HiParaGraphs, uma Linguagem de Especificação de Algoritmos de Controlo Paralelos e Hierárquicos

Andreia Melo, Valery Sklyarov, António Ferrari

Resumo **– Este artigo apresenta um método de especificação utilizado para descrever o comportamento de uma unidade de controlo digital. É uma linguagem formal que resultou da evolução da sua antecessora, os HGSs – Hierarchical Graph-Schemes. Denominada *HiParaGraphs* – Hierachical & Parallel Graphs, esta linguagem pretende acrescentar novas facilidades de paralelismo aos algoritmos de controlo, além de melhorar a especificação anterior com a introdução de novos parâmetros e diferentes tipos de nodos.**

Abstract **- This paper presents a specification method for describing the behavior of digital control units. It is a formal language that results from an evolution of its predecessor, the HGSs – Hierarchical Graph-Schemes. It is called HiParaGraphs – Hierarchical and Parallel Graphs, and the main objective was to add new facilities, mainly related to parallelism.**

I. INTRODUÇÃO

O trabalho desenvolvido na área da especificação e projecto de sistemas complexos e reactivos tem resultado em vários formalismos, tais como os Statecharts [1] e as redes de Petri [2]. Neste artigo descreve-se uma linguagem de especificação cujo objectivo é integrar algumas propriedades dos métodos já existentes e acrescentar novas funcionalidades. Os sistemas reactivos são caracterizados por reagir continuamente a eventos, isto é, a estímulos internos e externos. Estes sistemas são utilizados em inúmeras áreas, tais como em telecomunicações, sistemas operativos, sistemas de aviação, etc. Vamos aqui debruçar-nos sobre a parte de controlo de um dado sistema digital, também de características reactivas, a que chamaremos unidade de controlo.

A dificuldade reside na descrição do comportamento reactivo de forma clara e realista e ao mesmo tempo formal e rigorosa. O comportamento de um sistema reactivo é um conjunto de sequências de entrada permitidas e eventos de saída, condições, acções e eventualmente algumas restrições temporais.

Os Grafos Hierárquicos e Paralelos (HiParaGraphs - *Hierarchical & Parallel Graphs*) são uma linguagem de especificação gráfica formal de algoritmos de controlo que se baseiam no formalismo da máquina de estados finitos (FSM – *Finite State Machine*). Um HiParaGraph descreve algoritmicamente o comportamento de unidades de controlo digitais, sendo composto por diferentes tipos de nodos ligados entre si, por intermédio de arcos direccionados, que estabelecem o fluxo de execução do algoritmo. Desta forma, a sua construção é análoga à de um fluxograma.

A linguagem de especificação que irá ser aqui descrita resultou de uma evolução de linguagens anteriormente propostas, às quais se adicionou alguma funcionalidade. Inicialmente denominada GSs (*Graph-Schemes*) [3], esta linguagem especificava algoritmos de controlo simples que depois evoluíram para os Esquemas de Grafos Hierárquicos (*Hierarchical Graph-Schemes* – HGSs) [4]. Neste passo foi acrescentada a hierarquização dos algoritmos de controlo, composta por um algoritmo principal e vários sub-algoritmos que são executados um de cada vez, ou seja, um algoritmo invoca outro e este quando terminar a sua execução retorna àquele que o invocou. O passo posterior foi a especificação do paralelismo onde se acrescentou aos HGSs a possibilidade de invocar dois ou mais sub-algoritmos simultaneamente de forma a serem executados ao mesmo tempo. Este é o ponto actual da evolução e é onde pretendemos adicionar novas formas de sincronismo no âmbito da execução paralela. Como por exemplo, os sub-algoritmos podem iniciar a sua execução e não retornar à parte invocadora, podem ser invocados indirectamente a partir de dois ou mais sub-algoritmos que estejam a ser executados, ou até a parte invocadora pode ter a possibilidade de terminar um algoritmo que invocou, qualquer que seja o estado deste. Estas são algumas das características da linguagem de especificação gráfica que será descrita ao longo deste artigo, os HiParaGraphs.

Este artigo está dividido em sete secções. A primeira corresponde à presente introdução. Na segunda secção descreve-se brevemente as características gerais da linguagem. A secção três é dedicada à especificação linear dos circuitos de controlo. Na quarta secção é apresentada a funcionalidade hierárquica deste formalismo. O paralelismo dos HiParaGraphs é abordado na secção cinco. A sexta secção descreve as novas funcionalidades da ferramenta de software que suporta este método de especificação. A secção sete contém as conclusões.

II. CARACTERÍSTICAS GERAIS DOS HIPARAGRAPHS

Ao formalismo clássico das FSMs estão associados os diagramas de transição de estados. Estes diagramas não fornecem a noção de hierarquia e de modularidade, não

permitindo, consequentemente, a utilização de técnicas de decomposição descendente ou montagem ascendente durante o desenvolvimento de um sistema. O número de transições num diagrama pode ser bastante elevado. Por exemplo, um evento que causa transições iguais num grande número de estados, como é o caso de uma interrupção de elevada prioridade, é representado por um elevado número de arcos iguais, resultando assim numa quantidade enorme de transições e complicando bastante a representação. Outra desvantagem deste método é o crescimento exponencial do número de estados quando o sistema apenas cresce linearmente.

