Geração de imagem de teste usando segmentação simbólica Avaliação para Testes Unitários

Tahir Jameel, Mengxiang Lin
Laboratório de ambiente de desenvolvimento de software do estado
Universidade Beihang
Pequim, China
{tahir, mxlin}@nlsde.buaa.edu.cn

Ele Li, Xiaomei Hou
Laboratório de ambiente de desenvolvimento de software do estado
Universidade Beihang
Pequim, China
{lihe, xmhou}@ nlsde.buaa.edu.cn

***Resumo*— Este artigo apresenta uma nova técnica para gerar imagens de teste usando avaliação simbólica segmentar para testar aplicativos de processamento de imagem. As imagens são multidimensionais e diversas por natureza, o que leva a diferentes desafios para o processo de teste. Uma técnica é necessária para gerar imagens de teste capazes de encontrar caminhos de programa derivados por pixels de imagem. A técnica proposta é baseada na execução simbólica que é amplamente usada para geração de dados de teste nos últimos anos. Em aplicativos de processamento de imagem, operações de pixel como média, convolução etc. são aplicadas em um segmento de pixels de imagem de entrada chamado janela para uma única iteração e repetidas para a imagem inteira. Nossa ideia principal é imitar operações na janela de pixel usando valores simbólicos em vez de concretos para gerar restrições de caminho no programa em teste. As restrições de caminho geradas para diferentes caminhos são resolvidas para valores concretos usando nosso solucionador SAT simples e as soluções são capazes de guiar a execução do programa para os caminhos específicos. As soluções de restrições de caminho são usadas para gerar imagens de teste sintéticas para cada caminho identificado e as restrições de caminho que não são solucionáveis para valores de pixel concretos são relatadas como caminhos inviáveis. Desenvolvemos uma ferramenta IMSUIT que recebe uma função de processamento de imagem como entrada e executa o programa simbolicamente para a janela de pixels fornecida para gerar imagens de teste. A eficácia do IMSUIT é testada em diferentes módulos de um sistema de reconhecimento óptico de caracteres e o resultado mostra que ele pode criar com sucesso imagens de teste para cada caminho do programa em teste e é capaz de identificar caminhos inviáveis.**

Palavras-chave — execução simbólica, teste de processamento de imagem, teste de unidade

Eu. EUNTRODUÇÃO

Recentemente, o papel dos aplicativos de processamento de imagem aumentou consideravelmente, como imagens médicas, digitalização de documentos, bioinformática, sensoriamento remoto e uma série de outros aplicativos. A importância dos sistemas de imagem baseados em decisão aumenta a necessidade de um sistema confiável e bem testado. O teste de software é uma abordagem vital para identificar bugs, o que é realizado pela análise de software e geração de entradas causadoras de bugs. O objetivo do teste de software é mostrar a ausência de bugs, mas, na prática, ele mostra apenas os bugs que se manifestam durante o processo de teste. Além disso, é a parte mais cara do ciclo de vida de desenvolvimento de software e pode exceder 50% do custo geral [1]. Apesar de sua limitação e custo, o teste de software ajuda a melhorar a qualidade do software e reduz os esforços manuais para testar o programa.

Nosso objetivo nesta pesquisa é automatizar o processo de teste de aplicativos de processamento de imagem. Em particular, buscamos gerar automaticamente imagens de teste que podem atingir uma métrica de cobertura de código, como cobertura de caminho. Na última década, a importância da execução simbólica [4, 6] para geração automática de entrada de teste foi amplamente comprovada para programas sequenciais e simultâneos escritos em diferentes linguagens [5, 10]. No entanto, o teste de aplicativos de processamento de imagem apresenta diferentes desafios que devem ser abordados.

