**Universidade do Minho**

Mestrado Integrado em Engenharia Informática - 4º ano

## Métodos Formais em Engenharia de Software

2016/2017

# Verificação Formal

Módulo *VCGen*

David Moreira

21 de Junho de 2017

# Conteúdo

# 1 Introdução

Este trabalho surge no âmbito da Unidade Curricular de Verificação Formal, inserida na especialização em Métodos Formais em Engenharia de Software, e tem como objetivo o desenvolvimento de uma aplicação para gerar condições de verificação, para programas devidamente anotados, e validar essas mesmas condições, recorrendo a uma ferramenta de prova (*prover*).

A sintaxe da linguagem a ser usada para esses programas, foi escolhida no início do desenvolvimento, e foi desenvolvido um *parser* para reconhecer a sua sintaxe, e interpretar os programas e respetivas anotações, para os quais se pretendem gerar e validar as condições de verificação geradas.

# 2 Linguagem Alvo

A linguagem alvo escolhida trata-se de uma *simple language* (*sl*), à semelhança de uma linguagem imperativa como *C*, mas com algumas particularidades e diferenças na sintaxe.

A linguagem usada e o respetivo *parser*, reconhecem as construções básicas fixadas no enunciado do projeto. Tais como, variáveis do tipo inteiro, expressões do tipo inteiro e *boolean*, atribuições, sequências de instruções, estruturas condicionais (*if then else*) e cíclicas (*while do*).

No exemplo em baixo podemos ver a codificação do excerto de código dado no enunciado do projeto, utilizando a sintaxe da linguagem definida.

```
begin:
    while x < 1000 do:
        x = x + 1;
    end
end
```

# 3 Linguagem de Especificação

A linguagem de especificação prevê os mecanismos base para as anotações do código, tal como mencionado no enunciado do projeto. Sendo estes mecanismos, pré-condição (*pre*), pós-condição normal (*postn*), pós-condição execional (*poste*) e invariante({ ... })

A sintaxe apenas não prevê o tratamento de exceções (*try catch*).

No exemplo em baixo podemos ver a codificação do excerto de código dado no enunciado do projeto, com as respetivas anotações.

```
pre x > 100

begin:
    while x < 1000 do:
        {100 < x && x <= 1000}
        x = x + 1;
    end
end

postn x = 1000
poste false
```

Encontram-se no *Anexo 1* os restantes exemplos usados para testar e validar a aplicação

# 4 Infraestrutura de Desenvolvimento

Inicialmente, recorri à ferramenta *ANTLR* para produzir a gramática para gerar o *parser* da linguagem. Foi necessário especificar e fazer refletir na gramática todas as construções e também as anotações, das quais pretendemos reconhecer a quando da leitura dos programas. Encontra-se no *Anexo 2* do presente documento a gramática desenvolvida para a linguagem utilizada.

A ferramenta *GOM* foi utilizada para, juntamente com o *ANTLR*, gerar as estruturas de dados e ajudar na construção das Árvores de Sintaxe Abstratas (*AST*).

A ferramenta *TOM*, em ambiente *Java*, foi utilizada para interligar as duas ferramentas anterior, no sentido de gerar a *AST* para o programa a receber por *input*. Através dessa árvore, que vai sendo percorrida, e vão sendo gerados os valores e ações pretendidas, conforme vamos fazendo *match* das construção especificadas.

Foi utilizada programação estratégica durante a fase de construção da *AST*, mais especificamente, para ir registando as anotações do código, variáveis do programa e também para construir as condições de verificação.

É também aqui que traduzimos as condições de verificação geradas a partir da árvore de sintaxe, para o formato *standard SMT-LIB*, para ser possível mais à frente validar estas mesmas condições.

Podemos encontrar no *Anexo 3* as principais funções que são referidas a cima, que permitiram desenvolver todo o resto da aplicação com base na informação registada nestas funções. Sendo estas funções, as de registar as anotações, registar as variáveis do programa que são encontradas.

Por fim, foi utilizada a ferramenta *z3* para tentar validar as condições de verificação geradas pela aplicação, para um dado programa.

