Dissertação de Mestrado

Redes de Petri e VHDL na Especificação de Controladores Paralelos

João Miguel Lobo Fernandes

Departamento de Informática

Universidade do Minho

Braga, Julho 1994

Resumo

A unidade de controlo da maior parte dos sistemas digitais é normalmente estruturada como uma máquina de estados síncrona genérica (GSSM). Complexas máquinas deste tipo estão presentes em muitos projectos VLSI e são implementadas usando dispositivos de lógica programável. Actualmente, estão disponíveis, na maioria das plataformas CAD, linguagens de especificação, embora estas não disponibilizem directamente formas de modelar actividades concorrentes e cooperativas.

As Redes de Petri (de aqui em diante, simplesmente, RdP) são uma ferramenta gráfica muito poderosa para especificar e modelar o comportamento de controladores paralelos. Existem várias técnicas para análise das RdP que permitem validar formalmente as propriedades mais importantes do sistema modelado: vivacidade, segurança, inexistência de conflitos e determinismo. Inúmeros tipos de RdP foram propostos e usados para especificar ou modelar sistemas, quer pela imposição de restrições ao modelo básico, quer pela adição de características adicionais. Uma revisão dos tipos mais relevantes sugeriu que as GSSM são mais facilmente especificadas e implementadas por RdP seguras com transições guardadas e disparos síncronos. Adicionalmente, são também admitidos arcos inibidores e habilitadores.

VHDL é uma linguagem textual bastante potente, possibilitando a especificação, simulação e concepção de um sistema digital. As suas características são sumarizadas neste trabalho. É definido um subconjunto da linguagem, com a finalidade de facilitar a simulação e a síntese de controladores paralelos baseados em RdP. Apresentam-se formas alternativas de representar diagramas ASM e RdP, usando VHDL.

As plataformas de CAD electrónico actualmente disponíveis começam a aceitar especificações baseadas em RdP, mas ainda não exploram totalmente os benefícios do paradigma das RdP para análise e da compilação para VHDL para posterior utilização em ferramentas de simulação e síntese. Foi desenvolvida uma aplicação computacional para obviar estas limitações, com o cuidado de gerar código VHDL aceite por algumas plataformas de CAD. Esta aplicação aceita como entrada uma especificação textual baseada nas RdP, valida as propriedades do sistema modelado e converte — compila — a especificação para um dado subconjunto VHDL. O estudo de exemplos comprova a viabilidade da abordagem seguida, tendo sido testado o código VHDL nas ferramentas de domínio público ALLIANCE.

Palavras chave: Redes de Petri, Controladores Paralelos, VHDL, CAD Electrónico, Especificação de Sistemas Digitais.

Abstract

The control path of most digital systems is often structured as a generic synchronous state machine (GSSM). Complex GSSM are present in most VLSI designs or implemented using Programmable Logic Devices. Specification languages are currently available in most CAD packages, but they often lack appropriate support to express concurrent and cooperating activities.

Petri Nets (for short PNs) provide a powerful graphics tool to specify and model the behaviour of parallel controllers. Several techniques for PN analysis are available which allow a formal validation of the basic properties of a modelled system: liveness, safety, conflict-freedom and determinism. Several types of PNs have been used to specify and/or model systems, either by imposing restrictions to a basic model, or by adding extensions to it. A review of the most relevant ones suggests that a GSSM is best specified and modelled by a safe PN with guarded transitions and synchronous triggers; this PN should also support enabling and inhibitor arcs.

VHDL is a powerful text-based language to specify, simulate and design a digital system, and its basic features are here summarized. A subset of VHDL is defined, aiming the simulation and synthesis of a PN-based parallel controller specification. Alternative ways of representing ASM flowcharts and PNs using VHDL are presented.

Current electronic CAD tools start to accept PNs as a data input, but they do not fully exploit yet the benefits from both the PN paradigm for analysis and the compilation to VHDL for later simulation and synthesis. A new software environment was developed to overcome these limitations, which was designed to feed any CAD package that accepts certain VHDL subsets. This environment accepts a PN text-based specification, and then it validates its basic properties and converts — compiles — the specification into a VHDL subset notation. Some examples were run to test the environment, and later simulated and synthesized with ALLIANCE, a VHDL public domain package.

Keywords: Petri Nets, Parallel Controllers, VHDL, Electronic CAD, Digital Systems Specification.

Aos meus pais,

"Honos alit artes"

Cícero – Tusculanas, I, 2, 4

Agradecimentos

Agradeço ao Prof. Alberto Proença a incondicional disponibilidade que sempre demonstrou e, ainda, todos os conselhos e sugestões que me deu, os quais valorizaram enormemente este projecto.

Ao meu colega João Alexandre Saraiva, tenho de deixar expressa uma palavra de profundo agradecimento por todo o apoio técnico que incondicionalmente me prestou na área das gramáticas de atributos, bem como no funcionamento do produto de desenvolvimento. Sem a sua prestimosa colaboração, o compilador não teria sido construído tão rápida e eficazmente.

Agradeço ao Prof. Marian Adamski a enorme quantidade de artigos que me facultou, as conversas frutíferas que me proporcionou e os comentários que teceu ao trabalho realizado.

Deixo uma palavra de apreço às pessoas do Departamento de Informática e aos meus colegas de mestrado, que directa ou indirectamente, contribuiram para a realização deste trabalho. Uma referência especial tem de ser dada aos meus colegas de trabalho, Luís Paulo Santos, João Luís Sobral, António Esteves e António Tavares, pelo agradável clima de trabalho que me proporcionaram e pela amizade que sempre demonstraram.

Ao Prof. Pedro Henriques, agradeço a disponibilidade que demonstrou, no sentido de verificar a adequação dos métodos de compilação por mim utilizados.

Aos meus alunos António Manuel Dias e Paulo Jorge Almeida agradeço todo o código que produziram, que permitiu, em tempo útil, incluir um módulo de análise na aplicação desenvolvida.

Ao meu colega Bruno Dias agradeço toda a assistência e todo o apoio que me prestou no LaTeX.

Devo ainda agradecer às seguintes pessoas: Shishir Agarwal, Marco Ajmone Marsan, Carl Chang, Frank DiCesare, Javier Esparza, Luca Ferrarini, Greg Findlow, Paulo Flores, Geoffrey Frank, Luís Gomes, Tomasz Kozlowski, Tadao Murata, Zebo Peng, Franz Rammig, Alberto Sangiovanni-Vicentelli, Robert Saphiro, Dave Stotts, Ichiro Suzuki, Robert Valette, Eugenio Villar, Albert Wang, James Watson. Todas elas tiveram a gentileza de me enviar os artigos, relatórios e teses que lhes pedi, sem os quais o resultado final deste projecto teria sido, sem qualquer dúvida, mais pobre.

À Raquel agradeço todo o apoio, carinho e amor, sem os quais este projecto de mestrado não teria sido possível.

Finalmente, não posso esquecer os meus pais, a quem dedico este trabalho, e a minha irmã, pela forma carinhosa como sempre me apoiaram, durante o período que durou o projecto.

Índice

Capítulo 1

Introdução

"Dois mundos que se não podem confundir e que, vivendo à parte, com fins diferentes, caminham paralelamente na civilização, um com o título egrégio de bacharel, outro com o nome emblemático de futrica."

Eça de Queiróz in "O Conde de Abranhos"

1.1 Controladores Paralelos e Redes de Petri

Os controladores são um tipo de sistemas digitais, usados no controlo de sistemas de tipo discreto. Habitualmente, os sistemas digitais, se forem relativamente complexos, são divididos em duas partes distintas, mas cooperantes: uma estrutura de controlo e uma parte para dados (Control + Data Path) [Pro91]. Essa perspectiva é apresentada na figura 1.1. Estas duas partes estão relacionadas pelos sinais de controlo provenientes da estrutura de controlo para a parte de dados e pelos sinais dirigidos em sentido inverso.

Analogamente, os programas desenvolvidos numa qualquer linguagem de programação (C, Pascal, Assembly) podem ser vistos como sendo constituídos por algoritmos e estruturas de dados[1]. Aliás, estes factos estão intimamente ligados, pois, em última análise, os programas especificados numa qualquer linguagem de alto nível, após uma série de transformações, são executados numa máquina real.

A dimensão dos sistemas digitais tem vindo, ao longo dos tempos, a aumentar consideravelmente. Além disso, esses sistemas incluem, cada vez com maior frequência, acções simultâneas e computação paralela. O primeiro passo no projecto de qualquer sistema digital é o da sua especificação. O desenho de um sistema digital é iniciado, usualmente, com uma descrição de alto nível (leia-se descrição comportamental, algorítmica e informal) do comportamento desejado [KBK+90]. Numa fase posterior, faz-se a "tradução" dessa descrição para uma representação mais formal. No caso particular dos controladores programáveis (PLCs), Gomes et al. [GSG92] referem que existe uma série de métodos que os

[1] O título de um livro [Wir76] bastante famoso em Ciências da Computação ("*Programs = Algorithms + Data Structures*"), da autoria de N. Wirth, reflecte exactamente essa ideia.

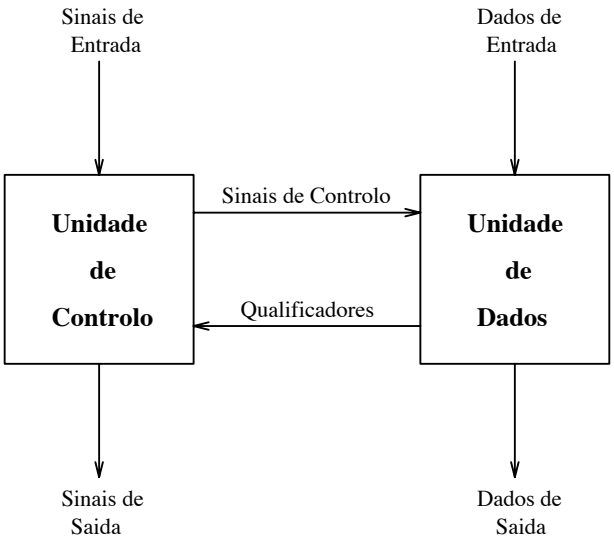


Figura 1.1: Estrutura Genérica de um Sistema Digital.

permite especificar como "*ladder diagrams*", linguagens lógicas e GRAFCET[2]. Algumas das linguagens de programação de PLCs baseiam-se em formalismos de carácter gráfico, adaptados para a modelação de sistemas sequenciais ou paralelos [Sil89]. Os diagramas de estado e os diagramas ASM são formalismos bastante utilizados para sistemas sequenciais, enquanto que, para sistemas paralelos, o sistema GRAFCET [BBFM79] é baseado nas RdP.

Os diagramas ASM são suficientes para a especificação de sistemas sequenciais. Mais concretamente, podem-se especificar controladores em que apenas um estado está activo, em cada momento — controladores sequenciais. Alguns fabricantes permitem que se implemente de um modo quase directo os diagramas ASM. Por exemplo, a linguagem PALASM-4 [AMD91] contém alguns comandos vocacionados para a escrita directa de diagramas ASM. Outras linguagens que também o permitem são: ABEL, CUPL e OrCAD-PLD [Pea91]. Considere-se o diagrama ASM apresentado na figura 1.2. Este diagrama representa uma máquina do tipo Mealy, o que significa que as saídas são dadas por uma função das entradas, quando a máquina se encontra estabilizada num dado estado. O respectivo código PALASM é apresentado no texto 1.1, assumindo-se a implementação do sistema num dispositivo PAL16V8.

A implementação final deste diagrama ASM poderia ser, caso se use a aplicação PALASM, um dispositivo PAL ou MACH. É precisamente a existência de ferramentas de apoio à especificação, conjuntamente com a generalização e uso fácil dos dispositivos de lógica programável, que tornou os diagramas ASM tão populares na especificação de sistemas sequenciais.

É possível, também, especificar um controlador paralelo (controlador em que podem estar activos vários estados simultaneamente), usando as técnicas inerentes aos diagramas ASM [PB91]. Para tal, há que dividir o controlador inicial num conjunto de controladores locais. Estes controladores locais serão sequenciais e a sua interligação faz-se através de sinais comuns ou bits de semáforo.

Qualquer uma das duas técnicas de interligação atrás referida é contudo de difícil aplicação, além de que a divisão resultante pode corresponder a uma implementação ineficiente. Esta abordagem traz ainda outros problemas: é também difícil verificar a existência de erros de

[2]Acrónimo da expressão em língua francesa: *Graph de Commande Etape-Transition*.

```
TITLE Exemplo Diagrama ASM
PATTERN A
REVISION 1.0
AUTHOR Joao Miguel Fernandes
COMPANY Departamento Informatica - U.M.
DATE Marco-94
;-----------------------------
CHIP _ASM PAL16V8
;-----------------------------
PIN  1 CLOCK               ; RELOGIO
PIN  2 IN1                 ; ENTRADA
PIN  3 IN2                 ; ENTRADA
PIN 10 GND
PIN 11 ENABLE              ; HABILITAR
PIN 12 OUT1  COMBINATORIAL ; SAIDA
PIN 13 OUT2  COMBINATORIAL ; SAIDA
PIN 14 OUT3  COMBINATORIAL ; SAIDA
PIN 18 STO   REGISTERED    ; V.ESTADO
PIN 19 ST1   REGISTERED    ; V.ESTADO
PIN 20 VCC
;-----------------------------
STATE
MEALY_MACHINE
START_UP := POWER_UP -> a
CLKF = CLOCK
;-----------------------------
;--- Transicoes de estado
a := IN1   -> b
  +  N_IN1 -> c
b := VCC   -> a
c := IN2   -> a
  +  N_IN2 -> b
;-----------------------------
;--- Equacoes para saidas
a.OUTF = IN1   ->  OUT1 * /OUT2 *  OUT3
       + N_IN1 ->  OUT1 * /OUT2 * /OUT3
b.OUTF =          /OUT1 * /OUT2 * /OUT3
c.OUTF =          /OUT1 *  OUT2 * /OUT3
;-----------------------------
;--- Atribuicao de estados
a = /ST1 * /STO
b = /ST1 *  STO
c =  ST1 * /STO
;-----------------------------
CONDITIONS
N_IN1 = /IN1
N_IN2 = /IN2
```

Texto 1.1: Código PALASM para diagrama ASM.

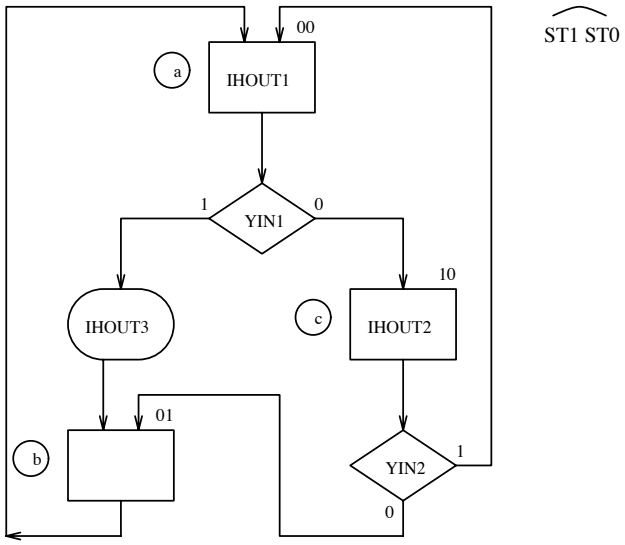


Figura 1.2: Diagrama ASM.

sincronização, tais como bloqueio[3] (quando dois controladores esperam indefinidamente um pelo outro) ou acesso múltiplo (quando uma operação na unidade de dados é pedida por dois controladores em simultâneo).

Assim, embora as técnicas baseadas em diagramas ASM sejam adequadas para sistemas puramente sequenciais, a sua aplicação é inapropriada para sistemas que exibam comportamento paralelo.

De entre os vários paradigmas de modelação, aquele que se baseia em RdP é o único que permite facilmente especificar subsistemas cooperantes, além de tornar possível a utilização de procedimentos formais de validação [VCBA83]. Adicionalmente, as RdP constituem uma linguagem gráfica fácil de compreender e de manipular, unívoca e universal. A utilização do GRAFCET é análoga ao uso de RdP. Todavia, há ligeiras diferenças que tornam difícil (ou sem valor) a aplicação de técnicas de análise aos modelos GRAFCET [Sil89].

Surge, assim, com naturalidade a seguinte questão:

Que formalismo se deve usar para especificar controladores paralelos?

A progressiva complexidade dos sistemas de controlo torna a sua concepção de difícil entendimento para a equipa de projecto e para os seus utilizadores. Este facto, segundo Silva [Sil89], torna conveniente:

- Fazer uma distinção bastante clara entre a especificação do sistema (algo parecido a um contrato) e a sua implementação;
- Proceder a uma verificação formal da especificação (na prática apenas são verificadas algumas características da especificação) antes de se proceder à sua implementação.

Esta divisão torna indispensável a existência de um formalismo que permita uma análise matemática dos sistemas. Desta forma, além de claro e preciso, um "bom" formalismo deve respeitar as seguintes características [Sil89]:

[3]Em língua inglesa usa-se o termo *deadlock*.

- Poder de expressão, permitindo que o modelo seja construído modularmente;
- Modelação clara e explícita de actividades e eventos;
- Representação gráfica que permita um diálogo entre utilizadores e projectistas;
- Existência de uma teoria forte de análise e verificação;
- Existência de aplicações para computador para auxílio da equipa de projecto, durante o período de verificação;
- Implementação fácil, usando um leque de soluções o mais vasto possível.

Neste sentido, o formalismo baseado em RdP parece ser o mais adequado, estando pois justificada a sua escolha, neste trabalho.

As RdP permitem especificar controladores paralelos. Essa especificação é feita a um nível bastante alto e, numa fase inicial, apenas se considera o comportamento do sistema, e não a sua implementação física. A especificação pode ser realizada de um modo hierárquico e modular, resultando ainda num maior nível de abstração.

É fundamental que a especificação, baseada nas RdP, possa ser implementada. Se isso não se verificar, o processo de especificação será de utilização discutível; eventualmente, poderá servir para simulação. Embora esse uso seja útil, durante determinadas fases do projecto, é imprescindível a possibilidade de implementar fisicamente o sistema.

A linguagem VHDL mostra-se, verdadeiramente, uma solução a considerar, como se justifica no capítulo 3. Primeiro, por se tratar de um *standard* que paulatinamente se tem imposto. Segundo, pelos diversos níveis de especificação que admite. Finalmente, por permitir que um sistema seja simulado e sintetizado, desde que a sua especificação obedeça a determinadas restrições.

O âmbito deste projecto restringiu-se à especificação da unidade de controlo, que poderá apresentar características paralelas. Adicionalmente, poder-se-ão usar macronodos, o que permite que a especificação seja feita de uma forma modular e hierárquica.

Neste trabalho, os sistemas são especificados a um nível alto, usando RdP. Através de uma série de rotinas, algumas propriedades são verificadas. Posteriormente, um processo automático permite obter o código VHDL respeitante ao sistema modelado pela RdP. Esse código poderá ser, então, usado para entrada em ferramentas de simulação ou de síntese.

O objectivo deste trabalho residia em verificar de que forma as RdP se mostram adequadas para especificar estruturas de controlo, com comportamento marcadamente paralelo, produzindo como resultado uma descrição da solução numa linguagem que permita a sua simulação e síntese automática com uma ferramenta de CAD. Adicionalmente, pretendia-se verificar se as técnicas de análise que as RdP disponibilizam, podem ser aplicadas aos controladores paralelos.

1.2 Estrutura do Relatório

A elaboração do relatório da presente dissertação de mestrado seguiu algumas das indicações e sugestões apresentadas em [Fer93].

O capítulo 2 é dedicado às RdP. São introduzidas as RdP de uma forma informal, seguindo-se a sua descrição formal. São ainda descritas algumas das técnicas de análise existentes para as RdP, assim como alguns tipos de RdP usados por variados autores. A finalizar, é apresentado o tipo de RdP escolhido para modelação de controladores paralelos, bem como algumas das características adicionais consideradas.

O capítulo 3 é dedicado inteiramente à linguagem de descrição de *hardware* VHDL. Faz-se uma introdução a essa linguagem, apresentando-se as suas características mais relevantes. A terminar o capítulo, foca-se a questão do subconjunto VHDL a usar para modelar diagramas ASM e RdP e apresentam-se alguns exemplos.

O capítulo 4 aborda todas as questões relativas à metodologia adoptada e à aplicação computacional desenvolvida. É feita uma revisão histórica dos vários métodos e produtos CAD propostos, na literatura, com a finalidade de especificar controladores paralelos, usando como suporte as RDP. De seguida, faz-se referência à linguagem ConPar desenvolvida, para permitir especificar controladores paralelos e apresenta-se o ambiente completo de desenvolvimento. São também discutidos, com algum detalhe, os módulos de análise de propriedades e de compilação desenvolvidos. É ainda considerado um exemplo de um controlador paralelo, para verificar a viabilidade da metodologia seguida.

No capítulo 5 apresentam-se as conclusões deste trabalho e são ainda dadas algumas pistas e sugestões para trabalho futuro.

Em apêndice são apresentados a linguagem desenvolvida, as opções da aplicação e um artigo realizado durante este trabalho.

Capítulo 2

Redes de Petri

"Petri nets are a model for procedures, organizations and devices where regulated flows, in particular information flows, play a role."

Wolfgang Reisig in "Petri Nets - an Introduction"

2.1 Introdução Informal

Nesta secção introdutória, pretende-se apresentar, de uma forma informal, as RdP. Introduzem-se os conceitos básicos como lugar, transição, marca, execução, disparo, capacidade, e discute-se a especificação hierárquica de sistemas.

2.1.1 Conceitos Básicos

As RdP têm-se mostrado bastante adequadas para a modelação de sistemas que exibem actividades assíncronas ou concorrentes. As áreas de aplicação têm sido diversas, das quais se salientam as seguintes:

- Protocolos de Comunicação;
- Avaliação de Desempenho;
- Modelação e Análise de Sistemas Distribuídos;
- Programação Concorrente e Paralela;
- Sistemas Flexíveis de Manufactura;
- Sistemas de Controlo Industrial;
- Sistemas de Eventos Discretos;
- Sistemas Computacionais de Multiprocessamento;

- Sistemas Computacionais *Data Flow*;
- Programação de Dispositivos Lógicos Programáveis (PLD);
- Estruturas e Circuitos Assíncronos;
- Sistemas Operativos;
- Sistemas de Informação;
- Linguagens Formais;
- Programação Lógica;
- Redes de Computadores Locais (LAN);
- Sistemas Legais;
- Redes Neurais;
- Filtros Digitais.

Na figura 2.1, apresenta-se uma RdP. É usual representar-se as RdP graficamente, pois essa forma funciona como um instrumento de trabalho, simulação, análise e visualização de extrema utilidade.

A representação gráfica das RdP permite a sua utilização para animação, ou seja, usando RdP para modelar um determinado sistema, é possível mostrar como o sistema se comporta [SV90].

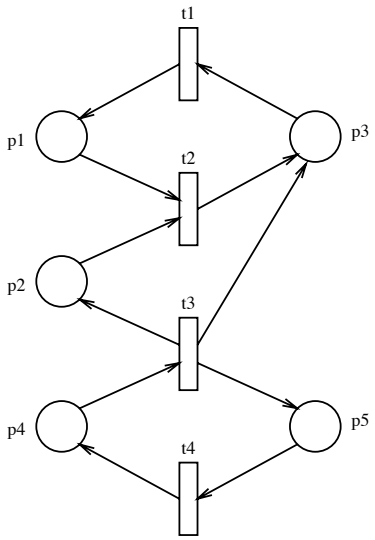


Figura 2.1: Exemplo de uma Rede de Petri.

O grafo associado à RdP modela as propriedades estáticas ou estruturais do sistema [Pet77]. O grafo contém dois tipos de nodos: círculos e rectângulos. Habitualmente, aos primeiros dá-se o nome de *lugares* e aos últimos chama-se *transições*.

Os nodos do grafo são ligados por arcos dirigidos, de lugares para transições e de transições para lugares. Não são permitidos arcos a ligarem nodos do mesmo tipo.

Se um arco está dirigido de um nodo i para um nodo j, diz-se que i é uma entrada de j, e j é uma saída de i. Por exemplo, na figura 2.1, a transição *t4* é uma entrada do lugar *p4* e é uma saída do lugar *p5*. Por outro lado, o lugar *p1* é uma entrada da transição *t2* e é uma saída da transição *t1*.

Consoante o tipo de sistema que se modela, é possível dar uma interpretação distinta aos lugares e às transições. No quadro 2.1 apresentam-se algumas dessas possíveis interpretações alternativas [Mur89].

Lugares de Entrada	Transições	Lugares de Saída
Pré-condições	Eventos	Pós-condições
Dados de entrada	Passo de computação	Dados de saída
Sinais de entrada	Processador de sinais	Sinais de saída
Recursos necessários	Tarefa	Recursos libertados
Condições	Claúsula em Lógica	Conclusões
Buffers	Processador	*Buffers*

Quadro 2.1: Algumas interpretações habituais dos lugares e das transições.

No entanto, é preciso algo mais além da simples estrutura atrás apresentada. Nesse sentido, introduz-se o conceito de marcação e marca. Uma *marca* é um conceito básico, assim como o conceito de lugar ou transição, na teoria das RdP. As marcas são atribuídas aos lugares e pode-se pensar que neles residem. Uma *marcação* é uma atribuição de marcas aos lugares. O número e a posição das marcas podem alterar-se durante a execução da RdP.

Graficamente, as marcas são representadas por um pequeno círculo a cheio (•). Na figura 2.2, os lugares *p1*, *p2* e *p4* contêm uma marca e os restantes lugares (*p3* e *p5*) não contêm marcas. A RdP apresentada na figura 2.2 diz-se *marcada*. Quando o número de marcas num dado lugar excede um determinado valor, é usual substituir os círculos por um número.

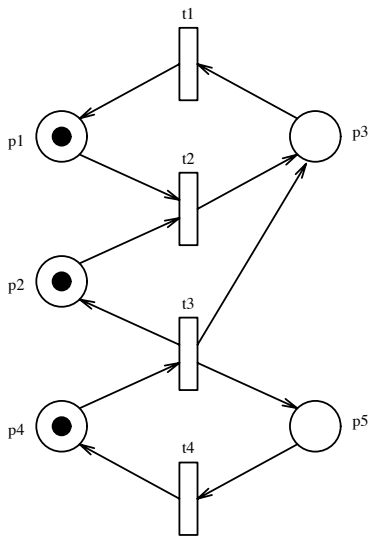


Figura 2.2: Exemplo de uma Rede de Petri marcada.

A *execução* de uma RdP é controlada pelo número e pela distribuição das marcas. Assim, a partir de uma determinada marcação, uma RdP é executada (ou o sistema que está a

ser modelado, é simulado), através do *disparo* de transições. Para que uma transição possa disparar, é necessário que esteja habilitada. Uma transição diz-se *habilitada*, se todos os lugares de entrada dessa transição contêm, pelo menos, uma marca. A transição, ao ser disparada, remove uma marca de cada um dos seus lugares de entrada e adiciona uma marca a cada um dos seus lugares de saída.

Por exemplo, retomando a figura 2.2, verifica-se que as transições *t2* e *t3* estão habilitadas. A transição *t2* está habilitada porque os seus lugares de entrada (*p1* e *p2*) estão marcados. O mesmo sucede com a transição *t3*, pois o seu único lugar de entrada (*p4*) está marcado. A transição *t1*, por seu lado, não está habilitada a disparar, porque o seu único lugar de entrada (*p3*) não está marcado. Deste modo, qualquer das transições *t2* e *t3* pode disparar. A escolha de qual a transição a disparar é feita de uma forma não-determinística[1] ou por forças que não são modeladas, isto é, não há qualquer regra que defina qual a transição que dispara, quando num dado instante, há um conjunto delas habilitadas a fazê-lo. Se a transição *t2* disparar, há que remover uma marca dos lugares *p1* e *p2* e adicionar uma marca ao lugar *p3*. A marcação resultante é apresentada na figura 2.3.

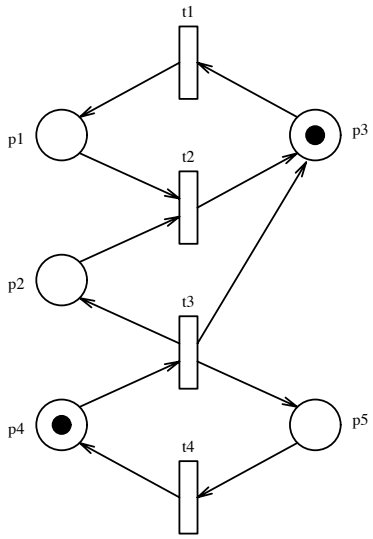


Figura 2.3: Rede de Petri após disparo da transição t2.

Considerando, agora, a RdP apresentada na figura 2.3, verifica-se que as transições *t1* e *t3* são as únicas que estão habilitadas. Se a transição *t3* disparar, a marcação resultante é a apresentada na figura 2.4.

Como se verifica na figura 2.4, o número de marcas num lugar pode ser diferente de zero e um. É o que acontece com o lugar *p3*.

A execução de uma RdP continua, enquanto houver transições habilitadas. Se, numa dada marcação, não houver qualquer transição habilitada, a execução estagna (pára). Pode ver-se a execução de uma RdP como uma sequência discreta de eventos[2].

[1] Esta característica das RdP reflecte o facto de em situações da vida real, onde vários factos podem suceder simultaneamente, a ordem da ocorrência dessses acontecimentos não ser única. Se o não-determinismo é vantajoso do ponto de vista da modelação, introduz todavia alguma complexidade na análise do comportamento da RdP [Pet77].

[2] Considera-se, neste caso, a interpretação que relaciona eventos do sistema modelado com transições no modelo baseado numa RdP.

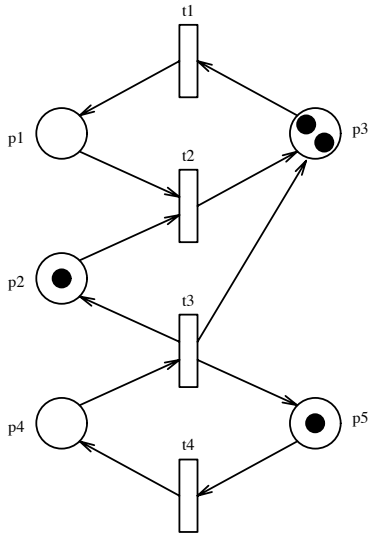


Figura 2.4: Rede de Petri após disparo da transição t3.

Nos exemplos atrás considerados, a cada arco estava associado um peso unitário, ou seja, o número de marcas adicionado ou subtraído a cada lugar, sempre que uma transição dispara, é igual a um.

É possível aumentar o poder de modelação das RdP, atribuindo pesos aos arcos. Se o peso de um arco (valor inteiro positivo) entre dois nodos for igual a k, esse facto pode ser interpretado como havendo um conjunto de k arcos a ligar os dois nodos em questão.

Dessa forma, para que uma transição esteja habilitada a disparar, é necessário que o número de marcas em cada um dos lugares de entrada dessa transição seja superior ou igual ao peso do arco que liga esse lugar à transição em questão. Se tal condição se verificar e se essa transição for disparada, é subtraído, a cada lugar de entrada dessa transição, um número de marcas igual ao peso do arco que liga o lugar à transição. Analogamente, a cada lugar de saída dessa transição é adicionado um número de marcas igual ao peso do arco que liga a transição ao lugar. Como exemplo, considere-se a figura 2.5, que modela a seguinte fórmula química:

$$2SO_2 + O_2 \rightarrow 2SO_3$$

Na figura 2.5(a), a transição t está habilitada pois o número de marcas no lugar SO_2 é 3 e no lugar O_2 é de 1, valores esses iguais ou superiores aos pesos dos arcos que ligam esses lugares à transição t. Assim sendo, se a transição t disparar, a marcação resultante é a que se apresenta na figura 2.5(b).

Também se considerou nos exemplos anteriores que cada lugar pode receber um número ilimitado de marcas. No entanto, para um grande número de sistemas reais, é lícito pensar-se que cada lugar tenha um limite superior para o número de marcas que pode conter. Por exemplo, uma sala de cinema tem um número limite de pessoas que pode receber e um parque de estacionamento não pode albergar mais do que um determinado número de automóveis.

Se se atribuir a cada lugar uma capacidade finita, uma transição passa a estar habilitada se se verificar uma outra condição: cada lugar de saída da transição não pode exceder a sua capacidade, ou seja, o número de marcas a adicionar ao lugar não pode fazer com que a respectiva capacidade seja excedida. Por exemplo, na figura 2.3, se a capacidade do lugar

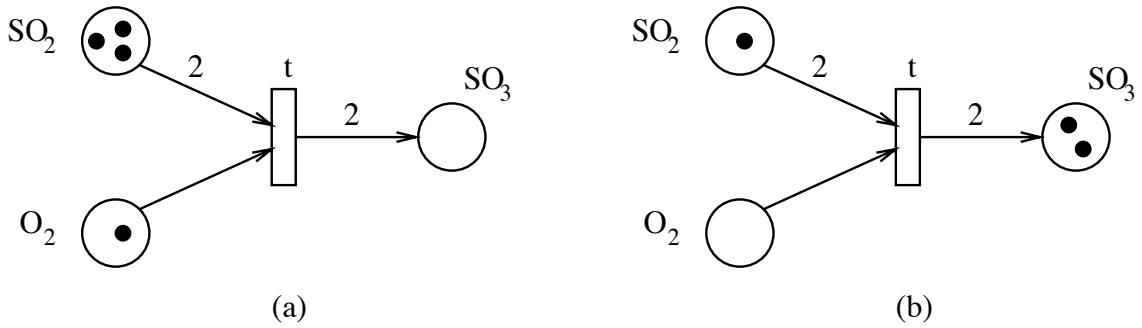


Figura 2.5: Reacção química, usando uma RdP. (a) A marcação antes do disparo da transição t. (b) A marcação após o disparo da transição t.

p3 for de uma marca, a transição *t3* deixa de estar habilitada.

Esta situação requer uma análise mais aprofundada. Admita-se que os lugares das RdP, a seguir consideradas, têm capacidade unitária. A generalização para RdP com lugares de capacidade superior a um não apresenta grande dificuldade.

Quando um dos lugares de saída inibe o disparo de uma transição está-se perante uma situação de *contacto*. À primeira vista pode não parecer justificado o facto de a transição não estar habilitada. Poder-se-ia propor que cada lugar de saída que estivesse marcado, antes do disparo da transição, assim permanecesse após o disparo da mesma. Vejam-se as implicações de tal proposta, através de alguns exemplos: um copo cheio poder-se-ia encher de novo, o outono pode começar quando já é outono, um lugar reservado poderia reservar-se novamente, um carro poderia mover-se para um local onde já se situa outro carro, etc. Alguns destes acontecimentos são possíveis, embora indesejáveis, e outros são mesmo impossíveis.

Com os exemplos atrás apresentados, emergem imediatamente três das propriedades mais características que identificam todos os modelos baseados em RdP: o assincronismo, o paralelismo e o não determinismo.

O *assincronismo* é evidente. Não há qualquer noção de tempo inerente a um modelo baseado em RdP. Existe apenas uma ordem parcial no disparo das transições (na ocorrência de eventos).

O *paralelismo* assenta na possibilidade de duas transições habilitadas, que não interajam uma com a outra, poderem ocorrer independente e paralelamente.

O *não determinismo* surge do facto de a ordem de disparo de duas ou mais transições simultaneamente habilitadas poder ser escolhida pelo utilizador da RdP. Mais concretamente, não existe nenhuma regra que indique qual das transições habilitadas deva disparar.

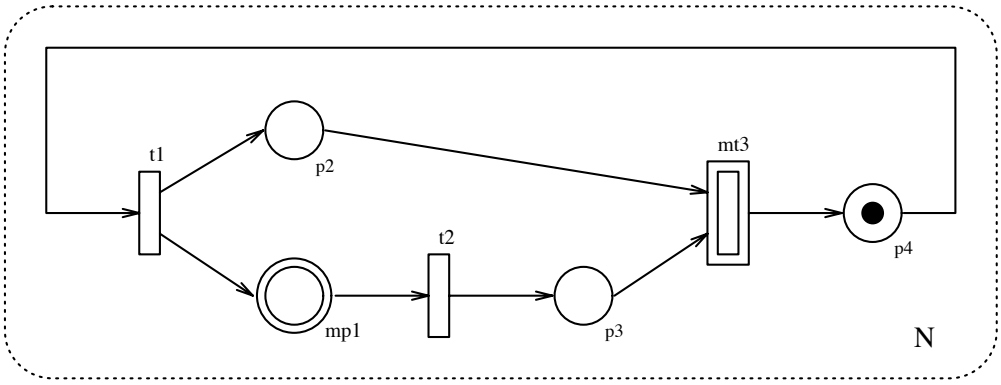
Em quase todos os sistemas reais, estas propriedades estão presentes. Por exemplo, numa fábrica, os empregados podem (e devem!) estar a trabalhar em simultâneo — paralelismo. O assincronismo também é evidente: o empregado de uma linha de montagem só pode tratar uma peça quando o anterior tiver terminado de a processar. Finalmente, o não determinismo resulta do facto de não se saber *a priori* qual a ordem de execução de dois eventos que podem ocorrer paralelamente (será que o empregado pega primeiro no prego ou no martelo?).

Um problema tradicional com a análise do espaço de estados para sistemas concorrentes (dos quais as RdP são um exemplo) reside na explosão exponencial no tamanho do espaço de estados relativamente ao tamanho do sistema. Esta explosão consome, não só tempo, como também capacidade de armazenamento [SR92]. Normalmente, diz-se que um dos problemas que as RdP apresentam é a *complexidade*, quando se pretende referir o problema atrás descrito. Para tentar obviar este problema, surgiram as RdP Hierárquicas.

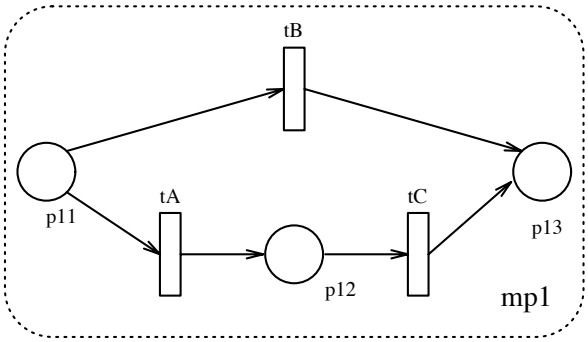
2.1.2 Redes de Petri Hierárquicas

A modelação de sistemas relativamente complexos é fiável somente quando existêm mecanismos de estruturação. De entre esses mecanismos, a modelação hierárquica é uma das características mais relevantes que as RdP apresentam. A modelação hierárquica de sistemas apresenta as seguintes vantagens [Feh93]:

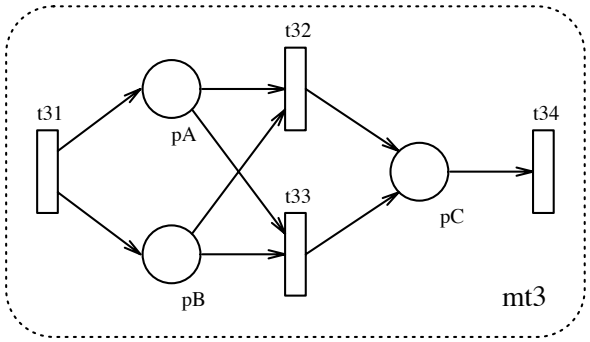
- Inspecção do sistema a vários níveis de detalhe;
- Visualização de partes seleccionadas do sistema (ex: o refinamento de um nodo);
- Re–utilização fácil e múltipla de partes do modelo.



