Artigo apresentado no 8. Simpósio de Computação Tolerante a Falhas, Campinas, SP, 7-9/7/1999.

Construção de Classes Autotestáveis

Eliane Martins, Cristina Maria Toyota
Instituto de Computação – Unicamp
CP 6176 CEP 13083-970
Campinas SP
{eliane, toyota}@dcc.unicamp.br

1.1 Resumo

Componentes reutilizáveis devem ser testados várias vezes: durante o desenvolvimento e a cada vez que haja alguma alteração (na classe ou no ambiente), pois novas falhas podem ser introduzidas. Recriar os testes a cada vez que uma classe é reutilizada causa um impacto no desenvolvimento de novas aplicações, contrariando um dos objetivos da reusabilidade, que é melhorar a produtividade.
Para reduzir o tempo e os custos com testes de componentes de hardware, são usados mecanismos embutidos de autoteste, ou BIST (de Built-In Self-Test), no projeto desses componentes. Fazendo um paralelo com os circuitos VLSI, foi proposta a aplicação de técnicas de BIST a componentes O-O. Tem-se então a criação de uma *classe autotestável*, que conteria uma especificação de teste, usada para a seleção das entradas de testes, e um conjunto de métodos que auxiliam a realização dos testes. Neste texto é apresentada a descrição do protótipo de uma ferramenta que tem por objetivo auxiliar à construção de classes autotestáveis, bem como a automatização da geração, execução e analálise dos resultados dos testes.

1.2 Abstract

Tests have to be repeated many times in the lifecycle of a reusable class: during development and after each change (in the class or in the environment), since new errors can appear. Regenerating tests each time a class is reused can severely impact the development of new applications, which goes against another goal of reusability: enhance productivity.
For hardware components, Built-In Self-Test (BIST) helps reducing time and cost of testing. Based on the self-testing concept different approaches were proposed for the use of BIST in the development of O-O components. A BIST-classe contains a test specification, used for test generation, and a set of methods usefull during test execution. This text presents the main features of a prototype tool aimed at helping in the development of self-testing (or BIST) classes, and also to allow the automation of testing activities: selection, execution and results analysis.

2. Introdução

A tecnologia de objetos tem sido cada vez mais utilizada, principalmente devido ao seu potencial para a reutilização de componentes de software, permitindo dessa forma reduzir o custo de produção de sistemas. Para que o reuso seja bem sucedido, é fundamental que o componente reutilizado seja confiável. Uma forma de garantir a

confiabilidade, bem como outras qualidades de um componente é através de testes. Componentes reutilizáveis devem ser testados várias vezes [PK90, Wey98]: durante o desenvolvimento, toda a vez que forem utilizados em um novo contexto, e a cada vez que sofram alguma alteração. Portanto é essencial que esses componentes sejam testáveis, ou seja, fáceis de testar, para reduzir os custos para a realização dos testes.
Em sistemas orientados a objetos a testabilidade de é afetada pelas características intrínsecas desse paradigma, tais como herança, polimorfismo com ligação dinâmica, encapsulamento e ocultação da informação.

Fazendo-se uma analogia entre circuitos integrados e classes [Hof89], surgiu a proposta de se aplicar às classes as técnicas de projeto para testabilidade (DFT, do inglês *Design For Testability*) e mais o conceito de autoteste, usados no projeto de circuitos VLSI. Tem-se com isso a proposta para a criação de *classes autotestáveis* [Bin94b]. Da mesma forma que no hardware, em que circuitos extras são embutidos em um componente voltados unicamente para funções de teste, mecanismos similares também devem ser acrescentados a uma classe. Diferentemente do hardware, esses mecanismos não precisam permanecer durante a fase operacional do sistema, sua introdução ou não podendo ser controlada através de diretivas de compilação.

No entanto, enquanto no hardware os testes gerados podem ser reaplicados a cada reuso do componente, no software isso nem sempre é possível pois, na maioria das vezes, os componentes sofrem alguma alteração quando são reutilizados. Para permitir que o reuso possa ser aplicável também aos testes, Harrold e McGregor propuseram a técnica *incremental hierárquica* [HM92], que leva em conta a hierarquia de herança, permitindo que, em alguns casos, seja possível reutilizar casos de testes gerados para a classe nos testes de suas subclasses.

Combinando o uso do conceito de autoteste e a técnica incremental hierárquica, o objetivo desse estudo é não só melhorar a testabilidade de componentes reutilizáveis, mas também permitir que o reuso se estenda também aos testes, sempre que possível. Para verificar a viabilidade da proposta foi construída uma ferramenta, a ConCAT (de *C*onstrução de *C*lasses *A*uto*T*estáveis), que dá apoio automatizado tanto ao desenvolvimento quanto ao uso dessas classes.

O texto está organizado da forma descrita a seguir. A seção 2 apresenta uma visão geral dos teste, apresentando o conceito de testabilidade e o impacto do paradigma O-O sobre a mesma. A seção 3 trata do projeto visando testabilidade e o conceito de autoteste, e sua aplicação em sistemas O-O. A seção 4 apresenta os principais aspectos que influenciaram no projeto e implementação da ConCAT. A seção 5 dá uma visão geral das funcionalidades oferecidas pela ferramenta. A seção 6 mostra alguns trabalhos existentes e, finalmente, a seção 7 apresenta as conclusões e trabalhos futuros.

3. Testes e testabilidade

Esta seção apresenta uma visão geral sobre testes de software, o conceito de testabilidade, e de como esta é afetada pelas características da orientação a objetos.

3.1 Visão Geral dos testes

Testes constituem uma atividade de validação que consiste em exercitar o software, fornecendo-lhe entradas, com o objetivo de encontrar falhas[1]. O processo de teste compreende as seguintes atividades [Pos96, cap1]: geração, execução and avaliação dos testes.

