GERAÇÃO AUTOMÁTICA DE DADOS DE TESTE

AUTOMATIC TEST DATA GENERATION

Carlos Miguel TOBAR Toledo*

ABSTRACT

The process for the elaboration and execution of tests has received, more and more, the importance that it deserves. Because of its nature, this process requires great quantities of effort and time, which result in its placement on a second level. The automation of this process can result in a significant reduction of effort and time requirements. This paper presents a number of techniques and tools which provide support for the generation of test data (some times test cases) and proposes a simplified classification for them.

KEY WORDS: Test, Test Data Generator

RESUMO

O processo de elaboração e execução de testes tem recebido, cada vez mais, a devida importância, porém, devido à sua natureza exige uma grande quantidade de esforço e de tempo, o que acaba resultando na sua colocação em segundo plano. A automatização desse processo pode contribuir significativamente para a redução de esforço e tempo. Este trabalho apresenta um levantamento de técnicas e ferramentas que auxiliam na geração de dados de teste (algumas vezes casos de teste) e propõe uma classificação simplificada para as mesmas.

PALAVRAS-CHAVE: Testes, Gerador Automático de Dados de Teste.

1. INTRODUÇÃO

A automatização do processo de testes pode levar à redução significativa de custos, pois os custos de testes correspondem a aproximadamente 50% do custo total do desenvolvimento de sistemas [4].

Uma das atividades que pode ser automatizada e corresponde a um dos mais importantes e difíceis aspectos do processo de testes, senão o mais importante e difícil, é a atividade de geração de dados de teste, ou seja, a atividade de identificar valores para as entradas do programa que satisfaçam determinado critério de teste.

A utilização de critérios de teste ocorre devido à impossibilidade, na maioria dos casos, de se realizar teste exaustivo. Idealmente, estabelece que a escolha de um subconjunto de entradas permite encontrar uma grande porção de defeitos e, portanto, é efetivo e permite um ganho de confiança em relação ao programa. Por outro lado, ao mesmo tempo, introduz a questão sobre a sua efetividade em testar.

Para sistemas reais, devido ao seu tamanho e complexidade consideráveis, a geração de um conjunto efetivo de dados de teste pode representar uma tarefa

(*) Coordenador dos Cursos de Especialização do II/PUCCAMP, Professor dos Cursos de Graduação em Análise de Sistemas e Engenharia de Computação do II/PUCCAMP, Mestre em Ciência da Computação pelo DCC-IMECC UNICAMP e Aluno de Doutorado em Engenharia de Computação e Automação Industrial no DCA-FEE UNICAMP. E-mail tobar@dca.fee.unicamp.br.

árdua envolvendo a análise de milhares de combinações, muitas vezes só realizável através da automação. Situação agravada se considerarmos mudanças nas especificações ou sua não corretude. Normalmente, para a realização de testes, assume-se que as especificações não apresentam problemas.

A ferramenta que auxilia na geração de dados de teste, seja completa ou parcialmente, é chamado TDG (Test Data Generator).

Foram pesquisados 6 diferentes artigos que descrevem TDGs. O objetivo desta pesquisa é o estudo de diferentes tipos de TDGs para identificar suas diferenças e propor uma classificação.

Existe uma série de conceitos que é necessária para o apropriado entendimento dos artigos pesquisados. Estes conceitos, via de regra, são discutidos nas primeiras partes dos artigos, e a terminologia envolvida, normalmente, varia de autor para autor.

ARTIGOS PESQUISADOS E COMENTADOS

Em seguida são apresentados, resumidamente, os artigos pesquisados na forma de comentários.

2. AUTOMATED GENERATION OF TESTCASE DATASETS [5]

O TDG está baseado em teste caixa branca, através de cobertura de estrutura de controle, de acordo com algum critério pré-estabelecido. A geração de dados de teste ocorre através da simulação de execução em sentido reverso (backtracking) de cada caminho em consideração:

. a identificação da estrutura de controle do programa, que é usada, por sua vez, para identificar uma coleção de padrões (caminhos), de acordo com um critério prestabelecido, e que permita exercitar o programa de maneira ampla;

. a escolha de valores do domínio de entrada do programa para exercitar o programa da maneira pretendida; este processo envolve a referência específica ao fonte do programa;

. a análise de dados de testes; após a identificação de variados conjuntos de caminhos, estes podem ser associados às especificações funcionais do programa.Cada caminho representa uma instanciação por dados de uma condição formal de verificação do programa.

