Uso de Instrumentação para a Construção de Programas Confiáveis

Arndt von Staa
(e-mail: arndt@inf.puc-rio.br)

PUCRioInf MCC 28/93

Abstract

In this paper we discuss the implementation of reliable programs using “almost” reliable components. The implementation method contributes also to the productivity increase when compared to traditional implementation based on testing and debugging. The implementation method is based on (i) the use of design and programming tools aiming at the construction of “almost” correct programs, (ii) the use of robust and auto-verifiable data strutures, (iii) the inclusion of executable assertions into the programs, (iv) the use of co-routines to verify the structural correctness of the data structures during idle cycles.

KEYWORDS: Assertions, Auto-verifiable data structures, Correctness, Debugging, Executable assertions, Instrumentation, Program testing, Robust data structures, Structural assertions

Resumo

É apresentado um método para o desenvolvimento de programas confiáveis, construídos a partir de componentes "quase" confiáveis. O método contribui para um significativo ganho de produtividade quando comparado com o desenvolvimento “clássico” baseado em teste e correção. O método baseia-se (i) no uso de ferramentas de projeto visando a criação de módulos "quase" corretos por construção, (ii) no uso de estruturas de dados auto-verificáveis; (iii) na incorporação de instrumentação de apoio à diagnose, (iv) na incorporação de assertivas executáveis em tempo real, (v) na incoporação de corotinas para a verificação das assertivas utilizando os ciclos ociosos da máquina.

PALAVRAS CHAVE: Assertivas, Assertivas estruturais, Assertivas executáveis, Auto-teste, Corretude, Diagnose, Estruturas de dados robustas, Instrumentação, Testes

1. Motivação

Software produto requer um nível elevado de qualidade. Apesar de todos os cuidados ao especificar, arquitetar, projetar, estruturar e codificar o software produto, sempre será necessário testá-lo [Gerhart76]. Em outras palavras, a prática tem demonstrado que a criação de módulos corretos por construção, assegurando a total correspondência entre o módulo e sua especificação, é um ideal não atingível na prática (pelo menos por enquanto). Necessita-se, então, testar programas, em adição ao uso de técnicas que visem assegurar a "quase" correção por construção.

Testes são *experimentos controlados* em que se procura evidência suficiente para confirmar a "teoria" que o módulo em teste se comporta **exatamente** tal como especificado. Um teste corretamente realizado utiliza um *critério de seleção de casos teste* [Goodenough75]. Cada caso teste define as condições a serem satisfeitas pelos dados de modo que determinado aspecto do módulo em teste seja exercitado. O conjunto de casos teste gerado deve ser minimal e deve *cobrir o módulo*, ou seja, deve testar (i) todo o módulo, e/ou (ii) todas as estruturas de dados utilizadas e/ou (iii) todas as especificações segundo um *critério de completeza* definido. Os casos teste devem ser confiáveis, ou seja, devem *independer da escolha* dos dados específicos, desde que estes satisfaçam as condições dos casos teste. Finalmente, o conjunto de casos teste deve ser *válido* de modo que, existindo *falha* (defeito de implementação) no módulo, seja gerado pelo menos um caso teste que exiba um *erro* (comportamento diferente do esperado).

Infelizmente, a prática mostra que dificilmente conseguimos atingir plenamente estes objetivos. Em outras palavras, testes tendem a ser experimentos parciais, não servindo como instrumentos confiáveis para assegurar a correspondência **total** entre a especificação e o segmento de programa testado. Em adição, em programas complexos, a cobertura da estrutura do módulo é, muitas vezes, economicamente inviável, conseqüentemente, porções significativas dos módulos poderão ficar sem suficiente exercitação. Da mesma forma, a cobertura estrutural de estruturas de dados complexas é inviável, conseqüentemente, sub-conjuntos significativos de estados dos dados poderão ficar sem suficiente avaliação experimental. No entanto, é justamente em sistemas complexos utilizando estruturas de dados complexas que mais necessitamos de garantia de qualidade tanto por construção como experimental.

Além dos problemas acima, existem ainda problemas inerentes ao processo de realização do controle de qualidade. Quando o programa e as estruturas de dados que utiliza forem complexos, a quantidade de elementos que influenciam a qualidade é grande. Neste caso, o confronto dos resultados esperados com os resultados obtidos passa a ter uma elevada chance de erro, em particular quando realizado por seres humanos. Em virtude disso pode-se inadvertidamente aceitar como correto um módulo contendo falhas.

