Mineração de regras para solução de problemas relacionados à fragmentação do Android

Marcus Adriano Ferreira Pereira, Marcelo de A. Maia

Faculdade de Computação - FACOM
Universidade Federal de Uberlândia (UFU)

marcusadriano.pereira@gmail.com,marcelo.maia@ufu.br

Abstract. *Android operating system is highly fragmented resulted from a combination of different versions of the operating system and thousands of different devices. Ideally, the developer is required to grasp the specific features of each available device (e.g., sensors, graphical resources, among others) to ensure an adequate user experience. However, large fragmentation level makes it impossible to systematically analyze all the possible options, which is responsible for failures in situations that occur only in specific versions of the API and/or device. An alternative for dealing with this problem is to rely on the crowd knowledge to find possible corrections for typical failures linked to fragmentation problems. This study aims to mine Android projects available on GitHub for associations between elements of the Android API that co-occur in code to mitigate a fragmentation-related problem. We observed that some rules are mainly linked to the user interface, and there is a concern of developers to use features of the new APIs, but also embrace users who use older device models but do not have enough resources for such. The rules could be used for recommending patches where the respective method calls occur.*

Resumo. *O sistema Android é altamente fragmentado resultado de uma combinação entre diferentes versões da API ligada ao sistema operacional e milhares de dispositivos diferentes. Idealmente, o desenvolvedor é levado a conhecer as características específicas de cada dispositivo disponível (e.g., sensores, recursos gráficos, dentre outros) para garantir que o usuário final do seu aplicativo tenha uma ótima experiência. Entretanto, a diversidade ligada à imensa fragmentação torna inviável uma análise sistemática de todas as opções possíveis, o que é responsável pelo aparecimento de falhas em situações que ocorrem apenas em versões específicas da API e/ou dispositivo. Uma alternativa para lidar com este problema é se apoiar no conhecimento da multidão para encontrar possíveis correções para falhas típicas ligadas a problemas de fragmentação. Este estudo tem como objetivo minerar projetos Android disponíveis no GitHub em busca de associações entre elementos da API do Android que co-ocorram no código para contornar um problema de fragmentação. Como resultado, observou-se que algumas regras estão ligadas principalmente a interface do usuário, há uma preocupação dos desenvolvedores em utilizar recursos das novas APIs e também alcançarem usuários que usam modelos mais antigos de dispositivos, mas que não possuem recursos suficientes para tal. Tais regras podem ser usadas na recomendação de patches onde as chamadas de métodos envolvidos ocorrem.*

1. Introdução

O Android é o sistema operacional para dispositivos móveis mais popular atualmente, ocupando aproximadamente 87% do mercado[1]. A alta adoção faz com que novos requisitos sejam rapidamente endereçados. Por isto, já existem mais de 10 versões diferentes do sistema, com 26 diferentes níveis de API[2] (*Application Programming Interface*). Além disso, como se trata de um sistema aberto, diferentes fabricantes o adotam (e.g., Samsung, LG, Motorola, HTC), fazendo com que atualmente o sistema esteja operando em mais de 16.000[3] tipos de dispositivos diferentes.

O sistema Android dispõe de uma infraestrutura elaborada para criar aplicativos usando a linguagem Java. Os aplicativos podem se adaptar a diferentes dispositivos, por exemplo: é possível criar diferentes arquivos XML de layout para diversos tamanhos de tela e o sistema determina qual layout deverá aplicar com base no tamanho da tela. Desenvolvedores podem consultar a disponibilidade dos recursos do dispositivo em tempo de execução se qualquer recurso do aplicativo exigir hardware específico, como uma câmera ou GPS.

