Análise Comparativa de Linguagens de Especificação de Unidades de Controlo Digitais

Andreia Melo, Valery Sklyarov

***Abstract* – The paper analyses and compares the most widely used textual and graphical languages for formal specification of digital control units. It demonstrates that graphical languages are more adequate for manual specification and usually it is easier to learn them. As a rule, commercially available tools for the design of digital systems support different languages.**

***Resumo* – Este artigo analisa e compara as linguagens formais gráficas e textuais mais utilizadas na especificação de unidades de controlo digitais. É demonstrado neste artigo que as linguagens gráficas são preferencialmente adoptadas na especificação manual e normalmente de mais fácil aprendizagem. As ferramentas de projecto de sistemas digitais disponíveis actualmente no mercado combinam duas linguagens, uma textual e outra gráfica.**

I. INTRODUÇÃO

As unidades de controlo são circuitos digitais sequenciais que reagem a um determinado conjunto de sinais de entrada actuando num conjunto de sinais de saída. A unidade de controlo constitui apenas uma parte de um sistema pois interage através de sinais binários com a unidade operacional do mesmo, como se pode ver na figura 1.

A unidade operacional é composta por elementos estáticos (memórias, somadores, contadores, descodificadores, etc) cuja operação é determinada pela sequência de sinais produzida pela unidade de controlo. A esta sequência que define o comportamento da unidade chama-se *algoritmo de controlo.* Os sinais de saída da unidade de controlo produzem operações elementares no processamento da informação pela unidade operacional.

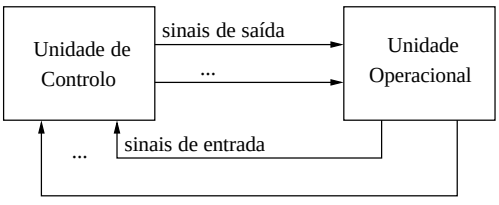

Figura 1 - Interacção entre uma unidade de controlo e uma unidade operacional.

A função de uma linguagem de especificação é descrever algoritmos de controlo. A linguagem deve produzir uma descrição clara, precisa e concisa. O algoritmo deve ser escrito completa e correctamente e de forma compreensível para que seja possível efectuar futuras modificações.

Nas várias secções deste artigo são apresentadas algumas linguagens de especificação, nomeadamente os Diagramas de Transição de Estados, PRALU, Statecharts, SDL, HGS e VHDL. Na secção VIII é efectuada uma análise comparativa das várias linguagens e na secção IX o artigo termina com algumas conclusões.

II. DIAGRAMAS DE TRANSIÇÃO DE ESTADOS

A Máquina de Estados Finitos (MEF) é o modelo normalmente utilizado para descrever o comportamento de uma unidade de controlo simples. Para usar este modelo é necessário identificar os estados da unidade de controlo. Um estado representa uma resposta da unidade aos estímulos fornecidos num dado intervalo de tempo. Devem ser definidas as transições entre estados, que podem ocorrer, dependendo ou não, de algumas condições previstas no algoritmo. Estas condições normalmente estão associadas aos estímulos recebidos da unidade operacional. Finalmente, é necessário definir os instantes de tempo onde devem ser actuados os sinais de saída.

Uma MEF é formalmente representada pelo 6-tuplo $< S, I, O, f, h, s_0 >$, onde:

- $S = \{s_0, ..., s_N\}$ é o conjunto finito dos estados da máquina, sendo s_0 o estado inicial
- $I = \{i_1, ..., i_M\}$ é o conjunto finito de entradas
- $O = \{o_1, ..., o_L\}$ é o conjunto finito de saídas
- $f : S \times I \rightarrow S$ é a função de transição que determina o estado seguinte a partir do estado actual e das entradas
- h é a função que determina as saídas da máquina; no caso da máquina de Moore as saídas dependem apenas do estado actual e por isso a função define-se $h : S \rightarrow O$; nas máquinas de Mealy as saídas dependem do estado actual e das entradas, logo a função define-se $h : S \times I \rightarrow O$

Os Diagramas de Transição de Estados são a linguagem gráfica vulgarmente utilizada para descrever uma MEF. Os estados são nós representados graficamente por circunferências ou elipses. Opcionalmente podem ser associadas etiquetas de identificação aos estados que podem ser colocadas dentro de cada nó. O estado inicial é representado por uma dupla circunferência. As transições são representadas por arcos direccionados onde são indicados o(s) valor(es) da(s) entrada(s) responsáveis pela transição.

A atribuição de valores aos sinais de saída pode ser associada a um estado ou a uma transição. No primeiro caso

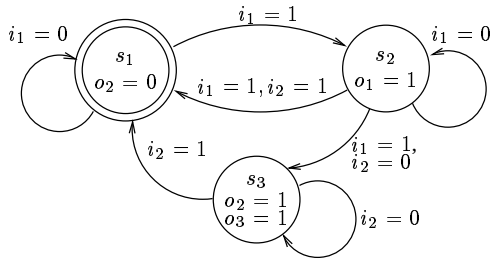

Figura 2 - Diagrama de Transição de Estados.

os valores de saída devem ser colocados dentro do nó. No segundo caso, os valores devem ser indicados perto do arco que representa a transição onde deve ser realizada a atribuição.

Na figura 2 está representado um exemplo de diagrama de transição de estados composto por três estados, s_1, s_2 e s_3, sendo s_1 o estado inicial. As transições dependem dos valores das entradas i_1 e i_2. Os sinais de saída são o_1, o_2 e o_3, que neste caso são activados dentro dos estados.

III. PRALU

A linguagem PRALU dedica-se à descrição de algoritmos de controlo assíncronos e paralelos [1]. Um algoritmo é um conjunto não ordenado de cadeias α_i que podem ser executadas sequencial ou paralelamente usando um mecanismo de controlo do tipo rede de Petri.

Uma cadeia é uma expressão do tipo $u_i : -e_i\ a_i \rightarrow v_i$. u_i e v_i são subconjuntos de $M = \{1, 2, ..., m\}$ e representam as etiquetas de início e de fim da cadeia. Os elementos de M são os *estados locais*. O *estado global* do algoritmo de controlo é o conjunto de estados locais activos num dado instante de tempo. v_i especifica o conjunto de estados locais para onde o algoritmo transita depois da cadeia terminar.

