PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 31/96

PG-08 Regras e Recomendações para o Uso de Instrumentação em Programas C e C++ - Versão 1.00 -

Arndt von Staa
(Editor)

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 31/96
Editor: Carlos J. P. Lucena Setembro, 1996

PG-08 Regras e Recomendações para o Uso de Instrumentação em Programas C e C++ - Versão 1.00 - *

Arndt von Staa
(Editor)

* Trabalho patrocinado pelo Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

Responsável por publicações:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brazil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br

PG-08 Regras e Recomendações para o uso de Instrumentação em Programas C e C++

- Versão 1.0 -

A.v. Staa[1]
(Editor)

arndt@inf.puc-rio.br

Laboratório de Engenharia de Software
Departamento de Informática
Pontifícia Universidade Católica
22453-900 Rio de Janeiro, Brasil

PUC-RioInf.MCC31/96, Setembro 1996

Resumo: Neste documento é apresentado um conjunto de critérios para a inclusão de instrumentação ativa em programas redigidos em C ou C++. São propostos diversos instrumentos: assertivas estruturais que verificam a integridade de estruturas de dados, exploradores que permitem examinar e possivelmente editar conteúdos de estruturas de dados e co-rotinas de validação que realizam os controles de integridade utilizando os tempos ociosos dos equipamentos. Com a adoção desta norma espera-se:

i. reduzir o esforço requerido para diagnosticar erros e defeitos contidos em programas.
ii. viabilizar o auto-teste através da inclusão de código capaz de monitorar o correto funcionamento do programa.
iii. aumentar a eficácia e a eficiência dos testes.
iv. estabelecer meios para controlar o correto funcionamento de programas concorrentes, ou possuidores de um número muito grande de alternativas de operação.

Palavras chave: assertivas, assertivas estruturais, corretude, instrumentação, controle de qualidade, testes.

Abstract: In this document active software instrumentation is described. The instrumentation is geared towards C or C++ programs. Several instruments are proposed: structural assertion verifiers, which assure integrity of the contents of data structures; structure explores, which support exploring and editing of the contents of data structures; and verifying coroutines which perform verification during idle times. When adopting this standard it is expected that:

---

[1] Trabalho apoiado por: CNPq, Bolsa de Pesquisador 300029/92-6, CENPES/Petrobrás, Itautec/Philco

- the effort required to diagnose errors will be perceptibly reduced.
- the code will be capable to verify the correctness of data structures while executing.
- efficiency and efficacy of tests will be increased.
- there are means to dynamically control correctness of parallel programs, or programs subjected to non repeatable event sequences.

Keywords: assertions, correctness, instrumentation, quality control, structural assertions, testing.

Histórico de evolução
PG-08 Regras e recomendações para o uso de instrumentação em programas

Gestor: Laboratório de Engenharia de Software
Departamento de Informática, PUC-Rio

Arquivo: Instrm01
Editado: 16 setembro, 1996
Impresso: 12 agosto, 2002

Documentos correlatos

PG-01	Convenções para a escolha de nomes de elementos em programas C e C++
PG-02	Regras e recomendações para o uso de constantes simbólicas em C e C++
PG-03	Regras e recomendações para a programação em C e C++
PG-04	Regras e recomendações específicas para a programação em C++
PG-05	Regras e recomendações para a inclusão de especificações no código de programas
PG-06	Regras e recomendações para fluxos de controle em algoritmos
PG-07	Regras e recomendações para avaliação de estruturas

Versão V1.01

Editores:
Arndt von Staa (PUC-Rio)

Status: Em uso

Data homologação: 01/jul/1996
Data entrada em vigor: 01/jul/1996

Data de início da próxima revisão 01/jan/1997

Descrição de evolução
correções ortográficas e sintáticas

Descrição da retroação

Versão V1.00

Editores:
Arndt von Staa (PUC-Rio)

Status: Em uso

Data homologação: 01/mar/1996
Data entrada em vigor: 01/mar/1996

Data de início da próxima revisão 01/jan/1997

Descrição de evolução

Descrição da retroação

Créditos

Revisores

André Derraik	(TeCGraf PUC-Rio)	Versão 1.0
Claudio de Oliveira	(PUC-PR)	Versão 1.0
Geraldo Machado Costa	(LES PUC-Rio)	Versão 1.0
Lincoln Nobumiti Kanamori	(Itautec Philco)	Versão 1.0
Pedro Alexandre O. Giovani	(Itautec Philco)	Versão 1.0
Pedro Jorge E. Hübscher	(LES PUC-Rio)	Versão 1.0
Renan Martins Baptista	(CENPES)	Versão 1.0
Rosa Maria Ramalho Correia	(Itautec Philco)	Versão 1.0

Apoio

CENPES Petrobrás
CNPq
Itautec Philco

Marcas registradas e nomes de produtos

Sumário

Objetivo

Esta norma estabelece regras e recomendações para a inclusão de código de instrumentação em programas C e C++.

Com a adoção desta norma espera-se:

i. reduzir o esforço requerido para diagnosticar erros e defeitos[2] contidos em programas.

ii. viabilizar o auto-teste através da inclusão de código capaz de monitorar o correto funcionamento do programa.

iii. aumentar a eficácia (capacidade de encontrar falhas) e a eficiência (rapidez com que as falhas são localizadas) dos testes.

iv. estabelecer meios para controlar o correto funcionamento de programas concorrentes, ou possuidores de um número muito grande de alternativas de operação.

Motivação

Uma parcela significativa do software desenvolvido requer um elevado nível de qualidade. Apesar do uso de instrumentos formais e apesar de todos os cuidados ao especificar, arquitetar, projetar, estruturar e codificar o software, sempre será necessário testá-lo [Gerhart 76]. Em outras palavras, a prática tem demonstrado que a criação de programas corretos por construção, assegurando a total correspondência entre o programa e a sua especificação, é, por enquanto, um ideal não atingível na prática.

Testes são *filtros estatísticos*, ou seja eles possuem uma certa probabilidade, menor do que um, de encontrar erros. Conseqüentemente, jamais poderemos assegurar a ausência de erros no programa meramente realizando testes. Assim, para desenvolver programas assegurando elevados níveis de qualidade são necessários diversas ações, entre elas:

- seguir um processo de desenvolvimento bem definido e capaz de levar a programas de qualidade assegurada [Paulk 93],
- especificar o que se espera que o programa ou componente venha a realizar, incluindo aí também os critérios de aceitação (critérios de avaliação da qualidade [Rocha 87]).
- utilizar técnicas de projeto e programação possuindo o potencial de assegurar a correção por construção,
- incluir assertivas no código dos programas [LG 86],
- incluir instrumentação no projeto e no código do programa,
- utilizar medições e inspeções sistemáticas amparadas em critérios de avaliação da qualidade, e
- realizar testes sistemáticos.

Testes são experimentos controlados em que se procura obter evidência suficiente para confirmar que o programa, ou componente em teste, se comporta exatamente tal como especificado. Um teste corretamente realizado utiliza um critério de seleção de casos teste bem definido [Goodenough 75]. Cada caso teste define as condições a serem satisfeitas pelos dados de modo que determinado conjunto de condições de funcionamento sejam exercitadas. Determinadas as condições de cada caso teste, escolhem-se dados que satisfaçam estas condições. De posse destes dados, são calculados os resultados esperados em conformidade com a especificação. Executa-se o programa, utilizando os dados de teste escolhidos. Finalmente, comparam-se os resultados obtidos no teste com os resultados esperados. Evidentemente, sem uma especificação precisa, também não se consegue testar, uma vez que não se consegue determinar os resultados esperados, nem confrontar o que era esperado com o que foi obtido.