Necessitamos, então, de técnicas alternativas para:

1- construir programas confiáveis a partir de componentes não confiáveis e utilizando métodos de construção não confiáveis;

2- permitir a elaboração econômica de módulos "quase" corretos por construção;

3- permitir a realização econômica de testes "quase" completos;

4- reduzir os riscos decorrentes de falha humana.

2. Projeto da estrutura de módulos

A utilização de técnicas de projeto em papel é desmotivante e eivada de falhas de transcrição. Assim, costumeiramente módulos são construidos utilizando-se editores de texto seqüenciais, o que torna difícil assegurar boa estruturação em todos os módulos que compõem o programa. Uma das soluções propostas é a utilização de ferramentas contendo editores de estrutura que permitam definir e organizar a estrutura dos módulos, permitam adicionar fragmentos de texto de código e de assertivas aos módulos, bem como permitam

a linearização destas estruturas para código seqüencial inteligível pelos compiladores [Staa92].

Foge ao escopo deste artigo discutir em detalhe o uso de ferramentas de apoio ao projeto de módulos. Entretanto, cabe ressaltar que as técnicas abordadas no restante deste artigo de pouco servirão sem um projeto cuidadoso dos módulos. Este projeto requer a definição de assertivas de entrada e saída e da estrutura interna do módulo. Requer, ainda, a definição das estruturas de dados e das correspondentes assertivas estruturais.

3. Assertivas executáveis

Ao projetar um módulo são definidas assertivas que capturem as condições de correto funcionamento do módulo [Staa87]. Tipicamente estas assertivas determinam as condições que os dados devem satisfazer ao entrar em uma função ou fragmento de estrutura de código, bem como as condições a serem satisfeitas ao sair desta função ou fragmento.

Em geral é possível traduzir uma parcela considerável destas assertivas em código executável. Assertivas executáveis reduzem o esforço de diagnose (identificação da falha que provocou o erro), uma vez que evidenciam a ocorrência de uma anomalia tão logo ela se torne perceptível, informando o que está diferente do esperado e onde isto se verificou. Por exemplo, seja a função *Inserir nó antes de nó dado* que insere um novo nó e uma lista duplamente encadeada. Esta função recebe um ponteiro para um nó desconectado de qualquer lista e um ponteiro para um nó pertencente a alguma lista duplamente ancadeada. Ao entrar podemos verificar se o novo nó está devidamente desconectado, bastando verificar se os ponteiros anterior e próximo deste novo nó estão devidamente nulos. Podemos verificar, ainda, se o nó da lista antes do qual se deve dar a inserção de fato existe. Ao sair podemos verificar se o nó sucessor do novo nó é o nó da lista fornecido e se o nó antecessor do nó da lista fornecido é o novo nó introduzido.

Assertivas executáveis reduzem o risco de falha humana, uma vez que são verificadas automaticamente. Assertivas executáveis viabilizam o uso de módulos quase corretos, uma vez que podem interceptar o prosseguimento da execução do programa, evitando, assim, as conseqüências do mau funcionamento.

Diversas linguagens fornecem construções de controle de assertivas (por exemplo a função "assert" em "C"). No entanto, estas construções são pouco satisfatórias, uma vez que não informam a semântica (o significado) da condição em erro e, ao descobrirem uma situação de erro, cancelam imediatamente a execução do programa sem realizar a “arrumação da casa”. O cancelamento imediato pode provocar conseqüências indesejáveis, como, por exemplo, deixar transações de alteração de banco de dados parcialmente executadas e, conseqüentemente, inutilizando o banco de dados.

Ao detectar uma condição de erro, a assertiva executável deve ativar um controlador de erro capaz de:

1- emitir as mensagens de erro, possivelmente incluindo valores de variáveis relevantes, por exemplo, o nome do arquivo afetado pelo erro, a identificação do módulo onde foi identificado o erro, etc.

2- ao operar em modo depuração, ativar um instrumento de exame do conteúdo de memória em termos dos tipos das variáveis e da especificação das estruturas de dados utilizadas. Por exemplo, ao encontrar um erro em uma lista, o instrumento permite exibir o conteúdo da lista em variados formatos.

3- ao operar em modo de produção, ativar um instrumento de recuperação que restaure a integridade dos arquivos;

4- decidir se continua ou se cancela. Dessa forma pode-se incluir assertivas executáveis sinalizando a ocorrência de situações *implausíveis*. Estas são condições de funcionamento que, embora corretas, têm baixissima probabilidade de ocorrência.

