Geração Automática de Cenários de Teste a partir de Gramática Livre do Contexto

Juliana Silva Herbert
e-mail: juliana@inf.ufrgs.br

Ana Maria de Alencar Price
e-mail: anaprice@inf.ufrgs.br

Instituto de Informática
Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500. Bloco IV. Porto Alegre, RS.

RESUMO

Teste de software constitui-se a estratégia mais utilizada para a validação de um programa, de acordo com a sua especificação. Somente através de um procedimento sistemático, pode-se garantir a eficácia desta etapa do ciclo de desenvolvimento. Para a realização de testes sistemáticos, desenvolveu-se o ambiente PROTESTE+, um sistema de validação automática de qualidade de software através de técnicas de teste e de métricas de complexidade. Este ambiente é configurável para diferentes linguagens de programação imperativas através da especificação de gramáticas. Testes de unidade e de subsistema (integração) utilizam módulos especiais denominados *drivers* e *stubs*, que representam um esforço adicional na fase de teste, já que estes têm que ser codificados. Para facilitar a geração destes módulos, e torná-la independente de linguagem seguindo o propósito do ambiente, desenvolveu-se uma gramática livre do contexto estendida, que gera automaticamente *drivers* e *stubs*. Esta geração foi validada de forma prática verificando-se a simplicidade da gramática, devido ao reduzido número de elementos utilizado na extensão da gramática original.

ABSTRACT

Software testing is the most used strategy to validate a program according to its specification. Only through software systematic testing, one can guarantee this life cycle phase effectiveness. To fulfil systematic testing, it has been developed the system PROTESTE+, an automatic environment for evaluating software quality through testing tecniques and complexity metrics. This environment is configurable to different imperative programming languages, through grammars specifications. Unit and subsystem (integration) testing uses special modules called drivers and stubs, which represent an overhead on the testing task, because they have to be coded. To turn it into an automatic task and language independent, as the environment's proposal, it has been developed an extended context free grammar, which generates drivers and stubs automatically. This generation has been validated in a practical way. The resulted grammar is very simple because it requires only small number of new constructions to be added to the original grammar.

1. Introdução

Teste é a técnica que tem sido mais utilizada na validação do software [FAI85]. Entretanto, o objetivo de garantir a confiabilidade do software somente é atingido se o teste é realizado de forma sistemática, utilizando-se, preferencialmente, uma ferramenta para a automatização do teste. Apesar desta constatação, as estratégias mais utilizadas são o teste por exaustão e o teste para um subconjunto do domínio dos dados de entrada, técnicas estas reconhecidamente incompletas [MYE79] [MOS93].

Por outro lado, constata-se que cerca de 60% dos custos e tempo dispendidos no desenvolvimento do software estão concentrados nesta fase e na manutenção [DEM87] [MOS93]. Uma das causas desta concentração de recursos é o fato de a maior parte das ferramentas CASE (*Computer Aided Software Engineering*) existentes no mercado ser dedicada às etapas de análise, projeto e codificação do sistema, fazendo com que as fases finais do ciclo de desenvolvimento sejam realizadas sem a utilização de técnicas avançadas e automatizadas [DAL94].

Os métodos de teste sistemático podem ser classificados em três grandes grupos: teste estrutural ("caixa-aberta"), teste funcional ("caixa-preta") e a análise de mutantes. No teste estrutural, sobre o qual trata este trabalho, a fase inicial compreende a análise minuciosa do código fonte do programa, a fim de obter informações relacionadas aos fluxos de controle e de dados, bem como a seleção de casos de teste [MAL92] [RAP85]. Esta análise, devido à quantidade de características consideradas e ao tamanho do programa, se realizada manualmente, é bastante propícia a erros, além de não se constituir em uma tarefa intelectualmente agradável. Desta forma, é evidente a necessidade da utilização de ferramentas que automaticamente orientem o processo de teste do sistema, tornando-o mais confiável e rápido [SIL94] [SIL95].