Os dispositivos podem ter instalado diferentes versões do Android e cada versão mais recente, em geral, possui novos elementos de API não disponíveis nas versões anteriores. O desenvolvedor tem liberdade para escolher o nível de API mínimo para seu aplicativo através do seguinte *snippet* presente no manifesto do aplicativo:

```
<uses-sdk
      android:minSdkVersion="8" android:targetSdkVersion="15" />
```

O atributo *minSdkVersion* determina qual será a versão mínima para o funcionamento do aplicativo. Quando o desenvolvedor usa funcionalidades de APIs mais recentes é necessário cautela para garantir o funcionamento do aplicativo caso esta funcionalidade não esteja disponível em APIs mais antigas, causando a incompatibilidade de *software*. Uma prática usada para lidar com este problema é verificar qual a versão corrente do sistema e tomar ações necessárias conforme o resultado desta verificação.

A classe *android.os.Build*[4] fornece algumas constantes que informam ao desenvolvedor sobre o ambiente no qual seu aplicativo é executado. O exemplo abaixo, retirado da classe ACTIONBARIMPL.JAVA do projeto aSM-clone disponível no *Github*[5], exemplifica uma verificação que trata se o aplicativo está sendo executado na *API* 8:

```
if (Build.VERSION.SDK_INT < Build.VERSION_CODES.FROYO) {
   mActionView.onConfigurationChanged(newConfig);
   if (mContextView != null)
      mContextView.onConfigurationChanged(newConfig);
}
```

A alta variabilidade em relação à combinação das diferentes opções de dispositivos com as diferentes versões de APIs do sistema leva a um fenômeno conhecido como **fragmentação do Android**, a qual impõe ao desenvolvedor a necessidade de saber se o

[1] https://goo.gl/PLjVzL
[2] https://developer.android.com/about/dashboards/index.html
[3] https://support.google.com/googleplay/answer/1727131?hl=en-GB
[4] https://developer.android.com/reference/android/os/Build.html
[5] https://github.com/yfli/aSM-clone

dispositivo na qual está sendo executado seu aplicativo possui recursos de *hardware* e/ou *software* para garantir o seu correto funcionamento.

Alguns trabalhos tem pesquisado a fragmentação do Android de maneira seletiva tomando como base alguns projetos populares do Android. Por exemplo, em [Wei et al. 2016], o código-fonte de algumas aplicações populares foi analisado de maneira manual para encontrar *snippets* onde possivelmente há problema relacionados à fragmentação, de forma a usar esta informação para elaborar um sistema de recomendação que, dado um código-fonte qualquer, identifica a ocorrência desses *snippets* e consegue indicar o tratamento adequado.

Entretanto, ainda há uma carência de estudos com uma análise mais abrangente de projetos Android e com automatização na busca por *snippets* de código que mitigam problemas de compatibilidade, para ampliar o conhecimento os pontos críticos de código impactados pela compatibilidade. Além disso, o mapeamento da ocorrência desses *snippets*, pode permitir a detecção de padrões nas soluções propostas em código disponível em repositórios, com o objetivo de facilitar a identificação e o tratamento de problemas cuja causa está relacionada à fragmentação.

Os objetivos deste trabalho são: 1) encontrar *snippets* de código em aplicativos Android disponíveis em repositórios de software, que tentam tratar algum tipo de problema de relacionado à fragmentação do Android 2) encontrar padrões utilizando técnicas de mineração de dados afim de encontrar regras de associação entre a co-ocorrência de chamadas de métodos nos *snippets* de código encontrados que possam ser usadas para entender melhor o problema relacionados à fragmentação e como possíveis regras de recomendação de soluções para este tipo de problema.

O restante deste trabalho está organizado como segue. A metodologia do estudo, com os objetivos, a descrição sobre a coleta e análise dos dados são descritas na Seção 2. Os resultados do estudo são apresentados na Seção 3. Os trabalhos relacionados são citados na Seção 4. As conclusões e os trabalhos futuros são expostos na Seção 5.

2. Metodologia do Estudo

2.1. Coleta de Dados

Os sistemas utilizados neste estudo foram: *GitHub*, *Boa*[6] e *R*[7]. Todos os dados foram retirados do *GitHub* por meio da plataforma *Boa*, onde um total de 66.536 projetos Android foram analisados.

