4º SBRC
RECIFE - 24 A 26 DE MARÇO 86

Analisador Sintático em Prolog para a Linguagem de Especificação de Protocolos Estelle(*)

Wanderley Lopes de Souza
CPC/DSC/Universidade Federal da Paraíba
Edilson Ferneda
DSC/Universidade Federal da Paraíba

Sumário

Neste artigo é proposto um analisador sintático, escrito na linguagem de programação lógica Prolog, para a linguagem de especificação de protocolos "Extended state transition language (Estelle)". Após a introdução, onde é esclarecida a motivação para o desenvolvimento deste trabalho, são expostas e definidas sintaticamente as principais construções de Estelle. Quanto à Prolog, são apresentados os seus conceitos fundamentais e um modelo para a transcrição sintática de Estelle em Prolog. Em anexo, o leitor encontrará um exemplo de especificação de um protocolo em Estelle, assim como a sintaxe aceita pelo reconhecedor.

1. Introdução

A fase crítica no ciclo de desenvolvimento de um sistema de comunicação é a construção de uma **especificação** precisa de seus compenentes. O uso exclusivo de linguagens naturais para a descrição de **protocolos** complexos, embora aparentemente facilita a compreensão, leva a especificações informais, frequentemente ambíguas. Uma especificação formal, além de servir como referência para o trabalho em equipe, poderá ser útil para o desenvolvimento de **ferramentas** que possibilitem: a **validação** da

(*) Trabalho realizado com auxílio fornecido pelo CNPq.

especificação; a implementação automática do protocolo e o teste da implementação.

Preocupados com a fixação de normas para a especificação de protocolos, órgãos internacionais de padronização, tais como, o "National Bureau of Standards (NBS)", o "Comité Consultatif International Télégraphique et Téléphonique (CCITT)" e a "International Standards Organization (ISO)", têm desenvolvido técnicas de descrição formal para protocolos de comunicação. Como consequência, foi criado um grupo de trabalho junto a este último organismo (ISO TC 97/SC16/W G1 subgroup B), que vem, no decorrer dos últimos anos, estabelecendo uma linguagem para a especificação formal de protocolos baseada num modelo de **máquina de estados finitos estendidos (MEFE)** e que recentemente foi denominada "**Extended state transition language (Estelle)**".

A construção de um **compilador** para a linguagem Estelle permitirá a obtenção automática de implementações de sistemas especificados nesta linguagem, facilitando a transportabilidade de tais implementações, uma vez que eliminará a codificação personalizada. Uma implementação automática será uma referência confiável, para a análise do comportamento de uma implementação desenvolvida manualmente (desde que ambas sejam derivadas da mesma especificação). Finalmente, este compilador poderá integrar-se a um sistema de **simulação**, que permitirá a validação de especificações escritas em Estelle.

Entre as vantagens apresentadas em [Warr 80] para o emprego da linguagem de programação lógica **Prolog** na construção de compiladores, em relação ao emprego de outras linguagens, destacamos: o esforço e o tempo requeridos para desenvolver o compilador são inferiores; o código obtido é mais preciso e legível, facilitando assim a documentação; extensões da linguagem

fonte e modificações do compilador são mais facilmente incorporadas. A interrogação maior que resta é a seguinte: o desempenho de um compilador em Prolog é suficientemente bom para a utilização prática desejada?

Entre os diversos elementos envolvidos na construção de um compilador, o **analisador sintático** ("parser") é o que mais incentiva o emprego de Prolog. O equacionamento do problema 'análise sintática', assim como a transcrição das regras gramaticais (de uma gramática livre de contexto) numa linguagem de programação, são realizados de forma natural em Prolog [ClMe 81]. Uma vez que a linguagem Estelle é proveniente de um modelo MEFE e a obtenção de uma gramática a partir de um autômato finito é factível, o uso de Prolog, para o desenvolvimento de um reconhecedor de especificações escritas em Estelle, parece ser bastante promissor.

2. A linguagem de especificação Estelle

A arquitetura de um sistema é definida por um conjunto de **módulos**, interconectados através de **canais**. Um módulo é visto pelo seu ambiente (demais módulos que compõem o sistema) como um conjunto de **portas**. Neste tipo de estrutura a comunicação é indireta, o que permite especificar módulos e conexões separadamente.

Um canal define um conjunto de **primitivas de interação** (que podem ser invocadas através deste tipo de canal), os valores possíveis dos parâmetros associados a estas primitivas e os papéis que os módulos, conectados às extremidades deste canal, deverão desempenhar (e.g., "user" e "provider"). Na especificação de um módulo cada porta é caracterizada pelo tipo de canal ao qual ela está associada.

A especificação de um módulo pode ser realizada através do seu refinamento em submódulos e através da definição do comportamento de cada submódulo.