Com esta motivação, foi desenvolvido nesta instituição o PROTESTE+, um ambiente de validação automática de qualidade de software através da utilização de técnicas de teste e métricas de complexidade [SIL95]. Este ambiente implementa as fases de teste de módulo, subsistema (integração) e de sistema, sendo configurável para diferentes linguagens de programação imperativas, a partir da especificação da BNF (*Backus Naur Form*) da linguagem. Para a realização do teste de módulo e do teste de subsistema, é necessária a simulação dos outros módulos do sistema, para possibilitar a sua execução. Esta simulação é realizada utilizando-se módulos *drivers* e *stubs*, que constituem o cenário para a execução dos testes.

Conservando o aspecto de configuração para diferentes linguagens de programação imperativas, sentiu-se a necessidade de definir uma maneira através da qual estes módulos fossem gerados automaticamente. Optou-se por gerá-los a partir da especificação de uma BNF adaptada, com diretivas e guardas que tornam a geração mais direta e simples.

O presente trabalho tem o objetivo de apresentar a gramática anteriormente citada, bem como descrever a sua implementação. São enfatizados aspectos relacionados às estruturas de dados, como informações provenientes da análise do código fonte, bem como à tradução da gramática e à geração dos módulos a partir desta, considerando o fluxo de dados modular e interprocedimental.

2. Teste Estrutural de Software

O teste estrutural é realizado com base na estrutura interna do programa. É dependente de implementação e analisa aspectos relacionados aos fluxos de controle e de dados de um módulo, subsistema ou sistema. É normalmente utilizada a análise de cobertura de caminhos, baseada em critérios [MAL92] [PRI90] [RAP85] [SIL94].

Os critérios de cobertura (ou de seleção de casos de teste) são condições que devem ser satisfeitas pelo teste. Um critério é válido se a execução de pelo menos um dos casos de teste detectar erros no programa. É ideal se fornecer um conjunto de casos de teste que detectem todos os erros que o critério se propõe a detectar [WEY80].

No teste de módulo, os critérios de cobertura fundamentam-se na seleção de um conjunto C de caminhos no Grafo de Fluxo de controle (GFC). O módulo original é isolado do resto do sistema, criando-se um cenário para a sua execução. Para tal, o módulo chamador deve ser substituído pelo *driver* e os módulos chamados por *stubs*. O módulo sendo testado é instrumentado, inserindo-se comandos que indiquem quais partes foram executadas em determinado momento. A partir do *trace* de execução, são verificados quais os caminhos do conjunto C foram executados. Novas execuções são então realizadas, a fim de executar todos os elementos de C.

Nos testes de subsistema e de sistema, os critérios selecionam caminhos no diagrama de chamadas, baseando-se na análise interprocedimental. No teste de subsistema, novamente um cenário para a execução deve ser criado, já que apenas alguns dos módulos do sistema original

estão sendo utilizados. A necessidade da quantidade de *drivers* e *stubs* envolvidos está relacionada diretamente à abordagem de integração utilizada (p.ex. *top-down, bottom-up*) [MYE79].

Os *drivers* utilizados devem simular o módulo principal, contendo comandos de inicialização de variáveis globais ao sistema ou parâmetros reais dos módulos testados, além da chamada a estes módulos. Os *stubs* simulam rotinas subordinadas à interface dos módulos testados, podendo realizar simples manipulações de dados, imprimir alguma verificação de entrada, e retornar o controle ao módulo chamador com o retorno de valores, quando assim for necessário. A geração de *drivers* e *stubs* dependente de linguagem é descrita em [LIM90].

A criação de cenários para a realização dos testes de módulo e de subsistema tem como resultado um sistema compilável e executável, com um menor escopo de análise do que o original, facilitando o trabalho de depuração de erros. Entretanto, a codificação de módulos *drivers* e *stubs* representa uma tarefa custosa adicional ao teste, além de ter a complexidade aumentada por ser baseada na análise de grande quantidade de informações. Por isso torna-se bastante importante que esta codificação seja realizada de forma automática, disponibilizando estes módulos em quaisquer momentos convenientes ao programador.

