2014 IEEE International Conference on Software Testing, Verification, and Validation

Geração Automatizada de Oráculos para Testes Recursos de interação do usuário de aplicativos móveis

Razieh Nokhbeh Zaeemÿ A
Universidade do Texas em Austin
nokhbeh@utexas.edu

Mukul R. Prasad
Fujitsu Laboratories of America
mukul@us.fujitsu.com

Sarfraz Khurshid
A Universidade do Texas em Austin
khurshid@ece.utexas.edu

Resumo—À medida que o uso de dispositivos móveis se torna cada vez mais onipresente, cresce cada vez mais a necessidade de testes sistemáticos de aplicativos (apps) executados nesses dispositivos. No entanto, testar aplicativos móveis é particularmente caro e tedioso, muitas vezes exigindo um esforço manual substancial. Embora os pesquisadores tenham feito muito progresso em testes automatizados de aplicativos móveis nos últimos anos, um problema-chave que permanece em grande parte não resolvido é o problema clássico do oráculo, ou seja, determinar a exatidão das execuções de teste. Este artigo apresenta uma nova abordagem para gerar automaticamente casos de teste, que incluem oráculos de teste, para aplicativos móveis. A base para nossa abordagem é um estudo abrangente que realizamos sobre defeitos reais em aplicativos móveis. Nossa principal chave, a partir deste estudo, é que existe uma classe de recursos que chamamos de recursos de interação do usuário, que está implicado em uma fração significativa de bugs e para os quais os oráculos podem ser construídos – de uma maneira agnóstica do aplicativo – com base em nosso entendimento comum de como os aplicativos se comportam. Apresentamos uma estrutura extensível que suporta esses oráculos de teste específicos de domínio, mas agnósticos de aplicação, e permite a geração de sequências de teste que aproveitam esses oráculos. Nossa ferramenta, QUANTUM, incorpora nossa abordagem para gerar casos de teste que incluem oráculos. Resultados experimentais usando 6 aplicativos Android mostram a eficácia do QUANTUM em encontrar bugs potencialmente graves, ao mesmo tempo em que gera conjuntos de testes compactos para recursos de interação do usuário.

I. INTRODUÇÃO

Os últimos anos testemunharam um crescimento explosivo no uso de dispositivos móveis e no número e variedade de aplicativos de software desenvolvidos para esses dispositivos. Aplicativos móveis, ou aplicativos como são chamados popularmente, geralmente são desenvolvidos em projetos pequenos e rápidos com recursos de teste escassos. Ao mesmo tempo, testar aplicativos móveis apresenta alguns desafios únicos, como o suporte a uma ampla gama de dispositivos, plataformas e versões, além de garantir a integridade da interface de usuário rica e altamente interativa característica de tais aplicativos [24]. Assim, há uma necessidade crescente de desenvolver ferramentas de teste automatizadas para apoiar o desenvolvimento de aplicativos móveis.

Pesquisadores fizeram progressos significativos no desenvolvimento de técnicas para suportar testes automatizados de aplicativos móveis [17], [2], [1], [10], [9]. No entanto, essas técnicas visam principalmente a geração de sequências de teste, deixando a tarefa de adicionar oráculos de teste [7], [23] a essas sequências de teste para o testador humano. Isso por si só pode ser um processo manualmente intensivo e se os oráculos não forem de qualidade suficientemente alta, pode comprometer potencialmente a eficácia dos casos de teste.

O objetivo deste artigo é abordar parcialmente o problema oracle no contexto da geração automatizada de casos de teste para aplicações móveis. Para concretizar este objetivo realizamos um estudo onde amostramos, estudamos e categorizamos os bugs relatados para vários aplicativos Android de código aberto populares. O estudo revelou que uma fração significativa de bugs pode ser atribuída a recursos de interação do usuário que são suportados pela plataforma móvel e simplesmente implementados por cada aplicativo. Esses recursos incluem apresentação de conteúdo ou recursos de navegação, como girar o dispositivo ou usar vários gestos para rolar ou ampliar as telas. Uma característica distintiva desses recursos é que eles são amplamente independentes da lógica central do aplicativo. Mais importante, muitas vezes há uma expectativa geral e de bom senso de como o aplicativo deve responder a um determinado recurso. Por exemplo, girar um dispositivo e depois girá-lo de volta deve trazer a tela precisamente de volta à tela inicial. Tais observações motivam nossa abordagem.

Apresentamos uma nova estrutura para criação de oráculos de teste para verificação de recursos de interação do usuário de aplicativos móveis, de maneira agnóstica de aplicativos. Nosso framework suporta geração de suítes de teste orientadas a modelo [21] onde cada teste gerado inclui tanto a sequência de teste a ser executada quanto as asserções correspondentes a serem verificadas (como oráculos de teste). Dado um modelo da interface do usuário do aplicativo móvel em teste, nossa estrutura usa sua biblioteca de oráculos extensível e integrada (para vários recursos de interação do usuário) e gera um conjunto de testes para testar de forma abrangente o aplicativo em relação aos recursos de interação do

Embora o objetivo básico de nossa estrutura seja permitir a geração de suítes de teste completas com oráculos de teste incorporados para recursos suportados, ela inclui mais duas técnicas para aprimorar ainda mais sua utilidade na prática. Em primeiro lugar, nossa estrutura suporta uma função de custo personalizável que define uma medida de custo para executar um determinado conjunto de testes e produz um conjunto de saída que provavelmente tem um custo de execução mínimo, enquanto verifica cada recurso. Em segundo lugar, nossa técnica de geração de teste insere vários oráculos de teste, para diferentes recursos, dentro de um único caso de teste, quando possível. Isso permite a verificação de várias propriedades na mesma execução de teste, compartilhando conceitualmente segmentos de execução comuns em diferentes testes, reduzindo assim o custo geral de execução do teste. Nossa técnica de geração de teste produz conjuntos de testes adequados aos recursos, que para o modelo fornecido exercitam todas as transições relevantes para cada recurso suportado e testam sua funcionalidade esperada.

Nossa ferramenta, QUANTUM, incorpora nossa estrutura e fornece um conjunto de ferramentas de botão totalmente automatizado para geração de casos de teste para aplicativos móveis. Nossos experimentos iniciais com o QUANTUM mostram que ele gera conjuntos de testes valiosos e fornece a base de uma abordagem promissora para testes mais eficazes de aplicativos móveis.

[y] Este autor foi estagiário no Fujitsu Labs of America para uma parte deste trabalho.

978-0-7695-5185-2/14 $ 31,00 © 2014 IEEE
DOI 10.1109/ICST.2014.31

Este artigo traz as seguintes contribuições:

Estudo de bugs. Realizamos um estudo abrangente de reais defeitos de aplicativos móveis e identificar uma família de recursos do aplicativo, que chamamos de recursos de interação do usuário. Observamos que essas características estão implicadas em um fração dos defeitos estudados. Além disso, são características da plataforma móvel e implementado por muitos aplicativos, mas não diretamente dependentes da lógica do aplicativo.

Testes baseados em recursos de aplicativos móveis. Apresentamos um nova forma de adequação do teste [6] no contexto de aplicativos móveis onde o objetivo é cobrir o modelo dado da interface do usuário do aplicativo exercitando cada transição relevante para qualquer recurso e verificando a funcionalidade esperada para o recurso.

Geração automática de oráculos para testar aplicativos móveis. Apresentamos uma biblioteca extensível de oráculos para vários recursos de interação do usuário. Nosso framework permite a autoria de oráculos de teste para recursos, de forma agnóstica de aplicação, para reutilização em vários aplicativos diferentes que devem oferecer suporte essas características. Esses oráculos são adequadamente instanciados por nossa técnica de geração de suíte de teste orientada por modelo.

Geração de suíte de teste de custo mínimo. Fornecemos uma técnica para gerar um conjunto de testes compacto, tentando minimizar uma função de custo personalizável. A suíte de testes também incorpora os oráculos para testar de forma abrangente o conjunto de recursos suportados.

Avaliação. Apresentamos nossa ferramenta, QUANTUM, para testes automatizados de aplicativos móveis e sua avaliação em 6 reais Aplicativos Android. A avaliação confirma que QUANTUM é capaz de gerar conjuntos de testes compactos, completos com testes oráculos, para testar as características identificadas. Esses conjuntos de testes são capazes de revelar uma série de bugs nas aplicações estudadas: A QUANTUM encontrou um total de 22 bugs, alguns deles particularmente sério, usando um total de 60 testes para esses 6 aplicativos.

II. ESTUDO DE ERRO

Conduzimos um estudo de bugs em 106 bugs retirados de 13 aplicativos Android de código aberto. O objetivo foi identificar oportunidades para geração automática de casos de teste, que incluem oráculos de teste, concentrando-se em bugs específicos para aplicativos móveis e explorando o conhecimento de domínio da plataforma móvel.

