Especificação e Simulação Interactiva de Algoritmos de Controlo Paralelos e Hierárquicos*

Andreia Melo, Valery Sklyarov, António Ferrari

Resumo – **O método de especificação apresentado neste artigo é utilizado para descrever o comportamento de uma unidade de controlo digital. Os algoritmos de controlo que gerem esta unidade seguem um modelo hierárquico, paralelo ou uma conjugação de ambos. Neste artigo é abordada a especificação e a simulação dos algoritmos de controlo paralelos e hierárquicos em simultâneo. O estudo de caso apresentado neste artigo consta da especificação da unidade de controlo do controlador CAN – *Controller Area Network* utilizando os HiParaGraphs.**

Abstract – **The specification method, which is presented in this paper, is used to describe the behavior of a digital control unit. The control algorithms that manage this unit may be hierarchical, parallel and both. In this paper is shown the specification and simulation of hierarchical and parallel control algorithms at the same time. The CAN - *Controller Area Network* protocol was used as a case study being specified using HiParaGraphs.**

I. INTRODUÇÃO

Existem sistemas digitais que se podem dividir em duas unidades, a unidade de controlo e a unidade de execução. A primeira, que vai ser abordada neste artigo, tem como função estabelecer a sequência de operações que deverão ser efectuadas pela unidade de execução. Assim, é necessário definir um algoritmo de controlo para decidir a sequência de operações. Dependendo da complexidade da unidade de controlo em causa, podemos utilizar vários tipos de algoritmos. O caso mais simples é a existência de um único algoritmo que estabelece uma sequência de operações, baseado ou não em valores lógicos de sinais de entrada, e por isso com características reactivas, actuando em sinais de saída binários. No entanto, a complexidade da unidade de controlo pode ser tal que a implementação de apenas um algoritmo não seja tão eficiente como a sua divisão em sub-algoritmos mais simples. Desta forma, temos um conjunto de sub-algoritmos que podem ser interligados de várias maneiras. De forma hierárquica, isto é, existe um conjunto de sub-algoritmos cujas invocações estão originadas segundo uma árvore. No topo da árvore encontra-se aquele a que chamamos sub-algoritmo principal porque tem como função iniciar a execução do algoritmo de controlo. Numa hierarquia um sub-algoritmo pode invocar outro, e assim sucessivamente, até que todos terminem a sua execução, voltando aos seus invocadores, isto é, percorrendo o caminho inverso. Por outro lado, podemos ter um conjunto de sub-algoritmos que pretendamos executar em paralelo. Neste caso não existe um algoritmo principal mas um conjunto de sub-algoritmos onde dois ou mais são executados em simultâneo. Por fim, o caso que vamos aqui abordar é um conjunto de sub-algoritmos cuja interacção é hieráquica e ao mesmo tempo paralela, ou seja, existe um algoritmo principal onde podem ser invocados vários sub-algoritmos simultaneamente e estes, por sua vez, também podem ter este comportamento. A especificação de todos estes tipos de algoritmos é efectuada recorrendo aos Grafos Hierárquicos e Paralelos – HiParaGraphs (***Hi****erarchical &* ***Para****llel* ***Graphs***) [1, 2]. 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.

A simulação de um algoritmo complexo, hierárquico e paralelo será aqui abordada. Foi efectuada no mesmo ambiente de desenvolvimento da própria especificação, ou seja no GraphBuilder [3, 4], e pode ser utilizada para efeitos de depuração. Durante este processo é possibilitada a intervenção do utilizador em qualquer instante. O modelo de implementação que suporta estes circuitos de controlo encontra-se ainda em desenvolvimento. No entanto, o método utilizado na simulação sugere uma arquitectura.

A especificação da unidade de controlo do controlador CAN – *Controller Area Network* [5, 6], é aqui apresentada como um estudo de caso com o objectivo de evoluir a linguagem aqui apresentada.

Este artigo está dividido em cinco secções. A primeira corresponde à presente introdução. Na segunda secção descrevem-se as características da linguagem de especificação responsáveis pela interacção entre os vários sub-algoritmos. A secção três é dedicada à simulação do algoritmo de controlo com possibilidade de interacção com o utilizador. Um estudo de caso é apresentado na secção quatro. A secção cinco contém as conclusões.

II. A LINGUAGEM DE ESPECIFICAÇÃO

Os HiParaGraphs são uma linguagem de especificação gráfica formal de algoritmos de controlo que se baseiam no formalismo da máquina de estados finitos (FSM –

* Este trabalho foi financiado pela bolsa de doutoramento da Fundação para a Ciência e Tecnologia SFRH/BD/971/2000.

Finite State Machine). Possuem as propriedades sequenciais e simples dos diagramas de transição de estados e combinam a hierarquia, modularidade e comunicação dos Statecharts [7] com o paralelismo das redes de Petri [8] e dos Algoritmos de Controlo Lógicos [9], utilizando uma notação gráfica e intuitiva.

Os vários módulos de um HiParaGraph possuem características diferentes e por isso foram divididos em dois tipos: **funções** e **procedimentos** [1]. As funções são sempre responsáveis por invocações hierárquicas porque o sub-algoritmo que as invoca necessita de esperar pela sua terminação. Quando esta necessidade de espera não acontece, geralmente no caso dos procedimentos, estes são invocados em paralelo, pois neste caso mais do que um sub-algoritmo está activo ao mesmo tempo.

Nas próximas sub-secções são apresentadas, ao nível comportamental, algumas características da especificação de uma unidade de controlo. Para as ilustrar são utilizados grafos dirigidos [10] onde cada nodo representa um sub-algoritmo e cada arco dirigido corresponde a uma possível invocação do sub-algoritmo a que se destina. Os retornos não são aqui representados por uma questão de simplicidade de representação. Dentro de cada nodo existe um identificador do sub-algoritmo, que consiste num número inteiro, para que se possam distinguir as várias invocações, até porque estas podem ocorrer de forma repetida. Os vários nodos são coloridos com cores diferentes de forma a representar possíveis caminhos de invocações e quais os sub-algoritmos activos num dado instante da execução do algoritmo de controlo.

A. A especificação de hierarquia

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. Um algoritmo de controlo quando especificado hierarquicamente possui um sub-algoritmo que invoca outro e este, por sua vez, será executado num nível hierárquico inferior. Quando este terminar a sua execução retorna ao algoritmo que o invocou, ou seja, ao nível imediatamente superior. Na fig. 1 é apresentado um grafo dirigido dividido em três partes pelas linhas a tracejado. Cada parte corresponde a um nível da hierarquia do algoritmo de controlo que este grafo pretende representar. Tal como já foi introduzido anteriormente, cada nodo representa um sub-algoritmo e como podemos ver na fig. 1 os nodos estão coloridos com cores diferentes. Aqui está ilustrado um exemplo que mostra um caminho de execução do algoritmo. A branco estão aqueles sub-algoritmos que não foram invocados. A cinzento claro estão indicados os sub-algoritmos que foram invocados para que seja fácil distinguir na figura o caminho percorrido, 1→3→5. Como se pode verificar, existe apenas um sub-algoritmo colorido em cada nível hierárquico. No último nível hierárquico, a cinzento escuro está representado o sub-algotimo que está a ser executado (5). Quando este terminar o fluxo do algoritmo corresponderá ao caminho inverso (retorno), 5→3→1.

Os HiParaGraphs permitem a especificação da hierarquia quando prevêm a invocação de funções, pois o sub-algoritmo que as invoca precisa de esperar pelo resultado por elas calculado. Só após a determinação deste resultado é que pode prosseguir pois este decidirá o caminho após o nodo que especifica a invocação da função.

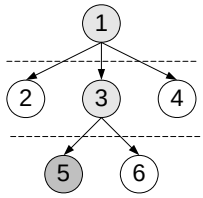

Figura 1. Grafo de representação de hierarquia.