Primeiro, para testar um software de processamento de imagens, lidamos com imagens semanticamente significativas. Em sistemas reais, essas imagens podem ser específicas ou gerais, por exemplo, o sistema de reconhecimento facial requer rostos e não rostos para teste, a classificação de tecidos cancerígenos e não cancerígenos requer imagens de ressonância magnética e conjuntos de imagens semelhantes para o sistema de recuperação de imagens baseado em conteúdo. Geralmente, esses sistemas são testados usando a classe disponível de imagens que eles processam e sua saída é analisada. Por exemplo, se testarmos um sistema de reconhecimento facial, damos diferentes imagens de teste ao sistema e verificamos se ele é capaz de combiná-las com o rosto certo no banco de dados de imagens. No entanto, ao fazer isso, estamos analisando a correção do algoritmo em vez do software. Nossa primeira observação é que imagens significativas são necessárias para o teste do algoritmo, não para sua implementação de software. Podemos usar imagens sintéticas para testar diferentes caminhos de um software e sua saída é analisada usando um oráculo de teste. Em segundo lugar, uma imagem é um dado de entrada multidimensional composto de diferentes níveis de brilho e uma grade 2D de posições de pixels; o que torna o processo de teste e a geração de dados de teste mais difíceis. Nossa segunda observação é que podemos fazer uso de um subconjunto de pixels de imagem para os quais as operações são aplicadas em uma única iteração e repetidas para a imagem inteira. Em terceiro lugar, a seleção de um oráculo de teste é baseada na semântica da aplicação de processamento de imagem. Por exemplo, se um programa classifica pixels de imagem em diferentes faixas de níveis de cinza, então o oráculo de teste deve ser baseado nos intervalos de pixels.

O teste manual é a técnica mais pronta sem usar recursos extras. Para esse propósito, o código é analisado e os casos de teste são gerados com base no fluxo de controle ou fluxo de dados do programa em teste. No entanto, o problema com essa abordagem é sua aplicabilidade. É um trabalho tedioso testar o sistema manualmente, especialmente quando os programas em teste têm dados multidimensionais. O teste exaustivo é uma escolha em sistemas críticos de segurança [2], mas no caso de aplicações de processamento de imagem,

978-1-4799-5604-3/14/$31,00 direitos autorais 2014 IEEE

não é viável testar o programa para todas as imagens de entrada possíveis em um tempo limitado. O teste aleatório [3, 8, 9] é uma abordagem simples, mas gera testes não direcionados. O tempo necessário para processar imagens é alto e o teste não direcionado carece de diversidade. O requisito é gerar imagens de teste que satisfaçam certa cobertura de código, como cobertura de caminho.

Este artigo apresenta um primeiro esforço para gerar automaticamente imagens de teste usando execução simbólica para testar aplicativos de processamento de imagem. A técnica proposta aborda os três desafios de testar aplicativos de processamento de imagem da seguinte maneira. O primeiro desafio é abordado pela criação de imagens sintéticas com base na semântica do programa. A técnica proposta visa atingir a cobertura do caminho e imagens de teste são geradas para cada caminho identificado. Os caminhos para os quais nenhuma imagem de entrada pode ser gerada são identificados como caminhos inviáveis. Testar cada caminho do programa usando imagens de teste aumenta a confiança na qualidade do sistema. O segundo desafio de lidar com dados multidimensionais em larga escala é abordado pela escolha de uma janela de pixels (4 vizinhos, 8 vizinhos etc.) como variáveis simbólicas e imagens de teste são geradas pela manipulação dessas variáveis. A ideia principal é explorar o fato de que certas operações são realizadas em uma janela de pixels selecionados e essas operações são repetidas para a imagem inteira para produzir a saída. O terceiro desafio é abordado pela seleção automática do oráculo de teste usando a semântica do programa em teste. Para um caminho de teste, os pixels da imagem resultante ocorrem em intervalos específicos de níveis de cinza que são usados como oráculo de teste.

Neste artigo, estendemos a execução simbólica tradicional para aplicativos de processamento de imagem usando um subconjunto de pixels chamado janela. Desenvolvemos uma ferramenta protótipo IMSUIT em Matlab que pode receber uma função de processamento de imagem escrita em Matlab como entrada e gerar imagens de teste automaticamente. Para fazer isso, geramos restrições em pixels de imagem durante a execução do programa em valores simbólicos. Em alguns casos, como a correspondência de padrões, essas restrições podem ser grandes o suficiente, usamos a simplificação de restrições para aumentar a velocidade da resolução de restrições. A restrição de caminho é uma expressão com algumas operações matemáticas e lógicas sobre variáveis de pixel. Para pixels de 8 bits, o valor dos níveis de cinza varia de 0 a 255. Os solucionadores de restrições [7] são usados para resolver as restrições. Usamos um solucionador simples baseado na geração de números aleatórios, pois o intervalo de valores de pixel não é alto. Para testar a eficácia do IMSUIT, aplicamos o IMSUIT em um sistema de reconhecimento óptico de caracteres (OCR) e geramos imagens de teste para seus módulos. O resultado mostra que, para cada caminho no programa em teste, as imagens são geradas para atingir a cobertura do caminho.

Este artigo tem três contribuições principais:

1. Uma nova ideia para gerar imagens de teste para testes unitários de aplicativos de processamento de imagem;
2. Desenvolvimento de um protótipo funcional IMSUIT;
3. Resultados experimentais mostrando a eficácia em aplicações reais de processamento de imagens.

