Síntese de Unidades de Controlo Descritas por Grafos dum Esquema Hierárquicos

Valery Sklyarov, António Adrego da Rocha

Resumo **- Este artigo descreve grafos hierárquicos que podem ser usados eficientemente para descrever o comportamento de unidades de controlo. A metodologia a empregar permite a síntese do esquema final construído com dispositivos de lógica programável. No entanto, também pode ser aplicada a esquemas construídos com elementos arbitrários. O dispositivo de controlo é modelado como uma máquina finita de estados com *stack*, o que torna possível suportar hierarquia nas operações a desempenhar. Ambos os modelos das máquinas de Mealy e Moore são considerados. O processo de síntese é dividido nos seguintes passos: conversão do grafo hierárquico numa tabela de transição de estados; codificação de estados; atribuição das micro operações e desenho do esquema final. Uma atenção especial é dedicada ao problema da sincronização. Esta metodologia combina a teoria de máquinas de estados com a programação orientada a objectos, permitindo o desenho de unidades de controlo com novas facilidades, tais como: extensibilidade; flexibilidade e reutilização. Estas técnicas recorrem o mais possível a estruturas predefinidas.**

Abstract **- This paper discusses Hierarchical Graph-Schemes that can be efficiently used in order to describe a behaviour of control units. The approach to be considered allows to synthesis the final scheme built from programmable logic devices. However it can be also applied to schemes constructed from arbitrary elements. The control device is modelled as a finite state machine with a stack memory which makes it possible to support hierarchical ordering of operations to be performed. Both Moore and Mealy machines are being considered. The process of synthesis is divided into the following steps: converting a hierarchical graph-scheme to a structural table; state encoding; micro operation assignment and designing the final scheme. A special attention paid to synchronisation problem. The approach combines finite state machines theory with object oriented programming which allows control units to be designed with new facilities, such as: extensibility, flexibility and reuse. The techniques use predefined frames and basic schemes (templates) as far as possible.**

I. INTRODUÇÃO

Os **grafos hierárquicos** (Hierarchical Graph-Schemes **HGS**) podem ser usados para descrever eficientemente o comportamento de unidades de controlo [1]. Foram inicialmente propostos em [2] e têm a seguinte descrição formal:

- Um HGS é composto de nodos rectangulares e losangulares. Cada HGS tem um nodo rectangular de entrada marcado com a etiqueta ***Begin*** e um nodo rectangular de saída marcado com a etiqueta ***End***;
- Os outros nodos rectangulares contêm **micro instruções** do conjunto $\Phi=\{Y_1,Y_2,...\}$ ou **macro instruções** do conjunto $\varphi=\{Z_1,Z_2,...\}$. Cada nodo em particular contem apenas um elemento do conjunto $\Phi\cup\varphi$. Posteriormente será mostrado que se um método especial de sincronização for usado, um nodo pode incorporar dois elementos, tais que, o primeiro pertença ao conjunto Φ, e o segundo pertença ao conjunto $e\tau$. Qualquer **micro instrução** inclui um subconjunto de *micro operações* do conjunto $Y=\{y_1,...,y_N\}$. Uma **micro operação** é um sinal de saída que causa uma operação simples no *datapath*, tal como activar um registo ou incrementar um contador. Qualquer **macro instrução** reune um subconjunto de *macro operações* do conjunto $Z=\{z_1,...,z_Q\}$. Cada **macro operação** é descrita por um HGS de nível mais baixo. Para começar, vamos assumir que cada macro instrução inclui apenas uma macro operação, ou seja estamos apenas a considerar processos sequenciais (não paralelos);
- Cada nodo losangular contem apenas um elemento do conjunto $X\cup\Theta$, onde $X=\{x_1,...,x_L\}$ é o conjunto das **condições lógicas** e $\Theta=\{\theta_1,...,\theta_I\}$ é o conjunto das **funções lógicas**. Uma *condição lógica* é um sinal de entrada que transporta o resultado de um teste, tal como, *overflow* ou igual a zero. Em robótica, por exemplo, as condições lógicas reflectem um estado cumulativo de vários sensores. Cada *função lógica* é calculada executando alguns passos sequenciais predefinidos, descritos por um HGS de nível mais baixo.

Considere-se o conjunto $E=\{\varepsilon_1,...,\varepsilon_V\}$, para o qual $E=Z\cup\Theta$. Cada elemento $\varepsilon_v\in E$ corresponde ao HGS Γ_v, que descreve um algoritmo que executa ε_v (se $\varepsilon_v\in Z$) ou um algoritmo que calcula ε_v (se $\varepsilon_v\in\Theta$). Em ambos os casos, um algoritmo é descrito usando um HGS de nível inferior. Consideremos que $Z(\Gamma_v)$ é o subconjunto de macro operações e que $\Theta(\Gamma_v)$ é o subconjunto de funções lógicas que pertencem ao HGS Γ_v. Se $Z(\Gamma_v)\cup\Theta(\Gamma_v)=\varnothing$ temos um grafo ordinário [3] ou apenas um nível de representação algorítmica.

De maneira a evitar recursividade infinita na execução do HGS, este deve ser verificado quanto à sua correcção [4]. Vamos pois, considerar apenas HGS correctos.

Um **algoritmo hierárquico** de um controlador lógico pode ser expresso por um conjunto de HGS que descrevem o núcleo principal e todas as partes do conjunto E. O núcleo principal é descrito pelo HGS Γ_0 a partir do qual a execução do algoritmo de controlo será iniciada. Uma execução do algoritmo será iniciada no nodo ***Begin*** de Γ_0 e será finalizada no nodo ***End*** de Γ_0. Todos os restantes HGS serão por consequência chamados por Γ_0 ou por outro HGS, que seja descendente de Γ_0. Para descrever os elementos $\varepsilon_1,...,\varepsilon_V$ usaremos os HGS $\Gamma_1,..., \Gamma_V$.

O uso de um HGS permite-nos desenvolver qualquer algoritmo de controlo complexo em partes, prestando atenção a um nível particular de abstracção (a um elemento particular do conjunto E). Cada componente de E pode ser verificado e testado independentemente. É sabido que existe apenas um método para lidar com a complexidade: "Dividir e conquistar". Esta ideia básica pode ser aplicada de diversas maneiras, por exemplo, com HGS que simplificam o desenho de algoritmos de controlo complexos. A figura 1 exemplifica uma descrição algorítmica composta por um núcleo principal, descrita pelo HGS Γ_0, e pelas macro operações $Z_1,..., Z_5$ descritas respectivamente pelos HGS $\Gamma_1,..., \Gamma_5$.

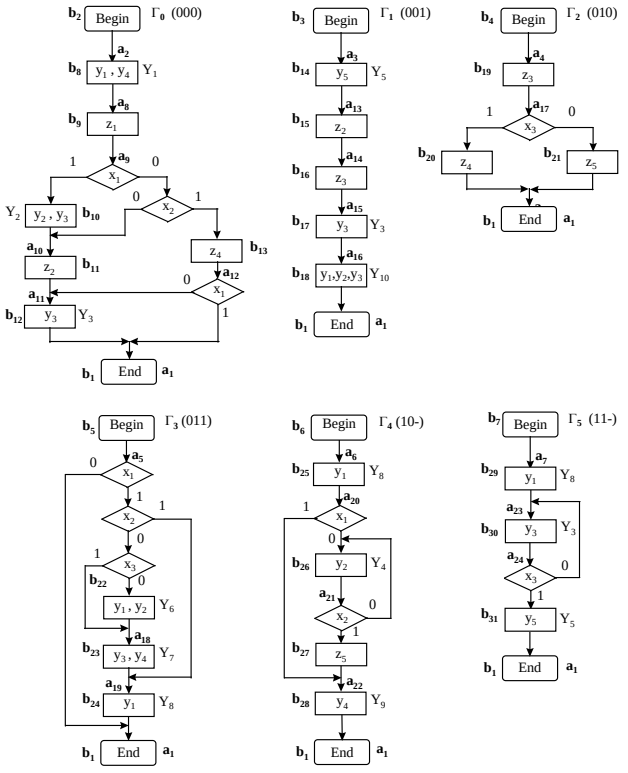

Figura 1. Descrição de um algoritmo com um grafo hierárquico

II. MÁQUINAS DE ESTADOS FINITAS HIERÁRQUICAS

Vamos considerar um grafo G_h que mostra vários níveis hierárquicos e que pode ser considerado como uma árvore. A raiz da árvore **m** corresponde ao núcleo principal HGS Γ_0 do nível 0. As folhas da árvore correspondem aos HGS que não contêm elementos do conjunto E. Estes HGS são grafos ordinários. Consideremos a seguinte sequência de HGS: Γ_0 (nível 0) $\Rightarrow \Gamma^1$ (HGS do nível 1) $\Rightarrow \Gamma^2$ (HGS do nível 2) $\Rightarrow$... , onde Γ^1 é o conjunto de HGS que são usados para descrever elementos do conjunto $Z(\Gamma_0)\cup\Theta(\Gamma_0)$, Γ^2 é o conjunto de HGS que são usados para descrever elementos dos conjuntos $\bigcup_{\gamma\in\Gamma^1} Z(\gamma)$ e $\bigcup_{\gamma\in\Gamma^1}\Theta(\gamma)$. A mesma maneira pode ser usada para determinar os outros conjuntos (Γ^3, Γ^4, etc.). A figure 2 mostra o grafo G_h respeitante ao HGS da figura 1.