Resumidamente, a plataforma Boa permite explorar um espelho dos projetos *Open Source* disponíveis no *GitHub*. O mais importante desta plataforma é a possibilidade de *visitar* diferentes partes do código e obter informações sobre o mesmo (e.g., revisões, autores do projeto, classes, métodos, expressões).

Com base neste cenário, procuramos buscar *snippets* onde há uma verificação, instrução *IF*, que contenha a expressão android.os.Build.VERSION.SDK_INT ou Build.VERSION.SDK_INT para verificar problemas de compatibilidade relacionados à API e também outras constantes da classe *Build* que ditam o nome do dispositivo, fabricante e algumas informações básicas, que nomearemos como "*outras*

[6]http://boa.cs.iastate.edu/

[7]https://cran.r-project.org/

condições" (e.g. Build.MANUFACTURER, Build.HARDWARE, Build.BRAND, Build.DEVICE, Build.TAGS, Build.ID, Build.TYPE, Build.VERSION.RELEASE). Por exemplo, dado um projeto X e o *snippet* de código abaixo, organizamos os dados conforme a Tabela 1.

```
package com.x.proj;
class A {
   void create() {
      if (Build.VERSION.SDK_INT < Build.VERSION_CODES.FROYO) {
         metodo1();
         metodo2();
      } else {
         metodo3();
         metodo4();
      }
   }
}
```

Tabela 1. Informações obtidas através do *snippet* acima

Projeto	Classe	API	Método	Método Chamado
X	com.proj.A	8	create	metodo1
X	com.proj.A	8	create	metodo2
X	com.proj.A	8	create	metodo3
X	com.proj.A	8	create	metodo4

2.2. Análise de Dados

Inicialmente, serão apresentadas as chamadas de métodos mais prevalentes em situações relacionadas à fragmentação.

Em relação à mineração de regras de associação, para descobrir quais relações são mais frequentes entre os diferentes tipos de métodos chamados, é aplicado o algoritmo *APRIORI* [Agrawal and Srikant 1994] ao conjunto de transações organizadas a partir dos dados existentes conforme a **Tabela 1**. As transações são organizadas por declaração de método, conforme exemplificado na **Tabela 2**. Para cada método declarado, os itens que fazem parte do conjunto de itens da transação são os métodos chamados dentro de IF's relacionados a mitigação de problemas associados à fragmentação.

Tabela 2. Exemplo de Transações

Método	Métodos Chamados
create	{metodo1, metodo2, metodo3, metodo4}

Os limiares empregados foram o mínimo 80% de confiança sobre cada regra encontrada, e em relação ao suporte, o mesmo foi ajustado de tal forma que ocorresse um número relevante de regras que pudessem ser manualmente analisadas. O critério para o percentual de confiança se deve pelo interesse em descobrir inter-relações fortes entre os métodos chamados que ocorrem constantemente a partir das regras encontradas.

3. Resultados

3.1. Prevalência de chamadas e Grupos de regras

Na **Tabela 3**, mostramos quais métodos são chamados mais frequentemente em situações relacionadas ao tratamento da fragmentação.

Tabela 3. Top-10 chamadas mais prevalentes: em condições para APIs e em *outras condições*

Chamadas - API	#	Chamada - outras condições	#
getActionBar	3880	AddDeviceSpecificCapability	696
setVisibility	3374	commit	550
getResources	2654	putBoolean	550
setDisplayHomeAsUpEnabled	2336	set	344
onConfigurationChanged	2179	setPreferenceBooleanValue	315
setDuration	1823	setClassName	174
animate	1660	sendKey	126
super.onPopulateAccessibilityEvent	1416	sleep	121
getWindow	1359	setPreferenceFloatValue	112
setFlags	1331	enterText	84

Em relação à mineração de regras de associação, a análise foi executada sobre 80.974 transações em nosso *dataset*, i.e., linhas como exemplificadas na **Tabela 1**. Mais de 82% das APIs verificadas estão entre a versão 8 e 14, pois de fato essas versões introduziram diversas mudanças no Android principalmente quando se diz respeito a interface gráfica do sistema. As principais regras observadas, estão presente na **Tabela 4**.

