Universidade Federal de Pernambuco

Centro de Informática

Ciência da Computação

Encontrando Diferenças de Integração entre Versões de um Programa com Abstrações de Mudança

Trabalho de Graduação

Aluno: Filipe Araujo de Almeida(faa@cin.ufpe.br)

Orientador: Prof. Dr. Marcelo Bezerra d'Amorim (damorim@cin.ufpe.br)

Recife, Dezembro de 2009

Universidade Federal de Pernambuco

Centro de Informática

Ciência da Computação

Encontrando Diferenças de Integração entre Versões de um Programa com Abstrações de Mudança

Trabalho de Graduação

Monografia apresentada à Universidade Federal de Pernambuco, como trabalho de conclusão do curso de Graduação em Ciência da Computação, para a obtenção do título de Bacharel em Ciência da Computação.

Aluno: Filipe Araujo de Almeida (faa@cin.ufpe.br)

Orientador: Prof. Dr. Marcelo Bezerra d'Amorim (damorim@cin.ufpe.br)

Recife, Dezembro de 2009

ASSINATURAS

Este Trabalho de Graduação representa o esforço do aluno **Filipe Araujo de Almeida**, orientado pelo professor **Marcelo Bezerra d'Amorim**, sob o título **"Encontrando Diferenças de Integração entre Versões de um Programa com Abstrações de Mudança"**. Todos abaixo estão de acordo com o conteúdo deste documento e os resultados deste trabalho de graduação.

_____

Marcelo Bezerra d'Amorim

(orientador)

_____

Filipe Araujo de Almeida

(aluno)

AGRADECIMENTOS

Aos meus pais e irmão por terem me acompanhado e apoiado durante todas as fases de minha vida.

À minha namorada, por ter sido paciente o bastante nos momentos em que não lhe pude dar atenção.

Ao colega Rafael Oliveira, por ter cedido sua implementação do classificador de branch e ter ajudado nos experimentos.

Ao meu orientador Marcelo d'Amorim, por sua dedicação e energia e pela disponibilidade que sempre demonstrou para a orientação do desenvolvimento deste trabalho.

E a todos que de alguma forma me ajudaram durante a minha formação.

RESUMO

Programas mudam com a implementação de novos requisitos, refactorings e correção de defeitos. Suítes de teste devem acompanhar a evolução do programa, caso contrário mudanças de comportamento não intencionais decorrentes das modificações podem não ser observadas. Nesse sentido, surge a preocupação de como gerar classificadores de teste que capturem diferenças de comportamento entre duas versões de um programa. Nossa abordagem será usar uma versão como referência para checar a conformidade com a versão que está sendo testada. Para isso, os *traces* de execução de sequências de teste rodadas sobre duas versões consecutivas do programa serão comparados. Desta forma, a entrada do classificador automático serão duas versões consecutivas de um programa e sequências de teste e a saída será um veredito: conforme ou diferente. Como o foco deste trabalho é encontrar diferenças de integração, o classificador procura dinstiguir efeitos colaterais locais (numa unidade do programa) de efeitos colaterais não locais (que se propagam para outros módulos do programa). Para tanto, o classificador deve abstrair dos *traces* partes que tenham sido geradas por código próximo às modificações sintáticas realizadas entre as versões.

SUMÁRIO

1. INTRODUÇÃO

Programas mudam com a implementação de novos requisitos, refactorings, e correção de defeitos. As suítes de teste devem acompanhar a evolução do programa, caso contrário mudanças de comportamento não intencionais decorrentes das modificações podem não ser observadas. Considere o caso em que uma suíte de teste passa em sua totalidade após mudanças no código. É possível que a suíte não tenha sido capaz de identificar importantes efeitos colaterais no sistema, como mudanças de estado e/ou fluxo de controle. Isso pode acontecer tanto por causa de cenários de teste insuficientes na suíte ou por causa de asserções insuficientes em cada sequência de teste. É importante observar que alguns efeitos colaterais são intencionais (por exemplo, devido à correção de defeitos ou atualização de features) e, portanto, não são nocivos. Por outro lado, efeitos colaterais não intencionais (por exemplo, devido à adição de um bug) são custosos e podem resultar em erros de regressão não identificados.

A comunidade de testes recentemente reconheceu a importância de identificar a inadequação de uma suíte para retestar programas modificados. Person et al. [1] e Orso e Xie [2] propõem a construção incremental de testes, de acordo com os novos requisitos de teste que as mudanças no software originam [3][4]. Dessa forma, uma ferramenta deve fornecer ao usuário casos de teste que revelem comportamentos diferentes quando executados em versões sucessivas de um programa. Em particular, casos de teste que revelem os efeitos de mudanças de unidades sobre outros módulos do sistema. A ferramenta pode tanto selecionar esses casos de teste entre casos de testes já existentes, bem como construir novos casos de teste. Assim, surgem duas importantes preocupações ortogonais: (i) como gerar sequências de teste que irão manifestar diferenças de comportamento e (ii) como gerar classificadores que capturem as mudanças de comportamento. Este trabalho foca na segunda preocupação.

Nossa abordagem será usar uma versão de um programa como referência para checar a conformidade com a versão que está sendo testada. Para isso, os traces de execução de sequências de teste rodadas sobre duas versões consecutivas do programa serão comparados. Desta forma, a entrada do classificador automático serão duas versões consecutivas de um programa e sequências de teste e a saída será um veredito: conforme ou diferente. Como o foco deste trabalho é encontrar diferenças de integração, o classificador procura dinstiguir efeitos colaterais locais (numa unidade do programa) de efeitos colaterais não locais (que se propagam para outros módulos do programa). Para tanto, o classificador deve abstrair dos traces partes que tenham sido geradas por código próximo às modificações sintáticas realizadas entre as versões – uma vez que se espera que essas partes devem de fato ser diferentes.

