EDENILSON JOSÉ DA SILVA

PREDTOOL: UMA FERRAMENTA PARA APOIAR O TESTE BASEADO EM PREDICADOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre do Curso de Mestrado em Informática, Setor de Ciências Exatas da Universidade Federal do Paraná.

Orientadora: Prof.ª Dr.ª Silvia Regina Vergilio

CURITIBA

2003

Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Edenilson José da Silva, avaliamos o trabalho intitulado, "*PREDTOOL: Uma Ferramenta para Apoiar o Teste Baseado em Predicados*", cuja defesa foi realizada no dia 25 de novembro de 2003, às dez horas, no Auditório do Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 25 de novembro de 2003.

Profª. Dra. Silvia Regina Vergilio
DINF/UFPR – Orientadora

Profª. Dra. Simone do Rocio Senger de Souza
UEPG - Membro Externo

Prof. Dr. André Luiz Pires Guedes
DINF/UFPR - Membro Interno

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
Setor de Ciências Exatas
Coordenação de Pós-Graduação em Informática

AGRADECIMENTOS

Este trabalho teve a cooperação de diversas pessoas, mesmo porque, sozinho, pouco pode ser realizado. Foram muitos amigos, colegas de trabalho, colegas de estudo, mestres, profissionais com quais mantive contato e pude solucionar dúvidas e trocar experiências

Algumas destas pessoas colaboram com idéias, outras com auxílio inestimável e préstimo de seu tempo na resolução de dúvidas; alguns ouviram as lamentações; outras ainda deram suporte, através de gestos, palavras e atos nas horas difíceis, onde a vontade e persistência para conclusão do trabalho pareciam minguar.

Algumas pessoas colaboram de forma muito especial, emprestando sua presença física e espiritual, nos momentos de dificuldade de execução ou raciocínio para desenvolver o trabalho.

A todas estas pessoas devo minha gratidão. Não ouso fazer menção de seus nomes, pois poderia me alongar demais, o que talvez me levasse a ser omisso, algo que consideraria imperdoável. Fica o agradecimento coletivo, com a certeza de que o tempo permitirá retribuir exponencialmente toda e qualquer ajuda prestada.

Mas, algumas pessoas, não posso deixar de citar. Primeiramente a minha família: Thatiana, minha esposa amada; Edenilson, Arthur e Erica; meus filhos e presentes de Deus para minha vida. A vocês, minha eterna gratidão pela compreensão das horas ausente. E a orientadora deste trabalho, professora Silvia Regina Vergilio, pela oportunidade ímpar de poder realizar este trabalho. Penso que, sem sua orientação sempre segura, paciente e competente, não teria finalizado esta dissertação.

SUMÁRIO

RESUMO

A atividade de teste é fundamental dentro da Engenharia de Software, especialmente para a melhoria da qualidade dos programas criados. Mas esta atividade, apesar de importante, tem sido colocada em segundo plano, muitas vezes em conseqüência da dificuldade de realização de testes de forma manual, devido ao consumo de tempo e a possibilidade de ocorrência de inúmeros erros. Para reduzir os custos e aumentar o número de defeitos revelados no teste, foram propostos diversos critérios de teste. Esses critérios têm como objetivo guiar o testador na seleção e na avaliação de conjuntos de casos de teste. Esta dissertação aborda critérios estruturais de teste, mais particularmente os critérios BOR (Boolean OperatoR testing) e BRO (Boolean and Relational Operator testing), baseados em predicados e que tem como objetivo revelar defeitos presentes em predicados compostos do programa em teste. Uma ferramenta que automatiza os critérios BOR e BRO foi implementada e é descrita. A ferramenta chama-se PredTOOL, e testa especificamente programas desenvolvidos na linguagem C. A implementação da ferramenta tornou possível a realização de um experimento de avaliação dos critérios BOR e BRO cujos resultados permitem a comparação desses critérios com dois outros critérios estruturais, Todos-Arcos e Todos Potenciais-Usos, implementados pela ferramenta Poke-Tool. Da análise dos resultados obtidos, é sugerida uma estratégia para aplicação dos critérios estruturais analisados.

ABSTRACT

The testing activity is a fundamental phase in the Software Engineering process, especially for improving the quality of the developed programs. But this activity, in spite of being important, has been put in a second plan, maybe due to the difficulty of accomplishing tests manually. This spents a lot of effort and time,and in general a great number of faults remains.To reduce the costs and to increase the number of defects revealed in the test, several testing criteria were proposed. Those criteria have as objective to guide the tester in the selection and evaluation of test case sets. This works focuses structural testing criteria, more particularly BOR (Boolean OperatoR testing) and BRO (Boolean and Relational Operator testing) criteria, that are based on predicates and have the goal of revealing faults in compound predicates of the program under testing. A tool that implements BOR and BRO criteria was implemented and it is described. The tool calls PredTOOL, and tests C programs. The implementation of the tool makes possible the accomplishment of an experiment for evaluation of BOR and BRO criteria. The obtained results allow the comparison of those criteria with two other structural criteria: All-edges and All Potential-Uses, implemented by the tool Poke-Tool. Those results are used to propose a strategy for application of the studied structural criteria.

LISTA DE FIGURAS

LISTA DE TABELAS

1. – INTRODUÇÃO

O panorama em que o software é desenvolvido está diretamente ligado à evolução dos sistemas computadorizados, mais especificamente do hardware. Junto com a mudança dos processadores da válvula para os dispositivos microeletrônicos sugiram softwares mais complexos e sofisticados. Este período ficou conhecido como uma nova revolução industrial [OSB79], também apontando para o papel essencial que a informação e o conhecimento controlados por computador teriam no século XXI.

Desde então muito se evoluiu em termos dos métodos e técnicas utilizados no desenvolvimento de software. Porém, mesmo após os progressos alcançados, alguns destes problemas perduram até hoje, como por exemplo: os custos elevados associados ao desenvolvimento de software, a necessidade de desenvolvê-lo em prazos cada vez menores e os freqüentes problemas encontrados no software produzido. Práticas de Engenharia de Software (aplicação de uma abordagem sistemática, disciplinada e quantificável para o desenvolvimento, operação e manutenção de software) têm sido adotadas em todas as fases do ciclo de vida para tentar minimizá-los.

Dentro deste contexto, uma das principais etapas no desenvolvimento de um sistema é o Teste de software. É muito importante conhecer os erros mais freqüentes no desenvolvimento de sistemas, pois, conhecendo os erros, é possível tentar evitá-los. Ao estudarmos os Fundamentos dos Testes de Software, precisamos analisar diversos pontos, entre eles [SAN94]:

- Os fatores de motivação dos Testes de Software;
- Importância dos Testes de Software;

- Relação custo/benefício dos Testes de Software;
- Os métodos de Testes existentes;

Enfim, fazer uma relação ponderante entre o que é necessário e o que é esperado de um bom teste de software.

Apesar da importância, a atividade de teste costuma ser a mais negligenciada do desenvolvimento de software, e por isso mesmo, é a área onde há maior possibilidade de ganho de escala, de produtividade e de qualidade, principalmente para pessoas que esperam o menor número de defeitos, ela não pode ser menosprezada. Para citarmos Deutsch [DEU79]:
"O desenvolvimento de sistemas de software envolve uma série de atividades de produção em que as oportunidades de injeção de erros humanos são enormes. Erros podem começar a acontecer logo no começo do processo, onde os objetivos... podem estar errônea ou imperfeitamente especificados, além de erros que venham a ocorrer em fases de projeto e desenvolvimento posteriores... Por causa da incapacidade que os seres humanos têm de executar e comunicar com perfeição, o desenvolvimento de software é acompanhado por uma atividade de garantia de qualidade."

A atividade de teste de software é um elemento crítico da garantia de qualidade de software e representa a última revisão das especificações, projeto e codificação [PRE00]. Salienta-se que a atividade de teste tem sido apontada como uma das mais onerosas no desenvolvimento de software. Apesar deste fato, Myers observa que aparentemente conhece-se muito menos sobre teste de software do que sobre outros aspectos e/ou atividades do desenvolvimento de software [MYE79].

O desenvolvimento de ferramentas para suporte à atividade de teste, é fundamental. Estas ferramentas propiciam maior qualidade e produtividade para a fase de testes, uma vez que essa atividade é muito propensa a erros, além de improdutiva se aplicada manualmente. Pressman [PRE00] diz que o uso atual de ferramentas automatizadas para teste de software está crescendo, e que as ferramentas de teste descendentes da primeira geração, descritas brevemente em seu livro, provocarão mudanças radicais na maneira como testamos software e, por conseguinte, melhorarão a confiabilidade dos sistemas baseados em computador.

Estas ferramentas são desenvolvidas baseadas em técnicas e critérios de teste de software, que são consideradas complementares porque acabam revelando diferentes tipos de erros. Por este motivo, a comparação entre as técnicas e os critérios de teste, tanto teórica quanto empiricamente, é de fundamental importância para se obter estratégias de aplicação desses critérios. Abaixo são citadas as principais técnicas de teste existentes:

- Técnica funcional (utiliza a especificação do software realizada na fase de análise): é do tipo caixa preta, por tratar o software como uma caixa cujo conteúdo é desconhecido e da qual só é possível visualizar o lado externo (os dados de entrada e as saídas destes dados). Dois passos são essenciais neste tipo de técnica: identificar as funções que o software deve realizar e criar os casos de teste capazes de checar se estas funções estão sendo realizadas. Um dos problemas relacionados com este tipo de técnica é que a especificação do programa é realizada de modo descritivo e não formal. Assim, os requisitos do teste derivados destas especificações são na maioria dos casos, também imprecisos e não formais.

- Técnica estrutural (utilização do código fonte para derivar os casos de teste): também conhecida como caixa branca, onde os aspectos de implementação são fundamentais na escolha dos casos de teste, pois baseia-se no conhecimento da estrutura interna do programa. Utiliza-se do grafo do programa ou grafo de fluxo de controle.

- Técnica baseada em erros (utiliza defeitos específicos comuns em linguagens de programação): A ênfase desta técnica está nos erros que o programador pode cometer durante o desenvolvimento. Entre as abordagens que podem ser utilizadas para detectar a ocorrência de erros encontram-se a Análise de Mutantes [DEM78] e a Semeadura de Erros [BUD81].

As técnicas descritas acima geralmente são associadas a um critério de teste, que são predicados utilizados para se considerar quando a atividade de teste deve ser encerrada, isto é, para considerar que um determinado programa já foi suficientemente testado e ajudar o testador na tarefa de selecionar os casos de teste.

Existem vários critérios de teste disponíveis na literatura que consideram diferentes aspectos dos programas ou da especificação para selecionar o conjunto de testes; entre estes critérios são de especial interesse nesse trabalho os critérios estruturais. Geralmente eles requerem a execução de caminhos no programa que devem exercitar elementos do código-fonte ou do grafo de fluxo de controle do programa. Eles podem ser:

- Critérios baseados no fluxo de controle [MAL91], [RAP85]: utilizam características de controle de execução do programa, como nós, arcos e caminhos do grafo de fluxo de controle; dentre estes podemos citar os critérios Todos-nós, Todos-Arcos e Todos-caminhos
- Critérios baseados no fluxo de dados [MAL91], [RAP85]: exploram a interação que envolve as definições e usos das variáveis no programas.
- Critérios baseados em predicados: têm o objetivo de descobrir erros nos predicados dos programas ou na especificação. Os critérios BOR (Boolean Operator Testing) e BRO (Boolean and Relational Operator Testing)[TAI93] são exemplos deste tipo de critério. Eles visam a execução de certos tipos de teste para cada predicado (ou condição) do programa, satisfazendo assim um conjunto mínimo de restrições que é criado a partir da análise dos operadores lógicos e booleanos do programa a ser testado.

Com a crescente complexidade do software, qualquer estratégia de teste sem o suporte de ferramentas, tende a ser trabalhosa e muito propensa a erros. Para demonstrar a importância, observe o trecho abaixo, extraído de Horgan [HOR92]: "*Com a crescente complexidade do software, qualquer estratégia de teste sem suporte de ferramentas tende a ser trabalhosa e propensa a erros. Para a aplicação efetiva de um critério de teste faz-se necessário o uso de ferramentas automatizadas que apóiem tal critério pois, do contrário, sua aplicação será limitada a programas muito simples.*" As ferramentas de teste visam reduzir a intervenção humana durante a atividade de teste, aumentando desta forma a qualidade e a produtividade dessa atividade, e influenciando de maneira direta na confiabilidade do software testado [MAL98].

Entre as principais ferramentas destacam-se a ferramenta Asset e Atac. A Asset (A System to Select and Evaluate Tests) [FRA85], foi desenvolvida na New York

University, para o teste de programas em Pascal. Ela utiliza os critérios de adequação baseados na análise de fluxo de dados definidos por Rapps e Weyuker [RAP82], [RAP85]. A Atac (Automatic Test Analysis for C) [HOR90] foi desenvolvida na Bell Comunications Research. Ela avalia a adequação de um conjunto de casos de teste a partir dos critérios Cobertura de Blocos, Decisões, Definições, P-Usos, C-Usos, e Todos-Usos, os quais baseiam-se nos critérios definidos por Rapps e Weyuker. A ferramenta Poke-Tool [CHA91] apóia a utilização da família de critérios Todos Potenciais-Usos, critérios estruturais baseados em fluxo de dados, propostos por Maldonado [MAL91], além dos critérios baseados em fluxo de controle Todos-Arcos e Todos-Nós, para programas escritos em diversas linguagens. A ferramenta Proteum [DEL97], apóia a utilização do critério Análise de Mutantes para programas escritos em linguagem C.

Para apoiar a utilização do critério BRO foi desenvolvida na Universidade da Carolina do Norte a ferramenta BGG [TAI93], que apóia o teste de programas escritos em Pascal e é descrita no item 3.6 .

1.1 – MOTIVAÇÃO

Dado o contexto exposto acima, destacam-se os seguintes itens que serviram de motivação para o trabalho aqui descrito.

1. A importância que o teste de software possui dentro da engenharia de software;
2. A importância de reduzir custos dessa atividade e aplicação de critérios de teste;
3. Existem diferentes critérios, mas eles são considerados complementares porque revelam diferentes tipos de erros, por isso é fundamental a comparação teórica e empírica entre critérios, a fim de se obter estratégias de aplicação dos mesmos;
4. O uso de ferramentas que apóiem a utilização desses critérios e que permitam a realização de experimentos é fundamental;

5. Os critérios BOR e BRO baseados em predicados têm o objetivo de revelar defeitos em predicados;
6. Na literatura existem poucas ferramentas para automatizar esse critério, e a única encontrada apenas realiza o teste de programas em Pascal [TAI93], por isso mesmo é que talvez existam poucos trabalhos comparando BOR e BRO com outros critérios estruturais, principalmente com critérios baseados em fluxo de dados, pois as principais ferramentas que apóiam a aplicação desses critérios são para programas escritos em C.

1.2 – OBJETIVOS DO TRABALHO

Este trabalho tem como foco principal o teste baseado em predicados, mais particularmente os critérios BOR e BRO. O trabalho tem como objetivo a implementação de uma ferramenta de suporte a esses critérios, a PredTool, que permite a geração dos elementos requeridos e a análise da adequação de um conjunto de casos de teste em relação a esses critérios. Essa análise tem como resultado uma cobertura, que poderá ser utilizada para decidir quando encerrar a atividade de teste.

A PredTOOL permite o teste de programas escritos em linguagem C e com ela foi possível a condução de um experimento com os critérios BOR e BRO, comparando-os com outros critérios estruturais. Os resultados desse experimento foram utilizados para propor uma estratégia de aplicação dos critérios analisados.

1.3 – ORGANIZAÇÃO DA DISSERTAÇÃO

Esta dissertação está dividida em sete capítulos. Neste primeiro capítulo foi introduzido o assunto tema desta dissertação e foram abordados a motivação e os objetivos do trabalho realizado.

O Capítulo 2 apresenta um resumo sobre o teste de software, sua importância, terminologia utilizada, as principais técnicas e critérios de teste e algumas ferramentas que automatizam esta tarefa.

O Capítulo 3 contém a uma descrição dos critérios BOR e BRO, baseados em predicados e que são o foco desse trabalho.

O Capítulo 4 apresenta aspectos funcionais e de implementação da ferramenta PredTool.

O Capítulo 5 apresenta um exemplo completo de funcionamento e uso da PredTool.

O Capítulo 6 apresenta a descrição do experimento conduzido com a PredTOOL e a comparação realizada com os critérios BOR e BRO e dois outros critérios estruturais, implementados pela ferramenta POKE-TOOL.

O Capítulo 7 contém as conclusões e desdobramentos desse trabalho.

O trabalho contém um apêndice (Apêndice A) que apresenta o código fonte dos programas utilizados no experimento descrito no Capítulo 6.

2 –TESTE DE SOFTWARE

Os termos *erro, falha, defeito* e *engano* são bastante conhecidos da teoria de testes, mas eles são utilizados de forma muito variada na literatura. Estes termos são definidos a seguir, segundo a classificação do IEEE, que tem realizado vários esforços de padronização, inclusive da terminologia utilizada no contexto de Engenharia de Software. O padrão IEEE número 610.12-1990 [IEE90] diferencia estes termos da seguinte maneira:

– *Defeito* (fault, bug) é um passo, processo ou definição de dados incorreto (exemplo: uma instrução ou comando incorreto);

– *Engano* (mistake) é uma ação humana que produz um resultado incorreto (exemplo: ação incorreta tomada pelo programador);

– *Erro* (error) é uma diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa constitui um erro; e,

– *Falha* (failure) é uma produção de uma saída incorreta em relação à especificação.

Nesta dissertação, quando citados no âmbito do teste de software, os termos engano, defeito e erro serão referenciados como defeito (causa), e o termo falha (conseqüência) como um comportamento incorreto do programa.

Geralmente, os erros são classificados em: erros computacionais e erros de domínio. O primeiro caso provoca uma computação incorreta, mas o caminho executado (seqüência de comandos) é igual ao caminho esperado. Já no segundo

caso, o caminho efetivamente executado é diferente do caminho esperado, ou seja, um caminho errado é selecionado.

O teste serve como uma revisão da especificação, projeto e codificação. Os testes também provam que as funções de um software trabalham como desejado, e analisam a conformidade do produto de acordo com os requisitos iniciais. Glen Myers [MYE79] estabelece três regras que podem ilustrar os objetivos de teste:

- A atividade de teste é o processo de executar um programa com a intenção de descobrir um erro.
- Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um erro ainda não descoberto. Um caso de teste é composto de uma entrada para o programa e da respectiva saída esperada.
- Um teste bem sucedido é aquele que revela um erro ainda não descoberto.

Salienta-se que não existe um procedimento de teste de propósito geral que possa ser usado para provar a corretitude de um programa. Apesar de não ser possível, através de testes, provar que um programa está correto, os testes, se conduzidos sistemática e criteriosamente, contribuem para aumentar a confiança de que o software desempenha as funções especificadas e evidenciar algumas características mínimas do ponto de vista da qualidade do produto [MAL98].

