PredTOOL – Uma Ferramenta para Apoiar o Teste Baseado em Predicados

Edenilson José da Silva
CEFET/PR Unidade do Sudoeste, GETIC
Pato Branco, Brasil, 85502-390
ede@pb.cefetpr.br

e

Silvia Regina Vergilio
UFPR, Departamento de Informática,
Curitiba, Brasil, 81531-970
silvia@inf.ufpr.br

Abstract

The testing activity is a fundamental phase in the Software Engineering process, especially for improving the quality of the developed programs. To reduce the costs and to increase the number of defects revealed in the test, several testing criteria were proposed. These criteria guide the tester in the selection and evaluation of test case sets. This works focuses on structural testing criteria, more particularly BOR (Boolean OperatoR testing) and BRO (Boolean and Relational Operator testing) criteria, that have the goal of revealing faults in compound predicates of the program under testing. A tool that implements the BOR and BRO criteria is described. This tool, named PredTOOL, supports the test of C programs. PredTOOL made possible the accomplishment of experiments with BOR and BRO criteria and the comparison of those criteria with two other structural criteria: All-edges and All Potential-Uses. The obtained results are used to propose a strategy for application of the studied structural criteria.
Keywords: Software Testing, Predicate Based Testing, Structural Testing Criteria.

Resumo

A atividade de teste é fundamental dentro da Engenharia de Software, especialmente para a melhoria da qualidade dos programas criados. Para reduzir os custos e aumentar o número de defeitos revelados no teste, foram propostos diversos critérios. Esses critérios têm como objetivo guiar o testador na seleção e na avaliação de conjuntos de casos de teste. Este trabalho aborda os critérios estruturais de teste, mais particularmente os critérios BOR (Boolean OperatoR testing) e BRO (Boolean and Relational Operator testing), que têm como objetivo revelar defeitos presentes em predicados compostos do programa em teste. Uma ferramenta que automatiza os critérios BOR e BRO é descrita. Essa ferramenta, chamada PredTOOL permite o teste de programas em linguagem C. A utilização da ferramenta tornou possível a realização de um experimento dos critérios BOR e BRO e a comparação desses critérios com dois outros critérios estruturais, Todos-Arcos e Todos Potenciais-Usos. Da análise dos resultados obtidos, é sugerida uma estratégia para aplicação dos critérios estruturais analisados.
Palavras chaves: Teste de Software, Testes Baseado em Predicados, Critérios de Teste Estrutural**.**

1 - Introdução

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 [11]. Salienta-se que a atividade de teste tem sido apontada como uma das mais onerosas no desenvolvimento de software.

Nesse contexto, 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. Estas ferramentas são desenvolvidas baseadas em técnicas e critérios de teste de software. As principais técnicas de teste são: a) Técnica funcional que utiliza aspectos funcionais do software para derivar os dados de teste; b) Técnica estrutural que utiliza o código fonte para derivar os casos de teste e baseia-se no conhecimento da estrutura interna do programa ou da especificação. Geralmente utiliza-se do grafo do programa ou grafo de fluxo de controle (GFC); e c) Técnica baseada em erros, que utiliza defeitos específicos, comuns em programação. A ênfase desta técnica está nos erros que o programador pode cometer durante o desenvolvimento.

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 [7], [14]. 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: a) Critérios baseados no fluxo de controle [7], [14]: 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; b) Critérios baseados no fluxo de dados [7], [14]: exploram a interação que envolve as definições e usos das variáveis no programa. Os principais critérios baseados em fluxo de dados são Todas-Definições, Todos-Usos, Todos-Du-Caminhos e Todos-Potenciais-Usos; e c) Critérios baseados em predicados, que 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) [15] 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 lógico/booleanos do programa em teste.

Os diferentes critérios de teste são considerados complementares pois podem revelar diferentes tipos de teste. Por isso, a realização de estudos empíricos é fundamental para se obter estratégias de aplicação dos critérios de teste existentes.