Concluindo, o trabalho não aborda diversos pontos importantes:

- como são tratados comandos repetitivos (loops) com número constante de iterações?
- como são tratadas condições compostas com operadores and, or e not?
- como são tratados caminhos infactíveis?
- como são tratados elementos de matrizes e registros, além de variáveis dinâmicas (ponteiros)?
- como são identificados os caminhos relativos a um critério? e
- mais fortemente, como se associam caminhos às especificações funcionais (mesmo que formais)?

3. TEST PLAN GENERATION USING FORMAL GRAMMARS [1]

O TDG em questão, é mais que um TDG, é um gerador de casos de teste, baseado em teste caixa preta, através de uma gramática formal que representa um autômata finito extendido (fsa). Este TDG é parte de um sistema maior que permite a execução dos casos de teste gerados.

O motivo da maioria do geradores se basearem na estrutura do programa (caixa branca) é a falta de técnicas formais para especificação de comportamento. Existem, no entanto, alguns tipos de sistemas que podem ser especificados através de gramáticas formais.

Dada a especificação do comportamento observável de um sistema, que pode ser modelado por um fsa, a geração de um conjunto de seqüências de testes ocorrerá através da análise da especificação e definição de seqüências de estímulos. Estes estímulos são, posteriormente, usados em uma simulação do automata para a definição dos resultados esperados.

A especificação é feita via uma descrição formal, que é usada como entrada por um Processador de Linguagem de Requisitos (RLP). O RLP é um compilador direcionado por tabelas (que dependem da aplicação e pode ser "customizado") e gera um fsa extendido, que, por sua vez, é usado como entrada para um Gerador de Planos de Teste (TPG). O TPG é responsável pela geração de seqüências de dados de teste e resultados esperados, que são usados como entrada por um Executor Automático de Testes (ATE). O ATE é o responsável pelos resultados dos testes.

O TPG produz um conjunto de scripts de testes executáveis. Cada script corresponde a uma seqüência de estímulos e das respostas esperadas.

Concluindo, o trabalho não aborda um ponto muito importante: como gerar aleatoriamente os scripts, a partir da gramática (autômata)?

Perceber que esta questão está relacionada diretamente à efetividade do teste, ou seja, quais e quantas combinações de entrada, das possíveis, devem ser escolhidas. Algo parecido com cobertura das especificações.

Reparar também que os casos de teste não têm condições de detectar partes implementadas que não são executadas (estão a mais). Para isso deveria ocorrer algum tipo de combinação com teste caixa branca.

4. AUTOMATIC GENERATION OF RANDOM SELF-CHECKING TEST CASES [2]

Os TDGs estão baseados em teste caixa preta de aplicações cujas entradas são especificações feitas em uma linguagem formal e geram dados de teste, isto é programas, de forma randômica. Estes programas-teste podem ser executados sobre o sistema que se quer testar e fazem com que o próprio sistema se auto verique, através de comparações parciais, de forma automática.

A motivação para a existência de TDGs Randômicos é a de que, existindo uma quantidade enorme de possibilidades para teste e uma vez definidos os dados de teste, estes apresentam "gaps" em relação à cobertura total das possibilidades. Mais que isso, a tendência é a manutenção desses dados de teste e, conseqüentemente, dos "gaps". Os TDGs Randômicos podem gerar dados de teste que permitem coberturas diferenciadas, cada vez que são executados. Note-se que essa motivação cria outro grave problema quando considera-se a importância dos testes de regressão.

Um TDG Randômico constitui uma ferramenta específica de cada sistema de software a ser testado.

A estratégia de geração randômica pode ser efetiva (não há comprovação), mas é contrária aos testes de regressão, que poderiam ser dispensados desde que a efetividade fosse comprovadamente alta. Novamente algo a haver com cobertura de especificações.

Reparar também que os casos de teste não têm condições de detectar partes implementadas que não são executadas (estão a mais). Para isso deveria ocorrer algum tipo de combinação com teste caixa branca.

5. AUTOMATED SOFTWARE TEST DATA GENERATION [4]

Os TDGs estão baseados em teste caixa branca e em execuções consecutivas do programa: para cada caminho (elemento requerido) realizam-se ajustes em um dado de teste gerado (inicialmente) de forma randômica, considerando fluxo de dados (definição e uso), até que o caminho seja percorrido; após cada ajuste acontece a execução parcial do programa.