$-e_i$ é uma conjunção de *operações de espera* (condições). e_i pode ser omitida se tomar o valor 1. As operações de espera terminam quando ocorre um dado evento. Se numa cadeia é necessário esperar pelo valor lógico 1 da condição, deve-se escrever $-e$, caso contrário deve-se escrever $-e'$.

a_i é uma sequência de *operações de actuação* que alteram o estado do sistema. Quando é do tipo $\rightarrow A \rightarrow B \rightarrow C \rightarrow ...$ significa que A, B e C são executadas sequencialmente e segundo a ordem indicada. Se a_i for do tipo $\rightarrow AB \rightarrow C \rightarrow ...$ significa que A e B são executadas em paralelo e quando ambas terminarem é executada C.

A *operação de transição* $\rightarrow v_i$ define a transição para a cadeia com a etiqueta v_i.

O conjunto $N_t \subseteq M$ contém os estados ou as etiquetas de início activas no instante de tempo t. Inicialmente deve ter apenas um valor que indica o estado global inicial do algoritmo.

Uma cadeia α_i é iniciada quando $u_i \subseteq N_t$ e $e_i = 1$ no instante de tempo t. Nesse instante u_i é removido de N_t, é executada a sequência a_i e a operação de transição $\rightarrow v_i$. Finalmente v_i é adicionada a N. Após a execução de α_i, $N = (N_t \setminus u_i) \cup v_i$.

As cadeias podem ser executadas sequencial ou paralelamente se uma operação de espera for verdadeira para mais do que uma cadeia. Se existirem cadeias com a mesma etiqueta inicial as operações de espera têm de ser mutuamente exclusivas. O algoritmo termina quando todas as cadeias ficarem passivas, ou seja, quando N possuir um valor terminal.

Esta linguagem permite quatro tipos de construções. A *derivação concorrente* é uma construção onde no final de uma cadeia são iniciadas simultaneamente duas cadeias diferentes, como por exemplo, $1 : -u \rightarrow AB \rightarrow 2.3$.

A *derivação alternativa* existe numa cadeia com condições de espera mutuamente exclusivas, como é o caso das cadeias $2 : -v'w \rightarrow A \rightarrow 4$ e $2 : -v \rightarrow A'B \rightarrow 5$. Após a execução da cadeia 2 é executada a 4 ou a 5.

A *convergência de derivação concorrente* ocorre quando duas cadeias estão activas simultaneamente e as suas operações de transição provocam o início de apenas uma cadeia, como por exemplo, $1 : -v'w \rightarrow A \rightarrow 3$, $2 : -v' \rightarrow B \rightarrow 4$ e $3.4 : -w \rightarrow A \rightarrow 5$.

Finalmente, a *convergência de derivação alternativa* acontece quando duas cadeias resultantes de uma derivação alternativa convergem para uma cadeia com a mesma etiqueta, como é o caso das cadeias $1 : -u'v' \rightarrow A \rightarrow 3$ e $2 : -uv \rightarrow B \rightarrow 3$.

As regras que garantem a correcção de um algoritmo descrito em PRALU e os mecanismos de verificação estão descritos em [2].

IV. STATECHARTS

A linguagem Statecharts [3] é utilizada na especificação de sistemas reactivos e embutidos. Tal como os Diagramas de Transição de Estados, os Statecharts baseiam-se na definição de estados. No entanto, aqui os estados podem estar encapsulados dentro de outros, criando-se a noção de hierarquia.

Quando um estado possui sub-estados descreve um tipo de execução concorrente ou ortogonal onde o controlo consiste na activação de apenas um sub-estado num dado instante de tempo. A ortogonalidade é a situação onde dois ou mais sub-estados estão relacionados pela operação AND, isto é, estão activos simultaneamente. Os componentes ortogonais podem ser sincronizados apenas por eventos comuns e podem afectar-se mutuamente apenas através de condições. Os estados que estão ligados entre si descrevem um tipo de execução sequencial. Neste caso diz-se que estão relacionados pela operação OR porque apenas um deles deve estar activo num dado instante de tempo. As operações AND e OR distinguem-se graficamente por uma linha a tracejado que separa os estados afectados pela primeira operação. Por defeito, quando não existe linha, os estados são afectados pela segunda operação.

Graficamente, um estado é representado por um rectân-

gulo de cantos arredondados. Um Statechart possui sempre um estado inicial por defeito. O estado inicial é identificado por uma seta que liga um ponto ao rectângulo associado a esse estado. O sistema descreve-se à custa das transições entre estados que graficamente são representadas por arcos direccionados.

Cada arco é etiquetado com um par *condição-acção* c/a. A condição de activação c é o predicado para a activação do estado e activação das saídas. Quando uma condição associada a uma transição toma o valor verdadeiro a transição realiza-se executando imediatamente a acção associada e transferindo o controlo para o estado alvo da transição. A acção pode ser nula ou uma sequência de uma ou mais acções primitivas, sendo uma acção primitiva uma activação de um estado ou de uma variável do sistema. A acção pode ser um evento externo ou interno, que neste caso pode ser utilizado para sincronizar outras transições entre estados ortogonais [4].

Os Statecharts assumem a existência de um relógio global que fornece informação temporal para sincronização e *time-outs*, ao qual todos os estados têm acesso. A semântica é definida por sequências temporais de alterações que ocorrem no sistema. As alterações correspondem às transições onde são gerados os eventos.

Quando é realizada uma transição a acção associada pode activar outras transições. Desta forma, a semântica pode conduzir a anomalias resultantes de conflitos entre reacções em cadeia. Estes conflitos podem ocorrer devido à possibilidade de actuação simultânea sobre variáveis partilhadas nas transições. Como os resultados são difundidos no sistema pode ocorrer um conflito de valores.

Os Statecharts dispõem de vários mecanismos de sincronização. Supondo que é especificado um estado que engloba um conjunto de estados, sendo um deles o estado inicial, um evento que seja especificado graficamente por uma seta que termine na caixa do estado englobante provoca a sincronização desse estado. Isto resulta na activação do estado inicial do estado englobante.

Outro exemplo de sincronização é a utilização de uma variável interna para actuar como evento com o objectivo de sincronizar outras transições.

Na figura 3 encontra-se um exemplo de um diagrama de um algoritmo de controlo descrito em Statecharts.