Tal como já foi dito, as condições de verificação são traduzidas para o formato *SMT-LIB*, sendo que, apenas foi necessário submeter o ficheiro para onde foram registadas essas mesma condições de verificação, ao *prover z3*, e recolher o resultado dado pelo mesmo.

# 5 Árvore de Sintaxe Abstrata (*AST*)

Durante a execução da aplicação, e após ser gerada a Árvore de Sintaxe Abstrata, foi guarda em formato *txt*, para posterior análise. É também gerado um ficheiro *dot* para a mesma *AST* e posteriormente, recorrendo à ferramenta *Graphviz*, é também gerada uma *AST* em formato imagem que reflete exatamente a árvore gerada.

Em baixo encontra-se a imagem produzida pela ferramenta *Graphviz*, através do ficheiro *dot* gerando durante a execução da aplicação para o programa dado. Encontra-se no *Anexo 4* a *AST* gerada em formato de texto e o código do ficheiro *dot*, para ser possível analisar a árvore nesse formato diferente, mas que representa o mesmo.

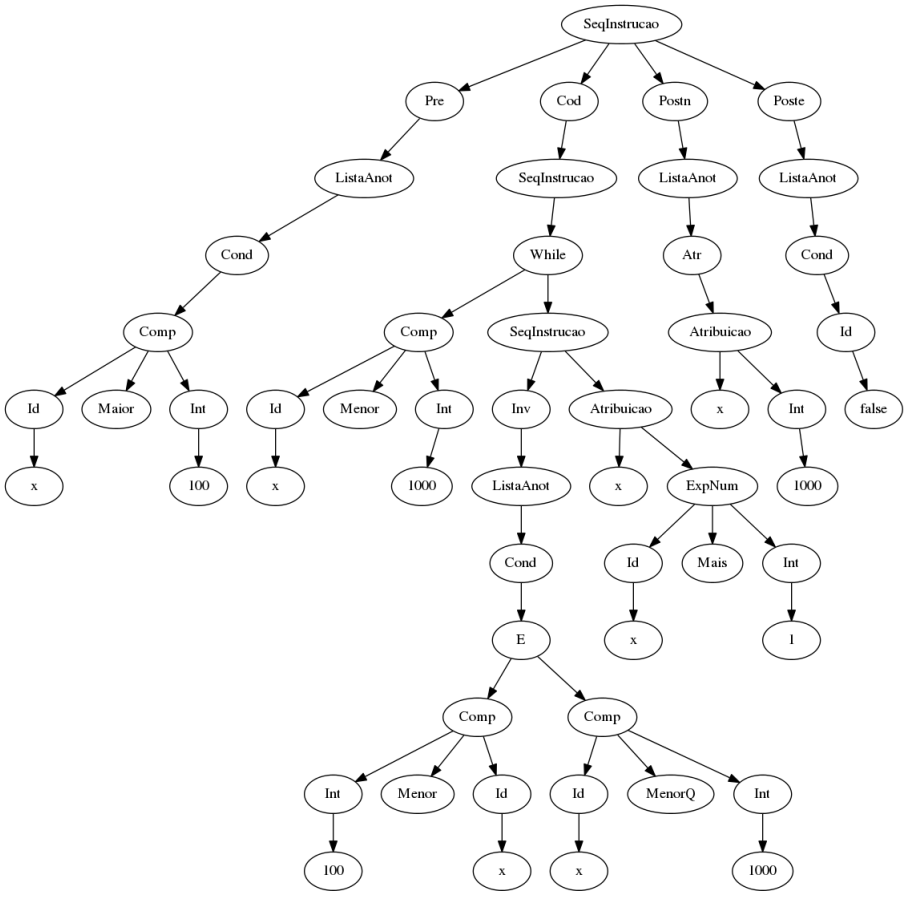


**Figura 1:** Árvore de Sintaxe Abstrata

# 6 Condições de Verificação

Tal como já foi explicado, as condições de verificação são geradas durante a execução da aplicação, mais especificamente, percorrendo a *AST* gerada, e posteriormente, são traduzidas para o formato *SMT-LIB*, para por fim tentar validar essas mesmas condições.