Tabela 4. Regras de Associação

Grupo	Regra	Suporte	Confiança
1	{alpha} → {setDuration}	0,59%	94,00%
	{alpha} → {animate}	0,56%	89,00%
	{setListener} → {alpha}	0,49%	88,00%
2	{execute} → {executeOnExecutor}	6,02%	92,00%
	{executeOnExecutor, getActivity} → {execute}	0,49%	100,00%
	{executeOnExecutor, startActivity} → {execute}	0,43%	100,00%
	{executeOnExecutor, getActionBar} → {execute}	0,43%	100,00%
	{setBackground} → {setBackgroundDrawable}	3,10%	89,00%
3	{getNumberOfCameras} → {getCameraInfo}	0,63%	86,00%
	{getCameraInfo, open} → {getNumberOfCameras}	0,46%	93,00%
4	{apply} → {commit}	3,06%	93,00%
	{edit} → {commit}	0,63%	100,00%
	{apply, edit} → {commit}	0,46%	100,00%
5	{setMessage, setPositiveButton, setTitle, show} → {voiceRecording}	1,03%	100,00%
5	{hide,setFlags} → {getActionBar}	4,40%	99,00%
	{getWindow,setFlags} → {getActionBar}	4,40%	94,00%
	{setDisplayHomeAsUpEnabled} → {getActionBar}	13,00%	90,00%
6	{setSupportZoom} → {getSettings}	2,00%	100,00%
	{setIcon,setMessage,setPositiveButton,setTitle,show} → {voiceRecording}	1,00%	100,00%

A semântica das regras encontradas quer dizer que os métodos co-ocorrem. Por exemplo, {`alpha`} → {`setDuration`} toda vez que ocorrer `alpha`, então ocorrerá `setDuration`. Estas regras foram encontradas levando em consideração os métodos chamados que estavam dentro de uma instrução *IF* de um determinado método. O agrupamento das regras se deu por um critério de um determinado grupo ter regras com chamadas comuns entre si ou serem semanticamente relacionados.

3.2. Análise Qualitativa

Na **Tabela 3** os métodos `getActionBar` e `setDisplayHomeAsUpEnabled` além de serem métodos recorrentes, também fazem parte do quinto grupo de regras da **Tabela 4**. Estes métodos são comumente usados para controlar as telas de interação com o usuário do aplicativo, em muitos casos o desenvolvedor opta por proporcionar ao usuário um botão que auxilie voltar a tela anterior, isto é definido através da combinação `getActionBar().setDisplayHomeAsUpEnabled(true)` a partir da API 11.

Outra combinação muito utilizada pelos desenvolvedores, quando a API em tempo de execução é menor que a 16 são os métodos `getWindow` e `setFlags`. Neste caso, o desejo do desenvolvedor é ajustar seu aplicativo para ocupar 100% do *display*, modo *fullscreen*, porém antes da API 16, isso só era possível combinando o `getWindow().setFlags(int, int)`, onde os parâmetros são constantes que definem para qual estado deseja se alterar a tela da aplicação. Após a API 16, os desenvolvedores podem alteram os *flags* de uma tela da aplicação, através dos arquivos XML que configuram toda a interface do usuário, não sendo necessário nenhum código Java.

Os métodos `onConfigurationChanged` e `onPopulateAccessbilityEvent` não fazem parte de alguma regra em questão, mas são métodos recorrentes pelo fato de muitos projetos analisados utilizarem bibliotecas que fazem um gerenciador melhor da interface, melhor dizendo, são bibliotecas que ajudam o desenvolvedor adaptar o seu aplicativo aos diferentes tipos e tamanhos de telas, a biblioteca usada em todos os casos foi a *ActionBarSherlock*[8]. Hoje em dia, o próprio sistema oferece bibliotecas para os desenvolvedores adaptarem seus aplicativos, tornando inviável a utilização de biblioteca de terceiros.

