir o software. Identificar as causas de uma falha é o processo que mais consome tempo no ciclo de vida do software. Existem trabalhos que abordam técnicas de como localizar bugs reportados no código-fonte mas não encontramos nenhum trabalho que relacione ou classifique bugs de acordo com máquina de estados. A tarefa de identificar a qual elemento da máquina de estados um bug reportado está relacionado é muito laboriosa. Diante desse problema, o objetivo desta pesquisa foi desenvolver uma técnica de classificação automática de bugs relacionados a máquina de estados. A técnica proposta é composta por uma taxonomia de bugs sob a perspectiva de máquina de estados e um algoritmo para classificação automática de bugs. Para avaliar a técnica proposta, realizamos um survey para avaliar a taxonomia e um experimento quantitativo em projetos de software reais para avaliar o algoritmo classificador. Como resultado verificamos que: (*i)* a taxonomia atende a demanda, com número de categorias e clareza adequadas para o seu objetivo; (*ii*) o algoritmo para classificação automática obteve um desempenho de 80% de *precision* e 80% de *recall* utilizando o algoritmo de aprendizagem de máquina KNN.

Abstract

Among the models approached by the UML language, there is the state machine model, which allows modeling the behavior of an isolated software object, showing explicitly how the object responds to external stimulus. When using the state machine model in a software project, the source code reflects the state machine such that a software error may be directly related to the state machine. Identifying the cause of an error related to the state machine model is not always a trivial task and can often go through the error assessment process, identify software elements that impacted by the error, determine its cause, and correct the software. Identifying the causes of a failure is the most time-consuming process in the software life cycle. There are works that address techniques for finding reported bugs in source code, but we have not found any work that lists or classifies bugs according to state machine. The task of identifying which state machine element a reported bug relates to is very laborious. Given this problem, the objective of this research was to develop a technique of automatic classification of bugs related to state machines. The proposed technique is composed of a bug taxonomy from the state machine perspective and an algorithm for automatic bug classification. To evaluate the proposed technique, we performed a survey to evaluate the taxonomy and a quantitative experiment in real software projects to evaluate the classifying algorithm. As result, we found that: *(i)* taxonomy meets demand, with a number of categories and clarity adequate for its purpose; (*ii*) the algorithm for automatic classification achieved a performance of 80% *precision* and 80% *recall* using the KNN machine learning algorithm.

Agradecimentos

Agradeço primeiramente a minha família, aos meus país (Melquisedec e Madalena) pela educação que recebi e pelo caminho que sigo hoje. Sou o que sou principalmente por causa deles, e por isso, sou grato. A minha namorada (Jacqueline) que esteve presente, me apoiou e me motivou a continuar mesmo que sem saber. Sua presença foi fundamental durante essa jornada. O meu orientador (Franklin) que tenho grande estima e admiração. Sou grato pela orientação que recebi, pela paciência que ele sempre teve, e pela sua vontade de sempre querer ajudar. Nas mais adversas situações, mesmo atarefado de coisas para fazer, ele sempre estava lá para ajudar um aluno com uma dúvida ou outra, para orientar Matheus e eu. Muito obrigado, professor. À Matheus (Matheuzim), por me acompanhar durante toda a pós-graduação, nos bons e maus momentos, na alegria e no desespero. Aos meus queridos amigos do SPLab por me proporcionar momentos de alegria e boas conversas na hora do café. Agradeço a Alysson e Indy pela companhia e ajuda que me deram diversas vezes e pelas risadas proporcionadas. Aos professores e demais funcionários da UFCG que contribuíram para minha formação como aluno. À coordenação de Aperfeiçoamento de Pessoal de nível Superior (CAPES), pelo apoio financeiro.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

O processo de desenvolvimento de software vem evoluindo desde do surgimento do software, várias técnicas e formas de desenvolver software surgiram e sucumbiram na história da computação. Durante a década de 1980, com o aumento da complexidade e da exigência por softwares com qualidade em um prazo de tempo menor, as técnicas e metodologias de desenvolvimento da época não acompanhavam mais a demanda, o que levou à crise do software [2].

Para atender à demanda, uma série de metodologias, técnicas de desenvolvimento, paradigmas de linguagem de programação, e conceitos surgiram [1]. Muitos autores geravam conhecimento semelhantes entre si para o desenvolvimento de software, estendendo ou acrescentando alguma pequena melhoria para algum conceito de desenvolvimento de software, de tal forma que muitos dos conceitos e metodologias de software elaborados na época só possuíam diferença entre nomenclaturas de termos e definições. Diante desse cenário surgiu um movimento dos principais pesquisadores em metodologia de software da época para unificar todo esse conhecimento de forma metódica [1]. Esse movimento deu origem à linguagem UML (*Unified Modeling Language*) [3], uma linguagem de modelagem de software dotada de uma série de 13 modelos para representar os diferentes aspectos do software.

Os modelos UML são classificados em modelos do tipo estrutural e do tipo dinâmico [4]. Os modelos do tipo estrutural definem os conceitos-chave do software, suas propriedades, e as relações entre os conceitos, um exemplo de modelo estrutural é o diagrama de classes [1]. Já os modelos do tipo dinâmico modelam o comportamento de objetos, podendo modelar: (*i*) o histórico de vida da interação de um único elemento do software com os demais, (*ii*)

a execução de uma sequência de atividades para descrever um processo, (*iii*) modelar o padrão de comunicação de um conjunto de objetos relacionados para mostrar como os objetos interagem para implementar um comportamento do software [1].

Os modelos do tipo dinâmico na UML 2.5 [1] são 4: máquina de estados, diagrama de sequência, diagrama de comunicação e diagrama de atividade. O modelo máquina de estado permite modelar o comportamento de um objeto isolado, explicitando como o objeto responde a eventos externos a ele. As respostas do objeto podem ser modeladas, a grosso modo, como mudança/permanência do seu estado atual, ou execução de ações. Quando se utiliza o modelo máquina de estados em um projeto de software, o código-fonte é reflexo da máquina de estados, de tal forma que um erro do software pode estar diretamente relacionado a sua máquina de estados. Dentre outras causas, o erro de software pode ser proveniente de: (*i*) má implementação da máquina de estados (o código não reflete a máquina) ou (*ii*) resultante de uma modelagem errada (a máquina não reflete o que o software deveria fazer). Este trabalho aborda o problema (*i*), ou seja, consideramos que a máquina de estados do software está especificada corretamente.

O processo de software, como definido por Sommerville, consiste no "conjunto de atividades relacionadas que levam à produção de um produto de software"[4]. Todo processo de software deve incluir no mínimo as atividades de especificação do software, implementação, validação, e evolução. A forma como um processo de software é aplicado pode ser utilizada como indicativo de qualidade do produto de software, existindo para isso métricas específicas de qualidade do software baseadas no processo de software [5].

Todas as atividades do processo de software estão sujeitas ao fator "erro humano". As principais causas que levam o software a apresentar falhas são: (*i*) definição de metas irrealísticas do projeto, (*ii*) imprecisão na estimativa dos recursos necessários, (*iii*) mau levantamento de requisitos [6]. Um exemplo clássico da importância de estabelecer um processo de software de qualidade é o da companhia *London Stock Exchange*, onde um erro no design de software do projeto e o péssimo gerenciamento do projeto custou à empresa $600 milhões [6]. Outro exemplo é o da companhia *Sydney Water Corp*, a maior companhia de fornecimento de água da Austrália, que teve um prejuízo de $32 milhões em um projeto para automatizar o sistema de cobranças. O projeto foi cancelado após a alegação de planejamento inadequado e taxa de mudança de requisitos alta [6].

Dentre todas as atividades do processo de software existe a atividade de *debugging*. *Debugging* ou *debug* é o processo de reproduzir falhas[1] avaliar gravidade, risco, e regiões impactadas; identificar a causa da falha; e corrigir o software analisado [4]. Identificar as causas de uma falha é a etapa que mais consome tempo, porque o time de desenvolvimento precisa identificar dentre o espaço de estados do software, o estado específico em que ocorreu a falha, o que implica tipicamente em analisar diversas variáveis [8]. Segundo Britton et al. (2013) a tarefa de *debug* custa em média 50% do custo total do desenvolvimento de software [9]. Um exemplo clássico da dificuldade em identificar as causas de falhas em software é descobrir se a causa da falha está no código fonte ou na definição dos requisitos do software.

1.1 Problema

Existem vários trabalhos que propõem técnicas para ajudar na identificação de *bugs* [10, 11, 12], tanto na fase de desenvolvimento de software, como na fase de manutenção durante o processo de *debug*. Sobre identificar as causas de falhas em software, existem algumas abordagens já consolidadas e dentre elas se destacam a análise estática e a análise dinâmica [12].

Técnicas de análise dos aspectos comportamentais do software são amplamente utilizadas para detectar anomalias no funcionamento de software. Na literatura, as técnicas de análise de aspectos comportamentais realizam a detecção de *bugs*, e comparam, por exemplo o modelo de máquinas de estados que foi implementado com o modelo de software da sua especificação [10, 13, 14, 15, 16], contudo elas não consideram *bugs* que foram encontrados após a fase de implantação do software ou seja, *bugs* que foram reportados ou pelo usuário final ou encontrados pela equipe de desenvolvimento durante a fase de manutenção do software. Um bug encontrado após a fase de implantação é um bug mais custoso para se tratar e geralmente possuir um texto em linguagem natural descrevendo o problema. Diferentemente de um bug encontrado antes da fase de implantação, onde sua descrição é mais objetiva e com texto escrito em linguagem mais técnica, podendo citar até possíveis ele-

[1]Em nosso trabalho utilizamos a terminologia do IEEE [7] para erro, falha e falta. Falta: também conhecido como bug ou defeito, uma implementação incorreta dos requisitos do software, resultante de ação humana. Erro: um estado incorreto do sistema decorrente de uma falta. Falha: Comportamento não esperado do software, decorrente de um erro.

mentos do código impactados pelo bug. Dessa forma, o bug encontrando após implantação geralmente precisa de mais análise, sendo mais custoso para resolver.

Para ilustrar essa situação vamos considerar que o seguinte bug tenha sido reportado: "Na janela de mensagens, o sistema notifica apenas o usuário que deu a permissão de passar". Assumindo que na especificação desse sistema, todos os clientes deveriam receber uma notificação, temos um problema pois apenas o usuário que deu a permissão foi notificado. Vamos considerar ainda que a causa desse bug esteja no código, ou seja, má implementação da máquina de estados. A máquina de estados desse sistema é representada pela Figura 1.1 e consiste de dois estados (*Estado 1* e *Estado 2*), o sistema muda de *Estado 1* para *Estado 2* quando a condição de guarda *podePassar* for atendida. Somente quando o sistema atinge o estado *Estado 2* é que a ação */notificarTodos* é executada.

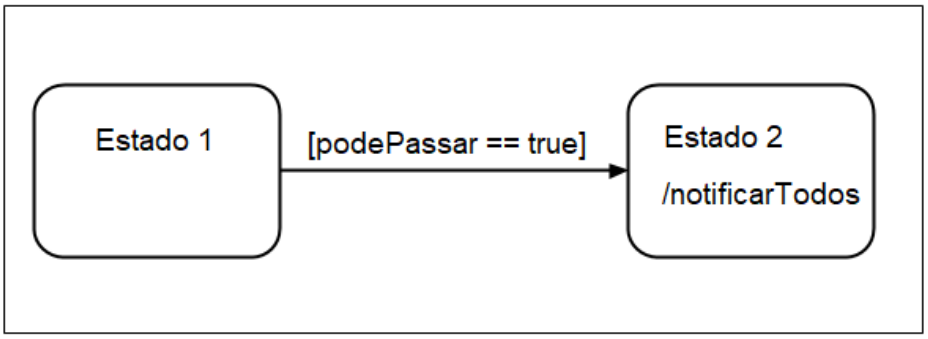


Figura 1.1: Exemplo de máquina de estados.

Para resolver esse bug o(a) desenvolvedor(a) irá analisar o código-fonte e a documentação do software, o que pode ser muito custoso. Automatizar a análise é algo útil e que facilitaria o trabalho do desenvolvedor. O problema de automatizar a análise pode ser dividido em dois subproblemas: (1) automatizar a análise do código-fonte, (2) automatizar a análise da documentação.

O problema que abordaremos neste trabalho é: *A tarefa de identificar se elementos da máquina de estados estão ou não relacionados a um bug reportado é muito laboriosa*. Perde-se muito tempo procurando-se a causa de bugs que estejam relacionados a máquina de estados. O problema da máquina de estados estar mal modelada não é abordado por este trabalho. Decidimos abordar especificamente com máquina de estados por identificarmos o problema em um sistema real de forma recorrente. A partir dessa constatação pesquisamos por outros

projetos e identificamos que o problema ocorrendo em projetos *opensource*.

1.2 Objetivos

O objetivo desta pesquisa é desenvolver uma técnica de classificação automática de *bugs* que estão relacionados à máquina de estados. Os objetivos específicos desta pesquisa consistem em:

- Criar uma taxonomia para classificação de bugs sob a perspectiva de máquina de estados;
- Desenvolver algoritmos para classificação automática de bugs na taxonomia elaborada utilizando recuperação da informação e aprendizagem de máquina;
- Analisar desempenho dos algoritmos para identificar o mais adequado para o contexto desta pesquisa.

Para classificar os *bugs* relacionados a máquina de estados, criamos uma taxonomia de *bugs*, onde cada categoria representa uma classe de bugs associada a um elemento da máquina de estado. A taxonomia tem como objetivo delimitar as causas do bug. Assim o bug classificado na taxonomia tem uma maior chance de ter suas causas delimitadas, fazendo com que a tarefa de *debug* seja menos custosa para o desenvolvedor(a).

Para realizar a classificação de *bugs*, a técnica de classificação de bugs proposta neste trabalho utiliza o algoritmo KNN de aprendizagem de máquina [17] e recuperação da informação [18].

1.2.1 Técnica de classificação automática de bugs

A técnica proposta classifica *bugs* sob a perspectiva de máquina de estados, para isso a técnica utiliza a ferramenta de recuperação da informação Apache Lucene [19] e a biblioteca de aprendizado de máquina *Scikitlearn* [20].

Para avaliarmos a eficiência da técnica, coletamos bugs reportados de projetos de software abertos que continham máquina de estados e utilizamos as métricas *precision* [21] e

recall [21]. Os resultados obtidos mostram que a técnica é eficiente para classificar bugs relacionados a máquina de estados, apresentando desempenho de pelo menos 80% de *precision* nos projetos analisados.

1.3 Relevância

O uso da abordagem proposta pode reduzir o tempo de *debug* de projetos que utilizem máquina estados, reduzindo assim o custo da atividade de manutenção de software, que corresponde a 50% do custo total de software [4]. Um dos motivos que levam a tarefa de manutenção de software a ser tão custosa, diz respeito às tarefas de *debug*, onde é preciso realizar análise do bug, verificar regiões impactadas, determinar a causa do bug e realizar a correção.

Existem várias configurações em que a máquina de estados pode ser má implementada no código. Quando o código não reflete o comportamento descrito pela máquina, existe uma violação do modelo. Essa violação pode ser difícil de identificar, principalmente para quem não está familiarizado com a máquina de estados. Uma taxonomia que realize o mapeamento dessas violações também ajuda na tarefa de *debug*, apontando as possíveis configurações da máquina de estados que causou o erro do software.

A verificação manual da taxonomia que faz o mapeamento das violações em máquina de estados também é uma tarefa que pode demandar muito tempo, pois exige que seja analisado cada cenário da taxonomia para verificar em quais deles um determinado bug melhor se encaixa. Sendo essa tarefa feita de forma automatizada, temos assim um meio rápido para auxiliar a identificar a causa de bugs relacionados a máquina de estados, com a possibilidade de reduzir o custo da tarefa de *debugging* para bugs relacionados a máquina de estados.

Além de auxiliar no processo de *debugging*, a técnica proposta pode auxiliar na compreensão do funcionamento do software. Diante da complexidade que um sistema pode ter, somente observar sua máquina de estados pode não ser o suficiente para sua compreensão. Tal situação pode acontecer principalmente com desenvolvedores com pouca experiência profissional ou recém chegado ao projeto. A técnica proposta pode auxiliar nesse sentindo expondo cenários de violações da máquina de estados que podem estar relacionados um determinado bug, fazendo por exemplo que o desenvolvedor analise o funcionamento correto

da máquina de estados e o confronte com o problema reportado no bug. Ao final o desenvolvedor poderá ter um melhor entendimento do funcionamento da máquina de estados do sistema e consequentemente do funcionamento de uma parte do sistema em si.

1.4 Escopo

Este trabalho aplica a técnica de classificação automática de bugs para projetos de software que disponibilizem o acesso à documentação e relatório de *bugs* do software. Na ferramenta onde implementamos a técnica proposta, os arquivos de entrada devem estar no formato CSV (*Comma Separated Value*) [22].

Na taxonomia de *bugs* também proposta neste trabalho, contemplamos os seguintes elementos de máquina de estados para a classificação: *state, transition, guard, internal/external activity, entry, exit, join, fork*, e *choice*, que são os elementos mais utilizados para modelagem em máquina de estado [23]. A taxonomia proposta não contempla os seguintes elementos: *shallowHistory, deepHistory, junction* e *terminate* devido ao baixo uso desses elementos na documentação de softwares [23], e por não termos encontrado nas bases de dados consultadas nenhuma documentação que contemplasse esses elementos.

Para verificar a eficiência da técnica utilizamos relatórios de *bugs* de projetos disponíveis na plataforma aberta *Github* [2].

A técnica de classificação automática de bugs necessita de um conjunto de treinamento para poder classificar os bugs de forma eficaz. Assim, é preciso um conjunto de bugs previamente classificados, do projeto de software a ser analisado, para utilizar a técnica.

1.5 Organização do documento

Os capítulos a seguir desta dissertação estão organizados da seguinte forma:

Capitulo 2: Fundamentação teórica. Apresenta os conceitos necessários para melhor compreender o contexto e proposta deste trabalho. Os conceitos abordados são: *Máquina de estados UML* e *Aprendizado de máquina*.

Capítulo 3: Técnica proposta. Apresenta a técnica de classificação automática de bugs

[2] www.github.com

relacionados à máquina de estados. A técnica é constituída pelos elementos: a taxonomia para classificação de bugs e o algoritmo classificador. Este capítulo descreve os elementos da técnica e a relação entre eles.

Capítulo 4: Avaliação. Descreve como a técnica de classificação foi avaliada, ou seja, como a taxonomia e o algoritmo classificador foram avaliados. Além disso, esse capítulo apresenta e discute os resultados dos experimentos realizados.

Capítulo 5: Trabalhos relacionados. Apresenta o resultado da revisão da literatura de trabalhos que relacionam máquina de estados e *debug*, expondo as limitações e relações desses trabalhos com a técnica de classificação automática de bugs proposta neste trabalho.

Capitulo 6: Conclusões. Apresenta as conclusões obtidas por esse trabalho diante dos resultados alcançados pelos experimentos, expõe as limitações observadas e apresenta sugestões de trabalhos futuros. Vamos utilizar os dados de pacientes que já têm o diagnóstico correto para *inferir* o diagnóstico de um novo paciente. Esse paciente é o que se encontra na quinta linha da Tabela 2.1. Para isso, primeiramente devemos calcular a distância do conjunto de características entre o paciente sem diagnóstico e cada paciente do conjunto treinamento. Seja P o conjunto de

Capítulo 2

Fundamentação Teórica

Para melhor compreender a técnica de classificação automática de bugs sob a perspectiva de máquina de estados, se faz necessário antes, introduzir os conceitos de máquina de estados, e de bug reportado (relatório de bug). Além disso, é preciso compreender alguns dos princípios de Aprendizagem de máquina pois tais princípios são utilizados pela técnica durante as etapas de análise e classificação de bugs.

2.1 Máquinas de Estados UML

Máquina de estados é um modelo UML [3] que descreve o comportamento de um único objeto, ou seja, máquina de estado descreve como um objeto muda sua configuração em resposta a eventos que ocorrem no meio ao qual o objeto está inserido.

Tipicamente, uma máquina de estados (*state machine* ou SM) é composta por estados e transições. Estado representa uma configuração do objeto em um momento específico. Um estado possui um nome que o descreve. Uma transição é um arco que possui um estado como *fonte* e um estado como *destino*. A transição representa a mudança entre estados.

Além de estados e transições, uma máquina de estados pode possuir ações associadas aos estados. As ações são classificadas em: (*i*) *entry actions*, que são executadas assim que o sistema atinge o estado que a contenha; (*ii*) *do actions* que são executadas após todas as *entry actions* enquanto o sistema permanecer no estado; (*iii*) *exit actions* que são executadas imediatamente antes do sistema sair do estado.

A transição pode conter um evento, uma condição e ações. Um evento é um estímulo

que dispara a transição. Após a ocorrência do evento, caso exista condição de guarda, esta deve ser satisfeita para que o sistema mude do estado *fonte* para o estado *destino*. Por fim, atividades são executadas antes do sistema atingir o estado *destino* da transição.

Os elementos dos tipos *estado* e *transição* são ditos elementos concretos, de tal forma que existem elementos que não são concretos, estes são chamados de elementos abstratos, e representam estados temporários em uma máquina de estados [3]. São os elementos definidos como *Pseudo-estados*. O grupo de elementos do tipo pseudo-estado é um sub-conjunto do conjunto de estados e é composto por elementos do tipo: *deepHistory*, *shallowHistory*, *join*, *fork*, *junction*, *choice*, *entryPoint*, *exitPoint*, e *terminate* [3].

Para ilustrar o funcionamento de máquina de estados, consideremos o exemplo de Rumbaugh *et.al* [1], onde uma máquina de estados descreve o comportamento de um painel secreto em uma sala [1]. Os autores descrevem o contexto: "[...] neste castelo, eu quero guardar meus tesouros em um cofre que seja difícil de encontrar. Então para revelar a fechadura do cofre, eu preciso remover uma vela estratégica de seu candelabro, mas isso apenas irá revelar a fechadura do cofre enquanto a porta do recinto estiver fechada" [1, pp. 104]. Uma vez que o controlador revela a fechadura, é possível inserir a chave para abrir o cofre, contudo existe uma medida extra de segurança, o cofre deve ser aberto apenas se a vela for re-inserida no candelabro. Caso contrário, o controlador solta um monstro para devorar o ladrão negligente. A Figura 2.1 mostra a máquina de estados do controlador do cofre.

A máquina de estados da Figura 2.1 possui três estados: *Wait*, *Lock*, e *Open*. Após remover a vela do seu candelabro o controlador deve verificar se a porta está fechada, esse comportamento é modelado pela guarda *doorClosed*, que se verdadeira, revela o painel secreto da fechadura. A ação de revelar a fechadura corresponde a ação */revealLock*. O evento *candleRemoved* corresponde ao disparo do evento que muda o estado do sistema de *Wait* para *Lock*.