Ainda segundo Myers [MYE79], *"Teste é o processo de executar um programa ou sistema com a finalidade de encontrar erros"*; assim, testes que efetivamente encontram erros estão contribuindo para a melhoria da qualidade de serviços. Isto vem sendo perseguido pelas diversas entidades que atendem a um público cada vez mais variado, seja pelo interesse em aumentar os lucros (como empresas privadas) ou pela necessidade de se adequar a um orçamento cada vez mais enxuto (como empresas públicas), ou mesmo acadêmicos interessados em aprimorar seus conhecimentos na atividade de teste. Compreender o que está acontecendo em uma organização específica e em seus projetos de software é crucial para que se possa planejar, controlar e melhorar o desenvolvimento de software [WAN00].

Mas por que testar não é simples? Porque para testar com eficiência, é preciso conhecer os sistemas a fundo e os sistemas não são, em geral, simples nem fáceis de entender. Segundo Vyssotsky [VYS73], *"primeiro e mais importante, se eu não*

sei o que o programa deve fazer, nenhum teste pode me garantir que ele está fazendo aquilo". Além de conhecer o funcionamento, testar um software pode ser uma tarefa difícil porque exige habilidade, conhecimento e experiência para que sejam obtidos bons resultados. Hetzel [HET87] aponta 4 fatores como essenciais para a atividade de teste:

1) a criatividade e a inteligência;

2) o conhecimento funcional;

3) a experiência na realização de testes e

4) a metodologia de teste.

Outro detalhe importante é o custo do teste; Knowles [KNO76] salienta que o custo do teste é significativo tecnologicamente, mas não adiciona nada visível ao produto. Este custo é justificado pela confiança na existência de um nível de aceitação de qualidade e desempenho ou uma indicação de que alguns tipos de defeitos estão ausentes. O rigor e o custo envolvidos nessa atividade dependem do impacto da ocorrência de falhas no sistema [ROC01]. Pressman [PRE00] cita: "*O destaque crescente do software como elemento de sistema e os 'custos' envolvidos associados às falhas são forças propulsoras para uma atividade de teste cuidadosa e bem planejada. Não é incomum que uma organização de software gaste 40% do esforço de projeto total em teste.*"

Analisando a citação de Pressman conclui-se que não é difícil gastar 40% ou até mais, em sistemas dos quais dependem vidas humanas (por exemplo, controle de vôo, monitoração de reatores nucleares). Nestes casos, Pressman diz que o tempo gasto com os testes pode custar de três a cinco vezes mais que todos os outros passos da Engenharia de Software juntos. Devido a esta importância, o teste deve ser feito da maneira correta para que um resultado positivo seja alcançado.

É importante observar também que o efeito total de corrigir defeitos muito tarde no processo, os custos de reparo serão aumentados grandemente. Quanto maior o atraso entre o defeito ser introduzido e detectado, maior será o custo da correção [GRA94].

Estudos e registros históricos apontados por Simões [SIM99], relatam que considerando-se a produtividade de um projeto de sistemas em termos de número

de pontos por função por unidade de tempo, os percentuais de tempo despendidos nas fases de um projeto podem ser, em média, distribuídos conforme a Figura 2.1

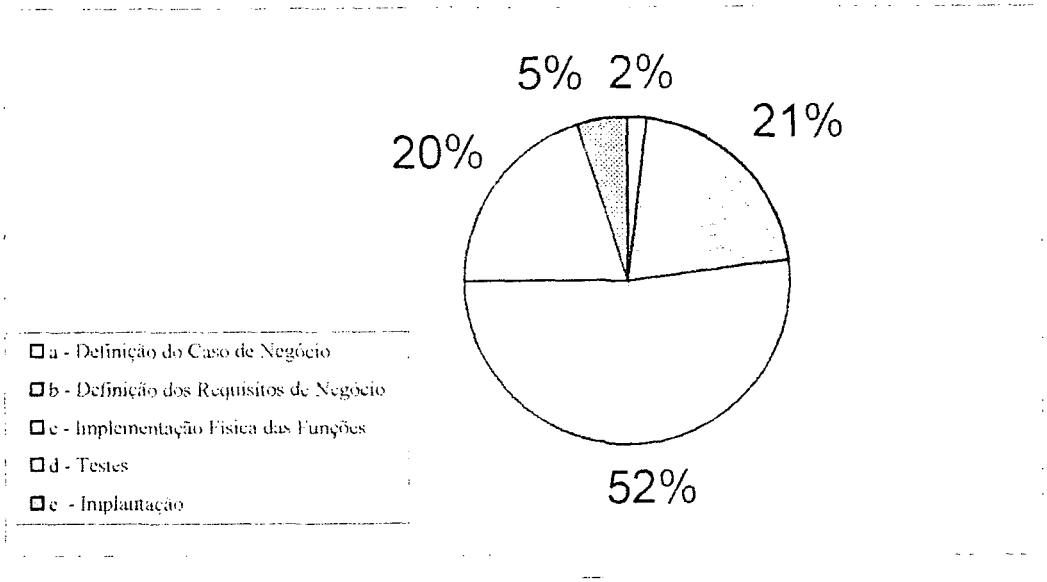

Figura 2.1- Tempo Gasto nas Fases do Projeto.

Observa-se que o tempo de teste em relação ao tempo total do projeto corresponde, em média, a 20%. Aplicando-se esse percentual ao custo total de desenvolvimento de um projeto, desconsiderando-se o custo de correção dos problemas encontrados, nota-se que essa é uma parcela significativa do custo total do projeto.

Além da questão do custo e do tempo que se gasta com testes tem-se uma série de vantagens em se aplicar testes adequadamente e com qualidade:

- reputação de mercado da empresa é afetada, em detrimento das experiências ruins dos clientes com software de má qualidade, com muitos defeitos;

- tempo gasto na correção dos defeitos na manutenção é um tempo não gasto no desenvolvimento de novos produtos;

- é necessário para certificação de gerenciamento de qualidade reconhecidos, (tal como ISO 9001), e realizando o processo de teste repetível, satisfaz-se um dos requisitos para o nível 2 do CMM, e com o processo de teste definido, satisfaz-se o nível 3 do CMM;

- a Legislação faz a empresa se responsabilizar por qualquer efeito da falha de software (Lei 9609/98), independente de a mesma ter sido ocasionada por negligência ou não;

- a credibilidade de uma empresa ou de um produto é construída lentamente e por um longo tempo. Mas uma falha séria no software pode destruir isso rapidamente;

- os efeitos de falhas de software dependem do sistema em que eles ocorrem, mas podem incluir inconveniência, aborrecimento, desinformação, perda de informação, perda de dinheiro, mágoa pessoal e até mesmo morte, no caso de sistemas tipo *safety-critical*.

- utilizando uma metodologia ou um processo de teste que tenha credibilidade garante-se que os componentes de alto risco do sistema são testados;

- evita problemas de omissão, teste inadequado de partes do sistema ou de concessão, testar partes do sistema que não são importantes ou que resultam em testes redundantes;

Duas questões são importantes na atividade de teste e devem ser levadas em consideração quando os testes são realizados: "Como os dados de teste devem ser selecionados?" e "Como decidir se um programa P foi suficientemente testado?". Levando-se em consideração estas questões, têm-se que os critérios para selecionar e avaliar conjuntos de casos de teste são fundamentais para o sucesso da atividade de teste. Estes critérios devem fornecer indicação de quais casos de tese devem ser utilizados para aumentar as chances de revelar erros. Se erros não forem revelados, tais critérios devem, estabelecer um nível de confiança na correção do programa. Uma atividade muito citada na condução e avaliação neste contexto é a *análise de cobertura*, que consiste basicamente em determinar o percentual de elementos requeridos por um dado critério de teste, que foram ou não exercitados pelo conjunto de casos de teste utilizado.

Os principais critérios para geração de dados de teste visam gerar dados que possam detectar a maioria dos defeitos com o mínimo de esforço e tempo. Alguns desses critérios estão descritos nas próximas seções.

2.1 – Critérios de Teste

Geralmente associados a uma técnica de teste estão diferentes critérios de teste. Um critério de teste é um predicado a ser satisfeito para se considerar a atividade de teste encerrada.

2.1.1 – Critérios estruturais

São também conhecidos como critérios de "caixa branca" (Figura 2.2) aplicado à partes pequenas do software, como módulos e por isto são classificados por Hetzel [HET84] como "testing in the small".

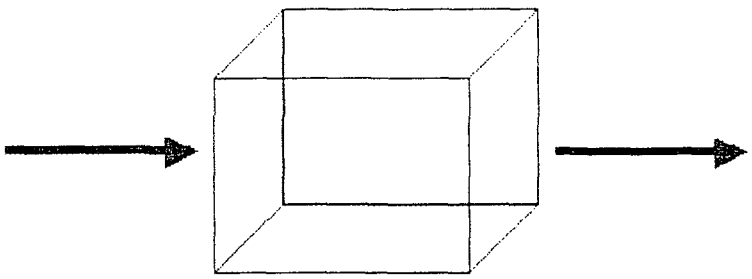

Figura 2.2 - Teste Caixa-branca

Para aplicar um critério estrutural é necessário conhecimento da estrutura interna da implementação dos programas, ou seja: o código fonte do programa. Em geral, a maioria dos critérios dessa técnica utiliza os chamados *grafos de fluxo de controle* (ou *grafo de programa*) como representação dos comandos dos programas. Os critérios estruturais mais conhecidos são:

2.2.1.1 - Critérios baseados em fluxo de controle.

Utiliza apenas características de controle da execução do programa. Os critérios mais conhecidos dessa classe são:

➡ **Todos-Nós** [PRE92] ou **Cobertura de declaração de código** [GRA94] – exige que a execução do programa passe, ao menos uma vez, em cada vértice do grafo

de fluxo, ou seja, que cada comando do programa seja executado pelo menos uma vez. Cobertura de declaração é o número de declarações exercitadas pelo conjunto de teste, divididos pelo número total de declarações no módulo sendo medido. Em um programa de 100 declarações, um conjunto de testes que atualmente usou 75 destas declarações, alcançaria 75% da cobertura.

➡ **Todos-Arcos ou Todos-Ramos** [PRE92] ou **Cobertura de decisão** [GRA94], [PRE92] – requer que cada aresta do grafo, ou seja, cada desvio de fluxo de controle do programa, seja exercitado pelo menos uma vez; Cobertura de decisão ou ramo é similar a cobertura de declaração, mas em vez do número de declarações serem contadas, são contados o número de ramos equivalentes ou decisões. Uma declaração *IF* tem dois ramos, o ramo verdadeiro e o ramo falso. Declarações *CASE* e laços são equivalentes para ramos, desde que eles possam ser expressos como declarações *IF*. Se um programa contém 20 ramos, e um conjunto de teste exercita 10 deles, então 50% de cobertura de ramos foi alcançada.

➡ **Todos-Caminhos** – requer que todos os caminhos possíveis do programa sejam executados [PRE92], [GRA94]. Em qualquer programa que executa um laço um número variável de iterações, há um número infinito de caminhos diferentes através do programa, logo cobertura de todos os caminhos é geralmente impraticável.

➡ **Outros critérios** – Ainda podem ser utilizados outros critérios estruturais, tais como: critério Boundary-Interior [HOW75] e Cobertura LCSAJ (Linear Code Sequence And Jump) [GRA94].

2.2.1.2 - Critérios baseados em fluxo de dados.

Utilizam informações do fluxo de dados do programa para determinar os requisitos de teste, selecionando caminhos de teste de um programa de acordo com as localizações das definições e usos de variáveis no programa. Esses critérios exploram as interações que envolvem definições de variáveis e referências a tais definições para estabelecerem os requisitos de teste [RAP82].

Exemplos dessa classe de critérios são os critérios definidos por Rapps e Weyuker [RAP82], [RAP85], requerem a execução de caminhos a partir da definição da variável até onde ela foi utilizada. Uma definição da variável ocorre quando um valor é armazenado em uma posição de memória. Um uso de uma variável ocorre quando é feita uma referência a essa variável. O uso da variável pode ser computacional (*c-uso*), quando é usada em uma computação, uma soma por exemplo (*lado direito da atribuição*) e *predicativo* (*p-uso*), quando usada em uma condição. Os principais critérios são:

➧ **Todas-Definições** – requer que cada definição de variável seja exercitada pelo menos uma vez, não importa se por um *c-uso* ou por um *p-uso*.

➧ **Todos-Usos** – requer que todas as associações entre uma definição de variável e seus subsequentes usos (*c-usos* e *p-usos*) sejam exercitadas pelos casos de teste, através de pelo menos um caminho livre de definição, ou seja, um caminho onde a variável não é redefinida.

➧ **Todos-Du-Caminhos** – exige que todos os du-caminhos do programa sejam cobertos. Um du-caminho c.r.a x, é dado por uma seqüência de nós $(n_1,...,n_j, n_k)$ onde n1 possui uma definição de x e n_k possui um uso de x e $(n_1,...,n_j, n_k)$ é um caminho livre de definição c.r.a x (ou ainda, $(n_1,...,n_j)$ possui um uso de x em um predicado e $(n_1,...,n_j, n_k)$é um caminho livre de definição c.r.a x e $(n_1,...,n_j)$ é um caminho livre de laços).

Baseados nos critérios de Rapps e Weyuker [RAP82], Maldonado [MAL91] definiu os critérios Potenciais-Usos, introduzindo o conceito de Potencial-Associação. Esses critérios não exigem o uso explícito de uma variável para que uma associação seja estabelecida. *O principal critério dessa família* é o critério:

➧ **Todos-potenciais-usos** [MAL89], [MAL98] – esse critério é satisfeito se, para todo nó *i* e para toda variável x para a qual existe uma definição em *i*, pelo menos um caminho livre de definição c.r.a x do nó i para todo nó e para todo arco possível de ser alcançado a partir de i seja executado.

Outros critérios baseados em fluxo de dados que podem ser utilizados: a família de critérios K-tuplas requeridas de Ntafos [NTA84]; Todos-potenciais-du-caminhos Todos potenciais-usos/du [MAL91], Todos-c-usos e alguns p-usos; Todos p-usos e alguns c-usos; Todos c-usos; Todos p-usos; [RAP82] entre outros.

2.2.1.3 - Critérios baseados na Complexidade.

Utilizam informações sobre a complexidade do programa para derivar os requisitos de teste [PRE00], [ROC01]. Um critério bastante conhecido dessa classe é:

➡ **Critério de McCabe ou Todos Caminhos Linearmente independentes** [MCC76] – utiliza a complexidade ciclomática do grafo de fluxo do programa. A medida de complexidade é utilizada para definir um conjunto de caminhos de execução. Casos de teste devem ser projetados para cada um dos caminhos, garantindo que cada instrução do programa seja executada pelo menos uma vez durante a atividade de teste. Essencialmente, esse critério requer que um conjunto de caminhos linearmente independentes do grafo de programa seja executado, esse teste também conhecido como teste do caminho básico.

O teste estrutural apresenta um problema, que refere-se a impossibilidade de se determinar automaticamente se um dado caminho é ou não executável. Assim, não existe um algoritmo que, dado um caminho completo qualquer, decida se o caminho é executável e forneça o conjunto de valores que causam a execução desse caminho [VER92]. Assim, é preciso a intervenção do testador para determinar quais são os caminhos não executáveis para o programa sendo testado. Um elemento requerido por um dado critério estrutural será não executável se não existir um caminho executável que o cubra. Com esse problema, uma grande quantidade de tempo e esforço são gastos determinando-se elementos não executáveis durante a aplicação dos critérios estruturais.

2.1.2 – Critérios funcionais

São conhecidos como de caixa preta, pois não levam em consideração como o programa foi implementado, ou seja, tratam o software como uma caixa cujo conteúdo e desconhecido e da qual só é possível visualizar o lado externo, ou seja, os dados de entrada fornecidos e as respostas produzidas como saída (Figura 2.3). Os dados de teste são derivados a partir da especificação e dos requisitos de software.

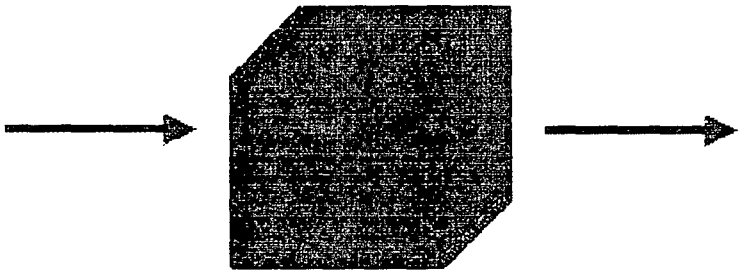

Figura 2.3 - Teste Caixa Preta

O teste caixa preta é classificado por Hetzel [HET84] como "testing in the large". O objetivo deste teste é validar os requisitos funcionais sem se preocupar com o funcionamento interno do programa. Focaliza o domínio da informação tratada pelo software (manipula as entradas e saídas) e procura descobrir erros na interface, de funções incorretas ou ausentes, nas estruturas de dados ou no acesso a bancos de dados externos, de inicialização e término além de erros de desempenho. Alguns exemplo de critérios de teste funcional são [PRE00]:

➡ **Particionamento em classes de equivalência** [PRE00] – utiliza 'classes de equivalência' que representam um conjunto de estados válidos ou inválidos para cada condição de entrada. Um dado de teste em uma mesma classe pode revelar o mesmo tipo de erro. A partir das condições de entrada de dados identificadas na especificação, divide-se o domínio de entrada de um programa em classes de equivalência válidas e inválidas. Em seguida seleciona-se o menor número possível de casos de teste, baseando-se na hipótese de que um elemento de uma dada classe seria representativo da classe toda, sendo que para cada uma das classes inválidas deve ser gerado um caso de teste distinto. Por exemplo, se os números 0 a 99 são válidos, então todos os números deveriam ser manuseados da mesma forma pelo software. Se o programa trabalha corretamente para o valor 7, por exemplo, ele provavelmente trabalhará também corretamente para 2, 35, 50, etc. Todos os valores de 0 a 99 estão na mesma classe de equivalência.

➡ **Análise do valor limite** [PRE00] – é um complemento ao critério particionamento em classes de equivalência, sendo que os limites associados às condições de entrada são exercitados de forma mais rigorosa; ao invés de selecionar-se qualquer elemento de uma classe, os casos de teste são escolhidos nas fronteiras das classes, pois para esses pontos estão associados grande número de defeitos. O espaço de saída do programa também é particionado e são

exigidos casos de teste que produzam resultados nos limites dessas classes de saída. Utiliza os possíveis valores de entrada para cada campo do software, de modo a testar os valores das extremidades (valores máximo e mínimo, máximo + 1 e mínimo - 1, para cada entrada).