Os 13 aplicativos Android de código aberto que selecionamos incluíram 6 aplicativos estudados em trabalhos publicados anteriormente sobre testes automatizados para aplicativos móveis [8], [1], [17], mais 6 aplicativos selecionados do repositório de código aberto Google Code e o exemplo do Bloco de Notas aplicativo fornecido para fins educacionais pelo Android oficial website (também estudado em trabalho anterior [17]). A tabela I lista os nome, função declarada e fonte para cada um dos 13 assuntos.

Nosso objetivo era escolher sujeitos de teste de um conjunto diversificado de categorias e funções de aplicativos. Os 6 aplicativos CMIS, Deli cious, OpenSudoku, MonolithAndroid, Wordpress e Nexes Manager, escolhidos a partir de trabalhos publicados anteriormente, refletem essa intenção. Além disso, aplicamos os seguintes cinco adicionais critérios para escolher os 6 aplicativos de repositórios de código aberto: (1) popularidade: uma classificação mínima de 3,5 de 5 no Google Play, (2) elevado número de instalações ativas: mínimo de 50.000, (3) ter comunidades de desenvolvimento ativas: a versão mais recente da fonte do aplicativo deve ter sido baixado pelo menos 1000 vezes, (4) banco de dados rico de problemas relatados: pelo menos 25 problemas relatados e (5) defeitos reproduzíveis: o aplicativo deve tem pelo menos alguns defeitos reproduzíveis em um Android padrão emulador. Critérios semelhantes foram usados em estudos anteriores

TABELA I: Sujeitos para estudo de bugs

Aplicativo	Função	Fonte
Ferramenta de criação de notas do bloco de notas		developer.android.com/tools/samples
CMIS Navegador CMIS		gc/android-cmis-browser
Delicious	Social Bookmarking	gc/android-delicious-bookmarks
OpenSudoku	Sudoku Game	gc/opensudoku-android
MonolithAndroid	3D Game	gc/monolithandroid
Ferramenta de blog do Wordpress		android.trac.wordpress.org
Gerenciador de Nexes	Gerenciador de Arquivos	github.com/nexes/Android-File-Manager
VuDroidGenericName	Visualizador de PDF	gc/vudroid
Cronômetro de cozinha	Cronômetro	gc/timer de cozinha
Jogador de golfinhos	Reprodutor de mídia	gc/jogador de golfinhos
AnkiDroidGenericName	Revisão do Flashcard	gc/ankidroid
Embaralhar	Organizador pessoal	gc/android-shuffle
K9 Mail	Cliente de e-mail	gc/k9mail

gc: https://code.google.com/p

de aplicativos Android [8], embora para propósitos um pouco diferentes. Ao navegar manualmente no Google Code com a seleção acima protocolo selecionamos os 6 aplicativos VuDroid, Kitchen Timer, Dolphin Player, AnkiDroid, Shuffle e K9Mail que possuem em média, uma classificação 4,3 de 5.500.000 instalações ativas, 10.700 downloads e 1.400 problemas relatados.

Geralmente, apenas uma pequena fração dos problemas registrados no repositório de bugs de um aplicativo são bugs verdadeiros e reproduzíveis. Muitos deles não podem ser reproduzidos e outros ainda são meramente solicitações de recursos. Para selecionar bugs para investigação adicional, por cada sujeito de teste examinamos manualmente cada problema registrado em seu repositório até termos 10 bugs reproduzíveis (exceto para Delicious, MonolithAndroid e Nexes, selecionados de trabalhos anteriores, que possuem pequenos repositórios de bugs onde poderíamos encontrar apenas 8, 6 e 2 defeitos reprodutíveis, respectivamente). Sem bug existem relatórios para o Bloco de Notas. Isso nos deu um total de 106 bugs.

Nós investigamos e categorizamos manualmente cada um dos 106 bugs, do ponto de vista dos oráculos de teste necessários para detectar eles. Identificamos 20 categorias além do aplicativo principal lógica. A Tabela II mostra essa categorização. Nós observamos que quase 75% dos bugs estudados não estão diretamente ligados a a lógica do aplicativo (apenas 27 bugs são categorizados em Application Logic), que nos inspirou a explorar caminhos para gerando automaticamente oráculos de teste adaptados para aplicativos móveis.

Agregamos ainda mais as categorias com base na automatização dos oráculos subjacentes. Oráculos que impõem o lógica de aplicação são muito específicas de aplicação e notoriamente difícil de gerar de forma totalmente automática. Além desta categoria, identificamos 3 grupos de oráculos. O primeiro grupo, Básico Oráculos, abrangem instâncias gerais de programas aberrantes comportamento como travamentos, travamentos ou rescisões ilegais que não são específicos do aplicativo, ou mesmo específicos para aplicativos móveis. Por exemplo, Exceções não capturadas pertencem a este grupo. O CMIS tem um bug em que “se o campo URL for deixado vazio, um null exceção de ponteiro é lançada”. Esses oráculos básicos já são amplamente utilizado em testes automatizados de software e, portanto, não particularmente interessante para a presente investigação. Outro grupo, denominado Oráculos Específicos de Aplicativos, não estão diretamente relacionados a a lógica do aplicativo, mas ainda pode ser muito específico do aplicativo. Por exemplo, oráculos para validar a Aparência Visual de um app pertencem a este grupo. O aplicativo MonolithAndroid tem um defeito em que “existem buracos na textura de fundo” de a renderização do aplicativo. Achamos que seria muito difícil gerar oráculos automatizados precisos para distinguir entre comportamento defeituoso nesses casos. Portanto, nosso trabalho não segmentar esta categoria também. No entanto, o grupo de App Agnostic

TABELA II: Categorização de Bugs.

Grupoÿ	**Oráculos Básicos**									**Oráculos agnósticos de aplicativos**				**Oráculos específicos do aplicativo**							
Categoria ÿ																					
Bloco de anotações																					
CMIS			4				1	1											2	2	
Delicioso			1													111	4				
OpenSudokuGenericName		1									1			1	2			3			2
MonolithAndroid				111							1				1		1				
WordPress			3						1		1		1						1	1	2
Gerenciador de Nexes																					2
VuDroidGenericName	1	31								2							1				1
Cronômetro de cozinha			1							2	1	1	1		2						3
Jogador de golfinhos	1				1	1	1										2	1			3
AnkiDroidGenericName			2					1		1		1			1		2				2
Embaralhar			1				1		1				1						1	2	3
K9 Mail		1										1		311							3
Total	**2**	**2**	**15**	**2**	**2**	**2**	**3**	**2**	**2**	**5**	**4**	**4**	**2**	**4**	**7**	**2**	**7**	**5**	**2**	**5**	**27**

Oráculos contém bugs para os quais os oráculos são significativamente mais complicado do que os oráculos básicos, mas suficientemente agnóstico que eles poderiam potencialmente ser gerados automaticamente. Encontramos categorias relativamente bem preenchidas, como Rotação, Ciclo de vida da atividade e erros de gestos neste grupo.

Os bugs de rotação se manifestam à medida que o dispositivo móvel é girado da orientação paisagem para retrato ou vice-versa. Há um entendimento comum de como os aplicativos geralmente respondem para rotação: o mesmo conteúdo deve ficar na tela, em um arranjo possivelmente diferente, e deve suportar o mesmo ações como antes. Além disso, as entradas de dados do usuário devem ser preservada após a rotação. VuDroid contém um exemplo de um erro de rotação em que “a seleção de guias é redefinida após a rotação o telefone”. Nosso estudo de bugs encontrou cinco bugs de rotação em três aplicativos. Erros de gestos formam outra categoria, semelhante à rotação bugs, onde gestos comuns, como aumentar e diminuir o zoom, e rolagem, produzem uma resposta que contradiz o senso comum expectativa. O estudo encontrou quatro erros de gestos em quatro aplicativos.

A interface gráfica do usuário de aplicativos Android é composta de componentes chamados Atividades, cada um correspondendo a um núcleo função do aplicativo. O comportamento de uma atividade deve estar de acordo com um ciclo de vida de atividade, uma máquina de estados finitos onde cada estado representa um estado de nível grosseiro (como ativo ou pausado)1. Os bugs do ciclo de vida da atividade correspondem ao comportamento aberrante exibido à medida que os componentes da atividade do aplicativo passam diferentes estados do ciclo de vida. Isso acontece, por exemplo, como um aplicativo é enviado para segundo plano, eliminado, retomado ou reiniciado. Semelhante para bugs de rotação, há uma compreensão do senso comum como os aplicativos devem se comportar quando são pausados ou eliminados. Por Por exemplo, quando um aplicativo é pausado e posteriormente retomado, ele deve preservar as entradas de dados do usuário na tela atual. Nosso estudo encontrou quatro bugs do ciclo de vida da atividade em quatro aplicativos. Por exemplo, o Wordpress tem um bug em que “o conteúdo desaparece se o aplicativo for para segundo plano”. Observe que erros de rotação e gesto podem compartilhar causas raiz com a atividade bugs do ciclo de vida. No entanto, essas categorias de bugs não são causalmente vinculado. Além disso, eles representam diferentes bugs do final perspectiva do usuário e, portanto, merecem ser testados independentemente.

