Geração Automática de Interface para Incorporação de *IP-Cores* em Ambientes SoC

Julio Alexandrino de Oliveira Filho

Dissertação submetida ao Centro de Informática
da Universidade Federal de Pernambuco
como requisito parcial para obtenção
do grau de Mestre em Ciência da Computação
Sob a orientação do Professor Dr. Manoel Eusébio de Lima
e co-orientação do Professor Dr. Paulo Romero Martins Maciel

Grupo de Engenharia da Computação
Centro de Informática
Universidade Federal de Pernambuco
Brasil
Abril, 2003

Dedicado à

Selma, minha mãe.

Julio, meu pai.

Mariana, minha irmã.

Geração Automática de Interface para Incorporação de IP-Cores em Ambientes SoC

Julio Alexandrino de Oliveira Filho

Dissertação submetida ao Centro de Informática
da Universidade Federal de Pernambuco
como requisito parcial para obtenção
do grau de Mestre em Ciência da Computação
Abril, 2003

Resumo

Este trabalho enfoca a síntese automática do processo de interface entre dois módulos comunicantes. Seu objetivo principal é a rápida incorporação de módulos de *IP-Cores* em projetos de ambientes *System-On-Chip*. Uma metodologia baseada no formalismo de redes de Petri é apresentada para descrever os protocolos individuais dos módulos e guiar a síntese do processo de interface. Posteriormente, o processo sintetizado é traduzido para uma descrição VHDL do circuito a ser implementado e pode ser utilizado para conectar estruturalmente os módulos comunicantes. O uso do formato intermediário em redes de Petri, nesta metodologia, permite a aplicação de diversas técnicas conhecidas de análise, verificação e validação, garantindo resultados "corretos por construção". Um ferramenta de CAD foi desenvolvida para realizar, de forma automática, as atividades propostas ao longo da metodologia. Um módulo *Multiply And Accumulate* preparado com uma interface *AMBA* foi incorporado á um sistema NIOS baseado em um barramento *AVALON*.

Agradecimentos

À Deus, por tudo.

Em especial ao Professor Manoel Eusébio, pela orientação firme e presente a este trabalho e pela amizade.

Ao Professor Paulo Romero Maciel, pelo vasto conhecimento apreendido sobre redes de Petri.

Aos Professores Edna Natividade, Sérgio Cavalcante e Marcília Andrade.

À Georgia, pela atenção, cuidado, carinho e companheirismo.

À Sônia, minha madrinha, pelo carinho.

Ao Professor Francisco (Chico) Luiz dos Santos e toda a equipe do projeto Aroma, pelo apoio constante e incondicional.

Aos amigos do Grupo de Engenharia da Computação, e em especial a Bruno Celso e Juliana Moura, pelo apoio na implementação do estudo de caso. A Cristiano Coelho, pelas conversas construtivas e iluminadoras ao longo do caminho.

Ao grande amigo Frederico Braga.

A todos os amigos, testemunhas pacientes das noites dedicadas a este trabalho.

Aos Beatles. A Chaves e Chapolin. A Renoir. Ao vampiro Lestat.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Os últimos cem anos da história humana foram palco de um surpreendente avanço tecnológico impulsionado pelo conhecimento científico e pelo complexo jogo dos sistemas econômicos. A tecnologia de sistemas eletro-eletrônicos, em especial, foi constantemente abastecida pelo contínuo progresso nos processos de fabricação de dispositivos semicondutores, e ao mesmo tempo sofreu a demanda crescente dos mercados de bens de consumo. Nos dias atuais, às portas de um novo século e um novo milênio, os sistemas eletrônicos permeiam a vida cotidiana dos grandes centros urbanos aos mais distantes pontos do planeta, alterando profundamente as relações sociais e a vida humana. O explosivo crescimento na capacidade de integração de dispositivos eletrônicos gerou um forte potencial de performance e funcionalidade mantendo os custos finais em níveis bastante atrativos para o mercado consumidor. Na prática, verifica-se um alto nível de incorporação destes sistemas em produtos como automóveis, eletrodomésticos, equipamentos de comunicação pessoal e médico-hospitalares. As maiores barreiras técnicas para este cenário dizem respeito às metodologias e ferramentas de suporte ao projeto dos sistemas eletrônicos, uma vez que nenhuma destas acompanhou de perto o avanço tecnológico dos processos de fabricação em silício. De forma resumida, as técnicas de projeto atuais remontam a sistemas construídos com relativamente poucos elementos e sobre restrições de performance, funcionalidade e consumo pouco exigentes. O futuro da integração

de sistemas eletrônicos e o uso pleno do potencial tecnológico dos dispositivos atuais depende fortemente do amadurecimento na forma de se projetar tais sistemas.

No tocante a metodologias e ferramentas que oferecem suporte ao projeto de sistemas eletrônicos, Chang [1] argumenta que é necessário uma completa mudança de paradigma, equivalente à ocorrida com o advento dos *application-specific integrated circuits* (ASICs), e que simples mudanças incrementais nas metodologias de projeto de CIs não serão suficientes. Este novo paradigma deve ser capaz de reduzir tempo e esforço no desenvolvimento, aumentar a previsibilidade dos resultados, lidar com a heterogeneidade envolvida nos novos sistemas e reduzir os riscos criados pelo aumento de complexidade, mantendo atrativos os aspectos de funcionalidade e custo. Neste sentido, a pesquisa por novos métodos e ferramentas aponta em duas direções concorrentes: o projeto baseado em alto nível de especificação e abstração e o reuso intensivo de módulos pré-fabricados e caracterizados.

As principais metodologias que propõem o projeto em alto nível de abstração aproveitam a característica híbrida de hardware e software, presente na grande maioria dos sistemas atuais, para construir o paradigma de *hardware/software co-design* [2], onde estes dois aspectos do projeto são planejados, implementados e integrados concorrentemente. O ponto forte das metodologias baseadas em *hardware/software co-design* é o tratamento do sistema em alto nível (como por exemplo especificação e/ou arquitetura) com suporte de ferramentas capazes de sintetizar, de forma integrada, a implementação do sistema nas suas características físicas (hardware) e programáveis (software). Para tal, é comum o uso de linguagens especiais desenvolvidas para especificação e manipulação a nível de sistema, como por exemplo, as linguagens de descrição de hardware (*VHDL, AHDL, Verilog*) [3] [4], linguagens para descrição de sistema (*SystemC, SpecC*) [5] [6] e a aplicação de métodos formais (*CSP, Redes de Petri*) [7].

A figura 1.1 esquematiza, de uma forma geral, as principais etapas envolvidas em uma metodologia de *hardware/software Co-Design*. Inicialmente é realizado um conjunto de análises para os requisitos e restrições do projeto, determinando sua natureza e estas análises são utilizadas para especificar o sistema. Neste momento, entram em jogo diversas linguagens de alto nível e esquemas de modelagem, con-

forme citado anteriormente, que envolvem linguagens de descrição de hardware e métodos formais. Esta especificação é então particionada determinando elementos a serem implementados em hardware e elementos a serem implementados em software. O processo de particionamento deve preservar a semântica do sistema, ou seja, não deve permitir alteração dos requisitos e restrições especificados. Posteriormente, as descrições particionadas são sintetizadas em suas respectivas bases e um sistema de comunicação entre estes é gerado. Uma etapa de verificação e validação ocorre paralelamente garantindo os resultados e servindo de base para estimativa e otimização de aspectos do sistema. Finalmente, o sistema é integrado, simulado e implementado em uma tecnologia alvo.

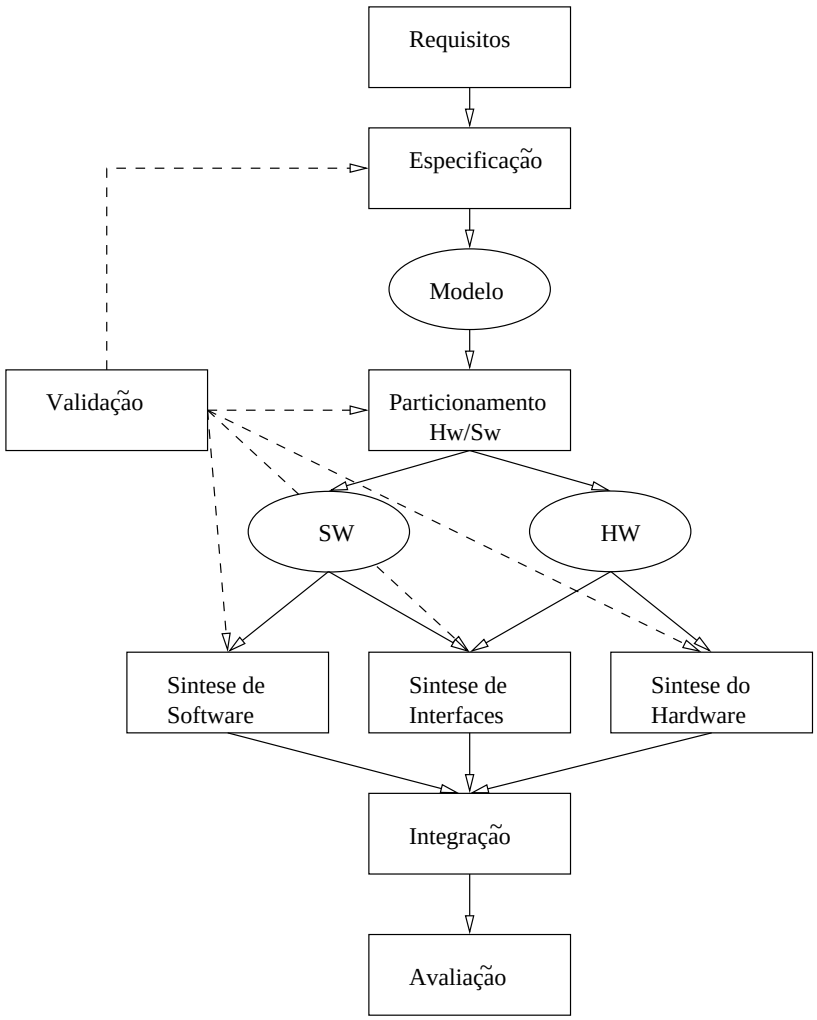


Figura 1.1: Metodologia de *Hardware/Software Co-Design(Fonte [8])*

A segunda abordagem ao problema do *gap* de produtividade, embora não dis-

sociada da anterior, é o reuso intensivo de módulos de hardware e software pré-projetados. Os sistemas são, segundo esta proposta, montados a partir de blocos de hardware e/ou software já bem caracterizados e conhecidos. Em software, estes blocos são comumente chamados de "componentes" e em hardware tomam a denominação de *"cores"*. Futuramente, os *cores*, baseados em uma política de "propriedade intelectual", substituirão os atuais circuitos integrados e serão peças dominantes no desenvolvimento de dispositivos eletrônicos. De fato, o aumento da escala de integração permite, atualmente, que sistemas inteiros sejam construídos em um só *chip* utilizando uma analogia aos sistemas de integração em placas de circuito impresso estabelecidos nas últimas três décadas. As especificações dos dispositivos de circuito integrado serão substituídos gradualmente por descrições sintetizáveis e em alto nível de sua funcionalidade e/ou estrutura, e a integração em um único dispositivo será realizado com métodos similares aos utilizados na montagem de placas dos sistemas atuais.

Nasce então um conceito largamente explorado neste trabalho, o *system-on-chip (SoC)*. Definimos *SoC* como um circuito integrado complexo que integra a maior parte dos elementos funcionais de um produto final em um só dispositivo. Em geral, um projeto de *SoC* incorpora um processador (programável e responsável pela execução da parte software do sistema) e *cores* de hardware periféricos altamente especializados, interligados convenientemente. A diferença em relação às metodologias de projeto atuais começa a tornar-se evidente por causa do aumento proporcional da complexidade com o número de elementos envolvidos. Em aproximadamente dez anos o número de portas lógicas (medida comumente utilizada para expressar o grau de integração) passou de cerca de 50.000 para mais de 10.000.000 portas em um único chip, numa proporção exponencial prevista pela conhecida *Lei de Moore*. Esta medida exige um novo paradigma de projeto com um forte apelo ao reuso de componentes pré-existentes, visando redução do tempo de projeto e conseqüente *time-to-market* do produto.

Na prática atual de projetos, o reuso de componentes acontece nas primeiras fases do projeto e é normalmente pautado pelo conhecimento do especialista. Em verdade, não existe um cultura estabelecida do reuso, a qual está fortemente ligado

ao fator humano, tornando-se proibitivo quando os sistemas passam a ser desenvolvidos por diferentes grupos e empresas em diferentes lugares e momentos. O conceito mais recentemente proposto para reusabilidade envolve fundamentalmente uma cultura de reuso planejado. "Reuso é um requisito para liderança a curto prazo e para sobrevivência a médio e longo prazo[1] [1]", afirmam os gigantes da indústria eletrônica. O caminho para sua implantação passa pela criação de módulos especificados em alto nível (funcionalidade) e uma política universal de integração.

É exatamente neste ponto, o da integração entre os módulos em SoC, que são construídas as bases deste trabalho. De forma mais clara, a facilidade de integração de um *core* em um número adverso de sistemas afeta diretamente sua reusabilidade pois determina predominantemente o esforço e tempo gastos na fase de integração do projeto. A interface de comunicação precisa ser simples o suficiente para permitir uma rápida incorporação do módulo ao projeto, no entanto, não é conveniente que seja excessivamente específica para um ambiente pois, certamente, o aproveitamento em outras plataformas será prejudicada.

A VSI Alliance [9] argumenta que os projetos de *cores* poderão ser caracterizados em dois grandes grupos:

IP Centric Os *cores* são criados de forma a serem facilmente customizados para reuso, com larga faixa de aplicação, habilidade de implementação em plataformas diferentes (como *standard-cell*, *FPGA*, *sea-of-gates*), facilidade de verificação e validação em ambientes diversos.

Integration Centric Os *cores* são elaborados para serem reutilizados sem customizações, voltados para nichos específicos de mercado e famílias restritas de aplicações, com planejamento prévio das fases de implementação e integração e reduzida necessidade de re-verificação e validação.

Os módulos do primeiro grupo (*IP-Centric*) tendem a assumir uma postura de padronização de sua interface de comunicação. Desta forma, os *cores* são gerados visando uma plataforma alvo onde são assumidas prerrogativas como características

[1]National Technology Roadmap for Semiconductors, 1994; e National Technology Roadmap for Semiconductors, 1997. Disponível no site `www.sematech.org/public/roadmap/index.htm`.

físicas (elétricas, tecnológicas) e protocolos de comunicação. Esta visão é promissora no aspecto de aumento da produtividade pois aspira ao que costuma-se chamar "*plug-and-play*", no entanto, não deve ser dominante pois dificulta a troca de módulos entre grupos que não utilizam as mesmas prerrogativas. Por outro lado, os membros do segundo grupo serão fabricados de forma a serem mais "genéricos" dificultando aspectos de validação e verificação, mas facilitando o intercâmbio entre equipes de projetos e extendendo o mercado alvo.

Os *cores* construídos segundo uma abordagem *IP Centric* possuem uma interface mais simples, no entanto, não necessariamente adequada a um padrão. Os circuitos auxiliares que permitirão a integração em um sistema maior são construídos na fase de incorporação do *core*.

Do estudo e entendimento dos desafios envolvidos na fase de incorporação dos *cores* no projeto de *SoCs* foi elaborada a proposta deste trabalho, visando primordialmente, contribuir para o desenvolvimento das metodologias de projeto e diminuir o contraste estabelecido entre a tecnologia e a técnica.

1.2 Proposta de Tese

A proposta deste trabalho insere-se na fase de incorporação dos *cores* ao fluxo de projeto de sistemas eletrônicos, em particular, de *systems-on-chip*. Tem como objetivo a determinação de uma metodologia para geração automática de processos de interface a serem utilizados na incorporação destes *cores*, e apresenta como produto final a implementação de uma ferramenta de CAD para realizar e validar as idéias propostas na metodologia.

A idéia geral do trabalho pode ser visualizada na figura 1.2. Consiste basicamente em facilitar o processo de conexão entre dois módulos comunicantes de um sistema através da geração automática do processo de interface entre eles. Na prática, a tarefa de rapidamente integrar os *cores* ao sistema ainda não é uma realidade por varias razões, incluindo:

- A falta de ferramentas eficientes para a integração de *cores* em ambientes *SoC*.

Atualmente, a integração de módulos ao projeto é uma atividade eminentemente manual e, portanto, susceptível a erros, forçando o projetista a lidar diretamente com funcionalidade, característica elétricas complexas e aspectos customizáveis de dezenas de pinos. A incorporação de *cores* em estágios mais elevados de abstração pode ser interessante para resolver estes problemas;

- Complexidade do projeto físico. Integrar *cores* é mais que construir o circuito de interconexão entre eles, e pode tornar-se uma tarefa desafiadora pois efeitos imprevisíveis como ruído e capacitâncias parasitas podem afetar a performance do sistema integrado.

- O projeto orientado a reusabilidade deve considerar a integração de *cores* de diversos fabricantes, no entanto, ainda não existem padrões estabelecidos. Mais ainda, o processo de integração deve considerar várias questões referentes às interfaces, como por exemplo temporização, que podem fazer com que o sistema apresente falhas, mesmo que os módulos estejam individualmente corretos.

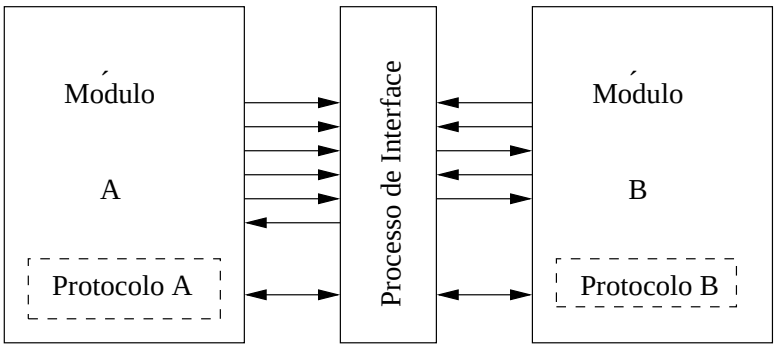


Figura 1.2: Geração automática do processo de interface.

O problema pode ser formulado como proposto por Gajski [10] e posteriormente por Passerone [11]: *"Dado dois módulos comunicantes e uma descrição dos dois protocolos que cada um deles utiliza para transferir dados, determinar uma interface tal que a transferência de dados seja consistente a ambos os protocolos. Um* ***processo de interface*** *responde adequadamente aos sinais de controle de ambos os protocolos e realiza o sequenciamento de dados entre eles, em outras palavras, traduz um protocolo no outro."*

A solução proposta por este trabalho é pode mostrar-se interessante, não só pelo fato de automatizar parte do processo de integração, mas também por ser baseada em um formalismo robusto (Redes de Petri) que permite:

- Uma geração automatizada e "correta por construção" do processo de interface;
- Pontos de entrada e saída da geração de interface definidos em alto nível de abstração (comportamental), adequado aos modernos modelos e metodologias de projeto, em especial de *SoCs*;
- Verificação, validação e simulação do processo gerado;

Além destas contribuições iniciais, a metodologia desenvolvida soma positivamente ao cenário atual de projetos de circuitos eletrônicos, pois:

- Reúne em um só fluxo de projeto os trabalhos de conversão automática de protocolo desenvolvidos por Bill Lin e Steven Vercauteren [12] baseados em redes de Petri e os processos de síntese de circuitos a partir da mesma base desenvolvidos por Jordi Cortadella, Alexander Yakovlev, et. al. [13] [14];
- Introduz, ainda que informalmente, as bases para uma nova técnica de síntese de código VHDL a partir de Redes de Petri, em um formato ainda não apresentado, ao melhor de nosso conhecimento, na literatura atual;

A figura 1.3 descreve rapidamente o fluxo de atividades proposto.