(a)



(b)



(c)

Figura 2.6: RdP hierárquica. (a) Nível mais abstracto. (b) Refinamento do macrolugar mp1. (c) Refinamento da macrotransição mt3.

A modelação hierárquica apresenta ainda uma outra vantagem adicional e de extrema importância: os variados testes aplicados às RdP podem ser efectuados também hierarquica-

mente. Assim, um dos grandes problemas que as RdP apresentam, a complexidade, é de alguma forma atenuado.

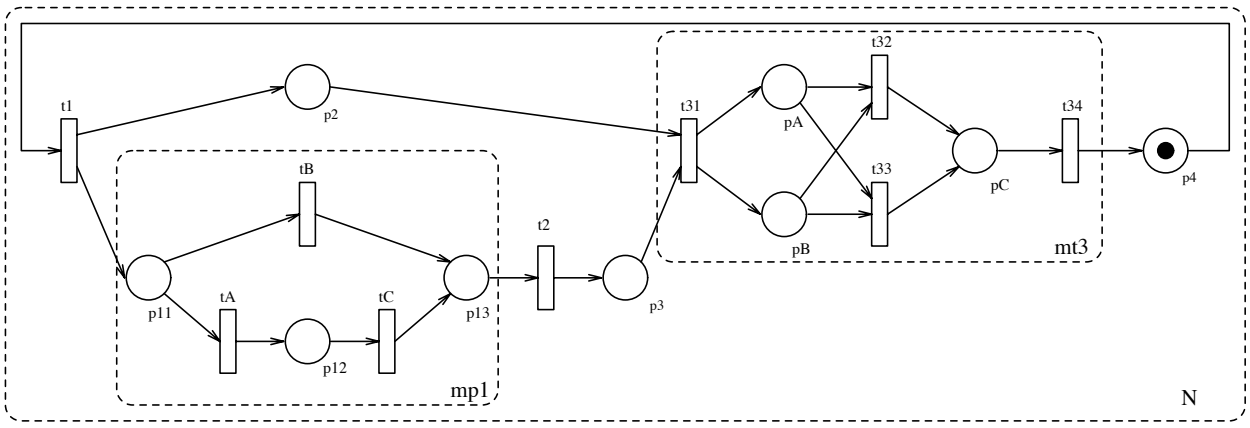


Figura 2.7: Linearização de uma RdP, após refinamento dos macronodos.

Uma subrede completa pode ser substituída por uma única transição ou por um único lugar, o que permite modelar a um nível mais abstracto, ou os lugares e as transições podem ser substituídos por subredes, de modo a fornecer informação mais detalhada. A figura 2.6 pretende ilustrar estas ideias. Nessa figura surgem dois macronodos. Um *macronodo* é um nodo que representa uma subrede. O macronodo *mp1* é um *macrolugar*, representado por duas circunferências concêntricas. O macronodo *mt1* é uma *macrotransição* e representa-se por dois rectângulos.

As *portas* de uma subrede são os nodos relacionados com o exterior da subrede. Para o macrolugar *mp1* (macrotransição *mt3*), o lugar *p11* (a transição *t31*) é uma porta de entrada e o lugar *p13* (a transição *t34*) é uma porta de saída.

As *chaves* de uma subrede são os nodos adjacentes à subrede, ligados por arcos. Para o macrolugar *mp1* (macrotransição *mt3*), a transição *t1* (tanto o lugar *p2* como o *p3*) é uma chave de entrada e a transição *t2* (o lugar *p4*) é uma chave de saída.

Na figura 2.7, apresenta-se a RdP que resulta da linearização da RdP hierárquica da figura 2.6.

2.2 Definição Formal

Na secção anterior, introduziu-se, de uma forma informal, as RdP para permitir uma primeira abordagem bastante acessível. Nesta secção, pretende-se definir formalmente as RdP. Para tal, será usada a notação apresentada por Reisig [Rei85].

Uma rede é definida da forma seguinte:

Definição 1 (Rede) *Um triplo $N = (S, T; F)$ é uma rede sse*[3] *os conjuntos S e T forem disjuntos ($S \cap T = \emptyset$) e $F \subseteq (S \times T) \cup (T \times S)$ é uma relação binária.* □

[3] *se e só se.*

Aos elementos do conjunto S dá-se, habitualmente, o nome de lugares[4] e aos elementos do conjunto T chama-se transições. A F dá-se o nome de *relação de fluxo.*

Notação 1 *Seja $N = (S, T; F)$ uma rede. Por vezes, denotam-se as três componentes S, T e F por S_N, T_N e F_N, respectivamente. Escreve-se, ainda, N para representar $S \cup T$, se daí não resultar qualquer confusão.* □

Definição 2 (Transições e Lugares de entrada e saída) *Sejam N uma RdP, $s \in S_N$ e $t \in T_N$. s diz-se um* lugar de entrada *de t e t diz-se uma* transição de saída *de s se existe um arco de s para t, i.e. sF_Nt. s diz-se um* lugar de saída *de t e t diz-se uma* transição de entrada *de s, se existe um arco de t para s, i.e. tF_Ns.* □

Definição 3 (Pré- e Post-conjunto) *Seja N uma rede. ${}^\cdot x = \{y \mid yF_Nx\}$ é o* pré-conjunto *de x e $x^\cdot = \{y \mid xF_Ny\}$ é o* post-conjunto *de x. Para $X \subseteq N$, sejam ${}^\cdot X = \bigcup_{x \in X} {}^\cdot x$ e $X^\cdot = \bigcup_{x \in X} x^\cdot$.* □

Definição 4 (Transições e Lugares fonte e destino) *Sejam N uma RdP, $s \in S_N$ e $t \in T_N$. s diz-se um* lugar fonte *se não tem qualquer transição de entrada, i.e. ${}^\cdot s = \emptyset$. s diz-se um* lugar destino *se não tem qualquer transição de saída, i.e. $s^\cdot = \emptyset$. t diz-se uma* transição fonte *se não tem qualquer lugar de entrada, i.e. ${}^\cdot t = \emptyset$. t diz-se uma* transição destino *se não tem qualquer lugar de saída, i.e. $t^\cdot = \emptyset$.* □

Definição 5 (Ciclo próprio) *Seja N uma rede. Um par $(s, t) \in S_N \times T_N$ é chamado* ciclo próprio *sse $sF_Nt \wedge tF_Ns$.* □

Definição 6 (Rede pura) *Uma rede N diz-se* pura *sse F_N não contém qualquer ciclo próprio.* □

Definição 7 (Elemento isolado) *Seja N uma rede. $x \in N$ diz-se* elemento isolado *sse ${}^\cdot x \cup x^\cdot = \emptyset$.* □

Definição 8 (Rede simples) *Uma rede N diz-se* simples *sse elementos distintos não têm os mesmos pré- e post-conjunto, i.e. $\forall x, y \in N : ({}^\cdot x = {}^\cdot y \wedge x^\cdot = y^\cdot) \Rightarrow x = y$.* □

Pode-se, agora, definir uma RdP, assumindo que o símbolo ω representa o infinito:

Definição 9 (Rede de Petri) *Um sextuplo $N = (S, T; F, K, M, W)$ é uma* Rede de Petri *sse $(S, T; F)$ é uma rede finita, $K : S \to \mathbb{N} \cup \{\omega\}$ dá uma* capacidade *(possivelmente infinita) para cada lugar, $W : F \to \mathbb{N} \setminus \{0\}$ atribui um* peso *a cada arco da rede e $M : S \to \mathbb{N} \cup \{\omega\}$ é a* marcação inicial*, respeitando as capacidades, i.e. $\forall s \in S : M(s) \leq K(s)$.* □

Notação 2 *Analogamente às redes, as componentes de uma RdP N podem ser denotadas por S_N, T_N, F_N, K_N, M_N e W_N.* □

[4] Em língua alemã: *Stellen.*

Definição 10 (Marcação) *Seja N uma RdP. Um mapeamento $M : S \to \mathbb{N} \cup \{\omega\}$ é uma* marcação *de N sse $\forall s \in S : M(s) \leq K(s)$.* □

Definição 11 (Transição M-habilitada) *Seja M uma marcação de uma RdP N. Uma transição $t \in T_N$ diz-se* M-habilitada *sse $\forall s \in {}^\cdot t : M(s) \geq W_N(s,t)$ e $\forall s \in t^\cdot : M(s) \leq K_N(s) - W_N(t,s)$.* □

Definição 12 (Marcação seguinte) *Seja M uma marcação de uma RdP N. Se uma transição $t \in T_N$ está M-habilitada, pode originar uma* marcação seguinte *M' de M, que obedece às seguintes regras para cada $s \in S_N$*

$$M'(s) = \begin{cases} M(s) - W_N(s,t) & \Leftarrow s \in {}^\cdot t \setminus t^\cdot, \\ M(s) + W_N(t,s) & \Leftarrow s \in t^\cdot \setminus {}^\cdot t, \\ M(s) - W_N(s,t) + W_N(t,s) & \Leftarrow s \in {}^\cdot t \cap t^\cdot, \\ M(s) & \textit{caso contrário.} \end{cases}$$

□

Diz-se que t dispara de M para M', e escreve-se notacionalmente $M[t > M'$.

Definição 13 (Conjunto de marcações) *Sejam M uma marcação de uma RdP N e $[M >$ o menor conjunto de marcações, tal que $M \in [M >$ e se $M_1 \in [M >$ e para algum $t \in T_N : M_1[t > M_2$ então $M_2 \in [M >$.* □

Na representação gráfica das RdP, os arcos $f \in F$ são etiquetados por $W(f)$ se $W(f) > 1$. A capacidade de um lugar $s \in S$ é representada por "$\kappa = K(s)$". A inscrição "$\kappa = \omega$" pode ser omitida, pois é admitida por defeito. Uma marcação M é representada pelo desenho de $M(S)$ marcas ou pelo símbolo ω, em cada lugar $s \in S$.

Definição 14 (RdP ordinária) *Uma RdP N diz-se* ordinária *se o peso de qualquer dos seus arcos for iguais a 1, i.e. $\forall f \in F_N : W_N(f) = 1$.* □

Definição 15 (Lugar k–limitado e seguro) *Um lugar p de uma RdP N diz-se k–*limitado*, se o número de marcas, nesse lugar, não é superior a um número finito k, para qualquer marcação atingível a partir da marcação inicial M_N, i.e. $\forall M' \in [M_N > : M'(p) \leq k$. Um lugar p diz-se* seguro *se for 1–limitado.* □

Definição 16 (RdP k–limitada e segura) *Uma RdP N diz-se k–*limitada*, se todos os seus lugares forem k–limitados, i.e. $\forall s \in S_N, \forall M' \in [M_N > : M'(p) \leq k$. Uma RdP N diz-se* segura *se for 1–limitada.* □

Definição 17 (RdP ilimitada) *Uma RdP N diz-se* ilimitada*, se não existir um número finito k, tal que N é k–limitada, i.e. $\forall k \in \mathbb{N}, \exists M' \in [M_N >, p \in P_N : M'(p) \geq k$.* □

Repare-se que um lugar que seja n–limitado, também é, por definição, $(n+1)$–limitado. Analogamente, uma RdP que seja n–limitada, também é $(n+1)$–limitada.

Definição 18 (Transição e RdP vivas) *Sejam N uma RdP e $t \in T_N$. t diz-se* viva *sse* $\forall M \in [M_N >, \exists M' \in [M >$: *$t$ é M'-habilitada. N diz-se* viva *sse* $\forall t \in T_N : t$ *é viva.* □

Definição 19 (Transição morta) *Sejam N uma RdP e $t \in T_N$. t diz-se* morta *sse* $\forall M \in [M_N >$: *t não é M-habilitada.* □

Relativamente às RdP hierárquicas, a seguinte terminologia é habitualmente usada [LF85].

Definição 20 (Grau de relação dum nodo) *Sejam N uma RdP e $x \in N$ um nodo de N. O* grau de relação de entrada, *D_{in}, de um nodo é dado pelo número de elementos do pré-conjunto desse nodo, i.e. $D_{in}(x) = \#{}^{\cdot}x$. O* grau de relação de saída, *D_{out}, de um nodo é dado pelo número de elementos do post-conjunto desse nodo, i.e. $D_{out}(x) = \#x^{\cdot}$. O* grau de relação, *D, de um nodo é dado pela soma dos graus de relação de entrada e saída desse nodo, i.e. $D(x) = D_{in}(x) + D_{out}(x)$.* □

Definição 21 (Macronodo) *Um* macronodo *é um nodo da RdP que representa uma subrede. Os macronodos encontram-se divididos em duas categorias:* macrolugares *e* macrotransições. □

Definição 22 (Grau de uma subrede) *Seja N uma RdP e M uma sua subrede, i.e. $S_M \subseteq S_M, T_M \subseteq T_M, F_M \subseteq F_M$. O* grau, *$Q$, de uma subrede é a soma dos graus de relação dos nodos na subrede, cujo valor é superior a 2, i.e. $Q(M) = \sum_{x \in M} D(x)$, para $D(x) > 2$.* □

Definição 23 (Porta de uma subrede) *Seja N uma RdP e M uma sua subrede. Uma* porta *de uma subrede é um seu nodo relacionado com o exterior da subrede. O conjunto das portas de uma subrede é dado por: $\{x' | x' \in M, x \in N \setminus M : xF_N x'\}$.* □

Definição 24 (Macrolugar e macrotransição) *Um* macrolugar *é uma subrede cujas portas são unicamente lugares. Uma* macrotransição *é uma subrede cujas portas são unicamente transições.* □

2.3 Análise de Redes de Petri

A modelação baseada em RdP permite a especificação de um determinado sistema. No entanto, são necessárias técnicas analíticas que permitam examinar uma RdP e determinar quais as suas propriedades.

Entre as propriedades que normalmente se pretende que um sistema goze, encontram-se, por exemplo, a vivacidade (ausência de bloqueios) ou a limitação (ausência de transbordo).

O facto de um sistema verificar um conjunto de propriedades previamente definido, não implica que o seu funcionamento seja, obrigatoriamente, aquele que se pretendia inicialmente. Contudo, é lícito considerar que se as tais propriedades se verificam, então o sistema não contém qualquer erro patológico.

Nesta secção faz-se uma abordagem às técnicas de análise de RdP actualmente disponíveis, que, segundo Murata [Mur89], podem ser divididas em três grupos principais:

1. Grafos de cobertura ou alcançabilidade[5];
2. Matriz;
3. Técnicas de redução e decomposição.

2.3.1 Grafo de Cobertura

Dada uma RdP, é possível obter, a partir da marcação inicial, tantas marcações novas, quantas as transições habilitadas. A partir das marcações novas obtidas, podem-se obter, novamente, mais marcações novas. Deste processo resulta uma representação arbórea das marcações.

A representação atrás descrita apresenta um problema. Se a RdP for ilimitada, a representação crescerá infinitamente. No entanto, pode-se obter uma árvore finita em que cada marcação atingível, ou está representada explicitamente por um nodo do grafo, ou então está "coberta" por um nodo [Pet81, Rei85, Mur89]. A essa árvore dá-se o nome de "*Árvore de Cobertura*".

Nesse sentido, introduz-se um símbolo especial ω, que pode ser visto como "infinito". Para qualquer valor inteiro n, esse símbolo apresenta as seguintes propriedades: $\omega > n$; $\omega + n = \omega$; $\omega - n = \omega$; $\omega \geq \omega$.

A árvore de cobertura para uma RdP N é construída segundo o seguinte algoritmo [Pet81]:

1. Colocar a marcação inicial M_N como a raiz e marca-lá como "nova".
2. Enquanto existirem marcações novas fazer:
   (a) Seleccionar uma marcação nova M.
   (b) Se M é igual a uma outra marcação, presente na árvore, não marcada como "nova", então marcar M como "velha" e voltar para o início do passo 2.
   (c) Se não há transições habilitadas em M, marcar M como "fim".
   (d) Enquanto existirem transições habilitadas em M, para cada transição habilitada t em M, fazer:
      i. Obter a marcação M' que resulta do disparo de t em M.
      ii. Se existe, no caminho da raiz até M, uma marcação M'' tal que $M'(s) \geq M''(s)$, para cada lugar s, e $M' \neq M''$, i.e. se M'' é coberto por M', então substituir $M'(s)$ por ω, para cada p em que $M'(s) \geq M''(s)$.
      iii. Introduzir M' como um nodo, desenhar um arco de M para M' com etiqueta t e marcar M' como "nova".

Este algoritmo termina, já que a árvore de cobertura de uma RdP é garantidamente finita [Pet81, Rei85].

Como exemplo para aplicação do algoritmo atrás descrito, considere-se a figura 2.8, onde se representa uma RdP. Na figura 2.9(a) é apresentada a respectiva árvore de cobertura. Cada marcação é representada pelo vector $\{M(p1), M(p2), M(p3), M(p4)\}$

[5]Em língua inglesa, os termos cobertura e alcançabilidade são conhecidos por *coverability* e *reachability*, respectivamente.

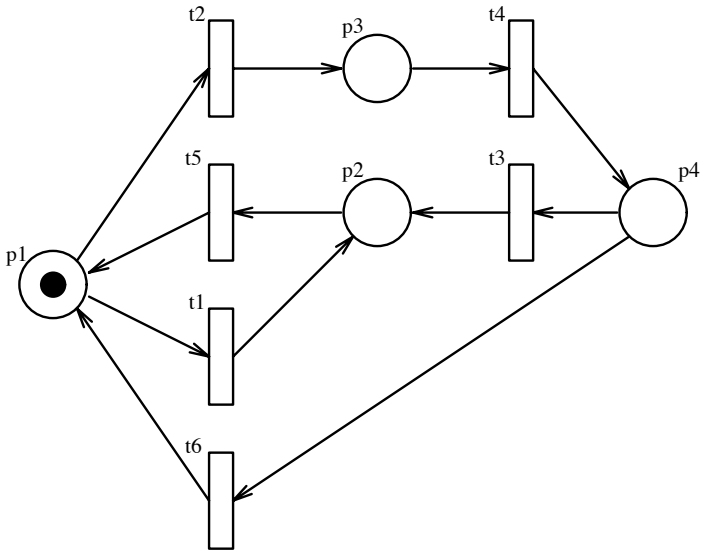


Figura 2.8: Exemplo de uma RdP.

O nodo {1,0,0,0}, presente na árvore de cobertura, corresponde à marcação inicial. Nesta marcação, existem duas transições habilitadas, *t1* e *t2*. Se disparar a transição *t1*, a marcação resultante é {0,1,0,0}. Se for disparada a transição *t2*, obtém-se a marcação {0,0,1,0}. Estes dois potenciais disparos são representados, na árvore de cobertura, pelos arcos que ligam o nodo {1,0,0,0} aos nodos {0,1,0,0} e {0,0,1,0}, respectivamente.

Na marcação {0,1,0,0}, apenas a transição *t5* está habilitada. A marcação resultante, após o disparo da transição *t5* é {1,0,0,0}, que corresponde ao nodo inicialmente introduzido (marcação inicial). O processo de construção da árvore de cobertura continuaria, até se esgotarem as marcações marcadas como "novas".

Com o intuito de tornar ainda mais compacta a representação, é mais comum construir-se o "*Grafo de Cobertura*". Basicamente, no grafo de cobertura não há nodos iguais, o que se traduz numa redefinição dos pontos 2.(a) e 2.(d)iii do algoritmo atrás apresentado. Contudo, o grafo e a árvore de cobertura dão origem a estruturas com o mesmo conteúdo de informação. Mais concretamente, pode-se saber quais as marcações atingíveis e que transições estão habilitadas nessas marcações.

Para a RdP considerada na figura 2.8, o grafo de cobertura é apresentado na figura 2.9(b). Atente-se no facto de o nodo {0,1,0,0}, por exemplo, surgir somente uma vez, ao contrário do que sucede na árvore de cobertura. Repare-se, também, que o número de arcos é o mesmo, tanto na árvore como no grafo.

O exemplo anterior serviu de exemplificação do algoritmo de construção do grafo e da árvore de cobertura, usando uma RdP bastante simples. Considere-se, agora, a RdP apresentada na figura 2.10. Aplicando o algoritmo a essa RdP, obtém-se o grafo de cobertura ilustrado na figura 2.11. Representa-se, no presente exemplo, cada marcação pelo vector $\{M(p1), M(p2), M(p3), M(p4), M(p5)\}$

O nodo {1,0,0,0,0}, presente no grafo de cobertura, corresponde à marcação inicial da RdP. Nessa marcação, há duas transições habilitadas: *t1* e *t2*. Quando a transição *t1* dispara, a marcação resultante é {0,1,0,0,1}. Nesta marcação, apenas a transição *t5* está habilitada. A marcação resultante do disparo de *t5* é {1,0,0,0,1}. Porém, verifica-se que, no grafo de cobertura, essa marcação não está explicitamente representada, aparecendo contudo o nodo

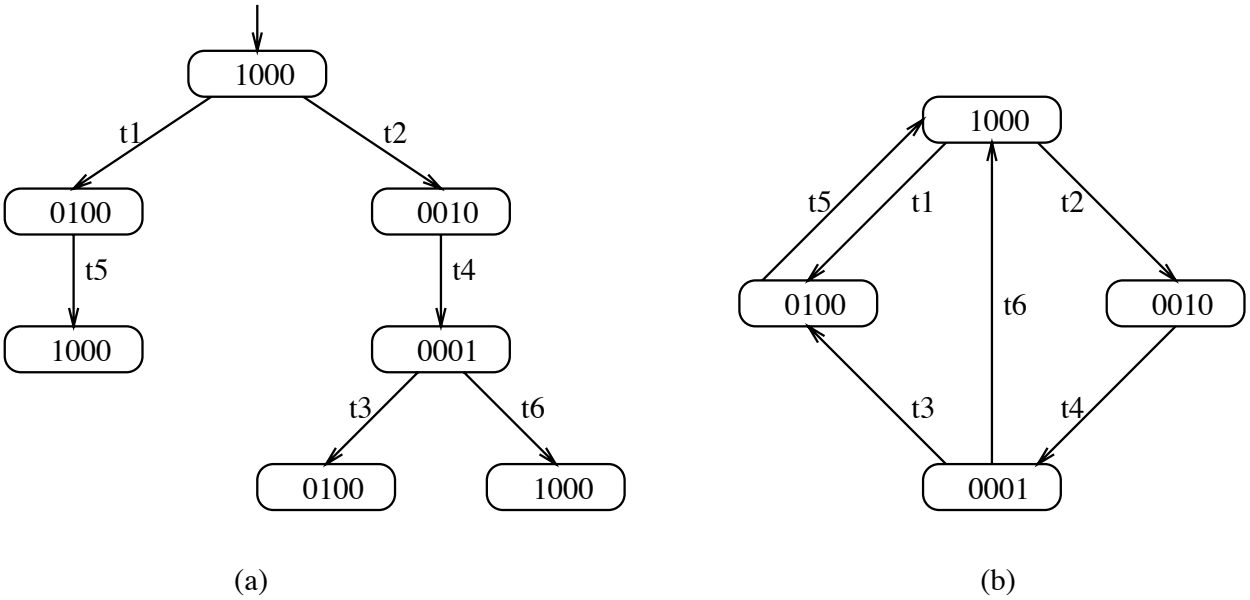


Figura 2.9: (a) Árvore de cobertura. (b) Grafo de cobertura.

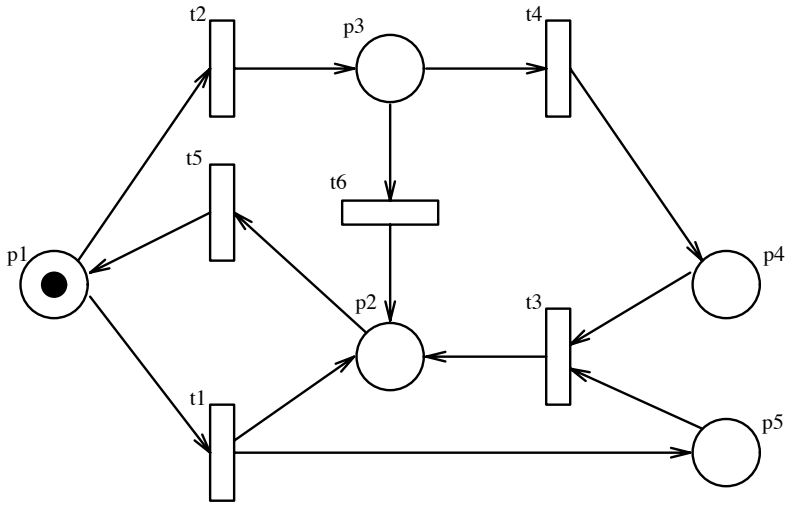


Figura 2.10: Exemplo de uma RdP mais complicada.

{1,0,0,0,ω}. Tal deve-se ao facto da condição do ponto 2.(d)ii do algoritmo se verificar, ou seja, a marcação inicial é coberta por esta nova marcação. Este novo nodo[6] corresponde à possibilidade da sequência *t1* - *t5* puder realizar-se inúmeras vezes, no que resulta um número elevado de marcas no lugar *p5*. A restante construção do grafo de cobertura seria efectuada até não existirem mais nodos novos.

Algumas das propriedades que podem ser estudadas, usando o grafo de cobertura T relativo a uma RdP N, são [Mur89]:

1. Uma RdP N é limitada sse não surge o símbolo ω em qualquer dos nodos de T.
2. Uma RdP N é segura sse apenas aparecem 0's e 1's nos nodos de T.
3. Se uma marcação M de N é atingível a partir de M_N, então existe um nodo M' de T tal que $M \leq M'$.

[6]Ver, também, os arcos etiquetados por *t1* e *t5* que ligam os nodos {1,0,0,0,ω} e {0,1,0,0,ω}.

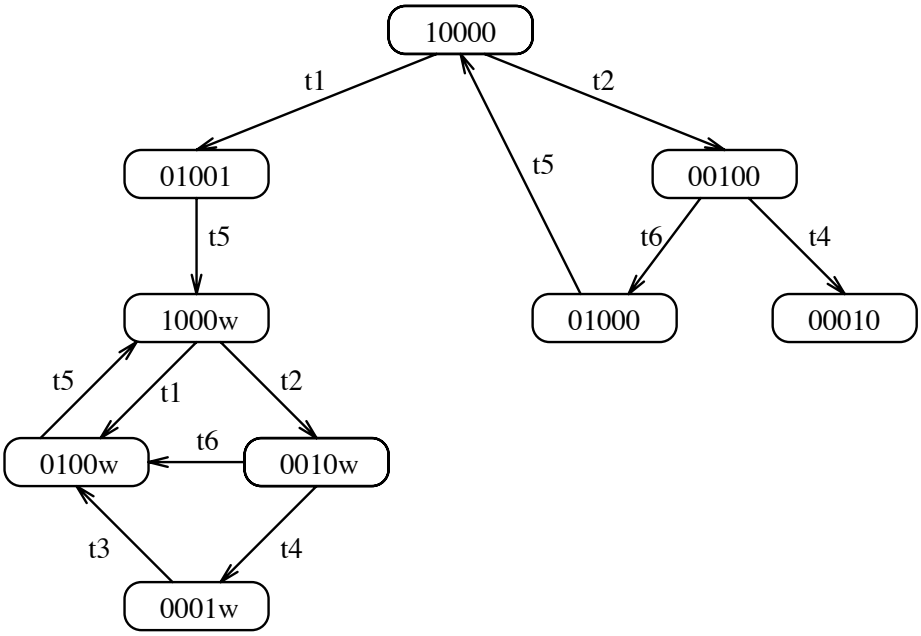


Figura 2.11: Grafo de cobertura.

4. Uma transição t é morta sse não há nenhum arco etiquetado por t.

Infelizmente, devido ao uso do símbolo ω, alguma informação pode perder-se (ex: ω pode representar apenas números pares), pelo que verificar se a RdP é viva ou se uma marcação é alcançável, a partir da marcação inicial, podem ser problemas sem solução se se basear a análise no grafo de cobertura.

2.3.2 Matriz

O estudo do comportamento das RdP pode também ser realizado através de uma representação algébrica linear [Rei85]. Em geral, este método baseia-se na matriz de incidência e nos invariantes lineares da RdP.

Definição 25 (Matriz de Incidência duma RdP) *Seja $N = (S, T; F, K, M, W)$ uma RdP. Para cada transição $t \in T$, seja o vector $\underline{t} : S \to \mathbb{Z}$ definido por:*

$$\underline{t}(s) = \begin{cases} -W(s,t) & \Leftarrow s \in {}^{\cdot}t \setminus t^{\cdot}, \\ W(t,s) & \Leftarrow s \in t^{\cdot} \setminus {}^{\cdot}t, \\ W(t,s) - W(s,t) & \Leftarrow s \in {}^{\cdot}t \cap t^{\cdot}, \\ 0 & \textit{caso contrário.} \end{cases}$$

Seja a matriz $\underline{N} : S \times T \to Z$ definida por $\underline{N}(s,t) = \underline{t}(s)$. □

Assim, para a RdP da figura 2.8, a matriz de incidência $\underline{N}$ é dada por:

$$\begin{array}{c} \\ p1 \\ p2 \\ p3 \\ p4 \end{array} \begin{array}{c} \begin{array}{cccccc} t1 & t2 & t3 & t4 & t5 & t6 \end{array} \\ \begin{pmatrix} -1 & -1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & -1 \end{pmatrix} \end{array}$$

A leitura desta matriz é simples. Os valores constantes numa coluna (cada coluna corresponde a uma transição) indicam as alterações que os lugares (cada linha corresponde a um lugar) sofrem no seu número de marcas, ou seja, $\underline{N}(s_i, t_j)$ descreve a alteração na marcação do lugar s_i quando a transição t_j dispara.

Por exemplo, a coluna etiquetada por *t1* indica que, quando esta transição dispara, é retirada uma marca do lugar *p1* (valor -1), é adicionada uma marca ao lugar *p2* (valor $+1$) e que os restantes lugares mantêm o seu número de marcas.

A representação matricial mostra-se exacta somente para RdP puras (ver definição 6). Se, adicionalmente, a RdP N for livre de contactos, então o comportamento da RdP pode ser totalmente derivado pela sua matriz $\underline{N}$ e pelo vector M_N (marcação inicial). Note-se que as marcações da RdP podem também ser representadas por vectores.

Um *p-invariante* de uma RdP é um conjunto de lugares, tal que o número de marcas no conjunto é constante, para toda a marcação alcançável a partir da marcação inicial, e que não tem nenhum subconjunto que é um p-invariante. Os p-invariantes são as soluções y do sistema de equações $\underline{N} \times y = 0$, com a restrição de os elementos de y serem 0 ou 1. Os p-invariantes são pois os conjuntos de lugares correspondentes a 1's nos vectores y.

Um *t-invariante* de uma RdP é um conjunto não vazio de transições que devem disparar para atingir a marcação M_i, a partir da marcação M_i. Os t-invariantes são as soluções inteiras x do sistema de equações $\underline{N}^T \times x = 0$. Os t-invariantes são os conjuntos de transições correspondentes a elementos não nulos nos vectores x.

Os p-invariantes e os t-invariantes podem ser usados para analisar as propriedades estruturais das RdP, ou seja, todas as propriedades que dependem da estrutura topológica e que são independentes da marcação inicial [Mur89].

2.3.3 Técnicas de Redução e Decomposição

Estas técnicas são aplicadas, normalmente, antes de aplicar qualquer um dos dois métodos anteriores. A sua aplicação transforma uma dada RdP, numa outra mais simples (leia-se com menos lugares ou transições), que mantém as propriedades da primeira. Desta forma, a aplicação dos métodos de análise, a realizar posteriormente, torna-se mais simples, o que se traduz num menor tempo de processamento. Estas técnicas tornam-se atractivas, quando a RdP é de dimensões consideráveis. Em sentido inverso, existem técnicas que são utilizadas para transformar um modelo mais abstracto num outro mais detalhado, para efeitos de síntese.

Existem bastantes técnicas de transformação, mas nesta subsecção ir-se-ão apresentar apenas as mais simples. Considerem-se as seis operações ilustradas na figura 2.12, que permitem analisar a RdP, em relação às seguintes propriedades: viva, segura e limitada. Sejam N e N' as RdP, respectivamente, antes e depois de qualquer das operações consideradas. Então N' é

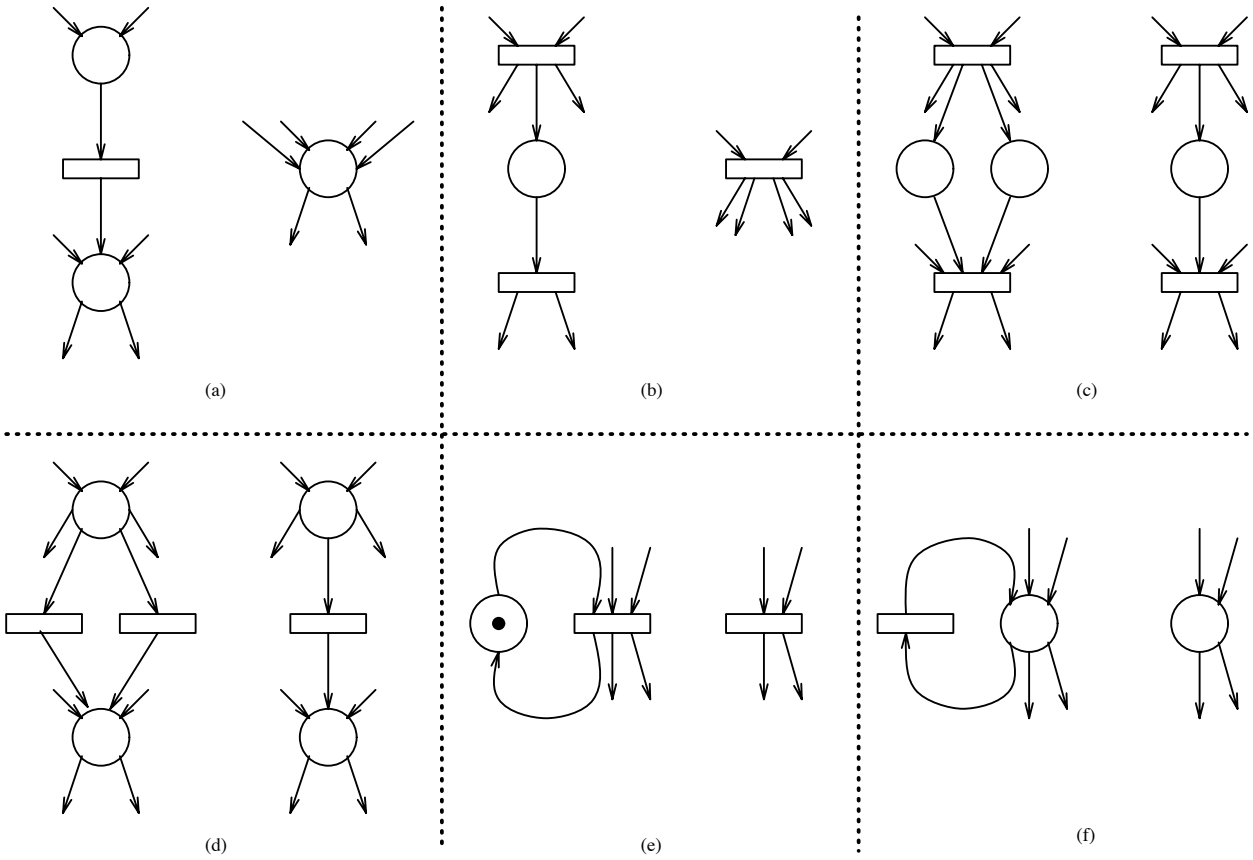


Figura 2.12: Seis operações de transformação.

segura, viva ou limitada, se e só se N também for segura, viva ou limitada, respectivamente.

- Fusão de lugares em série, como ilustrado na fig. 2.12(a).
- Fusão de transições em série, como ilustrado na fig. 2.12(b).
- Fusão de lugares em paralelo, como ilustrado na fig. 2.12(c).
- Fusão de transições em paralelo, como ilustrado na fig. 2.12(d).
- Eliminação de ciclos próprios de lugares, como ilustrado na fig. 2.12(e).
- Eliminação de ciclos próprios de transições, como ilustrado na fig. 2.12(f).

Um dos problemas, já apontados na secção 2.1, reside na complexidade, pelo que se torna importante e útil desenvolver métodos de transformação que permitam reduzir o tamanho da RdP, embora mantendo as propriedades a analisar. Valette [Val79] apresenta uma metologia que permite construir e analisar as propriedades (viva e segura) de uma RdP, caso se substitua uma das suas transições por uma macrotransição. Suzuki e Murata [SM83] generalizam o método anterior e apresentam ainda formas de refinar os lugares de uma RdP através de macrolugares.

2.4 Tipos de Redes de Petri

Nesta secção, pretende-se dar uma panorâmica sobre os diversos tipos de RdP sugeridos por diversos autores e investigadores. Estas variantes surgiram devido às mais variadas razões, nomeadamente pela necessidade de adaptar as características das RdP ao tipo específico de sistemas a modelar.

Felder et al. [FGP93] afirmam que as RdP, por si só, apenas facilitam a especificação dos fluxos de controlo em sistemas complexos. Foi, assim, necessário extender as RdP, em diversas direcções, com o objectivo de as tornar úteis na especificação de outros aspectos como: informação, funcionalidade e tempo.

Algumas das variantes são casos particulares do modelo básico, enquanto que outras são extensões a esse mesmo modelo. Desta forma, a escolha do tipo de modelo, para aplicação numa determinada área, não deve ser extemporânea, mas deve ser feita de uma forma cuidada e atenta.

2.4.1 Modelo Básico

Ao longo desta subsecção, far-se-á referência ao modelo básico das RdP. Por esse modelo, a que também se dá o nome de *Rede Lugar/Transição*, entende-se qualquer RdP que siga a definição 9, dada na secção 2.2.

Dos tipos de sistemas que podem ser modelados com o modelo básico de RdP, pode-se salientar, por exemplo, os sistemas químicos (ver figura 2.5) ou os sistemas produtor–consumidor. A figura 2.13 modela um destes sistemas.

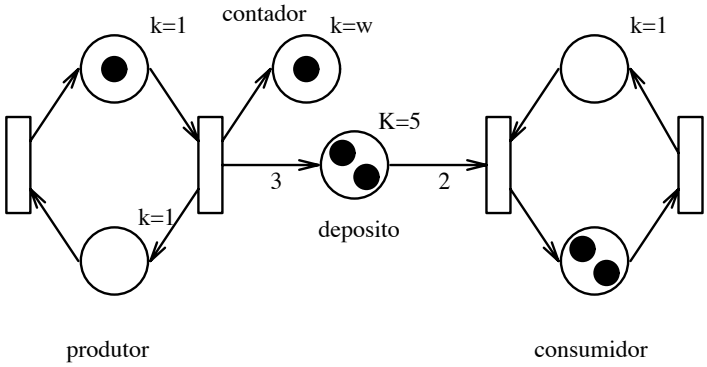


Figura 2.13: Sistema produtor–consumidor com capacidade de armazenamento de 5 unidades.

2.4.2 Vários Tipos

Na bibliografia consultada, encontraram-se diversas versões de RdP. Devido à multiplicidade de campos onde as RdP têm sido aplicadas com sucesso, os diversos autores usam, por vezes, uma taxonomia distinta. Mais problemático ainda é o facto de alguns autores darem o mesmo nome a tipos de RdP diferentes ou darem um nome igual para RdP com características distintas. Assim, existe a possibilidade de os termos que se introduzem, ao longo desta subsecção, para determinados tipos de RdP, não serem únicos.