Os Statecharts [1] resolvem estes problemas fornecendo uma representação modular, hierárquica e concorrente. A comunicação entre FSMs é também uma característica deste método.

As redes de Petri [2] são extremamente eficientes na descrição de sistemas concorrentes e paralelos, utilizando uma representação simples, também à base de estados (*places*) e de transições. Graficamente os *places* são circunferências e as transições são linhas. A interligação entre estes dois componentes é efectuada por intermédio de arcos direccionados. Uma desvantagem da representação visual deste método é a utilização de um número muito reduzido de símbolos que implica que a detecção das várias situações de sequenciação, sincronismo, paralelismo e concorrência numa descriçao complexa seja efectuada não de uma forma directa mas sim após uma análise da rede. A hierarquia e a modularidade também não são explicitamente representáveis.

Os HiParaGraphs resultam do estudo e da integração de métodos formais já existentes. Nasceram das propriedades sequenciais e simples dos diagramas de transição de estados e evoluíram de forma a combinar a hierarquia, modularidade e comunicação dos Statecharts com o paralelismo das redes de Petri e dos Algoritmos de Controlo Lógicos [6]. Utilizando uma notação gráfica e intuitiva, permitem o desenvolvimento top-down do circuito pretendido.

Como elementos gráficos temos nodos, cuja forma depende da função, e arcos direccionados. Os nodos encapsulam operações de teste em conjuntos de entradas e acções nas saídas. Os arcos direccionados interligam os vários tipos de nodos. Com apenas estes recursos visuais estamos aptos a descrever algoritmos de controlo lineares e sequenciais. No entanto, se considerarmos um algoritmo de controlo como um conjunto de sub-algoritmos, necessitamos de nodos com novas funcionalidades de forma a prever situações não só de sequenciação, como era o caso anterior, mas também de sincronismo e paralelismo. Por outro lado, os vários sub-algoritmos podem ser organizados segundo uma hierarquia e assim será necessária a especificação de invocações.

Resumidamente, a actualização desta linguagem baseou-se na adição de novos tipos de nodos (*Sync*, e *Switch*) e na utilização de parâmetros de entrada em sub-algoritmos, relativamente aos HGSs, que enriqueceram este método, fornecendo ao algoritmo novas funcionalidades que se revelaram interessantes durante o projecto das unidades de controlo.

Um HiParaGraph possui quatro categorias de nodos: **iniciação**, **terminação**, **activação** e **teste**.

Os nodos de iniciação são denominados *Begin*. Como são sempre os primeiros nodos de cada sub-algoritmo, poderão mostrar uma lista de parâmetros de entrada, cuja activação determinará o início da execução do respectivo algoritmo. Desta forma acrescentou-se a especificação de sincronismo.

Os nodos de terminação são *End* (obrigatório, para determinar o final de um sub-algoritmo) e *Halt* que pode ser usado para suspender a execução de um sub-algoritmo no caso de ser detectada uma situação de excepção.

Tal como o nome indica, os nodos de activação são utilizados para activar sinais. Os vários tipos de nodos que se enquadram nesta categoria são os Operacionais, que são usados tanto para activar sinais de saída como para invocar sub-algoritmos. A estes nodos acrescentaram-se indicadores de activação quer em sinais de saída quer em invocações de sub-algoritmos. Estes, por sua vez, podem ser especificados utilizando parâmetros de entrada e desta forma podem ser activados a partir de um ou mais sub-algoritmos diferentes durante a sua execução, isto é, a invocação de um dado procedimento poderá não depender unicamente de um mas de um conjunto de estados de sub-algoritmos diferentes. Para este efeito, ou seja, o sincronismo no caso de algoritmos paralelos, são utilizados os nodos *Sync*. Os nodos do tipo *Dummy* são um caso particular dos Operacionais pois como não actuam sobre qualquer sinal, representam um estado de espera. Finalmente, os *Assign* utilizam-se para devolver o resultado (verdadeiro ou falso) das funções.

Os nodos de teste são usados para testar o valor lógico dos sinais de entrada, a que chamamos condições lógicas. Nesta classe temos os nodos do tipo Condicionais com Condição Lógica, que testam as condições lógicas e os Condicionais com Função Lógica que são responsáveis pela invocação de uma função lógica. Os *Switch* encapsulam uma rede de nodos com condições lógicas pois verificam uma dada sequência destes sinais. A funcionalidade de cada tipo de nodo será descrita mais à frente com maior detalhe.