Considere o cenário no qual o programador muda a implementação de uma lista em um módulo do programa. Ele substitui uma lista ligada por uma implementação com array dinâmico, por exemplo. As mudanças nas partes do estado global que armazenavam listas ligadas não originam necessariamente mudanças na integração do sistema. Somente se pode observar uma diferença de comportamento quando outras partes do programa ativam a mudança (por exemplo, a execução lê uma parte modificada do estado e toma uma decisão de fluxo de controle baseada nessa leitura).

Este trabalho traz as seguintes contribuições:

- Apresentação de dois classificadores que usam diferentes heurísticas para identificar diferenças de integração entre duas versões de um programa. Nós propomos um classificador que reporta possíveis diferenças de integração a partir da observação de mudanças de estado e um classificador que reporta reais diferenças a partir de mudanças no fluxo de controle.

- Avaliação empírica tomando por base o Jtopas, um parser para dados textuais arbitrários. Nós usamos versões corretas e defeituosas do repositório SIR [5] e avaliamos a capacidade dos classificadores de

revelar defeitos injetados com a infra-estrutura do SIR, com relação ao classificador manual[1] também disponível no SIR. Os resultados experimentais mostram que os classificadores propostos deixaram de encontrar o erro em apenas um dos 31 casos em que pelo menos um dos classificadores encontra. Na média, os classificadores propostos revelam os erros em mais sequências de teste que o classificador manual. Os resultados sugerem que devemos considerar primeiramente o veredito do classificador de fluxo de controle, uma vez que, em comparação com o classificador de estados, o primeiro reporta menos alarmes falsos, mas deixa escapar mais erros. O experimento também indica que os classificadores propostos poderiam potencialmente ter encontrado mais erros caso as sequências de teste exercitassem mais código defeituoso (apenas 48% dos testes exercitam código que infecta o estado com um defeito).

Este trabalho está organizado da seguinte forma:

No Capítulo 2, apresentamos os dois classificadores propostos neste trabalho, bem como o framework utilizado pelos classificadores. Detalhamos a diferenciação de código-fonte, a instrumentação do código e a comparação de traces.

No Capítulo 3, discutimos as limitações conceituais e de implementação das técnicas apresentadas neste trabalho.

No Capítulo 4, apresentamos o experimento realizado e discutimos os resultados obtidos.

No Capítulo 5, apresentamos as conclusões do trabalho, bem como trabalhos relacionados e trabalhos futuros.

[1] Por classificador manual entendemos qualquer classificador que tenha sido manualmente escrito por um programador para, através de asserções, reportar se um teste passa ou não. Um teste do Junit é um exemplo de teste que utiliza um classificador manual.

2. CLASSIFICADORES PROPOSTOS

Os classificadores que este trabalho propõe recebem um par de *traces* (um para cada versão do programa, resultantes da execução de sequências de teste sobre as versões) como entrada e reportam como saída um veredito, indicando se há ou não diferença de comportamento entre as versões. O *trace* da execução de um teste sobre um programa é uma sequência de observações (por exemplo, estados visitados ou branches tomadas).

A Figura 1 destaca os principais passos do framework utilizado pelos classificadores.

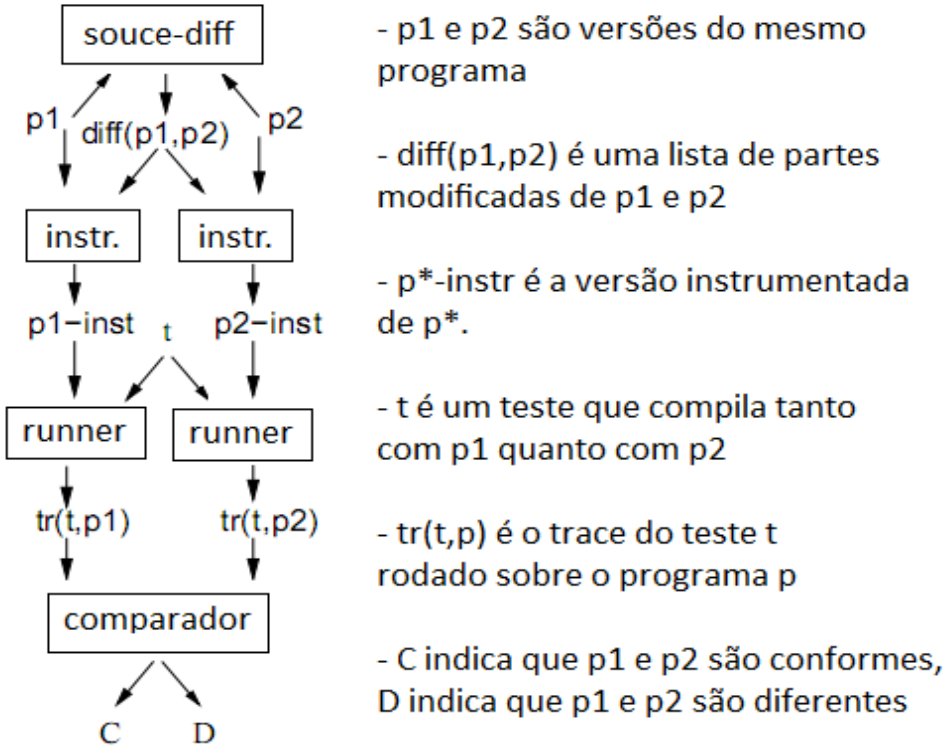

Figura 1 – Entradas e saídas dos principais passos do framework

Primeiramente, um algoritmo de diferenciação de código-fonte compara duas versões de um programa e identifica que partes mudaram. Em seguida,

um programa instrumentador lê a lista de partes modificadas e altera o código original das duas versões para coletar observações (gerar o *trace*) durante a execução do teste. Após a instrumentação, o mesmo teste é executado sobre cada versão instrumentada do programa. Por último, um comparador de *traces* verifica se há ou não conformidade entre os traces.

Este trabalho propõe duas instâncias desse framework: classificador por diferenciação (i) de estado e (ii) de branch. O primeiro classificador reporta possíveis mudanças de comportamento observadas a partir dos efeitos no estado do programa. O segundo reporta reais mudanças de comportamento observadas a partir dos efeitos no fluxo de controle do programa.