No caso da linguagem Estelle, cada módulo é representado por uma MEFE. As **transições** de uma MEFE são consequências das interações produzidas pelo ambiente e recebidas pelo módulo (**entradas**), ou dos eventos internos (**transições espontâneas**).Ao executar uma transição, o módulo pode gerar interações(**saídas**). As interações de um módulo com o seu ambiente são consideradas atômicas, ou seja, uma única entrada de um único módulo é tratada por vez. Como a partir de uma interação de entrada é possível a existência de várias transições executáveis e como a partir de um determinado estado existe uma transição espontânea executável, a especificação de um módulo é **não determinística.**

A descrição de um módulo contem uma variável, que define o **estado principal** do módulo, e um conjunto de **variáveis de estado adicionais.** Simultaneamente com os parâmetros da interação de entrada, elas determinam a transição a executar no módulo. A cada transição está associada uma **condição**, que deve ser satisfeita para a execução da transição, e uma ação,que atualiza os valores das variáveis e eventualmente gera as interações de saída (maiores detalhes relativos a tais conceitos são apresentados em [Lopes 85].

2.1. Especificação de um protocolo em Estelle

Uma vez definida a arquitetura do sistema a ser especificado, a descrição dos componentes deste sistema,em Estelle, assemelha-se a um programa Pascal (a sintaxe da linguagem Estelle têm suas bases na linguagem de programação Pascal da ISO [JeWe 74]).

Inicialmente são declarados os elementos globais: constantes e tipos. Símbolos do tipo '...' ou palavras do tipo 'primitive' indicam que a escolha é deixada a cargo da implementação. Em seguida, são definidos os canais, conjuntamente com as primitivas a serem iniciadas pelo usuário e pelo provedor de serviços. Finalmente, os módulos são declarados.

Após o cabeçalho de um módulo (declaração de seu nome e de suas portas) são definidos os seus elementos locais: constantes, tipos, rótulos, variáveis (entre as quais uma representando o estado principal), funções e procedimentos. O módulo é então inicializado e as transições são descritas.

Uma transição é declarada através das seguintes cláusulas: **from** (seguida do estado principal vigente), **to** (seguida do novo estado principal), **when** (seguida da interação de entrada) e **provided** (seguida das condições habilitadoras da transição). A fim de condensar a descrição das transições, a ordem destas cláusulas é irrelevante e pode haver uma combinação das mesmas. As transições espontâneas podem conter as cláusulas **delay** e **any**. Após a declaração das cláusulas de uma transição são atualizadas as variáveis de estado adicionais. Para a execução das transições, uma **priority** pode ser estabelecida (quanto maior é o valor do número inteiro, menor é a prioridade).

O Anexo I apresenta uma especificação em Estelle do protocolo bit-alternante [ISO 83]. A arquitetura do sistema é composta de 2 módulos (Alternating-Bit e Timer) e 3 canais (U_access_point, N_access_point e S_access_point). As entradas provenientes dos 02 primeiros canais são colocadas numa fila comum e as provenientes do último são colocadas numa fila individual. As saídas de um módulo para o canal são indicadas através da pa

lavra 'OUT', seguida da porta, do símbolo '.' e da interação.

O módulo Alternating-Bit possue as portas de acesso U, N e S, através das quais ele recebe e envia interações. Caso o módulo esteja operando como produtor, ele recebe dados do nível superior (porta U) através da primitiva SEND_request, transmite-os ao nível inferior (porta N) através de Data_request, e recebe o reconhecimento dos dados enviados através de Data_response. Caso o módulo esteja operando como consumidor, ele recebe dados do nível inferior através de Data_response, envia o reconhecimento ao produtor através de Data_request e ao receber uma solicitação do nível superior através de RECEIVE_request, entrega os dados através de RECEIVE_response. A distinção entre dado e reconhecimento é realizada no campo id da variável (tipo record) N Data.

Dois estados são possíveis para o módulo Alternating-Bit: ESTAB e ACK_WAIT. Para possibilitar a retransmissão de mensagens, 02 "buffers" são previstos: send-buffer e recv-buffer.Os procedimentos store, remove e a função retrieve permitem a manipulação desses "buffers". A escolha da estrutura de dados de buffer-type e consequentemente os detalhes dos procedimentos e funções relacionados aos "buffers" são deixados a cargo da implementação.

O módulo Timer permite a detecção de mensagens perdidas ou duplicadas. O produtor, ao enviar um dado, aciona o módulo timer (porta S)através da primitiva Timer_request, que por sua vez inicializa o temporizador. O fim da contagem é sinalizado ao produtor através de Timer_response. Caso o produtor esteja no estado ACK_WAIT, o dado é retransmitido e uma nova temporização é solicitada. Caso contrário, nenhuma atividade é realizada.

Ao receber um dado, o consumidor envia o reconhecimento com o mesmo número de sequência recebido. Posteriormente, ele comparà este número com o número de sequência esperado. Caso eles sejam iguais, o dado é armazenado e o número de sequência esperado é incrementado. Caso contrário, o dado é descartado.

O produtor, por sua vez, ao receber um reconhecimento com o número de sequência diferente do esperado, despreza este reconhecimento e permanece no estado ACK_WAIT. Uma vez expirada a temporização, o produtor retransmite o último dado enviado. Este procedimento, conjuntamente com o descrito no parágrafo acima, elimina mensagens duplicadas.