B. A recursividade

A rede de invocações ou a interacção entre os vários sub-algoritmos de um HiParaGraph pode ser tal que além de invocarem sub-algoritmos diferentes podem também invocar-se a eles próprios. Assim, este método permite a especificação de algoritmos de controlo recursivos. Tal como já foi mencionado em [1], a recursividade infinita pode ser detectada mediante a construção de um HiParaGraph equivalente. A correcta execução do algoritmo de controlo depende do modelo de implementação escolhido e dos recursos de hardware disponíveis. Os modelos de implementação que utilizam memórias de *stack*, que embora limitem a recursividade do algoritmo devido ao tamanho máximo da memória, (podendo ocorrer uma situação de *stack overflow*) são aqueles que melhor se adequam a este tipo de algoritmos. O conhecimento prévio da profundidade do *stack* e a construção do HiParaGraph equivalente, representando a árvore de invocações, poderá evitar esta situação.

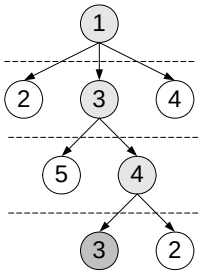

Fig. 2 – Grafo de representação de recursividade numa especificação hierárquica.

Na fig. 2 o grafo representa uma situação de recursividade porque o caminho percorrido pelo algoritmo de controlo ao longo da hierarquia é 1→3→4→3. Como podemos concluir, o sub-algoritmo identificado com o número 3 é invocado pela segunda vez, sendo assim este algoritmo é denominado recursivo pois estamos perante

um ciclo fechado de invocações: 1→3→4→3→4→3→... Este ciclo termina apenas quando se verificar uma condição tal que 1→3→4→3→4→3→…→3→4→2.

De notar que a condição necessária a um algoritmo de controlo para ser recursivo é que este seja hierárquico.

Se considerarmos a recursividade existente na fig. 2, vemos que apenas uma instância do sub-algoritmo 3 está activa num dado instante. No entanto, se este grafo fosse alterado como mostra a fig. 3, temos um sub-algoritmo que se invoca a ele próprio (3). Neste caso além de recursividade existe também reentrância pois o sub-algoritmo 3 é iniciado novamente antes da sua terminação. Para evitar a recursividade infinita, isto é, para que este ciclo de invocações não se feche definitivamente, deve existir sempre uma condição que permita a terminação do sub-algoritmo em questão antes que ele se invoque novamente a ele próprio ou uma invocação a um sub-algoritmo diferente (2) que já irá mudar o fluxo do algoritmo de controlo, tornando possível efectuar o caminho de retorno, quebrando a recursividade.

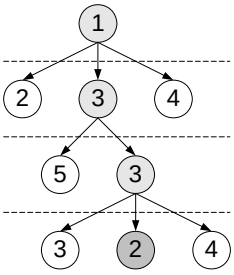

Figura 3. Grafo de representação simultânea de recursividade e reentrância.

C. *A especificação do paralelismo*

A especificação paralela é útil nos casos em que se pretende executar duas ou mais tarefas em simultâneo, como mostra o grafo da fig. 4. Todos os nodos estão coloridos a cinzento escuro porque estão activos, ou seja, a executar simultaneamente.

Figura 4. Grafo de representação de paralelismo.

Os HiParaGraphs suportam o paralelismo especificando a invocação de dois ou mais procedimentos dentro do mesmo nodo de activação. Neste caso, como irá ser detalhado mais adiante, os HiParaGraphs dispõem de indicadores de activação que diferenciam as várias formas de invocar sub-algoritmos dentro do mesmo nodo.

No entanto, existe outra forma de invocação, que será mencionada numa das secções posteriores, em que um sub-algoritmo pode ser invocado paralelamente aos restantes que já estão a ser executados, por intermédio de outro tipo de nodos integrados em sub-algoritmos diferentes.

D. *A especificação simultânea de hierarquia e paralelismo*

Um dos objectivos dos HiParaGraphs é a especificação simultânea da hierarquia e do paralelismo, isto é, uma combinação das situações apresentadas nas sub-secções anteriores. Para ilustrar este caso usamos o exemplo da fig. 5 onde se representa um algoritmo com quatro níveis hierárquicos. A principal diferença deste grafo relativamente aos anteriores é a activação de mais de um nodo por nível hierárquico. Os HiParaGraphs são capazes de invocar mais do que um sub-algoritmo ao mesmo tempo [1] e por isso aumentamos assim o número de nodos coloridos, como vemos na fig. 5. Existem quatro níveis hierárquicos e a execução foi iniciada em 1. Este invocou simultaneamente os sub-algoritmos 3 e 4, e o sub-algoritmo 3 invocou o 2 e o 5. Num dado instante, que mostra a figura, estão activos três sub-algoritmos em dois níveis hierárquicos diferentes.

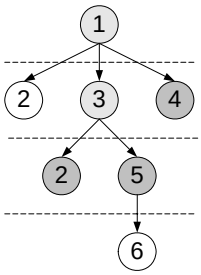

Figura 5. Grafo de representação simultânea de hierarquia e paralelismo.

De notar que na fig. 5 todos os sub-algoritmos activos são diferentes (4, 2 e 5). No entanto, e tal como já foi abordado anteriormente, (ver fig. 3) pode acontecer a ocorrência de uma invocação de um sub-algoritmo que já esteja a ser executado. A esta situação chamamos reentrância e será explicada na próxima sub-secção. Um algoritmo de controlo pode ser recursivo e reentrante simultaneamente quando um dos seus sub-algoritmos se invoca a ele próprio.

E. *A reentrância*

Os algoritmos hierárquicos e paralelos possuem mais do que um sub-algoritmo activo simultaneamente. Numa invocação, quer seja hierárquica quer paralela, o algoritmo de controlo poderá especificar a invocação de um sub-algoritmo que já esteja a ser executado. A este caso chama-se reentrância pois teremos de iniciar/entrar num sub-algoritmo já em execução.

Na fig. 6 mostra-se um grafo com quatro níveis hierárquicos e onde podemos ver que três sub-algoritmos estão activos simultaneamente, pois existem três nodos coloridos a cinzento escuro. No entanto, dois destes possuem a mesma identificação (3). Neste caso terá de ser

criada uma nova instância do sub-algoritmo invocado repetidamente.

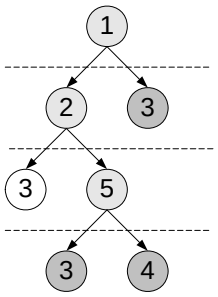

Figura 6. Grafo de representação de reentrância numa especificação hierárquica e paralela.

Os HiParaGraphs prevêm esta situação, que será abordada nas secções posteriores. Durante a simulação de um algoritmo de controlo com estas características, a ferramenta de simulação permite ao utilizador que disponha ou não de reentrância, isto é, esta é uma propriedade apresentada como opção pois o utilizador pode estar ou não limitado ao nível da implementação do respectivo circuito de controlo.

F. Os nodos utilizados em invocações de sub-algoritmos

Considerando um algoritmo de controlo complexo constituído por um conjunto de sub-algoritmos, são necessários nodos com capacidades de invocação e sincronismo que assegurem o correcto funcionamento da unidade de controlo, sendo esta especificada por um algoritmo paralelo e hierárquico. Tal como foi apresentado em [1], um HiParaGraph possui quatro categorias de nodos iniciação, terminação, activação e teste. No entanto, só os dois últimos serão aqui detalhados.

Os sinais de saída da unidade de controlo são os responsáveis pela produção da 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, que é equivalente à introdução de mais do que uma microoperação dentro de um único nodo.

Os procedimentos, por vezes também denominados macrooperações, pertencem ao conjunto $Z=\{z_1, z_2, \ldots, z_N\}$. Os nodos de activação dos HiParaGraphs podem simultaneamente especificar a activação de sinais de saída e a invocação de sub-algoritmos, ou seja, especificar elemento(s) do conjunto Y e elemento(s) do conjunto Z dentro do mesmo nodo. A cada sinal pode ser atribuído um indicador de activação, individualmente, cuja função será detalhada na secção seguinte.

Os nodos do tipo *Sync* são também utilizados para invocar procedimentos, mas neste caso de forma indirecta. 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. 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.