Na geração de testes são escolhidos os dados de entrada a serem aplicados durante os testes. Dado que em software é geralmente impossível a realização de testes exaustivos, é necessário escolher um subconjunto, finito, do domínio de entrada (o conjunto de todas as entradas possíveis, que pode ser infinito), que seja adequado para revelar as falhas presentes no software. Como as falhas existentes no software são desconhecidas, a escolha dos dados é feita com base em *critérios*, os quais servem para definir o conjunto finito de elementos a serem exercitados durante os testes. Esses elementos são parte do *modelo de teste* [Sie91], o qual, na maioria dos métodos existentes, pode representar a estrutura ou a especificação do software. Quando o modelo representa a estrutura do software, os critérios de teste usados são ditos estruturais ou caixa branca. Já nos critérios de teste ditos funcionais, ou baseados na especificação ou ainda caixa preta, o modelo de teste representa o comportamento esperado do software e suas características operacionais, tais como desempenho, confiabilidade.

Na execução dos testes os dados de entrada são aplicados ao software e observar as saídas produzidas. Quando um componente de software é testado isoladamente, devem ser criados componentes adicionais para auxiliar a realização dos testes; esses componentes incluem os *drivers* e *stubs*. Um *driver* é usado para ativar o componente em teste, fornecendo-lhe as entradas e observando as saídas; os *stubs* são usados para substituir os componentes que são usados (chamados) pelo componente em teste [Pre97, 17.3].

Na avaliação dos testes deve-se determinar [Pos96]: (i) se as saídas observadas correspondem ao que era esperado, e (ii) se a seqüência de testes aplicada ao software foi adequada. Para detectar desvios com relação à especificação (i.e, *defeitos*), é necessário um mecanismo que julgue se a saída observada em resposta a uma dada entrada corresponde ao que foi especificado. Esse mecanismo é chamado de *oráculo*; sua determinação é muito importante, pois se um oráculo inexiste ou é pouco confiável, as conclusões extraídas dos testes também não são confiáveis.

A seqüência de testes (coleção de dados de entrada aplicados) é adequada quando o modelo foi coberto, segundo o critério de teste escolhido. Portanto, a determinação da adequação depende do modelo ser baseado na especificação ou na implementação. No atual estado da prática a adequação dos testes baseados na especificação é determinada informalmente, pois em geral a especificação usada é informal. No caso de testes estruturais pode-se usar ferramentas (instrumentadores ou analisadores dinâmicos) para fornecer informações sobre as instruções, ramos ou outros elementos do programa, que foram exercitados durante os testes.

O esforço necessário para a realização das atividades apresentadas vai depender da testabilidade do software (ou do componente) em teste. O conceito de testabilidade e o impacto da orientação a objetos sobre a mesma são apresentados no próximo item.

---

[1] Como ainda não existe um consenso quanto à terminologia a ser usada em português, os termos usados aqui são baseados em [LL87] onde: uma falha ("fault") é a causa direta de um erro; um erro é a parte do estado do sistema que pode levar a um defeito ("failure").

3.2 Testabilidade

Existem diversas definições para a testabilidade de sistemas de software; segundo o padrão IEEE610.12-1990 (IEEE Standard Glossary of Software Engineering Terminology), tem-se que a *testabilidade* é uma indicação do quanto (1) um sistema ou componente facilita tanto o estabelecimento de critérios de teste quanto a realização de testes para determinar se esses critérios foram satisfeitos; e (2) os requisitos são expressos de forma a permitir tanto o estabelecimento de critérios de teste quanto a realização de testes para determinar se esses critérios foram satisfeitos.
Em outras palavras, a testabilidade é uma qualidade do software que se refere à facilidade com a qual critérios de teste podem ser estabelecidos e satisfeitos. Um sistema testável requer menor esforço para a realização dos testes e melhora a eficácia dos mesmos, i.e, o seu potencial para encontrar falhas.
Dois aspectos chaves para a testabilidade são: controlabilidade e observabilidade. Isso significa que, para se testar um componente é preciso ter a capacidade de controlar suas entradas e observar suas saídas. Em geral é difícil conseguir uma boa controlabilidade e observabilidade, em especial se o componente está embutido em outros componentes. Nesse aspecto os sistemas orientados a objetos, devido a suas características intrínsecas, apresentam alguns obstáculos à testabilidade, conforme será visto a seguir.

3.3 Impacto da orientação a objetos sobre a testabilidade

Os conceitos de O-O dados nesta seção são baseados em [BR98]. Os termos classe, objeto e atributos, sendo amplamente conhecidos, não serão definidos aqui. Um objeto encapsula dados (i.e, uma coleção de atributos) e os algoritmos que manipulam esses dados; esses algoritmos são chamados de *operações*. As operações são implementadas como *métodos*[2]. Objetos interagem através do envio de *mensagens*; a recepção de mensagem faz com que seja executada a operação correspondente da interface pública do objeto recipiente. O envio de mensagem corresponde portanto à invocação de uma operação declarada na interface pública de um objeto. A interface pública de um objeto compreende o conjunto de operações que podem ser chamadas por outros objetos.
A seguir será visto o impacto sobre a testabilidade de mecanismos tais como: encapsulamento, herança, polimorfismo e ligação dinâmica, relacionamentos. As referências [SR90, PK90, RSTC94, KGH+95, Bin97, BBP98] foram usadas como base.

1. **Encapsulamento**

O encapsulamento descreve o empacotamento de uma coleção de itens. A classe é, usualmente, o nível de encapsulamento em abordagens O-O. O encapsulamento permite que detalhes internos da implementação sejam escondidos do mundo externo (ocultação da informação). As diferentes metodologias e linguagens oferecem diferentes opções de visibilidade das características de uma classe (atributos e operações). Em C++, por exemplo, tem-se características *públicas* (visíveis por todos os objetos), *privadas* (visíveis internamente ao objeto, somente) e *protegidas* (visíveis internamente ao objeto ou aos objetos de classes derivadas).
Apesar da vantagem que o encapsulamento e ocultação da informação trazem em termos de melhoria da modularidade, elas limitam a controlabilidade e observabilidade.