As condições de verificação são armazenadas num ficheiro de *txt* para posterior análise, assim como as condições de verificação traduzidas para *SMT-LIB*, são armazenadas num ficheiro *smt2*, para posteriormente ser submetido à ferramenta *z3*, para validar essas mesmas condições.

Mais uma vez, para o exemplo dado no enunciado do trabalho, foram geradas as condições de verificação, que correspondem as que são também apresentadas nesse mesmo documento.

Condições de verificação (VCs) geradas:

```
VC1: x>100 => 100<x and x<=1000
VC2: 100<x and x<=1000 and x<1000 => 100<x+1 and x+1<=1000
VC3: 100<x and x<=1000 and not(x<1000) => x=1000
```

Depois de traduzidas para formato *SMT-LIB*:

```
(declare-fun x () Int)
(assert (not (=> (> x 100)(and (< 100 x)(<= x 1000)))))
(check-sat)
(assert (not (=> (and (< 100 x)(<= x 1000))(< x 1000)(and (< 100 (+ x 1)
   )(<= (+ x 1) 1000)))))
(check-sat)
(assert (not (=> (and (< 100 x)(<= x 1000))(not(< x 1000))(= x 1000 ))))
(check-sat)
```

Após submeter estas mesmas condições de verificação ao *prover*, o resultado obtido foi o seguinte:

- **VC1**: *sat*, ou seja, não se verifica
- **VC2**: *unsat*, ou seja, verifica-se
- **VC3**: *unsat*, ou seja, verifica-se

# 7 Conclusões e Trabalho Futuro

Apesar de já ter uma breve experiência com as ferramentas utilizadas, na Unidade Curricular de Análise e Testes de Software, tive algumas dificuldades em interligar todas as ferramentas e trabalhar com as mesmas, numa fase mais inicial do projeto.

Penso ter atingidos os objetivos base pretendidos para este projeto, apesar de se ter revelado um trabalho um pouco extenso, uma vez que não fiz a gestão mais apropriada do tempo, e também pelo facto de realizar o trabalho individualmente.

Como trabalho futuro, fica por concluir o tratamento de exceções, e colocar a aplicação a gerar e validar as condições de verificação, da mesma forma que faz para os restantes casos. Também seria interessante melhorar o *workflow* e interessam com a aplicação, uma vez que esta executa de uma só vez todos as funcionalidades implementadas e armazena em ficheiros (e produz imagens), de uma só vez, apresentando apenas o resultado da validação ao utilizador.

# 8 Anexos

## 8.1 Anexo 1

Atribuição

```
pre x >= 0 && x <= 100;

begin:
  x = x+100;
end

postn x <= 200;
poste false;
```

Estrutura condicional (*if then else*)

```
pre x >= -100 && x <= 100;

begin:
  if x < 0 then:
    x = x + 100;
  end
end

postn x >= 0 && x <= 200;
poste false;
```