Assim temos: $\Gamma^1=\{\Gamma_1,\Gamma_2,\Gamma_4\}$, $\Gamma^2=\{\Gamma_2,\Gamma_3,\Gamma_4,\Gamma_5\}$, $\Gamma^3=\{\Gamma_3,\Gamma_4,\Gamma_5\}$, $\Gamma^4=\{\Gamma_5\}$ e **m** é o núcleo principal do algoritmo. As micro operações $\mathbf{y}^+$ e $\mathbf{y}^-$ são usadas para respectivamente **incrementar** e **decrementar** o ponteiro do *stack* (stack pointer **sp**).

Se considerarmos qualquer HGS Γ_i para o qual $Z(\Gamma_i)\cup\Theta(\Gamma_i)=\varnothing$ então podemos aplicar métodos conhecidos de síntese lógica [1,3,4] de maneira a desenhar um esquema que implemente o comportamento pretendido.

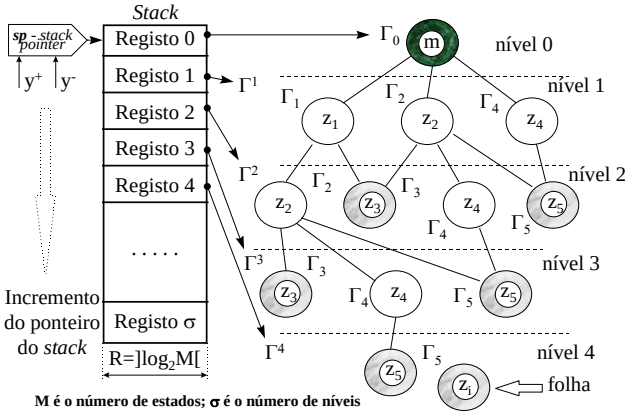

Figura 2. Grafo G_h usando *stack*

O problema pode então ser enunciado da seguinte forma: Como comutar entre os vários níveis? Este problema pode ser eficientemente resolvido usando **máquinas de estados finitas hierárquicas** (hierarchical finite state machine **FSM**) com *stack* (ver figura 3). O registo do topo do *stack* é usado para memória da FSM do HGS Γ_0. Suponhamos que é necessário executar o algoritmo do componente ε_v de Γ_0 e $\varepsilon_v\in Z(\Gamma_0)\cup\Theta(\Gamma_0)$. Nesse caso podemos **incrementar** o ponteiro do *stack* activando $\mathbf{y}^+$ e iniciar o novo registo, que está localizado no novo topo do *stack*, com o estado inicial de Γ_v. Na prática é conveniente seleccionar para tal estado o código

com tudo a zeros (00...0). Consequentemente o registo do topo anterior do *stack* mantém o **estado interrompido** de Γ_0, e o registo do topo actual do *stack* mantém o **estado de entrada** de Γ_v. A mesma sequência de passos pode ser aplicada para os outros níveis. Por consequência a dimensão total do *stack* σ+1 (o número de registos) não deve ser inferior ao número dos vários níveis de G_h, (ver figura 2). Quando a execução do HGS do nível j tiver terminado, temos que executar a sequência inversa de passos, de modo a retornar ao HGS interrompido. Neste caso basta apenas **decrementar** o ponteiro do *stack* activando $\mathbf{y}^-$.

Consideremos agora o esquema físico de base que vai ser usado para a síntese lógica (ver figura 3). O código no registo **Registo^h** indica qual o HGS que deve ser executado a seguir. De forma a usar o **Registo^h** é necessário construir um grafo especial associativo Γ_α, que descreve como comutar entre os vários HGS (por outras palavras, como invocar macro operações e funções lógicas). O Γ_α é composto de dois fragmentos [2] que são o *fragmento de distribuição* $D(\Gamma_\alpha)$ e o *fragmento principal* $M(\Gamma_\alpha)$. Todas as relações mútuas entre eles estão apresentadas na figura 4.

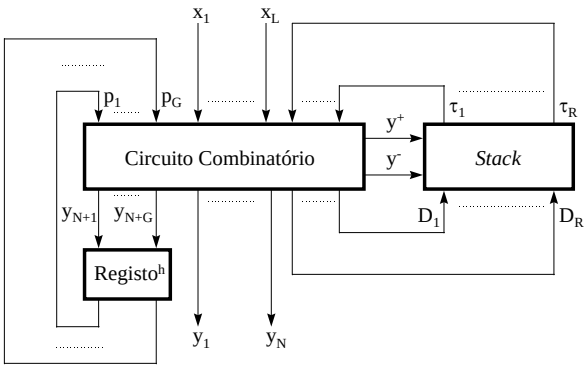

Figura 3. Estrutura básica da máquina de estados finita hierárquica

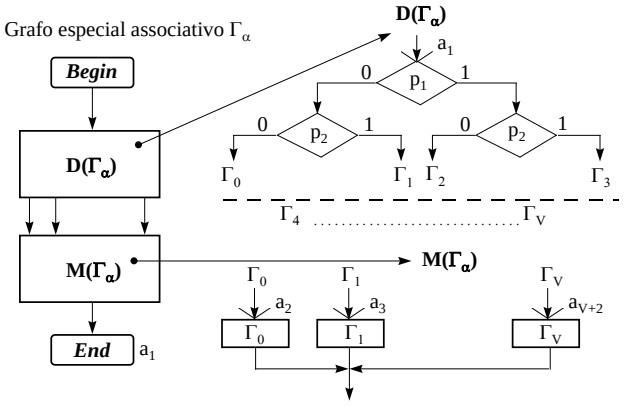

Figura 4. Divisão do grafo especial associativo Γ_α nos dois fragmentos $D(\Gamma_\alpha)$ e $M(\Gamma_\alpha)$

O fragmento $M(\Gamma_\alpha)$ contem V+1 nodos com os símbolos Γ_0, Γ_1,..., Γ_V, correspondentes aos HGS Γ_0, Γ_1,..., Γ_V. Se o nodo Γ_v está activado então o HGS Γ_v tem de ser executado.

O fragmento $D(\Gamma_\alpha)$ é composto apenas por nodos condicionais (losangos) e permite-nos seleccionar os nodos correctos do fragmento $M(\Gamma_\alpha)$. É equivalente a invocar o HGS que deve ser executado.

Vamos atribuir aos elementos do conjunto $E=\{\varepsilon_1,...,\varepsilon_V\}$ **códigos binários** com comprimento $G\geq]\log_2(V+1)[$ e não usaremos o código com tudo a zeros (00...0). Designemos $K(\varepsilon_v)=\{e_{v1}...e_{vG}\}$ como sendo o código de ε_v, onde $e_{vg}\in\{0,1,-\}$, g=1,...,G, e em que o símbolo "-" representa o valor binário *don't care*. Assim sendo, $K(\varepsilon_v)$ é o código binário do HGS do elemento ε_v. Os códigos dos HGS Γ_0, Γ_1,..., Γ_V da figura 1 estão escritos entre parêntesis à frente dos símbolos Γ_0, Γ_1,..., Γ_V. Vamos considerar a relação entre o HGS Γ_v (com v=1,...,V) e o produto $p_1^{e_{v1}}...p_G^{e_{vG}}$ ($p_g^0=\overline{p}_g$, $p_g^1=p_g$, $p_g^-=1$, g=1,...,G). Ao núcleo principal do algoritmo HGS Γ_0 é atribuído o produto $\bigwedge_{g=1}^{G}\overline{p}_g$. Consideremos a seguinte equação:

$$\textbf{\textit{Begin}}\rightarrow\alpha_0\Gamma_0\vee\alpha_1\Gamma_1\vee...\vee\alpha_V\Gamma_V\vee\alpha_{V+1}\textbf{\textit{Begin}},$$

onde α_0, α_1,..., α_V, α_{V+1} são as funções que forçam as respectivas transições a partir da etiqueta ***Begin*** para os nodos dos símbolos Γ_0, Γ_1,...,Γ_V, e

$$\alpha_0=\bigwedge_{g=1}^{G}\overline{p}_g,$$

$$\alpha_1=p_1^{e_{11}}...p_G^{e_{1G}},...,\alpha_V=p_1^{e_{V1}}...p_G^{e_{VG}},$$

$$\alpha_{V+1}=\overline{\alpha_0\vee\alpha_1\vee...\vee\alpha_V}.$$

Para os HGS $\Gamma_0,\Gamma_1,...,\Gamma_5$ da figura 1 e para o conjunto $E=\{\varepsilon_1,\varepsilon_2,\varepsilon_3,\varepsilon_4,\varepsilon_5\}$ ($\varepsilon_1=z_1$, $\varepsilon_2=z_2$, $\varepsilon_3=z_3$, $\varepsilon_4=z_4$, $\varepsilon_5=z_5$) podemos efectuar a seguinte codificação: $K(\varepsilon_1)=001$, $K(\varepsilon_2)=010$, $K(\varepsilon_3)=011$, $K(\varepsilon_4)=10-$, $K(\varepsilon_5)=11-$. Neste caso a equação pode ser expressa da seguinte forma:

$$Begin\rightarrow\overline{p}_1\overline{p}_2\overline{p}_3\Gamma_0\vee\overline{p}_1\overline{p}_2p_3\Gamma_1\vee\overline{p}_1p_2\overline{p}_3\Gamma_2\vee$$
$$\overline{p}_1p_2p_3\Gamma_3\vee p_1\overline{p}_2\Gamma_4\vee p_1p_2\Gamma_5.$$

De acordo com a equação construímos o grafo especial associativo Γ_α apresentado na figura 5.