Uma vez o controlador está no estado *Lock*, existem duas possibilidades de estado para ir após o disparo do evento *keyTurned*: *Open* ou o estado final. Embora um mesmo evento dispare duas transições diferentes, as guardas para cada um dos estados são mutualmente excludentes, indicando assim qual transição será disparada. Se a guarda *candleIn* for satisfeita o controlador executa a ação */openSafe* e muda o estado do sistema para *Open*, contudo se a guarda *candleOut* for satisfeita, o controlador executa a ação */releaseKillerRabbit* e a

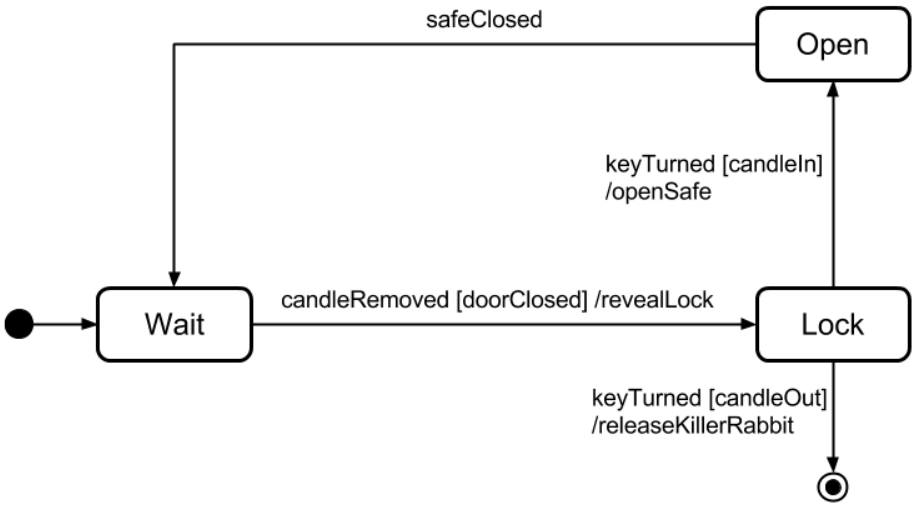


Figura 2.1: Máquina de estados do controlador para o painel secreto no castelo. Fonte: Rumbaugh *et.al.* (2004, pp. 104). [1]

máquina de estados alcança o estado final, indicando que o controlador deve ser reiniciado. Uma vez que o controlador está no estado *Open*, apenas o disparo do evento *safeClosed* que pode alterar o estado do controlador, que muda para o estado *Wait*.

Uma transição em máquinas de estado pode ter múltiplos eventos, condições de guarda e ações. Para garantir a correta execução de todos esses elementos, a máquina de estados possui uma ordem de prioridade. Essa ordem consiste em: Disparo de eventos, guardas, ações [3]. Considerando o exemplo do controlador do painel secreto em seu estado inicial, primeiramente o evento *candleRemoved* dispara e só depois disso que o controlador verifica se guarda *doorClosed* foi satisfeita, e somente se a guarda foi satisfeita é que a ação *revealLock* ocorre.

2.2 Aprendizagem de Máquina

Aprendizagem de máquina é uma área da inteligência artificial e é definida como o processo de extrair informação útil de uma massa de dados de forma automatizada [17]. Por informação útil entende-se informação que possua valor agregado e que possa ajudar na tomada de decisões. Empresas como a *Amazon Inc*, *Facebook* e *Google* [24] utilizam aprendizagem de máquina para proporcionar uma melhor experiência aos seus consumidores e aperfeiçoar

seus produtos.

O processo de extração de informação tem como objetivo levantar hipóteses sobre os dados analisados. As hipóteses levantadas são basicamente de dois tipos:

- Previsão: Considerando um histórico de dados com determinadas características, prever as características de um novo dado;
- Descrição: Explorar ou descrever um conjunto de dados.

Na previsão, os elementos da análise são tratados como um conjunto de entrada e um conjunto de saída, a técnica de aprendizagem de máquina recebe um conjunto de dados D e devolve uma saída S bem definida. Já na descrição, os elementos da análise não possuem uma saída bem definida [25].

Figura 2.2: Tipos de aprendizado.

Cada tipo de hipótese origina diversas técnicas de aprendizado de máquina. A aprendizagem de máquina em si pode ser vista como uma forma de aprendizado do tipo indutivo (se o dado possui tais características, então pode-se concluir x sobre o dado). O aprendizado indutivo é categorizado da forma como apresentado na Figura 2.2.

Os aprendizados do tipo *supervisionados* compreendem os que tratam as hipóteses do tipo *previsão*. Nesse tipo de aprendizado já conhecemos previamente as possíveis saídas do aprendizado. O aprendizado supervisionado pode ainda ser categorizado como *classificação* e *regressão*. Na classificação, a saída assume valores discretos (por exemplo: classificar se o filme é bom ou ruim). Na regressão a saída assume valores contínuos (por exemplo: qual o

valor de um imóvel daqui a 10 anos) e podem ser tratadas em um preditor contínuo (exemplo: regressão linear), ou categórico (exemplo: regressão logística).

Já os aprendizados do tipo *não supervisionados* compreendem os que tratam as hipóteses do tipo *descrição*. Nesse tipo de aprendizado, não conhecemos as possíveis saídas. O aprendizado não supervisionado pode ser categorizado em: sumarização, associação e agrupamento. No agrupamento os dados são agrupados de acordo com a similaridade (por exemplo, grupos de cliente similares). A sumarização consiste em encontrar uma descrição compacta para os dados. Por fim, a associação consiste em encontrar padrões frequentes de associações entre atributos.

Cada categoria (classificação, regressão, sumarização, associação e agrupamento) corresponde a uma família de algoritmos de aprendizagem de máquina, onde cada algoritmo aplica diferentes abordagens e configurações para extrair informações relevantes de uma massa de dados. O algoritmo de aprendizagem de máquina utilizado na abordagem desta dissertação foi o KNN (*K-Nearest Neighbors*), um algoritmo do tipo classificador e supervisionado.

O KNN classifica um elemento baseado em exemplos de outros elementos previamente classificados. Esse conjunto de elementos classificados é dito *conjunto treinamento*. O algoritmo realiza um cálculo para determinar a similaridade do elemento a ser classificado, com os elementos do conjunto treinamento. É um algoritmo reconhecidamente eficiente por trabalhar diretamente com informações corretas. O algoritmo consiste em:

1. Calcular a distância entre o exemplo desconhecido e os outros exemplos do conjunto de treinamento;

2. Identificar os K vizinhos mais próximos;

3. Utilizar o rótulo da classe dos vizinhos mais próximos para determinar o rótulo da classe do exemplo desconhecido, através de votação majoritária.

O cálculo da distância entre elementos é comumente feito utilizando-se a distância euclidiana, onde a distância entre elementos é dada pela equação:

$$d(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

Onde *d* é a distância a ser calculada, *p* é o elemento a ser classificado, *q* é um elemento do conjunto treinamento, *x* e *y* são características do elementos. Para uma melhor compreensão

do cálculo da distância, vamos considerar o exemplo fictício da Tabela 2.1.

Tabela 2.1: Exemplo KNN para diagnosticar paciente com problema no coração (Tabela elaborada pelo autor, todos os dados são fictícios).

Idade	Sexo	PAS	PAD	Peso	Fuma?	Bebe?	Dor no peito	Diagnóstico
54	M	130	85	92	0	1	1	1
30	F	110	78	53	0	0	0	0
52	M	131	87	84	1	1	1	1
41	M	120	80	78	0	0	1	0
45	M	120	81	76	0	1	0	???

Neste exemplo, temos dados de pacientes: idade, sexo, pressão arterial sistólica (PAS), pressão arterial diastólica (PAD), peso, se o paciente fuma, bebe, sente dor no peito, e o diagnóstico acusando ou não problema no coração. Na tabela o valor 0 tem significado negativo (não) e o valor 1 tem significado positivo (sim). Cada coluna representa uma informação do paciente, sendo o conjunto de colunas denominado *conjunto de características*.

Para o conjunto de características dados no exemplo, podemos levantar a seguinte hipótese:

- O conjunto de características do paciente está diretamente relacionado com o seu diagnóstico.

Para fins meramente ilustrativos, assumimos que a hipótese é verdadeira. O objetivo do algoritmo será então determinar o diagnóstico para problema no coração, de acordo com as características do paciente. O diagnóstico é a saída do algoritmo, e o conjunto de características é a entrada. O diagnóstico pode assumir dois valores: o paciente tem problema no coração, ou o paciente não tem problema no coração. Os valores que a saída pode assumir são denominados *rótulos*.

Vamos utilizar os dados de pacientes que já têm o diagnóstico correto para *inferir* o diagnóstico de um novo paciente. Esse paciente é o que se encontra na quinta linha da Tabela 2.1. Para isso, primeiramente devemos calcular a distância do conjunto de características entre o paciente sem diagnóstico e cada paciente do conjunto treinamento. Seja P o conjunto

de pacientes, sendo *p1*, *p2*, *p3*, *p4*, os pacientes do conjunto treinamento, e *p5* o paciente sem diagnostico, a distância euclidiana será calculada entre *p5* e cada elemento do conjunto treinamento. A Figura 2.3 contém o cálculo e valor das distâncias calculadas.

$\sqrt{(-9)^2+0^2+(-10)^2+(-4)^2+(-16)^2+0^2+0^2+(-1)^2}$	14
$\sqrt{15^2+10^2+3^2+23^2+0^2+1^2+0^2}$	29
$\sqrt{(-7)^2+0^2+10^2+(-4)^2+(-16)^2+0^2+0^2+(-1)^2}$	20
$\sqrt{4^2+0^2+0^2+1^2+(-2)^2+0^2+1^2+(-1)^2}$	4

Figura 2.3: Distância euclidiana entre características de pacientes. Fonte: Própria. (2019).

Para realizar esse cálculo a característica *sexo* foi excluída por questão de conveniência para o exemplo, pois como a característica *sexo* é do tipo qualitativa, se faz necessário primeiramente mapear essa característica para valores numéricos, verificar a influencia de cada sexo no resultado, para que assim fosse possível atribuir valores a cada um dos sexos. As características *Fuma?*, *Bebe?* e *Dor no peito* também são qualitativas mas com apenas dois valores possíveis (sim e não) podendo esses valores serem mapeados para "0"e "1"de forma direta.

Com a distância calculada o algoritmo KNN segue para determinar o rótulo da entrada. Para isso ele ranqueia as K menores distâncias calculadas, e através de votação majoritária determina que a classificação do elemento analisado é igual a classificação da maioria dos elementos com menor distância do conjunto treinamento.

Observando o exemplo anterior, vemos que para $K = 1$ temos que a menor distância é o valor 4, que corresponde ao paciente *p4* da Tabela 2.1, portanto o paciente *p5* seria classificado com a mesma classificação de *4* (não possui problema no coração). Para $K = 3$ temos as distâncias (4, 14, 20), que correspondem a *p4*, *p1* e *p3* respectivamente. Nesse caso, *p1* e *p3* são classificados como “possui problema no coração”, enquanto *p4* é classificado como “não possui problema no coração”. De acordo com a votação majoritária, a maioria dos elementos é classificado como “possui problema no coração”, portanto dessa vez *p5* é classificado dessa forma.

A classificação de um mesmo elemento pode mudar com a variação de *K*, a medida que a

maioria dos elementos mais próximos está propícia à mudança na classificação. A Figura 2.4 ilustra essa questão apresentando um elemento no centro que deseja-se classificar de acordo com a sua forma geométrica.

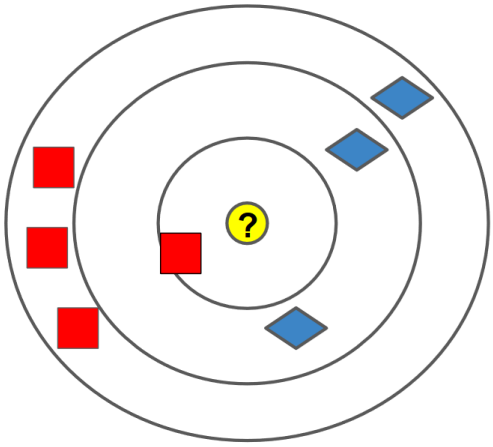


Figura 2.4: Variação da classificação do algoritmo KNN.

Seja o elemento central denominado c, ao seu redor temos outros elementos de diferentes tipos e distâncias. O elemento mais próximo de c é um quadrado. Dessa forma, para classificar c, utilizando o algoritmo KNN e considerando $K = 1$, temos que c é classificado como quadrado. Já para para $K = 3$ temos dois losangos e um quadrado mais próximos de c. Por maioria de votos, c é classificado como losango nesse caso. Para $K = 7$ temos novamente c classificado como quadrado. O valor de K geralmente é um número impar para não haver risco de empates na votação.

Capítulo 3

Técnica proposta

A técnica proposta é constituída por dois elementos: (*i*) taxonomia para classificação de bugs sob a perspectiva de máquina de estados e (*ii*) algoritmo classificador de bug. A taxonomia contempla um conjunto de elementos de máquina de estados UML, e os possíveis cenários onde podem ocorrer falhas relacionadas com máquina de estados. O algoritmo classificador utiliza técnicas de recuperação da informação [18] e aprendizado de máquina [26] para classificar um dado bug de acordo com a taxonomia. Este capítulo irá detalhar sobre a técnica proposta. A Seção 3.1 discorre sobre a taxonomia de bugs, enquanto a Seção 3.2 discorre sobre o algoritmo classificador e seu funcionamento.

3.1 Taxonomia para classificação de bugs

O objetivo de classificar bugs é ajudar na tarefa de depuração (*debugging*) de forma que, a classificação de um bug revele características de sua causa, tornando assim as tarefas de identificação das causas do bug e sua correção mais fáceis. Trabalhos anteriores já trataram de classificação de bugs com diferentes objetivos, como por exemplo [27, 28]. Contundo não encontramos nenhum trabalho propondo classificar bugs de acordo com o modelo de máquina de estados de software.

Com a finalidade de permitir a classificação automática de bugs de acordo com comportamentos especificados em máquina de estados, nossa técnica provê uma taxonomia de bugs que diz respeito a discrepâncias entre especificação da máquina de estados e sua implementação no código-fonte, bem como em quais e cenários tais discrepâncias podem ocorrer.

Esta seção foi dividida em duas sub-seções. Na sub-seção 3.1.1 apresentamos a taxonomia proposta neste trabalho, suas características e exemplos de uso. Na sub-seção 3.1.2 apresentamos o processo de construção da taxonomia, exibindo questões que foram surgindo durante o seu processo de avaliação.

3.1.1 Taxonomia para classificação de bugs

Cada categoria da taxonomia diz respeito aos principais elementos de máquina de estados UML [3], são eles: estado, guarda, transição, atividades internas, ações, *choice*, *fork*, e *join*. Esses não são todos tipos de elementos de máquina de estados, e sim os elementos mais utilizados na modelagem de software [23]. Assim sendo, a taxonomia e consequentemente a técnica de classificação de bugs, não contempla os elementos: *shallowhistory*, *deephistory*, *junction* e *terminate*. Decidimos não contemplar esses elementos por não encontrar nas bases de dados consultadas as especificações de máquina de estados que os utilizassem. Os projetos de software que encontramos modelam o comportamento desejado sem utilizar tais elementos. Juntamente com esse fato, Liu et al. [23] expõe que o formalismo UML para máquina de estados [3] por muitas vezes dificulta a modelagem de software, como consequência os modelos são adaptados, ou não utilizados com o mesmo rigor do formalismo UML, assim são poucos os casos onde uma especificação de máquina de estados vai precisar de fato de todos os elementos que a UML dispõe.

Para classificar um bug que está relacionado a máquina de estados, é preciso identificar em qual ou quais elemento(s) da máquina de estados o bug ocorreu. Por exemplo, considere um bug *b* onde a sua descrição relata que não é possível executar um dado comportamento modelado na máquina de estados. Assumindo que a especificação da máquina de estados está correta, duas possíveis causas do bug *b* são: (*i*) a máquina de estados, incorretamente, não alcança o estado que contém o comportamento reportado em *b*; ou (*ii*) a máquina de estados alcança o estado que contém o dado comportamento, mas falha em executá-lo. Podem existir ainda outras razões que levaram a esse bug, de forma que, a tarefa de identificar se a causa do bug está em (*i*), em (*ii*) ou em algum outro elemento relacionado à especificação da máquina de estados, é uma tarefa difícil e que consome muito tempo.

No exemplo do bug *b* observamos que a sua causa embora esteja relacionada à máquina de estados, existem ainda nuances como as descritas em (*i*) e (*ii*) que apontam para problemas

Tabela 3.1: Resumo da taxonomia de bugs relacionados com máquina de estados

Classificação	Número de cenários
Bug on state	5
Bug on guard	2
Bug on activity	5
Bug on entry	3
Bug on exit	2
Bug on join	2
Bug on fork	2
Bug on choice	3

distintos, e que levam a resoluções diferentes da causa do bug. Por exemplo, em (*i*) a pessoa que está corrigindo o problema pode pensar em verificar eventos de transição, funções que alteram o estado do sistema, etc. Enquanto em (*ii*) ele/ela vai se preocupar em um estado específico, funções que executam ações naquele estado, etc.

Bugs do tipo mencionado acima, são violações de comportamentos modelados na máquina de estados e estão relacionadas a elementos (por exemplo: estado, transição, ações, etc.) Mesmo identificando a qual elemento um bug (como o *b*) está relacionado, existem nuances como as descritas em (*i*) e (*ii*) que determinam qual a causa do bug. Identificamos e catalogamos tais nuances dos elementos de máquina de estados, e as rotulamos como *cenários*. Dessa forma a taxonomia é capaz de determinar a que tipo de elemento da máquina de estados um dado bug pertence, e em qual o cenário o bug viola a máquina de estados.

Neste trabalho propomos a taxonomia de bugs relacionados aos elementos de máquina de estados apresentados na Tabela 3.1, onde cada linha da coluna "Classificação"se refere a um elemento da máquina de estados, enquanto a outra coluna diz respeito ao número de cenários mapeados para a categoria da taxonomia de bugs. A taxonomia é o primeiro passo na técnica proposta, para identificar e corrigir bugs relacionados a máquina de estados de maneira mais simples e rápida, economizando tempo e esforço.

Para ilustrar como um bug pode ocorrer em um mesmo elemento mas em cenários distintos, vamos considerar o exemplo da Figura 3.1. Nela, temos uma máquina de estados que consiste de dois estados (*Estado 1* e *Estado 2*), uma condição de guarda onde o atri-

buto *podePassar* deve ser igual a *Verdadeiro* para a transição ser executada, e temos ainda a ação *notificarTodos* que é executada quando o estado do sistema alcança o *Estado 2*. Considerando que um bug foi reportado e que ele esteja relacionado à condição de guarda, os cenários possíveis onde o bug pode ocorrer são os seguintes: (1) a condição de guarda é satisfeita e a máquina de estados não muda de *Estado 1* para *Estado 2*, (2) a condição de guarda não é satisfeita e a máquina de estados muda de *Estado 1* para *Estado 2*.

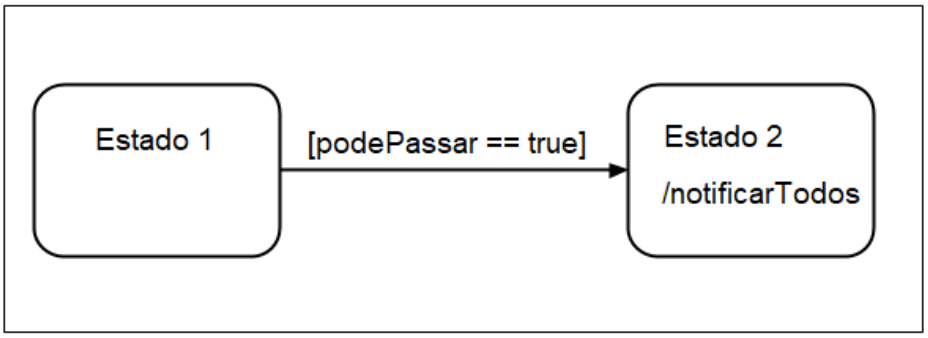


Figura 3.1: Exemplo de máquina de estados.

Em máquina de estados UML, cada tipo de elemento (*estado*, *transição*, *ação*) possui um conjunto de regras e configurações bem definidos. Quando um elemento desse não obedece uma regra ou configuração, temos então uma violação ao funcionamento correto de máquina de estados. A violação à máquina de estados pode acontecer dentre outras causas, por má implementação da máquina no código-fonte. A técnica proposta aborda somente problemas da má implementação da máquina de estados no código-fonte. A técnica não analisa se existe algum problema de especificação da máquina de estados e é destinada para qualquer pessoa que for realizar a tarefa de *debugging*.

Os cenários na classificação de *bugs* correspondem à combinação das configurações que violam o comportamento esperado da máquina de estados, durante a ocorrência do bug. Por exemplo, na Figura 3.1, a condição de guarda pode assumir o valor *Verdadeiro* ou *Falso*, então temos quatro cenários possíveis:

1. A condição de guarda é satisfeita e a máquina de estados não muda do *Estado 1* para *Estado 2*;

2. A condição de guarda não é satisfeita e a máquina de estados muda do *Estado 1* para *Estado 2*;

3. A condição de guarda é satisfeita e a máquina de estados muda do *Estado 1* para *Estado 2*;

4. A condição de guarda não é satisfeita e a máquina de estados não muda do *Estado 1* para *Estado 2*;

Os cenários 3 e 4 são comportamentos esperados de uma máquina de estados, logo não são situações configuradas como bug. Por outro lado, os cenários 1 e 2 são falhas que violam a especificação de máquina de estados e por isso devem ser consideradas *bugs*.

Além disso, a violação da máquina de estados pode resultar em um comportamento inesperado do sistema. Os cenários 1 e 2, por exemplo, possuem um efeito colateral. No cenário 1 a atividade interna *notificarTodos* não é executada pois a máquina de estados não alcança *Estado 2*. Já na combinação 2, *notificarTodos* é executado embora a condição *podePassar* não seja satisfeita. O bug geralmente é reportado pelo usuário final e em linguagem natural, sem termos/jargões próprios da área de TI. Uma possibilidade de descrição de bug reportado seria: "O sistema não executa a ação *notificarTodos* quando deveria". A causa desse bug pode estar na mudança de estado que não acontece como ocorre no cenário 1.

O número de comportamentos inesperados (cenários) pode crescer rapidamente dependendo do número de valores que os elementos da máquina de estados podem assumir, o que pode resultar em uma taxonomia muito grande para quem for analisar o bug reportado, fazendo com que a classificação do bug reportado não ajude o desenvolvedor a encontrar a causa do bug.