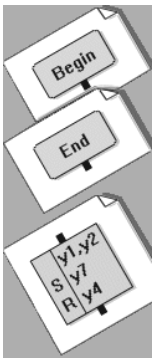
III. ESPECIFICAÇÃO NÃO HIERÁRQUICA

Os circuitos reactivos que estamos aqui a considerar, tal como já foi mencionado na introdução deste artigo, são aqueles que constituem a parte de controlo de um sistema digital que se pode dividir em unidade de controlo e unidade de execução. A primeira é a parte do sistema que pretendemos especificar o comportamento e cuja função é produzir uma sequência de operações na unidade de execução. Estas operações (elementares), são chamadas microoperações. $Y=\{y_1,\ldots,y_n\}$ é o conjunto de microoperações induzidas pelos sinais binários $y_1,\ldots,y_n$ da

unidade de controlo. Para activar a microoperação y_n (n=1,…N) o sinal y_n=1 deverá ser activado. Por vezes algumas microoperações são executadas em simultâneo (no mesmo ciclo de relógio) na unidade de execução. A sequência de activação das microoperações é determinada por funções de transição, isto é, funções Booleanas α_{ij} (i,j=1,…T) de variáveis Booleanas X={x_1,…x_L}. Estas variáveis correspondem às variáveis de entrada da unidade de controlo, também denominadas condições lógicas, e podem ser alteradas pelas microoperações.

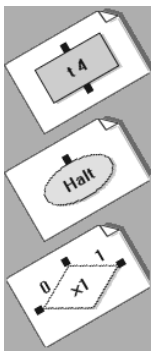
Um HiParaGraph é uma linguagem gráfica que dispõe de vários tipos de nodos para especificar facilmente o comportamento de um algoritmo de controlo. Recorrendo a dois tipos de nodos, teste e activação, um HiParaGraph é capaz de testar o valor binário de um sinal de entrada ou de uma expressão Booleana de variáveis de entrada e activar ou desactivar os valores das saídas. Os nodos possuem pontos de conexão onde se ligam os arcos direccionados permitindo assim estabelecer ligações com outros nodos. No entanto, existe uma restrição relativamente às ligações que consiste numa única ligação de um ponto de saída de um nodo. Os pontos de entrada podem estar ligados a um ou mais nodos. Algumas regras básicas são a obrigatoriedade de um nodo se situar num caminho existente entre o nodo *Begin* e o nodo *End*. Desta forma estamos a garantir que se encontra correctamente ligado e que não pertence a um ciclo infinito. As saídas dos nodos condicionais não se devem ligar à própria entrada pois desta forma estamos a assumir que existe um estado de espera que não está representado explicitamente no algoritmo e para este efeito existe um nodo específico denominado *Dummy*.

Os vários tipos de nodos que constituem um HiParaGraph não hierárquico são os seguintes:



Os nodos *Begin* e *End* representam os pontos de entrada e de saída do algoritmo de controlo, respectivamente, e a sua presença num HiParaGraph é obrigatória e única.

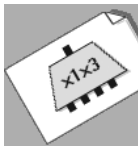
Originalmente [3], [4] os nodos operacionais eram utilizados para especificar a activação dos sinais de saída da unidade de controlo. No seu interior são colocadas listas de microoperações, que correspondem a esses sinais de saída. Além disto foi acrescentada a possibilidade de utilizar indicadores de activação nestes sinais de forma individual. Isto permite-nos activar (indicador SET - "S") ou descativar (indicador RESET - "R") um dado sinal durante um ou mais ciclos de relógio, ou durante um período de tempo correspondente a um ou mais estados do circuito. Os indicadores são colocados à esquerda da lista de microoperações que irá ser afectada. O número de ciclos de relógio durante o qual este nodo permanecerá activo pode também ser aqui especificado. É indicado por um número inteiro precedido pela letra "t" e por defeito é igual a 1.

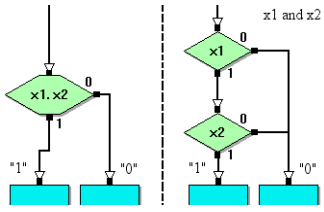


Os nodos *Dummy* são um caso particular dos nodos operacionais pois não contêm microoperações e são utilizados apenas para introduzir um estado de espera. Por defeito o seu tempo de activação é um ciclo de relógio, mas tal como no caso anterior, podem ser especificados múltiplos de ciclos de relógio. No exemplo da figura ao lado temos no topo um nodo *Dummy* que ficará activo durante 4 ciclos de relógio.

Os nodos *Halt* indicam o final da execução do algoritmo no caso de este ter de ser interrompido numa situação de erro ou de excepção. Quando a execução do algoritmo atinge este nodo assume-se que o próximo estado da execução corresponde a ele próprio. Difere do nodo *End* pois, no caso de uma hierarquia, o *End* significa retorno enquanto o *Halt* significa paragem. Os nodos condicionais possuem uma funcionalidade configurável pois podem conter quer uma condição lógica pertencente ao conjunto X={x_1,…,x_L}, que corresponde a uma entrada do circuito de controlo quer um subconjunto de condições lógicas às quais é aplicada uma função Booleana (por exemplo, AND, OR). Este nodo possui um ponto de conexão de entrada e dois pontos de saída que correspondem aos valores lógicos 0 e 1 do sinal binário da entrada ou do resultado da função Booleana de variáveis de entrada nele especificadas. Os nodos condicionais que possuem conjunções e disjunções de variáveis de entrada (Fig. 1) são úteis quer para encapsular por si só vários nodos condicionais num só, quer para facilitar o teste do valor lógico de uma condição mais extensa, composta por conjunções e disjunções (somas de produtos, produtos de somas, etc).