Não se pretende fazer uma descrição completa e exaustiva de todos os tipos de RdP propostos. Deseja-se, tão somente, mostrar alguns dos tipos mais habitualmente usados e referir, muito sucintamente, as suas características mais relevantes.

RdP Seguras

As *RdP seguras*, alternativamente denominadas por Sistemas Condição/Evento, são casos particulares do modelo básico em que o peso associado a todos os arcos é igual a um ($W : F \rightarrow \{1\}$) e a capacidade de qualquer lugar é também sempre igual a um ($K : S \rightarrow \{1\}$). Como cada lugar só pode ser encontrado em dois estados possíveis (com ou sem marca), é lícito considerar que cada lugar representa uma condição do sistema. Os eventos (i.e. as transições) alteram as condições do sistema (ver quadro 2.1).

Um exemplo típico em que a utilização das RdP seguras se mostra adequada é apresentado na figura 2.14, e permite modelar as estações do ano.

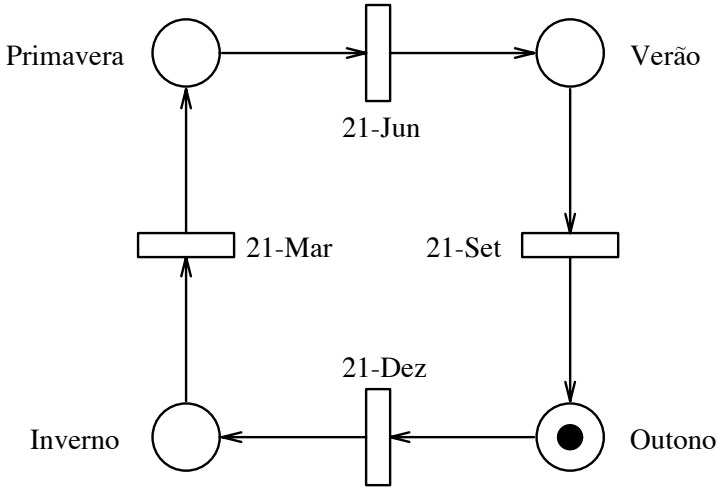


Figura 2.14: As estações do ano.

RdP Temporizadas

Diversos investigadores sentiram a necessidade de introduzir características temporais ao modelo básico de RdP. Várias propostas, no sentido de colmatar essa necessidade, foram apresentadas. As principais alternativas que caracterizam as diferentes propostas são [Mar90]:

- Associar aos nodos (lugares ou transições) da RdP uma temporização;
- A semântica do disparo, no caso de transições temporizadas (disparo atómico ou em três fases);
- A natureza das especificações temporais (determinística ou probabilística).

As *RdP Temporizadas* são extendidas relativamente ao modelo básico, através da associação dum atraso ao disparo das transições. Esse disparo é atómico, ou seja, as marcas

são removidas dos lugares de entrada e colocadas nos lugares de saída numa operação considerada indivisível, atómica. A especificação do atraso de disparo pode ser de natureza determinística (RdP Temporizada Determinística) ou probabilística (RdP Temporizada Estocástica). O atraso de tempo, associado às transições, representa a quantidade de tempo que deve passar, antes de a transição disparar.

O factor principal que limita a aplicabilidade das RdP Temporizadas assenta na complexidade da sua análise. O número de marcações atingíveis é, normalmente, muito elevado. Outro aspecto limitativo reside na presença de actividades (modeladas por transições) com duração de ordem de grandeza diferente (superior ou inferior) relativamente à(s) actividade(s) crítica(s) no desempenho global do sistema. Para obviar esse problema, Ajmone Marsan et al. [MBC84] introduziram as RdP Estocásticas Gerais[7].

Uma RdP Estocástica Geral apresenta dois tipos de transições:

- *Transições temporizadas*: associadas com atrasos probabilísticos, como sucede numa RdP Estocástica.
- *Transições imediatas*: disparam em tempo nulo e com prioridade relativamente às transições temporizadas.

Posteriormente, foram admitidos arcos inibidores (ver subsecção 2.4.3), consideraram-se diferentes níveis de prioridade para as transições imediatas e associaram-se pesos às transições de tipo imediato.

Com o intuito de introduzir características temporais de natureza probabilística, foram, também, feitas propostas nas duas direcções seguintes:

- Modelos de RdP em que as temporizações são associadas aos lugares.
- Modelos de RdP em que o disparo das transições temporizadas é feito em três fases distintas.

Quando o tempo é associado aos lugares, o disparo de uma transição habilitada remove marcas dos lugares de entrada e coloca marcas nos lugares de saída, numa operação atómica. Contudo, as marcas depositadas nos lugares de saída não ficam imediatamente disponíveis para habilitar outras transições. Essas marcas ficam apenas disponíveis, após o tempo associado ao lugar ter transcorrido.

No caso em que a temporização está associada a transições que disparam em três fases, uma transição habilitada dispara imediatamente e remove marcas dos lugares pertencentes ao pré-conjunto da transição em questão. Porém, não são depositadas marcas, nos lugares de saída, enquanto não passar o tempo associado à transição. As marcas permanecem "em transição", até transcorrer o atraso da transição. Dito de outro modo, o disparo de uma transição consiste nas três fases seguintes:

1. Uma fase de *início de disparo*, na qual se removem as marcas dos lugares de entrada;
2. Uma fase de *disparo em progresso*, à qual se associa o atraso de disparo;

[7]Em língua inglesa, *Generalized Stochastic Petri Nets*.

3. Uma fase de *fim de disparo*, na qual se colocam as marcas nos lugares de saída.

Normalmente, as transições temporizadas em três fases podem ser disparadas novamente, enquanto um outro disparo está em progresso.

RdP de Alto Nível

Outra limitação, notada principalmente em certas áreas, prende-se com a indistinção que existe ao nível das marcas. No modelo básico, as marcações são, em cada instante, determinadas apenas pelo número e distribuição das marcas pelos lugares. Para ultrapassar essa limitação, foram propostas várias soluções, todas elas passando pela individualização das marcas. Para tal, a cada marca é associada alguma informação.

Por exemplo, para modelar uma biblioteca com serviço de empréstimo domiciliário, cada livro corresponde a uma marca. A cada marca está associado um registo, contendo informação necessária para descrever o livro (título + autor + editor + ano + ...). Similarmente, os utentes da biblioteca são também modelados através de marcas. A cada marca correspondente a um utente será associado um registo de informação (nome + morada + telefone + bi + ...).

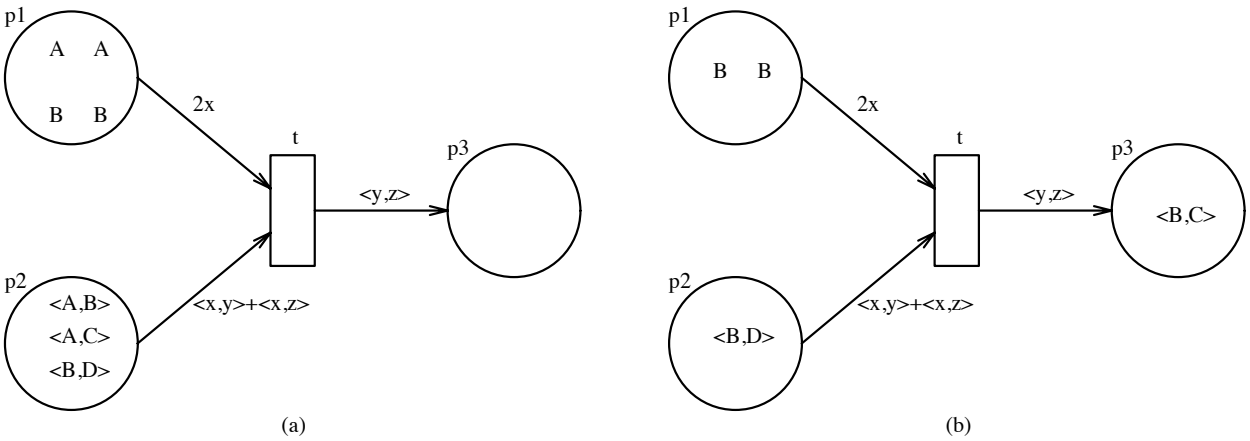


Figura 2.15: Regra de Disparo numa RdP de Alto Nível. (a) Marcação antes da transição t disparar. (b) Marcação após a transição t disparar.

É assim que surgem as RdP de Alto Nível, que incluem, as RdP Coloridas e as RdP com Marcas Individuais. Para ilustrar a regra de disparo das transições de RdP de Alto Nível, considere-se a figura 2.15(a). A RdP aí representada é constituída por três lugares e uma transição. Inicialmente, o lugar *p1* contém quatro marcas coloridas: dois A's e dois B's; o lugar *p2* contém três marcas coloridas (pares ordenados): $< A, B >$, $< A, C >$ e $< B, D >$; e o lugar *p3* não se encontra marcado.

Repare-se que os arcos estão etiquetados. A etiqueta presente num arco indica o número e o tipo de marcas "coloridas" que são removidas ou colocadas, respectivamente, nos lugares de entrada ou saída da transição, sempre que esta disparar. Nessa situação, sucedem os seguintes factos:

- O lugar *p1* perde duas marcas com a mesma cor x;
- O lugar *p2* perde duas marcas de diferentes cores, $< x, y >$ e $< x, z >$;

- O lugar *p3* recebe uma marca com cor $< y, z >$.

Usaram-se letras minúsculas para denotar variáveis, enquanto que para as constantes se utilizam letras maiúsculas.

Os arcos são etiquetados por nomes de variáveis, que representam parâmetros formais. Para uma mesma transição, uma variável com o mesmo nome, presente nos arcos de entrada ou saída, representa a mesma variável. Assim, uma transição diz-se habilitada a disparar se existirem marcas suficientes da cor "certa" em cada um dos lugares de entrada. Entende-se, neste caso, cor "certa" como a existência de uma substituição consistente das variáveis por constantes, de acordo com as etiquetas dos arcos. Por exemplo, a transição t está habilitada a disparar, pois há suficientes marcas nos lugares de entrada e fazendo $x = A, y = B, z = C$ obtém-se uma substituição consistente.

A marcação resultante do disparo da transição t é ilustrada na figura 2.15(b).

Uma aplicação com sucesso desta versão das RdP para implementação de um *Cascadable Nacking Arbiter* pode ser encontrada em [GS93]. Uma das variantes às RdP de Alto Nível, referenciada em [ELP93, FGP93, PALD93], tem por nome RdP Temporizada de Alto Nível[8]. A aplicação deste tipo de RdP tem-se revelado adequada na especificação de sistemas de tempo real.

2.4.3 Arcos Habilitadores e Inibidores

Em alguns sistemas, a possibilidade de descrever uma situação de prioridade ou de sincronização entre dois ou mais subprocessos quase independentes é necessária. Essa situação é frequente, quando os subprocessos partilham determinados recursos. A solução para esse problema passa por considerar a existência de arcos habilitadores e inibidores.

Uma prioridade pode ser modelada com o auxílio de uma arco inibidor, representado graficamente por uma linha a tracejado com uma pequena circunferência junto da transição.

Como exemplo ilustrativo, considere-se o comportamento de um sistema, modelado pela RdP apresentada na figura 2.16. Esse sistema é composto por dois subprocessos que partilham um mesmo recurso. O subprocesso A tem prioridade relativamente ao subprocesso B, no acesso ao referido recurso. Um conflito surge, quando ambos os subprocessos pretendem acesso simultâneo ao recurso. Nessa situação, o acesso deve ser atribuído ao subprocesso A e, após a utilização por parte deste, o acesso é dado ao subprocesso B. Um arco inibidor liga um lugar a uma transição e não permite que a transição dispare, ou seja, inibe o seu disparo, se o lugar contém uma marca. A introdução de arcos inibidores aumenta o poder de modelação, ao permitir testar se um lugar está vazio, sem marcas.

Por seu lado, o uso de arcos habilitadores permite modelar uma situação onde um dado evento só pode suceder se o sistema tiver atingido um determinado estado.

Como exemplo ilustrativo, considere-se o sistema representado na figura 2.17. Nesse sistema, os dois subprocessos quasi-independentes são sincronizados através dum arco habilitador, representado por uma linha a tracejado com uma seta junto à transição. O subprocesso B pode prosseguir quando o subprocesso A atinge o estado A1.

[8]Em língua inglesa: HLTPN, iniciais de *High Level Timed Petri Net.*

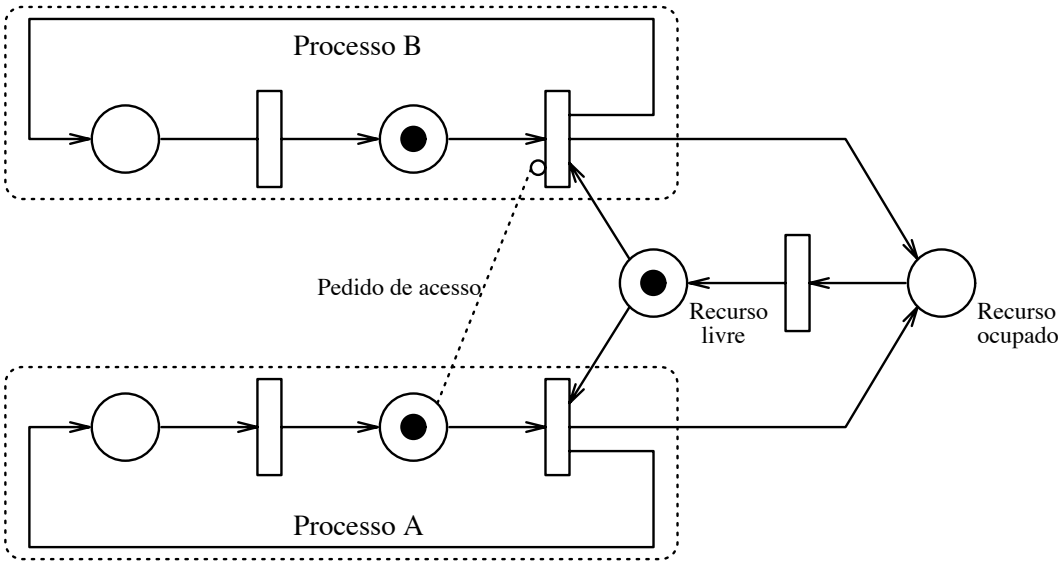


Figura 2.16: Sistema onde a prioridade entre dois subprocessos é modelada através de um arco inibidor.

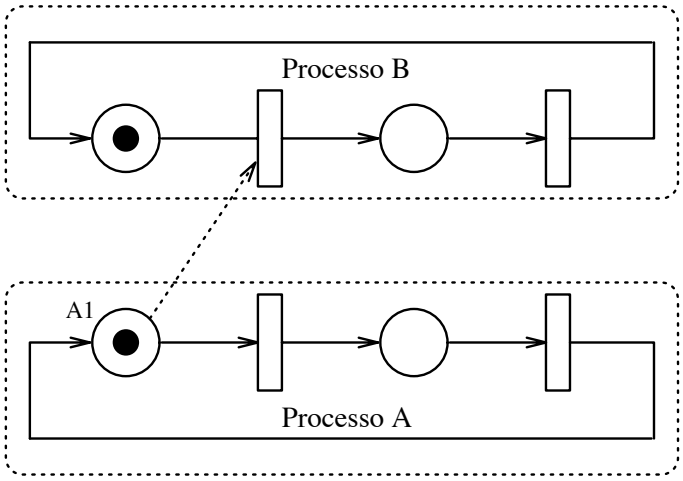


Figura 2.17: Sistema onde a sincronização entre dois subprocessos é modelada através de um arco habilitador.

Um arco habilitador liga um lugar a uma transição e habilita esta quando o lugar contém uma marca. Contudo, quando a transição dispara, a marca presente no lugar ligado à transição pelo arco habilitador não é removida. Quando o lugar não contém qualquer marca, a transição não está habilitada a disparar.

2.5 Selecção de um Tipo de Rede

A primeira decisão a tomar neste trabalho incidiu sobre a escolha do tipo de RdP mais adequado, para modelação de controladores paralelos.

Se for seleccionado um tipo forte demais, embora a modelação seja possível, a sua análise torna-se mais onerosa e difícil. Se, por outro lado, o tipo escolhido for fraco de mais, então nem sequer é possível modelar a totalidade dos sistemas pretendidos.

A escolha do tipo de RdP recaiu nas RdP seguras, visto que, segundo Peterson [Pet81], uma das propriedades mais importantes que uma RdP deve obedecer, para modelar um sistema digital, é a segurança. Se um lugar é seguro, então o número de marcas que pode conter é 0 ou 1. Assim, esse lugar pode ser, directamente, implementado por um *flip-flop*. A segurança permite que um lugar seja implementado por um *flip-flop*, mas, mais genericamente, um contador pode ser usado, se a RdP for limitada[9]. Por outro lado, a utilização das RdP seguras mostra-se uma extensão natural às máquinas de estado, sendo portanto fácil a transição destas para aquelas. É também possível com RdP seguras modelar as máquinas de estado sequenciais, pois um controlador sequencial é um caso particular de um controlador paralelo. Uma última vantagem resulta do facto de a análise das RdP seguras ser mais fácil se comparada com outros tipos (RdP Alto Nível ou RdP Temporizadas, por exemplo).

No entanto, sobre as RdP seguras, há que proceder a algumas alterações, para que aquelas possam efectivamente modelar qualquer controlador paralelo. A primeira alteração reside na colocação de condições (expressões lógicas formadas por variáveis de entrada, a que se chama *guardas*) nas transições [AVD76, FM91]. A estas transições pode-se dar o nome de *transições guardadas*. A esta versão de RdP, chamar-se-á *Redes de Petri Interpretadas* (abreviadamente RdPI).

As guardas associadas às transições fazem alterar a regra que define se uma transição de uma RdPI segura está ou não habilitada:

Regra 1 (Habilitação de uma transição guardada) *Uma transição duma RdPI segura diz-se habilitada a disparar, se se verificam os três factos seguintes: cada um dos seus lugares de entrada tem uma marca; cada um dos seus lugares de saída não contém qualquer marca e; a guarda associada a essa transição é verdadeira.*

Adicionalmente, para representar as acções do controlador, tem de se associar os sinais de saída aos lugares ou transições, onde eles devem ser activados [AVD76]. Um sinal de saída associado a um lugar representa um sinal do tipo Moore, enquanto que se a associação for entre uma transição e um sinal, este é do tipo Mealy [PB91].

Graficamente, ao longo deste trabalho, as transições e os lugares serão decorados da forma apresentada na figura 2.18. De notar que *cond* é uma expressão lógica das variáveis de entrada e que *instr* é uma lista dos sinais de saída que devem ser activos. Quando a guarda a associar a uma transição for uma tautologia (sempre verdadeira), não é necessária a sua indicação.

Cada lugar da RdPI é visto como um estado de controlo. O estado total do controlador é determinado pela marcação da RdP (distribuição das marcas pelos lugares). Como se limita a capacidade de cada lugar à unidade, um lugar, quando contém uma marca, representa um estado activo e, por conseguinte, cada um dos sinais de saída a ele associado é activado. Como é possível que dois ou mais lugares tenham simultaneamente uma marca, o sistema modela operações em paralelo, de um modo directo e dinâmico.

Um dos problemas que as RdP exibem no seu comportamento assenta no assincronismo, característica que não é de manipulação trivial em sistemas digitais, embora já haja alguma investigação no desenho de circuitos assíncronos. Para contornar os problemas que resultam do assincronismo das RdP, houve que proceder, novamente, a algumas alterações ao

[9] Se um lugar é $k-$*limitado*, então pode ser implementado por um contador com, pelo menos, $\log_2 k$ bits.

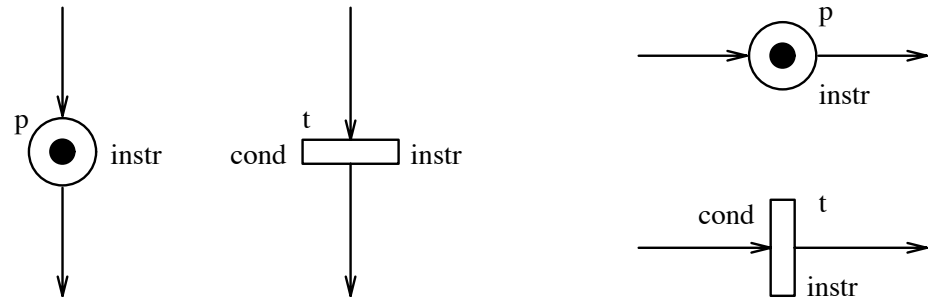


Figura 2.18: Decoração de lugares e transições de uma RdPSI. Desenho realizado (a) na vertical e (b) na horizontal.

comportamento habitual das RdP. No presente trabalho, assumiu-se que o controlador, ao nível mais alto, trabalha sincronamente.

A solução adoptada assenta na seguinte filosofia: as transições habilitadas num dado instante, não disparam instantaneamente, mas mantêm-se habilitadas até que surja um pulso (uma transição activa) no sinal de relógio. Quando a transição no sinal de relógio surge, todas as transições habilitadas disparam e uma nova marcação é calculada. A uma RdPI que segue este princípio chama-se *Rede de Petri Síncrona Interpretada* (abreviadamente RdPSI). Repare-se que nas RdP tradicionais, apenas dispara, em cada instante, uma das várias transições habilitadas a fazê-lo.

Assim, foi adoptada a seguinte regra, definida em [PKSB92], e que pressupõe ou permite pressupor a existência de um sinal de relógio ou sincronismo.

Regra 2 (Disparo Simultâneo) *Todas as transições habilitadas, num dado instante, disparam simultaneamente.*

É porém necessário garantir que em qualquer marcação alcançável, todas as transições habilitadas não estão em conflito. Atente-se, na figura 2.19(a). Se as transições *t1* e *t2* estiverem ambas habilitadas, no instante de sincronismo, há um conflito, visto que essas transições partilham um lugar de entrada. Uma segunda situação, a ter em conta, encontra-se ilustrada na figura 2.19(b). Nesta situação, a problema reside no facto de as transições *t3* e *t4* partilharam um lugar de saída, que será duplamente marcado, se as duas transições estiverem habilitadas, no mesmo pulso do sinal de relógio, o que viola a segurança pretendida para as RdPSI.

Suzuki [Suz84] introduziu o conceito de disparo simultâneo e afirma que este é útil na modelação e análise de sistemas em que ocorra, pelo menos, uma das duas seguintes situações:

- Os componentes do sistema requerem o mesmo tempo de execução;
- Os componentes do sistema operam sincronamente, regulados por pulsos de relógio.

É precisamente a segunda característica que interessa para a modelação de controladores paralelos, os quais, no presente trabalho, têm de ser, obrigatoriamente, síncronos.

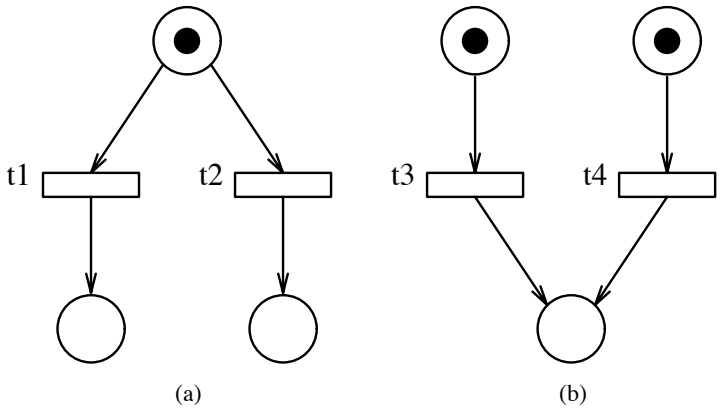


Figura 2.19: Possíveis conflitos numa RdPSI.

Adicionalmente, serão admitidos arcos habilitadores e inibidores. Repare-se, finalmente, que uma RdP segura não pode conter ciclos próprios, pois se tal sucedesse essa transição nunca dispararia — ver definição 11 e figura 2.20.

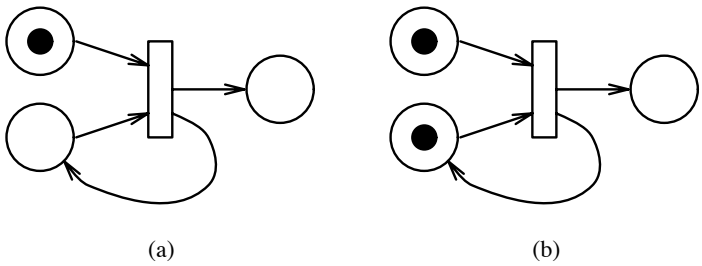


Figura 2.20: Ciclos próprios numa RdP segura.

2.5.1 Propriedades das RdPSI

As RdPSI que dão suporte aos controladores paralelos devem ser vivas, seguras, determinísticas e livre de conflitos [PB91, AVD76].

Uma RdP é *viva* se é possível, a partir de qualquer marcação, habilitar todas as transições através de uma dada sequência de disparo. Este facto assegura a inexistência de situações de bloqueio e código morto (transições nunca habilitadas ou lugares nunca marcados). O facto de não existir código morto garante que não haverá desperdício de silício na implementação final.

Uma RdP diz-se *segura* se cada lugar não contém mais do que uma marca, em cada instante. Se assim for, os lugares podem ser implementados por *flip-flops*. As transições numa RdP segura só podem disparar se os seus lugares de saída estiverem vazios, pelo que a segurança garante que as operações na unidade de dados não podem ser re-invocadas, antes de terem terminado completamente.

Uma RdP *determinística* garante que os recursos disponíveis na unidade de dados não podem ser simultaneamente partilhados ou invocados por mais do que um nodo da RdP. Deste modo, para cada marcação e conjunto de transições habilitadas, os sinais de controlo gerados devem ser mutuamente exclusivos.

Uma RdP é *livre de conflitos* se para todo o lugar marcado, não mais do que uma das

suas transições de saída está habilitada, em qualquer momento. Este facto é de grande importância, pois permite que o controlador se comporte de uma forma pré-definida e não arbitrária. A utilização de predicados nas transições permite resolver alguns conflitos. Por exemplo, o conflito existente na figura 2.19(a) pode resolver-se, guardando a transição *t1* com o predicado X e a transição *t2* com o predicado $\overline{X}$. Como estes predicados são mutuamente exclusivos (a sua conjunção $X \wedge \overline{X}$ dá como resultado falso), apenas uma das transições poderá estar habilitada, num dado momento.

2.5.2 Exemplo de uma RdPSI

Para se entenderem melhor as duas regras apresentadas nesta secção e o efeito provocado pela introdução de arcos habilitadores e inibidores, apresenta-se na figura 2.21 uma RdPSI, a qual servirá de exemplo, ao longo desta subsecção. Usou-se, nas figuras deste trabalho, o símbolo ' para indicar a negação de uma expressão booleana.

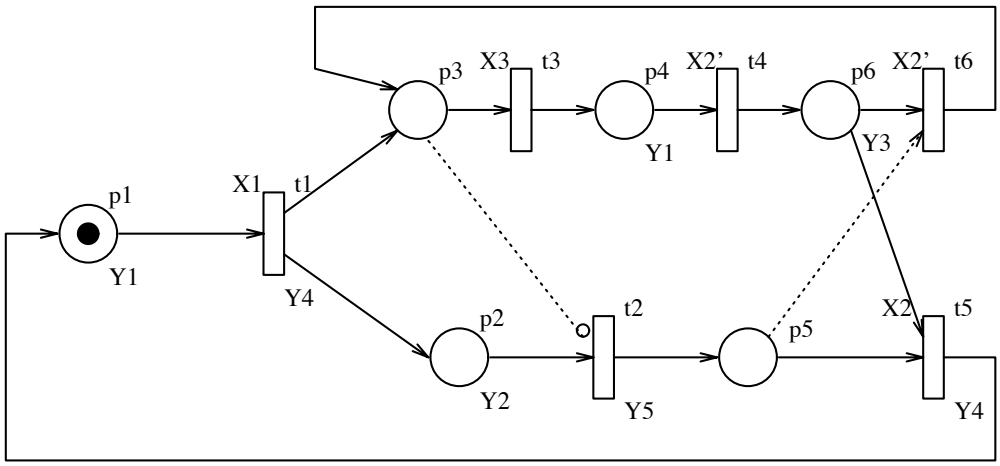


Figura 2.21: Exemplo de uma RdPSI.

No quadro 2.2, encontram-se todos os sinais envolvidos na descrição do sistema que a RdPSI pretende modelar. São esses sinais que permitem dar uma interpretação à RdP.

Sinais de Entrada	Sinais de Saída	
	Moore	Mealy
X1, X2, X3	Y1, Y2, Y3	Y4, Y5

Quadro 2.2: Sinais de Entrada e de Saída.

No presente exemplo, o vector $\{M(p1), M(p2), M(p3), M(p4), M(p5), M(p6)\}$ é usado para representar cada marcação. A partir da marcação inicial {1,0,0,0,0,0} verifica-se que a única transição potencialmente habilitada é *t1*. Essa transição está apenas potencialmente habilitada, já que se trata de uma transição guardada. Assim, só conhecendo o valor da variável de entrada *X1* se pode afirmar se a transição está ou não habilitada a disparar. Adimita-se que o valor de *X1* é, a partir de um dado momento igual a '1'. A variável de saída *Y4* é "imediatamente" activada, pois trata-se de uma saída do tipo Mealy. Quando surge uma transição activa no sinal de relógio, a transição *t1* dispara. O disparo dessa

transição retira a marca presente no lugar *p1* e coloca uma marca no lugar *p2* e outra no lugar *p3*, resultando a marcação {0,1,1,0,0,0} — ver figura 2.22.

Na marcação {0,1,1,0,0,0} a única transição potencialmente habilitada é *t3*, pois o seu único lugar de entrada *p3* contém uma marca e o seu lugar de saída *p4* está vazio. Por seu lado, a transição *t2* não está habilitada, uma vez que o lugar *p3* não está vazio. Dada a existência de um arco inibidor entre *p3* e *t2*, esta transição só pode disparar se *p3* não contiver qualquer marca.

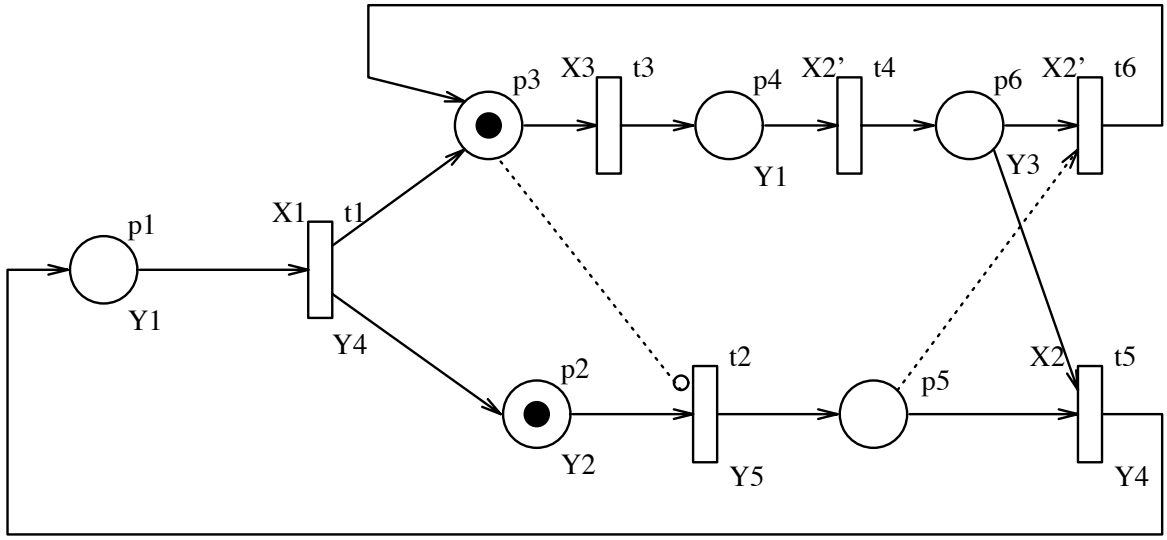


Figura 2.22: RdPSI após disparo da transição t1.

O disparo da transição *t3* far-se-á quando a variável *X3* for igual a '1', resultando a marcação {0,1,0,1,0,0} apresentada na figura 2.23.

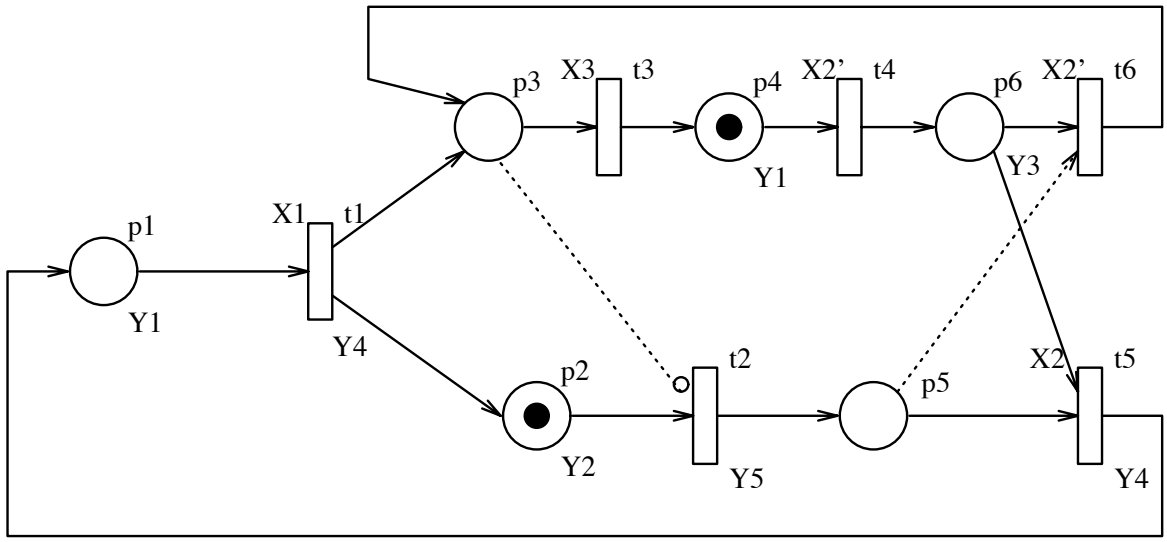


Figura 2.23: RdPSI após disparo da transição t3.

Nesta marcação, a transição *t2* está habilitada a disparar, pois a sua guarda é incondicionalmente verdadeira, o lugar *p2* está marcado, e os lugares *p3* e *p5* estão vazios. A transição *t4* está ou não habilitada, consoante o valor de *X2* seja '0' ou '1', respectivamente. Assim, se *X2* for igual a '1', na próxima transição activa no sinal de relógio, a única transição que

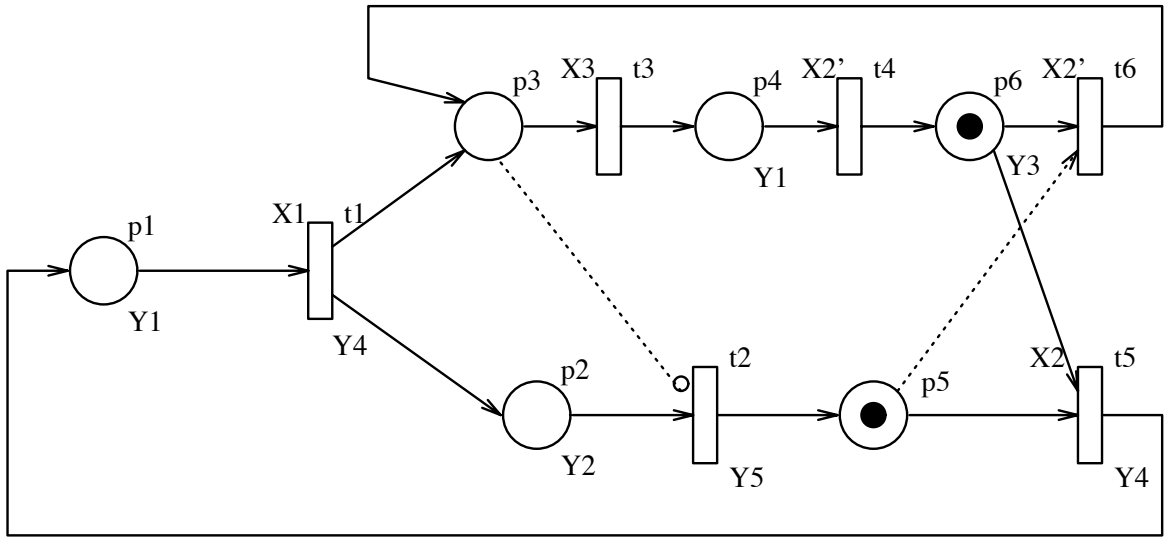


Figura 2.24: RdPSI após disparo simultâneo das transições t2 e t4.

dispara é *t2*. Se *X2* for '0', então, quando surgir um pulso no sinal de relógio, as transições *t2* e *t4* disparam em simultâneo (cf. regra 2).

Considerando esta última hipótese, apresenta-se a marcação resultante na figura 2.24. Neste caso, haveria, à partida, uma situação de conflito envolvendo as transições *t5* e *t6*. No entanto, como as suas guardas são disjuntas, essa situação desaparece. Consoante o valor da variável de entrada *X2*, apenas uma dessas transições está habilitada a disparar. Se *X2* for igual a '0', a transição *t6* está habilitada, e o seu disparo retira uma marca do lugar *p6* e coloca outra no lugar *p3* — ver figura 2.25. Repare-se que é necessária a presença de uma marca no lugar *p5*, mas aquela mantém-se após o disparo da transição. Este facto deriva da existência de um arco habilitador entre o lugar *p5* e a transição *t6*.

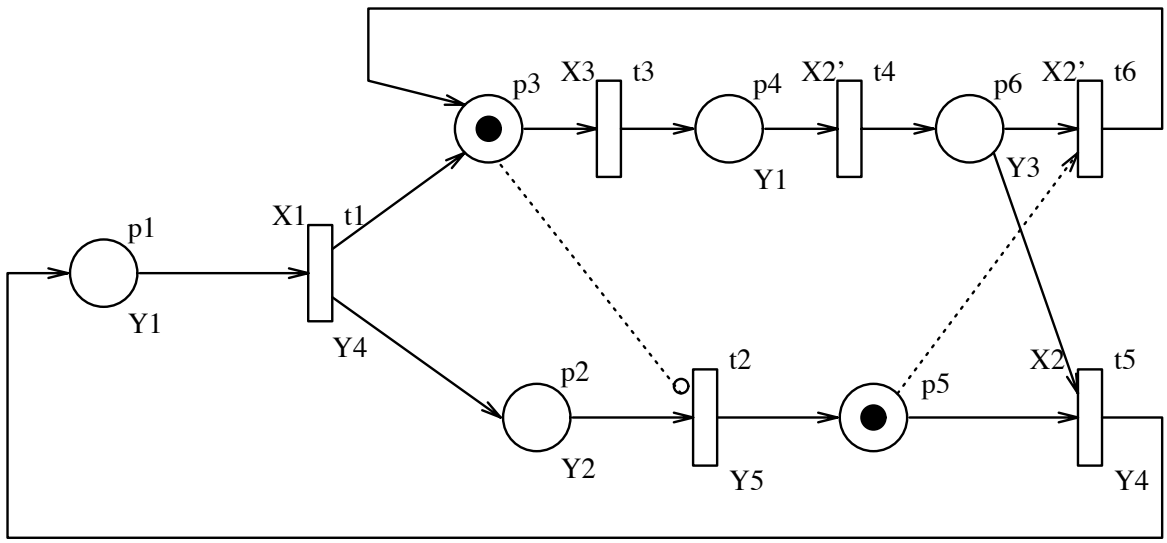


Figura 2.25: RdPSI após disparo da transição t6.