1http://developer.android.com/guide/components/activities.html#Lifecycle.

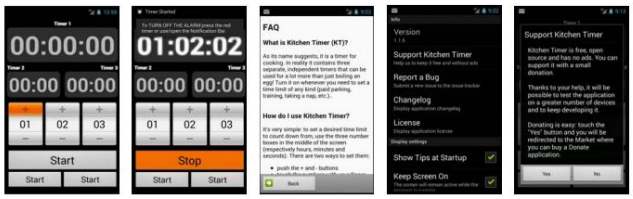
Finalmente, os erros de fuso horário ocorrem ao lidar com horários diferentes zonas. O estudo encontrou dois bugs de fuso horário em dois aplicativos. No entanto, embora seja agnóstico do aplicativo, esses bugs não surgem diretamente das interações do usuário e, portanto, são diferentes dos outros três categorias deste grupo.

O estudo de bugs revelou que muitos bugs em aplicativos móveis raízes compartilhadas além da lógica do aplicativo. Ainda nos ajudou identificar três categorias de oráculos para encontrar tais bugs: (1) o oráculos básicos, (2) os oráculos específicos do aplicativo e (3) o aplicativo oráculos agnósticos. Os oráculos agnósticos de aplicativos principalmente (com o exceção de bugs de fuso horário) correspondem a formas de interação com dispositivos móveis, comum não apenas entre aplicativos diferentes mas também entre várias plataformas móveis. O estudo descobriu que esses recursos de interação do usuário de aplicativos móveis são a causa de bugs em mais da metade dos aplicativos estudados. Inspirado por estes bugs, apresentamos nossas técnicas de teste e geração de oráculos para recursos de interação do usuário de aplicativos móveis.

III. EXEMPLO

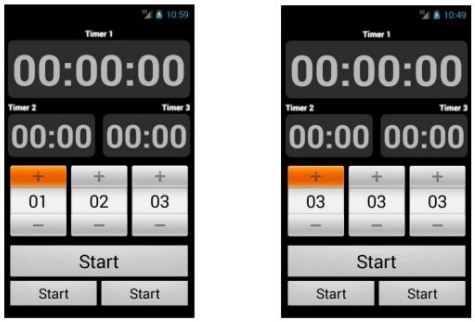
Nesta seção, apresentamos um aplicativo de exemplo para motivar nossos técnica para testar automaticamente os recursos de interação do usuário de aplicativos móveis. Para ilustração, usamos um bug real da Kitchen Timer, um aplicativo Android de código aberto. Como o nome sugere, Kitchen Timer é um temporizador para cozinhar. Contém três temporizadores que pode ser definido independentemente e disparar soando um alarme após contagem regressiva até zero. Além disso, o temporizador de cozinha fornece outras funcionalidades para alterar as preferências (por exemplo, o som do alarme, a cor do LED, nomes dos temporizadores), salve os temporizadores predefinidos, e muitos mais.

Enquanto usamos o temporizador de cozinha completo para o bug estudo e avaliação (Seções II e V), descrevemos um versão simplificada para este exemplo. A Figura 1 mostra instantâneos do temporizador de cozinha. Neste temporizador de cozinha simplificado, o usuário pode definir um temporizador usando os sinais de mais e menos na tela principal (Figura 1a). Os números da esquerda para a direita mostrar as horas, minutos e segundos para o temporizador a ser definido. Em seguida, o usuário pode iniciar o cronômetro pressionando o botão Iniciar (para chegar à Figura 1b) ou pará-lo pressionando o botão Parar. Além disso, ele pode selecionar Informações, Preferências ou Doação de no menu (para chegar às Figuras 1c, 1d e 1e). Todas essas telas rotação de suporte. Para simplificar, excluímos outras ações



(a) Temporizadores principais (b) Temporizador em execução (c) Informações (d) Preferências (e) Doação

Fig. 1: Instantâneos do temporizador de cozinha simplificado.



(a) O usuário define um temporizador. (b) Valores substituídos após duas rotações.

Fig. 2: Instantâneos de um Bug no Timer de Cozinha.

de Informações, Preferências e Doação do aplicativo simplificado.
O usuário pode, no entanto, voltar para a tela principal a partir deles.

Como mostra a Figura 2, o Kitchen Timer tem um bug que se manifesta quando o dispositivo é girado duas vezes. Se o usuário definir um cronômetro (Figura 2a) e, em seguida, girar o dispositivo móvel duas vezes antes de iniciar o cronômetro (Figura 2b), o valor de segundos substituirá os valores de minutos e horas, alterando o cronômetro de 1 hora, 2 minutos, e 3 segundos a 3 horas, 3 minutos e 3 segundos. Nossa ferramenta encontra esse bug automaticamente.

4. TESTE BASEADO EM RECURSOS DE APLICATIVOS MÓVEIS

As descobertas do nosso estudo de bugs nos motivaram a desenvolver uma abordagem para testar automaticamente os recursos de interação do usuário (recursos de interação, ou simplesmente recursos ao longo do artigo) de aplicativos móveis. Nossa abordagem proposta é descrita nesta seção. Começamos definindo algumas terminologias.

Definição 1 (recurso de interação). Um recurso de interação é uma ação suportada pela plataforma móvel, que permite que um usuário humano interaja com um aplicativo móvel, usando o dispositivo móvel e a interface gráfica do usuário (GUI) do aplicativo.
Além disso, um recurso de interação está associado a uma expectativa de senso comum de como o aplicativo móvel deve responder a essa ação.

Os recursos de interação incluem ações como girar um dispositivo móvel, gestos de uso geral, como aumentar/diminuir o zoom ou rolar, e ações que iniciam, pausam, matam ou retomam a operação de um aplicativo, levando seus componentes da GUI de atividade por vários estados em seu ciclo de vida . Esses recursos foram discutidos em nosso estudo de bugs. Além disso, recursos como os botões Voltar ou Para cima da plataforma Android2 também são válidos recursos de interação. Observe que a definição acima exclui um número de gestos comuns, como clique ou longClick, ou outros gestos personalizados, para os quais não há uma resposta padrão esperada dos aplicativos; é completamente específico do contexto e da aplicação. Como um determinado recurso de interação terá, em geral, um comportamento padrão esperado, entre aplicativos e diferentes plataformas móveis3, isso fornece um oráculo geral e independente do aplicativo para validar a resposta de um aplicativo ao exercício desse recurso. Assim, um componente-chave de nossa abordagem é criar esses oráculos reutilizáveis e empregá-los no teste de recursos de interação.

Seguimos uma abordagem orientada por modelo para gerar casos de teste para testar os recursos de interação de um determinado aplicativo móvel. O ponto de partida para nossa técnica é um modelo de estado finito do comportamento da GUI do aplicativo, que é definido a seguir.

Definição 2 (modelo GUI). Um modelo de GUI de um aplicativo é uma máquina de estados finitos M, denotada pela 4-tupla M = (S, s0, A, R), onde S é um conjunto finito de estados abstratos representando diferentes telas de GUI, s0 ÿ S é o estado inicial denotando a tela de abertura do aplicativo, A é um conjunto finito de ações específicas do aplicativo que o usuário pode executar na execução da lógica do aplicativo e R ÿ S × A × S é uma relação de transição que descreve transições entre estados em S em resposta a ações do usuário de A.

Duas telas GUI são representadas pelo mesmo estado abstrato em M se e somente se contiverem o mesmo conjunto de ações nos mesmos widgets. A única exceção a isso são as telas que mostram coleções de itens, como livros, arquivos, músicas, transações, etc., onde cada item suporta algum conjunto de ações. Neste caso, duas telas com números diferentes (diferentes de zero) de itens são interpretadas como o mesmo estado. Assim, o conteúdo de uma coleção é abstraído como vazio ou não vazio. Noções semelhantes de estados de GUI também foram usadas em trabalhos anteriores [11], [9]. O conjunto A inclui ações específicas do aplicativo, como cliques ou cliques longos, etc., em widgets específicos, mas não inclui recursos de interação suportados pela plataforma (por exemplo, rotação do dispositivo etc.). Acreditamos que isso também seja típico de modelos de GUI [25].

Observe que, embora a parte visível de uma tela GUI de um aplicativo móvel, vista em um dispositivo móvel, possa mudar executando uma ação como uma rotação do dispositivo, um zoom ou uma ação de rolagem, essas telas aparentemente diferentes ainda correspondem a o mesmo estado abstrato no modelo GUI. Definimos a noção de uma visão, denotada pelos símbolos w, para representar a porção visível dos estados abstratos do modelo GUI s. Assim, um estado s pode ter várias visualizações, geradas pelo exercício de diferentes recursos de interação disponíveis em s. Especificamente, usamos a notação ÿ(s, u, ÿ) e ÿ(s, u, +) para denotar, respectivamente, as duas visões diferentes do estado s antes e depois da ação (ou sequência de ações) u ser disparada, onde u corresponde a uma instância de exercício de um recurso de interação. A notação de visualização fornece uma noção relativa de tempo dos estados de amostragem (para sua visualização atual), antes e depois de exercitar os recursos de interação.