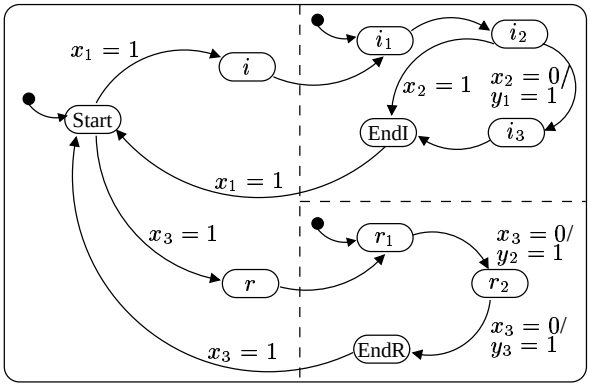

Figura 3 - Satecharts.

V. SDL

O desenvolvimento da linguagem SDL (Specification and Description Language) iniciou-se em 1972 pelo CCITT (International Telegraph and Telephone Consultative Committee) para especificar e descrever especificamente sistemas de telecomunicações, incluindo a comunicação de dados. Actualmente pode ser aplicada a sistemas interactivos e de tempo-real, a comunicações de satélite, equipamento médico, sistemas de controlo e protocolos de comunicação em automóveis.

A SDL permite descrever formalmente a arquitectura e o comportamento de sistemas [5] que são estruturados hierarquicamente em agentes ligados por canais. Existem dois tipos de agentes: agentes de bloco e agentes de processo. Os primeiros executam de forma concorrente e os segundos de forma interlaçada. Os agentes podem ser decompostos em sub-agentes. O comportamento dinâmico de um agente consiste em acções internas e troca de mensagens discretas (sinais) com outros agentes e com o ambiente onde o sistema está integrado. Para simplificar a terminologia daqui em diante os agentes de bloco serão chamados apenas de *blocos* e os agentes de processo por *processos*.

Um sistema é composto por um ou mais blocos ligados entre si e à fronteira que existe entre o sistema e o ambiente. As ligações são estabelecidas por canais. Um bloco pode ser decomposto em sub-blocos e canais resultando numa estrutura em árvore. Apenas os blocos simples, que não podem ser decompostos, é que contêm processos (ver figura 4).

Graficamente o sistema é representado pelo *diagrama do sistema* (ver figura 5) que é constituído por um nome, pela descrição dos sinais entre blocos e entre os blocos e o ambiente, pela descrição dos canais que interligam blocos e que ligam os blocos ao ambiente, pela descrição dos tipos de dados e dos blocos que compõem o sistema.

A descrição de um canal é composta pelo seu nome, por

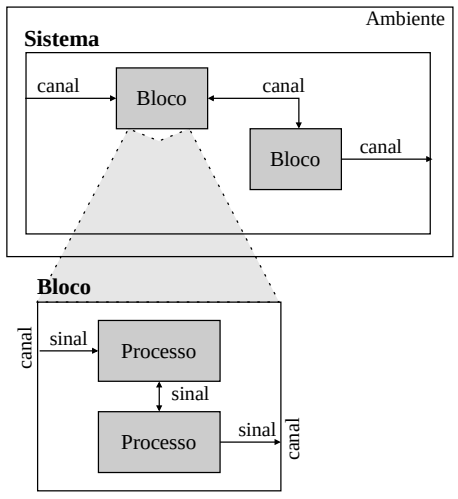

Figura 4 - SDL: Estrutura estática de um sistema.

uma lista com os nomes dos sinais que o canal transporta e pela identificação do local de terminação do canal (que pode ser um bloco ou o ambiente). O canal é representado por uma linha com uma seta a meio indicando o sentido do fluxo dos sinais. Se o canal for bidireccional devem ser colocadas setas nos dois sentidos. Para evitar ambiguidades o nome do canal deve ficar perto do respectivo símbolo. A lista dos sinais associados ao canal é colocada perto do símbolo e entre parêntesis rectos (ver na figura 5, por exemplo, $canal_2[s_1]$).

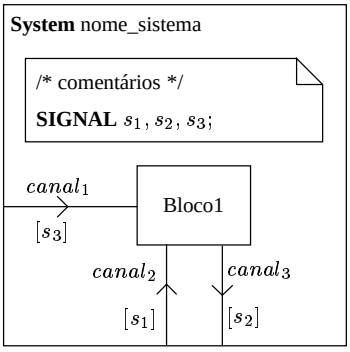

Figura 5 - SDL: diagrama de sistema.

Um bloco representa-se pelo *diagrama de bloco* (figura 6). No diagrama deve estar presente o nome do bloco, a descrição dos sinais internos ao bloco (por exemplo s_1, s_2, s_3) e a descrição dos percursos dos sinais que interligam os processos e que ligam o bloco ao ambiente. O símbolo que representa o percurso de um sinal é semelhante ao de um canal. A diferença é que a seta deve ficar colocada na extremidade da linha (ver pe_1 a pe_5). O diagrama deve incluir também as ligações dos canais internos e externos ao bloco. As ligações aos canais são definidas colocando o nome do canal junto à moldura do bloco, no exterior, suficientemente perto da extremidade do símbolo que representa o percurso de um sinal (ver as ligações aos canais $canal_1$, $canal_2$ e $canal_3$). Finalmente, o diagrama deve possuir a especificação dos processos que implementam o comportamento do bloco (ver P_1 e P_2).

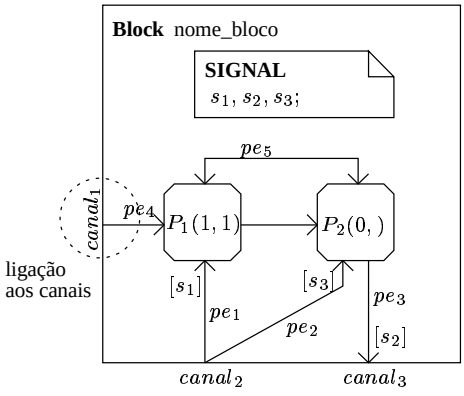

Figura 6 - SDL: diagrama de bloco.

Tal como os sistemas e os blocos, os processos também

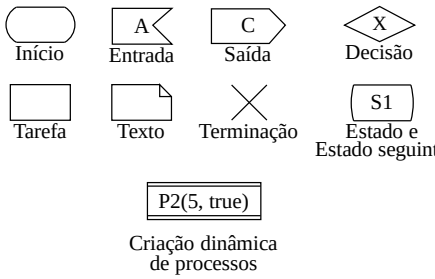

Figura 7 - SDL: Símbolos utilizados para descrever um processo.