A sequência de activação das microinstruções é determinada por funções de transição, isto é, funções Booleanas α_{ij} (i,j=1,…T) de variáveis Booleanas $X=\{x_1,\ldots x_L\}$. Estas variáveis correspondem às variáveis de entrada da unidade de controlo, também denominadas condições lógicas, e podem ser alteradas pelas microoperações, devido à possibilidade de existência de feedback nas unidades de controlo.

Para testar os valores lógicos destes sinais de entrada são usados os nodos de teste. A invocação de funções Booleanas, elementos do conjunto $F=\{f_1, f_2, \ldots, f_J\}$ é efectuada dentro dentro destes nodos e pode influenciar o fluxo de execução do algoritmo invocador a partir de um valor que é determinado e devolvido pela função, decidindo assim qual o próximo estado activo do circuito. Cada nodo invoca apenas uma função lógica, isto é, especifica apenas um elemento do conjunto F. O valor binário calculado pela função é devolvido utilizando um nodo do tipo *Assign*. Existem apenas dois nodos deste tipo dentro da especificação da função lógica, um especificando o valor lógico 0 e outro especificando o valor lógico 1. No entanto, se quisermos considerar funções f mais complexas, que devolvem outros valores além do 0 e do 1, deverá ser utilizado outro tipo de nodo e dentro do sub-algoritmo que especifica a função deverão existir tantos nodos *Assign* como todos os valores possíveis tomados pela função. Futuramente os HiParaGraphs incluirão esta forma de especificar funções com vários valores de retorno.

G. Os indicadores de activação em sinais de saída

Dentro dos nodos de activação podem ser utilizados indicadores de activação, quer em sinais de saída quer em invocações de procedimentos. Estes indicadores são atribuídos individualmente a cada sinal. No caso dos sinais de saída temos os indicadores SET – “S” e RESET – “R” de forma a ser possível colocar um sinal binário a 1 ou 0, respectivamente, durante mais do que um ciclo de relógio. Caso se pretenda efectuar estas operações apenas durante o ciclo de relógio em que o nodo está activo podem usar-se as opções de atribuição dos valores lógicos 0 ou 1 a cada sinal de saída. Estes indicadores já foram apresentados em [1] e o método utilizado na sua implementação detalhado em [3].

H. *As diferentes formas de invocação de sub-algoritmos*

De forma a distinguir as diferentes formas de invocação de sub-algoritmos, neste caso vamos considerar os procedimentos, os HiParaGraphs dispõem de indicadores de activação associados a cada procedimento e que podem ser os seguintes:

- NONE - nenhum
- WAIT - espera
- RESET - reiniciação
- STOP - terminação

Dentro do mesmo nodo de activação existem sempre sub-conjuntos de Z que estão afectados por estes indicadores, mesmo que sejam sub-conjuntos vazios.

Estes indicadores podem ser divididos em duas categorias. Enquanto NONE – " " e WAIT – "W" estão associados à invocação dos procedimentos, RESET – "R" e STOP – "S" estão indicados à intervenção sobre um dado procedimento que já está a ser executado. De seguida serão detalhadas as funções de cada indicador.

Um procedimento que seja especificado com NONE dentro de um dado sub-algoritmo significa que será iniciado e o sub-algoritmo que o invocou continuará a sua execução. Por outro lado, se for especificado com WAIT o sub-algoritmo invocador ficará à espera da terminação daquele que invocou para poder prosseguir. O indicador RESET fará com que o procedimento especificado, caso este esteja a ser executado, reinicie a sua execução, ou seja, activará imediatamente o seu nodo Begin. Contrariamente a este o indicador STOP terminará o procedimento especificado, caso esteja a ser executado, activando o respectivo nodo End.

Dentro do mesmo nodo de activação podem existir todas as combinações possíveis de sinais com indicadores de activação diferentes. Isto significa que poderá acontecer que o algoritmo invocador poderá especificar, por exemplo, procedimentos com os indicadores NONE e WAIT. Neste caso deve invocar todos eles e ficar apenas à espera da terminação daqueles especificados com WAIT.

III. A SIMULAÇÃO

GraphBuilder é a ferramenta de software que suporta esta linguagem de especificação. A sua constante evolução relativamente às versões anteriores [7, 8] consistiu na integração da hierarquia com o paralelismo ao nível da especificação e a possibilidade de simulação do algoritmo de controlo paralelo e hierárquico utilizando o mesmo ambiente de desenvolvimento.

I. *O algoritmo de simulação*

A simulação de um algoritmo de controlo paralelo e hierárquico é feita passo a passo, calculando ou determinando cada estado seguinte de cada sub-algoritmo que está a ser executado. Num HiParaGraph todos os sub-algoritmos que o constituem podem ser executados simultaneamente, assim como várias instâncias do mesmo sub-algoritmo nos casos de recursividade. A forma como é calculado o estado seguinte de cada um não pode depender da ordem pela qual os algoritmos são percorridos pois este cálculo não deverá ser influenciado pelos restantes.

A existência de um modelo de memória *stack* revelou-se fundamental durante este processo de forma a permitir a simulação de hierarquia e recursividade, pois nestes casos é necessário o conhecimento de alguns estados anteriores do algoritmo de controlo para que possam retornar ao seu invocador e ao estado em que este foi interrompido. No entanto, além da hierarquia e da recursividade temos o paralelismo e para tal o modelo de memória baseou-se num *stack* bidimensional onde a profundidade corresponde ao número de invocações hierárquicas de funções e a largura ao número de invocações paralelas de procedimentos. A fig. 7 pretende ilustrar uma possível forma do *stack* bidimensional durante uma simulação.

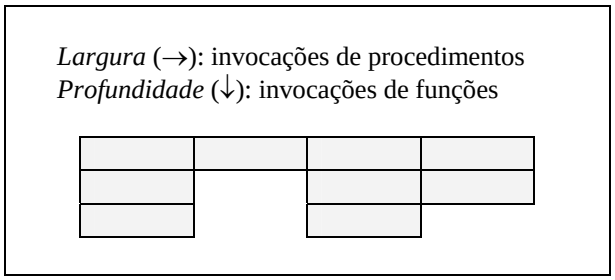

Figura 7. Possível estado do stack bidimensional durante uma simulação.

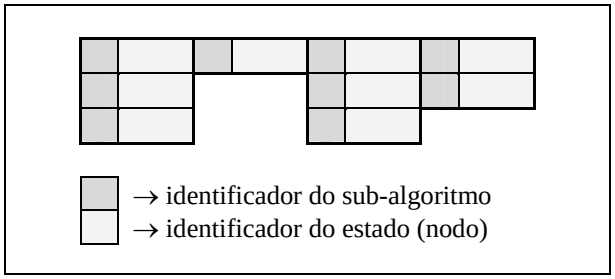

Figura 8. Os diferentes campos de cada registo do stack bidimensional.

Cada quadrícula da memória de *stack* repesentada na fig. 7 representa um registo. No entanto, e tal como faz parte do objectivo dos HiParaGraphs, de forma a flexibilizar o comportamento do algoritmo de controlo, cada registo é dividido em dois campos (ver fig. 8). O primeiro contém o identificador do sub-algoritmo e o segundo o identificador do estado dentro desse sub-algoritmo. Esta é uma forma de codificar os estados do algoritmo de controlo que se pretende que seja mais eficiente do que a codificação binária de todos os estados. Esta eficiência e flexibilidade revela-se ao nível da utilização dos indicadores de activação, como por exemplo RESET e STOP, onde é necessário efectuar a procura de um ou mais sub-algoritmos que já estejam a

executar, ou seja, que já estejam presentes no *stack*. Para identificar um sub-algoritmo nestas condições basta analisar o identificador do sub-algoritmo do último registo de cada coluna do *stack* bidimensional.

Tal como já foi mencionado anteriormente, a ordem pela qual são determinados os estados seguintes de cada sub-algoritmo não deve ser tida em conta porque teoricamente todos os procedimentos são executados ao mesmo tempo e embora possa haver interacção entre eles, o cálculo do novo estado de um dado sub-algoritmo não deve influenciar o cálculo de um novo estado de outro que esteja a ser executado em paralelo. Assim, na simulação efectuada em software foi utilizada uma estrutura de dados adicional, que consiste num conjunto de registos, para guardar o estado anterior da simulação. O seu número de registos é igual ao número de registos da largura do *stack* bidimensional. Por exemplo, no caso da fig. 8 esta estrutura possui os últimos quatro registos do *stack* bidimensional.