Com este exemplo, verifica-se que é possível dispararem duas ou mais transições em simultâneo e que as guardas permitem resolver situações de conflito, o que não aconteceria numa RdP, seguindo o modelo básico.

Capítulo 3

VHDL

"The VHSIC Hardware Description Language is an industry standard language used to describe hardware from the abstract to the concrete level."

Douglas L. Perry in "VHDL"

3.1 Caracterização da Linguagem

3.1.1 Introdução

Convencionalmente, o desenho de um projecto de sistemas de digitais electrónicos é obtido através de uma série de diagramas esquemáticos. Embora esta técnica esteja bem definida e a sua utilização seja conveniente para descrições ao nível da porta lógica, para sistemas mais complexos é necessário recorrer-se a uma linguagem de descrição de *hardware* (HDL – Hardware Description Language) [KBK+90].

As HDL são usadas no projecto de sistemas digitais, permitindo a sua concepção com uma notação de especificação clara, bem como a simulação, modelação e teste de soluções. A funcionalidade dos sistemas assim projectados pode ser representada hierarquicamente, de forma compacta e adequada, usando as HDL.

Contudo, durante a concepção de sistemas digitais de relativa complexidade, muito do tempo que se perde resulta da conversão entre formatos, por forma a usar diversas ferramentas.

Num ambiente ideal, a descrição de alto nível do sistema é facilmente legível por todos os participantes no projecto e define de forma precisa o *hardware*. À medida que o projecto prossegue, mais detalhes vão sendo acrescentados à descrição inicial, o que permite a simulação e o teste do sistema a vários níveis de abstracção. No último passo do projecto, a descrição inicial deu origem a uma descrição muito mais detalhada, que serve de entrada a um programa que sintetiza o sistema descrito.

Este processo existe, se estiver disponível uma HDL que permita descrever *hardware* a diferentes níveis, de molde a ser legível e perceptível para pessoas (utilizadores, gestores,

programadores) e para máquinas (programas, simuladores, compiladores) [Nav93]. Essa linguagem existe e dá pelo nome de VHDL. Foi precisamente a necessidade de um ambiente integrado de projecto e de uma linguagem de documentação para comunicar a diferentes níveis que ditou a criação de VHDL.

VHDL (VHSIC Hardware Description Language) surgiu como resultado do programa "Very High Speed Integrated Circuit" (VHSIC), organizado pelo Departamento de Defesa (DoD) dos E.U.A., nos inícios dos anos 80. Ao longo do decorrer desse programa, tornou-se clara a necessidade de uma linguagem normalizada para descrever a estrutura e a funcionalidade de circuitos integrados. É, dessa forma, que é criada e desenvolvida a linguagem VHDL [Per91].

O DoD, aquando do processo de criação da linguagem, elaborou um documento, onde se indicavam todas as características a que VHDL deveria obedecer. O primeiro ponto indicava que VHDL deveria ser utilizada para documentação, concepção de alto nível, simulação, síntese e teste de sistemas digitais, bem como servir de entrada para uma ferramenta de desenho ao nível físico. As restantes características deveriam suportar o seguinte:

- Concepção hierárquica e parametrizável;
- Utilização de bibliotecas;
- Existência de comandos sequenciais e concorrentes;
- Definição de tipos de dados;
- Existência de subprogramas (funções e procedimentos);
- Controlo temporal;
- Especificação estrutural.

Em 1986, VHDL foi proposta como norma IEEE, tendo finalmente sido aceite como tal (*standard* IEEE 1076-1987), após uma série de revisões e alterações, em Dezembro de 1987.

A construção da linguagem VHDL foi fortemente influenciada pela linguagem ADA [Led81]. Muitos dos conceitos foram emprestados por esta linguagem, entre os que se incluem [FS89]:

- Especificação separada de interfaces e implementações;
- Pacotes e a possibilidade de deles importar selectivamente declarações;
- Bibliotecas de programas e compilação separada;
- Tipos definidos pelo utilizador;
- Funções polimórficas;
- Declarações de funções aninhadas;
- Constantes enumeradas polimórficas;
- Vectores dinamicamente restringidos;

- Notação compacta para especificar vectores e estruturas em termos dos seus componentes.

Adicionalmente, VHDL acrescenta mais construtores e características para suportar aspectos relacionados directamente com *hardware*, que não são considerados numa linguagem de programação de âmbito geral [FS89]:

- Novas unidades de compilação: `entity` (interface de um dispositivo), `architecture` (corpo de um dispositivo) e `configuration` (implementação concreta de um dispositivo);
- Regras por defeito para configurar uma arquitectura, se não for explicitamente dada uma configuração;
- Atributos definíveis pelo utilizador e especificação dos seus valores iniciais;
- Objectos relacionados com sinais: atribuição, funções de resolução;
- Sinais e comandos guardados;
- Sincronização do tempo de simulação;
- Processos e respectivo escalonamento e comandos de espera (`WAIT`).

3.1.2 Níveis de Abstracção

Segundo Bergé et al. [BFMR92], a forma como o projectista entende um sistema electrónico varia de acordo com as fases de projecto e com o uso pretendido. Basicamente, distinguem-se três perspectivas: especificação, concepção e documentação.

A **especificação** define o interface e as características funcionais do sistema e inclui três aspectos principais:

1. O interface, que define a comunicação com o exterior;
2. O comportamento do sistema, que é feito independentemente da implementação;
3. As restrições, que podem condicionar a escolha de uma arquitectura adequada para o sistema.

Quando se projecta um sistema digital através de ferramentas automáticas de síntese, deve-se garantir que a especificação é aceite pela ferramenta. Tal facto pode implicar restrições no uso de alguns construtores da linguagem VHDL.

A **concepção** implementa uma descrição comportamental em termos de componentes interligados. Os componentes podem ser tirados de uma biblioteca existente ou especificados para futura concepção.

Uma HDL não deve estar apenas vocacionada para permitir comunicação entre o homem e a ferramenta CAD [BFMR92]. É também importante que possibilite a transferência de informação entre plataformas de diferentes fabricantes (comunicação entre ferramentas) e

```
process
  variable RESULT:INTEGER:=0;
begin
  wait on A, B, Cin;
  RESULT := BIT2INT(A)+BIT2INT(B)+BIT2INT(Cin);
  S <= BOOL2BIT((RESULT MOD 2)=1);
  Cout <= BOOL2BIT(RESULT>1);
end process;
```

Texto 3.1: Somador ao nível comportamental.

entre projectistas (comunicação entre homens). Resumindo, deve facilitar a **documentação** de um projecto.

VHDL como HDL pode ser usada para descrever qualquer das perspectivas do sistema atrás mencionadas, sendo recomendável uma noção prévia do vasto leque de níveis, estilos de descrição e domínios de aplicação disponibilizados pela linguagem [BFMR92].

A classificação das descrições de sistemas *hardware* é habitualmente baseada na informação que elas contêm. É usual considerar os seguintes níveis de abstracção: comportamental, fluxo de dados[1] e estrutural [BFMR92, Nav93]. Além disso, um modelo em VHDL pode combinar os três estilos atrás descritos, numa única descrição [BFMR92].

3.1.3 Descrição Comportamental

Os modelos descritos ao nível comportamental representam as transformações nos dados de entrada que produzem as saídas dos blocos, sem representar exactamente o fluxo de dados interno aos blocos. A este nível, os tipos de dados são, geralmente, muito mais abstractos, se comparados com os utilizados ao nível do hardware (bits e seus vectores) [Bol91].

Este tipo de descrição define a funcionalidade de um dado sistema ou dispositivo através de um algoritmo (programa) sequencial, sem referência a qualquer implementação estrutural. Este nível de descrição mostra-se apropriado para verificação e simulação de ideias e para documentação.

Um somador, por exemplo, será modelado por uma operação de adição. Se `A`, `B`, `Cin`, `S` e `Cout` são sinais (algures definidos) e se estão definidas as funções de conversão `BIT2INT` e `BOOL2BIT`, então pode-se representar o somador esquematicamente (figura 3.1) ou por uma descrição comportamental usando código VHDL (texto 3.1).

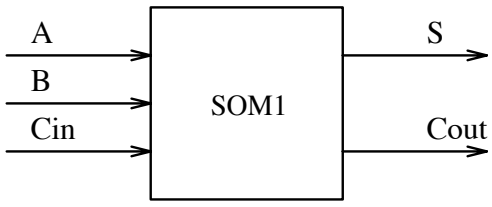


Figura 3.1: Somador completo de 1 bit funcional.

[1] Em terminologia anglo-saxónica, também se usam as iniciais RTL de *Register-Transfer Level.*

```
temp <= a XOR b AFTER 10ns;
s <= temp XOR cin AFTER 10ns;
cout <= (a AND b) OR (temp AND cin) AFTER 20 ns;
```

Texto 3.2: Somador ao nível fluxo de dados.

3.1.4 Descrição Transferência de Registos

Numa descrição fluxo de dados ou RTL, a atribuição de sinais é o construtor principal [Nav93]. O sistema é representado por um conjunto de equações concorrentes. Cada uma dessas equações envolve funções definidas pelo projectista e operadores lógicos e aritméticos que manipulam sinais de tipos arbitrariamente complexos. Essas equações expressam o fluxo da informação (dados) através dos módulos funcionais RTL, que as funções e os operadores implicitamente determinam. A comunicação entre os diversos elementos do sistema é conseguida através de sinais e barramentos.

O nível de detalhe do *hardware* presente em descrições fluxo de dados é suficiente para permitir identificar e descrever os componentes a sintetizar. As descrições fluxo de dados implicam uma arquitectura e uma implementação única. É possível encontrar uma implementação directa em *hardware*, mapeando os sinais em fios (ou *latches*) e os operadores RTL em módulos RTL.

A simulação destas descrições envolve o movimento dos dados pelos registos, *latches* e barramentos, e portanto é mais lenta que a obtida com descrições ao nível comportamental.

As descrições RTL são de alto nível (leia-se legíveis por um humano) e permitem que um processo automático se encarregue da sua conversão para a tecnologia pretendida, ou dito de outro modo, as descrições a este nível são comportamentais (os operadores detêm um dado significado) e implicam uma dada estrutura em termos dos módulos RTL.

Notando que **AND**, **XOR**, **OR** são operadores pré-definidos da linguagem, o somador pode ser representado pela figura 3.2 e por uma descrição comportamental em linguagem VHDL (texto 3.2).

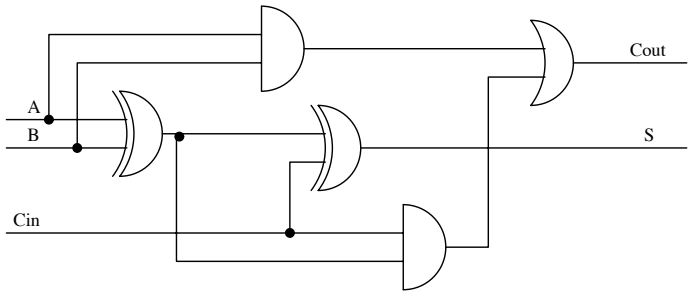


Figura 3.2: Descrição RTL.

3.1.5 Descrição Estrutural

Um circuito digital electrónico pode ser descrito como um módulo com entradas e saídas. Os valores eléctricos nas saídas são uma função dos valores das entradas. Uma forma possível de descrever o funcionamento de um sistema eléctrico é indicar como ele é composto [Ash90].

```
COMP1: SEMISOM port map (A,B,T1,T2);
COMP2: SEMISOM port map (T2,Cin,T3,S);
COMP3: OR_GATE port map (T1,T3,Cout);
```

Texto 3.3: Somador ao nível estrutural

Uma descrição estrutural lista os componentes do sistema e respectivas interligações. A funcionalidade dos componentes não é evidente (não faz parte da descrição), ou seja, os componentes são vistos como caixas negras com um determinado interface bem definido. O sistema resultante corresponde a um conjunto de peças interligadas. A funcionalidade dos componentes do sistema é definida noutro local, usando outros construtores da linguagem.

Todavia, o *hardware* é claramente descrito e um "simples" programa pode facilmente gerá-lo. O somador considerado, poderia ser descrito considerando a interligação de dois semi-somadores e uma porta lógica. O respectivo código em VHDL encontra-se no texto 3.3.

O respectivo diagrama é apresentado na figura 3.3, onde são usadas caixas negras sem qualquer semântica associada. Os nomes `SEMISOM` ou `OR_GATE` são identificadores arbitrários sem qualquer significado especial para a linguagem. Contudo, têm um determinado significado para os projectistas, pelo que a sua escolha deve ser feita com cuidado por forma a aumentar a legibilidade da descrição.

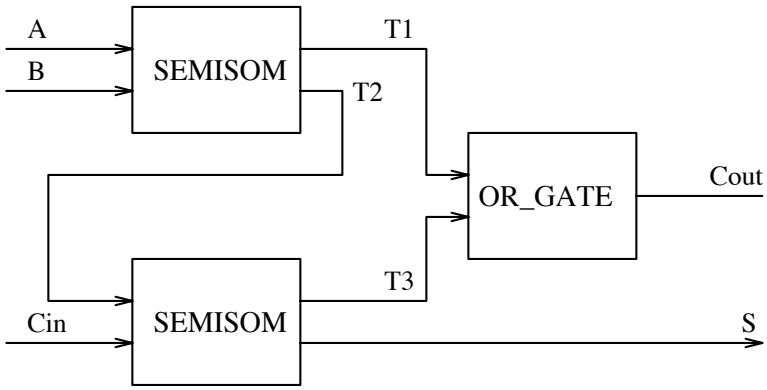


Figura 3.3: Descrição Estrutural.

Ao nível estrutural, os modelos representam blocos como um sistema hierarquizado de *netlists*. Geralmente, não se pretende escrever modelos puramente estruturais [Bol91].

3.1.6 Construtores VHDL

Nesta secção, pretende-se, de uma forma sucinta, apresentar alguns dos construtores da linguagem VHDL. A utilização dos construtores será feita na secção 3.2. Não cabe neste capítulo a apresentação de toda a linguagem VHDL, o que se tornaria fastidioso, dada a grandiosidade da linguagem. Para um conhecimento mais detalhado e aprofundado sobre VHDL, deve-se consultar a bibliografia referenciada ao longo do capítulo.

Sinais e Variáveis

VHDL admite, como qualquer linguagem de programação, variáveis mas também sinais. Estes representam a principal forma de dados e são usados para interligar os diversos componentes presentes num dado modelo [Per91]. Uma variável é definida como um objecto com um dado valor actual, ao passo que os sinais detêm uma história de valores. A esta história de valores dá-se usualmente o nome de "forma de onda". De uma forma simplista, pode entender-se os sinais como valores presentes em fios ou barramentos.

Uma declaração de uma lista de sinais de um dado tipo segue a seguinte sintaxe:

SIGNAL lista_sinais **:** tipo [**:=** valor_inicial]**;**

Como exemplo tome-se:

```
SIGNAL a : BIT := '1';
SIGNAL atraso, inicio : TIME;
```

Os sinais têm significância ao nível físico e têm uma componente temporal associada. O símbolo de atribuição para sinais é `<=`. Considere-se o seguinte exemplo:

```
a <= b;
```

Esta instrução tem a seguinte leitura: "a recebe o valor de b". O efeito da instrução é atribuir ao sinal `a` o valor actual do sinal `b`. A execução desta atribuição ocorre sempre que o sinal `b` mudar de valor.

O escalonamento para atribuição ao sinal do valor da expressão presente no lado direito da atribuição pode ser especificado usando a claúsula `AFTER`. Exemplo:

```
a <= b AFTER 7ns;
```

Esta instrução tem a seguinte leitura: "a recebe o valor de b depois de passarem 7 nanosegundos".

Dois conceitos importantes que estão relacionados com atribuições de sinais são: *eventos* e *transacções*.

Quando um determinado valor está escalonado para atribuição a um sinal após um dado período de tempo, diz-se que uma transacção ocorre. Uma transacção que não faz alterar o valor do sinal continua a ser uma transacção, mas não causa um evento no sinal. Um evento ocorre num sinal sempre que o seu valor se altera.

Uma transacção é representada por um par valor-tempo [Nav93]. Esse valor é o actual se a componente temporal for zero; caso contrário esse valor é um valor projectado no futuro, incluído no *driver* do sinal.

Associado a cada sinal existe um *driver*, que pode ser visto como um projecto para a "forma de onda" do sinal. O *driver* é uma lista de transacções , ou seja, uma lista de pares valor-tempo. O tempo indica o instante no qual o valor do sinal em causa é alterado para o novo

valor indicado. O *driver* contém sempre, pelo menos, uma transacção, que é inicialmente dada pelo valor por defeito especificado para o sinal, e também tem sempre uma transacção que indica o valor actual do sinal.

Todas as atribuições feitas a sinais, no corpo de uma arquitectura, são concorrentes. É possível ter várias atribuições a sinais no corpo de uma arquitectura. Essas atribuições estão simultaneamente activas e a ordem por que são escritas não é relevante. Este facto apresenta, contudo, algumas limitações. Só é possível fazer várias atribuições concorrentemente a um mesmo sinal, se for especificada uma função — dita de resolução — para calcular, a partir dos vários valores atribuídos, um valor único. A esses sinais chama-se *sinais resolvidos*.

As atribuições feitas a sinais em construtores sequenciais VHDL (por exemplo: um processo) são realizadas sequencialmente. Neste caso, a ordem pela qual são escritas as atribuições é importante.

Para as variáveis não foi associada a tal componente temporal, ou seja, qualquer atribuição ocorre imediatamente. A sua utilização é normalmente para guardar resultados intermédios (em descrições comportamentais).

Uma declaração de uma lista de variáveis de um determinado tipo é dada pela seguinte sintaxe:

VARIABLE lista_variáveis **:** tipo [:= valor_inicial]**;**

O símbolo de atribuição para sinais é `:=`. Veja-se o exemplo seguinte:

```
c := '1';
```

Entidades e Arquitecturas

Um modelo VHDL é constituído, na sua forma mais simples, por entidades e respectivas arquitecturas. Uma entidade define o interface entre o sistema e o seu ambiente. Normalmente, é uma lista de portas (entradas e saídas). A sintaxe para uma entidade é:

ENTITY entidade **IS**
    portas_de_entrada_e_saída
    parâmetros_físicos_genéricos
END entidade**;**

A arquitectura descreve a funcionalidade do sistema. Esta funcionalidade depende dos sinais de entrada e de saída e dos parâmetros definidos na descrição da interface. A sintaxe para uma arquitectura é:

ARCHITECTURE nome **OF** entidade **IS**
    comandos_declarativos
BEGIN
    especificação_da_funcionalidade_do_componente
    baseado_nas_entradas_e_saídas_e_nos_parametros
END nome**;**

Podem-se especificar várias arquitecturas com nomes diferentes para um dado componente (definido por uma entidade). Por exemplo, um componente pode ter uma arquitectura

```
ENTITY and2 IS
  PORT (a, b : IN BIT; c : OUT BIT);
END and2;
```

Texto 3.4: Entidade para componente AND2.

ao nível comportamental e outra ao nível estrutural. Esta situação é representada na figura 3.4. Uma das vantagens que esta facilidade permite assenta na possibilidade de, em instantes diferentes, se usar uma arquitectura diferente, consoante a finalidade pretendida. Por exemplo, para simulação, usa-se a arquitectura ao nível comportamental e para síntese é aconselhável o uso de uma arquitectura ao nível estrutural.

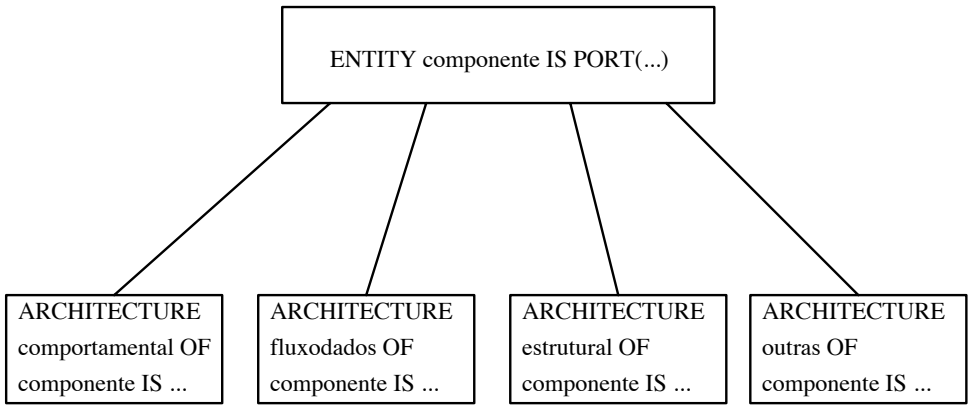


Figura 3.4: Uma entidade e as suas arquitecturas.

Como exemplo, considere-se o componente `AND2`, constante da figura 3.5.

Ter-se-á de escrever o pedaço de código VHDL, apresentado no texto 3.4 para descrever a interface do componente.

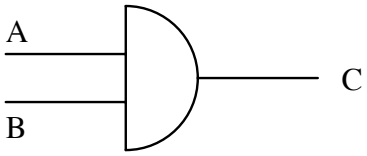


Figura 3.5: Componente AND2.

Basicamente, está-se a definir quais as portas de entrada (sinais `a` e `b`) e de saída (sinal `c`), ou seja, quais os sinais fronteiros. A arquitectura que exprime a funcionalidade do componente `AND2` é dada pelo texto 3.5.

Blocos

Os blocos são um mecanismo que permite agrupar logicamente áreas do modelo. A analogia com um sistema CAE típico é uma folha esquemática. Os blocos podem ser aninhados hierarquicamente.

```
ARCHITECTURE concorrente OF and2 IS
BEGIN
  c <= a AND b AFTER 5ns;
END concorrente;
```

Texto 3.5: Arquitectura para componente AND2.

Os blocos têm um comportamento interessante, conhecido por *bloco guardado*. Um bloco deste tipo contém uma expressão de guarda (do tipo booleano) que habilita ou não as atribuições dentro do bloco.

Além da hierarquia, a instrução `BLOCK` apresenta uma funcionalidade importante que consiste em permitir que certas declarações sejam encapsuladas na parte declarativa, que serão partilhadas por todos os comandos locais [BFMR92].

Quando usado para estruturar (no sentido de organizar) uma descrição, a instrução `BLOCK` tem a vantagem de permitir escrever rápida e concisamente, embora apresente algumas desvantagens.

Simplificadamente, um bloco pode ser visto como um componente instanciado. Se forem necessárias duas instâncias do mesmo componente, o código VHDL terá de ser duplicado. As desvantagens daqui decorrentes são óbvias: maior número de linhas de código, menor legibilidade, depuração e correcção mais dificultadas.

Para concluir, um bloco deve ser usado pelas funcionalidades atrás descritas (hierarquia e encapsulamento). Este construtor é ainda usado, com alguma frequência, para tradução automática de uma qualquer HDL para VHDL, embora não se possa esperar que desse processo resulte um código re-utilizável ou legível.

A sintaxe de um bloco é:

etiqueta : **BLOCK** [guarda]
    cabeçalho_bloco
    comandos_declarativos
BEGIN
    comandos_concorrentes
END BLOCK [etiqueta];

Dados

A linguagem VHDL admite sinais, variáveis e constantes de qualquer tipo pré-definido ou definido pelo utilizador. VHDL é uma linguagem fortemente tipificada, o que significa que qualquer utilização de dados obriga sempre a testar os tipos envolvidos.

Os tipos pré-definidos incluem os seguintes: `INTEGER`, `REAL`, `NATURAL`, `POSITIVE`, `BIT`, `BIT_ARRAY`, `BOOLEAN`, `CHARACTER`, `STRING`. Estes tipos não apresentam qualquer particular dificuldade para alguém familiarizado com linguagens de programação tipificadas, como por exemplo PASCAL ou ADA.

```
PROCESS (b)
BEGIN
  a <= b AFTER 5ns;
END PROCESS;

PROCESS (d)
BEGIN
  c <= d AFTER 8ns;
END PROCESS;
```

```
a <= b AFTER 5ns;
c <= d AFTER 8ns;
```

Texto 3.6: Dois processos equivalentes a duas atribuições concorrentes

Comandos Concorrentes

Os comandos concorrentes são relativamente comuns em qualquer HDL, uma vez que os componentes de qualquer circuito electrónico operam em simultâneo.

O processo é o comando concorrente fundamental [Bak93]. Todos os restantes comandos são ou representam processos. Cada processo é executado de forma concorrente relativamente aos restantes.

Uma atribuição feita a um sinal dentro de uma arquitectura é equivalente a um processo, que executa a pretendida atribuição. Assim, os dois fragmento de código VHDL, apresentados no texto 3.6 são equivalentes.

Os blocos têm por principal utilidade controlar as atribuições feitas a sinais guardados. Essas atribuições são concorrentes, pelo que um bloco pode ser visto equivalentemente como uma série de processos.

Comandos Sequenciais

O construtor `PROCESS` contém, no seu corpo, apenas comandos sequenciais. Porém, os processos em si são entendidos como comandos concorrentes. Um processo pode existir numa arquitectura, definindo desse modo regiões onde o código é executado sequencialmente.

A sintaxe de um processo é:

[etiqueta **:**] **PROCESS** [lista_sensibilidade]
    comandos_declarativos
BEGIN
    comandos_sequenciais
END PROCESS [etiqueta]**;**

Os comandos sequenciais que podem ser usados no corpo de um processo são idênticos aos que se encontrem numa outra linguagem de programação: `CASE`, `IF THEN ELSE`, `LOOP`, etc. Não será aqui especificada a sintaxe nem a semântica associadas a estes construtores. Para tal, consultar [Per91, Nav93, Ash90], por exemplo.

Considere-se novamente, como exemplo, o componente `AND2`, ilustrado na figura 3.5. A arquitectura que exprime a funcionalidade deste componente pode ser escrita, da forma que se apresenta no lado esquerdo do texto 3.7, usando um processo.

```
ARCHITECTURE sequencial OF and2 IS
BEGIN
  PROCESS (a,b)
    VARIABLE temp : BIT;
  BEGIN
    temp := a AND b;
    IF (temp='1')
      c <= temp AFTER 7ns;
    ELSIF (temp='0')
      c <= temp AFTER 5ns;
    ELSE
      c <= temp AFTER 7ns;
    END IF;
  END PROCESS;
END sequencial;
```

```
ARCHITECTURE sequencial OF and2 IS
BEGIN
  PROCESS
    VARIABLE temp : BIT;
  BEGIN
    temp := a AND b;
    IF (temp='1')
      c <= temp AFTER 7ns;
    ELSIF (temp='0')
      c <= temp AFTER 5ns;
    ELSE
      c <= temp AFTER 7ns;
    END IF;
    WAIT ON a,b;
  END PROCESS;
END sequencial;
```

Texto 3.7: Duas arquitecturas equivalentes ao nível comportamental para componente AND2.

Um processo pode conter uma lista de sensibilidade. Esta lista define os sinais que provocam a execução do processo, sempre que o valor de um ou mais desses sinais se alterar. No exemplo anterior, a lista de sensibilidade é dada pelos sinais `a` e `b`, logo a seguir à palavra `PROCESS`. Quando um destes sinais ou ambos mudar de valor, o processo é executado. O processo tem obrigatoriamente de conter uma lista de sensibilidade ou, alternativamente, um comando `WAIT`.

O comando `WAIT` permite a sincronização entre processos. Este comando suspende a execução de processos. A principal utilidade do comando `WAIT` reside em atrasar um processo por um dado período ou em modificar dinamicamente a lista de sensibilidade do processo em causa. A sintaxe genérica do comando `WAIT` é:

WAIT [**ON** lista_sinais] [**UNTIL** expressão] [**FOR** tempo];

A arquitectura `sequencial` da entidade `and2` pode ser reescrita como descrito no lado direito do texto 3.7.

O comando `WAIT` no fim do processo é equivalente à lista de sensibilidade no início do mesmo. O comando `WAIT` tem de estar no fim do processo, porque durante o processo de arranque do simulador todos os processos são executados uma vez, permitindo assim iniciar os valores dos sinais e das variáveis.

O comando `ASSERT` reveste-se de grande utilidade para detecção de erros e depuração de programas, na fase de simulação [Bak93]. Este comando verifica o valor lógico de uma determinada expressão booleana. Se o valor dessa expressão for verdadeiro, o comando não faz nada. Caso o valor da expressão booleana seja falso, o comando envia para o écrã uma mensagem definida pelo programador. O nível do erro indica se a execução deve continuar (quando se usa o valor `WARNING` ou `NOTE`) ou simplesmente parar (quando é usado o valor `ERROR` ou `FAILURE`).

A sintaxe de um comando `ASSERT` é a seguinte:

ASSERT expressão_booleana
    [**REPORT** mensagem]
    [**SEVERITY** nível_erro];

```
PROCESS (rst, set, relogio)
BEGIN
  ASSERT NOT (rst='1' AND set='1')
    REPORT "Set e Reset iguais a 1!"
    SEVERITY NOTE;
  IF set='1' THEN
    state <= '1' AFTER 6ns;
  ELSIF rst='1' THEN
    state <= '0' AFTER 6ns;
  ELSIF clk='1' AND clk'EVENT THEN
    state <= d AFTER 6ns;
  END IF;
END PROCESS;
q  <= state;
qb <= NOT state;
```

Texto 3.8: Exemplo de uso do comando ASSERT.

Considere-se o exemplo do texto 3.8. O processo modela o comportamento de um *flip-flop* do tipo D com sinais assíncronos para iniciar a saída aos níveis '0' (`rst`) ou '1' (`set`). Neste dispositivo, a combinação das entradas `set`=1 e `rst`=1 não é válida. Sempre que essa situação ocorrer, o simulador enviará uma mensagem ao utilizador a assinalar o facto.

O comando `ASSERT` apresentado neste exemplo é sequencial, já que se encontra num processo. Existe também uma versão concorrente deste comando, que apresenta a mesma sintaxe e que existe fora dos processos.

3.2 Selecção dum Subconjunto VHDL

A linguagem VHDL é adequada para simulação e modelação, pois foi desenvolvida com o cuidado de a tornar flexível e incluir construtores comportamentais de alto nível.

É devido à generalidade e flexibilidade da linguagem que nem todos os construtores podem ser usados em todo o tipo de aplicação. Para síntese, modelação e teste, diferentes subconjuntos VHDL devem ser usados [NS91]. Para síntese, por exemplo, o código usado deve incluir apenas os construtores que possam ser directamente traduzidos para o nível físico. Surge assim, o conceito de subconjunto VHDL.

Neste capítulo pretende-se identificar quais os comandos, construtores, operadores e demais elementos da linguagem VHDL, que serão usados, para tornar possível a modelação de controladores paralelos, descritos a partir de RdPSI. Pretende-se pois identificar o subconjunto VHDL, bem como o estilo de descrição, para modelar RdPSI. Antes disso porém, apresentam-se as várias abordagens encontradas para modelação em VHDL de Máquinas de Estados.

Um dos objectivos na escolha do subconjunto VHDL é que ele seja admitido tanto em ferramentas de simulação, como de síntese. Adicionalmente, pretende-se que o nível de abstração desse subconjunto seja o mais alto possível.

3.2.1 A Simulação e a Síntese

A linguagem VHDL atinge a sua plenitude em aplicações de simulação (o propósito para o qual foi criada), pelo que quase todas as ferramentas de simulação aceitam a totalidade da linguagem, ou, quando tal não acontece, o subconjunto aceite em simulação é tão ou mais vasto que o subconjunto de síntese.

Todavia, VHDL também é usada, com sucesso, noutras áreas de aplicação que não a simulação, nas quais se inclui a síntese. Porém, o seu propósito inicial fez com que a semântica para síntese não esteja totalmente definida.

Uma aplicação de síntese é um processo que se deseja o mais automático possível. Quando integrado num ambiente de projecto de sistemas digitais, o seu objectivo é transformar uma especificação abstracta numa descrição no nível mais baixo do domínio físico. Essa transformação é feita tendo em consideração as restrições que acompanham o projecto (área, velocidade, tecnologia, consumo de energia, testabilidade, ciclo de relógio, etc). O resultado desse processo é normalmente uma *netlist*.

A síntese de sistemas digitais pode decompor-se na seguinte sequência de tarefas de síntese mais simples: síntese de sistema, síntese de alto nível, síntese de transferência de registo, síntese lógica e mapeamento na tecnologia [MV94].

O objectivo da *síntese de sistema* é dividir um sistema em subsistemas, partindo da sua especificação abstracta e mínima. Esses subsistemas, que podem ser vistos como processos concorrentes que comunicam entre si, são caracterizados pela sua descrição comportamental ao nível algorítmico. Este tipo de síntese não tem ainda aceitação comercial.

Camposano et al. [CST91] definem a *síntese de alto nível* (também designada por síntese algorítmica) como sendo o processo que sintetiza uma estrutura física, a partir de uma descrição[2] comportamental. O resultado da síntese de alto nível é uma descrição estrutural com duas componentes: uma componente de manipulação de dados (unidades funcionais, unidades de memória, e unidades de interligação) e uma componente de controlo especificada por um diagrama de transição.

Muita investigação tem sido realizada nesta área, já que a síntese de alto nível ajuda a manipular os sistemas digitais mais complexos. Este tipo de síntese aceita como entrada uma especificação de alto nível, sendo este tipo de especificação, geralmente, mais pequena, mais facilmente gerada e entendida, menos errónea e mais rapidamente simulável que uma especificação equivalente de mais baixo nível.

A grande dificuldade para o sucesso da síntese de alto nível reside na modelação do tempo [CST91]. A razão assenta no facto de um modelo VHDL ser simulável, o que implica que o tempo esteja bem definido. Porém, para o processo de síntese de alto nível, o tempo não é propositadamente explicitado. Será a ferrramenta de síntese a determinar quando é que uma dada operação é exactamente executada [CST91].

A síntese a partir de código VHDL concorrente é essencialmente *síntese de transferência de registo* (ou simplesmente síntese RTL). Este tipo de síntese consiste em mapear os operadores VHDL para os componentes primitivos permitidos nas ferramentas a usar posteriormente, como por exemplo, sintetizadores lógicos. Os sistemas comerciais de síntese, na sua grande

[2]Note-se o emprego do termo *descrição* e não *especificação*, o que significa que foram já tomadas decisões fundamentais de realização.

maioria, situam-se a este nível. O resultado da síntese RTL, que consiste numa descrição de nível transferência de registo constituída por blocos de lógica combinatória e elementos de memória, é passado à síntese lógica

A ênfase da *síntese lógica* reside na optimização, ou mais especificamente na minimização lógica, tendo por objectivo obter a mínima área possível. Esta tarefa de síntese permite abstrair a lógica combinatória da tecnologia a utilizar e do tipo de projecto.

O *mapeamento na tecnologia* recebe como entrada uma rede de portas lógicas abstractas e produz um conjunto de células físicas de uma biblioteca de uma dada tecnologia.

Não há actualmente nenhum ambiente que inclua a totalidade das tarefas de síntese referidas. Nomeadamente, as sínteses de sistema e alto nível apresentam problemas de formalismo [MV94].

Dados

Um cuidado a ter resulta do facto de VHDL oferecer tipos de dados abstractos equivalentes aos que se encontram noutras linguagens de programação. Tendo em vista a síntese, todos esses tipos devem ser traduzidos para um formato que envolva bits, bytes ou RAM.

Por exemplo, um tipo enumerado com dois valores pode ser mapeado em formato bit. Outros tipos enumerados, assim como tipos escalares, são mapeados para palavras de tamanho apropriado. Uma estrutura (*record*) pode ser vista como a concatenação dos formatos correspondentes aos subtipos dos campos. Os vectores podem ser mapeados em RAM ou registos.

Os tipos de dados que envolvam apontadores (`access`) e ficheiros (`file`) não devem ser utilizados, caso se pretenda a síntese, pois não é trivial a sua conversão para o nível físico.

Relativamente às expressões lógicas, devem ser notadas as duas seguintes restrições para o processo de síntese [BFMR92]:

- Não usar operadores sobre tipos de dados não suportados.
- Banir o uso de operadores intrinsecamente ligados à simulação.

A primeira das restrições é óbvia; se não se podem usar os tipos de dados, também não se podem usar os respectivos operadores. Mais concretamente, se não se usam ficheiros nem apontadores, também não se usam operações de escrita ou leitura sobre ficheiros nem operações de alocação de memória para um apontador.

A segunda restrição resulta do facto de alguns dos operadores estarem directamente ligados ao processo de simulação e portanto não serem suportados na síntese. Como exemplo, refiram-se os atributos associados a sinais como `'TRANSACTION`, `'ACTIVE` e `'QUITE`.

3.2.2 Representação de Diagramas ASM

Os diagramas ASM (*Algorithm State Machines*) [Cla73] permitem definir Máquinas de Estados Finitas (abreviadamente FSM de *Finite State Machine*) e são de grande utilidade para simulação em VHDL. Os diagramas ASM são usados para representar a parte de controlo

```
ENTITY maq_estados IS
  PORT (clock, in1, in2 : IN BIT;
        out1, out2, out3 : OUT BIT);
END maq_estados;
```

Texto 3.9: Entidade para diagrama ASM.

de um sistema electrónico. Uma das características que identifica os diagramas ASM reside no facto de estes só poderem ter um estado activo, em cada instante, pelo que permitem somente modelar estruturas de controlo sequenciais. Apesar disso, a sua aplicação é enorme, tanto em meios académicos, como em ambientes industriais.

Ao longo desta secção, retomar-se-á como exemplo o diagrama ASM apresentado na figura 1.2. Para o sistema que este diagrama encerra, a entidade VHDL é dada pelo código presente no texto 3.9. Encontraram-se basicamente dois métodos para modelação de FSM [Bak93, Nav93], sobre os quais existem inúmeras variantes.

O primeiro método, utilizando um processo e o comando `CASE`, é apresentada no texto 3.10. Nesta descrição comportamental, a máquina de estados encontra-se garantidamente apenas num estado, em cada instante. O comando `CASE` é o mais apropriado pois escolhe apenas um dos casos, ignorando os restantes. O mesmo efeito seria conseguido, usando o construtor `IF .. THEN .. ELSIF .. END IF`.

A seguir ao comando `CASE`, o comando `WAIT` suspende o processo durante 1ηs. Depois de esgotado esse tempo, as saídas são colocadas a '0' e o processo é novamente executado. A razão para a suspensão por esse tempo é para criar um atraso entre a atribuição de um valor ao sinal que define o estado (`Actual`) e a sua utilização no comando `CASE`. Dado que `Actual` é um sinal, o novo valor atribuído fica disponível no ciclo seguinte de simulação, pelo que o atraso permite que o novo valor estabilize. Se o comando `WAIT` não estivesse presente, a atribuição do novo valor ao sinal `Actual` e a nova execução do processo ocorreriam no mesmo ciclo de simulação, causando a utilização do valor antigo de `Actual` pelo comando `CASE`.

Uma variante a este método consegue-se usando dois processos: um para determinar o próximo estado e outro para cálculo das saídas de controlo. Torroja et al. [TPU93] desenvolveram uma ferramenta que gera, a partir de uma especificação gráfica de uma FSM, o respectivo código VHDL. A arquitectura desse código, cujo estilo é idêntico ao atrás descrito, é dividida em dois processos, um para a parte combinatória e outro para os registos de estado —baseado em processos e em instruções `CASE`— e pode ser sintetizada, segundo os autores.