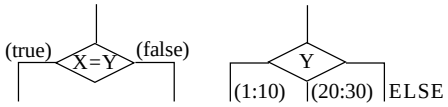

Figura 8 - SDL: Exemplos de utilização de um símbolo de decisão.

são descritos por *diagramas de processos*. Um diagrama deste tipo deve incluir o nome do processo, os parâmetros formais, as descrições das variáveis, as descrições dos temporizadores, a descrição do procedimento e do grafo do processo que descreve a máquina de estados finitos associada.

Um processo é composto por nove símbolos básicos, representados na figura 7. Os símbolos *início* e *terminação* servem para identificar onde começa e termina, respetivamente, o comportamento do processo. O símbolo *texto* é usado para declarar ou inicializar variáveis locais. A *tarefa* é um símbolo utilizado para especificar a manipulação dessas variáveis. Na *entrada*, *saída* e *estado/estado seguinte* são colocadas variáveis do respectivo tipo. A *decisão* serve para especificar o nome de uma variável da qual depende uma dada transição de estado. Este símbolo pode ter várias saídas, às quais se associam gamas de valores que a variável pode assumir. Na figura 8 estão representadas algumas maneiras de utilizar este símbolo. Finalmente, existe um símbolo específico para a criação dinâmica de um processo.

A transferência de dados entre processos pode ser realizada através de sinais (ver figura 9). Para este efeito deve existir um símbolo de saída e um símbolo de entrada correspondente especificando tipos compatíveis de variáveis. A ordem pela qual os valores são enviados num sinal depende da ordem como são definidos. Uma ou mais variáveis podem ser omitidas quer num símbolo de entrada quer num símbolo de saída. Neste caso esses valores não são recebidos nem enviados, respectivamente.

Em SDL um processo é uma máquina de estados finitos que pode manipular dados guardados em variáveis locais à máquina. Um processo está sempre associado a um tipo. Num sistema podem existir vários processos do mesmo tipo. Os processos podem ser criados estática ou dinamicamente por outro processo (processo-pai). No diagrama de bloco (figura 6) os processos são especificados com dois parâmetros. O primeiro indica o número de processos desse tipo que existem inicialmente e o segundo indica o número máximo de processos que podem existir.

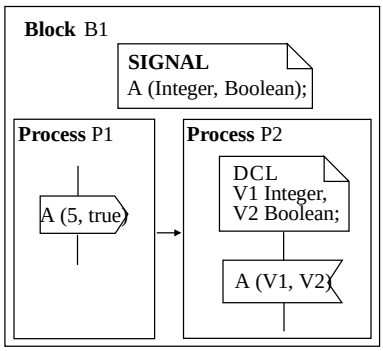

Figura 9 - SDL: Transferência de sinais entre processos.

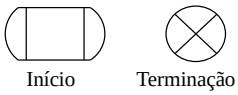

Figura 10 - SDL: Símbolos utilizados num procedimento.

Se este último for omitido significa que pode existir um número ilimitado de processos desse tipo.

Quer o processo criador quer o processo criado devem pertencer ao mesmo bloco. A terminação de um processo só pode ser feita pelo próprio.

Um processo pode incluir a descrição de um procedimento, que por sua vez também pode incluir outro procedimento. Um procedimento não é mais do que uma máquina de estados finitos dentro de um processo e graficamente é descrito como tal. A única diferença é a representação gráfica dos símbolos de início e de terminação, que estão representados na figura 10.

Um procedimento partilha o mesmo conjunto de sinais de entrada do processo a que pertence. No entanto, o seu conjunto de estados é independente dos do processo. Quando um procedimento executa, o processo ou o procedimento que o invocou é suspenso na transição que contém a chamada. Esta transição continua quando o procedimento chamado termina a sua execução.

A comunicação entre processos é realizada à custa da atribuição de endereços únicos a cada processo. Os endereços não são atribuídos pelo utilizador mas por uma máquina SDL abstracta no momento da criação do processo.

VI. HGS

Em 1958 Yanov [6] propôs a linguagem LS (Logic-Schemes) para descrever algoritmos de controlo. Posteriormente, Baranov [7], [8] propôs os GS (Graph-Schemes) para representar graficamente a notação anterior. Esse trabalho serviu de base à criação de HGS (Hierarchical Graph-Schemes) por Sklyarov [9]. Nesta fase de desenvolvimento adicionou-se aos GS a modularidade e a hierarquia.

Considere-se que a unidade de controlo é constituída pelo conjunto $Y = \{y_1, ..., y_n\}$ de sinais de saída. Estes sinais são também denominados *micro-operações*. Activar a micro-operação $y_n (n = 1, ...N)$ significa actuar sobre o sinal tal que $y_n = 1$. As micro-operações podem ser executadas sequencial ou simultaneamente (no mesmo ciclo de relógio) na unidade operacional, dependendo do algoritmo implementado na unidade de controlo. As transições entre estados dependem dos sinais de entrada $X = \{x_1, ..., x_n\}$, os quais são denominados *condições binárias*.

O algoritmo de controlo é dividido em vários módulos, que podem ser descritos por GS ou HGS, com as alterações necessárias à interligação dos vários módulos. Um algoritmo de controlo é sempre composto por um módulo principal, onde se inicia a execução, e por um conjunto outros de módulos.

Com a introdução da hierarquia surge a possibilidade de um módulo poder invocar outro. Quando ocorre uma invocação hierárquica o módulo que é chamado deve ser executado até atingir o seu nó *End*. Neste momento o fluxo de execução do algoritmo de controlo deve voltar ao módulo invocador. A definição de HGS permite descrever algoritmos recursivos pois não restringe o tipo de módulo a invocar, logo está incluída a possibilidade de um módulo se invocar a ele próprio ou a outro que já esteja a ser executado.

Além das micro-operações e das condições binárias os HGS utilizam as *macro-operações* e as *funções lógicas* que especificam a invocação de módulos. As macro-operações são activadas dentro de um nó *operacional* e as funções lógicas são colocadas dentro de nós *condicionais*. A função lógica é diferente da macro-operação porque devolve um valor binário ao módulo que a invoca.

O modelo da MEF é a base comum a todas estas linguagens. No entanto, enquanto na MEF o processo de especificação se concentra na determinação dos estados do circuito de controlo, estas concentram-se na sequência de atribuições aos valores dos sinais de saída. O formalismo de um HGS pode ser descrito da seguinte maneira. Sendo um algoritmo de controlo composto pelo conjunto de módulos M, a função que faz a associação entre as macro-operações e as funções lógicas e os módulos por estas invocados é:

$$h : \bigcup_{m \in M} Z_m \cup F_m \rightarrow M$$

sendo Z e F o conjunto de macro-operações e de funções lógicas, respectivamente. O módulo principal, onde se inicia a execução, é $m_0 \in Z$.