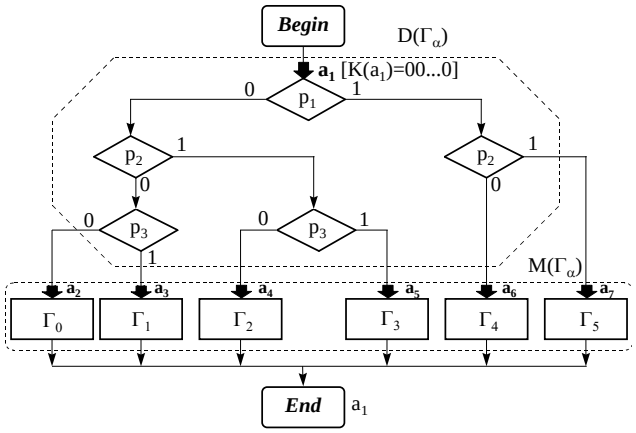

Figura 5. Grafo especial associativo Γ_α do HGS da figura 1

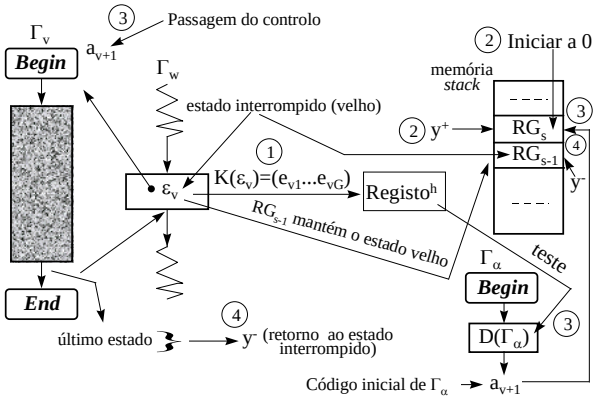

Figura 6. Sequência de acções que força a transição do HGS Γ_w para o HGS Γ_α (1–3) e retorno (4)

Vamos voltar ao esquema da figura 3. A FSM hierárquica comporta-se como uma FSM ordinária se não houver transições entre dois HGS. Se for necessário invocar um novo HGS Γ_v de forma a executar quer uma *macro operação*, quer uma *função lógica* é necessário executar a seguinte sequência de acções (ver figura 6):

1. O código $K(\varepsilon_v)=(e_{v1}...e_{vG})$ é armazenado no registo **Registo^h** (ver figura 3);
2. O ponteiro do *stack* é incrementado ($y^+=1$). Por consequência um novo registo RG_s do *stack* é usado como o registo actual da nossa FSM hierárquica. O registo anterior RG_{s-1} mantém o estado da FSM hierárquica interrompida (aquela que invocou o HGS Γ_v). O novo registo seleccionado RG_s é automaticamente iniciado com o código binário (00...0). Devemos portanto, atribuir o código **$K(a_1)$=00...0** ao estado **a_1**;
3. O código $K(\varepsilon_v)=(e_{v1}...e_{vG})$ atribuído ao **Registo^h** causa a transição para o estado inicial de Γ_v. Então Γ_v passa a ser responsável pelo controlo das operações, a partir deste ponto e até a sua execução terminar;
4. Depois da execução de Γ_v terminar é necessário gerar o sinal **y^-=1** de forma a retornar ao estado interrompido. Assim o controlo regressa ao estado que invocou Γ_v. As acções seguintes serão executadas da mesma maneira que numa FSM ordinária.

Se um HGS é usado para calcular uma *função lógica*, o valor a calcular será armazenado num qualquer *bit* do **Registo^h**. Isto é permitido porque o **Registo^h** não é usado enquanto decorre a execução do HGS. O HGS interrompido testará posteriormente este valor de forma a executar a respectiva transição.

III. O PROBLEMA DA SINCRONIZAÇÃO

Uma FSM hierárquica tem uma **sincronização** mais complexa que uma FSM ordinária. É necessário sincronizar os seguintes eventos (os três primeiros são comuns a uma FSM ordinária):

- Transições de estados;
- Gerar as variáveis de saída $y_1,...,y_N$ (micro operações);
- Fixar das variáveis de entrada $x_1,...,x_L$ (condições lógicas);
- Colocar da codificação desejada no **Registo^h**;
- Incrementar o ponteiro do *stack*;
- Decrementar o ponteiro do *stack*;
- Iniciar o novo registo do *stack* a zero.

É necessário mencionar que existem vários mecanismos de sincronização e que queremos considerar apenas um. Todas as necessárias convenções e assunções relacionadas com o mecanismo de sincronismo sugerido são enumeradas de seguida:

- A selecção do HGS a executar apenas pode ser feita no estado **a_1** com o código **00...0** (tudo a zeros);
- Apenas se pode decrementar o ponteiro do *stack* no estado **a_1** com o código **00...0**. O ponteiro do *stack* não pode ser decrementado se estiver posicionado no primeiro registo do *stack*, mesmo que, o sinal **y^-** tenha sido activado;
- As variáveis $y_{N+1},...,y_{N+G}$, y^+ podem ser activadas em qualquer estado com excepção do estado a_1;
- De forma a colocar informação no **Registo^h** o sinal **y_z** será gerado e activado em cada estado. Para iniciar o novo registo do *stack* será gerado o sinal **y_0**, que será activado se e apenas se o sinal **y^+** estiver activado. A sequência de eventos mencionada será a seguinte: *a)* $y_1,...,y_N,y_{N+1},...,y_{N+G}$; *b)* y_z; *c)* y^+, y^-; *d)* y_0. Esta

sequência pode ser obtida usando o **atraso físico** dos elementos lógicos. A este método daremos o nome de sequência assíncrona em circuitos síncronos, tais como FSM hierárquicas;

- Os valores das variáveis $y_1,...,y_N$, $y_{N+1},...,y_{N+G}$, $x_1,...,x_L$, $p_1,...,p_G$ são **estáticos**. São activados gerando os *sinais de sincronismo dinâmicos* apropriados e serão desactivados apenas depois de gerados novos *sinais de sincronismo dinâmicos*. Pelo contrário, os valores das variáveis y^+, y^-, y_z, y_0 são **dinâmicos**. Estas variáveis são usados para controlar a atribuição dos valores estáticos. São sinais gerados num período mínimo, o mais pequeno possível, mas suficiente para que os valores estáticos sejam correctamente activados ou desactivados. Este período mínimo depende do *chip* usado (em particular da sua tecnologia e dos atrasos físicos reais). As variáveis de entrada $x_1,...,x_L$ são estabilizadas no *datapath* (unidade de execução). As variáveis de saída $y_1,...,y_N$ são geradas na unidade de controlo (o sinal de sincronismo y_z pode ser usado para validá-las);
- Se pudermos iniciar todos os registos do *stack* previamente a zero, então o sinal y_0 é desnecessário, o que simplifica o mecanismo de sincronização;
- O método de sincronização considerado permite-nos combinar micro operações e macro operações no mesmo nodo de um HGS. De facto eles serão activados em períodos de tempo diferentes. Micro operações têm valores activos durante o actual período de tempo. A transição para o respectivo HGS (a execução da macro operação) será começada depois desse terminar.

A figura 7 exemplifica uma forma de onda possível para um sinal de sincronismo discreto de acordo com o mecanismo acabado de descrever.

Cada transição na FSM hierárquica é sincronizada pela transição ascendente do impulso discreto (para este efeito podem ser utilizados registos *latch*). Depois da transição ascendente, a sequência predefinida de impulsos de curta duração são gerados. Eles fixam os respectivos eventos descritos anteriormente e indicados na figura 7.

