Estratégia de Geração de Dados de Teste Baseada na Análise Simbólica e Dinâmica do Programa

Juliana Silva Herbert
Ana Maria de Alencar Price

juliana@inf.ufrgs.br
anaprice@inf.ufrgs.br

Curso de Pós-graduação em Ciência da Computação
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre, RS

Resumo

A geração automática de dados é considerada a tarefa mais crítica do teste estrutural. A obtenção de um conjunto de dados confiável, de propósito geral, não é computável. Assim sendo, os estudos de geração de dados devem basear-se em critérios específicos de teste. Os critérios utilizados para o teste estrutural abordados neste trabalho são os que selecionam caminhos a partir da análise dos fluxos de controle e de dados do programa. Para que a cobertura seja atingida, dados devem ser gerados de tal forma que permitam a execução de todos os caminhos selecionados. Por outro lado, a programação em lógica tem sido bastante utilizada para a implementação de protótipos de ferramentas de apoio às fases do desenvolvimento de software, inclusive à fase de validação. A linguagem Prolog, por exemplo, possui características, próprias para o processamento simbólico, tais como: formalismo DCG, reversibilidade e prova de teoremas, que são extremamente adequadas ao teste estrutural de software, mais especificamente à geração automática de dados para teste. Este artigo apresenta uma proposta de estratégia de geração automática de dados para o teste estrutural, baseada na programação em lógica. Nesta estratégia, são realizadas análises simbólica e dinâmica do programa, a partir das quais são extraídas informações que, combinadas, constituem-se em heurísticas para a geração de dados.

Palavras-chave: teste estrutural; geração de dados para teste; programação em lógica.

Abstract

Automatic generation of data is considered to be the most difficult task of structural testing. Obtaining a reliable general purpose data set is not computable. In this way, data generation studies must limit their scope to specifical error cathegories, which can be discovered by test criteria. The criteria used in structural testing are those that select paths based on the control and data flow program analysis. To cover all the program, data must be generated in such a way to allow the execution of all selected paths. On the other hand, logic programming has been very used to implement tools prototypes that support software development cycle phases, including validation phase. Prolog language, for example, has features adequated to symbolic processing, such as DCG formalism, reversibility and theorem proving, that are very suitable to software structural testing, specially to test data automatic generation. This paper presents a data automatic generation strategy to structural testing, based on the logic programming. In this strategy it is joined the program symbolic and dynamic evaluation, it is extracted information that, combined, can be data generation heuristics.

Keywords: structural testing; testing data generation; logic programming.

1. Introdução

À medida em que o projetista de software adquire maior experiência na construção de sistemas de software, novas regras de transformação aplicáveis ao processo de desenvolvimento são elaboradas, as quais, dificilmente, são incorporadas às ferramentas CASE em uso, dada a maneira pela qual estas, em geral, são construídas.

A aplicação da Programação em Lógica à construção de ferramentas CASE pode refletir de forma mais natural a dinâmica deste processo. O objetivo é facilitar a construção de sistemas especialistas para a Engenharia de Software que incluam heurísticas para apoiar a tomada de decisões, permitindo a adição incremental de conhecimento sobre o processo de desenvolvimento e a atualização periódica deste conhecimento especializado.

A aplicação do paradigma da programação em lógica permite a criação de ferramentas que extraiam informações essenciais do código, combinando-as de forma útil e eficiente, para aplicar transformações ao programa, testá-lo, analisá-lo e depurá-lo, entre outras aplicações. Além disso, após uma série de usos, a ferramenta pode incorporar à sua base de conhecimento informações obtidas a partir de sucessos e/ou falhas ocorridos [CAN92]. A análise passa a ser realizada de forma dinâmica, certamente mais completa e rica em informações.

O teste de programas pode ser definido como o processo de executar o sistema, com dados de entrada selecionados, a fim de analisar seu comportamento [PRE95]. Hamlet [HAM95] afirma: "As atividades de '*execução*' e de '*seleção de dados de entrada*' são as mais importantes na fase de testes: a **arte** de testar um programa inicia quando a **pessoa** que conduz o teste seleciona os valores de entrada. Um **computador** realiza a execução".

A geração de dados é considerada a tarefa mais crítica do teste estrutural. Um conjunto de dados de teste é adequado se o programa, caso correto, para o qual este conjunto foi gerado, reage de forma esperada, e se o programa incorreto apresenta erros. Se o conjunto de dados é adequado, então é confiável [DEM87]. Um procedimento de seleção de um conjunto de dados de teste válido e confiável, de propósito geral, não é computável. Desta forma, os estudos relacionados à geração de dados devem basear-se em critérios específicos de teste.

A aplicação de programação em lógica, através da linguagem PROLOG, no caso deste trabalho, favorece a construção de uma ampla base de conhecimento sobre o programa em análise, oferecendo também uma linguagem de consulta a esta base e permitindo que o conjunto de regras que extrai informações possa ser ampliado em tempo de execução.

Pesquisas que envolvem a construção de protótipos de ferramentas de teste são essenciais no desenvolvimento e na análise de técnicas de teste. A quantidade de informações envolvidas neste processo é bastante grande, o que inviabiliza o trabalho manual [MYE79]. Por outro lado, a maior parte das ferramentas encontradas na literatura (por exemplo, ASSET [FRA87], ATAC [HOR91], POKE-TOOL [MAL90] e PROTESTE+ [SIL95]), foram construídas utilizando linguagens procedimentais. O problema deste tipo de estratégia é o tempo e os esforços envolvidos em seu desenvolvimento e manutenção. Protótipos são considerados essenciais, porque apenas após implementar e executar várias vezes uma determinada técnica de teste, tem-se subsídios para modificá-la e aprimorá-la, o que tem como conseqüência direta, na maioria das vezes, modificações na implementação da ferramenta [HAM95].

Ou seja, embora a estratégia convencional, de utilizar linguagens de programação procedimentais seja mais adequada para a construção de ferramentas de apoio ao teste, que serão utilizadas na prática, a aplicação da programação em lógica, como linguagem de implementação ou ferramenta de prototipação, é mais adequada para a abstração de detalhes de baixo nível de implementação, tais como a definição de estruturas de dados mais complexas e procedimentos para sua manipulação, e para a ênfase nas técnicas de teste a serem implementadas, fatores importantes na fase

de proposição destas técnicas, que devem ser validadas de forma rápida e ser modificadas de forma prática.

