Um Método de Teste Funcional para Verificação de Componentes

Carina M. Farias e Patrícia D. L. Machado[1]
Departamento de Sistemas e Computação - Universidade Federal de Campina Grande
Caixa Postal 10.106, 58.109-970, Campina Grande - PB
{carina, patricia}@dsc.ufcg.edu.br

Resumo. O interesse no desenvolvimento de software baseado em componentes tem crescido substancialmente devido à promessa de redução de custos e tempo de desenvolvimento através do reuso. A maioria das metodologias existentes tem se concentrado nas fases de análise e projeto. Entretanto, o reuso efetivo de componentes está fortemente relacionado à confiabilidade dos mesmos. O principal objetivo deste trabalho é propor um método de teste funcional aplicável a componentes de software. Já que o processo de desenvolvimento utilizado influencia significativamente a testabilidade de sistemas, o método proposto estará integrado a um processo de desenvolvimento de componentes bem definido. Artefatos de teste são gerados a partir de especificações em UML (Unified Modelling Language). Um estudo de caso é apresentado para ilustrar a aplicabilidade do método.
Palavras-chave: Teste Funcional, Componentes, UML, Orientação a Objetos.

Abstract. Interest in component-based software development has increased significantly due its promise to reduce development costs and time through reuse. The majority of existing methodologies has focus in the analysis and design disciplines. Nevertheless, effective reuse of components is closely related to their reliability. The main goal of this work is to propose a method of functional testing to verify software components. Since testability of systems is greatly influenced by the development process chosen, the proposed method is integrated with a well-defined component development process. Test artifacts are generated from UML (Unified Modelling Language) specifications. A case study has been developed to illustrate the applicability of the method.
Keywords: Functional Testing, Components, UML, Object Orientation.

1. Introdução

A crescente preocupação com o aumento dos custos envolvidos no processo de desenvolvimento de software tem motivado o desenvolvimento de novas tecnologias capazes de produzir código eficiente, manutenível e compreensível, com recursos humanos e de tempo limitados.

Nesse sentido, as tecnologias centradas no reuso de software têm recebido uma atenção especial, já que o reuso favorece efetivamente a amortização dos custos do desenvolvimento do software entre seus usuários, além de possibilitar a redução no tempo de desenvolvimento.

Dentre os paradigmas de desenvolvimento que apresentam o reuso como uma de suas principais características, a engenharia de software baseada em componentes é uma das abordagens que vem mais crescentemente sendo adotada no desenvolvimento de software, tendo em vista que se trata de um paradigma capaz de combinar unidades pré-desenvolvidas, com o objetivo de reduzir o tempo de desenvolvimento, facilitar o acoplamento de novas

[1] Este trabalho e seus autores são apoiados pelo CNPq, processo 552190/2002-0. A primeira autora recebeu apoio financeiro da CAPES.

funcionalidades, ou a mudança de funcionalidades já existentes, e ainda promover o reuso de partes do software.

Podem ser encontradas várias definições de componente na literatura. Neste trabalho, estamos considerando a definição de C. Szyperski [17]: "Um componente de software é uma unidade de composição com interfaces especificadas através de contratos e dependências de contexto explícitas. Um componente de software pode ser distribuído independentemente e está sujeito a composição com outras partes".

Apesar dos benefícios, a engenharia de software baseada em componentes ainda apresenta muitas limitações e desafios que devem ser vencidos. Dentre estes desafios, destacamos neste trabalho a ausência de técnicas que testem efetivamente os componentes individualmente.

Um dos requisitos para se obter sucesso no uso de componentes é a verificação adequada de sua funcionalidade, tanto por parte do fornecedor do componente quanto por parte do cliente. O fornecedor precisa verificar o componente independente de contextos específicos de utilização. Por outro lado, o cliente precisa verificar o componente, possivelmente de propósitos mais gerais, dentro de um contexto específico. Teste é uma das técnicas de verificação de software mais utilizadas na prática. Se usado de forma efetiva, pode fornecer indicadores importantes sobre a qualidade e confiabilidade de um produto.

Teste funcional apresenta-se como uma ferramenta de suma importância tanto para o fornecedor quanto para o cliente do componente. Baseando-se numa descrição da funcionalidade do software, o teste funcional é capaz de checar se o software está de acordo com sua especificação independentemente da sua implementação. O fornecedor de um componente precisa garantir que as propriedades especificadas na interface do componente serão válidas para todos os possíveis contextos de uso deste componente. Em termos teóricos, essa garantia apenas pode ser obtida por meio de provas formais, dado que exista uma especificação precisa do componente e das propriedades do ambiente onde ele deverá executar. Não é possível alcançar essa garantia completamente por meio de testes, já que seria necessário executar o componente em todos os possíveis contextos, o que não é viável, ou mesmo possível. Entretanto, teste funcional pode ser usado para verificar propriedades do componente independentemente da sua implementação em contextos específicos. Se o conjunto de contextos for criteriosamente selecionado e representativo, é possível ampliar os resultados obtidos com os testes para um conjunto maior de contextos equivalentes [12]. Se os testes forem automáticos, poderão ser facilmente repetidos para verificar o componente em novos contextos. Finalmente, para os usuários do componente, teste funcional pode ser a única forma de testar o componente de forma satisfatória, já que normalmente o código fonte do componente não está disponível.

Neste trabalho, estamos propondo um método de teste funcional aplicável a componentes de software. A principal contribuição deste trabalho está em possibilitar a verificação de propriedades individuais dos componentes e empacotar artefatos e resultados de teste de forma adequada, facilitando o reuso e a composição de componentes pelos clientes. Vale ressaltar que o método de teste aqui proposto combina técnicas existentes e se integra a um processo de desenvolvimento de componentes, o que contribui para aumentar a testabilidade[2] dos componentes desenvolvidos. Artefatos de teste são gerados a partir de especificações em UML. É importante notar que uma metodologia de teste completa deve incorporar também técnicas de teste estruturais, além de permitir que sejam executados testes de integração e sistema, entretanto, isto está fora do escopo deste trabalho.

Alguns trabalhos relacionados podem ser encontrados. Em [10], é tratado o problema de inserir sistematicamente a atividade de teste no processo de desenvolvimento. Propostas de

[2]Propriedade que engloba todos os aspectos que facilitam a elaboração e execução dos testes de software [14].

técnicas para teste funcional de software orientado a objetos podem ser encontradas em [5,7,8,18,1]. Em [9], é abordado o problema de testar sistemas baseados em componentes. Já o teste funcional de componentes é tratado em [13,12,4]. Contudo, estes trabalhos, em geral, apresentam métodos e técnicas isolados, fora do contexto de um processo de desenvolvimento e não cobrem todas as etapas de um processo de teste.