A aplicação prática dos critérios de teste e a condução de experimentos somente são possíveis se ferramentas de teste estiverem disponíveis. Entre as principais ferramentas destacam-se: a ferramenta Asset [3], para o teste de programas em Pascal e a Atac [5] para programas C. Elas utilizam os critérios de adequação baseados na análise de fluxo de dados definidos por Rapps e Weyuker [13], [14], além de alguns critérios baseados em fluxo de controle. Para os estudos descritos nesse trabalho, utilizou-se a ferramenta Poke-Tool [1], que apóia a utilização da família de critérios Todos Potenciais-Usos, critérios estruturais baseados em fluxo de dados, propostos por Maldonado [7], além dos critérios baseados em fluxo de controle Todos-Arcos e Todos-Nós. Para apoiar a utilização do critério BRO foi desenvolvida na Universidade da Carolina do Norte a ferramenta BGG [15], que apóia o teste de programas escritos em Pascal.

Na literatura encontram-se diferentes trabalhos, e entre esses estudos existem os que comparam os critérios estruturais [2], [7], [16], [18], [19], [20], [21]. Com relação aos critérios BOR e BRO, baseados em predicados, são poucos os relatos desse tipo de estudo. Tai [15] mostra resultados de comparações empíricas desses critérios com o critério Todos-Arcos. Vouk et al [10] 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. Talvez isso se deva ao fato de haver uma única ferramenta para o critérios BOR/BRO para o teste de programas Pascal e a grande maioria de ferramentas para critérios baseados em fluxo de dados apóia o teste de programas C. Apesar disso, experimentos com tais critérios são importantes porque eles podem ressaltar o relacionamento empírico entre os critérios baseados em predicados e outros critérios estruturais.

Dado o contexto descrito acima, esse trabalho tem como objetivo descrever uma ferramenta de apoio ao teste baseado em predicados, ou seja, dos critérios BOR e BRO para programas escritos em linguagem C. Essa ferramenta, chamada PredTOOL, permite a condução de experimentos de comparação e avaliação dos critérios implementados. Assim sendo, um experimento com a PredTOOL e com a ferramenta Poke-Tool, foi realizado. Esse experimento realizou comparações com os critérios BOR e BRO com dois outros critérios estruturais, o critério Todos-Arcos, baseado em fluxo de controle, o critério Todos Potenciais-Usos, baseado

em fluxo de dados. Os resultados mostram uma relação empírica que pode ser utilizada para se propor uma estratégia de utilização dos critérios estudados.

O trabalho está organizado da seguinte maneira. Na Seção 2 é apresentada uma revisão do teste estrutural de software e dos critérios utilizados. Na Seção 3 é descrita a ferramenta PredTOOL. Na Seção 4 são discutidos os resultados do experimento realizado. Na Seção 5 estão as conclusões e desdobramentos desse trabalho.

2 –Teste Estrutural de Software

Glen Myers [9] estabelece três regras que podem ilustrar os objetivos de teste: 1) A atividade de teste é o processo de executar um programa com a intenção de descobrir um erro; 2) Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um erro ainda não descoberto; 3) Um caso de teste é composto de uma entrada para o programa e da saída esperada; e 4) Um teste bem sucedido é aquele que revela um erro ainda não descoberto.

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 [7], [14]. 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. Nessa seção os critérios de teste estruturais utilizados nesse trabalho são descritos. Esses critérios consideram aspectos da estrutura interna do programa ou especificação para derivar os dados de teste. Os critérios estruturais mais conhecidos são:

1) Critérios baseados em fluxo de controle: Utilizam apenas características de controle da execução do programa, e os critérios mais importantes são: Todos-Nós [12]; Todos-Arcos ou Todos-Ramos [12], e Todos-Caminhos [4], [12] que exigem respectivamente que todos os nós, arcos e caminhos do GFC (grafo de fluxo de controle) sejam exercitados pelos casos de teste.
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 acordo com as localizações das definições e usos de variáveis no programa. Os principais representantes deste tipo de critérios são: Todas-Definições; Todos-Usos; Todos-Potenciais-Usos [7], [8], [14]. O critério Todos-Potenciais-Usos requer a execução de caminhos para cobrir associações entre um nó do grafo i que define uma variável x, com outros nós do grafo que potencialmente usam x.
3) O teste baseado em predicados consiste na execução de determinados tipos de testes para cada predicado (ou condição) do programa. Os critérios BOR (Boolean OperatoR testing) e BRO (Boolean and Relational Operator testing) [15], são os mais conhecidos critérios baseados em predicados e são o foco desse trabalho. Por isso, eles serão descritos em detalhe na Subseção 2.1.