Na Seção 2.1, discutimos a diferenciação de código-fonte. Nas seções 2.2 e 2.3, discutimos o processo de instrumentação. Na Seção 3.4, discutimos a diferenciação dos *traces*.

2.1. Diferenciação de código-fonte

Neste passo, o framework identifica os elementos do código-fonte que foram modificados desde a última versão. O algoritmo recebe o código-fonte de duas versões e retorna uma lista dos métodos modificados (corpo do método e/ou assinatura), adicionados e removidos. Um caso especial é aquele dos métodos que sobrescrevem outro numa hierarquia de classes. Por exemplo, considere que um programador remove um método que sobrescreve outro e que um teste chame transitivamente[2] esse método. Na versão nova, a execução irá dinamicamente repassar a chamada para o outro método que estiver acima na hierarquia. Por esse motivo, o algoritmo considera todos os métodos da hierarquia com a mesma assinatura de um método modificado como modificados também. Para comparar o corpo do método nas duas versões, o algoritmo considera dois métodos iguais se as impressões de suas

[2] Um método *m* chama um método *n* transitivamente caso exista um caminho no grafo de chamadas do programa que leve de *m* a *n*.

árvores sintáticas abstratas forem iguais. Para isso, adaptamos um parser escrito em JavaCC [6].

2.2. Instrumentação (classificador de estado)

No contexto do classificador de estado, duas questões são levantadas com relação à intrumentação: (i) onde adicionar instruções para capturar o estado durante a execução e (ii) que partes do estado capturar, ou seja, que partes do estado global podem potencialmente revelar uma diferença.

Com relação à primeira questão, a instrumentação captura o estado no retorno de métodos modificados. O princípio é que mudanças de estado são oriundas de mudanças de código e podem somente ser observadas se elas são propagadas através de objetos ainda visíveis após a saída de um método modificado. Além disso, a instrumentação ignora chamadas a métodos modificados que ocorram dentro do contexto de um outro método modificado. Por exemplo, seja *m* um método qualquer. Se *m* chama o método não modificado *n*, que por sua vez chama o método modificado *o*, então a chamada ao método modificado *o* só será instrumentada caso *m* não seja um método modificado. O motivo para isso é que não podemos comparar os estados imediatamente posteriores às chamadas ao método *o* se os estados imediatamente anteriores potencialmente já diferem, uma vez que o método *m* foi modificado.

Essa abordagem diminui o número de falso-positivos, já que, continuando no exemplo acima em que m é um método modificado, se o estado imediatamente posterior à chamada de *o* for diferente nas duas versões isso não implica que haja uma diferença de comportamento, uma vez que o estado imediatamente posterior à chamada de m pode ser igual nas duas versões. Devemos ressaltar que o foco deste trabalho é encontrar diferenças de integração, logo, podemos desconsiderar diferenças locais que não se propagam para outros módulos do programa. A abordagem diminui ainda o

número de instruções adicionadas ao código instrumentado e, consequentemente, aumenta o desempenho do classificador.

A Figura 2 ilustra a pilha de chamada gerada pela execução do código instrumentado. A figura mostra um cenário em que o método modificado *A* chama transitivamente um outro método também modificado *B*. Esses são os únicos métodos modificados na pilha. Observe que o estado não é capturado quando a execução retorna do método modificado *B*, uma vez que ele foi chamado no contexto de execução do método modificado *A*. Assim, um *snapshot* do estado é capturado apenas no retorno do método modificado A.

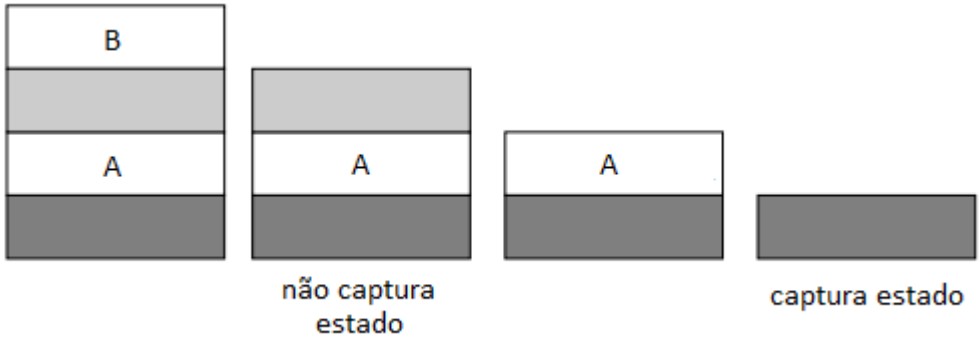

Figura 2 – Pilha de chamada de métodos modificados

Com relação à segunda questão, a instrumentação considera apenas o estado que é visível tanto para o método modificado quanto para o método que o chama. Isso inclui todo o estado alcançável pelo objeto que chama o método, parâmetros, retorno e campos estáticos.

Representação do estado

O classificador representa o estado com uma *hash* de 64 bits. Dois estados isomórficos produzem a mesma *hash*. Dizemos que dois estados são isomórficos caso os grafos de objetos que representam esses estados tenham a mesma estrutura (mesmo que tenham *ids* diferentes de objetos) e os

mesmos valores primitivos associados aos nós. Para produzir essa *hash*, computamos uma representação linear do estado com um array de inteiros (processo normalmente chamado de marshalling, serialização o linearização) e usamos a função de *hash* Jenkins [7] para computar um identificador único para o grafo de objetos. A Figura 3 exibe um pseudocódigo para o procedimento de linearização que utilizamos.

```
public class Linearizer {
  static DynamicIntArray seq;
  static Map ids;
  static final int NULL = -1;

  public static int[] linRoots(Object[] roots) {
    dyn = new DynamicIntArray();
    ids = new Map();
    foreach Object o in roots linObject(o);
    return dyn.toArray();
  }
  private void linObject(Object o) {
    if (o == null) id = NULL;
    else if (o in ids.keys()) id = ids.get(o);
    else id = ids.size();
    seq.add(id);
    if (id == ids.size()) { /* not seen */
      ids.put(o, id);
      foreach Field f in o.getField() {
        linObject(o.getField(f));
      }
    }
  }
}
```