Um modelo GUI M(S, s0, A, R) também pode ser representado como um grafo direcionado rotulado e enraizado G = V, E, r, A,L , de maneira direta. Aqui, nós V representam os estados S, o nó raiz r representa o estado inicial s0, as arestas E representam transições entre estados, consistentes com a relação de transição R, e a função de rotulagem L: E ÿ A rotula cada aresta com a ação a ÿ A responsável pela transição. Modelos de GUI podem ser construídos manualmente ou gerados automaticamente usando uma das técnicas de um crescente corpo de trabalho na geração de modelos de GUI para aplicativos móveis [1], [10], [25].

2Consulte http://developer.android.com/design/patterns/navigation.html.

3Aplicativos específicos podem, obviamente, optar por modificar essa resposta padrão.

Abordagem geral: nossa técnica gera conjuntos de testes compactos, completos com oráculos de teste, para testar de forma abrangente os recursos de interação de um determinado aplicativo móvel. A abordagem usa uma biblioteca extensível F de definições de recursos agnósticos e reutilizáveis, descritos na Seção IV-A. Dado um modelo de GUI do aplicativo fornecido pelo usuário, aumentamos automaticamente esse modelo com instâncias de recursos, usando as definições de recursos em F (Seção IV-B). Então, com base nos critérios de custo e adequação de teste definidos na Seção IV-C, automaticamente percorremos o modelo aumentado para criar sequências de teste compactas (Seção IV-D). Por fim, instanciamos automaticamente os oráculos de teste nas sequências de teste para obter um conjunto de testes completo.

A. Criação de Oracles para Teste de Recurso de Interação

Apresentamos uma estrutura extensível na qual os recursos de interação podem ser definidos de maneira agnóstica do aplicativo e armazenados em uma biblioteca. Ao testar um determinado aplicativo, nossa técnica instancia adequadamente os recursos da biblioteca, usando essas definições de recursos, e gera testes, completos com oráculos de teste, para testar cada recurso de forma abrangente.

Definição 3 (definição de recurso). A definição de traço de um dado traço de interação f é um triplo: uf , Df (s), Of (w1, w2) . uf = u1, u2,...,un é uma sequência de ações que exercita o recurso. Df (s) é a função de destino, que mapeia um dado estado s no qual o recurso pode ser exercido para um conjunto de estados Sf ÿ S que poderia resultar do exercício de f no estado s. Of (w1, w2) é o oráculo para o recurso f, onde w1 = ÿ(s1, uf , ÿ) é uma visão de algum estado s1 antes de disparar ações uf e w2 = ÿ(s2, uf , +) é uma visão de um estado s2 alcançado após disparar ações uf em algum estado anterior s1, possivelmente o mesmo estado s2.

Um aspecto crucial da definição de recurso acima é expressar uf , Df e Of de uma maneira agnóstica de aplicativo. Demonstramos como fazer isso abaixo, por meio de definições de recursos de exemplo de vários recursos de interação comuns. Outra restrição importante implícita na Definição 3 é que o conjunto de estados abstratos no modelo GUI deve ser fechado sob aplicação do recurso de interação, ou seja, exercitar o recurso em um dos estados não deve levar a aplicação a um estado abstrato fundamentalmente novo fora o modelo GUI. Essa restrição de bom senso também é válida para todos os recursos de interação em nosso conhecimento.

Beira. Nos exemplos a seguir, usamos ÿÿ(s) e ÿ+(s) como abreviação para ÿ(s, uf , ÿ) e ÿ(s, uf , +) respectivamente, já que uf é claro a partir do contexto.

Rotação dupla (DR): incorporamos o recurso de rotação do dispositivo móvel em uma definição do recurso de rotação dupla, que expressa o ato de girar um dispositivo móvel e depois girá-lo de volta à orientação original. Com esta ação, o aplicativo deve permanecer no mesmo estado. Além disso, a visão desse estado antes de uma rotação dupla após deve ser idêntica.

Isso é expresso na definição de recurso: DR = uf = , Df (s) = {s}, Of = (ÿÿ(s)=ÿ+(s)) . girar, girar **Eliminar e reiniciar (KR):** O sistema operacional pode optar por eliminar e reiniciar um aplicativo por vários motivos (por exemplo, pouca memória). Semelhante à rotação dupla, o aplicativo deve recuperar seu estado e visualização originais. Assim, KR = uf = , Df (s) = {s}, Of = (ÿÿ(s)=ÿ+(s)) . matar, reiniciar **Pausar e retomar (PR):** O aplicativo pode ser pausado (por exemplo, pressionando o botão Home do Android) e depois retomado. PR = , Df (s) = {s}, Of = (ÿÿ(s) = uf = pausar, retomar ÿ+(s)) . Matar e depois reiniciar, pausar e depois retomar são ambas as instâncias de transições de ciclo de vida de atividade que todos os aplicativos devem oferecer suporte.

Funcionalidade do botão Voltar (Voltar): O botão Voltar é um botão de hardware em dispositivos Android que leva o aplicativo , Df tela anterior. Back = uf = back {sp : sp ÿ pai(s)}, Of = (ÿÿ(s1)=ÿ+(s)) para a (s) = onde s1 ÿ Df (s). Neste caso, a função de destino produz um conjunto de destinos D(s) correspondentes a cada um dos nós pais (usando a noção teórica de grafos padrão de pai e filho) do estado atual s no modelo GUI.

Abrindo e fechando menus (Menu): O botão Menu de hardware em dispositivos Android abre e fecha menus personalizados que cada aplicativo define. Para esta definição de recurso Menu = uf = , Df (s) = cardápio, cardápio {s}, Of = (ÿÿ(s)=ÿ+(s)) .

Nos casos acima, o oráculo sempre foi uma afirmação de igualdade entre duas visões de estado apropriadas. Em geral, no entanto, o predicado oracle pode incluir operadores relacionais ou lógicos arbitrários. Por exemplo:

Zoom in (ZI): O zoom em uma tela deve trazer um subconjunto do que estava originalmente na tela. ZI = uf = , Df (s) = {s}, Of = (ÿÿ(s) ÿ ÿ+(s)) . mais Zoom

Diminuir o zoom (ZO): Diminuir o zoom de uma tela deve resultar em um superconjunto da tela original. ZO = uf = Reduzir o zoom , Df (s) = {s}, Of = (ÿÿ(s) ÿ ÿ+(s)) .

Rolagem (SCR): A rolagem para baixo (ou para cima) deve exibir uma tela que compartilha partes da tela anterior. SCR = uf = , Df (s) = rolar para baixo {s}, Of = (ÿÿ(s) ÿ ÿ+(s) = ÿ) .

Observe que a própria definição do recurso inclui uma implementação do oráculo, embora independente do aplicativo, que pode ser reutilizada em diferentes aplicativos. Assim, a semântica dos operadores utilizados nos oráculos é definida ali.

B. Aprimorando modelos de GUI com instanciações de recursos

Dado um modelo de GUI G = V, E, r, A,L do aplicativo de destino e uma biblioteca F de recursos de interação, especificados conforme discutido na Seção IV-A, o próximo passo em nossa abordagem é anotar G com todas as instanciações possíveis de cada recurso em F para produzir um modelo de GUI aumentado G+ = V,E+, r, A+,L+ . Especificamente, isso envolve adicionar um conjunto de arestas especiais rotuladas, chamadas arestas douradas, a G. Cada aresta dourada, ef (v1, v2) denota que o recurso f quando exercido no estado do vértice v1 leva a aplicação para o estado do vértice v2 . Além disso, ef é rotulado com uf , a sequência de ação do recurso f. Assim, o modelo aumentado G+ inclui o conjunto aumentado de arestas E+ = EÿEgolden, conjunto de ações aumentadas A+ = Aÿ fÿF uf , e função de rotulagem apropriadamente modificada L+ : A+ ÿ E+, onde Egolden são as arestas douradas e fÿ F uf são as ações para os recursos F que rotulam as bordas douradas.

O Algoritmo 1 mostra o procedimento para realizar o aumento do modelo GUI. O algoritmo itera sobre cada estado v no modelo GUI (linhas 3 ÿ 13) e cada recurso f na biblioteca F, instanciando f em v, conforme a definição do recurso. Ele calcula o conjunto de possíveis vértices de destino dSet, avaliando a função Df na definição do recurso (função destinationSet() na linha 5). Ele então itera sobre cada vértice de destino possível v1 (linhas 6ÿ11) criando e adicionando uma borda dourada, rotulada pela sequência de ação do recurso uf (linha 8), ao modelo aumentado G+. A Figura 3 mostra o modelo GUI de nossa versão simplificada do Kitchen Timer da Seção III, aumentada com bordas douradas para os recursos do botão Double Rotation e Back.