O artigo está organizado da seguinte forma. Na seção 2, uma breve visão geral da abordagem é apresentada usando um exemplo simples. Na seção 3, os detalhes da abordagem são discutidos. Na seção 4, a implementação da abordagem e sua avaliação são apresentadas.

```
                    .
:                  =     =    (              )
:   ;
:
:   =    ;
:     =  :   −
:        =  :   −
:          =              (  −  ,   −  );     =
:          (  −  ,  );     =            (  −  ,
:          +  );     =            (  ,   −  );
:          =              (  ,   );
:
:          =              (  ,   +  );     =
:          (  +  ,   −  );     =            (  +
:          ,  );     =            (  +  ,   +  );
:
:          = (    +     +     +     +     +     +     +     +    )/  ;    (      <
:          h     h    )
:                (  −  ,   −  ) =  ;
:
:                (  −  ,   −  ) =     ;
:
:
:
```

Figura 1 Código de exemplo

usando um aplicativo de processamento de imagem real. Finalmente, a seção 5 conclui a discussão.

II. OVISÃO GERAL

Usamos um exemplo simples para ilustrar o funcionamento do IMSUIT. Considere uma função*binário*na Fig. 1, que pega uma imagem de entrada em escala de cinza de 100 por 100 pixels e gera sua imagem binária. O programa seleciona uma janela de oito pixels vizinhos em termos de*eu e j*para uma dada localização de pixel para tirar a média da janela para fins de suavização. Então o programa aplica o limiar para converter os pixels cinzas em pixels pretos e brancos na imagem resultante. O programa tem dois caminhos, um é seguido quando a média da janela selecionada é menor que um limiar, enquanto o outro é seguido quando a média é maior ou igual a um limiar. Para gerar imagens de teste para tais funções, o IMSUIT precisa de dois tipos de entradas, uma é o programa em teste e a outra é o nome das variáveis que devem ser executadas simbolicamente. No exemplo acima, as variáveis $p_1$principal$_9$armazena os valores de oito pixels vizinhos de um pixel centralizado da imagem de entrada em uma única iteração. Os valores de $p_1$principal$_9$são alteradas iterativamente para a próxima janela de pixels vizinhos até que toda a imagem seja percorrida. Essas variáveis são fornecidas como entrada para IMSUIT para articular que essas variáveis devem ser tratadas como variáveis simbólicas durante a execução.

O IMSUIT gera restrições do programa em teste usando avaliação simbólica. O programa em teste é analisado linha por linha e os estados das variáveis do programa são armazenados em diferentes estruturas projetadas para variáveis, imagens, loops, ramificações e uma pilha é projetada para resolver o escopo de instruções de ramificação. Essas estruturas imitam a pilha de memória do programa durante a execução simbólica. A Tabela 1 mostra os estados das variáveis do programa executadas simbolicamente em cada etapa. Na linha 2, há uma atribuição simples de um valor concreto a uma variável. Para cada atribuição simples, é avaliado se a variável do lado esquerdo é uma nova variável ou já declarada.

Tabela 1 Estados Simbólicos

linha	média	debulhar	r	c	eu	eu	computador
1	?	?	?	?	?	?	?
2	?	128	?	?	?	?	?
3	?	128	100	?	?	?	?
4	?	128	100	100	?	?	?
5	?	128	100	100	2	?	i≤r-1
6	?	128	100	100	2	2	i≤r-1&j≤c-1
7~15	?	128	100	100	2	2	i≤r-1&j≤c-1
16	∑= ( )	128	100	100	2	2	i≤r-1&j≤c-1
17	∑= ( )	128	100	100	2	2	i≤r-1&j≤c-1 & ∑= ( )<128
18	O pixel da imagem de saída é atribuído 0						
19	∑= ( )	128 100 100 2				2	i≤r-1&j≤c-1 & ∑= ( )≥128
20	O pixel da imagem de saída é atribuído a 255						

Se uma nova variável for declarada, seu nome e valor serão armazenados na estrutura da variável, caso contrário, o valor da variável já declarada será substituído. As linhas 3 e 4 também são atribuições simples para novas variáveis. No caso de*para*loops, a condição do loop deve ser satisfeita para executar pelo menos uma única iteração. Os loops nas linhas 5 e 6 são aninhados, para executar as instruções no loop aninhado as condições de ambos os loops devem ser satisfeitas. Então a condição do caminho se torna:

Computador: ≤ − 1 e ≤ − 1 (1)