[2] Apesar de ter sido utilizado neste trabalho a linguagem C++, preferiu-se utilizar nesse texto o termo método em vez de função-membro, como eles são chamados nessa lingaugem. As funções usadas fora do escopo de uma classe são chamadas de funções globais.

Uma outra implicação é que a unidade de teste passa a ser a classe, e para testá-la é necessário criar uma instância sua, ou seja, um objeto. Mas nem todas as classes podem ser instanciadas diretamente, como é o caso das classes abstratas e genéricas.
Classes abstratas são projetadas para que outras classes herdem suas características, e não implementam essas características elas mesmas. Portanto elas não têm instâncias diretas, somente suas descendentes é que têm, ou seja, suas características só podem ser testadas através dos objetos instanciados a partir de suas descendentes.
Classes genéricas (ou *paramétricas*)[3] são definidas sem um tipo específico e devem ser parametrizadas para que se possa definir uma classe de um determinado tipo, e só então pode-se instanciar um objeto. Uma das dificuldades de se testar uma classe genérica está na escolha do tipo a ser selecionado para os testes que aumente o potencial para encontrar falhas.

2. **Herança**

Herança é um mecanismo que permite a uma classe (classe *derivada* ou *subclasse*) herdar características (métodos e atributos) de outra classe (classe *base* ou *superclasse*). Diz-se que há *herança múltipla* quando uma classe derivada herda característica de mais de uma classe base (estas constituem então suas classes *ancestrais*).
A herança é comumente usada de duas formas [BR98]: herança de implementação e herança de comportamento.
Na *herança de implementação* a classe derivada pode remover ou renomear algumas das características da classe base; em outras palavras, a classe derivada não é uma especialização da classe base, e portanto não é garantido que elas tenham o mesmo comportamento.
Na *herança de comportamento* a classe derivada se comporta como se fosse a classe base. À subclasse podem ser acrescentadas novas características além daquelas herdadas, e estas podem ser redefinidas ("overriden"). Por exemplo, pode ser dada uma nova implementação a uma operação herdada caso o seu comportamento não seja apropriado no contexto da subclasse.
A herança é um mecanismo muito importante para a reusabilidade mas apresenta uma série de dificuldades para os testes:

- A implementação do mecanismo de herança varia de uma linguagem para outra, portanto a estratégia a ser utilizada para os testes é altamente influenciada pela linguagem.
- Reuso implica em re-teste; mesmo que as características herdadas permaneçam inalteradas na subclasse, elas devem ser testadas no novo contexto. A herança de comportamento é melhor nesse caso, pois permite que as entradas de testes das partes que não foram modificadas possam ser reutilizados.
- A complexidade de uma classe vai aumentando quanto mais baixo ela estiver na hierarquia de herança e o uso de herança múltipla aumenta ainda mais essa complexidade. O aumento de complexidade implica em que o esforço necessário para a preparação dos testes é maior. Em especial para determinar que entradas de testes podem ser reaplicadas, quais devem ser modificadas. Obviamente que o uso de herança de implementação é um fator de complexidade a mais, podendo inclusive impedir o reuso de testes.

[3] Em C++ elas são declaradas como *template*.

- Modificações em uma ou mais ancestrais podem causar efeitos indesejáveis nas classes derivadas, tendo como implicação a necessidade de retestá-las.

3. **Polimorfismo e ligação dinâmica**

Polimorfismo significa "várias formas para o mesmo nome", e quando usado no contexto de orientação a objetos significa que diferentes tipos de objetos podem responder à mesma mensagem de maneiras diferentes. Existem diversas formas de se obter polimorfismo nas diferentes linguagens de programação; de acordo com a taxonomia usada em [BR98] pode-se ter polimorfismo:

- por coerção ("casting"): nesse caso a linguagem permite o mapeamento interno entre tipos, ou seja, se um contexto requer um elemento de um determinado tipo e um outro é fornecido, é verificado se é possível fazer uma conversão para o tipo desejado;
- por sobrecarga ("overloading"): nesse caso a linguagem permite que o mesmo nome de função seja usado mais de uma vez com diferentes tipos de parâmetros. A seleção da função apropriada é baseada no tipo dos parâmetros usados quando é feita a chamada;
- paramétrico: nesse caso a linguagem permite que uma única função trabalhe uniformemente com diferentes tipos. As classes genéricas, citadas anteriormente, são um exemplo desse tipo de polimorfismo;
- de inclusão (também chamado de polimorfismo de objeto [SR90]): nesse caso a linguagem permite que um objeto que é instância de uma subclasse possa substituir um objeto que é instância de uma de suas classes ancestrais. Em outros termos, um objeto pode ser usado seja como destinatário de mensagem, seja como parâmetro de alguma operação, no lugar de um objeto de uma classe ancestral.

As linguagens de programação orientadas a objetos implementam polimorfismo de inclusão e por sobrecarga associados ao mecanismo de herança. Por exemplo, em C++ um método declarado como virtual em uma hierarquia de classes pode ter diversas implementações; a versão a ser chamada é determinada em tempo de execução dependendo se o objeto é de uma classe derivada ou de uma de suas ancestrais.
A associação, em tempo de execução, entre uma operação e a sua implementação é denominada de ligação dinâmica ("dynamic" ou "late binding").
O uso de polimorfismo e ligação dinâmica acarreta nas seguintes dificuldades para os testes:

- a geração de testes baseados na implementação se torna mais complexa, pois o código a ser executado é mais dependente das condições de execução do que nas linguagens imperativas. Portanto uma análise estática do código para determinar as entradas de teste que satisfaçam a um determinado critério não é suficiente.
- O uso de herança de implementação pode causar a ocorrência de muitos erros de execução, pois a classe derivada não tem o mesmo comportamento de suas ancestrais. A dificuldade está em selecionar entradas de teste que permitam que esses esrros sejam detectados.
- A determinação de todos os usos polimórficos de um objeto ou característica é impossível.