---

[2] Um *erro* é uma discrepância entre o resultado esperado e o resultado obtido. Erros tornam inutilizáveis os resultados do momento em que ocorreu o erro em diante até um momento em que se dá a recuperação do erro. Afetam, portanto, a confiabilidade ou a segurança do programa ou componente. Um *defeito* é uma discrepância entre o desempenho esperado e o obtido. Defeitos não afetam a confiabilidade dos resultados. Uma *falha* é um conjunto de uma ou mais porções de código que, quando exercitadas, provocam um erro ou defeito. *Diagnose* é a atividade de determinar as falhas existentes nos componentes a partir de erros ou defeitos. *Depuração* é a atividade de completa, correta e comprovadamente remover as falhas identificadas.

Testes não asseguram a total correspondência entre a especificação e o programa ou componente sendo testado. Em adição, em programas complexos, tende a ser inviável realizar testes suficientemente completos[3], conseqüentemente, porções significativas dos diversos componentes que compõem o programa poderão ficar sem suficiente exercitação ao testar o programa. No entanto, justamente os programas complexos são os que mais necessitam terem sua qualidade assegurada de forma sistemática[4].

Além dos problemas acima, existem ainda problemas inerentes ao próprio processo de realização do teste. Quando o programa e as estruturas de dados que utiliza forem complexos, tende a ser muito grande tanto a quantidade de casos teste a realizar, como o número de condições em cada caso teste. Isto dificulta a geração de dados teste corretos para cada um dos casos teste, e aumenta a chance do inspetor de qualidade deixar de observar discrepâncias entre o resultado obtido e o esperado, levando-o a aceitar como correto um programa ou componente contendo falhas, embora o teste apresente discrepâncias.

Dadas todas estas dificuldades ao realizar testes, tornam-se necessárias técnicas para:

- permitir a realização econômica de testes suficientemente completos;
- reduzir os riscos decorrentes de falha humana;
- reduzir os custos da realização de testes e da depuração.

Uma das formas de se atingir estes objetivos é a utilização de instrumentação embutida no código. Nesta norma definimos regras e recomendações para a incorporação de instrumentos em programas. A instrumentação monitora o comportamento do programa, alertando a ocorrência de erros tão logo se tornem perceptíveis. Isto, por sua vez, contribui para reduzir o esforço de depuração.

Os conceitos, regras e recomendações são de aplicação geral. No entanto, para uniformizar os exemplos, utilizaremos as convenções de C e C++ nos exemplos.

Conceitos

A *instrumentação* é formada por código e dados adicionais incorporados ao programa com a finalidade de monitorar o seu funcionamento. A instrumentação constitui custo adicional, uma vez que não contribui diretamente para a prestação do serviço do programa. Por outro lado, a instrumentação transforma um programa intolerante a falhas em um programa robusto ou até em tolerante a falhas. Um *programa intolerante a falhas* é um programa incapaz de operar corretamente em condições adversas ou hostis, em particular quando os erros forem provocados por falhas contidas no próprio programa. Programas intolerantes a falhas podem provocar enormes danos, uma vez que tendem a propagar os erros simplesmente por serem incapazes de detectar situações de funcionamento anômalas. Um *programa robusto* é um programa que é capaz de observar a ocorrência de um erro e interceptar a execução de modo que o dano provocado pelo erro se mantenha confinado. Um *programa tolerante a falhas* é um programa robusto que possui mecanismos de recuperação, habilitando-o continuar operando confiavelmente, mesmo que prestando serviços parciais – *graceful degradation* –.

A instrumentação é utilizada durante os testes, podendo ser mantida nos programas em produção. A decisão de manter instrumentação em programas em produção é de ordem econômica. Se o custo operacional da instrumentação for menor do que o custo nominal de possíveis danos, é recomendável manter-se a instrumentação no programa.

A instrumentação cuidadosamente projetada e incluída nos componentes do programa é particularmente interessante ao desenvolver programas de forma incremental e durante a realização do beta-teste[5]. Em ambos os

---

[3] Um teste é *completo* se satisfizer um *critério de completeza*, por exemplo cobertura de todas as instruções. Um teste completo não assegura ausência de erro, meramente assegura que todas as condições, selecionadas segundo um determinado critério de escolha de casos teste, foram devidamente contempladas no conjunto de todos os casos teste realizados.

[4] Dizemos que a qualidade é *assegurada* quando for garantidamente maior ou igual que a qualidade especificada – *qualidade esperada* –. Dizemos que a qualidade é *acidental* quando a qualidade resultante é obra do acaso, mesmo que seja maior do que a especificada.

[5] *Alfa-teste* é o teste de um programa ou componente no contexto da organização que está criando ou mantendo o programa. *Beta-teste* é o teste de programas ou componentes realizado em uma ou mais organizações cliente. O beta-teste visa verificar o comportamento do programa em condições normais de uso. Visa também examinar a utilidade (funcionalidade) e utilizabilidade (interface do usuário) do programa. *Teste de regressão* é o teste

casos os componentes instrumentados alertam situações anômalas de funcionamento, sem para isto necessitarem formas especiais de operação, tais como são requeridas quando se utilizam depuradores[6].

Projeto da estrutura de módulos

São condições essenciais para se poder desenvolver programas e componentes possuindo qualidade assegurada:

- os programas e componentes devem ter sido bem especificados – especificação de requisitos, especificação da essência, e critérios de aceitação;
- os programas e componentes devem ter sido bem arquitetados e projetados, ou seja devem possuir uma boa organização interna além de uma especificação de projeto.
- os programas e componentes devem possuir interfaces pequenas, clara e completamente definidas.

As normas a seguir estabelecem regras e recomendação visando atingir os requisitos acima enumerados.

PG-01 Regras e recomendações para a escolha de nomes de elementos em programas C e C++

PG-02 Regras e recomendações para o uso de constantes simbólicas em C e C++

PG-03 Regras e recomendações para a programação em C e C++

PG-04 Regras e recomendações específicas para a programação em C++

PG-05 Regras e recomendações para a inclusão de especificações no código de programas

PG-06 Regras e recomendações para fluxos de controle em algoritmos

PG-07 Regras e recomendações para avaliação de estruturas

Assertivas

Ao projetar um componente são definidas assertivas. Estas capturam as condições de correto funcionamento do componente [Liskov 87]. Assertivas determinam as condições que os dados e estados devem satisfazer ao entrar em uma função ou fragmento de código — *assertivas de entrada*[7], *pré-condições* ou *hipóteses* —, bem como as condições a serem satisfeitas ao sair desta função ou fragmento — *assertivas de saída*, *pós-condições* ou *requisitos.* Em outras palavras, as assertivas de entrada definem as condições necessárias para que o código a seguir possa operar corretamente, e as assertivas de saída definem as condições que deverão ser verdadeiras para que se possa dizer que o código que antecede a assertiva tenha operado corretamente.