3. Ambiente PROTESTE+

PROTESTE+ é um ambiente interativo para apoio ao teste estrutural de software. O objetivo deste ambiente é auxiliar o programador no planejamento dos testes, nas tarefas de seleção de caminhos e de dados, e na execução, que envolve a realização dos testes com instrumentação e análise dos resultados [PRI90] [SIL94] [SIL95]. O teste pode ser realizado para módulos, subsistemas ou sistemas. Para isso, PROTESTE+ implementa onze critérios de seleção de casos de teste para módulo (considerando os fluxos de controle e de dados), três critérios para subsistemas ou sistemas, onze métricas de complexidade de módulo (que consideram os fluxos de controle, dados e o tamanho e volume do módulo).

Apesar de o ambiente ser dirigido ao teste estrutural, que é dependente do código fonte do programa, o sistema foi construído tendo uma parte significativa de suas rotinas independente da linguagem de entrada. Desta forma, o PROTESTE+ pode ser visto como um ambiente configurável para diferentes linguagens de programação imperativas. A fim de possibilitar esta independência de linguagem, as rotinas dependentes de linguagem foram automaticamente geradas através do ambiente SINLEX [ROS89], um gerador de compiladores baseado em gramática de

atributos. Assim, a configuração para uma nova linguagem de programação é realizada através da inserção de atributos e ações semânticas na BNF (*Backus Naur Form*) da linguagem. Esta BNF é submetida à análise do SINLEX que gera analisadores léxico e sintático, executando, quando reconhecidos determinados *tokens*, as ações semânticas correspondentes.

A execução das ações semânticas associadas à BNF tem como resultado tabelas com informações relativas ao código fonte, que são utilizadas pelas rotinas independentes de linguagem. A arquitetura geral do PROTESTE+ é apresentada na figura 1.

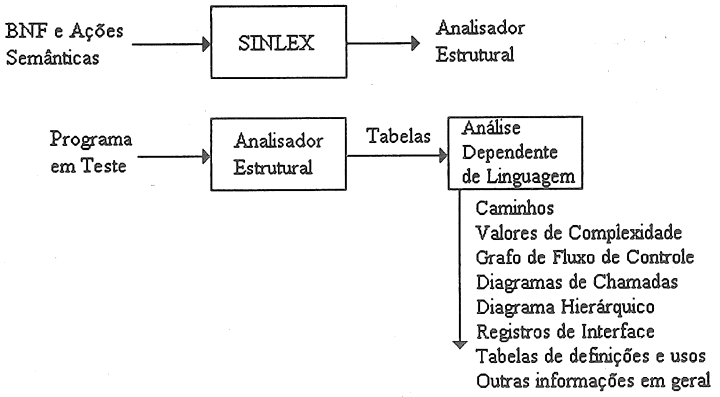

Figura 1. Arquitetura geral do ambiente PROTESTE+.

Devido ao objetivo do ambiente de permitir a configuração para diferentes linguagens de programação, projetou-se a construção de cenários para o teste de módulos e de subsistemas de modo a ser realizada também de forma automática e independente de linguagem de programação. Decidiu-se fazer esta geração a partir de uma gramática livre do contexto adaptada, já que esta seria um subconjunto estendido por um reduzido número de novos elementos, da BNF original da linguagem, não exigindo significativos conhecimentos adicionais do usuário configurador.

4. Informações para a Geração de *Drivers* e *Stubs*

Para a geração dos *drivers* e *stubs*, além da estrutura da linguagem especificada pela gramática, são necessárias informações extraídas diretamente do código do sistema. A fim de

obter estas informações (e outras utilizadas especificamente para critérios de seleção de casos de teste e métricas de complexidade), é realizada a análise sintática do código, pelo analisador estrutural, que contém ações semânticas associadas para a construção de tabelas. Estas tabelas são então utilizadas pelas rotinas independentes de linguagem (figura 1). Para cada módulo, é derivado um registro de interface, que é apresentado de forma mais detalhada na figura 2.

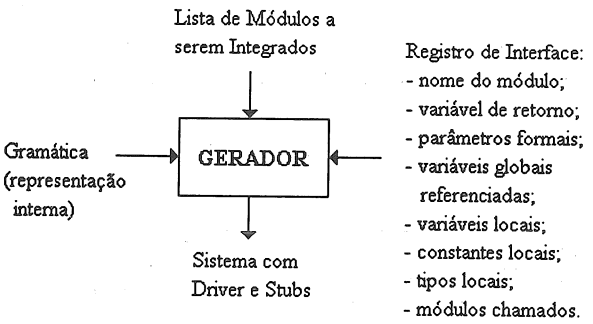

Figura 2. Geração de módulos *drivers* e *stubs* a partir da gramática e do registro de interface.