É claro que o desenvolvimento de assertivas executáveis adiciona esforço ao desenvolvimento do código. Por outro lado, reduz o esforço de diagnose de erros, contribuindo para a redução do custo total de desenvolvimento do programa. Cabe salientar que o uso de assertivas executáveis não torna inúteis as ferramentas de apoio à depuração que acompanham os compiladores modernos, no entanto reduzem as ocasiões em que se necessita utilizá-las.

Assumindo que os módulos que compõem um programa sejam imperfeitos, as assertivas executáveis devem, em princípio, permanecer nos programas, uma vez que elas constituem as redundâncias necessárias para a identificação, o mais cedo possível, das anomalias de funcionamento dos programas. No entanto, levar a inclusão de assertivas executáveis até as últimas conseqüências pode aumentar sensivelmente o esforço computacional realizado, inviabilizando o uso do programa em virtude de tempos de resposta intoleráveis. Deve-se, então, estimar o custo operacional da execução destas assertivas (tempo de execução x número de vezes que se espera ser executado) e confrontá-lo com o custo decorrente do dano que poderia ser provocado pelo mau funcionamento não interceptado.

Além de aumentarem o tempo de execução de programas, assertivas executáveis freqüentemente têm o inconveniente de controlarem somente condições locais de execução dos programas. Podem ocorrer situações em que estas condições locais estarão corretas do ponto de vista da assertiva e, no entanto, as estruturas de dados globais estão em erro. Por exemplo, seja um módulo capaz de copiar listas. Este módulo pecorre a lista origem do primeiro nó para o último, criando uma nova lista. Ao retornar, fornece o ponteiro para o primeiro nó da lista criada. Uma variante errada deste programa poderia retornar o ponteiro para algum nó da lista em situações extremamente particulares, por exemplo caso tenha sido necessário reorganizar a memória (“garbage collection”) mais do que uma vez durante a cópia. O processamento posterior utilizando este resultado errado poderá até dar a impressão de estar operando corretamente, desde que o ponteiro retornado aponte corretamente para a origem de um nó.

4. Estruturas de dados auto-verificáveis

Uma estrutura de dados é dita *auto-verificável* caso ela possua redundância suficiente para evidenciar erros estruturais sem requerer informações de estado dos programas que manipulem estas estruturas. Para podermos verificar a correção de uma estrutura de dados independentemente do estado de execução do programa, é necessário que ela seja auto-verificável. Isto é facilmente observado pelo fato que, sem informação de estado do programa, também não sabemos que assertivas de execução valem a cada instante.

Por exemplo, seja um conjunto de listas lineares em que cada um dos nós pertença exclusivamente a uma única lista. Esta propriedade poderá ser facilmente verificada caso todas as listas do conjunto forem duplamente encadeadas e caso exista uma lista adicional, a lista de listas, na qual o campo de informação de cada um dos nós aponte para a origem de uma das listas do conjunto. Esta lista de listas é evidentemente uma das listas do conjunto de listas. Note que o encadeamento pelo campo de informação não fere requisito de disjunção das listas, pois este depende somente dos campos de encadeamento.