Estados definem condições de funcionamento da plataforma. São exemplos de estados: arquivo aberto, cursor em determinada posição no vídeo, determinada cor selecionada, determinada fonte selecionada. Estados são usualmente registrados em alguma variável ou estrutura de dados, porém a mera alteração do valor destes elementos não muda o estado. Por exemplo, as variáveis *Linha* e *Coluna* podem ser utilizadas para indicar as coordenadas do cursor na janela. Porém, para efetivamente mudar a posição do cursor visualizado na tela, não basta alterar estes valores, é necessário transmiti-los para a função que controla a visualização do cursor.

Assertivas funcionais são utilizadas para argumentar a corretude[8] de um algoritmo. Um algoritmo estará correto se:

- o algoritmo pára, e

---

realizado após a alteração de um programa ou componente, com o intuito de verificar se as porções não alteradas continuam operando tal como especificado.

6 Um *depurador* (*debugger*) é uma ferramenta de apoio ao teste de programa que permite a execução controlada do programa, visualizando o progresso da execução através do programa fonte. Depuradores também permitem o exame e alteração do conteúdo de variáveis e de áreas de memória. Depuradores são usualmente fornecidos junto com os compiladores.

7 Uma *assertiva* é um conjunto de condições que os dados e estados de funcionamento devem satisfazer em determinado ponto do programa. As assertivas devem valer sempre, independentemente de como tenha sido realizado o processamento anterior ao ponto onde se encontra a assertiva. Assertivas podem conter hipóteses ou requisitos, estabelecendo critérios de aceitação tais como desempenho e facilidade de uso.

8 *Corretude: [correto + -(t)ude] S.f. propriedade (qualidade) de estar correto.* Preferimos utilizar um neologismo ao invés do termo *correção* para evitar a freqüente ambigüidade entre as acepções “corrigir” (tornar correto) e “estar correto”.

- sendo verdadeira a assertiva de entrada, possa ser demonstrada a validade da assertiva de saída após a execução do algoritmo.

Assertivas estruturais são assertivas que definem a validade de uma coletânea de dados ou estruturas de dados e dos estados associados a estes dados. Assertivas estruturais devem valer sempre que os correspondentes dados, estruturas de dados e estados não estiverem no processo de serem alterados.

Figura 1. Coordenadas ao exibir textos

Na Figura 1 são definidas as coordenadas de um cursor e de uma janela sobre um texto. Assume-se (hipótese) que todas as linhas tenham a mesma altura: *TamCrs*. Note que as assertivas somente valerão se esta hipótese for verdadeira. Caso a hipótese não seja verdadeira, não se pode dizer que o algoritmo esteja errado, porém é provável que ou a especificação esteja errada (não previu o caso de tamanhos de linha variáveis), ou o código que antecede a assertiva de entrada esteja errado (implementou algo que não foi especificado).

O seguinte conjunto de assertivas especifica o comportamento de um algoritmo *PosicionarJanela* cujo objetivo é posicionar a janela de tal forma que a linha corrente esteja sempre visível na janela. A linha corrente é a linha na qual se encontra o cursor.

```
AE:   TamTxt >= 0;
      TamJan >= TamCrs;
      0 <= LinCorr <= TamTxt
      Hipótese:   TamCrs constante para todas as linhas
                  LinCorr, TamTxt e LinJan são medidos em linhas
                  LinCrs, TamJan e TamCrs são medidos em pixel

      PosicionarJanela

AS:   0 <= LinCrs <= TamJan - TamCrs
            // Assegura o cursor no interior da janela
      0 <= LinJan <= LinCorr
            // Assegura a origem da janela antes ou na linha corrente
      LinCorr < LinJan + TamJan / TamCrs
            // Assegura a linha corrente antes do final da janela
      Se LinJan > 0 Então LinJan < TamTxt - TamJan / TamCrs
            // Maximiza o conteúdo da janela
```

Utilizando o mesmo exemplo, as assertivas estruturais serão:

```
      TamTxt, LinJan e LinCorr são medidos em número de linhas
```

```
TamJan, TamCrs e LinCrs  são medidos em pixel
A primeira linha é a linha 0
O primeiro pixel é o pixel 0

0 <= TamTxt
      // o texto é formado por zero ou mais linhas
0 <= LinCorr <= TamTxt
      // o cursor corrente está sobre uma das linhas do texto
      // ou sobre uma linha imediatamente a seguir da última
0 <= LinJan <= LinCorr
      // a origem da janela encontra-se antes ou sobre o cursor
      // corrente
TamCrs <= TamJan
      // o tamanho da janela é suficientemente grande para conter
      // pelo menos uma linha
0 <= LinCrs <= TamJan - TamCrs
      // o cursor visível encontra-se no interior da janela
LinCorr < LinJan + TamJan / TamCrs
      // o cursor corrente encontra-se antes do final da janela
Se LinJan > 0 Então LinJan < TamTxt - TamJan / TamCrs
      // a janela exibe pelo menos TamJan / TamCrs - 1 linhas do
      // texto, ou então está na origem do texto
Hipótese: TamCrs constante para todas as linhas.
```

Note que as assertivas estruturais contém, como era de se esperar, as assertivas de entrada e de saída da função *PosicionarJanela*. O uso de assertivas estruturais simplifica em muito a redação do programa, uma vez que elas sempre corresponderão às assertivas de entra e de saída dos algoritmos que manipulem a estrutura de dados. Por esta razão são chamadas de *invariantes da estrutura de dados.*

No texto da assertiva estrutural acima utilizamos uma linguagem semelhante à de expressões lógicas encontradas em programas, e também uma linguagem natural restrita. Embora as duas formas sejam redundantes entre si, a presença das duas aumenta a *inteligibilidade* e o *rigor* da especificação (expressões lógicas).

Assertivas executáveis

Em geral é possível traduzir uma parcela considerável das assertivas em código executável. Assertivas executáveis reduzem o esforço de diagnose, uma vez que evidenciam a ocorrência de uma anomalia tão logo ela se torne perceptível. Por exemplo, considere a função *Inserir novo elemento após elemento corrente* que insere um elemento em uma lista duplamente encadeada. Esta função recebe um ponteiro para um elemento desconectado de qualquer lista e um ponteiro para o elemento corrente da lista após ao qual deverá ser inserido o novo elemento. Ao entrar na função podemos verificar se o novo elemento está devidamente desconectado, bastando verificar se os ponteiros *anterior* e *próximo* deste novo elemento estão nulos. Podemos verificar, ainda, se o elemento da lista após ao qual se deve inserir está corretamente conectado aos seus vizinhos na lista, bastando para isto verificar se o antecessor do sucessor do elemento aonde se dará a inserção é este mesmo elemento. Ao sair da função podemos verificar se o elemento antecessor ao elemento introduzido é efetivamente o elemento da lista fornecido para a função de inserção, e se o novo elemento esta corretamente conectado aos seus vizinhos. O fragmento de código a seguir apresenta assertivas executáveis implementando texto descritivo anterior.

```
// pElemCorr ponteiro para o elemento da lista após ao qual será
//           inserido o novo elemento
// pElemNovo ponteiro para o novo elemento

pElemAux = pElemCorr->pProx ;
   if ( pElemAux != P_NIL )
   {
      if ( pElemAux->pAnt != pElemCorr )
      {
         Erro( ) ;
      }
```

```
   }
   if ( ( pElemNovo->pProx != P_NIL )
     || ( pElemNovo->pAnt  != P_NIL ))
   {
      Erro( ) ;
   }
```

Efetuar a inserção

```
if ( (   pElemNovo->pAnt  != pElemCorr )
    || ( pElemNovo->pProx != pElemAux  )
    || ( pElemCorr->pProx != pElemNovo ))
   {
      Erro( ) ;
   }
   if ( pElemAux != P_NIL )
   {
      if ( pElemAux->pAnt != pElemNovo )
      {
         Erro( ) ;
      }
   }
```

Assertivas executáveis reduzem o risco de falha humana, uma vez que verificam, durante a execução do programa, se o seu funcionamento está de acordo com o esperado. Assertivas executáveis viabilizam o uso de componentes quase corretos, uma vez que, caso ocorra um erro, interceptam o prosseguimento da execução do programa, evitando, assim, a propagação dos danos provocados por este erro. Ou seja, ao incluir assertivas executáveis estaremos aumentando a robustez dos programas.