Adicionalmente, a taxonomia de *bugs* deve classificar um bug de acordo com sua causa, então decidimos utilizar uma abordagem do tipo *overlapping*[1] para classificar os *bugs* com o intuito de simplificar a informação para o usuário e facilitar a tarefa de *debugging*. Essa taxonomia possibilita ao usuário a noção sobre o tipo de violação que ocorreu na máquina de estado, permitindo que o bug seja manipulado de forma mais eficiente do que uma taxonomia com todos os cenários possíveis. No exemplo anterior, a taxonomia deve classificar o bug como *Bug on State* e *Bug on Activity*, pois a execução da ação de fato não ocorreu, sendo assim um problema categorizado em *Bug on Activity*, mas o motivo da ação não ter sido executada foi porque o sistema não alcançou o estado seguinte, assim sendo, o problema

também é classificado em *Bug on State*.

Na Figura 3.2, encontra-se a taxonomia proposta completa, com os cenários para cada classe de bug.

Classificação	Sigla	Cenários
Bug on State	BS	Evento dispara, estado não muda
		Evento não dispara, estado muda
		Guarda satisfeita, estado não muda
		Evento dispara, guarda satisfeita, estado não muda
		Evento dispara, guarda não satisfeita, estado muda
Bug on Guard	BG	Evento não dispara, guarda é satisfeita, atividade da transição é executada, estado muda
		Evento não dispara, guarda não é satisfeita, atividade da transição é executada, estado muda
Bug on Activity	BA	Atividade interna não é executada
		Atividade da transição não executada
		atividade de transição é executada quando não deveria ser
		Evento dispara, atividade de transição não é executada
		Evento não dispara, atividade de transição é executada
Bug on Entry	BE	Estado é alcançado, entry action não executa
		Estado não é alcançado, entry action executa
		Estado é alcançado, entry action executa, mas não na ordem esperada
Bug on Exit	BX	Saída do estado e exit action não executa
		Exit action executa e estado não muda
Bug on Join	BJ	Todas as transições de chegada foram completadas, transição de saída não executa
		Pelo menos uma transição de chegada não foi completada, transição de saída executa
Bug on Fork	BF	A transição de chegada foi completada, pelo menos uma das transições de saída não é executada
		A transição de chegada não foi completada, pelo menos uma das transições de saída é executada
Bug on Choice	BC	A transição de chegada foi completada, a guarda foi satisteita, transição esperada não ocorre
		A transição de chegada não foi completada, a guarda foi satisteita, pelo menos uma das transições de saída é executada
		A transição de chegada foi completada, o fluxo deveria serguir o else, a transição de saída do else não é executada

Figura 3.2: Classificação de *bugs*.

Decidimos que cada categoria da taxonomia deve contemplar um tipo de elemento da

[1]Em uma abordagem do tipo *overlapping* as classificações não são tratadas como um conjunto disjunto, ou seja, não são mutualmente excludentes, podendo um bug ser classificado em mais de uma categoria. O motivo de se utilizar essa abordagem é verificar todas as possibilidades de classificação de um dado bug e identificar qual classificação que é a mais adequada.

máquina de estados. Chegamos a essa decisão após analisar seis amostras de máquina de estados de do projeto ePol. Percebemos que o que ajuda o depurador a corrigir bugs relacionados a máquina de estados, é identificar qual o elemento da máquina de estados está relacionado com o bug (por exemplo: o bug está relacionado com o estado *x*? Com a ação *a*?). O estado específico varia conforme cada projeto de software, assim generalizamos a taxonomia, onde cada categoria se refere a um tipo de elemento de máquina de estados (por exemplo: estado, ação, evento).

O processo de construção da taxonomia consistiu em: (*i*) levantar todas as combinações de violações de elementos de máquina de estados, originando os cenários; (*ii*) categorizar cada cenário elencado em (*i*); (*iii*) eliminar cenários redundantes; (*iv*) avaliar taxonomia e ajustar a taxonomia. Esse processo resultou em três versões, as versões iniciais foram avaliadas e ajustadas, resultando na versão final da taxonomia. A avaliação e ajuste da taxonomia teve como objetivo de facilitar a tarefa de depuração de bugs relacionados a máquina de estados. Em cada etapa balanceamos o número de cenários e categorias da taxonomia, considerando que: (*i*) um maior número de cenários aumentaria sua granularidade e dificultaria a tarefa de correção de bugs, pois o usuário teria que verificar um grande número de cenários para identificar em qual o bug se encaixa; (*ii*) um menor número de cenários diminuiria a granularidade da taxonomia, mas o cenário seria tão genérico que poderia não ajudar em nada o usuário. Foi considerando esses dois extremos que balanceamos e chegamos à taxonomia ilustrada na Figura 3.2, que foi avaliada através de questionário. Mais detalhes sobre o procedimento de avaliação encontra-se no Capítulo 4.

Alguns cenários podem parecer redundantes à primeira vista. O cenário *I* da categoria *Bug on State* é o mais abrangente da categoria, podendo englobar por exemplo o cenário seguinte (Guarda satisfeita, estado não muda), contudo é importante lembrar que todos os elementos de uma transição são opcionais, podendo existir casos em que a máquina de estados contenha apenas condições de guardas em suas transições, sem nenhum evento associado a elas. Essa é a razão de termos os cenário *I* e o cenário *III*, assim como os cenários *II* e *IV*.

3.1.2 Processo de construção da taxonomia para classificação de bugs

A primeira versão da taxonomia possuía sete categorias e 38 cenários. Nessa versão não existia a categoria *bug on guard* e a granularidade de seus cenários era alta. Os cenários

eram a combinação simples de valores dos comportamentos que os elementos da máquina de estados poderiam assumir, de forma que, esses comportamentos violavam a semântica de máquina de estados UML [3]. Por exemplo, uma transição que não dispara quando deveria, uma *entry action* que não é executada quando a máquina de estados alcança determinado estado, etc. Contudo, a simples combinação de valores gerou cenários redundantes, onde um cenário bem específico já era coberto por outro cenário mais genérico, ou um cenário de uma categoria era coberto por outro pertencente a uma categoria distinta. Para melhor compreensão desses problemas, analisemos a Tabela 3.2, que apresenta uma amostra de classificações e cenários da versão inicial da taxonomia.

Tabela 3.2: Amostra de classificações e cenários da primeira versão da taxonomia de bugs

Classificação	Cenarios
Bug on State	I - Evento dispara, estado não muda
	II - Evento não dispara, estado muda
	III - Guarda satisfeita, estado não muda
	IV - Guarda não satisfeita, estado muda
	V - Evento não dispara, guarda satisfeita, estado muda
	VI - Evento não dispara, atividade de transição não é executada, estado muda
Bug on Acvity	VII - Atividade interna não executada, quando deveria ser
	VIII - Atividade da transição não executada
	IX - Evento dispara, atividade de transição não é executada

Na Tabela 3.2 temos alguns cenários das categorias *Bug on State* e *Bug on Activity* na primeira versão da taxonomia. A ideia inicial para construir os cenários foi pensar em cada violação de comportamento dos elementos da máquina de estados. Assim sendo, para determinar os cenários da categoria *Bug on State*, por exemplo, catalogamos uma série de cenários com violações a comportamentos de máquina de estados, englobando elementos do tipo *estado* (*e.g* o estado do sistema não muda).

Os dois últimos cenários da categoria *Bug on State* (*V* e *VI*) são derivados do cenário *II*, com a adição de condição de guarda em *V* e uma atividade em *VI*. A informação mais importante nesses cenários é o fato de que o evento que habilita a transição não ocorreu e mesmo assim a máquina mudou de estado, já que estamos tratando de cenários da categoria

Bug on State. O fato de uma atividade de transição ser executada ou não nesse cenário é uma violação que se encaixa melhor na categoria *Bug on Guard*. A mesma linha de raciocínio é aplicada para a guarda em *V*. Além disso, preocupação da atividade de transição não ser executada em *VI* é contemplado pelo cenário *VIII* da categoria *Bug on Activity*. Assim o cenário *VI* foi removido. O mesmo se aplica aos cenários *VIII* e *IX*: o *VIII* engloba o *IX* sem que haja nenhuma informação relevante em *IX*, assim *IX* foi removido.

3.2 Algoritmo classificador

Para realizar a classificação de *bugs*, propomos a técnica de classificação automática de bugs. Essa técnica mescla princípios de recuperação de informação e aprendizado de máquina. No total, propomos e avaliamos o desempenho de três algoritmos classificadores: o puramente baseado em recuperação da informação, o puramente baseado em aprendizado de máquina e o baseado em aprendizado de máquina com pré-processamento da entrada usando recuperação da informação, sendo esse último uma combinação dos dois anteriores. Todos os algoritmos propostos foram implementados para serem avaliados e estão disponíveis em plataforma *opensource* [1].

Os seguintes algoritmos são contribuições desse trabalho: (*i*) o algoritmo puramente baseado em recuperação da informação, (*ii*) o algoritmo puramente baseado em aprendizado de máquina e (*iii*) o algoritmo baseado em aprendizagem de máquina com pré-processamento da entrada. Não foi escopo desse trabalho implementar e propor novos algoritmos de recuperação da informação e de aprendizagem de máquina. Utilizamos algoritmos já consolidados e prontos de recuperação da informação (disponibilizado pela biblioteca Lucene [19]) e de aprendizagem de máquina (disponibilizado pela biblioteca Scikit [20]). O algoritmo puramente baseado em recuperação da informação se mostrou pouco efetivo para classificar bugs. Os algoritmos baseados em aprendizagem de máquina tiveram desempenho equivalente, não havendo diferenças estatísticas significativas entre eles e tiveram desempenho superior em relação ao puramente baseado em recuperação da informação. A técnica proposta por nesse trabalho utiliza o algoritmo puramente baseado em aprendizado de máquina.

Observamos que o desempenho do algoritmo puramente baseado em recuperação da in-

[1] https://github.com/melqui-andrade/automaticBugClassification

formação não foi satisfatório para nosso contexto. Os algoritmos baseados em aprendizado de máquina tiveram desempenho equivalente, não havendo diferença estatística entre o desempenho deles. Em relação do desempenho do algoritmo puramente baseado em recuperação da informação, o desempenho dos algoritmos baseados em aprendizado de máquina foi superior, se mostrando mais eficazes.

3.2.1 Algoritmo puramente baseado em recuperação da informação

Com o objetivo de classificar de forma automática bugs de acordo com a taxonomia da seção anterior, elaboramos um primeiro algoritmo totalmente baseado em recuperação da informação (*IR-based*) e o implementamos em Java [29], utilizando a biblioteca Lucene [19] que já dispunha das técnicas de recuperação da informação implementadas e prontas para uso. Recuperação da informação (*Information Retrieval - IR*) é, como definido por Baeza e Ribeiro [18] , "Um campo da ciência da computação que lida com o armazenamento e recuperação de documentos de forma automatizada" (Baeza e Ribeiro [18, pp. 1], tradução nossa).

A ferramenta desenvolvida que implementa esse algoritmo recebe como entrada as máquinas de estados do software a ser analisado e um conjunto de bugs. As máquina de estados devem estar em conformidade com o modelo UML [3] e estar no formato XMI [30]. Desde que a máquina de estados esteja no formato XMI [30] e seguindo as definições UML [3], não importa a origem do arquivo, se foi extraída a partir de engenharia reversa, se foi modelada. Não é o foco da ferramenta se preocupar em como esse arquivo é gerado, desde que ele seja fornecido, a análise será feita.

Para utilizar qualquer ferramenta de recuperação da informação é preciso definir como a informação será representada. O modelo mais utilizado para isso é o modelo vetorial [18], onde cada termo dos documentos que queremos recuperar é indexado juntamente com a sua relevância para determinado documento. A relevância do termo é determinada basicamente pela razão da frequência em que o termo aparece em um documento específico pelo número de documentos que contém o termo [18].

O algoritmo *IR-based* recebe como entrada um relatório de bugs, *BR* (contendo uma lista de bugs reportados) e um conjunto de máquinas de estados *SM* do projeto de software a ser analisado. A saída do algoritmo é um conjunto de *4-uplas* (*b, ie, c, sc*), onde *b* é um

bug reportado de *BR*, *ie* é o conjunto de elementos das máquinas de estados de *SM* que estão relacionados a *b* (por exemplo: um estado *s*, uma transição *t*, etc.), *c* é o conjunto de categorias em que o bug foi classificado e *sc* são os cenários das categorias de *c*.

A categoria do bug é identificada a partir do cálculo de similaridade entre os elementos do relatório de bugs - precisamente, o título e descrição de cada bug (escritos em linguagem natural) - e os elementos da máquina de estados que, como explicado na Seção 3.1, são: *state*, *transition*, *guard*, *internal/external activity*, *entry*, *exit*, *join*, *fork* and *choice*.

Para realizar a classificação, primeiramente o algoritmo agrupa os elementos de *SM* em três grupos iniciais:

- *StateElements* - Contendo os elementos que são do tipo *state*;
- *PseudoStateElements* - Contendo os elementos do tipo *pseudo-state*: *fork*, *join* and *choice*;
- *TransitionElements* - Contendo os elementos que são do tipo *transition*.

Todo elemento de *SM* pertence a um dos grupos acima. Por exemplo: um *event*, um *guard* ou uma *transition action* estão sempre em uma transição, logo pertecem ao grupo *TransitionElements*. Já *entry*, *exit* e *do actions* ocorrem dentro de um estado, logo pertencem ao grupo *StateElements*.

Dessa forma, primeiro o algoritmo itera sobre todos os elementos de *SM* com o objetivo de separar os elementos nos grupos citados. Depois disso, o algoritmo analisa a relação entre cada bug e os elementos agrupados, como é indicado no Código-Fonte 3.1.

Código Fonte 3.1: Iterações do algoritmo para casar bugs e elementos de SM

```
1 groupAllElements(BR, SM)
2 for each bug b from BR, do:
3       for each transition t in SM.Transitions, do:
4         analyzeTransitions(b, t)
5       for each state s in SM.States, do:
6         analyzeStates(b, s)
7       for each pseudoState ps in SM.PseudoStates, do:
8         analyzePseudoStates(b, ps)
```

Como cada grupo possui elementos com características distintas, é preciso utilizar três funções para tratar cada grupo. Sendo assim, no Código-Fonte 3.1, *analyzeTransitions* (linha 4) trata os elementos de transição, *analyzeState* (linha 6) trata elementos de estado, e *analyzePseudoStates* (linha 8) trata elementos do tipo pseudo-estados. Essas funções são responsáveis por: (*i*) calcular a similaridade entre os termos de um dado bug e os elementos dos grupos; (*ii*) ranquear os *n* primeiros elementos *SM* de acordo com a similaridade calculada em (*i*); e (*iii*) classificar o bug de acordo com as categorias da taxonomia da Seção 3.1. Cada função segue uma sequência de passos própria para classificar os bugs. Como a técnica de classificação de bugs usa a abordagem *overlaping*, o mesmo bug passa por cada função com o objetivo de verificar se ele é classificável em categorias dos elementos de cada grupo, assim um bug pode ser classificado em mais de uma categoria.

A função *analyzeState* é detalhada no Código-Fonte 3.2. Essa função recebe como entrada um bug e um estado. Sua heurística procura casar elementos textuais do bug (título e descrição) com um ou mais elementos do grupo *StateElements* (*internal activities*, *entry actions*, ou *exit actions*). A função recupera todas as atividades internas do estado na linha 2, todas as *entry actions* na linha 9, *exit actions* na linha 16 e por fim o estado em si na linha 23 utilizando uma função *match* que aplica técnicas de *IR*.

Código Fonte 3.2: Pseudo código da função *analyzeState*

```
1  define analyzeState(bug b, state s){
2      define MIntActivities such that MIntActivities = match(b,
3      s.internalActivities);
4      if (MIntActivities.size > 0) then:
5              define bugClass as BugClassification:
6                  bugClass.bugID = b.identification;
7                  bugClass.classification = BugOnActivity;
8                  bugClass.elements = s.internalActivities;
9              Add bugClass on foundRelations;
10     define MentryActions such that MentryActions = match(b, s.entries);
11     if (MentryActions.size > 0) then:
12             define bugClass as BugClassification:
13                 bugClass.bugID = b.identification;
14                 bugClass.classification = BugOnEntry;
15                 bugClass.elements = s.entry;
```

```
16              Add bugClass on foundRelations;
17      define MexitActions such that MexitActions = match(b, s.exit);
18      if (Mexit.size > 0) then:
19              define bugClass as BugClassification:
20                  bugClass.bugID = b.identification;
21                  bugClass.classification = BugOnExit;
22                  bugClass.elements = s.exit;
23              Add bugClass on foundRelations;
24      Define Mstates such that Mstates = match(b, s);
25      If (Mstates.size > 0) then:
26            Define bug as BugClassification:
27                  bugClass.bugID = b.identification;
28                  bugClass.classification = BugOnState;
29                  bugClass.elements = s.toString;
30              Add bugClass on foundRelations;
31  }
```

Sempre que acontece pelo menos um casamento entre bug e um elemento de *StateElements*, uma classificação de bug é criada. Na classificação são armazenados o identificador do bug (para localizar o bug), a categoria da classificação, e os elementos da *SM* relacionados. Por exemplo, as linhas 4-7 do Código-Fonte 3.2 criam uma classificação de bug, sempre que pelo menos uma atividade interna casa com os termos do bug. Uma vez criada, a classificação do bug é adicionada a uma lista (*foundRelations*) para gerar um relatório de classificação de bugs (linhas 8, 15, 22 e 29).

De forma análoga, as funções que tratam *entry actions* (linhas 10-15) e *exit actions* (linhas 24-29) geram classificações de bugs para *entry actions* e *exit actions* respectivamente. No caso de ocorrer o casamento entre o bug e estado propriamente dito (linhas 24-29) o bug é classificado como *BugOnState*.

É importante notar que um bug pode ser classificado em mais de uma categoria. Por exemplo, um bug que relata um problema em alguma atividade interna de um estado pode ser classificado como *Bug on State* e *Bug on Activity*, não havendo nenhuma prioridade entre as classificações na técnica proposta.

As funções *analyzeTranstions* e *analyzePseudoStates* do Código-Fonte 3.1 funcionam de forma análoga. Em particular, a função *analyzeTranstions* trata de analisar os elementos do

tipo *guards*, *actions* e da transição em si para classificar (ou não) o bug em *BugOnGuard*, *BugOnActivity* ou *BugOnTransition*, respectivamente. A função *analyzePseudoStates* trata dos pseudo-estados *forks*, *join* e *choice* para classificar (ou não) o bug nas categorias de *BugOnFork*, *BugOnJoin* ou *BugOnChoice* respectivamente.

É importante enfatizar como a função *match* funciona, já que ela é utilizada em vários momentos no algoritmo (linhas 2, 9, 16, 23) e aplica técnicas de *IR*. Basicamente, essa função calcula a similaridade entre um bug *b* e um conjunto de elementos de máquina de estados *S*, através de *IR*. A similaridade entre elementos é determinada em função da frequência de termos (*tf*) semelhantes e o inverso dos termos nos documentos (*idf*). O *tf-idf* determina a similaridade entre bugs e elementos da máquina de estados contando a frequência em que cada termo dos elementos aparecem em cada descrição dos bugs, bem como em quantos bugs distintos cada termo aparece [19]. Além de calcular a similaridade, a função ranqueia os *n* elementos mais similares da entrada dada. Essa função retorna uma lista ordenada de elementos da máquina de estados. A ordem é dada de acordo com o grau de similaridade (por exemplo: elemento na ordem "1" foi o elemento classificado com maior grau de similaridade). Para isso a função utilizou um conjunto de técnicas *IR* já implementadas por uma biblioteca externa, o Lucene [19].

A função *match* adota *tf-idf* para indexar tanto bug (título + descrição), como elementos de máquina de estados. Ambos são representados em um modelo vetorial [18] e sua similaridade é calculada como o cosseno entre os dois vetores (bug e elemento SM). A função *match* também aplica técnicas de *IR* da seguinte forma: (*i*) quebra o texto em palavras (*tokenization*); (*ii*) remove sufixos e prefixos das palavras (*stemming*); e (*iii*) remove palavras irrelevantes (*stopwords*) como conjunções e preposições.

O Código-Fonte 3.3 mostra a função *match*, utilizada para fazer o casamento de todos os elementos processados. Como entrada, a função recebe um bug *b* e um conjunto de elementos de máquina de estados E. A *query* que fará a consulta de quais elementos estão relacionados ao bug *b* é definida na linha 2, e é composta pelo texto do título e descrição de *b*. Nas linhas 3 e 4 definimos o *index* onde iremos executar as consultas da *query q*, de forma que na linha 4 definimos que o processo a ser utilizado para indexar vai ser o executado pelo *StadardAnalyzer*, analisador já implementado no Lucene [19], que realiza as tarefas de *tokenization*, *stemming*, e remoção de *stop-words*. Os elementos de máquina de

estados são indexados nas linhas 5-9. A indexação ocorre sob a descrição do elemento e da máquina de estados ao qual o elemento pertence. Finalmente, a consulta é executada na linha 14, onde *scoreDocs* recebe os *n* melhores resultados da pesquisa, de acordo com o grau de similaridade.

Código Fonte 3.3: Pseudo-código da função *match*

```
1  define match(bug b, elements[] E){
2      define q as Query such that q = b.title + b.description;
3      define index as Index, where:
4          index.analizer = StandardAnalyzer;
5      for each element e in E, do:
6          define doc as Document;
7          doc.add(e.description);
8          doc.add(e.stateMachineID);
9          index.addDocument(doc);
10     define searcher as IndexSearcher where:
11         searcher.index = index;
12         searcher.query = q;
13         searcher.hitsPerPage = n;
14     define scoreDocs as searcher.run;
15     return scoreDocs.docs;
16 }
```

Embora a comparação de *strings* funcione para a maioria dos elementos de *SM*, essa abordagem não é eficiente para analisar condições de guarda, pois a falha de uma guarda é reportada em linguagem natural, que geralmente não corresponde aos termos da condição de guarda da especificação da máquina de estados. Para ilustrar essa situação, suponha o seguinte bug reportado, por um membro da equipe de desenvolvimento:

> Bug: verificadorPorta muda de *Aguardando* para *Fechado*, embora o status da porta seja "aberta".

Assumindo que a transição do estado *Aguardando* para o estado *Fechado* possui uma condição de guarda que é verdadeira quando *status* é igual a "aberta", somente comparação de *strings* não consegue relacionar a condição de guarda ao status reportado no bug. Os elementos textuais da condição de guarda não serão similares a descrição do bug, conse-

quentemente a função não conseguirá um *match* preciso. Isso é uma limitação do algoritmo puramente baseado em *IR*.