3. Programação em lógica e Prolog

A lógica é usada para representar problemas e obter suas soluções de uma maneira formal. Esses problemas são expressos por meio de asserções e das relações entre elas [Li 84] [ChLe 73].

Uma forma particular de lógica é o **cálculo de predicados de 1ª ordem**. Seus objetos são chamados **termos** e as relações entre estes objetos são as **fórmulas**.

Termos são definidos como:

(a) uma constante é um termo;

(b) uma variável é um termo;

(c) se f é um símbolo funcional n_ário e $t_1, t_2, \ldots, t_n$ são termos, então $f(t_1, t_2, \ldots, t_n)$ é um termo;

(d) todos os termos são gerados pela aplicação das regras acima.

Fórmulas em lógica de 1ª ordem podem ser definidas como:

(a) se P é um símbolo predicativo n_ário e $t_1, t_2, \ldots, t_n$ são

termos, então $P(t_1, t_2, \ldots, t_n)$ é uma fórmula atômica;

(b) se F e G são fórmulas, então $\neg F$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$ e $(F \leftrightarrow G)$ são fórmulas;

(c) se F é uma fórmula e x é uma variável, então $\forall xF$ e $\exists xF$ são fórmulas;

(d) todas as fórmulas são geradas pela aplicação das regras acima.

Como exemplo, as asserções

(1) Sócrates é um homem;

(2) todo homem é mortal;

(3) se todos os homens são martais e Sócrates é um homem, então, Sócrates é mortal.

ficariam, em fórmulas de cálculo de predicado, da seguinte maneira:

(1) homem (Sócrates)

(2) $(\forall x)$(homem (x) $\longrightarrow$ mortal (x))

(3) $(\forall x)$(homem (x) $\longrightarrow$ mortal (x))$\wedge$homem (Sócrates)$\rightarrow$ mortal(Sócrates).

Para fazermos manipulações formais, entretanto, fica muito difícil usarmos as fórmulas de cálculo de predicado. Podemos então simplificá-las para uma forma mais simples: a cláusula.

Uma **cláusula** é uma expressão do tipo

$$B_1, B_2, \ldots B_m \longleftarrow A_1, A_2, \ldots, A_n$$

onde $B_1, B_2, \ldots B_m, A_1, A_2, \ldots A_n$ são fórmulas atômicas, $n \geqslant 0$ e $m \geqslant 0$. Se a cláusula contém as variáveis $x_1, x_2, \ldots, x_k$, devemos interpretá-la como:

para todo $x_1, x_2, .., x_k$, B_1 ou B_2 ou ... B_m é válida se A_1 e A_2 e ..., A_n forem válidas.

Temos ainda alguns casos especiais:

(1) se n = 0, isto é,

$$B_1, B_2, \ldots, B_m \leftarrow$$

então para todo $x_1, \ldots, x_k$, B_1 ou B_2 ou ..., B_m são incondicionalmente verdadeiros.

(2) se m = 0, isto é,

$$\leftarrow A_1, A_2, \ldots, A_n$$

então: para todo $x_1, \ldots, x_k$, não há caso em que A_1 e A_2 e ... , A_n seja verdadeiro.

(3) se m = n = 0, isto é cláusula vazia,

$\leftarrow$

então: a cláusula é sempre falsa.

As cláusulas contendo no máximo uma conclusão são chamadas **cláusulas de Horn**. Há, portanto, 2 tipos de cláusulas de Horn:

(1) **cláusula com cabeça**

$$B \leftarrow A_1, A_2, \ldots, A_n$$

ou

$$B \leftarrow$$

(2) **cláusula sem cabeça**

$$\leftarrow A_1, A_2, \ldots, A_n$$

ou

$\leftarrow$ (cláusula vazia)

Qualquer problema solucionável pode ser expresso em cláusulas Horn, sendo umas delas uma cláusula sem cabeça e as restantes com cabeça. A cláusula sem cabeça deve ser interpretada como um questionamento sobre o problema. Por exemplo, a definição do fatorial

(1) O fatorial de 0 é 1;

(*) o fatorial de (x + 1) é (x + 1) multiplicado pelo fatorial de x.

seria representado em cláusulas Horn como:

fatorial (0,1) ←

fatorial (soma_1 (x), y) ← fatorial (x,A), mult(soma_1 (x), A,y)

onde "soma_1 (x)" indica a função x + 1; e "mult (a,b,c)" é uma fórmula predicativa mostrando que a * b é c.

Para questionarmos qual seria o fatorial de um certo valor, usaríamos a cláusula sem cabeça

← fatorial (4,x)

onde **x** é a variável a ser "instanciada" (*) com o resultado.