4. **Relacionamentos**

Objetos podem se relacionar uns com os outros de diferentes formas [Sie96, cap.2]: um objeto pode conter outro objeto; objetos podem enviar mensagens para outros objetos; objetos podem ser executados concorrentemente; objetos podem ser compatíveis ou incompatíveis uns com os outros.
Um objeto pode conter outros objetos de diferentes formas: uma classe pode ser declarada internamente a outra; uma instância de um objeto é criada dentro de outro objeto; através da herança, as características de uma classe são compartilhadas pelas classes derivadas.
No envio de mensagens um objeto invoca um método de outro objeto, havendo opcionalmente o uso de argumentos e valores de retorno; objetos podem ser passados como argumentos ou como valor de retorno. O envio de mensagens está ligado com a à visibilidade: o envio de mensagem corresponde à invocação de métodos na interface pública de um objeto[4]. Se um objeto está encapsulado em outro, não sendo visível externamente, não pode receber mensagens de outros objetos.
Algumas linguagens oferecem capacidade de concorrência, permitindo com isso que objetos possam pertencer a diferentes "threads", e seus métodos podem ser executados independentemente uns dos outros. Objetos concorrentes podem trocar mensagens entre si, de forma síncrona ou assíncrona.
Compatibilidade diz respeito à capacidade de um objeto de enviar mensagem a outro com base na natureza da mensagem. Compatibilidade está relacionada com herança, polimorfismo e ligação dinâmica, já apresentados anteriormente.
A variedade e complexidade dos relacionamentos entre objetos causam o chamado problema da interdependência complexa [KGH+95], que acarreta as seguintes dificuldades para os testes:

- É difícil entender uma classe, pois ela depende de muitas outras.
- É difícil saber por onde começar os testes, pois não há uma hierarquia do tipo coordenador-subordinado, como nas linguagens impertivas.
- Torna-se muito complicado construir "stubs".
- É difícil avaliar o impacto de modificações sobre o sistema.

Dadas as vantagens oferecidas pela orientação a objetos para a melhoria da qualidade de software, a melhoria da testabilidade é extremamente importante para garantia da confiabilidade e manutenibilidade de sistemas desenvolvidos segundo esse paradigma. Na seção a seguir será visto como técnicas usadas no hardware podem ser úteis nesse caso.

3.4 Projeto visando a testabilidade de classes

Na área de hardware o *projeto visando a testabilidade* ou DFT (do inglês design for testability) engloba várias técnicas que têm por objetivo facilitar os testes de circuitos integrados[CM91, KBJ+96]. Componentes extras são introduzidos para melhorar o acesso a pontos internos do circuito durante os testes. Para evitar a necessidade de se usar um testador externo, pode-se ainda incorporar mecanismos para a geração de

[4] Em C++ existe a noção de funções ou classes amigas ("friends"). Nesta relação um componente externo à classe tem visibilidade às características privadas de uma classe sem manter com a mesma uma relação de herança.

padrões de teste e análise dos resultados. Tem-se assim o conceito de *autoteste* ou BIST (de Built-In Self Test). A interface com as funcionalidades de autoteste pode ser simples, de modo que estas possam ser facilmente utilizadas nas diversas fases de teste de um componente.

Tomando por base o que é feito em hardware, R.Binder propôs a aplicação da noção de projeto visando a testabilidade no desenvolvimento de sistemas orientados a objetos. A testabilidade deve ser levada em conta ao longo de todo o processo de software [Bin94b]. Nesse estudo o enfoque está nas modificações a serem introduzidas no código para se conseguir alta testabilidade.. O autor recomenda a construção de *classes autotestáveis*, ou seja, à classe em teste original devem ser acrescentadas funcionalidades que permitam não só melhorar a controlabilidade e observabilidade de sistemas O-O, mas também dêem apoio à geração e análise automática dos testes. A vantagem com relação ao hardware é que essas funcionalidades não precisam fazer parte do sistema final: sua inserção pode ser controlada com o uso de diretivas de compilação.

A figura 1 mostra a configuração usada nesse trabalho para a elaboração de classes autotestáveis. Os componentes extras são [Bin94b]:

- a *especificação de testes*, que é o modelo de base usado na geração dos testes;
- *mecanismos embutidos de teste* ou BIT (em inglês, Built-In Tests), usados no controle e observação do estado interno, bem como na análise dos resultados.

Além disso é também necessário construir um *driver*, responsável por: (i) gerar as entradas de testes a partir da especificação embutida na classe, (ii) executar os testes utilizando os mecanismos embutidos de teste e (iii) analisar os resultados obtidos.
Com base nesses conceitos R.Binder propõe diversas formas de se implementar o conceito de classes autotestáveis. Neste trabalho foi adotada a que considera a utilização de um *driver* único (denominado pelo autor de *driver genérico*), a partir do qual seriam gerados os *drivers* específicos para cada classe.

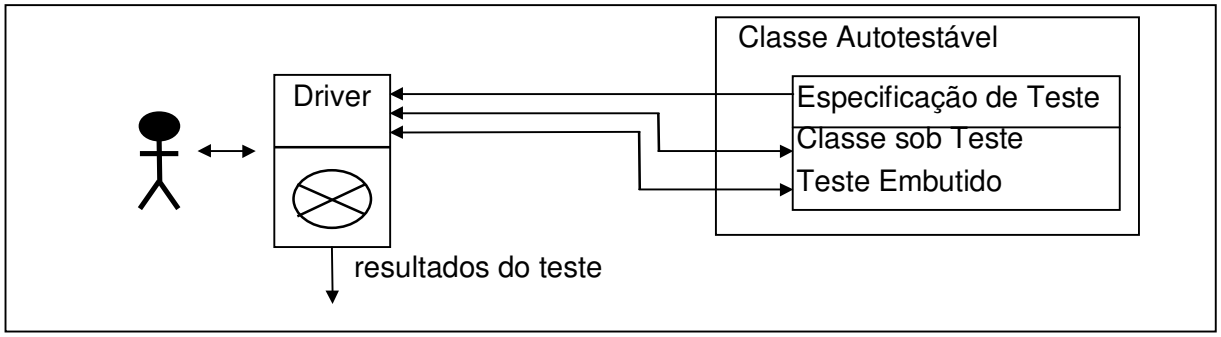

Figura 1: Classe autotestável.