Assim, cada passo da simulação pode ser dividido em três etapas:

1. A estrutura de salvaguarda é apagada e é criada uma nova com a mesma largura do *stack* bidimensional, para onde os estados actuais do *stack* são copiados.
2. Todas as colunas desta estrutura são percorridas e aqui são guardados os estados seguintes de cada sub-algoritmo.
3. Os novos valores contidos nesta estrutura são copiados para o *stack* bidimensional.

Em cada ciclo de relógio, ou ciclo de simulação, existe a possibilidade de invocar sub-algoritmos e/ou de estes retornarem ao seu invocador. Neste casos o *stack* bidimensional pode aumentar ou diminuir de tamanho em ambas as dimensões. Para tal, a estrutura de salvaguarda que é utilizada é composta por três campos de identificadores: o do sub-algoritmo, o do estado dentro desse sub-algoritmo e o da operação que deve ser efectuada *na coluna correspondente do stack bidimensional*. As operações a efectuar são pré-definidas e identificadas da seguinte maneira:

- NOOP – Nenhuma operação (utilizada na inicialização).
- PUSH – Adição de um novo registo de estado.
- POP – Remoção do último registo de estado.
- REFRESH – Alteração dos valores existentes do último registo de estado.
- REMOVE – Remoção de uma das colunas do *stack* bidimensional.
- FLUSH – Apaga todo o conteúdo da respectiva coluna do *stack*, embora não o remova.
- FLUSH_PUSH – Apaga todo o conteúdo da respectiva coluna do *stack* e é utilizado apenas na terminação do sub-algoritmo principal. De notar que esta operação corresponde à junção de duas pois as duas operações de FLUSH e PUSH, neste caso, devem ser efectuadas no mesmo ciclo de relógio.

Na fig. 9 é apresentada a simulação de um algoritmo de controlo constituído por cinco sub-algoritmos. Existe um algoritmo principal, isto é, aquele que é inicialmente executado, duas funções e dois procedimentos. De notar que este algoritmo foi construído e escolhido apenas como exemplo para a simulação simultânea da hierarquia e paralelismo.

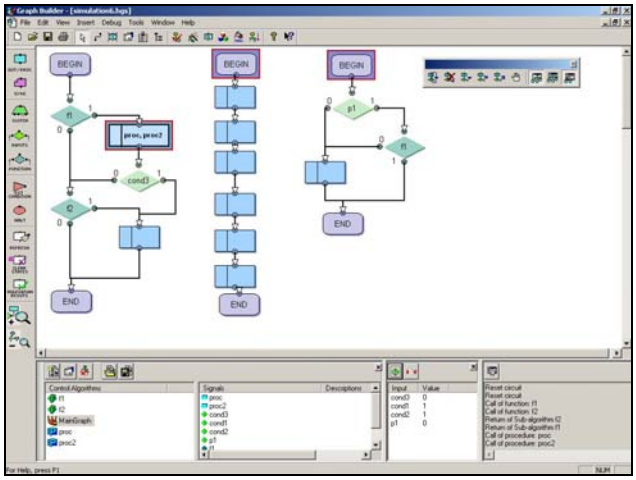

Figura 9. Simulação de um HiParaGraph utilizando o GraphBuilder.

Neste caso são visualizados três sub-algoritmos a executar em paralelo. De notar que os nodos de activação surgem vazios, ou seja, não activam nenhum sinal, porque para este exemplo a activação dos sinais de saída da unidade de controlo é um factor irrelevante. O que neste artigo se pretende evidenciar é apenas a interacção entre os vários sub-algoritmos de um HiParaGraph. À esquerda está representado o sub-algoritmo principal que possui um nodo activo, pois está realçado relativamente aos restantes. Este é um nodo de activação que especifica a invocação (com indicador NONE) de dois procedimentos. Estes, por sua vez, foram iniciados e por isso são visualizados à direita do sub-algoritmo invocador, onde podemos ver os nodos Begin activos de cada um deles. As janelas situadas na parte inferior da aplicação, são utilizadas para visualizar os sinais de entrada e o *log* da simulação, onde são fornecidas mensagens relativas às acções que estão a ser efectuadas pelo algoritmo de controlo. Na fig. 10 estas janelas estão representadas mais pormenorizadamente. À semelhança da visualização das entradas existe também uma outra janela que mostra todos os sinais de saída e os respecivos valores binários. No entanto, esta não está aqui presente porque tal como já se disse anteriormente, neste exemplo não foram utilizados sinais de saída pois eram irrelevantes para a demonstração do processo de simulação. Estas janelas são flutuantes, no entanto podem ser encostadas à parte inferior da janela principal, e possuem uma barra de ferramentas com botões para actuar nos valores binários dos sinais de entrada e fazer o seu reset, no caso da janela de entradas. O botão

existente no topo da janela de *log* serve apenas para apagar o seu conteúdo.

Na fig. 11, à esquerda, está representado o sub-algoritmo principal que no primeiro ciclo de simulação invoca a função f1. O momento da simulação mostrado nesta figura corresponde ao passo da simulação em que se consideram os valores lógicos dos sinais de entrada cond3 = 0 e p1 = 1, depois de f1 ter retornado com o valor lógico 1.

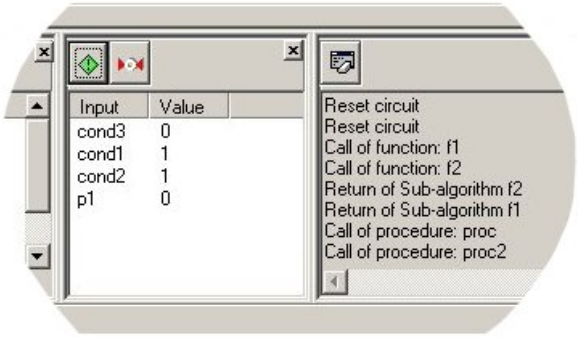

Figura 10. Pormenor da visualização dos valores das entradas e das mensagens de log.

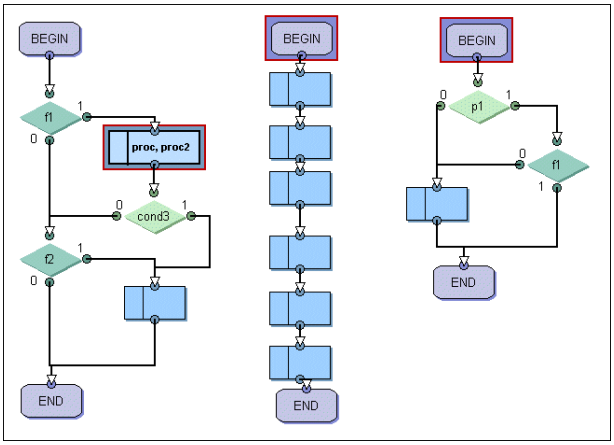

Figura 11. Invocação de dois procedimentos em paralelo.

Neste momento o estado do *stack* aumentou a sua largura de 1 (o sub-algoritmo principal tinha apenas invocado hierarquicamente a função f1) para 3, pois agora temos três sub-algoritmos a executar simultaneamente. Na fig. 12 é mostrado o estado do *stack* com os respectivos identificadores dentro de cada registo.

0	4	1	0	2	0

Figura 12. Estado do stack bidimensional relativamente ao estado da simulação da fig. 11.

Assim, no próximo ciclo de relógio teremos uma transição hierárquica do sub-algoritmo da esquerda porque como cond3 = 0 a função f2 deverá ser invocada. O segundo sub-algoritmo efectua uma simples transição de estado e por fim o terceiro, como p1 = 1, invoca f1, dando-se assim uma nova transição hierárquica. Como se pode ver na fig. 13, os nodos Begin das funções f2 (à esquerda) e f1 (à direita) encontram-se activos.

