Uso de Modelos da UML em Testes de Componentes[1]

Ivan Rodolfo Duran Cruz Perez, Eliane Martins, Júlio Esslinger Viégas

Instituto de Computação – Universidade Estadual de Campinas (Unicamp)
Caixa Postal 6.176 – 13.083-970 – Campinas – SP – Brazil

ivan.perez@students.ic.unicamp.br, eliane@ic.unicamp.br,
ra061970@students.ic.unicamp.br

Abstract. *A software component must be tested every time that is reused, to guarantee the quality of both the component itself and the system in which it is to be integrated. Since the source code of the components is generally not available for its users, we propose a method to generate test cases from an UML Activity Diagram that representing the component behavior. Path testing techniques are used to automatically derive test cases. In this paper we describe the method and show an example to illustrate it. A tool developed to support the approach is also briefly presented.*

Resumo. *Um componente de software deve ser testado a cada vez que é reutilizado, para garantir a qualidade tanto do componente quanto do sistema no qual será integrado. Como geralmente o código fonte dos componentes não é disponível para seus usuários, nós propomos um método para gerar casos de teste a partir de Diagramas de Atividades UML que representam o comportamento do componente. Técnicas de testes de caminhos são usadas para derivar os casos de teste automaticamente. Neste artigo, nós descrevemos o método e mostramos um exemplo para ilustrar o uso dele. Uma ferramenta foi desenvolvida para dar suporte à abordagem e ela é brevemente apresentada*

1. Introdução

Um componente é uma unidade com interfaces especificadas através de contratos e com dependências de contexto explícitas [Szyperski 1998]. O desenvolvimento de software baseado em componentes é cada vez mais utilizado atualmente. Um de seus principais atrativos é a possibilidade de redução do tempo e custo de desenvolvimento, através da reutilização de código. A reutilização por si só não garante a qualidade do sistema, sendo necessário a da realização de testes nos componentes, a cada reutilização e a cada integração ou uso em um novo contexto [Weyuker 1998].

Para realizar testes em componentes, a abordagem mais adequada é a de testes baseados na especificação, pois o código fonte nem sempre é acessível. Além do que, os componentes podem ser heterogêneos, provenientes de diferentes fornecedores, desenvolvidos em diferentes linguagens de programação, o que dificulta os testes

[1] Este projeto é parcialmente financiado pelo projeto Harpia, um convênio firmado entre a Secretaria da Receita Federal do Brasil com as instituições Unicamp e ITA; e parcialmente financiado pelo projeto CompGov (Finep nº 1843/04).

baseados em seu código fonte.

Neste trabalho propomos um método de geração de casos de testes a partir da especificação do componente. A especificação considerada aqui é baseada em modelos comportamentais. Os modelos são bastante aceitos e têm sido cada vez mais utilizados em testes de *software*. Algumas das principais técnicas de Testes Baseados em Modelos podem ser encontradas em [Beizer 1990] e [Binder 2000].

Uma das principais vantagens da geração de testes a partir de modelos é permitir que a especificação e implementação dos casos de teste comecem mais cedo no ciclo de desenvolvimento. Os casos de teste podem ser criados assim que a especificação do componente estiver pronta. Além disso, o uso de modelos na especificação possibilita a automatização da geração dos casos de testes. Uma vantagem em automatizar os testes é que a geração manual de casos de testes é tediosa e propensa a erros e omissões, especialmente para componentes mais complexos.

Este trabalho dá continuidade àquele desenvolvido em [Rocha 2005] onde é definido um componente testável. Um componente testável possui duas especificações: uma de comportamento e outra de contrato. Para a especificação de comportamento, Rocha propôs o uso de diagramas de atividades para especificar as interfaces providas e requeridas dos componentes. Para a especificação de contrato, são utilizadas assertivas, conforme recomendado pelo método de projeto por contrato (design-by-contract) [Meyer 1997].

O método proposto aqui consiste na realização dos seguintes passos para geração dos casos de teste: (i) modelagem do comportamento do componente usando o Diagrama de Atividades (DA); (ii) conversão do DA em um grafo; (iii) seleção de caminhos no grafo; (iv) especificação dos casos de testes correspondentes aos caminhos selecionados; (v) identificação dos dados de entrada necessários para ativar os caminhos selecionados; (vi) implementação dos casos de testes na linguagem de programação escolhida.

Os passos (ii), (iii) e (iv) foram automatizados através da implementação de um protótipo para apoiar os testes. A identificação dos dados (v) e implementação dos casos de teste (vi), são realizadas manualmente, e não serão abordadas de maneira detalhada neste trabalho.

O restante do texto está organizado da seguinte maneira: a Seção 2 apresenta alguns trabalhos relacionados a testes de componentes e testes baseados em modelos; a Seção 3 descreve o DA utilizado para modelar o componente no passo (i), bem como o estudo de caso exemplo; a Seção 4 define o grafo obtido a partir do DA e mostra um exemplo do grafo; a Seção 5 mostra exemplos de seleção de caminhos e dos casos de testes obtidos (iii) e (iv). A Seção 6 apresenta os resultados da execução dos testes, conclusões e perspectivas futuras deste trabalho.

2. Trabalhos Relacionados

Nos testes baseados em modelos (model-based testing ou MBT), os testes são obtidos total ou parcialmente a partir de um modelo do sistema em teste. Um modelo descreve informações essenciais dos aspectos a serem testados e, além disso, um modelo é mais simples de analisar quando o sistema está sendo estudado [Binder 2000]. Um dos

objetivos da criação de modelos é descrever o comportamento do sistema, ou parte dele, e abstrair o que não é o foco dos testes ou do desenvolvimento. MBT é considerado um tipo de teste caixa preta, dado que os testes são derivados a partir do modelo de comportamento e não do código fonte. No entanto alguns autores utilizam um modelo muito usado nos testes caixa branca, que é o grafo de fluxo de controle, para representar aspectos comportamentais. Na seção 2.1 apresentamos alguns estudos existentes sobre o assunto. Na continuação, na Seção 2.2 são apresentadas algumas abordagens de MBT usando UML.