A partir dessa arquitetura de alto nível foi projetado e implementado um protótipo de um sistema para a construção de classes autotestáveis. Para isso os seguintes pontos tiveram de ser definidos:

- o modelo de teste a ser utilizado;
- a linguagem para a especificação de teste;
- a forma de associação entre a especificação de teste e a classe;
- a forma de se instrumentar a classe com os mecanismos de teste embutidos
- o oráculo.

Esses pontos serão abordados nas próximas seções.

4. Estratégia adotada para os testes

A estratégia usada para a construção da ferramenta levou em conta os seguintes aspectos:

- Teste de classe

  A ferramenta visa auxiliar na melhoria da testabilidade de classes fornecendo um auxílio automatizado à realização de testes de classes.

- Teste baseado na especificação

  Optou-se pela utilização de um método de teste caixa-preta, denominado teste de fluxo de transação, definido em [Bei90, cap.4]. Uma transação, segundo o autor, é uma unidade trabalho do sistema, do ponto de vista do usuário. Transações representam ciclos de vida de um objeto, desde sua criação até sua destruição. O modelo de teste, é então o chamado grafo de fluxo de transação. O modelo mostra os diferentes modos de se criar um objeto, as diferentes tarefas ou processos que este pode realizar e os diferentes modos de destruí-lo. Cada caminho neste grafo representa uma transação. Os nós representam os processos em si (por exemplo, a execução de um método ou conjunto de métodos) e os arcos representam seqüências de execução válidas. O grafo de fluxo de transação pode ser construído a partir dos cenários de uso ("use cases") ou do modelo funcional da classe [Sie96]. Um exemplo de como construir esse modelo para uma classe é apresentado em [TM98].

  A escolha desse modelo atende uma das recomendações feitas em [Wey98], segundo a qual os testes de um componente reutilizável, tanto por seus desenvolvedores quanto por seus consumidores, devem procurar exercitar todos os usos possíveis desse componente, e esse é o objetivo dos testes desse tipo de modelo.

- Reuso de seqüências de testes

  A ferramenta deve possibilitar que o reuso seja também estendido aos testes. Assim, o usuário (ou consumidor) de uma classe poderia reutilizar seqüências de testes criadas pelo desenvolvedor (ou produtor) da classe, fazendo acréscimos ou alterações quando necessário. Como em O-O o reuso se dá geralmente através do mecanismo de herança, o que se pretende é que seqüências de teste geradas para a classe base possam ser utilizados nos testes das classes derivadas. Será usado aqui o mesmo conceito proposto por [Sie96], de se usar uma seqüência de teste incremental hierárquica, gerada segundo a *técnica incremental hierárquica* [HM92]. A técnica é dita hierárquica porque utiliza a ordenação parcial dada pela hierarquia de herança, e incremental porque utiliza os resultados dos testes de uma classe em um nível mais alto na hierarquia para reduzir os esforços necessários em

níveis subsequentes. Assim, para projetar a seqüência de teste para uma subclasse, as informações de teste da superclasse são incrementalmente atualizadas para refletir atributos e métodos novos ou redefinidos, conforme mostra a tabela 1, baseada em [Sie96, cap11].

Métodos	Descrição	Teste
Novos	São definidos na subclasse, e incluem aqueles que têm o mesmo nome de métodos da superclasse, mas têm parâmetros diferentes.	Realizar testes.
Herdados	São definidos na superclasse e não são nem sobrecarregados nem redefinidos na subclasse.	Testar somente se o método interage com métodos novos ou redefinidos.
Redefinidos	São definidos na superclasse e são sobrecarregados ou redefinidos na subclasse.	Testar reutilizando os modelos de teste funcionais da superclasse.

Tabela 1: Tipos de métodos e teste (ou reteste) exigido por cada tipo.

- Uso de Prolog na geração do *driver* de testes

  O *driver* genérico, que usa a especificação de testes para gerar o *driver* específico, foi desenvolvido em Prolog [CM87]. Os métodos de geração de teste são implementadas nessa linguagem. O uso de Prolog, uma linguagem lógica, em sistemas de testes não é novo [Ura83, Hof91] e apresenta diversas vantagens:

  - A linguagem possui um nível de abstração é mais alto do que linguagens imperativas como C, por exemplo, sendo mais adequada para o desenvolvimento rápido de protótipos.
  - A linguagem oferece mecanismo de *backtracking*, muito útil para implementar algoritmos de geração de testes e análise de resultados.
  - Facilita o acréscimo de novos estratégias de geração de testes.
  - Facilita a manutenção da ferramenta, pois o entendimento do sistema á bem mais fácil.

5. Descrição da ferramenta ConCat

5.1 Descrição Geral

A Figura 5-1 apresenta uma visão geral da ferramenta. As atividades dentro de uma oval são realizadas pelo usuário. Os itens a seguir apresentam os passos realizados para a construção e uso de classes autotestáveis.