Assim, da esquerda para a direita temos os seguintes sub-algoritmos a executar: f2, proc e f1. Neste momento temos duas hipóteses a considerar. Se o sinal de entrada cond1, como se mostra na fig. 12 for igual a 0, então o nodo *Assign* de valor 0 ficará activo. Por outro lado, e esta é a situação que vamos considerar, se cond1 = 1 então será necessário invocar novamente a função f2. No entanto, esta já está a ser executada (sub-algoritmo à esquerda) e por isso estamos perante uma situação de reentrância. Nesta aplicação é possível escolher entre executar ou não um algoritmo reentrante, isto é, se a reentrância for uma propriedade aceite pelo utilizador então f2 deverá ser invocada e visualizada à direita. Caso contrário, o sub-algoritmo f1 terá de esperar pela terminação da instância de f2, que já está a ser executada, para poder prosseguir.

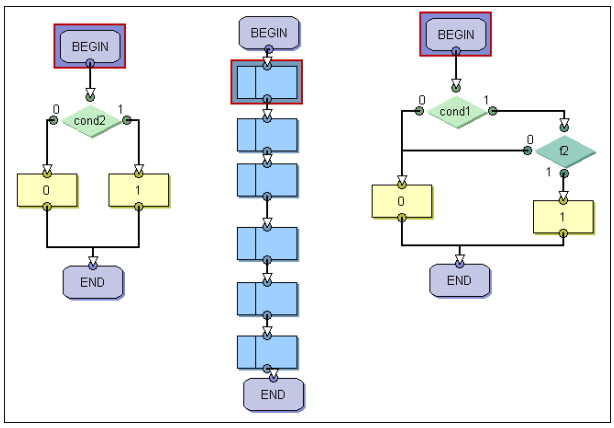

Figura 13. Invocações hierárquicas de sub-algoritmos que executam em paralelo, relativamente à fig. 11.

Na fig. 14 podemos ver pelo estado do *stack* as duas invocações hierárquicas (à esquerda e à direita) que ocorreram. O procedimento representado no centro da fig. 13 continua a sua execução e por isso o registo do *stack* na coluna a que lhe corresponde (ao centro) foi apenas actualizado com o identificador do novo estado activo.

0	3	1	2	2	3
4	0			3	0

Figura 14. Estado do stack bidimensional relativamente ao estado da simulação da fig. 13.

Esta é a situação que é apresentada na fig. 15. f1 permanece com o seu nodo Begin activo, à espera da terminação de f2, enquanto os restantes transitam de estado.

Três ciclos de relógio mais tarde, situação a que corresponde a fig. 16, a função f2 já retornou ao algoritmo invocador e por isso f1 já pode efectuar a invocação de f2 pela qual esperava. Nesta figura, da esquerda para a direita já podemos visualizar os sub-algoritmos: principal, proc e f2, com o respectivo nodo Begin activo.

Dois ciclos depois do estado da simulação representado na fig. 16, a função f2 atinge o seu nodo End ao mesmo tempo que no procedimento proc também o nodo End é activado. O algoritmo principal reinicia a sua execução, activando o respectivo nodo Begin. Esta situação está representada na fig. 17 e o correspondente estado do *stack* na fig. 18. Assim, ambos os sub-algoritmos proc e f2 vão terminar e por isso deixarão de ser visualizados.

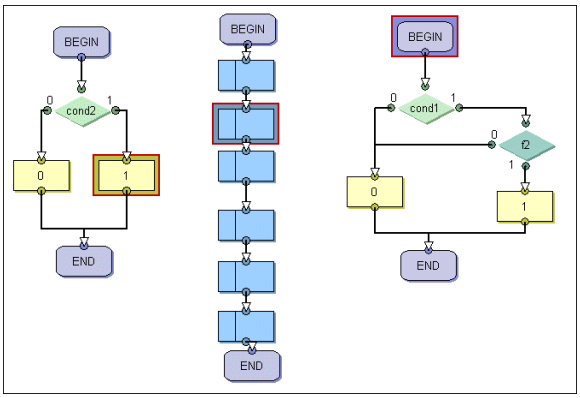

Figura 15. Espera de um sub-algoritmo pela terminação de outro para evitar a recursividade.

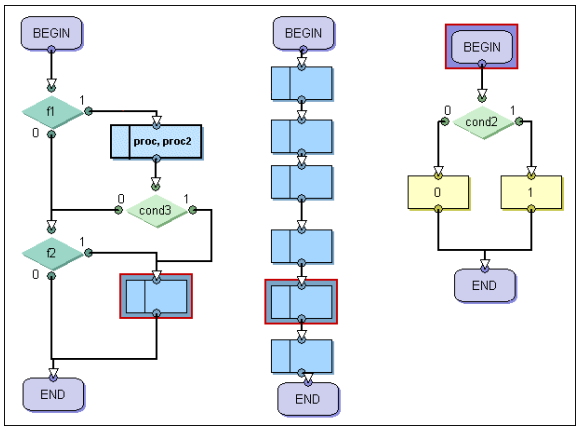

Figura 16. Invocação de f2 por f1 após a terminação da primeira.

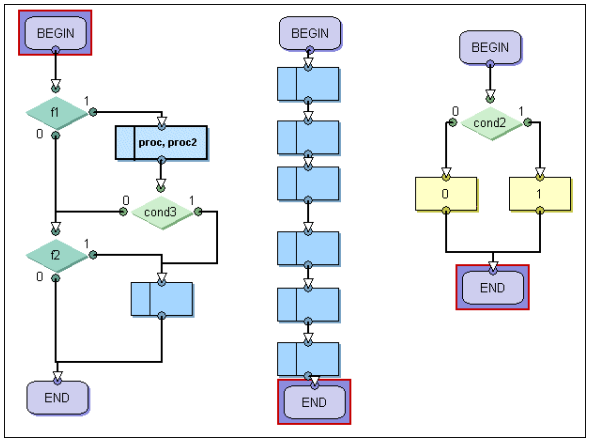

Figura 17. Terminação de proc e f2.

0	0	1	1	2	3
				3	5
				4	1

Figura 18. Estado do stack bidimensional relativamente ao estado da simulação da fig. 17.

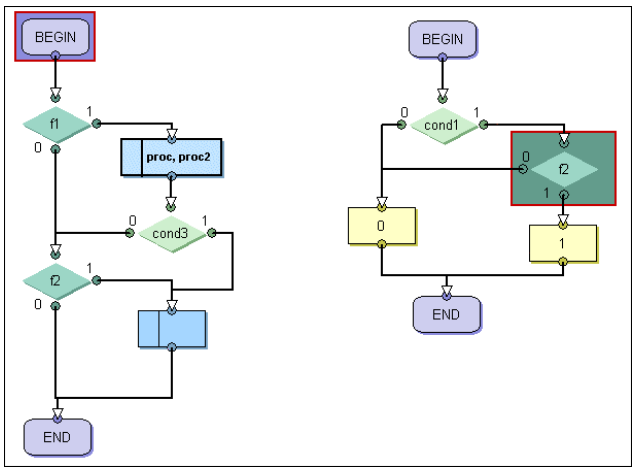

Figura 19. Retorno de f2 ao sub-algoritmo invocador f1.

Na situação apresentada na fig. 19 temos à esquerda o algoritmo principal e à direita temos f1. De notar que o primeiro permanece com o estado Begin activo porque o nodo que a este está ligado especifica a invocação da função f1 e que neste momento está a ser executada. Desta forma, evitando a reentrância, este sub-algoritmo terá de esperar que o da direita termine. Neste estado da simulação podemos verificar pela fig. 19 que a largura do *stack* bidimensional diminuiu, pois temos agora apenas dois sub-algoritmos a executar e um deles no segundo nível de hierarquia.

0	0	2	3
		3	5

Figura 20. Estado do stack bidimensional relativamente ao estado da simulação da fig. 19.

No final desta simulação podemos verificar que as dimensões máximas atingidas pela memória de *stack* foram 3x3, o que significa que obtivemos no máximo 3 invocações hierárquicas e 3 sub-algoritmos a executar em paralelo. Caso fosse permitida a reentrância as dimensões máximas obtidas certamente teriam sido superiores. De qualquer forma, a ferramenta GraphBuilder permite-nos ter conhecimento acerca do tamanho real do *stack* que devemos utilizar, caso estejamos na presença de um algoritmo de controlo simples, ou efectuar uma estimativa das dimensões necessárias ao correcto funcionamento do algoritmo na unidade de controlo pretendida.

