João Luiz de Andrade Neto

AUTOTESTCOVERAGE^{C}: UMA FERRAMENTA PARA OBTENÇÃO DE COBERTURA DE CÓDIGO PARA COMPONENTES ANDROID SEM USO DE INSTRUMENTAÇÃO

Trabalho de Graduação

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~graduacao

RECIFE
2018

Universidade Federal de Pernambuco
Centro de Informática
Graduação em Ciência da Computação

João Luiz de Andrade Neto

AUTOTESTCOVERAGE^{C}: UMA FERRAMENTA PARA OBTENÇÃO DE COBERTURA DE CÓDIGO PARA COMPONENTES ANDROID SEM USO DE INSTRUMENTAÇÃO

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Alexandre Cabral Mota*

RECIFE
2018

João Luiz de Andrade Neto

AutoTestCoverage^{C}: Uma ferramenta para obtenção de cobertura de código para componentes Android sem uso de instrumentação/ João Luiz de Andrade Neto. – RECIFE, 2018-

66 p. : il. (algumas color.) ; 30 cm.

Orientador Alexandre Cabral Mota

Trabalho de Graduação – Universidade Federal de Pernambuco, 2018.

1. Cobertuda de Código. 2. Sem Instrumentação de Código. I. Alexandre Mota. II. Universidade Federal de Pernambuco. III. Centro de Informática. IV. AutoTestCoverageC: Uma ferramenta para obtenção de cobertura de código para componentes Android sem uso de instrumentação

CDU 02:141:005.7

Trabalho de graduação apresentado por **João Luiz de Andrade Neto** ao programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **AutoTestCoverage^{C}: Uma ferramenta para obtenção de cobertura de código para componentes Android sem uso de instrumentação**, orientada pelo **Prof. Alexandre Cabral Mota** e aprovada pela banca examinadora formada pelos professores:

Prof. Alexandre Cabral Mota
Centro de Informática/UFPE

Prof. Juliano Manabu Iyoda
Centro de Informática/UFPE

RECIFE
2018

Eu dedico esse trabalho em especial à minha mãe, que sempre acreditou em mim. Também dedico a minha família, amigos e ao meu professor que me deu todo o suporte necessário para a conclusão desse trabalho.

Agradecimentos

Em primeiro, gostaria de agradecer a minha mãe, que sempre por mim lutou e acreditou em cada objetivo que um dia eu tracei para minha felicidade e futuro. Espero que ela possa comemorar essa conquista comigo, onde quer que esteja.

Queria agradecer a todos os familiares, em especial, meu pai que sempre quis um futuro melhor para mim e meu tio Cristiano Andrade, que sempre se preocupou que eu seguisse minha vida, sem perder o foco dos meus objetivos.

Desejo também agradecer ao CITi (Centro Integrado de Tecnologia da Informação) Empresa Júnior do CIn, lugar onde pude aprender muito sobre o mercado e adquirir mais experiência para projetos de vida cada vez maiores.

Agradeço também as pessoas do convênio CIn/Motorola, por estarem sempre dispostas a ajudar e contribuir de alguma forma. Em especial gostaria de agradecer Alice Arashiro, Marlom Jobson, Bazante, Eliot Maia, Guilherme e Virgínia.

Agradeço também Diego Morais, que me ensinou, que eu podia lidar com os problemas da vida de maneira mais tranquila e ter uma vida melhor com isso.

Também gostaria de agradecer aos professores do Centro de Informática que tiveram toda a paciência em compartilhar seus conhecimentos ao longo do curso, da forma que puderem. Em especial Alexandre Mota, Leopoldo e Silvio Melo.

Por fim, aos amigos que ao meu lado estiveram nos momentos mais difíceis e também nos momentos de alegria por cada conquista na vida, sempre me puxando para cima.

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

—MARTIN FOWLER

Resumo

Quando falamos da indústria de desenvolvimento de software, nos dias atuais, logo conseguimos ter em mente que diversas funcionalidades são diariamente implementadas e colocadas em produção para centenas ou milhares de usuários. Isso significa que constantemente várias linhas de código são adicionadas ou modificadas por desenvolvedores. Com esta realidade, se qualquer descuido na fase de testes ocorrer, fica fácil de se deparar com falhas e a degradação da experiência do usuário. Inspirado nisto, este trabalho se propõe a mostrar como desenvolvemos um meio de coletar dados de cobertura sobre aplicações Android, sem o uso de instrumentação de código, e como, através destes dados, pudemos propor uma forma de melhorar a seleção dos testes a serem executados em uma campanha, além de poder reportar sobre a qualidade dos testes executados por nossa parceira industrial. Assim, por transitividade, melhoramos a experiência final do usuário que vai vivenciar menos falhas.

Este trabalho é parte de uma parceira de pesquisa entre o CIn-UFPE e a Motorola Mobility.

Palavras-chave: Cobertura de Código Sem Instrumentação, Teste de Regressão, Seleção de Testes, Qualidade de Testes, Testes em Android.

Abstract

When we talk about the software development industry these days, we soon realize that several features are implemented daily and put into production for hundreds or thousands of users. This means that several lines of code are constantly being added or modified by developers. With this reality, if any carelessness in the testing phase occurs, it is easy to come across faults and degrade the user experience. Inspired by this, we have come to show how we have developed a way to collect coverage data about Android applications, without the use of code instrumentation, and how, through this data, we could propose a way to improve the selection of the tests to be executed in a campaign, besides being able to report on the quality of the tests performed by our industrial partner. Thus, by transitivity, we improve the final user experience that will face fewer bugs.

This work is a collaboration research between CIn-UFPE and Motorola Mobility.

Keywords: Code Coverage Without Instrumentation, Regression Testing, Test Selection, Test Quality, Android Testing.

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

CPU	Unidade Central de Processamento	29
ART	Android Runtime	29
HTTP	Protocolo de Transferência de Hipertexto	38
API	Interface de Programa de Aplicação	37
TBC	To Be Covered	36
MC	Manual Coverage	36
AST	Árvore Sintática Abstrata	31
ADB	Android Debug Bridge	36
SBP	Sample Based Profile	50

Sumário

1

Introdução

Neste capitulo será apresentado o contexto no qual este trabalho está inserido, bem como nossa proposta diante deste cenário, principais objetivos abordados, contribuições gerais e como este documento está estruturado.

1.1 Contexto

Quando falamos da indústria de desenvolvimento de software nos dias atuais, logo conseguimos ter em mente que diversas funcionalidades são diariamente implementadas e colocadas em produção para milhares de usuários. Isso significa que constantemente um número incalculável de linhas de código são modificadas diariamente. Com essa realidade, se um controle de qualidade eficaz não existir, fica fácil se deparar com falhas e diminuição da experiência do usuário ao utilizar um produto. Com o objetivo de identificar os problemas provenientes das modificações no software e evitar que esses erros cheguem até o mercado, empresas executam testes com o objetivo de melhorar a qualidade de seus produtos, bem como a experiência que entregam a seus usuários.

Com a prática de testes diários, galgando a melhoria de qualidade dos produtos, empresas precisam se adaptar ao processo desde a criação, seleção e execução de testes. Isto requer tempo e dinheiro, esperando assim que tal investimento traga um resultado positivo na diminuição dos defeitos escapados[1].

Porém existe outra preocupação quando se investe em um processo de teste, em se tratando da qualidade dos testes executados, que está relacionada com uma forma de medir o quão as campanhas de testes estão exercitando as regiões modificadas. Isto é feito em geral através do cálculo de regiões de código alcançadas durante a execução de testes, chamada cobertura de código. Geralmente a cobertura de código é medida através da instrumentação do código fonte, ou seja, adicionando trechos auxiliares de código, que servem como uma marcação para exibir as regiões que foram exercitadas ao usuário testador. Este tópico será melhor abordado no Capítulo 2. Uma forma simples e já estabelecida de se fazer isto é através

[1] Um defeito escapado é aquele que não foi encontrado pelo time de testes.

de uma ferramenta como a JaCoCo[2].

Entretanto, existem cenários onde o acesso ao código fonte é restrito apenas à leitura, não sendo possível alterar o mesmo, tornando assim o processo de obtenção e análise dos dados de cobertura mais difícil. Este é exatamente o cenário onde o presente trabalho está inserido.

Com o objetivo de obter informações de cobertura de código de formas alternativas que não exijam alteração do código fonte ou geração de múltiplos instaladores de uma mesma aplicação, nossa parceira industrial, a *Motorola Mobility*, propôs o cenário que iremos abordar ao longo deste trabalho. Neste documento mostraremos como viabilizamos as informações de cobertura de código necessárias no processo da parceira sem o uso de instrumentação de código, possuindo apenas acesso de leitura ao código fonte, dispositivos de teste e esforço humano.

1.2 Proposta deste Trabalho

Nossa proposta é identificar os métodos que foram modificados, através do código fonte, e criar uma abordagem que nos permita monitorar os dispositivos enquanto eles são testados. Obtemos assim os métodos da aplicação alvo que foram executados durante um teste, sendo possível calcular a cobertura de código percentual de uma campanha para uma determinada *apk*[3] do Android (em nosso parceiro industrial, esta *apk* está relacionada ao *teste de componente*). Esta proposta foi implementada em uma ferramenta chamada AutoTestCoverage[C], que atualmente se encontra em fase de validação e aperfeiçoamento. O desenvolvimento inicial dessa ferramenta foi dividido em dois módulos principais: (i) *To Be Covered (TBC)*, responsável por calcular os métodos que foram modificados entre diferentes versões de uma aplicação; (ii) *Manual Coverage (MC)*, nos provendo uma forma de monitorar os métodos exercitados em uma determinada execução em um dispositivo Android[4]. Estes módulos serão melhor detalhados no Capítulo 3, objetivando uma melhor compreensão do processo de desenvolvimento e harmonia dos mesmos.

1.3 Objetivos

O objetivo principal desse trabalho é obter os dados de cobertura sobre os testes executados pela nossa parceira, sem o uso de instrumentação de código, utilizando apenas o que nos foi permitido e seguindo os padrões de código da empresa, sendo necessário a homologação de cada funcionalidade implementada, pelo time encarregado de desenvolvimento de pesquisa. Com isso esperamos entregar valor à cliente, à indústria e à comunidade acadêmica.

[2] Biblioteca de cobertura de código em Java com o uso de instrumentação - https://www.eclemma.org/jacoco/
[3] apk - Extensão dada a um instalador do sistema Android.
[4] Site Oficial Android - https://www.android.com/

1.4 Contribuições

As principais contribuições deste trabalho são:

- Um processo que recebe duas versões de uma aplicação como entrada e informa como saída que métodos desta aplicação foram modificados entre estas duas versões;
- Uma ferramenta protótipo (chamada AutoTestCoverage^{C}) onde os conceitos apresentados neste trabalho foram implementados;
- Uma diferente abordagem para se calcular cobertura de código em dispositivos móveis sem o uso de instrumentação de código fonte;
- Alguns experimentos demonstrando a capacidade da abordagem desenvolvida, com diferentes configurações.

1.5 Estrutura do Documento

Este trabalho está organizado da seguinte maneira:

- **Capítulo 2** será introduzido sobre os principais conceitos usados neste trabalho. Teste de regressão, teste de caixa branca, cobertura de código, controle de versão, árvore sintática abstrata e o uso do *profiling*.
- **Capítulo 3** apresentará a solução proposta, bem como seu desenvolvimento, para obtenção dos dados de cobertura sem o uso de instrumentação do código fonte de uma aplicação Android.
- **Capítulo 4** explicará sobre os experimentos que foram executados a fim de comparar diferentes aspectos dentro da solução proposta.
- **Capítulo 5** será discutido a conclusão do trabalho, bem como os trabalhos relacionados e trabalhos futuros.

2

Conhecimento

Este capítulo apresenta de forma rápida alguns conceitos básicos relacionados aos seguintes tópicos: Testes de Regressão, Abordagem Híbrida Caixa Preta e Branca, Criação de Perfil (*Profiling*) de Sistemas, Cobertura de Código, Controle de Versão e Árvore Sintática Abstrata, com foco no que foi utilizado no presente trabalho.

2.1 Testes de Regressão

Segundo ROTHERMEL; HARROLD (1997), teste de regressão é uma custosa mas necessária atividade de teste, exercida em softwares modificados, para garantir que tais modificações estão corretas e não irão afetar outras regiões do código já funcionais e previamente validadas.

Este tipo de teste é rotineiramente empregado pelo nosso parceiro industrial, representando longas e complexas execuções de testes. Com isto, este trabalho considerou este tipo de teste como de maior prioridade (foco principal do presente trabalho), sem que os demais tipos de testes fossem descartados.

É importante explicar também que nosso parceiro realiza campanhas de testes de regressão reduzidas, seguindo um dos critérios relatados pelo trabalho ROTHERMEL; HARROLD (1997), não exercitando todos os testes existentes em sua base, por motivos de tempo e todo o esforço humano que seria necessário para tal. Para evitar a reexecução de todos os testes, também conhecida na literatura pelo termo *Retest All*, utiliza-se uma ferramenta para seleção e priorização de casos de teste, baseado em modificações do software testado, como pode ser visto em MAGALHãES et al. (2017), que também foi desenvolvida em outro trabalho com nosso parceiro. O presente trabalho foi utilizado como forma de avaliar a qualidade da seleção e priorização do trabalho MAGALHãES et al. (2017), através do percentual de cobertura de código atingido na execução dos testes selecionados em relação às regiões que sofreram modificações.

2.2 Abordagem Híbrida Caixa Preta e Branca

Em ambientes industriais voltados para desenvolvimento de dispositivos móveis, é comum se deparar com diferentes times de teste, responsáveis por diferentes validações após a etapa de desenvolvimento. O processo de desenvolvimento de testes por nossa parceira segue o modelo de caixa preta, onde o arquiteto de teste não tem conhecimento do código fonte, mas sabe quais ações ou entradas, devem ser executadas no dispositivo para testar uma determinada função, e quais reações, ou saídas, deve esperar para tais ações. Esse tipo de teste é frequentemente presente em trabalhos envolvendo a plataforma Android, como em YEH; HUANG; CHANG (2013) e AMALFITANO; FASOLINO; TRAMONTANA (2011) que criam diferentes abordagens para execução de testes.