3.2.2 Algoritmo baseado em aprendizagem de máquina com pré-processamento da entrada

O Algoritmo baseado em aprendizagem de máquina com pré-processamento da entrada (tec-ml+ir) funciona de maneira diferente do anterior, no sentido que não utiliza heurísticas para rotular um bug em determinada categoria e precisa de um conjunto de bugs previamente classificados na taxonomia de bugs para realizar o seu treinamento. O algoritmo é baseado em aprendizagem de máquina, (*Machine Learning* ou *ML*) que segundo (Michalski, *et al.*, Nguyen e Armitage [26, 31]) é um sub-campo da área de inteligência artificial que estuda e busca por modelos computacionais voltados para o processo de aprendizado.

O algoritmo classificador em questão utiliza *IR* em uma busca preliminar para localizar a quais elementos da máquina de estados o bug pode estar relacionado, e extrair *características* utilizadas pelo algoritmo de *ML*. Em seguida, um algoritmo de *ML* analisa os elementos textuais e as características extraídas para classificar o bug de acordo com a taxonomia de bugs. O algoritmo de *ML* utilizado foi o *KNN* (*K-Nearest Neighbor*), que por ser um algoritmo do tipo supervisionado requer um conjunto de entrada previamente classificado para que assim a máquina possa assimilar os padrões de bugs para cada categoria da classificação.

Os elementos fornecidos como conjunto treinamento para o algoritmo classificador são: (*i*) um *corpus B* composto de *bugs*, onde cada registro do *corpus* corresponde a um *bug*, contendo seu identificador, título, descrição, classificação; e (*ii*) as máquinas de estados do projeto de software **M**. Primeiramente *B* e *M* precisam ser pré-processados para gerar uma única entrada, a *tabela de características*. Essa tabela contém as colunas: *id*, *título*, *descrição*, *elementos-sm-relacionados*, *classificação*. Os elementos Id, título e descrição pertencem ao conjunto *B*, já explicados. A coluna *elementos-sm-relacionados* é resultado da aplicação da técnica de IR sobre os elementos textuais de *M* e *B*. Esse processo consiste em utilizar os textos dos elementos das máquinas de estados como *query* para recuperar bugs. O resultado de cada busca é um conjunto bug + elementos da máquina de estados, onde os elementos da máquina de estados são adicionados a coluna *elementos-sm-relacionados*. A

Código Fonte 3.4: Pseudo-código Algoritmo Classificador

```
1  algoritmoClassificador(arquivo){
2      dados = read(arquivo)
3      classificador = new Classificador(
4                          i = new Indexador(type = MatrixEsparsa),
5                          applicarTF-IDF(i),
6                          algoritmo = new KNN(k = 1, distancia = ('manhattan')
7                          )
8      treino, teste = separaTreinoTeste(dados, proporcao = 0.7)
9      classificador.treina(treino)
10     classificador.testa(teste)
11     classificador.calculaDesempenho()
12     proxBug = read(bugSemClassificacao)
13     classificador.classifica(proxBug)
14 }
```

coluna *classificação* corresponde à classificação prévia do *bug*, que consta na coluna *classificação* em *B*.

A construção do corpus foi feita de forma manual, analisando-se três projetos de software abertos no repositório *github*[2]. Foram eles: cristal-ise/kernel [3], exabgp [4], e POCS [5]. Procuramos por projetos que contivessem modelos de máquina de estados. Analisamos os bugs reportados com o status *fechado* em cada projeto, procurando evidências sobre a resolução do bug estar associada a elementos de máquina de estados, nos comentários dos colaboradores do projeto. Todo esse processo de montagem do *corpus* é descrito em detalhes no Capítulo 4.

O Código Fonte 3.4 apresenta o algoritmo classificador em pseudo código. O algoritmo não implementa técnicas de *IR* ou *ML*, e sim as utiliza, portanto os detalhes de como o algoritmo *KNN* é implementado por exemplo, está fora do escopo do algoritmo classificador. O algoritmo começa lendo o arquivo que contém a tabela de características (linha 1). Em seguida configura-se um classificador para que utilize o *KNN* como algoritmo (linhas 2-6).

[2]http://github.com

[3]https://github.com/cristal-ise/kernel

[4]https://github.com/Exa-Networks/exabgp

[5]https://github.com/panoptes/POCS

Nas linhas 2 a 6 do Código Fonte 3.4 um novo classificador é criado, e suas configurações definidas. Na linha 3 definimos que deve-se usar uma matriz esparsa para indexar os termos do documento de entrada. Essa forma de indexação é requerida para classificadores *multi-label* (classificação *multi-label*: o elemento pode ser classificado em mais de duas categorias). Na linha 4 definimos que iremos utilizar *TF-IDF* para quantificar a relevância dos termos indexados, a outra forma que a biblioteca fornece é a contagem simples da ocorrência dos termos. Conforme (Frakes e Baeza-Yates) demonstram, a simples contagem da ocorrência de termos não é um meio eficaz para recuperação da informação, pois podem existir vários documentos que possuem o termo procurado, sendo necessário calcular também o número de documentos em que cada termo aparece. Um conjunto de termos que ocorre com alta frequência provavelmente não terá tanta importância se aparecerem em todos os documentos pesquisados [18]. Dessa forma, entre escolher a contagem simples de termos e *TF-IDF*, optamos pela segunda, *TF-IDF*, para a tarefa de quantificar a relevância de termos [20]. Por fim, na linha 5 configuramos o algoritmo *KNN* já implementado pela biblioteca *scikit-learn* [20], que será utilizado no classificador. O k do *KNN* diz respeito à quantidade de elementos mais próximos que o algoritmo deve considerar durante a classificação.

Por padrão o valor de k é 5, contudo os experimentos [6]mostraram que o valor de $k = 1$ traz melhores resultados para classificação de bugs. Definimos ainda que o cálculo da distância dos elementos seja feito utilizando a distância *Manhattan*, o *KNN* permite utilizar diferentes tipos de cálculos de distância, sendo que por padrão o algoritmo utiliza a distância euclidiana, contudo os resultados dos experimentos[6] mostraram melhor desempenho do KNN utilizando a distância *Manhattan*.

Na linha 7 separamos o conjunto de treinamento para o algoritmo com proporção de 70% do conjunto de elementos fornecidos na linha 1. Esse valor é adotado como padrão para treinamento de técnicas de aprendizagem de máquina [20]. O treinamento e teste do classificador são feitos nas linhas 9 e 10. Por fim, o classificador está pronto para classificar um novo bug que não tem classificação prévia (linhas 11 e 12).

Para verificar se o algoritmo proposto está funcionando de maneira eficaz, o implemen-

[6]Experimentos realizados para avaliação do algoritmo classificador baseado em *ML*. Realizamos o experimento variando os parâmetros do algoritmo com o objetivo de avaliar a sua performance. Os detalhes serão discutidos no Capítulo 4

tamos na linguagem de programação Python utilizando a biblioteca *scikit-learn* [20], que implementa diversos algoritmos de aprendizagem de máquina. Uma vez implementado, submetemos a técnica a testes, tendo como alvo três projetos de software livre, disponíveis no repositório Github [7]. Esses três projetos foram escolhidos por disporem de sua base de dados de *bugs* e também as especificações de máquinas de estados. As rodadas dos experimentos consistiram em executar a técnica de classificação de bugs e calcular as métricas *precision* e *recall* alterando-se parâmetros do algoritmo classificador. A forma como alteramos os parâmetros do *KNN* e quais critérios foram utilizados, serão discutido no Capitulo 4. As decisões que levaram ao ajuste de parâmetros do algoritmo *KNN*, a inclusão do pré-processamento de texto, e qual o impacto causado com as mudanças entre os algoritmos serão discutidas no Capítulo 4.

[7] http://github.com

Capítulo 4

Avaliação

Este capítulo descreve como a técnica de classificação automática de bugs sob a perspectiva de estados foi avaliada. A técnica é constituída de duas partes: (*i*) a taxonomia para classificação de bugs; (*ii*) o algoritmo classificador. As avaliações foram estruturadas considerando a abordagem *Goal, Question, Metric - GQM* [32]. A seção 4.1 discorre sobre a avaliação da taxonomia de bugs, apresentando o planejamento, resultados e discussão do experimento para sua avaliação. A seção 4.2, discorre sobre a avaliação do algoritmo para classificação automática, apresentando o planejamento, resultados e discussão experimento para sua avaliação.

4.1 Avaliação da taxonomia

Para avaliar a taxonomia de classificação de bugs, elaboramos um estudo empírico. No experimento, o condutor submeteu bugs reportados para que o sujeito classificasse cada bug de acordo com a taxonomia. O experimento foi estruturado considerando o **GQM**, assim sendo, seu objetivo foi **analisar** a taxonomia de classificação de bugs **com a intenção de** verificar sua eficiência para classificar bugs relacionados a máquina de estados **com respeito a** sua completude e clareza **do ponto de vista dos** analistas de qualidade de software **no contexto de** correção de bugs de software. O experimento também visou avaliar a granularidade da taxonomia, ou seja, se a quantidade e nível de detalhamento de classes definidas para taxonomia está adequada para classificar bugs.

As subseções a seguir detalham o planejamento do experimento (Seção 4.1.1), metodo-

logia aplicada (Seção 4.1.2), resultados (Seção 4.1.3), e discussão (Seção 4.1.4).

4.1.1 Planejamento

Com o objetivo de avaliar a completude e clareza da taxonomia de bugs sob a perspectiva de máquina de estados, definimos que a avaliação da taxonomia seria feita através de questionário, onde os sujeitos do experimento pertenciam a uma equipe de desenvolvimento de um projeto de software real. Os bugs reportados eram provenientes desse projeto. O questionário foi disponibilizado para os sujeitos via *web* [1], onde os sujeitos, após realizarem a classificação de bugs, poderiam responder no dia e horário mais conveniente, até um prazo máximo estabelecido.

Para conduzir o experimento, a metodologia foi dividida nas seguintes fases:

I - Definição do projeto de software a ser analisado;

II - Definição do design de experimento;

III - Preparação do material de treinamento dos participantes;

IV - Execução do estudo piloto;

V - Execução do experimento;

VI - Análise dos dados.

Na fase I, definiu-se que o projeto de software a ser estudado foi o projeto ePol[2], um projeto de sistema de grande porte que está em funcionamento, com mais de 1800 registros de *bugs* reportados (dentre eles duplicados, inválidos e melhorias que precisaram passar por uma filtragem) e seis máquinas de estados, que representavam módulos do sistema. Escolhemos esse sistema primeiramente porque sua documentação continha máquina de estados, pelo fato de já se ter conhecimento de que existiam bugs reportados que estavam associados à máquina estados do sistema e pelo fato da equipe de colaboradores ser de fácil acesso. Para o experimento utilizamos um subconjunto de todos os *bugs* relatados e todas as seis máquinas de estados do sistema.

[1] Disponível em: https://goo.gl/forms/1n2lZo3XFv2hcQre2

[2] ePol - Sistema de Gestão da Atividade de Polícia Judiciária. Projeto da Polícia Federal do Brasil

Na fase II, elaboramos o questionário levando em consideração o objetivo do experimento. As questões foram elaboradas com o objetivo de verificar a completude, clareza e também granularidade da taxonomia de bugs proposta neste trabalho. O questionário utilizado encontra-se no Apêndice A. Além disso, analisamos o número de bugs disponíveis para análise manual, o contexto do projeto de software (por exemplo: alguns bugs correspondiam à máquina de estados obsoletas, outros bugs à máquina de estados atualizadas). Por fim, analisou-se a complexidade das máquinas de estados, definimos os papeis, materiais e a condução do experimento.

Na fase III elaboramos o material de treinamento para ser aplicado aos sujeitos (os colaboradores do projeto ePol que se voluntariaram para responder o questionário). O material de treinamento consiste em um documento contendo um resumo sobre máquina de estados, sobre a taxonomia e um exemplo ilustrando como utilizar a taxonomia para classificar bugs. O material de treinamento se encontra no Apêndice B.

Na fase IV executamos um estudo piloto para evidenciar problemas com o experimento elaborado, e realizar os ajustes necessários. Na fase V, executou-se o experimento propriamente dito e por fim, na fase VI realizou-se a análise dos dados coletados.

4.1.2 Metodologia

Primeiramente, para avaliar a taxonomia de classificação de bugs buscamos por um projeto de software que dispusesse da especificação de sua máquina de estado, e que os colaboradores do projetos estivessem dispostos a passar pelo treinamento e avaliar, de forma voluntária, a taxonomia de bugs. O projeto encontrado com essas características foi o projeto ePol. Um projeto de grande porte, que possui parceria da UFCG com a Polícia Federal do Brasil. O projeto contava na época do experimento com cerca de 30 colaboradores (analistas, gerentes, estagiários, etc), uma base com mais de 1800 registros de *bugs* relatados (abertos, fechados, duplicados, inválidos), e seis máquinas de estados. Além disso, a equipe de colaboradores do projeto era de fácil acesso, e se mostraram solícitos para colaborar com a avaliação. Esses fatores foram determinantes para a escolha do projeto ePol. Realizamos ainda uma busca por projetos abertos na plataforma *Github* que contivessem máquina de estados e bugs reportados. Encontramos apenas seis projetos. Para cada projeto encontrado, fez-se contato com a equipe de desenvolvimento e só obtivemos resposta de um dos projetos. Na resposta, o

colaborador do projeto informou que estava indisponível para auxiliar. Diante desse cenário, o projeto ePol foi o melhor qualificado para o experimento.

Como o projeto ePol é um projeto proprietário, todas as informações relacionadas ao projeto são sigilosas, não sendo possível apresentar as máquinas de estados do projeto neste trabalho.

Dos 1865 bugs e issues disponíveis, excluiu-se os com *status REJECTED*, *INVALID*, *DUPLICATE*, *WORKSFORME*, restando assim 1233 bugs disponíveis para o experimento.

Os bugs disponíveis foram submetidos a uma filtragem, utilizando *information retrieval* buscando descrições dos bugs relacionadas com elementos das máquinas de estados. A filtragem gerou um conjunto com 90 bugs para poder distribuir entre os voluntários.

Os papéis definidos para o experimento foram os de *sujeito* e *condutor*, onde cada sujeito é um colaborador do projeto ePol que se voluntariou para participar do experimento. O condutor foi o responsável por coordenar as etapas do experimento. Para o sujeito participar do experimento foi preciso que este se voluntariasse e fornecesse o nome. O nome foi utilizado somente para organizar os agendamentos com cada voluntário para realizar o experimento. O formulário completo que o voluntário teve acesso encontra-se no Apêndice C.

Os materiais utilizados foram:

- Material de treinamento;
- Lista de bugs para análise manual;
- Questionário para avaliação da taxonomia de bugs.

O material de treinamento é um documento com resumo da apresentação feita para o sujeito. Contém os conceitos sobre máquina de estados, seus principais elementos e exemplos de sua aplicação. O material de treinamento contém também os conceitos e definições da taxonomia, a tabela apresentando todas as categorias propostas e seus cenários. Por fim, o material de treinamento encerra-se com um exemplo de análise de bugs utilizando a taxonomia. O material de treinamento se encontra no Apêndice B. O questionário para avaliação da taxonomia de bugs encontra-se no Apêndice A. O formulário que foi disponibilizado para os sujeitos preencherem as respostas do experimento encontra-se no Apêndice C.

A preparação do experimento consistiu em duas etapas: (*i*) do ponto de vista do condutor,

e (*ii*) do ponto de vista do sujeito. Do ponto de vista do condutor, definiu-se as seguintes tarefas:

I - Preparar a entrada;

II - Submeter arquivos do relatório de bugs do projeto ePol para o algoritmo de classificação;

III - Selecionar, de forma aleatória, sujeitos para classificar bugs;

IV - Submeter lista de bugs para análise manual e questionário de avaliação da taxonomia aos sujeitos.

O passo descrito em I consiste em processar os arquivos das máquinas de estados do projeto ePol. Os arquivos foram disponibilizados em formato *XML* e serviram de entrada para o pré-processamento dos bugs que seriam analisados pelo sujeito. O pre-processamento foi realizado em II e como resultado, listou bugs que casaram com elementos das máquinas de estados do projeto. A essa listagem acrescentamos bugs que foram escolhidos aleatoriamente do conjunto de bugs disponíveis. Os bugs foram sorteados para serem analisados pelos sujeitos.

Em III, agendamos uma data adequada com cada sujeito que respondeu ao convite para realização do treinamento. Treinamos os sujeitos para classificar bugs de forma manual. Selecionamos de forma aleatória bugs obtidos em II. Selecionamos aleatoriamente um dos voluntários para execução piloto. Executamos o experimento piloto e realizamos os devidos ajustes no experimento. Por fim, executamos o experimento para os demais sujeitos.

Do ponto de vista do sujeito, definimos as seguintes etapas para o experimento:

I - Receber treinamento em data agendada com o Condutor;

II - Abrir lista de bugs para realizar análise manual;

III - Classificar bugs de acordo com a taxonomia de bugs;

IV - Submeter análise manual;

V - Responder questionário sobre a taxonomia de bugs;

VI - Submeter classificação.

Um total de sete sujeitos se voluntariaram para o experimento. Desses, um foi conduzido para o experimento piloto, com isso o conjunto de sujeitos do experimento foi de um total de seis pessoas.

A classificação dos bugs pode exigir mais ou menos esforço por parte de quem está fazendo a análise. Identificamos como fatores para classificação de bugs os seguintes elementos: *complexidade da máquina de estados*, *tamanho da descrição do bug* e *severidade* do bug reportado. A complexidade da máquina de estados pode ser medida através da *Complexidade de entendimento*. J.A Cruz-Lemus et al. [33] define em seu trabalho que a complexidade da máquina de estados está relacionada ao número de elementos de controle de fluxo (*transitions*, *states* e *actions*), e o número de elementos que a máquina de estados possui. A análise mostrou que não existia variação de complexidade entre as máquinas de estados, pois todas eram compostas somente por elementos do tipo *state* e *transitions* assim o fator complexidade foi descartado do design do experimento.

Definimos níveis para o fator *tamanho da descrição do bug*, pois o tamanho de cada descrição variava muito. Assim, calculamos a média dos tamanhos das descrições, chegando a um valor de 49 e definimos o valor *baixo* para os tamanhos abaixo da média, e o valor *alto* para os acima da média.

O fator *severidade* continha os valores: *bloqueante*, *crítico*, *normal*, *secundário*, *melhoria*. Excluimos o valor *melhoria* por entendermos que não se trata de um comportamento anômalo do sistema. Os níveis *normal* e *secundário* correspondendo ao nível *baixo* e os valores *bloqueante* e *crítico* correspondendo ao valor *alto*. Assim, definimos um *design* fatorial 2^2.

4.1.3 Resultados

Para avaliar a taxonomia de bugs sob a perspectiva de máquina de estados, aplicamos um questionário para sujeitos integrantes de um projeto de código fechado. O questionário teve como objetivo avaliar se: (*i*) a taxonomia cobre as categorias necessárias para classificar bugs relacionados a máquina de estados; (*ii*) a taxonomia contém os possíveis cenários onde um bug relacionado a máquina de estados pode ocorrer. O questionário continha as seguintes

questões:

- Q1: A taxonomia contempla as categorias necessárias para classificar bugs?
- Q2: Qual categoria você sentiu falta na taxonomia? (Responder caso a resposta da questão anterior seja “não”.
- Q3: Em uma escala de 1 a 5, onde 1 representa "muito clara"e 5 representa "muito confusa", como você classifica a descrição dos cenários de cada categoria?
- Q4: Em uma escala de 1 a 5, onde 1 representa "falta de cenários"e 5 representa "excesso de cenários", como você classifica o número de cenários da taxonomia?

A questão Q1 permite verificar a completude de taxonomia. Em caso de resposta negativa em Q1, o voluntário pode sugerir a categoria faltante em Q2. Já Q3 permite verificar a facilidade de entendimento das descrições dos cenários onde um bug pode ocorrer. Por fim, Q4 verifica a granularidade dos cenários da taxonomia. Um cenário com alta granularidade pode aumentar o nível de detalhes das possíveis causas do bug, contudo pode também aumentar o tamanho da saída da técnica de classificação de bugs, demandando mais tempo para leitura e entendimento da causa do bug.

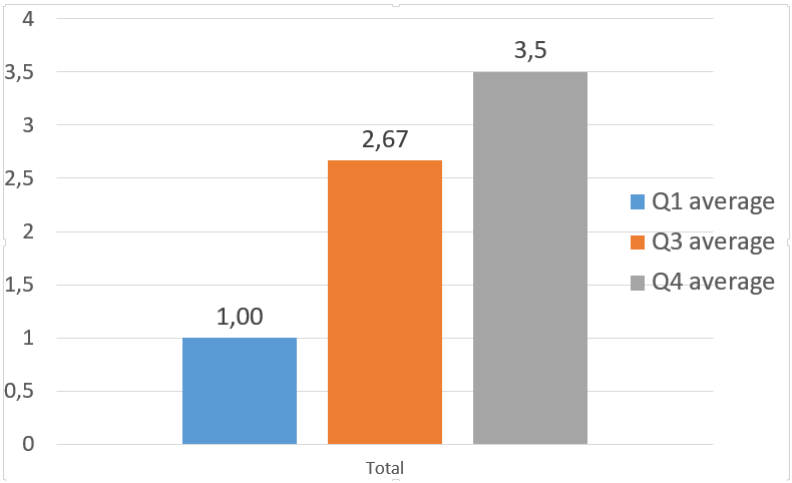


Figura 4.1: Média de valores das respostas do questionário

A Figura 4.1 apresenta o resultado das respostas obtidas no questionário. Existem duas possíveis respostas para Q1, onde “1” corresponde a “sim” e “0” a “não”. Todos os voluntários concordaram que a taxonomia contempla as categorias necessárias para classificar bugs

(100% responderam “sim”). O valor médio das respostas em Q3 foi de 2.67, indicando que as descrições dos cenários da taxonomia foi *claro*. O valor médio do das respostas em Q4 foi de 3.5, indicando que o número de cenários não é faltante nem excessivo, ou seja, o número de cenários da taxonomia é adequado.

4.1.4 Discussão

Em Q1, 100% dos voluntários responderam “sim”, mostrando que todos concordam que a taxonomia de fato contempla as categorias necessárias para classificar bugs relacionados a máquina de estados. Não houve resposta para Q2, reafirmando que os voluntários acreditam que não existe necessidade de acrescentar categorias para a taxonomia. Em Q3 quatro dos voluntários responderam com valor 1 ou 2, indicando que a maioria dos voluntários concordaram que a descrição dos cenários da taxonomia está clara, enquanto os demais voluntários relataram que os cenários da taxonomia deveriam ser mais claros. Um dos voluntários acabou se confundindo com os cenários das categorias *bug on state* e *bug on guard*. Em conversa com o sujeito após a realização do experimento, esse acabou afirmando que a confusão aconteceu pelo fato do treinamento não ter sido suficiente para sanar todas as suas dúvidas. A média obtida em Q3 mostra que as descrições dos cenários da taxonomia está clara, mas deve-se observar que a taxonomia por si só não é didática, precisando de um material de apoio para explanar sobre os cenários de cada categoria antes de utilizá-la propriamente.