J. A interacção com o utilizador

Durante a simulação o utilizador dispõe de uma barra de ferramentas, que no caso da fig. 9 pode ver-se em estado flutuante sobre o fundo da janela principal, onde se encontram todas as facilidades permitidas neste processo. É aqui que se inicia e termina a simulação e se simula a passagem de um ou mais ciclos de relógio, no caso da simulação passo a passo ou automática, respectivamente. No caso da simulação automática existe também a possibilidade de introdução de pontos de paragem (*breakpoints*) sobre os nodos de cada sub-algoritmo ou a pausa/interrupção do processo mediante a actuação de um dos botões da barra de ferramentas.

Todos os sinais de entrada e de saída da unidade de controlo considerada podem ser visualizados em janelas flutuantes, assim como o *log* da simulação. Inicialmente todos os sinais de entrada são colocados a zero mas durante a simulação o utilizador pode ir actuando sobre estes sinais binários entre cada ciclo de relógio, ou seja, antes de cada passo da simulação, para verificar o comportamento do algoritmo de controlo mediante as combinações binárias de entrada que desejar.

IV. UMA APLICAÇÃO PRÁTICA

O objectivo dos HiParaGraphs é disponibilizar um método fácil, rápido e eficaz de especificar unidades de controlo digitais. O CAN – *Controller Area Network*, é um protocolo de comunicação série que suporta a implementação de aplicações distribuídas e de tempo real e que é bastante utilizado nos sistemas electrónicos de automóveis. Estamos a falar de um barramento, num sistema *multi-master*, onde todos os nodos estão pendurados e são capazes de enviar e receber informação do barramento. Uma das grandes vantagens do protocolo CAN é a sua grande fiabilidade na transmissão. A quantidade de erros enviada por um nodo vai sendo medida e registada. Se o número de erros produzidos for grande o nodo será retirado do barramento. Este procedimento é possível devido à inexistência de endereços nos nodos e por isso estes podem ser acrescentados e retirados do barramento sem que haja interferências no sistema. Os endereços são substituídos por um sistema de envio de mensagens de diferentes prioridades. Cada mensagem CAN pode transmitir 0 a 8 bytes de informação (dados) mas este comprimento pode ser estendido utilizando segmentação, sendo a taxa máxima de transmissão especificada a 1Mbit/s.

Vários nodos podem requisitar simultaneamente o barramento para o envio de mensagens. Um transmissor envia uma mensagem a todos os nodos CAN (*broadcasting*) e cada nodo decide, baseando-se no identificador recebido se deve ou não processar aquela mensagem. Este identificador já determina a prioridade da mensagem no acesso ao barramento.

Os vários tipos de mensagens são:

- DATA FRAME – de dados, enviada por um transmissor a todos os nodos receptores.
- REMOTE FRAME – transmitida por um nodo para requisitar a transmissão de uma mensagem de dados com o mesmo identificador.
- ERROR FRAME – é transmitida por um nodo quando este detecta um erro no barramento.
- OVERLOAD FRAME – é utilizada para introduzir um atraso entre as mensagens DATA FRAME e REMOTE FRAME.

As mensagens DATA e REMOTE são separadas das restantes por um INTERFRAME SPACE que consiste numa mensagem de pelo menos três bits consecutivos a 1.

Uma mensagem do tipo DATA é composta por sete campos diferentes:

1. *Start of Frame* – composto por um bit igual 0 que delimita o início da mensagem.
2. *Arbitration* – campo de arbitragem, composto por um identificador de 11 bits e um bit RTR.
3. *Control* – composto por 6 bits, dois reservados e quatro para indicar o comprimento dos dados.
4. *Data* – campo que contém todos os bits de dados, desde 0 a 8 bytes.
5. *CRC* – contém uma sequência CRC (Cyclic Redundancy Code) e um bit delimitador.
6. *ACK* – campo de dois bits.
7. *End of Frame* – campo de 7 bits iguais a 1.

Uma mensagem do tipo REMOTE é equivalente à anterior pois possui todos estes campos à excepção do *Data*.

O protocolo CAN possui uma unidade de controlo para transferência de mensagens que os HiParaGraphs são capazes de descrever. Esta unidade possui características hierárquicas e paralelas e utiliza as condições de saída dos HiParaGraphs para especificar o comportamento do sistema em situações de erro. A descrição comportamental da unidade de controlo foi dividida em vários módulos. Existem três sub-algoritmos que são executados em paralelo desde o início e que estão representados nas figuras 21, 22 e 23. O primeiro dedica-se ao controlo da transferência das mensagens no barramento, o segundo ao controlo sobre a ocorrência de erros e o terceiro ao controlo do papel de emissor ou receptor de cada nodo CAN existente no barramento. Os dois últimos correspondem a duas máquinas de estado simples enquanto o primeiro é composto por quatro sub-algoritmos e possui hierarquia devido aos diferentes tipos de campos existentes numa mensagem, como foi apresentado acima.

Na fig. 21 estão visíveis todos os campos de uma mensagem, quer do tipo DATA quer REMOTE. O controlo da recepção/transmissão de cada bit daqueles campos constituídos por mais do que um bit é encapsulado noutro sub-algoritmo de controlo – temos portanto hierarquia. Entramos no campo de arbitragem quando é invocado hierarquicamente o procedimento DR_ARB (fig. 24a). Seguidamente passamos ao campo de controlo, invocando da mesma forma o procedimento DR_CTRL (fig. 24b). O

bit *rtr* dentro do nodo de teste que se segue determina se estamos perante uma mensagem do tipo DATA ou do tipo REMOTE. Os campos CRC e ACK estão representados pelos procedimentos DR_CRC (fig. 25a) e DR_ACK (fig. 25b) e tal como os anteriores também são invocados hierarquicamente. Todos os procedimentos são invocados com o indicador W, pois temos de esperar a sua terminação para passar ao campo seguinte da mensagem. Na parte inferior do sub-algoritmo está especificada a condição de saída que indica que se o sinal *error* estiver activo devemos sair deste algoritmo e invocar o FRM_ERROR (ver fig. 26a).

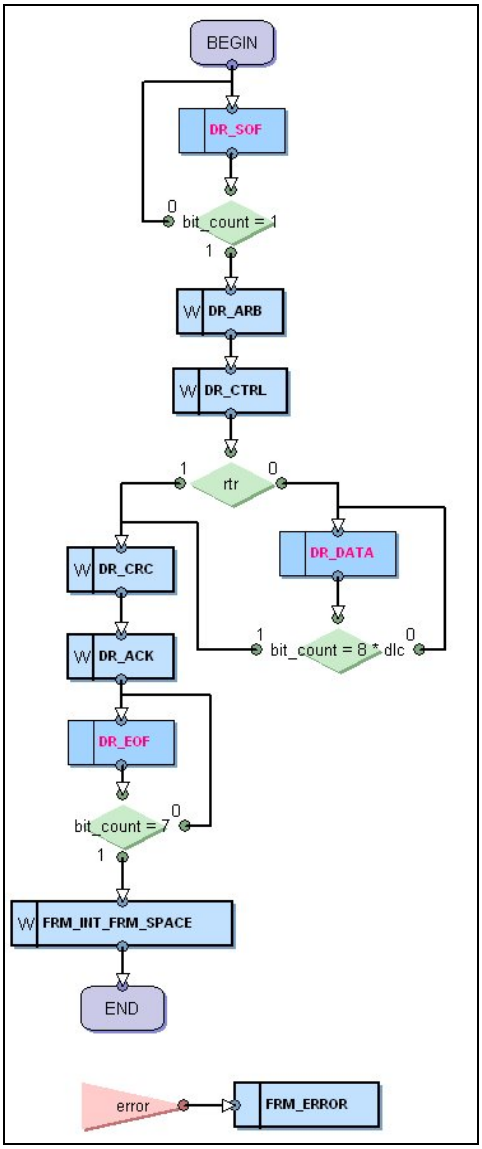

Figura 21. Sub-algoritmo pertencente à especificação da unidade de controlo do CAN responsável pelo controlo das transferências de mensagens.