O segundo método, ao nível fluxo de dados, utiliza blocos e guardas e tem a forma apresentada no texto 3.11. Os comandos para atribuição de valores às saídas não são guardados (falta a palavra `GUARDED`), pelo que podem ser colocados em qualquer local dentro da parte de comandos da arquitectura. No exemplo apresentado, as atribuições são colocadas junto aos blocos correspondentes aos estados onde as saídas são activadas.

```
ARCHITECTURE comport OF maq_estados IS
  TYPE VEstado IS (a, b, c);
  SIGNAL Actual : VEstado := a;
BEGIN
  PROCESS
  BEGIN
    CASE Actual IS
      WHEN a =>
        out1 <= '1';
        out3 <= '1' WHEN in1='1' ELSE '0';
        WAIT UNTIL clock='1'
        IF (in1='1') THEN
          Actual <= b;
        ELSE
          Actual <= c;
        END IF;
      WHEN b =>
        WAIT UNTIL clock='1'
        Actual <= a;
      WHEN c =>
        out2 <= '1';
        WAIT UNTIL clock='1'
        IF (in2='1') THEN
          Actual <= a;
        ELSE
          Actual <= b;
        END IF;
    END CASE;
    WAIT FOR 1ns;
    out1 <= '0';
    out2 <= '0';
    out3 <= '0';
  END PROCESS;
END comport;
```

Texto 3.10: Primeira arquitectura para diagrama ASM.

```
ARCHITECTURE fluxodados OF maq_estados IS
  TYPE TEst IS (a, b, c);
  TYPE VEst IS ARRAY(NATURAL RANGE <>) OF TEst;

  FUNCTION Eresolve (Est: IN VEst)
    RETURN TEst IS
  BEGIN
    ASSERT (Est'LENGTH=1)
      REPORT "FSM em mais que um estado!"
      SEVERITY ERROR;
    RETURN Est(1);
  END Eresolve;

  SIGNAL Actual: Eresolve TEst REGISTER := a;

BEGIN
  clocking : BLOCK (clock='1' AND NOT clock'STABLE)
  BEGIN
    s1 : BLOCK (Actual=a AND GUARD)
    BEGIN
      Actual <= GUARDED b WHEN in1='1' ELSE c;
      out1 <= '1' WHEN Actual=a ELSE '0';
      out3 <= '1' WHEN (Actual=c AND in1='1') ELSE '0';
    END BLOCK s1;
    s2 : BLOCK (Actual=b AND GUARD)
    BEGIN
      Actual <= GUARDED a;
    END BLOCK s1;
    s3 : BLOCK (Actual=c AND GUARD)
    BEGIN
      Actual <= GUARDED b WHEN in2='1' ELSE a;
      out2 <= '1' WHEN Actual=c ELSE '0';
    END BLOCK s1;
  END BLOCK clocking;
END fluxodados;
```

Texto 3.11: Segunda arquitectura para diagrama ASM.

3.2.3 Representação de Redes de Petri

Nesta subsecção, pretende-se identificar qual o código VHDL a usar para modelar RdP e, mais especialmente, RdPSI. Uma das características que se deseja que esse código apresente é a legibilidade, ou seja, pretende-se que estrutura da RdP seja facilmente perceptível, a partir do código VHDL [Aga90].

Nesse mesmo artigo, é referido que a utilização de funções de resolução é fundamental para a modelação de RdP em VHDL. Tal facto deve-se à possibilidade de um lugar de uma RdP poder ser marcado simultaneamente. Ora, se se pretender que as RdP sejam seguras, há que garantir que tal não sucede, o que pode conseguir-se, recorrendo a uma função de resolução.

Relativamente à implementação de RdP em VHDL, encontraram-se três alternativas que abaixo se descrevem. Como exemplo ao longo da presente secção, considere-se a RdPSI apresentada na figura 3.6.

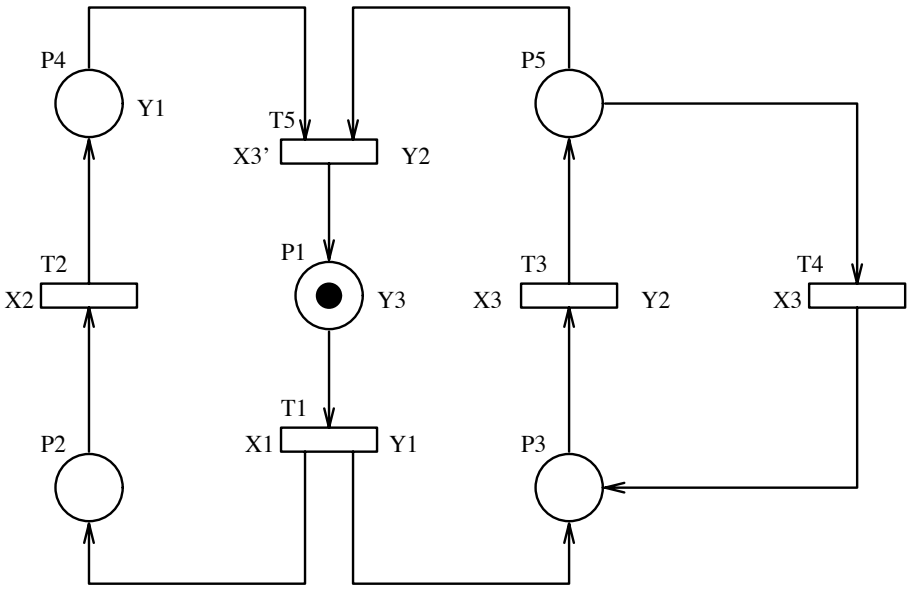


Figura 3.6: Controlador baseado numa RdPSI.

A primeira solução, apresentada no texto 3.13, foi adaptada de [BFMR92] e representa os lugares por sinais guardados do tipo `BIT` e do género (*kind*) `REGISTER`. Os lugares marcados são associados a sinais com o valor '1'. A função de resolução (obrigatória para sinais guardados) garante que a RdP está livre de conflitos no acesso aos lugares.

As transições são descritas por blocos guardados (ver subsecção 3.1.6). A guarda do bloco verifica se a transição está ou não habilitada a disparar

Uma segunda alternativa, apresentada por Pardey e Bolton [PB91, Bol91], usa um processo e um conjunto de atribuições concorrentes a sinais, para descrever o comportamento da RdPSI (ver texto 3.13). A entidade e o processo para inferir o registo de estado são idênticos aos usados em FSM (ver subsecção 3.2.2). No entanto, para o bloco de lógica combinatória, é usada uma descrição fluxo de dados, que involve uma lista de transições `Ti`, próximas marcações `NPi`, e saídas de controlo `Yi`.

O tipo de dados `REG_BIT` é o disponibilizado no produto ALLIANCE [GP93] e equivale a um subtipo resolvido do tipo `bit`, usando uma função de resolução que verifica que apenas

```
PACKAGE util_pkg IS
  FUNCTION petri_resolve(ARG:BIT_VECTOR) RETURN BIT;
  SUBTYPE place IS petri_resolve BIT;
END util_pkg;

PACKAGE BODY util_pkg IS
  FUNCTION petri_resolve(arg:BIT_VECTOR) RETURN BIT IS
  BEGIN
    ASSERT (arg'LENGTH<=1)
      REPORT "Conflict: Place already busy..."
      SEVERITY ERROR;
    IF (arg'LENGTH=1) THEN
      RETURN arg(arg'LEFT);
    ELSE
      RETURN ('0');
    END if;
  END petri_resolve;
END util_pkg;

ENTITY controller IS
  PORT (clock, reset, X1, X2, X3 : IN BIT;
        Y1, Y2, Y3 : OUT BIT);
END controller;

ARCHITECTURE dataflow OF controller IS
  USE WORK.util_pkg.ALL;
-- define places
  SIGNAL P1, P2, P3, P4, P5 : place REGISTER;
BEGIN
  INIT : BLOCK (NOT reset'STABLE AND reset='1')
    BEGIN  -- Initial marking
      P1 <= GUARDED '1';
    END BLOCK;

-- Transition T1
  T1 : BLOCK (P1='1' AND X1='1' AND clock'EVENT AND clock='1')
    BEGIN
      ASSERT NOT (P2='1' OR P3='1')
        REPORT "Marking a marked place"
        SEVERITY ERROR;
     -- unmark input places
      P1 <= GUARDED '0';
     -- mark output places
      P2 <= GUARDED '1';
      P3 <= GUARDED '1';
    END BLOCK;

-- Transitions T2 to T5 identical

  Y1 <= P4 OR (P1='1' AND X1='1');
  Y2 <= (P3='1' AND X3='1') OR (P4='1' AND P5='1' AND X3='0');
  Y3 <= P1;

END dataflow;
```

Texto 3.12: Primeira solução para RdP.

```
ENTITY controller IS
  PORT (clock, reset, X1, X2, X3 : IN BIT;
        Y1, Y2, Y3 : OUT BIT);
END controller;

ARCHITECTURE dataflow OF controller IS
  SIGNAL P1, P2, P3, P4, P5 : REG_BIT REGISTER;
  SIGNAL NP1, NP2, NP3, NP4, NP5 : BIT;
  SIGNAL T1, T2, T3, T4, T5 : BIT;
BEGIN
  PROCESS BEGIN
    WAIT UNTIL clock'EVENT AND clock='1';
    IF  reset='0' THEN
      P1 <= GUARDED '1';
      P2 <= GUARDED '0';
      P3 <= GUARDED '0';
      P4 <= GUARDED '0';
      P5 <= GUARDED '0';
    ELSE
      P1 <= GUARDED NP1;
      P2 <= GUARDED NP2;
      P3 <= GUARDED NP3;
      P4 <= GUARDED NP4;
      P5 <= GUARDED NP5;
    END IF;
  END PROCESS;

  T1 <= X1 AND P1;
  T2 <= X2 AND P2;
  T3 <= X3 AND P3;
  T4 <= X3 AND P5;
  T5 <= NOT X3 AND P4 AND P5;

  NP1 <= T5 OR (P1 AND NOT T1);
  NP2 <= T1 OR (P2 AND NOT T2);
  NP3 <= T1 OR T4 OR (P3 AND NOT T3);
  NP4 <= T2 OR (P4 AND NOT T5);
  NP5 <= T3 OR (P5 AND NOT (t4 OR T5));

  Y1 <= P4 OR T1;
  Y2 <= T3 OR T5;
  Y3 <= P1;

END dataflow;
```

Texto 3.13: Segunda solução para RdP.

uma atribuição está, em cada instante, activa. O tipo `REG_BIT` é idêntico ao tipo de dados `PLACE` apresentado no primeiro fragmento de código desta secção.

Adicionalmente, pode incluir-se uma série de comandos `ASSERT` que permite verificar, durante o processo de simulação, algumas das propriedades (não morta, segura, livre de conflitos e determinística) da RdP.

Os critérios que foram considerados na construção desta solução foram os que a seguir se descrevem [PB91]. A descrição deve:

- ser facilmente derivável, a partir da RdP;
- permitir entrada adequada, tanto em ferramentas de simulação, como de síntese, evitando assim os erros que se introduzem, quando se traduz um modelo que simula correctamente, num formato apropriado para síntese;
- sintetizar eficientemente, dado que o estilo de descrição influencia enormemente a qualidade do resultado do processo de síntese;
- usar o mesmo subconjunto VHDL que o suportado pelo sintetizador lógico.

A terceira e última hipótese, apresentada por Peng, consiste em converter a RdP temporal[3] para uma máquina de estados finita (ver subsecção 3.2.2). A geração da máquina de estados finita é um processo similar ao da geração do grafo de alcançabilidade. O código gerado tem por único fim a simulação da descrição de entrada. A metodologia usada bem como um exemplo de um controlador e respectivo código VHDL podem ser encontrados em [Pen92].

Relativamente a estas três alternativas consideradas, a terceira não se mostra atraente, no contexto deste projecto, por permitir apenas simular as RdP, tendo em conta que se pretende que a síntese seja possível. Restam, portanto, duas hipóteses. Dessas a segunda é a que apresenta mais vantagens. Primeiro, por ser mais simples. Segundo, por permitir entrada tanto em ferramentas de simulação, como de síntese. Finalmente e devido às razões apresentadas na subsecção 3.1.6 (maior número de linhas de código, menor legibilidade, depuração e correcção mais dificultadas) que resultam da utilização dos blocos, a primeira alternativa é menos sedutora. Estes aspectos motivaram a escolha da metodologia apresentada por Pardey e Bolton, para modelação de RdPSI em VHDL.

[3]Cada especificação é constituída por duas partes separadas, mas relacionadas: uma parte de controlo, modelada por uma RdP Temporizada (trata-se, na realidade, de algo similar a uma RdP Síncrona e Interpretada), e uma parte de dados [PK86]. Por esse motivo, é usado o termo "*Extended Timed Petri Net*" para se referir ao conjunto constituído pelas duas partes.

Capítulo 4

Metodologia Desenvolvida

Neste capítulo, apresenta-se, em primeiro lugar, as metodologias e ferramentas CAD que foram propostas, no sentido de permitir a especificação e a implementação de controladores paralelos. De seguida, é apresentado o ambiente de desenvolvimento e respectiva metodologia pensados para possibilitar a especificação, análise e síntese de controladores paralelos. Segue-se uma apresentação detalhada dos módulos de análise e compilação, que constituem a aplicação computacional ConPar desenvolvida no âmbito deste trabalho. A terminar, tenta-se mostrar a viabilidade da metodologia seguida e as potencialidades da aplicação ConPar, através da análise de alguns exemplos encontrados na literatura consultada.

4.1 Metodologias e Produtos CAD existentes

Na diversa bibliografia consultada, encontrou-se um conjunto de referências a metodologias ou ferramentas CAD desenvolvidas com o intuito de poder especificar, simular ou implementar controladores paralelos e/ou sequenciais, usando como suporte inicial as RdP. Nesta secção, pretende-se mostrar, muito resumidamente, esses métodos e produtos e verificar as suas principais potencialidades.

A especificação de controladores paralelos através de RdP mereceu a atenção de diversas equipas de investigação, desde há longo tempo. O primeiro trabalho nessa direcção, resultou no desenvolvimento de métodos unicamente para análise. Azema et al. [AVD76] apresentam uma metodologia que permite analisar e simular o comportamento de sistemas concorrentes, usando como suporte RdP Interpretadas. Essa metodologia foi posteriormente melhorada, como pode ser verificado em [VCBA83, CVBE83].

Os trabalhos seguintes, no final da década de 70, tiveram como resultado métodos com o propósito de permitir, não só a análise mas também a síntese de controladores paralelos, usando RdP Interpretadas.

Leung et al. [LMB77] descrevem duas metodologias que permitem implementar controladores em estruturas PLA (*Programmable Logic Array*), usando RdP Interpretadas para representar os controladores. Às transições associam-se as variáveis de entrada e aos lugares associam-se as variáveis de saída, pelo que são considerados apenas controladores do tipo Moore. A RdP especificada é decomposta em máquinas de estado e, seguidamente, para cada uma das máquinas de estado resultante, é construída uma tabela, ou gerado o grafo

de alcançabilidade e atribuído, a cada nodo deste, um código binário único. A finalizar, qualquer uma das representações (tabela ou grafo) é implementada em dispositivos PLA.

Augin et al. [ABA78] consideraram uma metodologia similar à anterior. Posteriormente, introduziram uma extensão ao método anterior [ABA80], que se baseava na construção do grafo de alcançabilidade para toda a RdP. Se a RdP fosse demasiado complexa, procedia-se à sua simplificação, por métodos de redução. Adicionalmente, era possível associar sinais de saída tanto a transições como a lugares, pelo que eram considerados controladores dos tipos Moore e Mealy.

Toulette e Parsy [TP79] introduziram um método heurístico para decompor uma RdP em várias máquinas de estado, implementadas directamente, atribuindo um elemento de memória a cada lugar.

A partir de meados da década de 80, foram apresentados algoritmos mais eficientes, novas metodologias e novas soluções, permitindo a implementação de controladores paralelos mediante o uso de tecnologia VLSI.

Brück et al. [BKKR86] descrevem um método, idêntico ao apresentado por Augin et al. [ABA78], que permite sintetizar controladores complexos a partir de descrições algorítmicas concorrentes. É usada, para especificação do sistema, uma RdP modificada, que inclui seis tipos distintos de transições, sendo cada uma delas implementada em estruturas *hardware* diferentes. A especificação é, primeiramente, dividida em máquinas de estado, que comunicam entre si de forma assíncrona. A implementação final de cada uma destas máquinas é feita com tecnologia CMOS.

A plataforma CAMAD [PKL88] usa como notação de entrada as RdP Temporizadas Extendidas. Cada especificação é constituída por duas partes separadas, embora relacionadas: uma parte de controlo, modelada por uma RdP Temporizada (trata-se, na realidade, de uma RdP de características idênticas às das RdP Síncronas e Interpretadas), e uma parte de dados [PK86]. A versão CAMAD, datada de 1992, recebe como entrada uma especificação algorítmica e gera uma descrição para a parte de dados (*netlist*) e um microprograma para o controlador [KP87]. A RdP de entrada permite construir o grafo de alcançabilidade, podendo-se, com base neste grafo, sintetizar o controlador microprogramado. O uso da linguagem VHDL é também considerado, embora a sua finalidade seja apenas a simulação [Pen92].

Balakrishnan et al. [BMBL88] propõem uma técnica que permite a síntese de controladores descentralizados, baseada na análise das interligações entre a unidade de controlo e a parte de dados. Para descrição da unidade de controlo é usado um dado tipo de RdP (são associados dois sinais a cada transição e uma nova regra de disparo é introduzida). O fluxo de informação entre as unidades de dados e de controlo é analisado e o controlador dividido num conjunto apropriado de subcontroladores, sendo cada um destes implementado por uma estrutura do tipo PLA. Uma das vantagens desta metodologia assenta no facto de os subcontroladores resultantes apresentarem área pequena, funcionarem a grande velocidade e serem facilmente testados.

Nishimura e Zaky [NZ89] apresentam um metodo de síntese que envolve componentes síncronos e assíncronos. Os sistemas são modelados por uma versão das RdP (*Design Petri Nets*), contendo três tipos distintos de transições (sem relógio, assíncronas ou síncronas). A RdP de entrada é decomposta num conjunto de máquinas de estado, com a restrição de todas as transições de uma subrede serem assíncronas ou síncronas relativamente ao

mesmo relógio. Cada uma das máquinas de estado é síntetizada com o auxílio das técnicas convencionais.

A metodologia descrita por Patel [Pat90] permite implementar controladores especificados por RdPSI. A especificação de um controlador é, primeiramente, mapeada directamente (codificação *one-to-one*) para células genéricas ao nível físico. Cada transição é mapeada numa porta lógica AND e cada lugar é implementado com um *flip-flop* do tipo D e uma porta lógica OR (que colecta todas as transições de entrada). A realização inicial é minimizada, seguindo-se as fases de mapeamento na tecnologia e optimização. Na abordagem apresentada, os controladores são implementados em dispositivos PLA or LCA (*Logic Cell Array*). Em adição, é apresentado um método que permite fazer uma estimativa do custo e velocidade resultantes.

Pardey e Bolton [PB91] introduzem um metodologia em VHDL para projecto de controladores paralelos síncronos, permitindo a representação e verificação ao nível RTL, a síntese lógica e o teste de consistência ao nível da porta lógica. São incluídos também algoritmos para decomposição da especificação e são disponibilizadas várias alternativas para codificação (atribuição de estados). O teste da RdP é feito através da inclusão de uma série de comandos `ASSERT` que permitem verificar algumas das propriedades, durante o processo de simulação. Esta metodologia não se encontrava, à data do artigo, totalmente automatizada. Por exemplo, o código VHDL tem de introduzir-se manualmente. A partir daí, o processo beneficia dos algoritmos considerados já implementados e de todas as ferramentas VHDL já existentes.

Ferrarini e Maffezzoni [Fer92, FM91] descrevem um ambiente para projecto de controladores lógicos, baseado em RdP seguras. Foram introduzidas guardas nas transições e permitido o uso de ciclos próprios e arcos inibidores. A grande ênfase desse ambiente consiste no conjunto de algoritmos desenvolvidos, que permite verificar formalmente a correcção do controlador.

Gomes et al. [GSG92, GSGGC93] apresentam uma metodologia para especificação e implementação de unidades de controlo, em que condicionantes de tempo real e concorrência de acções se mostrem características fundamentais. O método usa como suporte RdP Coloridas e Sincronizadas e integra capacidades de especificação, usando lógica imprecisa e temporizações. A implementação é realizada, com base na tradução da especificação do sistema (sequencial ou concorrente), numa máquina de estados, representando todo o sistema. Esta tradução é obtida mediante a construção da árvore de alcançabilidade.

PARIS [Koz93] é um conjunto de comandos que foram acrescentados à plataforma SIS [SSL+92], permitindo a especificação e a síntese de controladores paralelos, baseados em RdPSI. A descrição pode, entre outros, ser testada (viva, segura e determinística), reduzida (ao nível estrutural) ou traduzida para o formato BLIF [SSL+92]. A optimização do sistema e a sua síntese podem obter-se, usando todas as facilidades disponibilizadas pela plataforma SIS.

Pardey et al. [PABA94] apresentam ainda uma outra metodologia manual que possibilita a especificação dum controlador baseado em RdP, em linguagem PALASM-4 [AMD91]. A especificação do sistema pode ser simulada, usando o produto PALASM e a sua implementação possível, recorrendo-se ao uso de dispositivos de lógica programável.

De todas estas metodologias, a única que considera a existência da linguagem VHDL para fins de simulação e síntese é a apresentada por Pardey e Bolton. Contudo, a não automa-

tização do processo de geração de código VHDL, bem como o facto de os testes apenas se realizarem durante a simulação, tendo como base apenas os comandos `ASSERT` incluídos no código VHDL, limita um pouco a utilização da metodologia e não garante uma análise que se pretende o mais completa possível.

Será, de seguida, proposta uma metodologia que permite especificar controladores paralelos, usando uma linguagem textual apropriada. Os sistemas assim especificados são depois analisados, animados e é gerada uma representação em código VHDL. Antes de apresentar a metodologia adoptada, far-se-á uma ligeira referência à linguagem que permite a especificação dos controladores paralelos.

4.2 A Linguagem ConPar

A especificação dos controladores paralelos é feita através de uma linguagem textual desenvolvida para o efeito, no âmbito desta tese, de nome ConPar (Controladores Paralelos). Como base para o desenvolvimento desta linguagem foi utilizada a linguagem PNSF, definida por Kozlowski em [Koz93]. Esta linguagem é composta por descrições baseadas em regras na forma de sequentes lógicos de Gentzen (regras de produção extendidas), e faz-se uso também das lógicas de decisão e temporal [Ada91]. Foram igualmente analisadas outras linguagens, desenvolvidas para o mesmo efeito, nomeadamente as definidas por Valette et al. [VCBA83, CVBE83] ou Silva e Velilla [SV82]. Pode concluir-se, no entanto, que a capacidade de modelação destas não era superior à da linguagem ConPar.

A linguagem é não-procedimental (não-algorítmica), o que simplifica a programação de modelos baseados em RdP, pois permite validar as especificações [SV82].

No apêndice A, apresenta-se a sintaxe completa da linguagem, faz-se uma descrição mais detalhada dos vários construtores, e consideram-se alguns exemplos ilustrativos.

4.3 O Ambiente de Desenvolvimento

O ambiente completo de desenvolvimento, que permite a especificação, a análise, a animação, a simulação e a síntese de controladores paralelos baseados em RdPSI, encontra-se ilustrado na figura 4.1.

Na construção deste ambiente e respectiva metodologia adoptada, dedicou-se especial cuidado aos dois pontos seguintes, que se consideraram fulcrais:

1. Análise das propriedades da RdPSI;
2. Compilação das especificações para VHDL.

O primeiro destes pontos permite verificar a correcta especificação do controlador, evitando o recurso à heurística e à intuição, por parte da equipa de projecto. O segundo habilita a utilização da linguagem VHDL, beneficiando-se de todo o conjunto de ferramentas CAD já disponíveis para esta linguagem.

Para especificar um dado controlador baseado numa RdPSI, há que introduzir o seu código

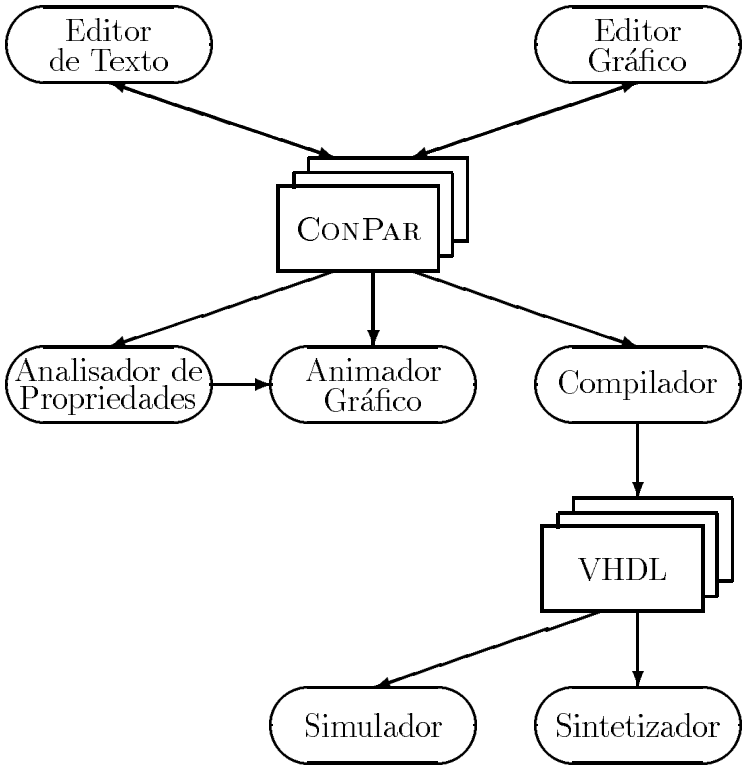


Figura 4.1: O Ambiente completo de Desenvolvimento.

em linguagem ConPar (ver secção 4.2), através de um qualquer editor de texto. Alternativamente, a especificação do controlador poder fazer-se recorrendo a um editor gráfico que produza como resultado um ficheiro em linguagem ConPar, ficando pois o utilizador liberto do conhecimento desta. Esta última funcionalidade não foi implementada por não se considerar indispensável na defesa da tese proposta.

Os ficheiros assim criados podem então servir como entrada, em qualquer dos três módulos subsequentes: Analisador de Propriedades, Animador Gráfico e Compilador.

O módulo *Analisador de Propriedades* tem por função verificar as propriedades (viva, segura, livre de conflitos) das RdP. Se alguma das propriedades não se verificar, o módulo assinala o facto, através de uma mensagem no écrã, podendo, caso se ache adequado, recorrer-se ao módulo *Animador Gráfico*, que permite, de uma forma interactiva, visualizar o comportamento da RdPSI e detectar quais as causas que motivaram o erro. O módulo de animação gráfica não foi implementado, já que também não se enquadrava no contexto deste trabalho.

Caso não seja assinalado nenhum erro grave para a RdP de entrada, pode considerar-se que o respectivo controlador está correctamente especificado[1], pelo que pode então utilizar-se o módulo *Compilador*. Este módulo faz a compilação da RdPSI de entrada, especificada em linguagem ConPar, para o respectivo código em VHDL, gerado ao nível fluxo de dados.

Os ficheiros VHDL gerados podem depois ser utilizados em ferramentas VHDL (na figura 4.1 apresentam-se, como exemplos, um simulador e um sintetizador), desde que o subconjunto,

[1]O facto de uma RdP não violar nenhuma das propriedades não significa que ela está bem especificada. A correcta especificação da RdP depende da "ideia" que a equipa de projecto tem. Contudo, se ela violar alguma das propriedades está, de certeza, mal especificada.

que essas ferramentas admitem, inclua todos os construtores e comandos que o módulo de compilação gera (ver apêndice B).

No Departamento de Informática da Universidade do Minho, encontravam-se disponíveis, à data da realização do presente trabalho, dois produtos de simulação e síntese para VHDL: ALLIANCE [GP93] e SYNT/MINT [Min94, Syn94]. Nesse sentido, houve a preocupação de gerar código VHDL que pudesse ser utilizado nas ferramentas do produto ALLIANCE, já que este se encontra disponível e é usado com sucesso, em muitas instituições. Trata-se de um produto, de domínio público, que corre em sistemas UNIX e que pode ser obtido gratuitamente, através dos serviços de rede, no endereço ftp `cao-vlsi.ibp.fr`. O produto SYNT/MINT não foi considerado, pois o subconjunto aceite não é compatível com o considerado neste trabalho.

Os módulos *Analisador de Propriedades* e *Compilador* foram incluídos na aplicação ConPar desenvolvida no âmbito deste trabalho. Nas duas secções seguintes, apresentam-se, mais detalhadamente, cada um desses módulos.

4.4 Análise de Propriedades

O módulo de análise verifica se as especificações de entrada são vivas e livre de conflitos. O teste relativo à segurança da RdPSI não é necessário efectuar-se, visto que se adoptou a regra de disparo forte, ou seja a capacidade dos lugares de saída é testada, antes de se disparar a respectiva transição. Deste modo, não é possível marcar um lugar de saída, se ele já contiver uma marca. A segurança nas RdPSI é violada, se surgir uma situação idêntica à apresentada na figura 2.19(a). Todavia, esse problema está salvaguardado, pois a RdPSI é testada relativamente a conflitos, ou seja, qualquer situação semelhante às apresentadas na figura 2.19 é detectada e comunicada ao utilizador.

Verificar se uma RdP é viva é uma operação, na maior parte dos casos, bastante demorada. Assim, no presente trabalho, restringiu-se essa verificação aos seguintes testes:

1. não existirem transições fonte, nem transições destino;
2. não existirem lugares fonte, nem lugares destino;
3. cada marcação ter, no mínimo, uma outra marcação seguinte [Koz93];
4. o disparo de cada transição aparecer no grafo de alcançabilidade da RdPSI [Koz93].

Estes testes não garantem que a RdP é viva, mas, caso não se verifiquem, mostram que a RdP não é viva. Os dois primeiros testes foram incluídos por forma a satisfazer o seguinte teorema [Mur89]:

Teorema 1 *Se uma RdP N é viva e segura, então não existem lugares fonte ou destino, transições fonte ou destino, i.e.,* $\forall x \in N : {}^{\cdot}x \neq \emptyset \neq x^{\cdot}$. □

Estes dois testes do tipo estrutural (não dependem da marcação inicial da RdP), são realizados, tomando directamente como entrada a especificação da RdP. O teste 3 garante que, a partir de uma dada marcação, é sempre possível disparar, pelo menos, uma transição,

resultando uma marcação diferente. O teste 4 assegura que toda a transição dispara, no mínimo uma vez, assegurando que a transição não é morta (ver definição 19).

Estes dois últimos testes, efectuados pelo módulo, são realizados com base no grafo de alcançabilidade, construído para o efeito. Porém, para o caso de uma RdPSI, a construção do respectivo grafo de alcançabilidade é feita de uma forma distinta, relativamente à apresentada na subsecção 2.3.1, devido ao facto de as RdPSI terem uma regra de disparo distinta relativamente ao modelo básico.

A partir do conjunto de transições habilitadas, ter-se-ão de considerar todas as combinações possíveis de condições associadas a essas transições.

Como exemplo, considere-se parte de uma RdPSI presente na figura 4.2(a). Admita-se que, num dado instante, apenas as transições *t1* e *t2* estão habilitadas. A primeira daquelas transições está guardada pela variável de entrada A, enquanto que a última é guardada pela variável de entrada B. As quatro combinações possíveis das variáveis A e B, bem como as respectivas transições habilitadas são apresentadas no quadro 4.1. Na figura 4.2(b) apresenta-se uma parte do grafo de cobertura, correspondente ao instante assumido, da RdPSI considerada.

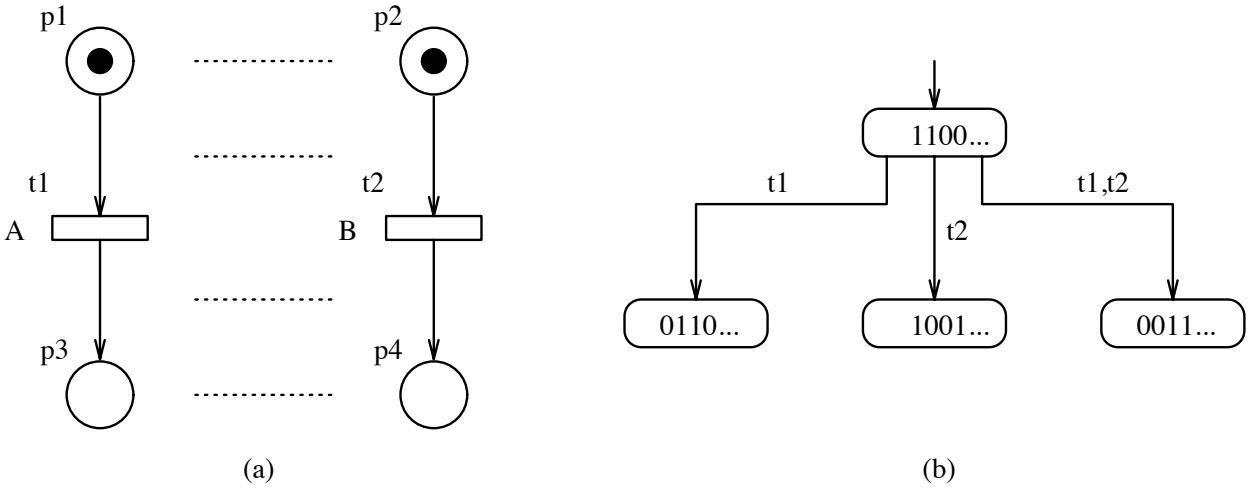


Figura 4.2: (a) Parte de uma RdPSI e (b) parte do respectivo grafo de cobertura.

A	B	Transições habilitadas
0	0	Nenhuma
0	1	t2
1	0	t1
1	1	t1 e t2

Quadro 4.1: Combinações das variáveis A e B e transições habilitadas.

Os testes atrás referidos são aplicados às partes do controlador de forma separada, ou seja, cada subcontrolador é testado individualmente. Os arcos inibidores ou habilitadores provenientes de lugares de outras partes, caso existam, são interpretados como sinais de entrada "normais".

Em relação aos macronodos, não é feita a sua expansão para efeitos de análise. Assim, a análise das propriedades é feita exclusivamente ao nível mais alto de abstracção. Para resolver esta questão, os resultados apresentados por Valette [Val79] ou Suzuki e Murata [SM83] deixam antever que é possível analisar, separadamente, a RdP mais abstracta e os macronodos e extrapolar os resultados para a RdP mais refinada. Outra solução, menos elegante, consiste em linearizar toda a RdP e aplicar os testes analíticos à RdP resultante. Esta solução tem a agravante de se aplicar a uma RdP potencialmente grande, o que torna a análise bastante demorada, perdendo-se assim algumas das vantagens resultantes da utilização de hierarquia.

4.5 Compilação para VHDL

A existência de um módulo de compilação —que recebe uma entrada escrita numa linguagem fonte (ConPar) e gera uma saída numa linguagem destino (VHDL)— sugere uma pequena abordagem sobre a teoria da compilação, dando especial relevo às gramáticas de atributos. De seguida apresentam-se os produtos utilizados para o desenvolvimento do módulo e discutem-se as opções implementadas neste.

4.5.1 Teoria da Compilação

De uma forma simplista, um compilador[2] é uma aplicação que recebe um programa escrito numa dada linguagem — a linguagem fonte — e o converte para um outro programa numa outra linguagem — a linguagem destino. O compilador deve ainda assinalar todos os erros encontrados no programa fonte [ASU86].

A tarefa de compilação divide-se em duas fases: análise e síntese. A fase de análise divide o programa fonte em pequenos pedaços e gera uma representação intermédia desse programa (nomeadamente uma árvore sintática). A fase de síntese gera o programa destino, a partir da representação intermédia obtida na fase anterior.

Cada uma das fases acima descrita é normalmente dividida em tarefas. A fase de análise engloba as seguintes: análise léxica, análise sintática, análise semântica e geração de código intermédio.

Estas tarefas dependem principalmente da linguagem fonte, e deseja-se que sejam o mais independentes possível da linguagem destino.

A fase de síntese, inclui as seguintes tarefas, que dependem da linguagem destino e da representação intermédia escolhida, mas são independentes da linguagem fonte: optimização de código e geração de código.

Cada uma destas seis tarefas recebe uma dada representação do programa fonte e altera essa representação ou cria uma nova, que servirá de entrada à tarefa seguinte. Faz-se, de seguida, uma curta descrição de cada uma destas tarefas.

A *análise léxica* é a primeira tarefa do processo de compilação. A sua função consiste em ler,

[2]Entenda-se o termo "compilador" no seu sentido mais lato, abrangendo, entre outros, tradutores, pré-processadores, interpretadores, *assemblers*, *pretty-printers*, etc.

da esquerda para a direita, os caracteres que formam o texto do programa fonte e agrupá-los em conjuntos com um dado significado colectico (palavras-chave, identificadores, constantes, sinais de pontuação, etc), a que usualmente se chama *lexemas*. Estes lexemas constituem o vocabulário da linguagem. Os vários lexemas encontrados, durante esta tarefa, são passados à tarefa seguinte.

A *análise sintática* recebe a sequência de lexemas e verifica se ela forma ou não uma frase válida da linguagem. Para definir quais as frases sintaticamente válidas da linguagem fonte, é comum usar-se uma gramática independente do contexto. Deste modo, determinar se uma sequência de lexemas é valida (i.e. se pertence à linguagem fonte) assenta em verificar se essa sequência é derivável a partir da gramática. Durante esta tarefa é habitual construir-se uma árvore que representa a estrutura sintática do programa fonte. Estas árvores são chamadas árvores de *derivação* ou *sintáticas*. Em resumo, esta tarefa transforma uma sequência de lexemas numa árvore sintática.

A *análise semântica* tem por função verificar se o programa segue as regras semânticas da linguagem fonte. Esta análise é feita, percorrendo a árvore sintática construída na fase anterior. A semântica das linguagens é, regra geral, definida por condições de contexto. Exemplos típicos de condições de contexto envolvem as chamadas *regras de escopo* e de *tipo*, ou seja, a obrigatoriedade de declarar um identificador antes de o utilizar e a validação de tipos nas expressões. É frequente, para efectuar estes testes, recorrer-se a uma estrutura de dados (uma tabela de símbolos) que guarda informação sobre os vários identificadores utilizados.

A *geração de código intermédio* tem por finalidade gerar uma representação intermédia do programa fonte. Esta representação pode ser, posteriormente, convertida para um programa escrito em qualquer linguagem destino. Pretende-se que a geração do código intermédio seja o mais simplificada possível, sendo também desejável que a conversão para qualquer linguagem destino seja possível e se faça de uma forma fácil. Nesta tarefa é também feita a *optimização* deste código, que é *independente da máquina*, já que o código intermédio é melhorado sem se ter em consideração qualquer característica da máquina destino.

A *optimização de código* tem a função de melhorar o código intermédio produzido na fase anterior. Geralmente, esta optimização refere-se à alocação de registos e utilização de sequências de instruções-máquina especiais. Nesta tarefa a optimização é *dependente da máquina*, ao contrário do que sucedia na tarefa anterior.

A *geração de código* é a última tarefa do processo de compilação. É nesta tarefa que se produz o programa destino, sendo feitas, por exemplo, a escolha dos modos de endereçamento, a selecção da instrução-máquina e a gestão de registos.