Num HGS deve garantir-se que qualquer que seja o módulo então ele deve ser invocado, quer por módulos diferentes, quer por ele próprio. Para descrever matematicamente esta propriedade vamos primeiro definir a função $i : M \rightarrow M$ que representa as invocações hierárquicas de um módulo. Se um módulo o é descrito por um Esquema de Grafos que possui nós que especificam invocações hierárquicas, a expressão:

$$i(o) = D : o \in M, D \subset M$$

significa que o módulo o invoca um sub-conjunto de módulos D.

A invocação de todos os módulos é garantida pela seguinte expressão:

$$\bigcup_{m \in M} i(m) = M \backslash \{m_0\}$$

que diz que a união de todos os módulos invocados deve ser igual ao conjunto de todos os módulos, à excepção do módulo principal, pois este inicia a execução do algoritmo e nunca deve ser chamado por outro módulo.

Cada módulo é formalmente definido pelo tuplo $< X, Y, Z, F, O, C, A, \{Begin, End\}, f, g, t, u, n >$, onde:

- X é o conjunto de entradas
- Y é o conjunto de saídas
- Z é o conjunto de macro-operações
- F é o conjunto de funções lógicas
- O é o conjunto de nós *operacionais*
- C é o conjunto de nós *condicionais*
- A é o conjunto de nós de *atribuição*
  A cada nó deste tipo está associado um valor de retorno.
- $\{Begin, End\}$ é o conjunto de nós que delimitam o módulo. $Begin$ é o nó que define o ponto de partida da execução e End é o nó que define o ponto de terminação da execução do módulo.
- $f : O \cup A \cup \{Begin\} \rightarrow O \cup C \cup A \cup \{End\}$
  Esta função define os arcos que partem das saídas dos nós *operacionais*, de *atribuição* e do *Begin*. Se $f(a) = b$ então existe um arco que liga a saída do nó a, do tipo *operacional*, *atribuição* ou *Begin* à entrada do nó b.
- $g : C \times \{0, 1\} \rightarrow O \cup C \cup A \cup \{End\}$
  Esta função define todos os arcos que partem das saídas dos nós *condicionais*.
- $t : O \rightarrow Y \cup Z$
  A função *t* atribui a um nó *operacional* o conjunto de saídas e macro-operações que devem ser activadas nesse nó.
- $u : C \rightarrow X \cup F$
  A função u atribui a um nó *condicional* o sinal de entrada ou função lógica que deve ser testado por esse nó.
- $n : A \rightarrow \{0, 1\}$
  Esta função associa a cada nó de *atribuição* um valor de retorno.

Graficamente um HGS é composto por seis tipos de nós, como mostra a figura 11, ligados por arcos direccionados. Nos HGS os nós *operacionais* podem conter uma ou mais micro-operações e apenas uma macro-operação. Dentro dos nós *condicionais* é permitido colocar apenas uma condição binária ou uma função lógica. O retorno das funções lógicas faz-se pelos nós de *atribuição*, que devem estar ligados ao nó *End*.

A execução de um HGS inicia-se no nó *Begin* do módulo principal. O nó seguinte é dado por $f(Begin)$.

Chamando x ao nó actual, se este for um nó do tipo *operacional* o nó seguinte também é dado pela expressão

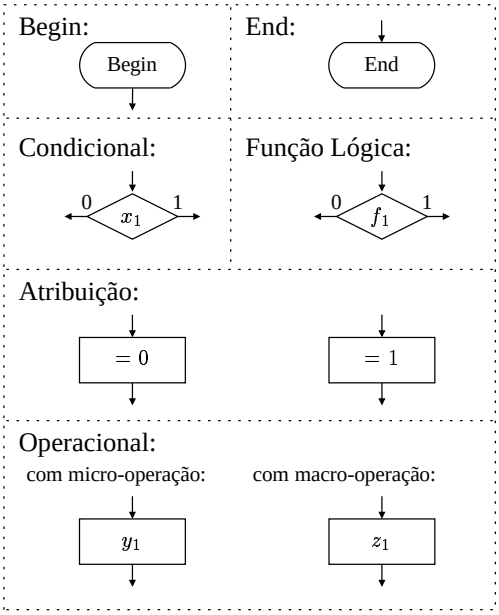

Figura 11 - HGS: Tipos de nós.

$f(x)$. Se x especificar apenas micro-operações então devem ser activadas as saídas dadas por $t(x)$. Se especificar uma macro-operação, isto é, se $t(x) \in Z$, o nó seguinte é o nó *Begin* do módulo que corresponde a essa macro-operação que é igual a $h(t(x))$. Quando terminar a execução do módulo $h(t(x))$, o fluxo de execução retoma ao módulo invocador. Quando isto acontece o nó seguinte é determinado por $f(x)$.

Se o nó *End* for o nó actual devem ser distinguidas duas situações. Por um lado, x pode pertencer ao módulo principal m_0, e então ou a execução do algoritmo de controlo termina ou recomeça no nó *Begin*, dependendo do tipo de implementação desejado. Por outro lado, se x pertence a qualquer outro módulo diferente de m_0, é realizado o retorno da invocação desse módulo.

No caso de um nó *condicional* ser o nó actual x o resultado de $u(x)$ deve ser testado. Se for uma condição binária, ou seja, se $u(x) \in X$, o nó seguinte é $g(x)$. Se for uma função lógica, ou seja, se $u(x) \in F$, o nó seguinte é o nó *Begin* do módulo $h(u(x))$, que inicia a sua execução. Quando este módulo terminar o fluxo de execução retoma ao módulo invocador cujo nó seguinte é dado por $g(x)$, tendo em conta o valor devolvido por $h(u(x))$.

Se o nó x for de *atribuição* significa que está a ser executada uma função lógica e que o sinal usado para fazer o retorno do valor da função irá tomar o valor $n(x)$ especificado neste nó. Em qualquer dos casos, valor 0 ou 1, o nó seguinte deve ser o nó *End* da função, resultante de $f(x)$.

Na figura 12 está representado um HGS composto por dois módulos, o módulo principal e uma macro-operação z_1.