A metodologia tem como ponto de entrada diagramas temporais que descrevem o comportamento das interfaces a serem integradas. Diagramas temporais fazem parte da cultura estabelecida entre projetistas, sendo uma forma comum de especificação em *data-sheets* e na documentação de módulos de *hardware.* Estes diagramas precisam ser marcados pelo projetista para estabelecer relações de causalidade entre os eventos. Os diagramas marcados são traduzidos em um formalismo matemático conhecido como Redes de Petri e manipulados para a geração do processo de interface. Durante esta fase são realizadas análise qualitativa de propriedades e verificação do processo gerado garantindo parâmetros adequados de construção. Finalmente, o processo de interface descrito em redes de Petri é convertido para uma descrição

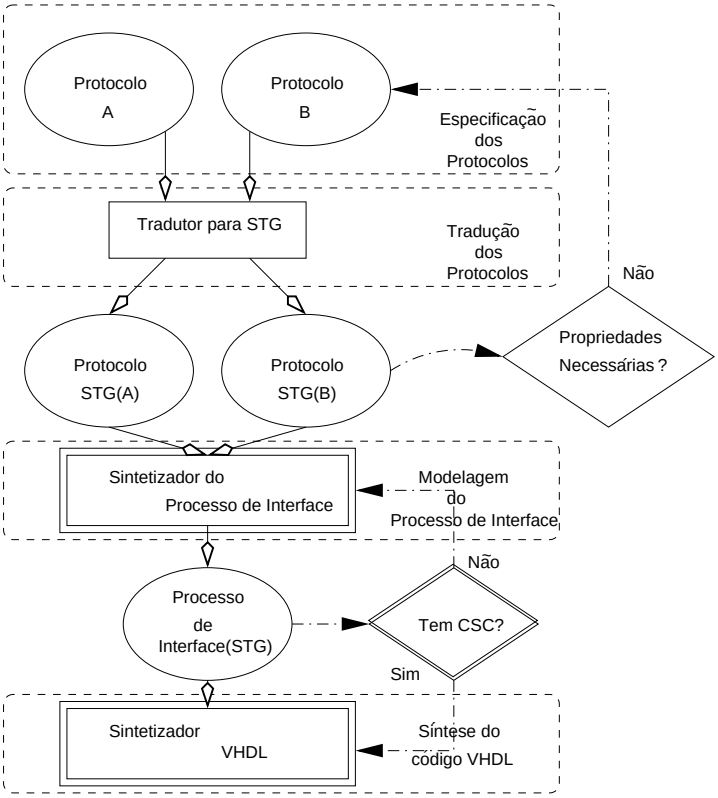


Figura 1.3: Fluxo da geração do processo de Interface

comportamental em VHDL do circuito o implementará. Esta descrição em VHDL é o ponto de saída desta metodologia e está pronto para ser incorporado ao sistema no processo de integração do *core* ao *SoC*. Maiores detalhes serão introduzidos posteriormente nas próximas seções.

Devido ao grande escopo de problemas envolvidos na metodologia proposta, assumimos alguns pontos simplificadores na implementação deste trabalho no sentido de delimitar o tamanho de espaço de soluções para o projeto:

- A comunicação é "ponto-a-ponto", ou seja, o processo de interface gerado não é compartilhado com nenhum outro módulo do sistema. Todavia a técnica pode ser utilizada hierarquicamente ou um dos módulos comunicantes pode ser um barramento, de forma a permitir a integração de vários elementos aos sistema;

- Assume-se que os protocolos individuais dos módulos possuem um ciclo bem definido para todas as possíveis operações envolvidas na transferência de dados. Isto implica que esta metodologia não contempla ainda transferências

comumente conhecidas como "modo *burst*" em que uma sequência de dados é transferido de uma só vez entre os módulos;

- A conversão de protocolo é realizada a nível dos sinais de controle. Admite-se que o *datapath* entre os módulos comunicantes está estabelecido. Esta é uma prerrogativa apenas simplificadora para permitir uma completa automação do procedimento. Não é necessária porém se permitimos a interação do projetista nas diversas fases da metodologia;

O trabalho desenvolvido mostra ser capaz de produzir soluções robustas e elegantes para uma fase crítica do projeto de *SoCs*, a integração de módulos, sendo ponto especialmente importante para as metodologias de projeto que evidenciam a reusabilidade de módulos.

1.3 Estrutura da Tese

Organizamos esta dissertação em seis capítulos principais. O Capítulo 1 reflete sobre a motivação em automatizar a integração de módulos ao projeto de sistemas digitais, em especial *SoCs*, delineando o problema a ser resolvido e delimitando o espaço de soluções iniciais desta proposta.

O Capítulo 2 introduz o estado da arte da geração de processos de interface e delineia os principais trabalhos que serviram de base para este estudo e o desenvolvimento da metodologia proposta.

O Capítulo 3 fornece as ferramentas formalizadoras desta proposta, definindo conceitos importantes a serem utilizados durante todo o trabalho. Uma breve abordagem ao formalismo de Redes de Petri e um estudo das propriedades importantes deste formalismo são explorados de forma a tornar clara a formulação do corpo metodológico proposto.

O Capítulo 4 delineia e formaliza a metodologia de geração do processo de interface e estabelece as bases para a ferramenta de CAD a ser desenvolvida com fruto deste trabalho.

O Capítulo 5 documenta a ferramenta de CAD implementada, e que realiza de

forma automática a geração do processo de interface e as análises e verificações necessárias para a construção correta deste.

Tendo em vista a validação do sistema, o capítulo 6 descreve um estudo de caso que utiliza a metodologia proposta para integração de um *core* em um sistema *SoC* baseado na plataforma *Excalibur Nios* [15].

Por fim, o Capítulo 6 discute de forma crítica os resultados obtidos, retirando conclusões e estabelecendo trabalhos futuros a serem desenvolvidos nesta promissora proposta.

1.4 Resumo

Neste capítulo, introduzimos ao leitor os problemas modernos do projeto de sistemas digitais, em especial, a diferença contrastante entre o potencial oferecido pelos atuais dispositivos semicondutores e as metodologias de projeto de sistemas para estes dispositivos. Argumentamos que a solução destes problemas reside na adoção de metodologias de projeto em alto nível de abstração, fortemente apoiados em reusabilidade e suportados por ferramentas automáticas de CAD. Introduzimos o problema geral de integração de módulos ao sistema e apresentamos a formulação do problema específico a ser abordado neste trabalho: a geração automática do processo de interface visando facilitar a incorporação de *cores* em ambientes *SoCs*. E delineamos a estrutura geral a ser apresentada ao longo deste trabalho.

Capítulo 2

Estado da Arte

Antes de delinearmos a explicação da técnica de geração do processo de interface estudada neste trabalho, convém projetar o cenário atual de desenvolvimento neste sentido. Basicamente, três abrangentes áreas de pesquisa merecem foco no intuito de contextualizar as propostas desta dissertação: metodologias de especificação e geração de interface, projeto de sistemas *on-chip* e *Ip-Cores*, e circuitos assíncronos.

2.1 Metodologias de Especificação e Geração de Interface

O projeto de sistemas digitais mapeia a funcionalidade de um sistema em um conjunto de componentes (*chips*, *IP-Cores,* memórias, processadores, blocos lógicos, etc.). Esta estrutura, no entanto, só forma uma arquitetura quando a malha de comunicação entre estes componentes é estabelecida [16]. A especificação desta malha de comunicação deve ir além da simples interconexão entre os blocos, sendo que, para a implementação em hardware ser realizada são necessárias informações sobre o conjunto de sinais e o protocolo de comunicação que trafega entre os módulos. Nos diversos fluxos de projeto atualmente estudados, dois cenários são possíveis: (seção 2.1.1) os sinais e os protocolos de comunicação seguem uma padronização, não podendo ser alterados; e/ou a interface de comunicação é customizada seguindo simplesmente as necessidades do projeto.

2.1.1 Síntese de Interfaces a nível de Sistema

Na primeira abordagem, a estrutura de pinos e os protocolos de comunicação dos módulos envolvidos oferecem pouca ou nenhuma customização. Impressionantemente, esta opção é a que oferece melhores oportunidades de reuso dos módulos, pois possuem, frequentemente, características de projeto direcionadas a integração [1]. Grandes grupos da indústria de sistemas eletrônicos [9] [17] apostam na padronização de interfaces e barramentos, em especial para projetos de sistemas *on-chip*.

A pergunta então a ser feita é: o que fazer quando deseja-se integrar dois módulos que não possuem compatibilidade de sinais e protocolo? A resposta é simples e reside em introduzir, entre os dois módulos, um processo de interface. A idéia é mostrada na figura 2.1 e consiste em gerar um circuito que permita a transferência consistente de dados entre os módulos através da conversão dos protocolos. Na literatura, este módulo interposto recebe nomes diversos, com pequena variação no significado: "processo de interface", "*wrapper*", "*bridge*", "*transducer*" e "conversor de protocolo", entre outros.

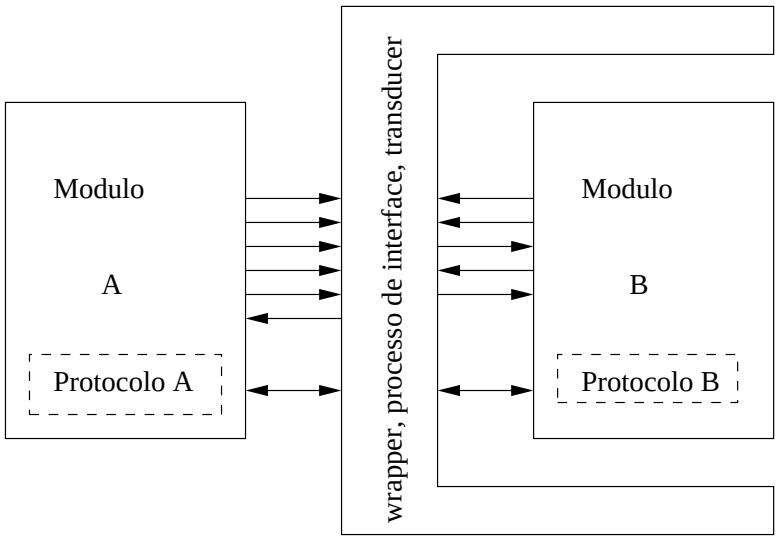


Figura 2.1: Ligando dois módulos comunicantes com protocolos incompatíveis

É importante enfatizar que a implementação deste circuito deve primar pela simplicidade, sob pena de impactar diretamente no custo de comunicação entre os módulos. Por outro lado, a não trivialidade desta compatibilização entre protocolos

impede que estes circuitos sejam vistos como simples *glue-logic,* ou seja, um circuito combinacional para conversão de sinais. Narayan e Gajski [10] argumentam que a geração do processo de interface deve contemplar diversos aspectos da incompatibilidade entre protocolos:

- Primeiro, não deve ser preciso especificar detalhes que não sejam explicitamente necessário a solução do problema. A especificação inicial, por exemplo, não deve conter requisitos como representar sinais de sincronismo com o mesmo *label*, informações de *time-out* que sejam inerentes ao funcionamento interno dos módulos e não da interface, entre outros;

- É interessante que a interface gerada seja "simulável". Ou seja, o projetista deve ser capaz de realizar, ao menos, uma verificação funcional do processo de interface. É valido acrescentar que este aspecto ganha importância, em especial, nos fluxos de trabalho que aspiram a grande índice de reusabilidade. Para estes, é interessante também a possibilidade de um modelamento que facilite análise, validação, teste e facilidades no mapeamento tecnológico.

- Finalmente, a tradução entre protocolos deve resolver inconsistências nos sinais de dados e prever políticas de arbitragem. Por inconsistência no *datapath* entende-se diferentes tamanhos no barramento de dados, representações diferentes (como por exemplo *big endian* ou *little endian*) e conversão de tipos de dados. As políticas de arbitragens definem a forma de sincronismo da comunicação a um nível mais global.

Com base nestes parâmetros pode-se analisar alguns trabalhos recentes.

Os trabalhos mais recentes que abordam a síntese de interface parecem ter como ponto de partida os estudos de Gaetano Borriello [18] [19], no final dos anos 80. Borriello introduz a especificação de protocolos através de grafos de eventos, ocorridos nos sinais da interface, como uma forma de estabelecer uma base para sincronização e sequenciamento dos dados. A partir desta especificação é realizada a síntese de *"transducers"*, ao que ele define como o circuito lógico que conecta dois blocos de circuitos, e que nada mais é que a combinação dos grafos individuais através

da fusão de nós com sinais de mesmo nome. O circuito em si é gerado utilizando uma estratégia de casamento de padrões de uma biblioteca com a estrutura do grafo sintetizado e depois otimizado.

A necessidade imposta ao projetista de demarcar os pontos de sincronismo na transferência dos dados através de sinais de mesmo nome é visto como um ponto fraco do trabalho, e o método utilizado para síntese do circuito final carece de suporte à análise e simulação. Entretanto, este é sem dúvida um marco na pesquisa da síntese de processos de interface. Quase todos os trabalhos que se seguiram utilizam-se da mesma base de especificação (grafo de eventos nos sinais), ou similar, e adotam idéias lançadas nesta pesquisa.

A tentativa de implementar as idéias de Borriello a partir de uma especificação de mais alto nível veio no trabalho de Sanjiv Narayan e Daniel Gajski [10]. Esta abordagem utiliza como ponto de partida códigos em linguagem de descrição de hardware (originalmente VHDL) detalhando o número de sinais de controle e a sequência de transferência dos dados. Os protocolos devem ser descritos com base em um conjunto de cinco tipos atômicos de operações, conforme indicado na tabela 2.1.

Operação Atômica	Equivalente HDL	Operação dual a ser implementada no processo de interface
Espera por evento em linha de controle	*wait until (Con = '1')*	*Con <= '1'*
Atribuição a linha de controle	*Con <= '1'*	*wait until (Con = '1')*
Leitura em linha de dado	*var <= Data*	*Data <= var*
Escrita em linha de dado	*Data <= var*	*var <= Data*
Interfalo de tempo	*wait for 100 ns*	*wait for 100 ns*

Tabela 2.1: Operações atômicas de um protocolo e suas implementações em HDL

Além de utilizar combinações de tipos básicos das operações definidas para montar o protocolo, o projetista precisa representá-los como um conjunto ordenado de "relações" (figura 2.2). Uma relação descreve um conjunto de atividades que devem ser executadas mediante a ocorrência de uma condição (um evento ou um intervalo de tempo). As relações são, então, agrupadas em blocos que representam unidades

de transferência dos dados. O processo de interface é sintetizado reordenando-se estes blocos em um só conjunto e substituindo cada operação atômica por seu dual (tabela 2.1).

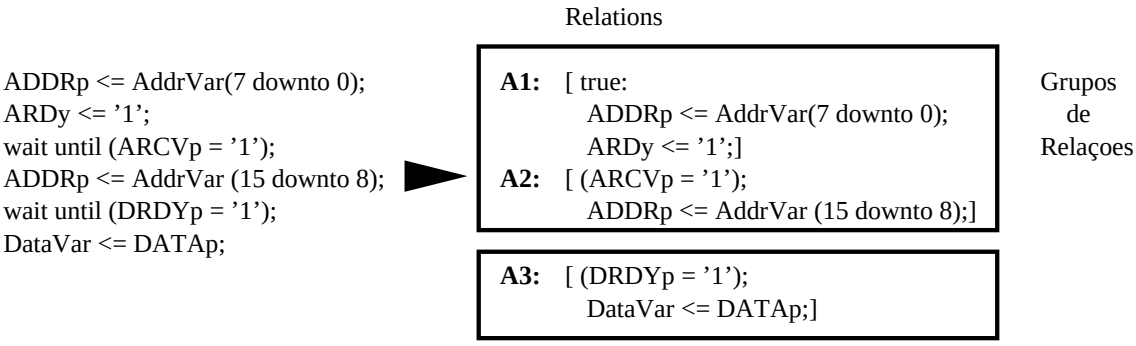


Figura 2.2: "Relações" de um protocolo

Esta metodologia oferece facilidades para implementação em diversas plataformas, bem como suporte à simulação, devido à sua abordagem através de linguagens de descrição de hardware. Todavia, há a falta de um formalismo que permita uma síntese "correta por construção" e facilite a análise de condições críticas. Requisitos temporais, por exemplo, podem ser utilizados para efeito de simulação, mas não são garantidos na implementação do circuito.

A construção de redes de interconexão pino a pino é proposta por Gupta [20] para a geração de circuitos combinacionais otimizados, em formato de *glue logic*. A especificação inicial parte de grafos ou diagramas que descrevem os requisitos temporais para a sinalização, indicando a janela de tempo em que um evento deve ocorrer em um sinal para que a operação seja realizada. Esta especificação é discretizada, como mostrado na figura 2.3 , em *slots* temporais e o comportamento de cada sinal é representado logicamente de forma a estabelecer relações booleanas entre os sinais de entrada e os sinais de saída do processo de interface.

As conexões são classificados em três grupos: tipo 1, pinos que devem ser ligados diretamente a fonte de alimentação (Vcc) ou terra (gnd); tipo 2, pinos que podem ser controlados diretamente por outro pino; e tipo 3, conexões que requerem portas lógicas entre pinos de entrada e pinos de saídas. O circuito lógico para conexão do terceiro tipo é construído a partir das relações booleanas presentes na discretização dos sinais.

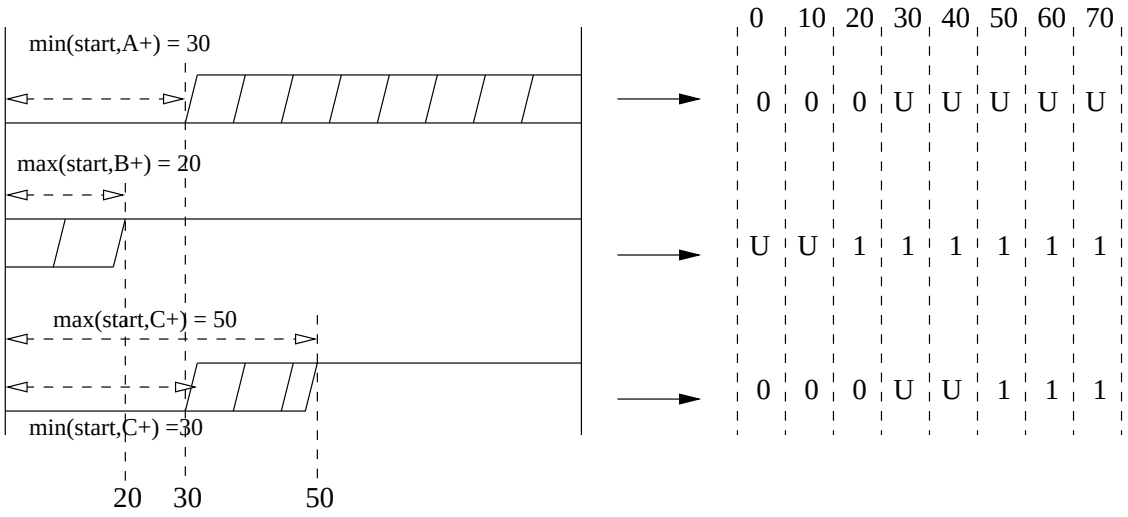


Figura 2.3: Representação discreta de diagramas temporais no SYNTERFACE

Um algoritmo, chamado de SYNTERFACE [20] é proposto para otimizar este processo em termos de área do circuito gerado. Na prática, este sistema lida somente com a compatibilização de sinais ao nível de seu comportamento lógico, ignorando aspectos mais gerais como inconsistências nos sinais de dados e "autonomia" do processo de interface. As relações de causalidade na especificação ficam sub-entendidas na marcação temporal, tornando difícil análises e verificações formais do resultado. Parece, no entanto, ser uma abordagem adequada para a geração de lógica auxiliar entre diversos módulos *off-the-shelf* presentes no mercado, como microcontroladores e memórias padrões.

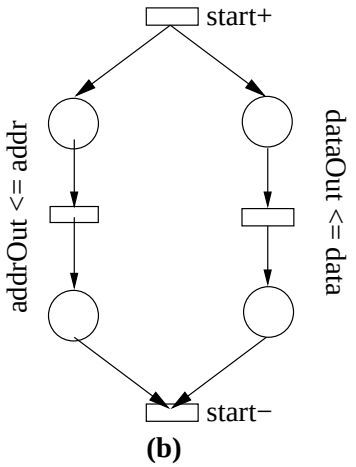
Apesar da contribuição relevante dos trabalhos destacados, a abordagem adotada por Bill Lin e Steven Vercauteren [21] [12] [22] para síntese do processo de interface foi a grande inspiradora deste trabalho. Estes pesquisadores desenvolveram, no IMEC, um compilador de alto nível chamado ***Integral*** [12]para a síntese de módulos de interface. A especificação da comunicação entre os módulos a serem integrados é fornecida em estilo de uma linguagem CSP (Occam, por exemplo) e descreve a transferência de dados a partir de transações realizadas em canais. A cada canal é atribuído um protocolo retirado de uma biblioteca e para o qual existe uma representação em formato intermediário de redes de Petri de alto nível [23]. A rede de Petri é refinada até que represente somente eventos de transição nos sinais, e um circuito assíncrono para a interface é gerado através do compilador *Assassin* [24].

O processo é melhor entendido seguindo o pequeno exemplo na figura 2.4. O programa (figura 2.4(a)) envia um dado através do canal *chDado* concorrentemente ao endereço no canal *chEndereço*. Ambos os canais obedecem ao protocolo *rtz* (um protocolo padrão tipo *return-to-zero,* presente na biblioteca de padrões) especificado. Este programa é transformado em uma rede de Petri e refinado até representar somente transições nos sinais (figuras 2.4(b) e 2.4(c)). O módulo de interface é, então, gerado como um circuito assíncrono baseado em uma biblioteca de portas lógicas especiais.