As instruções 7 a 15 inicializam as variáveis simbólicas com os valores de pixel da imagem de entrada. As variáveis $p_1$principal$_9$são tratadas como variáveis simbólicas, conforme fornecido na entrada. Essas variáveis são armazenadas na estrutura de variáveis simbólicas e as variáveis cujos valores são baseados nessas variáveis simbólicas também são tratadas como variáveis simbólicas. Na linha 16, as variáveis simbólicas são usadas para calcular a média e a variável*média*juntamente com seu valor simbólico são armazenados na estrutura da variável simbólica. A equação 2 mostra o valor atribuído à variável*média*em termos simbólicos:

= [1+ 2+ 3+ 4+ 5+ 6+ 7+ 9+ 9]/9 (2)

Um branch divide a execução do programa em dois caminhos e ambos os caminhos são executados em execução simbólica. O IMSUIT analisa as condições do branch e gera restrições de caminho para condições true e false. Na linha 17, uma condição de caminho é a seguinte:

Computador: ≤ − 1 & ≤ − 1 &∑9 =1 ( ) / 9 < 128 (3)

Enquanto a condição do caminho alternativo é:

¬PC: ≤ − 1 & ≤ − 1 &∑9 =1 ( ) / 9 ≥ 128 (4)

```
:
:
(   ,   _   )
:   =   ( );   ≠
:
:   ...   ( )
:
:   =   ( )   =
6:  ( );
:
:   _  ...
:   =   (  _  )
:   ≠
:   =   (   )
:
:
:
```

Figura 2 Algoritmo

A palavra-chave*fim*está associado ao escopo correto de*para* laço ou*se/senão**se*branch usando uma pilha. As condições de caminho extraídas da execução simbólica são resolvidas para valores concretos usando um solucionador de restrições simples com base na geração de números aleatórios. Um valor de pixel varia de 0 a 255 e para resolver uma condição de caminho, valores arbitrários são gerados a partir do intervalo. Se esses valores forem capazes de satisfazer a condição de caminho, então é uma solução possível, caso contrário, o processo é repetido para outro conjunto de números arbitrários. Uma condição de caminho pode ter uma solução específica ou várias soluções. As restrições acima têm várias soluções para cada caminho, variando de 0 a 127 e 128 a 255.

As soluções computadas para as condições de caminho são usadas para sintetizar imagens de teste e para cada caminho no programa em teste, pelo menos uma imagem de teste é criada. Como no caso acima, um caminho tem várias soluções e para gerar imagens de teste, o IMSUIT randomiza diferentes soluções. Uma imagem de teste garante que o programa em teste seguirá um caminho específico quando executado, fornecendo a imagem de teste como entrada. Isso cria facilidade de teste de oráculo e processo de depuração. No exemplo acima, duas imagens de teste são criadas para testar cada caminho. Quando uma imagem de teste é fornecida como entrada para um programa, a asserção não deve ser violada. Na Fig. 1, as linhas 18 e 20 definem os valores de pixel da imagem de saída que são usados como oráculo de teste. Por exemplo, para o caminho 1, quando a imagem de teste é executada, todos os pixels da imagem resultante devem ser zero e para o caminho 2 todos os pixels de uma imagem resultante devem ser 255. Se houver um caminho para o qual nenhuma imagem de teste pode ser criada, então o caminho é inviável ou pode ser parte de código morto.

III.UmAPROXIMAÇÃO

IMSUIT recebe um programa como entrada escrito na forma de um conjunto de instruções, que consiste em atribuições, instruções de ramificação e loops. Uma instrução de atribuição pode ser uma equação simples ou matemática com um operador de atribuição. instrução de atribuição atribui o valor avaliado do lado direito a uma variável do lado esquerdo. A atribuição pode ser por meio de uma chamada de função, um dígito simples, variável ou equação. O