➧ **Grafo de causa e efeito** [PRE00], [BEI90] – os critérios anteriores não exploram combinações das condições de entrada, e esta técnica realiza um planejamento de casos de teste que provê uma representação concisa de condições lógicas e ações correspondentes, baseando-se nas possíveis combinações das condições de entrada. Primeiramente, são levantadas as possíveis condições de entrada (causas) e as possíveis ações (efeitos) do programa. A seguir é construído um grafo relacionando as causas e efeitos levantados. Esse grafo é convertido em uma tabela de decisão a partir da qual são derivados os casos de teste.

Um dos problemas relacionados aos critérios funcionais é que muitas vezes a especificação do programa é feita de modo descritivo e não formal. Dessa maneira, os requisitos de teste derivados de tais especificações são também, de certa forma, imprecisos e informais. Como conseqüência, tem-se dificuldade em automatizar a aplicação de tais critérios, que ficam, em geral, restritos à aplicação manual. Por outro lado, para a aplicação desses critérios é essencial apenas que se identifiquem as entradas, a função a ser computada e a saída do programa, o que os tornam aplicáveis praticamente em todas as fases de teste (unidade, integração e sistema) [MAL98].

2.1.3 – Critérios baseados em erros

Utilizam certos tipos de defeitos comuns do processo de desenvolvimento de software para identificar erros e também para derivar requisitos de teste. Os casos de teste gerados são específicos para mostrar a presença ou ausência desses defeitos. A ênfase desses critérios está nos enganos que o programador ou projetista pode cometer durante o desenvolvimento e nas abordagens que podem ser usadas para detectar a sua ocorrência.

Exemplos de critérios baseados em erros:

➡ **Semeadura de Erros** (Error Seeding) [BUD81] [ROC01] – este critério exige que alguns erros sejam incluídos no programa, para que seja verificado durante o teste quais erros foram inseridos (semeados) no programa e quais são naturais. A idéia é verificar a relação entre o número de erros semeados revelados e o número de erros naturais revelados e obter uma indicação do número de erros naturais restantes.

➡ **Análise de mutantes** [DEM78], [VIN97] – O critério Análise de Mutantes utiliza um conjunto de programas ligeiramente modificados (mutantes) obtidos a partir de um determinado programa P para avaliar o quanto um conjunto de casos de teste T é adequado para o teste de P. O objetivo é determinar um conjunto de casos de teste que consiga revelar, através da execução de P, as diferenças de comportamento existentes entre P e seus mutantes.

2.2 – COMPARAÇÕES ENTRE CRITÉRIOS

A relação de inclusão é uma importante propriedade dos critérios, sendo utilizada para realizar uma avaliação teórica do relacionamento dos vários critérios. Esta relação estabelece uma ordem parcial entre os critérios, caracterizando uma hierarquia entre eles. O critério Todos-Ramos, por exemplo, inclui o critério Todos-Nós, ou seja, qualquer conjunto de casos de teste que satisfaz o critério Todos-Ramos também satisfaz o critério Todos-Nós, necessariamente. Quando não é possível estabelecer essa ordem de inclusão para dois critérios, como é o caso de Todas-Defs e Todos-Ramos, diz-se que tais critérios são incomparáveis [MAL98]. A Figura 2.4 mostra a relação de inclusão entre alguns critérios [RAP85], [MAL98],

Outro aspecto interessante de conhecer esta relação de inclusão, é que, para os testes serem mais rigorosos, devem ser utilizados critérios mais exigentes. Outro aspecto é que ao aplicar um critério mais abrangente, pode-se considerar que os critérios menos rigorosos foram também satisfeitos.

A relação de inclusão está relacionada ao "strength", que representa a dificuldade de satisfazer um critério dado que um outro já foi satisfeito. Segundo Wong et al [WON94], além do strength, dois ou fatores são geralmente utilizados para comparar critérios de teste: custo, dado pelo número de casos de teste

necessários para satisfazer um critério, e eficácia, dado pelo número de defeitos revelados.

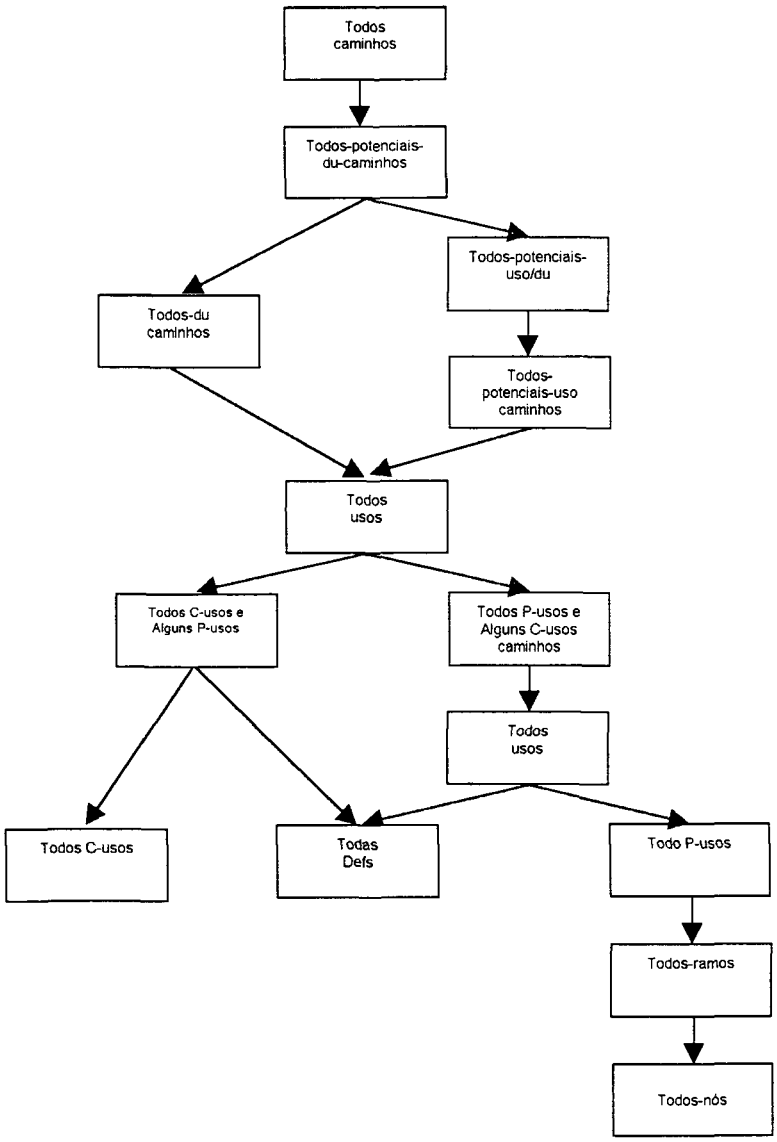

Figura 2.1 - Relação de Inclusão com Critérios Baseados em Fluxo de Dados e Controle

Diversos estudos empíricos têm sido conduzidos considerando os fatores mencionados acima. Segundo Harrold [HAR00], o teste de software pode ser considerada uma atividade que saiu do estado da arte para o estado da prática. Diversos critérios de teste foram propostos e ferramentas que dão apoio a utilização desses critérios foram implementadas, entretanto, é necessário agora encontrar formas econômicas para aplicação desses critérios e o uso dessas ferramentas. Nesse sentido, comparações empíricas entre critérios são fundamentais. Somente elas poderão indicar uma estratégia para a atividade de teste.

Na literatura encontram-se diferentes trabalhos, e entre esses estudos existem os que comparam os critérios estruturais [MAL91], [CHA92], [VER01], [WEI88],

[WEI91]. Alguns deles comparam os critérios estruturais e baseados em erros [SOA02], [WON94]. Tai [TAI93] mostra resultados de comparações empíricas entre os critérios BOR/BRO e os critérios Todos-Arcos. Vouk et al [PAR94] comparam o teste baseado em predicados com a estratégia N-versions, mas os critérios baseados em predicados não foram ainda comparados com os critérios baseados em fluxo de dados. Experimentos com tais critérios são importantes porque eles podem ressaltar o relacionamento empírico entre os critérios baseados em predicados e outros critérios estruturais.

2.3 – FERRAMENTAS DE TESTE DE SOFTWARE

Ferramentas de software ajudam muito na avaliação da cobertura e na execução de vários tipos de testes. Existem muitas ferramentas disponíveis e que atuam nos mais variados níveis de desenvolvimento de software, testando desde o código-fonte até a interface. O foco desse trabalho esta nas ferramentas que permitem a aplicação de critérios de teste. Essas ferramentas oferecem medidas de cobertura e são de especial interesse para a presente dissertação.

2.3.1 – A Ferramenta de Teste POKE-TOOL

A ferramenta POKE-TOOL [MAL98] [CHA91], apóia a aplicação dos critérios estruturais baseados em fluxo de controle Todos-Nós e Todos-Arcos, *assim* como os critérios Potenciais-Usos. A implementação inicial da POKE-TOOL foi desenvolvida em micros compatíveis com IBM/PC. O objetivo principal era a implementação de uma ferramenta específica para o auxílio ao teste aplicando os critérios Potenciais-Usos, permitindo, portanto a avaliação da eficácia desta família de critérios, e verificar sua relação com outros critérios existentes. Na versão inicial da POKE-TOOL, a interação com o usuário se dá através de uma seqüência de telas, formadas por menus. Através da seleção de opções no menu o usuário seleciona programas e visualiza os arquivos gerados.

Uma versão mais atual da POKE-TOOL está disponível para utilização nas estações SUN/UNIX e Linux. Nesta versão, o processo de teste é apoiado por programas distintos, os quais permitem a análise do programa em teste, a execução de casos de teste, a avaliação da cobertura atingida, e a visualização de

um grafo do programa. Tal versão está baseada em linhas de comando e serviu como um modelo para a implementação da ferramenta PredTOOL.

2.3.2 – Outras Ferramentas de Teste

A Proteum (Program Testing Using Mutants) [DEL97] é uma ferramenta de apoio ao critério Análise de Mutantes [VIN97] para programas escritos em C. Esta ferramenta foi desenvolvida pelo Grupo de Engenharia de Software do Instituto de Ciências Matemáticas de São Carlos – USP e está disponível para sistemas operacionais SunOs, Solaris e Linux. Entre os recursos que ela oferece estão a adequação ou geração de casos de teste T para um determinado programa P, e com base nas informações geradas o testador pode melhorar a qualidade de T até obter um conjunto adequando ao critério Análise de Mutantes.

A ferramenta Proteum/IM [MAL98], [VIN97] é uma extensão da Proteum que implementa um conjunto de operadores de mutação distinto e oferece características para testar a conexão entre as unidades de software.

Maldonado [MAL98] descreve outros esforços de ferramentas de teste de software. Entre elas destacam-se a ferramenta Asset e Atac. A Asset (A System to Select and Evaluate Tests) foi desenvolvida na New York University, para o teste de programas em Pascal. Ela utiliza os critérios de adequação baseados na análise de fluxo de dados definidos por Rapps e Weyuker [RAP82], [RAP85]. A Atac (Automatic Test Analysis for C) [HOR90] foi desenvolvida na Bell Comunications Research. Ela avalia a adequação de um conjunto de casos de teste a partir dos critérios Cobertura de Blocos, Decisões, Definições, P-Usos, C-Usos, e Todos-Usos, os quais baseiam-se nos critérios definidos por Rapps e Weyuker.

2.4 – Considerações Finais

Neste capítulo foram introduzidos os principais conceitos e terminologia para a atividade de teste, e descritos os principais critérios de teste existentes. É importante ressaltar a importância de se utilizar um critério de teste, pois, além de fornecer diretrizes para a geração de casos de teste, ele também fornece a

cobertura de um conjunto de casos de teste. A cobertura pode ser utilizada para comparar conjuntos de casos de teste e também para se avaliar a qualidade do teste realizado, fornecendo mecanismos para se considerar a atividade de teste encerrada.

A existência de uma ferramenta de teste é fundamental para a aplicação de um critério, e muito embora implementem técnicas e critérios diferentes, apresentam características globais bastante semelhantes. Assim, pode-se identificar conjuntos básicos de operações que caracterizam atividades pertinentes ao processo de *teste de software* e que foram consideradas para a implementação da ferramenta PredTOOL, descrita no Capítulo 4.

Os critérios baseados em predicados são critérios estruturais de teste, mas por serem o foco do presente trabalho, serão descritos com detalhes no próximo capítulo.

3 - TESTE BASEADO EM PREDICADOS

O teste baseado em predicados consiste na execução de determinados tipos de testes para cada predicado (ou condição) do programa, e várias estratégias de teste existentes são ineficazes e até mesmo impraticáveis para realizar teste de predicados com um ou mais operadores booleanos (chamados de predicados compostos), conforme mostrado por Tai em [TAI93].

Neste mesmo artigo, Tai apresenta dois novos critérios para análise de predicados compostos: BOR (Boolean OperatoR testing) e BRO (Boolean and Relational Operator testing). Demonstra-se ainda que ambas as estratégias são eficazes para detecção de erros em um programa e que, para um predicado com n, $n > 0$, operadores AND/OR, são necessários no máximo $(n + 2)$ e $(2 * n + 3)$ testes para satisfazer respectivamente os critérios BOR e BRO.

Por tratarem predicados compostos, os critérios BOR e BRO são os mais conhecidos critérios baseados em predicados e são o foco desse trabalho. Por isso, apresentam-se, a seguir, os principais conceitos necessários para o entendimento dos critérios BOR e BRO, bem como a técnica utilizada para a satisfação de tais critérios, extraída de [TAI93], [TAI95] e [TAI96].

3.1 - Conceitos Importantes

Predicados (ou condições) em um programa dividem o domínio de entrada deste programa em partições e definem os caminhos deste programa. Seja C um predicado de uma sentença *if* ou *while* em um programa P.

Assume-se que a execução de P com a entrada X atinge C. Se o resultado de C é incorreto devido a erros em C ou na(s) sentença(s) executada(s) antes de alcançar C, então um caminho incorreto de P será executado e provavelmente um resultado incorreto será produzido. A possibilidade de correção coincidental, i.e., uma execução de P com um caminho incorreto produzir um resultado correto, é muito pequena.

Através do teste de C é possível detectar não somente erros em C ou nas instruções executadas antes de atingí-lo, mas também erros nas instruções executadas após C.

Um predicado em um programa ou é um predicado simples ou então é um predicado composto. Um predicado simples é uma variável booleana ou uma expressão relacional, possivelmente com um ou mais operadores de negação. Uma expressão relacional tem a forma:

E1 <rop> E2

onde E1 e E2 são expressões aritméticas e <rop> é um dos 6 operadores relacionais: "<", "≤", "=", "≥", ">", e "≠". Um predicado composto consiste de pelo menos um operador binário booleano, dois ou mais operandos, possivelmente operadores de negação e parênteses.

Os operadores booleanos que podem estar presentes em um predicado são *OU* ("||") e *E* ("&&"), cada um com dois operandos. Um operando simples em um predicado composto refere-se a um operando sem operadores binários booleanos. Um operando composto em um predicado refere-se a um operando com pelo menos um operador binário booleano. Uma expressão booleana é um predicado sem expressões relacionais.

Para formalizar a ocorrência de váriaveis e operadores nas expressões apresentadas nesta dissertação, será utilizada a representação abaixo descrita:

- Bi com i > 0 denota uma variável booleana,
- Ei uma expressão aritmética;
- <rop> ou <rop_i> um operador relacional e

- <bop> ou <bop_i> um operador binário booleano.

Assume-se que um predicado não possui erros sintáticos. Se um predicado estiver incorreto com respeito ao que ele deve fazer, então um ou mais erros dos seguintes tipos ocorrem:

A) Erro no operador booleano (AND/OR incorretos ou operador de negação ausente/extra);

B) Erro na variável booleana (variável incorreta);

C) Erro nos parênteses (localização incorreta de parêntese);

D) Erro no operador relacional (operador relacional incorreto) e

E) Erro na expressão aritmética.

Erros dos tipos A, B e C e suas combinações são chamados de erros na expressão booleana. Erros dos tipos D e E, e suas combinações são chamados de *erros na expressão relacional*.

Um predicado incorreto contém ou um único erro ou múltiplos erros da mesma classe ou de classes diferentes. Diz-se que a existência de um ou mais erros no predicado C é detectada por um teste se a execução de C com este teste produz (executa) um resultado (caminho) inesperado de C. Um conjunto de testes para C garante a detecção de algum tipo de erro em C se a existência de tal erro pode ser detectada por pelo menos um elemento do conjunto de testes, desde que C não contenha erros de outros tipos.

Seja um predicado C* que contém o mesmo conjunto de variáveis de C e não é equivalente a C. Um teste (um conjunto de teste) distingue C de C* se C e C* produzem diferentes resultados sob o teste (em pelo menos um elemento do conjunto de teste). Se C contém erros e C* é a versão correta de C, um conjunto de teste é *insensitivo* a erros em C se este conjunto de teste não pode distinguir C de C*.

3.2 - Restrições Para Predicados

Para um predicado com n, n > 0 operandos simples

$$(<opd1> <bop1> <opd_2> \ldots \ldots <bop_{n-1}> <opd_n>),$$

onde $< opd_i>$, i > 0, denota o i-ésimo operando simples, uma *restrição-BR* (ou somente restrição) é definida como (D1, D2,.... Dn), onde Di, $0 < i \leq n$, é um símbolo especificando uma restrição na variável booleana ou expressão relacional em $<opd_i>$. Observa-se que um operando simples é uma variável booleana ou expressão relacional, possivelmente com um ou mais operadores de negação.

Para uma variável booleana B, os seguintes símbolos são usados para denotar diferentes tipos de restrições no valor de B:

- t: o valor de B é verdadeiro;
- f: o valor de B é falso;
- *: Não há restrição sobre o valor de B;

Para uma expressão relacional E1 <rop> E2, os seguintes símbolos são usados para denotar diferentes tipos de restrições sobre os resultados da expressão relacional:

- t: o valor da expressão relacional é verdadeiro;
- f: o valor da expressão relacional é falso;
- >: o valor de (E1 - E2) é maior do que zero;
- =: o valor de (E1 - E2) é igual a zero;
- <: o valor de (E1 - E2) é menor do que zero;
- $+\varepsilon$: o valor de (E1 - E2) é maior do que zero e menor ou igual a ε;
- $-\varepsilon$: o valor de (E1 - E2) é menor do que zero e maior ou igual a ε;
- *: não há restrição no valor da expressão relacional.

Uma restrição D para um predicado C é *coberta* (ou *satisfeita*) por um teste se durante a execução de C com este teste, o valor de cada variável booleana ou expressão relacional em C satisfaz a restrição correspondente em D. Considere a restrição (=,<) para o predicado ((E1 > E2) | ~ (E3 > E4)), onde ~ é o operador de negação. A cobertura de (=,<) para este predicado requer um teste fazendo E1 = E2 e E3 < E4. A cobertura de (t, +ε) para este predicado requer um teste fazendo E1 > E2 e $0 < E3 - E4 \leq \varepsilon$.

Um conjunto S de restrições para um predicado C é dito ser coberto (ou satisfeito) por um conjunto de teste T se cada restrição em S é satisfeita em C por pelo menos um teste em T. Um teste em T pode cobrir duas ou mais restrições em S.