A próxima seção apresenta resumidamente o processo de desenvolvimento de componentes utilizado para dar suporte ao método, chamado Componentes UML [6]. A Seção 3 apresenta a técnica de teste TOTEM que é a principal técnica usada pelo método. O método de teste é introduzido na Seção 4. e suas etapas são apresentadas em mais detalhes nas Seções 5., 6. e 7.. Um estudo de caso e seus resultados são apresentados nestas seções. Por fim, apresentamos as conclusões na Seção 8.

2. Componentes UML

A utilização efetiva do método de teste proposto neste trabalho é intimamente dependente do processo de desenvolvimento e dos artefatos produzidos durante este processo. Utilizamos, tanto na apresentação do método quanto no desenvolvimento do estudo de caso, o processo de desenvolvimento Componentes UML. Entretanto, qualquer outro processo de desenvolvimento pode ser utilizado, desde que o processo atenda às seguintes recomendações:

- Diagramas de seqüência devem ser produzidos para representar os cenários de uso do componente. Deve existir um diagrama para cada cenário, incluindo os cenários principais e alternativos.
- As operações presentes nas interfaces dos componentes devem ser especificadas usando OCL (*Object Constraint Language*).

Componentes UML abrange as seguintes etapas, em um processo iterativo e incremental:

- Definição de Requisitos: descobre os objetivos do software a ser desenvolvido.
- Modelagem de Componentes: identifica os componentes que irão compor o sistema final e gera a especificação destes componentes. As interações entre os componentes são também identificadas nesta etapa.
- Materialização de Componentes: fornece implementações dos componentes especificados, seja implementando a especificação, seja adquirindo um componente existente que satisfaça a especificação.
- Montagem da Aplicação: os componentes previamente implementados ou adquiridos e testados individualmente, são reunidos em um sistema e uma interface de usuário para este sistema é projetada de forma a se obter uma aplicação.
- Testes.
- Distribuição da Aplicação.

Este processo foi usado para exemplificar o método de teste por se tratar de um processo simples, que produz um conjunto mínimo de artefatos. As atividades e artefatos que devem ser desenvolvidos em cada uma das etapas citadas acima estão bem documentadas em [6], com exceção das duas últimas etapas. Neste trabalho, detalhamos a atividade de teste no contexto da metodologia Componentes UML, tornando-a ainda mais completa.

3. TOTEM

O objetivo do método de teste que estamos propondo é verificar as funcionalidades individuais de um componente. Dessa forma, ficamos particularmente interessados no uso dos diagramas de seqüência para derivar os casos de teste, já que estes diagramas representam a interação entre os objetos que compõem o componente para entregar uma determinada funcionalidade.

A principal técnica de teste que escolhemos para ser usada no método foi a técnica TOTEM (*Testing Object-orienTed systEms with the unified Modeling language*) [5]. Escolhemos esta técnica porque ela apresenta potencial para automação e usa artefatos UML e especificações OCL como fonte de geração dos testes.

TOTEM permite usar Diagramas de Caso de Uso, Diagramas de Seqüência ou Colaboração e Diagramas de Classe, para derivar os casos de teste [3], dados [4] e oráculos [5].

A técnica TOTEM propõe que cada diagrama de seqüência seja convertido em uma expressão regular cujo alfabeto são os métodos públicos dos objetos no diagrama de seqüência. A técnica considera que cada diagrama de seqüência abrange todos os cenários de uso para o caso de uso. Dessa forma, a expressão regular derivada do diagrama de seqüência é um produto de termos onde cada termo representa um cenário específico. A notação usada na construção dos termos é $Operacao_{Classe}$, denotando a operação que está sendo executada e a que classe a operação pertence.

A partir da expressão regular, obtém-se um grafo, que é utilizado para selecionar os casos de teste, ou seja, os caminhos que serão exercitados no sistema. Grafos são a estrutura mais comumente usada para representar modelos de teste, tanto funcional quanto estrutural [3,15].

Para construir os oráculos, a técnica propõe que sejam identificadas, inicialmente, as condições de execução de cada termo da expressão regular, ou seja, para cada cenário de uso deve-se identificar as condições de execução deste cenário. Deve-se identificar ainda se a execução do cenário causa ou não uma mudança no estado do componente e que mudança é esta. Identificamos ainda as mensagens que são retornadas para o ator do caso de uso em resposta à execução de cada um dos cenários representados na expressão regular. Todas essas informações são reunidas em uma tabela de decisão, utilizada para orientar a construção dos oráculos de teste.

Utilizamos esta técnica no método aqui proposto com algumas adaptações que serão comentadas na próxima seção.

4. O Método de Teste

Em geral, a atividade de teste é vista como a última atividade realizada no processo de desenvolvimento de software. Desse ponto de vista, é necessário que a implementação do software esteja completa para que a atividade se inicie.

Neste trabalho, consideramos que a atividade de teste deve ser aplicada em vários pontos do desenvolvimento e não apenas no final do processo. Mais que isso, consideramos que os testes não devem se aplicar apenas ao código do programa, mas a todos os artefatos produzidos durante o desenvolvimento do software. Aplicar o teste dessa forma aumenta as chances de se desenvolver um sistema que atenda às expectativas do cliente.

---

[3] Aspectos ou funcionalidades de um sistema a serem testados, expressos por critérios de entrada e saída.
[4] Valores que servem de entrada para a execução de um caso de teste.
[5] Procedimentos responsáveis por decidir se um teste obteve ou não sucesso.

Um processo de teste completo pode envolver as seguintes etapas [14]:

- Planejamento: Nesta etapa, são definidos que tipos de teste serão realizados e quais as expectativas com relação aos testes escolhidos.
- Especificação: Nesta etapa, são gerados os modelos de teste dos quais são derivados os casos de teste, dados e oráculos.
- Construção: Nesta etapa, os artefatos necessários para execução dos testes são criados. Os casos de teste e os oráculos identificados na etapa anterior são implementados utilizando-se alguma linguagem de programação.
- Execução: Nesta etapa, são executados os casos de teste desenvolvidos para o sistema na etapa anterior, sendo fornecidos os dados selecionados na etapa de especificação.
- Análise dos Resultados: Nesta etapa, os oráculos gerados na etapa de construção são utilizados para analisar se o programa em teste falhou ou não em um teste executado.