Em qualquer instante um nodo CAN pode estar num dos três estados representados na fig. 22:

1. FLT_ERROR_ACTIVE – significa que o nodo pode enviar para o barramento mensagens de erro do tipo ACTIVE_ERROR_FLAG. Tal como podemos ver na figura pelo sinal de saída *error_flag* = 0.
2. FLT_ERROR_PASSIVE – significa que o nodo pode enviar para o barramento mensagens do tipo PASSIVE_ERROR_FLAG, pela actualização do sinal *error_flag* = 1.
3. FLT_BUS_OFF – neste estado o nodo não pode enviar mensagens para o barramento.

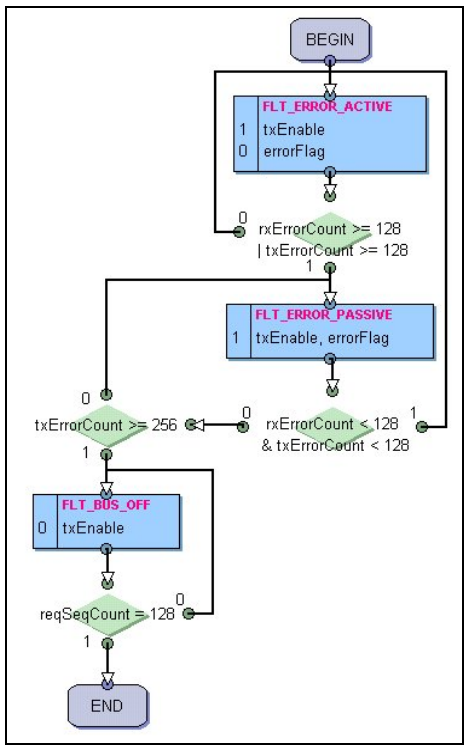

Figura 22. Sub-algoritmo pertencente à especificação da unidade de controlo do CAN responsável pelo controlo sobre a ocorrência de erros no barramento.

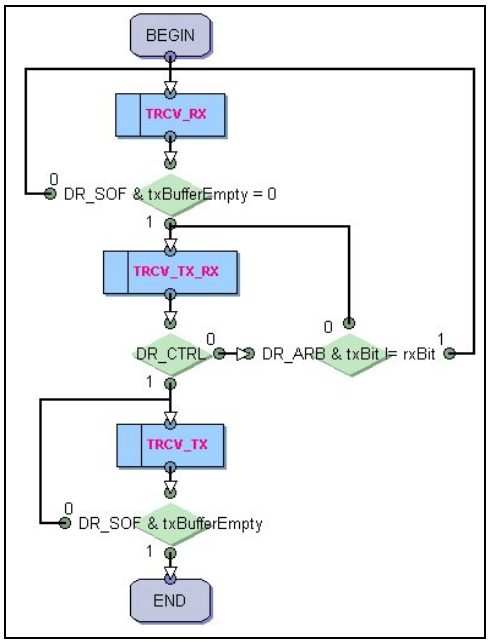

Figura 23. Sub-algoritmo pertencente à especificação da unidade de controlo do CAN responsável pelo controlo sobre a emissão e recepção de mensagens de cada nodo.

Na fig. 23 visualizamos também três estados denominados:

1. TRCV_RX – que significa que o nodo CAN está apenas em modo de recepção,
2. TRCV_TX_RX – significa que o nodo é capaz de transmitir e receber mensagens do barramento,
3. TRCV_TX – significa que o nodo está apenas em modo de transmissão.

O sub-algoritmo representado na fig. 23 explica como um nodo pode transitar entre estes três estados, pela verificação das condições especificadas dentro dos nodos de teste. Os nodos de teste com as condições escritas a letra maiúscula, como por exemplo DR_CTRL e DR_ARB pretendem testar se os respectivos sub-algoritmos com este nome estão activos.

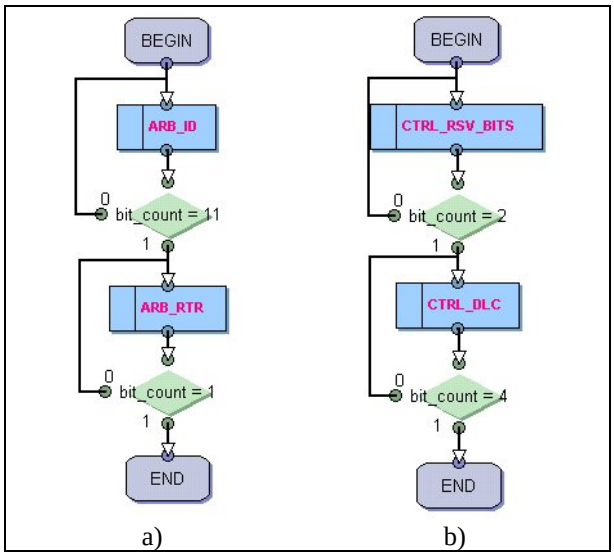

Figura 24. Sub-algoritmos pertencentes à especificação da unidade de controlo do CAN que especificam os procedimentos a) DR_ARB e b) DR_CRTL.

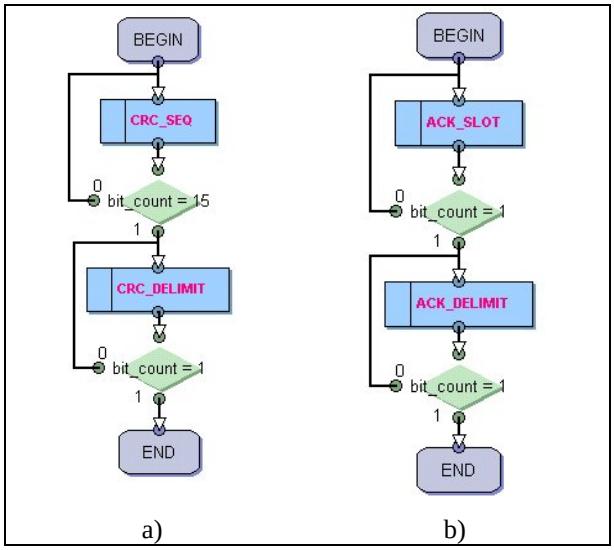

Figura 25. Sub-algoritmos pertencentes à especificação da unidade de controlo do CAN que especificam os procedimentos a) DR_CRC e b) DR_ACK.

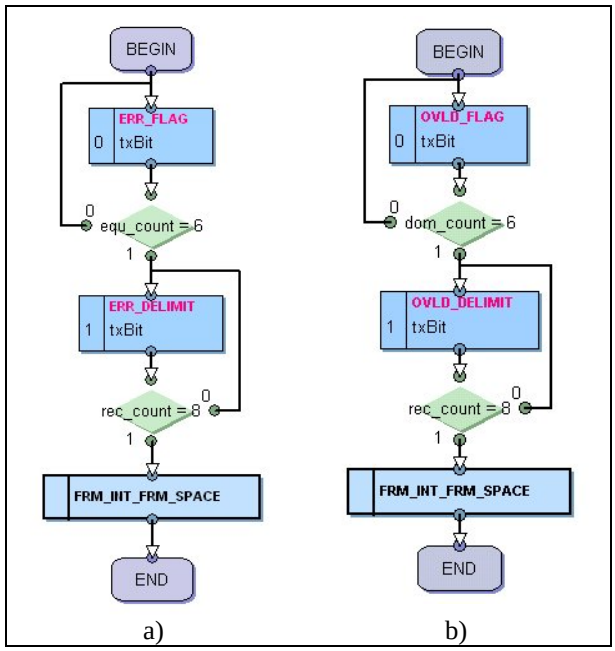

Figura 26. Sub-algoritmos pertencentes à especificação da unidade de controlo do CAN que especificam os procedimentos a) FRM_ERROR e b) FR_OVERLOAD.