Em Q4 obteve-se um valor médio de 3.5, considerando que o valor ideal para essa questão é 3, o valor obtido indica que o número de cenários da taxonomia está balanceado. Nessa questão, o valor 1 indica que a taxonomia não contém um número de cenários suficiente para descrever as causas do bug em uma dada categoria, e 5 indica que a taxonomia contém cenários em demasia, sendo que alguns cenários são desnecessários. O valor médio de 3.5 indica que o número de cenários da taxonomia está levemente inclinado para “excesso” mas não é possível identificar qual a causa do excesso apontado pelos voluntários. As possíveis causas para que alguns dos voluntários tenham apontado para o excesso de cenários pode estar relacionado com Q3, ou seja, o treinamento sobre a taxonomia não foi claro o suficiente para sanar todas as questões dos voluntários.

4.2 Avaliação do algoritmo classificador

Para avaliar o algoritmo classificador, elaborou-se um experimento quantitativo. O experimento foi estruturado considerando o **GQM**, de forma que seu objetivo é: **analisar** o algoritmo classificador, **com a intenção de** verificar sua eficiência para classificar bugs relacionados a máquina de estados **com respeito a** precisão e completude **do pondo de vista dos** analistas de software **no contexto de** correção de bugs de software.

As subseções a seguir detalham o planejamento do experimento (Seção 4.2.1), resultados (Seção 4.2.2) e discussão (Seção 4.2.3).

4.2.1 Planejamento

Com os resultados coletados do experimento descrito na Seção 4.1, avaliamos também o desempenho da técnica de classificação de bugs baseada puramente em *recuperação da informação* (tec-irBased). Para avaliar seu desempenho coletamos um corpus de bugs classificados manualmente por integrantes do projeto ePol. O corpus é constituído de 90 bugs aleatórios que foram reportados durante a utilização do sistema ePol. Os bugs foram classificados de acordo com a taxonomia de bugs sob a perspectiva de máquina de estados.

As métricas de avaliação utilizadas neste experimento foram *precision* [21] e *recall* [21]. *Precision* determina a habilidade de se classificar corretamente um bug nas categorias da taxonomia. *Recall* determina a habilidade de recuperar todos os bugs que estão associados com máquina de estados, em um conjunto de bugs relacionados/não relacionados com máquina de estados. Suas fórmulas são definidas como:

$$precision = \frac{verdadeirosPositivos}{verdadeirosPositivos + falsosPositivos}$$

$$recall = \frac{verdadeirosPositivos}{verdadeirosPositivos + falsosNegativos}$$

Avaliamos algoritmos de aprendizado de máquina que utilizamos na técnica de classificação automática de bugs, comparamos à eficiência de cada um em relação à eficiência

do algoritmo puramente baseado em *information retrieval* [18]. Nem todo algoritmo de aprendizado de máquina suporta a abordagem *overlapping* da técnica que propomos. Na abordagem *overlapping*, um bug pode ser classificado em mais de uma categoria na taxonomia. Essa funcionalidade foi implementada de forma direta em tec-irBased, de forma que o bug era analisado em cada uma das categorias da taxonomia de bugs. Já em aprendizado de máquina, essa funcionalidade se enquadra em um tipo específico de problema chamado de classificação multi-label [20]. Para esse tipo de classificação existe um grupo específico de algoritmos de aprendizagem de máquina.

A biblioteca *Scikitlearn* [20] escrita em Python é uma biblioteca que implementa, dentre outras, técnicas de aprendizado de máquina. Os algoritmos de classificação multi-label disponíveis no *Scikitlearn* são os seguintes: Passivo-agressivo; Regressão logística; SVN; Naive-Bayes; Label Propagation; KNN; Árvores de decisão; Rede neural multi-camada Perceptron (Perceptron).

Desses, apenas o KNN, árvores de decisão e Perceptron se mostraram adequados para o formato do corpus. Observamos que nos outros algoritmos, a biblioteca Scikitlearn apresentava erros ao realizar processamento de texto. Não sendo nossa prioridade depurar/corrigir o código da biblioteca, descartamos os algoritmos: passivo-agressivo, regressão logística, SVM, naive-Bayes e label propagation.

O experimento consistiu em: (i) dividir o corpus em conjunto de treino e conjunto de teste; (ii) para cada algoritmo (KNN, árvores de decisão e Perceptron) executar a sua fase de treino; (iii) executar fase de teste do algoritmo e (iv) verificar desempenho do algoritmo.

Em (i) criamos dois subconjuntos de bugs, um para treinar o algoritmo de classificação e outro para o treinamento. A divisão foi feita na proporção 70/30, tipicamente adotada em algoritmos de aprendizado de máquina 70% dos bugs foram utilizados para o treinamento do algoritmo e 30% para o teste.

Na etapa (ii), submetemos o subconjunto de treinamento para ajustar cada algoritmo. Uma vez que o treinamento foi executado com sucesso em (ii), executamos a etapa (iii) que consiste em submeter o conjunto de treinamento para cada algoritmo, sem a classificação dos bugs. O algoritmo deve por si só, classificar os bugs do conjunto de teste de acordo com o que foi aprendido na etapa (ii). Na etapa (iv), confrontamos a classificação feita pelo algoritmo em (iii) com a classificação do corpus do projeto ePol para poder calcular o desempenho do

algoritmo.

Nas fases (ii) e (iii) utilizamos como entrada o título e descrição de cada bug do corpus. O texto de cada bug foi processado utilizando indexação e TF-IDF para produzir valores numéricos. Os algoritmos de classificação utilizam esses valores para correlacionar os bugs com sua classificação.

Dentre os 90 bugs do corpus removemos seis bugs, onde quatro eram duplicados e dois correspondiam a bugs classificados como *Bug on Fork*. Como tivemos somente dois bugs nessa categoria, teríamos um bug para treinamento e outro para teste, não sendo suficiente para o algoritmo aprender a classificar na categoria *Bug on Fork*. Para um conjunto de bugs de uma dada categoria da taxonomia ser incluso no experimento, consideramos a proporção 70/30 utilizada anteriormente. Consideramos assim, que deveria existir pelo menos três bugs de uma dada categoria para esses serem utilizados no experimento, onde dois bugs seriam para treino, e um bug para teste. Por fim, nenhum dos bugs do corpus foi classificado na categoria *Bug on Join*, não sendo possível verificar a precisão do algoritmo para esse tipo de bug.

4.2.2 Resultados

Nas sub-seções a seguir apresentamos os resultados dos algoritmos classificadores. Na sub-seção 4.2.2.1 discorremos sobre os resultados da técnica puramente baseada em *IR* (**pureIR-tec**). Na sub-seção 4.2.2.2 discorremos sobre os resultados da técnica puramente baseada em aprendizado de máquina (**pureML-tec**). Na sub-seção 4.2.2.3 discorremos sobre os resultados da técnica baseada em aprendizado de máquina com pré-processamento da entrada (**ml+ir-tec**).

4.2.2.1 Resultado do experimento com técnica puramente baseada em *IR*

Do total de 90 bugs, 42 bugs foram classificados por seis vonluntários do projeto ePol, onde cada um classificou sete bugs. 24 desses bugs foram classificados como relacionados à máquina de estados (o projeto ePol contava na época com seis máquinas de estados). Os 24 bugs relacionados à máquina de estados foram classificados nas categorias da taxonomia de classificação de bugs.

Como resultado, obteve-se: (*i*) *corpus* de bugs classificados manualmente pela equipe de voluntários do ePol; (*ii*) o desempenho do algoritmo classificador utilizando puramente *IR*: *precision* = 0.2, *recall* = 0.082. O resultado em *ii* implica que, uma vez o bug classificado como relacionado à máquina de estados, (*pureIR-tec*) classifica-o na categoria correta em 20% dos casos. Adicionalmente, (*pureIR-tec*) mostrou que consegue identificar bugs que de fato estão relacionados com máquina de estados em 8.3% dos casos.

Devido os baixo desempenho de *pureIR-tec*, descartamos a possibilidade de um novo experimento usando a mesma técnica mas com o corpus dos projetos *opensource*. Priorizamos os experimentos da técnica utilizando aprendizagem de máquina com o objetivo de confrontar seus resultados com os da técnica *pureIR-tec*.

4.2.2.2 Resultado do experimento com técnica puramente baseada em aprendizado de máquina

O experimento com a técnica puramente baseada em aprendizado de máquina (**pureML-tec**) foi conduzido primeiramente no *corpus* resultante do experimento anterior. Esse corpus é composto por bugs de um projeto de software proprietário, dessa forma não era possível disponibilizar a base de bugs que utilizamos, dificultando a reprodução de nosso experimento. Somente após realizar o experimento com o projeto ePol foi que encontramos projetos de software *opensource* para utilizar em nosso experimento. Além disso, o resultado obtido com a técnica (*pureIR-tec*) nos motivou a buscar outras técnicas para aumentar a eficiência de nosso algoritmo classificador.

Pesquisamos por projetos *opensource* no repositório Github [3] e encontramos três projetos que disponibilizavam documentação com máquina de estados e repositório de bugs, foram eles: cristal-ise/kernel [4], exabgp [5], e POCS [6]. Os três projetos renderam um total de 62 bugs aptos para classificação de acordo com a taxonomia de bugs. Realizamos o experimento utilizando o corpus dos projetos *opensource* e também no corpus do projeto ePol para fazer o comparativo de desempenho das técnica baseadas em aprendizado de máquina.

Utilizou-se algoritmos de aprendizado de máquina neste experimento pois se mostra-

[3] www.github.com

[4] https://github.com/cristal-ise/kernel

[5] https://github.com/Exa-Networks/exabgp

[6] https://github.com/panoptes/POCS

ram mais eficientes do técnicas baseadas puramente em IR, contudo nem todo algoritmo de aprendizado de máquina suporta a abordagem *overlapping* da técnica de classificação de bugs. Essa característica foi implementada de forma direta em (*pureIR-tec*), de forma que o bug era analisado em cada uma das categorias da taxonomia de bugs.

Conforme explicamos na sub-seção 4.2.1, apenas os algoritmos KNN, árvores de decisão e Perceptron se mostraram adequados para o formato do corpus. Dessa forma, somente esses algoritmos foram utilizados na técnica de classificação automática.

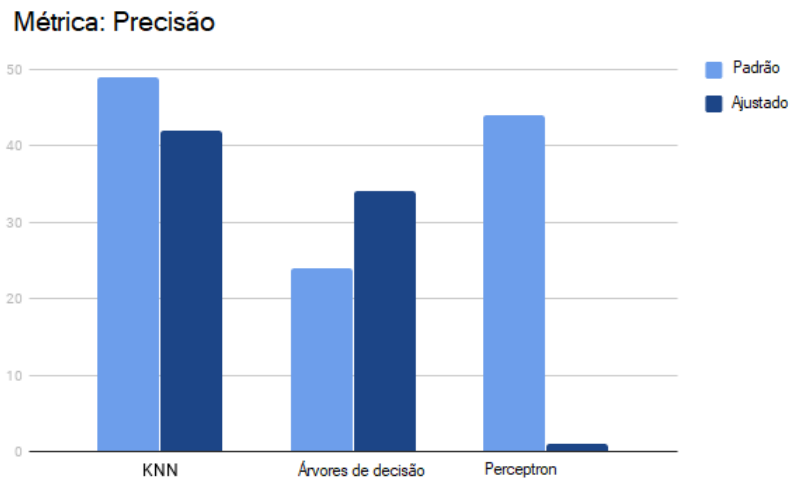


Figura 4.2: Precisão dos algoritmos com parâmetros: padrão e ajustado

A Figura 4.2 mostra desempenho de (*pureML-tec*) em relação à métrica *precision* para o conjunto de bugs do corpus resultante o experimento anterior. Na biblioteca *Scikitlearn*, cada algoritmo possui uma série de parâmetros que podem ser alterados com o objetivo de melhorar sua performance. Por exemplo, no KNN podemos alterar o valor de k (o número de vizinhos mais próximos) e observar se o desempenho do algoritmo melhora ou não. A biblioteca conta com uma ferramenta que possibilita uma busca, de maneira exaustiva, por melhores parâmetros para o algoritmo. O processo de busca é chamado de ajuste de algoritmo (*algorithm turning*), pois visa otimizar o desempenho do algoritmo, através do ajuste de parâmetros.

Além do processo *algorithm turning*, cada algoritmo de aprendizagem de máquina possui valores padrão (*padrão*) para seus parâmetros. Os valores padrão dos parâmetros do algoritmo KNN por exemplo, são: k = 5, distância = Euclidiana.

Comparamos o desempenho dos algoritmos com os valores indicados por essa busca em comparação aos valores *padrão* utilizados em cada algoritmo. Como podemos observar na Figura 4.3, os resultados da busca apresentaram valores que diminuíram a performance dos algoritmos KNN e Perceptron, enquanto a performance do algoritmo de árvores de decisão aumentou. O KNN se mostrou o algoritmo com melhor *precision*, (valor de 0.49) enquanto árvores de decisão obteve a pior performance (0.24), já o algoritmo Perceptron obteve uma precisão de 0.44. Os valores apresentados com a legenda *Ajustado* são os resultados obtidos após ao ajuste dos parâmetros dos algoritmos em questão, realizado pela biblioteca *Scikitlearning*.

A Figura 4.3 mostra o desempenho dos três algoritmos (KNN, árvores de decisão, Perceptron) em relação à métrica recall. Para o conjunto de bugs do corpus resultante o experimento anterior. O KNN obteve o melhor resultado dentre os três algoritmos e com os valores *padrão* para seus parâmetros, com um valor de 0.45, já o algoritmo de árvores de decisão obteve um *recall* de 0.30 utilizando valores de parâmetros resultantes da busca da biblioteca Scikitlearn, já o algoritmo Perceptron obteve um *recall* de 0.26. A análise dos resultados de cada algoritmo será discutida na sub-seção 4.2.3.

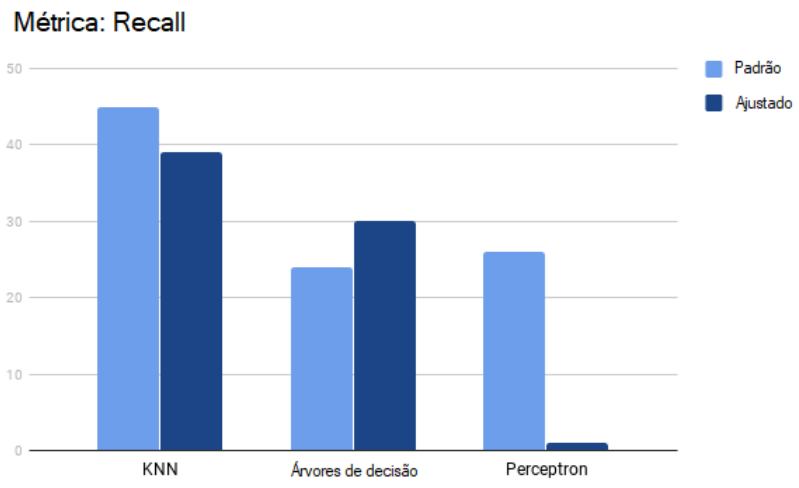


Figura 4.3: Recall dos algoritmos com parâmetros: *padrão* e ajustado

Os elementos da entrada são designados para os conjuntos de treinamento e teste de forma aleatória. Assim, um conjunto (treinamento/teste) A pode levar o algoritmo a um melhor desempenho do que um conjunto B, mesmo que ambos utilizem a proporção 70/30

pois uma configuração de elementos de um pode ser mais relevante para o treinamento do algoritmo do que outra.

Executamos o experimento testando várias configurações de parâmetros. Primeiramente, verificamos qual a melhor proporção para os subconjuntos de treinamento e teste. Além da proporção padrão (70/30), mediu-se a eficiência do algoritmo para as proporções 60/40 e 80/20. A proporção padrão (70/30) foi a que apresentou melhores resultados. Verificamos também o desempenho dos algoritmos utilizando apenas TF, sem utilizar IDF. Utilizando apenas TF, a precisão do algoritmo KNN para detectar *Bug on Activity* aumentou de 0.50 para 0.75, contudo a métrica *precision* caiu nas demais categorias, de forma que a métrica *precision* geral do algoritmo foi para 0.42, além de Recall diminuir de 0.45 para 0.39.

O algoritmo KNN foi o que apresentou melhor desempenho dentre os algoritmos utilizados pela técnica de classificação. Assim, descartou-se o algoritmo de árvores de decisão e o algoritmo Perceptron.

A seguir, executamos o experimento utilizando um conjunto de bugs de projetos de software *opensource*. Nesse caso não tínhamos à disposição voluntários para realizar a classificação manual de bugs, assim realizamos uma filtragem nos bugs reportados, buscando por bugs que já haviam sido solucionados e que continham termos das máquina de estados dos projetos na descrição e comentários dos colaboradores. Verificando a solução realizada e os comentários dos colaboradores, conseguimos evidências para a classificação dos bugs.

O experimento para avaliar (*pureML-tec*) utilizando o algoritmo KNN, foi conduzida nos seguintes projetos de software: cristal-ise/kernel [7], exabgp [8], e POCS [9]. Na Tabela 4.1 temos exemplos de bugs de cada projeto onde, por exemplo, na segunda linha, temos o bug de código identificador 55 na página do projeto kernel. O bug em questão diz respeito a uma regra da máquina de estados do sistema, no momento da transição entre os estados *done* e *completed* onde uma *flag* que está ativa só deve ser desativada imediatamente após a ativação de outra *flag*.

Um total de 62 bugs foram utilizados como entrada para o algoritmo de classificação, onde 16 bugs foram do projeto kernel, 26 do projeto exabgp, e 14 do projeto POCS. A Figura 4.4 apresenta os resultados do desempenho de (*pureML-tec*) em relação as métricas

[7] https://github.com/cristal-ise/kernel

[8] https://github.com/Exa-Networks/exabgp

[9] https://github.com/panoptes/POCS

Tabela 4.1: Exemplo de bugs reportados nos projetos opensource

ID	Projeto	Título	Descrição
248	exabgp	Match Multiple TCP-Flags with Hex value in Flowspec Rules	Since the flowspec support in Exabgp is mostly used for DDoS mitigation, it'd be nice to have the ability o match multiple tcp-flags in hex value like on the JunOS platform. For example, a tcp christmas tree packet will have the hex value of 0x29.
55	kernel	Activity is set only inactive when next Activity is "run", i.e. its runNext() is called.	Check CAExecutionTest 'Execute ElemAct using Done transition' test method. StateMachine are not used. This might not be a bug, but resolving this functionality could simplify AdvancementCalculator
23	pocs	Camera Stop/Shutdown	Have a better method for interrupting camera to guarantee we don't get in a locked out state with exposure. Should be able to build into the subprocess/async.subprocess easily.

precision e *recall*.

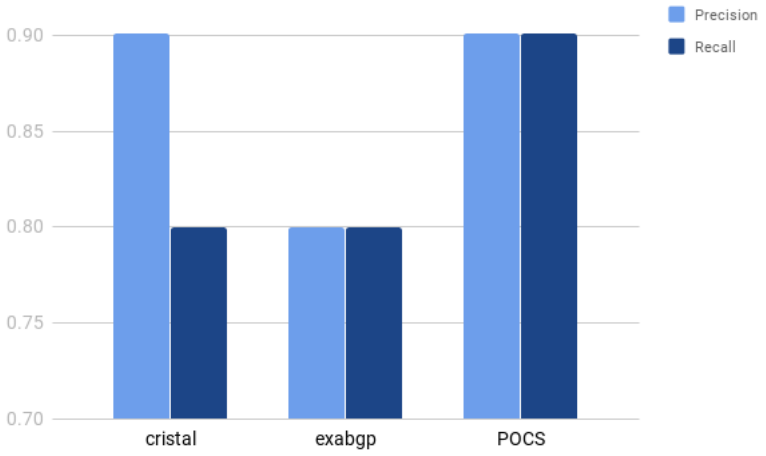


Figura 4.4: Desempenho da técnica de classificação baseada puramente em ML

No projeto *cristal* (*pureML-tec*) com KNN obteve *precision* de 0.9 e *recall* de 0.8, já no projeto exabgp (*pureML-tec*) com KNN obteve *precision* e *recall* igual a 0.8, por fim no projeto POCS (*pureML-tec*) com KNN obteve *precision* e *recall* igual a 0.9. Os valores obtidos em Figura 4.4 são os melhores resultados de 100 execuções de (*pureML-tec*). A cada rodada os bugs de cada projeto eram escolhidos aleatoriamente para compor o conjunto de treinamento e o conjunto de teste de (*pureML-tec*). As Figuras 4.5, 4.6 e 4.7 apresentam o desempenho de 100 rodadas de (*pureML-tec*) para cada projeto. Podemos observar na Figura 4.5 um baixo desempenho do algoritmo nesse projeto, que teve como causa a falta de objetividade das descrições dos bugs nesse projeto, conforme é explicado na sub-seção 4.2.3.

Além das métricas *precision* e *recall*, os gráficos apresentam a métrica *f1-score*, também conhecida como *f-measure*, corresponde a média harmônica entre *precision* e *recall*. Embora essa métrica não faça parte das métricas utilizadas para avaliar o desempenho dos algoritmos de classificação de bugs, ela fornece uma visão geral do desempenho dos algoritmos.