Além das etapas acima, em se tratando de teste de componentes, uma sexta etapa pode ser realizada. Nesta etapa, os artefatos e resultados dos testes obtidos nas etapas anteriores são empacotados juntamente com o componente desenvolvido a fim de fornecer ao usuário do componente facilidades para conhecer os testes executados e desenvolver novos testes, caso seja necessário [11].

O método aqui proposto fornece um conjunto de diretrizes e técnicas que auxiliam a execução e o acompanhamento de cada uma dessas etapas. Para tornar a descrição do método mais clara, integramos o método à metodologia Componentes UML apresentada na Seção 2.. Essa integração é mostrada na Figura 1.

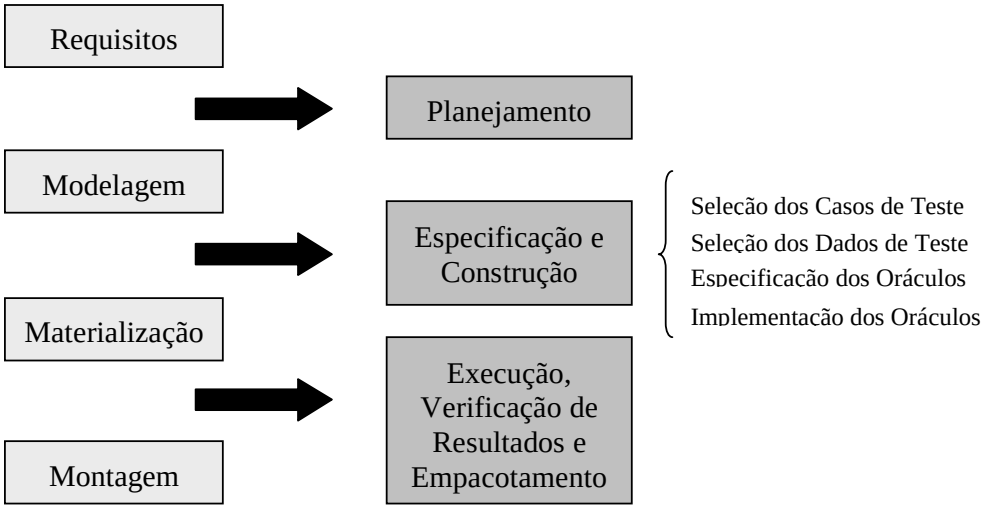

Figura 1: Integração das etapas de teste no processo de desenvolvimento.

As atividades de teste foram inseridas no processo da seguinte forma:

- O planejamento dos testes pode ser feito durante a definição dos requisitos. Esta atividade tem por objetivo definir que tipos de teste serão realizados e o que se espera da realização destes testes.
- A especificação dos testes deve ser feita à medida que a modelagem é realizada. Em linhas gerais, durante a etapa de modelagem, estaremos selecionando os casos

de teste, usando a técnica TOTEM [5], combinada com o aspecto estatístico do Cleanroom [15,19]. Os dados são também selecionados durante esta etapa. Para esta atividade usamos as orientações de teste por partições proposta em [16,2]. Por fim, os oráculos são gerados baseando-se na técnica TOTEM, a partir das especificações de pré e pós-condições feitas em OCL.

- A construção dos testes pode ser realizada em paralelo à sua especificação ou ao final da especificação.
- A execução dos testes e análise dos seus resultados é feita após a materialização dos componentes. Nesse momento, devemos também empacotar os artefatos de teste juntamente com o componente produzido.

Além dos aspectos citados acima, consideramos que os artefatos produzidos durante a modelagem passarão por sessões de inspeções que avaliarão a correção, completude e consistência desses artefatos. Para se ter testes efetivos, é importante que os artefatos sejam de boa qualidade, visto que retratam a funcionalidade desejada.

Nas subseções seguintes, detalhamos cada etapa do processo de teste e suas atividades.

Estudo de Caso. A fim de obter impressões iniciais sobre a aplicabilidade do método, realizamos um estudo de caso, onde a aplicação escolhida foi um Sistema de Reservas em Hotel, cujo objetivo é permitir a realização de reservas em diferentes hotéis pertencentes a uma rede de hotéis. No desenvolvimento deste estudo de caso, não nos limitamos a realizar apenas as atividades de teste; seguimos todo o processo de desenvolvimento a fim de complementar a especificação do sistema, encontrada em [6], implementá-lo e testá-lo. A preocupação com a testabilidade dos componentes nos levou a realizar sessões de inspeção e revisão dos modelos a fim de que produzíssemos modelos corretos, completos e consistentes. Entretanto, por limitações de tempo, tivemos um número muito reduzido de sessões, apenas duas, com duração média de 2 horas cada uma. Essa limitação não chegou a comprometer a qualidade dos artefatos gerados, mas acreditamos que, a depender do grau de complexidade da aplicação em desenvolvimento, um número maior de sessões pode ser necessário.

As atividades de teste foram aplicadas apenas ao componente Gerenciador de Hotéis, por ser o componente mais complexo. Este componente exporta uma interface, IFHotel, que comunica aos seus clientes os serviços que ele é capaz de fornecer, tais como, fazer reserva, cancelar reserva, incluir quarto etc. Cinco classes fazem parte da estrutura interna do componente e colaboram entre si para entregar os serviços declarados na interface do componente. O componente foi implementado usando a linguagem Java, bem como seus casos de teste e oráculos. Usamos a ferramenta Junit (www.junit.org) para implementar os casos de teste e oráculos. O estudo de caso será detalhado nos exemplos das próximas seções.

5. Planejamento dos Testes

O planejamento dos testes pode ser iniciado logo que os requisitos tenham sido definidos. Neste momento, os artefatos de análise (modelo conceitual e diagrama de casos de uso) estão sendo elaborados e são fontes de informações importantes para o plano de teste. Desenvolver o plano de teste neste momento permite que se tenha idéia da dimensão da tarefa de teste logo no início do projeto, permitindo que a distribuição de recursos durante o processo de desenvolvimento do software seja feita de forma mais consciente e racional.

No planejamento, são tomadas decisões do tipo [14]: quem executará os testes? Que partes serão testadas? Quando os testes serão executados? Que tipos de teste serão executados? O quanto cada parte será testada? Ao tomar todas essas decisões, estamos determinando os

recursos necessários, os métodos utilizados e a qualidade dos resultados do esforço de teste. Ao final dessa etapa, o plano de teste do componente deve estar concluído, indicando quantos casos de teste deverão ser desenvolvidos para cada caso de uso, quem desenvolverá e executará cada caso de teste e quando isso será feito.