Dispondo do conhecimento necessário a respeito dos testes criados por nossa parceira e sua necessidade pela informação de cobertura de código diante das limitações expostas, foi necessário criar uma abordagem que permitisse coletar informações de cobertura do código que é executado no dispositivo através da execução de testes caixa preta (mais sobre a abordagem de monitoramento na Seção 2.3), e ao mesmo tempo uma maneira de obter as regiões de código modificadas entre diferentes versões de uma determinada aplicação, para que assim seja possível concretizar o cálculo da cobertura de código.

Com o objetivo de auxiliar na concretização do trabalho proposto na Seção 1.2, a parceira nos disponibilizou acesso de leitura ao código fonte de algumas aplicações, sendo possível desenvolver um processo de análise e extração de informações sobre o mesmo, que foram utilizadas como nossa métrica de cobertura (este processo será melhor explicado na Seção 3.2.1). Por razões de utilizarmos o código fonte para extrair tais informações, mas enfatizando que o mesmo não foi utilizado para criação de casos de teste, consideramos a abordagem utilizada neste trabalho como uma abordagem híbrida de caixa preta e branca, onde o monitoramento do dispositivo se encaixa no primeiro e a análise da métrica, no segundo.

2.3 *Profiling* de Sistemas

O *profiling* (ou a Criação de um Perfil), trata-se da habilidade de monitorar e capturar eventos diversos em tempo de execução de um sistema ou aplicação, utilizando um agente chamado de *profiler*. Através do *profiler* é possível obter informações sobre os recursos necessários para cada evento e performance do sistema. Esta técnica, segundo LIANG; VISWANATHAN (1999), é amplamente utilizada no desenvolvimento de software, permitindo a análise de regiões mais custosas do código ou comportamentos diversos do mesmo em relação ao hardware, possibilitando todo um trabalho em cima de performance de um produto.

Existem diferentes tipos de *Profilers*[1], sendo esses responsáveis por monitorar diferentes recursos do sistema, como por exemplo o *CPU Profiler*, responsável por medir quanto do

[1] Profiler - Agente praticante do *profiling*, sendo esse o monitoramento de um sistema ou aplicação.

processador é necessário para a execução de diferentes eventos, ou também o *Memory Profiler*, responsável por medir o consumo de memoria por tais eventos, sendo ambos exemplos atuantes sobre um sistema ou aplicação específica.

Neste trabalho foi utilizado um *profiler* chamado Sample Based CPU Profiler, o qual se baseia na coleta de amostras da pilha de rastreamento[2] de uma aplicação. Isto é feito em tempo de execução Android Runtime (ART)[3], como podemos ver em sua rotina de execução descrita na Figura 2.1, e tem como objetivo medir a quantidade de processamento e o tempo gasto por uma aplicação durante a chamada de seus métodos, distribuído pelas diversas *Threads* presentes na Unidade Central de Processamento (CPU).

As amostras coletadas por este *profiler* são populadas com diversas linhas de depuração contendo informações a respeito dos métodos que foram exercitados durante a sua execução. Diferente do *Logcat* presente no *Android Debugger*, as informações geradas por este *profiler* não dependem de instrumentação prévia do código, nos permitindo obter informações relevantes para este trabalho, tais como, nome do método e seus parâmetros, sendo esses essenciais para sabermos quais métodos de uma aplicação foram cobertos durante a execução de um teste, sem realizar a instrumentação de código. Importante ressaltar que este *profiler* não necessita do código fonte da aplicação na máquina do usuário, apenas da aplicação instalada no dispositivo.

```
While(true):
  - Dorme por um curto intervalo de tempo;
  - Suspende todas as Threads;
  - Grava a pilha de rastreamento de todas as Threads
    que rodaram no último intervalo;
  - Atribui um custo unitário a essa pilha de rastreamento;
  - Continua a execução de todas as Threads;
```

Figura 2.1: Rotina de monitoramento do *Sample Based Profile*. Imagem original em LIANG; VISWANATHAN (1999)

2.4 Cobertura de Código

A cobertura de código é uma das métricas de avaliação da eficácia de testes em um ambiente de desenvolvimento de softwares, podendo essa ser explorada através da cobertura de requisitos ou de código quando se deseja ter conhecimento da abrangência de um teste. Iremos focar neste trabalho na cobertura de código, objetivando sanar uma necessidade da parceira de se obter esta cobertura sem o uso da instrumentação de código fonte.

[2] Também conhecida como Stack Trace.

[3] Android Runtime - Sistema de tempo de execução sobre o qual a plataforma Android é executada em versões acima da 5.0. - https://source.android.com/devices/tech/dalvik/

Segundo ZHENG et al. (2016), a cobertura de código trata-se de uma análise dinâmica de um programa, que tem como objetivo medir quanto do seu respectivo código foi varrido, baseado nas características de entrada desse programa para uma determinada execução. Para se obter uma cobertura de código, é preciso definir uma métrica a ser seguida (Arquivos, Classes, Métodos, Linhas), bem como uma abordagem de medição (por exemplo, Instrumentação do Código Fonte, Instrumentação do Código Compilado e Instrumentação Dinâmica).

É comum encontrar trabalhos na literatura utilizando cobertura de código para avaliar algum esforço realizado em um dado experimento HUANG et al. (2015) ou comparar diferentes abordagens no ambiente de testes DO et al. (2016). Neste trabalho foi proposto uma técnica de cobertura de código que se adequasse às necessidades da parceira, como será apresentado nos próximos capítulos, sendo definido a cobertura de métodos como nossa métrica e também uma abordagem de monitoramento não convencional para medir cobertura, através de um *profiler*, sendo esta, uma abordagem sem o uso de instrumentação do código, e nos provendo os métodos exercitados durante a execução de um teste.

Possuindo a métrica e a abordagem de captura dos dados, iremos apresentar também ao longo deste trabalho a solução desenvolvida que foi dividida em dois módulos. O primeiro módulo chamado *ToBeCovered* apresentará uma forma de se obter quais métodos foram modificados entre duas versões diferentes de uma mesma aplicação. Chamaremos essa informação de *ChangedMethods*. O segundo módulo chamado de *ManualCoverage* apresentará uma adaptação do *profiler* às necessidades da parceira para se obter os métodos executados durante os testes, chamaremos esses métodos de *CoveredMethods*. Assim como é possível observar na seguinte fórmula, com essas informações, podemos calcular a cobertura de métodos, chamada de *MethodCoverage*, de uma conjunto de testes ou de um único teste. Mais detalhes sobre os módulos desenvolvidos são discutidos no Capítulo 3.

$$MethodCoverage = \frac{ChangedMethods}{CoveredMethods}$$

2.5 Controle de Versão

O Controle de Versão se trata de uma maneira de gerenciar as diferentes versões de um produto em desenvolvimento, que registra as modificações realizadas nos diferentes arquivos de um projeto ao longo do tempo, sendo possível navegar em versões específicas quando necessário e manter um histórico de tudo que é modificado durante seu desenvolvimento[4].

Quando se fala em Controle de Versão, no ambiente industrial, é comum que toda empresa que desenvolve software utilize um sistema específico para realizar o controle de versão de seu software, garantindo uma série de pontos positivos quando se trata de qualidade do processo de produção, que vão desde segurança ao manter um histórico das modificações de seu projeto, agilidade ao permitir que várias pessoas desenvolvam diferentes partes de um programa

[4]Git - Sobre o Controle de Versão - https://goo.gl/7nFidF

de forma independente e também adaptabilidade ao ser possível gerar versões com diferentes comportamentos com bastante versatilidade SPINELLIS (2005).

No lado da literatura é muito comum encontrar projetos que fazem comparativos entre diferentes versões de um determinado programa, como em REIS; MOTA (2018), sendo necessário o uso de um sistema de controle de versão que permita realizar as devidas operações já implementadas nesse tipo de sistemas, com o objetivo de facilitar a manipulação das diferentes versões.

Neste trabalho foi usado o sistema de Controle de Versão das aplicações providas pela parceira industrial para que pudéssemos fazer comparativos entre diferentes versões dessas aplicações e extrair as informações necessárias para o cálculo da métrica de cobertura proposta.

2.6 Árvore Sintática Abstrata

Árvore Sintática Abstrata (AST) se trata de uma representação em Árvore da estrutura sintática de um código fonte implementado em uma linguagem específica.[5] O uso de AST é muito comum na literatura quando se fala em comparação entre versões diferentes de um mesmo código, como é possível ver em NEAMTIU; FOSTER; HICKS (2005), ou sobre comparação entre códigos diferentes para detecção de cópia como em CUI et al. (2010). O uso de uma análise estruturada do código fonte facilita quando é preciso identificar padrões no código de maneira mais ágil.

Neste trabalho foi utilizado esta interpretação de código para facilmente identificar declarações de métodos em código fonte Java, como podemos observar dois exemplos AST Java nas figuras 2.2 e 2.3. A primeira representa o código em uma versão *alfa* e a segunda apresenta uma modificação sobre os parâmetros da função em questão, mostrando a remoção de um parâmetro do código na versão *Beta*. Para realizar esse processo de análise, utilizamos uma biblioteca[6] na linguagem Python, que pudesse abstrair algumas operações e facilitar o processo de construção da AST e navegação pela mesma para obter os dados necessários.

Durante este trabalho não iremos nos aprofundar em AST, porem foi necessário ter o conhecimento mínimo necessário para facilitar a análise de um código Java, sendo importante sua a menção nesta seção com o objetivo de detalhar os conhecimentos envolvidos no trabalho.

[5] Árvore Sintática Abstrata - https://goo.gl/WTbnpN
[6] Javalang - https://github.com/c2nes/javalang