4.2.2.3 Resultado do experimento com técnica baseada em aprendizado de máquina com pré-processamento da entrada

O experimento com a técnica baseada em aprendizado de máquina com pré-processamento da entrada (**ml+ir-tec**) foi conduzido nos três projetos de software utilizados na segunda

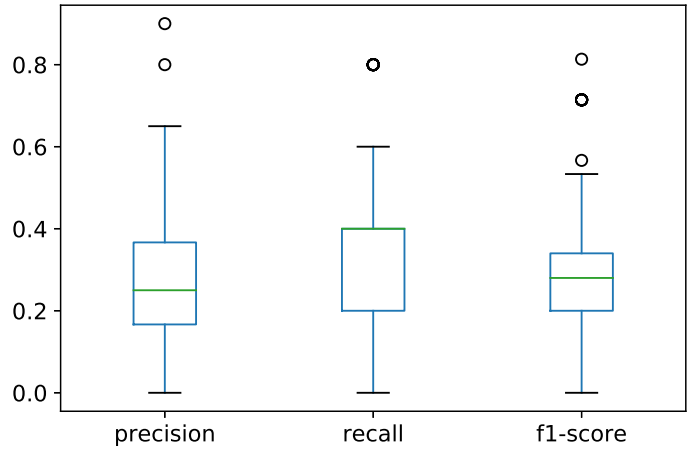


Figura 4.5: Desempenho de 100 rodadas de *pureML-tec* no projeto exabgp

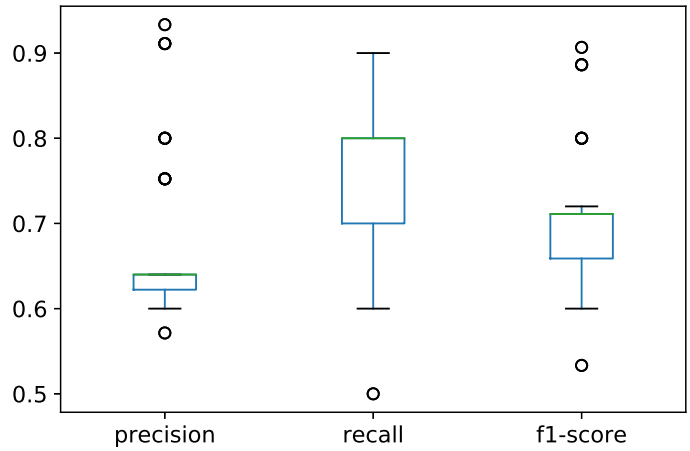


Figura 4.6: Desempenho de 100 rodadas de *pureML-tec* no projeto cristal

etapa do experimento anterior. O algoritmo de aprendizado de máquina utilizado por *ml+ir-tec* foi o KNN.

A forma como *ml+ir-tec* incorpora o algoritmo de aprendizagem de máquina é o mesmo de *pureML-tec*, de forma que a diferença desta é o pré-processamento dos bugs que servirão de entrada para *ml+ir-tec*. Utilizando IR, busca-se na descrições dos bugs, elementos relacionados à máquina de estados do projeto de software, caso a busca tenha sucesso, *ml+ir-tec* acrescenta uma *tag* à descrição do bug com o objetivo de facilitar aprendizado de máquina em *ml+ir-tec*, mais detalhes sobre encontra-se no Capítulo 3.

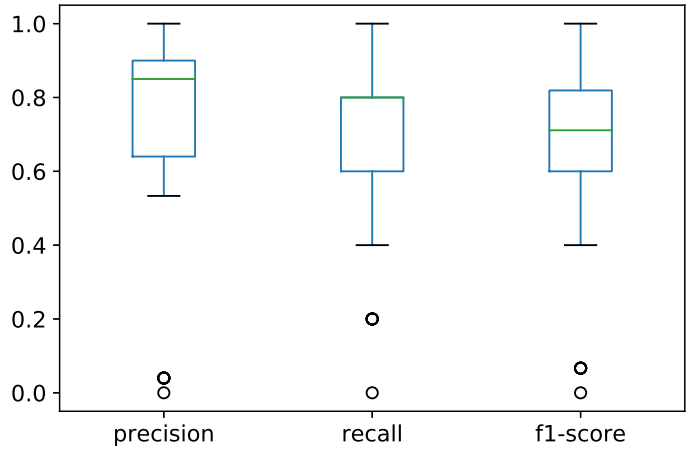


Figura 4.7: Desempenho de 100 rodadas de *pureML-tec* no projeto POCS

Este experimento foi conduzido nos projetos de software cristal-ise/kernel [10], exabgp [11], e POCS [12], de cada projeto extraimos respectivamente: 16, 26 e 14 bugs. Assim, um total de 62 bugs foram submetidos submetidos para *ml+ir-tec*. Ajustamos os parâmetros do algoritmo KNN da seguinte forma: k = 1, *distância* = *manhattan*, *proporção treino/teste* = 70/30. A Figura 4.8 apresenta os resultados do desempenho d*ml+ir-tec* em relação as métricas *precision* e *recall*.

No projeto *cristal ml+ir-tec* obteve *precision* de 0.9 e *recall* de 0.8, já no projeto exabgp *ml+ir-tec* obteve *precision* e *recall* igual a 0.8, por fim no projeto POCS, *ml+ir-tec* obteve *precision* e *recall* igual a 0.9. Os valores obtidos em Figura 4.8 são os melhores resultados de 100 execuções de *ml+ir-tec*. O melhor resultado de *ml+ir-tec* foi o mesmo resultado de *pureML-tec*, contudo o desempenho médio durante 100 rodadas de *ml+ir-tec* foi melhor do que *pureML-tec* no projeto cristal, e pior no projeto exabgp, conforme constata-se em Figura 4.9, Figura 4.10 e Figura 4.11.

4.2.3 Discussão

No que diz respeito aos resultados do desempenho dos algoritmos classificadores, a técnica puramente baseada em *information retrieval* (*pureIR-tec*) obteve pior desempenho, mos-

[10]https://github.com/cristal-ise/kernel
[11]https://github.com/Exa-Networks/exabgp
[12]https://github.com/panoptes/POCS

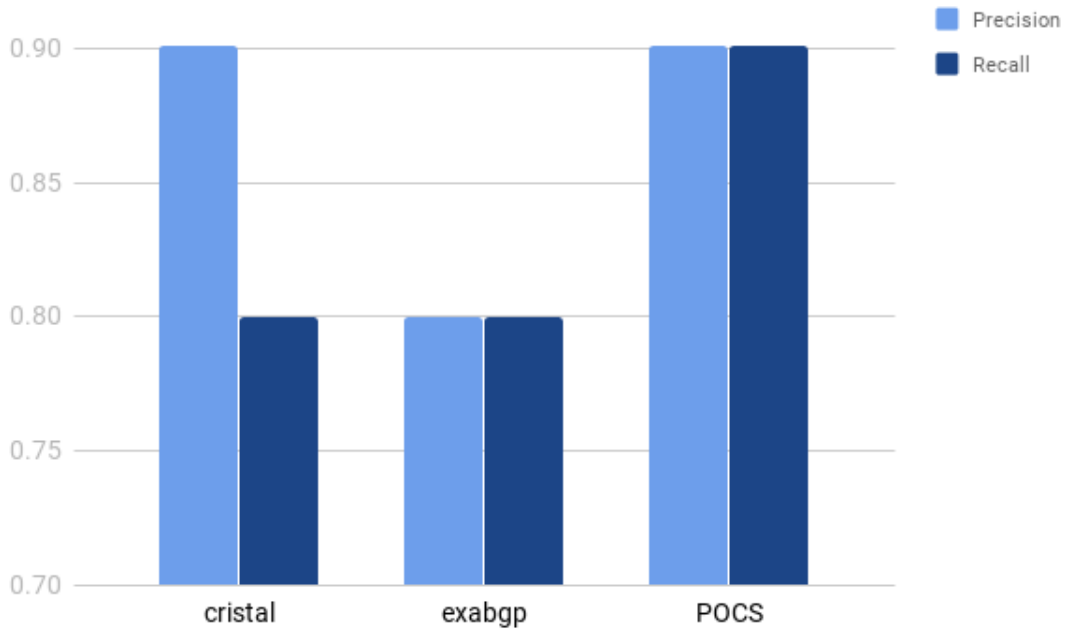


Figura 4.8: Desempenho da técnica de classificação baseada puramente em ML

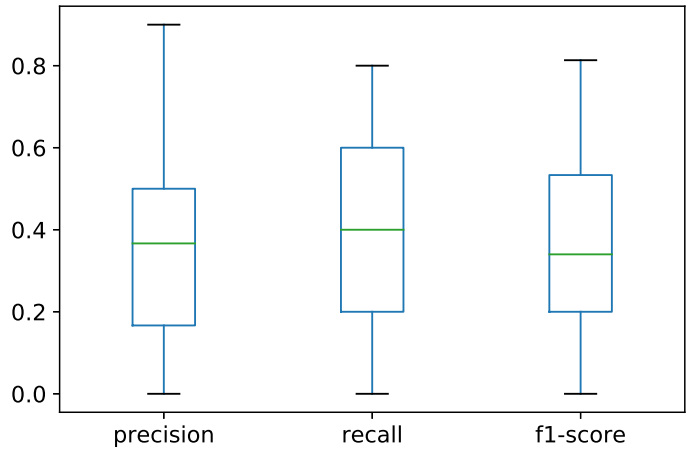


Figura 4.9: Desempenho de 100 rodadas de *ml+ir-tec* no projeto cristal

trando que utilizar somente *IR* não é suficiente para classificar bugs relacionados à máquina de estados de forma automática.

A técnica (*pureIR-tec*) obteve *precision* = 0.2 e *recall* = 0.083, mostrando assim uma capacidade de recuperar bugs pior do que uma classificação totalmente aleatória do bug (de forma aleatória uma suposta técnica teria 50% de chance em classificar o bug como relacio-

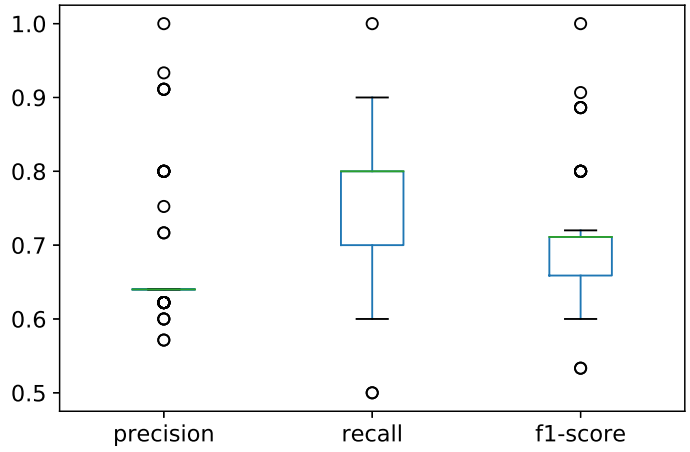


Figura 4.10: Desempenho de 100 rodadas de *ml+ir-tec* no projeto exabgp

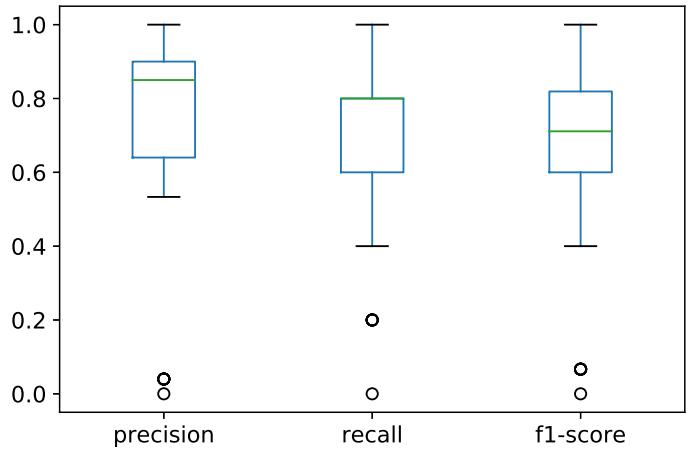


Figura 4.11: Desempenho de 100 rodadas de *ml+ir-tec* no projeto POCS

nado a máquina de estados ou não). A razão deste resultado se deu pela pouca quantidade de elementos textuais nas máquinas de estados do projeto de software. As máquinas de estados do projeto eram basicamente formadas por estados e transições, sendo que não havia nenhum evento associado à transições, tão pouco guardas e atividades. Observamos, portanto, que a documentação da máquina de estados do projeto analisado não tinha o nível de detalhes necessários para aplicar *pureIR-tec*.

Sabemos que o desempenho de técnicas de *IR* está diretamente relacionado à quantidade de texto relevante que é analisado [18]. Uma vez que *pureIR-tec* aplica *IR* nos títulos e des-

crições de bugs, investigamos se bugs de maior severidade, têm descrições mais detalhadas. Dessa forma, verificamos se existe uma correlação forte entre o tamanho do texto do bug, sua severidade, bem como com a classificação correta de bugs por *pureIR-tec*.

Para verificar se existe correlação forte entre a severidade de um bug e sua correta classificação por *pureIR-tec*, analisamos os quatro valores possíveis para severidade: (*critical*, *blocked*, *normal* e *secondary*). A segunda variável que analisamos foi o tamanho do texto do bug (*tamanho*). Essa variável pode assumir uma faixa muito grande de valores (texto de bugs não possuem tamanho específico), de forma que, analisar todos os níveis dos fatores (*severidade* e *tamanho*) tornaria o experimento inviável. Para contornar esse problema, aplicamos o design de experimento 2^n, onde os fatores *severidade* e *tamanho* podem assumir dois valores: *alto*, *baixo*.

Para mapear os valores do fator *tamanho* para *baixo*/*alto*, calculamos a mediana do tamanho do textos de todos os bugs do repositório de bugs. O valor de *tamanho* menor que a mediana foi mapeado para o valor *baixo*, enquanto o valor de *tamanho* maior que a mediana foi mapeado para o valor *alto*. Já para mapear os valores do fator *severidade*, definimos os valores *secondary* e *normal* como *baixo*, enquanto os valores *critical* e *blocked* foram mapeados como *alto*. A variável resposta desse experimento é a classificação do bug que nomeamos como *acerto*, que pode assumir dois valores possíveis: 1 se o bug foi corretamente classificado, 0 caso o bug não tenha sido corretamente classificado.

Executamos o teste de correlação Spearman [34] para determinar o coeficiente de correlação de Spearman: ρ. O valor do coeficiente pode variar entre $[-1, 1]$, onde -1 e 1 indicam uma perfeita correlação entre as variáveis, enquanto 0 indica que não há nenhuma correlação entre as variáveis. Diante disso, levantamos as seguintes hipóteses:

$$h0 : \rho = 0 \tag{4.1}$$

$$h1 : \rho \neq 0 \tag{4.2}$$

Onde em *h0*, temos a hipótese de que não existe correlação entre o tamanho e severidade de um dado bug com sua classificação, já em *h1* temos a hipótese de que existe correlação entre essas variáveis. Para rejeitar *h0*, ρ deve ser significativamente maior ou menor que 0, ou seja, $\rho > 0.5$ ou $\rho < -0.5$, implicando que a correlação é forte. Caso $-0.5 > \rho < 0.5$

Tabela 4.2: Resultado do teste de correlação de Spearman

		tamanho	severidade	acerto
tamanho	ρ	1.000	**0.039**	**-0.298**
	N	42	42	42
severidade	ρ	**0.039**	1.000	**-0.298**
	N	42	42	42
acerto	ρ	**-0.298**	**-0.298**	1.000
	N	42	42	42

não é possível rejeitar *h0* [34]. A Tabela 4.2 mostra o resultado do teste.

A Tabela 4.2 mostra que o valor obtido para ρ em todos os casos não é significativamente maior que 0. Os valores 0.039 e -0.298 se encontram na faixa $[-0.5, 0.5]$, implicando que não é possível rejeitar *h0*. Assim, não se pode dizer que há correlação entre o tamanho do texto e severidade do bug, com sua classificação.

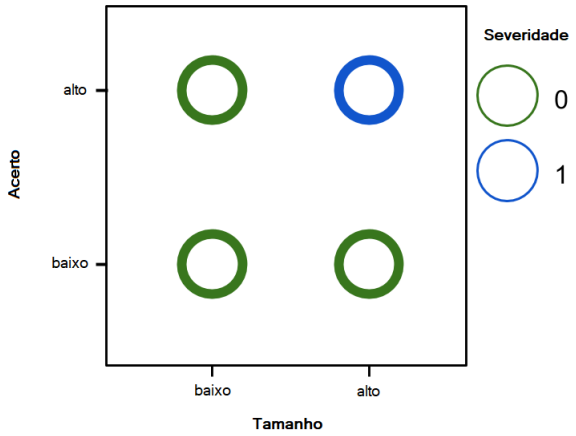


Figura 4.12: Gráfico de dispersão de correlação entre *tamanho*, *severidade*, e *acerto*

A correlação de Spearman supõe uma correlação linear entre as variáveis, mas é possível que exista uma correlação não linear entre as variáveis. Com o intuito de verificar a existência de uma correlação não linear entre o *tamanho*, *severidade* e *acerto*, plotamos um gráfico de dispersão de correlação para que fosse possível identificar qualquer tipo de correlação não linear entre as variáveis. A Figura 4.12 mostra o gráfico de correlação. Mais uma vez não observou-se qualquer correlação entra *tamanho*, *severidade* e *acerto*.

Considerando agora os resultados obtidos pela técnica baseada puramente em aprendi-

zado de máquina (*pureML-tec*), analisamos o desempenho de três algoritmos de aprendizado de máquina (KNN, árvores de decisão, redes neurais Perceptron). O algoritmo *KNN* foi o que se mostrou mais eficiente durante o experimento. O algoritmo de árvore de decisão é conhecido por obter boa performance (maior que 70%) utilizando uma base de dados pequena [35], contudo para essa boa performance o algoritmo pressupõe que a classificação a ser realizada é do tipo *one-label*, como a classificação de bugs que utilizamos é do tipo *multi-label* e a base de dados onde executou-se os experimentos é pequena, não há como esperar o resultado apontado por Antoniol *et.al.*[35]. Já o algoritmo de redes neurais Perceptron é conhecido por precisar de uma grande base de dados (na casa dos milhares de registros) [36] para que a rede consiga se estabilizar e obter boa performance. O processo de estabilização do algoritmo de redes neurais, é o processo em que a rede passa a obter sempre o mesmo desempenho, com diferentes instâncias. O algoritmo não é determinístico, de forma que obtivemos diferentes valores de desempenho (inclusive valores nulos) para um mesmo conjunto de entrada. A razão desse comportamento é o tamanho da base de dados utilizada não ser grande o suficiente para estabilizar a rede neural.

O algoritmo *KNN* apresentou o melhor resultado no experimento, realizamos alterações de forma manual em seus parâmetros, com o objetivo de melhorar sua performance. O *KNN* possui dentre outros parâmetros, o k, que representa o número de vizinhos mais próximos que o algoritmo irá considerar na votação para classificar um determinado bug. Variamos o valor de k na faixa de [1, n], onde n é o número de amostras dividido por dois. O desempenho do algoritmo *KNN* foi melhor com k = 1 e pode ser visto na Tabela 4.3, onde cada linha mostra a precisão e *recall* do algoritmo para cada categoria da classificação de bugs.

Executamos o treinamento e teste do algoritmo diversas vezes, cada execução é uma instância do algoritmo com diferentes bugs que foram distribuídos aleatoriamente para o conjunto de treinamento e teste, respeitando a proporção *70/30*. Como resultado observamos que o desempenho do algoritmo em cada categoria da taxonomia de bugs variou bastante. Na Figura 4.13 apresenta um comparativo da métrica *precision* das cinco instâncias que apresentaram melhores desempenho do KNN com diferentes conjuntos treinamento/teste, já em Figura 4.14 apresenta o comparativo da métrica recall das cinco melhores instâncias do KNN com diferentes conjuntos de treinamento/teste.

A instância 1 é a instância que se encontra na Tabela 4.3, e foi a instância que obteve

Tabela 4.3: Desempenho do algoritmo KNN por categorias da taxonomia

Classe	Precision	Recall
Não relacionado	0.54	0.88
Bug on State	0.71	0.45
Bug on Guard	1.00	0.50
Bug on Activity	0.33	0.40
Bug on Entry	1.00	0.33
Bug on Exit	0.00	0.00
Média	0.61	0.52

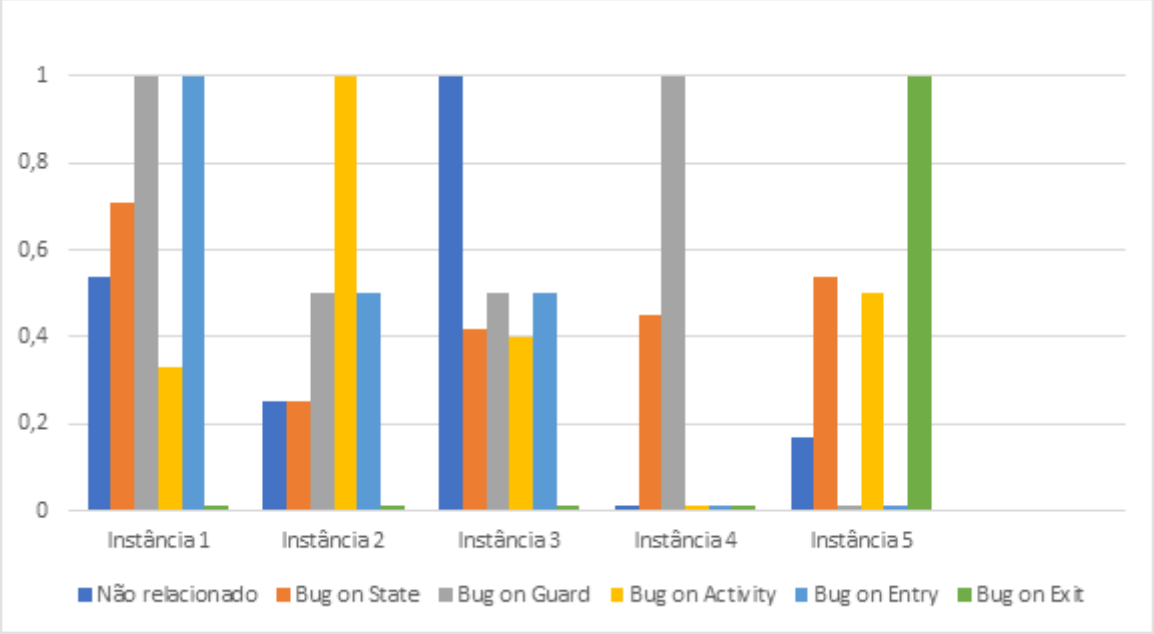


Figura 4.13: *Precision* do KNN para diferentes conjuntos de treinamento/teste

o melhor resultado de *precision* e *recall* simultaneamente. Podemos notar, na Figura 4.13, que outras instâncias do KNN obtiveram maior *precision* em alguma classe específica, como por exemplo em instância 3 a *precision* para bugs não relacionados foi de 1,00, contudo observa-se em Figura 4.14 que o *recall* em instância 3 para mesma classe foi de 0,09. Assim, podemos concluir que nesse caso menos bugs não relacionados a máquina de estados foram recuperados do que no caso da instância 1. Notamos na Figura 4.13 que o *recall* do algoritmo para classificar na categoria *bug on entry* foi abaixo da média geral do algoritmo. Isso se deve principalmente a baixa quantidade de bugs classificados nessa categoria disponíveis para treinar o algoritmo. O corpus extraído dos projetos não era balanceado, isso se deve principalmente à quantidades diferentes de bugs classificados nas categorias, onde algumas categorias possuíam mais elementos do que outras.