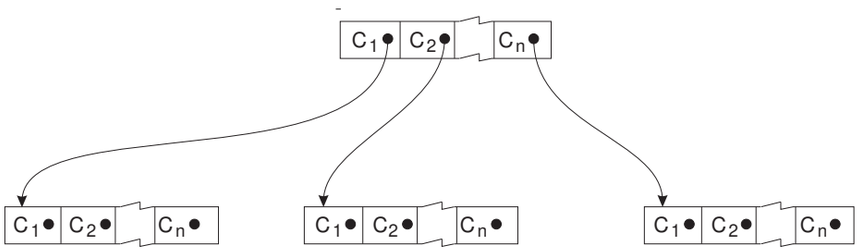

Figura 1. Fragmento de estrutura BTREE sem reundância alguma

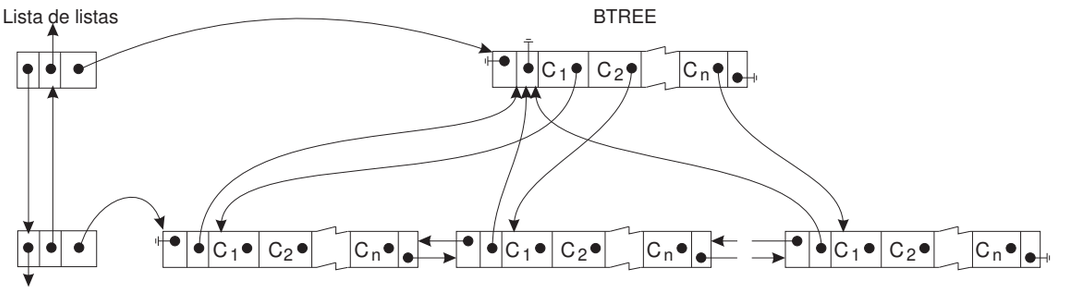

Figura 2. Fragmento de BTREE com redundâncias

Relações para todos os nós do conjunto de listas

```
Se existe Nó.Próximo então
    Nó.Próximo->Anterior == Nó
Se existe Nó.Anterior então
    Nó.Anterior->Próximo == Nó
```

Relações específicas para nós das listas da BTREE

```
NUM_MAX / 2 <= Nó.NumElem <= NUM_MAX

Para todos 1 <=  i <  j <=  Nó.NumElem
    Nó.Chave[ i ] < Nó.Chave[ j ] e
    Nó.Filho[ i ] != Nó.Filho[ j ] se um deles for != NULL
Se existe Nó.Próximo então
    Nó.Chave[ Nó.NumElem ] < Nó.Próximo->Chave[ 1 ]

Se existe Nó.Anterior então
    Nó.Anterior->Chave[ Nó.Anterior->NumElem ] < Nó.Chave[ 1 ]
```

Relações para as estruturas pai / filho

```
Se não Existe( Nó.Pai ) então
    Nó.Próximo  == NULL

    Nó.Anterior == NULL

Se Existe( Nó.Pai ) então

    Nó.Próximo != NULL ou Nó.Anterior != NULL

    Existe InxFilho tal que
       Nó.Pai->Chave[ InxFilho ] == Nó.Chave[ 1 ] e
       Nó.Pai->Filho[ InxFilho ] == Nó
```

Figura 3. Assertivas estruturais para nós da BTREE

Examinemos um outro exemplo. A figura 1 ilustra um fragmento de uma BTREE. Em uma BTREE, cada nó contém *NumElem* chaves. O valor de *NumElem* deve ser tal que pelo menos metade do espaço alocado para o nó seja ocupado. A cada chave está associado um ponteiro filho. Quando não nulo, este ponteiro aponta para outro nó da estrutura. Este outro nó contém chaves maiores ou iguais à chave correspondente ao ponteiro e menores do que a chave a seguir no nó. A organização da BTREE na figura 1 possui nenhuma redundância. Observando um nó qualquer da estrutura não temos meios de verificar se este nó pertence à BTREE e se está corretamente encadeado.

O encadeamento duplo é muitas vezes redundante, por exemplo para filas e pilhas. Ele é necessário para poder-se assegurar que cada nó adjacente a outro nó em uma lista, seja adjacente a somente este nó no conjunto de todas as listas. No processamento convencional, a lista de listas é redundante uma vez que os programas mantém origens das listas que utilizem em variáveis próprias. Ela é necessária para estabelecer uma estrutura de dados auto-verificável, já que não podemos lançar mão desta informação sem conhecer o estado interno dos programas.

A figura 2 ilustra o mesmo fragmento de BTREE, desta vez transformado em uma estrutura autoverificável. Cada "patamar" da árvore é uma lista. Cada uma destas listas é referenciada pela lista de listas. Cada lista é duplamente encadeada. Cada nó da árvore possui um ponteiro para o nó pai. A primeira chave de cada nó é igual à chave associada ao ponteiro filho correspondente. Com estas adições torna-se possível verificar se um dado nó está corretamente encadeado, independentemente do processamento realizado para se chegar a este nó.

5. Assertivas estruturais

Assertivas estruturais definem a corretude da organização de uma estrutura de dados. Ao entrar em uma função de manipulação da estrutura, as assertivas estruturais desta estrutura devem estar válidas, pois, de outra forma, o resultado da função poderia ficar comprometido. Ao sair é evidente que as assertivas devem valer, pois senão a função não teria operado corretamente. É claro que durante o processamento da função as estruturas podem passar por estados em que as assertivas estruturais não valem. Por exemplo, ao inserir um nó e uma lista duplamente encadeada, ao entrar e ao sair a lista é evidente que a lista deve estar corretamente encadeada. Porém, enquanto se está inserindo o nó, vai ocorrer um momento em que a lista estará parcialmente encadeada. A figura 3 mostra as assertivas estruturais da BTREE em acordo com o diagrama da figura 2.