Prolog é uma linguagem de programação prática, baseada num modelo próximo ao da programação lógica e na interpretação procedural de Kowalski para cláusulas Horn [Kowa 79]. Um programa em Prolog consiste de um conjunto de cláusulas, que são analisadas da esquerda para a direita e de cima para baixo. A diferença básica das cláusulas em Prolog para as clásulas Horn é sintática [ClMe 81]:

Cláusula Horn	Cláusula Prolog
$B \leftarrow A_1, A_2, \ldots, A_n$	$B :- A_1, A_2, \ldots, A_n.$
$B \leftarrow$	$B.$
$\leftarrow A_1, A_2, \ldots, A_n$	$?- A_1, A_2, \ldots, A_n.$

Além dessas diferenças, há ainda as seguintes peculiaridades:

(1) **Constantes** podem ser inteiras, reais ou átomos. Por átomos entende-se qualquer conjunto de caracteres que não definem

---

(*) do verbo inglês "To instance", significando a atribuição de um valor a uma variável ainda sem conteúdo.

um valor numérico nem uma variável;

(2) **variáveis** são definidas por um conjunto de caracteres iniciados por uma letra maiúscula;

(3) **termos compostos**, assim como os símbolos funcionais, têm seu identificador inicial definido como uma variável.

A principal estrutura de dados utilizada pelo Prolog é a lista. Uma lista é apresentada na forma

$$[a_1, a_2, \ldots, a_n]$$

com $n \geq 0$ e onde cada elemento a_i pode ser um termo ou uma lista. Outra maneira de representarmos esta lista é

$$[x \mid y]$$

onde **x** é a **cabeça** da lista (a_1), **y** é a **cauda** da lista ($[a_2, \ldots, a_n]$) e "|" representa a concatenação entre a cabeça e a cauda da lista.

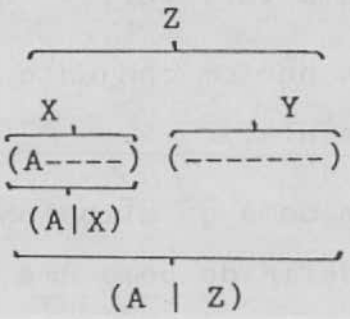
Peguemos como exemplo um conjunto de cláusulas que pode ser usado para manipular listas (concatenar, dividir, remover elementos da frente, de trás, etc.):

```
append ([ ], Y, Y).
append ([A| X ], Y, [A|Z]): - append (X,Y,Z).
```

A primeira cláusula informa que uma lista vazia unida com uma lista é a própria lista. A segunda cláusula informa que a união de uma lista [A|X] a uma segunda lista Y, gera uma terceira lista com a cabeça da primeira, ou seja, [A |Z]. Porém, para que isso seja verdadeiro,

```
append (X, Y, Z)
```

também o deve ser. Isto é ilustrado pela figura



As duas cláusulas acima definem os objetos e a relação entre eles. Estão na forma **declarativa**. Para se definir como o programa é executado usamos cláusulas na forma **procedural**.O exemplo acima pode ser usado como:

(a) reconhecedor de relações. Se o questionamento for, por exemplo,

```
?-  append ([a], [b,c], [a,b,c])
```

o resultado da execução será **sucesso**;

(b) gerador de respostas que satisfazem na relação. Se o questionamento for, por exemplo,

```
?-  append ([a], x, [a,b,c])
```

a resposta será **x = [b,c]**.

4. Analisador Sintático em Prolog para Estelle

A gramática de uma linguagem é definida a partir de um conjunto de símbolos e de regras que determinam a formação de sentenças desta linguagem. Por exemplo, um subconjunto da lingua portuguesa é determinado pelas seguintes regras gramaticais:

```
<sentença>    :: =  <sujeito> <predicado>
<sujeito>     :: =  <artigo><nome>
<predicado>   :: =  <verbo> <objeto> | <verbo>
<objeto>      :: =  <sujeito>
<artigo>      :: =   o | a
<nome>        :: =   menino | moça
<verbo>       :: =   comeu
```

É evidente a relação possível entre regras gramaticais e cláusulas Prolog [Warr 80]. O exemplo citado codificado em Prolog ficaria

```
sentença (X,Y) :- sujeito (X,Z), predicado (Z,Y)
sejeito (X,Y) :- artigo (X,Z), nome (Z,Y)
predicado (X,Y) :- verbo (X,Z), objeto (Z,Y)
predicado (X,Y) :- verbo (X,Y)
objeto (X,Y) :- sujeito (X,Y)
```

e o conjunto de símbolos, ou dicionário, seria as cláusulas

```
artigo ([o | X], X)
artigo ([a | X], X)
verbo ([comeu | X], X)
nome ([menino | X], X)
nome ([maçã | X], X)
```

Podemos, por artifícios de programação, deixar os argumentos de uma forma embutida.

Uma regra gramatical (em BNF) do tipo

```
A ::= BC
```

onde A, B e C são símbolos não-terminais, pode então ser descrito em Prolog pela cláusula

```
A :- B,C.
```

Convenções para regras gramaticais são facilmente transcritas em Prolog. A convenção

```
A ::- B|C
```

corresponde às seguintes cláusulas:

```
A :- B.
A :- C.
```

Uma regra do tipo

A ::=. empty ~

é representada em Prolog simplesmente como

A.

Um símbolo terminal 'x' aparecerá em Prolog como

terminal (**tipo** (**x**))

onde **x** é o terminal e **tipo** é o seu "parsing type" que pode ser **símbolo** ou **palavra**. Uma **palavra** deve ser seguida necessariamente de pelo menos um espaço em branco. Um **símbolo** pode ou não ser seguido de um espaço em branco.