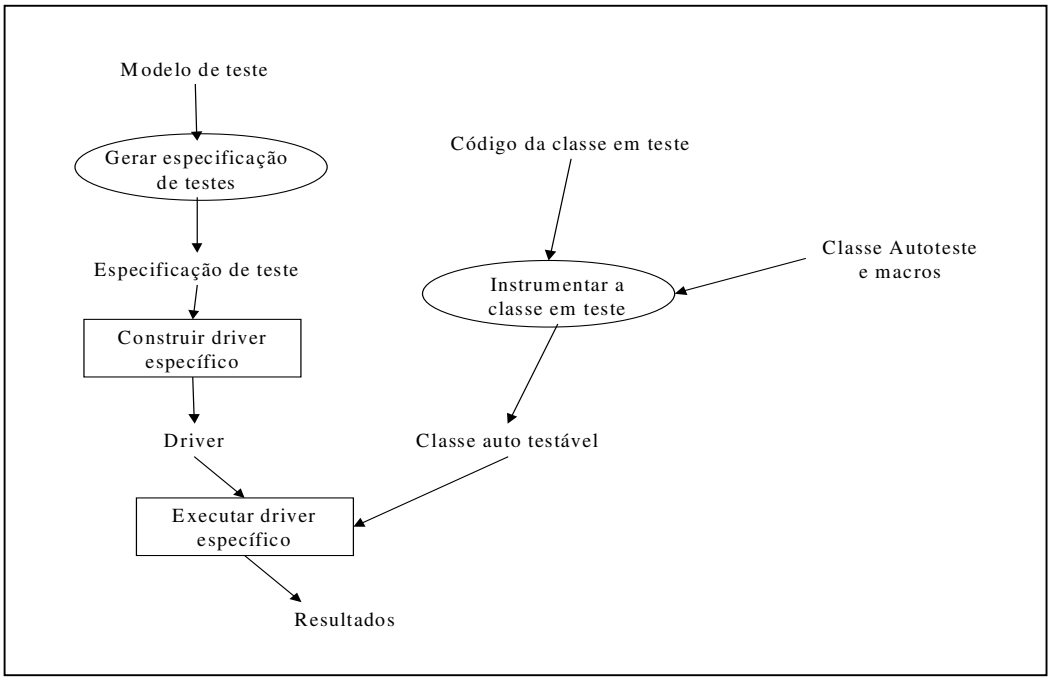

Figura 5-1. Visão geral da ConCAT

5.2 Construção da especificação de teste

O usuário deve construir inicialmente o modelo de fluxo de transação da classe e em seguida representar esse modelo na forma de fatos da linguagem Prolog. Esses fatos constituem a especificação de teste, cujos componentes são formato mostrados na figura 2.

```
Classe(   nome,
          Abstrata/concreta,          - indica se a classe é abstrata
          Nome_superclasse,
          Lista_arq_include  )        - lista de arquivos a serem incluídos na  compilação
Atributo( nome,
          Tipo                        - pode ser intervalo, conjunto, objeto ou ponteiro
          Dado1                       - para o tipo intervalo, indica o valor mínimo
          Dado2    )                  - para o tipo intervalo, indica o valor máximo
Método(   nome,
          Tipo_de_retorno,
          Classificação               - classificação dada conforme a tabela 1
          Nro_de_parâmetros)
Parâmetro(nome,
          Método_a_que_pertence,
          Tipo,                       - similar ao de atributo
          Dado1,
          Dado2   )
Nó(       identificador,
          Nó_inicial,
          Arcos_saída,
          Lista_de_métodos   )
Arco(     nó_origem,
          Nó_destino)
```

Figura 2 - Formato da especificação de teste

Uma vez fornecida a especificação de teste, esta deve ser associada à classe em teste (ou CeT), de forma a que toda vez que a classe seja reutilizada, sua especificação esteja disponível. Esta associação é feita nomeando-se o arquivo contendo a especificação de teste de acordo com um padrão: o nome da classe e a extensão "mt" representando modelo de teste. Desse modo, uma classe chamada "X" teria sua especificação de teste com o nome de "X.mt".

5.3 Instrumentar a classe em teste

Neste passo a CeT e seus métodos devem ser instrumentados, introduzindo-se capacidade de teste embutido: métodos de teste embutido e assertivas (ou predicados). Como essa capacidade de teste embutido deve ter visibilidade sobre o estado interno do objeto, a forma usada para tal foi criar uma classe abstrata, denominada **Autoteste**, contendo as definições dos métodos de teste embutido: um relator (ou de acesso, que permitem obter o estado do objeto) e outro para o teste de invariante da classe. Dessa forma garante-se que todas as classes autotestáveis terão uma interface padrão para interação com os mecanismos embutidos de teste. Cabe ao usuário redefinir a implementação desses métodos para a CeT. A classe Autoteste é mostrada na figura 3.

```
class Autoteste {
public:
        Autoteste ( ) { }
        ~Autoteste ( ) { }
        virtual void Testa_invariante ( ) { }
        virtual void Relator (FILE* arq)
};
```

Figura 3: A classe Autoteste.

O usuário deve também introduzir os predicados referentes às pré-condições e pós-condições de cada método. Para isso estão disponíveis as macros mostradas na figura 4. Esses predicados servem para avaliar, durante os testes, se o estado intermediário do objeto é válido ou não, e na atual versão da ferramenta é a única forma de oráculo existente.

```
#define Invariante(exp) \
  if (!(exp))           \
     throw "Invariante falhou...";

#define Pre_condicao(exp) \
 if (!(exp))     \
    throw "Pre-condicao falhou...";

#define Pos_condicao(exp) \
 if (!(exp)) \
    throw "Pos-condicao falhou...";
```

Figura 4: Macros para avaliar invariante de classe, pré-condições e pós-condições.

Ao término desse passo tem-se a classe autotestável, contendo uma referência para sua especificação, os métodos embutidos e os predicados.

5.4 Geração do driver específico

Neste passo o driver genérico é utilizado para gerar a seqüência de testes. Atualmente estão implementados os seguintes métodos:

- **Teste de fluxo de transação**, que consiste em percorrer o grafo, gerando as transações. Cada nó do grafo é visitado, sendo feita a escolha de um (ou todos) os métodos que compõem cada nó. Cada transação é vista pelo sistema como um objeto de teste [Sie96]. Um objeto de teste tem a seguinte forma:

  objeto_tst (Identificador, Lista_métodos, Caso_teste, Arq_Casos_teste)

  onde Lista_métodos representa uma lista dos métodos que compõem o objeto de teste, Caso_teste indica o caso de teste que testa este objeto e Arq_Casos_teste, o arquivo onde o caso de teste está armazenado.
  Se um objeto de teste é re-selecionado durante uma sessão de testes, as entradas geradas para esse objeto são reutilizadas. Se um objeto de teste é referente a uma subclasse, a reutilização das entradas de teste pode se dar conforme o indicado na tabela 1.