A especificação de funções Booleanas nos nodos condicionais permite diminuir o número de nodos por algoritmo e consequentemente facilita a tarefa de verificação e de síntese. Também por este motivo foi criado mais um tipo de nodo de teste a que chamamos *Switch*. O seu comportamento compara-se ao de um descodificador com um número de entradas variável. O número de variáveis de entrada que podem ser colocadas neste nodo pode ser 1, 2 ou 3 e por isso teremos 2, 4 ou 8 saídas, respectivamente.





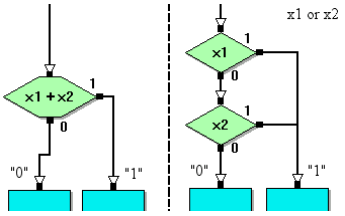


Fig. 1 – Encapsulamento de funções Booleanas AND e OR em nodos condicionais com duas variáveis.

A utilização destes nodos reduz significativamente o número de nodos condicionais numa construção equivalente. A introdução de uma variável de entrada implica a replicação de nodos condicionais já existentes, complicando assim a compreensão do algoritmo. Um nodo *Switch* com 3 entradas é equivalente a uma rede de 7 nodos condicionais, tal como mostra a Fig. 2. Como se pode concluir, uma construção deste tipo iria tornar o algoritmo de controlo mais complexo e sujeito a erros. A verificação das regras de especificação é simplificada pelo uso deste tipo de nodos visto que encapsulam uma função conhecida, que não precisa de ser construída manualmente, e que à partida cumpre todas as regras dos HiParaGraphs.

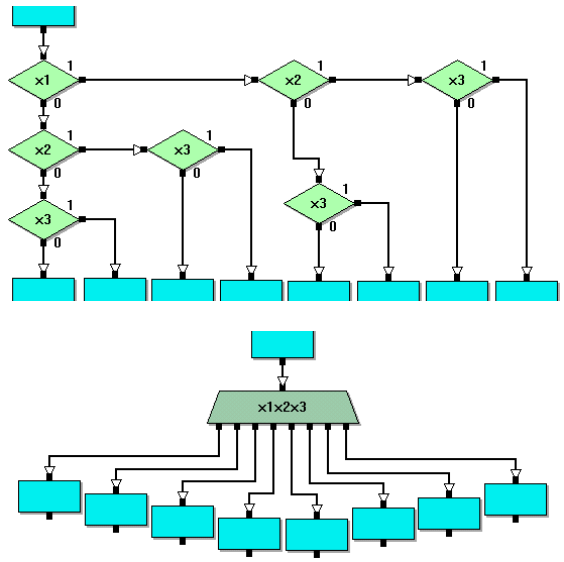


Fig.2 – Nodo Switch – construção equivalente com nodos condicionais.

O exemplo da Fig. 2 pretende mostrar a simplificação obtida pela utilização de um *Switch* de três variáveis. À saída deste nodo temos, da esquerda para a direita, todas as combinações binárias possíveis, de 0 a 7, respectivamente. A rede de nodos condicionais representada acima torna-se assim evitável eliminando-se uma possível fonte de erros e facilitando a tarefa de verificação automática.

Quer os nodos condicionais quer os nodos *Switch* podem ser ligados de forma a construírem funções mais complexas que deste modo são interpretadas de forma modular na própria especificação do algoritmo.

Os algoritmos de controlo especificados através de um HiParaGraph são sequenciais e podem ser modelados por uma máquina de estados finitos. Por exemplo, se considerarmos uma máquina de Moore, podemos associar os nodos de iniciação, activação e terminação aos vários estados do circuito, sendo as transições de estado efectuadas nos flancos ascendentes ou descendentes do sinal de relógio. O cálculo do estado seguinte depende dos nodos de teste existentes no algoritmo. A sequenciação, está representada na Fig. 3, onde é comparada com a especificação equivalente de uma rede de Petri. Estes algoritmos são equivalentes se considerarmos que o *place* (circunferência do topo da Fig.3b)) da rede de Petri que possui o *token* corresponde ao nodo operacional que activa a microoperação y_1. Os traços horizontais da rede de Petri representam transições, dependentes de condições, assim como os nodos de teste de um HiParaGraph. Na Fig. 3b), se a condição da primeira transição for verdadeira o *token* passará para o *place* seguinte, dando-se assim a transição de estado, caso contrário o *token* permanecerá no mesmo *place*. No caso do HiParaGraph, Fig.3a), se a condição lógica x_1 for verdadeira, do nodo operacional que activa y_1 passa-se para o nodo operacional que activa y_2. Por outro lado, se a condição for falsa, o estado activo do circuito continuará a ser o nodo operacional responsável pela activação de y_1.