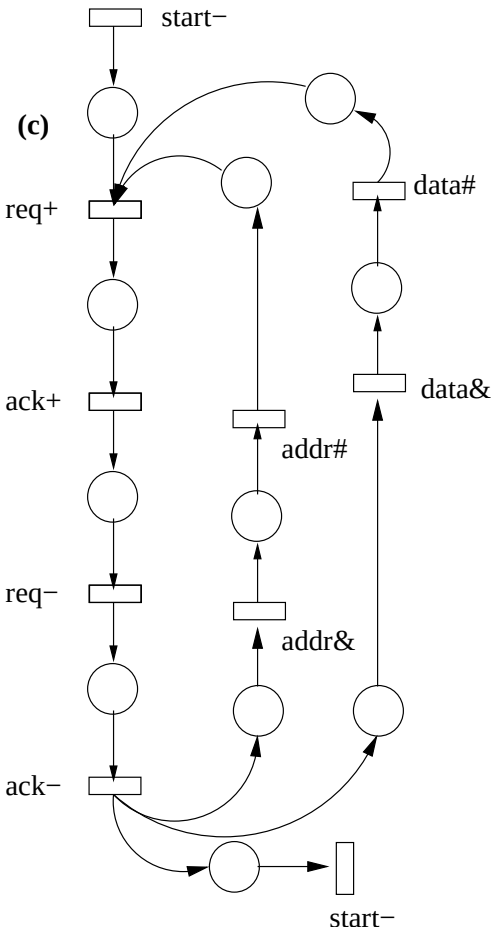
```
protocol rtzRead;
protocol rtzWrite;

process example;
in signal start;
channel addr<7:0>, data<7:0>: rtzRead;
channel addrOut<7:0>, dataOut<7:0> : rtzWrite;
{
   boolean x<7:0>, y<7:0>;
   start+;
   addr?y | data?x;
   addrOut!y| dataOut!x;
   start-;
}
```

(a)



(b)



(c)

Figura 2.4: Esquema de modelagem do *Integral*

Ao transformar a especificação inicial em um formato intermediário de redes de Petri, esta metodologia foi enriquecida com uma ferramenta formal e de fácil entendimento, a qual permite um modelamento racional para os pontos destacados anteriormente como "desafios" na geração do processo de interface:

1. Em primeiro lugar, a representação em Redes de Petri modela concorrência, paralelismo, sincronismo, entre outros aspectos, de uma forma natural. Isto evita que o projetista precise introduzir recursos extras (como sincronismo através dos nomes de sinais) para descrever o protocolo e modelar a forma como deseja o comportamento do processo de interface;

2. Este formato permite e encoraja a simulação, análise, verificação e validação do processo modelado, já possuindo para tal larga base de trabalhos desenvolvidos;

3. A metodologia pode ser gradualmente extendida para incorporar novos aspectos como tratamento dos tipos de dados a serem transmitidos, políticas de arbitragem, entre outros;

Convém reafirmar que o presente trabalho tem muitas idéias oriundas desta metodologia, aproveitando suas qualidades e tentando adaptá-las para a aplicação na rápida integração de *IP-Cores* em ambientes *SoC*.

Neste sentido procuramos:

- Retirar a idéia de biblioteca de padrões de protocolos, provendo ao projetista uma forma de especificar explicitamente o comportamento do protocolo estabelecido no canal;

- Adequar a descrição final do processo de interface a necessidades mais amplas de mapeamento tecnológico e de reusabilidade. Neste sentido substituímos a *netlist* de um circuito (fornecido pelo *Assassin*), por uma descrição em VHDL do processo de interface.

Quando o processo de interface é gerado de forma a integrar módulos especialmente preparados para o sistema, como em algumas metodologias de *hardware/software*

co-design, os problemas enfrentados têm natureza adversa do discutido até então e outras abordagens são assumidas. Detalharemos a seguir.

2.1.2 Geração de Interface em Sistemas de Hw/Sw Co-Design

Na maioria dos sistemas de *Hardware/Software Co-design*, consensualmente, o problema de comunicação entre módulos é modelado como canais entre um produtor e um receptor de dados [16]. Em muitos casos, estes canais compartilham um mesmo conjunto de recursos físicos de forma concorrente e exigem da metodologia de síntese uma percepção da arquitetura destes recursos. Assim, por exemplo, não somente uma especificação do canal e seu protocolo são necessários, mas também informações de se este canal é implementado em hardware e/ou software, se o meio é compartilhado por outros canais e como se dá o controle de acesso ao meio, entre outros aspectos.

Podemos então agrupar a pesquisa de síntese de comunicação para *Hw/Sw Co-design* em dois grandes grupos: aqueles que focam a modelagem e implementação do canal e aqueles que enfatizam a exploração de arquiteturas de comunicação. Estes dois grupos complementam-se em resultados.

Eisenring et. al. [25] [26] introduzem uma abordagem orientada a objeto que modela o sistema a partir de grafos de fluxo de dados. Dada uma partição de processos em hardware e software ele mapeia (e modela) os canais de comunicação segundo a semântica dos arcos deste tipo de grafo, ou seja, uma única fonte de dados, único consumidor, bufferizado com FIFO, leituras bloqueantes e escritas não bloqueantes. O modelo deste canal em hardware é apresentado na figura 2.5. Cada nó de hardware é um processo ou tarefa que consome e/ou produz dados. Estes dados são transmitidos através do circuito de I/O (bem definido na tipologia de dados) sob controle de uma lógica auxiliar que também ativa os processos e monitora o fim de suas execuções. Este sistema é conhecido na literatura como *Hardware/Software Interface Generator* ou HASIS.

Araújo [27] propõe a modelagem do canal com semântica similar a do HASIS, porém de uma forma mais abrangente não há distinção quanto a natureza do canal (ou seja, implementado em hardware ou software). Para tal o canal é dividido em

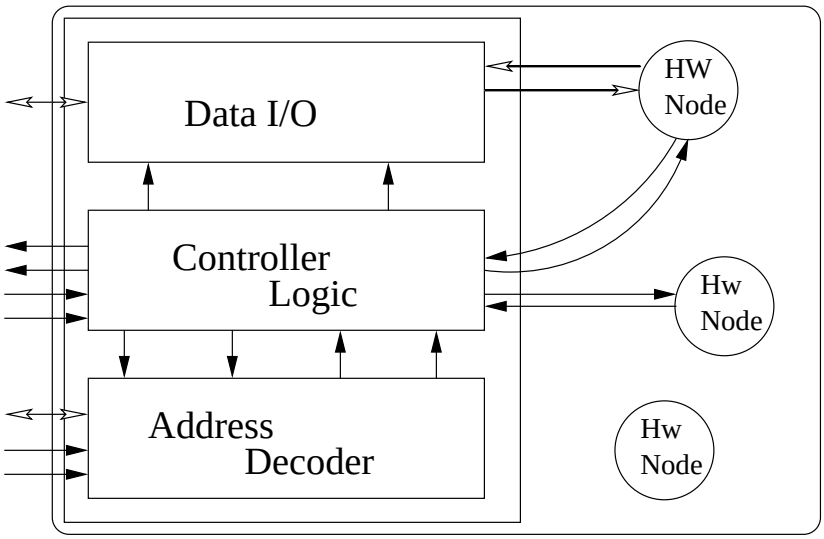


Figura 2.5: Modelo de Canal do HASIS

camadas (figura 2.6) onde a camada mais externa é responsável por implementar o controle de acesso ao meio e um protocolo genérico. Do ponto de vista dos módulos, no entanto, a interface deve ser bem definida, constituída da linha de dados e alguns poucos sinais de ativação e fim de execução. Esta abordagem é totalmente integrada e automatizada no processo de *hardware/software co-design* do PISH [2].

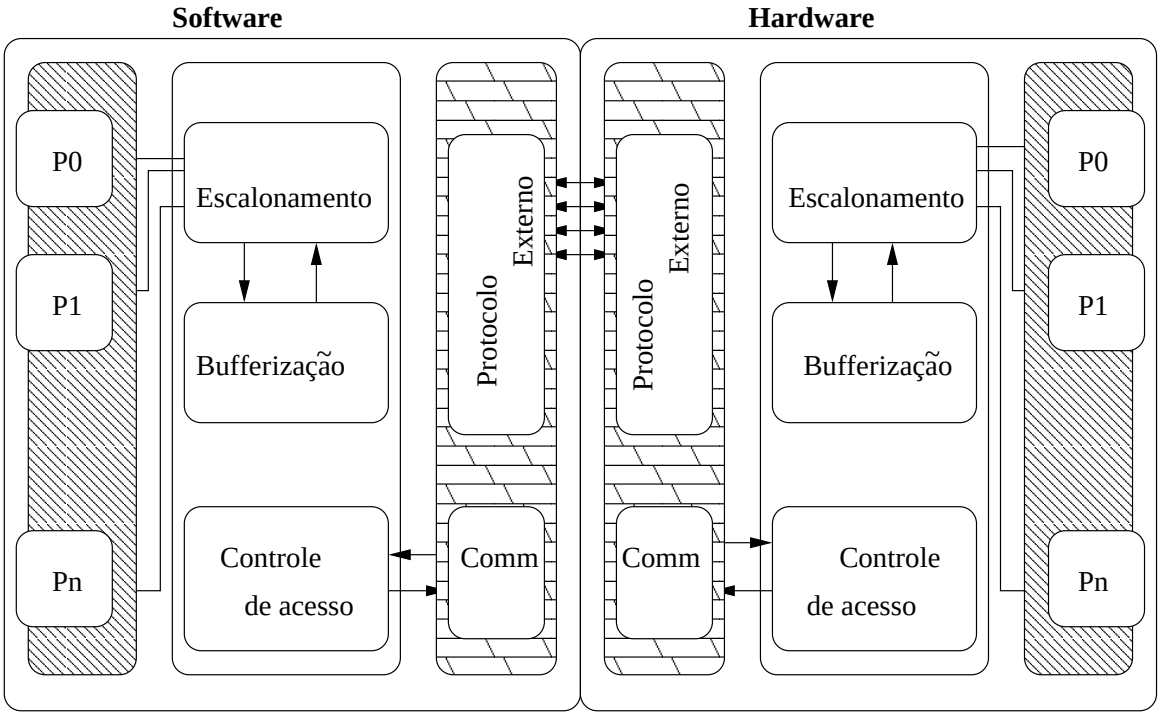


Figura 2.6: Canal *Hardware / Software* PISH

Em uma abordagem bastante simples para o segundo grupo, porém largamente utilizada, Tauro e Vahid [28], propõem que os canais sejam mapeados em elementos da arquitetura (barramentos ou FIFOs, por exemplo) para os quais já existe um conjunto de rotinas em C++ ou circuitos descritos em VHDL que os implementa. Esta biblioteca é conhecida na literatura como *Object-Oriented Communications Library* (OOCL) e permite que o projetista crie e simule o sistema inicial utilizando somente primitivas de comunicação como *send/receive*. Posteriormente estas primitivas são atribuídas e implementadas por portas físicas (ou de software) existentes na biblioteca.

Lahiri et. al. [29] utilizam a mesma abordagem baseada em bibliotecas, mas automatizam o processo de mapeamento das primitivas de comunicação nos respectivos elementos da arquitetura. Desta forma, propõem um algoritmo para explorar a arquitetura de comunicação com maior eficiência (menor custo).

2.2 *System-On-a-Chip* e *Ip-Cores*

A percepção clara da contribuição pretendida por este trabalho não pode ser completamente alcançada sem o entendimento da "revolução" dos *System-On-a-Chip*, ou mais comumente, *System-On-Chip* (SoC). A tecnologia de construção dos dispositivos semicondutores permite, na atualidade, um grau de integração tão elevado que sistemas inteiros podem ser construídos em uma só pastilha de silício. Daí o termo "sistemas em um chip", entretanto, este potencial não pode ser explorado com as mesmas técnicas de projetos utilizados para construir os sistemas digitais em placas de circuitos baseados em componentes *off-the-shelf*.

System On Chip, mais que uma classe de dispositivos, é um paradigma de projeto de hardware e consiste na integração de blocos funcionais de maior complexidade, configuráveis ou não, em um só ambiente de integração. É peculiar a este estilo o fato da maioria dos componentes adotarem padrões desenhados por terceiros, e cuja confiabilidade precisa ser garantida entre grupos distintos, não somente no interior de uma empresa ou grupo, mas entre as mais distantes (geográfica e culturalmente) equipes de projeto. Desta visão, nasce a atual necessidade de reuso de componentes

para compatibilizar a velocidade de projeto com as necessidades do mercado [30].

Chang et. al. defende que as metodologias de projeto atuais caminham para um projeto baseado em plataformas, conforme ilustrado na figura 7, saindo das tradicionais abordagens *timing driven design* e passando pelo projeto baseado em blocos pré-caracterizados. Estes segmentos variam de acordo com as tecnologias de suporte, o tamanho do projeto e o nível de reuso.

As abordagens tradicionais para projeto de *ASICs* de tamanho moderado (5000 - 250000 portas) é comumente chamada de *timing driven design.* Normalmente, os projetos são feitos a partir do zero, portanto sem nenhum reuso, e envolve poucas pessoas com um conhecimento razoavelmente homogêneo. As tecnologias observadas neste estilo são ferramentas para *floor-planning* e análises temporais. Área não é necessariamente um problema, transformando performance e potência em objetivos para a otimização.

Quando o enfoque do projeto toma uma direção voltada a aplicações e subsistemas[1] e várias equipes de projeto são envolvidas, surge a necessidade de basear o projeto em blocos pré-caracterizados de *hardware* e *software. Block based design* é, idealmente, modelado de forma comportamental a nível de sistema. Dá-se atenção a arquitetura do sistema e a comunicação entre os módulos. Tipicamente, muito do reuso é oportunístico, pois os módulos utilizados requerem re-verificação e, frequentemente, modificações. Este é um estilo mais rápido de projeto e capaz de lidar com maior complexidade (150.000 a 1.500.000 portas). O suporte tecnológico vem, primordialmente de ferramentas de análise e síntese de alto nível.

A necessidade, latente no mercado, por projetos ainda mais complexos (maiores que 350.000 portas) e em tempo recorde faz com que o projeto baseado em blocos evolua, adquirindo um caráter de reusabilidade extensiva, pré-imaginada e uma forte hierarquia. Denominados de *Plataform-Based Design,* estes sistemas são tipicamente baseados em microprocessadores que comunicam-se com módulos ligados a barramento. A necessidade de re-verificação e validação dos módulos é diminuída pelo planejamento prévio dos mesmos. A tecnologia de apoio a este estilo nascente

[1]Processamento embarcado, compressão de dados, correção de erros, entre outros.

são ferramentas de alto nível, ferramentas de *layout* focadas em integração de blocos e sistemas de verificação do módulos.

A tabela 2.2 ajuda a entender as diferenças entre estas abordagens:

Características do Projeto	*Timing Driven Design*	*Block Based Design*	*Plataform Based Design*
Complexidade	5000~250k portas	150k~15M portas	>350k portas
Nível do Projeto	RTL	Comportamental / RTL	Arquitetura
Equipe	Pequena e Focalizada	Multidisciplinar	Multi-grupo Multidisciplinar
Projeto primário	Lógica customizável	Blocos em contexto, interfaces customizáveis	Microprocessador Periféricos
Reuso	Nenhum	Oportunístico	Planejado
Arquitetura de Barramento	Nenhum	Customizado	Padronizado
Co-verificação *Hw/Sw*	Nenhuma	Funcionalidade e Interfaces	Interface *Hw/Sw*
Foco do particionamento	Limitado pela síntese	Funções	Funções / Comunicações

Tabela 2.2: Resumo comparativo para os estilos de projeto. (Fonte: *Surviving the SoC Revolution* [1])

Fortes grupos da indústria de eletrônica como o ***Virtual Socket Interface Alliance*** [9] e ***Open Core [31]*** caminham em concordância com estas idéias. É notavel, então o papel desempenhado por estes módulos pré-caracterizados de *hardware* e *software*, comumente conhecidos como *core* (núcleo).

Um *core* é definido como um bloco de projeto, maior que componentes RTL e que realizam alguma funcionalidade mais sofisticada ou elaborada[2] [32]. Esta sofisticação é revestida, frequentemente, de um conhecimento profundo do algoritmo implementado pelo circuito, bem como o *know-how* para sua validação, verificação e integração em outros sistemas. Ou seja, o projeto de um *core* é revestido de uma "elegância" baseada em conhecimento que pode caracterizar uma "propriedade intelectual". *Intelectual Property Cores* (*Ip-Cores*) são considerados os tijolos na construção das metodologias com reuso.

[2]Desta forma, um somador não pode ser considerado um *core*, mas um ULA com muitas operações sim.

O estudo por modelos de especificação, implementação, incorporação e comercialização de *Ip-Cores* são extensivamente explorados na literatura atual [32] [30] [33] [6] [34]. Bergamaschi [35] argumenta que a realidade de rapidamente montar um sistema a partir de *Ip-Cores* ainda não é uma realidade por várias razões, a maioria delas ligadas à fase de incorporação:

- A busca por um arquitetura ideal para um sistema não é uma tarefa trivial. O impacto do custo de comunicação é fundamental nesta busca [29];
- A integração de *cores* ao sistema é basicamente uma atividade manual e susceptível a erros, pois requer que o projetista lide com centenas de pinos, interconexões, protocolos de interface e características elétricas. Apesar da importância desta fase na performance do sistema[3], lidar com o circuito de integração não adiciona funcionalidades ao circuito, sendo gasto um tempo precioso sem acréscimo do valor agregado do produto;
- Verificação a nível de sistema é difícil;
- Faltam padrões estabelecidos na indústria [36] e/ou faltam ferramentas para síntese de interface eficientes, dificultando a integração de *Ip-Cores* de diferentes provedores;

A fase de integração destes módulos ao projeto é o foco direcionador do presente trabalho e o resultado final mostra que é possível construir metodologias e ferramentas eficazes na geração do processo de interface.

Por fim, é necessário visitar, ainda que rapidamente, as mais modernas abordagens para a síntese de circuitos assíncronos. Esta visão será útil no entendimento do código VHDL gerado como produto da metodologia proposta, entendendo suas potencialidades e limitações.

[3] A performance é diretamente afetada pelo custo de comunicação e a confiabilidade do circuito depende da ligação correta de seus componentes.

2.3 Síntese de Circuitos Assíncronos

O projeto clássico de circuitos assíncronos segue as bases lançadas por Huffman, nas conhecidas máquinas de Huffman. Nesta classe de circuitos, estruturados como uma máquina de estados, não são utilizados latches ou flip-flops subordinados a um sinal de *clock*. O estado é armazenado, na malha de realimentação do sistema. Tipicamente, exige-se que seja assumida uma suposição de que o circuito funciona em modo fundamental, ou seja, uma vez que ocorrem mudanças na entrada, não podem haver novas mudanças até o o sistema esteja estável. Sob estas condições, Huffman, Unger e McCluskey desenvolveram muito da teoria de circuitos assíncronos.

Desde que estas teorias foram lançadas, a área de pesquisa em circuitos assíncronos sofreu diversos altos e baixos, à medida que surgiam dificuldades tecnológicas à implementação destes circuitos que eram mais facilmente resolvidos por similares síncronos. As maiores barreiras em relação à síntese e implementação de circuitos assíncronos são:

- *Hazards*
- Mapeamento tecnológico
- Sistemas com alto grau de concorrência, dificultando o projeto.

Hazards, ou o potencial para o aparecimento de *glitches*, são uma importante consideração em qualquer projeto de sistema assíncrono. Em sistemas síncronos, o sinal de *clock* é utilizado para filtrar o efeito dos *glitches*, marcando adequadamente os momentos nos quais os sinais podem ser "lidos". Em sistemas assíncronos, por sua vez, estas instabilidades temporárias nos sinais (*glitches*) podem ser imterpretados como mudanças válidas nos sinais e causar um malfuncionamento do sistema. Diversos métodos para eliminação de *hazards* combinacionais, bem como problemas de *race*, foram propostas ao longo do tempo.

As diversas técnicas de eliminação de *hazards* baseiam-se em condições restritivas que incidem sobre a forma como o sinal propaga-se nos componentes lógicos e vias de interconexão. Estas condições acabam por dividir as técnicas segundo modelos de propagação do sinal, o que reflete diretamente nas características tecnológicas dos

dispositivos. De uma forma simples, circuitos projetados utilizando-se um determinado modelo de propagação deve ser implementado em uma plataforma tecnológica que respeite as condições impostas por este modelo.

Adicionalmente, as técnicas baseadas em máquinas de estado não são suficientemente adequadas apra lidar com o alto grau de concorrência exibido por sistemas assíncronos, gerando uma deficiência de ferramentas de CAD e algoritmos. Porém, de meados dos anos 80 até os dias atuais, diversas metodologias para síntese de circuitos assíncronos foram propostos, baseados em três categorias principais: máquinas de estados, métodos baseados em grafos e redes de Petri e métodos de tradução.