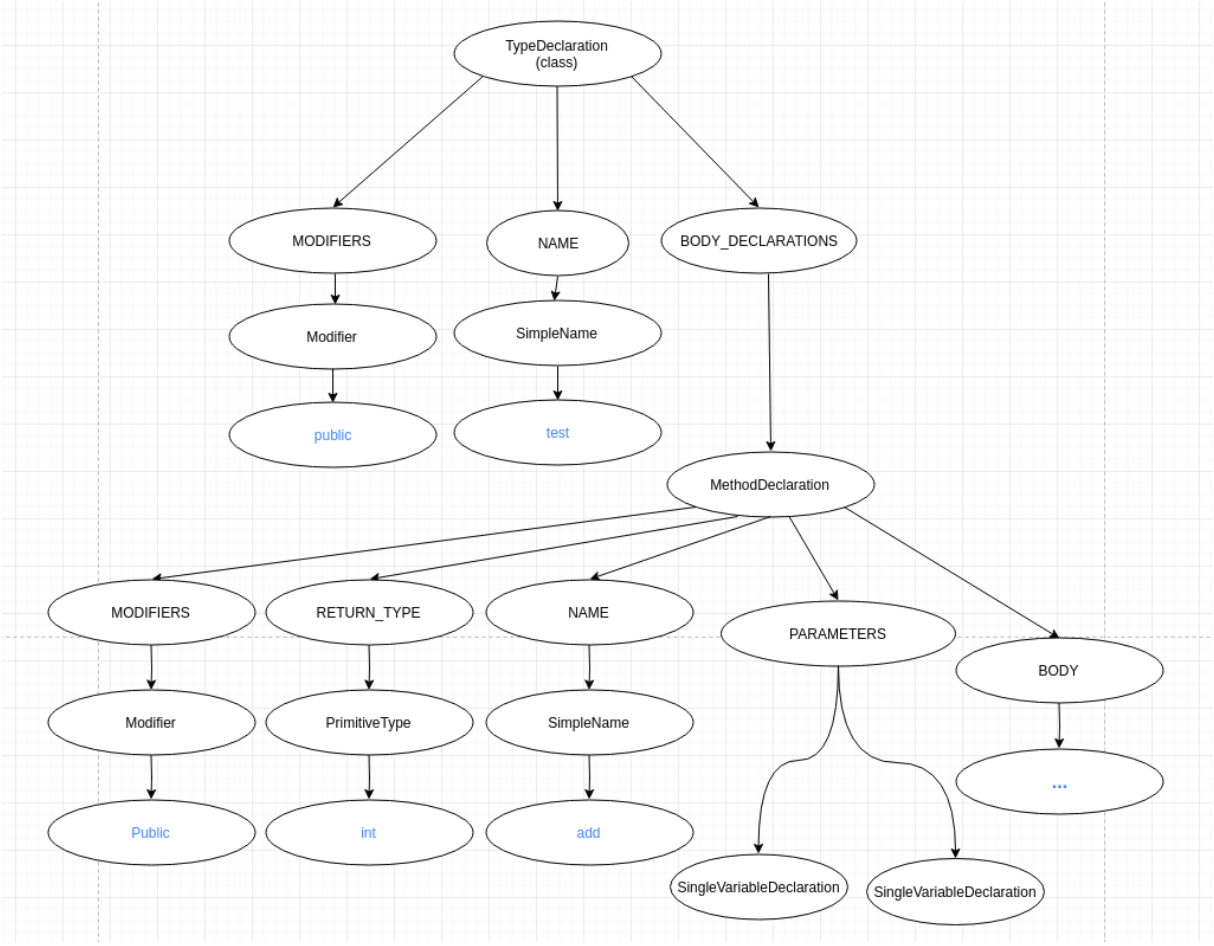

Figura 2.2: Ilustração de AST Java - Versão *Alfa*

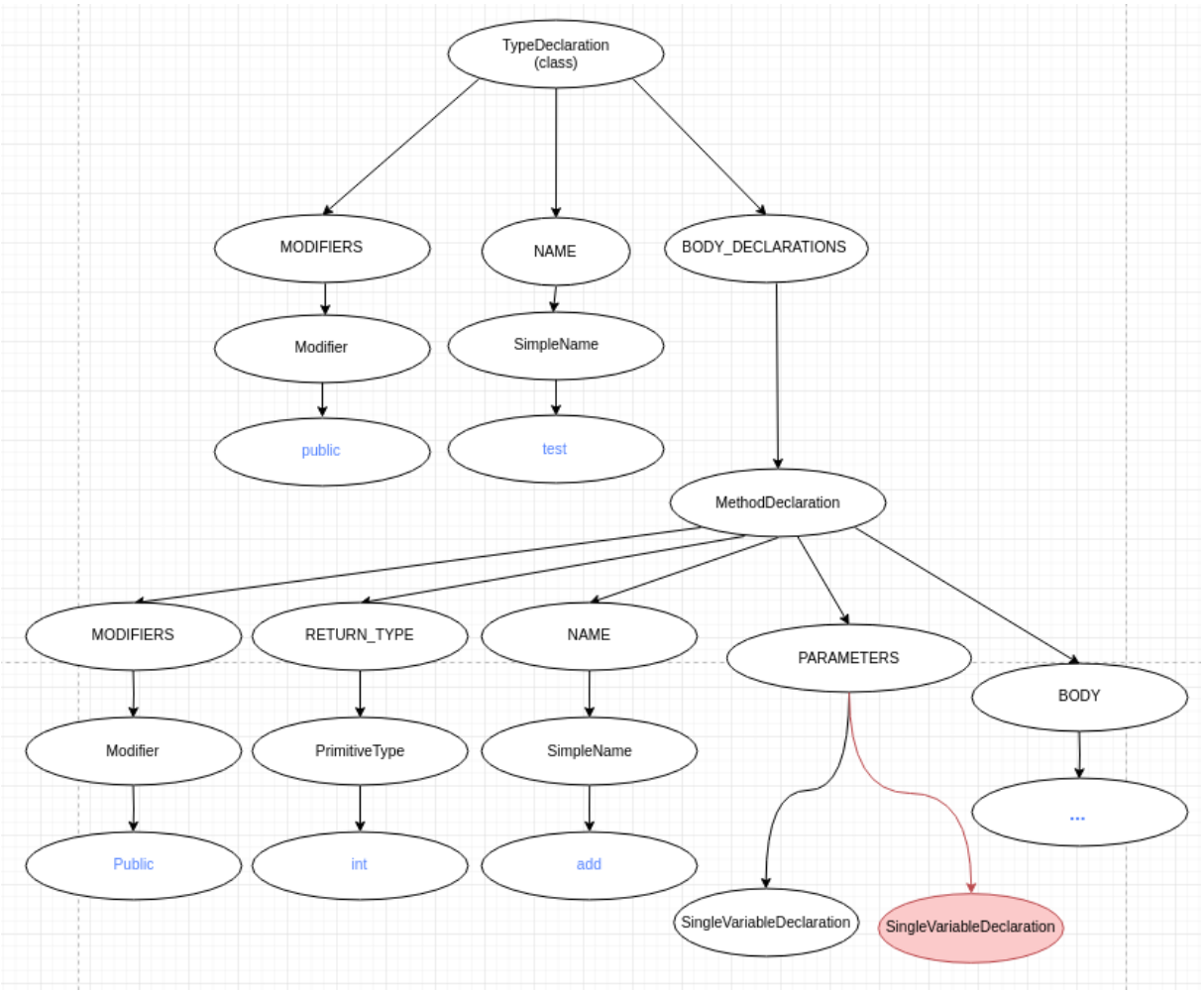

Figura 2.3: Ilustração de AST Java - Versão *Beta*

3

Desenvolvimento

Este capítulo apresenta todo o processo pelo qual a ferramenta de cobertura de código proposta como solução foi desenvolvida, sendo dividido em dois módulos, seguindo os padrões de modularidade de engenharia de software para uma melhor manutenção e adaptabilidade.

3.1 Requisitos do Sistema

O cenário em que realizamos este trabalho dispunha da execução de testes manuais e automatizados em dispositivos móveis. Dentro do ecossistema em que estávamos inseridos, onde a maior demanda presente era a de execuções manuais de teste, podíamos escolher nosso foco entre testes de regressão, exploratório ou testes de sanidade. Todos estes testes são executados diariamente por nossa parceira industrial. Por questão de as execuções manuais de regressão serem as de maior demanda para nosso parceiro, bem como serem as maiores e mais ricas em detalhes, decidimos utilizá-las como base para este trabalho.

Por uma limitação de não podermos instrumentar o código fonte, devido a possíveis alterações nos resultados dos testes, segundo a literatura KELL et al. (2012), buscamos alternativas que pudessem satisfazer a necessidade básica de um processo de Cobertura de Código. Então, definimos uma métrica de cobertura e uma forma de capturar esses dados, para então podermos inferir resultados percentuais de cobertura sobre o código testado, e também poder utilizar os dados obtidos em outras perspectivas e trabalhos realizados dentro da empresa.

Sobre o cenário e a limitação imposta, foi-nos liberado acesso de leitura ao código fonte das aplicações testadas e acesso a dispositivos móveis em fase de testes, para que pudéssemos criar uma solução adequada com o objetivo de obter os dados de cobertura e utilizá-los como forma de *feedback* para as atividades exercitadas.

Por fim, como nenhum dos processos de teste dentro da empresa envolvia obter dados de cobertura, tivemos uma maior autonomia para criar alternativas e realizar experimentos para validação de ideias. Realizamos também buscas na literatura, de trabalhos que se assemelhavam com nosso cenário e limitação. E dos vários trabalhos envolvendo testes em dispositivos Android, encontramos apenas o trabalho relatado em AZIM; NEAMTIU (2013), o qual utilizava uma

abordagem para obtenção dos métodos cobertos proveniente do *Android Debug Bridge (ADB)*[1], chamada Activity Manager (**am**) sendo categorizado como um *CPU Profiler* sem o uso de instrumentação, sendo viável diante de nossas limitações. Levamos em consideração esta abordagem ao executar experimentos e também ao comentar sobre os trabalhos relacionados.

3.2 O Protótipo da Ferramenta

O processo utilizado para solucionar o problema descrito anteriormente, levando em consideração o cenário e a limitação, foi implementado em uma ferramenta protótipo chamada AutoTestCoverage^{C} (^{C} significa *Componente*, ou seja, nosso foco é em aplicações Android específicas). Em outro trabalho de nosso grupo de pesquisa, há uma outra ferramenta cujo foco é a interação entre várias aplicações Android, sendo chamada AutoTestCoverage^{P}, onde o sobrescrito significa *Plataforma*).

Objetivando os princípios de modularização da engenharia de software, este protótipo possui dois módulos: o primeiro, chamado de *To Be Covered (TBC)*[2], é responsável pela definição do que deve ser coberto em uma execução, baseando-se na diferença de versões de uma aplicação; e o módulo dois, também conhecido como *Manual Coverage (MC)*[3], é responsável pela adaptação dos diferentes tipos de Profilers comparados neste trabalho, para obtenção dos métodos executados durante uma execução, sem o uso de instrumentação de código fonte (ver Figura 3.1). Esses módulos serão detalhados a seguir para um melhor entendimento de como chegamos aos resultado deste trabalho.

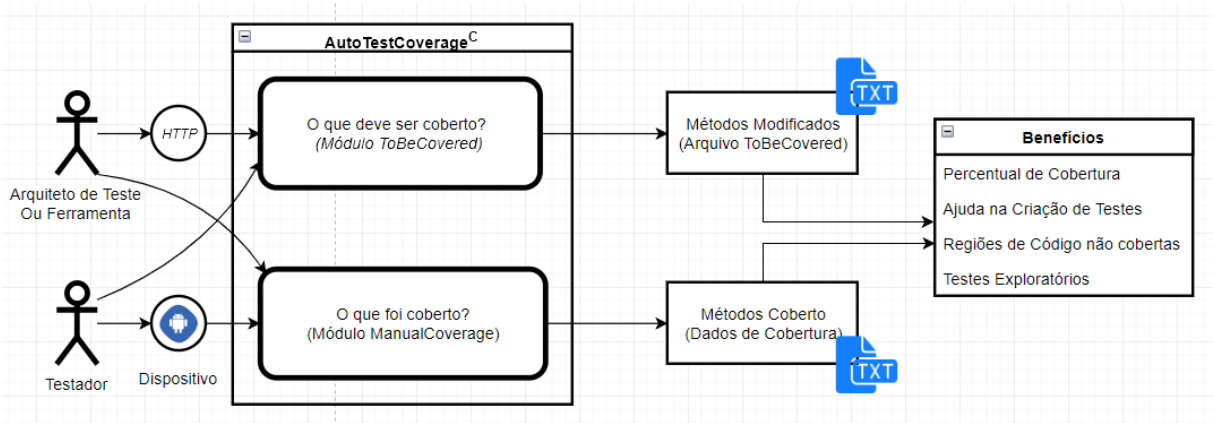

Figura 3.1: Visão geral da ferramenta AutoTestCoverage^{C}.

[1]O Android Debug Bridge (adb) é uma ferramenta de linha de comando versátil que permite a comunicação com uma instância de emulador ou com um dispositivo Android conectado.

[2]To Be Covered - Módulo responsável por calcular o que deve ser coberto.

[3]Manual Coverage - Módulo responsável por calcular a cobertura manual.

3.2.1 Módulo 1 - *To Be Covered* (TBC)

Este módulo, disponibilizado através de uma Interface de Programa de Aplicação (API), é ainda dividido em submódulos, com o objetivo de definir o que deve ser coberto em uma campanha de teste de regressão, baseado em diferentes versões do código fonte. Para concebê-lo, utilizamos o framework Flask[4], que utiliza a linguagem Python[5] para facilitar o desenvolvimento no ambiente web. Também utilizamos a tecnologia Git[6], responsável pelo controle de versões em repositórios de desenvolvimento. Estas tecnologias foram utilizadas nos submódulos que serão explicados com mais detalhes durante esta seção.

Foi necessário inicialmente definir uma métrica de cobertura que melhor se adequasse à necessidade. Por isso, foi feito uma análise sob o contexto em que a parceira se encontrava, sendo observado que a mesma possuía interesse sobre o quão eficiente os testes executados diariamente estavam de fato exercitando as regiões de código modificada nas aplicações. Com esse contexto, pensamos em diferentes métricas, dentre elas, arquivos, classes e métodos. Nossa decisão teria que lidar com o nosso cenário e limitação. Como nosso foco se tratava de obter uma informação que pudesse guiar o processo de teste, diante da execução de testes de regressão manual sobre aplicações, vimos que a métrica por arquivo ou classe seria muito superficial[7]. Decidimos que uma boa métrica seria dada pela cobertura de métodos, por ser possível extrair tal informação das execuções dos *profilers* usados, sendo assim uma métrica mais detalhada que arquivos ou classes.

A princípio este módulo foi pensado para que os arquitetos de teste pudessem utilizá-lo para melhor entender quais métodos foram modificados na hora de criar novos testes para a base. À medida que avançamos com o trabalho vimos que ela também poderia ser utilizada por testadores, como uma forma de feedback, podendo assim saber se a cobertura está ou não sendo efetiva enquanto o mesmo é executado.

Também é importante mencionar uma dependência presente neste módulo: a necessidade de possuir o código fonte da aplicação alvo disponível em uma máquina onde será executado AutoTestCoverage^{C}. Pois assim é possível comparar os arquivos modificados entre as diferentes versões da aplicação, permitindo a extração dos dados de interesse durante as etapas deste módulo.

As fases descritas a seguir irão melhor explicar como este módulo funciona para prover os métodos que devem ser cobertos em uma campanha de regressão baseado em duas diferentes versões, lembrando que para as seguintes etapas, tínhamos apenas acesso de leitura ao código fonte das aplicações testadas.

[4]Flask - http://flask.pocoo.org/
[5]Python - https://www.python.org/
[6]Git - https://git-scm.com/
[7]Esta métrica, no entanto, se aplica bem à ferramenta AutoTestCoverage^{P}

3.2.1.1 Fase 1 - Recebendo as versões desejadas e preparando os dados

Como este módulo se trata de uma API focada em uma aplicação Android específica, nesta fase nós recebemos uma entrada através de uma requisição Protocolo de Transferência de Hipertexto (HTTP), que possui em seu cabeçalho a aplicação alvo a ser testada. Assim como DO et al. (2016), precisamos também de duas versões desta aplicação para melhor se adaptar ao cenário de testes de regressão, permitindo o cálculo da diferença entre elas, sendo uma versão ***alfa***, mais antiga e ***beta*** uma mais atual a ser testada. Na Figura 3.2, podemos ver um exemplo de cabeçalho utilizado na requisição do presente módulo, contendo a aplicação a ser testada no campo *package*, a versão *alfa* e a versão *beta* desejada.

```
{
    "package": "com.package.name",
    "alfa_version": "01.20",
    "beta_version": "01.25"
}
```

Figura 3.2: Exemplo de Cabeçalho de Requisição do Modulo TBC

Após receber os dados de entrada, verificamos se o repositório da aplicação desejada se encontra presente e atualizado. Caso contrário, fazemos as devidas operações utilizando o Git para manter a consistência dos dados e garantir o bom funcionamento do processo. Após atualizar o repositório, é necessário manter o mesmo sempre na versão ***beta*** recebida como entrada, pois precisaremos ter acesso aos arquivos mais recentes para que nas seguintes fases possamos acessá-los.

3.2.1.2 Fase 2 - Calculando a diferença entre versões e filtrando modificações Java

Em seguida precisamos calcular a diferença entre essas duas versões passadas como entrada, para então saber quais arquivos foram modificados. Como temos interesse exclusivo nos arquivos Java[8], pois é neles que estão implementadas as funcionalidades da aplicação Android, precisamos filtrar apenas as modificações ocorridas nesses ficheiros.

Para calcular essa diferença, utilizamos o Git por questões de praticidade já que o mesmo consegue exercer esta função de maneira fácil. Podemos também nos aproveitar de uma propriedade presente no arquivo da diferença gerando pelo Git. Notamos que existe um padrão sobre cada arquivo modificado contendo o diretório para tais arquivos, facilitando então

[8] Java - https://goo.gl/BnXdEu

o processo de filtragem nesta fase, focando apenas em modificações que ocorreram em arquivos de extensão Java, reduzindo assim nosso escopo de modificações e dados a serem analisados.