Em termos práticos, não existe uma separação tão clara entre as tarefas atrás descritas. Geralmente, agrupam-se as fases que estão directamente relacionadas num mesmo bloco. Como ilustrado na figura 4.3, as análises léxica e sintática estão agrupadas num bloco que se designa por *Estruturação*, pois estas tarefas transformam o programa fonte numa outra representação (geralmente uma árvore), que define a estrutura sintática daquele programa. A análise semântica e a geração de código intermédio estão agrupadas no bloco *Tradução*, pois é função destas tarefas traduzir a linguagem fonte para uma representação intermédia. O bloco *Codificação* é constituído pelas tarefas de optimização e geração de código, que se relacionam apenas com a linguagem destino, e cuja função consiste em gerar código nesta linguagem.

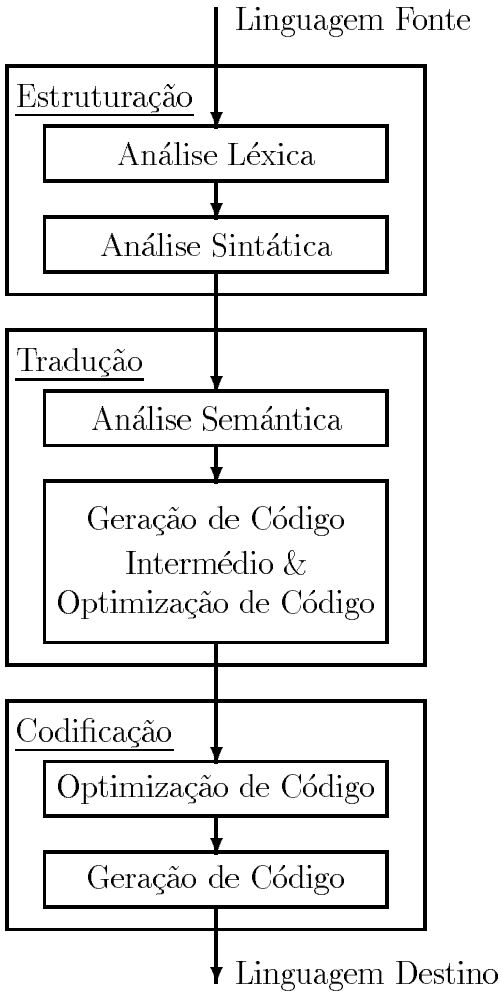


Figura 4.3: As tarefas da Compilação.

Uma das vantagens que resulta da divisão do processo de compilação em tarefas é o de permitir que se desenvolvam metodologias específicas para cada tarefa, de forma a construir ferramentas que, baseadas numa especificação, geram de uma forma automática procedimentos que implementam cada uma das tarefas. Um *sistema gerador de compiladores* é composto por um conjunto dessas ferramentas, produzindo implementações para cada tarefa do processo de compilação. Uma grande parte da implementação de um compilador pode, deste modo, ser gerada a partir de especificações, libertando o programador da tarefa morosa de desenvolver todo o compilador "manualmente".

A implementação de um compilador pode ser feita, utilizando duas abordagens distintas: a *tradução dirigida pela sintaxe* ou a *tradução dirigida pelos atributos*, baseadas respectivamente em *gramáticas tradutoras* e *gramáticas de atributos*. Na primeira abordagem, sempre que o analisador sintático reconhece uma produção da gramática, é feito o cálculo dos atributos, a análise semântica e a geração de código intermédio. Portanto, não é necessário construir toda a árvore sintática, uma vez que as tarefas que constituem o processo de tradução são executadas durante a análise sintática. Sendo assim, quando se reconhece o último lexema do texto fonte, a geração de código intermédio termina. Deste modo, as fases da análise semântica e da geração de código intermédio estão repartidas por vários procedimentos de pequena dimensão — as acções semânticas — e intercalados no analisador sintático.

A segunda abordagem, a compilação baseada em gramática de atributos, é conceptualmente mais próxima da estrutura apresentada na figura 4.3. Tal como na abordagem anterior, existe um analisador léxico e um analisador sintático. No entanto, a análise semântica e a geração de código intermédio não são feitas durante o reconhecimento sintático. Durante a análise sintática é apenas construída a árvore sintática. Posteriormente, e após a árvore estar totalmente construída, procede-se ao cálculo dos atributos e implementam-se as tarefas seguintes. Nesta abordagem, as tarefas de análise semântica e de geração de código intermédio são implementadas por um procedimento, designado por calculador de atributos, que determina o valor dos atributos. Este procedimento pode ser derivado automaticamente, a partir da gramática de atributos.

Gramáticas de atributos

As gramáticas de atributos já provaram ser um método apropriado para a especificação de computações em estruturas arbóreas. As áreas de aplicação das gramáticas de atributos são variadas, destacando-se a análise e tradução de linguagens de programação e a construção de compiladores e editores [Kas91].

Uma gramática de atributos consiste numa extensão à gramática independente do contexto, em que a cada símbolo da gramática são associados vários atributos. Os valores dos atributos são definidos por regras de cálculo associadas às produções da gramática independente do contexto. Estas regras de cálculo, especificam como se determinam os valores de uns atributos em função de outros atributos, definindo assim dependências entre os vários atributos [Alb91].

As gramáticas de atributos definem a estrutura das árvores correspondentes às frases da linguagem, os atributos associados aos nodos da árvore e as regras de cálculo. Os nodos da árvore podem ser vistos como estruturas com campos para guardar informação, que corresponde ao conjunto de valores possíveis que podem ser atribuídos aos atributos. Os

nomes dos campos correspondem aos atributos que os nodos contêm. Os atributos são divididos em dois subconjuntos disjuntos: os *atributos sintetizados* e os *atributos herdados*. Estes guardam informação relativa à parte da árvore que envolve o nodo ao qual estão associados. Os primeiros contêm informação da sub-árvore que tem raiz no nodo ao qual estão associados.

Um procedimento, que é designado por *calculador de atributos*, calcula as instâncias de atributos associadas aos vários nodos de uma árvore sintática. A ordem, pela qual as instâncias dos atributos são calculadas, é determinada com base nas suas regras de cálculo. A gramática de atributos impõe, implicitamente, uma ordem para o cálculo dos atributos, já que só é possível calcular o valor dum atributo, depois de todos os atributos de ele que depende já o terem sido.

Depois do cálculo dos atributos, o significado da árvore sintática é dado pelo valor dos atributos sintetizados associados à raiz dessa árvore.

As características mais frequentemente citadas das gramáticas de atributos são o seu carácter declarativo, o não determinismo e a localidade [Sar93]. O *carácter declarativo* resulta do facto de as gramáticas de atributos declararem os valores, mas não indicarem os passos a seguir para o seu cálculo. O *não determinismo* verifica-se, já que é possível, para uma mesma gramática de atributos, determinar mais do que uma ordem válida para o cálculo dos atributos. Por fim, a *localidade* advém do facto de uma regra semântica associada a um atributo de um símbolo definir um valor que apenas depende de valores de atributos associados a símbolos da mesma produção. Por este motivo, não é possível a utilização de variáveis globais.

Uma definição formal das gramáticas de atributos pode encontrar-se em [Alb91].

4.5.2 Técnicas e Produtos Usados

Como resultado da investigação realizada neste trabalho, pretendia-se desenvolver uma aplicação computacional que incluísse os módulos de análise de propriedades e de compilação, descritos na figura 4.1. A aplicação devia produzir, como resultado de maior valia, o código VHDL correspondente ao controlador.

A primeira decisão, para realização da dita aplicação, consistiu em usar uma ferramenta de desenvolvimento, em vez de desenvolver toda a aplicação manualmente[3]. As principais vantagens na utilização de uma ferramenta de desenvolvimento são [GE90]:

- Não ser necessário escrever um programa completo, mas apenas uma especificação;
- As ferramentas realizam vários e distintos testes de consistência à especificação;
- Escrever especificações, que são testadas automaticamente, reduz drasticamente o número de erros e aumenta a fiabilidade da aplicação desenvolvida.

Assim sendo, havia que escolher qual a ferramenta de desenvolvimento a usar. Os critérios mais importantes que determinaram a escolha da ferramenta seriam:

[3] Em [Wai93] pode encontrar-se um exemplo de um compilador produzido a partir de especificações. Nesse artigo, mostra-se como comparar a implementação "manual" de um compilador, com a sua implementação "automática", recorrendo a ferramentas próprias.

- O tempo de desenvolvimento, que se pretendia o mais curto possível;
- A portabilidade da aplicação gerada, que se desejava a maior possível.

Surgiram três alternativas possíveis, todas elas disponíveis no Departamento de Informática da Universidade do Minho:

1. *Lex & Yacc* [LMB90];
2. *Synthesizer Generator* (SG) [RT89].
3. *Compiler Tool Box* (CTB) [GE90];

A hipótese de utilização do *Lex & Yacc* só permite cobrir, de uma forma integrada, as tarefas relativas às análises léxica e sintática, enquanto que, se for utilizado um sistema gerador de compiladores, este já disponibiliza ferramentas que cobrem um maior número de tarefas de compilação.

O sistema SG permite gerar editores dirigidos pela sintaxe de uma determinada linguagem. Os editores são especificados na linguagem SSL (*Synthesizer Specification Language*), construída com base nos conceitos das gramáticas de atributos. A utilização deste sistema levanta um problema: o produto gerado fica dependente do interface gráfico, o que é um factor bastante limitativo relativamente ao grau de portabilidade pretendido. Além disso é sempre gerado um editor de texto dirigido pela sintaxe, o que retira, na fase de digitação dos ficheiros, alguma liberdade ao utilizador.

O CTB, a partir de diferentes especificações, gera automaticamente rotinas que efectuam as análises léxica, sintática e semântica. Para a análise léxica, utiliza-se o gerador de analisadores léxicos *Rex* [Gro87] (incluído no CTB), que produz analisadores léxicos, a partir de expressões regulares[4].

O gerador de analisadores sintáticos *Ell/Lalr* [Vie88] (incluído no CTB), produz automaticamente, a partir de gramáticas independentes de contexto, analisadores sintáticos[5], que incluem características importantes como, por exemplo, recuperação de erros [Gro89]. Finalmente, existe o programa *AE* que gera, de forma automática, um calculador de atributos, a partir de uma gramática de atributos. Esse calculador efectuará a tarefa de análise semântica. O CTB dispõe ainda de outras ferramentas que não foram usadas no contexto deste trabalho, pelo que não são aqui referidas.

Pelas razões atrás apresentadas, a escolha recaiu sobre o Compiler Tool Box (CTB). Este sistema permite desenvolver compiladores eficientes, disponibilizando ferramentas que abrangem a quase totalidade das tarefas de compilação (ver subsecção 4.5.1). As especificações utilizam uma linguagem própria [Gro87, SF91] e geram código em linguagem C [KR88] ou MODULA.

Com este sistema, as tarefas de análises léxica e sintática são cobertas na totalidade, tal como acontece com o *Lex & Yacc*. No entanto, além destas tarefas, o CTB também cobre a tarefa de análise semântica, através da implementação de um calculador de atributos. Todas as rotinas geradas pelo CTB são facilmente integráveis, já que foram pensadas em

[4] O *Rex* gera, em casos típicos, analisadores léxicos 5 vezes mais rápidos e 5 vezes mais pequenos que os criados pelo *Lex* [Gro87].

[5] O *Ell/Lalr* produz analisadores sintáticos mais rápidos que os produzidos pelo *Yacc* — entre 2 e 3 vezes. No entanto, o tamanho dos analisadores sintáticos produzidos pelo *Ell/Lalr* é ligeiramente maior [GE90].

conjunto, o que elimina os problemas que podem surgir, quando se usam ferramentas de sistemas distintos. Por exemplo, as rotinas que implementam a análise léxica devolvem os lexemas de forma a poderem ser usados, com extrema facilidade, pelas rotinas que executam a análise sintática.

Finalmente, o CTB disponibiliza ainda algumas funções que permitem manipular identificadores, fazer gestão da memória, alocar vectores dinamicamente e gerir uma tabela de cadeias de caracteres [Sar92].

No presente trabalho, tanto o código gerado pela ferramenta como o produzido manualmente foi em linguagem C, pelo que é possível portar, com relativa facilidade, a aplicação produzida.

4.5.3 Opções Implementadas

A aplicação detecta, a partir de um ficheiro de entrada, todo o tipo de erros léxicos, sintáticos e semânticos. São, em seguida, aplicados os algoritmos de análise à especificação de entrada, para verificar se existe algum problema. Após a fase de análise (ver subsecção 4.4) é gerado, pela aplicação ConPar, o código em linguagem VHDL que descreve o controlador de entrada. O tipo de código a gerar pode ser escolhido pelo utilizador. Uma primeira opção consiste em escolher se se deseja um bloco ou um processo para determinar qual a marcação inicial (quando o sinal `RESET` estiver activo) e qual a próxima marcação (quando surgir um pulso no sinal de relógio). A geração de um bloco permite que o código seja utilizado, por exemplo, nas ferramentas ALLIANCE.

Como justificado na subsecção 3.2.3, utilizou-se a metodologia apresentada por Pardey e Bolton, como base para o presente trabalho, embora se tenham feito ligeiras alterações ao estilo das descrições em código VHDL, de molde a poder seguir as ideias presentes neste trabalho.

Na metodologia de Pardey e Bolton, foi usada a regra de disparo fraca, ou seja não são testadas as capacidades dos lugares de saída, para que uma transição seja considerada habilitada. Este facto implica que a equação, presente no texto 3.13, que descreve a transição $T5$ da figura 3.6 é dada pela seguinte equação:

```
T5 <= NOT X3 AND P4 AND P5;
```

Repare-se que a transição fica habilitada a disparar (o sinal `T5` fica activo), quando a guarda é verdadeira (`NOT X3`) e os lugares de entrada $P4$ e $P5$ estão marcados (os sinais `P4` e `P5` estão activos). Nessa situação, o lugar $P1$ será marcado no próximo pulso activo do relógio, independentemente de estar ou não marcada, como se comprova pela seguinte equação:

```
NP1 <= T5 OR (P1 AND NOT T1);
```

São também incluídos comandos `ASSERT` que verificam se os lugares são seguros, se há alguma transição habilitada a disparar, se os sinais de saída são invocados deterministicamente e se há transições em conflito, permitindo assim ao utilizador detectar erros de concepção, durante a simulação.

Na metodologia seguida neste trabalho, a equação relativa à transição $T5$, considerada anteriormente, é dada pela seguinte equação:

```
T5 <= NOT X3 AND P4 AND P5 AND NOT P1;
```

Equivale isto a afirmar que a transição $T5$ só está habilitada a disparar se a guarda for verdadeira (`NOT X3`), os lugares de entrada $P4$ e $P5$ estiverem marcados (os sinais `P4` e `P5` estiverem activos) e o lugar de saída estiver livre (o sinal `P1` estiver inactivo). Este facto está de acordo com a regra 1 apresentada na secção 2.5 (regra de disparo forte).

Dos comandos `ASSERT` considerados por Pardey e Bolton, foram incluídos os relativos aos conflitos entre transições e à inexistência de transições habilitadas. Não foram incluídos, por não se considerar fundamental na defesa da presente tese, os comandos para verificar se a RdPSI é determinística. Em relação à segurança da RdP, o facto de se utilizar uma regra de disparo distinta, torna desnecessária a inclusão dos comandos `ASSERT` para testar essa propriedade.

Para controladores de reduzida complexidade, é comum usar-se apenas uma RdPSI para os especificar. Contudo, o controlador, a especificar, pode ser dividido pelo número de subcontroladores, que se ache adequado. Cada subcontrolador tem os seus próprios nodos (lugares e transições), que não podem ser partilhados, ou seja, um nodo não pode pertencer a mais do que um nodo. A ligação entre os diversos subcontroladores pode ser feita através de duas alternativas: as guardas associadas às transições, ou mediante a existência de arcos habilitadores ou inibidores a ligarem um lugar de um subcontrolador a uma transição de um outro subcontrolador (ver subsecção 2.4.3). Repare-se que se for usado um arco "normal" entre um lugar de um subcontrolador e uma transição de um outro subcontrolador, esses dois subcontroladores formam, na realidade, um único controlador. Cada subcontrolador é especificado, na linguagem ConPar por uma *parte*. Os nomes dos lugares, das transições, dos sinais de entrada ou saída especificados numa parte são considerados globais. Os sinais de entrada e saída são usados para definir a interface do controlador (`entity` em terminologia VHDL).

Como exemplo de um controlador especificado com base em duas RdPSI, considere-se a figura 4.4. A especificação deste controlador em ConPar encontra-se no texto A.5, em apêndice.

É também possível usar macronodos, para auxílio na fase de especificação do controlador. Assim, um nodo de uma RdPSI pode encapsular uma subrede. A aplicação admite apenas macrolugares, embora a linguagem tenha sido definida, de forma a já contemplar macrotransições.

Um macronodo pode ser visto como uma subrede parametrizável e re-utilizável[6], pois são indicados genericamente os seus sinais de entrada e saída, vistos como parâmetros. Cada utilização do macronodo terá de concretizar os parâmetros.

Considere-se, como exemplo, a RdP N e o macrolugar $mp1$ apresentados na figura 4.5. A concretização do macrolugar $mp1$, no âmbito da RdP N, é feita através do lugar $p1$, usando-se os sinais de entrada `X1` e `X2` e os sinais de saída `Y1` e `Y2`, em lugar dos parâmetros. Repare-se que seria possível, usar o macrolugar $mp1$, na RdP N, as vezes que fosse necessário. Este

[6]Uma visão alternativa permite considerar os macronodos como algo idêntico às rotinas, funções e procedimentos existentes nas linguagens de alto nível

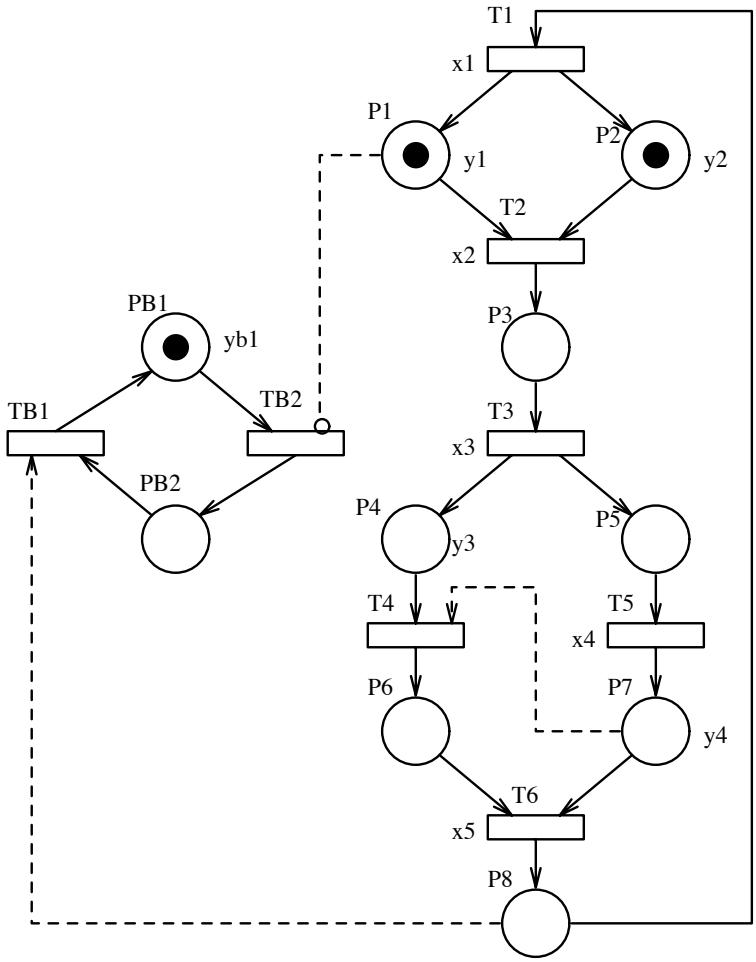


Figura 4.4: RdPSI especificada por duas partes.

facto obriga a acrescentar, no início dos nomes dos lugares e das transições do macrolugar, o nome do lugar $p1$.

Cada macronodo tem de conter apenas uma porta de entrada e uma porta de saída. A linearização da RdP é conseguida ligando-se as chaves de entrada do macronodo à porta de entrada e as chaves de saída do macronodo à porta de saída. A RdP resultante da linearização é apresentada na figura 4.6.

Por questões de versatilidade, a aplicação desenvolvida permite que o utilizador escolha se pretende seguir a metodologia apresentada nesta tese (utilizando a regra de disparo forte), ou a metodologia apresentada por Pardey e Bolton. Caso se opte por esta segunda alternativa, apenas é testada a RdPSI relativamente à existência de conflitos. Os restantes testes não são aplicados, pois seria necessário considerar algoritmos distintos, o que não se enquadra no contexto desta tese. O código VHDL gerado é distinto, como atrás se referiu, incluíndo os comandos `ASSERT` considerados, com excepção para os que testam se a RdP é determinística.

4.6 Exemplo de Aplicação

Nesta secção, pretende-se mostrar a viabilidade da metodologia seguida, testando o funcionamento da aplicação ConPar desenvolvida. Para esse fim, utilizar-se-á, como exemplo, o

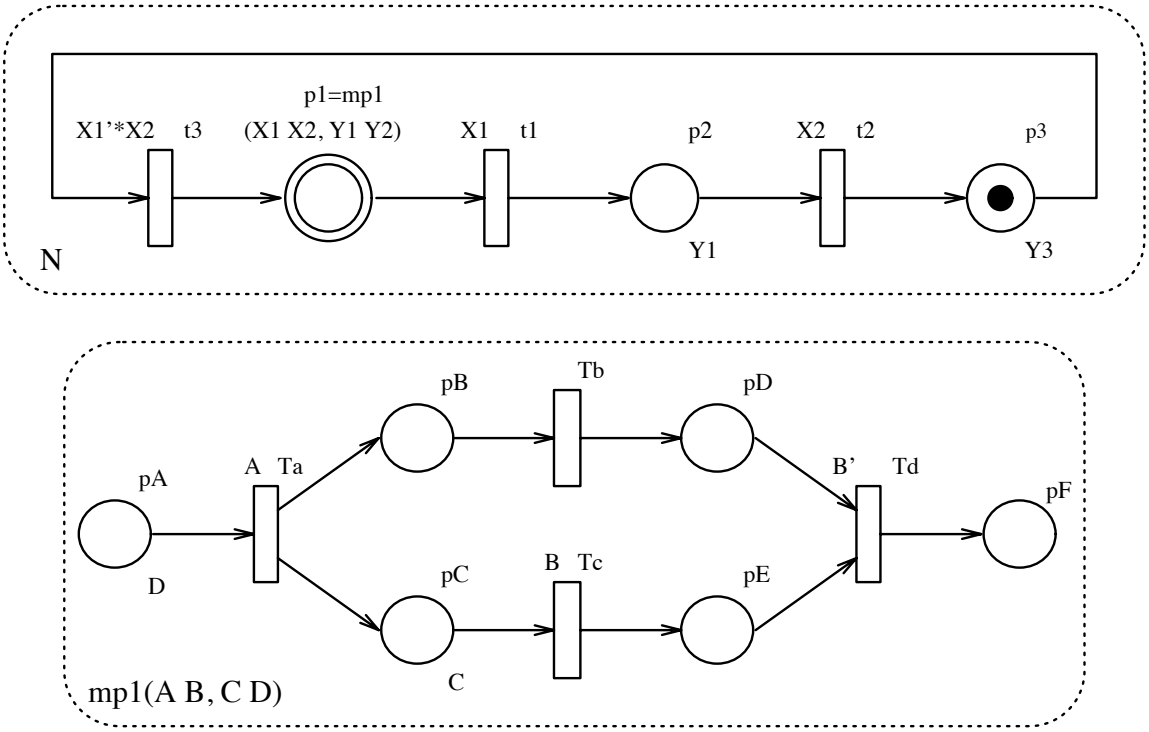


Figura 4.5: RdPSI especificada usando um macrolugar.

reactor encontrado em [Ada87].

4.6.1 O Sistema Reactor

O reactor permite controlar o funcionamento de um sistema de mistura e transporte de reagentes — ver figura 4.7(a). Os silos $S1$ e $S2$, onde estão armazenados os reagentes, alimentam os dois contentores $C1$ e $C2$, respectivamente. O conteúdo destes é despejado para o tanque T, onde são misturados, através do accionamento da ventoinha V (pás de uma misturadora). Quando não houver mais reagentes nos contentores, há que despejar a mistura para o carro, donde será descarregada, quando aquele atingir o fim da pista.

Os sinais `AbreS1` e `AbreS2` permitem accionar as válvulas que controlam a queda dos reagentes nos contentores. O controlo da queda dos reagentes para o tanque é feito pelos sinais `AbreC1` e `AbreC2`. Os sinais `C1Cheio` e `C2Cheio` (`C1Vazio` e `C2Vazio`) indicam quando os contentores $C1$ e $C2$, respectivamente, estão cheios (vazios). O sinal `TVazio` indica quando o tanque T se encontra vazio. Para accionar a ventoinha, deve-se activar o sinal $RodaV$, quando o sinal `TLimite`, estiver activo. Este sinal indica quando há suficientes reagentes para que a mistura possa ser remexida. Para despejar a mistura do tanque para o carro, tem de se activar o sinal `AbreT`, que acciona a válvula.

O movimento do carro é controlado pelos sinais `AvancaC` e `RecuaC`. O sinal `DespejaC` permite abrir a válvula colocada no fundo do carro, o que faz despejar o seu conteúdo. O sinal `CVazio` permite saber, quando já não há mais mistura no carro. Finalmente, os sinais `InicioPista` e `FimPista` indicam quando o carro atingiu os extremos da pista, momento em que se deve finalizar o seu movimento.

O processo inicia-se (ou continua), sempre que o sinal `Inicia` estiver activo. Adicional-

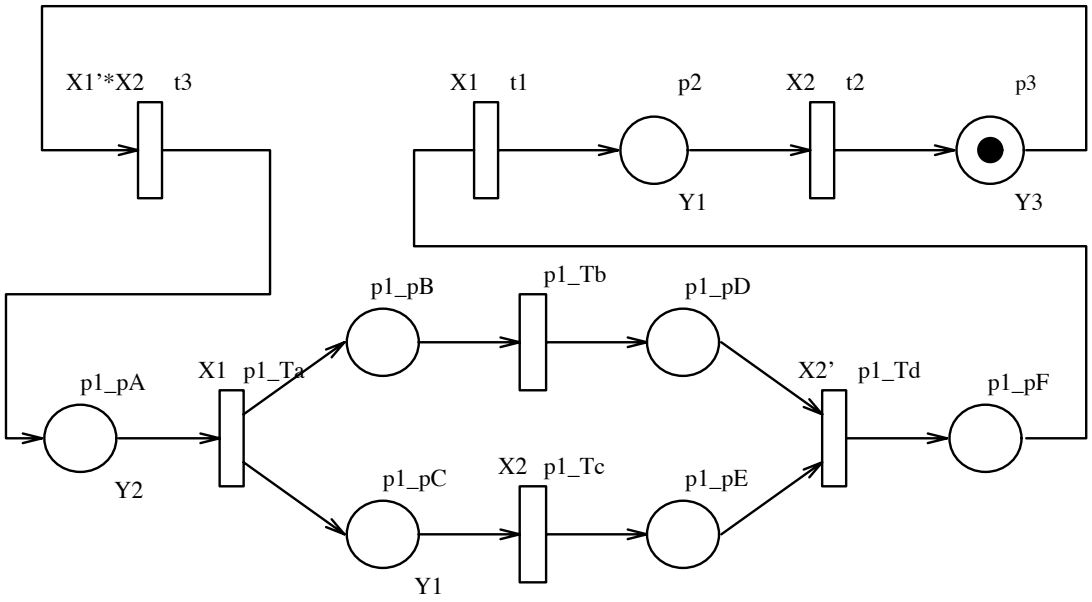


Figura 4.6: RdPSI após linearização do macrolugar.

mente, são considerados os sinais `Reset` (para iniciar o sistema de controlo) e `Relogio` (para sincronizar o controlador).

O sistema de controlo é representado pelo diagrama esquemático ilustrado na figura 4.7(b). A RdP Interpretada que descreve o comportamento do controlador é mostrada na figura 4.8.

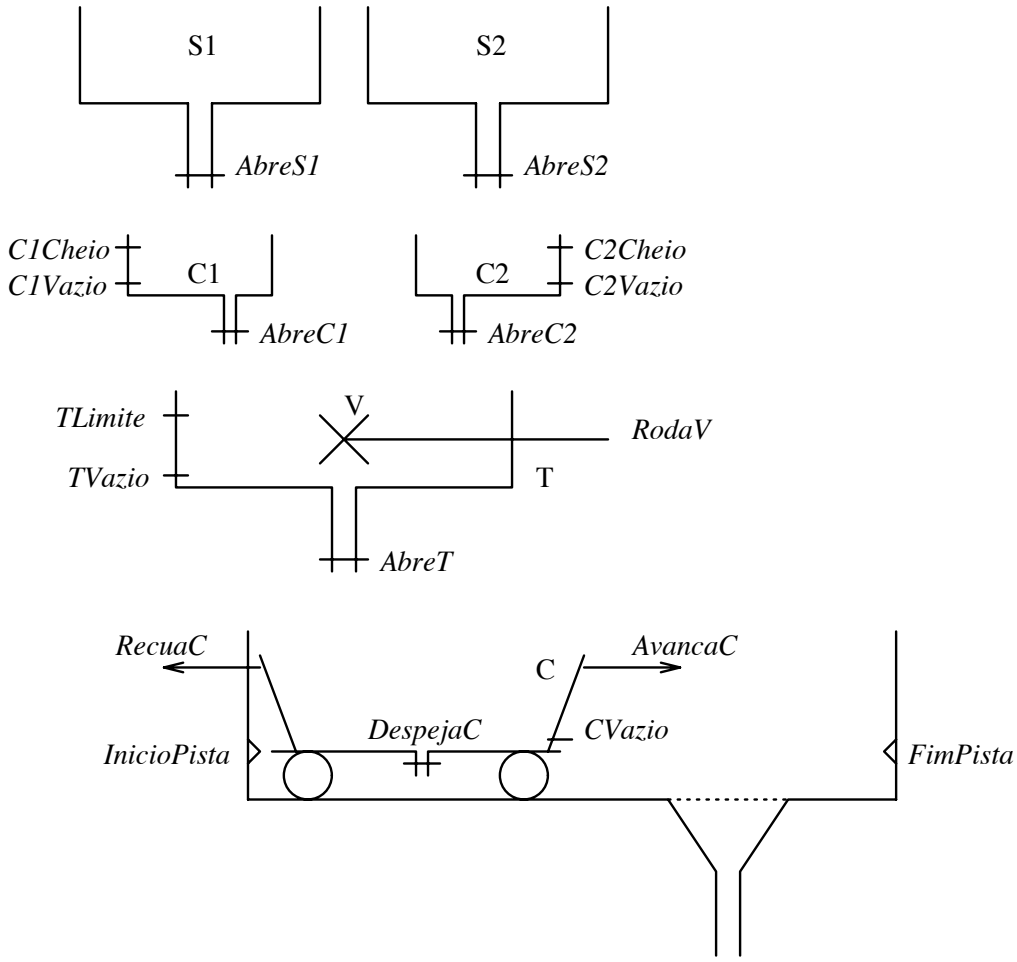
4.6.2 Utilização do Ambiente de Desenvolvimento

O código em linguagem ConPar relativo ao controlador do reactor pode ser introduzido em qualquer editor de texto e encontra-se listado no texto 4.1. A utilização deste código, como entrada na aplicação ConPar, produz os resultados apresentados no texto 4.2. O único problema detectado, durante a fase de análise, e reportado ao utilizador é relativo a um eventual conflito, originado por uma situação similar à apresentada na figura 2.19(a).

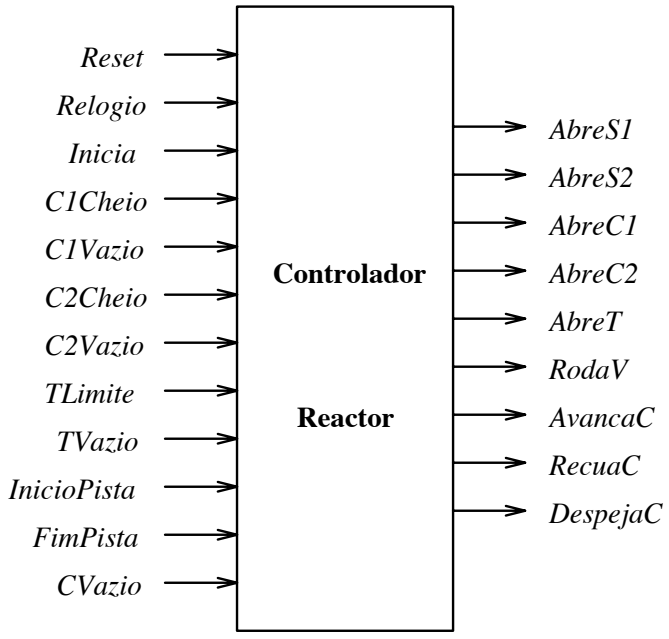
O facto de as transições *t5* e *t11* partilharem o lugar de entrada *p8*, produz uma mensagem de aviso. Repare-se que as transições em causa estão guardadas pelos predicados `TLimite` e `TVazio`, respectivamente. Contudo, pode-se constatar na figura 4.7, que é fisicamente impossível esses dois sinais estarem simultaneamente activos.

Por outro lado, as transições *t4* e *t6* partilham o lugar de saída *p8*, pelo que não podem estar simultaneamente habilitadas. Esta situação parece, à primeira vista, possível de suceder, uma vez que a transição *t4* não é guardada por qualquer sinal e a transição *t6* está guardada pelo predicado `TLimite'`. No entanto, uma análise atenta ao grafo de alcançabilidade permite concluir que tal nunca se verificará, uma vez que as duas transições nunca estarão simultaneamente habilitadas.

No texto 4.3 apresenta-se parte do conteúdo do ficheiro `Graph` gerado pela aplicação. Esse ficheiro mostra, em formato tipo vector, o grafo de alcançabilidade relativo à RdPSI. É também possível gerar o grafo, enumerando os lugares marcados e as transições disparadas, como se ilustra no texto 4.4.



(a)



(b)

Figura 4.7: Reactor. (a) Sistema de Mistura e Transporte. (b) Esquemático do Controlador.

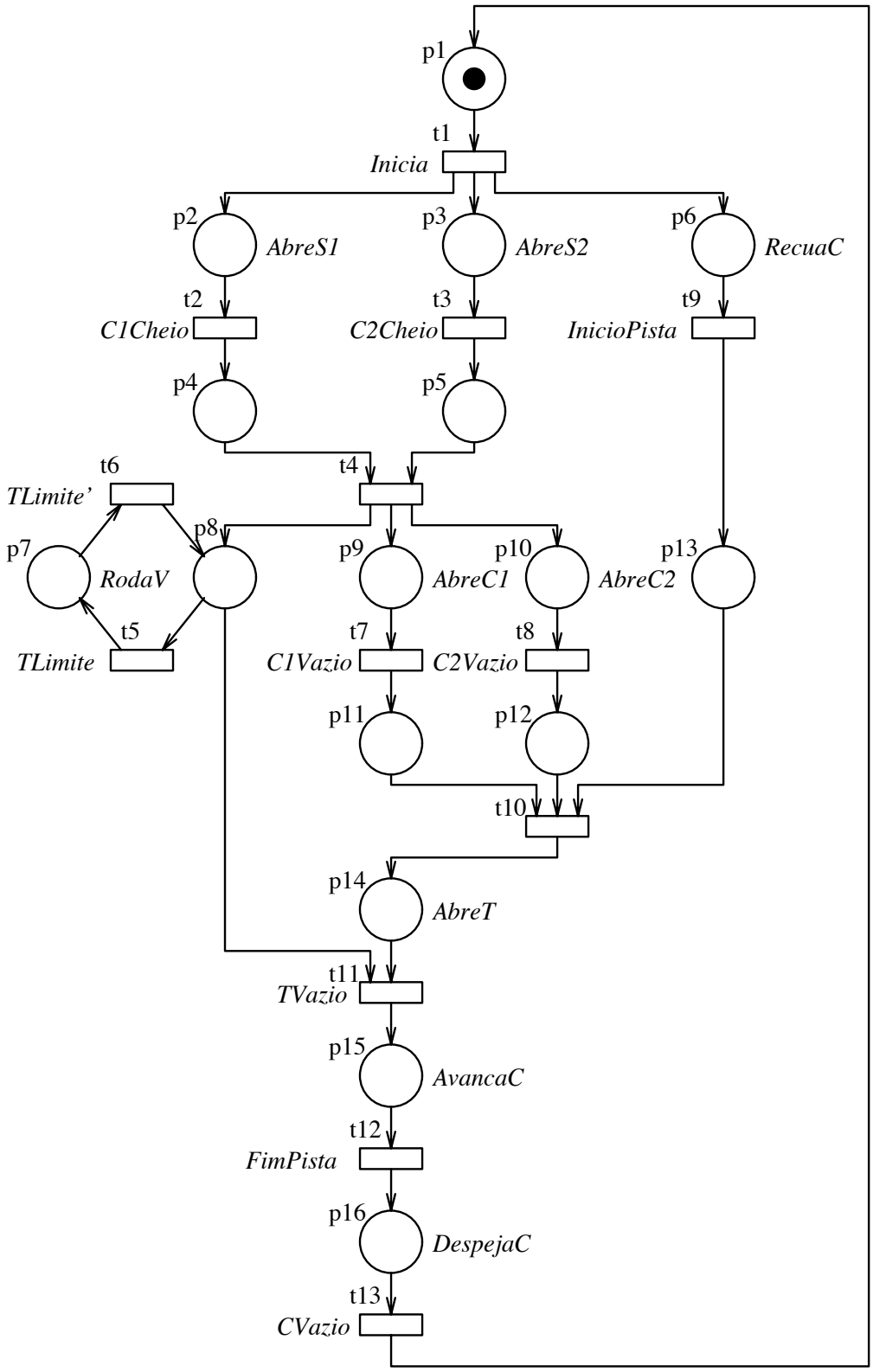


Figura 4.8: RdPSI para especificação do reactor.

Se se acrescentar, na RdPSI, um arco do lugar *p4* para a transição *t3*, a RdP resultante não é viva, uma vez que mais nenhuma transição fica habilitada a disparar, quando os lugares marcados forem *p5* e *p13*. Os resultados reportados são apresentados no texto 4.5. O grafo de alcançabilidade é gerado (ver texto 4.6), o que permite verificar as marcações produzidas até se atingir a situação de bloqueio.

Se, por outro lado, for adicionado um arco do lugar *p11* para a transição *t8*, a RdP resultante também não é viva, já que, a partir do momento em que a transição *t8* disparar, as únicas transições habilitadas são *t5* e *t6*. Os testes implementados detectam que há bloqueio, pois as transições *t10*, *t11*, *t12* e *t13* nunca foram disparadas. Os resultados reportados são apresentados no texto 4.7.

Como a fase de análise, para a RdPSI inicial, não reportou qualquer erro grave, é produzido o código VHDL[7] apresentado no texto 4.8. Foi usada a opção de compilação "+B" de forma a criar um bloco, em vez de um processo (ver apêndice B). Desta forma, é possível usar os produtos disponíveis no produto ALLIANCE. Ao ficheiro criado, deu-se o nome de `reactor.vbe`[8]. Mais exemplos de código VHDL gerado, a partir de especificações em linguagem ConPar, usando a aplicação desenvolvida, são apresentados no apêndice B.

Podem ainda ser gerados, caso o utilizador assim o pretenda, vários ficheiros com informação respeitante à especificação em linguagem ConPar. No exemplo apresentado no texto 4.2, além dos ficheiros com o código VHDL (`Tree`) e o grafo de alcançabilidade (`Graph`), foram gerados ficheiros com a lista de todos identificadores usados (`Ident`), a tabela de símbolos globais (`Table`) e a árvore sintática (`Tree`).