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 [17]. 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/esforço é gasta determinando-se elementos não executáveis na aplicação dos critérios estruturais.

2.1 – Critérios BOR/BRO

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. Através do teste de C é possível detectar não somente erros em C ou nas instruções executadas antes de atingi-lo, mas 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 ("&&"), 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 boleano. Uma expressão booleanas é um predicado sem expressões relacionais.

Para formalizar a ocorrência de variáveis e operadores nas expressões apresentadas neste artigo, será utilizada a representação abaixo descrita:

- Bi com i > 0 denota uma variável booleanas,
- Ei uma expressão aritmética;
- <rop> ou <ropi> um operador relacional e
- <bop> ou <bopi> um operador binário boleano.

Para um predicado com n, n > 0 operandos simples,

- (<opd1> <bop1> <opd2> ... ... <bopn-1> <opdn),

onde <opdi>, 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 booleanas ou expressão relacional em <opdi>. 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.

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, $+\varepsilon$) 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.

Os critérios BOR e BRO requerem basicamente um conjunto de restrições para os predicados do programa em teste e que um conjunto de casos de teste T cubra as restrições requeridas.

O algoritmo, extraído de [15], para gerar restrições requeridas para os critérios BOR e BRO é apresentado abaixo.

Sejam u = (u1,..., um) e v = (v1,..., vn), onde m, n > 0, são duas listas de elementos. A concatenação de u e v, denotada (u, v), é (u1,..., um, v1,..., vn). 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 têm mais do que dois elementos, A % B pode 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 duas condições:

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.

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 2.1 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) = S1_f % S2_f e T(C) = {S1_t * {f2}} $ {{f1} * S2_t}
Onde: f1 ∈ S1_f e f2 ∈ S2_f e (f1, f2) ∈ F(C). Assim, T(C) ∪ F(C) é um conjunto de restrições BOR(BRO) para C.

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

T(C) = S1_t % S2_t e F(C) = {S1_f * {t2}} $ {{t1} * S2_f }
Onde: t1 ∈ S1_t e t2 ∈ S2_t e (t1, t2) ∈ T(C). Assim, T(C) ∪ F(C) é um conjunto de restrições BOR(BRO) para C.

Tabela 2.1- 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 S1_f = S2_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#) = {S1_f * {t2}} ∪ {{t1} * S2_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 booleanass 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, S1_t = {(=)} e S1_f = {(>),(<)}. No caso de C2, S2_t = {(>),(<)} e S2_f = {(=)}. Seguindo a Regra 2, temos que T(C#) = S1_t % S2_t = {(=)} % {(>),(<)} = {(=,>), (=,<)}. Como T(C#) tem dois elementos, escolhe-se (=, >) como (t1, t2). Deste modo, F(C#) = {S1_f * {>}} ∪ {{=} * S2_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 2.1.

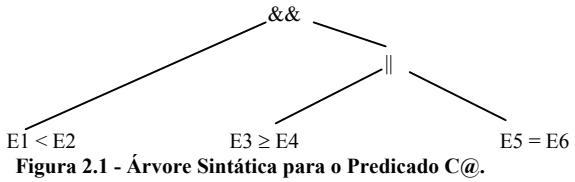

Figura 2.1 - Árvore Sintática para o Predicado C@.

Os seguintes passos são seguidos:

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

(2) Aplicar a regra 1 em 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 2.2, o processo de aplicação dos passos descritos acima é exibido com os conjuntos de restrições ao predicado C@.