TDGs orientados a caminhos têm como entrada o programa a ser testado e um critério de teste, a partir dos quais são gerados dados de teste que satisfaçam o critério.

Os TDGs orientados a caminhos realizam as seguintes operações básicas:

- construção do grafo de fluxo de controle do programa,
- seleção de caminhos que identifica o mais próximo conjunto de caminhos minimal que satisfaça o critério, e
- geração de dados de teste para cada caminho selecionado, de maneira a poder exercitá-lo.

Korel [4] apresenta uma abordagem alternativa, onde os dados de teste são desenvolvidos usando valores reais para as variáveis de entrada, denominada abordagem dinâmica, que está baseada em:

- execução real do programa,
- análise dinâmica de fluxo de dados, através de sua monitoração durante a execução do programa, e
- métodos de minimização de funções.

Durante o teste, se um fluxo de execução indesejável é observado em algum ramo, uma função com valores reais é associada ao ramo. Esta função resulta em um valor positivo quando o predicado do ramo é falso e em um valor negativo quando é verdadeiro.

Algoritmos de busca minimizada são usados para automaticamente localizar valores para as variáveis de entrada, que ocasionam o valor negativo para a função. Além disso, análise dinâmica de fluxo de dados é usada para determinar as variáveis de entrada, responsáveis pelo comportamento indesejável do programa, permitindo com isto o aumento de velocidade no processo de busca.

A abordagem dinâmica permite que matrizes e estruturas dinâmicas de dados possam ser manipuladas

adequadamente, pois durante a execução do programa todos os valores das variáveis, incluindo índices e ponteiros, são conhecidos.

Dado um caminho P, o objetivo do TDG é encontrar uma entrada x para o programa tal que P seja percorrido.

Concluindo, um ponto interessante para o qual não são apresentadas informações é a questão da seleção de caminhos, que de forma minimal, satisfaça o critério escolhido para o teste.

Salienta-se que esta proposta trata loops, elementos de matrizes, registros e variáveis dinâmicas.

A questão de condições compostas não é explicitada, mas é reconhecida como tratável.

6. CONSTRAINT-BASED AUTOMATIC TEST DATA GENERATION [3]

O TDG está baseado em teste caixa branca, através da produção de dados de teste que se aproximam da adequação relativa, ou seja, proporção de erros semeados em mutantes que são descobertos. O TDG é dito baseado em defeitos ou baseado em restrições.

Técnicas que escolhem dados de teste, para tentar mostrar a presença (ou ausência) de defeitos, são a origem para os TDGs baseados em defeitos, que são necessários devido à dificuldade inerente ao processo de geração de dados de teste.

Assume-se que o programa testado satisfaz suas especificações funcionais. Mais que isso, os operadores de mutação requerem explicitamente que os dados de teste satisfaçam critérios de cobertura de comandos e ramos, de valores extremos, perturbação de domínio e que modelem diretamente vários tipos de defeitos.

Uma das maneiras de gerar automaticamente dados de teste para matar mutantes é por meio do filtro de dados de teste não efetivos, o que pode ser feito através de restrições matemáticas, ou seja, através da solução de problemas algébricos.

As principais partes do TDG são o analisador de caminhos (AC), o gerador de restrições (GR) e satisfazedor de restrições (SR).

Para cada comando do programa original, o AC cria uma expressão/ restrição de caminho, tal que, se o dado de teste alcança o comando, a restrição será verdadeira. Idealmente seria interessante criar restrições para o inverso (se a restrição é satisfeita o comando será executado), porém isto é intratável.

Na prática, a satisfação de uma restrição de caminho garante a execução do comando alvo na ausência de loops.

O GR constrói as restrições de necessidade e de predicados e o SR pega cada restrição de necessidade, encontra o seu respectivo comando e faz a sua conjunção com a restrição de caminho. A solução dessa conjunção permite a geração do dado de teste. Se o dado de teste não pode ser gerado por algum motivo, a informação a respeito do motivo é apresentada ao testador.

Concluindo, tudo indica que os dados de teste gerados através de mutação são usados, posteriormente, para o teste efetivo do software, considerando que estes, por serem adequados ou relativamente adequados, são, potencialmente, capazes de detectar outros erros, que não os das mutações.

Falta à estratégia saber tratar loops e caminhos infactíveis.