2.1 Grafos de Fluxo de Controle

Os Grafos de Fluxo de Controle (GFC) são vastamente utilizados para realização de testes estruturais, também conhecidos como testes "caixa branca" [Beizer 1990]. O GFC é a representação gráfica da estrutura de controle de um programa, sendo útil como modelo para obtenção e visualização das informações do fluxo interno do programa em teste.

Testes de Caminhos (do inglês Path Testing) é o nome dado a uma família de critérios[2] de testes baseada na seleção de um conjunto de caminhos de teste em programa [Beizer 1990]. Um caminho de teste é a representação seqüencial de todas as sentenças exercitadas durante a execução do programa fonte, a partir de um conjunto de entradas fornecido. Os caminhos são criados a partir do GFC [Paige 1978] e [Beizer 1990].

Para representar o GFC, Beizer divide os elementos obtidos a partir do código fonte em três grupos: blocos de código, decisões e junções [Beizer 1990]. Um bloco de código é uma seqüência de sentenças no programa sem interrupções por decisões ou junções; uma decisão é um ponto do programa no qual o fluxo de controle pode divergir, como uma sentença *if then else, while, for* e demais similares. Por último, uma junção é um ponto no programa no qual o fluxo converge, como o fim de uma sentença *if, while* e demais sentenças de decisão correspondentes. Esses elementos constituem os nós ou vértices do GFC. As arestas representam o fluxo de controle entre os nós.

2.2 Grafos de Fluxo de Controle Comportamental

Em testes "caixa preta", alguns trabalhos propõem a criação de Grafos de Fluxo de Controle Comportamental (GFCC) a partir da especificação do sistema. Dentre estes, dois trabalhos foram importantes para a nossa abordagem, propostos por Parrish et al [Parrish et al 1993] e por Edwards [Edwards 2000], os quais serão descritos resumidamente a seguir.

Parrish et al propõem a criação automática de cenários de teste para classes. O GFCC proposto neste trabalho é formado por vértices que representam as operações da interface da classe em testes, ou mensagens; e as arestas ligam quaisquer dois vértices do GFCC, representam seqüências válidas de chamada dessas operações. Esse grafo foi denominado "Grafo de Fluxo da Classe", em inglês, *Class Flow Graph*. Parrish et al

[2] Um critério (de cobertura) é uma condição ou regra que determina quão bem um sistema (ou parte dele) foi testado.

propõem ainda um *framework* para apoiar a aplicação de diversos critérios de testes a partir desse modelo.

O trabalho apresentado por Edwards [Edwards 2000], segue a mesma abordagem proposta por Parrish et al. Ele propõe a criação do GFCC para realização de testes "caixa preta", em Componentes de Software. O GFCC pode ser criado de maneira semi-automática, se o componente for especificado na linguagem Resolve [Sitaraman e Weide 1994], usada para especificar contratos. O GFCC criado por Edwards representa as interações na interface provida do componente em teste.

Uma limitação dessas abordagens é o fato da equipe de testes ter de desenvolver os GFCCs diretamente, sendo que este modelo não é comumente usado na prática. Por essa razão o nosso método irá considerar o uso de modelos da UML, pois estes são mais utilizados na indústria. Diversas técnicas foram propostas com base nestes modelos, bem como ferramentas que as apóiem. A Seção 2.3 apresenta alguns desses trabalhos.

Outra limitação dos trabalhos de Parrish et al e Edwards é não prever a interação com classes ou componentes externos ao componente em testes, o que é considerado na abordagem proposta neste trabalho, como será visto na Seção 3.

2.3 Testes Baseados em Modelos UML

Alguns trabalhos têm sido desenvolvidos para criação de testes a partir de modelos criados nas fases de análise e projeto de software. As notações propostas pela UML são bastante utilizadas nessas fases, com isso a criação de testes a partir de modelos UML torna-se uma abordagem interessante. Alguns trabalhos se baseiam no Diagrama de Atividades (DA). Os DAs são usados para representar seqüências possíveis de execução entre as atividades e permitem até mesmo a representação de atividades concorrentes. Eles são úteis, por exemplo, para modelar fluxo de processo de negócios e *workflows* [Binder 2000, cp. 8.6].

Briand e Labiche propuseram uma metodologia de teste, chamada Totem [Briand e Labiche 2001]. Esta metodologia propõe a especificação de testes a partir de modelos de Casos de Uso. Um Diagrama de Atividades (DA) é usado para descrever as dependências seqüenciais entre os casos de uso. Cada atividade no DA representa um caso de uso. Um caminho no DA representa, desse modo, um cenário de uso do sistema. Cada caso de uso é detalhado através de diagramas de seqüência e/ou colaboração, que representam os cenários internos dos casos de uso. Com isso, um problema encontrado, é o número de casos de teste gerados devido às combinações dos cenários internos do caso de uso e cenários de uso do sistema, que tendem a atingir valores exponenciais.

Outro trabalho que usa DAs foi proposto por Hartmann et al, onde são usados os modelos UML para realizar testes de unidade, de integração e de sistemas [Hartmann et al 2000]. Os autores propõem a descrição detalhada dos casos de uso do sistema através de DAs. Assim, cada caso de uso é representado por um diagrama de atividades e a partir deles são criados os casos de teste. Um problema neste trabalho é que os modelos e as ferramentas de apoio aos testes são dependentes da ferramenta IBM Rational Rose.