A utilização do ficheiro `reactor.vbe`, juntamente com um ficheiro contendo vectores de teste, permite simular o funcionamento do controlador. O ficheiro com vectores de teste, de nome `reactor.pat`, encontra-se listado no texto 4.9. A simulação, usando o simulador ASIMUT [SGV93], foi executada, tendo apenas dado mensagens de aviso relativas ao facto de não haver, em determinadas situações, nenhuma transição habilitada a disparar (ver texto 4.10).

Após a simulação, o passo seguinte é a síntese do controlador. O sintetizador LOGIC, incluído no produto ALLIANCE, permite obter uma descrição estrutural a partir de uma especificação ao nível fluxo de dados. As mensagens resultantes da execução do sintetizador encontram-se no texto 4.10. O ficheiro `reactor.vst`[9] criado encontra-se listado no texto 4.11.

[7]Foram apagadas algumas linhas (assinaladas com `...`) do ficheiro produzido, para encurtar o seu tamanho, sem contudo diminuir a sua legibilidade.

[8]O nome por defeito é `Code`.

[9]Uma vez que este ficheiro continha 1004 linhas de código, apagaram-se algumas, por motivos de espaço.

```
<* Autor:     Joao Miguel Fernandes
   Data:      Julho-1993
   Projecto:  Tese de Mestrado
   Companhia: Universidade do Minho (Braga - Portugal)
*>

.clock RELOGIO
.input INICIA C1CHEIO C1VAZIO C2CHEIO C2VAZIO TLIMITE TVAZIO INICIOPISTA FIMPISTA CVAZIO
.output ABRES1 ABRES2 ABREC1 ABREC2 ABRET RODAV AVANCAC RECUAC DESPEJAC

.part REACTOR
.place p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 p12 p13 p14 p15 p16
.transition t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13

.net
t1:  p1  *             INICIA       |- p2 * p3 * p6;
t2:  p2  *             C1CHEIO      |- p4;
t3:  p3  *             C2CHEIO      |- p5;
t4:  p4  * p5                       |- p8 * p9 * p10;
t5:  p8  *             TLIMITE      |- p7;
t6:  p7  *            !TLIMITE      |- p8;
t7:  p9  *             C1VAZIO      |- p11;
t8:  p10 *             C2VAZIO      |- p12;
t9:  p6  *             INICIOPISTA  |- p13;
t10: p11 * p12 * p13                |- p14;
t11: p8  * p14 *       TVAZIO       |- p15;
t12: p15 *             FIMPISTA     |- p16;
t13: p16 *             CVAZIO       |- p1;

.MooreOutput
p2   |-  ABRES1;
p3   |-  ABRES2;
p6   |-  RECUAC;
p7   |-  RODAV;
p9   |-  ABREC1;
p10  |-  ABREC2;
p14  |-  ABRET;
p15  |-  AVANCAC;
p16  |-  DESPEJAC;

.marking p1
.e
```

Texto 4.1: Código ConPar para controlador do reactor.

```
> ConPar +I +T +R +W +B +O +G -M < reactor.cpar

================================================
ConPar Compiler: version 1.0
Copyright 1994 - J.M.Fernandes - U.M. - Braga - PORTUGAL
Email contact: miguel@di.uminho.pt
================================================

transition "t11" conflicts with transition "t5": shared input place

Petri Net Analysis is ok!

Reachability Graph written to file "Graph"
VHDL code written to file "Code"
Identifiers list written to file "Ident"
Table written to file "Table"
Parse Tree written to file "Tree"
```

Texto 4.2: Resultados produzidos pela aplicação ConPar durante a fase de análise.

```
INITIAL MARKING: 0000000000000001

PPPPPPPPPPPPPPP   ttttttttttttt    PPPPPPPPPPPPPPP
1111111987654321  1111987654321    1111111987654321
6543210           3210             6543210

1000000000000000 : 1000000000000 -> 0000000000000001
0100000000000000 : 0100000000000 -> 1000000000000000
0010000010000000 : 0010000000000 -> 0100000000000000
                 : 0000000010000 -> 0010000001000000
0010000001000000 : 0000000100000 -> 0010000010000000
0001110010000000 : 0001000010000 -> 0010000001000000
                 : 0001000000000 -> 0010000010000000
0001110001000000 : 0001000100000 -> 0010000010000000
                 : 0001000000000 -> 0010000001000000
0001100110000000 : 0000001010000 -> 0001110001000000
                 : 0000001000000 -> 0001110010000000
                 : 0000000010000 -> 0001100101000000
0001100101000000 : 0000001100000 -> 0001110010000000
                 : 0000001000000 -> 0001110001000000
                 : 0000000100000 -> 0001100110000000
0001011010000000 : 0000010010000 -> 0001110001000000
                 : 0000010000000 -> 0001110010000000
                 : 0000000010000 -> 0001011001000000
0001011001000000 : 0000010100000 -> 0001110010000000
                 : 0000010000000 -> 0001110001000000
                 : 0000000100000 -> 0001011010000000
0001001110000000 : 0000011010000 -> 0001110001000000
                 : 0000011000000 -> 0001110010000000
                 : 0000010010000 -> 0001100101000000
                 : 0000010000000 -> 0001100110000000
                 : 0000001010000 -> 0001011001000000
                 : 0000001000000 -> 0001011010000000
                 : 0000000010000 -> 0001001101000000
...
0000001101100000 : 0000111100000 -> 0001110010000000
                 : 0000111000000 -> 0001110001000000
                 : 0000110100000 -> 0001100110000000
                 : 0000110000000 -> 0001100101000000
                 : 0000101100000 -> 0001011010000000
                 : 0000101000000 -> 0001011001000000
                 : 0000100100000 -> 0001001110000000
                 : 0000100000000 -> 0001001101000000
                 : 0000011100000 -> 0000110010100000
                 : 0000011000000 -> 0000110001100000
                 : 0000010100000 -> 0000100110100000
                 : 0000010000000 -> 0000100101100000
                 : 0000001100000 -> 0000011010100000
                 : 0000001000000 -> 0000011001100000
                 : 0000000100000 -> 0000001110100000
0000000000111000 : 0000100001000 -> 0001001110000000
                 : 0000000001000 -> 0000001110100000
0000000000110010 : 0000100000010 -> 0001000000011000
                 : 0000100000000 -> 0001000000010010
                 : 0000000000010 -> 0000000000111000
0000000000101100 : 0000100000100 -> 0001000000011000
                 : 0000100000000 -> 0001000000001100
                 : 0000000000100 -> 0000000000111000
0000000000100110 : 0000100000110 -> 0001000000011000
                 : 0000100000100 -> 0001000000010010
                 : 0000100000010 -> 0001000000001100
                 : 0000100000000 -> 0001000000000110
                 : 0000000000110 -> 0000000000111000
                 : 0000000000100 -> 0000000000110010
                 : 0000000000010 -> 0000000000101100
0000000000000001 : 0000000000001 -> 0000000000100110
```

Texto 4.3: Grafo de alcançabilidade, em notação vectorial.

```
INITIAL MARKING: p1

p16               : t13  -> p1
p15               : t12  -> p16
p14 p8            : t11  -> p15
                  : t5  -> p14 p7
p14 p7            : t6  -> p14 p8
p13 p12 p11 p8    : t10 t5  -> p14 p7
                  : t10  -> p14 p8
p13 p12 p11 p7    : t10 t6  -> p14 p8
                  : t10  -> p14 p7
p13 p12 p9 p8     : t7 t5  -> p13 p12 p11 p7
                  : t7  -> p13 p12 p11 p8
                  : t5  -> p13 p12 p9 p7
p13 p12 p9 p7     : t7 t6  -> p13 p12 p11 p8
                  : t7  -> p13 p12 p11 p7
                  : t6  -> p13 p12 p9 p8
p13 p11 p10 p8    : t8 t5  -> p13 p12 p11 p7
                  : t8  -> p13 p12 p11 p8
                  : t5  -> p13 p11 p10 p7
p13 p11 p10 p7    : t8 t6  -> p13 p12 p11 p8
                  : t8  -> p13 p12 p11 p7
                  : t6  -> p13 p11 p10 p8
p13 p10 p9 p8     : t8 t7 t5  -> p13 p12 p11 p7
                  : t8 t7  -> p13 p12 p11 p8
                  : t8 t5  -> p13 p12 p9 p7
                  : t8  -> p13 p12 p9 p8
                  : t7 t5  -> p13 p11 p10 p7
                  : t7  -> p13 p11 p10 p8
                  : t5  -> p13 p10 p9 p7
...
p10 p9 p7 p6      : t9 t8 t7 t6  -> p13 p12 p11 p8
                  : t9 t8 t7  -> p13 p12 p11 p7
                  : t9 t8 t6  -> p13 p12 p9 p8
                  : t9 t8  -> p13 p12 p9 p7
                  : t9 t7 t6  -> p13 p11 p10 p8
                  : t9 t7  -> p13 p11 p10 p7
                  : t9 t6  -> p13 p10 p9 p8
                  : t9  -> p13 p10 p9 p7
                  : t8 t7 t6  -> p12 p11 p8 p6
                  : t8 t7  -> p12 p11 p7 p6
                  : t8 t6  -> p12 p9 p8 p6
                  : t8  -> p12 p9 p7 p6
                  : t7 t6  -> p11 p10 p8 p6
                  : t7  -> p11 p10 p7 p6
                  : t6  -> p10 p9 p8 p6
p6 p5 p4          : t9 t4  -> p13 p10 p9 p8
                  : t4  -> p10 p9 p8 p6
p6 p5 p2          : t9 t2  -> p13 p5 p4
                  : t9  -> p13 p5 p2
                  : t2  -> p6 p5 p4
p6 p4 p3          : t9 t3  -> p13 p5 p4
                  : t9  -> p13 p4 p3
                  : t3  -> p6 p5 p4
p6 p3 p2          : t9 t3 t2  -> p13 p5 p4
                  : t9 t3  -> p13 p5 p2
                  : t9 t2  -> p13 p4 p3
                  : t9  -> p13 p3 p2
                  : t3 t2  -> p6 p5 p4
                  : t3  -> p6 p5 p2
                  : t2  -> p6 p4 p3
p1                : t1  -> p6 p3 p2
```

Texto 4.4: Grafo de alcançabilidade, enumerando os lugares e as transições.

```
> ConPar < reactor.cpar

================================================
ConPar Compiler: version 1.0
Copyright 1994 - J.M.Fernandes - U.M. - Braga - PORTUGAL
Email contact: miguel@di.uminho.pt
================================================

there's a DEADLOCK situation in petri net when the marking is: p13 p5

Reachability Graph written to file "Graph"
```

Texto 4.5: Resultados produzidos pela aplicação ConPar, durante a fase de análise, após acrescentar arco entre o lugar p4 e a transição t3.

```
INITIAL MARKING: p1

p13 p5      <<< DEADLOCK situation!!! >>>
p6 p5  : t9  -> p13 p5
p6 p4 p3  : t3  -> p6 p5
p6 p3 p2  : t2  -> p6 p4 p3
p1  : t1  -> p6 p3 p2
```

Texto 4.6: Grafo de alcançabilidade, após detecção de uma situação de bloqueio.

```
> ConPar < reactor.cpar

================================================
ConPar Compiler: version 1.0
Copyright 1994 - J.M.Fernandes - U.M. - Braga - PORTUGAL
Email contact: miguel@di.uminho.pt
================================================

transition "t13" is dead (doesn't appear in reachability)!
transition "t12" is dead (doesn't appear in reachability)!
transition "t11" is dead (doesn't appear in reachability)!
transition "t10" is dead (doesn't appear in reachability)!

Reachability Graph written to file "Graph"
```

Texto 4.7: Resultados produzidos pela aplicação ConPar, durante a fase de análise, após acrescentar arco entre o lugar p11 e a transição t8.

```
ENTITY controller IS
  PORT (reset, inicia, c1cheio, ..., cvazio, relogio : IN BIT;
        abres1, abres2, abrec1, ..., recuac, despejac : OUT BIT);
END controller;

ARCHITECTURE dataflow OF controller IS
-- Place Signals
  SIGNAL p1, ..., p16  : REG_BIT REGISTER;
  SIGNAL Np1, ...,  Np16 : BIT;
-- Transition Signals
  SIGNAL t1, ..., t13 : BIT;

BEGIN
  PART : BLOCK (relogio='1' AND NOT relogio'STABLE)
  BEGIN
    p1 <= GUARDED Np1 WHEN reset='0' ELSE '1';
    ...
    p16 <= GUARDED Np16 WHEN reset='0' ELSE '0';
  END BLOCK;
-- Dataflow description for transitions
  t1 <= inicia AND p1 AND NOT p2 AND NOT p3 AND NOT p6;
  t2 <= c1cheio AND p2 AND NOT p4;
  t3 <= c2cheio AND p3 AND NOT p5;
  t4 <= p5 AND p4 AND NOT p8 AND NOT p9 AND NOT p10;
  t5 <= tlimite AND p8 AND NOT p7;
  t6 <= NOT tlimite AND p7 AND NOT p8;
  t7 <= c1vazio AND p9 AND NOT p11;
  t8 <= c2vazio AND p10 AND NOT p12;
  t9 <= iniciopista AND p6 AND NOT p13;
  t10 <= p13 AND p12 AND p11 AND NOT p14;
  t11 <= tvazio AND p14 AND p8 AND NOT p15;
  t12 <= fimpista AND p15 AND NOT p16;
  t13 <= cvazio AND p16 AND NOT p1;
-- Dataflow description for next place markings
  Np1 <= t13 OR (p1 AND NOT t1);
  Np2 <= t1 OR (p2 AND NOT t2);
  Np3 <= t1 OR (p3 AND NOT t3);
  Np4 <= t2 OR (p4 AND NOT t4);
  Np5 <= t3 OR (p5 AND NOT t4);
  Np6 <= t1 OR (p6 AND NOT t9);
  Np7 <= t5 OR (p7 AND NOT t6);
  Np8 <= t4 OR t6 OR (p8 AND NOT t5 AND NOT t11);
  Np9 <= t4 OR (p9 AND NOT t7);
  Np10 <= t4 OR (p10 AND NOT t8);
  Np11 <= t7 OR (p11 AND NOT t10);
  Np12 <= t8 OR (p12 AND NOT t10);
  Np13 <= t9 OR (p13 AND NOT t10);
  Np14 <= t10 OR (p14 AND NOT t11);
  Np15 <= t11 OR (p15 AND NOT t12);
  Np16 <= t12 OR (p16 AND NOT t13);
-- Output Signals Equations
  abres1 <= p2;
  abres2 <= p3;
  abrec1 <= p9;
  abrec2 <= p10;
  abret <= p14;
  rodav <= p7;
  avancac <= p15;
  recuac <= p6;
  despejac <= p16;
-- Transitions in conflict
  ASSERT NOT (t4 AND t6)
    REPORT "t4 and t6 are in conflict, because of output place p8."
    SEVERITY ERROR;
  ASSERT NOT (t5 AND t11)
    REPORT "t5 and t11 are in conflict, because of input place p8."
    SEVERITY ERROR;
-- No Enabled Transitions
  ASSERT NOT (t1='0' AND t2='0' AND ... AND t13='0')
    REPORT "Petri Net may be deadlocked"
    SEVERITY WARNING;
END dataflow;
```

Texto 4.8: Código VHDL, ao nível fluxo de dados, gerado para o controlador do reactor.

```
-- clock and reset
in       relogio B;
in       reset B;
-- the others 10 inputs
in       inicia B;
in       c1cheio B;
in       c1vazio B;
in       c2cheio B;
in       c2vazio B;
in       tlimite B;
in       tvazio B;
in       iniciopista B;
in       fimpista B;
in       cvazio B;
-- the 9 outputs
out      abres1 B;
out      abres2 B;
out      abrec1 B;
out      abrec2 B;
out      abret B;
out      rodav B;
out      avancac B;
out      recuac B;
out      despejac B;

begin
# reset=1: INITIAL MARKING (p1)
# No Outputs active, no transition enabled!
pat_0  : 0 1 0000000000 ?0?0?0?0?0?0?0?0?0;

# t enabled
pat_1  : 1 1 1000000100 ?0?0?0?0?0?0?0?0?0;

# reset=0:
# No Outputs active
pat_2  : 0 0 1000000100 ?0?0?0?0?0?0?0?0?0;

# CLOCK PULSE: (p2 p3 p6)
# Outputs: AbreS1 AbreS2 RecuaC
# t1 fired and t9 enabled
pat_3  : 1 0 1000000100 ?1?1?0?0?0?0?0?1?0;

# Outputs: AbreS1 AbreS2 RecuaC
# t9 enabled
pat_4  : 0 0 1000000100 ?1?1?0?0?0?0?0?1?0;

# CLOCK PULSE: (p2 p3 p13)
# Outputs: AbreS1 AbreS2
# t9 fired but no transition enabled!
pat_5  : 1 0 1000000100 ?1?1?0?0?0?0?0?0?0;

# t2 enabled, because C1Cheio
pat_6  : 0 0 1100000100 ?1?1?0?0?0?0?0?0?0;

# CLOCK PULSE: (p4 p3 p13)
# Outputs: AbreS2
# t2 fired but no transition enabled!
pat_7  : 1 0 1100000100 ?0?1?0?0?0?0?0?0?0;

# t3 enabled, because C2Cheio
pat_8  : 0 0 1101000100 ?0?1?0?0?0?0?0?0?0;

# CLOCK PULSE: (p4 p5 p13)
# No Outputs active
# t3 fired and t4 enabled
pat_9  : 1 0 1101000100 ?0?0?0?0?0?0?0?0?0;

# t4 enabled
pat_10 : 0 0 1101000100 ?0?0?0?0?0?0?0?0?0;

# CLOCK PULSE: (p8 p9 p10 p13)
# Outputs: AbreC1 AbreC2
# t4 fired but no transition enabled!
pat_11 : 1 0 1101000100 ?0?0?1?1?0?0?0?0?0;

# t5 enabled, because TLimite
pat_12 : 0 0 1000010100 ?0?0?1?1?0?0?0?0?0;

# CLOCK PULSE: (p7 p9 p10 p13)
# Outputs: AbreC1 AbreC2 RodaV
# t5 fired but no transition enabled!
pat_13 : 1 0 1000010100 ?0?0?1?1?0?1?0?0?0;

# t7, t8 enabled, because C1Vazio C2Vazio
pat_14 : 0 0 1010110100 ?0?0?1?1?0?1?0?0?0;

# CLOCK PULSE: (p7 p11 p12 p13)
# Outputs: AbreC1 AbreC2 RodaV
# t7, t8 fired and t10 enabled
pat_15 : 1 0 1010110100 ?0?0?0?0?0?1?0?0?0;

# t10,t6 enabled, because TLimite'
pat_16 : 0 0 1010100100 ?0?0?0?0?0?1?0?0?0;

# CLOCK PULSE: (p8 p14)
# Outputs: AbreT
# t6,t10 fired but no transition enabled!
pat_17 : 1 0 1010100100 ?0?0?0?0?1?0?0?0?0;

# t11 enabled, because TVazio
pat_18 : 0 0 1010101100 ?0?0?0?0?1?0?0?0?0;

# CLOCK PULSE: (p15)
# Outputs: AvancaC
# t11 fired but no transition enabled!
pat_19 : 1 0 1010110100 ?0?0?0?0?0?0?1?0?0;

# t12 enabled, because FimPista
pat_20 : 0 0 1010101010 ?0?0?0?0?0?0?1?0?0;

# CLOCK PULSE: (p16)
# Outputs: DespejaC
# t12 fired but no transition enabled!
pat_21 : 1 0 1010110010 ?0?0?0?0?0?0?0?0?1;

# t13 enabled, because CVazio
pat_22 : 0 0 1010101001 ?0?0?0?0?0?0?0?0?1;

# CLOCK PULSE: (p1)
# No Outputs active
# t13 fired and t1 enabled
pat_23 : 1 0 1010110001 ?0?0?0?0?0?0?0?0?0;
end;
```

Texto 4.9: Ficheiro com vectores de teste para simular o comportamento do controlador.

```
> asimut -b reactor reactor output
Initializing ...
Searching ‘reactor‘ ...
Compiling ‘reactor.vbe‘ (Behaviour) ...
Making Bdd ...

Searching pattern file : ‘reactor‘ ...
Restoring ...

Linking ...
###----- processing pattern 0 -----###
WARNING : ‘‘ assert violation on cell controller : "petri net may be deadlocked"
###----- processing pattern 1 -----###
###----- processing pattern 2 -----###
###----- processing pattern 3 -----###
###----- processing pattern 4 -----###
###----- processing pattern 5 -----###
WARNING : ‘‘ assert violation on cell controller : "petri net may be deadlocked"
###----- processing pattern 6 -----###
###----- processing pattern 7 -----###
WARNING : ‘‘ assert violation on cell controller : "petri net may be deadlocked"
###----- processing pattern 8 -----###
###----- processing pattern 9 -----###
###----- processing pattern 10 -----###
###----- processing pattern 11 -----###
WARNING : ‘‘ assert violation on cell controller : "petri net may be deadlocked"
###----- processing pattern 12 -----###
###----- processing pattern 13 -----###
WARNING : ‘‘ assert violation on cell controller : "petri net may be deadlocked"
###----- processing pattern 14 -----###
###----- processing pattern 15 -----###
###----- processing pattern 16 -----###
###----- processing pattern 17 -----###
WARNING : ‘‘ assert violation on cell controller : "petri net may be deadlocked"
###----- processing pattern 18 -----###
###----- processing pattern 19 -----###
WARNING : ‘‘ assert violation on cell controller : "petri net may be deadlocked"
###----- processing pattern 20 -----###
###----- processing pattern 21 -----###
WARNING : ‘‘ assert violation on cell controller : "petri net may be deadlocked"
###----- processing pattern 22 -----###
###----- processing pattern 23 -----###

> logic -s reactor reactor
Compiling ‘reactor.vbe‘ (Behaviour) ...
Running Mapping Standard Cell...
============================  INITIAL COST  ================================
Total number of litterals   = 185
Reduced number of litterals = 153
Register litterals          = 112
Number of binary operators  = 83
Cone maximum                = 17
Depth maximum               = 7
Delay maximum               = 3.500
=========================================================================
Running rules generator on '/user/alliance/alliance-2.0/cells/scr'...
cry_y Unused
sum_y Unused
tie_y Unused
115 rules generated
.....................................

Critical path Signal (null) : 0 (rodav : 21762)

MBK Driving './reactor.vst'...

============================  RESULTS  ================================
Number of cells used             = 14
Number of gates used             = 99 (25 inverters)
Number of grids                  = 497 (125244)
Depth maximum (in gates)         = 7 (8 eq. negative gates)
=========================================================================
```

Texto 4.10: Resultado da simulação e síntese do ficheiro `reactor.vbe`, usando o simulador ASIMUT e o sintetizador LOGIC.

```
-- VHDL structural description generated from `reactor`

-- Entity Declaration
ENTITY reactor IS
  PORT (
  reset : in BIT;       -- reset
  inicia : in BIT;      -- inicia
  c1cheio : in BIT;     -- c1cheio
  ...
  relogio : in BIT;     -- relogio
  abres1 : out BIT;     -- abres1
  ...
  despejac : out BIT;   -- despejac
  vdd : linkage BIT;    -- vdd
  vss : linkage BIT     -- vss
  );
END reactor;

-- Architecture Declaration
ARCHITECTURE structural_view OF reactor IS
  COMPONENT nao3_y
    port (
    i0 : in BIT;        -- i0
    i1 : in BIT;        -- i1
    i2 : in BIT;        -- i2
    f : out BIT;        -- f
    vdd : in BIT;       -- vdd
    vss : in BIT        -- vss
    );
  END COMPONENT;
  ...
  COMPONENT ms_y
    port (
    i : in BIT; -- i
    l : in BIT; -- l
    t : out BIT;        -- t
    vdd : in BIT;       -- vdd
    vss : in BIT        -- vss
    );
  END COMPONENT;

  SIGNAL t13 : BIT;     -- t13
  SIGNAL t11 : BIT;     -- t11
  ...
  SIGNAL auxreg4 : BIT; -- auxreg4

BEGIN
  ...
  auxsc117 : nao3_y
    PORT MAP (
    vss => vss,
    vdd => vdd,
    f => auxsc117,
    i2 => reset,
    i1 => auxsc121,
    i0 => auxsc119);
  ...
  p16 : ms_y
    PORT MAP (
    vss => vss,
    vdd => vdd,
    t => despejac,
    l => auxsc9,
    i => auxsc8);
  ...
  p1 : ms_y
    PORT MAP (
    vss => vss,
    vdd => vdd,
    t => auxreg16,
    l => auxsc9,
    i => auxsc139);
end structural_view;
```

Texto 4.11: Código VHDL, ao nível estrutural, gerado pelo sintetizador LOGIC.

Capítulo 5

Conclusão e Trabalho Futuro

5.1 Conclusão

Neste trabalho, foi estudado o problema da especificação de controladores paralelos. Como formalismo, para atingir tal fim, utilizou-se o paradigma de modelação das RdP. Estas mostraram-se adequadas para o efeito, devido à sua capacidade para modelar dinamicamente sistemas com actividades paralelas e às técnicas formais de análise que disponibilizam, embora tenha havido a necessidade de proceder a algumas modificações, relativamente ao modelo básico das RdP. Essas modificações consistiram basicamente na introdução de guardas nas transições e de acções nos lugares e transições— a interpretação — e em considerar os disparos das transições regulados por um sinal de sincronização. Por estes factos, ao tipo de RdP resultante deu-se o nome de RdP Síncrona Interpretada (RdPSI).

Foi descrita uma linguagem, de nome ConPar, que permite especificar os controladores paralelos baseados no tipo de RdP adoptado. Foi ainda desenvolvida uma aplicação computacional que, usando como entrada ficheiros escritos na referida linguagem, verifica algumas propriedades da RdPSI que serve de modelo ao controlador e gera a respectiva descrição em código VHDL, para posterior simulação e síntese.

A análise das RdPSI é realizada, na aplicação desenvolvida, apenas ao nível mais alto de abstracção. No caso de se utilizarem macronodos, para se especificar o controlador, aqueles não são refinados para efeitos analíticos. Para entrar em consideração com os macronodos, há, pelo menos, duas alternativas possíveis. A primeira consiste em linearizar toda a RdP (refinar todos os seus macronodos) e proceder, então, à análise da RdP resultante. Embora esta solução garanta que a análise é aplicada a toda a RdP, tem a agravante de se aplicar a uma RdP potencialmente muito grande (com muitos nodos e arcos), o que torna o processo de análise extremamente demorado e perdendo-se assim algumas das vantagens resultantes da utilização do conceito de hierarquia. A segunda opção consiste em aplicar as metodologias apresentadas por Valette [Val79] ou Suzuki e Murata [SM83], que analisam, separadamente, a RdP mais abstracta e os macronodos, e que extrapolam os resultados para a RdP mais refinada. Todavia, esta solução requer alguma investigação adicional, pois os métodos referidos aplicam-se ao modelo básico das RdP e, no presente trabalho, há que considerar as implicações resultantes da interpretação e, principalmente, do carácter síncrono das RdPSI.

O código VHDL gerado, que descreve o comportamento do controlador especificado, encon-

tra-se ao nível fluxo de dados, o que permite utilizar a maioria das ferramentas CAD para VHDL disponíveis actualmente no mercado, a menos de ligeiras alterações. Porém, o futuro reservado para os sistemas digitais de complexidade razoável reside na utilização de especificações algorítmicas de alto nível. Há pois que considerar, num futuro que se avizinha próximo, a representação de controladores baseados em RdP a este nível de abstracção. A linguagem VHDL facilita a existência, para uma dada entidade, de várias arquitecturas, fornecendo mecanismos para selecção de uma dada delas.

O trabalho realizado considerou a integração da aplicação ConPar, num ambiente de desenvolvimento completo suportado por uma metodologia apropriada, que possibilita a especificação, a análise, a animação, a simulação e a síntese de controladores paralelos baseados em RdPSI. O exemplo analisado permite concluir que a metodologia adoptada é viável e que os resultados produzidos são aceitáveis.

5.2 Trabalho Futuro

Nesta secção pretendem-se deixar algumas indicações sobre o trajecto que o trabalho desenvolvido, pode tomar, de molde a melhorá-lo. As duas grandes linhas de investigação a seguir consistem, como atrás se referiu, na geração de código VHDL a níveis de abstracção mais elevados e em determinar alguns teoremas e regras que permitam testar a RdPSI hierarquicamente. Adicionalmente, poder-se-á considerar qualquer dos pontos que a seguir se descrevem.

5.2.1 Enriquecimento da Linguagem ConPar

A linguagem ConPar pode considerar-se simples, pelo que não disponibiliza grandes facilidades para escrita de programas. Estudar possíveis alterações e extensões à linguagem proposta parece ser uma direcção a tomar. Como exemplo, a linguagem poderia admitir notação vectorial para entradas, saídas, transições e lugares. Outra funcionalidade que poderá revestir-se de grande utilidade reside na criação e manipulação de bibliotecas de macronodos[1]. Finalmente, alargar a linguagem de forma a incluir, além da unidade de controlo, a unidade de dados do sistema digital, aumentará o grau de utilização da aplicação desenvolvida.

5.2.2 Múltiplos Sinais de Sincronização

Os modelos considerados admitem apenas um único sinal de sincronismo. Seria útil, para determinadas finalidades, permitir a especificação de mais do que um desses sinais. Estudar as suas consequências a nível de comportamento e que alterações introduzir nas técnicas analíticas é outra linha de acção que pode tomar-se.

[1] Algo idêntico às bibliotecas do C, por exemplo.

5.2.3 Editor e Animador Gráficos

Dos módulos presentes no ambiente completo de desenvolvimento, apresentado na secção 4.3, não foram implementados, no contexto deste trabalho, o editor e o animador gráficos. A introdução das RdPSI através de um editor de texto está mais sujeita a erros. Uma primeira solução para atenuar este problema passa pelo desenvolvimento de um editor dirigido pela sintaxe da linguagem ConPar. Mais atractivo do ponto de vista de utilização, será desenvolver uma aplicação que permita a introdução gráfica das RdPSI, tentando adaptar, por exemplo, o sistema proposto por Viegas [Vie94].

A possibilidade de poder animar graficamente as RdPSI reveste-se de alguma importância no processo de trabalho, especialmente se forem encontrados erros de concepção pelo módulo de análise. Existem algumas alternativas para desenvolver o animador gráfico como, por exemplo, usar um sistema genérico de modelação com potencialidades de animação (como sugerido por Pina [Pin93]) ou uma das muitas ferramentas de animação de RdP e adaptá-la às RdPSI.

Estes dois módulos de carácter gráfico devem apresentar o mesmo tipo de interface grfico, para facilitar o seu grau de utilização, na metodologia proposta. A integração dos dois módulos numa única aplicação computacional será a solução ideal, pois obriga o utilizador a aprender apenas um novo programa.

5.2.4 Melhoria da Análise das RdPSI

Além da análise dos macronodos, que ficou em aberto, a fase de análise não está completa. Actualmente, todos os testes são realizados com base no grafo de alcançabilidade e na estrutura da RdP. Uma próxima preocupação será incluir o maior número possível de testes, uma vez que nem todos os testes estão implementados (falta verificar se as RdPSI são determinísticas e o teste relativo à vivacidade não é completo). Uma segunda direcção será verificar a equidade de outras técnicas de análise (ver secção 2.3), especialmente o uso de técnicas de redução, que poderão tornar mais simples a fase de análise.

Referências Bibliográficas

[ABA78] M. Augin, F. Boeri, e C. André. New Design using PLAs and Petri Nets. In *International Symposium on Measurement and Control*, pp. 864–9, Atenas, Grécia, 1978.

[ABA80] M. Augin, F. Boeri, e C. André. Systematic Method of Realization of Interpreted Petri Nets. *Digital Processes*, 6:55–68, 1980.

[Ada87] Marian Adamski. Direct Implementation of Petri Net Specification. In *7th International Conference on Control Systems and Computer Science CSCS7*, pp. 74–85, 1987.

[Ada91] Marian Adamski. Parallel Controller Implementation using Standard PLD Software. In W. R. Moore & W. Luk, editor, *FPGAs*. Abingdon EE&CS Books, 1991.

[Aga90] Shishir Agarwal. Thinking Petri Nets Through VHDL. In *1990 VHDL Fall User's Group Meeting*, pp. 51–59, Oakland, E.U.A., Outubro 1990.

[Alb91] Henk Alblas. Introduction to Attribute Grammars. In *Lecture Notes in Computer Science*, número 545, pp. 1–15. Springer-Verlag, 1991.

[AMD91] Advanced Micro Devices. *PALASM 4 User's Manual*, 1991.

[Ash90] Peter J. Ashenden. *The VHDL Cookbook*. Dept. Computer Science, University of Adelaide, 1ª edição, Julho 1990.

[ASU86] Alfred V. Aho, R. Sehti, e J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachussets, E.U.A., 1986.

[AVD76] Pierre Azema, Robert Valette, e Michel Diaz. Petri Nets as a Common Tool for Design Verification and Hardware Simulation. In *Proceedings of the 13th ACM/IEEE Design Automation Conference*, pp. 109–16, Junho 1976.

[Bak93] Louis Baker. *VHDL Programming with Advanced Topics*. John Wiley & Sons, 1993.

[BBFM79] J. C. Bossy, P. Brard, P. Faugère, e C. Merlaud. *Le GRAFCET: sa pratique et ses applications*. Educalivre, Paris, França, 12ª edição, 1979.

[BFMR92] J.-M. Bergé, A. Fonkoua, S. Maginot, e J. Rouillard. *VHDL Designer's Reference*. Kluwer Academic Publishers, 1992.

[BKKR86] B. Brück, B. Kleinjohann, T. Kathofer, e F. J. Rammig. Synthesis of Concurrent Modular Controllers from Algorithmic Description. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pp. 285–92, 1986.

[BMBL88] M. Balakrishnan, A. K. Majumdar, D. K. Banerji, e J. G. Linders. Synthesis of Decentralised Controllers from High Level Description. *Microprocessing and Microprogramming*, 22(3):217–29, 1988.

[Bol91] Martin Bolton. VHDL and its Use in VLSI Design. In *Microcomputer '91 - design, practice, education*, número 39 in Konferencje, pp. 149–58. Prace Naukowe Instytutu Cybernetyki Technicznej Politechniki Wroclawskiej, 1991.

[Cla73] C. A. Clare. *Design Logic Systems Using State Machines*. McGraw-Hill, 1973.

[CST91] R. Camposano, L. F. Saunders, e R. M. Tabet. VHDL as Input for High-Level Synthesis. *IEEE Design & Test of Computers*, pp. 43–9, Março 1991.

[CVBE83] M. Couvoisier, R. Valette, J. M. Bigou, e P. Esteban. A Programmable Logic Controller based on a High Level Specification Tool. In *IECON 83 – IEEE Annual Conference on Industrial Electronic*, Novembro 1983.

[ELP93] R. Elmstrøm, R. Lintulampi, e M. Pezzé. Giving Semantics to SA/RT by Means of High-Level Timed Petri Nets. *Real-Time Systems*, 5(2/3):249–71, Maio 1993.

[Feh93] Rainer Fehling. A Concept of Hierarchical Petri Nets with Building Blocks. In G. Rozenberg, editor, *Advances in Petri Nets 1993*, volume 674 de *Lecture Notes in Computer Science*, pp. 148–68. Springer-Verlag, 1993.

[Fer92] Luca Ferrarini. An Incremental Approach to Logic Controller Design with Petri Nets. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(3):461–73, 1992.

[Fer93] António J. Fernandes. *Métodos e Regras para elaboração de Trabalhos Académicos e Científicos*. Porto Editora, Porto, Portugal, 1993.

[FGP93] M. Felder, C. Ghezzi, e M. Pezzé. High-Level Timed Petri Nets as a Kernel for Executable Specifications. *Real-Time Systems*, 5(2/3):235–48, Maio 1993.

[FM91] Luca Ferrarini e Claudio Maffezzoni. Designing Logic Controllers with Petri Nets. In H. A. Baker, editor, *5th IFAC Symposium Computer-Aided Design in Control Systems*, pp. 319–24, Julho 1991.

[FS89] Rodney Farrow e Alec G. Stanculescu. A VHDL Compiler Based on Attribute Grammar Methodology. *ACM*, pp. 120–130, 1989.

[GE90] Josef Grosch e Helmut Emmelmann. A Tool Box for Compiler Construction. Compiler Generation Project, Report no. 20, GMD - Universität Karlsruhe, Janeiro 1990.

[GP93] Alain Greiner e François Pêcheux. ALLIANCE: A complete Set of CAD Tools for teaching VLSI Design. In *Third Eurochip Workshop on VLSI Design Training*, pp. 230–7, Grenoble, França, 1993.

[Gro87] Josef Grosch. Rex - A Scanner Generator. Compiler Generation Project, Report no. 5, GMD - Universität Karlsruhe, Dezembro 1987.

[Gro89] Josef Grosch. Efficient and Comfortable Error Recovery in Recursive Descent Parsers. Compiler Generation Project, Report no. 19, GMD - Universität Karlsruhe, Dezembro 1989.

[GS93] H. J. Genrich e R. M. Shapiro. A design of a Cascadable Nacking Arbiter. Março 1993.

[GSG92] Luís Gomes e A. Steiger-Garção. Especificação e Realização de Controladores utilizando Redes de Petri Coloridas e Sincronizadas integrando Lógica Imprecisa. *Revista Robótica e Automatização*, (10), Novembro 1992.

[GSGGC93] Luís Gomes, A. Steiger-Garção, Luís Gama, e Nuno Correia. Programação de Controladores Utilizando Redes de Petri. *Ingenium - Revista da Ordem dos Engenheiros*, VIII(72):43–52, Julho 1993.

[Kas91] Uwe Kastens. Attribute Grammars as a Specification Method. In *Lecture Notes in Computer Science*, número 545, pp. 16–47. Springer-Verlag, 1991.

[KBK+90] C.A. Kuszynski, T. Busfield, A.M. Koelmans, M.R. McLauchlan, e D.J. Kinniment. Graphical Representation of a Hardware Description Language. *IEE Proceedings*, 137(6):462–8, Novembro 1990.

[Koz93] Tomasz Kozlowski. Petri-net-based CAD tools for parallel controller synthesis. Dissertação de Mestrado, University of Bristol, Inglaterra, Março 1993.

[KP87] K. Kuchcinski e Zebo Peng. Microprogramming Implementation of Timed Petri Nets. *The VLSI Journal*, (5):133–44, 1987.

[KR88] B. W. Kernighan e D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, E.U.A., 2ª edição, 1988.

[Led81] Henry Ledgard. *ADA: An Introduction*. Springer-Verlag, 1981.

[LF85] Kwang-Hyung Lee e Jöel Favrel. Hierarchical Reduction Method for Analysis and Decomposition of Petri Nets. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-15(2):272–80, 1985.

[LMB77] K. C. Leung, C. Michel, e P. Le Beux. Logical Systems Design using PLAs and Petri Nets: Programmable Hardwired Systems. In *Proceedings of the 1977 IFIP Congress*, pp. 607–11, Amesterdão, Países Baixos, Agosto 1977. North-Holland.

[LMB90] John R. Levine, Tony Mason, e Doug Brown. *lex & yacc*. O'Reilly & Associates, 2ª edição, 1990.

[Mar90] M. Ajmone Marsan. Stochastic Petri Nets: an elementary introduction. In G. Rozenberg, editor, *Advances in Petri Nets 1989*, volume 424 de *Lecture Notes in Computer Science*, pp. 1–29. Springer-Verlag, 1990.