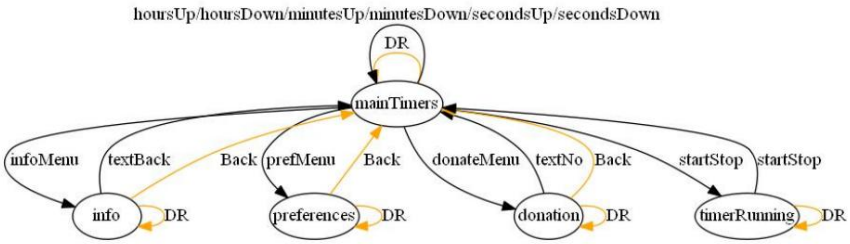


Fig. 3: Modelo Simplificado de Temporizador de Cozinha.

Algoritmo 1: Algoritmo de Aumento do Modelo GUI

Entrada : G = V, E, r, A, L: modelo GUI original do aplicativo de destino
F: Biblioteca de recursos
Saída: G+ = V,E+, r, A+, L+ : **início** do modelo **1 de** GUI aumentada

```
2  G+ ÿ G foreach
3      v ÿ V do
           // Iterar sobre cada vértice (estado) de G foreach f ÿ F do
4
               // Iterar sobre cada recurso em F dSet =
5              destinationSet(v, G+, f) foreach v1 ÿ dSet do e ÿ
6              createEdge(v, v1); setEdgeLabel(e, getAction(f))
7                  markGolden(e) addEdge(e, G+)
8
9
10
11             fim
12         fim
13     fim
14     de retorno G+
15 final
```

C. Definição do Conjunto de

Testes Dado um modelo de GUI aumentada G+ , a fase final do nosso técnica gera um conjunto de testes com o seguinte objetivo.

Objetivo do teste: um conjunto **compacto** de testes para testar de **forma abrangente** os recursos de interação do aplicativo em teste.

Um teste é uma sequência de ações (ou seja, uma sequência de arestas ou um caminho em G+) começando no estado inicial r, com cada ação possivelmente seguida por uma verificação oracular. Portanto, um teste Cada ai ÿ A+ is pode a1, o1,...,an, on . qualquer ação (incluindo aquelas que exercem recursos de ser representado como interação) permitida pela GUI do aplicativo. Cada oi é uma verificação de oráculo ou nenhuma operação. Assumimos que as verificações oracle são livres de efeitos colaterais, ou seja, não alteram o estado do aplicativo.

Definimos um critério de adequação de teste para concretizar a noção de um conjunto de testes “abrangente” declarado no objetivo do teste.
Como os aplicativos móveis são sistemas orientados a eventos e os recursos de interação são elementos da GUI do aplicativo, desenvolvemos um critério que é motivado pelas noções de cobertura de caminho e cobertura de fluxo de eventos usadas em trabalhos anteriores sobre testes de GUI [14].
Isso contrasta com os critérios baseados em cobertura de código, como cobertura de linha ou ramal, que seriam mais apropriados para testes funcionais da implementação do software, em vez de testar seus recursos de plataforma de alto nível, como no nosso caso.
Intuitivamente, dizemos que um conjunto de testes cobre um recurso, se ele contém testes para exercitar e validar cada instância possível de exercício desse recurso naquele aplicativo. Simplificando, isso implica exercitar o recurso em cada estado da GUI. Dado um conjunto de testes T, um recurso de interação f de uma biblioteca de recursos F e um modelo de GUI aumentado G+ = V,E+, r, A+,L+ , conforme definido na Seção IV-B, definimos a adequação de T no teste de f em relação a G+ como segue.

Definição 4 (Cobertura do recurso de interação). Um conjunto de testes T cobre uma característica f iff ÿs ÿ S : ÿt ÿ T,t = , ÿj, k, 0 ÿ j<k ÿ n tal que a1,...,aj leva o a1, o1,...,an, em aplicativo do estado inicial s0 para declarar e ok = Of . s, aj+1,...,ak = uf , Como não há funções de custo padrão ou amplamente quantificamos a “compacidade” de testes para priorizar testes usados em observação de senso comum de que conjuntos de testes grandes são difíceis para configurar, executar e manter. O tamanho de um conjunto de testes pode ser medido pelo número de testes que ele contém, bem como pelo número acumulado de operações (ações ai) no conjunto de testes como um todo. Propomos uma função de custo personalizável que captura isso.

Definição 5 (Custo de um conjunto de testes). O custo de um conjunto de testes T é custo(T) = ÿ ÿ |T| + ÿ ÿ ÿtÿT |t|, onde ÿ e ÿ são coeficientes positivos.

O coeficiente ÿ mede o custo relativo de desenvolvimento e manutenção de uma suíte, que escala com o número de testes em uma suíte. O coeficiente ÿ quantifica o custo de execução de ações e afirmação de oráculos que é proporcional ao número de operações.

D. Geração de sequência de teste com base em recursos

Lembre-se de que os recursos de interação são ortogonais à lógica central do aplicativo e sua função geralmente é ajudar o usuário a navegar ou acessar o conteúdo no aplicativo alterando o estado da GUI do aplicativo. Além disso, exercitar um recurso de interação em um determinado estado não tem efeitos colaterais em termos do modelo GUI, ou seja, o efeito de exercitar esse recurso é limitado a uma única tela GUI e não tem impacto nas ações downstream. Essa observação é muito importante, pois nos permite misturar e combinar arbitrariamente instâncias de vários recursos (e seus oráculos de teste) em um único caso de teste, desde que reduza o custo do conjunto de testes, conforme a Definição 5. Como cada instância de cada recurso já está registrado em nosso modelo de GUI aumentada G+ (atendendo ao critério de adequação de teste da Definição 4), nosso problema de geração de teste pode ser declarado da seguinte maneira.

Problema de geração de suíte de teste: Dado um modelo de GUI aumentada G+ , gere uma suíte de teste de custo mínimo, de modo que cada borda dourada em G+ seja coberta por pelo menos um teste na suíte.

Pode-se mostrar que o problema acima é NP-difícil, reduzindo o problema de cobertura mínima de caminho [20] a este problema.
Omitimos a prova detalhada aqui por falta de espaço.

Propomos um algoritmo guloso para este problema NP-difícil.
Além disso, introduzimos duas otimizações para reduzir ainda mais

Algoritmo 2: Algoritmo Traversal

Entrada : G+ = V,E+, r, A+, L+ : Modelo GUI aumentado do aplicativo
Saída: T: Conjunto de Testes

```
1  começo
2      CE ÿ ÿ
3      pilha ÿ ÿ
4      L ÿ sortWithBFS(G+)
5      foreach s ÿ L fazer
6          enquanto ÿ(s, y) ÿ saída(s), st(s, y) ÿ E ÿ CE do
7              foreach e ÿ mais curtoP athBF S(r, s) do
8                  pilha.push(e)
9                  CE ÿ CE ÿ {e}
10             fim
11             c ÿ s
12             parar ÿ falso
13             enquanto ¬para fazer
14                 se ÿ(c, v) ÿ Saída(c), st(c, v) ÿ E ÿ CE
                   então
15                     CE ÿ CE ÿ {(c, v)}
16                     pilha.push((c, v))
17                     c ÿ v
18                 fim
19                 senão pare ÿ verdadeiro
20
21             fim
22             T ÿ T ÿ pilha
23             pilha.limpar()
24         fim
25     fim
26     retornar T
27 final
```

o custo associado aos recursos de cobertura. Algoritmo 2 mostra um pseudo-código do algoritmo de travessia que propomos. A entrada para este algoritmo é o modelo de grafo aumentado. Usamos um conjunto para acompanhar as bordas cobertas CE e uma pilha para registrar o sequência de teste. Primeiro, na Linha 4, ordenamos os nós com base em seus aumentando a distância da raiz usando uma busca em largura (BFS) e manter a lista ordenada em L. Por exemplo, podemos ordenar os nós da Figura 3 como mainTimers, informações, preferências, doação, timerRunning . Então, trabalhando na lista L em Linha 5, selecionamos o próximo nó s que descobriu a saída bordas (Linha 6). Em nosso exemplo, o primeiro nó da lista com arestas de saída descobertas são mainTimers (já que ainda não cobriu quaisquer arestas). Usamos o caminho mais curto da raiz para este nó (salvo por meio de BFS executado anteriormente) como o prefixo de todas as sequências a serem geradas a partir dele. Linhas 7 para 10 iterar pelo caminho mais curto e (1) marcar as arestas como visitados adicionando-os ao CE e (2) empurrando-os para a pilha. A lógica por trás do uso de tal prefixo é minimizar o custo associado a tomar arestas para chegar a um determinado nó, onde o começa a exploração de bordas douradas descobertas.