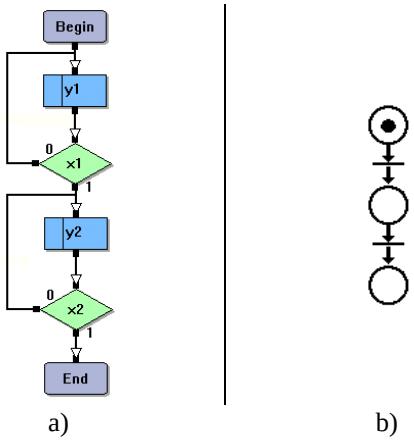


Fig. 3 – Especificação de sequenciação a) num HiParaGraph e b) numa rede de Petri.

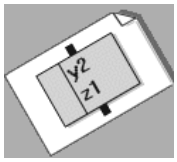
IV. ESPECIFICAÇÃO HIERÁRQUICA

A especificação hierárquica de um algoritmo de controlo justifica-se quando o comportamento de uma unidade de controlo é descrito por um algoritmo extenso, de tal forma que se poderia dividir em módulos, facilitando tanto a compreensão da especificação como a implementação do respectivo circuito. A descrição formal das máquinas de estados finitos hierárquicas (HFSM – *Hierarchical Finite State Machine*) [5] diz que apenas um nível hierárquico está activo em cada instante. Um dado algoritmo que é implementado como uma FSM invoca outro que será executado num nível hierárquico inferior.

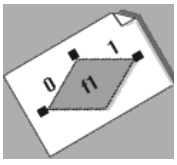
Quando este terminar a sua execução retorna ao algoritmo que o invocou, ou seja, ao nível imediatamente superior.

Os HiParaGraphs, herdando algumas características dos HGSs [4], disponibilizam uma descrição modular de algoritmos de controlo. Quando especifica uma hierarquia um HiParaGraph é composto por um conjunto de sub-algoritmos. Os vários módulos podem possuir características diferentes e por isso foram divididos em dois tipos: **funções** e **procedimentos**. Esta divisão baseou-se apenas na utilização ou não de parâmetros de saída, respectivamente. Tal como é usual em algumas linguagens de programação, como por exemplo o Pascal, as funções devolvem valores ao algoritmo onde são invocadas e os procedimentos apenas executam um conjunto de tarefas sem devolver ou calcular resultados.

Assim, um HiParaGraph, descrevendo uma unidade de controlo hierárquica, necessita de adicionar novos tipos de nodos ao conjunto apresentado na secção anterior.

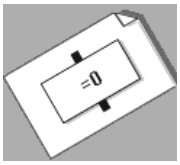


Para representar as invocações a procedimentos (macrooperações) são utilizados nodos operacionais, tais como foram descritos anteriormente, aos quais se adiciona uma macrooperação do conjunto $Z=\{z_1, z_2, \ldots, z_N\}$. Estes nodos podem simultaneamente especificar a activação de sinais de saída.



A utilização de funções deve-se à necessidade de poder influenciar o fluxo de execução do algoritmo invocador a partir de um valor que é determinado e devolvido pela função, tal como uma condição lógica, decidindo assim qual o próximo estado activo do circuito. Aos nodos de teste que se utilizam para invocar funções chamam-se nodos condicionais com funções lógicas e contêm um elemento do conjunto $F=\{f_1, f_2, \ldots, f_J\}$.

Para devolver o resultado (verdadeiro ou falso) de uma função são usados dois nodos de activação do tipo Assign



que devem pertencer à especificação da função, um com o valor 0 e outro com o valor 1. Ambos devem ser ligados ao nodo End, pois o retorno do resultado deverá ser a última operação efectuada por este sub-algoritmo.

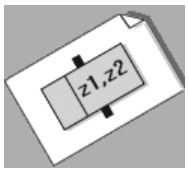
Recursividade

Os vários sub-algoritmos podem invocar outros e até mesmo eles próprios. Desta forma um HiParaGraph poderá especificar **algoritmos de controlo recursivos**. Não existe uma regra que evite a especificação de algoritmos infinitamente recursivos mas, no entanto, esta situação pode ser detectada mediante a construção de um HiParaGraph equivalente, não hierárquico, ao qual se extraem todos os nodos de activação (excepto aqueles utilizados para invocar sub-algoritmos). Isto resulta numa árvore de invocações onde facilmente se detectam ciclos, que podem corresponder a situações de recursividade infinita. Se existir sempre um caminho a partir de um dado nodo para o *End*, significa que a hierarquia especificada não é infinitamente recursiva. Os possíveis ciclos de invocações presentes nesta construção podem eventualmente ser interrompidos por nodos condicionais, ou seja, a execução cíclica pode depender de alguma variável de entrada. Neste caso a execução correcta do algoritmo depende do modelo de implementação escolhido e dos recursos de hardware disponíveis. Por exemplo, um modelo de implementação de uma HFSM que utilize um *stack* está limitado, no que diz respeito à recursividade, devido à sua profundidade. Se o número de invocações recursivas ou até mesmo o número de níveis hierárquicos da máquina for maior do que a profundidade disponível do *stack*, então a dada altura da execução do circuito de controlo chegaremos a uma situação de *stack overflow*. O conhecimento prévio da profundidade do *stack* e a construção do HiParaGraph equivalente, representando a árvore de invocações, poderá evitar esta situação.