3.2.1.3 Fase 3 - Obtendo os métodos e suas informações

Nesta fase, nosso principal objetivo é obter informações sobre os métodos presentes nos arquivos Java que foram modificados, para facilitar a próxima etapa deste módulo. Com este objetivo iremos utilizar os dados gerados na Fase 2, que nos provê exatamente o diretório para esses arquivos, facilitando o acesso para verificação do código fonte.

Para reunir as informações necessárias, tais como todos as declarações de método de um arquivo, seus respectivos nomes e caminhos de acordo com o padrão canônico[9] (um exemplo do padrão, *com.example.app.classname.methodName()*), primeira e a última linha do escopo deste método, parâmetros e modificadores, utilizamos uma biblioteca do Python, já mencionada no Capítulo 2 na seção 2.6, que traduz um parser da linguagem Java para implementação Python, possibilitando assim navegar sobre a AST de cada arquivo modificado e extraindo tais informações.

3.2.1.4 Fase 4 - Correlação entre as regiões modificadas e o código fonte

Possuindo agora as regiões modificadas dos arquivos Java e as informações sobre cada método contido nesses arquivos, podemos agora relacionar essas informações e inferir quais métodos de quais arquivos foram modificados e quais são os métodos que devem ser exercitados em uma campanha de regressão.

Para fazer isso, utilizamos outro padrão encontrado em arquivos de diferença gerados pelo Git, que nos informa a linha do arquivo em que a modificação começa, e quantas linhas foram modificadas, permitindo assim que cruzemos estas informações com os dados que possuímos anteriormente sobre a primeira e última linha de um método, o que nos provê o seu intervalo de escopo. Assim, podemos de fato dizer se a modificação está ou não dentro de um método e qual é este método, agrupando todos os métodos que foram modificados entre a as versões ***alfa*** e ***beta***.

3.2.1.5 Fase 5 - Preparando o arquivo final

Por fim, possuindo agora os métodos que foram modificados entre as duas versões, podemos formatá-los em um arquivo que chamamos de arquivo TBC, se tratando de uma lista contendo tais métodos escritos seguindo o padrão canônico de Java, anteriormente mencionado na Fase 3.

Este mesmo padrão será utilizado no formato dos dados obtidos durante a execução do módulo de cobertura, facilitando assim a comparação entre o que deve ser coberto e o que de fato foi coberto, possibilitando uma forma mais fácil de analisar a campanha de testes executado.

[9] Java Docs - https://goo.gl/wHsbws

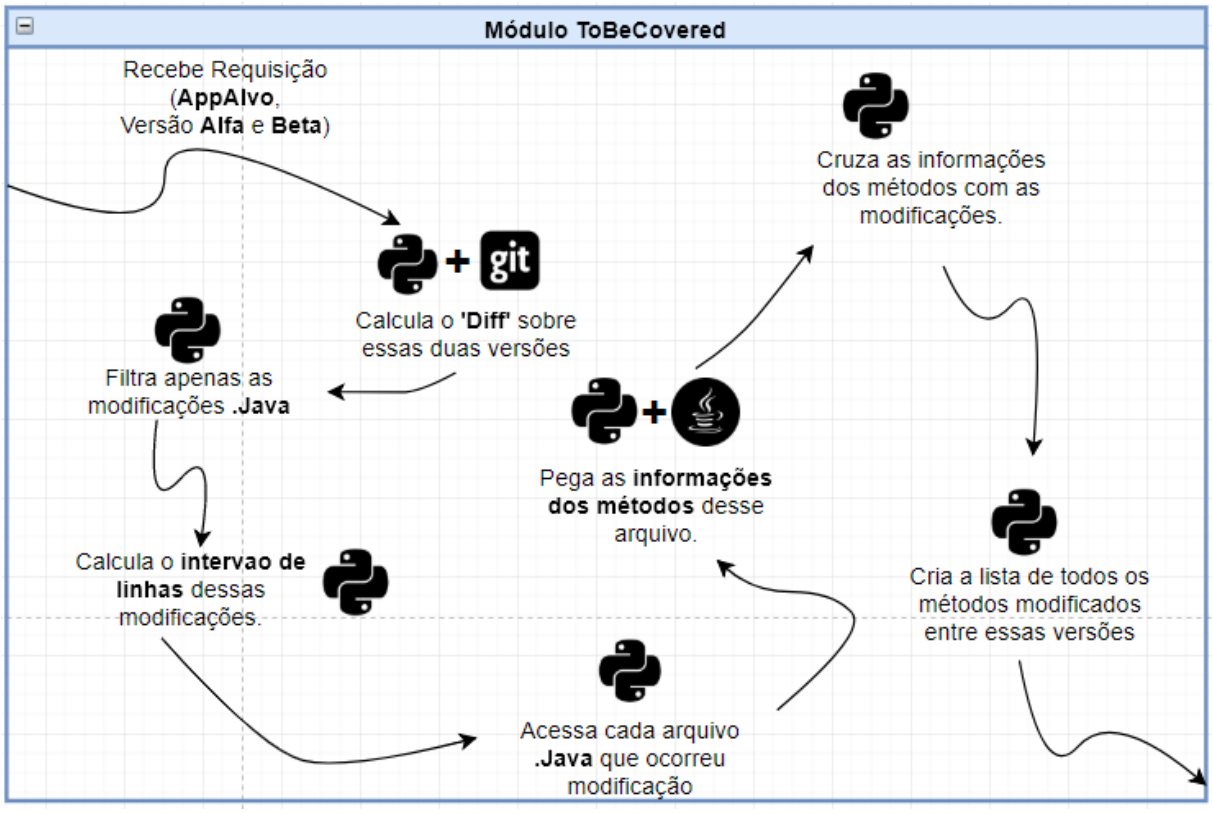

Figura 3.3: Visão geral do Módulo *ToBeCovered*

3.2.1.6 O Algoritmo geral *ToBeCovered*

Para uma melhor compreensão das fases listadas anteriormente, apresentamos o **Algoritmo 1** que junta todas as etapas de forma simples para realizar o processo proposto anteriormente, bem como a Figura 3.5, que apresenta de forma visual o passo a passo explicado nesta seção.

3.2.2 Módulo 2 - *Manual Coverage* (MC)

O segundo módulo deste trabalho consiste de uma aplicação local de monitoramento de dispositivos Android, e tem o objetivo capturar os métodos (levando em consideração a métrica definida no módulo um) que são executados em uma aplicação do dispositivo, durante a prática de um teste, podendo ser utilizado sobre cada teste de uma campanha, provendo assim um relatório de cobertura baseado nos métodos cobertos versus os métodos que deveriam ser cobertos, sendo esses últimos calculados pelo módulo um.

Para conceber este módulo foi utilizado o *framework* web, Django[10] que utiliza a linguagem Python para facilitar a criação de ferramentas no ambiente web, também foi utilizado a linguagem Java com o objetivo de criar uma interface entre o Python e o ambiente Android. Essas tecnologias foram em conjunto utilizadas nos sub-módulos explicados a seguir.

[10]Django Framework - https://www.djangoproject.com/

Algorithm 1 ToBeCovered - Collecting Methods Modifications

Input: *TargetApp* and *alfa* Version and *beta* Version of App
Output: List of Changed Methods *TBC*

```
1:  function GETTOBECOVERED
2:                                                          ▷ Phase 1
3:      TBC ← ∅
4:      Repo ← UPDATEREPOTOVERSION(TargetApp, beta)
5:                                                          ▷ Phase 2
6:      AllModifs ← GETREPODIFF(Repo, alfa, beta)
7:      JavaModif ← FILTERJAVADIFFS(AllModifs)
8:                                                          ▷ Phase 3
9:      JavaFileList ← GETCHANGEDJAVAFILESDIRECTORIES(JavaModif)
10:     MethodsInfo ← GETMETHODSINFO(JavaFileList)
11:
12:                                                         ▷ Phase 4
13:     for each Java Modification JM : JavaModif do
14:         ChangedMethod ← GETMODIFMETHODS(MethodsInfo[JM.Dir], JM)
15:         if ChangedMethod then
16:             TBC ← TBC ∪ {ChangedMethod}
17:         end if
18:     end for
19:                                                         ▷ Phase 5
20:     return TBC
21: end function
```

A princípio este módulo foi pensado visando ser utilizado pelos testadores para melhor entender quais métodos eram cobertos durante a execução dos testes de uma campanha. À medida que avançamos com o trabalho, vimos que ela também poderia ser utilizada por arquitetos de teste, como uma forma de *feedback* aos testes criados por eles, podendo assim ter uma resposta se os testes estão ou não exercitando as regiões de código necessárias.

É importante mencionar que este módulo possui como dependência um dispositivo de teste conectado à máquina em que a ferramenta se encontra e também com o modo de depuração ativo, bem como a aplicação alvo a ser testada na versão *beta* instalada. Só assim será possível ter acesso aos métodos que são chamados por esta aplicação, possibilitando a extração dos dados cobertos para posteriores conclusões.

As seguinte sessões descritas abaixo irão melhor explicar como o Módulo 2 funciona para prover o que o foi coberto em uma campanha de testes. Novamente é importante relembrar que, para exercitar os seguintes passos, nós tínhamos apenas acesso de leitura ao código fonte das aplicações a serem testadas e dispositivos em fase de teste.

3.2.2.1 A interface da ferramenta

Como este módulo trata-se de um programa que precisa estar rodando na máquina do testador para que seja possível existir uma conexão com o dispositivo a ser testado, foi necessário criar uma interface gráfica com o objetivo a facilitar o uso por parte dos testadores.

Foi pensado em um conjunto de telas simples que o processo pudesse comportar, envolvendo as etapas de autenticação, para ter controle de acesso e uso, uma tela de pesquisa de campanha, de maneira que o testador pudesse pesquisar por um conjunto de testes e uma tela principal de cobertura (pode ser observada na Figura 3.4), onde seria possível executar as ações necessárias para obtermos e salvar os dados provenientes dos testes executados em uma campanha.

3.2.2.2 Fase 1 - Autenticação e pesquisa de uma campanha

O processo de uso do Módulo 2 se inicia com a necessidade de autenticação do usuário testador, pois só assim seria possível acessar a base de informações de teste utilizados pela parceira e disponibilizar esses testes ao usuário.

Após autenticado, o usuário será direcionado para a página de pesquisa de uma campanha de testes, onde deve inserir um código referente a uma campanha desejada. Caso exista a campanha buscada, ele será redirecionado para a página principal. Caso contrário, ele será informado que a campanha não existe, até que o código seja válido.

3.2.2.3 Fase 2 - Sincronizando os dados do dispositivo e Aplicação

Na página principal, o usuário deve selecionar o dispositivo a ser testado, que deve estar conectado ao computador, bem como a aplicação alvo a ser testada, instalada no dispositivo em

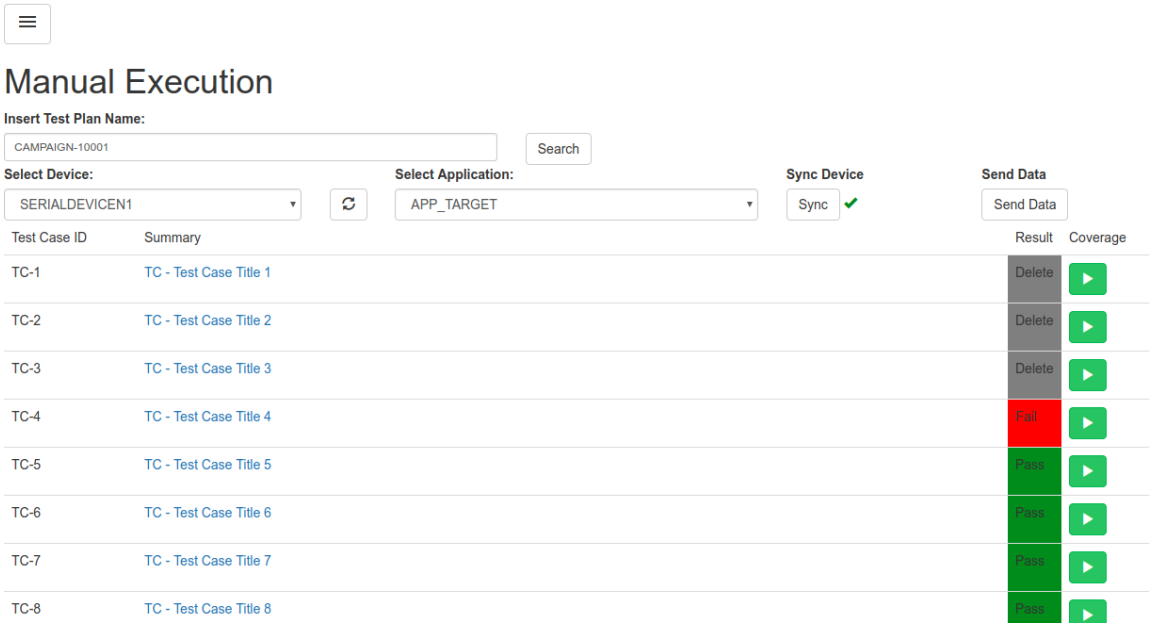

Figura 3.4: Tela principal da AutoTestCoverage[C]

uma versão ***beta***. ao selecionar corretamente tais informações, é necessário sincronizar os dados do dispositivo. Esse procedimento se trata de obter informações do dispositivo a ser testado, como por exemplo, seu código serial, versão do sistema instalada, modelo, entre outros dados e também o pacote da aplicação selecionada, (ex: ***'com.app.teste'***). Tais informações serão armazenadas junto com os dados capturados nas próximas fases de maneira mais consistente e enriquecendo o uso posterior. Importante explicar que, este Módulo não possui a necessidade do código fonte da aplicação na máquina do testador.

3.2.2.4 Fase 3 - O executável *Cobertura.jar*

Antes de entrarmos na fase de captura dos métodos em si, é importante explicarmos como de fato conseguimos monitorar uma aplicação Android e obter os métodos executados sem o uso de instrumentação de código. Até chegarmos na solução atual, passamos por algumas tentativas que iremos descrever nesta sessão.