## 8.2 Anexo 2

Ficheiro *g.g*

```
grammar g;

options {
  output=AST;
  ASTLabelType=Tree;
  tokenVocab=gTokens;
}

@header {
        package src;
}

@lexer::header {
        package src;
}


idTipo          : ('int' -> ^(DInt))
                        ;

tipo            : (INT -> ^(Int INT))
                        ;

prog            : pre* 'begin' programa* postn* poste* EOF -> ^(SeqInstrucao
    pre? programa* postn? poste?)
                        ;

pre             : 'pre' anotacao+ -> ^(Pre ^(ListaAnot anotacao*))
                ;

postn           : 'postn' anotacao+ -> ^(Postn ^(ListaAnot anotacao*))
                ;

poste           : 'poste' anotacao+ -> ^(Poste ^(ListaAnot anotacao*))
                ;

inv             : '{' anotacao '}' -> ^(Inv ^(ListaAnot anotacao*))
                ;

anotacao        : (atribuicao ';' -> ^(Atr atribuicao)
                        | condicao ';' -> ^(Cond condicao))
                        ;

programa        : blocoCodigo -> ^(Cod blocoCodigo)
                        ;

instrucao       : (if_st -> if_st | while_st -> while_st)
                        ;
```

```
if_st           : 'if' condicao 'then' blocoCodigo ( else_st -> ^(If
   condicao blocoCodigo else_st)
                        | -> ^(If condicao blocoCodigo ^(SeqInstrucao) ))
                        ;

else_st         : 'else' (blocoCodigo -> blocoCodigo | if_st -> if_st )
                        ;

while_st        : 'while' condicao 'do' blocoCodigo -> ^(While condicao
   blocoCodigo)
                        ;

blocoCodigo : ':' inv? codigo* 'end' -> ^(SeqInstrucao inv? codigo*)
                        ;

codigo          : (atribuicao ';' -> atribuicao
                        | instrucao -> instrucao)
                        ;

condicao        : condicao_ou ('?' expr ':' condicao -> ^(Condicional
   condicao_ou expr condicao)
                        | -> condicao_ou)
                        ;

condicao_ou : (condicao_e -> condicao_e) ('||' c=condicao_e -> ^(Ou
   $condicao_ou $c))*
                        ;

condicao_e      : (condicao_comp -> condicao_comp) ('&&' c=condicao_comp ->
   ^(E $condicao_e $c))*
                        ;

condicao_comp   : (condicao_ig -> condicao_ig) (('>' c=condicao_ig -> ^(Comp
    $condicao_comp ^(Maior) $c)
                                | '<' c=condicao_ig -> ^(Comp $condicao_comp
                                    ^(Menor) $c)
                                | '>=' c=condicao_ig -> ^(Comp
                                   $condicao_comp ^(MaiorQ) $c)
                                | '<=' c=condicao_ig -> ^(Comp
                                   $condicao_comp ^(MenorQ) $c)))*
                                ;

condicao_ig : (expr -> expr) (('!=' e=expr -> ^(Comp $condicao_ig ^(Dif) $e)
                        | '==' e=expr -> ^(Comp $condicao_ig ^(Igual) $e)))*
                        ;

atribuicao      : ID '=' condicao -> ^(Atribuicao ID condicao)
                        ;
```

```
expr            : (exprNum -> exprNum) (('+' e=exprNum -> ^(ExpNum $expr ^(
   Mais) $e)
                        | '-' e=exprNum -> ^(ExpNum $expr ^(Menos) $e)))*
                        ;

exprNum         : (op -> op) (('*' o=op -> ^(ExpNum $exprNum ^(Vezes) $o)
                        | '/' o=op -> ^(ExpNum $exprNum ^(Divide) $o)
                        | '%'  o=op -> ^(ExpNum $exprNum ^(Mod) $o)))*
                        ;

op                      : (opU ID -> ^(opU ^(Id ID))
                        | opU tipo -> ^(opU tipo)
                        | tipo -> tipo
                        | ID -> ^(Id ID))
                        ;

opU                     : ( '+' -> ^(Pos)
                        | '-' -> ^(Neg)
                        | '!' -> ^(Nao))
                        ;


CHAR
@after {
    setText(getText().substring(1, getText().length()-1));
}
        : '\'' ( '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\') | ~('\''|'\\') )
           '\''
        ;

fragment
DIGITO
        : ('0'..'9')+
        ;

INT
        : ('0' | '1'..'9' DIGITO*)
        ;

ID
        : LETRA ( LETRA | '0'..'9' )*
        ;

fragment
LETRA
        : 'a'..'z' | 'A'..'Z' | '_'
        ;

WS
        : (' '|'\r'|'\t'|'\u000C'|'\n') {$channel=HIDDEN;}
    ;
```

## 8.3 Anexo 3

```
%strategy Strategy_Anotacoes() extends Identity() {
        visit Instrucao {
                Pre(anot) -> {
                        ant = pre = anot_string(`anot);
                        antSmt = preSmt = smt_string(`anot);
                        `TopDown(Strategy_ID(0)).visit(`anot);
                }
                Postn(anot) -> {
                        postn = anot_string(`anot);
                        postnSmt = smt_string(`anot);
                        `TopDown(Strategy_ID(1)).visit(`anot);
                }
                Poste(anot) -> {
                        //
                }
        }
}

%strategy Strategy_ID(int id_inst) extends Identity() {
        visit Expressao {
                Id(id) -> {
                        if(id_inst == 0 && !(variaveis.containsKey(`id))){
                                variaveis.put(`id+"", `id+"");
                                variaveis_smt.put(`id+"", `id+"");
                                smt.add("(declare-fun␣"+ `id +"␣()␣Int)");
                        }
                        if(!(variaveis.containsKey(`id)) && !(variaveis_na.
                            contains(`id)))
                                variaveis_na.add(`id+"");
                }
        }
}
```