6. Co-rotina de verificação

Em ambientes de computação mono-usuário, por exemplo micros, é comum ocorrerem significativos tempos ociosos em virtude da espera por ação (tecla ou movimentação de "mouse") do usuário. Estes tempos ociosos podem ser utilizados, por exemplo, para realizar a verificação da correção estrutural das estruturas de dados. Esta verificação é agora realizada sem afetar o tempo real do processamento de transações.

O processamento de verificação pode ser bastante extenso. Para tornar viável a verificação imperceptível ao usuário, é necessário particionar o processo de verificação em uma seqüência de pequenos passos de verificação. Cada um destes passos deverá requerer um tempo de processamento tão pequeno (algo em torno de 0,05 milisegundos) que possa ocorrer entre ações sucessivas do usuário sem que ele perceba qualquer delonga significativa.

Ao realizar um passo de verificação não devem ocorrer alterações de estado nas estruturas de dados sendo verificadas, uma vez que durante a realização destas alterações as assertivas sendo controlada podem não valer. Isto torna necessário que cada passo, uma vez iniciado, processe até o final, sem ser interrompido por ações do usuário. Em adição, cada passo de verificação deve examinar completamente um aspecto. No exemplo da BTREE, um passo de verificação corresponderia a validar um nó.

A quebra do verificador em passos deve permitir que o processamento possa suspender ao término de cada passo e retomar a execução com o passo a seguir, independentemente do processamento realizado entre estes instantes. Para isto é necessário manter-se informação de estado capaz de informar o progresso do verificador e capaz de passar informação de um passo para o seguinte. Ou seja, temos que tornar o processo de verificação em uma co-rotina. Note que esta co-rotina deve ser absolutamente transparente ao programa de produção, conseqüentemente a informação de progresso não pode ser incorporada à estrutura de dados sendo verificada, devendo ser mantida interna à co-rotina de verificação.

Além de construir o verificador como uma co-rotina, temos que desenvoler um pequeno controlador de ativação do verificador. Este nada mais é do que um ciclo que repete enquanto o computador estiver esperando por ação do usuário. A cada ciclo é ativado o verificador e, consequentemente, é realizado um passo de verificação. Após um

Operações sobre a co-rotina de validação

`CriaCorotina( IdEstrutura ) ;`
— Cria a corotina, habilitando-a a operar sempre que for ativada.

`ReinicializaCorotina( IdEstrutura ) ;`
— Restaura a corotina para validar a partir do início da estrutura.

`AtivaCorotina( IdEstrutura ) ;`
— Realiza um passo de validação da estrutura. Esta chamada é inserida nos pontos de ociosidade (exemplo: espera por ação do usuário) e/ou em pontos de baixa criticalidade com relação a tempo de resposta (exemplo: geração de relatórios).

`DestróiCorotina( IdEstrutura ) ;`
— Cancela a corotina, cessando todas as atividades de validação.

A função AtivaCorotina tem o esquema:

```
Se ExisteCorotina( IdEstrutura ) então
    Preserva contexto afetado
    Restaura contexto de validação
    Efetua um passo de validação
       este passo deve requerer pouco esforço
    Salva contexto de validação
    Restaura contexto afetado
```

Figura 4. Esquema da co-rotina de validação.

número finito de ativações o verificador deverá ter concluido todos os passos necessários para a verificação integral da estrutura de dados.

Resta resolver um pequeno problema de sincronismo. Caso a transação disparada pela ação do usuário realize uma alteração na estrutura de dados, o estado corrente do verificador pode tornar-se incompatível com o estado da estrutura de dados. Este problema pode ser resolvido através de um “time-stamp” atualizado pelos processadores de transação sempre que estes alterem o conteúdo da estrutura de dados. Agora basta o verificador comparar o "time-stamp" da estrutura sendo verificada com o "time-stamp" que ela tinha ao iniciar a verificação a cada vez que for acionado um passo de verificação, Caso sejam diferentes, reinicia-se a verificação.

A figura 4 ilustra a organização de uma co-rotina de validação de assertivas estruturais.