Vários tipos de informações são necessários para o teste estrutural de um programa, sendo estas classificadas, principalmente, em relação ao fluxo de controle e ao fluxo de dados. Informações relacionadas ao fluxo de controle representam o grafo de fluxo de controle do programa e seu diagrama de chamadas de funções [SIL94] [SIL95]. Informações relacionadas ao fluxo de dados intra e interprocedimental do programa incluem escopos de declarações de variáveis, registros de parâmetros formais e reais e definições e usos de variáveis. A base de conhecimento resultante pode ser consultada a partir de cláusulas Prolog, além de poder ser validada utilizando técnicas de validação de base, propostas na literatura por [BEN93], [JAF93] e [MEN93], entre outros. Tais técnicas podem ser utilizadas para validar a base em relação a possíveis problemas de relacionamentos entre informações, tais como variáveis definidas mas não usadas, variáveis usadas mas não definidas e alguns casos de construções em laços sem fim, através da análise de sua invariante. Desta forma, tem-se as informações relativas ao código do programa validadas, permitindo que relacionamentos entre estas sejam explorados de forma estática e dinâmica, para entender, testar e depurar o programa.

1.1. Trabalhos Relacionados

Existem trabalhos, na área de projeto, desenvolvidos em Prolog. Apesar de não estarem relacionados diretamente à área de validação, o trabalho de Tse [TSE94] é citado por envolver a construção de uma base de conhecimento, a partir da qual são aplicadas regras de transformação ao programa. Em [TSE94], Prolog foi utilizado para representar diagramas de fluxo de dados, diagramas estruturados e dicionários de dados. Especificações expressas em forma de uma hierarquia de diagramas de fluxo de dados codificadas em Prolog são transformadas automaticamente em diagramas estruturados e avaliadas através de um conjunto de critérios oriundos da metodologia do "Projeto Estruturado". Se o resultado da avaliação não satisfizer os critérios estabelecidos, o sistema retrocede (*backtracking*) e produz um novo diagrama estruturado que é submetido a uma nova avaliação. Na ferramenta em questão, os implementadores consideraram heurísticas definidas por especialistas e engenheiros de software experientes na aplicação da Metodologia do Projeto Estruturado para o desenvolvimento de sistemas.

Informações relacionadas ao fluxo de controle e, principalmente, ao fluxo de dados, podem ser obtidas a partir de código fonte, através do processo de engenharia reversa [HOL87]. De acordo com [CAN92], a maior parte das ferramentas que realizam engenharia reversa fornecem informações genéricas ou minuciosas demais, dificultando a tarefa do programador que mantém o software. Informações genéricas somente têm utilidade no processo inicial de análise do programa, enquanto que detalhes desejados geralmente não são apresentados de forma coesa e interrelacionada, representando também um problema para a tarefa de manutenção do software.

Outros autores já consideraram a programação em lógica como ferramenta de auxílio para o teste de programas e para a geração automática de dados. Entre eles, são citados quatro autores que mais se relacionam ao trabalho: Khanna [KHA91], Cross [CRO91], Hamlet [HAM95] e Hoffman [HOF91]:

Khana aplica a programação em lógica para a construção de uma árvore de decisões, a partir do processo de análise. A árvore é usada para a avaliação de predicados de caminho e para a avaliação simbólica das variáveis de saída. A estratégia baseia-se nas "Múltiplas Teorias Dinâmicas" em lógica, organizadas em uma estrutura de árvore. Expressões condicionais geram duas teorias por exemplo, uma para o valor verdadeiro e outra para o valor falso. As teorias não armazenam o ambiente simbólico completo, evitando uma grande quantidade de informações armazenadas (um dos problemas do método de Korel [KOR90]). É utilizado um mecanismo de herança para passar os valores simbólicos da "teoria-pai" para as teorias descendentes. As teorias são então analisadas de forma simultânea, em uma seqüência particular, no lugar de examiná-las em ramos distintos, a partir da teoria

raiz. Os problemas neste método, de acordo com Khanna, são a dificuldade de construção de um provador de teoremas e de um solucionador de inequações.

Cross analisa a cobertura das decisões realizada por dados de teste utilizados em execuções anteriores, utilizando um sistema especialista para gerar novos dados para coberturas adicionais. Regrás heurísticas modificam os casos previamente utilizados para que a cobertura desejada seja atingida. Estas heurísticas baseiam-se: no tipo e domínio das variáveis de entrada, no cálculo percentual da diferença entre o caso de teste e o limite da condição e no caso de teste que mais se aproximou do limite da condição. Estas heurísticas não detectam caminhos não executáveis e não garantem que o mesmo caminho será executado até a condição em questão, quando uma variável tem seu valor alterado.

Hamlet apresenta um método geral para implementar protótipos de ferramentas de teste de programas, utilizando Prolog. A linguagem Prolog é apresentada como um ambiente auto-suficiente, no qual técnicas de teste podem ser definidas e implementadas. As seguintes tecnologias são utilizadas no método proposto em [HAM95]:

- Programas auto-instrumentados: o comportamento dos programas é analisado sem que seja necessário monitorá-los. Seu código é modificado, inserindo-se comandos, na própria linguagem de implementação, que sinalizam condições relacionadas ao estado de execução do programa.
- Geradores de *parsers* baseados em tabelas: construção do *parser* a partir da especificação da gramática da linguagem.
- Bases de conhecimento: Prolog pode expressar fatos sobre programas e execuções, em uma base, que pode ser utilizada em consultas interativas. O estilo declarativo de Prolog permite que sejam **descritos** métodos de análise de software.

O paradigma de análise utilizado é o seguinte [HAM95]:

- o programa é analisado pelo *parser*, gerando fatos Prolog sobre a análise estática;
- a auto-instrumentação é realizada, a fim de gerar fatos Prolog sobre a análise dinâmica;
- o programa é executado, gerando os fatos sobre a análise dinâmica;
- segmentos de código Prolog são colocadas em bibliotecas, a fim de descrever a análise a ser realizada a partir dos fatos armazenados na base;
- a interface do programa com o usuário é o sistema de consultas Prolog, que permite que a biblioteca citada ajude na investigação da base gerada.

Hoffman apresenta técnicas para escrever casos de teste em Prolog, que automaticamente testam módulos implementados em C. Alguns pontos fracos foram identificados nos trabalhos acima citados. Estes pontos são apresentados a seguir, sendo alguns deles resolvidos pela estratégia apresentada neste trabalho (contribuições - Seção 1.2).