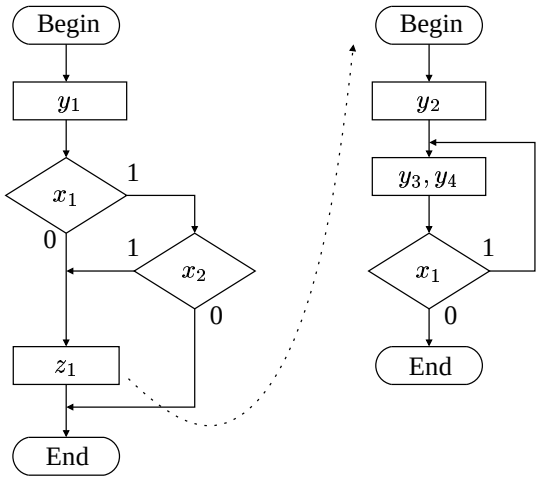

Figura 12 - HGS - módulo principal e uma macro-operação.

VII. HDL

Actualmente existem várias linguagens HDL (Hardware Description Languages) que permitem descrever sistemas digitais, incluindo unidades de controlo. As linguagens que têm sido adoptadas na descrição de circuitos digitais apresentam-se integradas nas ferramentas de síntese lógica disponíveis comercialmente e que dipõem de todos os requisitos de projecto de um circuito digital, desde a especificação até à implementação.

A Synplicity [10] desenvolveu a ferramenta Synplify Pro que inclui um explorador de MEF. A função desse explorador é detectar uma MEF na descrição do circuito que é construída pelo projectista. Várias codificações de estados são experimentadas pela aplicação que escolhe aquela que apresenta melhor desempenho de acordo com as restrições temporais especificadas. Graficamente as MEF são representadas por diagramas que derivam dos diagramas de transição de estados. Esta ferramenta inclui interfaces para simuladores de código HDL, nomeadamente o NC-Verilog, NC-VHDL, Active-VHDL, ModelSim e SpeedWave.

O StateCAD [11] fornecido pela Xilinx [12] é um editor gráfico de diagramas de estado. Inclui a ferramenta StateBench para gerar *test benches*, efectuar a verificação comportamental e *wizards*. Depois de validar um diagrama o StateCAD gera automaticamente código HDL que se pode simular e sintetizar directamente a partir do diagrama. O código HDL pode ser na linguagem VHDL, Verilog ou Abel. Este software permite detectar condições que não são usadas, bloqueio num dado estado, condições indeterminàveis, erro de síntaxe e porções incompletas do diagrama de estados.

A MentorGraphics [13] dispõe de uma ferramenta para projecto chamada Renoir Personal Designer que consiste num ambiente gráfico de depuração de HDL. Em vez do processo manual de produção de uma descrição do circuito em VHDL estrutural, o Renoir permite desenhar diagramas de blocos hierárquicos e gerar o respectivo código HDL automaticamente. Contudo, se for preferível a escrita de código HDL, a ferramenta é capaz de extrair daí informação de forma a construir diagramas de blocos e MEF.

O editor de diagramas captura a estrutura hieráquica usando blocos, segundo a metodologia *top-down*, ou componentes, segundo a metodologia *bottom-up*. Um bloco pode representar um diagrama de blocos, uma MEF ou código HDL. Os blocos podem ser ligados por sinais, barramentos ou *bundles*.

O editor de MEF baseia-se numa notação derivada dos diagramas de transição de estados. Suporta descrições hierárquicas e concorrentes, tipos de dados abstractos e interrupções. A marcação de estados pode ser efectuada manual ou automaticamente.

As linguagens mais utilizadas neste tipo de ambientes de projecto são o VHDL e o Verilog. São liguagens textuais e têm como objectivo descrever o comportamento ou a estrutura, ou ambos, de um sistema digital.

Uma linguagem deste tipo permite criar um modelo do comportamento esperado de um circuito antes de ser projectado e implementado. Esse modelo alimenta um simulador que permite verificar a correcção da solução escolhida. O código desse modelo normalmente é reutilizado para se obter uma descrição mais detalhada do circuito que é fornecida de seguida a um compilador lógico cuja saída servirá para configurar um dispositivo lógico programável com a função desejada.

Nesta secção escolheu-se o VHDL para exemplificar a descrição de uma unidade de controlo numa linguagem deste tipo. A linguagem VHDL foi desenvolvida por várias empresas, contratadas pelo Departamento de Defesa dos Estados Unidos da América, com o objectivo de descrever sistemas e componentes de hardware. A linguagem foi standardizada pelo IEEE em 1987.

A descrição de um componente é composta pela especificação da interface e da arquitectura. A especificação da interface é identificada pela palavra-chave **entity** e descreve todas as características externas do componente, nomeadamente os portos de entrada e de saída. A especificação da arquitectura é identificada pela palavra-chave **architecture** e descreve a funcionalidade do componente. Esta especificação pode ser comportamental ou estrutural. No primeiro caso, utilizando construções de programação e no segundo caso, utilizando componentes já existentes. Um componente, ou uma entidade, pode ser descrito por mais do que uma arquitectura.

A característica mais importante do VHDL que permite descrever um circuito de forma estrutural e hierárquica é a possibilidade de instanciação de vários componentes do mesmo tipo.

Para descrever o comportamento de uma entidade ou componente são usados processos (construção **process**) identificados por uma palavra-chave com o mesmo nome. Um processo está sempre activo e executa de forma concorrente com outros processos. Um processo é constituído por uma secção de declarações e uma secção de instruções. Na secção de declarações pode incluir uma

lista de sinais, aos quais é sensível, e que se chama *lista sensitiva*. A secção de instruções está sempre activa. Lá existe um conjunto de instruções que são executadas sequencialmente, em tempo zero, e que são disparadas por eventos que ocorrem nos sinais que fazem parte da lista sensitiva. Nesta secção podem ser invocadas funções e procedimentos e podem atribuir-se valores a sinais utilizando as construções *if*, *case* e os ciclos *for* e *while*.

A linguagem VHDL dispõe de duas formas para efectuar a transferência de valores entre pontos do programa, que são as variáveis e os sinais. Os sinais podem ser usados quer dentro de um bloco com características concorrentes quer dentro de um bloco com características sequenciais. Podem ser globais mas só podem ser declarados dentro de blocos concorrentes. Além dos sinais existem as variáveis que são normalmente utilizadas para guardar valores intermédios. Só podem ser declaradas dentro de um bloco sequencial e a sua visibilidade está restrita a esse bloco.