## 8.4 Anexo 4

Ficheiro *txt*

```
SeqInstrucao(Pre(ListaAnot(Cond(Comp(Id("x"),Maior(),Int(100))))),Cod(
    SeqInstrucao(While(Comp(Id("x"),Menor(),Int(1000)),SeqInstrucao(Inv(
    ListaAnot(Cond(E(Comp(Int(100),Menor(),Id("x")),Comp(Id("x"),MenorQ(),Int
    (1000)))))),Atribuicao("x",ExpNum(Id("x"),Mais(),Int(1)))))),Postn(
    ListaAnot(Atr(Atribuicao("x",Int(1000))))),Poste(ListaAnot(Cond(Id("false
    "))))
```

Ficheiro *dot*

```
digraph visitable {

    ordering=out;

      p [label="SeqInstrucao"];

      p1 [label="Pre"];

        p -> p1;

      p1_1 [label="ListaAnot"];

        p1 -> p1_1;

      p1_1_1 [label="Cond"];

        p1_1 -> p1_1_1;

      p1_1_1_1 [label="Comp"];

        p1_1_1 -> p1_1_1_1;

      p1_1_1_1_1 [label="Id"];

        p1_1_1_1 -> p1_1_1_1_1;

      p1_1_1_1_1_1 [label="x"];

        p1_1_1_1_1 -> p1_1_1_1_1_1;

      p1_1_1_1_2 [label="Maior"];

        p1_1_1_1 -> p1_1_1_1_2;

      p1_1_1_1_3 [label="Int"];

        p1_1_1_1 -> p1_1_1_1_3;

      p1_1_1_1_3_1 [label="100"];
```

```
  p1_1_1_1_3 -> p1_1_1_1_3_1;
p2 [label="Cod"];
  p -> p2;
p2_1 [label="SeqInstrucao"];
  p2 -> p2_1;
p2_1_1 [label="While"];
  p2_1 -> p2_1_1;
p2_1_1_1 [label="Comp"];
  p2_1_1 -> p2_1_1_1;
p2_1_1_1_1 [label="Id"];
  p2_1_1_1 -> p2_1_1_1_1;
p2_1_1_1_1_1 [label="x"];
  p2_1_1_1_1 -> p2_1_1_1_1_1;
p2_1_1_1_2 [label="Menor"];
  p2_1_1_1 -> p2_1_1_1_2;
p2_1_1_1_3 [label="Int"];
  p2_1_1_1 -> p2_1_1_1_3;
p2_1_1_1_3_1 [label="1000"];
  p2_1_1_1_3 -> p2_1_1_1_3_1;
p2_1_1_2 [label="SeqInstrucao"];
  p2_1_1 -> p2_1_1_2;
p2_1_1_2_1 [label="Inv"];
  p2_1_1_2 -> p2_1_1_2_1;
p2_1_1_2_1_1 [label="ListaAnot"];
  p2_1_1_2_1 -> p2_1_1_2_1_1;
p2_1_1_2_1_1_1 [label="Cond"];
```

```
  p2_1_1_2_1_1 -> p2_1_1_2_1_1_1;
p2_1_1_2_1_1_1_1 [label="E"];
  p2_1_1_2_1_1_1 -> p2_1_1_2_1_1_1_1;
p2_1_1_2_1_1_1_1_1 [label="Comp"];
  p2_1_1_2_1_1_1_1 -> p2_1_1_2_1_1_1_1_1;
p2_1_1_2_1_1_1_1_1_1 [label="Int"];
  p2_1_1_2_1_1_1_1_1 -> p2_1_1_2_1_1_1_1_1_1;
p2_1_1_2_1_1_1_1_1_1_1 [label="100"];