Os problemas encontrados na representação proposta por Khanna [KHA91], foram o tamanho dos predicados especificados nos nodos mais inferiores da árvore e a dificuldade de representação e execução simbólica de estruturas de controle do tipo "laço". O método de geração de dados proposto por Cross [CRO91] não define como as transformações que o programa realiza com as variáveis de entrada devem ser tratadas, bem como não garantem que o caminho executado até determinado momento de teste continue sendo executado.

O método proposto por Hamlet [HAM95] não define procedimentos para a geração de dados para o teste. Além disso, utiliza o procedimento de "auto-instrumentação" de programas, o que ocasiona modificações em seu código, procedimento complexo e desaconselhável.

1.2. Contribuições do Trabalho

As principais contribuições do presente trabalho são:

- a tradução automática de fatos Prolog, da base de conhecimento, para cláusulas Prolog para a execução simbólica;
- a representação utilizada para as cláusulas de execução simbólica, que permite a execução de qualquer caminho de programa especificado, inclusive a iteração de laços. Tais cláusulas, além de serem simples, podem ser diretamente transformadas em cláusulas para a execução real, trocando-se os operadores relacionais e incluindo testes de valores reais nas cláusulas de decisão.

Em relação aos problemas identificados nos trabalhos relacionados à geração de dados, apresentados na Seção 1.1, a estratégia de geração de dados de teste, no ambiente LOGTEST (Seção 4), apresenta as seguintes melhorias:

- a representação de cláusulas de execução simbólica e real é simples e compacta;
- os predicados gerados têm seu tamanho aumentado a medida em que novas condições são a ele agregadas, apenas em momentos de execução simbólica ou real. Por exemplo, um laço influencia no tamanho do predicado com exatamente o número de condições igual ao número de iterações que foram realizadas (em relação ao métodos proposto por Khanna [KHA91]);
- no procedimento de execução simbólica, encontram-se completamente definidos os tratamentos de variáveis, sejam elas de entrada ou internas à rotina analisada (em relação ao método proposto por Cross [CRO91];
- a "auto-instrumentação" [HAM95] é evitada utilizando-se cláusulas Prolog para a execução real, geradas no mesmo processo de geração de cláusulas para a execução simbólica, a partir da base de conhecimento com informações estáticas. Desta forma, evita-se modificações no código original.

Uma questão importante não solucionada nos trabalhos citados, e também não foi resolvida por este trabalho, é a existência de caminhos não-executáveis. Maiores estudos devem ser realizados referentes a esta questão. A infra-estrutura já criada no LOGTEST permite que técnicas para a identificação de caminhos não-executáveis sejam implementadas rapidamente, para sua fácil validação.

1.3. Organização do Trabalho

O artigo foi organizado da seguinte forma: a primeira seção apresentou a motivação do trabalho, esclarecendo seus objetivos e pontos de contribuição. Na segunda seção, são apresentadas algumas considerações sobre a geração de dados no teste estrutural. A Seção 3 apresenta as características da programação em lógica que foram consideradas importantes para a construção do ambiente LOGTEST, descrito sucintamente na Seção 4. E, finalmente, na Seção 5, é apresentada a estratégia de geração de dados proposta, inserida no contexto do LOGTEST. A Seção 6 apresenta as conclusões alcançadas e pontos de extensão. E, finalmente, a Seção 7 apresenta as referências bibliográficas nas quais este trabalho se baseia.

2. Geração de Dados no Teste Estrutural

O processo de teste estrutural é direcionado por critérios de seleção de caminhos de teste, baseados na análise dos fluxos de controle e de dados de módulo e de sistema (por exemplo, critérios propostos por Maldonado [MAL92] e Rapps&Weyuker [RAP85]). O teste estrutural utiliza-se do código do programa para, baseado em um critério de seleção de caminhos, selecionar caminhos de teste. A partir deste ponto, surgem dois problemas altamente complexos:

- determinação de dados que executem os caminhos selecionados;
- identificação de caminhos selecionados não executáveis (*infeasible paths*) [VER94].

A identificação de caminhos impraticáveis requer a utilização de um provador de teoremas. A construção de um provador de teoremas tem como conseqüência direta altos custos [GOL94]. Além disso, sua utilização tem como pré-requisitos a existência de uma especificação formal do sistema, além do conhecimento dos programadores na área de verificação formal, fatores raramente existentes nos sistemas de software atuais. A fim de minimizar tais problemas, decidiu-se utilizar, neste trabalho, a execução simbólica, que, apesar de continuar necessitando de um provador de teoremas, não torna necessária a existência da especificação formal do programa [DAR78]. O programa é especificado através das cláusulas Prolog, as quais produzem, no final da execução simbólica, predicados de caminho e contextos de variáveis que são utilizadas diretamente no provador de teoremas.

Desta forma, a construção de um provador de teoremas é bem mais simples de ser feita em uma linguagem declarativa. Até o presente momento o provador não foi implementado no ambiente, mas já está sendo estudado e deverá ser uma das próximas extensões futuras do mesmo.

A execução simbólica é a mais promissora dentre as técnicas para a determinação de dados para a execução do caminho selecionado [HOW78]. O programa é executado com valores simbólicos e é produzido um predicado de caminho que, se resolvido (a partir de um sistema de inequações), determina os intervalos das variáveis de entrada que ocasionam a execução do caminho. O predicado gerado pode, eventualmente, ser uma expressão complexa que deve ser simplificada, ou então, não ter solução [COE91].

Um teste simbólico do programa é equivalente a uma grande quantidade de testes reais [DAR78]. O único requisito para que a execução simbólica seja viável é a existência da definição formal da linguagem de programação em uso. A execução simbólica está entre a verificação e o teste convencional. Por exemplo, um dos problemas da verificação formal é a identificação da invariante dos laços. Este problema não existe na execução simbólica, já que pode-se executar o laço um número arbitrário de vezes, dependendo do interesse do testador. Dependendo de como o comportamento simbólico é expresso formalmente, os valores gerados por um interpretador simbólico podem ser utilizados tanto para o teste convencional (com valores reais) quanto para a prova formal de correteza do programa [DAR78]. Para aplicar a execução simbólica para a geração de dados, é necessário:

- um interpretador simbólico para o programa;
- um simplificador de expressões geradas pela execução;
- um solucionador de equações;
- um provador de teoremas.