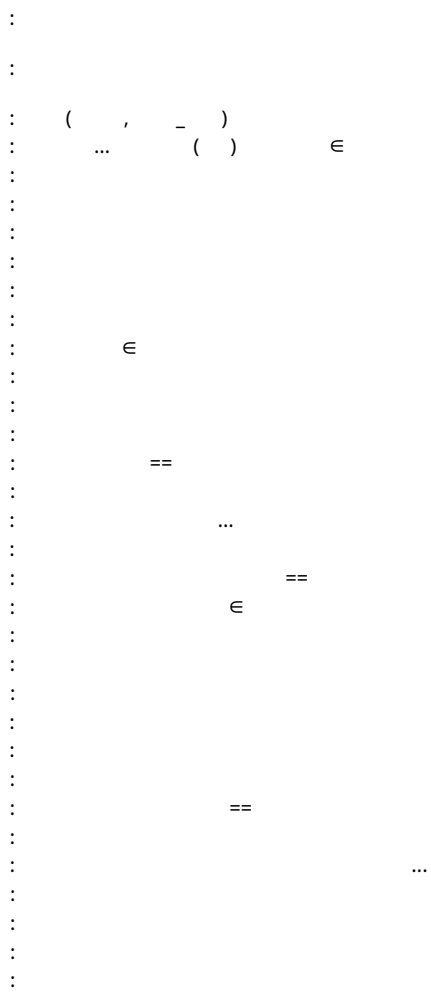

Figura 3 Avaliador Simbólico

instruções de ramificação têm uma expressão booleana seguida por instruções e então termina seu escopo. Uma condição de ramificação pode ser simples consistindo de uma única condição ou complexa consistindo de múltiplas condições concatenadas. Pode haver instruções para condição falsa de expressão booleana também. Uma instrução de loop tem uma variável de loop com um valor inicial e um valor final. Dentro do loop, há instruções e então termina seu escopo. O IMSUIT visa extrair caminhos de programa de acordo com as estruturas de programa acima e gera imagens de teste para cada caminho.

Para executar o programa simbolicamente, precisamos saber sobre as construções e semânticas do programa. A Fig.3 mostra uma visão geral do algoritmo IMSUIT. Um programa de teste é tomado como entrada com uma lista de variáveis necessárias para executar simbolicamente. O programa é lido linha por linha e passado para um avaliador simbólico, que analisa a declaração e extrai as informações úteis necessárias para executar o programa simbolicamente. Essas informações são usadas para atualizar a pilha e as estruturas que armazenam os estados do programa. Após extrair as informações necessárias, a declaração é executada simbolicamente e as restrições de caminho são atualizadas. Uma vez que o programa é processado, as restrições de caminho para diferentes caminhos são extraídas. Essas restrições de caminho podem ser complexas, o que é simplificado para agilizar o solucionador de restrições, pois

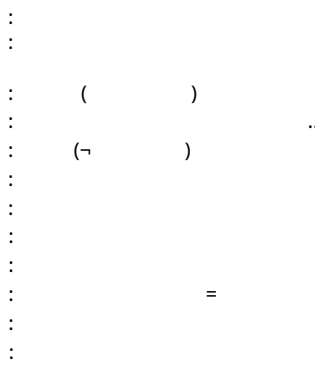

Figura 4 Solucionador de Restrições

é a parte computacionalmente mais cara. Para encontrar soluções concretas, a restrição de caminho simplificada é passada para um solucionador de restrições. As soluções concretas são usadas para gerar imagens de teste. Enquanto isso, se a restrição de caminho não puder ser resolvida, então o caminho é inviável. Os caminhos inviáveis são relatados ao usuário.

Na avaliação simbólica, as informações no programa em teste são extraídas e analisadas sem executá-lo. O objetivo da avaliação simbólica é extrair a semântica da estrutura do programa. Diferentes estruturas são projetadas para armazenar estados do programa, por exemplo, variáveis, variáveis simbólicas, imagens, loops e ramificações. Uma pilha é projetada para resolver o escopo de loops e instruções de ramificação. O avaliador simbólico pega uma declaração do programa em teste e uma lista de variáveis simbólicas. A Fig. 3 mostra como o avaliador simbólico funciona. Primeiramente, o token de string da declaração é examinado para palavras-chave específicas da linguagem de programação, por exemplo*se, senão, para*etc. Se o token for uma palavra-chave, a linha será analisada de acordo com os valores esperados específicos para essa palavra-chave. Por exemplo, se a palavra-chave for*se*então o avaliador simbólico espera uma expressão booleana após o*se* palavra-chave. Em segundo lugar, se o token não for uma palavra-chave, ele será verificado para uma palavra reservada específica para a linguagem de programação, por exemplo, em Matlab*clc, figura*etc. Essas palavras reservadas não contribuem na execução simbólica, mas são usadas apenas para exibir imagens, fechar arquivos etc. Terceiro, se o token não for uma palavra reservada, a linha será verificada para uma declaração de atribuição. Se a linha tiver um operador de atribuição, seu lado direito será analisado. O lado direito da equação pode ser uma chamada de função cuja saída é atribuída à variável no lado esquerdo. Caso contrário, a declaração pode ser uma atribuição simples com um dígito ou variável no lado direito ou uma equação consistindo de variáveis, dígitos e operadores cujo valor é computado e atribuído à variável no lado esquerdo. Os estados das variáveis são atualizados na estrutura das variáveis de acordo. As ramificações são importantes na execução simbólica. Quando um*se*ou *senão* palavra-chave é encontrada, sua condição booleana é analisada. Uma condição pode ser simples ou combinações de várias condições. As várias condições são concatenadas usando*&*ou /operadores. Variáveis de condição e operadores lógicos são armazenados em uma estrutura que é mantida para instruções de ramificação. Conforme discutido anteriormente, a execução simbólica segue ambos os caminhos de uma ramificação, para esse propósito a condição falsa também é computada e armazenada na estrutura de condições junto com a condição verdadeira. Em ramificações aninhadas, a condição verdadeira do pai é concatenada