Além das informações extraídas do código do sistema em teste, o usuário deve fornecer a lista de módulos a serem integrados. Se for adotada a abordagem *top-down* ou *bottom-up*, esta lista é sugerida pelo PROTESTE+. A partir desta lista, o ambiente determina quais as variáveis globais e parâmetros devem ser inicializados, bem como quais os módulos que devem ser chamados pelo *driver* ou módulo principal em questão. Estas variáveis são apresentadas ao usuário, que fornece os valores de inicialização.

5. Gramática para Geração de *Drivers* e *Stubs*

Uma gramática pode ser vista como um mecanismo formal para especificar de forma finita uma linguagem potencialmente infinita [HOP79]. A análise de um programa, de acordo com uma determinada linguagem de programação, é realizada basicamente em três fases: análise léxica, sintática e semântica. A análise sintática pode ser concebida de forma a ser um processo coordenador, acionando o analisador léxico quando saber o próximo *token* seja necessário, e chamando funções semânticas correspondentes ao estado atual da análise [ROS89]. Para a estrutura sintática da linguagem, é geralmente utilizada uma gramática livre do contexto, que pode

ser reconhecida por uma máquina de pilha, e por funções semânticas responsáveis pela verificação do contexto.

Uma gramática livre do contexto é definida por regras de produção que possuem um símbolo não-terminal à esquerda e uma seqüência de símbolos terminais e não-terminais à direita [AHO86], podendo ser classificadas em várias categorias (LR, LL, gramática de operadores e outras), de acordo com o tipo de analisador sintático que as suporta. A gramática considerada neste trabalho baseia-se em gramáticas do tipo LL(1) [HOP79]. Este tipo de gramática é fatorada e sem recursividade à esquerda. Para o seu reconhecimento, pode ser realizada a análise *top-down*, que parte do símbolo inicial e tenta construir a árvore de derivação para a sentença pela expansão do símbolo não-terminal mais à esquerda. Esta análise *top-down* é realizada por um analisador descendente recursivo [AHO86]. A gramática não deve ser ambígua, ou seja, a uma mesma sentença, não deve corresponder mais de uma árvore de derivação.

A geração de módulos *drivers* e *stubs* é realizada a partir da especificação de uma gramática livre do contexto estendida. Esta extensão foi necessária para condicionar a geração dos comandos da linguagem especificada, de acordo com o registro de interface. A gramática, construída a partir de um subconjunto da gramática original, conta as produções que geram inicializações, declarações de variáveis e chamadas a módulos. A gramática é composta pelos seguintes quatro conjuntos de símbolos: terminais, não-terminais, diretivas e guardas.

Terminais podem ser palavras reservadas ou delimitadores, entre aspas. Ex.: "program", ";". Os não-terminais ***driver*** e ***stub*** são símbolos iniciais e definem o início da geração dos módulos correspondentes. As diretivas são representadas entre os sinais < e > e executam funções pré-definidas. Podem ser consideradas como ações semânticas associadas às ações sintáticas, e suas funções pré-definidas realizam tarefas como recuperação de informações do registro de interface (*retorno_função*, *decl_var*, *decl_tipo*, *decl_const*, *tipo_var*, *tipo_tipo*, *tipo_const*, *parâmetros_stub*), definição de identificadores para *drivers* e *stubs* (*nome_driver* e *nome_stub*), e montagem do código fonte na ordem definida pela sintaxe da linguagem (*chamada*, *stubs*, *módulo* e *inicialização*). As chamadas a funções são realizadas de acordo com o estado atual da análise. Posteriormente, é apresentado um exemplo de produção com as diretivas *<decl_const>* e *<tipo_const>*, que buscam o tipo de informação especificado do registro de interface do módulo testado.