2.3.1 Máquinas de Estado Assíncronas

Muitos trabalhos recentes são centrados no projeto de máquinas de estados assíncronas de "modo *burst*", ou *"burst-mode"* [37]. Especificações *burst-mode* originaram-se informalmente dos trabalhos de Davis [38]. Davis propôs a idéia de máquinas de estados que esperariam por conjunto de mundanças na sua entrada (*input burst*) e então, responderia com um conjunto de mudanças na saída (*output burst*). A contribuição chave é que , diferentemente da abordagem tradicional, as mudanças na entrada poderiam ser descorrelacionadas, ou seja, acontecerem em qualquer ordem, portanto, estas máquinas podem operar com maior grau de concorrência. Porém, somente em trabalhos mais recentes [37] foram propostos métodos para a síntese destes circuitos sem a existência de *hazards*. Para tal, foi proposto um método de implementação chamado *3D* [39].

Adicionalmente, foi proposto por Yun e Dill uma extensão no formalismo de especificação que ficou conhecido como *extended burst-mode.* Esta nova abordagem permitiu um aumento no grau de concorrência e melhorou os aspectos práticos de implementação. O *Extended Burst-Mode* pode ser utilizado para sintetizar controladores e máquinas de estado para interface entre sistemas síncronos e assíncronos, ond e o *clock* global pode ser visto como uma das entradas do controlador.

2.3.2 Petri Nets e Métodos baseados em grafos

Redes de Petri e grafos de estado também são largamente utilizados para especificação, projeto e síntese de circuitos assíncronos. Diversos métodos de síntese utilizam redes de Petri restritas, baseadas em grafos marcados, que modelam bem aspectos de concorrência, mas impõem restrições à escolhas entre sinais.

Um número significativo de algoritmos de síntese, análise e mapeamento tecnológico foram desenvolvidos para estes sistemas [13] [40] [41] [22], visando a construção de circuitos livres de *hazard* e com um mínimo de estados. Ferramentas de CAD completas estão disponíveis para a aplicação destes métodos, como por exemplo, o pacote *SIS* (Berkeley) e o *Petrify*.

As metodologias abordadas nesta dissertação são baseadas nestes métodos, em especial nos trabalhos ligados à ferramenta *Petrify [42].*

2.3.3 Métodos de Tradução

Métodos de tradução especificam um sistema assíncrono através de uma linguagem concorrente de programação de alto-nivel. Exemplos comuns incluem variantes de *CSP e Occam.* O programa e iterativamente transformado até um programa em baixo nível, que pode ser mapeado diretamente em um circuito. Estes métodos podem ser utilizados, com bons resultados, para a síntese de estruturas de controle ou dados.

A mais conhecidas destas linguagens é a *Tangram*, utilizada por alguns grandes fabricantes como Philips. Esta linguagem foi desenvolvida por van Berkel et. al. e detêm um ótimo suporte atual em termos de ferramentas de CAD.

2.4 Resumo

Neste capítulo foram apresentados os mais recentes avanços de pesquisa nas três áreas principais que oferecem suporte a este trabalho: a geração e síntese de processos de interface, projeto de sistemas *on-chip* e *IP-Cores* e projeto de sistemas assíncronos.

Um breve histórico recente das metodologias de geração do processo de interface situa este trabalho entre as mais recentes iniciativas neste sentido e assinala as linhas principais desta pesquisa. O projeto de sistemas *on-chip* e de módulos de hardware foi enfatizado, mostrando seu crescimento promissor e intimamente ligado com as mais modernas metodologias de projeto que têm como base o reuso de módulos de hardware. Por fim, uma rápida introdução dos conceitos estudados na síntese de sistemas assíncronos.

Capítulo 3

Redes de Petri: Propriedades, análise e aplicações na geração de interface.

Antes de avançar no detalhamento da técnica de geração de interface, é importante visitar algumas noções do formalismo, introduzido em 1962 por Carl Adam Petri [43] e, largamente conhecido como Redes de Petri. Todos os procedimentos deste trabalho estão diretamente ligados às redes de Petri, e sendo esta uma vasta área de pesquisa, detalharemos neste capítulo somente as noções necessárias para o completo entendimento desta aplicação.

Redes de Petri é uma ferramenta de modelagem baseada em um formalismo matemático e com uma interpretação gráfica extremamente amigável. È uma ferramenta promissora para descrever e estudar sistemas de processamento caracterizados como concorrentes, assíncronos, distribuídos, paralelos, não-determinísticos e/ou estocásticos [44]. Seu uso para uso em ferramentas e metodologias de projeto de hardware torna-se particularmente interessante pois estas aplicações requerem um formalismo rigoroso, mas também de fácil compreensão pelo projetista.

Tradicionalmente, por exemplo, um dos modelos mais utilizados para modelamento de hardware são as máquinas de estado finito (MEF). A principal característica das MEF é que o sistema é definido de uma forma seqüencial. Os melhores esforços para representar aspectos como concorrência e paralelismo, neste tipo de abordagem, levam a *interleavings* e composições de várias máquinas de estado, incorrendo, frequentemente, em problemas de explosão de estados. As redes de Petri,

em contra-partida, oferecem uma abordagem natural à representação dos conceitos de paralelismo e concorrência devido a sua semântica. Adicionalmente, as metodologias para análise, verificação e validação de modelos baseados em rede de Petri são largamente estudadas e estabelecidas na literatura, e em especial interesse deste trabalho, diversas abordagens enfocam a modelagem de sistemas de hardware [13].

No presente trabalho, as redes de Petri desempenham o papel de linguagem intermediária para a representação da especificação dos protocolo e do modelo do processo de interface, sendo palco de todas as operações de análise, verificações, transformações e síntese. Como ilustrado na figura 3.1, a especificação dos protocolos iniciais[1] pode partir de diferentes bases (CSP, SystemC, diagramas temporais, máquinas de estado), devendo ser apropriadamente convertida para o modelo de rede de Petri deste trabalho. Da mesma forma, diversos formatos para a saída do sistema podem ser explorados. Os formatos de entrada e saída escolhidos para serem implementados são marcados em linha cheia.

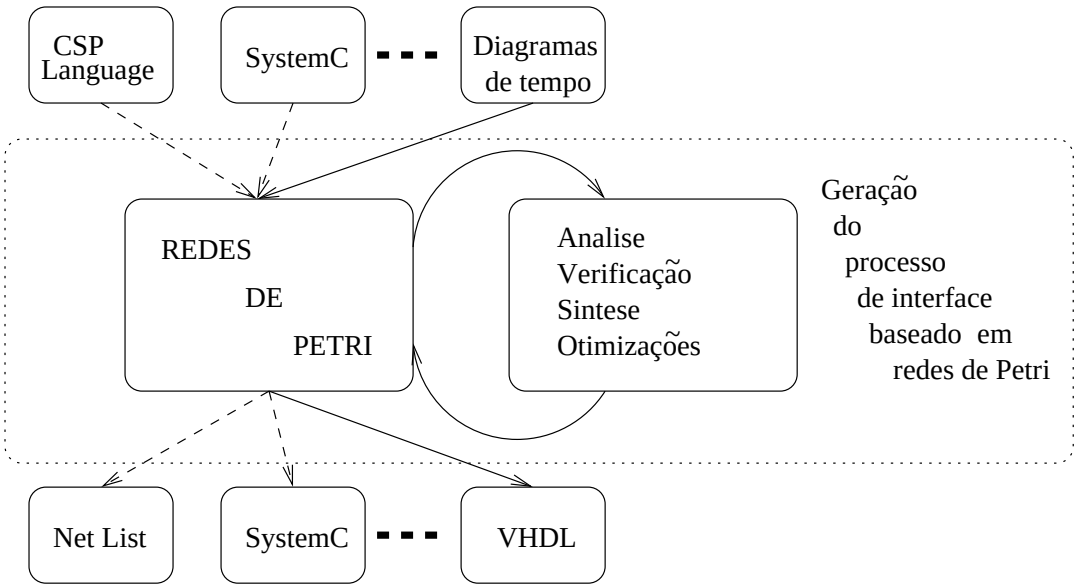


Figura 3.1: Redes de Petri como linguagem intermediária

[1] A especificação dos protocolos individuais dos módulos a serem integrados constituem o ponto de partida desta metodologia, conforme será explicado futuramente.

3.1 Redes de Petri

Definition 3.1.1 Uma **rede de Petri [44]** é uma tupla $N = \langle P, T, F, Mo \rangle$, onde $P = \{p_1, p_2, \cdots, p_m\}$ é um conjunto finito de lugares, $T = \{t_1, t_2, \cdots, t_n\}$ é um conjunto finito de transições, $F \subseteq (P \times T) \cup (T \times P)$ é um conjunto de arcos que marca a relação de fluxo e $M_o : P \rightarrow 0, 1, 2, \cdots$ é a marcação inicial da rede, $P \cap T = \varnothing$ and $P \cup T \neq \varnothing$.

Sistemas podem ser modelados descrevendo-se seus estados e as mudanças entre estes. Nas redes de Petri isto é feito atribuindo-se cada estado a uma marcação $M : P \rightarrow 0, 1, 2, \cdots$e seguindo uma regra comportamental bem estabelecida (regra de disparo) para a transição entre os estados:

1. Uma transição t é dita **habilitada**, em uma marcação M, denotada $M[t >$, se cada lugar de entrada p possui ao menos uma marca, ou *token;*

2. Uma transição habilitada pode ou não "disparar", dependendo se aquela transição realmente ocorre no sistema real;

3. O disparo de uma transição habilitada retira uma marca de cada lugar de entrada e coloca um *token* em cada lugar de saída.

Alguns modelos de redes de Petri atribuem pesos aos arcos, o que modifica levemente as regras de disparo. Neste trabalho, no entanto, consideramos somente as redes ditas **ordinárias**, ou seja, redes em que o peso de qualquer arco é unitário.

Uma interpretação gráfica bastante intuitiva pode ser montada para esta família de modelos. A figura 3.2(a) mostra uma rede de Petri simples onde a transição t_1 encontra-se habilitada, pois seus lugares de entrada p_1 e p_2 contém ao menos uma marca cada. Ao disparo de t_1, a nova marcação será como indicada em 3.2(b).

Uma marcação M é dita **alcançável**, a partir de M', se existe ao menos uma sequência realizável $\sigma = t_1 t_2 \cdots t_k$ que transforme M' em M, ao que denotamos $M'[\sigma > M$. Denominamos $[Mo >$ ao conjunto de todas as marcações alcançáveis a partir de Mo. Adicionalmente, dado um nó x qualquer da rede, onde $x \in P \cup T$, o conjunto de nós de entradas é denotado $\bullet x$ e o conjunto de nós de saída como

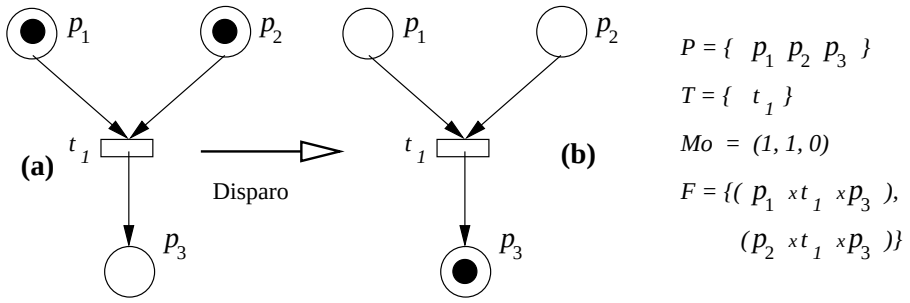


Figura 3.2: Rede de Petri

$x\bullet$. Desta forma, por exemplo, a transição t_1, na figura 3.2, tem como pré-conjunto $\bullet t_1 = \{p_1, p_2\}$ e pós-conjunto $t\bullet = \{p_3\}$.

Basicamente, existem dois tipos de propriedades a serem estudadas em modelos de rede de Petri: aquelas que dependem da marcação inicial da rede e estão intimamente ligados com o comportamente dinâmico do modelo; e aquelas independentes da marcação inicial, analisáveis a partir da estrutura da rede. Analisaremos apenas algumas propriedades comportamentais, pela sua importância às idéias abordadas no trabalho.

3.1.1 Propriedades Comportamentais das Redes de Petri

As propriedades comportamentais de uma Rede de Petri exprimem o comportamento dinâmico do sistema modelado, através dos possíveis estados alcançáveis e o jogo semântico das transições entre estes estados [45]. Daí então argumentar-se que a base fundamental do estudo destas propriedades dinâmicas consiste em uma noção de alcançabilidade. A análise de alcançabilidade em redes de Petri é o problema de determinar se uma determinada marcação M_n pertence ao conjunto de marcações alcançáveis $[Mo>$. Neste contexto, algumas propriedades adquirem caráter prático interessante na modelagem de sistemas de hardware, tais como:

Definition 3.1.2 Uma rede de Petri N é dita **limitada**, *k-bounded*, ou simplesmente *bounded* [44], se o número de *tokens* em cada lugar não excede um número finito k para qualquer marcação $M \in [Mo>$.

A figura 3.3 ilustra este conceito. A rede não é limitada pois não existe um limite finito para o número de *tokens* que o lugar p_3 acumula. Sistemas de hardware são

comumente modelados de forma que os lugares representam *buffers,* registradores ou recursos do sistema. A análise de *boundedness* pode, por exemplo, indicar possíveis sobrecarga no uso destes recursos.

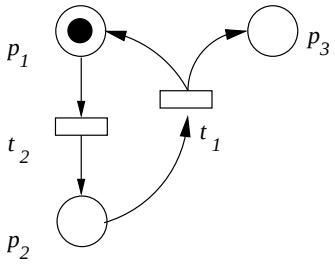


Figura 3.3: Exemplo de rede não limitada

Definition 3.1.3 Uma rede de Petri N é dita **segura, 1-*safe*** ou simplesmente ***safe*** [44] se ela é 1-*bounded.*

A figura 3.4(a) é um exemplo de rede *safe*, note que em todas as marcações alcançáveis da rede[2] nenhum lugar possui mais que um *token.* Em contrapartida, a mesma rede, com marcação inicial levemente diferente (figura 3.4(b)), pode acumular até 2 *tokens* em cada lugar, não sendo portanto *safe.* Note ainda, que a propriedade de *safeness* garante, por definição, que a rede é *bounded.*

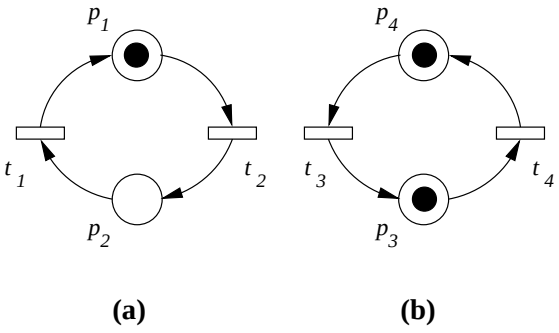


Figura 3.4: *Safeness* depende da marcação inicial

Definition 3.1.4 Uma rede de Petri é dita ***live [44]*** se para qualquer marcação $M \in [Mo >$, qualquer transição t poderá ser habilitada através de uma sequência apropriada de disparos.

[2] A propriedade de *safeness* depende da marcação inicial.

O conceito de *liveness* está intimamente ligado à ausência de travamentos (*deadlocks*) no sistema. Se uma rede não é *live* significa que existe uma marcação M que "trava" o sistema, ou seja, novas mudanças de estado não podem ocorrer pois nenhuma transição está apta a disparar. Por exemplo, a rede na figura 3.5(a) não é *live*, pois ao disparo de $\sigma = t_3 t_4$ nenhuma transição estará habilitada, ao contrário da rede em 3.5(b). A propriedade de *liveness* é particularmente interessante em especificações de protocolo e modelos de interface baseados em redes de Petri, pois garante que estes serão livres de travamento, defeito considerado grave para um circuito de interface.

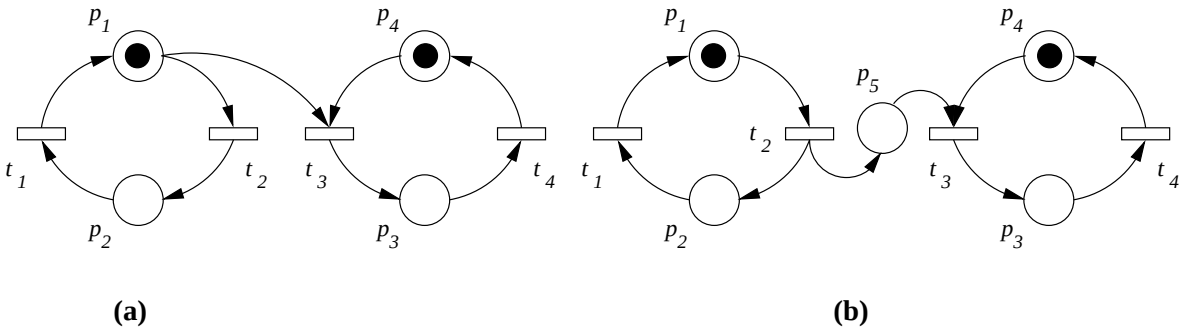


Figura 3.5: *Liveness*

Definition 3.1.5 Uma rede de Petri N é dita **reversível [44]** se, e somente se, para toda marcação $M \in [Mo>$, Mo é alcançável a partir de M.

De maneira simples, a reversibilidade garante que existe ao menos uma sequência σ de transições, que ao ser disparada leva o sistema a seu estado inicial, qualquer que seja o estado atual.

Para finalizar o conjunto de propriedades básicas das redes de Petri que serão exploradas ao longo deste trabalho, é necessário definir o conceito de persistência de uma rede. A persistência garante que se uma transição vem a tornar-se habilitada, ela permanecerá assim até que seja disparada. A figura 3.6 apresenta duas situações de **conflito**, onde o disparo de uma transição desabilita a outra. Note que estas redes não são persistentes, pois uma transição pode tornar-se habilitada e, em seguida, desabilitada sem que tenha disparado.

Definition 3.1.6 Uma rede de Petri N é dita **persistente [44]** se, e somente se, para quaisquer duas transições $t_1, t_2 \in T$, o disparo de uma não desabilitará a outra.

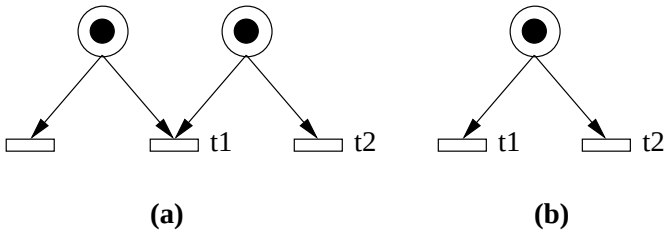


Figura 3.6: **(a)** Conflito, **(b)** Escolha

Como forma de facilitar o estudo dos algoritmos de análise e manipulação das redes de Petri, costuma-se classificá-las em subclasses através da imposição de restrições a sua estrutura [46]. Neste aspecto podemos agrupar as redes em 5 classes[3] (figura 3.7) :

State Machine é uma rede de Petri onde cada transição t tem exatamente um lugar de entrada e exatamente um lugar de saída, ou seja, $|\bullet t| = |t\bullet| = 1, t \in T$;

Marked Graph é uma rede onde cada lugar p tem exatamente uma transição de entrada e exatamente uma transição de saída, ou seja, $|\bullet p| = |p\bullet| = 1, p \in P$;

Free Choice é uma rede onde cada arco de um lugar é o único arco de saída ou o único arco de entrada de uma transição, ou seja, para qualquer $p \in P, |p\bullet| \leq 1 \vee \bullet(p\bullet) = \{p\}$;

Extended Free Choice é uma rede tal que $p_1\bullet \cap p_2\bullet \neq \emptyset \Rightarrow p_1\bullet = p_2\bullet$, para todo $p_1, p_2 \in P$;

Asymmetric Choice é uma rede (também conhecida como *simple net*) tal que $p_1\bullet \cap p_2\bullet \neq \emptyset \Rightarrow p_1\bullet \subseteq p_2\bullet \vee p_1\bullet \supseteq p_2\bullet$, para todo $p_1, p_2 \in P$;

As restrições impostas a cada uma destas classes limita seu poder de modelamento, entretanto, facilita o estudo de métodos para análise e verificação dos modelos

[3]Considera-se, nesta classificação, que todas as redes são ordinárias, ou seja, o peso de seus arcos é unitário.