  p2_1_1_2_1_1_1_1_1_1 -> p2_1_1_2_1_1_1_1_1_1_1;
p2_1_1_2_1_1_1_1_1_1_2 [label="Menor"];
  p2_1_1_2_1_1_1_1_1_1 -> p2_1_1_2_1_1_1_1_1_1_2;
p2_1_1_2_1_1_1_1_1_1_3 [label="Id"];
  p2_1_1_2_1_1_1_1_1_1 -> p2_1_1_2_1_1_1_1_1_1_3;
p2_1_1_2_1_1_1_1_1_1_3_1 [label="x"];
  p2_1_1_2_1_1_1_1_1_1_3 -> p2_1_1_2_1_1_1_1_1_1_3_1;
p2_1_1_2_1_1_1_1_1_2 [label="Comp"];
  p2_1_1_2_1_1_1_1_1_1 -> p2_1_1_2_1_1_1_1_1_2;
p2_1_1_2_1_1_1_1_1_2_1 [label="Id"];
  p2_1_1_2_1_1_1_1_1_2 -> p2_1_1_2_1_1_1_1_1_2_1;
p2_1_1_2_1_1_1_1_1_2_1_1 [label="x"];
  p2_1_1_2_1_1_1_1_1_2_1 -> p2_1_1_2_1_1_1_1_1_2_1_1;
p2_1_1_2_1_1_1_1_1_2_2 [label="MenorQ"];
  p2_1_1_2_1_1_1_1_1_2 -> p2_1_1_2_1_1_1_1_1_2_2;
p2_1_1_2_1_1_1_1_1_2_3 [label="Int"];
  p2_1_1_2_1_1_1_1_1_2 -> p2_1_1_2_1_1_1_1_1_2_3;
```

```
p2_1_1_2_1_1_1_1_2_3_1 [label="1000"];
  p2_1_1_2_1_1_1_1_2_3 -> p2_1_1_2_1_1_1_1_2_3_1;
p2_1_1_2_2 [label="Atribuicao"];
  p2_1_1_2 -> p2_1_1_2_2;
p2_1_1_2_2_1 [label="x"];
  p2_1_1_2_2 -> p2_1_1_2_2_1;
p2_1_1_2_2_2 [label="ExpNum"];
  p2_1_1_2_2 -> p2_1_1_2_2_2;
p2_1_1_2_2_2_1 [label="Id"];
  p2_1_1_2_2_2 -> p2_1_1_2_2_2_1;
p2_1_1_2_2_2_1_1 [label="x"];
  p2_1_1_2_2_2_1 -> p2_1_1_2_2_2_1_1;
p2_1_1_2_2_2_2 [label="Mais"];
  p2_1_1_2_2_2 -> p2_1_1_2_2_2_2;
p2_1_1_2_2_2_3 [label="Int"];
  p2_1_1_2_2_2 -> p2_1_1_2_2_2_3;
p2_1_1_2_2_2_3_1 [label="1"];
  p2_1_1_2_2_2_3 -> p2_1_1_2_2_2_3_1;
p3 [label="Postn"];
  p -> p3;
p3_1 [label="ListaAnot"];
  p3 -> p3_1;
p3_1_1 [label="Atr"];
  p3_1 -> p3_1_1;
p3_1_1_1 [label="Atribuicao"];
  p3_1_1 -> p3_1_1_1;
```

```
    p3_1_1_1_1 [label="x"];
      p3_1_1_1 -> p3_1_1_1_1;
    p3_1_1_1_2 [label="Int"];
      p3_1_1_1 -> p3_1_1_1_2;
    p3_1_1_1_2_1 [label="1000"];
      p3_1_1_1_2 -> p3_1_1_1_2_1;
    p4 [label="Poste"];
      p -> p4;
    p4_1 [label="ListaAnot"];
      p4 -> p4_1;
    p4_1_1 [label="Cond"];
      p4_1 -> p4_1_1;
    p4_1_1_1 [label="Id"];
      p4_1_1 -> p4_1_1_1;
    p4_1_1_1_1 [label="false"];
      p4_1_1_1 -> p4_1_1_1_1;
}
```