Então, usando c como um ponteiro para o nó atual, que é inicialmente definido como s, na Linha 14, escolhemos uma aresta descoberta indo fora de c. Pegamos essa borda, marcamos como coberta (Linha 15), empurre-o para a pilha (Linha 16) e atualize c para o destino desta borda de acordo (Linha 17). Quando chegamos a um nó que não tem borda de saída descoberta, a sequência de teste atual é completo e definimos stop como True na Linha 19. O atual pilha faz uma sequência de teste e continuamos gerando mais sequências e adicionando-as a T que é o conjunto de testes e é a saída deste algoritmo. Por exemplo, o primeiro sequência de teste que é gerada é mostrada como T0 em Não Otimização na Tabela III. Esta tabela exibe o conjunto de testes nosso algoritmo guloso gera para o modelo simplificado de

TABELA III: Sequências de teste para a Figura 3.

Sem otimização	
T0 = horasPara cima, horas para baixo, minutos para cima, minutos para baixo, segundos para cima, segundos para baixo, infoMenu, textBack, prefMenu, Back, donateMenu, textNo, startStop, startStop, DR	
T1 = infoMenu, Voltar	T2 = infoMenu, DR
T3 = prefMenu, DR	T4 = doarMenu, Voltar
T5 = doarMenu, DR	T6 = startStop, DR
#Tests = 7, Custo(T) = 34	
Otimização de priorização ativada	
T0 = DR, horasPara cima, horasPara baixo, minutosPara cima, minutosPara baixo, segundosPara cima, segundosPara baixo, infoMenu, Voltar, prefMenu, Voltar, donateMenu, Voltar, startStop, DR, startStop	
T1 = infoMenu, DR, textoVoltar	T2 = prefMenu, DR
T3 = doarMenu, DR, textNo	
#Tests = 4, Custo(T) = 28	
Otimizações de priorização e truncamento ativadas	
T0 = DR, horasPara cima, horasPara baixo, minutosPara cima, minutosPara baixo, segundosPara cima, segundosPara baixo, infoMenu, Voltar, prefMenu, Voltar, donateMenu, Voltar, startStop, DR	
T1 = infoMenu, DR	T2 = prefMenu, DR
T3 = doarMenu, DR	
#Tests = 4, Custo(T) = 25	

Cronômetro de cozinha. A suíte de testes possui 7 testes a um custo total de 34, com ÿ e ÿ ambos definidos como 1 na função de custo.

Introduzimos duas otimizações para aumentar nossa base algoritmo de travessia. A primeira otimização chamada priorização prioriza as bordas douradas sempre que houver e bordas descobertas regulares (não douradas) saindo de um nó, uma vez que o objetivo do algoritmo de travessia é cobrir bordas douradas. Para implementar essa otimização, o método outGoing() no Algoritmo 2 retorna as bordas douradas primeiro. A Tabela III exibe a saída do algoritmo de travessia com este otimização incorporada. Por exemplo, no início de T0 em Otimização de Priorização Ligado, quando o Golden a borda DR está disponível, ela é tomada antes de qualquer outra borda. Isto otimização torna o conjunto de testes menor e diminui o custo de 34 a 28.

A segunda otimização, chamada de truncamento, usa a observação de que um teste pode ser truncado após a última borda dourada ele cobre e excluído se não cobrir bordas douradas. Truncamento é aplicável em uma fase de pós-processamento em qualquer suíte de teste. A Tabela III mostra o resultado da combinação das duas otimizações (aplicando truncamento no resultado da otimização de priorização) o que faz com que o custo do conjunto de testes caia para 25.

Uma vez que as sequências de teste são geradas, inserimos oráculos por aumentando as sequências de teste de duas maneiras. Em primeiro lugar, adicionamos automaticamente instrumentação apropriada antes e depois ações relevantes em sequências de teste, para registrar dinamicamente o visualização atual de cada estado da GUI, à medida que o teste está sendo executado. Em segundo lugar, instanciamos automaticamente oráculos Of do definições de recursos para fazer verificações nas visualizações de estado registradas pela instrumentação.

E. Implementação

A ferramenta QUANTUM incorpora nossa abordagem. QUANTUM atualmente suporta o teste dos seguintes recursos4: rotação, matando e reiniciando, pausando e retomando, e botão Voltar.

Existem quatro etapas principais no uso do QUANTUM.

Passo 1: QUANTUM recebe um (manual ou automaticamente gerado) da GUI do aplicativo como um arquivo XML.

QUANTUM adiciona automaticamente bordas douradas para o

4A funcionalidade de zoom in e out não está disponível no JUnit e Frameworks Robotium, portanto, não os incluímos em nossa ferramenta.

conjunto de recursos suportados. Então, QUANTUM gera uma representação gráfica do modelo GUI usando o programa de pontos5 para que o usuário possa validar visualmente o modelo. A Figura 3 é um exemplo de representação gráfica que QUANTUM gerou.

Passo 2: Uma vez validado o modelo, QUANTUM percorre o modelo usando algoritmos de travessia para gerar suítes de teste. QUANTUM oferece as seguintes opções para percorrer o modelo: (1) nosso algoritmo descrito na Seção IV-D e (2) um algoritmo básico de busca em profundidade (DFS) que cobre todos os arestas para servir como uma linha de base para comparação. Em cima do nosso algoritmo de travessia, cada uma das otimizações pode ser ligado ou desligado de forma independente. Ao percorrer o modelo, QUANTUM gera um conjunto de testes JUnit6 . Os testes usam uma combinação do Robotium7 e JUnit para interagir com aplicativos Android.

Passo 3: No conjunto de testes gerado, QUANTUM insere automaticamente (1) instrumentação para registrar visualizações de estados, e (2) oráculos após o exercício de cada borda dourada. Gravação visualizações de estados podem ser feitas por meio de várias interfaces de usuário fornecido por uma plataforma móvel. Experimentamos dois interfaces da plataforma Android: Hierarchy Viewer8 e tirar instantâneos gráficos.

O Hierarchy Viewer é uma ferramenta para depurar interfaces de usuário de aplicativos que exibem a hierarquia e as propriedades dos itens no tela. Uma interface programática não está disponível para Hierarquia Visualizador para ser usado por testes, então implementamos um usando Java reflexão. A hierarquia e as propriedades dos itens na tela, fornecidos pela visão de estado, são então comparados por oráculos.

Instantâneos gráficos são obtidos de dentro dos testes JUnit e são então comparados usando processamento de imagem. Na corrente implementação do QUANTUM, instantâneos dos estados são registrados automaticamente e a comparação é baseada em um algoritmo básico de diferenciação de imagem que usa o sistema de coloração Red-Green Blue para comparar imagens pixel por pixel e permite um limiar ajustável de diferença. Desde o os estados são renderizados no mesmo dispositivo e na mesma tela, é concebível que a comparação básica de imagens possa ser boa o suficiente. De fato, tirar instantâneos gráficos provou ser mais fácil para usar do que o Hierarchy Viewer para o atualmente implementado conjunto de recursos, deu menos falsos positivos e foi mais rápido.

Etapa 4: Agora o conjunto de testes está completo e pode ser executado para testar o aplicativo em execução em um dispositivo ou emulador Android. Cada caso de teste percorre e verifica várias bordas douradas. Após a execução de cada teste, é fornecido um log que contém o resultado da verificação de cada borda dourada como Aprovado ou Reprovado. Dentro Além disso, QUANTUM tira instantâneos do aplicativo e fornece junto com o instantâneo esperado para cada falha. Esses instantâneos facilitam a identificação de falsos positivos, avaliando a gravidade dos bugs e depuração.

V. AVALIAÇÃO

Avaliamos QUANTUM em 6 aplicativos Android (3 aplicativos de aplicativos estudados anteriormente e 3 aplicativos dos aplicativos que selecionamos, como discutido na Seção II) para responder à seguinte pesquisa perguntas: (1) O QUANTUM pode encontrar bugs em aplicações reais? (2) Quão efetivo é o QUANTUM em termos da razão de reais bugs para falsos positivos (FP's)? (3) Quão compactos são os testes suítes geradas por QUANTUM?

5http://www.graphviz.org
6 http://junit.org
7https://code.google.com/p/robotium
8http://developer.android.com/tools/help/hierarchy-viewer.html

TABELA IV: Erros encontrados automaticamente com QUANTUM.

Aplicativo							
Bloco de notas (versão N/A)	8	22	7	22	9	4	3	2	2
OpenSudoku (versão 1.1.5)	7	67	11			4	5	2	3
Gerenciador de Nexes (versão 2.1.8)	15					3	8	2	7
VuDroid (versão 1.4)	6	16		3		0	3	0	2
Temporizador de cozinha (versão 1.1.6)	8	37		13		5	8	2	4
K9Mail (versão 4.317)	16	53		8		1	7	1	4
Total	**60**	**217**	**51**	**17**	**34**			**9**	**22**

A. RQ 1 e 2: Encontrando Bugs Reais

Dados os modelos GUI criados manualmente, usamos QUANTUM para gerar suítes de teste automaticamente. Primeiro usamos nossa travessia algoritmo com ambas as otimizações, juntamente com o oráculo de processamento de imagem. Em seguida, executamos automaticamente as suítes de teste em um emulador Android com root (executando o nível de API do Android 4.3 18 com uma CPU Intel Atom (x86), 512 MB de cartão SD e resolução WVGA800).