Os tipos de dados disponibilizados por esta linguagem são vários e apropriados a descrições comportamentais de alto-nível, nomeadamente *integer*, *real*, *array*, *pointer*, entre outros. Existem três tipos de operadores: lógicos, relacionais e aritméticos. O VHDL permite a redefinição de operadores.

A comunicação entre processos é realizada segundo um modelo de memória partilhada, baseada em sinais, cujas atribuições de valores podem ser feitas por qualquer processo, tornando-se visíveis a todos os outros processos.

A sincronização é feita de duas formas. A primeira, baseada na lista sensitiva do processo, garante que a execução do processo é iniciada quando ocorre um evento em pelo menos um dos sinais pertencentes à lista. A segunda, pela instrução **wait**, que suspende o processo até que ocorra um evento em pelo menos um dos sinais especificados naquela instrução ou até que ocorra uma determinada condição especificada.

Em código VHDL é possível descrever qualquer sistema descrito pelas linguagens apresentadas anteriormente. No entanto, é apresentado a seguir o código que descreve uma MEF. A MEF está encapsulada na entidade chamada *unidade_controlo*. É constituída por uma entrada INPUT, um vector com duas saídas OUTPUT e quatro estados. O processo SYNC_PROC realiza as transições de estado e a reinicialização da máquina. O processo COMB_PROC é sensível aos sinais de entrada e calcula o estado seguinte. A activação dos sinais de saída coincide com a activação do estado.

```
entity unidade_controlo is
    port(CLOCK, RESET: in STD_LOGIC;
    INPUT: in STD_LOGIC;
    OUTPUT: out STD_LOGIC_VECTOR(1 downto 0));
end unidade_controlo;

architecture BEHAV of unidade_controlo is

– definição do tipo de dados que representa um estado
type STATE_TYPE is (S0, S1, S2, S3);
attribute ENUM_ENCODING: STRING;
attribute ENUM_ENCODING of STATE_TYPE: type is
"00 01 10 11";
– declaração das variáveis que representam
– o estado actual e o estado seguinte
signal CS, NS: STATE_TYPE;

begin
    SYNC_PROC: process(CLOCK, RESET)
    begin
        if (RESET='1') then
            CS<=S0;
        elsif (CLOCK'event and CLOCK='1') then
            CS<=NS;
        end if;
    end process;
    COMB_PROC: process (CS, INPUT)
    begin
        case CS is
            when S0 =>
                OUTPUT <= "00";
                NS <= S2;
            when S1 =>
                OUTPUT <= "00";
                NS <= S1;
            when S2 =>
                OUTPUT <= "10";
                if (INPUT = '0') then
                    NS <= S3;
                elsif (INPUT = '1') then
                    NS <= S1;
                end if;
            when S3 =>
                OUTPUT <= "01";
                NS <= S1;
            end case;
        end process;
    end BEHAV;
```

VIII. Análise Comparativa

Os Diagramas de Transição de Estados são fáceis de construir se o algoritmo de controlo for simples. Se o algoritmo possui um grande número de estados e/ou um grande número de transições o diagrama torna-se bastante complexo e de difícil compreensão. Além deste inconveniente também não é capaz de representar o funcionamento do circuito de forma algorítmica, isto é, não é capaz de mostrar uma sequência de passos que produzam a sequência de acções baseadas nos valores de entrada.

Uma das desvantagens do método de especificação PRALU é o conjunto de cadeias não fornecer uma fácil percepção do fluxo do algoritmo. Embora seja uma notação relativamente simples para descrever situações de paralelismo, este método não suporta hierarquia, ou seja, não dispõe de recursos (quer textuais quer gráficos) para representar uma descrição hierárquica. A hierarquia poderá ser introduzida à custa de uma notação pré-estabelecida das etiquetas atribuídas às cadeias para que se torne mais intuitiva a noção de módulo. Todos os sinais envolvidos nas invocações e retornos hierárquicos dos módulos têm de ser descritos explicitamente à custa de operações de espera, no caso do algoritmo que é invocado, e de operações de acção, no caso do algoritmo que retorna, que deve sinalizar a sua terminação. As descrições das operações de espera e de acção devem ser o mais compactas possível, caso contrário a descrição

	Diag. T. Estados	PRALU	Statecharts	SDL	HGS	VHDL
Modularidade	0	0	1	3	4	4
Hierarquia	0	0	2	2	4	3
Paralelismo	0	4	4	3	0	4
Visualização do fluxo do algoritmo	3	2	3	3	4	2
Aprendizagem	4	3	4	1	4	2
Total	7	9	14	12	16	15

Tabela I
SUMÁRIO DA FACILIDADE NA APRENDIZAGEM E NA DESCRIÇÃO DE ALGUMAS PROPRIEDADES PELAS VÁRIAS LINGUAGENS.

textual torna-se incompreensível. As vantagens deste método residem indubitavelmente na capacidade de descrição do paralelismo e na existência de mecanismos de verificação formais [2], [14].

Os Statecharts acrescentam aos diagramas de transição de estados a possiblidade de descrever algoritmos usando a noção de hierarquia e de ortogonalidade. No entanto, a modularidade é uma propriedade que não é facilmente representada num diagrama de Statecharts. O algoritmo completo deve ser descrito num único diagrama, segundo a definição original dos Statecharts.

Os Statecharts são uma linguagem gráfica que combina o formalismo baseado em estados reminiscente das MEF com os diagramas de transição de estados. Esta linguagem adiciona aos diagramas:

- A comunicação por difusão por meio de variáveis globais - A difusão é o disparo de transições causado por um dado evento em todos os componentes a que este evento está associado. O disparo é a permissão que um evento anterior dá para que possa ocorrer um evento posterior.
- A modularidade e hierarquia, decompondo estados em sub-estados - A hierarquia permite o agrupamento de transições comuns entre estados.
- O historial dos estados passados.

Uma desvantagem deste método, que parece herdada dos diagramas de transição de estados é a difícil percepção do fluxo do algoritmo de controlo devido à grande quantidade de transições entre estados.

A hierarquia é modelada por estados compostos por sub-estados. A combinação destas duas características resulta na desvantagem da dissimulação das interligações entre estados hierárquicos, isto é, as invocações e os retornos hierárquicos não são evidentes num diagrama de Statecharts porque são representadas por transições tal como o são as transições entre quaisquer sub-estados.