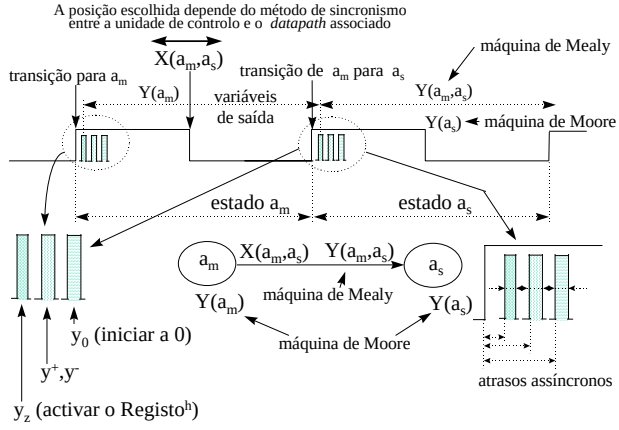

Figura 7. Forma de onda do sinal de sincronismo para FSM hierárquica

Deve ser mencionado que todas as ligações externas do esquema (ver figura 3) podem ser previamente estabelecidas para um número infinito, por assim dizer, de aplicações. Neste caso a adaptação particular do esquema pode ser simplesmente obtida por programação (ou reprogramação) dos seus componentes [1]. Por conseguinte podemos considerar esquemas básicos [1] para implementar FSM hierárquicas.

IV. SÍNTESE DE UMA MÁQUINA DE ESTADOS HIERÁRQUICA DE MEALY

A síntese é baseada em algoritmos sugeridos em [4] e inclui os seguintes passos.

Passo 1. Conversão do HGS numa **tabela estrutural** que é usada da mesma forma que uma tabela de transições de estados de uma FSM.

Passo 2. Codificação dos estados.

Passo 3. Optimização da lógica combinatória e desenho do esquema final.

A conversão do HGS é subdividida em três passos: *1)* marcar o HGS com etiquetas (ou com estados); *2)* armazenar na tabela estrutural todas as transições entre as etiquetas (entre os estados); *3)* transformar a tabela estrutural obtida numa tabela estrutural ordinária. Para marcar o HGS é necessário executar as seguintes acções:

- A etiqueta a_1 é atribuída a todos os nodos dos HGS $\mathbf{\Gamma_0}$, $\mathbf{\Gamma_1}$,..., $\mathbf{\Gamma_V}$, $\mathbf{\Gamma_\alpha}$ que contêm a etiqueta ***End*** e à entrada do fragmento $D(\Gamma_\alpha)$ como mostra a figura 4;
- As etiquetas $\mathbf{a_2,a_3,...,a_{V+2}}$ são atribuídas: *1)* respectivamente às entradas dos nodos a que chegam setas provenientes dos nodos ***Begin*** dos HGS Γ_0, Γ_1,..., Γ_V, de forma que, a_2 é usado em Γ_0, a_3 é usado em Γ_1, etc.; *2)* respectivamente às entradas dos nodos do fragmento $M(\Gamma_\alpha)$ que contêm $\mathbf{\Gamma_0}$, $\mathbf{\Gamma_1}$,..., $\mathbf{\Gamma_V}$, de forma que $\mathbf{a_2}$ marca $\mathbf{\Gamma_0}$, $\mathbf{a_3}$ marca $\mathbf{\Gamma_1}$, etc. (ver figuras 4 e 5);
- As etiquetas $\mathbf{a_{V+3},...,a_M}$ são atribuídas às seguintes entradas: *1)* aquelas a que chegam setas provenientes das saídas de nodos rectangulares nos HGS Γ_0, Γ_1,..., Γ_V; *2)* às que são entradas de nodos losangulares que contêm *funções lógicas* (designemos o subconjunto destas etiquetas como $\mathbf{A^{lf}}$); *3)* às que são entradas de nodos rectangulares contendo *macro operações* (designemos o subconjunto destas etiquetas como $\mathbf{A^{mo}}$);
- É proibido repetir a mesma etiqueta (com excepção de $\mathbf{a_1}$) nos vários HGS;
- Se alguma entrada já tiver sido marcada, não pode ser marcada outra vez. Ou seja, não é permitido marcar nenhuma entrada com mais do que uma etiqueta.

Depois de aplicado o método descrito ao HGS da figura 1 obtemos o *HGS marcado* com as etiquetas $a_1,..,a_{28}$ que se vê na figura 1.

Para construir a tabela estrutural é necessário executar as acções a seguir descritas:

- Armazenar todas as transições:

$a_1 \overline{p}_1 \ldots \overline{p}_G a_2;$
$a_1 p_1^{e_{11}} \ldots p_G^{e_{1G}} a_3;$
. . . . . .
$a_1 p_1^{e_{V1}} \ldots p_G^{e_{VG}} a_{V+2};$

- Para todos os HGS é necessário armazenar as seguintes transições:

$a_m X(a_m,a_s) Y(a_m,a_s) a_s;$
$a_m X(a_m,a_s) a_s;$

Onde **$X(a_m,a_s)$** é o produto das variáveis de entrada (condições lógicas) e funções lógicas que causam a transição de **a_m** para **a_s**, **$Y(a_m,a_s)$** é o subconjunto das variáveis de saída (micro operações) que têm papel activo após a transição (o tempo real de validação de $Y(a_m,a_s)$ depende do mecanismo de sincronização). O primeiro tipo de transição passa apenas por um nodo rectangular e qualquer número de nodos losangulares ($X(a_m,a_s)=1$ é admissível). O segundo tipo de transição passa apenas através de nodos losangulares. Neste caso, se $a_s \in \{a_1\} \cup A^{lf} \cup A^{mo}$ então não é permitido passar nenhuma transição através da etiqueta a_s. Podemos iniciar uma transição a partir de a_s ou finalizar uma transição em a_s,, mas nunca ambas. Em alguns casos podemos apenas usar o segundo tipo de transição. Nos restantes casos é aconselhável usar o primeiro tipo de transição;

- Cada transição é armazenada numa linha da tabela estrutural. Todas as transições do mesmo estado são agrupadas. Os grupos são separados por linhas horizontais (ver artigo [1], para obter informação adicional acerca de tabelas estruturais).

A tabela 1 é a tabela estrutural do HGS da figura 1. Esta tabela não pode ser ainda considerada uma tabela estrutural ordinária (ver por exemplo, [1]) porque contem símbolos do conjunto E (macro operações neste caso, e também funções lógicas noutras aplicações).

Tabela 1.

A_m	$K(a_m)$	a_s	$K(a_s)$	$X(a_m, a_s)$	$Y(a_m, a_s)$
a_1	00000	a_2	00001	$\overline{p}_1\overline{p}_2\overline{p}_3$	-
		a_3	00010	$\overline{p}_1\overline{p}_2 p_3$	-
		a_4	00011	$\overline{p}_1 p_2\overline{p}_3$	-
		a_5	00100	$\overline{p}_1 p_2 p_3$	-
		a_6	00101	$p_1\overline{p}_2$	-
		a_7	00110	$p_1 p_2$	-
a_2	00001	a_8	00111	1	y_1,y_4
a_3	00010	a_{13}	01100	1	y_5
a_4	00011	a_{17}	10000	1	**z_3**
a_5	00100	a_1	00000	$\overline{x}_1$	-
		a_1	00000	$x_1 x_2$	y_1
		a_{19}	10010	$x_1\overline{x}_2 x_3$	y_3,y_4
		a_{18}	10001	$x_1\overline{x}_2\overline{x}_3$	y_1,y_2
a_6	00101	a_{20}	10011	1	y_1
a_7	00110	a_{23}	10110	1	y_1
a_8	00111	a_9	01000	1	**z_1**
a_9	01000	a_{10}	01001	x_1	y_2,y_3
		a_{10}	01001	$\overline{x}_1\overline{x}_2$	-
		a_{25}	11000	$\overline{x}_1 x_2$	-
a_{10}	01001	a_{11}	01010	1	**z_2**
a_{11}	01010	a_1	00000	1	y_3
a_{12}	01011	a_1	00000	$\overline{x}_1$	y_3
		a_1	00000	x_1	-
a_{13}	01100	a_{14}	01101	1	**z_2**
a_{14}	01101	a_{15}	01110	1	**z_3**
a_{15}	01110	a_{16}	01111	1	y_3
a_{16}	01111	a_1	00000	1	y_1,y_2,y_3
a_{17}	10000	a_{26}	11001	x_3	-
		a_{27}	11010	$\overline{x}_3$	-
a_{18}	10001	a_{19}	10010	1	y_3,y_4
a_{19}	10010	a_1	00000	1	y_1
a_{20}	10011	a_{21}	10100	$\overline{x}_1$	y_2
		a_1	00000	x_1	y_4
a_{21}	10100	a_{28}	11011	x_2	-
		a_{21}	10100	$\overline{x}_2$	y_2
a_{22}	10101	a_1	00000	1	y_4
a_{23}	10110	a_{24}	10111	1	y_3
a_{24}	10111	a_1	00000	x_3	y_5
		a_{24}	10111	$\overline{x}_3$	y_3
a_{25}	11000	a_{12}	01011	1	**z_4**
a_{26}	11001	a_1	00000	1	**z_4**
a_{27}	11010	a_1	00000	1	**z_5**
a_{28}	11011	a_{22}	10101	1	**z_5**