A Tabela IV resume os resultados da localização de bugs. #testes é o tamanho do conjunto de testes gerado para cada aplicativo usando nosso algoritmo de travessia com truncamento e priorização otimizações. Cada teste abrange várias bordas douradas e testes vários recursos, gerando conjuntos de testes compactos. #As sertions mostra o número total de asserções no conjunto de testes, que é igual ao número de bordas douradas. #Falhas é o número de afirmações que falharam. Nós investigamos manualmente o falhas e identificar bugs reais e falsos positivos. Alguns esses bugs ou falsos positivos foram revelados mais de uma vez. Portanto, mostramos o número de falsos positivos distintos e bugs nas duas últimas colunas.

A QUANTUM encontrou um total de 22 bugs em 6 aplicativos. Esses bugs incluíam 12 bugs de rotação, 1 bug de matar e reiniciar, 5 pausando e retomando bugs e 4 bugs do botão Voltar. Exemplos dos bugs são os seguintes.

Pausando e retomando o bug no K9Mail: O usuário finalmente encontra um e-mail depois de pesquisar na caixa de entrada por algum tempo, mas enquanto lê o e-mail ele recebe um telefonema (que pausa K9 Mail). Após o término da chamada, o K9Mail continua, mas de volta para a caixa de entrada, necessitando realizar a busca novamente.

Eliminando e reiniciando o bug no K9Mail: O funcionamento sistema decide matar o K9Mail por causa da pouca memória enquanto o usuário está compondo um e-mail. K9Mail não salva o e-mail como rascunho, excluindo o conteúdo do e-mail.

Erro de rotação no temporizador de cozinha: explicado na Seção III.

Bug de rotação no OpenSudoku: Girar o dispositivo fecha o pop-up personalizado para inserir números e os descarta.

Bug de rotação no VuDroid: Rotação limpa a seleção da guia.

Bug de rotação no Nexes Manager: se houver um vazio pasta que não tem permissão (leitura, gravação, etc.), girando o dispositivo faz com que o ícone da pasta desapareça.

Bug do botão Voltar no Timer de cozinha: indo para submenus e voltar faz com que os botões fiquem fora de foco.

Encontramos dois desses bugs já relatados e aceitos nos repositórios de bugs dos aplicativos correspondentes. Todos os outros bugs eram novos. Relatamos isso aos respectivos desenvolvedores e estão aguardando a confirmação dos bugs deles.

QUANTUM relatou um total de 9 falsos positivos distintos.

TABELA V: Compacidade dos Conjuntos de Testes Gerados.

	Básico	Nosso Algoritmo + Trunc.	+ Antes.	+ Ambos	DFS
Aplicativo					
Bloco de notas 73 11	15	59	12 65	8 44 35 114	
OpenSudoku 85 10	13	67	9 67	51 30 138	
Gerente Nexes 38 200 24		7 149 26 174	15 127	97 354	
VuDroidGenericName	8	41 7 14	38 6 36 174	15 71	
Cronômetro de cozinha	113 14 30 228		113 8 196	21 187 16	75 70 310
K9 Mail	25		148 76 490		

No entanto, 4 desses falsos positivos foram devido a uma inconsistência na instrumentação de teste do Android, o que causou para agir de forma diferente quando pausado programaticamente (por testes) ou através da GUI do emulador (ao confirmar manualmente os bugs). Outros 2 falsos positivos foram devido à sensibilidade do tempo de alguns estados do aplicativo. Por exemplo, quando um cronômetro está sendo executado em Temporizador de cozinha, a rotação altera os valores do temporizador, não porque é um bug, mas sim porque o estado de um cronômetro em execução muda com tempo. Essa sensibilidade ao tempo geralmente é abstraída de o modelo de GUI do aplicativo para obter concisão. Outros 2 falsos positivos manifestados porque, se a GUI do aplicativo fornecer um visual botão voltar na tela, apertando este botão visual e então o botão Voltar do hardware não leva o aplicativo para o original Estado. O 1 falso positivo restante pode ser considerado um bug, dependendo da intenção do designer do aplicativo.

B. RQ 3: Compacidade dos Conjuntos de Testes Gerados

Comparamos nosso algoritmo de geração de teste com uma linha de base DFS, na geração de casos de teste que cobrem todas as bordas douradas. Por o conjunto de recursos implementados (rotação, matar e reiniciar, pausando e retomando, e botão Voltar) medimos o compacidade dos conjuntos de testes gerados em termos do número de testes e o custo do conjunto de testes quando gerado por cada um dos seguintes algoritmos: (1) Nosso algoritmo básico; (2) Nosso algoritmo mais a otimização de truncamento, que trunca os casos de teste após a última borda dourada e ignora o teste casos que não cobrem nenhuma borda dourada; (3) Nosso algoritmo além da otimização de priorização, que prioriza arestas ao atravessar o modelo; (4) Nosso algoritmo mais ambos otimizações de truncamento e priorização; e (5) A DFS algoritmo começando na raiz.

A Tabela V mostra os resultados experimentais. A função custo é calculado com ÿ e ÿ definidos como 1. Como esta tabela mostra, nosso algoritmo mostra uma clara melhoria em relação ao DFS, em termos do número de testes, bem como o custo de execução do teste suítes. Além disso, o truncamento e a priorização melhoram a resultados quando aplicados separadamente (exceto para Kitchen Timer, para cujo truncamento não melhora o número de testes ou custo). Em alguns casos (Notepad, Nexes Manager e VuDroid), o truncamento produz melhores resultados em comparação com a priorização, enquanto em nos demais casos a priorização é mais efetiva. Felizmente, o otimizações são compatíveis e combináveis e usando tanto deles produz resultados ainda melhores para todos os aplicativos estudados.

Ameaças à validade: Para minimizar as ameaças à validade interna, automatizamos todo o processo de geração e execução de testes e identificamos manualmente erros reais de falsos positivos. Para abordar a validade externa, experimentamos 7 previamente estudaram aplicativos e estabeleceram um critério para escolher outros 6 populares aplicativos de repositórios de código aberto, conforme discutido na Seção II. Com relação à validade de construto, seguimos rigorosamente nossos algoritmo de travessia e técnicas de geração de oráculo, usadas frameworks bem conhecidos Robotium e JUnit, e manualmente investigou testes gerados para alguns dos aplicativos.

VI. TRABALHO RELACIONADO

Nosso trabalho tenta abordar o problema clássico do oráculo [7], [23], no contexto de aplicativos móveis. Na prática, oráculos de teste são normalmente especificados manualmente, muitas vezes no à custa de tempo e esforço substanciais. Há um corpo rico de trabalho que visa aliviar este problema de longa data gerando oráculos automaticamente. Especificação de software técnicas de mineração ou inferência de modelo são frequentemente usadas para isso. propósito. Uma pesquisa abrangente de inferência de propriedade da API técnicas de Robillard et al. descreve muitas dessas técnicas [22]. Técnicas automatizadas de geração de oráculos geralmente gerar oráculos de propósito geral para testes funcionais; eles são não específico para qualquer plataforma ou classe de aplicativos de software ou quaisquer aspectos do comportamento do software. No entanto, um estudo empírico recente estudo de Nguyen et al. sobre o custo e a eficácia oráculos automatizados conclui que sua taxa de falsos positivos é muitas vezes proibitivamente alto para uso prático [18].

Nossa técnica proposta não gera automaticamente oráculos de propósito geral, mas cai em um corpo relacionado de trabalho que usa oráculos criados manualmente com base no domínio conhecimento específico, que são adequadamente instanciados durante o teste e usados para testar aspectos muito específicos, às vezes não funcionais, do comportamento do software. Por exemplo, o A ferramenta TODDLER usa um oráculo feito à mão que detecta padrões de acesso à memória em loops para identificar o desempenho bugs em software [19]. Nosso trabalho anterior usou diferencial testes [12] para automação oracle no contexto de testes implementações de navegadores da web [26], bem como detectar erros entre navegadores em aplicativos da web [4], ou seja, discrepâncias em comportamento de aplicativos da web em diferentes navegadores da web, usando oráculos de teste projetados especificamente para esses domínios específicos aplicações de testes diferenciais. Nosso trabalho neste artigo explora características de aplicativos móveis e da plataforma móvel projetar oráculos para testar uma importante classe de interface de usuário funcionalidades de aplicativos móveis. Hu et ai. também contratar um especialista oráculo para testar aplicativos móveis Android, que implementa e verifica a especificação do ciclo de vida da atividade9 para Android aplicativos [8]. No entanto, este é um único oráculo, enquanto nosso abordagem propõe uma estrutura extensível que abrange todo um classe de propriedades – recursos de interação do usuário.