Caso o tipo do símbolo seja **real**, **inteiro** ou uma cadeia de caracteres (**string**), teremos em Prolog

terminal (**tipo**)

ainda se for do tipo **ident**, teremos

terminal (ident, [**classe**, **oper**]).

Aqui, **classe** indica a classe do identificador (inteiro, booleano, ...) e **oper** a operação a ser executada (**decl** para declaração, **verif** para verificar sua declaração e **retira** para a retirada de sua declaração). Esse segundo argumento é necessário para a verificação e tratamento de erros semânticos. [Gerb 83][Wirt 76].

Têm-se também, terminais do tipo **inteiro** definindo rótulos

terminal (inteiro, [rótulo, **oper**]).

Além dos argumentos definidos acima, às vezes é necessário um outro para a verificação e tratamento de erros sintáticos. Assim, quando a falta do terminal deve ser considerada um erro, uma lista com os delimitadores a serem procurados na se

quência de entrada é acrescentada entre os argumentos do terminal. Por exemplo, as regras gramaticais abaixo:

```
<constd>::= "const" <defconst> | empty
<defconst>::= <ident> "=" <constant> ";" <seqdefconst>
<seqdefconst>::= <defconst> | empty
<constant>::= <optsing><numconst> | <string>
<sign>::= "+" | "-"
<optsign>::= <sign> | empty
<numconst>::= <inteiro> | <real> | <ident>
```

são descritos em Prolog pelas cláusulas

```
const:- terminal (palavra ("const")), defconst.
constd.
defconst:- terminal (ident, [const, decl]),
     terminal (símbolos ("="), [";"]), constant,
     terminal (símbolo (";"), [";"]), seqdefconst.
seqdefconst:- defconst.
seqdefconst.
constant:- optsign, numconst.
constant:- terminal (string).
sign:- terminal (símbolo ("+")).
sign:- terminal (símbolo ("-")).
optsign:- sign.
optsign.
numconst:- terminal (real).
numconst:- terminal (inteiro).
numconst:- terminal (ident, [const, verif]).
```

O subconjunto da linguagem Estelle que consideramos é definido por 245 regras gramaticais (vide Anexo II) [GeBo 83] . Além das cláusulas de igual número, são necessárias mais algumas (aproximadamente 25) para dar suporte ao tratamento de sím

bolos terminais.

Nesta primeira implementação, utilizou-se o interpretador micro-Prolog(*) disponível no equipamento Nexus 1600(**), sendo que a memória ocupada pelo parser foi em torno de 40K. A sintaxe deste interpretador é diferente do Prolog padrão.As principais diferenças são [Clar 84]:

(1) "if" no lugar de ":-";

(2) "and" como indicador de conjunção em vez de ",";

(3) questionamentos são feitos na forma

$$is(A_1 \text{ and } \ldots \text{ and } A_n)$$

(4) o controle de "backtracking" é feito explicitamente através de uma cláusula "/";

(5) as listas são representadas entre **(** e **)** ao invés de **[** e **]**;

(6) os elementos das lista e os argumentos das cláusulas são separados por um espaço em branco no lugar de ",".

Além dessas diferenças sintáticas, para otimização do tempo de execução, os argumentos de controle estão explícitos . Assim, uma regra gramatical, como por exemplo

<const>::=<optsign> <numconst> | <string>

em micro-Prolog ficará

```
constant (X Y) if
        optsign (X Z) and
        numconst (Z Y) and
        /
constant (X Y) if
        terminal (string (X Y))
```

A maior limitação imposta pelo interpretador utiliza-

(*) micro-Prolog é um produto da Logic Programming Associates Ltda.

(**) Nexus 1600 é um produto da Scopus Tecnologia.

do e que restringiu os testes realizados com o protocolo Bit-alternante especificado em Estelle (vide Anexo I), foi a existência de um limite máximo de elementos permitidos numa lista de entrada.

5. Conclusão

As principais vantagens da utilização de Estelle, para a especificação de protocolos de comunicação, residem na simplicidade desta linguagem e na facilidade de extrair-se, a partir de uma descrição Estelle, um autômato para o emprego dos métodos de validação associados

Um compilador Estelle, além de permitir a obtenção de implementações automáticas, pode ser usado junto a um sistema de simulação visando a validação de especificações escritas nesta linguagem.

O analisador sintático construído em Prolog para este compilador e os resultados animadores dos testes realizados com o protocolo Bit-alternante especificado em Estelle, justificam o desenvolvimento da totalidade do compilador. Para tal, deve ser utilizado um interpretador Prolog mais eficiente, que deverá estar disponível numa máquina de maior porte.

6. Referências

[ChLe 73]: C. Chang, R.C.Lee, 'Symbolic Logic and Mechanical Theorem Proving', Academic Press, 1973.

[Clar 84]: K.L.Clark, 'Micro-PROLOG: Programming in Logic', Prentice - Hall Internacional, 1984.

[ClMe 81]: W.F.Clocksin, C.S. Mellish, 'Programming in Prolog', Springer - Verlag, 1981.