Do lado direito da **Tabela 3** há alguns métodos frequentes quando desenvolvedores filtram dispositivos específico, por exemplo, o método `set`, o quarto mais chamado da lista ocorre em muitos projetos, mas exatamente na mesma classe, mesmo método e um mesmo dispositivo em questão, o *Behold II* da *Samsung*. Isso ocorre, porque todos os projetos usam a biblioteca *ZXing*[9], utilizava a chamada `set` para desativar o *flash* da câmera do dispositivo em questão de maneira forçada.

Os métodos `putBoolean` e `commit` também ocorrem da mesma forma como citada acima. Em análise, o cabeçalho da classe que utiliza os métodos contém a licença Apache[10] 2.0 do Android, logo o que nos leva entender que seja uma classe do próprio código do Android, além do mais, o comentário no fragmento de código *IF* que contém os métodos diz haver uma reclamação de usuários para os dispositivos *Motorola Cliq/-Dext*, *Huawei Pulse* e *U8230*, *Motorola Backflip*, *Motorola quench* e novamente o *Samsung Behold*. O comentário no código expressa um *bug* por um *driver* gráfico comum entre os dispositivos, forçando o aplicativo ativar modo de segurança, usando o método `putBoolean` para setar a configuração *SAFE_MODE* como ativa e `commit` para aplicar as configurações.

As regras referentes ao Grupo 1 da **Tabela 4** ocorrem pelo fato dos elementos que fazem parte da interface do usuário, denominados *View*[11], passaram a receber métodos para realizar animações a partir da API 11. A partir disso, houve a implementação da classe `android.animation.ObjectAnimator`[12] na qual permite realizar animações através dos métodos que fazem parte das regras. Antes disso, para realizar animação, os desenvolvedores utilizavam o método `setVisibility` para deixar o elemento visível ou invisível.

As regras do Grupo 2, ocorrem sempre em relação aos métodos `execute` e

[8]http://actionbarsherlock.com/

[9]https://github.com/zxing/zxing

[10]https://www.apache.org/licenses/LICENSE-2.0

[11]https://developer.android.com/reference/android/view/View.html

[12]https://developer.android.com/reference/android/animation/ObjectAnimator.html

`executeOnExecutor`. O segundo método citado foi implementado a partir da API 11 e a semelhança entre esses dois métodos é dita pelo fato de ambos executarem o início de uma tarefa em paralelo usando a classe `AsyncTask`. Porém a partir da API 11, o sistema Android limitou a classe `AsyncTask` para que apenas uma tarefa paralela seja executada por vez, para evitar erros em tarefas paralelas. Para reverter isso, os desenvolvedores passaram a utilizar o método `executeOnExecutor` com um parâmetro especial (`THREAD_POOL_EXECUTOR`).

No Grupo 3, as regras giram em torno das câmeras presentes no dispositivo. A classe na qual permite obter detalhes sobre as câmeras, `android.hardware.Camera.CameraInfo`, passaram estar presentes no sistema a partir da API 9, logo, faz sentido usar uma câmera se o aplicativo está executando em um sistema de versão 9 ou superior. Além do mais, é interessante ao desenvolvedor conhecer sobre qual direção as câmeras presentes no dispositivo estão, se é uma câmera frontal ou uma câmera que fica na parte traseira, para tal é usado o método `getCamera` e por fim, para abrir uma câmera é utilizado o método `open`.

Já os métodos `setBackground` e `setBackgroundDrawable`, presentes no Grupo 4 da **Tabela 4** são ou não executados a partir de uma verificação em relação a API 16, caso o aplicativo esteja em uma versão inferior, o correto será usar o método `setBackgroundDrawable`, o contrário, o correto é usar `setBackground`, pois este é um novo substituto ao método anterior.

As regras do Grupo 6 foram extraídas do *dataset* proveniente de *outras condições*. É possível observar que se trata de como o dispositivo lida especificamente com gravação de áudio.