Estas ferramentas não são triviais de implementar em linguagens de programação procedimentais. Por outro lado, as linguagens declarativas apresentam facilidades para a programação de processadores como os acima citados.

3. Programação em Lógica

A programação em lógica apresenta características particularmente aplicáveis à solução dos problemas mencionados na Seção 2:

- é orientada ao processamento simbólico: construir um reconhecedor sintático em uma linguagem de programação em lógica, tal como Prolog, constitui-se em escrever uma gramática. A especificação de um interpretador requer um pouco mais de esforço, que pode ser considerado mínimo se comparado ao esforço necessário para codificar um interpretador em uma linguagem procedimental;
- devido também ao processamento simbólico, desenvolver um simplificador de expressões, bem como um solucionador de equações, é um processo bem menos complexo do que em uma linguagem procedimental;

- um programa Prolog pode ser visto como se fosse um conjunto de axiomas, dos quais deseja-se derivar um teorema.

Cláusulas Prolog (escritas em DCG - *Definite Clause Grammar*), descrevendo a gramática de uma linguagem de programação, podem constituir analisadores léxico e sintático para os programas codificados nesta linguagem. Neste processo de reconhecimento do programa, informações do programa sobre fluxos de dados e de controle podem ser extraídas de forma direta, e, então, relacionadas para que casos e dados de teste sejam derivados. A partir de sucessivas execuções do programa, simbólicas e/ou reais, a base de conhecimento do programa pode ser incrementada, de forma a facilitar a geração de dados para a execução de caminhos selecionados.

A geração automática de dados que ocasionem a execução de determinadas partes do programa é um problema freqüentemente abordado na literatura. Combinando estratégias de teste de caminhos com a programação em lógica, são aproveitadas características do Prolog, tais como *backtracking*, construção de base de conhecimento através de regras, e modificação de cláusulas de reconhecimento em tempo de execução. Além disso, heurísticas podem ser identificadas e definidas, a partir da base de conhecimento, a fim de determinar os dados mais adequados para a execução do programa (por exemplo, as heurísticas definidas por Rombaldi [ROM96]).

A depuração de programas, procurando segmentos destes que contenham erros e corrigindo-os, é outra área desamparada por ferramentas em geral. Além de ser uma tarefa complexa, a depuração não costuma ser muito atrativa devido ao grande número de dificuldades de entendimento do programa, como por exemplo, relacionamentos entre variáveis usadas em diferentes segmentos. Sistemas baseados em conhecimento podem ser utilizados para facilitar tal tarefa, pois seu poder de depuração está associado à base de conhecimento que armazena informações relativas ao programa. Outras utilidades são o armazenamento de alternativas de implementação de determinadas funções (planos de algoritmos), causas de erros mais comuns e seu reparo, além do uso de assertivas para comparação da especificação com o código do programa.

4. LOGTEST

LOGTEST é um ambiente de apoio ao teste estrutural, inicialmente desenvolvido para analisar programas codificados em Pascal, implementado em Prolog, em desenvolvimento no Instituto de Informática, UFRGS. As seguintes capacidades encontram-se disponíveis atualmente:

- análise estrutural da linguagem Pascal, através de DCG e construção de uma base de conhecimento do programa, a partir de informações extraídas da análise estrutural;
- consultas à base de conhecimento, auxiliando o programador no entendimento do programa;
- seleção de caminhos para teste, com os critérios apresentados em [MAL92] e [RAP85];
- geração automática de cláusulas para a execução simbólica do programa, as quais permitem que o caminho seja especificado pelo usuário ou por um critério de seleção de casos de teste, utilizando o mesmo conjunto básico de cláusulas;
- estratégia de geração de dados baseado em heurísticas propostas por Rombaldi [ROM96]. Tais heurísticas examinam os predicados e as computações de caminhos obtidos na execução simbólica, obtendo a função inversa das operações, com o objetivo de cobrir ramos do grafo de fluxo de controle que não tenham sido executados;
- estratégia de geração de dados descrita na Seção 5 deste artigo;
- geração de slices estáticos e dinâmicos, que são trechos de programas relacionados à determinada variável, isolados para diminuir a complexidade da depuração de código [SIL96];
- depuração baseada em conhecimento, com o armazenamento e reconhecimento de planos do programa [PAL96].

Apesar do ambiente ter sido configurado, inicialmente, para o reconhecimento de programas em Pascal, pode-se configurá-lo para outra linguagem procedimental, através da representação da gramática da nova linguagem em formato DCG. A seguir é apresentado um exemplo de execução do processamento básico de LOGTEST, consistindo da geração da base de conhecimento e da tradução dos fatos Prolog da base para cláusulas de execução simbólica do programa. É utilizado um programa simples e pequeno a fim de facilitar o entendimento das cláusulas geradas. A base é obtida através da análise estrutural, e as cláusulas de execução simbólica através da submissão da base a um tradutor, também construído em Prolog. A Figura 1 apresenta o código da rotina "potencia" e seu grafo de fluxo de controle. A Figura 2 apresenta os fatos gerados a partir da análise estrutural, enquanto que na Figura 3 estão representadas as cláusulas Prolog, geradas pelo tradutor, para a execução simbólica. Este último segmento, quando executado com um caminho, gera fatos Prolog com valores simbólicos, os quais são apresentados na Figura 4. A seguinte representação é utilizada na Figura 2:

- controlFlowGraph(Programa,NroNodos): relaciona o nome do programa (*Programa*) com o número de nodos (*NroNodos*) do grafo de fluxo de controle correspondente;
- node(Programa,NroNodo,TipoNodo): relaciona o nodo de número *NroNodo*, do programa *Programa*, com o tipo *TipoNodo* (*begin, while, repeat, until, sequential, empty, end*, entre outros);
- edge(Programa,NroNodo1,NroNodo2): representa o arco que tem origem no nodo *NroNodo1*, chegando ao nodo *NroNodo2*, do programa *Programa*.
- varDeclaration(Programa,Variável,TipoVariável): relaciona a declaração da variável *Variável*, no programa *Programa*, com o tipo *TipoVariável*;
- varDefUse(Programa,NroNodo,Ordem,Variável,TipoRef,Valor): A variável *Variável* tem um tipo de referência *TipoRef* (definição ou c-uso [RAP85]) no nodo *NroNodo*, em relação ao *Programa*, na *Ordem* de apresentação no código do nodo (que é a mesma para definições e c-usos). Se *TipoRef* indicar uma definição, então *Valor* representa o valor de definição de *Variavel*. Se *TipoRef* indicar um c-uso, então *Valor* contém uma lista vazia ([]).
- varPUse(Programa,NroNodo1,NroNodo2,Variável,Predicado): Os nodos de números *NroNodo1* e *NroNodo2* são os nodos origem e destino do arco que possui o *Predicado* associado, onde há um p-uso da *Variável*.

```
procedure potencia(x,y:integer; var z:real);
var p: integer;
begin
 p:=0; z:=0;
 if y>0 then p:=y
        else p:=-y;
 z:=1;
 while p<>0 do begin
                 p:=p-1;
                 z:=z*x;
               end;
 if y<0 then z:=1/z;
end;
```