[GeBo 83]: G.W.Gerber, G.v. Bochmann, 'A Parser for an FDT Language', Université de Montréal, 1983.

[Gerb 83]: G.W. Gerber, 'Une Methode d'Implantation Automatisée de Systémes Spécifiés Formellement', Université de Montréal,1983, 89-107.

[ISO 83]: ISO TC 97/SC16/WG1 Subgroup B, 'A FDT based on an extended state transition model', 1983.

[JeWi 74]: Jensen and Wirth, 'Pascal: User manual and report', Springer-Verlag, 1974.

[Kowa 79]: R. Kowalski, 'Logic for Problem Solving', Elsevier North - Holland, 1979.

[Li 84]: D.Li, 'A Prolog Database System', Research Studies Press, 1984.

[Lopes 85]: W. Lopes de Souza, 'Utilização dos conceitos de Módulo, Porta e Canal em Especificações Formais de Serviços, Protocolos e Interfaces de Comunicação', anais do 3º Simpósio Brasileiro sobre Redes de Computadores, Rio de Janeiro, 1985, 25.1-25.23.

[Warr 80]: d.M.D. Warren, 'Logic Programming and Compiler Writing', Software - Pratice and Experience, vol. 10, 97 - 125,John Wiley Sons, 1980.

[Wirt 76]: N. Wirth, 'Algorithms + Data Structures = Programs', Prentice-Hall, 1976, 320-330.

Anexo I

Especificação do protocolo Bit-Alternante em Estelle

```
const
  retran_time = 10;
  empty = 0;
  null = 0;
type
  data_type = ... ;
  seq_type = ... ; (* para o bit alternante, usar 0..1 *)
  id_type = (DATA, ACK);
  timer_type = (retransmit);
  ndata_type = record
                 id : id_type;
                 data : data_type;
                 seq : seq_type
               end;
  msg_type = record
               msgdata : data_type;
               msgseq : seq_type
             end;
  buffer_type = ... ;
  int_type = ... ; (* deve ser um inteiro *)
(* definições dos canais *)
channel U_access_point(User, Provider);
  by User :
       SEND_request(UData : data_type);
       RECEIVE_request;
  by Provider :
       RECEIVE_response(UData : data_type);
channel S_access_point(User, Provider);
```

```
  by User :
       Timer_request (Name : timer_type; Time : int_type);
  by Provider :
       Timer_response (Name : timer_type);
channel N_access_point (User, Provider);
  by User :
       Data_request(NData : ndata_type);
  by Provider :
       Data_response (NData : ndata_type);
module Alternating_Bit(U : U_access_point (Provider)common queue;
                       N : N_access_point(User)common queue;
                       S : S_access_point(User)individual queue);
var
  send_seq : seq_type;
  recv_seq : seq_type;
  send_buffer : buffer_type;
  recv_buffer : buffer_type;
  p,q : msg_type;
state : (ACK_WAIT, ESTAB);
        EITHER = [ACK_W_AIT, ESTAB];
predicate Ack_OK;
  begin
    Ack_OK := (NData.id = ACK) and (NData_seq = send.seq)
  end;
procedure send_data (msg : msg_type);
var s : ndata_type;
begin
  s.id := DATA;
  s.data := msg.msgdata;
  s.seq := msg.msgseq;
  out N.DATA_request (s)
```

```
end;
procedure send_ack(msg : msg_type);
var a : ndata_type;
begin
  a.id := ACK;
  a.data := null;
  a.seq := msg.msgseq;
  out N.DATA_request (a)
end;
procedure deliver_data (msg : msg_type);
begin
  out U.RECEIVE_response (msg.msgdata)
end;
procedure store (var buf : buffer_type; msg : msg_type);
primitive;
procedure remove(var buf : buffer_type; msg : msg_type);
primitive;
function retrieve(buf : buffer_type) : msg_type;
primitive;
procedure inc_send_seq;
begin
  send_seq := (send_seq + 1) mod 2
end;
procedure inc_recv_seq;
begin
  recv_seq := (recv_seq + 1) mod 2
end;
initialize
  begin
    state to ESTAB;
    send_seq := 0;
```

```
    recv_seq := 0;
    send_buffer := empty;
    recv_buffer := empty
  end;
(* transições *)
trans
  from ESTAB to ACK_WAIT when U.SEND_request
    begin
      p.msgdata := UData;
      p.msgseq := send_seq;
      store (send_buffer,p);
      send_data (p);
      out S.TIMER_request (retransmit, retran_time)
    end;
  from ACK_WAIT to ACK_WAIT when S.TIMER_response
  provided Name = retransmit
    begin
      p := retrieve (send_buffer);
      send_data (p);
      out S.TIMER_request (retransmit, retran_time)
    end;
  from ACK_WAIT to ESTAB when N.DATA_response
  provided Ack_OK
    begin
      remove(send_buffer, NData.msg);
      incr_send_seq
    end;
  from ESTAB to ESTAB when S.TIMER_response
  provided Name = retransmit
    begin
      (* nada é realizado *)
```

```
    end;
  from EITHER to SAME when N.DATA_response
  provided NData.id = DATA
    begin
      q.msgdata := NData.data;
      q.msgseq := NData.seq;
      send_ack(q);
      if NData.seq = recv_seq
        then
          begin
            store(recv_buffer, q);
            incr_recv_seq
          end
    end;
  from EITHER to SAME when U.RECEIVE_request
  provided not buffer_empty (recv_buffer)
    begin
      q := retrieve(recv_buffer);
      deliver_data(q);
      remove(recv_buffer, q.msg)
    end;
module Timer(S : S_access_point(Provider) individual queue);
var
  timervalue : arrey [timer_type] of integer;
  next_timer_value : array [timer_type] of integer;
  index : timer_type;
initialize
  begin
    for index  := retransmit to retransmit do
    (* index deve assumir todos os valores possíveis de timer_type *)
```

```
        timervalue [index] := 0;
        next_timer_value [index] := 0
      end
    end;
trans
  when S.Timer_request
    begin
      timervalue [Name] := 0; (* cancela o timervalue precidente *)
      next timervalue [Name] := Time
    end;
trans
  any timer_index : timer_type do
    provided next_timer_value [timer_index]> 0
    no delay
      begin
        timervalue [timer_index ] := 0;
        next_timervalue [timer_index] := 0
      end;
trans
  any timer_index : timer_type do
    provided timervalue [timer_index]> 0
    delay (timervalue [timer_index])
      begin
        timervalue [timer_index] := 0;
        out S.Timer_response [timer_index]
      end;
```