De forma a transformá-la numa tabela estrutural ordinária é necessário executar as seguintes as acções:

- Se uma linha da coluna $Y(a_m,a_s)$ contem uma macro operação $\varepsilon_v=z_j \in Z$ então tem de ser substituída por novas variáveis de saída do tipo y^+ e $y_{N+1},\ldots,y_{N+G}$. Estamos a escolher apenas as variáveis y_{N+i} para as quais $e_{vi}=1$ (relembrar que e_{vi} é o valor do *bit* número i no código em que $\varepsilon_v=z_j$);

- Suponhamos que $a_u \in A^{lf}$ e o respectivo nodo losangular contem a função lógica $\theta_k=\varepsilon_v$ Neste caso em todas as linhas $Y(a_u,a_s)$ temos de adicionar as novas variáveis de saída y^+ e $y_{N+1},...,y_{N+G}$ (respeitando todas as regras enunciadas no ponto anterior);
- Todos os símbolos θ_k na coluna $Y(a_m,a_s)$ são substituídos por y_{N+1}. Devemos fazê-lo porque queremos retornar o valor calculado de θ_k do HGS invocado para o HGS invocador, através do **Registo**^{h} (por exemplo, podemos usar o primeiro *bit*);
- Todos os símbolos θ_k na coluna $X(a_m,a_s)$ são substituídos por novas variáveis de entrada p_1 que é dependentemente de θ_k o valor invertido no caso de $\bar{\theta}_k$ ou o valor directo no caso de θ_k (todas as explicações necessárias foram referidas no ponto anterior);
- No estado a_1 é necessário activar a variável $\mathbf{y}^-$. Para isso y^- é adicionado em todas as linhas $Y(a_1,a_s)$.

A tabela 2 é a tabela estrutural ordinária construída a partir da tabela 1 e aplicando as acções acima descritas. O símbolo y^- está escrito junto ao estado a_1. Isto é equivalente a adicionar y^- em todas as linhas de $Y(a_1,a_s)$. Informação mais detalhada acerca de tabelas estruturais e sua utilização é descrita em [1,3,4]. Na tabela 2 a coluna $K(a_s)$ foi eliminada.

Tabela 2.

A_m	$K(a_m)$	a_s	$X(a_m, a_s)$	$Y(a_m, a_s)$	$F(a_m, a_s)$
a_1 **y**$^-$	00000	a_2	$\bar{p}_1\bar{p}_2\bar{p}_3$	-	D_5
		a_3	$\bar{p}_1\bar{p}_2 p_3$	-	D_4
		a_4	$\bar{p}_1 p_2\bar{p}_3$	-	D_4D_5
		a_5	$\bar{p}_1 p_2 p_3$	-	D_3
		a_6	$p_1\bar{p}_2$	-	D_3D_5
		a_7	$p_1 p_2$	-	D_3D_4
a_2	00001	a_8	1	y_1,y_4	$D_3D_4D_5$
a_3	00010	a_{13}	1	y_5	D_2D_3
a_4	00011	a_{17}	1	$\mathbf{y^+,y_z^2,y_z^3}$	D_1
a_5	00100	a_1	$\bar{x}_1$	-	-
		a_1	x_1x_2	y_1	-
		a_{19}	$x_1\bar{x}_2x_3$	y_3,y_4	D_1D_4
		a_{18}	$x_1\bar{x}_2\bar{x}_3$	y_1,y_2	D_1D_5
a_6	00101	a_{20}	1	y_1	$D_1D_4D_5$
a_7	00110	a_{23}	1	y_1	$D_1D_3D_4$
a_8	00111	a_9	1	$\mathbf{y^+,y_z^3}$	D_2
a_9	01000	a_{10}	x_1	y_2,y_3	D_2D_5
		a_{10}	$\bar{x}_1\bar{x}_2$	-	D_2D_5
		a_{25}	$\bar{x}_1x_2$	-	D_1D_2
a_{10}	01001	a_{11}	1	$\mathbf{y^+,y_z^2}$	D_2D_4
a_{11}	01010	a_1	1	y_3	-
a_{12}	01011	a_1	$\bar{x}_1$	y_3	-
		a_1	x_1	-	-
a_{13}	01100	a_{14}	1	$\mathbf{y^+,y_z^2}$	$D_2D_3D_5$
a_{14}	01101	a_{15}	1	$\mathbf{y^+,y_z^2,y_z^3}$	$D_2D_3D_4$
a_{15}	01110	a_{16}	1	y_3	$D_2D_3D_4D_5$
a_{16}	01111	a_1	1	y_1,y_2,y_3	-
a_{17}	10000	a_{26}	x_3	-	$D_1D_2D_5$
		a_{27}	$\bar{x}_3$	-	$D_1D_2D_4$
a_{18}	10001	a_{19}	1	y_3,y_4	D_1D_4
a_{19}	10010	a_1	1	y_1	-
a_{20}	10011	a_{21}	$\bar{x}_1$	y_2	D_1D_3
		a_1	x_1	y_4	-
a_{21}	10100	a_{28}	x_2	-	$D_1D_2D_4D_5$
		a_{21}	$\bar{x}_2$	y_2	D_1D_3
a_{22}	10101	a_1	1	y_4	-
a_{23}	10110	a_{24}	1	y_3	$D_1D_3D_4D_5$
a_{24}	10111	a_1	x_3	y_5	-
		a_{24}	$\bar{x}_3$	y_3	$D_1D_3D_4D_5$
a_{25}	11000	a_{12}	1	y^+,y_z^1	$D_2D_4D_5$
a_{26}	11001	a_1	1	y^+,y_z^1	-
a_{27}	11010	a_1	1	y^+,y_z^1, y_z^2	-
a_{28}	11011	a_{22}	1	y^+,y_z^1, y_z^2	$D_1D_3D_5$

A partir da tabela 2 podemos desenhar a unidade de controlo usando algoritmos de síntese lógica (ver por exemplo [1,3,4]). Vamos assumir que queremos desenhar a unidade de controlo usando lógica programável tipo PLA (programmable logic arrays) [1]. Para o nosso exemplo, necessitamos de uma PLA(11,15,41) (com 11 entradas, 15 saídas e 41 produtos). A programação da PLA é apresentada na figura 8.

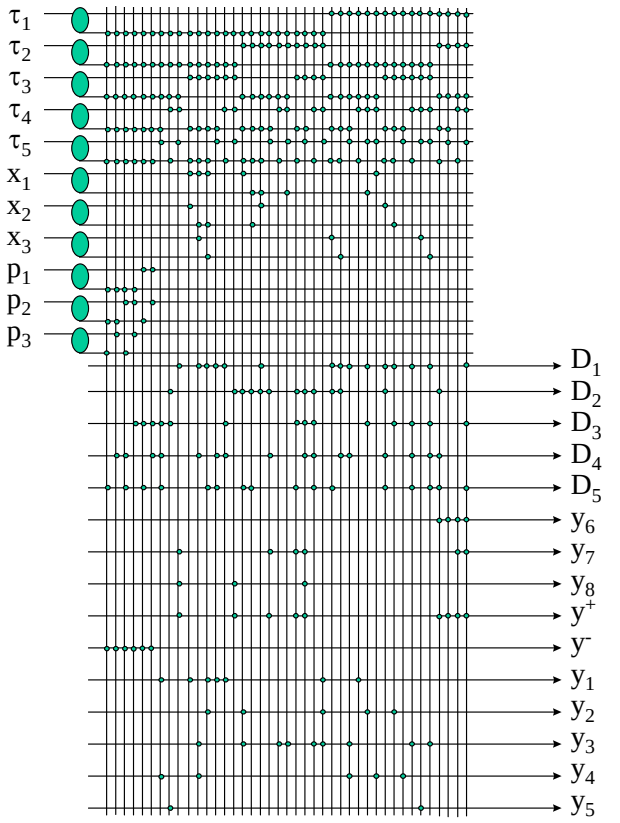

Figura 8. Programação da PLA

Todas as ligações necessárias, usadas no esquema final estão apresentadas na figura 9. Foi usada uma codificação de estados arbitrária (ver tabela 2) e as funções *booleanas* não foram optimizadas. No entanto, podem-se aplicar todos os métodos considerados em [1,3,4] para os passos 2 e 3. O principal objectivo deste artigo é a conversão do **HGS** numa tabela estrutural ordinária que contem toda a

informação necessária para ser sintetizada. É necessário mencionar que o exemplo demonstrativo do uso de funções lógicas em HGS foi apresentado no artigo [4, pág. 47-60].