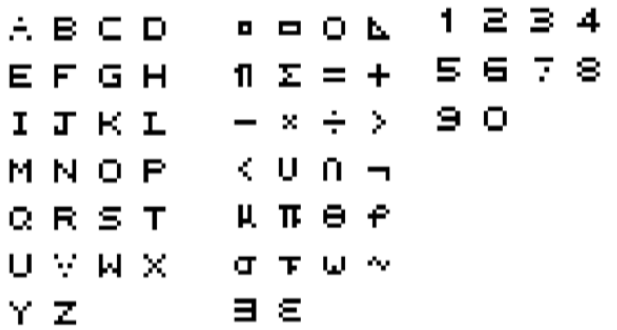

Figura 5 Imagem de teste gerada para OCR

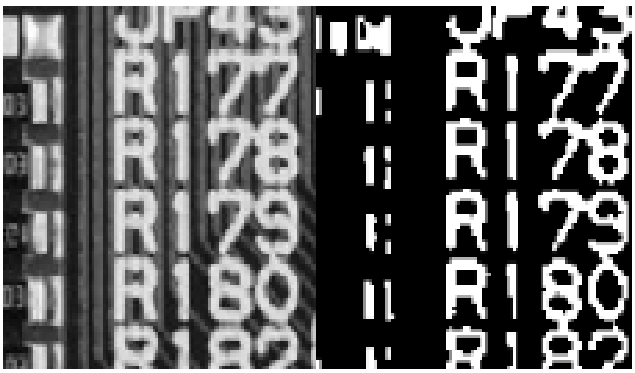

Figura 6 Imagem Binarizada

com a criança. Enquanto que no caso de*senão*a falsa condição do pai*se* condição é concatenada com a condição booleana da criança usando*&* operador. Essas condições de ramificação são usadas para atualizar a restrição de caminho na execução simbólica. As variáveis de condição são principalmente as variáveis simbólicas ou variáveis computadas usando variáveis simbólicas porque as condições são geralmente aplicadas para classificar pixels ou função de pixels. Os loops no programa têm uma condição booleana para executar as iterações do loop e seu término. A condição do loop também é concatenada com a condição do caminho após a extração da instrução. Sempre que uma condição ou uma ramificação ocorre, a pilha é atualizada para resolver seu escopo.

Para uma declaração de atribuição, se o tipo for uma atribuição simples, a variável do lado esquerdo é uma nova variável ou já declarada. Sempre que uma variável é analisada, ela é pesquisada na estrutura existente de variáveis. O lado direito da equação também é analisado se for um número, então o valor é escrito no campo de valor da estrutura da variável. Mas se o lado direito for uma variável, então o valor da variável é extraído da estrutura da variável e copiado para a variável do lado esquerdo em seu campo de valor na estrutura das variáveis. Se a equação for matemática ou lógica, então a equação é analisada para cada operando e operador. A equação é armazenada em termos de variáveis e variáveis simbólicas que são avaliadas durante a execução simbólica. Ao mesmo tempo, as variáveis são verificadas se forem simbólicas, então os valores são armazenados na estrutura da variável simbólica. As variáveis simbólicas são geralmente os pixels de imagens nas quais diferentes operações são realizadas. Elas são inicializadas pelos pixels da imagem e então seus valores simbólicos são alterados durante a execução. As variáveis cujos valores são computados usando variáveis simbólicas também são tratadas como variáveis simbólicas e seus valores simbólicos são armazenados na estrutura da variável simbólica.