4. Trabalhos Relacionados

Han e colegas [Han et al. 2012] estudaram se de fato a fragmentação do Android em relação aos dispositivos e às diferentes versões poderiam impactar de forma específica nas aberturas de problemas em *issue trackers*. Eles mostraram que de fato a fragmentação impacta em problemas específicos de fabricantes e que uma abordagem baseada em análise de tópicos pode ser útil no rastreamento deste tipo de problema. Este trabalho de certa forma, suporta nossa hipótese de que há problemas de fragmentação disponíveis em repositórios de software, em particular, endereçados por *patches* identificáveis no código fonte.

O trabalho que mais se relaciona ao nosso objetivo foi proposto por Wei e colegas [Wei et al. 2016], onde os autores estudaram problemas reais de compatibilidade coletados de aplicativos Android. Foram caracterizados os sintomas e as causas-raiz, e identificados padrões comuns para a solução dos problemas. Com isto, propuseram uma técnica denominada de FicFinder para identificar problemas de compatibilidade em aplicativos Android. Um dos problemas com esta abordagem é que as regras de identificação de problemas de compatibilidade foram definidas a partir de 191 problemas do conjunto de dados estudados. O objetivo em nosso trabalho foi permitir a obtenção de regras de uma maneira mais abrangente a partir de mineração de um grande volume de projetos que estejam disponíveis em repositórios de software.

Outro aspecto relacionado a este trabalho é o uso de regras de associação para mineração de co-ocorrências de chamadas de métodos em código-fonte relacionado

à fragmentação do Android. Mineração de regras de associação tem sido aplicada com sucesso na busca por significado de co-ocorrências de diversos tipos diferentes de informações em repositório. Talvez um dos trabalhos que mais influenciaram a comunidade foi o de Zimmermann e colegas [Zimmermann et al. 2004], cujo o objetivo era recomendar localizações de mudanças com base no seguinte raciocínio: "*Programadores que mudaram estas funções também mudaram ...*".

Dantas e Maia [Dantas and Maia 2016] estudaram a co-ocorrência de anomalias de código com o uso de regras de associação. Foi observado que determinados tipos de anomalias tendem a co-ocorrer mais por razões que podem ser explicadas.

5. Conclusões e Trabalhos Futuros

Um dos fatores sobre a aplicação prática direta dos resultados é que a base do *Boa* na qual oferece as cópias dos projetos do *GitHub*, somente possui informações até o ano de 2015 sendo assim um fator limitante, pois o Android recebe atualizações frequentes. Algo muito frequente que ocorreu ao decorrer da análise das regras, foi a indisponibilidade de repositórios no *GitHub*, uma vez que estar presente no *Boa* não garante mais a existência no *GitHub*. Sendo assim, uma alternativa de mineração diretamente em repositórios atuais se faz necessária.

Como resultado geral, observou-se que algumas regras estão ligadas principalmente a interface do usuário, em determinados níveis de API que tiveram mudanças mais significativas. As regras relacionadas a dispositivos são menos frequentes mas podem ser apresentadas principalmente para casos onde o aplicativo precisa dar suporte para dispositivos específicos. Tais regras podem ser usadas na recomendação de *patches* onde as chamadas de métodos envolvidos ocorrem.

Referências

Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In *20th Int. Conference on Very Large Data Bases (VLDB)*, pages 487–499.

Dantas, C. E. and Maia, M. d. A. (2016). Uma análise da associação de co-ocorrência de anomalias de código com métricas estruturais. In *Proc. of the IV Workshop on Software Visualization, Maintenance and Evolution - VEM'2016*, pages 1–8, Maringá, PR.

Han, D., Zhang, C., Fan, X., A. Hindle, K. W., , and Stroulia, E. (2012). Understanding Android fragmentation with topic analysis of vendor-specific bugs. In *Working Conference on Reverse Engineering - WCRE'2012*, page 83–92.

Wei, L., Liu, Y., and Cheung, S. C. (2016). Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proc. of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 226–237.

Zimmermann, T., Weisgerber, P., Diehl, S., and Zeller, A. (2004). Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE'04, pages 563–572, Washington, DC, USA.