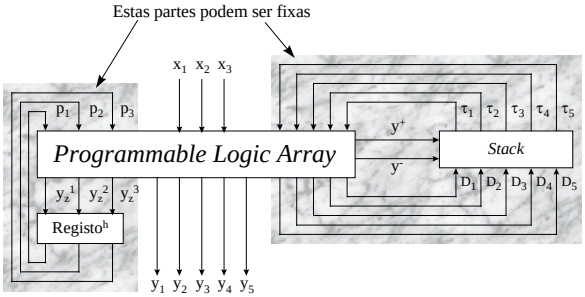

Figura 9. O esquema final ($y_z^1=y_6$; $y_z^2=y_7$; $y_z^3=y_8$)

Frequentemente é necessário ter em conta os **constrangimentos** impostos pelos componentes lógicos usados [1]. Para resolver este problema é possível modificar o procedimento de marcação do HGS inserindo **estados extras,** que nos permitem partir transições entre estados que são críticas (transições que infringem os constrangimentos). Para esse propósito é usado o método indicado em [1,4]. A abordagem sugerida em [1,3,4] é baseada na utilização de esquemas designados de um nível ou multinível de utilização de componentes de lógica programável (ou reprogramável) de tipo PLA, PAL, GAL, ROM, RAM, etc. (componentes a que geralmente se chamam dispositivos de lógica programável **PLD** [1]). Esta abordagem apresenta a vantagem de todos os grupos de ligações futuras serem conhecidas e a complexidade do esquema só depende do número total de componentes (por exemplo, PLAs). Na prática o nível final de complexidade pode ser estabelecido na primeira fase do desenvolvimento. O comportamento do esquema (capacidades funcionais) podem ser extendidas com a adição de novos componentes sem reprogramar os componentes já utilizados (qualquer acrescento não afecta a estrutura). Todos os métodos de síntese lógica e optimização sugeridos em [1,3,4] podem ser combinados com os métodos acima mencionados e directamente aplicados ao desenho de unidades de controlo descritos através de **HGS** e modelados por **FSM hierárquicas**.

Em conclusão deste ponto queremos realçar que os HGS apresentam um conjunto de vantagens. Simplificam a descrição de algoritmos de controlo, providenciam uma maneira natural para a sua decomposição, introduzem ordem no processo de desenho, etc. Estas vantagens podem ser facilmente demonstradas mesmo para o nosso exemplo trivial (ver figura 1). Se apresentássemos o mesmo algoritmo de controlo usando um grafo ordinário (não hierárquico) [3] então a nossa descrição (ver figura 10) tornar-se-ia muito mais complexa e a respectiva FSM adquiriria muitos mais estados (36 estados para a máquina de Mealy e 40 estados para a máquina de Moore).

Obviamente que podemos aplicar métodos de minimização de estados, mas, isso implica um esforço extra e mais tarde perder-se-iam algumas propriedades do esquema (por exemplo, a extensibilidade das instruções, etc.). Para algoritmos complicados de controlo as vantagens da abordagem descrita são essenciais e óbvias.

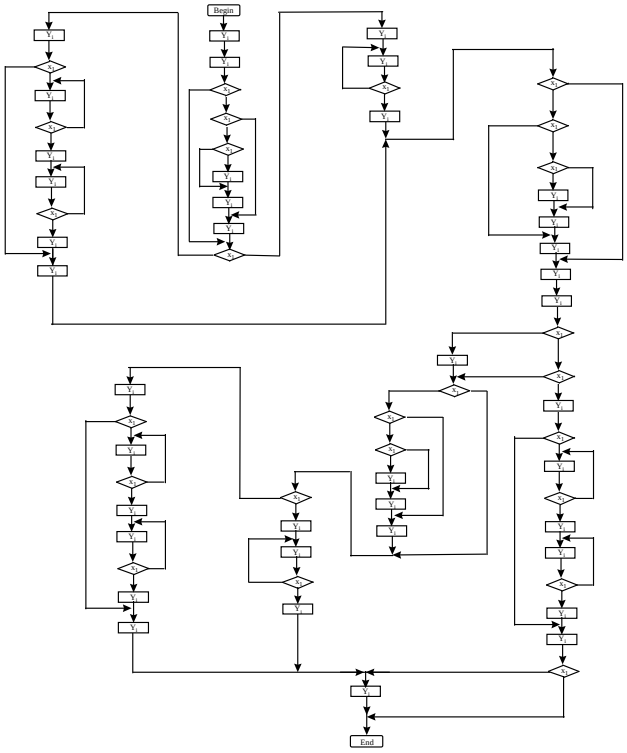

Figura 10. Descrição do algoritmo de controlo da figura 1 com um grafo ordinário

V. SÍNTESE DE UMA MÁQUINA DE ESTADOS HIERÁRQUICA DE MOORE

A síntese inclui os mesmos passos acima considerados. A diferença está relacionada com o passo 1 e é explicada em detalhe de seguida. Este passo é igualmente subdividida nos três passos apresentados na secção IV. No primeiro passo é necessário executar as seguintes acções:

- A etiqueta $\mathbf{a_1}$ é atribuída a todos os nodos de $\mathbf{\Gamma_0}$, $\mathbf{\Gamma_1}$,..., $\mathbf{\Gamma_V}$ que contêm a etiqueta ***End*** e ao nodo ***Begin*** do grafo Γ_α apresentado na figura 4;
- As etiquetas $\mathbf{a_2,a_3,...,a_{V+2}}$ são atribuídas: *1)* respectivamente aos nodos ***Begin*** dos HGS Γ_0, Γ_1,..., Γ_V de forma que, $\mathbf{a_2}$ é usado em $\mathbf{\Gamma_0}$, $\mathbf{a_3}$ é usado em $\mathbf{\Gamma_1}$, etc.; *2)* respectivamente aos nodos do fragmento $M(\Gamma_\alpha)$ que contêm os $\mathbf{\Gamma_0}$, $\mathbf{\Gamma_1}$,..., $\mathbf{\Gamma_V}$ de forma, que $\mathbf{a_2}$ marca $\mathbf{\Gamma_0}$, $\mathbf{a_3}$ marca $\mathbf{\Gamma_1}$, etc.;
- As etiquetas $\mathbf{a_{V+3},...,a_M}$ são atribuídas aos seguintes nodos e entradas: *1)* nodos rectangulares nos HGS Γ_0, Γ_1,..., Γ_V; *2)* entradas dos nodos losangulares que contêm *funções lógicas*;
- É proibido repetir a mesma etiqueta (com excepção de $\mathbf{a_1}$) nos vários HGS;

- Se algum nodo de entrada já tiver sido marcado, não pode ser marcado outra vez. Ou seja, não é permitido marcar nenhum nodo de entrada com mais do que uma etiqueta.

Depois de aplicado o método descrito ao HGS da figura 1 obtemos o *HGS marcado* com as etiquetas b_1,..,b_{31} que se vê na figura 1 (as etiquetas b_i foram usadas de maneira a distingui-las das etiquetas a_i da máquina de Mealy anteriormente considerada).

Os segundo e terceiro passos são semelhantes à máquina de Mealy. Na prática eles apenas diferem no seguinte. Para a máquina de Moore não é permitido usar as transições do tipo $a_m X(a_m,a_s) Y(a_m,a_s) a_s$ (apenas se usam as transições que passam através de nodos losangulares). A tabela 3 apresenta a tabela estrutural ordinária da máquina de estados hierárquica de Moore, que pode ser usada para os passos seguintes da síntese lógica. Tal como nas máquinas de Mealy podem ser utilizados os métodos de síntese e as ideias consideradas em [1,3,4].

Tabela 3.