Uma vantagem do uso de diagrama de atividades para modelagem de testes é que sua representação é mais adequada para a geração de testes, se comparada com representações como a do Diagrama de Seqüência. Com isso, os DAs podem ser usados

para representar o fluxo de execução tanto no nível de sistema como mostrado nos trabalhos acima quanto no nível de componentes. Nossa abordagem, assim como a de Hartmann et al, usa somente DAs para os testes, tornando nosso método amplamente viável tanto no meio acadêmico, como na indústria. Os DAs são usados em dois níveis para representar o comportamento do componente que serão mostrados a seguir.

Uma desvantagem das abordagens apresentadas nas Seções 2.2 e 2.3 é não considerar a criação de *stubs*. Um *stub* é um substituto dos componentes requeridos e serve principalmente para isolar o componente em teste dos demais componentes dos quais ele depende. O uso de *stubs* permite um melhor controle da execução do componente durante os testes e, além disso, facilita o diagnóstico das falhas reveladas pelos testes. Uma apresentação mais detalhada sobre *stubs* pode ser obtida em [Binder 2000, c.19; Meszaros 2004].

A criação de *stubs* pode demandar um grande esforço de testes, pois estes são geralmente criados manualmente e são específicos para um caso de teste. Uma preocupação do método proposto é permitir que os *stubs* também possam ser criados de forma automatizada, juntamente com os casos de testes. Para isso, o modelo de comportamento do componente deve representar as interações com as interfaces requeridas, pois estas são necessárias para a criação dos *stubs*. Esse aspecto é abordado na Seção 3.2; o protótipo desenvolvido para mostrar a factibilidade do método proposto é descrito em 5.2.

Um outro problema importante na geração de testes diz respeito à obtenção dos dados de testes, ou seja, os valores dos parâmetros das operações das interfaces providas do componente. Esse aspecto ainda não é coberto pelo método apresentado aqui; por ora esses valores devem ser preenchidos manualmente.

3. Especificação e Modelagem do Componente

Para ilustrar a apresentação do método, será utilizado o componente *CruiseControl* [Cruise Control 2006] como exemplo. O componente é descrito na Seção 3.1. Seu modelo de comportamento é apresentado na Seção 3.2, utilizando o DA da UML 2.0 [OMG 2004] pois esta oferece maiores recursos, em especial, para a especificação de tratamento de exceção.

3.1 Especificação do Componente CruiseControl

O componente *CruiseControl* compõe o sistema de injeção eletrônica de alguns carros de luxo, responsável por manter a velocidade do carro constante. Funciona como um piloto automático: o motorista define uma velocidade a ser mantida (*Velocidade de Cruzeiro*) e o *CruiseControl* obtém o controle sobre aceleração e freio para manter essa velocidade estável. As funcionalidades básicas presentes nesse componente estão descritas na Tabela 1.

Tabela 1 Serviços oferecidos pelo componente CruiseControl.

Operações	Descrição
ON	Inicia o sistema de CruiseControl (independente da velocidade do veículo ou marcha);
OFF	Desliga o sistema de CruiseControl;
SET	Define qual será a velocidade a ser mantida pelo veículo, acionado apenas quando o sistema estiver em ON, e se a velocidade e a marcha forem válidas. Imediatamente após a velocidade ser definida, o CruiseControl passa a controlar o carro (aceleração e freios) para manter a velocidade constante. O controle do carro volta ao motorista automaticamente caso o freio ou acelerador sejam acionados;
TIP-UP	Aumenta a velocidade de cruzeiro, em 3 km/h;
TIP-DOWN	Diminui a velocidade de cruzeiro, em 3 km/h
RESUME	Retorna a ação do sistema de CruiseControl sobre o veículo caso o valor da velocidade já tenha sido definido anteriormente

O componente possui uma interface provida, *ICruiseControl*, que contém as operações correspondentes às funcionalidades citadas. O *CruiseControl* interage ainda com três outros componentes do carro: Transmissão, Injeção e Ignição, e Freio ABS, representados por três interfaces requeridas: *ITransmissao, IInjecaoIgnicao* e *IABS*.

3.2 Modelagem usando Diagramas de Atividades

No método proposto, o DA é usado para representar os cenários de uso do componente em teste. Idealmente, esse modelo é criado pelos desenvolvedores durante a especificação do componente. Caso não exista, este diagrama deve ser construído manualmente pelo projetista de testes, complementando a especificação do componente. Apesar do DA permitir a representação de atividades concorrentes, este aspecto ainda não é considerado neste estudo.

O DA é decomposto em dois níveis [Rocha 2005]: o primeiro ilustra o comportamento da interface provida do componente, representando os cenários de uso, enquanto o segundo ilustra as interações de cada operação das interfaces providas com as interfaces requeridas. O primeiro nível será designado por Diagrama de Cenários de Uso do componente (DCU). A Figura 1 (a) mostra o DCU para o componente *CruiseControl*.

No DCU, cada operação da interface provida do componente é representada por uma *ação* do diagrama, contendo a assinatura completa da operação. Uma *transição* entre duas operações A e B indica que existe uma dependência de controle entre A e B, ou seja, B só pode ser iniciada depois que A terminou. O lançamento de exceções pelas operações da interface é representado por uma aresta de exceção ligando a ação a um nó final. Na Figura 1(a), por exemplo, a exceção *ExpChaveDesligada* é lançada pela operação *on()* e não é tratada pelo componente, que termina lançando a exceção.