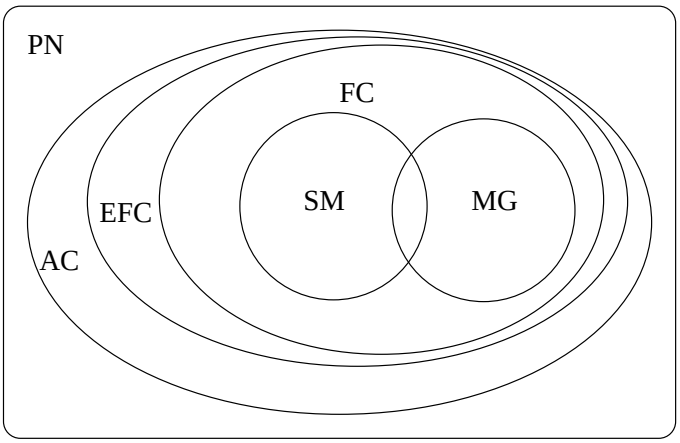
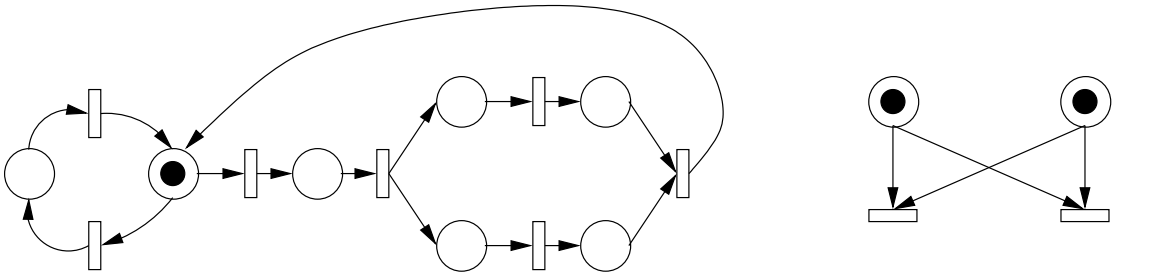


Figura 3.7: Sub-classes de redes de Petri. (Fonte [44])

pois estabelece limites à complexidade dos algoritmos de manipulação da rede. Assim, por exemplo, a classe das *state machine* não pode representar conceitos como paralelismo e sincronismo e a classe das *marked graph* não é capaz de modelar naturalmente escolhas, entretanto, a verificação das propriedades de *liveness* e *safeness* é bastante simples, com complexidade linear.

Alguns trabalhos enfocam a especificação de protocolos e síntese de circuitos a partir de redes *marked graph* [47] [21] [48], no entanto, seu poder expressivo ainda está aquém das necessidades verificadas na geração do processo de interface. Por outro lado, adotar uma abordagem sem restrições às redes utilizadas para modelar protocolos de entrada e o próprio processo de interface poderia incorrer em uma extrema dificuldade (quando não inviabilidade) da verificação de certas propriedades e manipulações. Por este motivo, adotamos a partir de agora os modelos da classe *free-choice*. Esta classe de redes pode expressar escolhas, paralelismo e sincronismo de eventos, no entanto, deve ser livre de conflitos, ou seja, a escolha atribuída a um lugar p deve ser independente da quantidade de marcas em qualquer outro lugar p'. A figura 3.8 ilustra estes conceitos. Redes de Petri *Free Choice* são largamente estudadas na literatura por seu poder de modelagem e por facilitar algoritmos de análise [49]. Checar *liveness, safeness* e persistência, por exemplo, pode ser feito com algoritmos de complexidade polinomial. A aplicação de transformações que preservem estas propriedades (como as utilizadas na síntese do processo de interface) é simples e direta, na maioria dos casos.



(a) Free−Choice Petri Net (b) Conflito (escolha condicionada)

Figura 3.8: Free Choice Petri Nets e conflito não modelável

Os métodos para análise em Rede de Petri podem ser classificados em três grupos: equações algébricas de matrizes, técnicas de redução-decomposição e árvore (grafo) de cobertura (alcançabilidade). As técnicas algébrico-lineares são baseadas na observação de que a ocorrência do disparo de uma transição sempre causa uma mesma mudança relativa no número de *tokens* em cada lugar [45]. Esta mudança relativa pode ser descrita como sistemas de equações dinâmicas. As técnicas de redução-decomposição utilizam transformações na rede, de forma a reduzir sua complexidade (eliminando redundância, por exemplo) sem alterar a propriedade a ser analisada/verificada. Apesar destas técnicas constituírem fortes ferramentas para análise, enfocaremos somente o terceiro método, o da construção da árvore de alcançabilidade.

Dada uma rede N com marcação inicial Mo é possível obter novas marcações através do disparo das transições habilitadas. Destas marcações outras novas podem ser obtidas, sucessivamente. Esta "execução" da dinâmica da rede pode ser representada por uma árvore, onde os nós são as marcações e os arcos saindo de um nó são as possíveis transições que podem ser disparadas naquela marcação. Esta representação pode crescer infinitamente para redes não-limitadas, então convém introduzir um símbolo especial w, que pode ser entendido como "infinito". w é tal que para qualquer inteiro n, $w > n, w \pm n = w$ e $w \geq w$.

A árvore de cobertura de uma rede de Petri pode então ser construída segundo o algoritmo 1 [44]:

Um exemplo da construção da árvore de cobertura está ilustrudo na figura 3.9. Para a marcação inicial indicada, t_1 e t_3 estão habilitadas. O disparo de t_1 leva a um *dead-*

Algorithm 1 Construção da árvore de cobertura

1) Nomeie a marcação Mo como *raiz* e marque-a como *nova*;
2) Enquanto existirem marcações *novas* faça:
2.1) Selecione uma marcação *nova* M;
2.2) Se M é idêntica a alguma marcação no caminho da *raiz* até M, então marque M como *velha* e vá para a nova marcação.
2.3) Se não existem transições habilitadas em M, marque-a como *morta*;
2.4) Enquanto existir transições habilitadas em M, faça, para cada transição habilitada:
2.4.1) Obtenha a marcação M' que resulta do disparo de t em M;
2.4.1) No caminho da *raiz* até M, se existir uma marcação M'', tal que $M'(p) \geq$ $M''(p)$ para todo lugar p, então reponha $M'(p)$ por w todos os lugares p tal que $M'(p) > M''(p)$;
2.4.3) Introduza M' como um nó, desenhe um arco denominado t de M a M', e marque M" como *nova*;

lock ($M_1 = (001)$) onde não existem mais transições livres. Disparando t_3, uma nova marcação $M_2 = (110)$ é alcançada, de forma que $M_2 \geq M_0$. M_2 é então substituída por $(1w0)$. Em M_2, as transições t_1 e t_3 continuam habilitadas, de forma que ao disparar t_3 a mesma marcação é obtida $M_5 = (1w0)$ e deve ser marcada como *velha*. Assim, sucessivamente, o disparo de t_3 leva a uma nova marcação $M_3 = (0w1)$, onde apenas a transição t_2 está habilitada. Disparando-a chegamos à última marcação, $M_4 = (0w1)$, que agora torna-se *velha*.

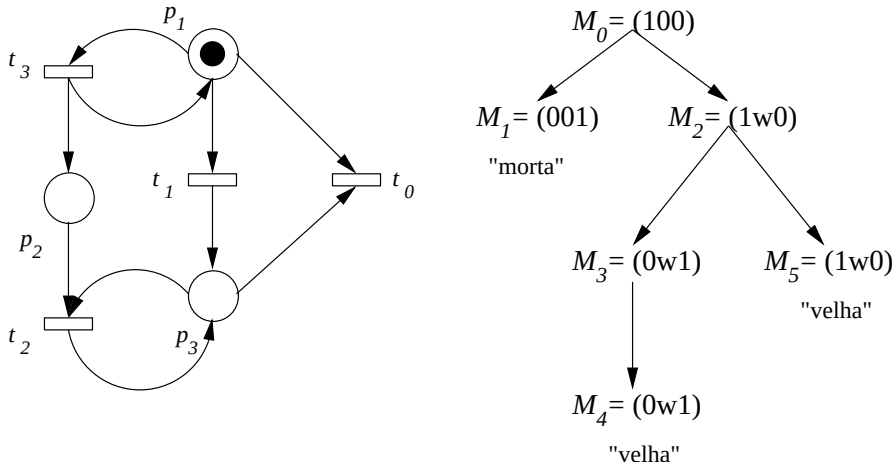


Figura 3.9: Árvore de cobertura

A construção da árvore de cobertura pode ser um trabalho "força bruta" em muitos casos, no entanto, uma vez construída, todas as propriedades discutidas

anteriormente podem ser analisadas e/ou verificadas com facilidade. Assim, por exemplo [49]:

1. A rede será *bounded* se e somente se, nenhum nó da árvore de cobertura tiver um w na marcação;

2. A rede será *safe* se e somente se as marcações de todos os nós tiverem apenas uns e zeros;

3. Se a rede for *safe* e *bounded*, então uma data transição t aparecer em ao menos um arco e, além disso, se a rede for reversível, então ela é *live*;

4. A rede será persistente se e somente se para todo nó, o disparo de uma transição não desabilita as outras transições habilitadas;

Além disso, a árvore de cobertura, ou uma variação de sua estrutura, conhecida como grafo de cobertura ou alcançabilidade, será a base de muitas operações ao longo da síntese do código VHDL do processo de interface. Para redes *bounded*, a árvore de cobertura é também chamada de árvore de alcançabilidade, pois seus nós formam o conjunto das marcações alcançáveis a partir da marcação inicial.

O grafo de cobertura (e mais especificamente para redes *bounded*, grafo de alcançabilidade) é um grafo direcionado $G = (V, E)$, onde V é o conjunto de todos os nós com marcações distintas da árvore de cobertura, e E é o conjunto de arcos associados às transições $t \in T$, que indicam as possíveis transições entre os nós em V. A figura 3.10 mostra o grafo de cobertura da rede exemplificada em 3.9.

Uma vez introduzidos os conceitos básicos de redes de Petri necessários ao entendimento deste trabalho é preciso formalizar o modelo de rede de Petri utilizado. Este modelo é um caso especial de redes de Petri anotadas, largamente conhecido pelo nome de *Signal Transition Graphs*(STG).

3.2 Signal Transition Graphs

Signal Transition Graphs (*STG*) foram introduzidos por T. A. Chu [50], em meados dos anos 80, como um caso especial dos modelos baseados em redes de Petri anotada.

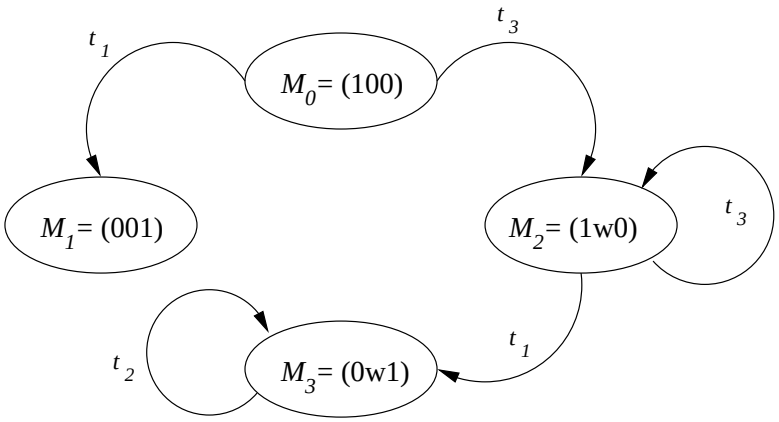


Figura 3.10: Grafo de cobertura da rede na figura 3.9

Nas redes de Petri anotadas as transições e/ou lugares recebem *labels* que expressam ou capturam algum aspecto do objeto modelado. No caso de *signal transition graphs*, as transições recebem *labels* que representam eventos ocorridos em sinais físicos, como por exemplo uma transição de subida $0 \rightarrow 1$ ou a colocação de um dado válido em um barramento.

Apesar de *STGs* representarem apenas um pequeno grupo de fenômenos que podem ser modelados com redes de Petri anotadas, esta simplicidade diz respeito somente ao alfabeto de *labels* atribuídos às transições. Nenhuma restrição maior é imposta, inicialmente, à estrutura da rede de Petri em si. Na prática, outras restrições serão estabelecidas para tornar o sistema modelado realizável e/ou facilitar operações sobre o modelo. A maior vantagem de utilizar *STGs* no projeto de sistemas digitais é a comprovada eficiência deste modelo em difinir causalidade e paralelismo a nível de sinais lógicos [13].

Definition 3.2.1 ***Signal Transition Graph*** é uma tupla $\langle N^*, S, \lambda \rangle$, onde

1. N^* é uma rede de Petri da classe *Free Choice,* adicionalmente, é *live* e *safe*;

2. $S = S_I \cup S_O \cup S_{Int}$ tal que $S_I = \{s_{i1}, s_{i2}, \cdots\}$ é um conjunto de sinais de entrada[4], $S_O = \{s_{o1}, s_{o2}, \cdots\}$ é um conjunto de sinais de saída e $S_{Int} = \{s_{int1}, s_{int2}, \cdots\}$ é um conjunto de sinais internos. Adicionalmente, $S_I, S_O \subseteq \{simple, data\}$ ou seja, os sinais de entrada e os sinais de saída podem ser

[4]Os sinais de entrada serão representados, em ilustrações, de forma sublinhada.

simples (sinais singulares) ou dados (barramentos e linhas de dado), e $S_{Int} \subseteq \{simple\}$, ou seja, os sinais internos somente podem ser sinais simples.

3. $\lambda : T \rightarrow \{S_I \times \{+,-\} \cup \{\sharp, @\}\} \cup \{S_O \times \{+,-\} \cup \{\sharp, @\}\} \cup \{S_{Int} \times \{+,-\} \cup \{\varepsilon\}\}$ é uma função de nomeação que atribui a cada transição exatamente um evento de um sinal. Dado um sinal s qualquer:

   (a) $s+$indica uma transição de subida $0 \rightarrow 1$;

   (b) $s-$ indica uma transição de descida $1 \rightarrow 0$;

   (c) $s\sharp$ indica que o dado na linha de dados tornou-se válido;

   (d) $s@$ indica que o dado foi invalidado; e

   (e) $s\varepsilon$ indica uma operação *dummy,* que não tem efeitos práticos.

4. Somente podem realizar eventos $\{\#, @\}$os sinais de entrada e/ou saída marcados como *data*. Similarmente, somente podem realizar eventos $\{+,-\}$sinais marcados como *simple*. Os eventos $\{\varepsilon\}$ somente podem ser atribuídos a sinais internos, portanto, *simple*.

A figura 3.11 ilustra um exemplo de um *STG* que descreve um protocolo tipo *return-to-zero (RTZ)* em uma operação de escrita do módulo A no módulo B. Quando o dado está válido, uma transição de subida no sinal $req+$ é realizada requisitando o início da transação. Quando o módulo receptor está pronto para capturar o dado, este sinaliza através de uma transição de subida no sinal $ack+$[5]. O sinal de requisição é então retirado (uma transição $0 \rightarrow 1$ no sinal req) seguido de um *acknowledge* ($0 \rightarrow 1$ no sinal ack). O dado então pode ser retirado e um novo ciclo inicia-se. Note que a "execução" da rede de Petri gera um *trace* de eventos que exprime o comportamento do protocolo.

É necessário ainda, fazer duas restrições ao modelo. Em primeiro lugar, o *STG* deve ser **válido**, em segundo, precisa ser ***output-persistent***.

[5]Sinais de entrada do módulo A, como o ack, aparecem sublinhados.

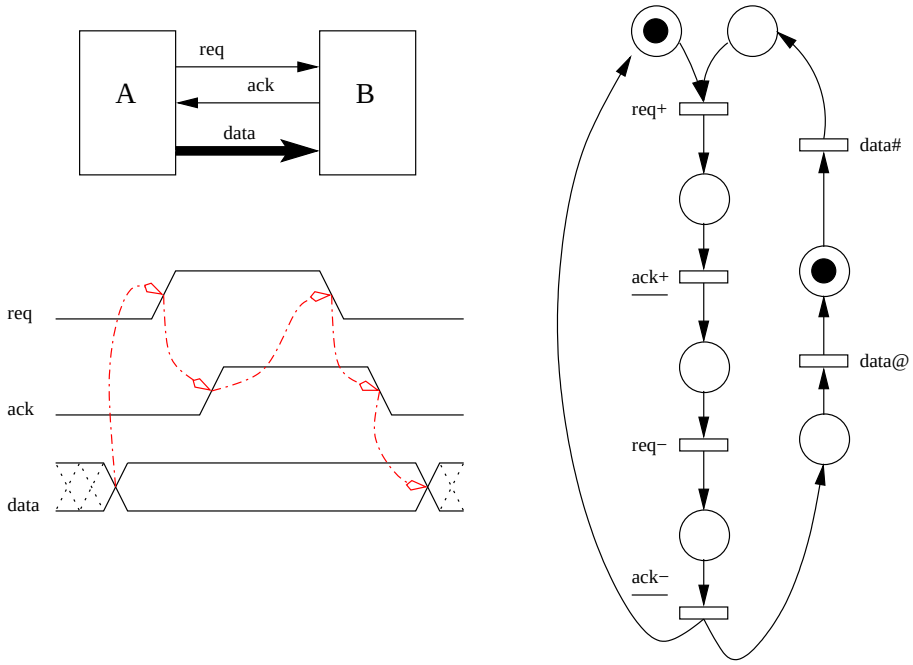


Figura 3.11: Descrição de um protocolo *rtz* em formato *STG*

Definition 3.2.2 Um *STG* é dito **válido**, ou **consistente**, se e somente se, para todas as possíveis sequências σ de disparo das transições, o *trace* de eventos gerado respeite as seguintes condições:

1. Há único evento possível para um dado sinal s, após uma $s+$ é $s-$. Ou alternativamente, o único evento possível para um dado sinal s, após $s-$ é $s+$. De forma similar para o par de eventos $s\sharp$, $s@$.

2. A primeira mudança em um sinal s é consistente com o estado inicial, ou seja, somente pode haver uma transição $0 \rightarrow 1$ se o sinal s é inicialmente igual a zero e, de forma similar, somente pode haver uma transição $1 \rightarrow 0$ se o sinal for inicialmente 1.

Conforme será estudado com mais detalhes posteriormente, é interessante, para o processo de síntese lógica, codificar os estados do grafo de alcançabilidade, gerado pelo *STG*, com um código binário que reflita o estado dos sinais em uma determinada marcação. A propriedade de validade é condição necessária e suficiente para garantir uma codificação consistente deste grafo de alcançabilidade.

Em outras palavras, a persistência em sinais de saída ou internos garante que um evento nestes sinais não será desabilitado pelo disparo de outro. A figura 3.12

mostra possíveis situações que tornam a rede não válida. Note que a execução da sequência $\sigma = t_1 t_2$, em 3.12(a) implica no *trace* seqüencial inválido $s+ \rightarrow s+$. Na situação em 3.12(b), a violação ocorre na execução em paralelo das transições t_2 e t_3, uma vez que, quando a transição $s- \rightarrow s+$ ocorre, é impossível determinar se o disparo foi em t_3 ou em t_2. A razão pela qual é necessária a propriedade de *safeness* é exposta em 3.12(c). Ao primeiro disparo da transição, o sinal s sofre uma mudança $0 \rightarrow 1$, porém a transição continua habilitada e ao disparar gera uma violação da validade.

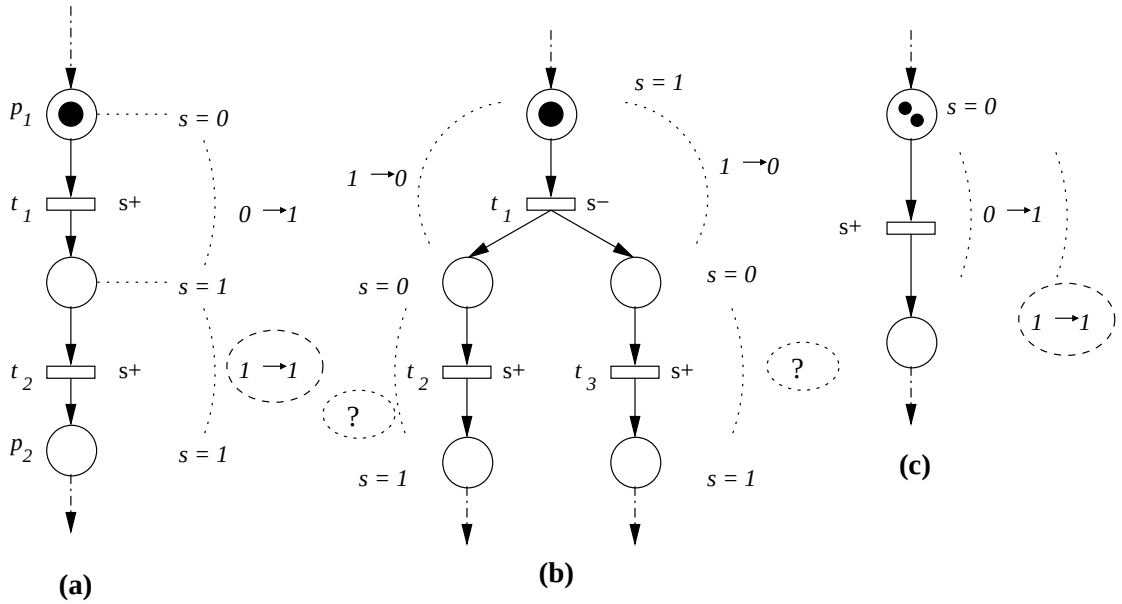
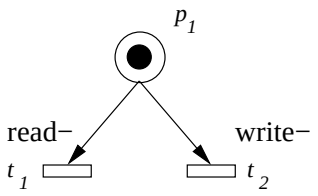


Figura 3.12: Possíveis violações da validade de um *STG*

Definition 3.2.3 Um *STG* é ***output-persistent*** se a rede de Petri é persistente em relação aos sinais internos e de saída, ou seja, não-entradas.