C e C++ definem a função `assert` (utilizada em C++ através da macro `ASSERT`). Esta função pode ser utilizada para controlar a validade de assertivas. Caso a expressão lógica do argumento da função seja falso, é emitida uma mensagem indicando a linha de código fonte que contém a função e o programa é cancelado. Em outras linguagens a função `assert` não está definida, porém é fácil criar uma subrotina com a mesma finalidade.

Em muitas ocasiões a função `assert` não é satisfatória, uma vez que cancela imediatamente a execução, sem informar os valores que provocaram o cancelamento e sem realizar a necessária *arrumação da casa*[9]. O cancelamento imediato pode provocar conseqüências indesejáveis, como, por exemplo, deixar transações de alteração de banco de dados parcialmente executadas, inutilizando o banco de dados utilizado no teste.

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

i. emitir as mensagens de erro, possivelmente incluindo valores de variáveis relevantes, por exemplo, o nome do arquivo afetado pelo erro, a identificação do módulo e da linha de código onde foi identificado o erro, os nomes e valores das variáveis que não satisfazem as assertivas, etc.

ii. ao operar em *modo de depuração*, ativar um *explorador de dados* (*browser*). Este é um instrumento que permite examinar o conteúdo das variáveis e das estruturas de dados. O explorador de dados deve exibir os dados em um formato inteligível e consistente com a semântica das estruturas de dados sendo exploradas. Por exemplo, ao encontrar um erro em uma lista, o instrumento deve exibir o conteúdo da lista nos variados formatos admitidos pelos elementos da lista, devendo permitir também a exploração[10] dos elementos da lista segundo os diversos encadeamentos existentes. Em alguns casos pode ser interessante permitir que o explorador de dados possa editar as estruturas, permitindo, desta forma, que a pessoa realizando testes

---

[9] *Arrumação da casa* é o conjunto de operações necessárias para deixar íntegros os estados e as estruturas de dados permanentes da estação de trabalho após o término de execução, normal ou não, do programa. São exemplos de operações: esvaziar *bufferes* contendo dados ainda não gravados, fechar arquivos, efetuar o *roll back* de transações incompletamente realizadas, desligar *locks* permanentes, e restaurar o estado normal dos periféricos.

[10] *Exploração* é a ação de examinar os elementos das estruturas de dados, e navegar entre eles segundo os ponteiros definidos.

corrija a estrutura de dados, possibilitando, assim, a continuação dos testes. Os depuradores que acompanham os compiladores modernos permitem a exploração dos dados, mas, em geral, não o fazem de uma forma consistente com a semântica da estrutura de dados sendo explorada.

iii. decidir se pode continuar ou se deve cancelar. Dessa forma pode-se incluir assertivas executáveis sinalizando a ocorrência de situações não *plausíveis*. Estas são condições de funcionamento que, embora corretas, têm baixa probabilidade de ocorrência.

iv. caso seja decidido pelo cancelamento, ativar um instrumento de recuperação que arrume a casa.

É claro que o projeto e o desenvolvimento das assertivas executáveis, dos controladores de erro, dos exploradores e recuperadores de dados acima identificados, adicionam esforço ao desenvolvimento do programa. Por outro lado, estes instrumentos reduzem significativamente o esforço de diagnose, contribuindo para a redução do custo total de desenvolvimento do programa. Cabe salientar que o uso de assertivas executáveis não torna inúteis as ferramentas tais como depuradores, no entanto reduzem as ocasiões em que se necessita utilizá-las, bem como o tempo despendido em seções de depuração.

Assertivas executáveis em código de produção

A possibilidade de componentes de um programa serem imperfeitos torna desejável a permanência das assertivas executáveis nos programas de produção. Isto decorre do fato delas permitirem identificar, o mais cedo possível, as anomalias de funcionamento dos programas e, conseqüentemente, evitar a propagação de danos, tornando, assim, mais robustos os programas.

O projeto e a codificação dos instrumentos consome uma parcela significativa de recursos. Sempre que um componente de um programa for modificado, pode se tornar necessário utilizar os instrumentos para apoiar o controle de qualidade da modificação. Conseqüentemente, instrumentos não devem ser eliminados do código, e sim, devem ser desativados, de modo que não consumam recursos computacionais durante o uso produtivo. Instrumentos devem ser desativados a medida que o tempo médio entre erros sinalizados pela instrumentação em questão ultrapassar um nível mínimo considerado aceitável.

Assertivas executáveis aumentam o esforço computacional realizado. Este aumento de esforço pode conflitar com o uso produtivo do programa. Deve-se, então, estimar o custo operacional da instrumentação – 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. Caso o custo operacional seja menor do que o custo correspondente ao dano, recomenda-se que a instrumentação permaneça ativa no código de produção.

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 adicionais às contidas na própria estrutura de dados. Para podermos verificar a correção de uma estrutura de dados independentemente do estado de execução do programa, é necessário que ela seja auto-verificável.

A seguir vamos ilustrar como transformar uma estrutura de dados em uma estrutura auto-verificável. Considere uma estrutura *conjunto de listas lineares* em que cada um dos elementos deve pertencer a uma única lista. Para saber quais são todas as listas em uso, é necessário que a estrutura *conjunto de listas* identifique cada uma das listas do conjunto. Evidentemente o programa que utiliza estas listas conterá variáveis identificando estas listas, porém estas variáveis constituem informação adicional à estrutura de dados *conjunto de listas.* Para resolver este problema, a estrutura *conjunto de listas* pode conter uma lista de listas. Cada elemento desta lista de listas referencia a origem de uma das listas do conjunto. Evidentemente a lista de listas é uma das listas do conjunto de listas, portanto a lista de listas possui um elemento que referencia ela mesma.

Em uma lista linear cada elemento pertence a exatamente uma lista. Portanto, cada elemento necessita saber qual é o seu antecessor caso haja, pois senão um mesmo elemento poderia ter dois antecessores, violando a regra de pertinência a exatamente uma lista. Uma solução para este problema é tornar as listas duplamente encadeadas.