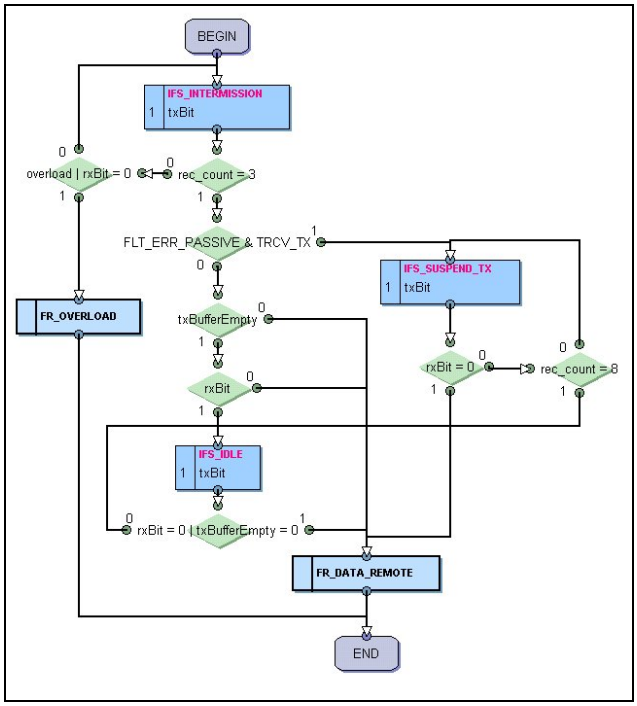

Figura 27. Sub-algoritmo pertencente à especificação da unidade de controlo do CAN que especifica o procedimento FRM_INT_FRM_SPACE.

Esta unidade de controlo foi utilizada como um estudo de caso de forma a avaliar as capacidades de especificação dos HiParaGraphs e eliminar as suas deficiências, acrescentando-lhes alguma funcionalidade.

A visualização do nome do estado associado a um nodo de activação foi uma das facilidades visuais (opcionais) introduzidas na aplicação pois revelou-se útil na compreensão da descrição comportamental dos módulos apresentados, como por exemplo os das figuras 22, 23, 24 e 25, pois como não activam nenhum sinal de saída ficariam vazios e, consequentemente, sem nenhum significado visível ao projectista. O nome do estado aparece sempre escrito no topo do nodo de activação, por cima de todos os sinais, com cor diferente e em maiúsculas. No caso da fig. 25a temos visualizados os estados CRC_SEQ e CRC_DELIMIT.

A introdução dos nodos com condições de saída nos HiParaGraphs deveu-se ao estudo deste exemplo de especificação. Como se pode ver na fig. 21, existe uma condição de saída na parte inferior do sub-algoritmo, que especifica o sinal *error*. Isto significa que se em qualquer estado do circuito esta condição se verificar, ou seja, se *error* estiver activo, o sub-algoritmo FRM_ERROR será imediatamente invocado. Esta forma de saída de um sub-algoritmo dependente de uma condição evita o teste desta em todos os estados possíveis do sub-algoritmo, o que complicaria demasiado o aspecto gráfico, a conversão para código VHDL e a respectiva implementação do circuito.

Os sinais de entrada que estão especificados dentro dos nodos condicionais mostram expressões, como por exemplo, *rtr*. A avaliação do valor lógico de um sinal binário dentro destes nodos é efectuada determinando se este tem o valor verdadeiro (1). No caso das expressões *bit_count = 8 * dlc*, *bit_count = 1* e *bit_count = 7* da fig. 21, a avaliação é efectuada de outra forma. Embora estas sejam mais explícitas para quem especifica a unidade de controlo, terão de ser substituídas por outros sinais binários aquando da conversão do algoritmo em código VHDL. Esta forma livre de especificar condições dentro dos nodos é útil e intuitiva ao projectista, pois demonstra uma interligação entre a unidade de controlo e a unidade de execução de um sistema, mas este terá de analisar todos os sinais de entrada (que é uma tarefa fácil utilizando o GraphBuilder) para posteriormente atribuir outros nomes antes da conversão para VHDL, pois poderá haver incompatibilidade entre os nomes das condições do tipo *bit_count = 8 * dlc*, e a sintaxe do VHDL.

A existência evidente de uma interligação entre a unidade de controlo e a unidade de execução é a especificação de sinais de entrada do tipo *bit_count = x*. Assumem-se aqui que existem vários contadores de bits, que vão sendo recebidos do barramento e contados. Quando for atingido um determinado valor que é especificado dentro do nodo de teste então a condição verifica-se. Devem existir quatro contadores diferentes de bits recebidos do barramento, que contam:

1. *bit_count* – todos os bits independentemente do seu valor lógico;
2. *rec_count* – todos os bits com o valor lógico 1;
3. *dom_count* – todos os bits com o valor lógico 0;
4. *equ_count* – todos os bits consecutivos iguais.

V. CONCLUSÕES

A especificação modular de algoritmos de controlo através de HiParaGraphs suporta explicitamente a hierarquia e o paralelismo. 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. A ferramenta de software GraphBuilder implementa todo o suporte para esta linguagem, desde a construção gráfica até à simulação e conversão em código VHDL.

A simulação dos algoritmos de controlo especificados por HiParaGraphs prevê a existência de hierarquia, paralelismo e ambos em simultâneo, recursividade e reentrância, podendo a última ser utilizada como opção devido a possíveis limitações de implementação por parte do projectista. A simulação, tal como foi apresentada neste artigo pode ser utilizada para efeitos de depuração do circuito de forma a serem detectados erros no comportamento deste antes de ser convertido em VHDL e posteriormente sintetizado, facilitando assim, mas não evitando, a depuração e teste do circuito após a sua implementação.

Como estudo de caso para evolução desta linguagem, assim como da ferramenta GraphBuilder, foi utilizada a especificação da unidade de controlo do protocolo CAN. Este estudo resultou na introdução de um certo grau de liberdade na especificação:

- do teste de sinais de entrada, que pode ser unicamente um sinal binário ou mais do que um afectados por uma função Booleana (*and, or,* ou *xor*). Por outro lado, se as condições especificadas tiverem o mesmo nome de um sub-algoritmo isto significa que estamos a testar se este sub-algoritmo se encontra activo naquele instante.
- na visualização dos estados no topo dos nodos de activação e
- na introdução de um novo tipo de nodo com condições de saída de sub-algoritmos.

Assim, podemos concluir que os HiParaGraphs são uma linguagem gráfica cada vez mais poderosa e com possibilidades de evolução na construção de unidades de controlo digitais. O ambiente GraphBuilder é uma ferramenta de trabalho que tem os HiParaGraphs como suporte, disponibilizando-lhes todas as funcionalidades e interface até agora apresentadas. A unidade de controlo deste protocolo pode ser construída e simulada no ambiente GraphBuilder e facilmente convertida em código VHDL comportamental sintetizável.

REFERÊNCIAS

[1] Andreia Melo, Valery Sklyarov, António Ferrari, *HiParaGraphs, uma Linguagem de Especificação de Algoritmos de Controlo Paralelos e Hierárquicos*, Electrónica e Telecomunicações, Vol. 3, Nº 3, pp. 214-221, Janeiro de 2001.

[2] Andreia Melo, Valery Sklyarov, António Ferrari, *Specification and Implementation of Hierarchical and Parallel Control Algorithms Using HiParaGraphs*, aceite para publicação nas actas de CAD DD'2001 - 4th International Conference on "Computer-Aided Design of Discrete Devices", Minsk, Bielorrúsia, Novembro 2001.

[3] 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.

[4] A. Melo, V. Sklyarov, *Ambiente Integrado para Especificação, Projecto e Verificação de Unidades de Controlo em FPGAs*, Electrónica e Telecomunicações, vol. 2, n.º 4, Janeiro 1999, pp. 477-485.

[5] Road vehicles – *Interchange of digital information – Part 1: Controller area network data link layer and medium access control*; ISO 11898.

[6] Road vehicles – *Controller area network (CAN) – Part 2: High-speed medium access unit*; ISO 11898.

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

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

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

[10] Diestel R., *Graph Theory*, Springer, New York, 2ª ed., 2000.