Anexo II

Sintaxe aceita pelo analisador sintático

```
<axiom> = <seqsect> .
<seqsect> = <section> ";" <seqsect> / empty.
<section> = <channel> / <module> / <process> / <refinemt> .
<channel> = <constd> <typed> "channel" <ident>
            "(" <rolelist> ")" ";" <byclause> "end" <ident> .
<rolelist> = <ident> <seqident> .
<seqident> = "," <rolelist> / empty.
<byclause> = "by" <rolelist> ":" <signal> <byclause> / empty.
<signal> = <ident> <signalpara> ";" <signal> / empty.
<signalpara> = "(" <paradef> ")" / empty.
<seqparadef> = ";" <paradef> / empty.
<paradef> = <rolelist> ":" <ident> <seqparadef> .
<module> = "module" <ident> ";" <portlist> "end" <ident> .
<portlist> = <rolelist> ":" <array> <ident> "(" <ident> ")" ";"
             <portlist> / empty.
<array> = "array" "[" <indextype> <seqindext> "]" "of" / empty.
<indextype> = <simpletype> .
<seqindext> = "," <indextype> <seqindext> / empty.
<refinemt> = "refinement" <ident> <signalpara> "for" <ident>
             ";" <refbody>"end" <ident> .
<refbody> = <seqsect> <instance> <intconnec> <extconnec> / empty.
<instance> = <rolelist> ":" <ident> "with" <ident> <lparacint>
              ";"<seqinst> .
<seqinst> = <instance> / empty.
<intconnec> = "internal" "connection" <connectn> / empty.
<extconnec> = "external" "connection" <connectn> / empty.
<portspec> = <ident> "." <ident> <optindex> .
<connectn> = <portspec> "=" <portspec> ";" <seqconnectn>.
```

```
<seqconnectn> = <connectn> / empty.

<optindex> = "[" <constant> <listconst> "]"  / empty.

<process> = "process" <ident> <signalpara> "for" <ident>  ";"
            <procbody> "end" <ident> .

<qchannel> = "queued" <rolelist> ";" / empty.

<procbody> =  <qchannel> <constd> <typed> <pvard> <init>
              <procfuncd><trans> / empty.

<pvard> = "var" <procvar> / empty.

<procvar> = "state" ":" "(" <rolelist> ")" ";" <seqvardecl>
            / <vardecl> .

<stateset> = <ident> "=" "[" <seqsetint> "]" ";" <stateset>
             / empty.

<init> = "initialize" <stateset> "begin" <initstatmt> <seqstatmt>
         "end" ";" / empty.

<initstatmt> = "state" ":=" <ident> / <plainstatmt> .

<trans> =  "trans" <seqclause> <opttrans> .

<opttrans> = <trans> / empty.

<seqclause> = <clause> <seqclause>
              / <opttag> <block> ";" <seqtrans> .

<clause> = "any" <paradef> "do" / "with" <variable> "do"
           /"when" <ident> <vparam> "." <ident>
           /"from" <rolelist> /"to" <nextmstate>
           /"save" <ident> <vparam> "." <ident>
           /"provided" <expression> / "priority" <idorint> .

<seqtrans> = <seqclause> / empty.

<opttag> = <ident> ":" / empty.

<vparam> = "[" <constant> <listconst>"]" / empty.

<listvariable> = "," <variable> / empty.

<nextmstate> = <rolelist> / "same".

<idorint> = <ident> /<integer> .

<block> = <labeld> <constd> <typed> <vard> <procfuncd>
          "begin" <statmt> <seqstatmt> "end".

<labeld> = "label" <integer> <seqinteger> ";" / empty.

<seqinteger> = "," <integer> <seqinteger> / empty.
```

```
<constd> = "const" <defconst> /empty.

<defconst> = <ident> "=" <constant> ";" <seqdefconst> .

<seqdefconst> = <defconst> / empty.

<constant> = <optsign> <numconst> / <string> .

<sign> = "+" / "-".

<optsign> =<sign> / empty.

<numconst> = <integer> / <real> / <ident> .

<typed> = "type" <deftype> / empty.

<deftype> = <ident> "=" <type> ";" <seqdeftype> .

<seqdeftype> = <deftype> / empty.

<type> =<simpletype> / <optpack> <typstruct> / "^" <ident> .

<simpletype> = "(" <rolelist> ")"
               / <sign><numconst> ".." <constant>
               / <string> ".." <constant>