Como nos tínhamos apenas acesso de leitura ao código fonte da aplicação foi possível mapear cada método existente dentro da aplicação, usando um parser como explicado no Módulo 1. Com isso, nossa primeira tentativa de saber quais métodos eram executados, se tratava de definir *breakpoints*[11] em cada um dos métodos de uma aplicação e, à medida que os métodos fossem chamados, os *breakpoints* eram ativados e liberados, marcando assim cada um dos métodos exercitados. Essa primeira tentativa se mostrou muito custosa e deixava o dispositivo muito lento, impactando diretamente no resultado do teste.

Uma segunda tentativa veio quando descobrimos uma ferramenta da Google chamada

[11] Breakpoint - Pausas escolhidas no código. Por questões de compreensão escolhemos manter o termo em inglês.

Android Device Monitor[12] que conta com uma série de funcionalidades para analisar o contexto de uma aplicação e do dispositivo. Dentre as diversas funções dessa plataforma, existia uma com um potencial muito grande diante do nosso contexto, chamada ***Sample Based CPU Profiling ou Sample Based Profile***[13], que monitora amostras sobre as múltiplas *threads* que são executadas sobre o sistema em tempo de execução ART, também exibindo a *thread* a qual cada método foi alocado, bem como o quantidade utilizada do processador para tal execução. Esta ferramenta gera um arquivo temporário quando executado. Este arquivo possui todos os metadados necessários para exibir as informações descritas em uma interface própria. Explorando-o foi possível obter a informação dos métodos executados sem realizar a instrumentação de código, o que nos permitiu seguir com o trabalho tendo uma forma de captura para medir a cobertura.

Após adquirir essa informação e tentar implantar o uso dessa ferramenta, vimos que seria muito incômodo para qualquer cliente presente no mesmo contexto, inserir uma ferramenta inteira para utilizar apenas uma funcionalidade. Começamos a investigar como era feito a captura dos dados ao utilizar a funcionalidade denominada ***Sample Based Profile***, e com isso foi possível identificar o uso de uma biblioteca Java, criada pela Google, chamada *ddmlib* que utiliza diversas funções para manipular informações de baixo nível sobre o sistema em tempo de execução ART, do Android. Em conhecimento desta biblioteca foi possível isolar a função de *profiling* mencionada, adaptando-a ao nosso contexto, criando assim um executável Java chamado *Cobertura.jar*, que pudesse ser portado para outra interface gráfica mais adaptada à prática, sendo assim agora possível obter os métodos exercitados por um teste sem o uso de instrumentação. Notamos também outra função de *profiling* presente na biblioteca, chamada *Method Tracer* que, por sua vez, realiza instrumentação em tempo de execução na aplicação alvo, também gerando um arquivo contendo métodos executados. Porém, o Method Tracer pode gerar uma sobrecarga maior no dispositivo. Levamos em conta esta abordagem nos experimentos executados no Capítulo 4.

Com a modularização da função de *profiling* através do executável Java, temos um arquivo de saída sempre que o mesmo é executado, contendo os métodos que foram exercitados durante o monitoramento. Para chegar neste arquivo, é necessário extrair apenas as informações relevantes ao nosso contexto do arquivo bruto que pela função utilizada do ddmlib. O processo de filtragem de métodos para geração do arquivo de saída se tratava de identificar as linhas contendo o pacote da aplicação alvo, sendo possível obter informações como o caminho até o método, seus parâmetros e tipo de retorno. Todo o processo de execução do *profiling* e filtragem da saída gerada será explicado de forma simplificada no Algoritmo 2 descrevendo o *Cobertura.jar*.

[12] Android Device Monitor - https://developer.android.com/studio/profile/monitor

[13] Por razões de facilidade para o leitor, foi mantido o nome original da técnica descrita na documentação oficial. A explicação do termo está presente na seção 2.3.

Algorithm 2 CoverageMonitor - Coverage Process and Filter

Input: *appPackage*, *deviceSerial*, *outputFile* and *profilingFrequency*
Output: File with convered methods in a monitored execution

```
1:  function COVERAGEMONITOR
2:                                                              ▷ Phase 1
3:      DebugBridge ← CREATEBRIDGECONNECTION(deviceSerial, appPackage)
4:      DebugBridge.STARTMETHODPROFILING(profilingFrequency)
5:
6:      while NoStopEventReceived do KeepMonitoring
7:                                                  ▷ Waiting for Stop Event
8:      end while
9:                                                  ▷ On Stop Event Received
10:     RawTraceFile ← DebugBridge.STOPMETHODPROFILING(deviceSerial)
11:
12:                                                 ▷ Filter data and create output file
13:     CoveredMethods ← FILTERCOVEREDMETHODS(RawTraceFiles)
14:     CREATEOUTPUTFILE(outputFile, CoveredMethods)
15: end function
```

3.2.2.5 Fase 4 - Capturando os Métodos Executados

Após o usuário realizar a sincronização do dispositivo, ainda na página principal, é possível começar a captura dos métodos em cada teste de uma campanha previamente listada, utilizando o executável Java de forma transparente. O usuário agora tem acesso a uma opção que o permite começar a cobrir um teste específico por vez. Feito isso, a aplicação alvo começará a ser monitorada e o testador seguirá com a execução do teste manual de maneira que, enquanto executa uma série de ações na aplicação, os diversos métodos que são exercitados são capturados e adicionados à lista de métodos que foram ativados para aquele teste.

Quando o teste chega ao fim, o testador deve então parar o monitoramento, e assim finalizar a construção da lista de métodos. Após isto, os dados gerados serão indexados em uma base textual (usamos o *Solr*[14]) com o objetivo de uso posterior para calcular a cobertura desejada no trabalho e também servir de insumo para outras ferramentas que desejam consumir tais informações. Este processo pode ser repetido para cada um dos testes de uma campanha, resultando na cobertura total de uma campanha como veremos na próxima seção. A Figura 3.5 contem uma forma mais visual de entender o fluxo de etapas do presente Módulo.

3.2.3 O Algoritmo de Cálculo da Cobertura

Após obtermos os dados referentes aos métodos que devem ser cobertos, visto na Seção 3.2.1, e adquirir os métodos que foram cobertos na Seção 3.2.2, facilmente podemos inferir o percentual de cobertura de cada teste executado. O Algoritmo 3 a seguir mostra como é feito o

[14] Solr - http://lucene.apache.org/solr/

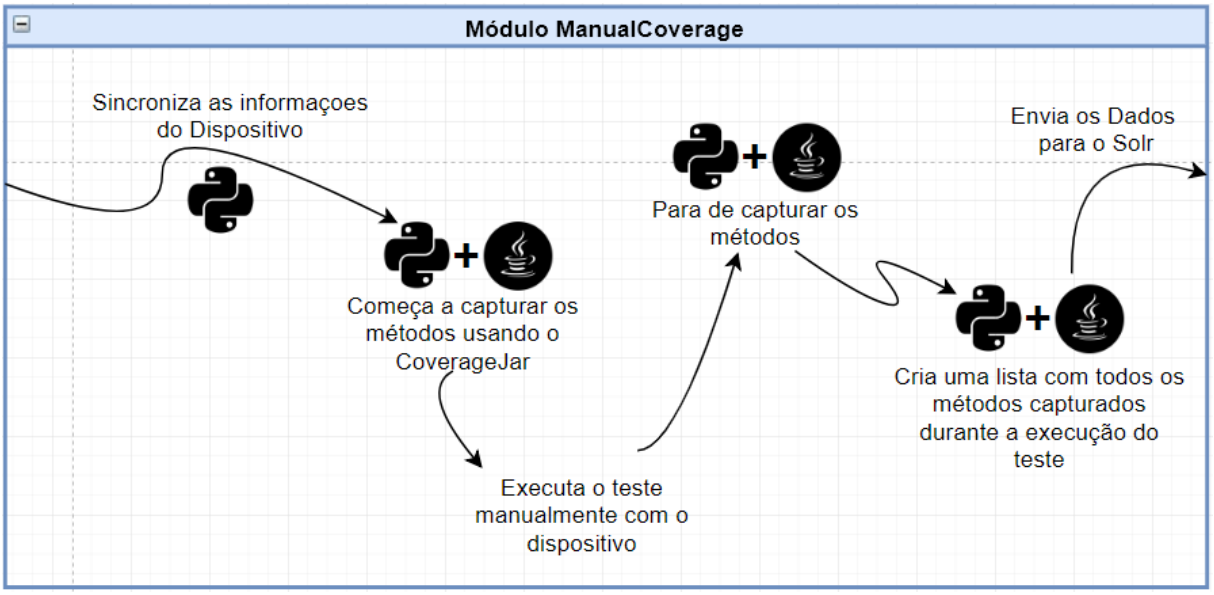

Figura 3.5: Visão geral do Módulo *ManualCoverage*

cálculo desse percentual para fins informativos sobre uma campanha.

A abordagem apresentada neste capítulo foi validada através de experimentos que serão apresentados no próximo capítulo e também com a nossa parceira, principal interessada na validação deste trabalho.

Algorithm 3 Calculate Coverage - Calculate a Test Plan Coverage Percentage

Input: *TargetApp* and *alfa* Version and *beta* Version of App, *TestPlanId*
Output: Total Coverage Percentage from one Test Plan Execution

```
1:  function CALCULATECOVERAGE
2:                                                          ▷ Phase 1
3:      TBC ← GETTOBECOVERED(TargetApp, alfa, beta)
4:      CoverageFiles ← GETCOVERAGEDATAFILESFROMSOLR(TestPlanId)
5:                                                          ▷ Phase 2
6:      UniqueMethodList ← GETUNIQUEMETHODS(CoverageFiles)
7:      CoveredMethods ← ∅
8:
9:      for each Method M : UniqueMethodList do
10:         if M ∈ TBC then
11:             CoveredMethods ← CoveredMethods ∪ {M}
12:         end if
13:     end for
14:                                                         ▷ Phase 5
15:     CoveredMethodsLength ← LENGTH(CoveredMethods)
16:     ToBeCoveredLength ← LENGTH(TBC)
17:     TotalCoverage ← DIVIDE(CoveredMethodsLength, ToBeCoveredLength)
18:     return TotalCoverage
19: end function
```

4

Experimentos

Neste capítulo será apresentada uma análise para validação da ideia proposta. Com o objetivo de melhor entender as diferentes técnicas encontradas, e também sanar a falta de informações de cobertura em ambientes de desenvolvimento industriais, realizamos experimentos comparativos entre as diferentes técnicas para obtenção dos dados sem o uso de instrumentação.

Objetivando se aproximar da realidade, os experimentos executados foram guiados por testadores profissionais (para que não houvesse interpretações dúbias a respeito dos cenários executados), utilizando dispositivos em fase de teste, casos de testes reais e focado na validação de uma aplicação específica. Dentro do ambiente manual de testes e diante destas condições, realizamos alguns experimentos com o objetivo de coletar dados que pudessem responder a algumas hipóteses levantadas durante o trabalho. Tais hipóteses serão apresentadas e discutidas neste capitulo.

Característica do Ambiente Experimental. Importante lembrar que todos os dados foram coletados utilizando o mesmo dispositivo L, em uma única versão do sistema e da aplicação. Apesar de não podermos revelar o nome comercial do dispositivo L, ele é um celular que consta de processador *1.8 Giga Hertz* de *8 Núcleos*, *3 Gigabytes* de memória RAM e *32 Gigabytes* de armazenamento e uma placa de vídeo dedicada, rodando a versão 8.0 do sistema Android.

Durante a realização dos experimentos utilizamos quatro aplicações: **A, H e C**. A **aplicação A** trata-se de um assistente de usabilidade que utiliza sensores de proximidade, tais como acelerômetro e giroscópio. A **aplicação H** trata-se de um assistente para usuários poderem testar as diferentes funcionalidades do dispositivo passando por cada sensor e hardware que impacte na usabilidade do usuário. A **aplicação C** realiza operações com a câmera do dispositivo.

4.1 Primeira Hipótese

Hipótese: Dadas duas versões de uma aplicação em teste, conseguimos calcular o conjunto de métodos que foram modificados (armazenados no arquivo denominado pela sigla *TBC*, a qual vem de *To Be Covered*).

4.1.1 Considerações

Com acesso de leitura ao repositório de uma aplicação, é possível comparar diferentes versões do código através dos *commits*[1] realizados durante seu desenvolvimento. É através dessa informação e utilizando a ideia explicada na Seção 3.2.1 que pretendemos obter quais métodos foram modificados entre duas versões e poder montar um arquivo denominado *TBC*.

4.1.2 Experimentos

Mediante o acesso ao código fonte, foi possível executar o módulo *To Be Covered* sobre duas versões da **aplicação A**, uma versão *alfa*, mais antiga, e uma versão *beta*, mais atual. Com essas informações foi possível rodar o algoritmo desenvolvido neste trabalho para inferir quais métodos foram modificações ou adicionados, possibilitando a criação de uma lista contendo os métodos que deveriam ser exercitados em uma campanha de regressão, referente a tais versões.

O resultado dessa execução apresentou um total de ***159 métodos*** únicos. Para verificar a autenticidade dos métodos identificados, conferimos mediante documentos que apresentavam as modificações em código realizados por versão e também do código fonte, que os dados coletados estavam corretos. Assim constatou-se, para este exemplo, verdadeira a hipótese em questão, possibilitando o cálculo de uma quantidade alvo de métodos a serem cobertos em uma campanha de regressão, baseado na diferença entre as versões.

Fizemos o mesmo procedimento sobre a aplicação H, para um outro intervalo de regressão, e foi possível calcular um arquivo que continha ***68 métodos***. Ou seja, de uma versão *alfa* para uma *beta*, 68 métodos foram modificados ou adicionados a esta aplicação, esses representando os métodos de maior potencial de falhas, que precisam ser testados. Esse valor foi verificado utilizando código fonte da aplicação e também as notas de desenvolvimento de cada versão lançada.

4.2 Segunda Hipótese

Objetivo: É possível medir o percentual de cobertura de código de uma campanha de testes através da técnica *Sample Based Profile (SBP)*.