Figura 3 – Procedimento de linearização

Dynamic binding

Se um método adicionado ou removido é alcançável, ele deve ser chamado por algum outro método no programa. Métodos que chamam um método adicionado ou removido serão geralmente métodos modificados. No entanto, isso não ocorre em duas situações: (i) sobrescrita de métodos e (ii) refinamento de métodos. No primeiro caso, se o método adicionado ou

removido sobrescrever outro, pode não haver nenhuma mudança de código em outras partes do programa, mas haverá mudanças dinâmicas devido ao *dynamic binding*. Por esse motivo, nós também capturamos o estado no retorno dos métodos-pai (métodos sobrescritos) do método adicionado ou removido. No segundo caso, o programador substitui o tipo de um parâmetro do método por um supertipo ou o retorno no método por um subtipo. No momento, nossa implementação não dá suporte a esse caso.

Implementação

Usamos a linguagem orientada a aspectos AspectJ [8] como ferramenta de instrumentação. Um aspecto é uma construção de modularização suportada por extensões de algumas linguagens populares, como C e Java [9]. Um aspecto consiste de *pointcut*(*s*) e *advice*(*s*). Um *pointcut* define vários pontos de instrumentação – *join points* na terminologia de aspectos – e um advice define o que fazer quando a execução alcançar esses pontos. A principal aplicação de aspectos é na separação interesses transversais, que interceptam diversos módulos de uma aplicação. O uso de aspectos pode, por exemplo, diminuir o espalhamento (quando um único interesse aparece em várias unidades do código) e o entrelaçamento (quando vários interesses aparecem na mesma unidade do código). O principal motivo de usarmos aspectos foi sua interface declarativa para instrumentação de programas, o que facilita diferentes configurações de experimentos.

A Figura 4 mostra o aspecto que o classificador de estado utiliza. O aspecto define um *pointcut* que intercepta qualquer chamada de método modificado desde que essa chamada não ocorra dentro do contexto de um método modificado. Os rótulos `m-sig_i` referem-se às assinaturas dos métodos modificados fornecidas pelo procedimento de diferenciação de código. O construtor `call` captura as chamadas aos métodos com as assinaturas especificadas. O construtor `cflowbelow(pcut)` intercepta todos os *join points* que ocorram entre a entrada e a saída do *join point p* capturado por `pcut`, exceto o próprio p. Por exemplo, considere a cadeia de chamada de métodos *a()* → *b()* → *c()* → *d()* → ..., sendo os métodos *b* e *d* métodos

modificados. As chamadas a b e d pertencem ao pointcut call. A chamada a *d* pertence ao pointcut `cflowbelow(call)`, já que ela ocorre no contexto da chamada a *b*. Portanto, apenas *b* é parte do joinpoint resultante. O advice declarado processa os dados depois do retorno de chamadas que `pcut` intercepta. Nós omitimos do aspecto o processamento de exceções por simplificação.

O corpo do advice computa uma representação linear dos estados do retorno do método (`null` em caso de `void`), o objeto pelo qual é chamado o método (null em caso de método estático), parâmetros e campos estáticos. Em seguida, o advice computa o valor do *hash*, como descrito anteriormente e escreve essa informação em um arquivo. A princípio, podemos reportar também o número da linha e o método que chama o método modificado.

```
public aspect StateSpectra {
  Object[] staticFields = /* static fields */;

  // definition of instrumentation points
  pointcut pcut() :
    (call (m-sig_1) || ... || call (m-sig_n)) &&
   !(cflowbelow((call (m-sig_1) || ... )));

  // advice defines what to do on instr. points
  after(): returning (Object ret) pcut() {
    List roots = new ArrayList();
    roots.add(ret); /* adds result */
    roots.add(thisJoinPoint.getTarget()); /* recv */
    roots.add(thisJoinPoint.getArgs()); /* params */
    roots.add(staticFields); /* static fields */
    log(hash(Linearizer.linRoots(linRoots)));
  }
}
```

Figure 4 – Instrumentação do classificador de estado

Otimização

Várias chamadas a métodos modificadas podem ocorrer dentro do contexto de loops, aumentando significativamente o tempo de execução do classificador. Por esse motivo, modificamos a instrumentação para capturar o estado apenas após a última chamada a um método modificado numa sequência de chamadas a esse método. Para implementar essa otimização, adicionamos um elemento no aspecto que sempre armazena o último estado visitado, bem como o último método, sem, no entanto, realizar a linearização. A Figura 5 mostra esta otimização. Para simplificar, omitimos o tratamento da última observação, já descrito na Figura 4.

```
public aspect StateSpectraLoopOptimized {
  ...
  List behindRoots;  Method behindMeth;
  after(): returning (Object ret) pcut() {
    Method thisMeth = thisJoinPoint.getSignature();
    if (behindMeth isNotNull &&
          thisMeth isDiff behindMeth) {
      log(hash(Linearizer.linRoots(behindRoots)));
    }
    List thisRoots = new ArrayList();
    // add objects to roots
    ...
    behindRoots = thisRoots;
    behindMeth = thisMeth;
  }
}
```

Figura 5 – Instrumentação otimizada com abstração de loop

2.3. Instrumentação (classificador de branch)

A diferenciação de estado reporta diferenças potenciais de comportamento causadas por uma mudança real de estado observada. Um trace da diferenciação de estado contém observações do estado ao final da

chamada de métodos com corpo modificado. A diferenciação de branch, no entanto, opera de uma maneira diferente.

Conceitualmente, a diferenciação de branch procura capturar diferenças reais (em vez de potenciais) de comportamento, já que ela reporta mudanças no fluxo de controle decorrentes de mudanças no código. Desta forma, um trace da diferenciação de branch contém observações do fluxo de controle de fora das regiões de código modificadas.