[MBC84] M. Ajmone Marsan, G. Balbo, e G. Conte. A Class of Generalized Stochastic Petri Nets for the Performance Analysis of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2):93–122, Maio 1984.

[Min94] Synthesia, Kista, Suécia. *MINT User's Guide*, Janeiro 1994.

[Mur89] Tadao Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–80, Abril 1989.

[MV94] Fernando Moreira e Luís Vidigal. O Ambiente de Síntese Automática de Circuitos Integrados. In *1º Encontro Nacional do Colégio de Engenharia Electrotécnica*, pp. 175–80, Lisboa, Portugal, Maio 1994. Ordem dos Engenheiros.

[Nav93] Zainalabedin Navabi. *VHDL: Analysis and Modelling of Digital Systems*. McGraw-Hill, 1993.

[NS91] Z. Navabi e J. Spillane. Synthesis of VLSI Circuits from Behavioral Descriptions. *Microelectronics Journal*, 22(5/6):7–13, 1991.

[NZ89] Richard Nishimura e Safwat G. Zaky. Synthesis of a Petri Net Based Control Flow Model. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, volume 1, pp. 313–8, Maio 1989.

[PABA94] James Pardey, Alain Amroun, Martin Bolton, e Marian Adamski. Parallel Controller Synthesis for Programmable Logic Devices. a publicar em *Microprocessors and Microsystems*, 1994.

[PALD93] J. Puente, A. Alonso, G. León, e Carlos Dueñas. Distributed Execution of Specifications. *Real-Time systems*, 5(2/3):213–34, Maio 1993.

[Pat90] M. R. K. Pate. Random Logic Implementation of Extended Timed Petri Nets. *Microprocessing and Microprogramming*, 30:313–320, 1990.

[PB91] James Pardey e Martin Bolton. Logic Synthesis of Synchronous Parallel Controllers. *Proceedings of the IEEE International Conference on Computer Design*, pp. 454–7, 1991.

[Pea91] Maureen Pearce. Report on Existing CAD Support for Programmable Devices. *Computer-Aided Engineering Journal*, pp. 160–6, Agosto 1991.

[Pen92] Zebo Peng. Digital System Simulation with VHDL in a High-level Synthesis System. *Microprocessing and Microprogramming*, 35:263–70, 1992.

[Per91] Douglas L. Perry. *VHDL*. McGraw-Hill, 1991.

[Pet77] James L. Peterson. Petri Nets. *Computing Surveys*, 9(3):223–52, Setembro 1977.

[Pet81] James L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, E.U.A., 1981.

[Pin93] António M. Pina. SCBA - Simulação Concorrente Baseada em Agentes. In *XX SEMISH*, Florianópolis, Brasil, 1993.

[PK86] Zebo Peng e Krzysztof Kuchcinski. Synthesis of Control Structures from Petri Net Descriptions. *Microprocessing and Microprogramming*, 18:335–40, 1986.

[PKL88] Zebo Peng, Krzysztof Kuchcinski, e Bryan Lyles. CAMAD: A Unified Data Path/Control Synthesis Environment. In *Proceedings of the IFIP TC-10 Conference on Design Methodologies for VLSI and Computer Architecture*, Setembro 1988.

[PKSB92] James Pardey, Tomasz Kozlowski, Johnatan Saul, e Martin Bolton. State Assignments Algorithms for Parallel Controller Synthesis. *Proceedings of the IEEE International Conference on Computer Design*, pp. 316–9, 1992.

[Pro91] Alberto J. Proença. Advanced Controller Design. In *Microcomputer '91 - design, practice, education*, número 39 in Konferencje, pp. 191–205. Prace Naukowe Instytutu Cybernetyki Technicznej Politechniki Wroclawskiej, 1991.

[Rei85] Wolfgang Reisig. *Petri Nets - An Introduction*, volume 4 de *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, Heidelberg, Alemanha, 1985.

[RT89] T. Reps e T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Texts and Monographs in Computer Science. Springer-Verlag, Nova Iorque, E.U.A., 1989.

[Sar92] João Saraiva. CTB - Utilitários. Technical report, Departamento de Informática, Universidade do Minho, Braga, 1992.

[Sar93] João Saraiva. Cálculo de Atributos Concorrente. Dissertação de Mestrado, Departamento de Informática, Universidade do Minho, Braga, Portugal, Julho 1993.

[SF91] Pedro P. Silva e Adalberto C. Ferreira. O Gerador de Analisadores Sintácticos ELL. Technical report, Departamento de Informática, Universidade do Minho, Braga, Janeiro 1991.

[SGV93] P. Bazargan Sabet, A. Greiner, e H. N. Vuong. ASIMUT: A Public Domain VHDL Simulation Tool. In *Fourth Eurochip Workshop on VLSI Design Training*, pp. 62–5, Toledo, Espanha, 1993.

[Sil89] Manuel Silva. Logical Controllers. In A. De Carli, editor, *IFAC Low Cost Automation: Techniques, Components and Instruments, Applications*, volume II, pp. F157–F166bis, Milão, Itália, Novembro 1989. IFAC.

[SM83] Ichiro Suzuki e Tadao Murata. A Method for Stepwise Refinement and Abstraction of Petri Nets. *Journal of Computer and Systems Sciences*, 27(1):51–76, Agosto 1983.

[SR92] P. David Stotts e J. Cyrano Ruiz. Synthesezing a Global Net State from Synchronized Local Pieces. Technical Report TR-92-13, Computer and Information Sciences, University of Florida, Janeiro 1992.

[SSL+92] Ellen Sentovich, Kanwar Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul Stephan, Robert K. Brayton, e Alberto Sangiovanni-Vicentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, Department of Electrical Engineering and Computer Science, University of California, Berkeley, E.U.A., Maio 1992.

[Suz84] Ichiro Suzuki. On the Notion of Simultaneous Firings of Transitions in Petri Nets and Marked Graphs. In *27th Midwest Symposium on Circuits and Systems*, volume II, pp. 716–9, West Virginia University, 1984.

[SV82] M. Silva e S. Velilla. Programmable Logic Controllers and Petri Nets: A Comparative Study. In *IFAC Software for Computer Control*, pp. 83–8, Madrid, Espanha, 1982.

[SV90] Manuel Silva e Robert Valette. Petri Nets and Flexible Manufacturing. In G. Rozenberg, editor, *Advances in Petri Nets 89*, volume 424 de *Lecture Notes in Computer Science*, pp. 376–417. Springer-Verlag, 1990.

[Syn94] Synthesia, Kista, Suécia. *SYNT 1.2 User's Guide*, 1994.

[TP79] J. M. Toulette e J. P. Parsy. A Method for Decomposing Interpreted Petri Nets and Its Utilization. *Digital Processes*, 5(3-4):223–34, 1979.

[TPU93] Y. Torroja, I. Pompa, e J. Uceda. Automatic Synthesis of FSM from Graphical Descriptions. In *Fourth Eurochip Workshop on VLSI Design Training*, pp. 72–7, Toledo, Espanha, 1993.

[Val79] R. Valette. Analysis of Petri Nets by Stepwise Refinements. *Journal of Computer and System Science*, (18):35–56, 1979.

[VCBA83] R. Valette, M. Courvoisier, J.M. Bigou, e J. Albukerque. A Petri Net Based Programmable Logic Controller. In *IFIP First International Conference on Computer Applications in Production and Enginnering*, Abril 1983.

[Vie88] Bertram Vielsack. The Parser Generators Lalr and Ell. Compiler Generation Project, Report no. 8, GMD - Universität Karlsruhe, Dezembro 1988.

[Vie94] Paulo Manuel Viegas. Geração de Controladores de Diálogos Assíncronos Usando Redes de Petri. Dissertação de Mestrado, Dep. Informática, Universidade do Minho, Braga, Portugal, 1994.

[Wai93] W. M. Waite. A Complete Specification of a Simple Compiler. Technical Report CU-CS-638-93, Depart. of Computer Science, University of Colorado at Boulder, Janeiro 1993.

[Wir76] Niklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Cliffs, New Jersey, E.U.A., 1976.

Lista de Figuras

Lista de Textos

Lista de Quadros

Apêndices

Apêndice A

Linguagem ConPar

"Quando mostrei a minha torre, Airam Josef
não perguntou sequer para que servia.
Olhou curiosa, levemente desapontada
e disse: como se chama? de Babel respondi.
E subimos a rampa."

Ana Maria Ferreira in "Babel"

Neste apêndice, apresenta-se a sintaxe completa da linguagem ConPar, faz-se uma descrição detalhada dos vários construtores, e consideram-se alguns exemplos ilustrativos. Será usada a notação Backus-Naur Form (BNF) para apresentar a sintaxe da linguagem.

A.1 Estrutura Genérica

A estrutura genérica de um texto fonte em ConPar é a seguinte:

<ControladorParalelo> ::=
    <Cabeçalho>
    (<MacroLugar> | <MacroTransição>) *
    <Parte> +
    [**.PREDICATEDESCRIPTION** <Descrição_Predicado> +]
    .E

Podem-se introduzir comentários no texto fonte. Os comentários podem ser aninhados o número de níveis que se pretender. Para abrir um comentário usa-se a sequência de caracteres: "< *". Para fechar um comentário, deve-se usar a sequência "* >". Considerem-se os exemplos apresentados no texto A.1.

A linguagem não diferencia as letras minúsculas das maiúsculas. Assim, é indiferente escrever-se a palavra–chave "**.marking**" de qualquer dos seguintes modos: "`.marking`", "`.Marking`", "`.MARKING`", "`.MaRkInG`". O mesmo é válido para qualquer outra palavra–chave ou identificador usado na linguagem.

```
<* Inicio de comentario ...
   ... Fim de comentario *>

<*
Comentario aninhado ...
<* comentario dentro de comentario ...
   ...
   fim de comentario dentro de comentario *>
...
Fim de comentario aninhado
*>
```

Texto A.1: Exemplos de comentários

A.2 Sinais Globais

Os sinais globais representam todos os sinais que podem ser usados por qualquer parte constituinte do controlador. Nestes sinais incluem-se: relógio, entradas, saídas e predicados.

Genéricamente, o bloco dos sinais globais tém a seguinte forma:

<Cabeçalho> ::=
.CLOCK <ident>
[**.INPUT** <ident> +]
[**.OUTPUT** <ident> +]
[**.PREDICATE** <ident> +]

A.3 Macronodos

Os macronodos (macrolugares e macrotransições) podem ser vistos como subrotinas de uma qualquer linguagem de programação, tais como: C, PASCAL, etc.

Os macronodos permitem especificar um controlador de forma hierárquica. No lugar do macronodo, deve ser colocada a RdP que aquele representa.

A sintaxe de um macronodo é dada por:

<MacroLugar> ::=
.MACROPLACE <ident> (<ident> * , <ident> *)
.INTERFACE <ident> , <ident>
<MacroNodoCabeçalho>
.NET <Transição> +
[**.MOOREOUTPUT** <Saída_Moore> +]
[**.PREDICATEDESCRIPTION** <Descrição_Predicado> +]
[<Marcação_Inicial>]

<MacroTransição> ::=
.MACROTRANSITION <ident> (<ident> * , <ident> *)
.INTERFACE <ident> , <ident>
<MacroNodoCabeçalho>
.NET <Transição> +
[**.MOOREOUTPUT** <Saída_Moore> +]
[**.PREDICATEDESCRIPTION** <Descrição_Predicado> +]

As interfaces são as portas de macrolugares ou macrotransições (ver definição 24).

A.4 Partes

As partes representam as várias redes presentes num controlador, i.e. os vários subcontroladores que compõem o controlador global. As partes podem ou não estar ligadas. Por exemplo, o sistema representado na figura 2.16, pode ser descrito por duas partes, correspondentes aos processos A e B.

Os lugares e as transições definidos nas partes são globais, pelo que lugares ou transições presentes em qualquer parte devem ter nomes distintos.

A sintaxe de uma parte é dada por:

<Parte> ::=
.PART <ident>
<ParteCabeçalho>
.NET <Transição> +
[**.MOOREOUTPUT** <Saída_Moore> +]
[**.PREDICATEDESCRIPTION** <Descrição_Predicado> +]
<Marcação_Inicial>

A.5 Transições

Para cada transição presente numa parte ou num macronodo tem de se escrever a respectiva especificação. Esta especifica os lugares de entrada e saída, a guarda e os sinais de saída (do tipo Mealy). A sintaxe de uma especificação de uma transição é dada por:

<Transição> ::=
<ident> **:** <Condições> |- <Consequências> **;**

O símbolo <ident> especifica a transição em causa.

O símbolo <Condições> representa uma lista (separada por asteriscos '*') de lugares de entrada e de uma guarda (opcional). A guarda tem de ser um único nome. Se a guarda for constituída por um único sinal de entrada (negado ou não), pode esse sinal ser especificado directamente. Se a guarda contiver mais do que um sinal (entradas ou lugares), a expressão lógica respectiva terá de ser indicada num predicado (ver à frente) e usado esse predicado na especificação. Quando há arcos inibidores ou habilitadores a ligar um lugar a uma dada transição, a sua especificação é feita obrigatoriamente num predicado (para distinguir dos lugares de entrada "normais"). Um arco inibidor é especificado negando o nome do lugar, ao passo que um arco habilitador é especificado simplesmente referindo o nome do lugar de origem.

O símbolo <Consequências> especifica uma lista de lugares de saída e de sinais de saída (tipo Mealy, ou seja, activos sempre que a transição estiver habilitada a disparar).

A.6 Saídas de tipo Moore

Para cada lugar onde existam sinais de saída activos (tipo Moore), é necessário escrever um fragmento de código que siga a seguinte sintaxe:

<Saída_Moore> ::=
    <ident> |- <Saídas_Activas> ;

O símbolo <ident> especifica o lugar em causa.

O símbolo <Saídas_Activas> representa uma lista (separada por asteriscos '*') dos sinais de saída activos no respectivo lugar.

A.7 Predicados

Quando a guarda de uma transição é composta por uma expressão envolvendo arcos habilitadores ou inibidores ou mais do que um sinal de entrada tem de usar-se um predicado. O predicado é usado na especificação da transição.

A definição de um predicado segue a seguinte sintaxe.

<Descrição_Predicado> ::=
    <ident> = <Função_Lógica> ;

O símbolo <Função_Lógica> representa uma expressão lógica envolvendo sinais de entrada e lugares. São usados os seguintes operadores:

Símbolo	Significado
*	AND lógico
+	OR lógico
!	NOT lógico

Quadro A.1: Símbolos para definição de expressão lógicas.

Podem ainda usar-se parênteses curvos para agrupar expressões lógicas.

Tomem-se os exemplos apresentados na figura A.1, considerando que `xi` são sinais de entrada, `pi` são lugares, `ti` são transições e `predi` representam nomes de predicados.

Para as transições `t1` e `t2` não são necessários predicados, uma vez que as suas guardas são compostas por um único sinal de entrada — complementado no caso da transição `t2`. As restantes transições necessitam de predicados. A transição `t3` tem uma guarda cuja expressão lógica envolve dois sinais de entrada. Para a transições `t4` e `t5`, teve de usar-se um predicado, pois existe um arco habilitador com origem no lugar `p4C` e um arco inibidor com origem no lugar `p5C`, respectivamente. A transição `t6` representa o caso mais genérico, quando a guarda da transição envolve uma expressão lógica dos sinais de entrada e arcos inibidores ou habilitadores.

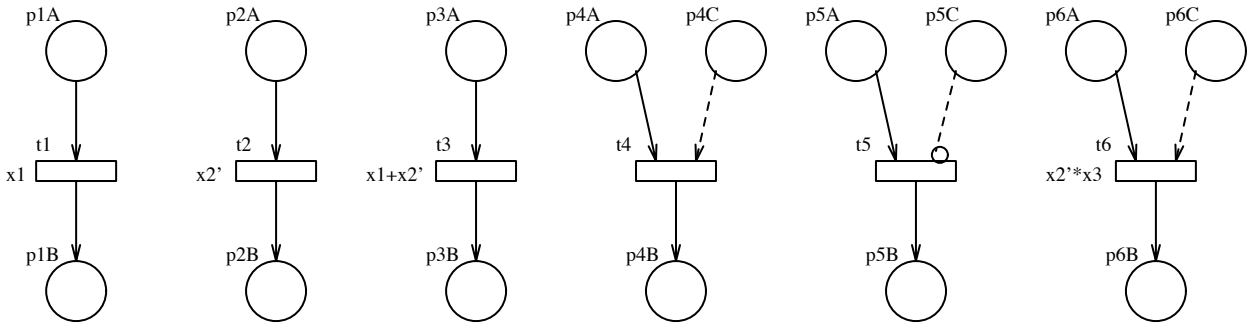


Figura A.1: Algumas transições de uma RdPSI.

```
t1: p1A *  x1   |- p1B;
t2: p2A * !x2   |- p2B;
t3: p3A * pred3 |- p3B;
t4: p4A * pred4 |- p4B;
t5: p5A * pred5 |- p5B;
t6: p6A * pred6 |- p6B;
...
pred3 = x1+!x2;
pred4 = p4C;
pred5 = !p5C;
pred6 = (!x2*x3)*p6C;
```

Texto A.2: Alguns exemplos de como usar predicados.

A.8 Marcação Inicial

Para cada parte ou macrolugar têm de se iniciar quais os lugares que estão inicialmente marcados. A sintaxe da especificação da marcação inicial é dada por:

<Marcação_Inicial> ::=
.MARKING <ident> +

O símbolo <ident> especifica cada um dos lugares a marcar inicialmente.

A.9 Exemplos

Como primeiro exemplo, considere-se a RdPSI apresentada na figura 2.21. O código em linguagem ConPar que descreve esse controlador é apresentado no texto A.3.

O segundo exemplo a considerar é a RdPSI ilustrada na figura 3.6, cujo respectivo código em linguagem ConPar se encontra no texto A.4.

Considere-se agora um controlador especificado com base em duas RdPSI, conforme se apresenta na figura 4.4. A especificação deste controlador em ConPar é apresentada no texto A.5.

Finalmente, considere-se um controlador onde se utiliza um macrolugar — ver figura 4.5. O respectivo código em linguagem ConPar é o presente no texto A.6.

```
<*
   Autor:     Joao Miguel Fernandes
   Data:      10-Nov-1993
   Projecto:  Tese de Mestrado
   Companhia: Universidade do Minho
              (Braga - Portugal)
*>

<* Header *>
.clock CLOCK
.input X1 X2 X3
.output Y1 Y2 Y3 Y4 Y5

<* Part *>
.part SIPN
.place p1 p2 p3 p4 p5 p6
.transition t1 t2 t3 t4 t5 t6
.predicate pt5 pt6
.net
t1: p1*X1    |- p2*p3*Y4;
t2: p2*pt2   |- p5*Y5;
t3: p3*X3    |- p4;
t4: p4*!X2   |- p6;
t5: p5*p6*X2 |- p1*Y4;
t6: p6*pt6   |- p3;
.MooreOutput
p1 |- Y1;
p2 |- Y2;
p4 |- Y1;
p6 |- Y3;
.predicateDescription
pt2 = !p3;
pt6 = !X2*p5;
.marking p1
.e
```

Texto A.3: Código ConPar para RdPSI da fig. 2.21.

```
<*
   Autor:     Joao Miguel Fernandes
   Data:      30-Mar-1994
   Projecto:  Tese de Mestrado
   Companhia: Universidade do Minho
              (Braga - Portugal)
*>

.clock RELOGIO
.input X1 X2 X3
.output Y1 Y2 Y3
<*
PARTE CONTROLADOR
*>
.part controller
.place P1 P2 P3 P4 P5
.transition T1 T2 T3 T4 T5
.net
T1: P1 * X1 |- P2 * P3 * Y1;
T2: P2 * X2 |- P4;
T3: P3 * X3 |- P5 * Y2;
T4: P5 * X3 |- P3;
T5: P4 * P5 * !X3 |- P1 *Y2;
.MooreOutput
P1 |- Y3;
P4 |- Y1;
.marking P1
.e
```

Texto A.4: Código ConPar para RdPSI da fig. 3.6.

```
.clock RELOGIO

<*
PARTE PRINCIPAL
*>
.part main
.input x1 x2 x3 x4 x5
.output y1 y2 y3 y4
.place P1 P2 P3 P4 P5 P6 P7 P8
.transition T1 T2 T3 T4 T5 T6
.predicate PRED4
.net
T1: P8 * x1 |- P1 * P2;
T2: P1 * P2 * x2 |- P3;
T3: P3 * x3 |- P4 * P5;
T4: P4 * pred4 |- P6;
T5: P5 * x4 |- P7;
T6: P6 * P7 * x5 |- P8;
.MooreOutput
P1 |- y1;
P2 |- y2;
P4 |- y3;
P7 |- y4;
.PredicateDescription
PRED4 = P7;
<* Marcacao inicial *>
.marking P1 P2

<*
PARTE CONTADOR
*>
.part contador
.output yb1
.place PB1 PB2
.transition TB1 TB2
.predicate PRED1 PRED2
.net
TB1: PB2 * pred1 |- PB1;
TB2: PB1 * pred2 |- PB2;
.MooreOutput
Pb1 |- yb1;
.PredicateDescription
PRED1 = P8;
PRED2 = !P1;
<* Marcacao inicial *>
.marking Pb1

.e
```

Texto A.5: Código ConPar para RdPSI da fig. 4.4.

```
<*
   Autor:     Joao Miguel Fernandes
   Data:      31-Mar-1993
   Projecto:  Tese de Mestrado
   Companhia: Universidade do Minho
              (Braga - Portugal)
*>

<* CABECALHO *>
.CLOCK relogio
.INPUT x1 x2
.OUTPUT y1 y2 y3

.MACROPLACE mp1(A B, C D)
.INTERFACE pA, pF
.PLACE pB pC pD pE
.TRANSITION tA tB tC tD
.NET
tA: pA*A |- pB*pC;
tB: pB   |- pD;
tC: pC*B |- pE;
tD: pD*pE*!B |- pF;
.MOOREOUTPUT
pA |- D;
pC |- C;

<*
PARTE UNICA
*>
.PART unique
.PLACE p1=mp1(x1 x2,y1 y2) p2 p3
.TRANSITION t1 t2 t3
.PREDICATE pred3
.NET
t1: p1*x1    |- p2;
t2: p2*x2    |- p3;
t3: p3*pred3 |- p1;
.MOOREOUTPUT
p2 |- y1;
p3 |- y3;
.PREDICATEDESCRIPTION
pred3 = !x1*x2;
<* Marcacao inicial *>
.MARKING p3

.E
```

Texto A.6: Código ConPar para RdPSI da fig. 4.5.

A.10 Gramática da Linguagem ConPar

Nesta secção, apresenta-se a gramática da linguagem ConPar.

<ControladorParalelo> ::=
    <Cabeçalho>
    (<MacroLugar> | <MacroTransição>) *
    <Parte> +
    [**.PREDICATEDESCRIPTION** <Descrição_Predicado> +]
    .E

<Cabeçalho> ::=
    .CLOCK <ident>
    [**.INPUT** <ident> +]
    [**.OUTPUT** <ident> +]
    [**.PREDICATE** <ident> +]

<MacroLugar> ::=
    .MACROPLACE <ident> (<ident> * , <ident> *)
    .INTERFACE <ident> , <ident>
    <MacroNodoCabeçalho>
    .NET <Transição> +
    [**.MOOREOUTPUT** <Saída_Moore> +]
    [**.PREDICATEDESCRIPTION** <Descrição_Predicado> +]
    [<Marcação_Inicial>]

<MacroTransição> ::=
    .MACROTRANSITION <ident> (<ident> * , <ident> *)
    .INTERFACE <ident> , <ident>
    <MacroNodoCabeçalho>
    .NET <Transição> +
    [**.MOOREOUTPUT** <Saída_Moore> +]
    [**.PREDICATEDESCRIPTION** <Descrição_Predicado> +]

<MacroNodoCabeçalho> ::=
    .PLACE <Nodo> +
    .TRANSITION <Nodo> +
    [**.PREDICATE** <ident> +]

<Parte> ::=
    .PART <ident>
    <ParteCabeçalho>
    .NET <Transição> +
    [**.MOOREOUTPUT** <Saída_Moore> +]
    [**.PREDICATEDESCRIPTION** <Descrição_Predicado> +]
    <Marcação_Inicial>

<ParteCabeçalho> ::=
    [**.INPUT** <ident> +]
    [**.OUTPUT** <ident> +]
    .PLACE <Nodo> +
    .TRANSITION <Nodo> +
    [**.PREDICATE** <ident> +]

<Transição> ::=
    <ident> **:** (<Condição> **||** ***) |-** <ident> **||** ***) ;**

<Condição> ::=

```
        <ident>
      | ! <ident>

<Saída_Moore> ::=
      <ident> |- (<ident> || *) ;

<Descrição_Predicado> ::=
      <ident> = <Função_Lógica> ;

<Função_Lógica> ::=
      <ident> <FLAux>
      | ! <Função_Lógica> <FLAux>
      | ( <Função_Lógica> ) <FLAux>

<FLAux> ::=
      + <Função_Lógica>
      | * <Função_Lógica>
      | ε

<Marcação_Inicial> ::=
      .MARKING <ident> +

<Nodo> ::=
      <ident> <NodoAux>

<NodoAux> ::=
      = <ident> ( <ident> * , <ident> * )
      | ε
```

Apêndice B

Aplicação ConPar

A invocação do aplicação é feita, na linha de comando do Sistema Operativo, e apresenta a seguinte sintaxe:

```
ConPar [opcoes] [< Fich_entrada] [> Fich_saida]
```

Se `Fich_entrada` ou `Fich_saida` não forem especificadas é usado a *stdin* e a *stdout*, respectivamente. A aplicação aceita qualquer das opções apresentadas no quadro B.1.

Símbolo	Significado
H	apresenta uma mensagem com ajuda
$+I$	escreve no ficheiro `Ident` a lista dos identificadores
$-I$	não escreve a lista dos identificadores (por defeito)
$+X$	escreve no ficheiro `Fich_saida` a lista de *hashing* dos identificadores
$-X$	não escreve a lista de *hashing* (por defeito)
$+T$	escreve no ficheiro `Table` a tabela global de símbolos
$-T$	não escreve a tabela global de símbolos (por defeito)
$+R$	escreve no ficheiro `Tree` a árvore sintáctica
$-R$	não escreve a árvore sintáctica (por defeito)
$+G$	escreve no ficheiro `Marking` o grafo de alcançabilidade
$-G$	não escreve o grafo de alcançabilidade (por defeito)
$+W$	escreve no ficheiro `Code` o código VHDL (por defeito)
$-W$	não escreve o código VHDL
$+B$	escreve o código VHDL com um bloco (por defeito)
$-B$	escreve o código VHDL com um processo
$+O$	escreve o código VHDL, seguindo a metodologia apresentada neste trabalho (regra disparo forte)
$-O$	escreve o código VHDL, seguindo a metodologia apresentada por Pardey e Bolton [PB91] (regra disparo fraca)

Quadro B.1: Opções da aplicação ConPar.

O ficheiro com código VHDL relativo ao controlador descrito em `Fich_entrada` é colocado no ficheiro `Code`.

Apresenta-se, no texto B.1, o código gerado[1] relativo ao controlador da figura 2.21. Este código foi obtido, usando a opção "$+B$". Pode pois ser simulado e sintetisado, usando as ferramentas do produto ALLIANCE. O código VHDL obtido para esse mesmo controlador, mas usando a opção "$-B$", razão pela qual é gerado um processo para controlo da marcação, encontra-se no texto B.2.

No texto B.3 apresenta-se o código gerado relativo ao controlador da figura 4.4. Este código foi obtido, usando a opção "-B",

O código gerado relativo ao controlador da figura 4.5 é o constante do texto B.4.

[1]Foram apagadas algumas linhas (assinaladas por ...) do ficheiro produzido, para encurtar o seu tamanho, mantendo contudo a legibilidade do mesmo. O mesmo foi feito para os restantes exemplos.

```
-- VHDL file generated by ConPar version 1.0

ENTITY controller IS
  PORT (reset, x1, x2, x3, relogio : IN BIT;
        y1, y2, y3, y4, y5 : OUT BIT);
END controller;

ARCHITECTURE dataflow OF controller IS

-- Place Signals
  SIGNAL p1  : REG_BIT REGISTER;
  SIGNAL Np1 : BIT;
  SIGNAL p2  : REG_BIT REGISTER;
  SIGNAL Np2 : BIT;
  ...
  SIGNAL p6  : REG_BIT REGISTER;
  SIGNAL Np6 : BIT;
-- Transition Signals
  SIGNAL t1, t2, t3, t4, t5, t6 : BIT;

BEGIN
  PART : BLOCK (relogio='1' AND NOT relogio'STABLE)
  BEGIN
    p1 <= GUARDED Np1 WHEN reset='0' ELSE '1';
    p2 <= GUARDED Np2 WHEN reset='0' ELSE '0';
    ...
    p6 <= GUARDED Np6 WHEN reset='0' ELSE '0';
  END BLOCK;

-- Dataflow description for transitions
  t1 <= x1 AND p1 AND NOT p2 AND NOT p3;
  t2 <= p2 AND NOT p5 AND (NOT p3);
  t3 <= x3 AND p3 AND NOT p4;
  t4 <= NOT x2 AND p4 AND NOT p6;
  t5 <= x2 AND p6 AND p5 AND NOT p1;
  t6 <= p6 AND NOT p3 AND (NOT x2 AND p5);

-- Dataflow description for next place markings
  Np1 <= t5 OR (p1 AND NOT t1);
  Np2 <= t1 OR (p2 AND NOT t2);
  Np3 <= t1 OR t6 OR (p3 AND NOT t3);
  Np4 <= t3 OR (p4 AND NOT t4);
  Np5 <= t2 OR (p5 AND NOT t5);
  Np6 <= t4 OR (p6 AND NOT t5 AND NOT t6);

-- Output Signals Equations
  y1 <= p4 OR p1;
  y2 <= p2;
  y3 <= p6;
  y4 <= t1 OR t5;
  y5 <= t2;

-- Transitions in conflict
   ASSERT NOT (t1 AND t6)
    REPORT "t1, t6 in conflict (output place p3)."
    SEVERITY ERROR;
   ASSERT NOT (t5 AND t6)
    REPORT "t5, t6 in conflict (input place p6)."
    SEVERITY ERROR;

-- No Enabled Transitions
  ASSERT NOT(t1='0' AND t2='0' AND ... AND t6='0')
    REPORT "Petri Net may be deadlocked"
    SEVERITY WARNING;

END dataflow;
```

Texto B.1: Primeiro exemplo de código VHDL gerado pela aplicação, usando a opção +B.

```
-- VHDL file generated by ConPar version 1.0

ENTITY controller IS
  PORT (reset, x1, x2, x3, relogio : IN BIT;
        y1, y2, y3, y4, y5 : OUT BIT);
END controller;

ARCHITECTURE dataflow OF controller IS

-- Place Signals
  SIGNAL p1, Np1 : BIT;
  SIGNAL p2, Np2 : BIT;
  ...
  SIGNAL p6, Np6 : BIT;
-- Transition Signals
  SIGNAL t1, t2, t3, t4, t5, t6 : BIT;

BEGIN
  PROCESS BEGIN
    WAIT UNTIL relogio'EVENT and relogio='1';
    IF reset='0' THEN
      p1 <= Np1;
      p2 <= Np2;
      ...
      p6 <= Np6;
    ELSE
      p1 <= '1';
      p2 <= '0';
      ...
      p6 <= '0';
    END IF;
  END PROCESS;

-- Dataflow description for transitions
  t1 <= x1 AND p1 AND NOT p2 AND NOT p3;
  t2 <= p2 AND NOT p5 AND (NOT p3);
  t3 <= x3 AND p3 AND NOT p4;
  t4 <= NOT x2 AND p4 AND NOT p6;
  t5 <= x2 AND p6 AND p5 AND NOT p1;
  t6 <= p6 AND NOT p3 AND (NOT x2 AND p5);

-- Dataflow description for next place markings
  Np1 <= t5 OR (p1 AND NOT t1);
  Np2 <= t1 OR (p2 AND NOT t2);
  Np3 <= t1 OR t6 OR (p3 AND NOT t3);
  Np4 <= t3 OR (p4 AND NOT t4);
  Np5 <= t2 OR (p5 AND NOT t5);
  Np6 <= t4 OR (p6 AND NOT t5 AND NOT t6);

-- Output Signals Equations
  y1 <= p4 OR p1;
  y2 <= p2;
  y3 <= p6;
  y4 <= t1 OR t5;
  y5 <= t2;

-- Transitions in conflict
  ASSERT NOT (t1 AND t6)
    REPORT "t1, t6 in conflict (output place p3)."
    SEVERITY ERROR;
  ASSERT NOT (t5 AND t6)
    REPORT "t5, t6 in conflict (input place p6)."
    SEVERITY ERROR;

-- No Enabled Transitions
  ASSERT NOT(t1='0' AND t2='0' AND ... AND t6='0')
    REPORT "Petri Net may be deadlocked"
    SEVERITY WARNING;

END dataflow;
```

Texto B.2: Código VHDL gerado pela aplicação, usando a opção -B.

```
-- VHDL file generated by ConPar version 1.0

ENTITY controller IS
  PORT (reset, x1, x2, x3, x4, x5, relogio : IN BIT;
        yb1, y1, y2, y3, y4 : OUT BIT);
END controller;

ARCHITECTURE dataflow OF controller IS

-- Place Signals
  SIGNAL pb1, Npb1 : BIT;
  SIGNAL pb2, Npb2 : BIT;
  SIGNAL p1, Np1 : BIT;
  SIGNAL p2, Np2 : BIT;
  ...
  SIGNAL p8, Np8 : BIT;
-- Transition Signals
  SIGNAL tb1, tb2 : BIT;
  SIGNAL t1, t2, t3, t4, t5, t6 : BIT;

BEGIN
  PROCESS BEGIN
    WAIT UNTIL relogio'EVENT and relogio='1';
    IF reset='0' THEN
      pb1 <= Npb1;
      pb2 <= Npb2;
      p1 <= Np1;
      p2 <= Np2;
      ...
      p8 <= Np8;
    ELSE
      pb1 <= '1';
      pb2 <= '0';
      p1 <= '1';
      p2 <= '1';
      ...
      p8 <= '0';
    END IF;
  END PROCESS;

-- Dataflow description for transitions
  tb1 <= pb2 AND NOT pb1 AND (p8);
  tb2 <= pb1 AND NOT pb2 AND (NOT p1);
  t1 <= p8 AND x1 AND NOT p2 AND NOT p1;
  t2 <= p1 AND p2 AND x2 AND NOT p3;
  t3 <= p3 AND x3 AND NOT p5 AND NOT p4;
  t4 <= p4 AND NOT p6 AND (p7);
  t5 <= p5 AND x4 AND NOT p7;
  t6 <= p6 AND p7 AND x5 AND NOT p8;

-- Dataflow description for next place markings
  Npb1 <= tb1 OR (pb1 AND NOT tb2);
  Npb2 <= tb2 OR (pb2 AND NOT tb1);
  Np1 <= t1 OR (p1 AND NOT t2);
  Np2 <= t1 OR (p2 AND NOT t2);
  Np3 <= t2 OR (p3 AND NOT t3);
  Np4 <= t3 OR (p4 AND NOT t4);
  Np5 <= t3 OR (p5 AND NOT t5);
  Np6 <= t4 OR (p6 AND NOT t6);
  Np7 <= t5 OR (p7 AND NOT t6);
  Np8 <= t6 OR (p8 AND NOT t1);

-- Ouput Signals Equations
  yb1 <= pb1;
  y1 <= p1;
  y2 <= p2;
  y3 <= p4;
  y4 <= p7;

END dataflow;
```

Texto B.3: Segundo exemplo de código VHDL gerado pela aplicação.

```
-- VHDL file generated by ConPar version 1.0

ENTITY controller IS
  PORT (reset, x1, x2, relogio : IN BIT;
        y1, y2, y3 : OUT BIT);
END controller;

ARCHITECTURE dataflow OF controller IS

-- Place Signals
  SIGNAL p1_pf  : REG_BIT REGISTER;
  SIGNAL Np1_pf : BIT;
  SIGNAL p1_pe  : REG_BIT REGISTER;
  SIGNAL Np1_pe : BIT;
  ...
  SIGNAL p1_pa  : REG_BIT REGISTER;
  SIGNAL Np1_pa : BIT;
  SIGNAL p2  : REG_BIT REGISTER;
  SIGNAL Np2 : BIT;
  SIGNAL p3  : REG_BIT REGISTER;
  SIGNAL Np3 : BIT;
-- Transition Signals
  SIGNAL t1, t2, t3 : BIT;
  SIGNAL p1_td, p1_tc, p1_tb, p1_ta: BIT;

BEGIN
  PART : BLOCK (relogio='1' AND NOT relogio'STABLE)
  BEGIN
    p1_pf <= GUARDED Np1_pf WHEN reset='0' ELSE '0';
    p1_pe <= GUARDED Np1_pe WHEN reset='0' ELSE '0';
    p1_pd <= GUARDED Np1_pd WHEN reset='0' ELSE '0';
    p1_pc <= GUARDED Np1_pc WHEN reset='0' ELSE '0';
    p1_pb <= GUARDED Np1_pb WHEN reset='0' ELSE '0';
    p1_pa <= GUARDED Np1_pa WHEN reset='0' ELSE '0';
    p2 <= GUARDED Np2 WHEN reset='0' ELSE '0';
    p3 <= GUARDED Np3 WHEN reset='0' ELSE '1';
  END BLOCK;

-- Dataflow description for transitions
  t1 <= x1 AND p1_pf AND NOT p2;
  t2 <= x2 AND p2 AND NOT p3;
  t3 <= p3 AND NOT p1_pa AND (NOT x1 AND x2);
  p1_td <= NOT p1_pf AND p1_pd AND p1_pe AND NOT x2;
  p1_tc <= NOT p1_pe AND p1_pc AND x2;
  p1_tb <= NOT p1_pd AND p1_pb;
  p1_ta <= NOT p1_pc AND NOT p1_pb AND p1_pa AND x1;

-- Dataflow description for next place markings
  Np1_pe <= p1_tc OR (p1_pe AND NOT p1_td);
  Np1_pd <= p1_tb OR (p1_pd AND NOT p1_td);
  Np1_pc <= p1_ta OR (p1_pc AND NOT p1_tc);
  Np1_pb <= p1_ta OR (p1_pb AND NOT p1_tb);
  Np1_pa <= t3 OR (p1_pa);
  Np1_pf <= (p1_pf AND NOT t1);
  Np2 <= t1 OR (p2 AND NOT t2);
  Np3 <= t2 OR (p3 AND NOT t3);

-- Output Signals Equations
  y1 <= p2 OR p1_pc;
  y2 <= p1_td OR p1_pa;
  y3 <= p3;

-- No transitions in conflict

-- No Enabled Transitions
  ASSERT NOT (t1='0' AND ... AND p1_ta='0')
    REPORT "Petri Net may be deadlocked"
    SEVERITY WARNING;

END dataflow;
```

Texto B.4: Terceiro exemplo de código VHDL gerado pela aplicação.

Apêndice C

Comunicação apresentada no Encontro Nacional do Colégio de Engenharia Electrotécnica

Neste apêndice, apresenta-se a comunicação apresentada no 1ºEncontro Nacional do Colégio de Engenharia Electrotécnica, organizado pela Ordem dos Engenheiros, nos dias 12 e 13 de Maio de 1994, em Lisboa. O artigo reflete basicamente os resultados produzidos pela investigação realizada, no decorrer deste trabalho, tendo sido incluído na documentação do referido encontro.