A figura 3.13 ilustra um exemplo de rede *output-persistent*. Ambas as transições t_1 e t_2 estão associadas a sinais de entrada.

Baseado nos conceitos discutidos até o momento, podemos formalizar as características desejadas para os modelos em redes de Petri a serem abordados neste trabalho.



(a)

Figura 3.13: A transição de descida informa a operação a ser realizada. Escolha baseada em sinais de entrada.

3.2.1 *STG* para os modelos de protocolo e processo de interface

É importante deixar claro quais as características desejadas pelos modelos em rede de Petri a serem utilizados no decorrer desta dissertação, e tecer comentários a respeito das restrições implicadas na modelagem. As redes de Petri que descrevem os protocolos de interface originais dos módulos a serem integrados, bem como a rede sintetizada a partir destes e que modela o processo de interface devem ser *Live and Safe Free Choice Petri Nets* (LSFCPN) [49], ou seja:

- São da classe de redes *Free Choice*;
- São *live;*
- São *safe* e conseqüentemente *bounded*;

Podem, portanto, capturar aspectos de paralelismo e sincronismo a serem desempenhados pelo processo de interface. A escolha incondicionada, característica modelável pela classe de redes *Free Choice*, no entanto, deve sofrer mais uma restrição. Isto ocorre pois a propriedade de persistência deve ser garantida para transições que representem eventos em sinais de saída. Neste sentido:

- Escolhas somente são permitidas entre sinais de entrada, ou seja, a rede é *output-persistent*;

Algumas abordagens [21] [48] requerem que a rede seja completamente persistente, o que transforma a classe de redes de Petri utilizada em *marked graphs*. Na prática,

a capacidade de modelar escolha é interessante na geração de processos de interface, pois, uma grande família de protocolos utiliza um ou vários sinais para indicar a operação a ser realizada, e a adoção de tamanha restrição é fator limitador sério. Por outro lado, estas sinalizações, onde a escolha deve acontecer, são freqüentemente feitas em sinais de entrada do processo de interface, de forma que a restrição de *output persistency* é completamente aceitável.

Adicionalmente, é necessário garantir que a interface retorne a seu estado inicial após uma operação, portanto:

- A rede de Petri deve ser reversível;

De fato, a reversibilidade da rede é uma propriedade desejada, mas que implica em uma das limitações deste método. Foi discutido anteriormente que a presente metodologia não era aplicável a protocolos que utilizassem transferências em modo *burst*. Neste tipo de transação o módulo transmissor executa apenas uma vez a sinalização necessária para envio dos dados e depois encadeia uma sequência de tamanho não definido de transferências, sem contudo, retornar ao estado inicial. Este comportamento pode levar a problemas na hora de modelar o sistema como um *STG* reversível.

Fica então claro porque não é possível modelar esta classe de protocolos. Em outras palavras, as operações devem ser atômicas, transferindo um dado a cada execução e retornando, necessariamente, a seu estado inicial. Por fim, uma última restrição:

- O *STG* modelado pela rede de Petri deve ser válido em todos os possíveis *traces*.

Estas são as condições necessárias para o modelamento dos protocolos e do processo de interface.

Uma restrição a mais é necessária para a síntese de circuitos, a partir deste modelo. Contudo, antes de discutí-la, convém apresentar os conceitos da síntese de hardware digital a partir de *signal transition graphs*.

3.3 Síntese de Circuitos Assíncronos a partir de *STGs*

As modernas abordagens à síntese de circuitos digitais, em especial circuitos assíncronos, foram discutidas na seção 3.3 deste trabalho. Em particular, detalharemos nesta seção a teoria introduzida por Jordi Cortadella et al. [51] [52] [53], para a especificação e síntese de controladores assíncronos que seguem um modelo temporal *speed-independent*. Esta teoria é largamente conhecida na literatura e foi implementada em uma ferramenta chamada *Petrify* [42].

Em linhas gerais, a metodologia proposta parte de uma especificação feita em *STG* e deriva um circuito lógico para sua implementação seguindo etapas similares à síntese lógica clássica. Por este motivo diz-se que a síntese é baseada em minimização lógica. Para cada sinal de saída ou interno, o procedimento acha uma equação booleana (ou **cobertura)** ao longo dos estados em que é desejado que o sinal tenha valor lógico igual a 1. Esta cobertura é então mapeada em uma interconexão física de portas lógicas de forma apropriada a garantir que o conjunto não apresenta *hazards*.

As restrições impostas à especificação inicial em *STG*, são as mesmas discutidas na seção 3.2.1:

1. *Boundedness*, e mais especificamente, 1-*safeness*;

2. O *STG* precisa ser válido para que a codificação binária dos estados no grafo de alcançabilidade permita uma interpretação significativa, em termos dos estados binários de onde serão derivadas as equações booleanas;

3. O *STG* precisa ser *output-persistent* para que o circuito a ser implementado seja livre de *hazards* comportamentais, provenientes da própria especificação.

Explicaremos o procedimento proposto pela metodologia através de um exemplo simples, discutido em [13], e ilustrado na figura 3.14. A entrada do sistema é o *STG* (3.14(a)), onde x e y são sinais de saída do sistema. Inicialmente, são verificadas as propriedades necessárias discutidas anteriormente. É fácil verificar, no exemplo proposto, que o *STG* é uma rede de Petri 1-*safe*, válida e persistente em relação ao sinal y e x.

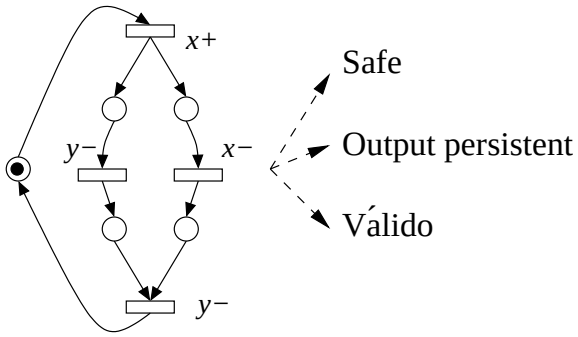


Figura 3.14: Análise de propriedades necessárias a implementação

O segundo passo consiste em construir o grafo de alcançabilidade da rede de Petri, atribuindo, a cada nó ou estado, um vetor binário onde cada posição representa o valor lógico de um sinal. De fato, este grafo codificado pode ser visto como um grafo dos possíveis estados do sistema. O grafo de alcançabilidade, ilustrado na figura 3.15, foi codificado com o vetor $\langle x, y \rangle$. Inicialmente os sinais encontram-se em repouso "$(0, 0)$", após o disparo de $x+$o sistema é codificado com o estado "$(1, 0)$". Neste estado, tanto o sinal x quanto o sinal y estão habilitados a transicionar. Note que o disparo de $y+$ leva o sistema ao novo estado "$(1, 1)$", no entanto, se o evento $x+$ ocorre em primeiro lugar o sistema migra a um novo estado, mas que possui uma codificação igual ao estado inicial (indicado na figura). Diz-se então que o sistema não possui "codificação completa de estado" ou mais comumente, não possui *complete state encoding* (*CSC*).

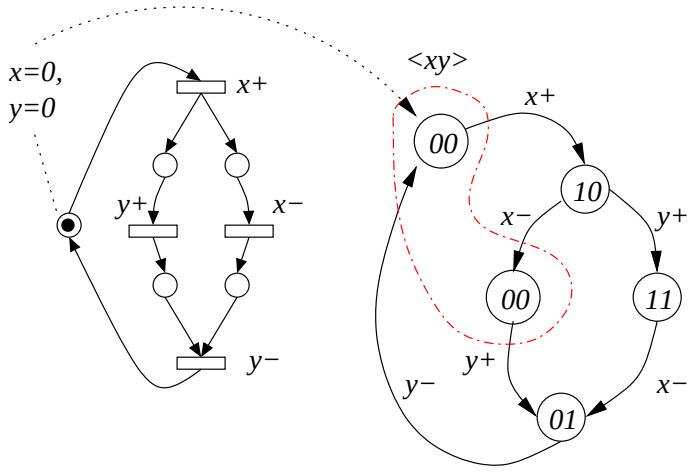


Figura 3.15: Não existe codificação completa de estados

Surge, então, uma nova restrição a ser imposta ao modelo em rede de Petri

(STG), uma condição necessária para síntese lógica: o sistema deve ter **codificação de estado completa**. Os métodos para resolução do problema de CSC merecem atenção especial, por serem implementados na ferramenta de CAD proposta nesta dissertação, e serão discutidos posteriormente. Aplicados estes métodos, um novo sinal é introduzido ao sistema exemplificado conforme ilustra a figura 3.16. Este novo sinal é considerado um sinal interno, e o sistema, observando apenas os sinais de entrada e saída, mantém o mesmo comportamento do sistema original. Chamamos a isto **equivalência observacional**.

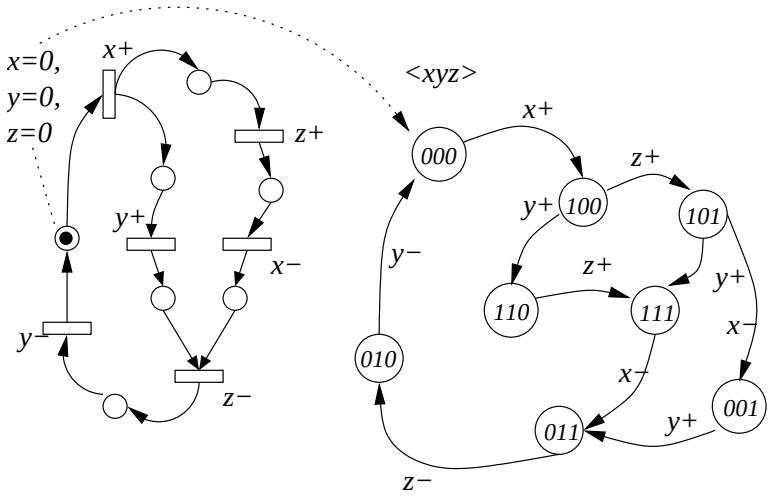


Figura 3.16: Codificação completa de estados através da inserção de sinais

Definition 3.3.1 Dois sistemas são **equivalentes observacionalmente** [54] com respeito a um conjunto observável de sinais se seus comportamentos não podem ser distinguidos observando-se este conjunto de sinais.[6]

Novamente, o grafo de alcançabilidade é construído e codificado (figura 3.16). Note que o problema de CSC foi eliminado e agora, todos os possíveis estados do sistema possuem codificação singular. As três propriedades inicialmente colocadas, mais o CSC, são condições necessárias e suficientes para a implementabilidade do sistema. Uma vez garantidas pode-se prosseguir à síntese lógica.

Na síntese lógica [41], é construído, para cada sinal não-entrada (saídas e sinais internos), um mapa de Karnaugh do estado futuro deste sinal, seguindo as seguin-

[6]Para uma definição formal de equivalência observacional refira-se a [55].

tes regras. Dado um sinal y, os estados do grafo de alcançabilidade podem ser classificados em quatro conjuntos:

Região de excitação positiva de y: Um estado pertence a $RE(y+)$ se $y = 0$ e $y+$é um evento habilitado neste estado;

Região de excitação negativa de y: Um estado pertence a $RE(y-)$ se $y = 1$ e $y-$ é um evento habilitado neste estado;

Região quiescente positiva de y: Um estado pertence a $RQ(y+)$ se $y = 1$ e não há eventos habilitados de y neste estado;

Região quiescente negativa de y: Um estado pertence a $RQ(y-)$ se $y = 0$ e não há eventos habilitados de y neste estado;

A função de estado futuro para o sinal y, em um estado s é

$$f_y(s) = \begin{cases} 1 \ if \ s \in RE(y+) \cup QR(y+) \\ 0 \ if \ s \in RE(y-) \cup QR(y-) \\ x \quad otherwise \end{cases}$$

onde x representa um "*don't care*" na minimização booleana. Seguindo o exemplo proposto, um mapa de Karnaugh para o sinal interno z pode ser visto na figura 16. O passo seguinte é mapear a "cobertura" dos estados onde a função recebe valor igual a 1 em uma interconexão de elementos de circuito que realize a função booleana sem *hazards*. Este processo é conhecido como **mapeamento tecnológico [56] [51].**

O mapeamento tecnológico é uma atividade altamente dependente da biblioteca de componentes (portas) lógicos. Assim, por exemplo, as seguintes arquiteturas são tipicamente utilizadas para a síntese de circuitos assíncronos do tipo *speed-independent*:

- cada sinal de saída ou interno é mapeado em uma porta lógica **complexa e atômica**; *Complex Gates* como são conhecidos estes dispositivos, são portas lógicas complexas que implementam atomicamente (uma só porta) toda uma função lógica. Assim, por exemplo, um porta *nand* de duas entradas é uma

complex gate para a função $f = \overline{xy}$. O problema é ter uma biblioteca de componentes vasta o bastante para adequar-se a maioria das funções booleanas. Mais ainda, este método é proibitivo, por exemplo, para mapeamentos tecnológicos em FPGAs, por exemplo, uma vez que o conjunto de portas disponível já está determinado fisicamente.

- cada sinal de saída ou interno é mapeado em um elemento de memória, ou *latch* (como um flip-flop RS, ou *C-elements*), controlados por duas funções de excitação, uma para *set* e outra para *reset* do sinal. As funções de excitação são implementadas com elementos complexos e atômicos [47] [48]. O mesmo problema anterior é verificado.

- similar à abordagem anterior, no entanto, as funções de excitação são decompostas em portas lógicas simples ou *latches*. Esta abordagem é mais ampla, podendo adequar-se inclusive ao ambiente de dispositivos reconfiguráveis, como *FPGAs*, desde que sejam observadas restrições quanto ao roteamento que possam violar as premissas do modelo *speed-independent*.

A figura 3.17, mostra o mapeamento tecnológico do sistema exemplificado em um biblioteca que contempla o último caso (o *latch* do sinal x é *reset-dominant* e o do sinal z é *set-dominant*).

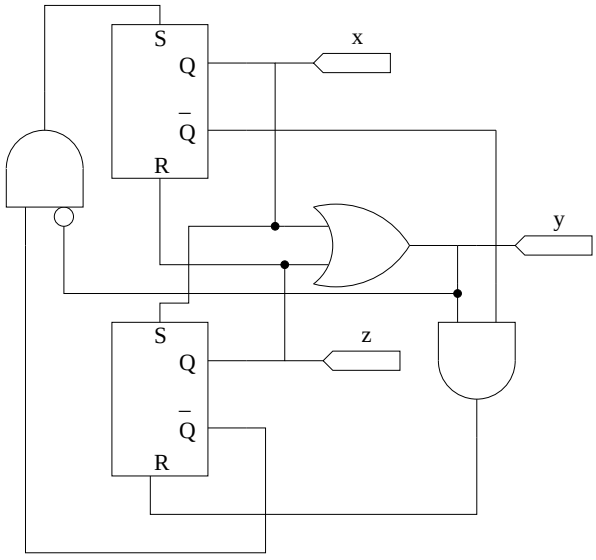


Figura 3.17: Implementações considerando diversas bibliotecas

3.3.1 Métodos e Ferramentas para a Codificação Completa de Estados (*CSC*)

Desde a introdução dos modelos *STG* como base para especificação e modelagem de sistemas, diferentes técnicas foram propostas para solucionar o problema do *CSC*, a maioria baseadas na introdução de sinais adicionais.

Uma abordagem utilizando minimização de máquinas de estado foi proposta por Luciano Lavagno et. al. [57]. Este método é capaz de fornecer com exatidão o número mínimo de sinais necessários para solucionar a codificação de estados, no entanto, as soluções obtidas são, em sua maioria, sub-ótimas, por causa da perda de informação ocorrida na minimização de estados. Adicionalmente, exige do projetista um esforço adicional para transformar o *STG* em uma máquina de estados finitos.

Josep Carmona [54] propôs uma técnica para resolver o problema através de modificações estruturais diretamente na rede de Petri do *STG*. Apesar de ser um método direto e simples, introduz uma quantidade de sinais adicionais maior do que a realmente necessária para solucionar o conflito, requerendo otimização posterior do circuito.

Mais recentemente, uma teoria baseada em regiões foi introduzida, capaz de solucionar de forma eficiente a maioria dos conflitos existentes em sistemas práticos [14] [58]. A técnica foi estudada e adotada ao longo deste trabalho, sendo implementada na ferramenta de CAD. Detalharemos alguns aspectos de sua abordagem mas omitiremos algumas provas referentes à teoria de regiões, por serem bastante extensas e saírem do escopo deste trabalho[7].

Seja S um sub-conjunto de estados de um grafo de alcançabilidade $G = (V, E)$ obtido a partir de um *STG*, tal que $S \subseteq V$. Então:

- Se um estado $s \notin S$ e $s' \in S$, então dizemos que uma transição $\overrightarrow{sas'} \in E$ **entra** em S;
- Se um estado $s \in S$ e $s' \notin S$, então dizemos que uma transição $\overrightarrow{sas'} \in E$ **sai** de S;

[7] O leitor interessado pode ver [58].

- Caso contrário, dizemos que a transição $\overrightarrow{sas'} \in E$ **não cruza** S.

Definition 3.3.2 Uma **região [13]** é um sub-conjunto de estados no qual todas as transições anotadas com o mesmo *label* de evento e têm exatamente a mesma relação entrada/saída.

Baseado neste conceito, diz-se que uma região r é **pré-região** de um evento e se existe uma transição $\overrightarrow{ses} \in E$ que sai de r. De forma análoga, r é uma **pós-região** de um evento e se existe uma transição que entra em r. Pré e pós regiões de r são denotadas como $\triangleright r$ e $\triangleleft r$, respectivamente. Adicionalmente, os conceitos de região de excitação (e quiescente) explicados anteriormente podem ser expressos como o máximo conjunto conectado de estados $ER(e)$ tal que , para todo estado de $ER(e)$, existe uma transição $\overrightarrow{sa}$.

Em linhas gerais, o problema de *CSC* pode ser resolvido particionando-se o conjunto de estados do grafo de alcançabilidade em dois sub-conjuntos a serem codificados diferentemente através da inserção de um sinal s, de forma que em uma parte da partição $s = 1$ e na outra, $s = 0$. Para implementar esta codificação, no entanto, é preciso inserir transições apropriadas nos estados que formam a borda entre estas duas partições.

Considere o exemplo na figura 3.18. A partição $r = r_1, \bar{r} = r_2$ separa todos os pares conflitantes, sendo portanto, uma boa escolha na tentativa de resolver o problema de *CSC*. As bordas entre estas duas partições ($EB(r_1)$ e $EB(r_2)$) serão transformadas em regiões de excitação (positiva e negativa) do sinal a ser inserido. Os outros estados constituirão as regiões de quiescência deste sinal. O novo sinal é inserido observando-se a técnica detalhada em [58] e produz o sistema de transição apresentado na figura 3.18.