Após extrair as informações por avaliação simbólica, a declaração é executada simbolicamente. Durante a avaliação simbólica, uma mensagem também é gerada para execução simbólica contendo as informações semânticas. Por exemplo, informações semânticas são geradas para diferentes declarações, como ramificação, atribuição simples, leitura de imagem, substituição de variável simbólica etc. e sinalizadores específicos são definidos. O avaliador simbólico encontra o sinalizador e executa a tarefa correspondente. O avaliador simbólico executa o programa em valor simbólico em vez de valores concretos.

Sempre que encontra uma ramificação, ele segue ambos os caminhos. A restrição de caminho também é atualizada para cada execução de instrução de ramificação. A condição de um*para*loop é semelhante a um*se* condição. Condições de caminho são geradas para cada caminho agregando todas as condições de ramificação sobre variáveis simbólicas. A condição de caminho pode ser uma sequência de declarações lógicas e matemáticas concatenadas. Em alguns casos, a condição de caminho pode ser simplificada. A resolução de restrições é computacionalmente a parte mais cara do sistema. Ao simplificar a condição de caminho, podemos torná-la mais rápida. Usamos um simplificador de restrições para simplificação de restrições.

Um solucionador de restrições verifica se uma condição de caminho pode ser resolvida para valor concreto. Desenvolvemos um solucionador de restrições mostrado na Fig. 4, com base na geração de números aleatórios. Normalmente, as variáveis simbólicas são pixels cujos valores são armazenados em variáveis de 8 bits que variam de 0 a 255. O intervalo de números aleatórios é pequeno e um solucionador SAT baseado em gerador de números aleatórios pode encontrar a solução concreta para uma restrição de caminho rapidamente. Uma vez que a solução para a condição de caminho é computada, ela é usada para gerar imagens de teste. Várias imagens podem ser geradas se houver várias soluções para a restrição de caminho. Uma única imagem contendo diferentes soluções concretas também pode ser gerada. Essas imagens de teste são fornecidas como entrada para o programa em teste e a saída do programa é avaliada. Por exemplo, no limiar global, o valor do pixel da imagem cinza é verificado. Se o pixel for menor que 128, então o pixel da imagem binarizada resultante é atribuído a 0, caso contrário, 255. Existem várias soluções para suas condições de caminho. As várias soluções são randomizadas nas imagens de teste. Quando a imagem de teste do caminho 1 é executada, a imagem resultante é toda preta. Enquanto isso, quando a imagem de teste para o caminho 2 é executada, a imagem resultante é toda branca.

IV. EuMPLEMENTAÇÃO E AVALIAÇÃO

Desenvolvemos uma ferramenta IMSUIT em Matlab para gerar imagens de teste para funções escritas em Matlab. Ela consiste em 2K linhas de códigos. O IMSUIT é testado em um sistema de reconhecimento óptico de caracteres real, que tem cinco módulos diferentes. A Tabela 1 mostra os resultados das imagens de teste criadas e seus caminhos. A primeira função é*limiar global*, que binariza a imagem dada usando valores de limite. Esta função tem dois caminhos e para testar precisamos de pelo menos uma imagem para cada caminho. Existem 128 soluções diferentes para ambas as restrições de caminho. A primeira imagem criada tem

Tabela 2 Resultados dos Módulos do OCR

Sr.	Função	Linhas de Código	Não de Caminhos	Sem teste Imagens
1	Limiar global	31	2	2
2	Limiar suave	64	4	4
3	Alfabetos OCR	164	27	27
4	Números OCR	106	11	11
5	Caracteres especiais OCR	106	27	27

valores aleatórios de pixels menores ou iguais a 128 e o segundo tem valores aleatórios maiores que 128. Quando a primeira imagem de teste é dada como entrada para a função, sua saída é toda preta, enquanto a saída da segunda imagem é toda branca. A violação do oráculo de teste aponta para um bug na função. A segunda função *limiar suave* binariza a imagem de entrada e também suaviza a imagem de saída para eliminar ruídos que causam problemas no reconhecimento de caracteres. Esta função usa três bandas diferentes de valores de pixel para binarizar a imagem de entrada. Esta função usa 8 vizinhos de um pixel centralizado e calcula sua média. A média é comparada aos três limites diferentes. Esses 9 pixels são considerados variáveis simbólicas na execução. A função tem quatro caminhos diferentes e para cada caminho uma imagem de teste é criada. A terceira função é *Alfabetos OCR* que busca padrões de alfabeto na imagem fornecida. Esta função tem 27 caminhos e 27 imagens são geradas para cada caminho. A quarta função é *Números OCR* que busca padrões numéricos na imagem de teste fornecida. IMSUIT gera imagem de teste para números usando as restrições de caminho. Ele tem 11 caminhos e 11 imagens de teste são geradas para testar esta função. A quinta função é *Caracteres especiais OCR* que busca padrões de caracteres especiais na imagem de entrada. Há 26 caminhos diferentes para essa função e 26 imagens de teste são geradas para testá-los.