Cada operação no DCU, caso interaja com alguma interface requerida, é representada em um segundo nível do diagrama. A hierarquia é indicada pelo símbolo de *tridente* na ação, como ilustrado na Figura 1 (a) para as operações *on(), set() e off().* Cada uma destas ações é decomposta em outro diagrama de atividades, que chamaremos de Diagramas de Interação entre Operações (DIO).

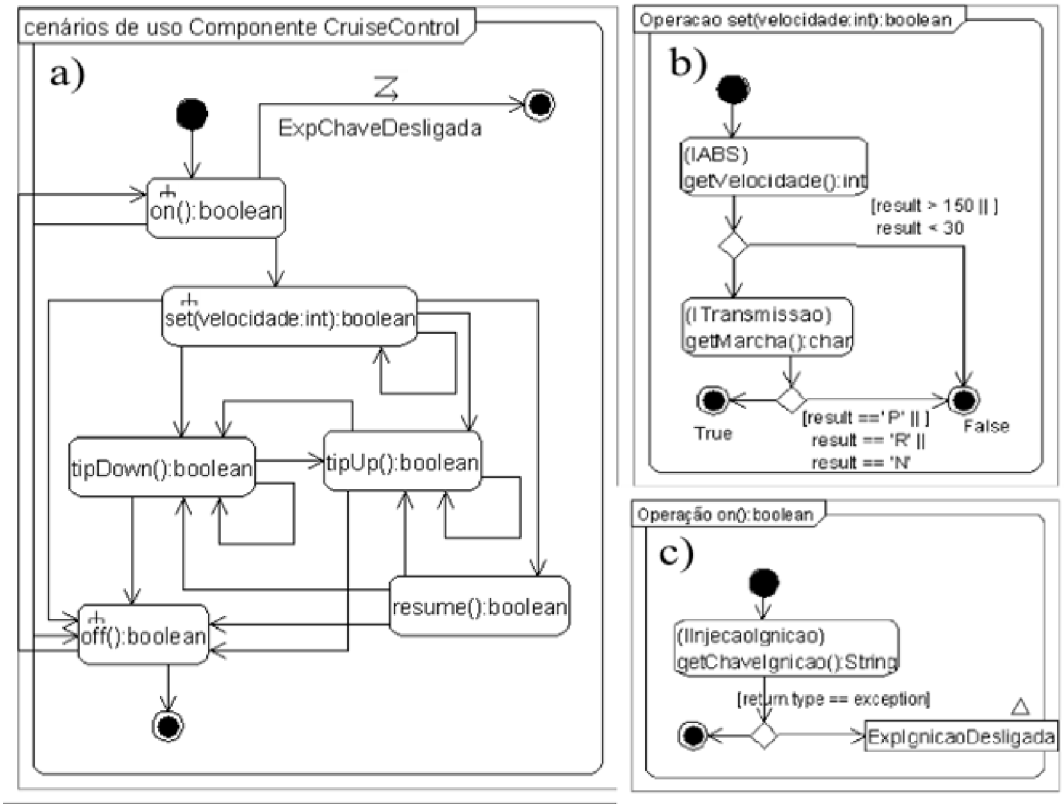

Figura 1 (a) DCU do componente CruiseControl; (b) DIOs das Operações set e; (c) on

O DIO de uma operação da interface provida ilustra sua interação com as interfaces requeridas: quais são chamadas, e como o fluxo de controle da operação provida varia de acordo com o valor retornado pela operação requerida. Cada *ação* neste diagrama representa uma operação de uma interface requerida, sendo que as ações são agrupadas em partições, de acordo com as interfaces requeridas a que pertençam.

As Figuras 1 (b) e (c) representam os DIOs relativos às operações *set()* e *on()*, respectivamente. Os DIOs são necessários para a criação dos *stubs*: cada partição (interface requerida) corresponde a um *stub* e cada operação representada neste diagrama representa uma operação que o *stub* deve conter. Vale observar que um *stub*, por ser um componnte substituto, não precisa implementar todas as operações da interface requerida, mas somente aquelas necessárias para que o caso de teste seja executado.

4. Montagem do GFC Comportamental

Nosso objetivo é aplicar critérios de teste de caminhos, descritos na Seção 2, aos Diagramas de Atividades representando o comportamento do componente em testes. Para aplicar estes critérios, são criados GFC correspondentes aos DAs criados conforme indicado na Seção 3. A Seção 4.1 mostra como os Diagramas de Atividades propostos na Seção 3.2 podem ser convertidos em GFCs. Em seguida, a Seção 4.2 mostra como os vários GFCs serão interligados, formando o Grafo de Fluxo de Controle Inter-atividades (GFCI).

4.1 Criar os Grafo de Fluxo de Controle a partir dos DAs

Conforme mostrado em 3.2, dois tipos de modelos são criados: o DCU e os DIOs. O GFC criado para cada DIO é chamado aqui de Grafo de Fluxo de Controle de Atividades (GFCA). Os GFCAs correspondentes aos DIOs do componente de exemplo possuem representação similares a apresentada na Figura 1.

De maneira mais formal, o GFCA, criado a partir de um Diagrama de Atividades, é um grafo direcionado que pode ser definido por uma tupla (*V*, *A*, *s*, *T*), onde *V* é um conjunto de vértices do GFCA representados pelas atividades do DIO. Em seguida, *A* é um conjunto que representa as arestas do grafo que correspondem às arestas do DIO, *s* é o vértice de entrada do GFCA, tal que, $s \in V$ e equivale ao vértice inicial do DIO. *T* é um conjunto que contém os vértices de saída do GFCA, tal que, $T \subset V$ e os vértices contidos em *T* são representados pelos vértices finais do DIO. Assim, o conjunto de vértices $V \setminus s, T$ (todos os vértices de *V* exceto *s* e o subconjunto *T*) é denominado conjunto de vértices internos do GFCA.