Como nada se diz a respeito de como são produzidas as restrições de caminhos, nada se pode concluir quanto ao tratamento de elementos de matrizes, registros e variáveis dinâmicas.

A estratégia foi experimentalmente analisada quanto à sua efetividade e os dados parecem indicar que, apesar de uma escolha aleatória de valores em sub-domínios restritos, a adequação é superior a 90%.

7. AUTOMATICALLY GENERATING TEST DATA FROM A BOOLEAN SPECIFICATION [6]

O TDG está baseado em teste caixa branca, através da estratégia de impacto significativo de cada variável em uma especificação em formato de expressão booleana. A geração de casos de teste se faz através de um algoritmo que parcialmente soluciona a satisfatibilidade da expressão, mas também a sua insatisfatibilidade.

A base dos algoritmos é uma especificação em formato de expressão booleana, a qual é analisada para que cada uma de suas componentes possa impactar no resultado final, seja ele verdadeiro ou falso. Esta estratégia é não determinística.

A especificação é transformada na forma canônica disjuntiva (DNF canônica), ou seja, soma de produtos, onde cada produto contém uma instância de cada uma das variáveis participantes, seja negada ou não.

Um caso de teste é uma atribuição de valores verdadeiros ou falsos às variáveis da expressão (fórmula).

Uma instância de uma variável em um dos produtos é chamada de literal e tem um impacto significativo, em um determinado caso de teste, se todos os valores dos outros literais sendo os mesmos, o valor final da expressão depende do valor dessa variável.

A estratégia em questão envolve a seleção de casos de teste que demonstram o impacto significativo de cada variável nos possíveis valores da fórmula.

Um teste deve envolver pontos verdadeiros e pontos negativos, estes são chamados pontos se considerarmos o espaço das possíveis combinações de valores das variáveis da fórmula. Um ponto verdadeiro ocasiona a fórmula ser avaliada para verdadeiro e um ponto falso para falso.

A estratégia básica garante a detecção de alguns tipos de erros de implementação pois os pontos verdadeiros únicos garantem o resultado verdadeiro e os pontos falsos próximos garantem o falso. Ao mesmo tempo alguns outros tipos de erros podem não ser detectados ou não o serão.

Concluindo, esta estratégia preocupa-se com a efetividade dos casos de teste gerados.

Reparar também que os casos de teste não têm condições de detectar partes implementadas que não são executadas (estão a mais). Para isso deveria ocorrer algum tipo de combinação com teste caixa branca.

8. CONCLUSÕES

Segundo Korel [4] existem três diferentes tipos de TDGs:

- orientados a caminhos, com exemplo no próprio trabalho [4];
- de especificação de dados, com exemplo em [5] e [1];
- randômicos, com exemplo em [2].

Artigos apresentando as quatro referências estão resumidos na segunda parte da monografia, juntamente com os trabalhos de [3], apresentado como um TDG baseado em defeitos, e [6], apresentado como baseado em uma especificação lógica.

Notamos que os trabalhos [5] e [1] são bem diferentes, para poderem constituir um mesmo tipo de TDG, como proposto por Korel [4]. Há sim uma semelhança entre os trabalhos de [1] e de [6], pois os mesmos se baseiam em especificações formais.

Pode-se dizer que os trabalhos de [1], [2] e [6] apresentam abordagens caixa preta. Enquanto que [5], [4] e [3] apresentam a abordagem caixa branca.

Os trabalhos de [5] e [3] são semelhantes na medida em a geração de dados de teste é realizada a partir da solução de expressões algébricas. Porém a motivação e a forma de criação das expressões diferem nos dois trabalhos.

Outra semelhança que pode ser vista entre os trabalhos de [3] e [6] é o uso de expressões algébricas lógicas normalizadas, embora ressalte-se que o primeiro usa o programa fonte para produzir as expressões e a segunda usa as especificações.

Os trabalhos de [1] e de [2] baseiam-se em linguagens, através das quais é possível descrever seqüências de entradas para o programa que se quer testar. Mais que isso, ambos provêm, além de dados de teste, resultados esperados e executores automáticos de testes. As diferenças, no entanto, existem na medida em que em [1] o esquema é genérico e adaptável a diversos e diferentes tipos de sistemas, enquanto que em [2] os esquemas são específicos. Isto se deve ao próprio tipo de linguagem usado: em [1] a linguagem descreve um autômata finito e em [2] descreve especificamente uma linguagem de programação computacional de alto nível.