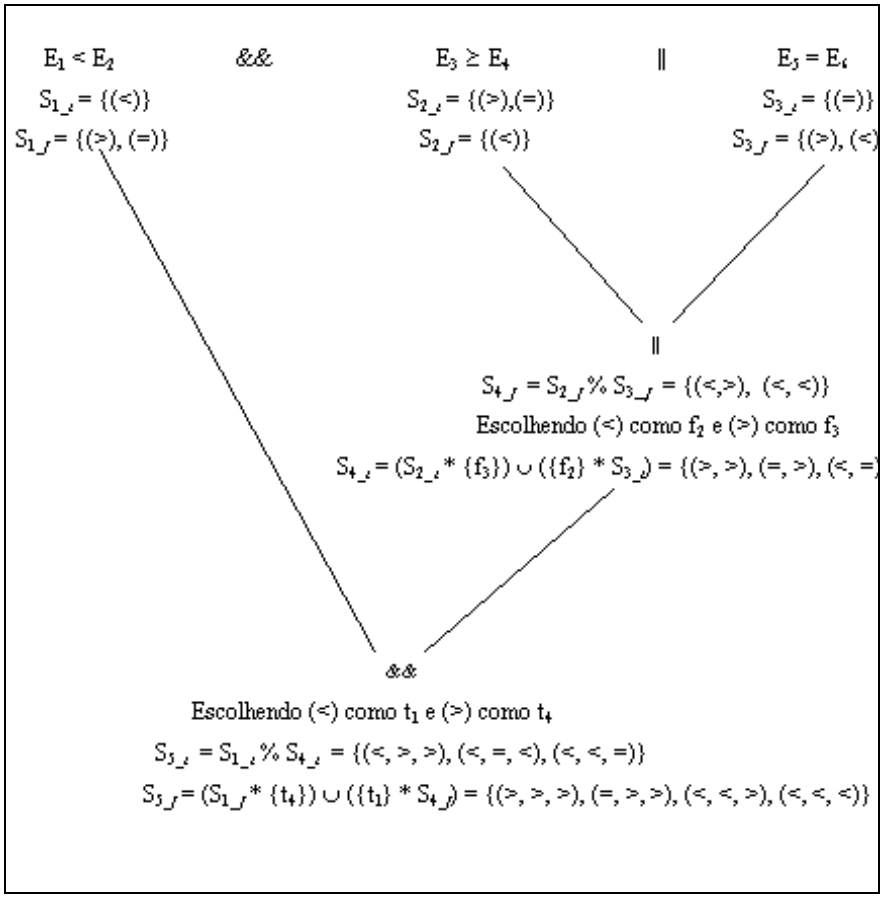

Figura 2.2 - Restrições BRO para o predicado ((E1 < E2) && ((E3 > E4) || (E5 = E6))),

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.

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. Assuma que P contém 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. Atenta-se para o fato de que o problema de determinar se uma restrição para um predicado é não executável é, em geral, indecidível [15]; problema análogo aos elementos não executáveis requeridos pelos outros critérios estruturais

3 – A Ferramenta PredTOOL

A Figura 3.1 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:

1) MÓDULO GERADOR – Identificação dos predicados: Esta função deve identificar os predicados da unidade a ser testada. A PredTOOL utiliza uma linguagem intermediária 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.

2) 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.

3) 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.

4) 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 pelo usuário. Os casos de teste fornecidos são também armazenados em arquivos.

5) AVALIADOR – Este módulo verifica se o conjunto de restrições executadas pelos casos de teste satisfaz 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 PredTOOL, atualmente, 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.