O fragmento de código C a seguir (faltam as declarações de dados), verifica se a estrutura satisfaz as regras estabelecidas:

```
ValidarListaLista( pOrgListaLista )
{
   pElemListaLista = pOrgListaLista ;
   while ( pElemListaLista != NULL )
```

Figura 2. Fragmento de Btree não auto-verificável

```
   {
      assert( pElemListaLista->pOrgLista->pAnt == NULL ) ;
      ValidarLista( pElemListaLista->pOrgLista ) ;
      pElemListaLista = pElemListaLista->pProx ;
   }
}

ValidarLista( pOrgLista )
{
   pElemLista = pOrgLista ;
   while ( pElemLista != NULL )
   {
      if ( pElemLista->pProx != NULL )
      {
         assert( pElemLista->Prox->pAnt  == pElemLista ) ;
      }
      if ( pElemLista->pAnt  != NULL )
      {
         assert( pElemLista->pAnt->pProx == pElemLista ) ;
      }
      pElemLista = pElemLista->pProx ;
   }
}
```

O encadeamento duplo é necessário para poder-se assegurar que cada par de elementos adjacentes em uma lista, sejam adjacentes somente entre si. A lista de listas é necessária, uma vez que não podemos saber quais são todas as listas em uso sem conhecer o estado interno de cada um dos componentes do programa. Se examinarmos o número de bytes consumidos nessa redundância, a medida que o número de elementos por lista aumenta e o tamanho do valor de cada elemento aumenta, o espaço relativo ocupado pelos dados redundantes vai diminuindo. Por exemplo, considerando 5 listas com em média 15 elementos, cada elemento com um tamanho médio de 30 bytes, o espaço relativo adicional necessário é de cerca de 13%, considerando ponteiros de 32 bits.

Examinemos um outro exemplo. A Figura 2 ilustra um fragmento de uma BTREE. Estas estruturas são utilizadas para aumentar o desempenho de acesso a dados contidos em tabelas ordenadas e armazenadas em bancos de dados. Em bancos de dados os nós das listas correspondem a páginas de tamanho constante e os ponteiros a identificadores de páginas. Em uma BTREE, cada elemento contém *MaxChaves* chaves das quais *NumChaves* estão em uso. Pelo menos metade das chaves devem estar em uso, ou seja deve valer sempre: *MaxChaves/2<=NumChaves<=MaxChaves*. Cada chave está associada a um valor ou a um ponteiro para um elemento filho. O elemento filho de um determinado elemento *X* contém chaves maiores ou iguais à chave associada ao ponteiro em *X* e menores do que a chave imediatamente a seguir de *X*, considerando-se os elementos de um patamar da árvore. A organização da BTREE na Figura 2 possui nenhuma redundância. Observando um elemento qualquer da estrutura não temos meios de verificar se este elemento pertence à BTREE e se está corretamente encadeado.

O primeiro passo para tornar auto-verificável a estrutura da Figura 2 é encadear todos os elementos irmãos de um mesmo patamar da árvore. Este encadeamento deve ser duplo. Pode-se, assim assegurar que a lista de elementos

de um mesmo patamar é bem formada. O segundo passo é encadear duplamente os elementos filho com os elementos pai. Desta forma torna-se possível examinar se todos os elementos de um determinado patamar são filhos de elementos pertencentes ao patamar imediatamente acima. O terceiro passo é criar uma lista de listas tal como descrita no exemplo anterior. Isto permite verificar se o conjunto de listas de elementos está corretamente formado. Finalmente, a primeira chave de um elemento filho deve ser igual à chave no correspondente elemento pai. Desta forma pode-se verificar se o encadeamento pai filho é consistente. O resultado desta adição de redundância pode ser visto na Figura 3.

Figura 3. Uma estrutura btree auto-verificável

A seguir apresentamos as assertivas estruturais da estrutura da Figura 3, utilizando texto em português. Esta é uma das possíveis formas de se redigir uma especificação da estrutura de dados.

- Cada patamar da árvore é uma lista de elementos da árvore.
- O patamar raiz é uma lista de exatamente um elemento.
- O elemento inicial de cada uma das listas patamar é referenciado pela lista de listas.
- Cada lista, inclusive a lista de listas, é duplamente encadeada.
- Cada elemento da árvore possui um ponteiro para o elemento pai. Caso o elemento seja o elemento raiz da BTREE, o ponteiro pai será nulo.
- Exceto para o elemento raiz da BTREE, a primeira chave de cada elemento da árvore é igual à chave associada ao ponteiro filho contido no correspondente elemento pai.
- Exceto para o elemento raiz da BTREE, o número de chaves utilizadas em cada elemento deve ser maior ou igual à metade das chaves possíveis no elemento.
- As chaves contidas em um elemento estão ordenadas em ordem crescente. A última chave de um elemento é menor do que a primeira chave do elemento seguinte na lista de elementos.

Os fragmentos de código C++ a seguir ilustram as assertivas estruturais executáveis da BTREE implementadas de acordo com o diagrama da Figura 3. Note que para redigir este código foi necessário transformar a especificação informal em português, para a definição formal utilizada no código. Com intuito de simplificar o código, utilizamos ponteiros para implementar a BTREE. Numa implementação utilizando arquivos, os ponteiros devem ser substituídos por chaves e o acesso a valores apontados deve ser realizado por funções de acesso a registros contendo essas chaves. O fragmento de programa verifica se um determinado elemento *pElem* da BTREE está correto com relação às assertivas estruturais. A validação de toda a BTREE pode ser realizada caminhando-se pela BTREE em alguma ordem. A cada vez que um determinado elemento da BTREE for visitado, ele será validado de acordo com o código a seguir.

```
// Verificar se o encadeamento com os elementos vizinhos está correto
     if ( ( pElem->pProx == P_NIL )
       && ( pElem->pAnt  == P_NIL ) )
     {
     // É elemento raiz da BTREE
        ASSERT( pElem->pPai == P_NIL )
        ASSERT( PertenceListaLista( pElem )) ;
     } else
     {
     // É elemento normal
```

```
         ASSERT( MAX_CHAVES / 2    <= pElem->NumChaves ) ;
         ASSERT( pElem->NumChaves  <= MAX_CHAVES       ) ;
         ASSERT( pElem->pPai != P_NIL )
         if ( pElem->pProx != P_NIL )
         {
            ASSERT( pElem->pProx->pAnt == pElem ) ;
         }
         if ( pElem->pAnt != P_NIL )
         {
            ASSERT( pElem->pAnt->pProx == pElem ) ;
         } else
         {
            ASSERT( PertenceListaLista( pElem )) ;
         }
      }

   // Verificar se a ordenação das chaves está correta
      for ( i = 0 ; i < pElem->NumChaves - 1 ; i++ )
      {
         ASSERT( pElem->Filho[ i     ].Chave <
                 pElem->Filho[ i + 1 ].Chave ) ;
      }
      if ( pElem->pProx != P_NIL )
      {
         ASSERT( pElem->Filho[ pElem->NumChaves - 1 ].Chave <
                      pElem->pProx->Filho[ 0 ].Chave ) ;
      }
      if ( pElem->pAnt != P_NIL )
      {
         ASSERT( pElem->pAnt->Filho[ pElem->pAnt->
                 NumChaves - 1 ].Chave < pElem->Filho[ 0 ].Chave ) ;
      }

   // Verificar se o encadeamento pai->filho está correto
      for ( i = 0 ; i < pElem->NumChaves ; i++ )
      {
          ASSERT(( pElem->Filho[ i ].pFilho )->Filho[ 0 ].Chave ==
                        pElem->Filho[ i ].Chave ) ;
          ASSERT(( pElem->Filho[ i ].pFilho )->pPai == pElem ) ;
      }

   // Verificar se o encadeamento filho->pai está correto
      if ( pElem->pPai != P_NIL )
      {
         Encontrou = FALSE ;
         for ( i = 0 ; i < ( pElem->pPai )->NumChaves ; i++ )
         {
            if (( pElem->pPai )->Filho[ i ].Chave ==
                        pElem->Filho[ 0 ].Chave )
            {
               ASSERT(( pElem->pPai )->Filho[ i ].pFilho == pElem ) ;
               ASSERT( !Encontrou ) ;
               Encontrou = TRUE ;
               break ;
            }
         }
         ASSERT( Encontrou ) ;
      }
```

Co-rotinas de verificação

O processamento de verificação de uma estrutura completa pode ser bastante demorado. No entanto, em plataformas dedicadas a um único usuário é comum ocorrerem longos tempos ociosos em virtude da espera por ação do usuário (tecla ou movimentação de *mouse*). Estes tempos ociosos podem ser utilizados para realizar a verificação da correção estrutural das estruturas de dados. Dessa forma a verificação é realizada sem afetar o tempo real do processamento de transações.