A instrumentação da diferenciação de branch acontece em dois passos. O primeiro passo consiste em instrumentar cada ponto de branch do programa. Por exemplo, a instrumentação traduz o fragmento de programa `if(`*e*`){...} else{...}` para `if(`*e*`){•;...} else{••;...}`, onde • refere-se ao statement `log.branch(true)` e •• refere-se ao statement `log.branch(false)`. O segundo passo consiste em desabilitar a instrumentação de branch se ela ocorre no contexto da chamada a um método modificado. Essa abordagem é similar à que foi usada na diferenciação de estado. Ela ignora observações originadas em código modificado.

Implementação

A instrumentação de branch é implementada com um visitor simples [10] em uma árvore sintática abstrata do programa. A nossa implementação do classificador de branch tomou como base a implementação descrita em [11]. Estendemos essa implementação com a adição do aspecto da Figura 6, que modifica o programa para parar a geração de *trace* no contexto de métodos modificados. Da mesma forma que o aspecto da Figura 4, esse aspecto obtém uma lista dos métodos modificados (`m-sig_i`) obtida pelo procedimento de diferenciação de código-fonte descrito anteriormente. Usamos duas linguagens de instrumentação porque AspectJ não dá suporte a *pointcuts* para instruções de branch.

```
public aspect BranchSpectra {
  pointcut pcut() :
    (call(m-sig_1) || ... || call(m-sig_n)) &&
   !(cflowbelow((call(m-sig_1) || ... )));

  before() : pcut(){ log.disable(); }
  after() : pcut(){ log.enable(); }
}
```

Figura 6 – Aspecto de instrumentação de branch

Optamos por abstrair do *trace* as decisões de branch que ocorrem dentro do contexto de métodos modificados pelo mesmo motivo que a instrumentação de estado não captura o estado dentro do contexto de métodos modificados. Partimos do princípio de que é natural que áreas do código chamadas por código modificado possam ter decisões de branch diferentes nas duas versões sem que isso seja um defeito no contexto de testes de integração.

2.4. Comparação de *trace*

Dados dois traces da forma $tr_{t, v1} = \sigma_{v1,1} \ldots \sigma_{v1,n}$ e $tr_{t, v2} = \sigma_{v2, 1} \ldots \sigma_{v2,n}$ resultantes da execução de um teste *t* sobre as versões *v1* e *v2* do programa, o classificador realiza uma comparação entre $tr_{t, v1}$ e $tr_{t, v2}$ para determinar se há diferença de comportamento entre as versões. O classificador reporta uma diferença de comportamento quando ele encontra algum $i \leq n$ para o qual $\sigma_{v1,i} <> \sigma_{v2, i}$. É importante que o instrumentador insira instruções de observação de forma consistente nas duas versões: o classificador só pode comparar observações geradas a partir dos mesmos pontos do programa.

3. LIMITAÇÕES

Destacamos abaixo limitações conceituais e de implementação dos classificadores.

Diferenças potenciais e reais. A versão atual do classificador de estado reporta uma diferença assim que ele observa uma diferença no estado que possa ser propagado para outras partes do programa comum a ambas as versões. Partimos do princípio de que o estado diferente pode ser lido no teste que reporta a possível diferença ou em outro teste semelhante. No entanto, é possível que esse estado possa não ser lido em nenhuma execução real do programa.

Mudanças fora do corpo dos métodos. Atualmente os classificadores consideram mudanças em corpo de métodos. Mas observe que é possível introduzir bugs com a inicialização incorreta de campos, declaração de *import* e diversas outras mudanças fora dos corpos dos métodos. Nesses casos, o classificador só poderá observar a diferença caso os efeitos se propaguem através de um método modificado.

Nenhum efeito. Considere o caso em que o programador introduz um defeito mudando a expressão `x + y` para `x - y`. Considere também que a parte do programa que contém essa mudança só é alcançável com um teste cuja execução armazene 0 na variável `y` no ponto da avaliação da expressão. Como a avaliação produz o mesmo valor, nenhum efeito pode ser observado.

Referências de objeto. O classificador de estado compara apenas isomorfismo de uma projeção do estado global, sendo essa projeção o estado que é alcançável na entrada e na saída da chamada ao método. Considere, por exemplo, que o programador mude o *statement* de retorno de um método que recebe o como argumento de `return o` para `return o.clone()`. Como o valor da primeira expressão é o mesmo do valor da segunda expressão, o classificador de estado não vai reportar a diferença.

4. AVALIAÇÃO EXPERIMENTAL

Este capítulo descreve a avaliação experimental. Na Seção 4.1, descrevemos o objeto do estudo. Na Seção 4.2, detalhamos as métricas utilizadas nos experimentos. Na Seção 4.3, descrevemos a configuração do experimento. Na Seção 4.4, apresentamos os resultados obtidos.

4.1. Objeto do estudo

Usamos o Jtopas [12], um parser open-source para textos arbitrários. Utilizamos várias versões corretas e defeituosas disponíveis no Software-Artifact Infrastructure Repository (SIR) [5]. O SIR dá suporte à injeção de defeitos artificiais no código, ou seja, defeitos intencionalmente criados para causar falha no software. Além das diferentes versões do programa, também utilizamos as sequências de teste disponíveis no SIR com classificador manual (asserções programadas manualmente).

4.2. Métricas

Nos experimentos analisamos o número de testes que caíram em cada uma das seguintes categorias: P-D, F-D, P-C e F-C. Os rótulos P e F indicam, respectivamente, passa e falha no classificador manual. Os rótulos D e C indicam, respectivamente, diferença e conformidade nos classificadores de estado e de branch. Por exemplo, P-D identifica execuções de teste que passam (P) no classificador manual, mas o classificador que propomos revela uma diferença de comportamento (D).

Também consideramos nos experimentos os percentuais de testes em cada categoria que exercitam um defeito (ou seja, infestam o estado do programa com um erro). Este número pode ser comparado com a soma das

categorias F-D e F-C que mostram o número de testes que originalmente falham com o classificador manual.