As guardas são utilizadas como estruturas de decisão, para definir qual será a expansão de um não-terminal. A estrutura de uma guarda é a seguinte: *#tipo_guarda* (x *op* 0) *% forma sentencial %*, onde *tipo_guarda* pode ser um comando de seleção, do tipo *if*, ou de repetição, do tipo *while*. Uma guarda do tipo *if* é avaliada somente uma vez, enquanto que uma guarda do tipo *while* é avaliada até que a condição associada seja falsa. *forma sentencial* é o trecho da produção que será expandido se a avaliação das guardas resultar em verdadeira. *op* é um operador relacional do tipo >, <, =, ≤ ou ≥. *x* é um dos campos do registro de interface que representam o número de elementos de determinada categoria da interface do módulo (por exemplo, variáveis globais, constantes locais). Exemplificando estes conceitos, seja a seguinte produção da gramática:

DECLARATION_CONST -> #while (NumConstGlob>0) % <decl_const> "=" <tipo_const> %.

Neste exemplo, a guarda *while* testa a quantidade de constantes globais armazenadas no registro de interface a serem declaradas. Enquanto houver constantes, a produção entre os símbolos "%" e "%" será expandida. As diretivas acima especificadas representam que inicialmente, o nome da constante deverá ser buscado no registro de interface, um sinal de "=" será escrito no arquivo e, posteriormente, o tipo da constante, que é o seu valor, será também recuperado do registro de interface.

6. Processo de Geração do Código dos Módulos Driver e Stubs

A análise da gramática para a geração dos *drivers* e *stubs* é realizada somente uma vez, no momento de configuração do ambiente para uma nova linguagem e informações são armazenadas em arquivo para serem futuramente recuperadas. Para o usuário final do PROTESTE+, esta implementação é transparente, sendo apenas necessário indicar quais os módulos para a integração. Para o usuário configurador da linguagem, o conhecimento exigido é o formato da gramática. A partir da lista de módulos a serem integrados, da gramática e dos registros de interface, é gerado um arquivo fonte, para ser compilado e executado (figura 2).

A expansão de um elemento depende do seu tipo: um terminal é simplesmente copiado para o arquivo fonte; um não-terminal é substituído por sua definição; e a expansão de uma diretiva é a execução da função correspondente. A lógica para a geração do código é a seguinte: a partir do símbolo inicial DRIVER, ou STUB, são expandidos todos os símbolos que estão do seu lado direito, gerando um arquivo fonte.

Na geração de *drivers*, a partir da lista de módulos a serem integrados, são consultados os registros de interface correspondentes e quais as funções devem ser chamadas pelo *driver* em questão, pois algum dos módulos sendo integrados pode estar sendo chamado por outro da lista. Posteriormente, as variáveis globais referenciadas em cada módulo da lista são agrupadas, a fim de não repetir declarações no módulo *driver*. Os *stubs* são gerados sem um corpo de comandos específico, contendo apenas, no caso de uma função, o retorno esperado pelo módulo chamado. Tanto os *drivers* como os *stubs* gerados podem ter seu código modificado, a fim de incluir computações necessárias para a aplicação, ou uma rotina de geração de dados aleatórios, por exemplo.

7. Exemplos de Gramáticas e de Código Gerado

A seguir, são apresentadas as gramáticas correspondentes à geração de *drivers* e *stubs* na linguagem Pascal e na linguagem C.

```
DRIVER -> "program" <nome_driver> ";"
          DECLARATIONS
          <stubs> <módulos>
          "begin" CORPO "end" ".".

DECLARATIONS -> # if (NumConstGlob > 0) % "const" DECLARATION_CONST %
                # if (NumTipoGlob > 0)  % "type" DECLARATION_TYPE  %
                # if (NumVarGlob > 0)   % "var" DECLARATION_VAR    %.

DECLARATION_CONST -> # while (NumConstGlob > 0) % <decl_const> "=" <tipo_const> %.

DECLARATION_TIPO -> # while (NumTipoGlob > 0) % <decl_tipo> "=" <tipo_tipo> %.

DECLARATION_VAR -> # while (NumVarGlob > 0) % <decl_var> "=" <tipo_var> %.

CORPO -> # while (numVarInit > 0) % <inicialização> ";" %
           <chamada> ";" CORPO_.

CORPO_ -> # while (numChamadas > 0) % <chamada> ";" %.

STUB -> # if (eh_Function = 0)  % "procedure" %
        # if (eh_Function <> 0) % "function" %
        <nome_stub>
        # if (numParametros > 0) % "(" <parametros_stub> ")" %
        # if (eh_Function <> 0)  % ":" <retorno_função> %
        ";" CORPO_STUB.

CORPO_STUB -> "begin"  #if (eh_Function <> 0) % <nome_stub> ":" "=" <valor_retorno> %
"end".
```