4.2 Grafo de Fluxo de Controle Inter-Atividades

Para a representação do DCU, que contém os cenários de uso das operações das interfaces providas do componente, é proposta a criação de um GFC interligando todos os GFCAs. Este GFC é chamado de Grafo de Fluxo de Controle Inter-atividades (GFCI), e baseia-se na proposta de Sinha et al, do Grafo de Fluxo de Controle Interprocedimental [Sinha et al 2001]. Na Seção 2.1 foi mostrado como obter um GFC para um programa. O GFC é criado para cada procedimento do programa. O que fazer quando um programa possui vários procedimentos? Sinha et al propuseram um Grafo de Fluxo de Controle criado para programas que contêm vários procedimentos.

Nesse grafo, denominado GFC Interprocedimental, cada procedimento do programa é representado por um GFC. Em cada GFC as chamadas para outros procedimentos são representadas através de um tipo de vértice específico, de *chamada*. Uma aresta de chamada interliga este vértice ao vértice de entrada do procedimento que é chamado. Analogamente, vértices de *retorno* são criados para representar o retorno ao ponto de chamada. Uma aresta de retorno liga um vértice de saída no procedimento ao vértice de retorno no ponto de chamada.

Da mesma forma que o trabalho de Sinha et al, aqui os GFCAs, que representam os DIOs, são interligados em um GFC Inter-atividades (GFCI), que representa o DCU. Um GFCI pode ser definido pela tupla (*G*, *C*, *R*, *I*, s_s, t_s), tal que, G é o conjunto de GFCAs que irá compor o GFCI, C é um conjunto de Vértices de Chamada, R um conjunto de Vértices de Retorno, *I* é um conjunto de arestas que interconectam os GFCAs, chamadas de arestas Inter-atividades (arestas de chamada e arestas de retorno). Por último, s_s e t_s representam os vértices de entrada e saída global do GFCI.

Na Figura 2 é mostrado o exemplo de um GFCI criado a partir do DCU e dos DIOs mostrados na Figura 1. Os vértices de chamada e retorno são representados com fundo cinza e com letras em negrito; já as arestas inter-atividades são representadas de forma tracejada. O GFCA da operação *off()* não foi representado para simplificação e para não comprometer a legibilidade da Figura.

Os Vértices de Chamada (*Ch.*) são criados a partir das ações do DCU que são

descritas através dos DIOs, como exemplo, na Figura 2, as operações *on()* e *set()* apresentadas no DCU da Figura 1. Os Vértices de Retorno (*Ret*) do GFCI representam os pontos de retorno de cada vértice de chamada. Foram definidos dois tipos de Vértice de Retorno: Retorno Final e Retorno por Exceção (Retorno Exp. na Figura 2).

O Retorno Final é para representar situações em que o GFCA chamado termina normalmente, na Figura 2 (a) o vértice Final do GFCA (*set*) está ligado ao *Ret. Set(velecidade:int):boolean* que representa o Retorno Final (normal) do GFCA. O Retorno por Exceção é usado para representar situações onde é lançada uma exceção pela operação da interface requerida e esta exceção é propagada para o componente em teste. A representação das exceções propagadas é feita através dos parâmetros de saída no DIO. Um exemplo na Figura 2 (c) é o parâmetro *ExpIgnicaoDesligada*.

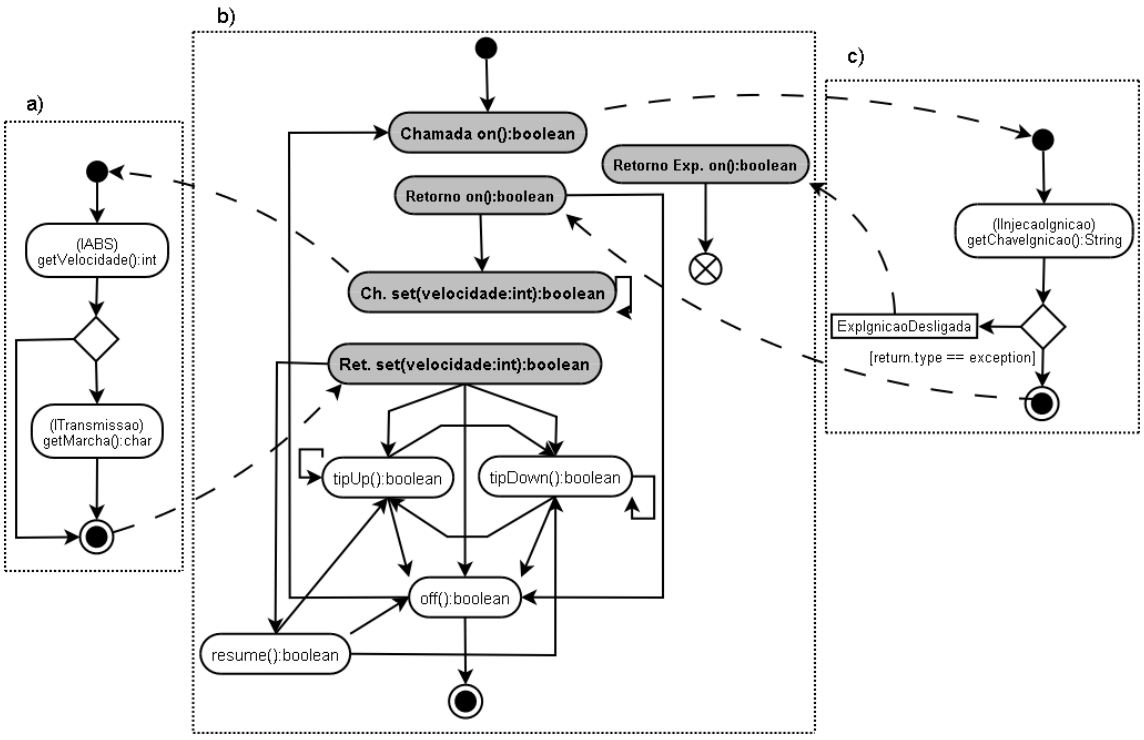

Figura 2 a) GFCA da operação set; b) GFCI do CruiseControl; c) GFCA da operação on.