3.3 - Critérios BOR e BRO

A estratégia de teste do operador booleano (BOR) para uma expressão booleana requer um conjunto de testes para garantir a detecção de erros no operador booleano, incluindo operadores AND/OR incorretos e operadores de negação ausentes/extras.

A estratégia de teste do operador booleano para um predicado é similar, isto é, requer um conjunto de testes para garantir a detecção de erros nos operadores booleanos do predicado.

A estratégia de teste do operador booleano e relacional (BRO) para um predicado requer um conjunto de testes para garantir a detecção de erros nos operadores booleano e relacional, incluindo operadores AND/OR incorretos, operadores de negação ausentes/extras e operadores relacionais incorretos.

Definição: Um conjunto S de restrições-BR para um predicado C é um conjunto de restrições BOR (BRO) para C contanto que, se um conjunto de teste T para C cobre S, então T é um conjunto de teste BOR (BRO) para C.

3.4 - Geração de Conjuntos de Restrições para os Critérios BOR e BRO

Nesta seção, apresentam-se os algoritmos para gerar os conjuntos de restrições BOR e BRO para predicados compostos, extraídos de [TAI93].

Sejam $u = (u_1, ..., u_m)$ e $v = (v_i, ..., v_n)$, onde m, n > O, são duas listas de elementos. A concatenação de u e v, denotada (u, v), é $(u_1, ..., u_m, v_1, ..., v_n)$. Sejam A e B dois conjuntos de listas. A $\cup$ B denota a união de A e B, A * B o produto de A por B, e |A| o tamanho de A. Já A % B é chamada de "onto" de A para B, sendo o conjunto mínimo de (u, v) tal que u|v está em A|B e cada elemento em A|B aparece com, o u|v pelo menos uma vez. Em outras palavras, A % B é um conjunto mínimo de (u,v) tal que u e v estão em A e B respectivamente, cada elemento de A aparece como u pelo menos uma vez, e cada elemento de B aparece como v pelo menos uma vez. |A % B| é o máximo entre |A| e |B|. Se ambos A e B tem mais do que dois elementos, A % B podem formar diversos conjuntos e retorna qualquer um deles. Por exemplo, considere C = {(a), (b)} e D = {(c), (d)}. C % D tem dois valores possíveis: {(a,c),(b,d)} e {(a,d),(b,c)}. Seja E = {(a), (b)} e F = {(c),(d),(e)}, E % F tem 6 valores possíveis: {(a,c),(b,d),(a,e)}, {(a,c),(b,d),(b,e)}, {(a,c),(a,d),(b,e)}, {(b,c),(a,d),(b,e)}, {(b,c),(a,d),(a,e)}, {(b,c),(b,d),(a,e)}.

Seja X uma restrição, que é formada por valores "t", "f", ">", "=" e "<" para um predicado C, o valor produzido por C em qualquer entrada que satisfaça X é o mesmo, e é chamado de C(X) - então uma restrição para C pode ser vista como uma entrada de C. X cobre ou está associado com o ramo verdadeiro ou falso de C se C(X) = verdadeiro ou C(X) = falso. Seja S um conjunto de restrições para C. S pode ser dividido em dois conjuntos: S_t e S_f, onde $S_t(C)$ = {X está em S | C(X) = t} e $S_f(C)$ = {X está em S | C(X) = f}.

Seja w uma entrada que satisfaz X para C. O valor produzido por C com w é C(X). Seja C" um predicado que tem o mesmo conjunto de variáveis de entrada que C. O valor produzido por C" sobre a entrada w pode ser chamado de C"(X) sob uma das suas condições abaixo:

1 - C" difere de C somente nos operadores booleanos e cada restrição em X é ou "t" ou "f", e

2 - C'' difere de C somente nos operadores booleanos e relacionais e cada restrição em X é ou "t" ou "f", para uma variável booleana em C, ou ">", "=", ou "<" para cada expressão relacional em C.

Assume-se que (1) ou (2) são verdadeiros. Um teste que satisfaz X para C(C'') pode distinguir C(C'') de C''(C) se e somente se C(X) $\neq$ C''(X). Assim, X *distingue* C de C'' se C(X) $\neq$ C''(X). Para um conjunto de restrições S para C, C''(S) = C(S) se para cada restrição X em S, C''(X) = C(X); caso contrário, C''(S) $\neq$ C(S) e S *distingue* C de C''.

Para uma variável booleana, seu conjunto de restrições BOR ou BRO é definido como {(t),(f)}. Para uma expressão relacional (E1 <rop> E2), seu conjunto de restrições BOR é definido como {(t),(f)} e seu conjunto de restrições BRO como {(<), (=), (>)}. Sejam C1 e C2 predicados. S1 e S2 são os conjuntos de restrições BOR (BRO) para C1 e C2 respectivamente. A seguir é mostrado como construir os conjuntos de restrições BOR (BRO) para (C1 || C2) e (C1 && C2) usando S1 e S2. Na Tabela 3.1 são mostradas as restrições possíveis para uma expressão relacional *R*. Sejam S1 e S2 conjuntos de restrições BOR (BRO) para C1 e C2, respectivamente, para gerar os conjuntos BOR e BRO aplicam-se as seguintes regras.

Regra 1: Para C = (C1 || C2)

$F(C) = S_{1\_f} \,\%\, S_{2\_f}$

$T(C) = \{S_{1\_t} * \{f2\}\} \;\$\; \{\{f1\} * S_{2\_t}\}$

Onde: f1 $\in S_{1\_f}$ e f2 $\in S_{2\_f}$ e (f1, f2) $\in$ F(C). Assim, T(C) $\cup$ F(C) é um conjunto de restrições BOR(BRO) para C.

Regra 2: Para C = (C1 && C2)

$T(C) = S_{1\_t} \,\%\, S_{2\_t}$

$F(C) = \{S_{1\_f} * \{t2\}\} \$ \{\{t1\} * S_{2\_f}\}$

Onde: t1 $\in S1_{\_t}$ e t2 $\in S2_{\_t}$ e (t1, t2) $\in$ T(C). Assim, T(C) $\cup$ F(C) é um conjunto de restrições BOR(BRO) para C.

Tabela 3.1- Restrições Para a Expressão Relacional *R*

Operador Relacional	Restrições para os predicados	
	$S\_t(R)$	$S\_f(R)$
=	{=}	{<,>}
≠	{<,>}	{=}
>	{>}	{<, =}
<	{<}	{=, >}
≥	{>, =}	{<}
≤	{=, <}	{>}

Exibe-se, na seqüência, a construção do conjunto de restrições BOR para o predicado composto C# definido como (E1 = E2) && (E3 ≠ E4).

Sejam C1 e C2 as expressões (E1 = E2) e (E3 ≠ E4), nesta ordem. Para a construção do teste BOR para C1 e C2, $S1\_t = S2\_t = \{(t)\}$ e $S_{1\_f} = S_{2\_f} = \{(f)\}$. De acordo com a Regra 2, temos que $T(C\#) = S1\_t \,\%\, S2\_t = \{(t,t)\}$. Uma vez que t1 = t2 = t, $F(C\#) = \{S_{1\_f} * \{t2\}\} \cup \{\{t1\} * S_{2\_f}\} = \{(f,t),(t,f)\}$. Assim, {(t,t), (f,t),(t,f)} é um conjunto de restrições BOR para C#. Observa-se que para a expressão (J1 && J2), onde J1 e J2 representam variáveis booleanas distintas, tem o mesmo conjunto de restrições BOR que C#.

Exibe-se também, a construção do conjunto de restrições BRO para C#. Para o teste BRO de C1, S_{1_t} = {(=)} e S_{1_f} = {(>),(<)}. No caso de C2, S_{2_t} = {(>),(<)} e S_{2_f}= {(=)}. Seguindo a Regra 2, temos que $T(C\#) = S_{1\_t} \,\%\, S_{2\_t}$ = {(=)} % {(>),(<)} = {(=,>), (=,<)}. Como T(C#) tem dois elementos, escolhe-se (=, >) como (t1, t2). Deste modo, $F(C\#) = \{S_{1\_f} * \{>\}\} \cup \{\{=\} * S_{2\_f}\} = \{(>,>),(<,>),(=,=)\}$. Desta forma, {(=,>),(=.<),(>,>).(<,>),(=,=)} é um conjunto de restrições BOR para C#.

Para o predicado C@, denotado por ((E1 < E2) && ((E3 > E4) || (E5 = E6))), o conjunto de restrições BRO é construído considerando-se a árvore sintática da Figura 3.1.

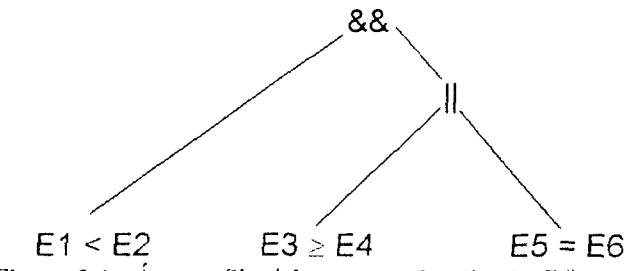

Figure 3.1 - Árvore Sintática para o Predicado C@.

Os seguintes passos são seguidos:

(1) Gerar o conjunto de restrições BRO S1, S2 e S3 para (E1 < E2), (E3 ≥ E4) e (E5 = E6), respectivamente.

(2) Aplicar a regra 1 para S2 e S3 e construir um conjunto de restrições S4 para ((E3 ≥E4) || (E5 = E6)).

(3) Aplicar a regra 2 sobre S1 e S4 e construir o conjunto de restrições S5 para C@.

Na Figura 3.2, o processo de aplicação dos passos descritos acima é exibido com os conjuntos de restrições ao predicado C@.

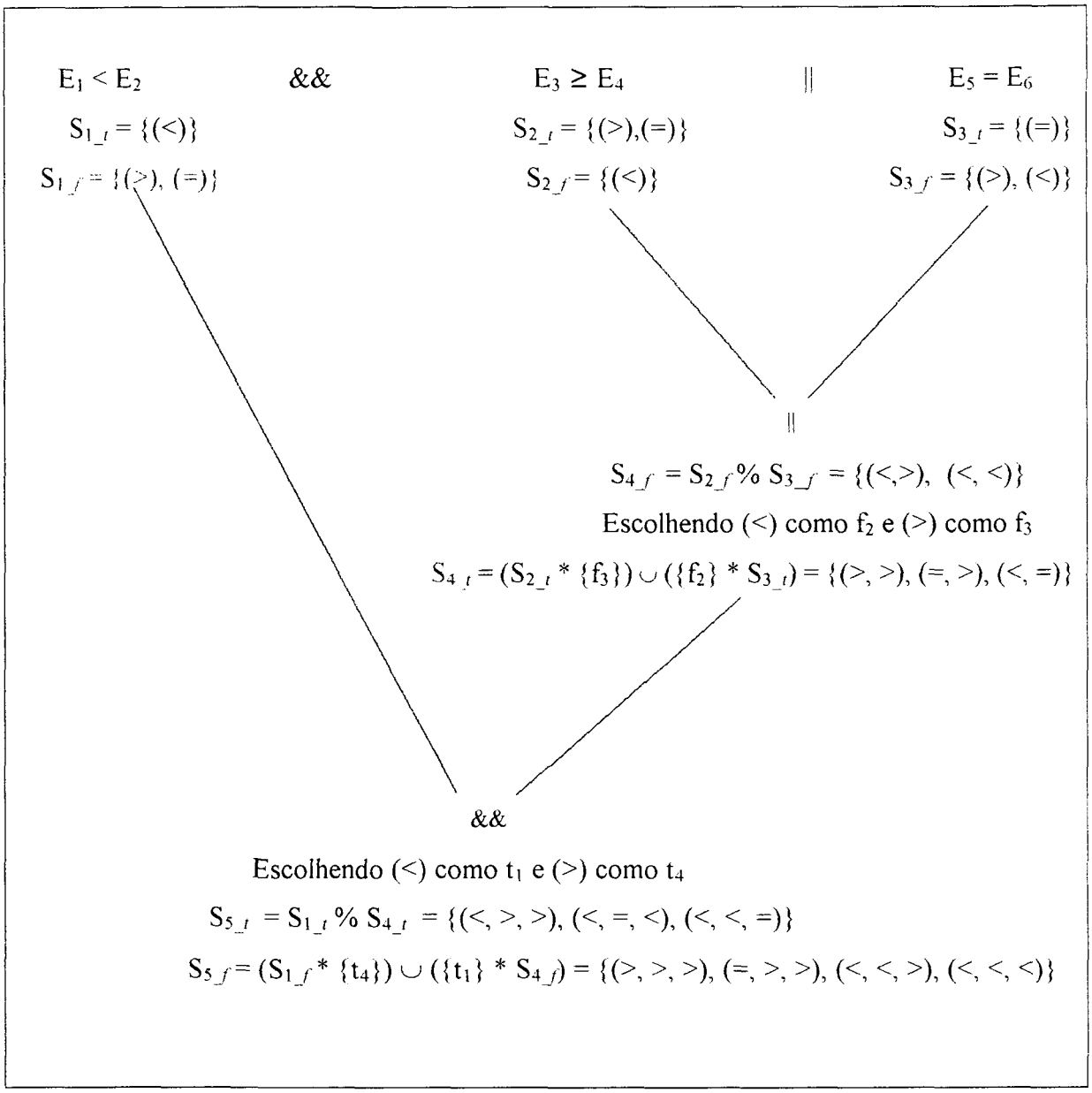

Figura 3.2 - Restrições BRO para o predicado ((E1 < E2) && ((E3 > E4) || (E5 = E6))),

Aplicando a mesma técnica, é possível mostrar que {(t,t,f),(t,f,t),(f,t,f),(t,f,f)} é um conjunto de restrições BOR para C@ e também para (J1 && (J2 || J3)), onde J1, J2 e J3 representam variáveis booleanas distintas. Observa-se isto na Figura 3.3

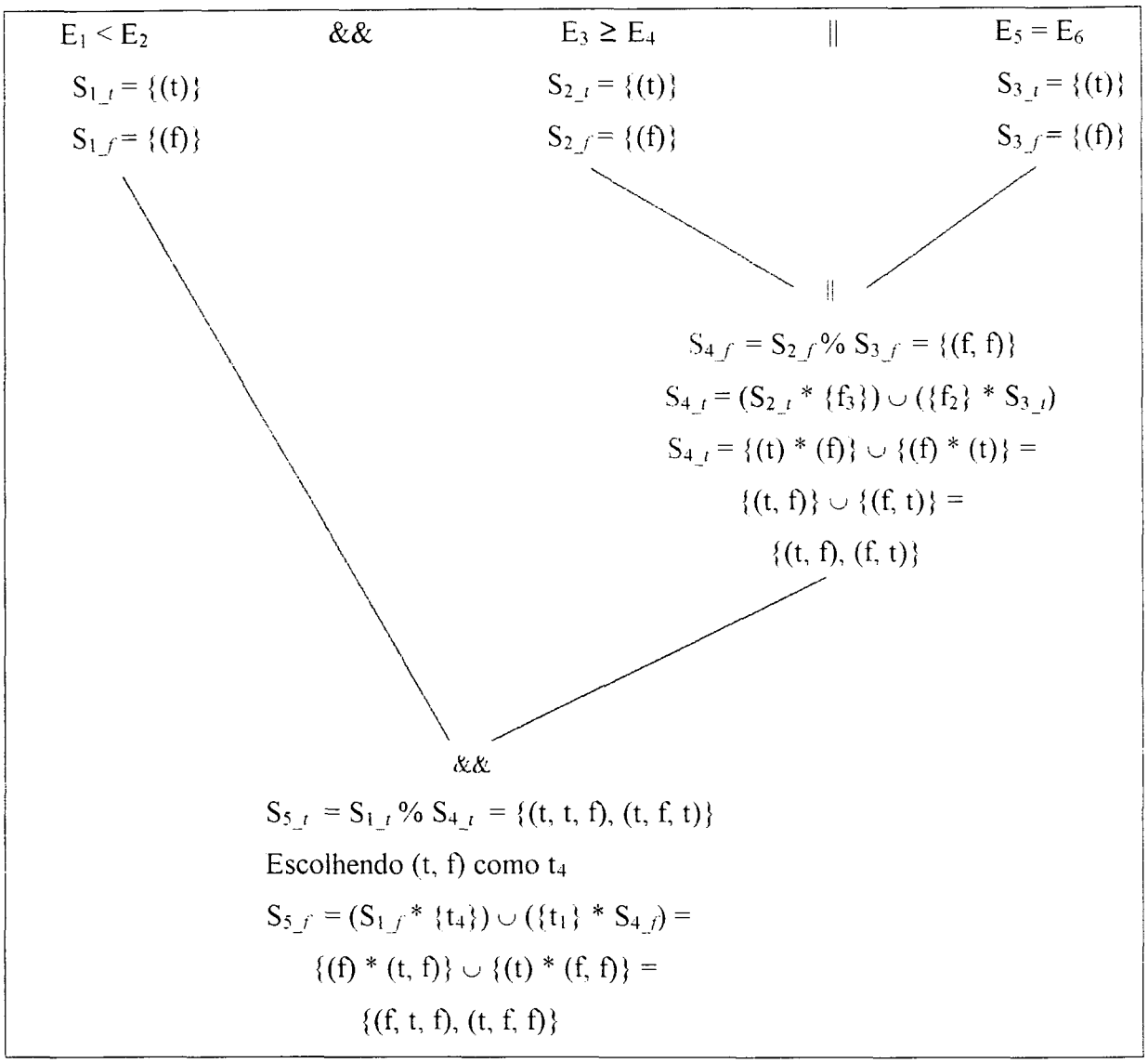

Figura 3.3 - Restrições BOR para o predicado ((E1 < E2) && ((E3 > E4) || (E5 = E6))),

3.5 - Aplicação dos Testes BOR e BRO

Como os predicados aparecem na especificação e na implementação de um programa, os critérios BOR e BRO podem ser usados nos testes baseados em programas e nos testes baseados em especificações. Aplicar um dos critérios a um programa *P* envolve:

(1) A geração do conjunto de restrições BOR (BRO) para cada predicado em *P* e

(2) A geração de testes de *P* para cobrir cada uma destas restrições pelo menos uma vez.

Um caminho de *P* pode ser definido como uma seqüência de ramos em *P*. Seja um caminho - restrição de *P* um caminho de *P* onde cada ramo é substituído por uma restrição. Uma maneira de executar o passo 2 é:

(2.1) - Selecionar um conjunto de caminhos - restrições de *P* para cobrir cada uma das restrições geradas em (1) pelo menos uma vez e

(2.2) - Gerar um teste para cada caminho - restrição selecionada.