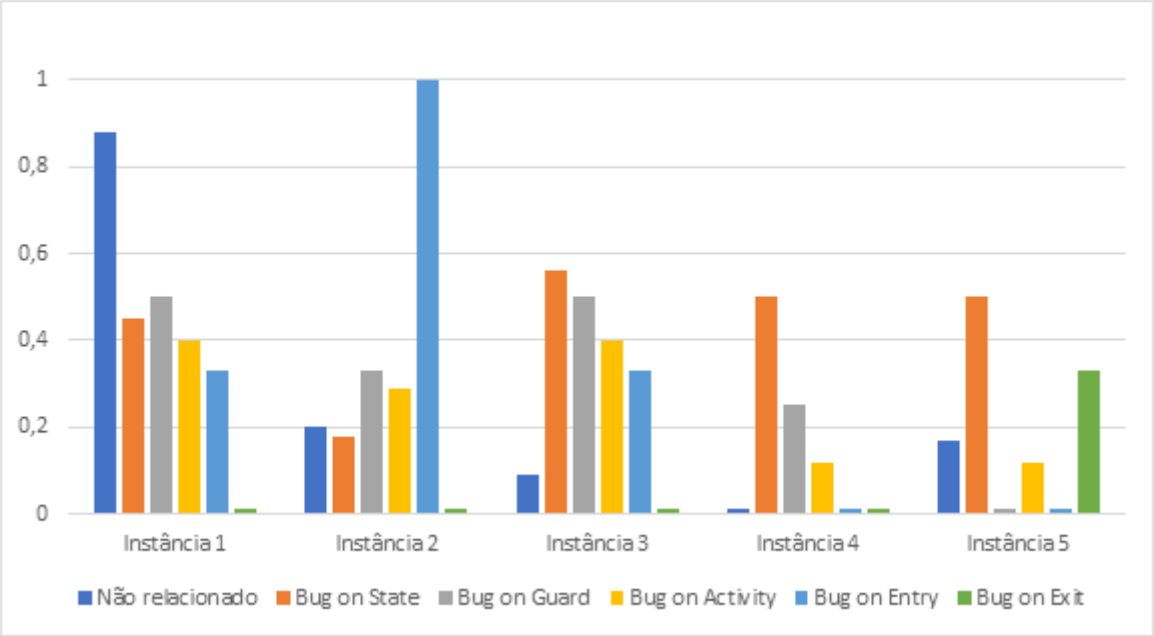


Figura 4.14: Recall do KNN para diferentes conjuntos de treinamento/teste

Comparando-se o resultado do algoritmo KNN com o algoritmo baseado puramente em IR (*ir-based*), observamos um ganho de 300% em *precision* e 650% em *recall*. O desempenho do algoritmo cresceu exponencialmente, o que fez surgir a seguinte pergunta:

- Quais foram as razões para que o KNN tivesse desempenho tão superior ao *ir-based*?

Para responder essa pergunta comparamos a classificação KNN e a classificação do *ir-based*. No comparativo, para cada bug classificado coletamos: identificador, título, descri-

Tabela 4.4: Relação de bugs analisados simultaneamente pelo KNN e ir-based

Bug_id	IR_Classification	KNN_Classification	User_Classification
57	nr	nr	bs/ba
248	bs	bs/bg	nr
318	bs	nr	nr
353	nr	nr	nr
578	nr	nr	nr
1088	nr	bs	nr
1338	nr	ba	bg/ba
1544	bs	ba	ba
1605	nr	bs	bs/bex
1953	bs	be	be
1994	bs	nr	nr
1997	nr	bs/ba	ba/be
2182	nr	be	be

Legenda: nr: não relacionado; bs: Bug on State; bg: Bug on Guard; ba: Bug on Activity; be: Bug on Entry; bx: Bug on Exit

ção, classificação KNN, classificação *ir-based* e classificação dos especialistas. Nem todos os bugs analisados no segundo experimento (*pureML-tec*) foram analisados no primeiro (que utilizou *ir-based*) de forma que descartamos os bugs que não foram analisados no primeiro experimento.

Observando a Tabela 4.4 notamos que apenas os bugs 353, 578 e 1088 foram classificados corretamente pelo algoritmo *ir-based*. Esses bugs foram classificados como não relacionados à SM, contudo o KNN classificou o bug 1088 como *Bug on State*. Investigando os outros bugs da Tabela 4.4 que foram classificados pelo KNN como *Bug on State* (bugs: 248, 1605, 1997), encontramos um padrão entre esses bugs: o tamanho do texto e termos em comum. O tamanho dos textos possui em média 334 caracteres, enquanto a média geral do tamanho dos textos dos bugs no experimento é de 469. Os termos encontrados em comum nos três bugs foram termos de um *template* adotado pela equipe de desenvolvimento para reportar bugs,

não sendo relevante para a classificação dos bugs.

Observa-se que o tamanho do texto do bug 1088, que é 308, está bem abaixo da média de tamanho dos bugs de experimento, 469. O texto do bug em si contém pouca informação, possui os termos do *template* de bugs (por exemplo: "usuário", " que você está fazendo?", "qual erro aconteceu?") e ainda possui os termos que estavam presentes em classificações cuja a categoria é de fato *Bug on State*. A soma de todos esses fatores levaram o KNN a errar na classificação. Mostrando a limitação do algoritmo com relação a bugs com textos de tamanhos muito abaixo da média de tamanho.

Um caso interessante aconteceu com o bug 57 que se encontra na primeira linha Tabela 4.4. O fato de ambos os algoritmos o classificaram como não relacionado a SM nos motivou a rever a sua classificação manual. Ficou constatado que de fato, esse bug não está relacionado com máquina de estados, assim o erro ocorreu de fato na fase de classificação manual desse bug.

Os bugs não relacionados com SM de número 318 e 1994 foram classificados erroneamente pelo algoritmo *ir-based* como *Bug on State*. Já no caso do bug 1994, o algoritmo recuperou um termo encontrado nas SM e no título do bug.

Os bugs 1544 e 1953 foram recuperados por ambos os algoritmos, contudo o *ir-based* errou nas classificações. Os bugs 1544 e 1953 foram associados a estados das SM. Em ambos os casos, a classificação só não foi correta pelo *ir-based* pela ausência de *actions* nas SM do projeto. Em uma SM mais detalhada o algoritmo *ir-based* poderia classificar corretamente esses bugs.

Por fim, os bugs 1997 e 2182 poderiam ser recuperados pelo *ir-based* se o algoritmo considerasse o nome da máquina de estados como entrada, contudo a heurística para classificação teria que ser alterada, pois nesse caso o bug teria relação somente com o nome da máquina de estados.

Na Tabela 4.5 observa-se o desempenho de *pureML-tec* nos projetos *open-source*.

O desempenho de *pureML-tec* foi melhor nos projetos *open-source* do que no projeto ePol. O principal fator para o menor desempenho médio de *pureML-tec* no projeto ePol se deu pelo fato do baixo desempenho do algoritmo para classificar bugs em categorias específicas. Conforme apresentado em Tabela 4.3, *pureML-tec* obteve *precision* e *recall* igual a zero na categoria *Bug on Exit*; 0.33 e 0.40 na categoria *Bug on Activity*.

Tabela 4.5: Desempenho de *pureML-tec* em projetos *open-source*

Projeto	*precision*	*recall*
cristal	0.9	0.8
exabgp	0.8	0.8
POCS	0.9	0.9

O baixo desempenho de *pureML-tec* em algumas categorias influenciaram a média de desempenho total da técnica. Observou-se que o desempenho de *pureML-tec* nessas categorias foi baixo pelos seguintes motivos: (*i*) poucos bugs nessas categorias para treinar o algoritmo de aprendizagem de máquina; (*ii*) os bugs classificados nessa categoria eram heterogêneos, de forma que não existiam elementos textuais relevantes em comum entre bugs da mesma categoria para que o algoritmo KNN conseguisse estabelecer um padrão, de forma eficiente, para os bugs dessas categorias. Em resumo, no projeto ePol, havia muitas categorias de bugs mas poucos em determinas categorias, enquanto nos projetos *open-source* havia poucas categorias, mas para cada categoria havia muitos bugs.

Observando o desempenho de *pureML-tec* nos projetos *open-source*, nota-se que seu desempenho foi mais baixo no projeto *exabgp* (*precision* = 0.8, *recall* = 0.8), muito embora esse projeto foi o de onde extraiu-se o maior número de bugs (26 bugs). Nos três projetos tem-se poucos bugs de fato classificados como relacionados a máquina de estados. (6 bugs no cristal, 5 no *exabgp*, 4 no *POCS*), contudo a descrição dos bugs nos projetos cristal e *POCS* são mais objetivas. No projeto *exabgp* tem-se uma média de 200 palavras para cada bug, enquanto no projeto cristal essa média é de 20 palavras e no projeto *POCS* a média é de 26 palavras.

Além de todos os projetos *open-source* disporem de poucos bugs para o treinamento do algoritmo de aprendizado de máquina, o projeto *exabgp* contém ainda mais um fator pesando contra o seu desempenho: seus bugs possuem muitos termos irrelevantes (e que não são considerados *stop-words*). O número elevado de termos acaba afetando negativamente o desempenho do algoritmo KNN, fazendo com que o algoritmo não consiga, de forma tão eficiente, estabelecer um padrão para os bugs de cada categoria. Apesar desse fator, a técnica *pureML-tec* atinge um valor alto de *precision* e *recall* (0.8).

Considerando a nuance observada no projeto *exabgp* durante a avaliação de *pureML-*

tec, decidimos verificar se o desempenho do algoritmo poderia ser melhorado, fazendo com que termos diretamente relacionados com elementos das máquinas de estados do projeto se apresentassem maior influência (maior peso) durante a fase de treino do algoritmo KNN. Chegou-se assim a técnica *ml+ir-tec*, onde as descrições de bugs que contém termos da máquina de estados do projeto, recebem um terno indicador de classificação. Os resultados do experimento para avaliar *ml+ir-tec* mostram que não houve diferença significativa entre os melhores casos de *pureML-tec* e *ml+ir-tec*. De fato, o valor de *precision* e *recall* foram os mesmos alcançados por *pureML-tec*.

Para verificar se, de fato, não há diferença significativa entre os desempenhos de *pureML-tec* e *ml+ir-tec*, executamos ambas as técnicas escolhendo aleatoriamente os conjuntos de treinamento. Nessa execução, escolhemos aleatoriamente 100 vezes o conjunto de treinamento e teste para cada projeto *open-source* e calculamos o desempenho do algoritmo a cada rodada. Como resultado, obtivemos o desempenho médio de ambas as técnicas (*pureML-tec* e *ml+ir-tec*) em cada projeto.

Os resultados mostram que o desempenho médio entre *pureML-tec* e *ml+ir-tec* são semelhantes, mostrando que não houve ganho no desempenho de *ml+ir-tec* em comparação a *pureML-tec*.

4.2.4 Ameaças à validade e limitações

Neste estudo identificamos três ameaças à validade do experimentos realizados, listadas a seguir:

Ameaça de validade interna, imitação de tratamento: No experimento para avaliar irBased-tec, existe a possibilidade do sujeito ter observado como outro fez a classificação manual de bugs, podendo influenciar na classificação manual. Cada sujeito recebeu um conjunto de bugs distintos, contudo não se pôde controlar a comunicação entre os sujeitos, pois eles ficaram livres para escolher o horário e local para realizar a classificação.

Ameaça de validade de constructo, viés mono-método: As métricas utilizadas para verificar a eficácia da técnica proposta neste estudo foram *precision* e *recall*. Como eficiência é um conceito abstrato, talvez existam outras formas de medi-la, o que pode trazer uma ameaça de viés-mono método para o estudo, mesmo que as métricas *precision* e *recall* sejam universalmente aceitas para verificar eficiência de algoritmos no que diz respeito à recuperação da

informação.

Ameaça de validade externa, interação de seleção e tratamento: As amostras utilizadas (os bugs) foram extraídas de projetos de software específicos. Logo, as amostras são representativas da população desses projetos de software, e não é possível generalizar as conclusões deste estudo para além da população dos projetos de software aqui tratados. Especificidade das variáveis: Tamanho da base de dados utilizada pode não ter sido suficiente para realizar o treinamento dos algoritmos baseados em aprendizagem de máquina. As bases de dados encontradas não contemplavam todas as categorias da taxonomia de forma homogênea, esse fato pode ter afetado o desempenho das técnicas baseadas em aprendizado de máquina. A técnica de recuperação da informação pode ter influenciado negativamente nos algoritmos de classificação durante a construção dos corpus pois bugs relacionados à máquina de estados podem não ter sido recuperados, contribuindo também para o problema do tamanho da base de dados.

4.2.5 Sugestões para novos experimentos

Durante a análise dos experimentos percebeu-se que algumas modificações poderiam ser feitas ao experimento com o objetivo de verificar a eficiência dos algoritmos classificadores. Uma sugestão para um novo experimento seria utilizar o corpus dos projetos *open-source* para verificar a eficiência do algoritmo KNN em *ml+ir-tec*, variando os fatores *k* e *proporção do conjunto treinamento/teste*. Por limitações de tempo e pela razão de que *ml+ir-tec* mostrou desempenho estatisticamente semelhante a *pureML-tec*, não executamos experimentos considerando a interação entre os fatores *k* e *proporção do conjunto treinamento/teste*.

Outra sugestão de experimento é utilizar todos os bugs classificados do projetos *open-source* como treinamento para as técnicas de classificação e usar as técnicas para classificar bugs com *status* = *open* para sugerir as causas de bugs não resolvidos para a equipe de desenvolvimento. O objetivo dessa sugestão é obter *feedback* da comunidade de projetos *open-source* sobre a técnica.

Capítulo 5

Trabalhos Relacionados

Para o estudo do estado da arte realizamos uma revisão da literatura procurando por trabalhos que relacionem máquina de estados com a tarefa de depuração. O protocolo utilizado para realizar a revisão da literatura, os motores de busca utilizados e a quantidade de trabalhos encontrados se encontram no Apêndice D.

Máquinas de estado podem ser utilizadas para verificação de inconsistências no sistema. Existem várias abordagens na literatura para se alcançar esse objetivo [16, 37, 13, 38, 14, 15, 10], elas variam de acordo com uma série de fatores, dentre eles:

- Disponibilidade da máquina de estados do software;
- Impossibilidade de acessar o *log* do sistema;
- Necessidade de instrumentar o software;
- A linguagem de programação do software;
- Presença de paralelismo no software.

Li et al. [38] utiliza uma abordagem que leva em conta um software que possui disponível a máquina de estado, mas que não precisa que o software a ser analisado seja previamente instrumentado. Desenvolveu-se um algoritmo para automaticamente instrumentar o software e outro algoritmo para gerar casos de testes aleatórios para gerar o trace de execução do software. Assim, a solução proposta instrumenta o software, gera o seu trace de execução e compara a consistência com a máquina de estados fornecida. Essa solução compara a

consistência da máquina de estados da especificação com a máquina de estados inferida, buscando divergências entre elas. Os autores afirmam que a abordagem pode ser utilizada para detectar bugs, máquinas de estado inconsistentes, construídas a partir de engenharia reversa, de sistemas legados.

Beschastnikh et al. [14] propõe uma solução para gerar um modelo dito semelhante a máquina de estados a partir do trace de execução do software a ser analisado, sem a necessidade de instrumentar o código-fonte analisado. O objetivo do trabalho é facilitar a tarefa de inspeção de logs de sistemas não concorrentes. A solução proposta é uma ferramenta, o *Synoptic*, e é dedicado para sistemas não concorrentes escritos em *java*. O software a ser analisado pela ferramenta precisa ser previamente instrumentado. A ferramenta permite que o usuário especifique uma expressão regular para analisar o *log* do software e gerar um modelo de grafos da execução do programa.

Em outro trabalho, Beschastnikh et al. [13] desenvolve o *CSight*, uma ferramenta que estende os algoritmos da técnica proposta no parágrafo anterior para atuar em sistemas concorrentes. O software a ser analisado pela solução dos autores necessita estar previamente instrumentado e cada evento do arquivo de *log* necessita conter um vetor com data e hora da ocorrência do evento. Isso se faz necessário por se tratar de análise de sistemas concorrentes, assim utilizar a data e hora em que o evento ocorreu é uma forma que os autores encontraram para garantir a ordem parcial dos eventos. A solução proposta se embasa na ordem parcial dos eventos minerados para garantir a precisão dos modelos inferidos.

Lorenzoli et al. [16] propõem uma técnica para comparar modelo do software implementado com o modelo do software especificado, permitindo assim verificar divergências entre os modelos. Para validação da técnica os autores desenvolveram o *GK-Tail* uma ferramenta para verificar relações entre os dados gerados por um software em execução e os dados da máquina de estados do documento de requisitos do software. A técnica gera uma máquina de estados estendida que contém os *traces* de execução do software analisado.

O estudo proposto por Mariani, Pezze e Santoro [39] estende a técnica proposta por Lorenzoli et al. [16] tendo como resultado o *GK-Tail++*. O que motivou esse trabalho foi a baixa performance/escalabilidade do *GK-Tail* na análise de sistemas de grande porte, requerendo horas de processamento para a produção de um modelo e por consequência tornando o uso do *GK-Tail* inviável. O problema foi resolvido com um refinamento do algoritmo pro-

posto por Lorenzoli et al, e como resultado do estudo obtiveram um ganho de performance de 50% a %75 comparado com o *GK-Tail*.

No trabalho de Aarts et al. [10] é proposta uma abordagem que estende a máquina de estado finito com o conceito de máquina de estado mínima. O objetivo é abstrair alguns dos comportamentos do software analisado, para assim facilitar o entendimento de algum comportamento em específico do software enquanto se oculta outros comportamentos. Utilizando um algoritmo que combina técnicas de *machine learning*, observação de comportamentos de componentes, e inferência de máquina de estados, a técnica analisa as máquinas de estados do software para realizar as abstrações.

A abordagem de Cui et al. [15] propõe reconstruir o estado do programa que finalizou de forma inesperada a partir de *dumps* de memória do sistema para fazer uma triagem do relatório de falhas (*crash report*). No crash report há informação do *dump* de memória do sistema quando o programa quebrou e uma máquina de estados finitos parcial do sistema. A solução tenta reconstruir a máquina de estados finitos para fazer uma análise completa do sistema e poder catalogar o bug reportado.

A análise de aspectos comportamentais de software permite ainda a detecção automática de *bugs*, realizando busca por comportamentos ou estados que induzem o software a um impasse (por exemplo: *deadlocks*, *loop* infinito). Tal análise possui vantagens e desvantagens em relação a outras técnicas, por exemplo: uma desvantagem é que esse tipo de análise requer a execução do programa e tipicamente isso implica em instrumentação do software para registrar propriedades do estado do software em tempo de execução [11]. Além disso, algumas técnicas precisam que o software seja instrumentado de forma manual, onde o desenvolvedor precisa inserir anotações no código [37][13] o que implica em um custo adicional de recurso por parte da equipe de desenvolvimento.

Embora os trabalhos encontrados utilizem máquina de estados para detectar *bugs* e anomalias em software, nenhum dos trabalhos propõe fazer a análise de *bugs* já reportados com relação a máquina de estados. A ideia de determinar a relação entre *bugs* reportados e elementos da máquina de estados aborda um problema real e pode facilitar a tarefa de depuração de software. O trabalho que propomos tem como objetivo investigar e propor uma solução contemplando esse gap no estado da arte, desenvolvendo uma técnica para classificação automática de *bugs* sob a perspectiva de máquina de estados.

Capítulo 6

Conclusões

No trabalho foi apresentada uma técnica para classificação automática de bugs relacionados a máquina de estados. A técnica é constituída pela taxonomia para classificação de bugs e pelo algoritmo classificador. Para elaboração da taxonomia foi preciso ponderar entre os níveis de detalhamento dos cenários de cada categoria, sendo necessária uma avaliação qualitativa para realizar os devidos ajustes. O algoritmo classificador se mostrou mais eficaz quando utilizou o algoritmo de aprendizagem de máquina KNN e mostrou que de fato pode ser utilizado para classificação automática de bugs.

A técnica proposta neste trabalho está limitada a classificar bugs de acordo com a taxonomia de bugs sob a perspectiva de máquina de estados. Os bugs que a técnica proposta analisa devem ser de softwares que utilizam máquina de estados em sua documentação. A técnica proposta necessita de um conjunto de treinamento, que consiste em um conjunto de bugs previamente classificados.

As contribuições desse trabalho foram:

- Uma técnica para classificação automática de bugs relacionados a máquina de estados;
- Taxonomia para classificação de bugs relacionados a máquina de estados;
- Corpus com 64 bugs *opensource* classificados de acordo com a taxonomia de bugs[1];
- Experimento apontando que determinados algoritmos puramente baseados em recuperação da informação apresentaram baixo desempenho para a tarefa de classificação de

[1]Disponível em: https://github.com/melqui-andrade/bug-classification

bugs no contexto deste trabalho.

Para trabalhos futuros, almejamos:

- Utilizar outras técnicas de aprendizado de máquina para o algoritmo classificador e avaliar o seu desempenho;
- Implementar a funcionalidade de coletar *feedback* das resoluções dos bugs para que a ferramenta automaticamente utilize o *feedback* no conjunto treinamento, dispensando assim a necessidade de se possuir um conjunto treinamento inicial para utilizar a técnica de classificação automática;
- Utilizar os bugs coletados dos projetos *opensource* utilizados no experimento para classificar bugs ainda sem resolução. Enviar a classificação feita pela técnica para as equipes de desenvolvimento dos projetos e obter seus *feedbacks*.

Bibliografia

[1] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.

[2] Edsger W Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[3] OMG Object Management Group. Unified modeling language^{TM}(uml®) version 2.5, 2015.

[4] Ian SOMMERVILLE. *Software Engineering. Eight Edition. 824 s*. Harlow: Pearson Education Limited, 8 edition, 2007.

[5] Judith A Clapp. *Software quality control, error analysis, and testing*. William Andrew, 1995.

[6] Robert N Charette. Why software fails [software failure]. *Ieee Spectrum*, 42(9):42–49, 2005.

[7] IEEE Standards Committee et al. Ieee std 610.12-1990 ieee standard glossary of software engineering terminology. *online] http://st-dards. ieee. org/reading/ieee/stdpublic/description/se/610.12-1990 desc. html*, 1990.

[8] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[9] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. *University of Cambridge-Judge Business School, Tech. Rep*, 2013.

[10] Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits Vaandrager. Automata learning through counterexample guided abstraction refinement. In *International Symposium on Formal Methods*, pages 10–27. Springer, 2012.

[11] Thomas Ball. The concept of dynamic analysis. In *Software Engineering—ESEC/FSE'99*, pages 216–234. Springer, 1999.

[12] Nachi Nagappan Brendan Murphy Philip J. Guo, Tom Zimmermann. Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In *Proceedings of the 32th International Conference on Software Engineering (ICSE)*. Association for Computing Machinery, Inc., May 2010.

[13] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering*, pages 468–479. ACM, 2014.

[14] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 267–277. ACM, 2011.