Para a linguagem C:

```
DRIVER -> DECLARATIONS
          <stubs>
          <módulos>
          "void main()"
          "{" CORPO "}".

DECLARATIONS -> # if (NumConstGlob > 0) % DECLARATION_CONST %
                # if (NumTipoGlob > 0)  % DECLARATION_TYPE  %
                # if (NumVarGlob > 0)   % DECLARATION_VAR   %.

DECLARATION_CONST -> # while (NumConstGlob > 0) % "#define" <decl_const> "=" <tipo_const> %.

DECLARATION_TIPO -> # while (NumTipoGlob > 0) % "typedef" <decl_tipo> "=" <tipo_tipo> %.

DECLARATION_VAR -> # while (NumVarGlob > 0) % <tipo_var> <decl_var> %.

CORPO -> # while (numVarInit > 0) % <inicialização> ";" %
         <chamada> ";" CORPO_.

CORPO_ -> # while (numChamadas > 0) % <chamada> ";" %.

STUB -> # if (eh_Function = 0)  % "void" %
        # if (eh_Function <> 0) % <retorno_função> %
        <nome_stub>
        # if (numParametros > 0) % "(" <parametros_stub> ")" %
        CORPO_STUB.

CORPO_STUB -> "{"
              #if (eh_Function <> 0) % "return" <valor_retorno> %
              "}".
```

A partir da gramática para a geração de módulos *drivers* e *stubs* na linguagem Pascal, apresentada anteriormente, é apresentado um exemplo de sistema, onde os módulos A e B são integrados inicialmente.

No exemplo acima, a variável global *cGlob* não foi declarada nem inicializada, pois nenhum dos módulos sendo integrados a utilizam. O *stub* D_st possui um comando de inicialização em seu interior, pois trata-se de uma função. As inicializações de variáveis globais e do retorno da função devem se fornecidas pelo usuário por arquivo, ou por interação direta, de acordo com a implementação do PROTESTE+.

```
program exemplo;
var
  aGlob, bGlob, cGlob: byte;

procedure B;
begin
  bGlob := bGlob + 1;
end;

procedure C(paramC: byte);
begin
  cGlob := paramC + 1;
end;

function D: byte;
begin
  D := 2;
end;

procedure A;
var
  a1Loc, a2Loc: byte;
begin
  C(a1Loc);
  a2Loc := D + aGlob;
end;

begin
  readln(aGlob,bGlob,cGlob);
  A;
  B;
end.
```

Sistema com *driver* e *stubs*:

```
program exemplo_drv;
var
  aGlob, bGlob: byte;

procedure B;
begin
  bGlob := bGlob + 1;
end;

procedure C_st(paramC: byte);
begin
end;

function D_st: byte;
begin
  D_st :={valor forn. pelo us. };
end;

procedure A;
var
  a1Loc, a2Loc: byte;
begin
  C_st(a1Loc);
  a2Loc := D_st + aGlob;
end;

begin
  aGlob:={valor forn. pelo us. };
  bGlob:={valor forn. pelo us. };
  A;
  B;
end.
```

8. Conclusões

A aplicação do teste estrutural sistemático somente é viável se utilizada uma ferramenta que o suporte, devido à grande quantidade de informações que devem ser gerenciadas e à natureza dos processos envolvidos. Como a construção de uma ferramenta envolve um grande esforço relacionado ao tempo e aos custos, este processo deve ser realizado de forma a torná-la o mais genérica possível. Por este motivo, o PROTESTE+ foi concebido como um ambiente configurável para diferentes linguagens de programação imperativas.