Este método consome tempo, pois existem poucas ferramentas para seleção automática de caminhos e testes. Uma alternativa é testar *P* com um conjunto de testes existente *T*, que pode conter testes gerados aleatoriamente a partir do domínio de entrada de *P*. Durante ou após o teste de *P* com *T*, as restrições satisfeitas por *T* são determinadas. Portanto, as restrições não satisfeitas em *P* são usadas para guiar a seleção de caminhos - restrições e a geração de testes adicionais para *P*.

A implementação de programas baseados nos critérios BOR e BRO requer uma ferramenta para medir a cobertura das restrições em um programa. Tal ferramenta é mais complexa do que as que servem para medir a cobertura de ramos em um programa. Uma das razões é o aumento de complexidade durante a execução de um predicado composto: o resultado de cada variável booleana ou expressão relacional precisa ser armazenado.

Uma dificuldade maior no teste de software é que um caminho selecionado de um programa pode ser não executável (ou seja: não existem entradas para executar o caminho). Além disso, o problema de determinar se um caminho é executável é geralmente indecidível. Tais dificuldades também existem para caminhos - restrições. Um caminho - restrição é não executável devido a uma ou mais restrições no caminho.

Uma restrição para o predicado C é não executável para C se ela nunca pode ser satisfeita por um teste para C. Por exemplo, a restrição (t, t) é não executável para o predicado ((E1 > E2) | (E1 = E2)), já que o valor de E1 nunca pode ser ao mesmo tempo maior do que E2 e igual a E2. Se todos os operandos simples em um

predicado C são independentes uns dos outros (ou seja: sem variáveis comuns), então cada restrição para C poderá ser executável.

Uma restrição para um predicado em um programa P é não executável para P se ela nunca pode ser satisfeita por um teste para P (Uma restrição não executável para um predicado é não executável para qualquer programa contendo este predicado). Assuma que P contem a seguinte sentença:

if (X > Y) then if ((X <= Z) || (Z > Y)) then ...;

A restrição (<, <) é executável para ((X <= Z) || (Z > Y)) e sua cobertura requer um teste fazendo X < Z < Y. Contudo, tal teste nunca irá alcançar ((X <= Z) || (Z > Y)), pois ele não pode satisfazer o predicado X > Y. Logo, a restrição (<, <) para ((X <= Z) || (Z > Y)) é não executável para P.

Se alguma restrição gerada para atender aos critérios BOR e BRO for não executável, então é impossível cobrir todas as restrições geradas e não há garantia para a detecção de erros nos operadores booleanos e relacionais em um programa. A existência de restrições não executáveis em um programa não implica necessariamente na existência de erros. Entretanto, a identificação de restrições não executáveis em um programa pode ajudar na detecção e localização dos erros. Atenta-se para o fato de que o problema de determinar se uma restrição para um predicado é não executável é, em geral, indecidível [TAI93]; problema análogo aos elementos não executáveis requeridos pelos outros critérios estruturais apresentados no Capítulo 2.

3.6 - Ferramenta BGG

Uma ferramenta chamada BGG [TAI93] foi inicialmente desenvolvida na Universidade da Carolina do Norte para medir o teste de cobertura de ramos e vários outros tipos de métricas de fluxo de dados de programas em Pascal. A BGG também foi estendida para gerar o conjunto de restrições BRO nos programas em Pascal e avaliar a cobertura desse critério em relação a um conjunto de dados de teste.

3.7 – Considerações Finais

Neste capítulo foram apresentados os conceitos que envolvem o teste baseado em predicados e os critérios BOR e BRO. Uma exemplificação de como obter as restrições para estes critérios e a aplicação dos testes BOR e BRO também foram mostradas. Os critérios BOR e BRO têm como objetivo a detecção de erros em predicados e podem ser aplicados a predicados compostos com maior probabilidade de revelar defeitos. Entretanto, existem poucas ferramentas de aplicação desse critérios, entre essas, destaca-se a BGG, descrita na Seção 3.6, que apóia o teste de programas Pascal. Para permitir a aplicação dos critérios BOR e BRO na linguagem C e permitir a comparação desses critérios com outros critérios estruturais foi implementada a PredTOOL, descrita nos próximos capítulos.

4 – A FERRAMENTA PREDTOOL

Neste capítulo apresentam-se a arquitetura e os principais aspectos funcionais de implementação da ferramenta de teste PredTOOL, que apóia a utilização dos critérios BOR e BRO, descritos no capítulo anterior. O objetivo é apoiar o teste de unidades e programas escritos em linguagem C.

Com o desenvolvimento da ferramenta pretende-se, além de auxiliar o uso prático dos critérios mencionados, viabilizar a realização de comparações entre estes critérios e os demais critérios de teste estrutural.

4.1 - Arquitetura da Ferramenta

Segundo Deutsch [DEU82], existem 5 funções básicas que devem ser realizadas por uma ferramenta de suporte às atividades de teste de programas:

(1) Análise do código fonte e criação de uma base de dados;

(2) Geração de relatórios baseada em análise estática do código fonte, que indique problemas potenciais ou existentes e identifique a estrutura de dados e de controle do software;

(3) Instrumentação do código fonte permitindo a coleta de dados relativos à execução de casos de teste;

(4) Análise dos resultados dos testes e geração de relatórios; e

(5) Geração de relatórios de apoio ao teste para auxiliar na organização das atividades de teste e para derivar conjuntos de entrada (casos de teste) para testes específicos.

Para cumprir com estes objetivos, a ferramenta PredTOOL tem como entrada o programa a ser testado, e um conjunto de casos de teste. Será produzido o conjunto de restrições BOR e um conjunto BRO, para cada predicado do programa e uma relação das restrições não cobertas pelo conjunto de dados fornecido e a cobertura total atingida.

A Figura 4.1 mostra a arquitetura da ferramenta. Composta por módulos que se comunicam através de arquivos. Nela, os retângulos representam os módulos, os losangos representam as entradas fornecidas pelos usuários e os círculos os produtos gerados. A seguir, tem-se uma breve descrição de cada módulo:

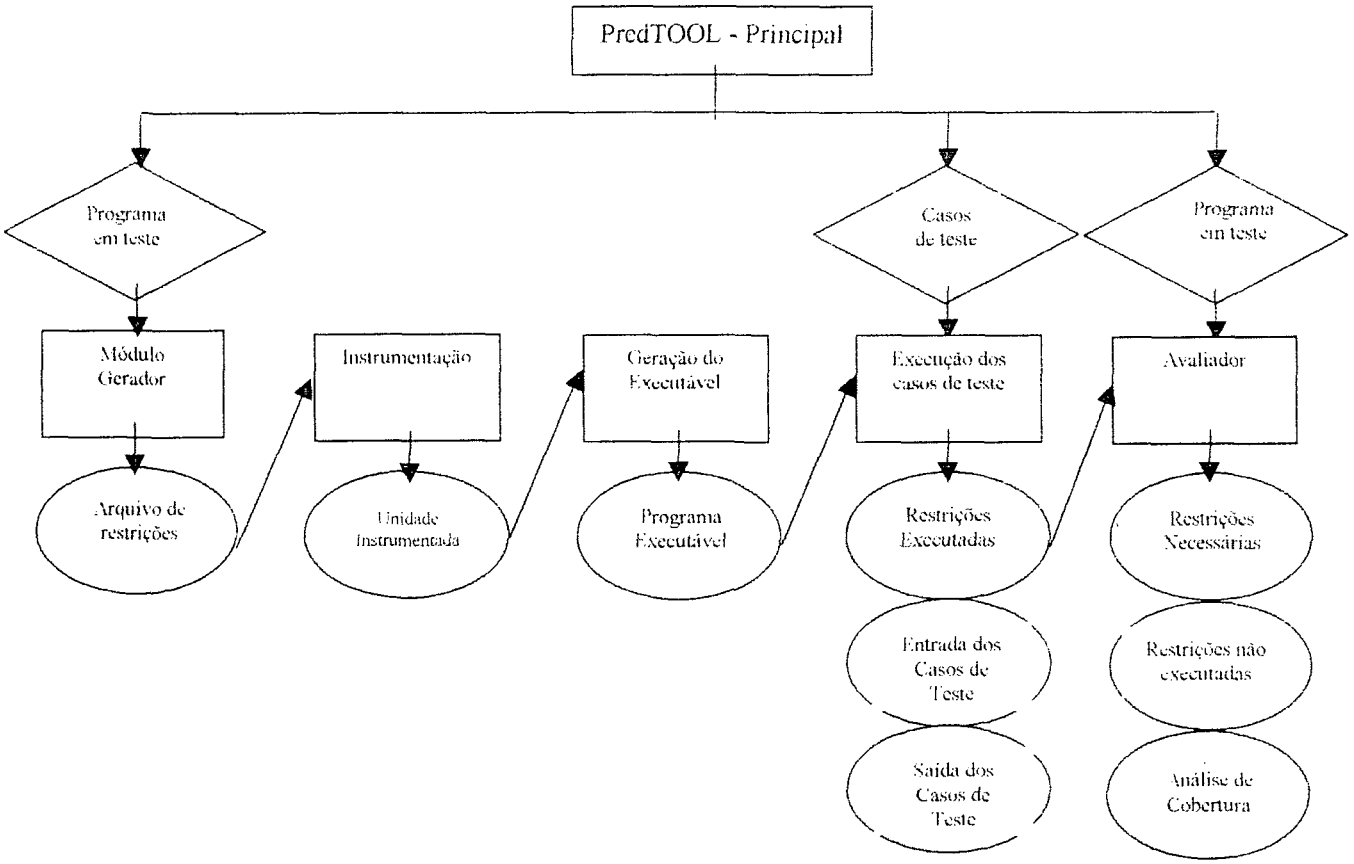

Figura 4.1 – Arquitetura da Ferramenta.

PRINCIPAL – Identificação dos predicados:

Esta função deve identificar os predicados da unidade a ser testada. Isto é realizado utilizando o arquivo .nli gerado pela ferramenta de Teste POKE-TOOL. Trata-se de uma linguagem intermediária que a PredTOOL utiliza para localizar no código-fonte a exata posição onde encontram-se os predicados (condições) do

programa, salvando os mesmos em arquivo. Com estas informações pode-se então passar para o próximo passo, a instrumentação.

INSTRUMENTAÇÃO

A instrumentação insere comando de escrita, chamados de pontas de prova, no programa fonte. Tais comandos basicamente inserem no programa todas as combinações possíveis de restrições aplicadas a cada predicado. O objetivo é gerar uma nova versão do programa - a versão instrumentada - para produzir um "trace" da execução dos casos de teste e obter a informação de quais restrições (*do conjunto total de restrições possíveis*) *foram executadas com os casos de* teste. É importante garantir que o programa instrumentado tenha o mesmo comportamento do programa em teste, ou seja, a semântica do programa em teste não é afetada pela instrumentação.

GERAÇÃO DO EXECUTÁVEL

O programa instrumentado deve então ser compilado pelo testador para gerar o executável da versão instrumentada. Este arquivo será então utilizado para receber como entrada os casos de teste desejados para verificar a cobertura que será atingida.

EXECUÇÃO DOS CASOS DE TESTE

Com o programa executável obtido na etapa anterior, deve-se proceder a execução da unidade em teste, produzindo o conjunto de restrições executadas pelos casos de teste fornecidos. Os casos de teste fornecidos são também armazenados em arquivos.

AVALIADOR

Esta função verifica se o conjunto de restrições executadas pelos casos de teste satisfazendo os critérios BOR/BRO. Em caso negativo, produz uma relação das

restrições requeridas pelo critério e não executadas. Uma medida percentual da cobertura provida pelo conjunto de casos de teste também é gerada, isto é, uma relação entre as restrições executadas e as restrições requeridas é produzida.

Entre as restrições requeridas, podem existir restrições não executáveis. A ferramenta desenvolvida, em sua primeira versão, não fornece nenhum suporte para determinação da executabilidade de uma determinada restrição, por ser este um problema indecidível e de difícil solução. O usuário deverá determiná-la manualmente.

4.2 - Aspectos de Projeto e Implementação

A PredTOOL baseia-se no funcionamento da POKE-TOOL [MAL91], [MAL98], [CHA91] e será orientada para sessão de trabalho. Na sessão de trabalho, o usuário realiza todas as tarefas de teste, a saber: análise estática da unidade, submissão de casos de teste; avaliação dos casos de teste e gerenciamento dos resultados de teste. A execução da ferramenta será dividida em duas fases: uma estática e outra dinâmica.

Na fase estática a ferramenta analisa o código fonte, obtém as informações necessárias para trabalhar com os predicados e instrumenta o código, inserindo pontas de prova no programa em teste e gerando uma nova do mesmo, que viabiliza a posterior avaliação da adequação de um dado conjunto de casos de teste. Terminada esta fase, a ferramenta apresenta, o conjunto de restrições mínimo para atender aos critérios BOR/BRO. Com estas informações o usuário pode projetar seus casos de teste a fim de que eles executem as restrições exigidas.

A fase dinâmica consiste no processo de execução e avaliação de casos de teste. Porém, antes de executar os casos de teste, é necessário que se gere o programa executável a partir da versão instrumentada da unidade a ser testada. Após a execução dos casos de teste, estes serão avaliados. O resultado da avaliação é um conjunto de restrições que restam ser executadas para satisfazer os critérios e o percentual da cobertura do conjunto de casos de teste. Ainda nesta fase, a ferramenta fornecerá o conjunto de restrições executadas e as entradas e saídas

dos casos de teste. O processo de execução/avaliação deve continuar até que todas as restrições tenham sido cobertas ou detectadas como não executáveis, ou ainda que se tenha atingido a cobertura desejada.

A Figura 4.2 apresenta o DFD Nível Zero da ferramenta PredTOOL.

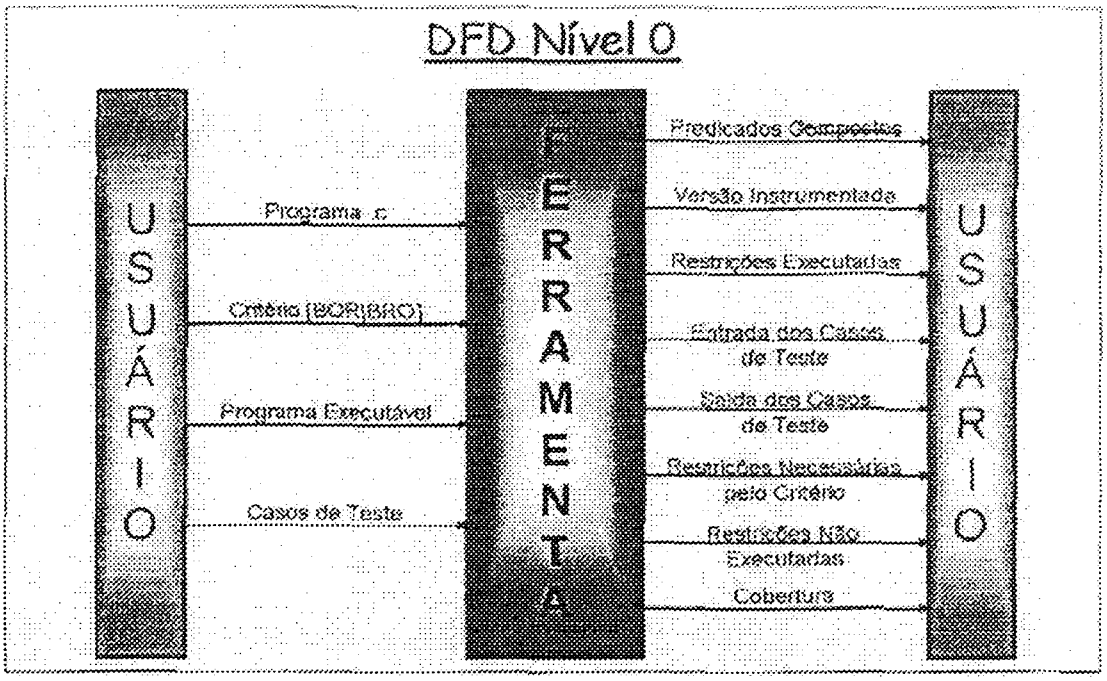

Figura 4.2 - Diagrama de Fluxo de Dados - Nível Macro da Ferramenta

O usuário representa a entidade externa ao sistema, e irá interagir diretamente com a ferramenta, enviando informações necessárias e recebendo um conjunto de relatórios para guiar a atividade de teste. Todos os relatórios serão arquivos gerados pela ferramenta e poderão ser visualizados sempre que o usuário sentir necessidade. Expandindo este diagrama, obtêm-se o nível 1, com informações mais detalhadas (Figura 4.3).

Etapa 1.0 – IDENTIFICAR OS PREDICADOS E GERAR RESTRIÇÕES.

Este processo é a responsável por parte da análise estática do código fonte da unidade. Esta análise é fundamental para a aplicação dos critérios BOR/BRO, pois estes critérios são caixa branca, ou seja, derivam seus requisitos para o teste a partir da estrutura da unidade. Uma análise estática inicial é feita para determinar

todos os predicados do programa. A etapa 1.0 gera um arquivo com os restrições mínimas a serem coberturas, no formato (*nome-do-arquivo*.rbb) e gera também, uma nova versão do programa .c original, com a identificação dos predicados no formato *nome-do-arquivo*_pp.c, para que seja possível gerar uma versão instrumentada.

Para obter o programa .c modificado é preciso fazer análises léxica, sintática e semântica específicas para o código fonte da unidade em teste. Essa análise é facilitada com o uso do módulo .nli da ferramenta POKE-TOOL. Este módulo gera, a partir de um arquivo .c, dois arquivos com extensão .li e .nli. Esses arquivos contêm, respectivamente, para os principais comandos de desvio de fluxo de controle do programa em teste e também para o número do nó do GFC do programa correspondente a esses comandos.