A Tabela 2 mostra um comparativo entre a quantidade de vértices e arestas criadas nos GFCI correspondente aos respectivos DIOs e DCU. Pode-se observar que houve aumento no número de vértices na criação do GFCI, devido à inserção dos vértices de retorno para fluxo normal e de exceção. Esses vértices são associados aos vértices de chamada das operações *on()*, *off()* e *set()*. O número de vértices e transições adicionais é proporcional a quantidade de DIOs presentes no modelo. Assim, a transformação dos modelos que representam o sistema em GFCI não apresenta um grande impacto, tornando a abordagem viável.

Tabela 2 Nós e transições nos diagramas e grafos de fluxo de controle.

	Diagrama de atividades		GFC	
Diagrama	**N° de nós**	**N° de transições**	**N° de nós**	**N° de arestas**
DCU	9	20	13	23
on():Boolean	5	4	5	6
off():Boolean	9	10	9	11
set(velocidade:int):boolean	7	7	7	8

5. Seleção de Caminhos e Obtenção dos Casos dos Testes

Esta seção apresenta uma rápida descrição da seleção de testes a partir do GFCI. Uma descrição mais detalhada dos algoritmos usados será objeto de um futuro trabalho. Em seguida, é apresentado o protótipo desenvolvido para mostrar a viabilidade do método aqui proposto.

5.1 Criação de testes

Para criação dos casos de teste o primeiro passo consiste na seleção de caminhos de teste completos no GFCI. Um caminho é dito completo se começa no nó inicial, *s*, e termina no nó final, *t*. Cada caminho completo no grafo corresponde a um cenário de uso do componente.

Um exemplo de caminho de teste completo é mostrado na Figura 3. O caminho representado na Figura contém as atividades do DCU, que deu origem ao GFCI do sistema, e atividades de dois DIOs, que deram origem a dois GFCAs, da operação *on()* e da operação *off()* respectivamente. As informações obtidas neste caminho serão necessárias na criação dos casos de teste.

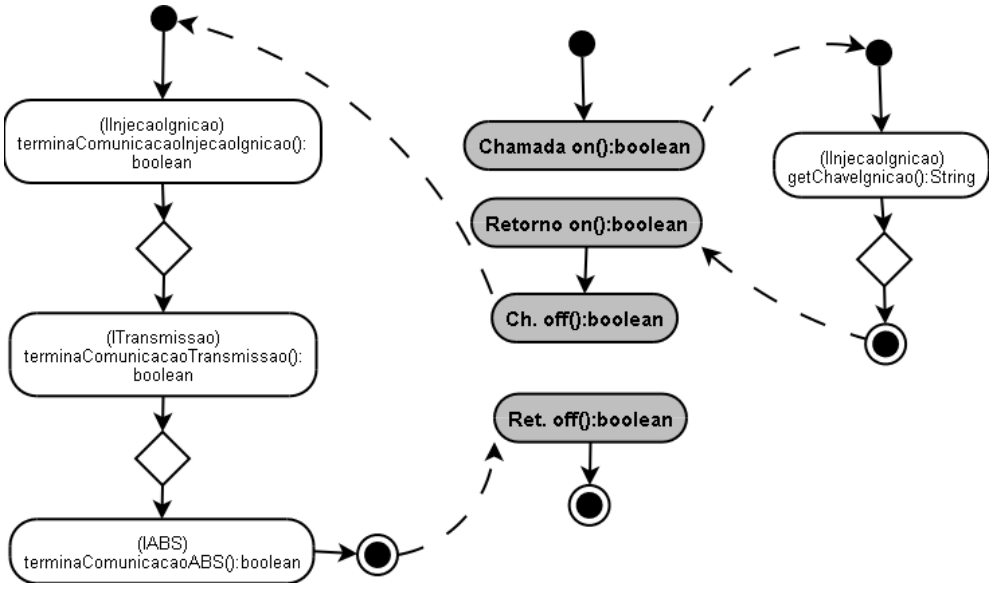

Figura 3 Caminho de teste selecionado na cobertura de ramos.

Em seguida, para uma escolha de caminhos adequados, é necessário definir o critério de cobertura usado para selecionar os caminhos de teste [Beizer 1990, cp. 2.3.4]. Os critérios de cobertura utilizados para selecionar caminhos neste trabalho foram baseados

naqueles mostrados por Beizer, sendo os principais: cobertura de todos os nós; cobertura de todas as arestas (ou ramos); e, por último, cobertura de todos os caminhos.

Dado que o critério de cobertura de todos os caminhos pode gerar um número infinito de caminhos, devido à existência de ciclos (ou *loops*) no GFC, limitamos o número de vezes que um nó aparece em um caminho, conforme proposto no critério de caminhos de nível-i [Paige 1978], onde *i* é a quantidade de vezes que um vértice pode se repetir em um caminho. Assim os caminhos foram selecionados para atender cada um dos três critérios, tornando possível uma comparação da quantidade caminhos selecionados para cada critério. O caminho de exemplo mostrado na Figura 3 foi selecionado usando o critério de cobertura de todos os ramos.