Para medir eficiência de tempo e espaço, consideramos o tempo necessário para executar a suíte de teste sobre o programa instrumentado e o tamanho do trace por teste.

4.3. Setup

Conduzimos dois tipos de experimentos de acordo com a organização das versões no repositório SIR.

A primeira configuração verifica a conformidade da versão *n* com relação à versão *n* + *1* com a injeção de algum defeito. Nós usamos testes da versão *n* (no Jtopas, esses testes compilam em ambas as versões). Neste setup, a versão *n* + *1* tem uma quantidade significativa de mudanças de código com relação à versão *n*. No entanto, apenas em alguns casos os testes da versão *n* exercitam um defeito em *n* + *1*. Isso ocorre pois um defeito pode ocorrer em partes novas da versão *n* + *1* as quais nenhum teste na suíte *n* cobre. Usamos o rótulo “v(n),v(n+1)F*” para identificar este setup experimental. Este setup simula um cenário em que o programador roda uma suíte de regressão depois de mudanças no código sem aumentar a suíte com novos cenários de teste.

A segunda configuração verifica a conformidade da versão *n* com relação à versão *n* com a injeção de algum defeito. Neste setup existe uma quantidade bastante reduzida de mudanças entre as duas versões. No entanto, os testes visitam com mais frequência as mudanças comparando-se ao setup anterior, uma vez que a suíte de teste corresponde à versão modificada. Usamos o rótulo “v(n),v(n)F*” para identificar este setup.

O setup “v(n),v(n+1)F*” considera um total de 37 versões defeituosas em meio a 4 versões corretas A, B, C e D. As versões defeituosas se distribuem da seguinte forma: 10 Bs, 12 Cs e 15 Ds. O setup “v(n),v(n)F*” considera um total

de 36 versões defeituosas em meio ao mesmo número de versões corretas. A distribuição de versões defeituosas é a seguinte: 10 Bs, 12 Cs e 14 Ds.

Em ambos os setups, executamos as sequências de teste 3 vezes para cada versão – uma com o classificador manual disponível no SIR, outra com o classificador de estado e outra com o classificador de branch.

4.4. Resultados obtidos

A Tabela I classifica as execuções de teste de acordo com as saídas (vereditos) dos classificadores.

Tabela 1 – Resultados dos classificadores

setup	verdicts	#tests		% tests infected	
		state	cflow	state	cflow
v(n),v(n+1)F*	P-D	505	16	**29.90**	100.00
	F-D	26	20	100.00	100.00
	P-C	1307 (2447)	4243	7.27	5.51
	F-C	0 (1)	7	NaN	100.00
v(n),v(n)F*	P-D	317	189	100.00	99.47
	F-D	85	77	100.00	100.00
	P-C	526 (4424)	5078	12.45	8.27
	F-C	0 (6)	14	NaN	100.00

A coluna “setup” se refere ao setup experimental. Usamos os rótulos “v(n),v(n+1)F*” e “v(n),v(n)F*” para indicar os setups explicados anteriormente. A coluna “vereditos” relaciona as saídas do classificador manual com os classificadores automáticos propostos. Os resultados foram sumarizados nos 4 grupos mencionados anteriormente: “P-D”, “P-C”, “F-D” e “F-C”. A coluna “# tests” mostra o número de testes em cada um desses grupos para os classificadores de estado e de branch (cflow). A coluna “% infected” mostra o número percentual de testes que exercitam defeitos. O repositório SIR documenta a localização de cada defeito. Nós manualmente adicionamos

instruções nas versões defeituosas para logar a execução do código defeituoso. Note que indicamos entre parêntesis a quantidade de execuções de testes que não geram nenhum dado de trace. Essas execuções não puderam ser avaliadas pelo classificador, pois nenhum método modificado foi chamado pelas sequências de teste. Isso é improvável de acontecer no classificador de branch já que todas as partes do programa, exceto as partes modificadas, geram dados de trace.

A Tabela II exibe a saída de cada classificador para cada versão.

Tabela 2 – Quantidade de testes que revelam defeitos por classificador por setup

setup	conf.	# fault-revealing tests			# infected
		manual	state	cflow	
v(n),v(n+1)F*	B06	1	1	17	17
	B07	0	**93**	0	95
	B08	0	**42**	0	95
	C01	**1**	0	0	1
	C02	0	**1**	0	1
	C03	6	6	0	6
	D04	1	16	1	44
v(n),v(n)F*	B01	1	2	1	3
	B02	1	1	2	3
	B03	1	1	2	2
	B05	1	18	18	18
	B06	1	15	31	31
	B09	1	2	3	3
	B10	1	3	1	5
	C01	1	2	1	3
	C03	1	10	1	10
	D04	1	16	1	44
	D06	1	11	11	14
	D07	1	58	58	59
	D08	1	11	11	12
	D09	1	54	54	55
	D12	0	**1**	0	1
	D13	1	67	64	69
	D14	0	63	0	69
	D15	0	63	0	69
	D16	1	0	3	3

A coluna “setup” indica o setup experimental e a coluna “conf.”, a configuração de versão usada nesse setup. Por exemplo, o rótulo “B06” no setup “v(n),v(n+1)F*” indica o cenário em que se compara a versão correta A com a versão B com um defeito injetado (identificador do defeito 6). A coluna “# fault-revealing tests” exibe o número de sequências de teste que o classificador manual classifica corretamente como falha ou diferente (D), no caso dos classificadores de estado e branch. Considerarmos que o classificador reportou corretamente uma falha ou diferença apenas quando o teste exercita um defeito. A coluna “# infected” exibe o número de testes infectados em uma suíte.

A Tabela III sumariza o custo de classificação. As linhas “size” e “time” reportam os valores centrais de tendência das distribuições de tamanho por teste e tempo por suíte respectivamente. Cada célula reporta média ($\bar{x}$), mediana ($\hat{x}$) e desvio-padrão (σ). Para os valores de tempo, não consideramos tempo de compilação, apenas o tempo da execução das duas versões.