A partir do arquivo .nli são geradas as expressões, contendo os predicados associados a cada nó e que serão utilizados no próximo passo para gerar as restrições que serão armazenadas no arquivo que segue o mesmo formato de nome que o arquivo instrumentado, mas com a extensão .rbb

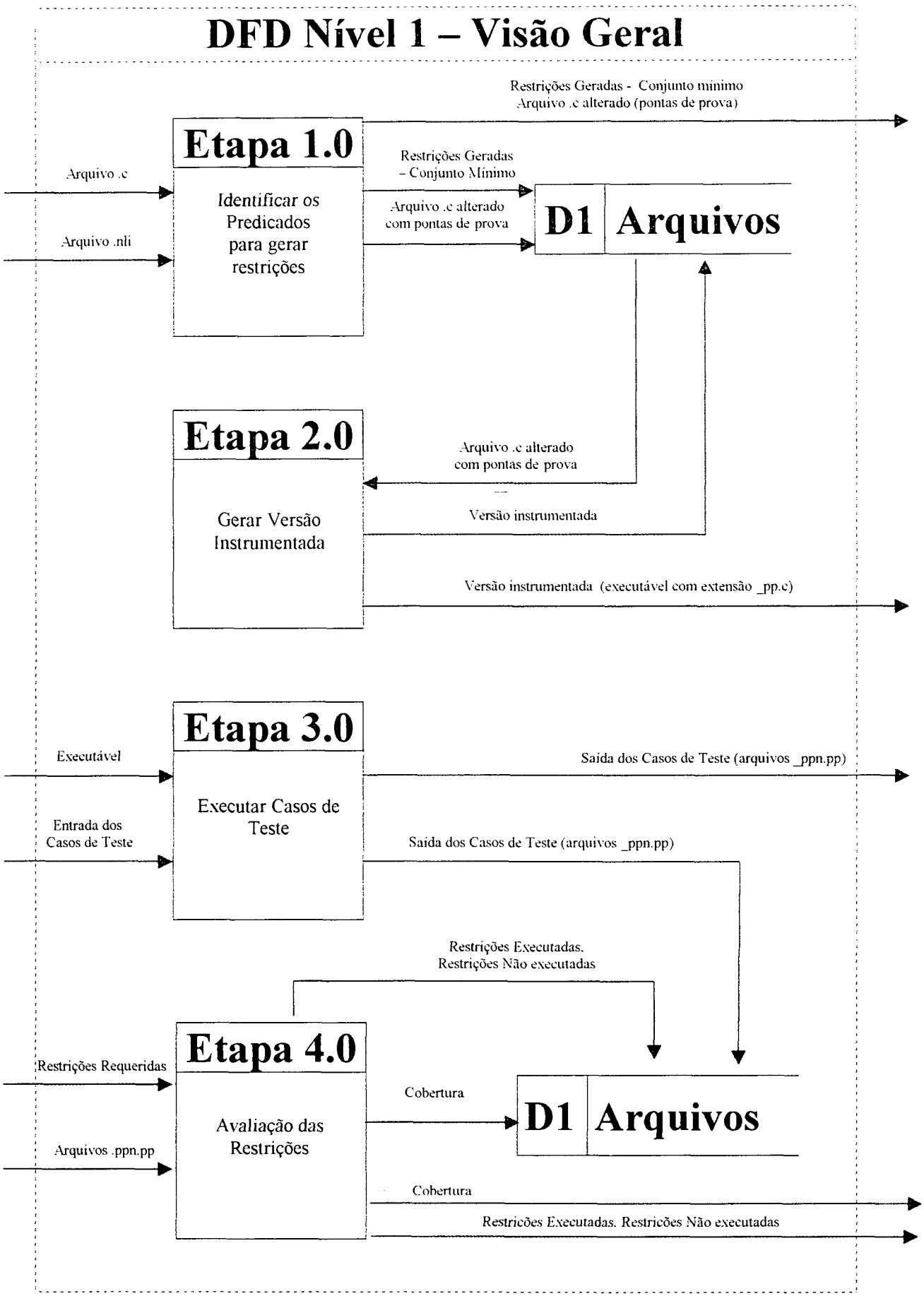

Figura 4.3 - Diagrama de Fluxo de Dados Nível 1

Etapa 2.0 - GERAR VERSÃO INSTRUMENTADA

O processo 2.0 tem como entrada o arquivo com os predicados identificados pelo processo 1.0. O processo de inserir pontas de prova na unidade modificada. gerando uma nova unidade é chamado de *instrumentação*. Esse processo gera então, uma versão da unidade em teste preparada para as atividades de teste e depende essencialmente da identificação dos predicados na etapa 1.0. As pontas de prova — instruções de escrita - geram um "trace" das restrições executadas por um dado caso de teste, viabilizando análises posteriores, tal como a análise da cobertura de um dado conjunto de casos de teste. A unidade instrumentada será armazenada no arquivo *nome-do-arquivo*_pp.c.

ETAPA 3.0 - EXECUTAR CASOS DE TESTE

Este processo requer que o arquivo *nome-do-arquivo*_pp.c, gerado no processo anterior, seja compilado, de modo a gerar um arquivo .exe, que deverá ser executado pelo usuário.. Este processo permite que o usuário informe os casos de teste que deseja executar. No final da entrada de cada caso de teste, é gerado um arquivo que contém a ponto de prova inserida pelo caso de teste.

ETAPA 4.0 - AVALIAR RESTRIÇÕES

Nesta etapa é realizada a análise da adequação de um dado conjunto de casos de teste em relação aos critérios BOR/BRO, sendo que o avaliador calcula as restrições requeridas para satisfazer os critérios BOR/BRO, as restrições executadas e fornece a cobertura atingida pelos casos de teste.

4.3 – Considerações Finais

Foi apresentada neste capítulo a arquitetura da PredTOOL, e foram descritos os aspectos de implementação dos módulos que integram a ferramenta. O principal objetivo foi o de mostrar uma visão macro da ferramenta e das etapas que devem ser seguidas para obter as restrições requeridas e cobertura dos programas que estão em teste. Um exemplo de execução da PredTOOL é apresentado no próximo capítulo.

5 – UTILIZAÇÃO DA PREDTOOL

Para ilustrar o funcionamento da ferramenta PredTOOL, um exemplo utilizando o programa mostrado na Figura 5.1. é apresentado, ilustrando todos os processos descritos no capítulo anterior, para a geração automatizada das restrições requeridas e apresentação final da cobertura atingida com os casos de teste. Com o objetivo de resumir a apresentação, o programa utilizado contém apenas um predicado composto.

PROGRAMA EXEMPLO.C

Programa Fonte

```
#include <stdio.h>
int a, b, c, x;
main()
{
 printf(" digite c -->");
 scanf("%d", &c);
 printf(" digite x -->");
 scanf("%d", &x);
   a=((2*c)+3);
   b=4*x;
 if((a>=10)&&(b<=20)&&(c<=10))
   printf (" mensagem 1 \n");
 else
   printf (" mensagem 2 \n");
 printf("\nc=%d, x=%d, a=%d, b=%d, \n", c, x, a, b);
       return 0;
}
```

Arquivo .nli gerado pela Poke-Tool

$DCL	0	1	18	1
$DCL	0	21	15	2
@main	1	0	0	0
$DCL	1	38	6	3
{	1	46	1	4
$S01	1	53	24	5
$S02	1	83	16	6
$S03	1	105	24	7
$S04	1	135	16	8
$S05	1	157	12	9
$S06	1	172	6	10
$IF	1	184	2	11
$C(03)01	1	186	27	11
{	2	0	0	0
$S07	2	220	26	12
}	2	0	0	0
$ELSE	3	249	4	13
{	3	0	0	0
$S08	3	260	26	14
}	3	0	0	0
$S09	4	292	55	15
$RETURN	4	351	9	16
}	5	362	1	17

Figura 5.1 – Arquivo Fonte e .nli Utilizados Como Exemplo de Uso

5.1 - Módulo Gerador das Restrições.

Para iniciar o uso da PredTOOL, deve-se especificar na linha de comando de execução da ferramenta, o arquivo para ser realizada a geração das restrições como mostrado na Figura 5.2.

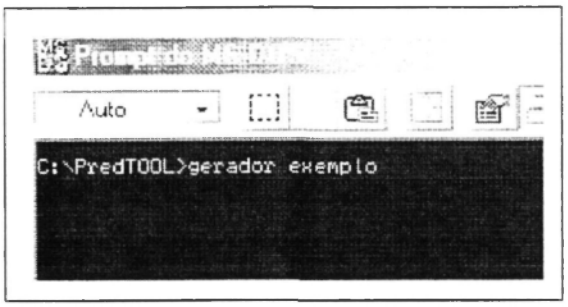

Figura 5.2 – Exemplo de Utilização do Módulo Gerador da PredTOOL.

Deve-se informar o nome do arquivo sem extensão, pois a PredTOOL automaticamente reconhecerá o arquivo fonte e o arquivo .nli para iniciar o processo de criação das restrições requeridas. A Figura 5.3 mostra esta etapa sendo realizada, onde se podem visualizar as restrições requeridas pelo predicado do programa utilizado como exemplo. A execução inicia com a leitura dos parâmetros do arquivo .nli, e baseando-se nos parâmetros contido neste arquivo, o programa lê a expressão no arquivo fonte e repete esta operação até que todas as expressões tenham sido lidas e armazenadas.

Os parâmetros de leitura das expressões do arquivo fonte são obtidos a partir das quatro primeiras colunas do conteúdo do arquivo .nli. Quando o valor da primeira coluna iniciar com $C, trata-se do indicativo de uma expressão correspondente a um conjunto de condições de um predicado. Neste caso, os valores das outras colunas representam respectivamente: o nó no qual se encontra esse determinado predicado, a posição no arquivo fonte onde a expressão inicia e o comprimento da expressão. O valor da quinta coluna não é considerado.

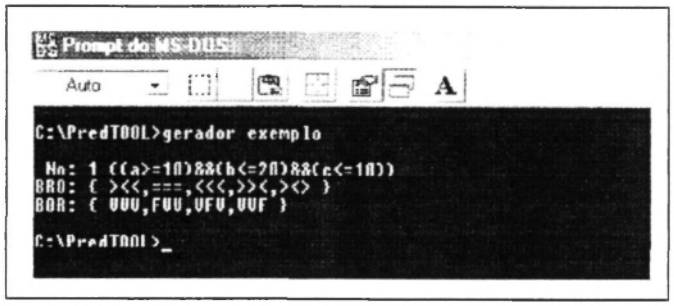

Figura 5.3 – Geração das Restrições Requeridas Pela PredTOOL.

Após a leitura das expressões, inicia-se a geração das restrições BOR/BRO para cada uma das expressões. A ferramenta faz a leitura da primeira expressão, traduzindo-a numa árvore binária gramatical e percorre a árvore fazendo a geração das restrições nó a nó, em pós-ordem. No final da geração, o nó raiz conterá as restrições para toda a expressão. Então, estas restrições são gravadas em arquivo e uma nova expressão é então lida. Esse ciclo termina quando forem geradas restrições para todas as expressões de cada nó existente no programa.

Ao final deste etapa, o arquivo com as restrições requeridas é criado e pode ser visto na Figura 5.4. O nome deste arquivo segue o padrão descrito no capítulo anterior, ou seja: exemplo_pp.rbb. Os valores dos campos de cada linha deste arquivo são separados pelo símbolo ';'.

```
1;((a>=10)&&(b<=20)&&(c<=10));1;1;,1 1 1 ,;1;3;,0 1 1 ,1 0 1 ,1 1 0 ,;1;2;6 7 7 ,10 10 10 ,;1;3;7 7 7
,6 6 7 ,6 7 6 ,;
```

Figura 5.4 –Conteúdo do arquivo de restrições do exemplo utilizado.

O valor do primeiro campo representa o nó nli da expressão; o segundo campo mostra a expressão de onde será gerada as restrições requeridas. O terceiro campo tem a função de indicar a existência ou não das restrições para o critério BOR ou BRO, caso o testador tenha optado por qualquer um dos critérios ou ambos. Na versão apresentada, a PredTOOL sempre gera as restrições para ambos os critérios, desta forma, este campo terá sempre o valor igual a 1. O próximo campo indica a quantidade de restrições existentes para o critério e o último campo mostra as restrições requeridas. Para melhor entender o arquivo gerado, a Tabela 5.1 mostra em detalhes o que cada campo do arquivo representa. Os campos foram separados em linhas diferentes apenas para facilitar a leitura.

Tabela 5.1 – Detalhamento do Arquivo de Restrições Gerado.

Arquivo de Restrições	Significado de cada item
1;	Número do Nó;
((a>=10)&&(b<=20)&&(c<=10));	Expressão (predicado);
1; 1; 1 1 1;	Restrições Bor_t?; Numero de restrições Bor_t; Restrições;
1; 3; 0 1 1, 1 0 1, 1 1 0;	Restrições Bor_f;? Numero de restrições Bor_f; Restrições;
1; 2; 6 7 7, 10 10 10;	Restrições Bro_t?; Numero de restrições Bro_t; Restrições;
1; 3; 7 7 7, 6 6 7, 6 7 6;	Restrições Bro_f?; Numero de restrições Bro_f; Restrições;

Nesta etapa também é gerado o arquivo que contém as pontas de prova, chamado de arquivo instrumentado. O arquivo instrumentado do exemplo utilizado pode ser visto na Figura 5.5.

```
/*************************************************************
* Instrumentador das Restrições dos Critérios Ber e Bor
*************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <io.h>
#include "comum.h"

#define OPER(ival,ival2) ival>ival2  ? G_OP : ival == ival2 ? EQ_OP : ival<ival2 ? L_OP : 0

#define REALIZAOP(ival,ival2,op)   op == G_OP ? ival>ival2 : op == L_OP ? ival<ival2 : op ==
LE_OP ? ival<=ival2 : op==GE_OP ? ival>=ival2 : op==EQ_OP ? ival==ival2 : op == NE_OP ?
ival!=ival2 : op == AND_OP ? ival&&ival2 : op==OR_OP ? ival||ival2 : 0;
#define IMPRIME_NUM_NO(fp,no)  fprintf(fp,"%d;",no);
#define IMPRIME_BRO(fp,op)     fprintf(fp,"%d",op);
#define IMPRIME_BOR(fp,r)      fprintf(fp,",%d ",r);
#define IMPRIME_BOR(fp,r)      fprintf(fp,",%d ",r);
#define IMPRIME_FIM(fp)              fprintf(fp,"\n");

FILE *fppp;
int no_atual=-1;

void AbreArquivo(char *nome)
{

        char nome_arquivo[512];
        struct _finddata_t fileinfo;
        long ret;
        int i;

        for(i=0;i<MAX_CASOS_TESTES;i++)
        {
                sprintf(nome_arquivo,"%s%d.pp",nome,i+1);
                ret=_findfirst(nome_arquivo,&fileinfo);
                _findclose(ret);

                if(ret == -1)
                {
                        fppp=fopen(nome_arquivo,"a+");
                        break;
                }
        }
}
int pp(int n_no,int ival,int ival2, int tipo,int fim)
{
        int r;

        r=REALIZAOP(ival,ival2,tipo);

        if(tipo != AND_OP && tipo != OR_OP)
        {
                if(n_no != no_atual)
                {
                        no_atual=n_no;
                        IMPRIME_NUM_NO(fppp,n_no);
                }

                IMPRIME_BRO(fppp,OPER(ival,ival2));
                IMPRIME_BOR(fppp,r);

        }
        if(fim)
        {
                 IMPRIME_FIM(fppp);
                 no_atual=-1;
        }
```

```
        return c;
}



#include <stdio.h>
int a, b, c, x;
main()
{
        AbreArquivo("exemplo_pp");


    printf(" digite c -->");
    scanf("%d", &c);
    printf(" digite x -->");
    scanf("%d", &x);
        a=((2*c)+3);
        b=4*x;
    if(pp(1,pp(1,pp(1,a,10,9,0),pp(1,b,20,8,1),12,0),pp(1,c,10,8,0),12,1))
            printf (" mensagem 1 \n");
        else
            printf (" mensagem 2 \n");
    printf("\nc=%d,  x=%d, a=%d,   b=%d,  \n", c, x, a, b);

        fclose(fppp);

return 0;
}
```

Figura 5.5 – Arquivo Instrumentado do Programa Exemplo.c

Da mesma forma que ocorre com a nomenclatura do arquivo de restrições, o nome do arquivo que contém as pontas de prova segue o padrão proposto anteriormente, sendo denominado neste caso de exemplo_pp.c O arquivo apresentado na Figura 5.5 deve então ser compilado para gerar o executável que receberá os casos de teste.

Com o programa executável, deve-se então proceder a entrada os casos de teste para tentar cobrir todas as restrições requeridas. Note-se que neste processo podem existir restrições não executáveis, e cabe ao testador conferir esta situação. Por este motivo, a cobertura de 100% das restrições pode não ser alcançada.

Os arquivos gerados também seguem o padrão da nomenclatura adotada em toda a ferramenta, assim, o arquivo exemplo_pp1.pp conterá as pontas de prova do caso de teste1; o arquivo exemplo_pp2.pp conterá as pontas de prova do caso de teste1 e assim por diante, para cada novo caso de teste que for adicionado. Importante ressaltar que a inclusão de novos casos de teste não sobrescrevem casos de teste anteriores, os novos casos são adicionados no conjunto para análise da cobertura.

A Tabela 5.2 mostra os dois casos de teste inseridos no exemplo aqui descrito.

Tabela 5.2 – Detalhamento dos Arquivos de Testes Submetidos no Exemplo.

Número do caso de teste	Nome do caso de teste	Valores digitados para as variáveis		Arquivo Gerado	*Conversão do arquivo gerado nas restrições*	Significado do arquivo gerado
		C	X			
01	exemplo_pp1.pp	5	10	1; 7,1 6,0 6,1	1; <,T >,F >,T	Nó 1; Restrições BRO: >><; Restrições Bor = TFT
02	exemplo_pp2.pp	1	1	1; 7,1 7,1 7,0	1; <,T <,T <,F	Nó ; Restrições BRO: <<<; Restrições Bor = FTT

Os valores do arquivo gerado seguem a seguinte legenda:

0 = F (false);

1 = T (true);

6 = > (maior);

7 = < (menor);

10 = = (igual).

A ordem em que o arquivo é gerado deve ser invertida para que as restrições encontrem-se na ordem correta, isto devido a execução da função pp (ponta de prova) do programa instrumentado, que gera os dados na ordem de precedência das operações. Assim, para a correta geração das restrições elas devem ser invertidas na ordem em que o arquivo é gerado.

Ao se executar o programa instrumentado, os traces correspondentes a cada dado de teste serão gerados e posteriormente utilizado na avaliação.

5.2 - Módulo Avaliador das Restrições.

Esse módulo utiliza-se dos arquivos gerados nas etapas anteriores para realizar a avaliação das restrições e apresentar a cobertura atingida com os casos de teste inseridos.

A Figura 5.6 mostra como deve-se especificar na linha de comando de execução da PredTOOL o comando para realizar a avaliação dos casos de teste.

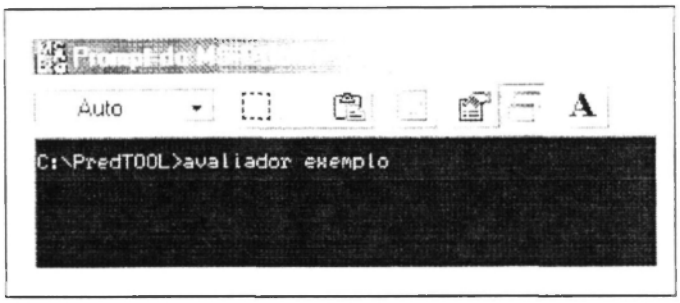

Figura 5.6 – Exemplo de Utilização do Módulo Avaliador da PredTOOL.