Exemplo 1 (Planejamento). Durante o planejamento, utilizamos a técnica de análise de riscos para definir quais funcionalidades do componente seriam testadas e quantos casos de teste poderiam ser desenvolvidos para cada funcionalidade. A partir da análise de risco chegamos à conclusão que funcionalidades como Fazer Reserva e Cancelar Reserva eram mais críticas para o funcionamento do sistema, portanto demandavam um esforço de teste maior que outras funcionalidades, como, por exemplo, Incluir Quarto. Para estas funcionalidades mais críticas, ou de uso mais freqüente, foi desenvolvida uma quantidade maior de casos de teste. No total foram implementadas 6 classes de teste e 23 casos de testes distribuídos entre as classes como mostra a Tabela 1. Além das 6 classes de teste construídas para testar as funcionalidades do componente, foram implementadas ainda mais duas classes de teste para fazer teste de unidade das classes Quarto e TipoQuarto.

Classe de Teste	Casos de Teste
Fazer Reserva	5
Remover Tipo Quarto	4
Cancelar Reserva	4
Incluir Quarto	4
Incluir Tipo Quarto	3
Alterar Tipo Quarto	3

Tabela 1: Quantidade de Casos de Teste Implementados por Classe de Teste.

Nos exemplos seguintes, consideraremos apenas a funcionalidade Fazer Reserva.

6. Especificação dos Testes

A especificação dos testes pode ser iniciada logo que a especificação do componente tenha sido iniciada. A especificação do componente contém informações importantes sobre a solução adotada para o problema e é usada para derivar os casos de teste, dados e oráculos.

6.1 Selecionando os Casos de Teste

Teste exaustivo é na maioria das vezes inviável. Já que os diagramas de seqüência fornecem uma visão dos diversos cenários de uso do componente, resolvemos utilizar, neste método, os diagramas de seqüência para decidir quais cenários serão testados. O planejamento nos diz quantos cenários de cada funcionalidade do componente serão testados, mas não nos diz quais são esses cenários. Usamos então a combinação de duas técnicas de teste, TOTEM e Teste Estatístico (Cleanroom), para selecionar os cenários de uso do componente a serem testados.

Embora a técnica TOTEM tenha se mostrado muito interessante na geração dos modelos de teste a partir dos diagramas de seqüência, percebemos que a construção de diagramas de seqüência que englobam tanto o cenário principal quanto os cenários alternativos não é uma prática comum, nem tão pouco é uma atividade trivial. Propomos então adaptar a técnica para gerar a expressão regular a partir de vários diagramas de seqüência. Neste caso, cada

diagrama deverá representar um cenário de uso diferente, e a expressão regular gerada no final representará então todos os possíveis cenários de uso extraídos dos diagramas de seqüência.

A técnica TOTEM não apresenta um critério de seleção de casos de teste bem definido. Dessa forma, a fim de fornecer critérios de seleção dos cenários, e possibilitar uma seleção mais automática e direcionada dos cenários de uso, incorporamos à técnica TOTEM o aspecto estatístico da técnica de teste usada no Cleanroom.

A técnica de teste usada no Cleanroom propõe a construção de um modelo de uso do sistema que representa todos os possíveis usos do sistema e suas probabilidades de ocorrência. Este modelo é expresso normalmente por meio de um grafo direcionado, representando uma cadeia de Markov, onde estados são conectados através de arcos de transição. Cada arco representa um estímulo para o sistema, que o faz mudar de estado, e possui um valor de probabilidade associado. Os casos de teste são gerados percorrendo-se o modelo, partindo-se do seu estado inicial até o estado final. A seqüência de estímulos que leva o sistema do seu estado inicial ao estado final, através de um determinado caminho no modelo, é definida baseando-se nas probabilidades das transições.

Ao combinar as duas técnicas, derivamos um modelo de uso da expressão regular obtida a partir dos diagramas de seqüência. Neste modelo de uso são inseridos dois vértices representando o início e o fim da seqüência de troca de mensagens entre os objetos. Cada troca de mensagem dá origem também a um novo vértice. As transições são rotuladas com uma mensagem, que é uma chamada a uma operação da classe, no formato Classe.Operação. As transições devem conter ainda uma probabilidade de ocorrência variando de 0 a 1. O fim da seqüência de troca de mensagens dá origem a uma transição ligando o vértice da última chamada de operação ao vértice que representa o fim da seqüência. É importante ressaltar que todo este processo de geração de expressões regulares e conversão para modelo de uso deve ser automatizado, a fim de aumentar as chances de aplicação prática deste método.

É importante notar que, no nosso grafo, as transições que chegam ao vértice final são rotuladas com um comentário que dá um significado textual para o caminho percorrido no grafo. Essas transições representam sempre o fim da troca de mensagens representada nos diagramas de seqüências. Em termos de automação, o rótulo inserido nessas transições não está adequado. É importante definir uma forma que esteja condizente com os demais rótulos do grafo. Esse é um ponto importante que deve ser melhorado em trabalhos futuros.

Para selecionar os caminhos do grafo de forma automática, podemos usar um algoritmo que percorra o grafo e, em cada nó visitado, use uma função de distribuição de probabilidade que selecione com chances diferentes, de acordo com as probabilidades atribuídas, o próximo arco a ser seguido. Para isto, a soma total das probabilidades dos arcos deixando o nó tem que ser 1. A função de probabilidade é facilmente implementada.

Vale ressaltar que esta técnica usada no Cleanroom pode garantir que os usos esperados com maior freqüência pelo sistema sejam testados. Porém, pode-se também optar por uma combinação de escolha desta forma com uma escolha mais determinística, constituída de um ou mais cenários indicados pelos usuários.

Tendo selecionado os caminhos ou casos de teste que serão executados, o próximo passo é selecionar dados que nos levem por esses caminhos. Entretanto, a técnica proposta para gerar os oráculos de teste na TOTEM pode facilitar a escolha dos dados. Por esse motivo, estamos gerando os oráculos primeiro e em seguida selecionando os dados de teste.