- Teste de dados, para selecionar os valores de atributos e dos parâmetros dos métodos. Na implementação atual os dados são gerados aleatoriamente, de acordo com a descrição dos valores possíveis que consta da especificação de teste. São considerados valores do tipo inteiro, real e cadeias de caracteres. Para atributos ou parâmetros na forma de ponteiros ou objetos, cabe ao usuário completar as informações requeridas.

Antes de iniciar os testes o usuário deve completar a seqüência de teste com os elementos que ainda não puderam ser gerados automaticamente pelo gerador, tais como:

- *stubs*, que substituem objetos usados pelo objeto em teste, e
- ponteiros usados em atributos ou parâmetros de métodos.

O próximo passo é compilar e ligar a seqüência de teste aos arquivos necessários: casos de teste, classe sob teste, classe Autoteste e macros. O arquivo executável funciona como um *driver* específico para a classe sob teste. Este *driver* ao ser executado gera um arquivo de resultados, contendo os casos de teste fornecidos, os resultados observados e as exceções sinalizadas, quando for o caso.

6. Trabalhos Existentes

Existem muitos trabalhos na área de testes de sistemas orientados a objetos, portanto não temos a pretensão de conhecê-los todos. Serão apresentados portanto alguns trabalhos, que de alguma forma influenciaram esse estudo: alguns referentes ao uso de testes caixa preta, outros sobre testabilidade.
Sobre o uso de testes caixa-preta podemos citar a metodologia FREE, apresentada em [Bin94a], em que o modelo de testes é baseado em máquina de estados. A metodologia

do grafo de teste, proposta em [HS95] também é baseada em modelo na forma de máquina de estado. O grafo de teste é um modelo de estados de uma classe, contendo os estados alcançáveis pelas seqüências de teste. Neste trabalho uma classe Driver gera as entradas de testes, percorrendo esse grafo para gerar as entradas de teste e aplicá-las à ao objeto. Uma outra classe, Oráculo, que implementa o modelo de teste, é usada para a análise dos resultados.

Ainda em termo de testes caixa preta, Doong e Frankl apresentam a abordagem ASTOOT [DF94], baseada em especificações algébricas, usadas para representar tipos abstratos de dados. O oráculo é baseado na equivalência observacional do comportamento externo do objeto: pares de seqüências de testes (contendo a mesma cadeia de invocação de métodos) são aplicados a dois objetos da mesma classe, verificando-se se os estados de ambos são equivalentes. Os aspectos teóricos para a aplicação de técnicas de teste baseadas na especificação em sistemas orientados a objetos é apresentada em [BBP98]. São apresentadas diversas hipóteses que permitem reduzir o tamanho de seqüências de teste, e diversas alternativas para a obtenção de um oráculo.

Também são usados modelos baseados nos relacionamentos entre classes. Em [HM92], por exemplo, a hierarquia de herança é usada para reduzir a quantidade de testes gerados no teste de subclasses. Kung et al apresentam um ambiente para testes e manutenção de sistemas, OOTM [KGJ+95], baseada em diversos modelos: ORD(Object Relation Diagram), representando os relacionamentos entre objetos (herança, agregação, associação, instanciação de classes genéricas, entre outros), que é usado para indicar uma ordem para o teste de classes com base no grau de dependência entre elas, com o objetivo de minimizar o número de *stubs*. Outros diagramas como o BBD (Block Branch Diagram), representando a estrutura de controle de um método e sua interface com outros métodos, bem como o OSD (Object State Diagram), similar ao modelo dinâmico da metodologia OMT [RBP+94], também são usados.

Com relação ao aspecto de projeto visando a testabilidade, que é o foco desse trabalho, tem-se o trabalho apresentado em [Hof89], onde é proposto o uso de assertivas (predicados) e métodos embutidos de teste para melhorar a testabilidade de módulos (o trabalho não enfoca a orientação a objetos). A geração automática de *drivers* usando um programa em Prolog, também inspirou o nosso estudo. Em [RST94] é apresentado um estudo mostrando empiricamente como o uso de assertivas melhora a testabilidade de programas. Wang et al. [WKC+97] descrevem diversos mecanismos de teste que podem ser embutidos nos métodos de classes para controlar os principais construtos de controle: condição, seqüência e iteração. Esses mecanismos podem ser herdados, da mesma forma que outras características de uma classe.

7. Conclusão

Neste trabalho foi apresentado o uso do conceito de autoteste, aplicado em circuitos VLSI, para a melhoria da testabilidade de classes. O que se pretende dessa forma é facilitar o reuso, permitindo que componentes reusáveis possam ser testados com facilidade sempre que for necessário.

Para mostrar a viabilidade da proposta foi construído um protótipo de uma ferramenta de auxílio à construção e utilização de classes autotestáveis.

A ferramenta visa auxiliar o produtor de uma classe a introduzir os mecanismos embutidos de testes (métodos relatores, para obtenção do estado interno e assertivas,

para checagem do estado interno em tempo de execução) e para associar a especificação de teste à classe. Para o usuário da classe um gerador, escrito em Prolog, permite construir automaticamente um driver que irá aplicar a seqüência de testes à classe e analisar os resultados. A seqüência de testes é gerada com base em modelo da classe na forma de um grafo de fluxo de transação que descreve as seqüências válidas de invocação de métodos de um objeto. Com o uso da técnica incremental hierárquica o gerador pode determinar quais testes precisam ser gerados, quais podem ser utilizados, nos testes de uma subclasse. Combinando os testes de fluxo de transação com testes de dados são geradas não só as cadeias de invocação de métodos, mas também os parâmetros desses métodos.