7. Conclusão

Neste artigo propusemos um método de desenvolviemnto de programas visando alcançar elevada confiabilidade a partir de módulos "quase" confiáveis. Este método utiliza intensivamente o computador como parceiro no desenvolvimento e no controle da execução de programas. O método baseia-se nos seguintes princípios:

1- especificação cuidadosa de cada módulo e das estruturas de dados manipuladas por estes módulos;

2- projeto cuidadoso da estrutura interna de cada módulo utilizando uma ferramenta de apoio ao projeto e à geração de código dos módulos;

3- aplicação de critérios de controle de qualidade ao projeto da estrutura e do código dos módulos, levando a módulos quase corretos por construção;

4- desenvolvimento de instrumentação de apoio à diagnose de erros e de exame do conteúdo e recuperação de estruturas de dados;

5- inclusão de assertivas executáveis capazes de disparar a instrumentação sempre que for detectada uma situação anomala dos dados manipulados;

6- desenvolvimento de uma co-rotina composta por diversos passos de verificação das assertivas estruturais das estruturas de dados utilizadas pelo programa. A execução de todos os passos da co-rotina assegura a validade destas estruturas de dados. A identificação de uma anomalia durante a execução de um passo de verificação dispara a execução da instrumentação de apoio.

7- desenvolvimento de um "mini" controlador de multiprogramação que ativa a co-rotina de verificação sempre que o computador esteja esperando por ação do usuário.

8- informação da ocorrência de alterações de conteúdo das estruturas de dados através de "time-stamps" atualizados pelos processadores de transações.

Este método foi utilizado para o desenvolvimento do ambiente TALISMAN [Staa92]. O nível de registro das bases de dados de TALISMAN é formado por listas de elementos. Cada elemento é formado por um par chave e valor de tamanho variável. Estas listas são armazenadas em memória virtual simulada. Os elementos das listas são acessíveis via B-Trees. O nível de registro é implementado por módulos para tratamento de listas genéricas, de dicionários e de B-Trees. Para apoio ao desenvolvimento foram desenvolvidos módulos para a exploração e edição força bruta do conteúdo de memória e para verificação co-rotinada da integridade estrutural. Todos estes módulos foram projetados utilizando MOSAICO e, posteriormente, importados para TALISMAN.

O método aqui descrito foi utilizado para controlar a qualidade dos módulos de acesso a registros da base de dados de TALISMAN. Grande parte dos testes foi realizada por pessoas sem conhecimento algum de programação. Para realizar os testes foi definido um roteiro de teste de acordo com um critério de seleção visando a cobertura de estrutura de dados. As pessoas encarregadas pelos testes efetuavam a entrada de dados segundo este roteiro, controlavam a correção dos dados teste fornecidos através do uso da instrumentação de inspeção do conteúdo da estruturas de dados, e registravam eventuais anomalias reportadas pelos verificadores. A partir destes relatórios de teste foi realizada a depuração dos módulos.

Depois de concluida a validação básica tanto dos módulos de acesso com dos módulos de instrumentação e verificação, passou-se à realização de testes estatísticos. Durante estes testes foram criadas e integralmente excluídas estruturas com cerca de 10 Mbytes de dados armazenados, utilizando diversas funções de distribuição para a geração de chaves e valores, inclusive com repetição.

Os módulos estão em uso há alguns anos tendo sido reportados zero erros. O tempo de desenvolvimento dos módulos de acesso e da instrumentação foi de cerca de 2 meses-homem. O tempo dedicado ao teste e à depuração destes módulos foi de menos de duas semanas. É evidente que estas estatísticas não são generalizáveis. São um forte indicador de que há um significativo ganho de produtividade quando do uso do método aqui relatado.

Referências bibliográficas

[Gerhart76] Gerhart, S.L.; Yelowitz, L. "Observations of fallibility in applications of modern programming methodologies"; *IEEE Transactions on Software Engineering*; setembro 1976

[Goodenough75] Goodenough, J.B.; Gerhart, S.L.; "Toward a theory of test data selection"; *IEEE Transactions on Software Engineering*; junho 1975

[Staa87] Staa, A.v.; *Engenharia de Programas*; Livros Técnicos e Científicos; Rio de Janeiro 1987, 2a. edição

[Staa92] *Ambiente de Desenvolvimento de Sistemas TALISMAN, Manual do Usuário*; STAA Informática Ltda. Rio de Janeiro; 1992