Como realizado anteriormente, deve-se informar o nome do arquivo sem extensão; assim a PredTOOL irá automaticamente identificar os arquivos criados nas etapas anteriores e realizar a avaliação dos arquivos gerados. Caso algum arquivo esteja faltando, uma mensagem será disparada avisando o testador. Caso tudo ocorra normalmente a Figura 5.7 mostra o arquivo final onde estão os casos de teste, a cobertura de cada um, as restrições requeridas e executadas, as restrições requeridas e não executadas e a cobertura obtida:

```
Caso de Teste:  exemplo_pp1.pp

No: 1  ((a>=10)&&(b<=20)&&(c<=10))
Total de Cobertura das Restricoes BRO: 20.00%
Total de Cobertura das Restricoes BOR: 25.00%

Caso de Teste:  exemplo_pp2.pp

No: 1  ((a>=10)&&(b<=20)&&(c<=10))
Total de Cobertura das Restricoes BRO: 20.00%
Total de Cobertura das Restricoes BOR: 25.00%

----------------------------------------

Total de Cobertura das Restricoes:
----------------------------------------
No: 1  ((a>=10)&&(b<=20)&&(c<=10))

Total de Cobertura das Restricoes BRO: 40.00%
{ (>><)  (<<<) }
Restrições BRO Não Cobertas: 60.00%
{ (><<)  (===)  (><>) }
Total de Cobertura das Restricoes BOR: 50.00%
{ (TFT)  (FTT) }
Restrições BOR Não Cobertas: 50.00%
{ (TTT)  (TTF) }
```

Figura 5.7 – Arquivo Exemplo_pp.txt com Avaliação Final da PredTOOL do Programa Exemplo.c

5.3 – Considerações Finais

Nesse capítulo foi descrito o funcionamento da ferramenta PredTOOL, que dá apoio ao teste baseado em predicados, especificamente nos critérios BOR/BRO. Para isto, o programa exemplo.c foi utilizado, e as diversas etapas para geração dos arquivos necessários ao funcionamento da ferramenta foram mostradas desde o seu início até a geração do arquivo final com a avaliação das restrições e apresentação da cobertura atingida com os casos de teste inseridos. Note-se que apenas dois casos de teste foram submetidos, apenas para fins de demonstração, então nenhum caminho não executável foi determinado. e nem a cobertura completa foi atingida.

O próximo capítulo apresenta resultados de um experimento de avaliação da PREDTOOL e dos critérios BOR e BRO. Detalhes sobre cobertura, custo e restrições não executáveis serão apresentados.

6 – EXPERIMENTO DE VALIDAÇÃO

Este capítulo apresenta um experimento que compara os critérios baseados em predicado, implementados pela PredTOOL - BOR/BRO - com dois outros critérios implementados pela POKE-TOOL - Todos Potenciais-Usos (PU - critério baseado em fluxo de dados) e Todos-Arcos (ARCS - critério baseado em fluxo de controle). O experimento utilizou 6 programas escritos na linguagem C, alguns deles extraídos de [KER81] e primeiramente utilizados por Weyuker [WEY90. São apresentados os objetivos do experimento, a descrição dos passos adotados e uma análise com a conclusão dos resultados obtidos, através de tabelas comparativas com a avaliação empírica dos critérios testados, culminando com a proposição de uma estratégia para aplicação dos mesmos.

6.1 – Descrição do Experimento

A descrição funcional dos programas escolhidos encontra-se na Tabela 6.1. O Apêndice A apresenta o código-fonte de cada programa.

Tabela 6.1 - Descrição Funcional dos Programas Utilizados no Experimento.

Programa	Descrição do Programa
bbsort.c	Programa que utiliza o método de bolha de ordenação. É solicitado a quantidade de números a serem ordenados, apresentando ao final os números ordenados.
compress.c	Encurta uma string padronizando a repetição de caracteres, substituindo a seqüência de quatro ou mais caracteres iguais por ~Nx. N corresponde à posição da letra no alfabeto para uma repetição de x. Grupos maiores que 26 são quebrados em 2.
entab.c	Substitui strings de brancos por tabs, produzindo a mesma saída visual, mas com menos caracteres.
expand.c	Com a entrada padronizada pelo programa COMPRESS.C na forma *~Nx* descrita acima, este programa retorna a entrada na sua forma normal,
find.c	Permuta os elementos de um array de forma que todos os elementos à esquerda do índice serão menores ou iguais a este e os elementos à direita serão maiores ou iguais ao índice.
getcmd.c	Decodifica o tipo de comando digitado.

O experimento consistiu dos seguintes passos:

1º – Gerar um conjunto inicial de testes T baseado na funcionalidade de cada programa sem entretanto aplicar nenhum critério funcional;

2º – Submeter T na ferramenta de teste POKE-TOOL;

3º – Verificar a cobertura obtida com T para o critério mais fraco, ARCS, e após, para o critério PU. Para ambos os critérios, identificar os itens não executáveis e gerar os casos de teste adicionais para cobrir todas as restrições possíveis. As Tabelas 6.2 e 6.3 mostram para cada programa o número de elementos requeridos, de elementos não executáveis encontrados, a cobertura obtida e o número de casos de teste efetivos, ou seja, o número de casos de testes que realmente contribuíram para o aumento da cobertura. Os conjuntos TARCS e TPU, que correspondem respectivamente aos conjuntos ARCS adequado e PU adequado são compostos somente de casos de teste efetivos.

4º – Submeter o conjunto de testes T na PredTOOL e gerar os casos de testes adicionais para obter os conjunto de TBOR e TBRO, respectivamente BOR e BRO adequados. Os resultados encontram-se nas Tabelas 6.4 e 6.5.

5º – Submeter os conjuntos TARCS e TPU, na PredTOOL, para obter a cobertura obtida para os critérios BOR e BRO. As Tabelas de 6.6, 6.7, 6.8 e 6.9 mostram os resultados obtidos.

6º – Submeter os conjuntos TBOR e TBRO na POKE-TOOL para obter a coberta dos critérios ARCS e PU. As Tabelas 6.10, 6.11, 6.12 e 6.13 mostram os resultados deste passo.

Tabela 6.2 - Resultados da Aplicação de Todos-Arcos.

CRITÉRIO ARCS - RESULTADOS				
Programa	Número de Associações Requeridas	Número de Associações não executáveis	Número de casos de teste efetivos (TPU)	Cobertura total do Critério PU (em %)
bbsort.c	7	0	3	100,0
compress.c	10	0	3	100,0
entab.c	11	0	4	100,0
expand.c	7	0	3	100,0
find.c	13	0	4	100,0
getcmd.c	15	0	15	100,0
T o t a l G e r a l	63	0	32	100,0

Tabela 6.3 - Resultados da aplicação de Todos Potenciais-Usos.

CRITÉRIO PU - RESULTADOS				
Programa	Número de Associações Requeridas	Número de Associações não executáveis	Número de casos de teste efetivos (TPU)	Cobertura total do Critério PU (em %)
bbsort.c	53	1	6	98,2
compress.c	51	3	10	98,1
entab.c	87	13	17	78,5
expand.c	44	14	9	68,1
find.c	175	51	14	83,7
getcmd.c	15	0	15	100,0
T o t a l G e r a l	425	82	71	80,7

Tabela 6.4 - Resultados da Aplicação do Critério BOR.

CRITÉRIO BOR – RESULTADOS				
Programa	**Número de Restrições Requeridas**	**Número de Restrições não executáveis**	**Número de casos de teste efetivos**	**Cobertura total no programa (em %)**
bbsort.c	10	0	2	100,0
compress.c	20	0	3	100,0
entab.c	18	0	4	100,0
expand.c	12	0	4	100,0
find.c	21	0	3	100,0
getcmd.c	30	0	16	100,0
Total Geral	111	0	32	100,0

Tabela 6.5 - Resultados da Aplicação do Critério BRO.

CRITÉRIO BRO – RESULTADOS				
Programa	**Número de Restrições Requeridas**	**Número de Restrições não executáveis**	**Número de casos de teste**	**Cobertura total no programa (em %)**
bbsort.c	15	01	04	93,4
compress.c	30	07	07	72,7
entab.c	27	06	05	77,8
expand.c	18	04	04	77,8
find.c	32	02	05	93,7
getcmd.c	45	01	16	97,8
Total Geral	167	21	41	87,4

Tabela 6.6 - Cobertura de TARCS na PredTOOL. – Critério BOR

CRITÉRIO BOR – RESULTADOS COM TARCS		
Programa	Número de Restrições Cobertas	Cobertura total no programa (em %)
bbsort.c	10	100,0
compress.c	19	95,0
entab.c	17	94,5
expand.c	12	100,0
find.c	21	100,0
getcmd.c	29	96,7
Total Geral	108	97.7

Tabela 6.7 - Cobertura de TARCS na PredTOOL - Critério BRO.

CRITÉRIO BRO – RESULTADOS COM TARCS		
Programa	**Número de Restrições cobertas**	**Cobertura total no programa (em %)**
bbsort.c	10	60,0
compress.c	17	56,7
entab.c	18	66,7
expand.c	13	72,3
find.c	27	84,4
getcmd.c	41	66,7
Total Geral	126	75,4

Tabela 6.8 - Cobertura de TPU na PredTOOL - Critério BOR.

CRITÉRIO BOR – RESULTADOS COM TPU		
Programa	**Número de Restrições cobertas**	**Cobertura total no programa (em %)**
bbsort.c	10	100,0
compress.c	20	100,0
entab.c	17	94,5
expand.c	13	100,0
find.c	21	100,0
getcmd.c	29	96,7
Total Geral	109	98,1

Tabela 6.9 - Cobertura de TPU na PredTOOL - Critério BRO.

CRITÉRIO BRO – RESULTADOS COM TPU		
Programa	**Número de Restrições cobertas**	**Cobertura total no programa (em %)**
bbsort.c	13	100,0
compress.c	17	100,0
entab.c	19	94,5
expand.c	13	100,0
find.c	30	100,0
getcmd.c	41	96,7
Total Geral	133	79,6

Tabela 6.10 - Cobertura de TBOR na POKE-TOOL - Critério ARCS.

CRITÉRIO ARCS– RESULTADOS COM TBOR		
Programa	**Número de Restrições cobertas**	**Cobertura total no programa (em %)**
bbsort.c	7	100,0
compress.c	10	100,0
entab.c	11	100,0
expand.c	7	100,0
find.c	13	100,0
getcmd.c	15	100,0
Total Geral	63	100,0

Tabela 6.11 - Cobertura de TBOR na POKE-TOOL - critério PU.

CRITÉRIO PU – RESULTADOS COM TBOR		
Programa	**Número de Restrições cobertas**	**Cobertura total no programa (em %)**
bbsort.c	28	65,1
compress.c	39	76,4
entab.c	40	45,9
expand.c	14	31,8
find.c	84	48,0
getcmd.c	15	100,0
Total Geral	220	51,7

Tabela 6.12 - Cobertura de TBRO na POKE-TOOL - Critério ARCS.

CRITÉRIO ARCS – RESULTADOS COM TBRO		
Programa	**Número de Restrições cobertas**	**Cobertura total no programa (em %)**
bbsort.c	7	100,0
compress.c	10	100,0
entab.c	11	100,0
expand.c	7	100,0
find.c	13	100,0
getcmd.c	15	100,0
Total Geral	63	100,0

Tabela 6.13 - Cobertura de TBRO na POKE-TOOL - Critério PU.

CRITÉRIO PU – RESULTADOS COM TBRO		
Programa	**Número de Restrições cobertas**	**Cobertura total no programa (em %)**
bbsort.c	31	72,9
compress.c	41	80,4
entab.c	47	54,0
expand.c	26	65,0
find.c	105	60,0
getcmd.c	15	100,0
Total Geral	265	62,3

6.2 – Análise dos Resultados

Nesta seção os resultados obtidos são analisados de acordo com os fatores custo (obtido pelo número necessário de casos de teste para cobrir as restrições) e strength (relacionado com a dificuldade de aplicação).

6.2.1– Custo

A POKE-TOOL não separa as condições nas declarações de fluxo de controle para gerar os arcos requeridos, é suficiente executar as condições *else* e *true* de cada condição. Em alguns casos, quando as expressões não contém predicados compostos, os critérios ARCS e BOR são muito similares. Talvez por causa disto, os critérios requerem exatamente o mesmo número total de casos de teste, 32, muito embora, BOR exija a execução de quase duas vezes mais elementos (restrições). BRO requer 41; estes números de elementos requeridos e casos de teste necessários, não são significativamente maiores do que BOR.

Observou-se que o critério mais exigente é PU. Ele requer 71 casos de teste, mais de duas vezes o número de casos de teste requeridos por ARCS e BOR. Observou-se também que o programa getcmd requer um grande número de casos de teste para os critérios ARCS, BOR e BRO, mas não para PU. Este fato é explicado pela análise dos programas e da complexidade de McCabe [PRE00]. O programa getcmd tem a maior complexidade. ARCS, BOR e BRO são critérios associados com as condições nos comandos de desvio de fluxo de controle, tais como if, while, case, etc. Assim, maior o número de comandos de desvio de fluxo de controle, isto é, a complexidade de McCabe, maior o número de elementos requeridos por estes critérios e conseqüentemente maior o número de casos de teste necessários. Entretanto, PU é um critério baseado em fluxo de dados, então, existem outras características que influenciam o número de elementos requeridos. Talvez, por causa disto, o programa que requer mais casos de teste é o entab. c e o que requer mais associações é o find.c e não o getcmd.c como é para os outros critérios.

Outro ponto a ser considerado é o número de elementos não executáveis. Eles podem aumentar o esforço e o custo de testar porque isto é determinado

manualmente. Os critérios ARCS e BOR não requerem elementos não executáveis. PU e BRO requisitaram a mesma porcentagem de elementos não executáveis, cerca de 20%.

6.2.2 – Strength

Para analisar o strength, utilizam-se as tabelas de 6.6 a 6.13.

Os conjuntos TBOR e TBRO sempre alcançaram 100% de cobertura para o critério ARCS. Este fato mostra que tanto TBOR como TBRO são conjuntos ARCS adequados para todos os programas no experimento. Isto representa que o critério ARCS é o mais fácil de ser satisfeito.

Os conjuntos TARCS obtiveram de coberturas relativamente altas para o critério BOR; somente 03 restrições BOR não foram cobertas, mas o mesmo não ocorreu para o critério BRO; 20 restrições executáveis não foram cobertas (24% delas). Isto mostra que é mais fácil satisfazer o critério BOR do que o BRO.

Com respeito ao TPU, observou-se que as coberturas BOR e BRO não foram 100%. Não existe diferença entre a cobertura dos conjuntos TPU e TARCS para BOR. Mas a cobertura do critério BRO alcançada por TPU, é maior que a cobertura do conjunto TARCS. Pode-se concluir com isto, que é mais fácil satisfazer BRO dado que PU foi satisfeito do que dado que ARCS foi satisfeito.

TBOR e TBRO cobriram, respectivamente, 64,1% e 77,2% das associações executáveis. Nem PU incluiu BOR e BRO como também BOR e BRO não incluíram PU. Eles são incomparáveis. Entretanto, pode-se observar que é mais fácil satisfazer PU dado que BRO foi satisfeito do que dado ARCS ou BOR forem satisfeitos. É mais difícil satisfazer PU do que os outros critérios.

Estes resultados mostram uma relação empírica entre os critérios estudados. Esta relação, baseada no strength, pode ser considerada para propor uma estratégia de aplicação dos critérios, ou seja, para estabelecer uma ordem para aplicá-los:

ARCS → BOR → BRO → PU

6.3 – Considerações Finais

Esse capítulo apresentou um experimento que mostra resultados de uma avaliação empírica de critérios de teste baseados em predicados, envolvendo os critérios BOR e BRO e outros critérios estruturais: Todos-Arcos e Todos Potenciais-Usos. As ferramentas PredTOOL e POKE-TOOL foram utilizadas.

Os resultados foram analisados com relação a dois fatores: custo e strength. Observa-se que o critério PU é o mais custoso e o mais difícil de satisfazer. Os critérios BOR e ARCS são os mais práticos e menos custosos, além de não apresentarem elementos não executáveis.

No entanto, deve-se considerar que esses casos de teste exigidos a mais pelo critério PU podem implicar em um maior número de defeitos revelados. Portanto, a ordem proposta na seção anterior deve considerar também os módulos mais críticos e a confiabilidade requerida para o sistema. O critério PU poderia ser aplicado somente nesses casos.

7 – CONCLUSÕES E TRABALHOS FUTUROS

Nesta dissertação foi apresentada uma ferramenta que apóia a atividade de testes de software, fundamentada em critérios baseados em predicados, BOR/BRO propostos inicialmente por Tai em [TAI93]. Também foram apresentadas algumas ferramentas de teste de software e conceitos pertinentes, bem como um estudo empírico analisando a ferramenta PredTOOL e comparando os critérios implementados com outros critérios estruturais, disponíveis na ferramenta POKE-TOOL.

A implementação de uma nova ferramenta se deve ao fato de não ter sido constatada a existência de trabalhos que implementem os critérios mencionados para a linguagem C. Apenas foi encontrada uma ferramenta que apóia o uso dos critérios BOR e BRO para o teste de programas Pascal.

Dada a dificuldade da aplicação manual de qualquer critério de teste e a ampla utilização da linguagem C, a ferramenta PredTOOL fornece uma opção informatizada dos critérios BOR/BRO em detrimento ao teste manual. Assim sendo, tem-se a expectativa de que a PredTOOL irá contribuir com a atividade de teste, melhorando a confiabilidade das unidades testadas que a utilizarem e, conseqüentemente, reduzindo o grau de defeitos que podem ser encontrados, principalmente os defeitos em predicados. O apoio automatizado contribui para a redução de esforços e custos da atividade de teste.

Outra contribuição da PredTOOL é permitir a realização de experimentos de comparação entre critérios, tais como o experimento descrito no Capítulo 6. Analisando-se os resultados obtidos empiricamente, conclui-se que o critério PU é o critério mais forte; ele requer o maior número de casos de teste e não foi satisfeito por nenhum outro critério. Os elementos não cobertos pelos outros critérios podem implicar em erros revelados, e podem ser explorados em um trabalho futuro.

O relacionamento entre os critérios obtido neste experimento pode ser utilizado para estabelecer uma estratégia de aplicação destes critérios. Por exemplo, poder-se-ia aplicar primeiramente um critério mais fraco para obter o conjunto de teste

inicial T e após isto gerar os casos de teste adicionais para cobrir os outros critérios seguindo a ordem de aplicação apresentada no Capítulo 6. Na maioria dos casos, não se aplicam todos os critérios devido aos altos custos. O critério PU é o mais forte, e muito caro para ser utilizado, poderia ser aplicado somente em módulos críticos do sistema ou em softwares onde a confiabilidade requerida é bastante alta. Esta estratégia tal como aqui descrita pode reduzir os esforços e o custo de teste.

7.1 - Trabalhos Futuros

Abaixo são listados os principais desdobramentos desse trabalho.