Para tornar 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. Em adição, cada passo de verificação deve examinar completamente um determinado aspecto da estrutura de dados. No exemplo da BTREE, um passo de verificação corresponde a validar um elemento da estrutura.

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 estruturais sendo controladas podem não valer. Isto torna necessário o bloqueio (*lock*) da estrutura de dados enquanto estiver sendo processado o passo de verificação.

A quebra do verificador em passos de verificação requer que o processamento possa suspender ao término de cada passo e retomar a execução com o passo a seguir, independentemente do processamento que possa ter sido realizado entre o instante da suspensão e o da retomada. 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. Um processo de verificação que satisfaça estes requisitos é uma *co-rotina*[11]. Note que esta co-rotina deve ser absolutamente transparente ao programa de produção, conseqüentemente a informação de progresso da co-rotina 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 desenvolver um pequeno controlador de ativação do verificador. Este nada mais é do que um ciclo que repete enquanto o computador estiver esperando por alguma ação do usuário. A cada ciclo é ativado o verificador e, conseqüentemente, é realizado um passo de verificação. Após um número finito de ativações o verificador deverá ter concluído todos os passos necessários para a verificação integral da estrutura de dados.

Resta resolver um pequeno problema de sincronismo. Caso o processamento entre o instante de suspensão e o de retomada da co-rotina 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 de uma estrutura de dados. Agora basta que, ao iniciar um passo de verificação, o verificador compare o *time-stamp* registrado ao iniciar a verificação em progresso. Caso *time-stamp* da estrutura sendo verificada seja diferente do *time-stamp* contido no verificador, ocorreu uma alteração. Uma solução conservadora é reiniciar o caminhamento na origem da estrutura sendo verificada.

O esboço de código C++ a seguir ilustra uma classe utilizada para verificar uma estrutura. Caso se esteja programando em C, é necessário criar a estrutura de dados contendo a informação de controle e passá-la como parâmetro para as funções. Note a semelhança deste conjunto de funções com uma função geradora tal como definida na norma *PG-06 Regras e recomendações para fluxos de controle em algoritmos.*

```
// Classe do verificador
   VerificarEstrutura

// Métodos da classe

   VerificarEstrutura( pEstrutura ) ;
         // Cria um objeto verificador vinculado à estrutura a ser
         // verificada
```

[11] Uma *co-rotina* é um sub-programa que memoriza o estado em que suspendeu a execução para, mais tarde, poder retomar a execução a partir exatamente deste estado.

```
IniciarVerificador( )
      // Muda o estado corrente do verificador de modo que
      // recomece a validar do início da estrutura

VerificarPasso( )
      // Efetua um novo passo de verificação a cada ativação.
      // Ativações sucessivas verificam toda a estrutura.
   // O esquema do algoritmo é o seguinte:
      if ( pOrigemEstrutura.ObterTimeStamp() !=
                TimeStampVerificador )
      {
         IniciarVerificador() ;
      }
      else {
         Bloquear a estrutura contra alterações
         Preservar o contexto externo afetado pelo verificador
         Restaurar o contexto do verificador na última suspensão
         Efetuar o passo de verificação corrente
         Se encontrou erro
         {
            Ativar o explorador da estrutura
            Decidir como continuar
         }
         Avançar para o próximo passo de verificação
         Salvar o contexto do verificador
         Restaurar contexto externo
         Retirar o bloqueio da estrutura
      }

BOOL AcabouVerificador( )
      // Retorna TRUE caso o verificador tenha examinado toda a
      // estrutura de dados. Os métodos IniciarVerificador,
      // VerificarPasso e AcabouVerificador formam uma função
      // geradora que poderá ser utilizada em estruturas de
      // repetição

~VerificarEstrutura( )
      // Elimina o objeto verificador
```

Controle de acesso a espaços de dados

Programas em C e C++ utilizam espaços de dados alocados dinamicamente e uma profusão de ponteiros. Ponteiros oferecem sérios riscos de integridade a programas. Além disso, erros de uso de ponteiros podem ser muito difíceis de diagnosticar devido à dificuldade de se estabelecer relações de causa e efeito. Ao programar utilizando a biblioteca de classes MFC, pode-se utilizar diversas funções e macros de controle tornadas disponíveis por esta biblioteca. Nos demais casos torna-se necessário criar seu próprio controle.

Recomenda-se que seja criadas funções de alocação e liberação de espaços de memória para fins de depuração. Em C a função de alocação possui o esqueleto descrito a seguir.

```
   /* Declaração de tipo utilizado para alocar espaços controlados */
      typedef struct tgControle
      {
         struct tgControle * pAnt ;
         struct tgControle * pProx ;
         size_t              Tamanho ;
         unsigned            idTipo ;
         char                ControleAntes[ TAM_C ] ;
         char                Valor[         TAM_C ] ;
         char                ControleApos[  TAM_C ] ;
```

```
} tpControle ;
```

Figura 4. Lista de espaços alocados

Este tipo é utilizado para alocar os espaços solicitados. Os ponteiros `pAnt` e `pProx` estabelecem uma lista duplamente encadeada de todos os espaços alocados e ainda não liberados. O valor `idTipo` deve ser consistente com o tipo semântico[12] associado ao ponteiro. Este campo permite verificar se os tipos estão sendo utilizados corretamente em tempo de execução. O valor de `Tamanho` é o tamanho útil alocado para o valor do usuário. Os valores de `ControleAntes` e `ControleApos` são constantes diferentes de zero e de combinações de caracteres ASCII que são utilizados para verificar se o usuário acidentalmente extravasou áreas ao atribuir valores. A Figura 4 mostra graficamente a organização da lista de espaços alocados.

```
/* Função de alocação de memória */
   void * MeuMalloc( size_t Tamanho )
   {
      tpControle * pEspaco ;
      pEspaco = ( tpControle * ) malloc( sizeof( tpControle ) +
                                         Tamanho - TAM_C ) ;
      if ( pEspaco == NULL )
      {
         return NULL ;
      }
   /* Iniciar os valores de controle do espaço */
      pEspaco->pAnt      = NULL ;
      pEspaco->pProx     = pOrgLista ;
      pOrgLista          = pEspaco ;
      pEspaco->idTipo    = TIPO_NULO ;
      pEspaco->Tamanho   = Tamanho ;
      memcpy( pEspaco->ControleAntes ,   VALOR_C , TAM_C ) ;
      memcpy( pEspaco->Valor + Tamanho , VALOR_C , TAM_C ) ;
      return ( void * ) &( pEspaco->Valor ) ;
   }
```