Tabela 3 – Tempo de execução dos classificadores (em segundos) e tamanho dos *traces* (quantidade de observações)

		setup	
		v(n),v(n+1)F*	v(n),v(n)F*
size ($\bar{x}/\hat{x}/\sigma$) per test	state	0.5/0/0.7	0.2/0/3.1
	cflow (x10^4)	20.6/10.6/30.6	24.2/12.5/41.0
time ($\bar{x}/\hat{x}/\sigma$) per suite	orig	2.2/1.6/1.1	3.3/1.6/2.4
	state	11.7/10.6/4.7	12.6/10.8/6.3
	cflow	32.1/24.4/22.0	26.9/28.1/10.2

Discussão

A partir da Tabela I, podemos notar que os classificadores que propomos têm o objetivo de revelar diferenças de comportamento. Algumas diferenças são intencionais, já outras não são (por exemplo, erros). Em princípio, as diferenças reportadas que não decorrem em erros ou são intencionais ou são uma limitação de nossa abordagem. De qualquer forma, o percentual de testes

classificados com o rótulo “D” que são infectados por defeitos é bastante alto, exceto para o classificador de estado no primeiro setup (em negrito).

É importante notar que nem sempre que o classificador manual reporta uma falha (F) os classificadores propostos também reportarão. No caso da diferenciação por estado, objetos com o mesmo valor mas com referências diferentes não podem ser diferenciados pelo nosso classificador de estado, enquanto que o classificador manual pode fazer essa diferenciação através do operador de igualdade de referência (por exemplo, o operador == na linguagem Java). No caso da diferenciaçao por branch, é possível que um defeito capturado por uma asserção no classificador manual não seja detectado pelo classificador de branch, uma vez que esse defeito não leve a mudança alguma no fluxo de controle.

Na Tabela II, observa-se a eficiência relativa de cada classificador. A tabela exibe o número de testes em que cada classificador detecta uma falha. A tabela apenas mostra os casos em que ao menos um classificador detecta uma falha (em 38 casos a versão era defeituosa, mas nenhum classificador foi capaz de revelar o defeito com as sequências de teste disponíveis, já que nenhuma delas exercita a falha; em 4 casos a versão era defeituosa e nenhum classificador foi capaz de revelar o defeito, mesmo havendo ao menos um teste que exercitasse a falha).

A partir da tabela, podemos ver que em 6 casos o classificador manual deixou de reportar uma falha, contra 2 casos do classificador de estados e 8 casos do classificador de branch.

Discutiremos abaixo alguns casos relevantes, identificados na tabela com o rótulo “*”.

Na versão C01, tanto o classificador de estado quanto o de branch deixam de reportar o defeito. Após inspeção manual desse caso, percebemos que isso ocorreu pois uma asserção do classificador manual interrompeu a execução do teste antes que os classificadores propostos identificassem a diferença. A asserção verifica a igualdade de referências de objetos. Sem a interrupção ambos os classificadores teriam detectado a diferença.

Na versão D12, a modificação de uma condição de branch resulta, para certos valores de entrada, em uma atribuição errada do valor de uma variável. Apenas uma sequência de teste leva a execução para o defeito. A diferenciação por estado revela o defeito. Como nenhuma asserção verifica a corretude do estado, o classificador manual não é capaz de revelar o erro. Depois de o estado se tornar infectado, nenhuma decisão de branch acessa a variável defeituosa. Dessa forma, o classificador de branch também não identifica a diferença.

Na versão D16, os classificadores manual e de branch identificam o defeito, diferentemente do classificador de estado. A mudança consistiu da transformação de um campo em estático. Várias partes do programa lêem e escrevem nesse campo. O classificador manual lê o valor desse campo usando uma instância específica da classe que contém esse campo e encontra um valor diferente do esperado quando outras instâncias modificam o campo estático. Essa mudança resulta em uma mudança no fluxo de controle. No entanto, não existiram métodos modificados cujos parâmetros ou retorno pudessem alcançar esse campo. Dessa forma, o classificador de estado deixou de identificar esse defeito.

A partir da Tabela III, podemos notar que o classificador de estado requer menos tempo e espaço que o classificador de branch. O custo de uma observação deveria ser, a princípio, maior para o classificador de estado (o processo de linearização precisa atravessar o grafo de objetos inteiro que tanto o método chamador quanto o método chamado podem acessar), mas a diferenciação por estado somente captura o estado em retornos de chamadas a métodos modificados. O custo de uma observação para a diferenciação de branch, por outro lado, é bastante baixo, já que só é necessário armazenar um bit de informação[3]. No entanto, o classificador captura as decisões ao longo de toda a execução fora do contexto dos métodos modificados. Daí o tamanho dos traces de branch ser em média quatro ordens de grandeza maior que o tamanho dos traces de estado.

[3] Para cada decisão de branch visitada, o *trace* apenas precisa armazenar se a execução tomou um caminho ou outro, ou seja, se a expressão booleana foi avaliada como verdadeira ou falsa.

Os resultados indicam que tanto o classificador de estado quanto o classificador de branch podem ajudar a detectar diferenças de integração em versões de um programa a partir de sequências de teste que exercitem as mudanças. A tabela II mostra que esses classificadores revelam diferenças em mais sequências que o classificador manual, o que diminui as chances de um erro deixar de ser detectado.

No entanto, é possível que os classificadores propostos reportem um alarme falso. No caso do classificador de estado isso acontece em quase 70% dos casos (casos em que o classificador de estado detecta uma diferença, mas o teste não está infectado). Essa alta taxa de alarme falso se deve ao fato de as sequências de teste disponíveis não terem sido construídas com o foco na integração do código modificado. Como muitas vezes o teste chama diretamente o método modificado, o estado imediatamente posterior à chamada é diferente. No entanto, isso não quer necessariamente dizer que seja um defeito.

É importante observar também que em 52% dos casos (38 de 73) nenhuma sequência de teste exercita o defeito. Esse resultado, em conjunto com o que foi descrito no parágrafo anterior, confirma que expansão de súites de teste (*test suite augmentation)* é importante para a evolução de software [4], principalmente levando-se em conta as modificações efetuadas no código.