Com relação à ferramenta PredTOOL, poderão ser realizados os seguintes trabalhos que permitirão a redução dos custos da ferramenta:

- Implementação de uma interface gráfica para facilitar o uso da ferramenta;
- Implementação de mecanismos para auxiliar a identificação de restrições não executáveis;
- Tratamento de variáveis apontadores e de "aliases" que não são atualmente tratados pela ferramenta;
- Com relação à comparação entre os critérios. Poderão ser conduzidos novos experimentos para:
    - Avaliar o fator eficácia, utilizando-se de programas incorretos;
    - Comparar os critérios BOR e BRO com outros critérios estruturais, tais como os descritos no Capítulo 2;
    - Utilizar um outro conjunto de programas, incluindo programas mais complexos;
    - Comparar os critérios BOR e BRO com o critério Análise de Mutantes, um critério baseado em erros, e implementado pela ferramenta Proteum.

REFERÊNCIAS BIBLIOGRÁFICAS

[BEI90] BEIZER, B. **Software testing techniques**. 2. ed. New York: Van Nostrand Reinhold, 1990.

[BUD81] BUDD, T. A. **Mutation analysis: ideas, examples, problems and prospects, computer program testing**. North-Holand Publishing, 1981.

[CHA91] CHAIM, M. L. **POKE-TOOL - uma ferramenta para suporte ao teste baseados em fluxo de dados.** Tese de Mestrado, DCA/FEEC/Unicamp, Campinas – São Paulo. 1991.

[CHA92] CHAIM, M. L.; MALDONADO, J.C.; VERGILIO, S. R. **Critérios potenciais usos: análise de aplicação de um benchark.** VI Simpósio Brasileiro de Engenharia de Software, pgs 357-371 . Gramado- RS, Brasil, 1992.

[DEL97] DELAMARO, M. E.; MALDONADO, J. C.; **Proteum: A Tool for the Assessment of Test Adequacy for C Programs**, Artigo apresentado no Workshop do Projeto Validação e Teste de Sistemas de Operação, 1997.

[DEM78] DEMILLO, R. A; LIPTON, R. J. **Hints on test data selection: help for practicing programmer**. IEEE Computer, v. 11, n. 4, p. 34-41, 1978.

[DEU79] Deutsch, M., **Verification and Validation** in Software Engineering, Prentice-Hall, 1979.

[DEU82] Deutsch, M.; **Software Verification and Validation**, Prentice-Hall, 1982.

[FRA85] Frankl F. G. Weyuker E. J. **Data flow testing tools**. In Softfair II, pages 46–53, San Francisco, CA, December 1985.

[GRA94] GRAHAM, D. R. Testing. In: **Encyclopedia of software engineering**. J. Wiley,. v. 2, p. 1330-1353. 1994.

[HAR00] HARROLD, Mary Jean. In: **Future of software engineering, 2nd internacional conference on software engineering.** June, 2000.

[HET84] Hetzel, W., **The Complete Guide to Software Testing**, QED Information Sciences, 1984.

[HET87] HETZEL, W. **Guia completo ao teste de software.** Rio de Janeiro. Editora Campus. 1987.

[HOR90] HORGAN, J. R.; LONDON, S. **ATAC- Automatic Test Coverage Analysis for C Programs.** Bellcore Internal Memorandum. June - 1990.

[HOR92] HORGAN, J. R.; MATHUR, A. P.; **Assessing Testing Tools in Research and Education**. IEEE Software, V.9, Nº 6, Dezembro 1992.

[HOW75] HOWDEN, W. E. **Methodology for the generation of program test data**. IEEE Transactions on software engineering, v. 24, n. 5, p. 554-559, May 1975.

[IEE90] IEEE Standard Glossary of Software Engineering Terminology. Padrão 610.12. New York: IEEE CS Press, 1990.

[KER81] KERNINGHAN, B. W.; PLAUGER, E. J. **Software tools in pascal.** Addison-Wesley Publishing Company Reading. Massachusetts, USA.. 1981

[KNO76] KNOWLES, R. **Automatic testing systems and aplication.** Mcgraw Hill. 1976

[MAL89] MALDONADO, J. C.; CHAIN, M. L.; JINO, M. **Arquitetura de uma ferramenta de teste de software de apoio aos critérios potenciais-usos**. In: CONGRESSO NACIONAL DE INFORMÁTICA, 22., 1989, São Paulo. **Anais...** São Paulo: SUCESU, 1989

[MAL91] MALDONADO, J. C.; **Critérios Potenciais-Usos: Uma Contribuição ao Teste Estrutural de Software**; Tese de Doutorado, DCA/FEE/UNICAMP - Campinas, SP, Brasil, 1991.

[MAL98] MALDONADO, J. C.; VINCENZI, A. M. R.; BARBOSA, E. F.; SOUZA, S. R. S. **Aspectos teóricos e empíricos de teste de cobertura de software.** In: ESCOLA DE INFORMÁTICA DA SBC DA REGIÃO SUL, 6., maio 1998. Anais... Blumenau: SBC, 1998.

[MCC76] McCabe, T.. **A Software Complexity Measure**, IEEE Transactions on Software Engineering, V. 2, N. 6, Dezembro, 1976.

[MYE79] MYERS, G.; The Art of Software Testing, Wiley, 1979.

[NTA84] NTAFOS, S.C. **On required element testing**. IEEE Transactions on Software Engineering, v. 10, n. 6, nov. 1984

[OSB79] OSBORNE, A., **Running Wild. The Next Industrial Revolution**. Osborne/McGraw-Hill, 1979.

[PAR94] PARDKAR, A.; TAI, K. C.; VOUK, M. A. **Empirical studies of precate-basead softwae testing**. In IEEE Sump. Software Reliablility. Pages 55-65. 1994.

[PRE00] PRESSMAN, R. S.; **Engenharia de Software**. Tradução de Jose Carlos Barbosa; revisão técnica José Carlos Maldonado, Paulo César Masiero, Rosely Sanches. Editora Makron Books, 1995.

[PRE92] PRESSMAN, R. S. **Software engineering: a practitioner's approach**. New York:McGraw-Hill, 1992.

[RAP82] RAPPS, S.; WEYUKER, E. J.; **Data Flow Analysis Techniques for Test Data Selection**, *Proceedings of International Conference on Software* Engineering, páginas 272-278, 1982.

[RAP85] RAPPS, S.; WEYUKER, E. J.; **Selecting Software Test Data using Data Flow Information,** IEEE Transactions on Software Engineering, SE-11(4), Abril, 1985.

[ROC01] ROCHA, A. R. C.; MALDONADO, J. C.; WEBER, K. C. **Qualidade de software - teoria e prática**. São Paulo: Prentice Hall, 2001

[SAN94] SANDERS, Joc e CURRAN, Eugene. Software Quality. Ed.AddisonWesley,1994

[SIM99] SIMÕES, C. A. **Planejamento, especificação e execução dos testes**. Developers' Cio Magazine**,** Rio de Janeiro, v. 4, n. 40, p. 22-24, dez. 1999.

[SOA02] SOARES, I. W,; VERGILIO, S. R. **Mutation analysis and constraint-basead criteria: results form an empirical evaluation in the context on software testing.** In III Latin American Test Workshop. Uruguai, February 2002.

[TAI93] TAI, K. C.; **Predicate-based test Generation for computer programs.** Proceedings of International Conference on Software Engineering, páginas 267 -276, IEEE Press. 1993.

[TAI95] TAI, K. C.; PARADKAR, A.; **Tesf generation for boolean expressions**. Proceedings of International Symposium on Software Reliability Engineering, páginas 106-115, 1995.

[TAI96] TAI, K. C.; PARADKAR, A.; VOUK, A. M., **Automatic test generation foor predicates**. Proceedings of Internaticonal Symposium con Software Reliability Engineering, páginas 66 - 76, 1996.

[VER01] VERGILIO, S. R.; MALDONADO, J. C.; JINO, M. **Constraint based criteria: An approach for test case selection in the structural test.** Journal of Eletronic Testing. Vol 17(2): 175-183. April 2001.

[VER92] VERGILIO, S. R.; MALDONADO, J. C.; JINO, M. **Caminhos não executáveis na automação das atividades de teste.** In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE. Gramado,. p. 343-356. nov. 1992

[VIN97] VINCENZI, A. M. R.; BARBOSA, E. F.; DELAMARO, M. E.; SOUZA, S. R. S.; MALDONADO, J. C.; **Critério Análise de Mutantes: Estado Atual e Perspectivas**. Artigo apresentado no Workshop do Projeto Validação e Teste de Sistemas de Operação, 1997.

[VYS73] VYSSOTSKY, V. **A. Common sense in desing testable software. Program test methods**. Prentice-Hall, cap.6 p. 41-48. 1973

[WAN00] WANGENHEIM, C. G.; WANGENHEIM, A. **Mensuração no melhoramento da qualidade de software**. Developer's Cio Magazine. Rio de Janeiro, v.5 n. 52, p. 28-32, dez. 2000.

[WEI88] WEYUKER, E. J. **An empirical study of the complexity fo data flow testing**. In Proceedings Of The Second Workshop On Software Testing, Verification E Validation, pages 188–195, Computer Science Press, Banff-Canada, July, 1988.

[WEI90] WEYUKER, E. J. **The cost of data flow testing: an empirical study.** IEEE Transactions On Software Testing, Verification, Validation and Analysis. pages Vol. SE-16(2)121-128, February 1990.

[WEI91] WEYUKER, E. J. WEISS, S. N. HAMLET. R. G. **Comparison of program testing strategies**. In 4th Symposium on Software Testing, Analysis and Verification, pages 154–164, Victoria, British Columbia, Canadá, 1991. ACM Press.

[WON94] WONG, A. P.; MALDONADO, J. C. **Mutation versus all-uses: an empirical evaluation of cost, strength e effectivenes.**. In Software Quality and Productivity – Theory, Practice, Education and Trainning. Hong-Kong, December, 1994.

APÊNDICE A

PROGRAMAS FONTE

Este apêndice mostra o código fonte dos seis programas utilizados no experimento descrito no Capítulo 4.

PROGRAMA BBSORT.C

```
#include <stdio.h>

int A[30], N;

main(){
int j, i;
  printf("Number of elements = ");
  scanf("%d\n",&N);
  for(i=0; i<N; i++){
    scanf("%d", &A[i]);
    printf("\n");
  }
  sort();
  for(i=0; i<N; i++){
    printf("A[%d]= %d\n", i, A[i]);
  }
}


sort()
{
  int pos=0, i1=0, i2, high;
  while (i1<N-1){              /*  Scan array */
    high = A[i1];
    i2 = i1+1;
    pos = i1;
    while (i2<N){          /* Largest */
      if (A[i2]>high){
        high=A[i2];
        pos=i2;
      }
      i2=i2+1;
    }                     /* Exchange */
    i2=A[i1];
    A[i1]=high;
    A[pos]=i2;
    i1=i1+1;
  }
}
```

PROGRAMA COMPRESS.C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void putrep(n,c)

int n;
int c;

/* imprime a representacao de n caracteres c */

{
  int i;

  while((n>=4)||((c=='~')&&(n>0)))
  {
    putchar('~');
    if (n < 26)
       putchar(n-1+'A');
    else
       putchar(26-1+'A');
    putchar(c);
    n = n - 26;
  }
  /* se o numero de repeticoes restantes <4 imprimir o caracter n vezes */
  for (i=n; i>0; i--)
    putchar(c);
}

void compress()

/* Encurta uma string padronizando a repeticao de caracteres,
   substitui uma sequencia de 4 ou + caracteres por ~nx, onde
   n e' a letra A para uma repeticao de x, B para duas, e as-
   sim por diante. Grupos maiores que 26 sao quebrados em 2. */

{
  int n,lastc,c;

  n = 1;
  lastc = getchar();
  while (lastc != EOF)
  {
   c=getchar();
   if (c == EOF)
   {
     if ((n>1)||(lastc=='~'))
       putrep(n,lastc);
     else
       putchar(lastc);
   }
   else
   {
    if (c==lastc)
      n++;
    else if ((n>1)||(lastc=='~'))
         {
           putrep(n,lastc);
           n = 1;
```

```
          }
          else
            putchar(lastc);
    }
    lastc = c;
   }
}

main()
{
  compress();
}
```

PROGRAMA ENTAB.C

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

int tabpos(col, tabstops)
int col;
int *tabstops;

/* Retornar 1 se col e' em tabstop */

{
  if (col>=100)
    return 1;
 else
   return (tabstops[col]);
}

void settabs(tabstops)
int * tabstops;

/* Retornar 1 se i e' um
tabstop, isso e' as colunas 0,4,8...*/

{
  int i;
  for (i=0; i<100; i++)
    if ((i % 4) == 0)
     tabstops[i] = 1;
    else
     tabstops[i] = 0;
}

void entab()

/*  Substitui strings de brancos por tabs. Produz visualmente a
    mesma saida, mas com menos caracteres                        */
{
 int c, col, newcol;
 int tabstops[100];
 settabs(tabstops);
 col = 0;

 do
 {
   newcol = col;
   c = getchar();

   while ( c == ' ')
   {
     newcol++;
     if (tabpos(newcol,tabstops))
       {
         putchar(9);
         col = newcol;
       }

       c = getchar();
   }
```

```
   while (col<newcol)
   {
    putchar(' ');
    col++;
   }
   if (c!=EOF)
   {
     putchar(c);
     if (c=='\n')
       col = 0;
     else
       col++;
   }
 }while (c!=EOF);
}




void main()
{
   entab();
}
```

PROGRAMA EXPAND.C

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

void expand()
/* Obtem de uma entrada padronizada por compress,
   uma entrada normal */

{
 int c,n;

 c=getchar();

 while(c != EOF)
 {
       if(c != '~')
            putchar(c);
       else
       {
            c=getchar();

            if(isupper(c)!= 0)
            {
          n = c - 'A' + 1;

              c = getchar();

          if (c != EOF)
          {
            for (n=n; n>=1; n--)
              putchar(c);
          }
          else
          {
            putchar('~');
            putchar(n-1+'A');
          }

        }
            else
        {
          putchar('~');
          if (c != EOF)
            putchar(c);
         }
       }

       c=getchar();
 }

}

void main()

{
  expand();
}
```

PROGRAMA FIND.C

```
/*-------------------------------------------------
 *
 *    Program Name:       Find ( C version)
 *
 *
 *-------------------------------------------------
 */

int a [256+1];

void find(int n,int f)
{
    int  b;
    int   m, ns, i, j, w;

    b = 0;
    m = 1;
    ns = n;

   while ((m < ns) || b)
   {   if   (!b)
       {  i = m;
          j = ns;
       }
       else
          b = 0;
   /*-------------------------------*/
       if   (i > j)
       {    if   (f > j)
                 {  if  (i > f)
                        m = ns;
                    else
                        m = i;
                 }
            else
                 ns = j;
       }
   /*-------------------------------*/
       else
       {    while  (a[i] < a[f])
                   i = i + 1 ;
            while  (a[f] < a[j])
                   j = j - 1 ;
            if   (i <= j)
            {    w = a[i];
             a[i] = a[j];
                 a[j] = w;
                 i = i + 1;
                 j = j - 1;
            }
            b = 1;
       }
   }
}

/*-------------------------------------------------------------------
 *
 *    Program Name:       main function of Findl ( C version)
 *
 *    Description:
 *
```

```
 *      INPUT:  an array a and an index f
 *
 *      OUTPUT: permutes the elements of a so that all elements
 *              to the left of position f are less than or equal
 *              to a[f] and all elements to the right of position
 *              f are greater than or equal to a[f].
 *
 *      ORIGIN: C. Hoare.  Proof of a program: Find.
 *              Commun. ACM, 14(1):39, 1971
 *
 *      OUR VERSION: This C version is translated from the
 *                   Pascal version used by Phyllis Frankl.
 *
 *   ATTENTION:
 *      The index of a C array ........
 *
 *
 *   BUG:
 *      This program gives an infinite loop if the value of
 *      n leass than the value of f.  This is because the while
 *      loop a[i] < a[f] inside the function "find".
 *      (a[f] is not initialized)
 *
 *
 *      Example:
 *      >>>
 *      >>> cat btest31c.3
 *      >>> 3       [ n = 3 ]
 *      >>> 8       [ f = 8 ]
 *      >>> 100     [ a[1]  ]    [ We skip a[0] in C ]
 *      >>> 30      [ a[2]  ]
 *      >>> 150     [ a[3]  ]
 *      >>>
 *      >>> a.out <btest31c.3
 *          |
 *          |
 *          |------  an infinite loop
 *
 *
 *-------------------------------------------------------------------
 */


/* int      a[max + 1];   */
/*================================================*/
                        /*  Because we skip index = 0, we need to declare
*/
                        /*  "a" as an integer  array with MAX + 1
elements. */
                        /*
*/
                        /*  Ref: p249 C by Dissection
*/

/*================================================*/

main()
{  char  *mystr;
   int  i;
   int N, F;

   char  *gets();
```

```
   void  find();

   mystr = (char *) malloc (80);  /*-------------------------------
                                   * allocate space to mystr
                                   *-------------------------------
                                   */

   scanf("%d", &N);
   gets(mystr);          /*----------------------------------------
                          *  We use gets() to mimic readln() in the
                          *  original Pascal program.
                          *  gets() reads a string into s from stdin
                          *  until a new-line character is read.
                          *----------------------------------------
                          */
   scanf("%d", &F);
   gets(mystr);

   for (i=1;i<=N;i++)
   {
      scanf("%d", &a[i]);
      gets(mystr);
   }


   find(N, F);
   printf("%5d\n", N);
   printf("%5d\n", F);

   for (i=1; i<=N; i++)
      printf("%5d\n", a[i]);
}
```

PROGRAMA GETCMD.C

```
#include <stdio.h>
#include <stdlib.h>
#include "cte.h"

cmdtype getcmd (buf)
char *buf;
/* Decodifica o tipo de comando */
{
  char cmd[3];
  cmd[0] = buf[0];
  cmd[1] = buf[1];
  cmd[2] = '\0';

  if (strcmp(cmd,"fi")==0)
    return(FI);
  else if (strcmp(cmd,"nf")==0)
    return(NF);
  else if (strcmp(cmd,"br")==0)
    return(BR);
  else if (strcmp(cmd,"ls")==0)
    return(LS);
  else if (strcmp(cmd,"bp")==0)
    return(BP);
  else if (strcmp(cmd,"sp")==0)
    return(SP);
  else if (strcmp(cmd,"in")==0)
    return(IND);
  else if (strcmp(cmd,"rm")==0)
    return(RM);
  else if (strcmp(cmd,"ti")==0)
    return(TI);
  else if (strcmp(cmd,"ce")==0)
    return(CE);
  else if (strcmp(cmd,"ul")==0)
    return(UL);
  else if (strcmp(cmd,"he")==0)
    return(HE);
  else if (strcmp(cmd,"fo")==0)
    return(FO);
  else if (strcmp(cmd,"pl")==0)
    return(PL);
  else
    return(UNKNOWN);
}
 main(argc, argv)
int argc;
char *argv[];

{
   if(argc <= 1)
    {
     printf("Erro numero de argumentos\n");
     exit(1);
    }
  printf("%d", getcmd(argv[1]));

}
```

CTE.H

```
typedef enum {BP,BR,CE,FI,FO,HE,IND,LS,NF,PL,RM,SP,TI,UL,UNKNOWN} cmdtype;
```