As constantes `VALOR_C` e `TAM_C` definem, respectivamente o valor de controle de extravasão de área e o tamanho deste valor. Um espaço de dados somente estará íntegro caso estes valores estejam presentes em todos

[12] O *tipo computacional* corresponde ao tipo declarado da variável, por exemplo `int`, `long`, `char*`, `tpMeuTipo`. O *tipo semântico* determina o significado do valor. Por exemplo: “velocidade em m/s”, “distância em m”, “nome de pessoa”, “nome de item de estoque”. Considere, por exemplo uma variável *string X*. Se esta variável contém nomes de pessoas, é evidente que não se pode atribuir a ela nomes de itens de estoque, embora do ponto de vista tipo computacional tal atribuição seja permitida.

os elementos alocados. O usuário de `MeuMalloc` não deve saber da existência dos atributos de controle de integridade, portanto a função deve retornar o ponteiro para o atributo `Valor` da estrutura. A variável `pOrigemAlloc` é o ponteiro para a origem da lista de espaços alocados.

```
/* Função de validação de um ponteiro para espaço alocado */
   void ValidarPonteiro( void * Ponteiro , tpTipoArea idTipo )
   {
      tpControle * pEspaco ;
   /* Calcular a origem real do espaço */
      pEspaco = ( tpControle * ) (( char * )
           Ponteiro - ( sizeof( tpControle ) - TAM_C - TAM_C )) ;
   /* Controlar a correspondência de tipos */
      if ( idTipo != TIPO_QUALQUER )
      {
         if ( pEspaco->idTipo != IdTipo )
         {
            TratarErro( TIPO_NAO_CASA ) ;
         }
   /* Controlar os delimitadores do espaço útil */
      if ( memcmp( pEspaco->ControleAntes , VALOR_C , TAM_C )
                != 0 )
      {
         TratarErro( EXTRAVASOU_ANTES ) ;
      }
      if ( memcmp( &( pEspaco->Valor[ pEspaco->TamElem ]) ,
                VALOR_C , TAM_C ) != 0 )
      {
         TratarErro( EXTRAVASOU_APOS ) ;
      }
   /* Controlar os encadeamentos na lista de espaços alocados */
      if ( pEspaco->pAnt != NULL )
      {
         if ( ( pEspaco->pAnt )->pProx != pEspaco )
         {
            TratarErro( ENCADEAMENTO_ANTES ) ;
         }
      } else {
         if ( pOrgLista != pEspaco )
         {
            TratarErro( NAO_EH_INICIO_LISTA ) ;
         }
      }

      if ( pEspaco->pProx != NULL )
      {
         if ( ( pEspaco->pProx )->pAnt != pEspaco )
         {
            TratarErro( ENCADEAMENTO_APOS ) ;
         }
      }
   }
```

Esta função verifica se os valores de controle em torno da área destinada ao usuário estão íntegros. Verifica também se o encadeamento do espaço está íntegro. Em virtude da aritmética utilizada para determinar a origem da estrutura, esta função não é garantidamente portátil. Diferentes plataformas possivelmente requerem algum ajuste devido à inclusão de campos de enchimento para assegurar o alinhamento dos diversos campos.

```
/* Função de liberação de espaço alocado */
```

```
void MeuFree( void * Ponteiro )
{
   tpControle * pEspaco ;
   ValidarPonteiro( Ponteiro ) ;
   pEspaco = ( tpControle * ) (( char * )
        Ponteiro - ( sizeof( tpControle ) - TAM_C - TAM_C )) ;
/* Desencadear o espaço */
   if ( pEspaco->pAnt != NULL )
   {
      ( pEspaco->pAnt  )->pProx = pEspaco->pProx ;
      ( pEspaco->pProx )->pAnt  = pEspaco->pAnt ;
   } else {
      pOrgLista = pEspaco->pProx ;
   }
/* Liberar o espaço */
   free( ( void * ) pEspaco ) ;
}
```

Esta função elimina um espaço de dados previamente alocado. Antes de desalocar, verifica se a estrutura está corretamente formada.

```
/* Função para a definição de um tipo de ponteiro */
   void DefinirTipoEspaco( void *   Ponteiro ,
                           unsigned IdTipo    )
   {
      tpControle * pEspaco ;
      ValidarPonteiro( Ponteiro ) ;
      pEspaco = ( tpControle * ) (( char * )
           Ponteiro - ( sizeof( tpControle ) - TAM_C - TAM_C )) ;
   /* Definir o tipo do espaço */
      if ( idTipo == TIPO_QUALQUER )
      {
         pEspaco->IdTipo = IdTipo ;
      } else if ( ( pEspaco->idTipo == TIPO_QUALQUER )
                || ( pEspaco->idTipo == TIPO_NULO     ))
      {
         pEspaco->IdTipo = IdTipo ;
      } else
      {
         TratarErro( REDEFINICAO_TIPO ) ;
      }
   }
```

Tipos de instrumentos

Um *depurador* permite o monitoramento simbólico e interativo do programa em execução, desde que tenha sido compilado de forma apropriada. Embora seja uma ferramenta muito poderosa, a sua utilização indisciplinada pode gerar enorme perda de tempo. Antes de iniciar uma sessão de depuração deve-se conhecer os erros encontrados, e deve-se ter hipóteses bem fundamentadas quanto à região do código que pode conter a falha sendo procurada. Depuradores são usualmente fornecidos junto com os compiladores.

Um *dump* é uma impressão ou exibição formatada do conteúdo de memória ou de arquivos. Dumps mais elaborados formatam os valores das variáveis e dos campos de estruturas de acordo com o tipo computacional declarado, antecedendo o valor exibido com o nome simbólico da variável ou do campo. Dumps mais simples exibem o conteúdo da memória em formato hexadecimal sem incluir títulos. No caso de dumps hexadecimais, cabe ao usuário encontrar o valor das variáveis de interesse, utilizando um mapa de memória para determinar os endereços de interesse. Alguns sistemas operacionais tornam disponíveis dumps hexadecimais e alguns processadores de linguagem fornecem dumps simbólicos, muitas vezes sob a forma de *post-mortem dump*, ou seja dumps gerados caso o programa seja cancelado em virtude de algum erro de execução.

Um *trace* é uma seqüência de impressões ou exibições de algumas variáveis selecionadas. O valor destas variáveis ou campos é exibido sempre que a execução passa por determinado ponto do código, ou quando o valor for modificado. Traces são muitas vezes implementados através de instruções de impressão ou exibição incluídas no código.

Um *explorador de dados* é um módulo interativo especificamente projetado e implementado para exibir dados selecionados, podendo navegar sobre ponteiros, referências e chaves contidas nas estruturas de dados ou arquivos. A exibição mostra os valores corretamente formatados e identificados segundo o seu significado. Os valores podem estar em memória ou podem ser lidos de arquivos. Usualmente exploradores permitem a alteração dos dados sendo exibidos. Desta forma pode-se corrigir erros durante os testes e continuar executando estes testes. Pode-se também deturpar os valores contidos em estruturas de dados afim de verificar se os validadores de assertivas estruturais estão correta e completamente implementados.