4.2.1 Considerações

Ao se obter o arquivo *TBC* de uma aplicação, e também os dados de cobertura referentes a cada teste de uma campanha, é possível medir o percentual de cobertura alcançado em cima das modificações de uma campanha de regressão. Baseando-se nos dados da primeira hipótese, onde foi calculado o arquivo *TBC* pra a aplicação **A**, realizamos o monitoramento da campanha de

[1] Commit - Nome dado a um conjunto de modificações registrado em um sistema de controle de versão.

testes selecionados para uma regressão referente às versões *alfa* e *beta*, possibilitando o cálculo do percentual coberto alcançado.

4.2.2 Experimento

Com o nosso aplicativo de cobertura denominado de *Cobertura.jar*, devidamente configurado sob a técnica escolhida, realizamos o monitoramento da aplicação **A** para duas campanhas selecionadas, com o objetivo de medir a cobertura de código dos testes selecionados para a regressão desejada. Cada teste executado gerava um arquivo de saída contendo os métodos cobertos que, ao agrupar o resultado de todos os testes, daria o conjunto de métodos cobertos na campanha. O arquivo *TBC* continha ***159 métodos*** modificados entre as versões utilizadas, e o percentual de cobertura calculado baseada nos métodos cobertos pela campanha se encontra na Tabela **4.1**.

Tabela 4.1: Cobertura Alcançada Por Campanha

Campanhas	Testes selecionados	Cobertura Alcançada
Campanha A	57 Testes	57,8%
Campanha B	40 Testes	64,1%

Nossa conclusão para este experimento foi a de que a cobertura de código sobre a campanha A, **57,8% [57/159]**, teve uma cobertura menor que a selecionada para a campanha B **64,1% [102/159]**. Isso ocorreu pois alguns testes selecionados para a campanha B podiam ser mais relevantes para as modificações da regressão, que os da campanha A. Com isso, foi possível comprovar a hipótese em questão, mas também levantar questionamentos sobre a confiabilidade dos dados em questão, sendo proposta uma análise mais detalhada na próxima hipótese.

4.3 Terceira Hipótese

Hipótese: Dadas as aplicações selecionadas neste estudo, a cobertura obtida pela técnica *Sample Based Profile* não é única, mesmo usando uma taxa de captura recomendada pela Google de uma captura a cada 1000μs.

4.3.1 Considerações

Para analisar a presente hipótese, vamos utilizar a técnica *SBP* da biblioteca *ddmlib*, que se baseia na coleta de amostras periódicas à pilha de rastreamento de uma aplicação, como explicada anteriormente na Seção 2.3. O período de coleta é definido em microssegundos e tem como padrão o valor 1000μs, podendo ser alterado.

4.3.2 Experimento

Com o *Cobertura.jar*, devidamente configurado sob a técnica escolhida, realizamos o monitoramento da execução nas 3 (três) aplicações **A, H e C** em modo de depuração. Um caso de teste foi escolhido de forma aleatória da base de testes de cada aplicação. Em seguida, cada teste escolhido foi executado **10 vezes** consecutivas da mesma maneira, sendo coletada uma cobertura (quantidade de métodos exercitados) para cada execução, contendo os métodos capturados na sessão. A Tabela **4.2** apresenta os dados coletados para cada aplicação utilizando a técnica escolhida com período de 1000μs.

Observando a Tabela 4.2, podemos notar que a quantidade de métodos cobertos em cada execução, por aplicação, não é a mesma. Isto está relacionado ao contexto altamente concorrente e complexo de aplicações de celulares. A técnica mostrou-se bem instável quanto a quantidade de métodos coletados, apresentando variações durante as execuções, diminuindo sua confiabilidade em relação a informação de cobertura.

A grande diferença entre a quantidade de métodos por aplicação é facilmente explicada devido às diferentes naturezas de cada aplicação testada.

Tabela 4.2: Métodos Coletados Por Aplicação - *Sample Based Profile* - 1000μs

Aplicações	Quantidade de Métodos Cobertos									
Aplicação A	34	33	28	28	91	28	27	27	27	27
Aplicação H	222	197	229	184	202	189	231	196	224	197
Aplicação C	2571	2189	2041	2325	2553	2579	2193	2413	2620	2033

4.4 Quarta Hipótese

Objetivo: Dadas as aplicações selecionadas neste estudo, é possível monitorá-las utilizando as técnicas *Sample Based Profile*, *Method Tracer* e *Activity Manager*.

4.4.1 Considerações

Verificado que a técnica *SBP* possui alguns problemas de instabilidade na última hipótese, foi pensado em analisar todas as técnicas identificadas ao longo da pesquisa, a fim de melhor concluir sobre a captura dos dados realizada por elas.

A primeira técnica, já previamente explicada, será *SBP*, onde utilizaremos os dados coletados para a hipótese anterior, presentes na Tabela 4.2. A segunda delas, chamada *Method Tracer* (MT), é um *profiler* de CPU que se baseia na instrumentação do código compilado em tempo de execução, não necessitando de alteração prévia do código fonte. Esta técnica se encontra implementada na biblioteca *ddmlib*. A terceira técnica, chamada *Activity Manager* (AM), também é um *profiler* de CPU funcionando de forma semelhante à primeira técnica,

porém encontra-se implementada de forma mais acessível através do aplicativo de depuração (ADB) do Android.

A segunda e a terceira técnicas necessitam de uma aplicação alvo a ser monitorada, como entrada, e realizam o monitoramento de forma contínua, exigindo mais do dispositivo, podendo apresentar quedas de desempenho e travamentos. Assim como na *terceira hipótese*, é esperado coletar dados provenientes dessas técnicas, permitindo a obtenção dos métodos executados ao realizar ações sobre uma aplicação, sem realizar instrumentação do código fonte da aplicação.

4.4.2 Experimento

Semelhante ao experimento realizado para coletar os dados da *terceira hipótese*, utilizamos o *Cobertura.jar* devidamente configurado para utilizar também a técnica *MT*, bem como o uso do ADB para mensurar a técnica *AM*, sendo executados de forma independente uma da outra. Um caso de teste foi escolhido de forma aleatória, para cada aplicação **A, H e C**. Em seguida, cada teste foi executado **10 vezes** consecutivas da mesma maneira, sendo coletado uma saída para cada execução, contendo os métodos capturados na sessão. As tabelas **4.2 (SBP)**, **4.3 (MT)** e **4.4 (AM)** apresentam os dados coletados para cada aplicação utilizando as técnicas citadas anteriormente.

Tabela 4.3: Métodos Coletados Por Aplicação - *Method Tracer*

Aplicações	Quantidade de Métodos Cobertos - Method Tracer									
Aplicação A	52	49	49	49	49	49	49	49	49	49
Aplicação H	364	364	364	364	364	364	362	366	301	342
Aplicação C	-	-	-	-	-	-	-	-	-	-

Tabela 4.4: Métodos Coletados Por Aplicação - *Activity Manager*

Aplicações	Quantidade de Métodos Cobertos - Activity Mnager									
Aplicação A	60	42	42	42	42	42	42	42	42	42
Aplicação H	292	346	290	271	307	307	340	353	353	273
Aplicação C	-	-	-	-	-	-	-	-	-	-

Ao obter os dados referentes à execução dos testes sob uma aplicação, vimos que era possível sim utilizar as técnicas apresentadas nesta seção para saber quais métodos estavam sendo exercitados durante a execução dos testes. Assim como na *terceira hipótese*, foi possível constatar uma variação dos métodos cobertos em cada execução, mesmo reproduzindo todas às vezes da mesma forma o teste selecionado. Como podemos observar na Figura **4.1**, *SBP* apresentou grande instabilidade para as aplicações **C e H** e se mostrou mais estável na aplicação **A**. Já *MT* apresentou grande estabilidade nas aplicações **A** e **H**, como podemos observar na

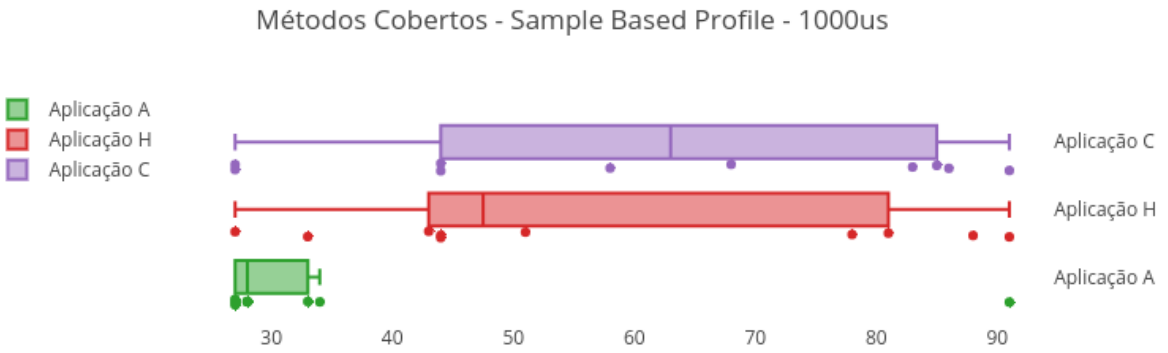

Figura 4.1: Métodos Coletados - Boxplot - *Sample Based Profile* - 1000us

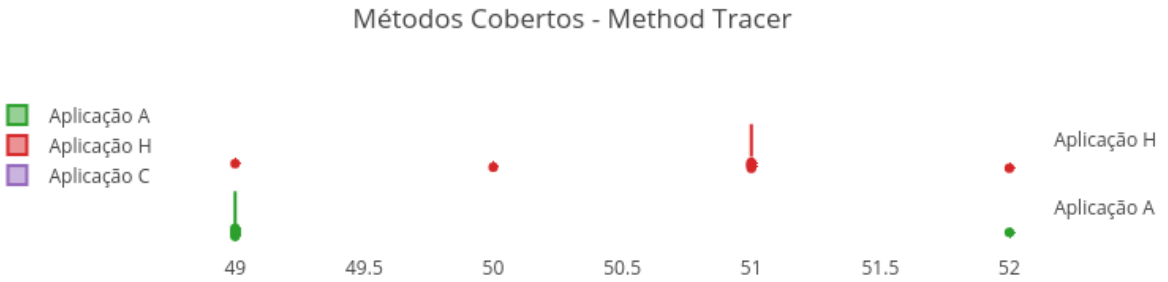

Figura 4.2: Métodos Coletados - Boxplot - *Method Tracer*

Figura **4.2**. Por fim, *AM* mostrou-se estável na aplicação **A** e instável na **H**, como visto na Figura **4.3**.

Em ambas técnicas não foi possível obter qualquer dado referente ao monitoramento da **aplicação C**. Isso ocorreu devido à sobrecarga gerada por essas técnicas, gerando travamentos e outros comportamentos adversos, fazendo com que a aplicação não executasse como esperado.

Podemos concluir que, das técnicas estudadas, *MT* se destacou como a mais estável, e *SBP* como a mais instável.

4.5 Quinta Hipótese

Objetivo: A técnica *Sample Based Profile 1000*μs captura uma quantidade de métodos executados menor que as técnicas *Method Tracer* e *Activity Manager*

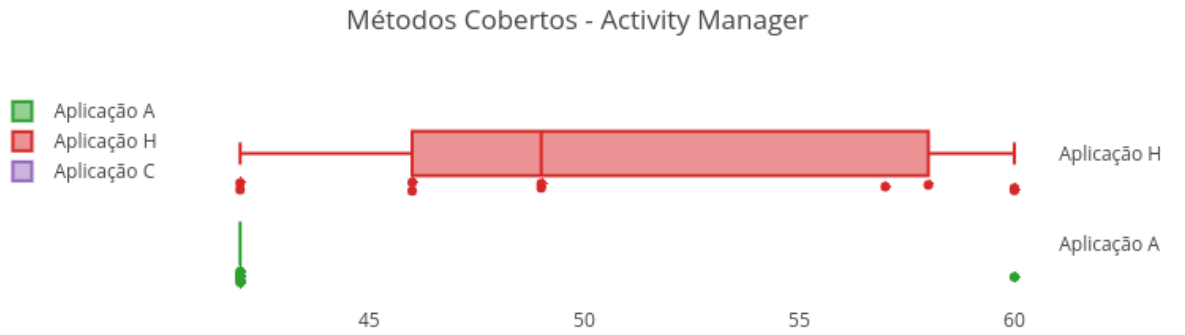

Figura 4.3: Métodos Coletados - Boxplot - *Activity Manager*

4.5.1 Experimento

Avaliando as tabelas **4.2**, **4.3** e **4.4**, geramos os gráficos das figuras **4.4** e **4.5**. Com base nestes gráficos, fica mais fácil tirar conclusões sobre a quantidade de métodos capturados para cada técnica, bem como avaliar sua instabilidade.