a_m	$K(a_m)$	a_s	$K(a_s)$	$X(a_m, a_s)$	$F(a_m, a_s)$
b_1 **y⁻**	00000	b_2	00001	$\bar{p}_1\bar{p}_2\bar{p}_3$	D_5
		b_3	00010	$\bar{p}_1\bar{p}_2 p_3$	D_4
		b_4	00011	$\bar{p}_1 p_2\bar{p}_3$	D_4D_5
		b_5	00100	$\bar{p}_1 p_2 p_3$	D_3
		b_6	00101	$p_1\bar{p}_2$	D_3D_5
		b_7	00110	$p_1 p_2$	D_3D_4
b_2, -	00001	b_8	00111	1	$D_3D_4D_5$
b_3, -	00010	b_{14}	01101	1	$D_2D_3D_5$
b_4, -	00011	b_{19}	10010	1	D_1D_4
b_5, -	00100	b_1	00000	$\bar{x}_1$	-
		b_{24}	10111	x_1x_2	$D_1D_3D_4D_5$
		b_{22}	10101	$x_1\bar{x}_2\bar{x}_3$	$D_1D_3D_5$
		b_{23}	10110	$x_1\bar{x}_2x_3$	$D_1D_3D_4$
b_6, -	00101	b_{25}	11000	1	D_1D_2
b_7, -	00110	b_{29}	11100	1	$D_1D_2D_3$
b_8, y_1,y_4	00111	b_9	01000	1	D_2
b_9, $\mathbf{y^+,y_z^3}$	01000	b_{10}	01001	x_1	D_2D_5
		b_{11}	01010	$\bar{x}_1\bar{x}_2$	D_2D_4
		b_{13}	01100	$\bar{x}_1x_2$	D_2D_3
b_{10}, y_2,y_3	01001	b_{11}	01010	1	D_2D_4
b_{11}, $\mathbf{y^+,y_z^2}$	01010	b_{12}	01011	1	$D_2D_4D_5$
b_{12}, y_3	01011	b_1	00000	1	-
b_{13}, $\mathbf{y^+,y_z^1}$	01100	b_{12}	01010	$\bar{x}_1$	D_2D_4
		b_1	00000	x_1	-
b_{14}, y_5	01101	b_{15}	01110	1	$D_2D_3D_4$
b_{15}, $\mathbf{y^+,y_z^2}$	01110	b_{16}	01111	1	$D_2D_3D_4D_5$
b_{16},$\mathbf{y^+, y_z^2,y_z^3}$	01111	b_{17}	10000	1	D_1
b_{17}, y_3	10000	b_{18}	10001	1	D_1D_5
b_{18}, y_1, y_2,y_3	10001	b_1	00000	1	-
b_{19},$\mathbf{y^+, y_z^2,y_z^3}$	10010	b_{20}	10011	x_3	$D_1D_4D_5$
		b_{21}	10100	$\bar{x}_3$	D_1D_3
b_{20}, $\mathbf{y^+,y_z^1}$	10011	b_1	00000	1	-
b_{21},$\mathbf{y^+, y_z^1,y_z^2}$	10100	b_1	00000	1	-
b_{22}, y_1,y_2	10101	b_{23}	10110	1	-
b_{23}, y_3,y_4	10110	b_{24}	10111	1	$D_1D_3D_4D_5$
b_{24}, y_1	10111	b_1	00000	1	-
b_{25}, y_1	11000	b_{26}	11001	$\bar{x}_1$	$D_1D_2D_5$
		b_{28}	11011	x_1	$D_1D_2D_4D_5$
b_{26}, y_2	11001	b_{26}	11001	$\bar{x}_2$	$D_1D_2D_5$
		b_{27}	11010	x_2	$D_1D_2D_4$
b_{27},$\mathbf{y^+, y_z^1,y_z^2}$	11010	b_{28}	11011	1	$D_1D_2D_4D_5$
b_{28}, y_4	11011	b_1	00000	1	-
b_{29}, y_1	11100	b_{30}	11101	1	$D_1D_2D_3D_5$
b_{30}, y_3	11101	b_{30}	11101	$\bar{x}_3$	$D_1D_2D_3D_5$
		b_{31}	11110	x_3	$D_1D_2D_3D_4$
b_{31}, y_5	11110	b_1	00000	1	-

VI. MACRO OPERAÇÕES PARALELAS

Na secção I partimos do princípio de que cada macro instrução incluiria apenas uma macro operação. No caso geral podemos eliminar esse constrangimento. Suponhamos que uma macro instrução **“**$_i$ é composta por mais que uma macro operação (digamos z_1, z_2, z_3). Por consequência elas devem ser executadas **em paralelo**. A ideia de implementar macro instruções em paralelo foi intruzido em [5]. Neste caso o esquema da unidade de controlo torna-se mais complexo. Como anteriormente a memória da FSM hierárquica é baseada em *stack*, mas

cada nível do *stack* deverá ser composto de **β registos** com **β>1** (ver figura 11). O valor de β define o número máximo de macro operações implementáveis em paralelo. Cada registo tem o tamanho R e em geral R<]$\log_2$M[. O *stack* tem R entradas D_1,...,D_R e R saídas τ_1,..., τ_R ligadas ao esquema combinatório (estamos a assumir que todos os registos são composto de *flip-flops* de tipo D). As entradas e as saídas do *stack* são comuns a todos os registos. O **ciclo de relógio** (clock cycle) (ver figura 11,b) da unidade de controlo é dividido em *β sub-relógios*. Cada um afecta o respectivo registo do mesmo nível e altera o seu estado. Suponhamos que é necessário executar a **macro instrução $\blacksquare_i$**, apresentada na figura 11,a.

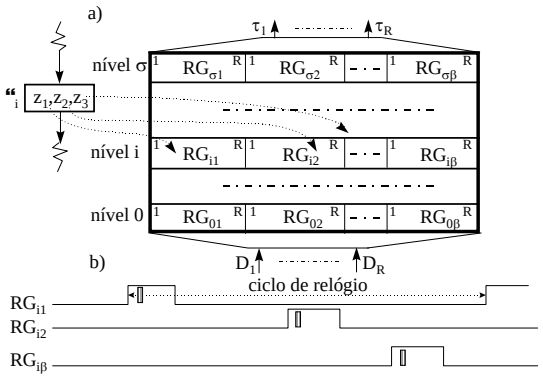

Figura 11. (a) Estrutura da memória *stack* ; (b) Forma de onda do impulso de sincronismo para execução de macro operações em paralelo

Neste caso é necessário executar a seguinte sequência de eventos:

- Um código especial [5] deve ser armazenado no **Registo^h** que será mais tarde usado para identificar todas as macro operações da macro instrução **$\blacksquare_i$**. O código é mais complexo do que para as macro operações sequenciais;
- O ponteiro do *stack* é incrementado (y^+=1). Por conseguinte um novo **subconjunto** de registos RG_{i1},...,$RG_{i\beta}$ será seleccionado, registos esses que devem ser iniciados com zeros;
- Todos os registos RG_{i1},...,$RG_{i\beta}$ são inspeccionados activando os respectivos sub-relógios. Quando algum registo RG_{ij} é seleccionado é carregado com o respectivo código inicial da macro operação. De forma a reconhecer a macro operação é necessário analisar o código armazenado no **Registo^h**;
- Cada ciclo de relógio seguinte causa transições sequenciais nos vários HGS (Γ_1, Γ_2, Γ_3 no nosso exemplo). O primeiro impulso do ciclo de relógio (ver figura 11,b) altera o estado do registo RG_1 que é responsável pela macro operação **z_1** (ver figura 11,a). Então é executada a necessária transição no HGS **Γ_1**. A próxima transição no HGS Γ_1 será executada quando o primeiro impulso do ciclo de relógio seguinte for activado. O segundo impulso causará uma transição semelhante no HGS **Γ_2**, etc. Quando o ciclo de relógio tiver terminado, todas as transições em todos os HGS paralelos terão sido sequencialmente executadas;
- Se alguma macro operação tiver terminado, esperará até que a última macro operação incluída na mesma macro instrução tenha terminado completamente;
- Depois de todas as macro operações da macro instrução terem terminado, o esquema combinatório activa a saída **y$^-$** que recolocará o *stack* no estado interrompido.

Esta abordagem é basicamente boa, porque: permite usar a mesma parte combinatória do esquema para registos diferentes (para executar diferentes HGS, HGS paralelos incluídos), minimiza o número de ligações externas e torna possível organizar pseudo execuções paralelas de macro operações. Por outro lado aumenta o ciclo de relógio, consequentemente aumenta o seu período e deteriora o desempenho da unidade de controlo. Contudo, isto é admissível para muitas aplicações práticas, tais como, o controlo de equipamento tecnológico, robótica, etc.

Deve ser mencionado que apresentámos uma **ideia básica** da abordagem e que foram omitidos todos os detalhes da implementação real. Esses detalhes tornariam a apresentação demasiado extensa para serem apresentados neste artigo. Existem no entanto, alguns problemas que deverão ser investigados, analisados e desenvolvidos no futuro, tais como:

- Como providenciar uma **melhor sincronização**, de forma a aumentar o desempenho. Por exemplo, seria uma boa ideia usar o *self synchronous approach*;
- Como aumentar o **desempenho** de unidades de controlo paralelas usando métodos de reorganização do esquema lógico. De forma a resolver este problema é vantajoso investigar e estimar as FSM hierárquicas com um número variável de ciclos de relógio (ver figura 11,b), invocar outras abordagens (por exemplo, ter em consideração uma colecção de FSM hierárquicas interactivas), etc.;
- Como **optimizar esquemas lógicos** de FSM hierárquicas e como desenhar **estruturas predefinidas** e ***templates*** [1] que simplifiquem o processo de produção de unidades de controlo para uma variedade de aplicações práticas;
- Como lidar com o problema da **recursividade** nos HGS, quando a sequência de invocação das respectivas macro instruções (funções lógicas) se torna infinita.