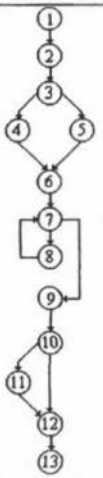

Figura 1. Código do programa "potencia" e seu grafo de fluxo de controle.

```
controlFlowGraph(potencia,13).

node(potencia,1,begin).
node(potencia,2,sequential).
node(potencia,3,if_then_else).
node(potencia,4,sequential).
node(potencia,5,sequential).
node(potencia,6,sequential).
node(potencia,7,while).
node(potencia,8,sequential).
node(potencia,9,empty).
node(potencia,10,if_then).
node(potencia,11,sequential).
node(potencia,12,empty).
node(potencia,13,end).

edge(potencia,1,2).
edge(potencia,2,3).
edge(potencia,3,4).
edge(potencia,3,5).
edge(potencia,4,6).
edge(potencia,5,6).
edge(...

varDeclaration(potencia,p,integer).

varDefUse(potencia,1,1,x,def,input).
varDefUse(potencia,1,2,y,def,input).
varDefUse(potencia,2,1,p,def,0).
varDefUse(potencia,2,2,z,def,0).
varDefUse(potencia,4,1,y,use,[]).
varDefUse(potencia,4,2,p,def,y).
varDefUse(potencia,5,1,y,use,[]).
varDefUse(potencia,5,2,p,def,-y).
varDefUse(potencia,6,1,z,def,1).
varDefUse(potencia,8,1,p,use,[]).
varDefUse(potencia,8,2,p,def,p-1).
varDefUse(potencia,8,3,z,use,[]).
varDefUse(potencia,8,4,x,use,[]).
varDefUse(potencia,8,5,z,def,z*x).
varDefUse(potencia,11,1,z,use,[]).
varDefUse(potencia,11,2,z,def,1/z).

varPUse(potencia,3,4,y,y>0).
varPUse(potencia,3,5,y,not y>0).
varPUse(potencia,7,8,p,p<>0).
varPUse(potencia,7,9,p,not p<>0).
varPUse(potencia,10,11,y,y<0).
varPUse(potencia,10,12,y,not y<0).
```

Figura 2. Fatos Prolog representando as informações da rotina *potencia*.

```
node(1) :- asserta(value(pp,emptyPath,1,0)),
        asserta(value(x,x,1)),
        asserta(value(y,y,1)).

node(2) :- asserta(value(p,0,2)),
        asserta(value(z,0,2)).

edge(3,4) :- value(pp,Path,_,_),
        value(y,Y,_),
        Path2 = Path + (Y>0),
        asserta(value(pp,Path2,3,4)).

node(4) :- value(y,Y,_),
        P=Y,
        asserta(value(p,P,4)).

edge(3,5) :- value(pp,Path,_,_),
        value(y,Y,_),
        Path2 = Path + (not Y>0),
        asserta(value(pp,Path2,3,5)).

node(5) :- value(y,Y,_),
        P = -Y,
        asserta(value(p,P,5)).

node(6) :- asserta(value(z,1,6)).

edge(7,8) :- value(pp,Path,_,_),
        value(p,P,_),
        Path2 = Path + (P<>0),
        asserta(value(pp,Path2,7,8)).

node(8) :- value(p,P,_),
        P2 = P-1,
        asserta(value(p,P2,8)),
        value(z,Z,_),
        value(x,X,_),
        Z2=Z*X,
        asserta(value(z,Z2,8)).

edge(8,7).

edge(7,9) :- value(pp,Path,_,_),
        value(p,P,_),
        Path2=Path+(not P<>0),
        asserta(value(pp,Path2,7,9)).

node(9).

edge(10,11) :- value(pp,Path,_,_),
        value(y,Y,_),
        Path2=Path+(Y<0),
        asserta(value(pp,Path2,10,11)).

node(11) :- value(z,Z,_),
        Z2=1/Z,
        asserta(value(z,Z2,11)).

node(12).
node(13).
```

Figura 3. Cláusulas Prolog geradas pelo tradutor para a execução simbólica.

A partir da base de conhecimento inicial sobre o programa (Figura 2), são geradas as cláusulas para a execução simbólica (Figura 3). Da execução simbólica, tem-se como resultado predicados de caminho e contextos de variáveis (valores simbólicos, em pontos de decisão do programa - Figura 4). A geração das cláusulas de execução simbólica é dependente dos tipos dos nodos. Caso um nodo seja do tipo seqüência, então a cabeça da regra será *node(NoNodo)*. Caso contrário, os nodos seletores (com dois ou mais arcos originando-se do nodo), a cabeça da regra será *edge(Nodo1,Nodo2)*. O corpo da regra é construído pesquisando-se os fatos correspondentes ao nodo ou arco da cabeça da regra, na ordem especificada pelo argumento *Ordem*, extraindo, então, informações sobre o predicado de caminho e o contexto das variáveis. Assim, dependendo do tipo do fato consultado na base, as seguintes cláusulas são geradas:

- varDefUse(potencia,1,1,x,def,input) → asserta(valor(x,x,1)).
- varDefUse(potencia,4,1,y,uso,[]) → valor(y,Y,_).
- varPUse(potencia,3,4,y,y>0) → valor(pp,Caminho,_,_), valor(y,Y,_), Caminho2 = Caminho + (Y>0), asserta(valor(pp,Caminho2,3,4).

Os fatos do tipo *value(Var,Valor,Nodo)* e *value(Var,Valor,Nodo1,Nodo2)* armazenam o *Valor* da variável *Var*, em relação ao *Nodo* ou em relação ao arco (*Nodo1,Nodo2*). O único valor de *Var* associado a um arco, no momento de execução simbólica, é *pp*, que armazena o predicado de caminho acumulado até o ponto considerado. Seu valor inicial é *emptyPath*, e posteriormente, vai assumindo o valor da concatenação dos predicados que vão sendo executados.