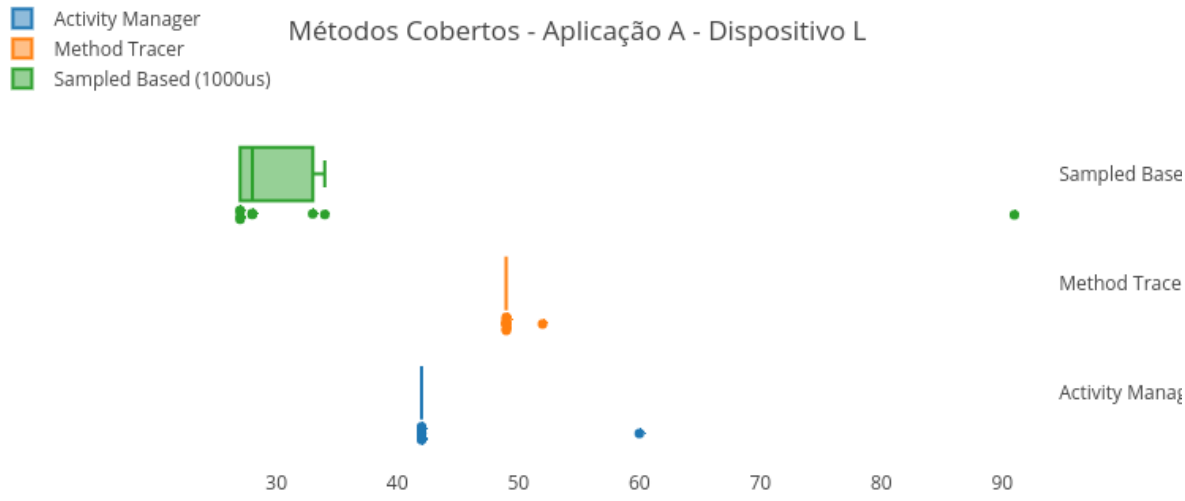

Figura 4.4: Métodos Coletados - Boxplot Panorâmica - Aplicação A

De acordo com as figuras **4.4** e **4.5**, podemos ver que a quantidade de métodos coletados pela técnica *SBP* é menor que as demais, excluindo a aplicação C. Houve uma sobrecarga e mudança do comportamento da aplicação nas técnicas *MT* e *AM*, resultando em nenhum dado gerado para a aplicação C, por isso os traços presente nas tabelas. Com isso podemos concluir que, utilizando o período padrão de 1000μs, não é possível coletar a mesma quantidade de métodos que as demais técnicas. Porém, esta frequência nos permitiu realizar as execuções por

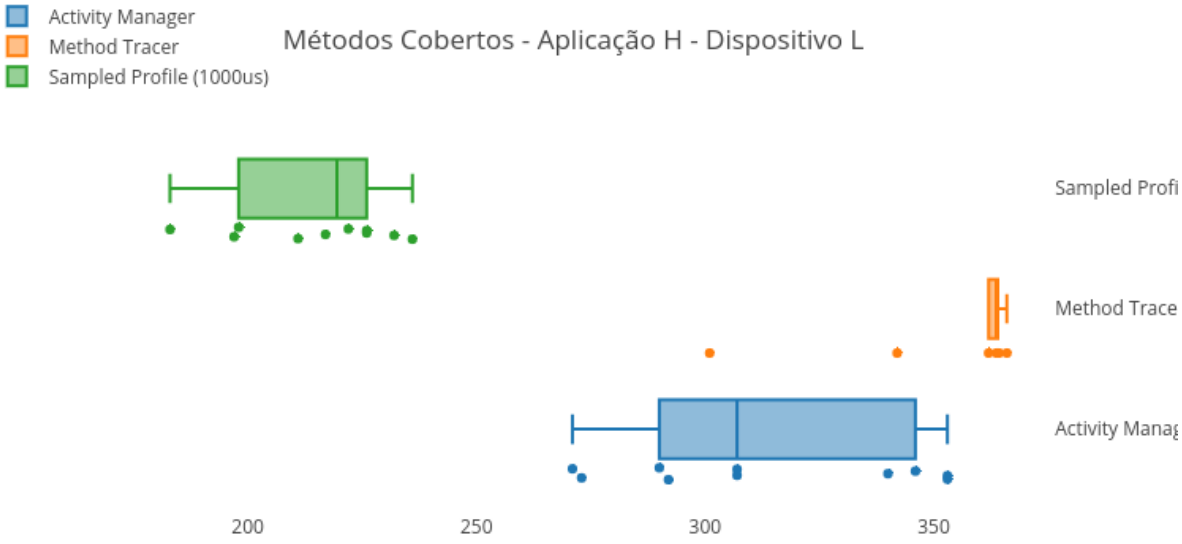

Figura 4.5: Métodos Coletados - Boxplot Panorâmica - Aplicação H

completo (sem interferências), independente da aplicação testada.

Outro ponto relevante é quanto à confiabilidade das técnicas. Podemos observar que *SBP* possui uma alta variabilidade. Já *AM* parece variar de acordo com a aplicação testada, enquanto que *MT* se manteve mais estável diante de todas as execuções e aplicações testadas.

4.6 Sexta Hipótese

Objetivo: A quantidade de métodos capturados utilizando a técnica *Sample Based Profile* é inversamente proporcional ao intervalo de tempo de coleta das amostras, desde que o período de amostragem seja superior a 100μs.

4.6.1 Experimento

Semelhante ao experimento realizado para coletar os dados nas hipóteses anteriores, utilizamos o *Cobertura.jar* devidamente configurado para utilizar a técnica *SBP* com diferentes taxas. Um caso de teste foi criado tendo conhecimento do código da aplicação, visando entender melhor os métodos capturados. Em seguida, o teste foi executado **10 vezes** consecutivas da mesma maneira, sendo coletada uma saída para cada execução, contendo os métodos capturados na sessão (ver **Tabela 4.5**).

Avaliando a **Tabela 4.5** e o gráfico relacionado na **Figura 4.6**, é possível tirar algumas conclusões sobre a quantidade de métodos capturada diante da técnica em questão, para diferentes frequências de amostra.

À medida que o período de amostragem aumentava (até o limite de 100μs), a quantidade

Tabela 4.5: Comparativo de Métodos Coletados - Aplicação A

Técnica	Métodos Coletados - Aplicação A									
Activity Manager	60	42	42	42	42	42	42	42	42	42
Method Tracer	52	49	49	49	49	49	49	49	49	49
Sample Based Profile - 1000us	34	33	28	28	91	28	27	27	27	27
Sample Based Profile - 500us	50	36	32	26	32	26	33	33	32	28
Sample Based Profile - 100us	52	41	39	40	42	38	40	35	34	42
Sample Based Profile - 30us	40	39	40	40	41	40	35	37	33	41

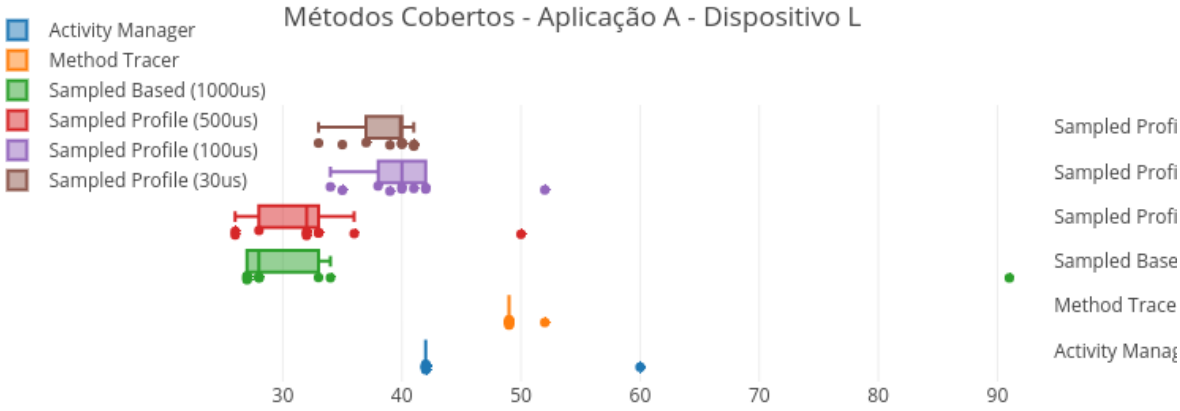

Figura 4.6: Boxplot Métodos Coletados - Boxplot - Aplicação A

de métodos coletados diminuía. Tal comportamento se manteve em outras aplicações, mostrando um padrão, comprovando a hipótese em questão.

Analisando de forma semelhante (ver **Tabela 4.6** e **Figura 4.7**), só que agora considerando uma outra aplicação, podemos ver que o comportamento do *SBP* chegou a coletar mais métodos que as demais técnicas ao atingir uma período inferior a 100μs), mas o resultado não é conclusivo porque a relação de inversão não foi mantida conforme a hipótese.

Portanto, o período até 100μs seria o que confere a melhor coleta mesmo considerando os intervalos de variação e que valida nossa hipótese.

Tabela 4.6: Comparativo de Métodos Coletados - Aplicação H

Técnica	Métodos Cobertos - Aplicação H									
Activity Manager	292	346	290	271	307	307	340	353	353	273
Method Tracer	364	364	364	364	364	364	362	366	301	342
Sample Based Profile - 1000us	222	197	229	184	202	189	231	196	224	197
Sample Based Profile - 500us	340	270	272	264	256	271	325	269	272	285
Sample Based Profile - 100us	374	319	321	377	378	411	377	329	383	418
Sample Based Profile - 30us	358	338	376	367	392	412	415	484	370	374

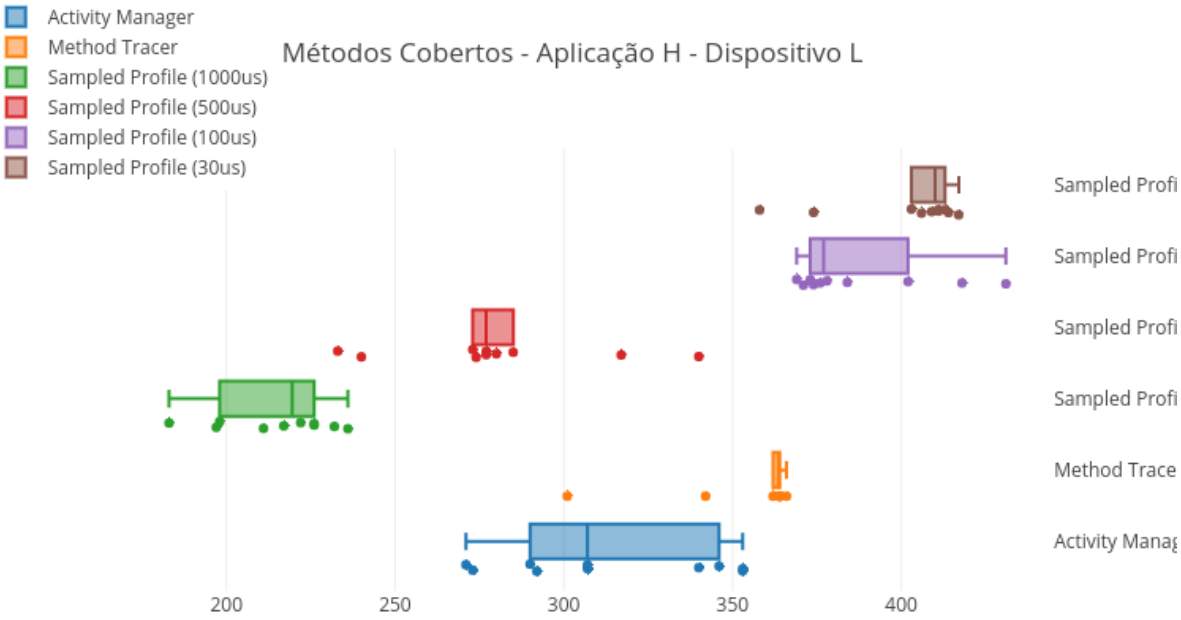

Figura 4.7: Boxplot Métodos Coletados - Boxplot - Aplicação H

4.7 Sétima Hipótese

Objetivo: Usando a mesma técnica de *Sample Based Profile*, verifica-se que o conjunto de métodos coletados com um período (intervalo de coleta) maior não está contido no conjunto com período menor.

4.7.1 Considerações

No experimento anterior, vimos que a quantidade de métodos aumentava à medida que diminuímos o intervalo de coleta até o limite de 100μs. Porém, não analisamos lá se esta coleta adicional mantinha a anterior. Isto é, era conservativa. Aqui realizamos um experimento simples que comparava os métodos coletados de acordo com o período de 1000μs e a de 30μs.

Foi utilizada uma **aplicação K**, criada exclusivamente para fins de validação desse experimento como uma forma de *baseline*. Trata-se de uma listagem simples de cartões (ou seja, não requer nada de especial do hardware do celular), criada para fins de teste da hipótese em questão. Um teste foi criado para esta aplicação e o mesmo executado **20 vezes** consecutivas e iguais, seguindo as diretrizes utilizadas nas hipóteses passadas. O resultado da coleta está presente na **Tabela 4.7**

Tabela 4.7: Comparativo de Métodos Coletados - Aplicação K

Frequências	Métodos Cobertos - Aplicação K																			
Sample Based Profile - 1000us	13	16	17	18	14	15	12	14	15	15	14	13	13	15	16	10	13	12	16	17
Sample Based Profile - 500us	17	16	13	16	19	15	19	20	15	17	12	13	15	15	15	15	15	18	18	15
Sample Based Profile - 100us	21	20	23	19	14	18	22	23	17	21	24	17	16	19	19	25	19	19	20	19
Sample Based Profile - 30us	24	23	20	24	26	25	17	28	19	20	25	21	25	21	22	21	22	29	22	20

4.7.2 Experimento

Comparando os métodos coletados para 1000μs com o de 30μs, nas **20 execuções**, foi possível obter a quantidade de métodos que não estavam presentes na de 30μs, mas apareciam na de 1000μs. Podemos acompanhar o resultado dessa análise na **Tabela 4.8**. Mesmo sabendo que um intervalo de 30μs aumenta a quantidade de métodos coletados com relação aos intervalos maiores, a possibilidade de ocorrer tais perdas de dados motivou a hipótese atual.

Tabela 4.8: Comparativo de Frequências - 1000μs para 30μs - Aplicação K

Quantidade de Métodos	Diferença de Coleta - 1000us para 30us - Aplicação K																			
Faltantes	0	0	1	1	0	3	3	0	0	1	0	0	0	0	2	0	0	0	0	3
Novos	11	7	4	7	12	13	8	14	4	6	11	8	12	6	8	11	9	17	6	6

Neste cenário onde a aplicação é pequena podemos notar um baixo número de métodos faltantes (presente em 1000μs e ausente em 30μs), deixando clara a existência de perdas, mesmo possuindo um número de métodos novos bem superior. Fazendo a mesma análise para uma aplicação H na **Tabela 4.9**, fica ainda mais evidente a perda entre as trocas de frequência, podendo assim concluir que a hipótese é verdadeira.