Assim, propomos uma classificação diferente para os TDGs apresentados nos artigos estudados. Esta classificação leva em consideração o tipo de entrada que cada TDG necessita e nada mais é que a classificação a respeito do tipo de teste utilizado como base (caixa preta ou caixa branca), assim temos:

- especificação formal;
- programa fonte do software a ser testado.

O primeiro tipo de TDG permite a geração de casos de teste e pode ter complexidade variada em relação ao tipo de formalismo usado:

- expressões booleanas como em [6] - a mais simples;
- linguagem de transição de estados como em [1] (autômata finito extendido) - complexidade média;
- linguagem genérica de programação como em [2] - a mais complexa.

O segundo tipo de TDG permite apenas a geração de dados de teste, devido à sua natureza de testador caixa branca. Explícita ou implicitamente requer um critério de cobertura:

- [5] e [4] requerem explicitamente o critério, dando flexibilidade ao testador na sua escolha e, indiretamente, na efetividade do resultado (considerando que cada tipo de cobertura tenha efetividade distinta, como proposto por Weyuker);
- [3] usa um critério de cobertura implícito, procura-se satisfazer o critério de morte de todos os mutantes não equivalentes ao programa original.

Ressalte-se que o trabalho de [4] recebe como entrada um critério de cobertura, que pode não ser baseado em fluxo de dados, porém utiliza fluxo de dados para melhorar o desempenho do processo de geração de dados de teste.

Existe um TDG caixa branca que se baseia em defeitos (mutação), porém não fica claro se os outros permitem a escolha de critérios de cobertura baseados em fluxo de dados ou, se apenas, fluxo de controle.

Os TDGs caixa branca orientam a geração do dado de teste em relação a cada elemento requerido pelo critério de cobertura, de forma a criar um predicado específico do elemento requerido.

Os TDGs estudados na sua maioria utilizam-se do recurso de simulação, quais sejam: os caixa branca ([5], [4] e [3] e os caixa preta complexos ([2]).

Todos os TDGs utilizam a geração randômica em graus variados:

- escolha sujeita a restrições como em [5];
- escolha livre, posteriormente adaptada, como em [4];
- escolha segundo um padrão de produção como em [1] e [2];
- escolha em um sub-domínio reduzido, devido à estratégia usada, como em [3] e [6].

Geração de dados de teste através de uma abordagem caixa branca representa o risco de: não se detectar a falta de funcionalidades e de apenas exercitar o software, sem validá-lo.

A utilização de TDGs, como de qualquer outra ferramenta automatizada, no processo de teste constitui mais uma variável a ser considerada na avaliação e depuração de resultados, diretamente relacionada com a corretude da própria ferramenta.

Alerta-se que os dados (ou casos) de teste obtidos a partir de TDGs não são suficientes para o teste. Há necessidade, de pelo menos, teste de unidade e de integração. Outros testes podem vir a ser necessários, caso se detecte pedaços não testados, para isso é necessário algum tipo de instrumentação para verificar a efetividade dos dados de teste gerados.

Existe ainda a necessidade de muita pesquisa e experimentação para a obtenção de TDGs genéricos ou que possam atender a qualquer classe de programas, todavia nota-se que para alguns tipos de programas a automatização do processo de testes é uma realidade.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] J. A. Bauer & A. B. Finger Test Plan Generation using Formal Grammars Proc. 4th. Int. Conf. on Soft. Eng'g, sep/79, pp. 425-432.

[2] D. L. Bird & C. U. MunoZ Automatic Generation of Random Self-Checking Test Cases IBM Sys. Journal, 22 (3), 1983, pp. 229-245.

[3] R. A. DeMillo & A. J. Offutt Constraint-Based Automatic Test Data Generation IEEE ToSE, 17 (9), sep/91, pp. 900-910.

[4] B. Korel Automated Software Test Data Generation IEEE ToSE, 16 (8), aug/90, pp. 870-879.

[5] E. F. Miller & R. A. Munoz Automated Generation of Testcase Datasets SIGPLAN Notices, 10 (6), jun/75, pp. 51-58.

[6] E. Weyuker, T. Goradia & A. Singh Automatically Generating Test Data from a Boolean Specification IEEE ToSE, 20 (5), may/94, pp. 353-363.