As variáveis de entrada, sejam elas parâmetros formais, ou valores fornecidos pelo usuário, são as que aparecem nos predicados de caminho resultantes. Todas as outras variáveis intermediárias têm seu conteúdo computado com base nas variáveis de entrada (o conteúdo é armazenado em fatos).

O acesso às regras é realizado a partir de caminhos selecionados por um critério. A partir de um caminho, que é representado por uma lista de nodos, são consultados os tipos dos nodos, nas cláusulas nodo/2, da base inicial. Caso o tipo seja *begin, end, sequencial, repeat, ou vazio*, então será executada a cláusula com a cabeça *node(Nodo)*, onde *Nodo* representa o número do nodo em questão. Caso o tipo do nodo seja *if_then, if_then_else, for, until, while* ou *case*, então deverá ser executada a cláusula com a cabeça *edge(Nodo1,Nodo2)*, pois trata-se de um predicado.

```
[1,2,3,5,6,7,8,7,8,7,9,10,11,12,13]   /* Lista com o caminho executado simbolicamente.   */
value(pp,emptyPath,1,0).              /* Inicialização do predicado de caminho "pp".     */
value(x,x,1).                         /* Inicialização das variáveis de entrada "x" e "y". */
value(y,y,1).
value(p,0,2).                         /* Variável "p" tem o valor 0 no nodo 2.           */
value(z,0,2).
value(pp,emptyPath + (not y > 0),3,5). /* "pp" recebe a concatenação de "(not y>0)" no arco (3,5). */
value(p,- y,5).
value(z,1,6).
value(pp,emptyPath + (not y > 0) + (- y < 0),7,8).
value(p,- y - 1,8).
value(z,1 * x,8).
value(pp,emptyPath + (not y > 0) + (- y <> 0) + (- y - 1 <> 0),7,8).
value(p,- y - 1 - 1,8).
value(z,1 * x * x,8).
value(pp,emptyPath + (not y > 0) + (- y <> 0) + (- y - 1 <> 0) + (not - y - 1 - 1 <> 0),7,9).
value(pp,emptyPath + (not y > 0) + (- y <> 0) + (- y - 1 <> 0) + (not - y - 1 - 1 <> 0) + (y < 0),10,11).
value(z,1 / (1 * x * x),11).
```

Figura 4. Resultados produzidos pela execução simbólica do caminho [1,2,3,5,6,7,8,7,8,7,9,10,11,12,13].

5. Proposta de Estratégia para a Geração de Dados

A estratégia de geração de dados aqui proposta é realizada de forma iterativa, tendo como principais tarefas: teste do programa com dados reais, análise de cobertura do caminho executado, execução simbólica, simplificação e determinação do intervalo do domínio em relação ao predicado de caminho. Este processo é realizado através das seguintes etapas, apresentadas (de forma diagramática) nas Figuras 5 e 6:

1) um conjunto de subcaminhos do programa é selecionado, de acordo com o critério escolhido;
2) o programa é executado, primeiramente, com dados reais (execução real);
3) a análise de cobertura é realizada, a fim de detectar quais subcaminhos selecionados pelo critério não foram executados;
4) o caminho executado no passo 2 é executado simbolicamente;
5) a primeira avaliação simbólica inicia no primeiro nodo do programa, e vai até o último nodo do caminho sendo considerado. Nas próximas iterações, a execução inicia do ponto aonde a execução simbólica parou, quando estava avaliando um caminho que inclui parte (pelo menos o primeiro nodo) do subcaminho em questão;
6) todas as variáveis (que sempre são variáveis de entrada), com exceção de uma, do predicado de caminho resultante, são substituídas pelos valores que foram atribuídos a elas na execução real anterior, e o predicado de caminho é avaliado, para a determinação de limites do domínio de entrada da variável livre do predicado, a fim de forçar a execução daquele subcaminho;
7) um valor dentro deste limite é escolhido para a variável livre, e o programa é executado com um novo conjunto de dados;
8) o processo retorna à etapa 3, até que todos os subcaminhos praticáveis sejam executados.

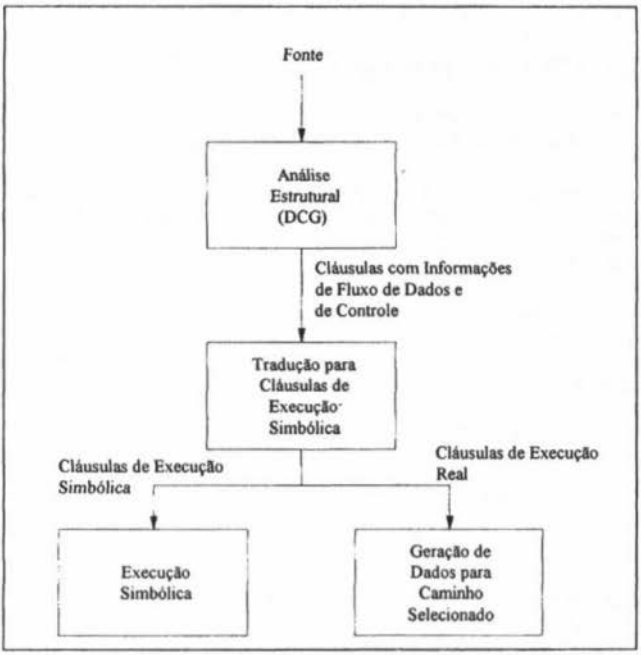

Figura 5. Construção do ambiente para geração de dados de teste.

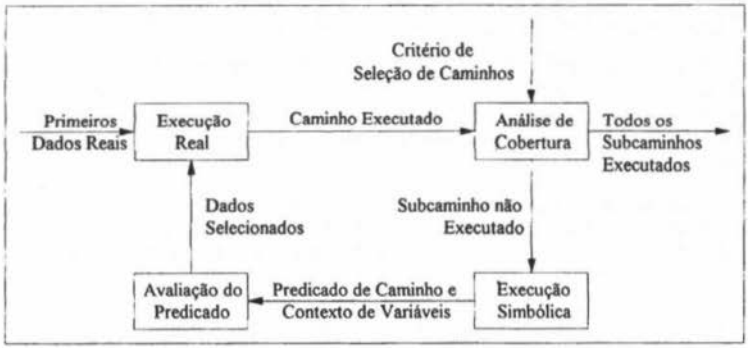

Figura 6. Geração dos dados para teste utilizando execução simbólica e real.