               / <integer> ".." <constant>
               / <ident> <optconst> .

<optconst> = ".." <constant> / empty.

<optpack> = "packed" / empty.

<typstruct> = "array" "[" <simpletype><seqsimplet> "]" "of" <type>
              /"record" <field> "end"
              /"set" "of" <simpletype>
              /"file" "of" <type> .

<seqsimplet> = "," <simpletype> <seqsimplet> / empty.

<field> = <fixedpart> <seqfield>
          /"case" <ident> <typselect> "of" <variant> .

<fixedpart> = <rolelist> ":" <type> / empty.

<seqfield> = ";" <field> / empty.

<typselect> = ";" <ident> / empty.

<variant> = <constant> <listconst> ":" "(" <field>")"
            <seqvariant> /empty.

<seqvariant> = ";" <variant> / empty.

<listconst> = "," <constant> <listconst> / empty.

<vard> = "var" <vardecl> / empty.

<vardecl> = <rolelist>":" <type> ";" <seqvardecl> .
```

```
<seqvardecl> = <vardecl> / empty.

<procfuncd> = <pfheader> ";" <pfbody> ";" <procfuncd> / empty.

<pfheader> =  "procedure" <ident> <lpara>
            /"predicate" <ident> <lpara>
            /"function"  <ident> <lpara>":" <ident> .

<pfbody> = <block> / "external" / "forward" / "primitive" /"...".

<lpara> = "(" <spara> <seqspara> ")"  / empty.

<seqspara> = ";" <spara> <seqspara> / empty.

<spara> = <rolelist> ":" <ident>
          /"var" <rolelist> ":" <ident>
          /"procedure" <ident> <lpara>
          /"function" <ident> <lpara> ":" <ident> .

<factor> = <real> / <string> / <integer> / "..." / "nil"
           /"|" <seqsetint> "|" / "(" <expression> ")"
           /"not" <factor> /<ident> <seqfactid> .

<seqfactid> = <lseqvaria> / "(" <index> ")".

<index> = <expression> <seqindex> .

<seqindex> =  "," <index> / empty.

<lseqvaria> = "[" <index> "]" <lseqvaria>
              /"." <ident> <lseqvaria>
              /"^" <lseqvaria>
              / empty.

<seqsetint> = <setint> <lseqset> / empty.

<lseqset> = "," <setint> <lseqset> / empty.

<setint> = <expression> <seqxset> .

<seqxset> = ".." <expression> / empty.

<term> = <factor> <seqfact> .

<seqfact> = <opermult> <term> / empty.

<opermult> = "*" / "/" / "div" / "mod" / "and".

<simplexp> = <optsign> <term> <seqterm> .

<seqterm> = <operadd> <term> <seqterm> / empty.

<operadd> = "+" / "-" / "or".

<expression> = <simplexp> <seqsimplexp> .

<seqsimplexp> = <operel> <simplexp> /  empty.
```

```
<operel> = "=" / "< " / " >" / "<>" / "<=" / ">=" / "in".

<statmt> = <integer> ":" <plainstatmt> / <plainstatmt>.

<plainstatmt> = "goto" <integer> / <ident> <appendix>
                /"out" <ident> <seqind> "." <ident><lparacint>
                /"nextstate" <newstate>
                /"begin" <statmt> <seqstatmt> "end"
                /"if" <expression> "then" <statmt> <optelse>
                /"case" <expression>"of" <case> <seqcase> "end"
                /"repeat" <statmt> <seqstatmt>
                "until" <expression>
                /"while" <expression> "do" <statmt>
                /"for" <ident> ":=" <expression><direction>
                <expression> "do"<statmt>
                /"with" <variable> "do" <statmt>
                / empty.

<lparacint> = "(" <index> ")" / empty.

<newstate> = <ident> / "same".

<seqstatmt> = ";" <statmt> <seqstatmt> / empty.

<optelse> = "else"  <statmt> / empty.

<seqcase> = ";"  <case><seqcase>  / empty.

<case> = <constant><listconst> ":" <statmt> / empty.

<direction> = "to" / "downto".

<variable> = <ident> <lseqvaria> <listvariable?.

<appendix> =<lparacint> / <lseqvaria> ":=" <expression>.

<seqind> = "[" <index> "]" <seqind> / empty.
```