O uso do conceito de autoteste combinado à técnica incremetal hierárquica tem um impacto no desenvolvimento das classes:

- O desenvolvedor deve acrescentar à classe os elementos embutidos de teste, bem como a especificação. Cada modificação feita à classe deve ser refletida na especificação de teste. Apesar de não ser ainda muito comum, essa prática é altamente recomendável, pois melhora não só a testabilidade mas também o entendimento da classe, com isso melhorando sua manutenibilidade.
- Alguns dos mecanismos oferecidos pela orientação a objetos, tais como: herança múltipla, herança de implementação e o uso de classes/funções amigas em C++, não podem ser usadas. De toda a forma o uso desses mecanismos é altamente desaconselhável, pois aumenta a complexidade do sistema, tornando-o mais propenso a falhas.

O desenvolvimento do protótipo foi um primeiro passo que serviu para mostrar a viabilidade de se construir uma ferramenta de auxílio à construção e utilização de classes autotestáveis. Naturalmente que extensões devem ser feitas em trabalhos futuros para acrescentar outras estratégias para a geração de dados e também para abordar aspectos não tratados aqui, em especial o uso de classes compostas. O uso de concorrência também não é enfocado, pois o C++, a linguagem alvo desse trabalho, não possui mecanismos explícitos para tal.
O segundo passo consiste em utilizar classes autotestáveis para estudar a viabilidade da proposta. Para isso o protótipo construído está sendo usada nos testes de uma aplicação real, que é o Sistema de Telemetria e Telecomando desenvolvido pela Divisão de Sistemas de Solo do Instituto Nacional de Pesquisas Espaciais (INPE), em S.José dos Campos.

Referências Bibliográficas

[Bei90] B. Beizer. *Software Testing Techniques*. International Thomson Computer, 2ª edição, 1990.

[BBP98] S.Barbey, D.Buchs, C.Péraire, "Issues and Theory for Unit Testing of Object-Oriented Software", 1998. Obtido via Internet no URL: http://www.di.epfl.ch.

[Bin94a] R. V. Binder, "The FREE Approach to Testing Object-Oriented Software: An Overview", Chicago, RBSC Corporation, 1994. Obtido via Internet no URL: http://www.rbsc.com

[Bin94b] R.V.Binder, "Design for Testability with Object-Oriented Systems", *Communications of the ACM*, v. 37, nº. 9, set./94, p. 87-101.

[Bin97] R. V. Binder, "Testing Object-Oriented Software: A Status Report", Chicago, RBSC Corporation, 1997. Obtido via Internet no URL: http://www.rbsc.com.

[BR98] L.E.Buzato, C.M.F.Rubira. "Construção de Sistemas Orientados a Objetos Confiáveis". 11ª Escola de Computação. Rio de Janeiro, 1998.

[CM87] W.F.Clocksin, C.S.Mellish. *Programming in Prolog*, Sprnger Verlag, 3ª. ed, 1987.

[DF94] R.K.Doong, P.G.Frankl. "The ASTOOT approach to testing object-oriented programs". In: *ACM Trans. on Software Engineering and Methodology*, 3(2), abr/94, p.101-130.

[Hof89] D.M.Hoffman. "Hardware Testing and Software ICS", in: anais do *Pacific NW Software Quality Conference*, Protland, Oregon, EUA, set/1989, pp234-244.

[HM92] M. J. Harrold e J. D. McGregor, "Incremental Testing of Object-Oriented Class Structures". Clemson University, 1992.

[HS91] D. M. Hoffman e P. Strooper. "Automated Module Testing in Prolog". in: *IEEE Transactions on Software Engineering*, v. 17(9), set./1991, p. 934-943.

[HS95] D. M. Hoffman e P. Strooper. "The Testgraph Methodology". in: *Journal of Object-Oriented Programming*, nov-dez/1995, p. 35-41.

[KJG+95] D. Kung, J. G., et al., "Developing an Object-Oriented Software Testing and Maintenance Environment". in: *Communications of the* ACM, v. 38, nº. 10, out./95, p. 75-87.

[PK90] D.E.Perry e G.E.Kaiser, "Adequate Testing and Object-Oriented Programming". in: *Journal of Object-Oriented Programming*, jan-fev./1990, pp13-19.

[Pos96] R.M.Poston. *Automating specification based software testing*. IEEE Computer Society Press, 1996.

[Pre95] Pressman, Roger S., "*Engenharia de Software*", Makron Books, São Paulo, 1995.

[RBP+94] J. Rumbaugh, M. Blaha, et. al., "*Modelagem e Projetos Baseados em Objetos*". Editora Campus, Rio de Janeiro, 1994.

[Sie96] S. Siegel, *"Object Oriented Software Testing: a Hierarchical Approach"*. John Wiley & Sons, 1ª edição, New York, 1996.

[SR90] M. D. Smith e D. J. Robson, "Object-Oriented Programming – the Problems of Validation". in: *Intern. Conference on Software Maintenance*, EUA, 1990.

[SR92] M. D. Smith e D. J. Robson, "A framework for testing object-oriented programs". in: *Journal of Object-Oriented Programming*, vol. 5, nº. 3, jun./1992.

[RST94] Reliable Software Technology Corporation. "Testability of Object-Oriented Systems". Relatório técnico 95-01, dez/94. Obtido via Internet no URL: http://www.rstcorp.com.

[TM98] C. M. Toyota e E. Martins. "Reutilização em Teste de Software Orientado a Objetos: Metodologia para Construção de Classes Autotestáveis". in: *IX Conferência Internacional de Tecnologia de Software - Qualidade de Software.* Curitiba, jun./1998.

[UP83] H. Ural e R. L. Probert. "User-Guided Test Sequence Generation". in: *Protocol Specification, Testing and Verification III*. North-Holland, 1983, p. 421-436.

[Wey98] E.Weyuker. "Testing component-based software: a cautionary tale". *IEEE Software*, pp 54-59, Sept-Oct/1998.

[WKC+97] Y.Wang, G.King, I.Court, M.Ross, G.Staples. "On Testable Object-Oriented Programming". ACM Software Engineering Notes, 22(4), jul/1997, pp 84-90.