V. ESPECIFICAÇÃO PARALELA HIERÁRQUICA

Para especificar o comportamento de sistemas reactivos, como é o caso das unidades de controlo que estamos a considerar, pode-se utilizar vários tipos de linguagens com diferentes filosofias e objectivos, mas preferencialmente orientadas ao estado. Este é o caso dos HiParaGraphs cujo objectivo é descrever o comportamento de um sistema que suporte hierarquia e paralelismo. Combinando algumas propriedades dos Statecharts [1], redes de Petri [2] e Algoritmos de Controlo Lógicos [6], obtivemos uma forma de integrar numa linguagem gráfica as características disponibilizadas por cada uma destas linguagens.

Da definição formal de HFSM apenas um nível hierárquico está activo em cada instante e tal como foi dito na secção anterior, quando pretendemos invocar um



procedimento colocamos uma macrooperação dentro de um nodo operacional.

No entanto, se em vez de apenas uma macrooperação forem especificadas duas ou mais dentro do mesmo nodo, isto significa que no instante em que este nodo for atingido elas serão invocadas simultaneamente. Assim, os procedimentos deverão ser executados em paralelo. Esta situação está representada na Fig. 4, onde também é comparada com uma construção equivalente numa rede de Petri. No exemplo da Fig. 4b), se a condição atribuída à transição for verdadeira, o *token* transitará do seu *place* para ambos os *places* que existem após a transição, ficando assim os dois activos simultaneamente. A Fig. 4a) possui a representação equivalente de um HiParaGraph, onde o nodo operacional especifica a invocação de duas macrooperações simultaneamente. Após este nodo serão activados os nodos *Begin* de cada um dos procedimentos. Neste caso o algoritmo invocador

prossegue a sua execução, mas poderia não o fazer mediante a utilização de um indicador.

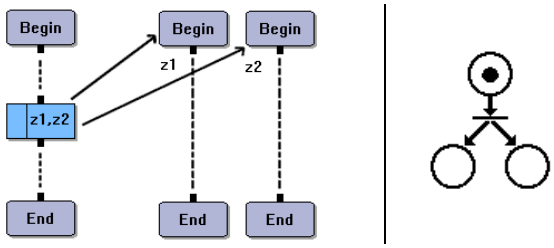
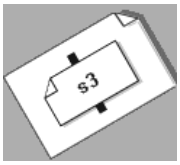


Fig. 4 – Especificação de paralelismo a) num HiParaGraph e b) numa rede de Petri.

Tal como as microoperações podem ter indicadores de activação associados, as macrooperações podem ser especificadas de forma a que o algoritmo que as invoca espere ou não pela sua terminação para poder prosseguir a execução. Assim, uma macrooperação afectada pelo indicador W (*wait*), utilizado nos HiParaGraphs, significa que o algoritmo invocador deve esperar pelo seu retorno. O indicador W é atribuído individualmente a cada macrooperação, o que significa que quando colocado num nodo operacional, pode afectar mais do que uma macro, ou um subconjunto daquelas que lá existem. Neste caso o algoritmo deve esperar pelas macrooperações com W mas caso as restantes ainda não tiverem terminado a sua execução, ele prosseguirá.

Não só os nodos operacionais que possuem macrooperações são os responsáveis pela invocação de procedimentos. Os nodos do tipo *Sync* são utilizados para invocações indirectas que especificam situações de sincronismo, tal como nas redes de Petri (ver Fig. 5).

Nestes nodos são colocados sinais pertencentes ao conjunto $S=\{s_1, s_2, \ldots, s_M\}$ e que são utilizados na comunicação entre sub-algoritmos.



Os nodos *Sync* foram criados para que seja possível invocar um procedimento ou função quando se atingem determinados estados de sub-algoritmos diferentes. Para tal é também necessário especificar parâmetros de entrada nos sub-algoritmos invocados desta forma. Tal como mostra a Fig. 5a), os nodos *Sync*, pertencentes a procedimentos ou funções diferentes, activam os sinais s_2 e s_4, eventualmente em instantes de tempo diferentes. O novo procedimento só inicia a sua execução quando os sinais que são especificados nos seus parâmetros de entrada estiverem ambos activos, ou seja, espera pela activação dos dois nodos *Sync* para poder iniciar a sua execução. Os algoritmos que possuem os nodos do tipo *Sync* prosseguem a sua execução normalmente, pois estão apenas a actualizar o valor de um sinal de saída.