Exemplo 2 (Seleção de Casos de Teste). Para selecionar os casos de teste, usamos os diagramas de seqüência produzidos durante a modelagem dos componentes. A Figura 2 mostra o diagrama de seqüência que representa o cenário principal para o caso de uso Fazer Reserva. Para este caso de uso foram produzidos ainda mais 4 diagramas de seqüência

representando os cenários alternativos: hotel inexistente, tipo de quarto inexistente, período inválido e indisponibilidade de quartos. A expressão regular obtida é mostrada a seguir:

$$fazerReserva_{GerenciadordeHoteis}.getHotel_{GerenciadordeHoteis}$$
$$+$$
$$fazerReserva_{GerenciadordeHoteis}.getHotel_{GerenciadordeHoteis}.fazerReserva_{Hotel}.$$
$$getTipoQuarto_{Hotel}$$
$$+$$
$$fazerReserva_{GerenciadordeHoteis}.getHotel_{GerenciadordeHoteis}.fazerReserva_{Hotel}.$$
$$getTipoQuarto_{Hotel}.isPeriodoReservaOk_{Hotel}$$
$$+$$
$$fazerReserva_{GerenciadordeHoteis}.getHotel_{GerenciadordeHoteis}.fazerReserva_{Hotel}.$$
$$getTipoQuarto_{Hotel}.isPeriodoReservaOk_{Hotel}.$$
$$verificarDisponibilidade_{Hotel}$$
$$+$$
$$fazerReserva_{GerenciadordeHoteis}.getHotel_{GerenciadordeHoteis}.fazerReserva_{Hotel}.$$
$$getTipoQuarto_{Hotel}.isPeriodoReservaOk_{Hotel}.\ verificarDisponibilidade_{Hotel}.$$
$$gerarCodReserva_{Hotel}.Reserva_{Reserva}$$

Esta expressão é uma soma de produtos, onde cada termo representa um cenário de uso do componente. Por exemplo, o termo 3 representa o cenário onde a reserva não pode ser criada porque o período não é válido. O modelo de uso para esta expressão é mostrado na Figura 3. Neste modelo de uso, o vértice 0 representa o início da seqüência de troca de mensagens e o vértice 1 representa o final dessa seqüência. Cada cenário é representado por um caminho que pode ser percorrido no grafo, iniciando-se no vértice 0 e terminando no vértice 1. Por exemplo, o caminho que passa pelos vértices 0,2,3,4,5,1 representa o cenário onde a reserva não é criada porque o tipo de quarto não existe no hotel desejado.

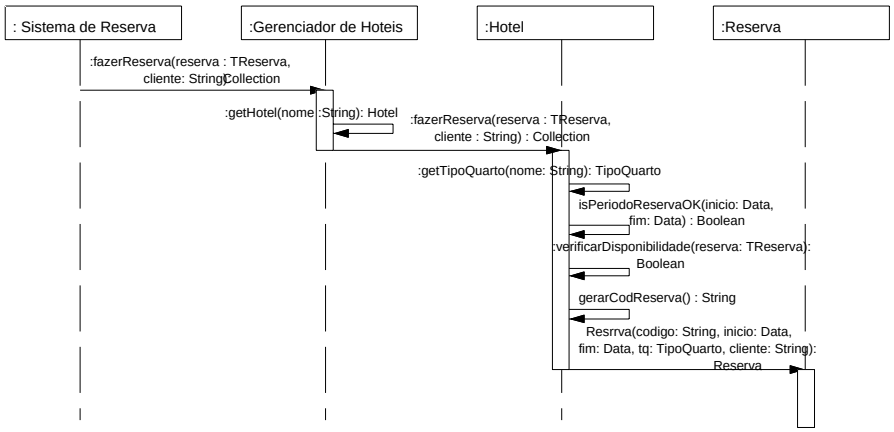

Figura 2: Diagrama de seqüência para o Caso de Uso Fazer Reserva - Cenário Principal: A reserva é criada com sucesso

É importante notar que as transições foram ponderadas com um valor de probabilidade. As probabilidades são atribuídas a fim de tentar garantir que os testes reflitam ou cubram os

usos mais esperados para o sistema. Por exemplo, ao atingir o vértice 3, o grafo pode tomar dois caminhos diferentes: ir para o vértice 1, indicando que o uso do componente é finalizado porque o hotel solicitado não pertence à cadeia, fato que tem 10% de chance de ocorrer, ou ir para o vértice 4, dando continuidade ao uso, o que tem 90% de chance de acontecer. Essas probabilidades podem ser definidas baseando-se no conhecimento dos usuários sobre o domínio do problema ou em dados históricos obtidos do uso de outras aplicações pertencentes ao mesmo domínio.

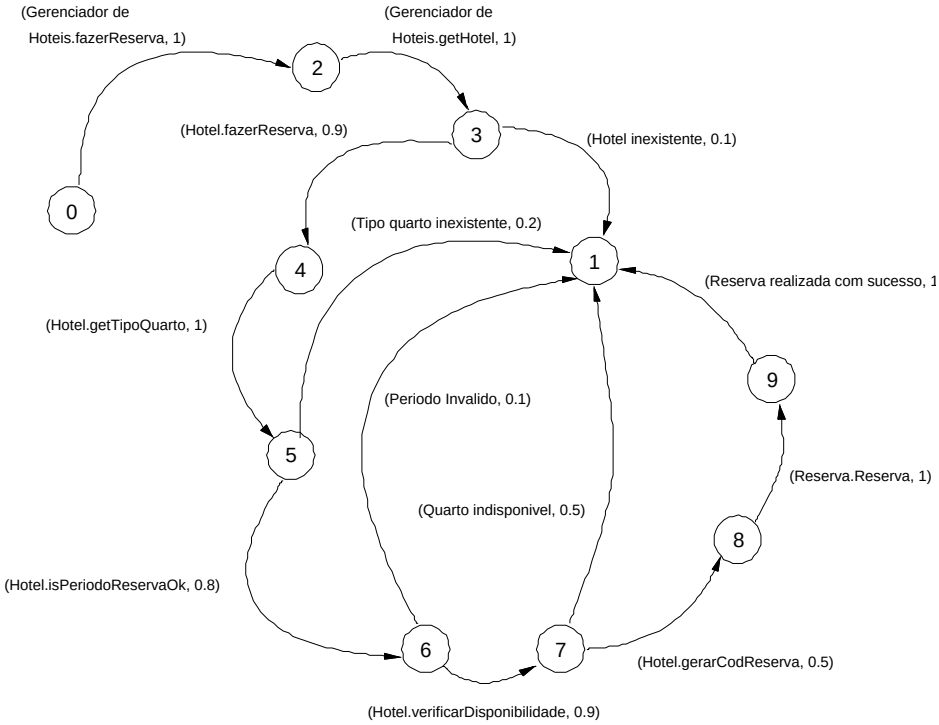

Figura 3: Modelo de Uso para o Caso de Uso Fazer Reserva

Neste exemplo, exercitamos todos os caminhos possíveis no grafo, mas, numa situação onde não houvesse recursos suficientes para cobrir o grafo totalmente, poderiam ser selecionados alguns caminhos baseando-se nas probabilidades definidas no grafo. Dessa forma, em cada vértice que se ramifica, seria escolhida a ramificação com maior probabilidade de ocorrência. A execução da etapa de planejamento é fundamental para definir o esforço de teste possível de ser realizado com os recursos disponíveis.