A linguagem SDL é uma linguagem ao nível do sistema e por isso um pouco complexa para a descrição de unidades de controlo. Apenas uma pequena parte dos recursos disponibilizados por esta linguagem é que podem ser utilizados para esse fim. A dificuldade na percepção de um algoritmo de controlo descrito em SDL é comparável à dos Statecharts, embora seja menor, pois o fluxo de controlo não é muito evidente quando se trata de hierarquia e de chamadas dos vários módulos.

Os HGS, ao contrário da linguagem SDL, são um método de descrição exclusivamente dedicado aos circuitos de controlo. É uma linguagem gráfica cuja aprendizagem é comparável à dos diagramas de transição de estados e à dos Statecharts. De todas as linguagens apresentadas esta é a que melhor modela as situações de hierarquia. No entanto, não dispõe de recursos para representar o paralelismo.

A linguagem VHDL é uma linguagem textual para descrever qualquer sistema em hardware. Isto significa que qualquer unidade de controlo descrita pelas outras linguagens é seguramente descrita em VHDL. Relativamente a esta linguagem, neste artigo é analisada apenas a estrutura da código que é normalmente utilizado para descrever uma unidade de controlo. Embora a modularidade seja visível, a assimilação do fluxo de execução do algoritmo de controlo só é conseguida após análise detalhada do código. Esta dificuldade reside no facto de o VHDL não ser uma linguagem gráfica como o SDL e os HGS.

A tabela I sumariza e compara as propriedades das linguagens analisadas ao longo deste artigo. As propriedades são a modularidade, hierarquia, paralelismo, visualização do fluxo do algoritmo e facilidade na aprendizagem. A cada uma delas é atribuído um valor contido numa escala de 0 a 4. O valor 0 significa a incapacidade da linguagem em descrever a referida propriedade e os valores de 1 a 4 representam a facilidade crescente da linguagem em representar essa propriedade. As classificações atribuídas a cada linguagem foram somadas e os resultados obtidos encontram-se na última linha da tabela. A linguagem HGS foi a melhor classificada.

IX. CONCLUSÕES

Uma unidade de controlo é considerada uma entidade constituída por um conjunto de sinais de entrada e um conjunto de sinais de saída. O comportamento da unidade é descrito por um algoritmo onde se estabelece a sequência de activação dos sinais de saída que pode depender do

valor dos sinais de entrada.

O algoritmo pode ser escrito em várias linguagens formais de especificação. Algumas foram apresentadas e analisadas neste artigo. A análise foi realizada para as seguintes características: modularidade, hierarquia, paralelismo, visualização do fluxo do algoritmo e facilidade na aprendizagem. A comparação das linguagens revelou que não existe uma que seja capaz de apresentar todas estas propriedades melhor do que todas as outras. De notar que relativamente à linguagem VHDL está a analisar-se apenas a capacidade de descrição de uma unidade de controlo que vulgarmente é modelada por uma MEF. A linguagem HGS, de todas as propriedades que consegue descrever é aquela a que foi atribuída a melhor classificação na análise comparativa. Além disso é uma linguagem gráfica e por isso mais atractiva. A única desvantagem deste método de descrição é a impossibilidade de representação de situações de paralelismo.

As ferramentas de projecto de sistemas digitais disponíveis actualmente no mercado combinam uma linguagem gráfica com uma linguagem textual para descrever o comportamento de unidades de controlo. A linguagem gráfica baseia-se nos diagramas de transição de estados com algumas extensões. A linguagem textual é uma linguagem HDL pois permite realizar a síntese e efectuar todos os passos que restam do projecto do sistema até à sua implementação. Estas ferramentas permitem converter a descrição gráfica na descrição textual. O código HDL gerado pode ser alterado de forma a cumprir requisitos de especificação que não tenham sido contemplados pela descrição gráfica. Pode-se assim concluir que as linguagens gráficas integradas nestas ferramentas não são suficientemente completas para descrever o comportamento da unidade de controlo desejada. Surge então a necessidade de uma linguagem gráfica de especificação que seja tão poderosa como os HGS e que acrescente o paralelismo. Em [15] foi proposta pelo autor a linguagem HiParaGraphs (*Hierarchical and Parallel Graphs*) derivada dos HGS que lhe adiciona o paralelismo e a comunicação entre módulos.

REFERÊNCIAS

[1] A. Zakrevskij, "Sequent model for representation of digital systems behavior", em *The International Workshop on Discrete-Event System Design - DESDes'01*, Polónia - Junho de 2001.

[2] B. Steinbach e A. Zakrevskij, "Parallel automaton: Basic model, properties and diagnostics", em *4th International Workshop Boolean Problems*, Freiberg, Alemanha - Setembro de 2000.

[3] D. Harel, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, , no. 8, pp. 231–271, 1987.

[4] D. Drusinsky e D. Harel, "Using statecharts for hardware description and synthesis", vol. 8, no. 7, pp. 798–807, Julho de 1989.

[5] M. Löwis J. Fischer, E. Holz e A. Prinz, "Sdl 2000: A language with a formal semantics", em *The Third Workshop on Rigorous Object-Oriented Methods - ROOM*, Reino Unido - Janeiro de 2000.

[6] Y. Yanov, "About logic schemes of algorithms", *Problems of Cybernetics*, , no. 1, pp. 75–127, 1958, Em russo.

[7] S. Baranov, *Synthesis of Microprogrammed Automata*, Energy Publishing Company, 1974, Em russo.

[8] S. Baranov, *Synthèse des Automates Microprogrammés*, Éditions de Moscou, 1983.

[9] V. Sklyarov, "Hierarchical graph-schemes", , no. 2, pp. 82–87, 1984, Em russo.

[10] Synplicity, "Synplify pro", Janeiro de 2004, URL: http://www.synplicity.com.

[11] Xilinx, "Statecad", Janeiro de 2004, URL: http://www.statecad.com.

[12] Xilinx, "Xilinx", Janeiro de 2004, URL: http://www.xilinx.com.

[13] Mentor Graphics, "Renoir personal designer", Janeiro de 2004, URL: http://www.mentor.com.

[14] D. Cheremisinov, "Deriving programs from parallel algorithms of logical control", em *The International Workshop on Discrete-Event System Design - DESDes'01*, Polónia - Junho de 2001.

[15] V. Sklyarov A. Melo e A. Ferrari, "Hiparagraphs, a specification language for hierarchical and parallel control algorithms", em *Electrónica e Telecomunicações*, Janeiro de 2001, vol. 3, pp. 214–221.