Por outro lado, na Fig. 5b), a rede de Petri possui uma transição que, embora a sua condição seja verdadeira, só pode disparada quando os *places* que a precedem possuírem *tokens*, tal como está representado na figura. Assim, esta transição equivale àquela que existe a seguir ao nodo *Begin* da Fig. 5a). Neste caso, tal como mostra a figura, pressupõe-se a existência de dois sub-algoritmos diferentes pois não é possível num HiParaGraph existir dois estados activos no mesmo sub-algoritmo.

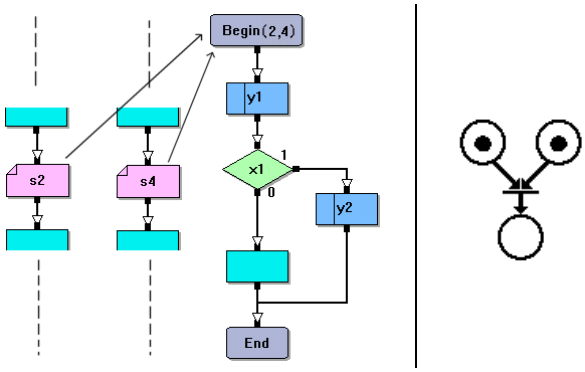


Fig. 5 – Especificação de sincronismo a) num HiParaGraph e b) numa rede de Petri.

VI. SOFTWARE – EDIÇÃO GRÁFICA

A aplicação de software que suporta esta linguagem de especificação chama-se GraphBuilder. A sua versão anterior [7], [8] foi melhorada de forma a integrar as novas características da especificação relativas ao paralelismo.

A janela principal desta aplicação encontra-se na Fig. 6, mostrando um HiParaGraph já construído. Além da especificação esta ferramenta integra também outras facilidades, nomeadamente a verificação, marcação de estados (Mealy ou Moore), optimização, conversão para código VHDL comportamental sintetizável e depuração. No entanto, estas tarefas não serão aqui explicadas.

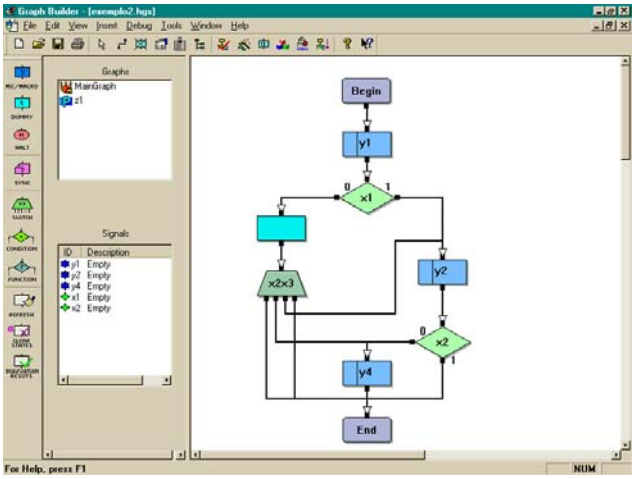


Fig. 6 – Janela principal da ferramenta GraphBuilder.

Nas secções anteriores foram apresentados os diferentes tipos de nodos que constituem um HiParaGraph. Todos eles pertencem a uma hierarquia de classes implementáveis em Visual C++ e cuja estrutura está representada na Fig. 7. Esta estrutura hierárquica foi modificada relativamente às versões anteriores, pois com o aumento do número de nodos foi necessária uma divisão de classes de acordo com a funcionalidade de cada um.

Como estamos na presença de uma especificação hierárquica, a primeira classe representa a hierarquia e chama-se *CHierGraph*. Uma hierarquia é composta por sub-algoritmos e por isso esta classe inclui a *CGraphScheme* que por sua vez possui vários nodos (no mínimo 2, *Begin* e *End*). Assim, esta inclui a classe *CNode* e a partir daqui são derivados os quatro tipos de nodos de que dispomos: iniciação – *CInitiationNode*, terminação – *CTerminationNode* , teste – *CTestNode* e activação – *CActivationNode*. Como temos apenas um tipo de nodo para iniciar os vários sub-algoritmos, deriva-se a classe *CBeginNode* (nodo *Begin*) da *CInitiationNode*. No caso dos nodos de terminação temos as classes *CEndNode* (nodo *End*) e *CHaltNode* (nodo *Halt*) derivadas da *CTerminationNode*. Para teste deriva-se da classe *CTestNode* a *CLogCondNode* (nodos condicionais com condições lógicas), *CLogFuncNode* (nodos condicionais com funções lógicas) e *CSwitchNode* (nodo *Switch*). Finalmente, para activação, deriva-se da classe *CActivationNode* a classe *COpNode* (nodos operacionais), *CDummyNode* (nodos *Dummy*), *CAssignNode* (nodos *Assign*) e *CSyncNode* (nodos *Sync*).

Todas estas classes estão armazenadas numa DLL (*Dynamic Link Library*) que é carregada quando a aplicação é iniciada.