6.2 Gerando os Oráculos

A definição precisa dos oráculos de teste é imprescindível para automatizar a atividade de teste. A técnica TOTEM propõe que seja construída uma tabela de decisão para cada expressão regular que representa os cenários de uso do componente. Esta tabela deve conter as condições de realização do uso, especificadas em OCL, e as ações que serão tomadas pelo componente diante da ocorrência do uso. As principais fontes de informação para construção desta tabela de decisão são as pré e pós-condições definidas para as operações das classes que fazem parte do componente.

Para cada termo da expressão regular, é necessário identificar suas condições de execução e expressá-las em OCL. É importante ressaltar que a especificação OCL é construída com base

no modelo de informação - parte "visível ao usuário do componente".

Tendo identificado as condições de execução de cada cenário do caso de uso, o próximo passo é definir que mudanças de estado ocorrem no componente com a execução do mesmo. Além disso, deve-se identificar que mensagens são retornadas para o ator do caso de uso.

Por fim, a tabela de decisão que resume todas as informações necessárias para a construção dos casos de teste e oráculos do caso de uso é produzida. Cada linha da tabela representa um termo na expressão regular, que representa um cenário de uso. Para cada termo são então associadas as condições de execução, as mensagens retornadas para o usuário e a indicação se a sua execução causa ou não uma mudança de estado no componente.

As informações coletadas durante esta etapa do processo poderão auxiliar na seleção dos dados de teste e servirão de base para a implementação dos casos de teste e dos oráculos.

Exemplo 3 (Geração de Oráculos de Teste). Para cada termo da expressão regular mostrado no Exemplo 2, é necessário identificar suas condições de execução e expressá-las em OCL. Para exemplificar, mostramos a condição de execução, expressa em OCL, para o termo 3 da expressão regular para o caso de uso Fazer Reserva[6]. Este termo indica que o hotel desejado pertence à rede de hotéis, o tipo de quarto desejado existe no hotel informado, mas o período informado para reserva não é válido. Para este caso, temos a seguinte condição de execução:

```
C:  Gerenciador de Hoteis.hoteis→exists
      (h: Hotel | h.nome=reserva.hotel and
                  (h.tiposQuarto→exists
                   (tq: TipoQuarto | tq.nome=reserva.tipoQuarto) and
                   (reserva.dInicio≤Today() or
                    reserva.dInicio≥reserva.dFim)))
```

Tendo identificado as condições de execução de cada cenário do caso de uso, o próximo passo é definir que mudanças de estado ocorrem no componente com a execução do caso de uso. Além disso, deve-se identificar quais mensagens são retornadas para o ator do caso de uso.

Para o exemplo do caso de uso Fazer Reserva, temos as seguintes mensagens possíveis:

I : Hotel não cadastrado.
II : Tipo de quarto não existe no hotel desejado.
III : Período inválido.
IV : Nenhum quarto disponível do tipo desejado no período e hotel informados.
V : Reserva criada com sucesso.

A mudança de estado prevista para a execução deste caso de uso é a seguinte[7]:

```
Gerenciador de Hoteis→exists
    (h: Hotel | h.nome = reserva.hotel and
                (h.reservas→ select(r | r.codigo=codReserva and
                                        r.dFim=reserva.dFim and
                                        r.dInicio=reserva.dInicio and
                                        r.tipoQuarto.nome=reserva.tipoQuarto and
                                        r.cliente=reserva.cliente and
                                        r.cancelada = false)→size = 1))
```

A tabela de decisão que resume todas as informações necessárias para a construção dos casos

[6]O contexto usado na condição é Gerenciador de Hoteis : : fazerReserva
[7]O contexto dessa expressão é também Gerenciador de Hoteis : : fazerReserva

de teste e oráculos do caso de uso Fazer Reserva é mostrada na Tabela 2.

Versões	Condições					Ações					
						Mensagem					Mudança de Estado
	A	B	C	D	E	I	II	III	IV	V	
1	Sim	Não	Não	Não	Não	Sim	Não	Não	Não	Não	Não
2	Não	Sim	Não	Não	Não	Não	Sim	Não	Não	Não	Não
3	Não	Não	Sim	Não	Não	Não	Não	Sim	Não	Não	Não
4	Não	Não	Não	Sim	Não	Não	Não	Não	Sim	Não	Não
5	Não	Não	Não	Não	Sim	Não	Não	Não	Não	Sim	Sim

Tabela 2: Tabela de Decisão para o Caso de Uso Fazer Reserva

6.3 Selecionando os Dados

Selecionar os dados de entrada para um caso de teste é ainda um dos maiores problemas enfrentados por testadores de software, principalmente dados que sejam capazes de revelar comportamentos anômalos. A natureza variável dos dados envolvidos no teste de um componente dificulta muito a geração automática destes dados, especialmente por estarmos tratando de objetos. No método proposto, limitamo-nos a fornecer apenas orientações sobre como selecionar os dados necessários para a execução dos casos de teste.

Dentre as técnicas sistemáticas existentes para seleção dos dados de teste, pode-se destacar a técnica de particionamento por equivalência [16,2]. Essa técnica parte do princípio que os dados de entrada de um programa podem ser agrupados em classes que apresentam características comuns e que o programa se comporta da mesma forma para todos os membros de uma mesma classe. Usando essa técnica, o trabalho de selecionar os dados de teste consiste em identificar as partições e escolher dados particulares dentro de cada partição. A identificação das partições deve ser baseada na especificação e documentação do software. Neste método, podemos usar as condições de execução definidas durante a geração dos oráculos para identificar partições adequadas ao teste. A escolha dos dados tanto pode ser feita de forma aleatória, como de forma mais direcionada, a fim de obter dados mais prováveis de revelar erros. Na escolha direcionada consideramos os dados encontrados nos limites da partição, por representarem normalmente valores atípicos, e dados considerados típicos, encontrados no meio da partição.

Exemplo 4 (Seleção de Dados de Teste). Considere os Exemplos 2 e 3. O cenário exemplificado para seleção de dados é a situação onde a reserva não é realizada porque o período não é válido. Um período inválido é caracterizado da seguinte forma:

- Data de início da reserva é menor ou igual à data corrente;
- Data de início da reserva é maior ou igual à data final da reserva.