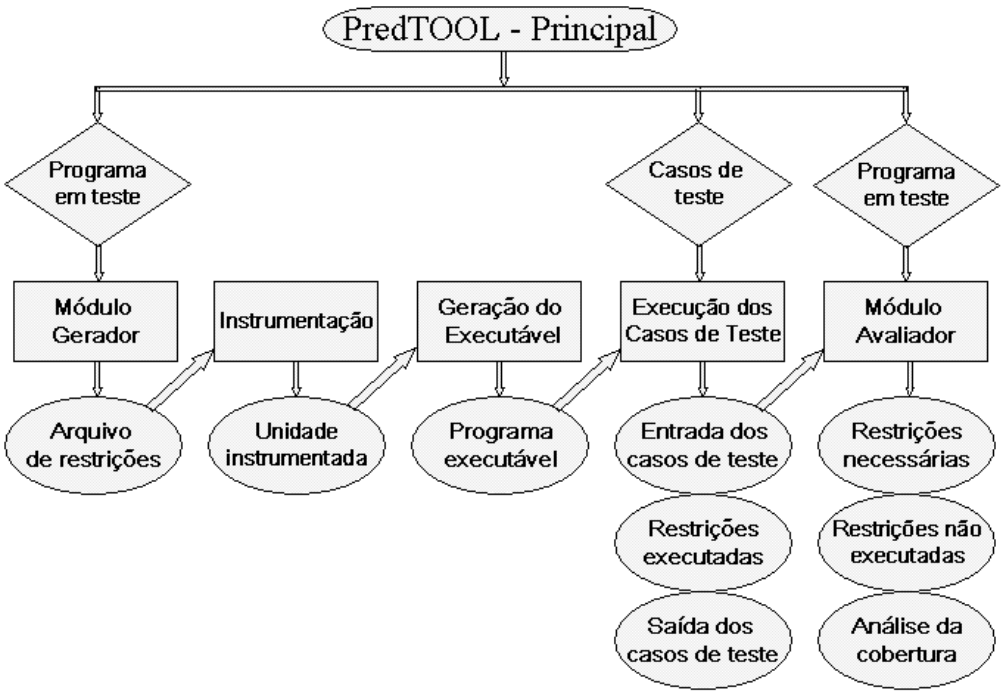

Figura 3.1 Principais Módulos da Ferramenta PredTOOL

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 3.2.

Deve-se informar o nome do arquivo sem extensão, pois a PredTOOL automaticamente reconhecerá o arquivo fonte para iniciar o processo de criação das restrições requeridas. Na figura também pode-se visualizar as restrições requeridas pelo predicado do programa utilizado como exemplo. A execução inicia com a leitura dos parâmetros do arquivo com o programa em linguagem intermediária 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.

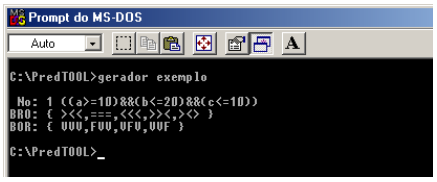

Figura 3.2 – Geração das Restrições Requeridas Pela PredTOOL.

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 dos nós do programa.

Ao final desta etapa, o arquivo com as restrições requeridas é criado e pode ser visto na Figura 3.3. 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 3.3 –Conteúdo do arquivo de restrições do exemplo utilizado.

O valor do primeiro campo representa o nó 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.

Nesta etapa também é gerado o arquivo que contém as pontas de prova, chamado de arquivo instrumentado. Esse arquivo gerado deve então ser compilado para obter 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. 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 3.1 mostra os dois casos de teste inseridos no exemplo descrito.

Tabela 3.1 – 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 1; 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 utilizados na avaliação.

O módulo AVALIADOR 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. 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 3.4 mostra o arquivo final onde estão os casos de (Tabela 3.1), 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 Restrições BRO: 20.00%
Total de Cobertura das Restrições BOR: 25.00%

Caso de Teste:  exemplo_pp2.pp
No: 1  ((a>=10)&&(b<=20)&&(c<=10))
Total de Cobertura das Restrições BRO: 20.00%
Total de Cobertura das Restrições BOR: 25.00%
----------------------------------------
Total de Cobertura das Restrições:
----------------------------------------
No: 1  ((a>=10)&&(b<=20)&&(c<=10))

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

Figura 3.4 – Arquivo de exemplo com avaliação final da PredTOOL do programa testado.

4 – Experimento

Apresenta-se 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 [6] e primeiramente utilizados por Weyuker [19].

4.1 – Descrição do Experimento

A descrição funcional dos programas escolhidos encontra-se na Tabela 4.1.

Tabela 4.1 - 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 ad-hoc de testes T baseado na funcionalidade de cada programa;

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 elementos não executáveis e gerar os casos de teste adicionais para cobrir todos os elementos executáveis. As Tabelas 4.2 e 4.3 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 conjuntos TBOR e TBRO, respectivamente, BOR e BRO adequados. Os resultados encontram-se nas Tabelas 4.4 e 4.5.

5º – Submeter os conjuntos TARCS e TPU, na PredTOOL, para obter a cobertura para os critérios BOR e BRO. As Tabelas de 4.6, 4.7, 4.8 e 4.9 mostram os resultados obtidos.

6º – Submeter os conjuntos TBOR e TBRO na POKE-TOOL para obter a cobertura dos critérios ARCS e PU. As Tabelas 4.10, 4.11, 4.12 e 4.13 mostram os resultados deste passo.

Tabela 4.2 - Resultados da Aplicação de Todos-Arcos.

CRITÉRIO ARCS - RESULTADOS				
Programa	Requeridas	Não executáveis	(TPU)	Cobertura 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
Total	63	0	32	100,0

Tabela 4.3 - Resultados Todos Potenciais-Usos.