VII. INTERRELAÇÃO ENTRE MÁQUINAS DE ESTADOS FINITAS HIERÁRQUICAS E A PROGRAMAÇÃO ORIENTADA A OBJECTOS

Consideremos o seguinte problema [1]: desenhar a unidade de controlo que executará **□** e que satisfará o conjunto de condições $\nearrow$, dado o conjunto de instruções **□**={$\square_1$,...,$\square_k$} e constrangimentos $\nearrow$={$\nearrow_1$,...,$\nearrow_p$}. Cada instrução particular **$\square_i$**∈**□** pode ser descrita por um **grafo Γ_i**. Se nenhum dos grafos Γ_1,...,Γ_k contiver macro operações e funções lógicas, então eles são grafos ordinários [3]. Assim obtemos um nível de representação

de □. De maneira a sintetizar o esquema cujo comportamento foi descrito pelos $\Gamma_1,...,\Gamma_k$ podemos ligá-los num **grafo único** usando a ideia que foi demonstrada na figura 4. Neste caso o fragmento $D(\Gamma_\alpha)$ selecciona o grafo apropriado Γ_i por análise de códigos de instruções predefinidos do conjunto □ (como anteriormente podemos usar para esse propósito as variáveis $\mathbf{p_1,...,p_G}$ com $G=\lceil \log_2 k \rceil$). Se o grafo Γ_i tiver sido seleccionado, será armazenado o estado inicial de Γ_i no registo simples que vai ser usado como memória da FSM. Depois de Γ_i ter terminado o controlo passará para o nodo ***Begin*** de Γ_α. Então podemos repetir o processo e seleccionar a instrução seguinte do conjunto □, etc.

Discutamos agora a **representação multinível de** □ que é a seguinte $□=□^0 \cup □^1 \cup ... \cup □^\sigma$ onde o conjunto $□^0$ inclui instruções do nível *0*,..., o conjunto $□^\sigma$ inclui instruções do nível σ [1].

Vejamos a instrução $□_i \in □^j$ ($j<\sigma$). $□_i$ foi descrito pelo **HGS Γ_i** do **nível j** que no caso geral integra *micro operações*, *condições lógicas*, *macro operações* e *funções lógicas*. Cada macro operação foi descrita por um HGS de nível inferior. Então podemos dizer que Γ_i **encapsula** variáveis de entrada e de saída (dados) e operações complexas (macro operações e funções lógicas) que podem ser vistas como **funções de controlo** (compare-se com o encapsulamento na programação orientada a objectos). Finalmente o encapsulamento permite separar a definição de uma instrução da sua implementação. Por outras palavras queremos focar-nos na acção das instruções em vez de na sua implementação.

A capacidade do HGS para encapsular dados e funções leva-nos a estabelecer uma semelhança entre as noções de uma *instrução* na descrição de uma FSM e uma *classe* na programação orientada por objectos (OOP). Por outro lado, a **FSM minúscula** que implementa a instrução pode ser vista como um **objecto** (a mesma ideia foi realçada em [6, pág. 90,91]). Finalmente a unidade de controlo completa pode ser considerada como uma colecção de *FSM interactivas* (ou como uma colecção de objectos interactivos). De facto "an object has state, behaviour and identity" [6, pág. 83]. Qualquer FSM também tem estado, comportamento e identidade. "The structure and behaviour of similar objects are defined in their common class" [6, pág. 83]. No nosso caso a estrutura e o comportamento de FSM semelhantes estão definidas no seu HGS comum (ou instrução).

O paradigma de programação da OOP é: "Decide which classes you want. Provide a full set of operations for each class. Make commonality explicit by using inheritance" [7, pág. 22]. O *paradigma do desenvolvimento* que pode ser considerado como um alicerce para a síntese de unidades de controlo é praticamente o mesmo. Decide quais as **instruções** que queres. Providencia um conjunto completo de **macro operações** e de **funções lógicas** para cada instrução. Torna as semelhanças entre várias instruções explicitas usando a **herança**. Tal abstracção permite-nos aplicar algumas ideias atractivas de OOP às FSM hierárquicas, o que torna possível fornecer novas facilidades às unidades de controlo a desenhar, tais como extensibilidade, flexibilidade e reutilização. Consideremos essas facilidades com um pouco mais de detalhe.

A **extensibilidade** pode ser obtida por introdução de novas instruções, de forma a ampliar o conjunto de instruções □. Na abordagem considerada isto pode ser feito da forma mais simples possível, fazendo simples alterações no grafo Γ_α (ver figura 4) sem influenciar partes do esquema já desenhadas.

A **flexibilidade** pode ser ganha providenciando o refinamento das instruções. Neste caso vale a pena considerar **instruções derivadas**. Genericamente falando elas são melhores do que as suas predecessoras (*instruções de base*). Podemos refinar uma instrução nova (derivada) estendendo o seu comportamento para alem do herdado da **instrução de base**. Para fazê-lo podemos *adicionar novas* macro operações e funções lógicas ou *redefinir* macro operações e funções lógicas *herdadas*. A última ideia leva-nos à definição de **macro operações virtuais** e **funções lógicas virtuais** que estão intimamente relacionadas com **estados virtuais** de FSM hierárquicas.

Genericamente falando **reutilização** denota a capacidade de um dispositivo ser utilizado novamente. Por vezes queremos adicionar funcionalidade ou alterar o comportamento. Nesta abordagem que estamos a considerar (ver também [1]) não necessitamos de começar o desenvolvimento do esquema, outra vez a partir do princípio. O novo esquema **herda** a parte invariável do desenho anterior e é apenas adicionada (ou substituída) a parte que é diferente no novo contexto.

VIII. CONCLUSÃO

Os **grafos hierárquicos** fornecem uma representação multinível natural de algoritmos de controlo que podem ser vistos em vários níveis de abstracção. Eles fornecem uma boa separação da *interface* da unidade de controlo da sua *implementação*. É muito importante que suportem *hierarquia*.

A abordagem considerada torna possível sintetizar uma unidade de controlo que implementa um dado comportamento descrito por um HGS. O esquema da unidade pode ser construído tendo por base *estruturas predefinidas* e *templates* consideradas em [1,3,4]. De forma a fornecer à unidade de controlo novas facilidades, tais como extensibilidade, flexibilidade e reutilização tentámos combinar os resultados de duas áreas bem conhecidas e intimamente relacionadas, e que são a teoria de *máquinas de estados finitas* e a *programação orientada a objectos*. Contudo esta tentativa deve ser considerada como o primeiro passo. O trabalho futuro será continuado nas seguintes direcções:

- Construir **estruturas predefinidas orientadas para HGS** e ***templates*** (esquemas básicos) [1,3,4] que possam ser usadas para uma variedade de aplicações práticas, que suportem *hierarquia* e que executem macro operações em paralelo;

- Desenvolver novos métodos eficientes de **síntese lógica** destinados a usarem as estruturas predefinidas e *templates* mencionadas no ponto anterior;
- Procurar semelhanças entre várias instruções usando explicitamente a **relação de herança**. Investigar **instruções derivadas** e as suas propriedades. Desenvolver novos métodos eficientes e componentes de *hardware* que proporcionem a **redefinição** e a **substituição** de micro operações (funções lógicas). Estes métodos estão intimamente relacionados com as macro operações virtuais e funções lógicas virtuais. A abordagem respectiva envolve a ideia de **polimorfismo** no desenho de FSM (e leva-nos à introdução de FSM polimórficas com estados ordinários e virtuais);
- Fornecer **suporte em *run-time*** baseado na reprogramação de componentes usados no esquema depois da unidade de controlo ter sido desenhada (durante a execução do algoritmo). Estas questões estão intimamente relacionados com ***early binding*** (estática) e ***late binding*** (dinâmica) de macro operações e funções lógicas;
- Fornecer um **mecanismo de *exception handling*** baseado na ideia [1].

Toda a informação adicional relacionada com grafos ordinários e a sua utilização para a síntese lógica de máquinas de estados finitas pode ser encontrada em [8].

REFERÊNCIAS

[1] Valery Sklyarov Applying Finite State Machine Theory and Object-Oriented Programming to the Logical Synthesis of Control Devices. Electronica e Telecomunicacoes, 1996, N 6, pp 515-529.

[2] Valery Sklyarov Hierarchical Graph-Schemes. Latvian Academy of Science, Automatics and Computers, Riga, 1984, N 2, pp 82-87.

[3] Samary Baranov, Valery Sklyarov. Digital Devices Based on Programmable Matrix LSI. Moscow, Radio and Communications, 1986, 272 p.

[4] Valery Sklyarov Synthesis of Finite State Machines Based on Matrix LSI. Minsk, Science and Technique, 1984, 287 p.

[5] Valery Sklyarov Parallel Graph-Schemes and Finite State Machines Synthesis. Latvian Academy of Science, Automatics and Computers, Riga, 1987, N 5, pp 68-76.

[6] Grady Booch Object-Oriented Analysis and Design. Second Edition. The Benjamin/Cummings Publishing Company, Inc., 1994, 589 p.

[7] Bjarne Stroustrup The C++ Programming Language. Second Edition, Addison Wesley Publishing Company, 1994, 691 p.

[8] Samary Baranov Logic Synthesis for Control Automata. Kluwer Academic Publishers, 1994, 392 p.