A Fig. 5 mostra as imagens de entrada geradas para alfabetos, números e caracteres especiais. As imagens criadas para outras funções são visualmente sem sentido, por isso não são mostradas. A Fig. 6 mostra uma imagem em escala de cinza no lado esquerdo que é pré-processada usando *limiar suave* função. Os efeitos de suavização são claramente visíveis na imagem binarizada no canto superior direito da figura. Alfabetos e números são claramente visíveis na imagem e esta imagem está pronta para ser dada como entrada para alfabetos OCR, números OCR e caracteres especiais OCR. No entanto, essas funções podem ser testadas usando imagens de teste mostradas na Fig. 5. Se um conjunto de imagens de teste for testado cobrindo todos os caminhos de uma função, então a função pode ser usada para imagens reais com maior confiança.

V. Conclusão

Testar um programa com a ajuda de dados de teste é uma maneira básica de verificar a funcionalidade do programa em teste. Neste artigo, usamos avaliação simbólica segmental para a geração de imagens de teste para aplicativos de processamento de imagem e demonstramos sua utilidade com a ajuda do sistema de reconhecimento óptico de caracteres. Este é o primeiro esforço para gerar imagens de teste automaticamente para testes unitários de módulos de processamento de imagem. A ferramenta IMSUIT é uma ferramenta protótipo desenvolvida para Programas Matlab, no entanto, podem ser implementados para outras linguagens de programação. O IMSUIT é capaz de receber funções de entrada e gerar imagens de teste para todos os seus caminhos usando avaliação simbólica e relatando os caminhos inviáveis. Desenvolvemos um solucionador SAT simples com base na geração de números aleatórios. As restrições de caminho dos sistemas de processamento de imagem compreendem pixels e os valores de pixels são de 0 a 255 para imagens de 8 bits. Esse fato torna possível o uso de um solucionador SAT simples. A ferramenta é testada em diferentes módulos do sistema de reconhecimento óptico de caracteres. O resultado mostra que ele criou com sucesso imagens de teste para cada caminho do programa.

Um Reconhecimento

Gostaria de agradecer ao conselho de bolsas de estudo chinês por proporcionar um ambiente de pesquisa propício.

Referências

[1] Beizer, B., 1990. Técnicas de Teste de Software, segunda edição. Internacional

[2] Knight, John C., Kevin G. Wika e Shannon Wrege. "Testes exaustivos como técnica de verificação." Submetido ao Simpósio Internacional sobre Testes e Análise de Software. 1996. J. Clerk Maxwell, Um Tratado sobre Eletricidade e Magnetismo, 3ª ed., vol. 2. Oxford: Clarendon, 1892, pp.68-73.

[3] Ciupa, Ilinca, et al. "Encontrando falhas: Testes manuais vs. testes aleatórios vs. relatórios de usuários." Engenharia de Confiabilidade de Software, 2008. ISSRE 2008. 19º Simpósio Internacional sobre. IEEE, 2008..

[4] King, James C. "Execução simbólica e teste de programa." Comunicações do ACM 19.7 (1976): 385-394..

[5] Cadar, Cristian, Daniel Dunbar e Dawson R. Engler. "KLEE: Geração automática e não assistida de testes de alta cobertura para programas de sistemas complexos." OSDI. Vol. 8. 2008

[6] Cadar, Cristian e Koushik Sen. "Execução simbólica para testes de software: três décadas depois." Communications of the ACM 56.2 (2013): 82-90.

[7] Armand, Michaël, et al. "Uma integração modular de solucionadores SAT/SMT para Coq por meio de testemunhas de prova." Programas e Provas Certificados. Springer Berlin Heidelberg, 2011. 135-150.

[8] Korel, Bogdan. "Geração automatizada de dados de teste de software." Engenharia de Software, IEEE Transactions em 16.8 (1990): 870-879.

[9] Thevenod-Fosse, Pascale e Helene Waeselynck. "STATEMATE aplicado a testes estatísticos de software." ACM SIGSOFT Software Engineering Notes 18.3 (1993): 99-109.

[10] Anand, Saswat, Corina S. Păsăreanu e Willem Visser. "JPF–SE: Uma extensão de execução simbólica para java pathfinder." Ferramentas e Algoritmos para a Construção e Análise de Sistemas. Springer Berlin Heidelberg, 2007. 134-138.