Em resumo, a estratégia consiste em :

1. Encontrar todas as partições do grafo de alcançabilidade que podem ser apropriadamente aproveitadas para criar as regiões de excitação e quiescência do sinal a ser inserido;

2. Estimar o custo/benefício da inserção de um sinal para cada uma destas partições;

3. Selecionar a melhor partição;

4. Inserir apropriadamente o sinal, preservando consistência, *safeness* e *output-persistency*.

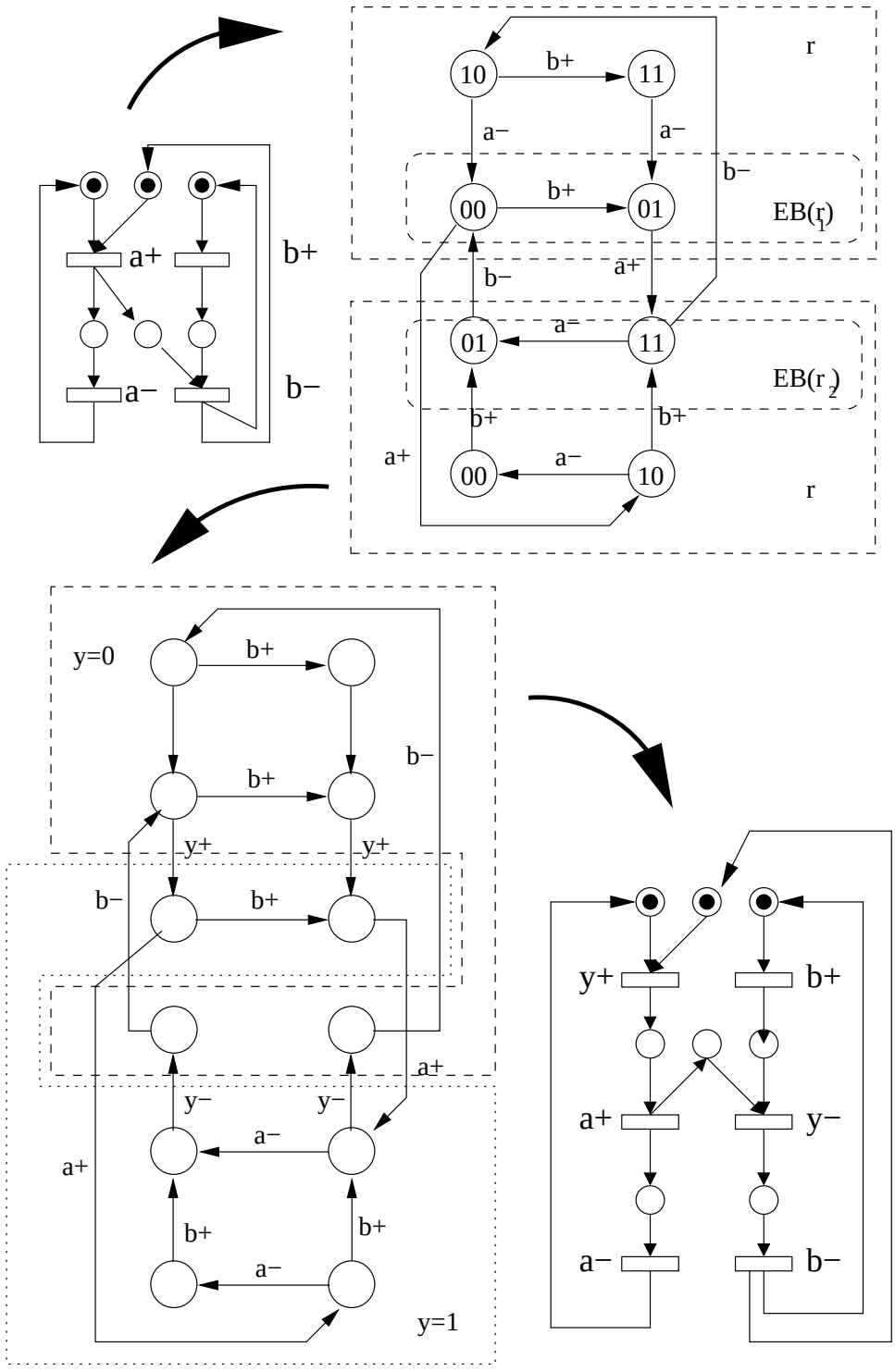


Figura 3.18: Estratégia para resolução do *CSC*

Note que nem todos os problemas de conflito foram resolvidos. Isto acontece por que alguns dos estados conflitantes encontravam-se na borda utilizada para a inserção do sinal. Repete-se então o processo, recursivamente, até o final de todos os problemas de conflito. Este método garante uma convergência para a solução se todos os sinais já existentes no sistema podem ser utilizados para construir as regiões de excitação.

3.4 Resumo

Este capítulo mostra uma pequena revisão nos conceitos de Redes de Petri, um formalismo matemático adequado para modelagem de sistemas com paralelismo e concorrência. Evidencia, em especial, as propriedades necessárias para seu uso na modelagem dos protocolos individuais dos módulos a serem integrados, na síntese do processo de interface e na implementação de circuitos assíncronos segundo um modelo *speed-independent*.

Uma classe de redes de Petri, chamada ***Live and Safe Free Choice Petri Net*** será utilizada ao longo de todo este trabalho, em forma de grafos de transição de sinais (*STG*), introduzidos em [50]. Além de pertencer a esta classe, a rede deve ser **persistente em relação aos sinais não-entrada** e deve ter um grafo de alcançabilidade com codificação de estados consistente, ou seja, deve ser **válida**. Adicionalmente, o grafo de alcançabilidade deve obedecer a uma codificação completa de estados. A técnica utilizada na ferramenta de *CAD* foi rapidamente discutida.

Estabeleceu-se assim as condições necessárias e suficientes para o desenvolvimento das atividades de modelagem e síntese da metodologia proposta. Esta metodologia será detalhada a seguir.

Capítulo 4

Metodologia para a Geração de Interface

4.1 Overview

O diagrama ilustrado na figura 4.1, mostra o fluxo geral de atividades a ser desenvolvido na geração do processo de interface.

A primeira etapa consiste na especificação dos protocolos individuais dos módulos a serem interligados. Esta especificação pode ser feita em alto nível, utilizando linguagens de descrição de hardware ou sistema (VHDL, SystemC) ou diagramas marcados (máquinas de estado, diagramas temporais). Neste trabalho, no entanto, discutimos somente as especificações realizadas através de diagramas temporais marcados.

A partir destas especificiações, em uma segunda etapa, cada protocolo de cada componente é então traduzido para um formato intermediário baseado em redes de Petri, mais especificamente *STGs*. A função desta camada é descrever o comportamento especificado de cada protocolo como um conjunto de transições de sinais. Esta tradução estabelece, através da semântica de redes de Petri, relações de precedência, causalidade, paralelismo e sincronismo entre os eventos que ocorrem nos sinais das interfaces. Estes eventos e relações são extraídos da especificação inicial.

Note que, ao final deste processo, os modelos traduzidos são submetidos a um exame de propriedades. Caso não existam condições suficientes e necessárias para

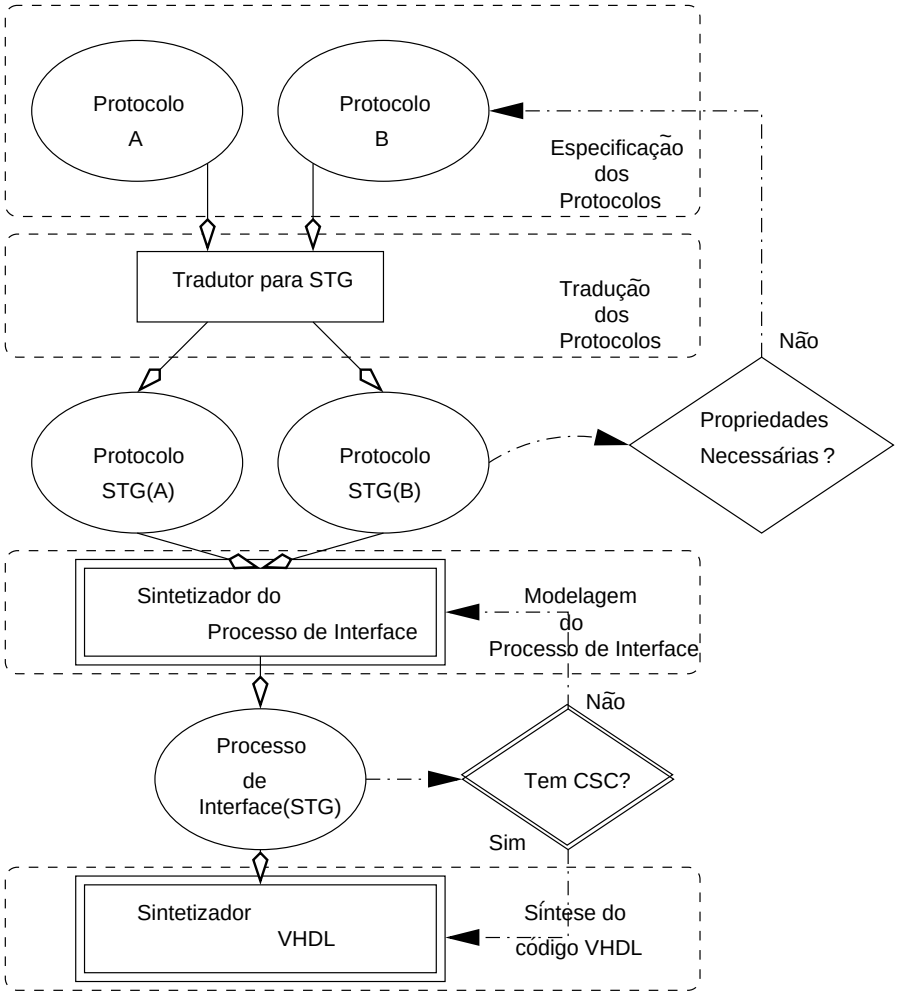


Figura 4.1: Fluxo de Atividades

prosseguir, isto indica que as especificações iniciais não são suficientes ou que não foram corretamente elaboradas. As especificações iniciais precisam, então, ser revistas.

Na terceira etapa seguem as operações de transformação e composição que transformarão os protocolos individuais (já em STG) em um único modelo, descrito em formato STG. Este novo modelo já não descreve mais protocolos, mas o comportamento do processo de interface. Este procedimento preserva as propriedades verificadas anteriormente, ou seja, o modelo do processo de interface é correto por construção, desde que os protocolos individuais estejam corretos.

Na qualta e última etapa da geração de interface ocorre a síntese. Nesta fase do processo, a descrição em STG da interface é convertida para um código VHDL sintetizável.

A síntese do processo de interface não resolve, no entanto, problemas de *CSC* já existentes nas descrições dos protocolos e pode, inclusive, inserir novos. Por este motivo, ao final desta etapa, o modelo do processo de interface deve ser verificado contra problemas de codificação de estados. Se estes existirem, o processo de síntese pode ser recursivamente executado, adicionando-se a cada rodada novas informações (sinais) para a resolução destes problemas.

Quando o modelo do processo de interface está correto[1] o sistema pode ser utilizado para:

- Verificação funcional do processo de interface;
- Síntese do circuito, como uma netlist;
- Síntese de código em linguagem de descrição de hardware (VHDL, Verilog)[2];

Neste capítulo, a síntese de código VHDL sintetizável será estudada em detalhes. O produto final deste procedimento é um *firmware core*, ou seja, uma descrição sintetizável do processo de interface, a qual pode ser facilmente integrada ao sistema de forma estrutural, unindo os dois módulos comunicantes.

Ao longo deste capítulo um exemplo simples, porém suficiente, será apresentado de forma a facilitar o entendimento das transformações a serem realizadas em cada etapa do processo de criação do circuito de interface. Ao final de cada seção, detalharemos como esta etapa foi implementada na ferramenta de CAD, fruto desta dissertação.

4.2 Especificação dos protocolos individuais

A especificação dos protocolos de cada um dos módulos a serem interligados gera as entradas do processo de geração da interface. Em teoria, a especificação destes protocolos pode ser feita utilizando-se qualquer base. Pode-se, por exemplo, descrever o comportamento da interface em *SystemC* [5], ou em VHDL [4], desde que estas

[1] Apresenta as propriedades de *safeness, output-persistency, validade e CSC* em redes da classe *free-choice*.

[2] Este código pode ser sintetizável ou com fins de simulação funcional e temporal.

especificações possam ser refinadas até que descrevam o comportamento em termos dos eventos que ocorrem em cada sinal da interface.

A forma como este refinamento é realizado, ou como utilizar estas plataformas de maneira a especificar adequadamente os protocolos, está fora do escopo desta dissertação. O método escolhido para realizar a especificação inicial, neste trabalho, foram os diagramas temporais marcados, por dois motivos principais:

1. Os diagramas temporais fazem parte da cultura geral disseminada entre os projetistas de hardware. Desde cedo, projetistas aprendem a "ler" o comportamento de uma interface através destes esquemas gráficos. Sua disseminação é quase universal, estando presente em *data-sheets*, *data-books*, manuais técnicos, livros, etc...

2. Os diagramas temporais são facilmente traduzidos para um *Signal Transition Graph (STG)*, que constitui uma classe das redes de Petri anotadas. Esta facilidade de tradução se dá pelo fato de os diagramas de tempo já transmitirem naturalmente o comportamento da interface em forma de eventos ocorridos nos sinais.

Os diagramas temporais puros (sem anotação), como são normalmente encontrados, podem incluir, de forma implícita, uma relação de causalidade entre os eventos. Durante a anotação dos diagramas temporais, estas relações são tornadas explícitas segundo o desejo do projetista. Tomemos, por exemplo, uma operação de escrita a ser realizada de um módulo A em um módulo B, com protocolos diferentes. A figura 4.2 descreve os protocolos de cada módulo através de dois diagramas temporais, ainda não marcados.

Apesar de não exibir marcas, um projetista "sabe" que, em uma operação de escrita, o módulo A disponibiliza os dados no barramento, seleciona o dispositivo destino e indica a operação de escrita. Só então, leva o sinal *enable* para 1, indicando que o dado pode ser lido. A partir daí, um pulso no sinal *wait* informa que a operação já pode ser concluída e os sinais podem ser desabilitados. Este entendimento "implícito" precisa ser claramente exposto através de marcas que estabelecem

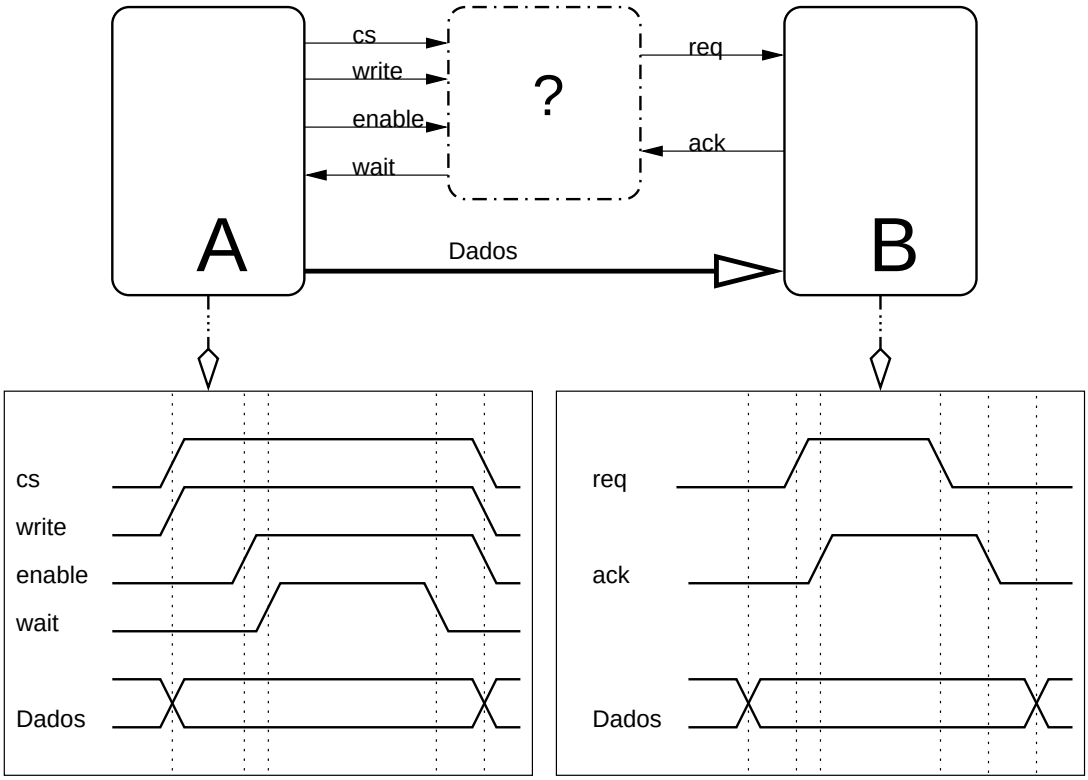


Figura 4.2: Operação de escrita de A em B. Especificação dos protocolos.

a relação de causalidade entre os sinais. A figura 4.3 mostra estas marcas para o exemplo dado.

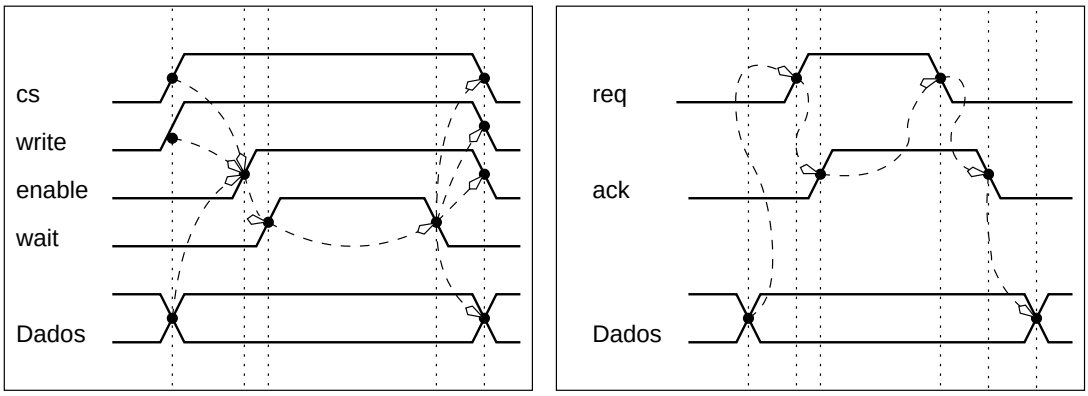


Figura 4.3: Protocolos anotados.

Para a maioria dos casos, contudo, não existe apenas uma forma possível de anotar o protocolo. O projetista pode optar por estabelecer estas relações explorando diferentes graus de paralelismo. Na figura 4.3 as transições nos sinais *cs*, *write* e *Dados* anteriores ao evento de subida do sinal *write* ocorrem de forma concorrente. O projetista poderia, no entanto, descrevê-los sequencialmente, estabelecendo uma

ordem de prioridade entre estes. Uma pergunta surge: quais os critérios de validação desta marcação? Para a finalidade de geração do processo de interface, estes critérios são:

- A marcação deve cobrir todas as possibilidades de execução do protocolo consideradas como válidas na execução de uma transação.
- A rede de Petri gerada no passo seguinte deve ser uma *Live and Safe Free Choice Petri Net* (*LSFCPN*). Note que esta atividade pode ser iterada com a seguinte, uma vez que a análise destas propriedades é de fácil resolução.
- As relações devem estabelecer um protocolo **válido**, ou seja, uma transição de subida somente pode ocorrer se o sinal estiver originalmente em estado 0, e analogamente, uma transição de descida somente deve ocorrer se o sinal for originalmente 1.
- Escolhas podem ser estabelecidas desde que os sinais marcados sejam sinais de "saída". No exemplo anterior, uma operação de leitura poderia ser diferenciada da de escrita se não houvesse a transição *write*+. Note que o sinal *write* é saída do módulo *A*.

Restrições temporais dos sinais não foram estudados nesta primeira abordagem, no entanto, defendemos que esta preciosa informação deve ser contabilizada em futuros trabalhos. A segunda etapa é a tradução desta especificação em redes de Petri anotadas detalhadas a seguir.

4.3 Tradução dos protocolos para *STG*

A especificação inicial de cada componente precisa ser "traduzida" para um formato intermediário independente da linguagem utilizada (SystemC, diagramas temporais, VHDL, entre outras). O formato intermediário escolhido foi um sub-conjunto das classes de redes de Petri anotadas chamadas *signal transition graphs* (*STG*). Este formalismo foi introduzido em meados dos anos 80 por Chu [50] e seu formalismo foi introduzido na seção 3.2.

A primeira atividade da etapa de tradução é estabelecer a direção e natureza dos sinais. Quanto a direção, os sinais devem ser classificados como:

Sinais de entrada: São sinais de controle "lidos" pelo módulo, em sua interface, durante a execução de uma transação.

Sinais de saída: São sinais "controlados" pelo módulo, em sua interface, durante a execução de uma transação.

Sinais bidirecionais: Não aplicável aos sinais de controle, somente aos dados.

Estas são as possíveis direções dos sinais descritos na especificação inicial dos protocolos.

Quanto a sua natureza, estes sinais são classificados como:

Sinais simples: São aqueles formados por uma única via elétrica e são normalmente parte da sinalização de controle da interface. Aos sinais simples somente podem ser atribuídos eventos de transição de subida (denotados com um $+$) e descida (denotados com um $-$).

Sinais de dado: São sinais formados por uma ou mais vias elétricas e correspondem aos barramentos de dados e endereçamento. Aos sinais de dados somente podem ser atribuídos eventos de validação[3] do dado (denotados com um $\sharp$) ou invalidação[4] (denotados com um @).

Durante a síntese do processo de interface, contudo, uma terceira classe surge, denominada **sinais internos**. Estes sinais são sinais de saída, no entanto não são controlados ou lidos por nenhum dos módulos a serem interligados, e sim pelo processo de interface. Os **sinais internos** são sempre sinais simples. A tabela 4.1 mostra a classificação dos sinais no exemplo da figura 4.2.

A segunda atividade é a síntese da rede anotada. Para tal, é necessário que a especificação inicial seja refinada até representar o comportamento do protocolo a nível de transições nos sinais. Este nível de abstração é naturalmente capturado

[3]Data assertion.
[4]Data de-assertion.

Sinal	Direção	Natureza
cs	saída	simples
write	saída	simples
wait	entrada	simples
enable	saída	simples
req	entrada	simples
ack	saída	simples
Dados	saída (A), entrada(B)	dado

Tabela 4.1: Classificação dos sinais nos protocolos especificados na figura 4.2.

nos diagramas de tempo, como discutido anteriormente. Cada evento descrito no protocolo é mapeado em uma transição do STG e formarão os nós do grafo.

Os arcos entre cada transição descrevem as relações de causalidade descritas entre os eventos na especificação do protocolo. Desta forma um arco entre dois eventos, denotado como $e_1 \rightarrow e_2$, significa que o evento e_1 **precede** o evento e_2 . Em outras palavras, o evento e_2 está habilitado a disparar após a ocorrência do evento e_1.