[15] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P Kemerlis. Retracer: triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering*, pages 820–831. ACM, 2016.

[16] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*, pages 501–510. ACM, 2008.

[17] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[18] William B Frakes and Ricardo Baeza-Yates. *Information retrieval: data structures and algorithms*. Prentice Hall PTR, 1992.

[19] Apache Jakarta. Apache lucene-a high-performance, full-featured text search engine library. *Apache Lucene*, 2004.

[20] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[21] Cyril Goutte and Eric Gaussier. A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In *European Conference on Information Retrieval*, pages 345–359. Springer, 2005.

[22] Yakov Shafranovich. Common format and mime type for comma-separated values (csv) files. 2005.

[23] S Liu, Y Liu, É André, C Choppy, J Sun, B Wadhda, and JS Dong. A formal semantics for the complete syntax of uml state machines with communications. In *Proceedings of the 10th International Conference on Integrated Formal Methods, IFM*, volume 38613, pages 8–23, 2013.

[24] Scott Galloway. *The four: the hidden DNA of Amazon, Apple, Facebook and Google*. Random House, 2017.

[25] John D Kelleher, Brian Mac Namee, and Aoife D'arcy. *Fundamentals of machine learning for predictive data analytics: algorithms, worked examples, and case studies*. MIT Press, 2015.

[26] Ryszard S Michalski, Jaime G Carbonell, and Tom M Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[27] Carl E Landwehr, Alan R Bull, John P McDermott, and William S Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys (CSUR)*, 26(3):211–254, 1994.

[28] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ACM SIGPLAN Notices*, volume 51, pages 517–530. ACM, 2016.

[29] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The java language specification (java se 8 edition). *Sun Microsystems*, 2015.

[30] OMG XMI. Omg xmi specification, v1. 2, 2002.

[31] Thuy TT Nguyen and Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials*, 10(4):56–76, 2008.

[32] Victor R Basili-Gianluigi Caldiera and H Dieter Rombach. Goal question metric paradigm. *Encyclopedia of software engineering*, 1:528–532, 1994.

[33] José A Cruz-Lemus, Ann Maes, Marcela Genero, Geert Poels, and Mario Piattini. The impact of structural complexity on the understandability of uml statechart diagrams. *Information Sciences*, 180(11):2209–2220, 2010.

[34] J.L. Myers, A.D. Well, and R.F. Lorch. *Research Design and Statistical Analysis: Third Edition*. Taylor & Francis, 2013.

[35] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, page 23. ACM, 2008.

[36] Ron Kohavi. Scaling up the accuracy of naive-bayes classifiers: a decision-tree hybrid. In *KDD*, volume 96, pages 202–207. Citeseer, 1996.

[37] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[38] Xuandong Li, Xiaokang Qiu, Linzhang Wang, Bin Lei, and W Eric Wong. Uml state machine diagram driven runtime verification of java programs for message interaction consistency. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 384–389. ACM, 2008.

[39] Leonardo Mariani, Mauro Pezze, and Mauro Santoro. Gk-tail+ an efficient approach to learn software models. *IEEE Transactions on Software Engineering*, 2016.

Apêndice A

Questionário para avaliação da taxonomia

Pesquisa - Taxonomia de bugs

Visão Geral

Esta pesquisa é conduzida por Melquisedec Lima, um pesquisador da Universidade Federal de Campina Grande.
O objetivo desta pesquisa é avaliar a Taxonomia de bugs sob a perspectiva de máquina de estados. Especificamente, deseja-se avaliar a utilidade e facilidade de uso da taxonomia considerando-se suas classes e propriedades, verificando se a taxonomia possui potencial para ajudar no debugging de software.

Observação: Todos os dados providos nesta pesquisa serão utilizados apenas para propósitos acadêmicos. Nenhuma informação pessoal será divulgada e sua privacidade está garantida.

* Required

1. **Nome** *

2. **Termo de consentimento** *
   Check all that apply.
   - [ ] Eu li e aceito participar desta pesquisa

Questionário

3. **A taxonomia contempla as categorias necessárias para classificar bugs sob a perspectiva de máquina de estados?** *
   Mark only one oval.
   - ◯ Sim
   - ◯ Não

4. **(Responder caso a resposta da questão anterior tenha sido "não") Que categoria você sentiu falta na taxonomia de bugs?**

5. **Em uma escala de 1 a 5, onde 1 representa "muito clara" e 5 representa "muito confusa", como você classifica a descrição dos cenários de cada categoria?** *
   Mark only one oval.

	1	2	3	4	5	
muito clara	◯	◯	◯	◯	◯	muito confusa

6. **Em uma escala de 1 a 5, onde 1 representa "falta de cenários" e 5 representa "excesso de cenários", como você classifica o número de cenários da taxonomia? (note que: 3 representa que o número de cenários está ok)** *
Mark only one oval.

	1	2	3	4	5	
falta de cenários	◯	◯	◯	◯	◯	excesso de cenários

Apêndice B

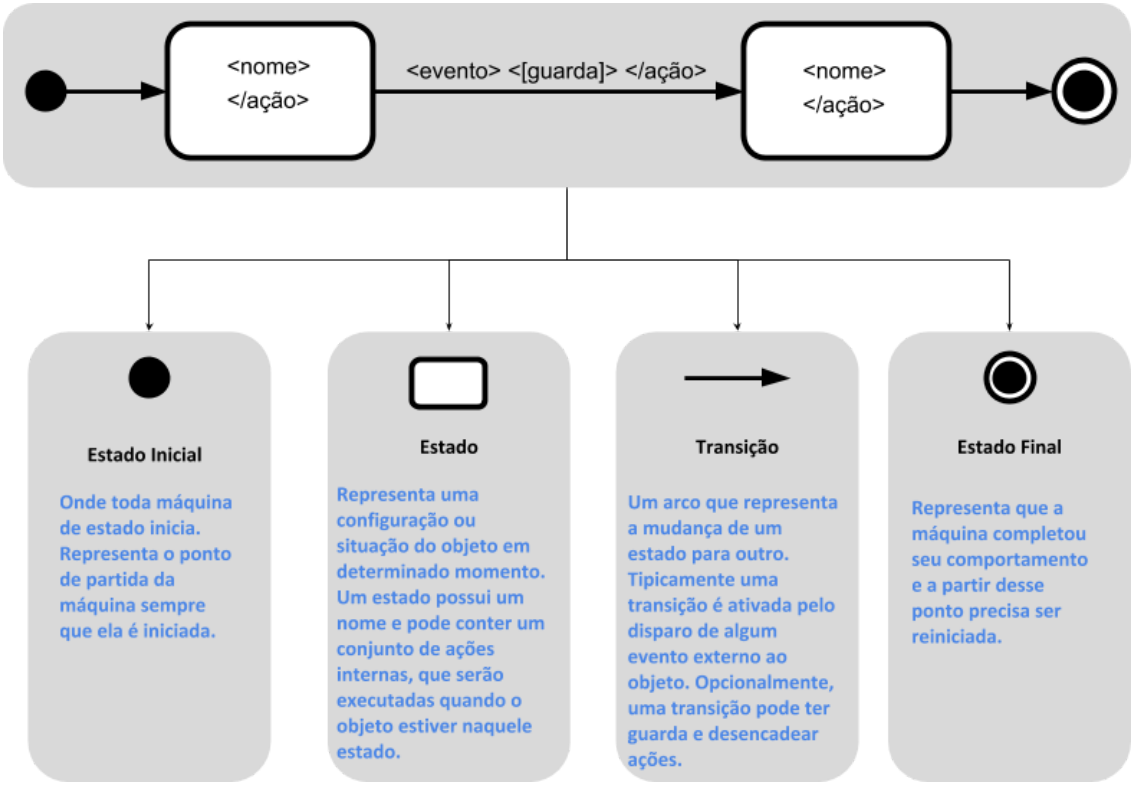
Treinamento análise de bugs

Treinamento análise de bugs

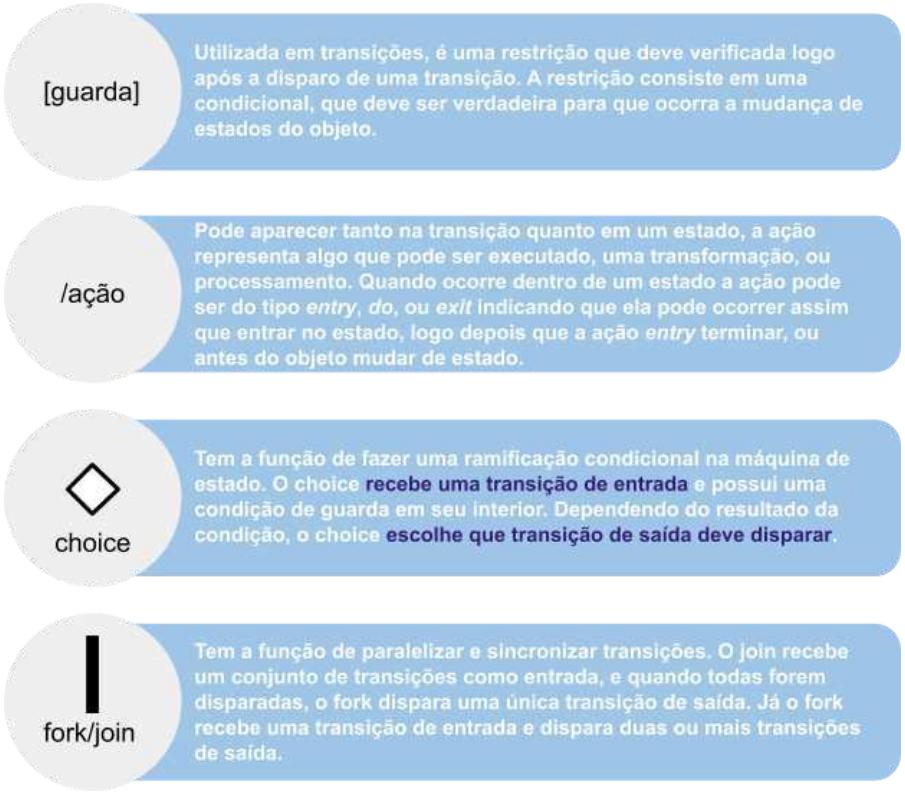
Sobre máquinas de estados

Máquina de estados é um modelo que descreve como um objeto individual muda sua configuração (estado) em resposta aos eventos que ocorrem no meio ao qual o objeto está inserido. Máquina de estados é um modelo UML do tipo dinâmico.

Uma máquina de estados é composta tipicamente por estados e transições. Para entender melhor o que significa cada elemento da máquina de estados, veja a figura abaixo:



Além desses elementos, a máquina de estados pode conter ainda:



Sobre a taxonomia de bugs

O objetivo da taxonomia de bugs é ajudar o desenvolvedor a identificar mais rapidamente, a causa de bugs relacionados a máquina de estados. A taxonomia foi desenvolvida pensando-se nos tipos de elementos que compõem uma máquina de estados, assim a taxonomia conta com oito categorias, onde cada categoria corresponde a um tipo de elemento da máquina de estado.
A tabela a seguir mostra a taxonomia em si:

Classificação	Sigla	Cenários
Bug on State	BSt	Trigger dispara, estado não muda
		Trigger não dispara, estado muda
		Guarda satisfeita, estado não muda
		Trigger dispara, guarda satisfeita, estado não muda
		Trigger dispara, guarda não satisfeita, estado muda
Bug on Guard	BGd	Trigger não dispara, guarda é satisfeita, atividade da transição é executada, estado muda
		Trigger não dispara, guarda não é satisfeita, atividade da transição é executada, estado muda
Bug on Activity	BAc	Atividade interna não é executada
		Atividade da transição não executada
		atividade de transição é executada quando não deveria ser
		Trigger dispara, atividade de transição não é executada
		Trigger não dispara, atividade de transição é executada
Bug on Entry	BEn	Estado é alcançado, entry action não executa
		Estado não é alcançado, entry action executa
		Estado é alcançado, entry action executa, mas não na ordem esperada
Bug on Exit	BEx	Saída do estado e exit action não executa
		exit action não executa e sai do estado
Bug on Join	BJn	Todas as transições de chegada foram completadas, transição de saída não executa
		Pelo menos uma transição de chegada não foi completada, transição de saída executa
Bug on Fork	BFk	A transição de chegada foi completada, pelo menos uma das transições de saída não é executada
		A transição de chegada não foi completada, pelo menos uma das transições de saída é executada
Bug on Choice	BCh	A transição de chegada foi completada, a guarda foi satisteita, transição eperada não ocorre
		A transição de chegada não foi completada, a guarda foi satisteita, pelo menos uma das transições de saída é executada

		A transição de chegada foi completada, o fluxo deveria serguir o else, a transição de saída do else não é executada

A ideia da taxonomia é identificar a que elemento da máquina de estados o bug e está relacionado e qual é o possível problema que está acontecendo na máquina de estados. A lista de problemas que pode ocorrer corresponde aos cenários da tabela da taxonomia. Cada categoria possui uma série de cenários descrevendo comportamentos que não são esperados para o elemento daquela categoria.

Exemplo de uso

A fim de exemplificar os conceitos apresentados vamos mostrar um exemplo de máquina de estados e um possível bug reportado. Seja a máquina de estados da Figura 1 uma máquina representando o comportamento de um software que notifica a todos os interessados quando o usuário obtém acesso aos registros do software.

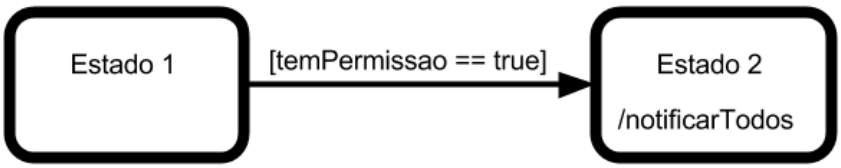


Figura 1. Máquina de estados do sistema de notificação sobre acesso.

O seguinte bug foi reportado:

Título do bug	Descrição do bug
O sistema não executa a ação notifyAll quando deveria	Após o usuário obter a permissão para prosseguir, o sistema não exibe a mensagem de acesso liberado para os demais.

Pela descrição do bug vemos que a condição de guarda é satisfeita pois a descrição “Após o usuário obter a permissão” nos leva a pensar que temPermissao == true.
O problema aqui então pode ser que **a condição de guarda é satisfeita mas o sistema não muda de estado** (State 1 para State 2) e assim não chega em State 2 para executar a ação, ou **o sistema atinge State 2 mas não executa a ação**. Assim esse bug pode ser classificado:

Bug on State	Trigger dispara, guarda satisfeita, estado não muda
Bug on Activity	Atividade interna não é executada

Apêndice C

Formulário disponibilizado para o sujeito durante experimento

An®lise manual de bugs

Vis² o Geral
Esta pesquisa ¶ conduzida por Melquisedec Lima, um pesquisador da Universidade Federal de Campina Grande.
O objetivo desta pesquisa ¶:
1) classificar bugs de acordo com a Taxonomia de bugs proposta. Especificamente, identificar se um bug est®ou n² o
relacionado a uma m®quina de estados, e, caso esteja, classific®-lo em uma das oito categorias da Taxonomia de bugs
proposta, sob a perspectiva de m®quina de estados.
2) avaliar a Taxonomia de bugs sob a perspectiva de m®quina de estados. Especificamente, deseja-se avaliar a utilidade e
facilidade de uso da taxonomia considerando-se suas classes e propriedades, verificando se a taxonomia possui
potencial para ajudar no debugging de software.
Para alcan´ar os objetivos acima, montamos este survey que est®dividido em duas partes, onde a primeira parte diz respeito a an®lise manual de bugs, e a segunda parte ¯ avalia´² o da taxonomia de bugs.
Termos
A an®lise manual e avalia´² o da taxonomia s² o realizadas por volunt®rios do projeto ePol e a qualquer momento o volunt®rio pode desistir de responder o survey sem qualquer ×nus para si.
Todos os dados providos neste formul®rio ser² o utilizados apenas para propÆsitos acad¸micos.
Nenhuma informa´² o pessoal ser®divulgada e sua privacidade est®garantida.
O participante da pesquisa pode solicitar uma cÆpia de sua resposta, ou tirar qualquer dßvida atrav¶s do contato: melquisedec@copin.ufcg.edu.br

*Required

1. Termo de consentimento *
Check all that apply.

- [ ] Li e aceito participar desta an®lise

An®lise de bugs: Instru´Þes

1. Para realizar a an®lise manual de bugs, o volunt®rio deve ter passado pelo treinamento sobre classifica´² o de bugs sob a perspectiva de m®quina de estados, onde ser®apresentada a taxonomia de bugs. Caso tenha alguma dßvida, ou precise relembrar algo, os links a seguir podem ajudar:
Sequ¸ncia de passos que devem ser seguidos: https://goo.gl/c71To3
Sobre a taxonomia de bugs: https://goo.gl/D1RMm2 .
Sobre a an®lise de bugs: https://goo.gl/LdL9ND

2. Para cada bug da lista de bugs existe uma quest² o do tipo checkbox neste formul®rio contendo em sua descri´² o o identificador do bug, e dez op´Þes para classificar o bug, sendo uma op´² o para classific®-lo como "n² o relacionado como m®quina de estados" (as demais correspondem ¯s oito categorias da Taxonomia de bugs). Existe ainda uma d¶cima op´² o (outro) que deve ser marcada caso o bug analisado n² o se enquadre em nenhuma das op´Þes anteriores.

3. O Volunt®rio deve escolher em quais das op´Þes o bug analisado se enquadra.

4. O prazo m®ximo para envio do formul®rio ¶ at¶ o dia 17/11/2017.

5. Ao final o volunt®rio deve submeter o formul®rio com todos os bugs analisados.

An®lise dos bugs

Link para lista de bugs para analisar: https://goo.gl/NEC35A

Sequência de passos que devem ser seguidos: https://goo.gl/c71To3
Sobre a taxonomia de bugs: https://goo.gl/D1RMm2 .
Sobre a análise de bugs: https://goo.gl/LdL9ND

2. Bug 1
Check all that apply.

- [ ] Bug não relacionado a máquina de estado
- [ ] Bug of state
- [ ] Bug of guard
- [ ] Bug of activity
- [ ] Bug of entry
- [ ] Bug of exit
- [ ] Bug of join
- [ ] Bug of fork
- [ ] Bug of choice
- [ ] Other: _____

3. Bug 2
Check all that apply.

- [ ] Bug não relacionado a máquina de estado
- [ ] Bug of state
- [ ] Bug of guard
- [ ] Bug of activity
- [ ] Bug of entry
- [ ] Bug of exit
- [ ] Bug of join
- [ ] Bug of fork
- [ ] Bug of choice
- [ ] Other: _____

4. Bug 3
Check all that apply.

- [ ] Bug não relacionado a máquina de estado
- [ ] Bug of state
- [ ] Bug of guard
- [ ] Bug of activity
- [ ] Bug of entry
- [ ] Bug of exit
- [ ] Bug of join
- [ ] Bug of fork
- [ ] Bug of choice
- [ ] Other: _____

5. Bug 4
Check all that apply.

- [ ] Bug não relacionado a máquina de estado
- [ ] Bug of state
- [ ] Bug of guard
- [ ] Bug of activity
- [ ] Bug of entry
- [ ] Bug of exit
- [ ] Bug of join
- [ ] Bug of fork
- [ ] Bug of choice
- [ ] Other: _____

6. Bug 5
Check all that apply.

- [ ] Bug não relacionado a máquina de estado
- [ ] Bug of state
- [ ] Bug of guard
- [ ] Bug of activity
- [ ] Bug of entry
- [ ] Bug of exit
- [ ] Bug of join
- [ ] Bug of fork
- [ ] Bug of choice
- [ ] Other: _____

7. Bug 6
Check all that apply.

- [ ] Bug n² o relacionado a m®quina de estado
- [ ] Bug of state
- [ ] Bug of guard
- [ ] Bug of activity
- [ ] Bug of entry
- [ ] Bug of exit
- [ ] Bug of join
- [ ] Bug of fork
- [ ] Bug of choice
- [ ] Other: _____

8. Ir para avalia´²o da taxonomia *
Check all that apply.

- [ ] Continuar

Avalia´²o da taxonomia de bugs

9. A taxonomia contempla as categorias necess®rias para classificar bugs sob a perspectiva de m®quina de estados?
Mark only one oval.

- ◯ Sim
- ◯ N² o

10. (Responder caso a resposta da quest² o anterior tenha sido "n² o") Que categoria voc¸ sentiu falta na taxonomia de bugs?

_____

11. Em uma escala de 1 a 5, onde 1 representa "muito clara" e 5 representa "muito confusa", como voc¸ classifica a descri´²o dos cen®rios de cada categoria?
Mark only one oval.

	1	2	3	4	5	
muito clara	◯	◯	◯	◯	◯	muito confusa

12. Em uma escala de 1 a 5, onde 1 representa "falta de cenários" e 5 representa "excesso de cenários", como você classifica o número de cenários da taxonomia? (note que: 3 representa que o número de cenários está ok)
Mark only one oval.

	1	2	3	4	5	
falta de cenários	◯	◯	◯	◯	◯	excesso de cenários

Apêndice D

Strings de busca utilizadas na revisão da literatura

1 String de busca utilizada na revisão da literatura

- String 1: ((identify OR recognize OR verify OR mapping) AND (state machine OR finite-state machine OR finite-state) AND (bugs OR error OR fault));
- String 2: ((extract OR obtain or get or infer) AND (state machine OR finite-state machine OR finite-state) AND (software code OR code OR software)).

2 Busca por trabalhos relacionados

O resultado visto em Tabela 1 foi obtido através das duas strings de busca, e foram utilizadas em três motores de busca.

Tabela 1: Quantidade de artigos retornados nos motores de busca utilizados na revisão sistemática.

	ACM Digital Library	IEEE eXplore	SCOPUS
Artigos encontrados	84	17	6

Os motores de busca utilizados foram os seguintes:

- ACM Digital Library (`portal.acm.org`);
- IEEE eXplore (`ieeexplore.ieee.org`);
- SCOPUS (`scopus.com`).

Para que determinado artigo fosse incluído ou excluído do grupo de resultados obtidos foram adotados critérios de inclusão e exclusão de artigos. Os critérios de inclusão foram dois e consistem em: O trabalho ajuda a responder as questões de pesquisa; o trabalho foi publicado em conferência ou periódico conhecido(a) nacionalmente ou internacionalmente. O critério de exclusão adotado foi: O artigo não se relacionam com os temas das questões de pesquisa.

A revisão foi divida em três fases para a seleção dos artigos que deveriam ser inclusos no grupo de resultados obtidos, em cada fase verificou-se se o item analisado tinha relação com as questões de pesquisa. Na primeira fase analisou-se o título do artigo, na segunda fase analisou-se as palavras-chave, e na última fase analisou-se o *abstract* do artigo.