5. CONCLUSÃO

Este trabalho propôs classificadores que utilizam duas versões de um programa para identificar efeitos colaterais resultantes das mudanças de código. Os classificadores recebem como entrada uma sequência de teste, além das duas versões do programa, e retornam como saída um veredito, indicando se há ou não diferença de comportamento.

A principal aplicação dos classificadores é a automação do teste de integração. Desta forma, mudanças de comportamento nas regiões de código modificado são desconsideradas (abstraídas). São levadas em conta as diferenças de comportamento que ocorrem em outros módulos do programa.

O classificador de estado reporta potenciais mudanças de comportamento entre duas versões, já que o estado alterado mesmo que lido pode não modificar a saída do programa. O classificador de branch, por sua vez, reporta reais mudanças de comportamento ao relatar alterações no fluxo de controle.

Os resultados dos experimentos realizados demonstraram que em apenas 1 dos 31 casos estudados ambos os classificadores deixaram de reportar um erro que o classificador manual reportou. Em todos os 6 casos em que havia testes infectados e o classificador manual deixou de reportar o erro, os classificadores propostos conseguiram reportar.

5.1. Trabalhos relacionados

Reps et al. [13] [14] são pioneiros no uso de *traces* de execução para verificação de conformidade. A diferenciação de branch exposta neste trabalho é semelhante à desses autores, com a diferença que omitimos a maior parte do trace ao realizarmos abstrações de mudanças.

Dois tipos de espectro foram propostos na literatura para verificação de conformidade: estrutural [13] [15] e valor [15] [12]. O espectro estrutural inclui

observações relacionadas à estrutura do programa, tais como expressão visitada, *statement, branch* e método. O espectro de valor inclui observações relacionadas a dados, tais como saídas do programa e estado. No geral, não é possível comparar o quão efetivos são os espectros estrutural e de valor.

Xie e Notkin [16] [17] propuseram um algoritmo conceitualmente semelhante ao que propomos neste trabalho para a diferenciação de estados. A principal diferença é que a abstração de mudanças permite que o classificador que propomos elimine partes dos *traces*, aumentando a precisão e a eficiência.

5.2. Trabalhos futuros

Os classificadores propostos neste trabalho podem ser integrados a um gerador automático de testes que foque em mudanças para gerar as sequências. Com testes automaticamente gerados especificamente para exercitar mudanças, a princípio os classificadores se tornam mais precisos.

Os classificadores podem ser transformados em uma extensão da ferramenta AQUA [18], um analisador dinâmico que implementa uma máquina virtual Java. Com o AQUA, é possível reduzir a granularidade de estruturas modificadas para branches ou instruções, em vez de métodos, como os classificadores estão atualmente implementados.

Os classificadores podem também ser aprimorados para servirem também como localizadores de falhas. Com algumas alterações de implementação é possível reportar não apenas que houve uma diferença de estado ou de brach tomada, mas em que linha de que classe isso ocorreu. Assim, os classificadores podem servir como ferramentas que auxiliem na depuração.

6. REFERÊNCIAS BIBLIOGRÁFICAS

[1] S. Person, M. Dwyer, S. Elbaum e C. Pasareanu, “Differencial symbolic execution”, in *Proc. Of Foundations of Software Engineering (FSE)*, 2008, pp. 226-237.

[2] A. Orso e T. Xie, “Bert: Behavioral regression testing”, in *Proc. Of the International Workshop on Dynamic Analysis (WODA),* 2008, pp. 36-42.

[3] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso e M. Harrold, “Matrix: Maintenance-oriented testing requirements identifier and examiner”, in *Proc. Of Testing: Academic and Industrial Conference – Practice and Research Techniques (TAIC-PART)*, 2006, pp. 137-146.

[4] R. Santelices, P. Chittimalli, T. Apiwattanapong, A. Orso e M. Harrold, “Test-suite augmentation for evolving software”, in *Proc. Of Automated Software Engineering (ASE)*, 2008, pp. 218-227

[5] Site do SIR: http://sir.unl.edu.

[6] Site do JavaCC: https://javacc.dev.java.net.

[7] B. Jenkins, “Algorithm alley: Hash functions”, *Dr. Dobbs Journal of Software Tools,* vol. 22, no. 9, pp. 183-200, 1997.

[8] Site do AspectJ: http://eclipse.org/aspectj/

[9] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier e J. Irwind, “Aspect-oriented programming”, in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997, pp. 220-242.

[10] E. Gamma, R. Helm, R. Johnson e J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[11] R. A. S. Oliveira e M. B. d’Amorim, “Uso de Espectros de Execução para Análise de Modificações em um Programa”, trabalho de conclusão do curso de graduação em Ciência da Computação. *Centro de Informática, Universidade Federal de Pernambuco (UFPE),* 2009.

[12] Site do Jtopas: http://jtopas.sourceforge.net.

[13] T. Ball e J. R. Larus, “Efficient path profiling” , in *Proc. Of the ACM/IEEE International Symposium on Microarchitecture,* 1996, pp. 46-57.

[14] T. Reps, T. Ball, M. Das e J. Larus, “The use of program profiling for software maintenance with applications to the year 2000 problem”, in *Proc. Of the European Software Engineering Conference and the Symposium on Foundations of Software Engineering (ESEC/FSE)*, 1997, pp. 432-449.

[15] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu e L. Yi, “An empirical investigation of the relationship between spectra differences and regression faults”, *Journal of Software Testing, Verification and Reliability*, vol. 10, no. 3, pp. 171-194, 2000.

[16] T. Xie e D. Notkin, “Checking inside the black box: Regression testing based on value spectra differences”, in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2004, pp. 28-37.

[17] T. Xie e D. Notkin, “Checking inside the black box: Regression testing by comparing value spectra”, *IEEE Transactions on Software Engineering (TSE)*, vol. 31, no. 10, pp. 869-883, 2005.

[18] Site do AQUA: http://www.cin.ufpe.br/~damorim/aqua/