Há um corpo crescente de pesquisas focadas em automação testes de aplicativos móveis. As técnicas propostas abrangem a gama completa de tecnologias de testes aleatórios [8], [11], para geração de casos de teste baseada em execução simbólica [2], [16], testes baseados em modelos, testes combinatórios e suas combinações [17], [9]. No entanto, a ênfase aqui é gerar sequências de teste para maximizar a cobertura do código, por o objetivo de testes funcionais. O problema do oráculo não é diretamente abordados nestes artigos. Está implícito que os oráculos seria especificado manualmente ou usaria o oráculo simples correspondente a uma falha catastrófica quando a aplicação trava, trava ou lança uma exceção. Por contraste, o foco de nossa abordagem é justamente abordar o oráculo problema, para uma classe de não-funcionais e específicos da plataforma funcionalidades de aplicativos móveis.

9http://developer.android.com/training/basics/activity-lifecycle/index.html

Nossa abordagem para geração de sequência de teste se enquadra na ampla área de teste baseado em modelo. O modelo pode ser especificado manualmente ou extraído automaticamente da aplicação em teste. Na verdade, há um corpo rico e ativo de trabalho em engenharia reversa de tais modelos a partir da interface do usuário de aplicativos GUI [13], aplicativos web [15] e, mais recentemente, aplicativos móveis [1], [10], [25]. No entanto, nossa abordagem é independente do método usado para produzir o modelo e, portanto, é ortogonal a essas técnicas.

O objetivo da geração de sequência de teste baseada em modelo é extrair um conjunto de casos de teste concretos com base no comportamento representado no modelo. A maioria das técnicas nesta categoria faz isso resolvendo heuristicamente alguma variante do problema de cobertura de caminho mínimo NP-Hard [20], normalmente guiado por alguns critérios de análise de suporte e suficiência de suíte de teste. Memon et ai. propõem vários critérios de adequação de teste para testes de GUI com base na cobertura de eventos e seqüências de eventos no modelo GUI [14]. Arlt et ai. usar uma análise estática leve para calcular dependências de dados entre manipuladores de eventos de um aplicativo GUI e usar isso para orientar a escolha de sequências de teste do modelo GUI [3]. Ganov et al., por outro lado, focam no problema de gerar valores adequados para os parâmetros de entrada de sequências de teste abstratas extraídas de um modelo GUI e empregam execução simbólica para calcular esses valores de parâmetros [5]. Nguyen et ai. abordar o mesmo problema usando técnicas de teste combinatório para incorporar valores de dados especificados pelo usuário em sequências de teste abstratas [17]. O objetivo de todas as técnicas acima é o teste funcional do aplicativo e, mais especificamente, extrair casos de teste que maximizem a cobertura do código do aplicativo. Por outro lado, nossa geração de sequência de teste destina-se a exercitar exaustivamente um conjunto de recursos de interação do usuário específicos da plataforma. Isso leva a diferentes alvos de teste, funções de custo e, finalmente, um conjunto diferente de algoritmos de passagem de modelo do que aqueles por abordagens de teste funcional puro.

VII. CONCLUSÃO

Neste artigo, apresentamos uma nova abordagem para gerar automaticamente casos de teste, completos com oráculos de teste, para aplicativos móveis. Foi motivado por um estudo abrangente que realizamos de defeitos reais em aplicativos móveis. Através deste estudo, identificamos uma classe de recursos chamados recursos de interação do usuário, que foram implicados em uma fração significativa de bugs e para os quais oráculos podem ser construídos, de maneira agnóstica de aplicativos, com base em nosso entendimento comum de como os aplicativos se comportam. Nossa abordagem, conforme incorporada por nossa ferramenta QUANTUM, inclui uma estrutura extensível que suporta esses oráculos de teste específicos de domínio, mas agnósticos, e gera automaticamente um conjunto compacto de sequências de teste, incluindo oráculos de teste, para testar de forma abrangente os recursos de interação do usuário de um determinado aplicativo móvel. Nossa avaliação experimental inicial do QUANTUM em 6 aplicativos Android de código aberto foi bastante promissora: o QUANTUM encontrou um total de 22 bugs, alguns deles particularmente graves, usando um total de 60 testes para esses 6 aplicativos. Para trabalhos futuros, gostaríamos de aumentar o conjunto de oráculos atualmente suportados pelo QUANTUM, avaliá-lo mais extensivamente em um conjunto maior de aplicativos e explorar a possibilidade de estender nossa abordagem básica de geração de oráculos além do conjunto de recursos de interação do usuário relatados neste trabalhos.

RECONHECIMENTO

Agradecemos a Guowei Yang pelas discussões detalhadas e comentários sobre este trabalho. Este trabalho foi financiado em parte pelo Fujitsu Labs of America (SRA No. UTA12-001194) e pela National Science Foundation (NSF Grant No. CCF-0845628 e CNS-1239498).

REFERÊNCIAS

[1] D. Amalfitano, AR Fasolino, P. Tramontana, S. De Carmine e AM Memon. Usando a extração de GUI para testes automatizados de aplicativos Android. Em ASE, 2012.

[2] S. Anand, M. Naik, MJ Harrold e H. Yang. Teste concólico automatizado de aplicativos de smartphone. Em FSE, 2012.

[3] S. Arlt, A. Podelski, C. Bertolini, M. Schaf, I. Banerjee e AM Lembrete. Análise estática leve para testes de GUI. In ISSRE, 2012.

[4] SR Choudhary, M. Prasad e A. Orso. X-PERT: Identificação precisa de problemas entre navegadores em aplicativos da web. In ICSE, 2013.

[5] SR Ganov, C. Killmar, S. Khurshid e DE Perry. Análise de ouvinte de eventos e execução simbólica para testar aplicativos GUI. In ICFEM, 2009.

[6] J. Goodenough e S. Gerhart. Em direção a uma teoria de seleção de dados de teste. TSE, (2):156-173, 1975.

[7] NÓS Howden e E. Miller. Introdução à Teoria dos Testes. 1978.

[8] C. Hu e I. Neamtiu. Automatização de testes de GUI para aplicativos Android. Em AST, 2011.

[9] CS Jensen, MR Prasad e A. Møller. Teste automatizado com geração de sequência de eventos direcionada. Em ISSTA, 2013.

[10] ME Joorabchi e A. Mesbah. Engenharia reversa para dispositivos móveis iOS formulários. Em WCRE, 2012.

[11] A. Machiry, R. Tahiliani e M. Naik. Dynodroid: Uma geração de entrada sistema para aplicativos android. Em FSE, 2013.

[12] W. McKeeman. Testes diferenciais para software. Técnico Digital Journal, 10(1):100-107, 1998.

[13] AM Memon, I. Banerjee e A. Nagarajan. Extração de GUI: Engenharia reversa de interfaces gráficas de usuário para teste. Em WCRE, 2003.

[14] AM Memon, ML Soffa e ME Pollack. Critérios de cobertura para testes de GUI. Em FSE, 2001.

[15] A. Mesbah, A. van Deursen e S. Lenselink. Rastreamento de aplicativos da Web baseados em ajax por meio da análise dinâmica das alterações de estado da interface do usuário. TWEB, 6(1):3:1–3:30, 2012.

[16] N. Mirzaei, S. Malek, CS Pasareanu, N. Esfahani e R. Mahmood. Testando aplicativos Android por meio de execução simbólica. Notas de Engenharia de Software, 37(6):1–5, 2012.

[17] CD Nguyen, A. Marchetto e P. Tonella. Combinando testes baseados em modelos e combinatórios para geração eficaz de casos de teste. Em ISSTA, 2012.

[18] CD Nguyen, A. Marchetto e P. Tonella. Oráculos automatizados: um estudo empírico sobre custo e eficácia. Em FSE, 2013.

[19] A. Nistor, L. Song, D. Marinov e S. Lu. Criança: detectando problemas de desempenho por meio de padrões semelhantes de acesso à memória. In ICSE, 2013.

[20] SC Ntafos e SL Hakimi. Em problemas de cobertura de caminho em dígrafos e aplicações para testes de programas. TSE, 5(5):520-529, 1979.

[21] M. Pezze e M. Young. Teste e análise de software - processo, princípios e técnicas. 2007.

[22] MP Robillard, E. Bodden, D. Kawrykow, M. Mezini e T. Ratchford. Técnicas automatizadas de inferência de propriedades de API. TSE, 39(5):613-637, 2013.

[23] EJ Weyuker. A suposição do oráculo de teste de programa. Em ICSS, 1980.

[24] L. Williamson. Uma cartilha de desenvolvimento de aplicativos móveis: um guia para equipes corporativas que trabalham em projetos de aplicativos móveis. Documento técnico da IBM Software Thought Leadership, 2013.

[25] W. Yang, MR Prasad e T. Xie. Uma abordagem de caixa cinza para geração automatizada de modelos de GUI de aplicativos móveis. Em FASE, 2013.

[26] R. Nokhbeh Zaeem e S. Khurshid. Testar a geração de entrada usando programacao dinamica. Em FSE, 2012.