Para a primeira situação, identificamos três partições:

- Data de início da reserva é anterior à data corrente;
- Data de início da reserva é igual à data corrente;
- Data de início da reserva é posterior à data corrente;

A técnica propõe que sejam selecionados dados nos limites e no meio da partição. Seguindo esta orientação, para a primeira partição, selecionamos a data de início para um dia antes da data corrente e para vários dias antes da data corrente, por exemplo, dez dias antes. A segunda partição tem apenas uma data disponível, que é a data de início igual à data corrente.

Então selecionamos também esta data para executar o caso de teste. Por fim, a terceira partição não apresenta dados que levem o programa a percorrer o caminho que estamos interessados em testar, porque todos os dados desta partição nos leva a um período válido. Então, nenhum valor foi selecionado desta partição. Em resumo, para testar a situação em que a reserva não é realizada porque o período é inválido, selecionamos os seguintes dados:

- Início da reserva igual à data de ontem.
- Início da reserva igual a 10 dias antes da data atual.
- Início da reserva igual à data atual.

O mesmo processo foi aplicado para selecionar os dados que serviram de entrada para os demais casos de teste.

7. Construção, Execução e Verificação de Resultados

Nesta etapa, usamos as informações geradas até o momento para implementar os casos de teste e oráculos. As tabelas de decisão construídas durante a geração dos oráculos são muito úteis.

Os casos de teste e oráculos para cada caso de uso são implementados através de uma classe de teste. Usamos principalmente as informações existentes na tabela de decisões, elaborada durante a especificação dos oráculos, para implementar esta classe. A classe contém um método para testar cada cenário definido na tabela de decisão. Para cada cenário, implementamos no método correspondente as condições de execução definidas na tabela. Por fim, verificamos a mensagem que é retornada e a ocorrência ou não de mudança de estado e comparamos com as definições da tabela para indicarmos se o teste obteve sucesso ou não.

Exemplo 5 (Implementando os Testes). Para exemplificar a construção dos testes, vamos analisar alguns métodos da classe TestaFazerReserva. No Código 1, mostramos o método testFazerReservaHotelInvalido(). Neste método, é testada a situação onde uma reserva não é criada porque o hotel informado não é válido.

```
1   // Realizar reserva em um hotel invalido
2   public void testFazerReservaHotelInvalido() {
3           // Define o hotel da reserva
4           tRes.setHotel("Hotel Ouro Branco");
5
6           // Tenta realizar a reserva no hotel definido
7           Vector r = (Vector)g.fazerReserva(tRes);
8
9           // Verifica se a reserva foi realizada
10          assertEquals(new Integer(4), r.elementAt(0));
11          assertEquals(null, g.getHotel("Hotel Ouro Branco"));
12  }
```

Código 1: Método testFazerReservaHotelInvalido() da classe TestaFazerReserva

Na linha 4, definimos o hotel onde desejamos realizar a reserva. Neste caso, de acordo com as condições estabelecidas na tabela de decisão, o hotel definido não faz parte da rede de hotéis, portanto, não está registrado no gerenciador de hotéis, representado pelo objeto g, referenciado na linha 7. Nesta linha 7, o caso de teste é executado através da chamada ao método fazerReserva(), definido na interface GerenciadorDeHoteis. Na linha 10 verificamos se o retorno do método fazerReserva() está correto. Nesta situação, em que o hotel onde se pretende fazer a reserva não pertence à rede, espera-se que o código retornado pelo método

seja 4. Essa informação é obtida da especificação da operação fazerReserva(). A linha 11 analisa se ocorreu mudança no estado do componente. Neste caso, nenhuma mudança é esperada (de acordo com a tabela de decisão), então verificamos se o hotel onde tentamos realizar a reserva continua não fazendo parte da rede, ou seja, o hotel não está registrado no Gerenciador de Hotéis, portanto, esperamos que o retorno do método getHotel() seja null.

Vejamos agora a situação onde a reserva não é realizada porque o tipo de quarto informado não é válido no hotel onde se deseja fazer a reserva. O método que testa esta situação é mostrado no Código 2.

```
1   // Fazer reserva para um tipo de quarto inexistente no hotel
2   public void testFazerReservaTipoQuartoInvalido() {
3
4       // Define o hotel e otipo de quarto da reserva
5       tRes.setHotel(nomeHotel);
6       tRes.setTipoQuarto("Simples");
7
8       // Tenta realizar reserva com os dados definidos
9       Vector r = (Vector)g.fazerReserva(tRes);
10
11        // Verifica se a reserva foi realizada
12  assertEquals(new Integer(1), r.elementAt(0));
13      assertEquals(null, h.getTipoQuarto("Simples"));
14  }
```

Código 2: Método testFazerReservaTipoQuartoInvalido() da classe TestaFazerReserva

As linhas 5 e 6 definem o hotel e o tipo de quarto da reserva. O hotel é um hotel pertencente à rede de hotéis, portanto está registrado no Gerenciador de Hotéis. Entretanto, o tipo de quarto não deve existir no hotel. Essas informações foram tiradas da seção Condições, da tabela de decisão. Na linha 9, o caso de teste é executado através da chamada ao método fazerReserva(), da interface GerenciadorDeHoteis. As linhas 12 e 13 analisam o resultado da execução do caso de teste. Na linha 12, é verificado se o retorno do método está correto. Esperamos neste caso que seja retornado o código 1, de acordo com a especificação da operação fazerReserva(). A linha 13 verifica se ocorreu alguma mudança de estado no componente. Mais uma vez, nenhuma mudança é esperada, então esperamos que o tipo de quarto continue não existindo no hotel informado, portanto é esperado que o retorno do método getTipoQuarto(), da classe Hotel, seja null.

A execução dos casos de teste detectou apenas alguns poucos "bugs" na implementação das classes. Nenhum erro nas funcionalidades do componente chegou a ser detectado. Acreditamos que isso tenha acontecido por três motivos principais:

- A complexidade das funcionalidades do componente é baixa.
- A qualidade da especificação do componente está muito boa.
- A distância existente entre a especificação OCL e a linguagem de programação Java é pequena, o que facilita a produção de um código mais correto.

Enfim, o fato de não ter detectado bugs não implica que os testes planejados foram ineficazes. Na prática, processos onde existe um comprometimento em desenvolver o software correto desde o início (correto por construção) são caracterizados por um número mínimo de bugs detectados ao final [15]. Isto ocorre devido, principalmente, às sucessivas revisões empregadas. As atividades preliminares de teste auxiliam bastante na melhoria da qualidade de artefatos e implementabilidade do sistema. No final, os testes automáticos gerados podem constituir uma poderosa ferramenta de documentação (especificação executável do sistema)

que pode ser usada não só na certificação da qualidade do produto, mas também em testes de regressão decorrentes da acomodação de novas funcionalidades.