Uma vez selecionado um caminho, tem-se a informação de quais *stubs* serão necessários durante a execução do caso de teste correspondente, bem como as operações a serem definidas para cada *stub*. Por exemplo, no caminho mostrado na Figura 3, será criado um *stub* para a interface *IInjecaoIgnicao*, com a operação *getChaveIgnicao()*. Esta operação conterá somente o valor de retorno requerido para a execução do caminho selecionado. Um exemplo da especificação de *stubs* é apresentado na Figura 4 da Seção 5.2.

5.2 Protótipo

Para automatizar o método de teste foi implementado o protótipo de uma ferramenta. Esse protótipo é composto de vários módulos. Em uma primeira parte, um módulo efetua a leitura dos DAs especificados em um arquivo XMI (XML Metadata Interchange) [OMG 2005] e, a partir dos DAs, cria o GFCI correspondente. O GFCI obtido, sobre os diagramas do estudo de caso, possui um total de 34 vértices e 48 arestas, contando as arestas normais e inter-atividades.

Um outro módulo implementa os critérios de cobertura mencionados em 5.1 para seleção de caminhos de teste no GFCI. Os resultados obtidos para seleção de caminhos para o modelo do exemplo foram: i) 8 caminhos para a cobertura de nós; (ii) 18 caminhos para a cobertura de arestas e (iii) 11 caminhos para cobertura de caminhos com limite de uma repetição de nó por caminho, e 514 caminhos para duas repetições.

Apesar do critério de cobertura de caminhos ser aquele com maior potencial para revelar a presença de falhas, a grande quantidade de caminhos selecionados torna sua aplicação muito difícil para componentes maiores e mais complexos.

Além de selecionar caminhos, outro módulo do protótipo é responsável por criar os casos de teste em formato XML, definido em um trabalho prévio [Rocha 2005], compatível com uma ferramenta de execução de testes, a CBDUnit [Guerra et al 2005].
A Figura 4 mostra a especificação do caso de teste da Figura 3. Cada método da interface provida, mostrado no DCU (ver Figura 1), corresponde, no arquivo XML, a um *MethodCall*. As interações com as interfaces requeridas (*stubs*), obtidas através dos DIOs mostrados na Figura 1, estão explícitas no caso de teste e são representadas por *Interaction*.

Conforme se pode observar na Figura 4, os casos de teste ainda não possuem os dados, especificados pelos campos *value*, representando os parâmetros das operações na interface provida, bem como os valores de retorno das interações, que representam os

stubs. Assim, é necessária a intervenção do testador para criá-los manualmente, bem como os resultados esperados para cada caso de teste.

Para uma avaliação inicial do método, os casos de teste gerados para cobertura de todos os ramos forma implementados e executados. Para isso, após a geração dos casos de teste, foram selecionados os dados de entrada, para tornar os casos de teste executáveis. Um caso de teste é executável quando existem dados de entrada para exercitar a seqüência de chamadas do caso de teste.