O armazenamento de contextos de variáveis e predicados de caminho intermediários tem o objetivo de reaproveitar segmentos de caminhos já executados simbolicamente. A escolha de dados reais, seja para o teste do programa, para as variáveis livres ou para o predicado de caminho, é direcionada por heurísticas tais como as propostas por Cross e Korel [CRO91] e [KOR90]. Por exemplo, a variável livre a ser escolhida no predicado de caminho deve ser aquela que aparece em menor número de vezes no predicado, já que esta, teoricamente, tem maior possibilidade de garantir que a parte do subcaminho executada anteriormente será repetida, com o novo conjunto de dados escolhido.

5.1. Exemplo de Aplicação da Metodologia

A seguir, é apresentado um exemplo da aplicação da metodologia de geração de dados de teste proposta, utilizando, para tal, a rotina "potencia", apresentada na Figura 1.

O critério escolhido para a seleção de caminhos é o todos-usos [RAP85], o qual selecionou os subcaminhos abaixo, representados no formato de listas Prolog. Subcaminhos contidos em outros foram considerados, já que, apesar de que a cobertura dos mesmos possa ocorrer executando os subcaminhos que os contêm, estes subcaminhos maiores podem não ser executáveis.

[1,2,3,4,6,7,8],
[1,2,3,4],
[1,2,3,5],
[1,2,3,4,6,7,8,7,9,10,11],
[1,2,3,4,6,7,8,7,9,10,12],
[4,6,7,8],
[4,6,7,9],
[6,7,8],
[6,7,9,10,11],
[8,7,8] e
[8,7,9,10,11].

Como dados iniciais de entrada, para a execução real, utilizou-se x=3 e y=1. Com este conjunto de dados, o programa executa o caminho [1,2,3,4,6,7,8,7,9,10,12,13]. Realizando-se uma análise de cobertura, verifica-se que os subcaminhos [1,2,3,5], [1,2,3,4,6,7,8,7,9,10,11], [4,6,7,9], [6,7,9,10,11], [8,7,8] e [8,7,9,10,11] não foram executados. O caminho [1,2,3,4,6,7,8,7,9,10,11] tem a

seqüência dos primeiros nodos em comum com o caminho executado. Verifica-se que a partir da execução do nodo 10, houve um desvio de execução. É, então, realizada a execução simbólica deste caminho, tendo-se como predicado de caminho resultante a expressão: (y>0) and (y<>0) and (y<0). Como as condições atômicas (y>0) e (y<0) são conflitantes, pode-se concluir que este caminho não é executável.

Analisando-se agora o subcaminho: [1,2,3,5], nota-se que este tem em comum com o caminho executado os três primeiros nodos, sendo o predicado do nodo 3 o responsável pelo desvio da execução. Executando simbolicamente o subcaminho [1,2,3,5] (aproveitando a parte inicial já executada anteriormente), tem-se o predicado de caminho resultante (y<=0). É, então, selecionado um valor para y que satisfaça esta condição: y=0. O programa é executado com x=3 e y=0, executando o caminho [1,2,3,5,6,7,9,10,12,13]. Com isso, o subcaminho [1,2,3,5] é executado. Os subcaminhos restantes são analisados de maneira similar aos apresentados anteriormente (através de execuções simbólicas e reais).

6. Considerações Finais

O presente artigo propõe uma estratégia para a geração automática de dados para o teste estrutural, a qual está sendo implementada no sistema LOGTEST, ambiente de validação de software, desenvolvido em Prolog.

A aplicação da programação em lógica na construção de LOGTEST permitiu que o tempo de desenvolvimento fosse bastante reduzido, em relação à utilização de uma linguagem procedimental. Possibilitou também que fosse utilizado um alto nível de abstração, já que a linguagem permite especificar os problemas sem que precisem ser fornecidas soluções para estes problemas. A base de conhecimento gerada permite uma maior economia de espaço de armazenamento. A facilidade de configuração de uma nova linguagem de reconhecimento dos programas também existe, já que o processo de análise estrutural foi realizado de uma maneira bem simples, utilizando a DCG e algumas cláusulas Prolog.

A estratégia de geração de dados proposta combina execução simbólica e a execução de dados reais, e tem a grande vantagem de utilizar um conjunto de fatos e regras relativamente compacto para sua realização. Com LOGTEST, todas estas informações estão disponíveis a partir da base de conhecimento e do estabelecimento de relações entre os dados que ali constam.

Existem algumas discussões na literatura sobre a função da inteligência e do conhecimento na construção de ambientes de engenharia de software. Nos anos 70 e 80 acreditava-se que a aplicação de um sistema de Inteligência Artificial suficientemente poderoso à área resolveria completamente os problemas. Entretanto, mais recentemente, observou-se que o conhecimento é tão importante ou até mais importante do que a inteligência [RIC88]. A característica principal da estratégia aqui proposta é o gerenciamento do conhecimento obtido a partir das análises dinâmica e simbólica do programa.

A geração automática de dados é uma necessidade no teste estrutural. Com a implementação desta estratégia, constatou-se que a simples escolha de dados para executar caminhos sugeridos pelos critérios não é suficiente. Deve-se combinar alguma estratégia do teste funcional, como a divisão do domínio de entrada em partições de equivalência, análise de limites do domínio, por exemplo [MYE79]. A divisão dos dados de entrada em partições, onde cada dado representa o comportamento de todos os outros dados de sua partição fornece maior confiabilidade aos testes realizados [VER93].

Os seguintes aspectos negativos e/ou limitações foram identificados, decorrentes do uso de Prolog na construção do ambiente LOGTEST:

- não foi ainda criada uma interface unificada para o ambiente, exigindo que o usuário tenha que lembrar corretamente dos nomes das cláusulas que iniciam cada parte do processo;

- apenas rotinas únicas estão sendo analisadas pelo ambiente. Relações interprocedimentais ainda não estão sendo consideradas;
- o desempenho do sistema não foi otimizado, tendo um tempo de resposta não imediato à medida em que a base ou os caminhos executados aumentam de tamanho;
- quando o interpretador Prolog responde "*no*", existem duas possibilidades: ou a consulta tem realmente como resposta um "não", ou o usuário esqueceu um argumento ou digitou alguma letra errada no nome do predicado.