No teste de unidade ou de subsistema, devem ser utilizados *drivers* e *stubs* a fim de simular o comportamento dos componentes do sistema que interagem com o(s) módulo(s) em teste, sem que a atenção seja dispersada para estes componentes. A gramática apresentada foi definida com

este propósito. Sua sintaxe é simples, apesar de ser uma representação formal do subconjunto de comandos da linguagem. O subconjunto da linguagem a ser representado é bastante reduzido, e tem-se a vantagem de que este já foi definido pelo usuário configurador, quando a gramática completa foi submetida ao PROTESTE+, a fim de configurá-lo para uma nova linguagem.

Outro aspecto positivo observado, é que o conhecimento exigido do usuário configurador relacionado às estruturas de dados se limita ao conhecimento de alguns campos do registro de interface. Acredita-se que com uma documentação clara e objetiva, este conhecimento possa ser utilizado de forma simples. A inicialização de variáveis globais e/ou passadas como parâmetros é direcionada, de forma a evitar que o usuário tenha que identificar estas variáveis manualmente.

A gramática foi implementada para o PROTESTE+, na versão Pascal. Os resultados obtidos permitiram que fossem estudadas técnicas de teste e métricas de complexidade para módulos, de forma bastante prática e eficiente.

9. Referências Bibliográficas

[AHO86] AHO, A. & ULLMAN, J. **Principles of Compiler Design**. Addison-Welley. California, 1986.

[DAL94] DALAL, S. R. & McINTOSH, A. A. When to Stop Testing for Large Software Systems with Changing Code. **IEEE Transactions on Software Engineering, vol. 20(4)**. Apr. 1994. Pp. 93-104.

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

[FAI85] FAIRLEY, R. **Software Engineering Concepts**. McGraw-Hill. New York. 1985.

[HOP79] HOPCROFT, J. E. & ULLMAN, J. D. **Introduction to Automata Theory, Languages, and Computation**. Addison-Welley. Massachussets. 1979.

[LIM90] LIMA, V. L. F. **Software para Teste de Módulos com a Geração de Drivers e Stubs**. Projeto de Diplomação. Instituto de Informática, UFRGS. Julho, 1990.

[MAL92] MALDONADO, J. C. et al. Critérios Potenciais Usos: Análise da Aplicação de um Benchmark. **VI Simpósio de Engenharia de Software**. Nov. 1992. Gramado, RS. Pp. 357-371.

[MOS93] MOSLEY, D. J. **The Handbook of MIS Application Software Testing**. Yourdon Press Computing Series. Prentice-Hall, Inc. New Jersey. 1993.

[MYE79] MYERS, G. J. **The Art of Software Testing**. John Wiley & Sons. New York. 1979.

[PRI90] PRICE, A. M. A. Ambiente de Apoio ao Teste Estrutural de Programas. **IV Simpósio de Engenharia de Software**. Águas de São Pedro, SP. Out. 1990.

[RAP85] RAPPS, S. & WEYUKER, E. Selecting Software Test Data Using Data Flow Information. **IEEE Transactions on Software Engineering, vol. SE-11(4)**. Apr. 1985. Pp. 367-375.

[ROS89] ROSA, F. R. **SINLEX - Um Ambiente de Desenvolvimento de Processadores de Linguagens**. Instituto de Informática, UFRGS. Trabalho de diplomação. 1989.

[SIL94] SILVA, J. B. & PRICE, A. M. A. Métrica de Complexidade de Software Baseada em Critério de Seleção de Casos de Teste. **VIII Simpósio Brasileiro de Eng. de Software**. Out. 1994. Curitiba, PR. Pp. 453-469.

[SIL95] SILVA, J. B. **PROTESTE+: Ambiente de Validação Automática de Qualidade de Software através de Técnicas de Teste e de Métricas de Complexidade. Dissertação de Mestrado**. CPGCC-UFRGS. 1995.

[WEY80] WEYUKER, E. J. & OSTRAND, T. J. Theories of Program Testing and the Application of Revealing Subdomains. **IEEE Transactions on Software Eng., vol. SE-6(3)**. May, 1980. Pp. 236-246.