Os casos de teste em XML foram então executados usando o *framework* Junit [Object Mentor Inc 2007]. Um caso de teste pode ser estruturado em quatro partes distintas, que são executadas em seqüência [Binder 2000, cp. 19]: preparação (*setup*), execução, verificação e término (*teardown*). A parte denominada execução contém as operações no caminho de teste selecionado. A criação dos *stubs* e sua configuração para que retorne os valores requeridos pelo caso de teste são feitas na preparação ou *setup* do caso de teste. A etapa de término (*teardown*) finaliza o caso de teste.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- Author: Júlio  -->
<!-- Date: 22/03/2007 20:39:30  -->
- <TestSuite component="CruiseControl" classpath="CruiseControl">
  - <TestCase seq="0" name="" objective="">
    - <MethodCall interface="CruiseControl" name="on">
      - <Interaction type="normal" name="" operation="getChaveIgnicao">
          <Stimulus type="normal" dataType="String" value="" />
        </Interaction>
        <Exp type="normal" dataType="boolean" value="" />
        <Result type="normal" dataType="boolean" value="" />
        <Verdict />
      </MethodCall>
    - <MethodCall interface="CruiseControl" name="off">
      - <Interaction type="normal" name="" operation="terminaComunicacaoInjecaoIgnicao">
          <Stimulus type="normal" dataType="boolean" value="" />
        </Interaction>
      - <Interaction type="normal" name="" operation="terminaComunicacaoTransmissao">
          <Stimulus type="normal" dataType="boolean" value="" />
        </Interaction>
      - <Interaction type="normal" name="" operation="terminaComunicacaoABS">
          <Stimulus type="normal" dataType="boolean" value="" />
        </Interaction>
        <Exp type="normal" dataType="boolean" value="" />
        <Result type="normal" dataType="boolean" value="" />
        <Verdict />
      </MethodCall>
    </TestCase>
  - <TestCase seq="1"  ...
```

Figura 4 Exemplo de Caso de Teste gerado para o componente CruiseControl.

6. Resultados Obtidos e Trabalhos Futuros

Para validar o método proposto foram executados casos de teste para cobertura de ramos, implementados conforme mostrado na Seção 5. No total foram 18 casos de teste (Seção 4), para cada caminho selecionado no GFCI (Seção 3.2). Observamos que: (i) os casos de testes criados foram executáveis, ou seja, foi possível criar dados de entrada para executar os caminhos de teste; (ii) as informações obtidas foram úteis para construção dos casos de teste e *stubs* necessários; (iii) a especificação comportamental e especificação das interfaces providas e requeridas possibilitaram antecipar as fases de

projeto e especificação dos casos de teste; (iv) a modelagem comportamental para obtenção dos casos de teste pode facilitar o reuso dos testes caso seja necessário testar mais componentes com especificações similares.

Apesar da cobertura de código não ser um objetivo na seleção dos testes baseados em modelos, determinou-se o quanto do código foi coberto pelos testes realizados. Os resultados da análise foram de cobertura de 89% dos ramos e 85% das instruções do componente CruiseControl. As instruções não cobertas correspondem a alguns métodos do tipo *get* e *set*, que, por não serem representados no modelo comportamental[3], não foram cobertos. A cobertura dos testes depende de quão completa é a especificação comportamental do componente; os métodos não especificados tiveram sua cobertura comprometida.

Os testes também revelaram a presença de duas falhas: (i) a operação *set()*, quando usada duas vezes consecutivas no caminho, não armazenava a velocidade de entrada na segunda vez em que era chamada; (ii) quando se chamava *on()* e *off()* duas vezes consecutivas, a operação *on()* não interagia mais com os *stubs* na segunda chamada, ou seja, o componente não iria interagir, em fase operacional, com os componentes requeridos. É importante ressaltar que essas falhas não haviam sido detectadas pelos testes baseados no perfil de uso fornecido pelo cliente.

Cumpre observar que o CruiseControl foi desenvolvido em uma disciplina do curso de pós-graduação do IC-Unicamp, baseado em uma especificação de um sistema real, em que o cliente foi uma empresa fornecedora desse tipo de sistema. Os casos de testes baseados no perfil de uso foram fornecidos pela empresa cliente.

Uma vantagem do método proposto é que a implementação dos casos de teste e *stubs* pode ser automatizada, com isso diminuindo o custo e tempo demandado na fase de projetos de teste, especialmente os testes de unidade. Para uma avaliação mais precisa, como por exemplo, do custo computacional e dos tempos para geração dos testes, aplicações em componentes maiores devem ser realizadas em trabalhos futuros.

Além disso, outro trabalho importante é dar suporte a criação de dados de teste ou a manutenção das entradas e saídas esperadas usando um pool de dados[4] predefinidos (fornecido) pelos testadores. Encontra-se em andamento o desenvolvimento de uma ferramenta de apoio ao método, integrando os módulos que constituem o atual protótipo com o ambiente de execução de testes da CBDUnit.

Referências

Beizer, B. (1990) “Software Testing Techniques”, 2nd Edition.

Binder, R. V. (2000) “Testing object-oriented systems”, Addison-Wesley.

Briand, L. e Labiche, Y. (2001) “A uml-based approach to system testing”. In: UML’01: Proceedings of the 4th International Conference on The Unified Modeling Language,

[3] Em desenvolvimento OO, métodos desse tipo só são representados em modelos de mais baixo nível de abstração.

[4] Conjunto de dados necessários para executar os cenários exercitados pelos casos de teste.

Modeling Languages, Concepts, and Tools. Springer-Verlag, London, UK, pp. 194–208.

Cruise Control (2006) “Cruise Control” http://en.wikipedia.org/ wiki/Cruise_control última consulta em 25/08/2006.

Edwards, S. H. (2000) “Black-box testing using flowgraphs: An experimental assessment of effectiveness and automation potential”. Software Testing, Verification and Reliability, 10(4), pp. 249-262.

Guerra, P. A. C., Araújo, C. S., Rocha, C. R. e Martins, E. (2005) “Cbdunit - uma ferramenta para testes unitários de componentes”. In: Anais de XIX Simpósio Brasileiro de Engenharia de Software - XII Sessão de Ferramentas.

Hartmann, J., Imoberdorf, C. e Meisinger, M. (2000) “UML-Based integration testing” ACM SIGSOFT Software Engineering Notes Volume 25 , Issue 5 Pages: 60 - 70 ISBN:1-58113-266-2.

Meszaros, G. A (2004) “Pattern Language for Automated Testing of Indirect Inputs and Outputs using XUnit”. In Proceedings of PLOP’2004.

Meyer, B. (1997) “Object-Oriented Software Construction”. Prentice Hall, segunda edição.

OMG (Object Management Group) (2004) “UML 2.0 Superstructure Final Adopted Specification”, OMG document ptc/03-08-02, 2004.

OMG (Object Management Group) (2005) “Meta Object Facility (MOF) 2.0 XMI Mapping Specification”, OMG document formal/05-09-01.

Object Mentor Inc (2007) Junit.org. http://www.junit.org.

Paige, M. R. (1978) “An analytical approach to software testing”, Proceedings COMPSAC.

Parrish, A.S. Borie, R. B. e Cordes, D. W. (1993) “Automated flow graph-based testing of object-oriented software modules” - Journal of Systems and Software - portal.acm.org

Rocha, C. (2005) “Um Método de Testes para Componentes Tolerantes a Falhas”. Campinas: Instituto de Computação da UNICAMP. 153 p. (Dissertação de Mestrado).

Sinha, S. e Harrold, M. J. (2001) “Interprocedural Control Dependence”. ACM Transactions on Software Engineering and Methodology, 10(1), Pages 209-254.

Sitaraman M. e Weide, B.W. (1994) “Component-based software engineering using RESOLVE”. ACM SIGSOFT Software Enigineering Notes.

Szyperski, C. (1998) “Component Software: Beyond Object-Oriented Programming”, Addison-Wesley.

Weyuker, E. (1998) “Testing Component-Based Software: A Cautionary Tale”, IEEE Software, 15(5): 54-59, September/October 1998.