Como primeira extensão ao LOGTEST pode ser citada a construção de um provador de teoremas e de um solucionador de equações, necessários para a completa automação da estratégia de geração de dados apresentada. Além disso, pretende-se estender o ambiente para o teste de sistemas constituídos por mais de um módulo. Pretende-se, também, explorar mais o paradigma da programação em lógica, verificando a viabilidade de adicionar ao ambiente as seguintes funcionalidades: verificação formal do programa (através do uso de assertivas, por exemplo), aprendizagem de padrões dos programas analisados (o que deverá exigir maiores estudos e esforços para a definição do processo). Um experimento controlado também constitui-se em uma interessante extensão para este trabalho, comparando-o, empiricamente, com outros trabalhos da área.

7. Referências Bibliográficas

[BEN93] BENCH-CAPON, T. et al. Two Aspectos of the Validation and Verification of Knowledge-Based Systems. **IEEE Expert, no. 6**. Jun. 1993.

[CAN92] CANFORA, G. et al. A Logic-Based Approach to Reverse Engineering Tools Production. **IEEE Transactions on Software Engineering, vol. 18(12)**. Dec. 1992.

[COE91] COEN-PORISINI, A. et al. Software Specialization Via Symbolic Execution. **IEEE Transactions on Software Engineering, vol. 17(9)**. Setp. 1991.

[CRO91] CROSS, J. H. et al. Expert System Assisted Test Data Generation for Software Branch Coverage. **Data & Knowledge Engineering, no. 6**. Jun., 1991.

[DAR78] DARRINGER, J. A.; KING, J. C. Applications of Symbolic Execution to Program Testing. **Tutorial Software Methodology**. C. V. Ranamoortht & R. T. Yeh. IEEE Catalog. EHO. 1978.

[DEM87] DeMILLO, R. A. et al. **Software Testing and Evaluation**. The Benjamin/Cummings Publishing Company, Inc. 1987.

[FRA87] FRANKL, P. G.; WEYUKER, E. J. A Data Flow Testing Tool. **Proceedings SoftFairII, Software Development Tools, Tecniques and Alternatives**. San Francisco, 1987. Pp. 46-53.

[GOL94] GOLDBERG, A. et al. Applications of Feasible Path Analysis to Program Testing. **Proceedings of ISSTA 94**. Seattle, USA. 1994.

[HAM95] HAMLET, D. Implementing Prototype Testing Tools. **Software Practice and Experience, vol. 25(4)**. April, 1995.

[HOF91] HOFFMAN, D. M.; STROOPER, P. Automated Module Testing in Prolog. **IEEE Transactions on Software Engineering, vol. 17(9)**. Sept. 1991.

[HOL87] HOLBROOK, H. B. & THEBAUT, S. M. A Survey of Software Maintenance Tools that enhance Program Understanding. **SERC-TR-9-F**. Software Engineering Research Center. Univ. Florida/Purdue Univ., 1987.

[HOR91] HORGAN, J. R.; LONDON, S. Data Flow Coverage and the C Language. **Proceedings Symposium on Software Testing, Analysis and Verification (TAV4)**. Victoria, BC. Oct. 1991. Pp. 87-97.

[HOW78] HOWDEN, W. E. DISSECT - A Symbolic Evaluation and Program Testing System. **IEEE Transactions on Software Engineering, vol. 4(1)**. Jan. 1978.

[JAF93] JAFAR, M. & BAHILL, A. T. Interactive Verification of Knowledge-Based Systems. **IEEE Expert, no. 2**. Feb. 1993.

[KHA91] KHANNA, S. Logic Programming for Software Verification and Testing. **The Computer Journal, vol. 34(4)**. Jul. 1991.

[KOR90] KOREL, B. Automated Software Test Data Generation. **IEEE Transactions on Software Engineering, vol. 6(8)**. Aug. 1990.

[MAL92] MALDONADO, J. C. et al. Critérios Potenciais Usos: Análise da Aplicação de um Benchmark. **In: SIMPÓSIO DE ENGENHARIA DE SOFTWARE, 6., 1992**. Gramado, 1992.

[MEN93] MENGSHOEL, O. J. & DELAB, S. Knowledge Validation: Principles and Practice. **IEEE Expert, no. 6**. Jun. 1993.

[PAL96] PALAVRO, I. Depuração de Software. **Projeto de Diplomação**. II/UFRGS. Porto Alegre, RS. 1996.

[PRE95] PRESSMAN, R. **Engenharia de Software**. Editora McCrow Books. 1995.

[RAP85] RAPPS, S.; WEYUKER, E. Selecting Software Test Data Using Data Flow Information. **IEEE Transactions on Software Engineering, vol. 11(4)**. Apr. 1985.

[RIC88] Rich, C.; Waters, R. C. Automatic Programming: Myths and Prospects. **IEEE Computer**, Aug. 1988.

[ROM96] ROMBALDI, V. Heurísticas para Geração de Dados de Teste. **Dissertação de Mestrado**. CPGCC/UFRGS. Porto Alegre, RS. Maio 1996.

[SIL94] SILVA, Juliana B. da; PRICE, A. M. de A. Métrica de Complexidade de Software baseada em Critério de Seleção de Caminhos de Teste. In: Anais do **VIII Simpósio Brasileiro de Engenharia de Software**, p. 471-485. Curitiba, PR. Out. 1994.

[SIL95] SILVA, Juliana B. da PROTESTE+: Ambiente de Validação Automática de Qualidade de SW através de Técnicas de Teste e de Métricas de Complexidade. **Dissertação de Mestrado**. II/UFRGS. Porto Alegre, RS. Fev. 1995.

[SIL96] SILVA, C. R. da. Slices no Processo de Teste Estrutural de Software. **Projeto de Diplomação**. II/UFRGS. 1996.

[TSE94] TSE, T. et al. The Application of Prolog to Structured Design. **Software Practice and Experience, vol. 24**. 1994.

[VER93] VERGÍLIO, S. R. at al. Uma Estratégia para Geração de Dados do Teste. **VII Simpósio Brasileiro de Engenharia de Software**. P. 306-319. Out. 1993.

[VER94] VERGÍLIO, S. et al. Caminhos não-executáveis no Teste de Integração: Caracterização, Previsão e Determinação. **In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 8., 1994**. Curitiba, 1994.