Empacotamento. Com o empacotamento pretendemos que os artefatos e resultados dos testes realizados sejam reunidos em um pacote e disponibilizados junto com o componente. Acreditamos que essas informações sejam de grande valia para os clientes dos componentes, já que estes poderão executar novamente os testes já realizados e até mesmo utilizar novos dados, o que contribui para aumentar a confiança dos clientes no componente que eles estão utilizando. Consideramos que os artefatos de teste poderão ser disponibilizados como um componente à parte, tendo sua interface composta pelos métodos especificados nos oráculos e pela especificação dos dados de teste utilizados.

8. Conclusões

Apresentamos neste trabalho, um método de teste para verificação de componentes que cobre cada uma das principais etapas de um processo de teste e está integrado a um processo de desenvolvimento de componentes. Embora o método tenha sido desenvolvido visando a verificação de componentes, pode-se perceber que ele pode ser facilmente adaptado e utilizado para verificar software orientado a objetos. O método ainda tem a vantagem de sugerir a disponibilização para os clientes dos componentes dos casos de teste, oráculos e dados que foram usados na execução dos testes, na forma de um componente à parte.

Por fim, produzimos um estudo de caso, que mesmo simples, nos deu as primeiras impressões sobre a utilização do método:

Quanto à quantidade e utilidade dos artefatos a produzir. O processo de desenvolvimento que escolhemos sugere a produção de poucos artefatos, ou seja, Componentes UML sugere que sejam criados apenas artefatos que nos ajudem a compreender melhor o domínio do negócio e a descobrir uma solução adequada para o problema. Procuramos desenvolver um método que usasse apenas os artefatos já sugeridos na metodologia de desenvolvimento para a produção dos testes. Dessa forma, adicionamos apenas a produção das expressões regulares a partir dos diagramas de seqüência e derivamos modelos de uso a partir das expressões regulares. Na prática, as expressões regulares e os modelos de uso são modelos equivalentes, apenas apresentados de formas diferentes. Sendo assim, efetivamente, apenas um novo artefato foi gerado com a finalidade de se desenvolver os testes dos componentes.

Quanto ao grau de dificuldade de aplicar o método. O fato do método ter sido sistematicamente inserido no processo de desenvolvimento facilitou significativamente a sua utilização prática, visto que temos uma definição das etapas que devemos seguir, das atividades que temos que realizar e da seqüência em que esse processo acontece. Visto de forma isolada, fora do contexto do processo de desenvolvimento, acreditamos que o método apresentaria um grau de dificuldade maior em termos de aprendizagem e aplicação prática.

Quanto à manutenção do sistema face aos diferentes artefatos desenvolvidos. O problema de manter os artefatos atualizados face às mudanças ocorridas no software não foi resolvido com este método. Obviamente aumentamos o número de artefatos produzidos, portanto o trabalho de manter os artefatos também cresceu. Entretanto, acreditamos que o desenvolvimento de ferramentas de suporte ao método possa ao menos minimizar o impacto da mudança sobre os artefatos de teste, ou mesmo eliminá-lo por completo.

É necessário ainda fazer ajustes e complementar alguns aspectos do método aqui apresentado, especialmente os aspectos relacionados à automação e abrangência do método. A construção de ferramentas computacionais de suporte é fundamental para viabilizar sua aplicação prática.

Referências

[1] S. Barbey, D. Buchs, and C. Peraire. A theory of specification-based testing for object-oriented software. *Lecture Notes in Computer Science*, 1150: 303–320, 1996.

[2] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 2nd edition, 1990.

[3] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, 1995.

[4] S. Beydeda and V. Gruhn. An integrated testing technique for component-based software. In *International Conference on Computer Systems and Applications*, pages 328–334. IEEE Computer Society Press, 26–29 Junho 2001.

[5] Lionel Briand and Yvan Labiche. A UML-based approach to system testing. *Lecture Notes in Computer Science*, 2185: 60–70, 2001.

[6] John Cheesman and John Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

[7] Philippe Chevalley and Pascale Thevenod-Fosse. Automated generation of statistical test cases from UML state diagrams. In *Proc. of the 25th Annual Int. Computer Software and App. Conference (COMPSAC 2001)*, pages 61–72, Chicago, Outubro 2001. ACM Press.

[8] Roong-Ko Doong and Phyllis G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. on Soft. Eng. and Methodology*, 3(2): 101–130, 1994.

[9] Jean Hartmann, Claudio Imoberdorf, and Michael Meisinger. UML-based integration testing. *ACM Trans. on Software Engineering and Methodology*, 8(11): 60–70, 2000.

[10] Zhenyi Jin and Jeff Offutt. Integrating testing with the software development process. Technical Report ISSE-TR-95-112, George Mason University, Agosto 1995.

[11] Patrícia D. L. Machado. Testing from structured algebraic specifications. In *AMAST: 8th Int. Conf. on Algebraic Meth. and Soft. Technology*, volume 1816, *LNCS*. Springer, 2000.

[12] Patrícia D. L. Machado and Don T. Sannella. Unit testing for CASL architectural specifications. In *27th International Symposium on Mathematical Foundations of Compouter Science*, volume 2420, LNCS. Springer-Verlag, Agosto 2002.

[13] Eliane Martins, Cristina Toyota, and Rosileny Yanagawa. Constructing self-testable software components. In *Proceedings of the 2001 Int. Conf. on Dependable Systems and Networks (DSN '01)*, pages 151–160, Washington - Brussels - Tokyo, Julho 2001. IEEE.

[14] John D. McGregor and David A. Sykes. *A Practical Guide to Testing Object-Oriented Software*. Object Technology Series. Addison-Wesley, 2001.

[15] Stacy J. Prowell, Carmen J. TRammell, Richard C. Linger, and Jesse H. Poore. *Cleanroom Software Engineering: Technology and Process*. SEI. Addison-Wesley, 1999.

[16] I. Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, 1996.

[17] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[18] H. Watanabe, H. Tokuoka, W. Wu, and M. Saeki. A technique for analyzing and testing object-oriented software using coloured Petri nets. In *Asia Pacific Software Engineering Conference*, pages 182–195. IEEE Computer Society Press, 1998.

[19] J. Whittaker and M. Thomason. A Markov Chain Model for Statistical Software Testing. *IEEE Transactions on Software Engineering*, 20: 812–824, Outubro 1994.