Os sub-algoritmos de um HiParaGraph são entidades autónomas e podem ser reutilizados em diferentes algoritmos e consequentemente em diferentes unidades de controlo. Assim, a ferramenta GraphBuilder permite gravar não só um HiParaGraph completo (objecto da classe *CHierGraph*), mas também os seus componentes, isto é, permite a gravação individual de cada sub-algoritmo (objectos do tipo *CGraphScheme*).

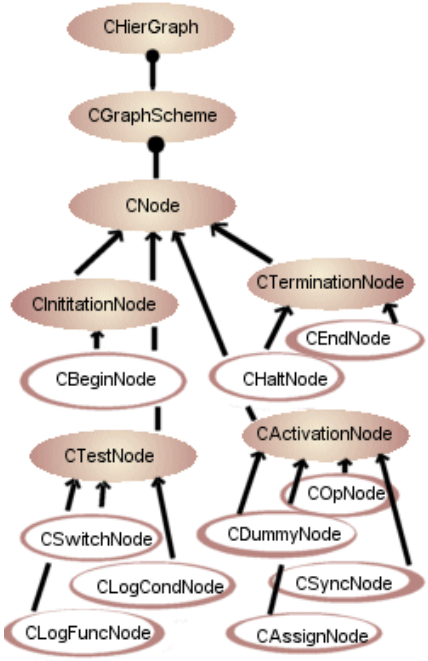


Fig. 7 – Hierarquia de classes para implementação de um HiParaGraph.

VII. CONCLUSÕES

A especificação de algoritmos de controlo através de HiParaGraphs suporta explicitamente hierarquia (o que não acontece com as redes de Petri e os Algoritmos de Controlo Lógicos) e paralelismo. Embora os Statecharts possuam também estas características, não existe uma forma automática de sintetizar e implementar os circuitos especificados por este método. Isto não acontece com os HiParaGraphs que além de tentarem combinar as características e vantagens dos vários métodos formais mencionados, são sintetizáveis e implementáveis. Embora não tenha sido descrito neste artigo, a ferramenta de software que suporta este método, integra uma aplicação que é capaz de converter os HiParaGraphs em código VHDL comportamental sintetizável. Essa aplicação, denominada SynGraph, utiliza um *template* em código VHDL que pode ser usado para sintetizar máquinas de estados finitos, ao qual efectua as alterações necessárias para descrever o comportamento do circuito desejado. O conjunto de ferramentas de software desenvolvidas para suportar este método de especificação permite modificar e expandir facilmente os algoritmos de controlo construídos em qualquer fase do projecto do circuito [9], [10].

Com este método pretende-se automatizar o projecto de unidades de controlo partindo de uma especificação facilmente extensível e sintetizável. Os HiParaGraphs são uma descrição gráfica formal independente do modelo de implementação do circuito.

REFERÊNCIAS

[1] David Harel, "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming, Nº8, pp. 231-271, 1987.

[2] Tadao Murata, "Petri Nets: Properties, Analysis and Applications", Proceedings of the IEEE, Vol. 77, Nº4, pp.541-590, Abril 1989.

[3] Samary Baranov, "Synthesis of Microprogrammed Automata", Energy Publishing Company, 1974 (em russo).

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

[5] Arnaldo Oliveira, "Modelos, Métodos e Ferramentas para Implementação de Unidades de Controlo Virtuais", Dissertação de Mestrado em Engenharia Electrónica e de Telecomunicações, Universidade de Aveiro, Julho 2000.

[6] A. D. Zakrevskij, "Parallel Algorithms for Logical Control", Institute of Engineering Cybernetics of NAS of Belarus, Minsk, 1999.

[7] A. Oliveira, A. Melo, V. Sklyarov, "Specification, Implementation and Testing of HFSMs in Dinamically Reconfigurable FPGAs", actas de FPL'99, 9th International Workshop on Field Programmable Logic and Applications, Glasgow, Reino Unido, pp. 313-322, Agosto 1999.

[8] Andreia Melo, "Especificação, Optimização e Teste de Algoritmos de Controlo Hierárquicos", Dissertação de Mestrado em Engenharia Electrónica e de Telecomunicações, Universidade de Aveiro, Janeiro 2000.

[9] Sklyarov, V., Monteiro, R. S., Lau, N., Melo, A., Oliveira, A., Kondratjuk, K., "Integrated Development for Logic Synthesis Based on Dynamically Reconfigurable FPGAs", *Field-Programmable Logic and Applications*, 8th International Workshop, FPL '98, Tallinn, Estonia, Agosto 1998.

[10] Sklyarov, V., Lau, N., Oliveira, A., Melo, A., Kondratjuk, K., Ferrari, A. B., Monteiro, R. S., Sklyarova, I., "Synthesis tools and design environment for dynamically reconfigurable FPGAs", *SBCCI98 XI Brazilian Symposium on Integrated Circuit Design*, Búzios, Rio de Janeiro, Brazil, Setembro 1998.