CRITÉRIO PU - RESULTADOS				
Programa	Requeridas	Não executáveis	Efetivos (TPU)	Cobertura 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
Total	425	82	71	80,7

Tabela 4.4 - Resultados da Aplicação do Critério BOR.

CRITÉRIO BOR – RESULTADOS				
Programa	Requeridas	Não executáveis	Efetivos	Cobertura (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	111	0	32	100,0

Tabela 4.5 - Resultados da Aplicação do Critério BRO.

CRITÉRIO BRO – RESULTADOS				
Programa	Requeridas	Não executáveis	Casos de teste	Cobertura (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	167	21	41	87,4

Tabela 4.6 - TARCS na PredTOOL. – Critério BOR

CRITÉRIO BOR – RESULTADOS COM TARCS		
Programa	Restrições Cobertas	Cobertura a (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	108	97.7

Tabela 4.7 - TARCS na PredTOOL - Critério BRO.

CRITÉRIO BRO – RESULTADOS COM TARCS		
Programa	Restrições Cobertas	Cobertura a (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	126	75,4

Tabela 4.8 - TPU na PredTOOL - Critério BOR.

CRITÉRIO BOR – RESULTADOS COM TPU		
Programa	Restrições Cobertas	Cobertura (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	109	98,1

Tabela 4.9 - TPU na PredTOOL - Critério BRO.

CRITÉRIO BRO – RESULTADOS COM TPU		
Programa	Restrições Cobertas	Cobertura (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	133	79,6

Tabela 4.10 - TBOR na POKE-TOOL - Critério ARCS.

CRITÉRIO ARCS– RESULTADOS COM TBOR		
Programa	Restrições Cobertas	Cobertura (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	63	100,0

Tabela 4.11 - TBOR na POKE-TOOL - critério PU.

CRITÉRIO PU – RESULTADOS COM TBOR		
Programa	Restrições Cobertas	Cobertura (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	220	51,7

Tabela 4.12 - TBRO na POKE-TOOL - Critério ARCS.

CRITÉRIO ARCS – RESULTADOS COM TBRO		
Programa	Restrições Cobertas	Cobertura (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	63	100,0

Tabela 4.13 - TBRO na POKE-TOOL - Critério PU.

CRITÉRIO PU – RESULTADOS COM TBRO		
Programa	Restrições Cobertas	Cobertura (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	265	62,3

4.2 – Análise dos Resultados

Os resultados 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) [21].

4.2.1– 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 [11]. O programa *getcmd.c* 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%.

4.2.2 – Strength

Para analisar o strength, utilizam-se as Tabelas de 4.6 a 4.13. 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, e pode ser vista na Figura 4.1:

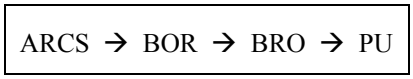

Figura 4.1 – Estratégia de Aplicação dos Critérios do Experimento.

5 – Conclusão

Esse trabalho descreveu a ferramenta PredTOOL, que apóia a utilização dos critérios BOR e BRO para programas escritos em C. A ferramenta contribui com a atividade de teste, melhorando a confiabilidade dos programas testados nessa linguagem, amplamente utilizada. Ela permite que uma estratégia de teste, que envolva a aplicação de diferentes e complementares critérios seja utilizada, pois a aplicação manual desses critérios é impraticável.

Uma outra contribuição da PredTOOL é a condução de experimentos. Apresentou-se 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 no experimento.

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 relacionamento entre os critérios obtido neste experimento foi 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

proposta. 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.

Outros experimentos, comparando-se os critérios BOR e BRO com critérios baseados em erros poderão ser realizados. Outros fatores de comparação, como a eficácia e outro conjunto de programas poderão ser considerados.

A ferramenta PredTOOL também pode ser modificada. Pretende-se construir uma interface gráfica para tornar seu uso mais fácil e implementar mecanismos que auxiliem o usuário na determinação de restrições não executáveis.

Referências

[1] 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.

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

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

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

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

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

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

[8] 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.

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

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

[11] 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.

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

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

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

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

[16] 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.

[17] VERGILIO, S. R.; MALDONADO, J. C.; JINO, M. *Caminhos não executáveis na automação das atividades de teste***.** In: Simpósio Brasileiro De Engenharia De Software. p. 343-356. nov. 1992

[18] WEYUKER, E. J. *An empirical study of the complexity of 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.

[19] 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.

[20] 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.

[21] 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.