Tabela 4.9: Comparativo de Frequências - 1000μs para 30μs - Aplicação H

Quantidade de Métodos	Diferença de Coleta - 1000us para 30us - Aplicação H																			
Faltantes	47	14	44	4	9	9	20	4	39	6	4	8	8	9	10	3	22	5	23	8
Novos	183	155	191	187	199	232	204	292	185	183	238	193	187	214	202	205	192	193	166	183

4.8 Discussão geral

Uma vantagem a técnica *SBP* foi que os dispositivos apresentavam poucos travamentos em comparação com as outras técnicas empregadas, sendo uma forma de se obter dados de cobertura sem o uso de instrumentação e sem gerar tanta sobrecarga, diante do cenário proposto pelo trabalho.

À medida que os valores ficavam muito pequenos, era possível notar alguns travamentos no dispositivo em uso. Executamos em outro dispositivo com uma quantidade de recursos superior e foi possível notar que alguns valores de frequência que geravam travamentos no primeiro dispositivo, conseguiam rodar sem sobrecarregar no segundo, mostrando-se também uma estratégia dependente da quantidade de recursos disponível no dispositivo.

Foi possível levantar também que, das técnicas avaliadas, *SBP* foi a que se mostrou mais instável quanto a quantidade de métodos capturados durante as execuções. Porém ela foi a única técnica que permitiu executar todas as aplicações sem gerar travamentos aos dispositivos. Além disso, a técnica *MT* se destacou como a mais estável, porém gerando muita sobrecarga no dispositivo. Já a abordagem *AM* gerou também sobrecarga no dispositivo, se destacando apenas pela simplicidade de uso, já que a mesma se encontra implementada nas funcionalidades do ADB.

5

Conclusão

Neste trabalho nós apresentamos uma estratégia para calcular a cobertura de código em componentes Android, sem o uso de instrumentação de código fonte, utilizando *profilers* de CPU. Nós desenvolvemos uma ferramenta chamada AutoTestCoverage^{C}, em dois módulos, que implementa os conceitos apresentados ao longo deste trabalho. O primeiro módulo, chamado *To Be Covered*, presente na Seção 3.2.1, objetiva o cálculo dos métodos que sofreram modificações entre duas versões de uma aplicação. Já o segundo módulo, chamado *Manual Coverage* presente na Seção 3.2.2, trata-se de uma adaptação dos *profilers* estudados para o cenário em questão, permitindo a obtenção dos métodos executados em tempo de execução.

Realizamos alguns experimentos ao longo deste trabalho utilizando dados reais obtidos através da nossa parceira, tendo auxílio de testadores experientes que puderam acompanhar e validar a realização destes experimentos, com o objetivo de validar algumas hipóteses. Dos experimentos realizados, pudemos levantar algumas conclusões relevantes sobre a solução proposta ao problema da parceira. Apesar de os resultados obtidos em algumas hipóteses mostrar certa incerteza quanto à solução proposta, no geral a ferramenta foi vista com bons olhos pela parceira, já que a mesma consegue trazer ganhos à qualidade do processo de testes da empresa. Por exemplo, medindo os métodos executados através de *profilers* de sistema, obtemos dados atualmente inexistentes quanto à cobertura de código dos testes realizados. Além disso, ter uma maneira de se calcular quais métodos foram modificados através do arquivo *TBC*, mostrou-se uma informação valiosa para alcançar uma melhor cobertura.

Foi possível levantar pontos positivos e negativos quanto à abordagem *Sample Based Profile* em relação às outras citadas, *Method Tracer* e *Activity Manager*. O primeiro ponto de destaque para a abordagem *SBP* se deu na baixa sobrecarga gerada por ela à aplicação em teste, diferente das demais que apresentavam perdas de performance e travamentos. Esta vantagem é possível graças à estratégia de consultas periódicas à pilha de chamadas da aplicação, como explicados anteriormente, sendo diferente das demais que realizam instrumentação do código compilado em tempo de execução, gerando uma maior sobrecarga. Tal vantagem é um grande diferencial diante do nosso cenário, já que uma das necessidades era o baixo impacto nas aplicações enquanto se executava o teste para obter uma cobertura sem interferências, além

de poder monitorar aplicações de natureza mais complexas. Partindo do fato que a abordagem *SBP* permite alterar o período de coleta das amostras, temos outro ponto que consideramos positivo em relação às outras abordagens (que não permitem ajustar nenhum parâmetro da execução): permite-se ajustar a abordagem de acordo com a quantidade de recursos disponíveis, acarretando em uma forma mais adaptável. Em contrapartida, *SBP*, por se basear em coleta de amostras, possui uma alta variabilidade dos dados coletados, gerando uma incerteza quanto a esta informação. Em muitos casos analisados, a abordagem apresentou uma quantidade de métodos coletados inferior às outras abordagens, mesmo reduzindo o período de amostras em valores relevantes a ponto de aplicações apresentarem queda de performance. Foi possível constatar também outro ponto negativo na última hipótese, quando comprovamos a existência de perda de informação entre diferentes taxas de coleta das amostras. Porém este ponto não pode ser comparado com as demais abordagens já que não é possível ajustar nenhum parâmetro da execução, tornando ainda incerta a conclusão de perda de informações à respeito das demais abordagens.

Mesmo sabendo que precisamos nos aprofundar nas abordagens e tentar diferentes estratégias, foi possível constatar que tivemos um grande avanço em analisá-las. Sendo este um trabalho inicial, os resultados se mostraram positivos, apontando que estamos seguindo no caminho certo, apenas precisando de alguns ajustes para alcançar melhores resultados.

5.1 Trabalhos Relacionados

Os trabalhos relacionados à pesquisa em questão dão enfase à necessidade de utilizar ferramentas de análise dinâmica em dispositivos móveis, buscando obter informações com o objetivo de melhorar a qualidade do produto criado. Os trabalhos listados a seguir não necessariamente utilizam as tecnologias utilizadas neste trabalho, mas foram os trabalhos que mais se relacionavam com partes do nosso contexto e por isso achamos necessário uma comparação entre o presente trabalho e estes.

O trabalho reportado em MACHIRY; TAHILIANI; NAIK (2013) relata, assim como o nosso, a criação de uma ferramenta de análise dinâmica para aplicações Android. Enquanto nós focamos numa realidade de testes manuais, o foco do trabalho em questão é outro, partindo para técnicas automatizadas para testar as aplicações. Este trabalho possui uma característica em comum com o nosso: utilizam a métrica de cobertura de código como forma de avaliação para medir a eficiência da ferramenta. Diferente do nosso trabalho, eles utilizam um software que realiza instrumentação do código fonte, chamado *Emma*[1], utilizando uma abordagem que altera o código fonte para que seja possível monitorar as regiões executadas durante um teste. Em nosso trabalho, o código fonte se fez necessário apenas para calcular as modificações entre duas versões, não sendo necessário tê-lo para realizar o monitoramento.

Um segundo trabalho mais alinhado com o nosso, é apresentado em AZIM; NEAMTIU

[1] Emma - http://emma.sourceforge.net/

(2013), que também relata a criação de uma ferramenta para análise dinâmica de aplicações, porém com o foco em exploração sistemática. Esta diz-se não precisar do código fonte para realizar as execuções exploratórias na aplicação, pois realiza análise estática do código compilado da aplicação, técnica que achamos interessante e possivelmente abordaremos nos trabalhos futuros. Para avaliar a ferramenta desenvolvida em AZIM; NEAMTIU (2013), foi utilizado a métrica de cobertura de telas e de métodos, este último se assemelhando ao nosso trabalho. Para os coletar os métodos executados no trabalho em questão, foi utilizado o *profiler* presente no ADB chamado *Activity Manager*, que comparamos com outras abordagens em nosso trabalho. Porém, nada foi explicado a respeito da abordagem utilizada no trabalho relacionado, diferente do que buscamos fazer. Além disso, ao analisar os dados referentes à abordagem utilizada no trabalho referenciado, começamos a levantar questionamentos sobre a validade dos dados de cobertura presentes no trabalho, visto que, em múltiplas execuções iguais, o resultado apresentou variações em nosso trabalho, sendo que nenhum comentário a respeito de variações foi apresentado no trabalho analisado.

Nenhum outro material relevante e de cunho científico foi encontrado a respeito das abordagens discutidas neste trabalho, se tratando ainda de algo novo a ser explorado e investigado a fundo. Outros trabalhos envolvendo técnicas de monitoramento instrumentada foram encontrados, porém tornaria os comentários redundantes a respeito. O trabalho descrito em AZIM; NEAMTIU (2013) foi o único relato que encontramos do uso de *profilers* para obter os métodos executados sem instrumentar o código, porém como vimos no presente trabalho, a abordagem *Activity Manager* deixa a desejar em alguns cenários, incluindo em nosso contexto industrial, apresentando quedas de desempenho e travamentos.

No geral é importante mencionar que existem diversos trabalhos que envolvem métricas de cobertura e ferramentas para análise dinâmica em dispositivos Android, não necessariamente relacionados diretamente ao nosso, mas que fortalecem a ideia trabalhada e mostra que existe uma demanda forte por informações em cima de uma plataforma que gera muitos dados, mas que ainda exige um grande esforço para se extraí-los.

5.2 Trabalhos Futuros

Como trabalhos futuros, precisamos realizar novos experimentos em diferentes cenários, com o objetivo de descobrir mais sobre a abordagem sugerida e as demais mencionadas no trabalho, a fim de compreender seu comportamento. Através do estudos em diferentes aplicações, pretendemos entender de fato o motivo de ocorrer variações na quantidade de métodos coletados durante um monitoramento.

Queremos também melhorar o processo de cálculo das modificações entre versões para que não seja necessário o código da aplicação. Como citado nos trabalhos relacionados, utilizando apenas análise estática do código compilado obtido através do arquivo *apk* que pode ser extraído diretamente do dispositivo testado, é possível reduziríamos uma dependência na

hora de analisar as modificações, possibilitando uma maior praticidade no uso da ferramenta.

Além disso, esperamos utilizar uma frequência sobre a abordagem *SBP* que permita rodar de forma satisfatória em qualquer dispositivo e aplicação, e com o auxilio de um grafo de chamadas de funções (CFG) de uma aplicação, complementar a informação de cobertura obtida durante uma execução, já que, durante este trabalho, foi constatado que existe sim, perdas durante a captura.

Com isso em mente, queremos atingir diferentes times de teste dentro da empresa parceira, através de adaptações para que seja possível rodar não só sobre testes de regressão, mas também sobre execuções exploratórias e automatizadas, aumentando assim nossa quantidade de dados para analisar e sugestões de melhorias de usuários.

Referências

AMALFITANO, D.; FASOLINO, A. R.; TRAMONTANA, P. A GUI Crawling-Based Technique for Android Mobile Application Testing. In: IEEE FOURTH INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION WORKSHOPS, 2011. **Anais...** [S.l.: s.n.], 2011. p.252–261.

AZIM, T.; NEAMTIU, I. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. **SIGPLAN Not.**, New York, NY, USA, v.48, n.10, p.641–660, Oct. 2013.

CUI, B. et al. Code Comparison System based on Abstract Syntax Tree. In: IEEE INTERNATIONAL CONFERENCE ON BROADBAND NETWORK AND MULTIMEDIA TECHNOLOGY (IC-BNMT), 2010. **Anais...** [S.l.: s.n.], 2010. p.668–673.

DO, Q. et al. Regression Test Selection for Android Applications. In: IEEE/ACM INTERNATIONAL CONFERENCE ON MOBILE SOFTWARE ENGINEERING AND SYSTEMS (MOBILESOFT), 2016. **Anais...** [S.l.: s.n.], 2016. p.27–28.

HUANG, S.-Y. et al. ABCA: android black-box coverage analyzer of mobile app without source code. In: IEEE INTERNATIONAL CONFERENCE ON PROGRESS IN INFORMATICS AND COMPUTING (PIC), 2015. **Anais...** [S.l.: s.n.], 2015. p.399–403.

KELL, S. et al. The JVM is Not Observable Enough (and What to Do About It). In: SIXTH ACM WORKSHOP ON VIRTUAL MACHINES AND INTERMEDIATE LANGUAGES, New York, NY, USA. **Proceedings...** ACM, 2012. p.33–38. (VMIL '12).

LIANG, S.; VISWANATHAN, D. Comprehensive Profiling Support in the Java Virtual Machine. In: COOTS. **Anais...** [S.l.: s.n.], 1999.

MACHIRY, A.; TAHILIANI, R.; NAIK, M. Dynodroid: an input generation system for android apps. In: JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, 2013., New York, NY, USA. **Proceedings...** ACM, 2013. p.224–234. (ESEC/FSE 2013).

MAGALHãES, C. et al. Evaluating an Automatic Text-based Test Case Selection Using a Non-Instrumented Code Coverage Analysis. In: ND BRAZILIAN SYMPOSIUM ON SYSTEMATIC AND AUTOMATED SOFTWARE TESTING, 2., New York, NY, USA. **Proceedings...** ACM, 2017. p.5:1–5:9. (SAST).

NEAMTIU, I.; FOSTER, J. S.; HICKS, M. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. **SIGSOFT Softw. Eng. Notes**, New York, NY, USA, v.30, n.4, p.1–5, May 2005.

REIS, J.; MOTA, A. Aiding exploratory testing with pruned GUI models. **Information Processing Letters**, [S.l.], v.133, p.49 – 55, 2018.

ROTHERMEL, G.; HARROLD, M. J. A Safe, Efficient Regression Test Selection Technique. **ACM Trans. Softw. Eng. Methodol.**, New York, NY, USA, v.6, n.2, p.173–210, Apr. 1997.

SPINELLIS, D. Version control systems. **IEEE Software**, [S.l.], v.22, n.5, p.108–109, Sept 2005.

YEH, C.-C.; HUANG, S.-K.; CHANG, S.-Y. A Black-box Based Android GUI Testing System. In: PROCEEDING OF THE 11TH ANNUAL INTERNATIONAL CONFERENCE ON MOBILE SYSTEMS, APPLICATIONS, AND SERVICES, New York, NY, USA. **Anais...** ACM, 2013. p.529–530. (MobiSys '13).

ZHENG, Y. et al. Comprehensive Multiplatform Dynamic Program Analysis for Java and Android. **IEEE Software**, [S.l.], v.33, n.4, p.55–63, July 2016.