Um *recuperador de dados* é um módulo especificamente projetado e implementado para recuperar estruturas de dados danificadas por alguma razão. Recuperadores de dados são especialmente interessantes quando forem capazes de recuperar arquivos parcialmente destruídos.

Um *arrumador da casa* é um módulo especificamente projetado para restaurar os estados da estação de trabalho ao seu estado normal, e para restaurar os arquivos permanentes, assegurando a validade das respectivas assertivas estruturais.

Um *eco* é um instrumento que exibe os valores recebidos e retornados por uma função. Poder ser utilizado, também, para exibir os valores recebidos de arquivos e de dispositivos de entrada. Ecos permitem verificar se os dados de entrada fornecidos para um componente correspondem aos valores definidos para os casos teste. Permite também verificar se o resultado retornado é consistente com a especificação da função independente do contexto em que esta função esteja operando. Desta forma ecos permitem apoiar testes detalhados. Ao invés de exibir valores de entrada e saída, ecos podem ativar exploradores de dados capazes de exibir todos estes valores.

Ao invés de incorporar código gerador do eco na função a ser instrumentada, pode-se desenvolver uma função cujo objetivo é estritamente o de gerar ecos. Desta forma pode-se controlar a função na forma com que será utilizada em produção. O esquema de código a seguir ilustra esta forma de implementar

```
// Chamada contida no programa cliente
   ...
#ifdef _DEBUG
   ... FuncECO( p1, p2, ... ) ... ;
#else
   ... Func( p1, p2, ... ) ... ;
#endif
...

// Implementação da função eco
#ifdef _DEBUG
   tipof FuncECO( tipo1 p1, tipo2 p2, ... )
   {
      tipof ValorRet ;
      exibir o eco dos valores de entrada, inclusive globais
      ValorRet = Func( p1, p2, ... ) ;
      exibir o eco dos valores de saída, inclusive globais
      return ValorRet ;
   }
#endif

// Implementação da função de produção
   tipof Func( tipo1 p1, tipo2 p2, ... )
   ...
```

Uma *janela de visita* é um instrumento controlado por uma condição de ativação. Diz-se que a janela de visita está *aberta* sempre que o correspondente instrumento estiver sendo executado. Usualmente o instrumento contido em uma janela de visita é um explorador de dados. Janelas de visita podem ser abertas por solicitação do usuário, por exemplo através da seleção de um comando contido em algum menu. Isto pode ser conveniente

durante o desenvolvimento de programas manipulando estruturas de dados complexas. Janelas de visita podem ser abertas ao encontrar uma assertiva executável em erro. Finalmente, janelas de visita podem ser associadas a transações. Sempre que uma transação contendo uma janela passar por um ponto de controle, ela será aberta. As demais transações passam pelos pontos de controle sem ativar a instrumentação.

Definição da norma

Recom. 1: Evite o uso de depuradores.

Embora depuradores sejam instrumentos poderosos, o seu uso indisciplinado leva a enormes perdas de tempo. Um uso disciplinado de depuradores requer que se disponha de erros de execução observados e de hipóteses razoavelmente fundamentadas quando às possíveis causas de tais erros. Para isso, é necessário que se tenha realizado alguns testes, e tenha registrado as discrepâncias observadas entre resultados esperados, calculados a partir da especificação, e resultados obtidos.

Regra 2: Todos os instrumentos devem ser deixados no código fonte.

Regra 3: Instrumentos permanentes devem ser tratados como código normal. Instrumentos utilizados somente durante o teste de componentes devem estar contidos em um controle de compilação condicional. A estrutura de código a utilizar é:

```
#ifdef _DEBUG
   código do instrumento
#endif
```

Regra 4: Utilize a constante `_DEBUG` para controlar a compilação condicional dos instrumentos normais.

A linha de ativação do compilador deve conter o parâmetro `/D_DEBUG` caso a compilação deva gerar o código de instrumentação. Alguns compiladores não permitem definir símbolos no comando de ativação do compilador. Neste caso incorpore uma declaração `#define _DEBUG` no arquivo de parâmetros dependentes de plataforma `PDR<id compilador>.INC` descrito na norma *PG-03 Regras e recomendações para a programação em C e C++*.

Recom. 5: Utilize constantes `_DEBUG1`, `_DEBUG2` etc. para controlar a compilação condicional em diferentes níveis de granularidade.

Utilizando diferentes níveis de granularidade de depuração, quanto maior o número, menos instrumentação estará ativa. Antes do primeiro comando de inclusão, insira um fragmento de código semelhante a:

```
#ifdef _DEBUG2
   #undef _DEBUG1
   #undef _DEBUG
#endif
#ifdef _DEBUG1
   #undef _DEBUG
#endif
```

Regra 6: Funções e métodos utilizados exclusivamente pela instrumentação devem estar contidos em um controle de compilação condicional. O controle de compilação condicional deve envolver tanto a implementação da função ou do método como também as respectivas declarações de protótipos.

Recom. 7: Todas as estruturas de dados complexas devem ser auto-verificáveis.

Estruturas auto-verificáveis permitem a inclusão de instrumentação de controle de integridade das estruturas de dados. Desta forma reduz-se em muito o custo dos testes e da depuração. O custo de espaço de memória adicional necessário para conter as redundâncias de dados é, de maneira geral, pequeno quando comparado com a economia conseguida ao testar ou depurar o programa.

Recom. 8: Para todas as estruturas de dados complexas deve ser desenvolvido um explorador e um validador estrutural.

Recom. 9: Para todos os arquivos não texto deve ser desenvolvido um explorador capaz de exibir o conteúdo formatado de registros selecionados do arquivo.

Recom. 10: Inspecione e teste o validador estrutural com rigor.

Regra 11: Funções e métodos que alteram o encadeamento de estruturas de dados complexas, devem conter antes do `return` uma chamada ao validador estrutural na forma:

```
#ifdef _DEBUG
   ValidarTodaEstrutura( ) ;
#endif
```

Caso a estrutura de dados possa se tornar muito grande, crie um validador que verifique o elemento corrente e todos os elementos adjacentes. Embora isto não assegure a integridade de toda a estrutura, certamente verificará a integridade dos elementos afetados pela função ou método.

Recom. 12: Programas fazendo intensivo uso de estruturas de dados dinamicamente alocadas, devem conter instrumentação de controle de acesso através de ponteiros e referências.

Recom. 13: Ao controlar acesso a espaços dinamicamente alocados, inclua instrumentação verificando se o tipo semântico do espaço de dados é consistente com o tipo semântico associado ao ponteiro.

Programas que utilizam estruturas de dados com grande volume de encadeamentos são difíceis de testar e depurar. Reduz-se esta dificuldade controlando o uso de dados dinamicamente alocados tal como discutido na seção *0 Controle de acesso a espaços de dados.*

Bibliografia

[Gerhart 76] Gerhart, S.L.; Yelowitz, L. "Observations of fallibility in applications of modern programming methodologies"; *IEEE Transactions on Software Engineering*; setembro 1976

[Goodenough 75] Goodenough, J.B.; Gerhart, S.L.; "Toward a theory of test data selection"; *IEEE Transactions on Software Engineering*; junho 1975

[LG86] Liskov, B.; Guttag, J.; *Abstraction and Specification in Program Development*; McGraw Hill; 1986

[Microsoft93] Microsoft; *Books On-Line*; Documentação em CD-ROM do compilador e ambiente de programação Visual C/C++, versão 1.5; 1993