Para transformar este grafo em uma rede de Petri, é necessário apenas atribuir a cada arco um lugar intermediário. O lugar será o receptor de *tokens* após o disparo de uma transição e sua visualização facilita a percepção de "estado" do sistema. Considerando uma rede de Petri como definido em 3.1.1, cada marcação alcançável $M \in [Mo >$ corresponde a um estado do sistema e as transições habilitadas em uma dada marcação descrevem os eventos que podem ocorrer neste estado. Desta forma, o conjunto de *traces* obtido com a execução da rede cobre todas as variações válidas do protocolo. A figura 4.4 mostra a tradução dos protocolos especificados e anotados anteriormente no exemplo proposto[5]. Note que os sinais de entrada estão sublinhados.

Para a modelagem de escolha em um sinal, um lugar é utilizado para representá-la. A figura 4.5 demonstra este conceito através da introdução de uma fictícia operação de leitura adicionada ao protocolo do módulo A exposto anteriormente. Note que se o evento *enable*+ ocorre antes de *write*+, a transação ocorre segundo descrito nos laços em 4.5(a), caso contrário nos laços em 4.5(b). A escolha deve ser inserida observando-se dois aspectos:

[5]Baseado nas relações de causalidade estabelecidas na figura 4.3.

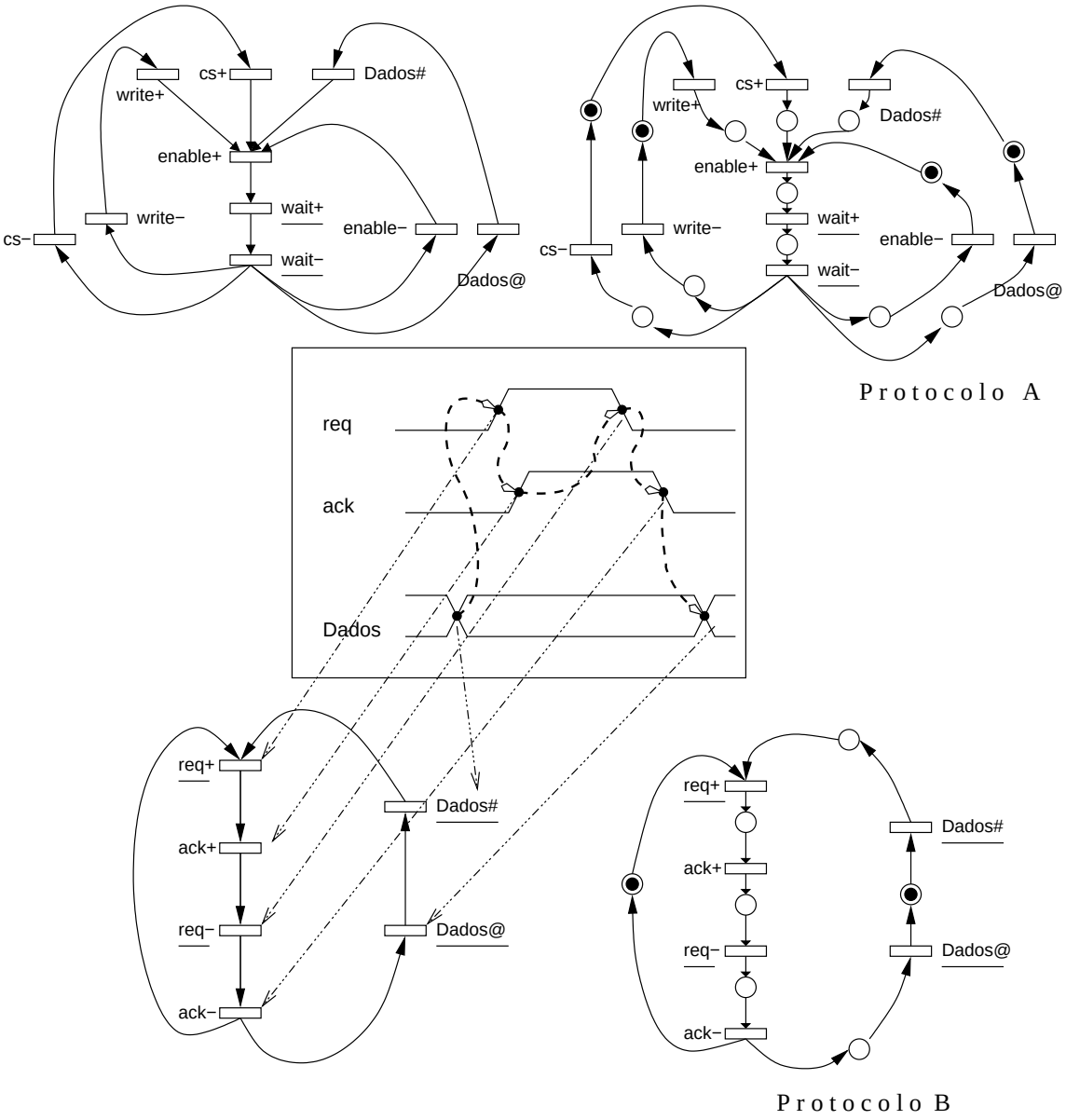


Figura 4.4: Tradução dos diagramas temporais para *STG*

- Os sinais envolvidos na escolha são sinais de *saída* da interface modelada. No exemplo dado, *write* e *enable* são sinais de saída.

- Não violar a **validade** da rede. Ou seja, transições de subida e descida, em um mesmo sinal, devem suceder-se. Aparentemente, em nosso exemplo, se a escolha fosse modelada com a inserção de apenas um lugar (como indicado em "escolha proibida", figura 4.5) o modelo poderia ser simplificado. Note no entanto, que a rede não seria válida pois em uma operação de leitura o evento *write*− poderia ocorrer sem que o sinal *write* estivesse originalmente em 1.

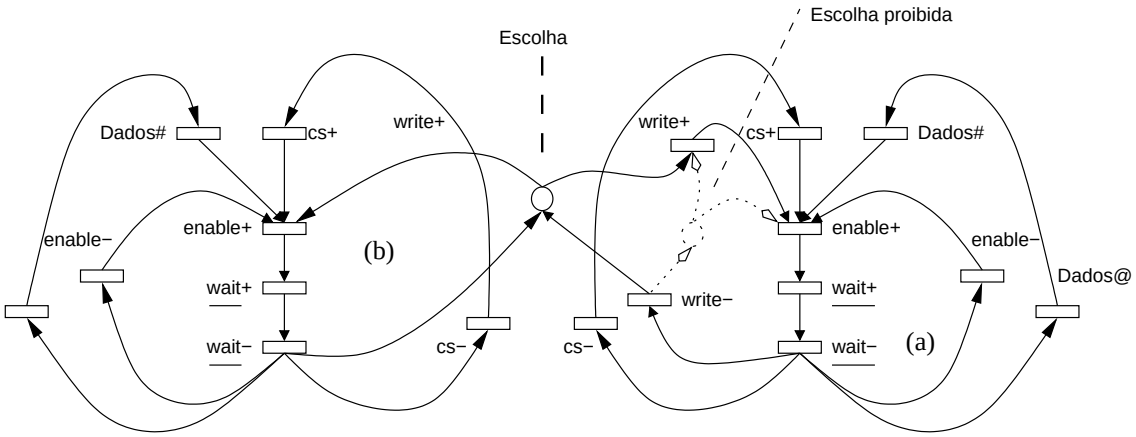


Figura 4.5: Modelando escolhas

4.3.1 Verificação de Propriedades

A ferramenta de CAD implementada neste trabalho recebe, como entrada, os dois protocolos individuais dos módulos já traduzidos em formato *STG*. Não é necessário, contudo, que o projetista preocupe-se em atender os requisitos necessários na primeira modelagem. A ferramenta da CAD é capaz de identificar e indicar que propriedades estão faltando em cada protocolo, de forma que o projetista pode, baseado nestas informações, refinar o modelo inicial.

As propriedades verificadas pela ferramenta de CAD nesta etapa são:

Boundedness Caso a rede esteja acumulando indefinidamente *tokens* em um lugar isto pode gerar uma explosão no número de estados do sistema. A rede precisa ser *bounded*[6].

Safeness Além de limitada, a rede deve ser *1-safe*[7].

Liveness A presença de *dead-locks* é verificada no sistema. *Dead-locks* podem ser causados quando o projetista estabelece de forma inapropriada as relações de causalidade entre os eventos[8].

Reversibilidade O sistema é capaz de retornar ao estado inicial, a partir de qual-

[6]O conceito de *boundedness* foi introduzido na definição 3.1.2.
[7]O conceito de *safeness* foi introduzido na definição 3.1.3.
[8]O conceito de *liveness* foi introduzido na definição 3.1.4.

quer marcação alcançável[9]?

Validade A rede deve ser válida para todos os possíveis *traces* de eventos[10].

Free Choice A rede deve ser livres de conflitos estruturais, ou seja, deve ser da classe das *free choice.* Se esta propriedade existir em conjunto com a de *liveness* e *safeness,* a rede pertence às *LSFCPN.* Adicionalmente, exige-se que as operações de escolha sejam feitas somente sobre sinais de "saída" nesta etapa.

Todas estas propriedades são analisadas através da construção de uma árvore de cobertura da rede, e, quando necessário, de seu grafo de alcançabilidade. A única excessão é a propriedade de *free-choiceness* que é verificada diretamente na estrutura da rede.

Boundedness e *safeness* são verificados se nenhum nó da árvore de cobertura é ilimitado e se todos os nós contém marcações com 1's e 0's, respectivamente [44]. *Liveness* implica que não existem nós marcados como "morto" na árvore de cobertura[11]. Reversibilidade e validade são analisados percorrendo os *traces* na árvore ou grafo de alcançabilidade.

Quando estas propriedades estão presentes em ambos os *STGs* que descrevem os protocolos, o projetista pode partir para a próxima etapa: a síntese do processo de interface.

4.4 Síntese do Processo de Interface

A etapa de síntese do processo de interface tem como finalidade descrever o comportamento do circuito a ser interposto entre os módulos comunicantes. Esta etapa tem como entrada os protocolos individuais de cada módulo, descritos a nível de transições nos sinais de sua interface, devidamente traduzidos para o formato *STG* e com as propriedades discutidas anteriormente verificadas. Doravante, denominaremos as entradas desta etapa como **protocolos padronizados**, para distingui-los

[9] O conceito de reversibilidade foi introduzido na definição 3.1.5.
[10] O conceito de validade foi introduzido na definição 3.2.2.
[11] Ver discussão na seção 3.1.1.

das descrições iniciais que ainda podiam conter erros ou não estarem adequadamente formatadas. Ao final do processo exposto nesta seção, teremos como produto uma única rede de Petri, no formato *STG*. Esta nova rede não mais descreve os protocolos individuais, mas sim, o comportamento do processo de interface.

A síntese do processo de interface é realizada através de transformações e composições aplicadas às redes de entrada. A álgebra envolvida nestas transformações foi, em sua maioria, desenvolvida por Gjalt G. de Jong e Bill Lin [22] [12] e preservam as propriedades anteriormente discutidas[12]. Os passos a serem aplicados são: modificação dos protocolos padronizados para criação dos pontos de sincronismo, composição paralela das redes e espelhamento.

4.4.1 Modificação dos protocolos padronizados e Composição Paralela

O passo inicial é criar os pontos através dos quais serão feitas as sincronizações de atividades entre os dois protocolos. A identificação dos eventos a serem utilizados como pontos de sincronismo pode ser feita, alternativamente, por dois métodos: através de indicação explícita do projetista ou automaticamente, através de um procedimento baseado em uma heurística.

Para realizar uma indicação explícita, o projetista precisa criar, ainda na especificação inicial, relações de causalidade entre os dois protocolos. Estas relações de causalidade serão ignoradas nas etapas anteriores, sendo utilizada somente neste passo. É importante que esteja claro para cada relação criada, quais eventos são a fonte do sincronismo e quais os receptores. São considerados fontes de sincronismo os eventos que, ao serem disparados, podem habilitar o disparo de um evento receptor.

A figura 4.6 ilustra duas relações criadas manualmente na especificação inicial do nosso exemplo. O disparo da transição $req+$ no protocolo B somente estará habilitado após a ocorrência do sinal $wait+$ no protocolo A. Da mesma forma, o sinal $wait$ somente poderá ter uma transição $1 \rightarrow 0$, uma vez que o evento $ack-$

[12] As provas formais de que estas transformações preservam as propriedades de *safeness, liveness, boundedness, free-choice, validade e reversibilidade* estão detalhadas em [22].

tenha ocorrido no protocolo B.

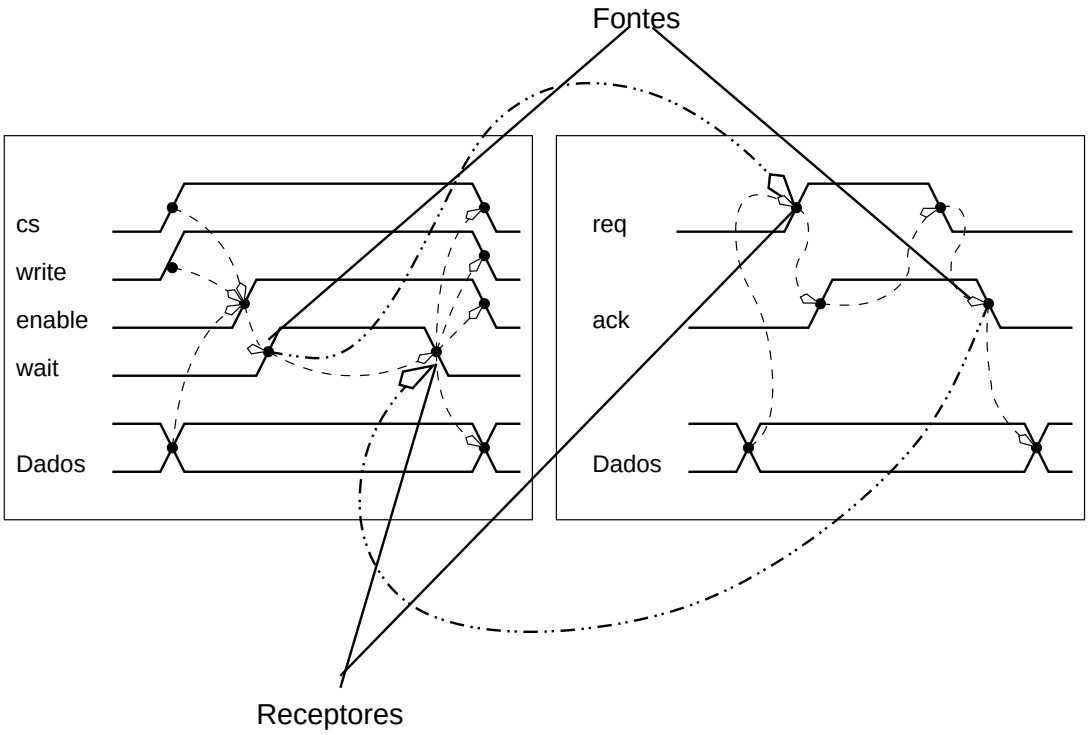


Figura 4.6: Relações de causalidade para sincronismo.

O segundo método para identificação dos pontos de sincronismo procura determinar regiões onde o dado transmitido está válido ou não. Quando uma emissão do dado é feita por um dos módulos, este precisa indicar ao outro módulo através de um evento gerado em um sinal de controle. De maneira análoga, antes do dado ser retirado do barramento, o módulo receptor precisa indicar que esta operação já pode ser realizada.

Para a primeira condição consideramos que qualquer transição, cujo antecessor é um lugar de saída da transição marcada com o evento de validação[13] do dado, pode ser utilizado como fonte e receptor de uma relação de sincronismo. A fonte será no protocolo responsável pela colocação do dado no barramento e o receptor no protocolo complementar. Esta relação estabelece para ambos os protocolos o início da região onde o dado é válido.

O protocolo do módulo A, na figura 4.7, é responsável pelo envio do dado. Neste

[13]Indicado com o símbolo $\sharp$.

caso, a transição *enable+* será disparada somente quando o dado for válido[14] e pode ser escolhida como "fonte" de uma relação de causalidade para sincronismo. Analogamente, a transição *req+*, no protocolo do módulo B, tem como pré-condição o disparo do evento $Dado\sharp$ e é candidata a "receptora" da mesma relação de causalidade.

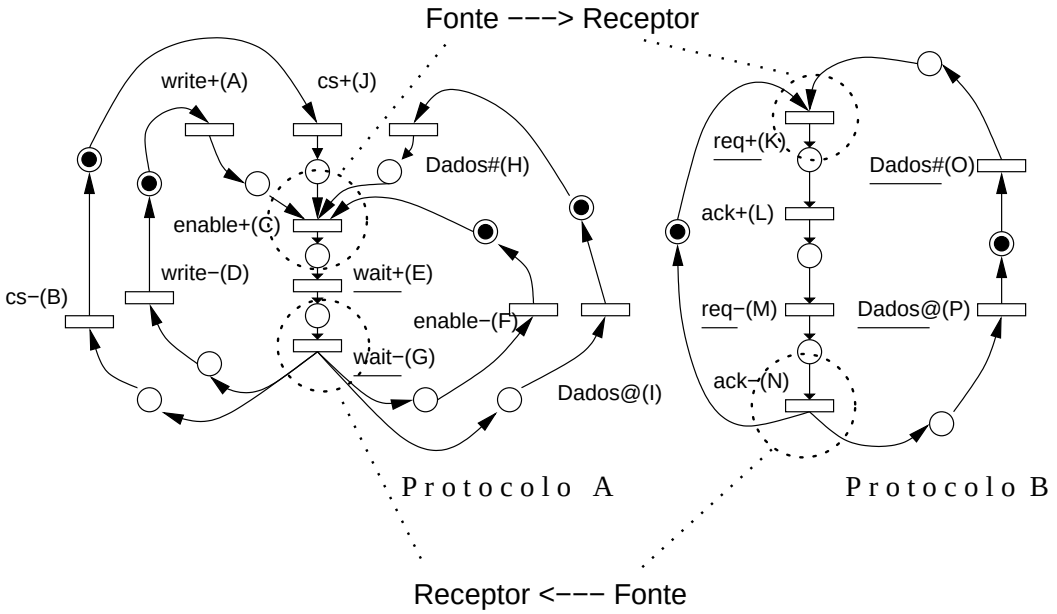


Figura 4.7: Identificando automaticamente pontos de sincronismo entre os protocolos

De forma similar pode-se descobrir a região onde o dado não é mais válido. Nesta situação, todas as transições cujo lugar de saída é antecessor do evento de invalidação[15] do dado formam um conjunto de pré-condições que pode ser utilizado como fonte ou receptor de uma relação de causalidade para sincronismo. Para estes casos, a fonte da relação encontra-se no protocolo que sinaliza que os dados podem ser retirados. O receptor da relação está no protocolo responsável pela retirada dos dados no barramento.

O protocolo do módulo B, na figura 4.7, indica ao protocolo do módulo A, que os dados podem ser retirados, com o disparo do evento *ack−* (fonte da relação de causalidade). Isto implica que o evento *wait−*, pré-condição para a retirada

[14]Nada impede que outros eventos ocorram entre a colocação do dado e o disparo de *enable+*. Veja por exemplo *cs+* e *write+*.
[15]Indicado com o símbolo @.

dos dados, somente poderá ser disparado após a sinalização do módulo B, sendo portanto, receptor da relação de causalidade.

É necessário, portanto, formalizar esta identificação.

Definition 4.4.1 Dado um STG $G_1 = \langle N_1^*, S_1, \lambda_1 \rangle$, denomina-se **conjunto** $\prod$ ao conjunto de transições $\{t \mid t \in T_1 \wedge \lambda(t) \in \{S_O \times \sharp\}\}$. e indica-se como $\prod(G_1)$. Adicionalmente, denomina-se **conjunto** $\prod^*$, e indica-se como $\prod^*(G_1)$, ao conjunto de transições $\{t \mid t \in T_1 \wedge \lambda(t) \in \{S_I \times \sharp\}\}$.

Definition 4.4.2 Dado um STG $G_1 = \langle N_1^*, S_1, \lambda_1 \rangle$, denomina-se **conjunto** $\coprod$ ao conjunto de transições $\{t \mid t \in T_1 \wedge \lambda(t) \in \{S_I \times @\}\}$. e indica-se como $\coprod(G_1)$. Adicionalmente, denomina-se **conjunto** $\coprod^*$, e indica-se como $\coprod(G_1)$, ao conjunto de transições $\{t \mid t \in T_1 \wedge \lambda(t) \in \{S_O \times @\}\}$.

Definition 4.4.3 Dado um STG $G_1 = \langle N_1^*, S_1, \lambda_1 \rangle$ e $t_k \in \prod(G_1)$, denomina-se de **conjunto fonte de validação de** t_k ao conjunto de transições $\{t \mid t \in T_1 \wedge \bullet(\bullet t) \subseteq \{t_k\}\}$, e indica-se $\bigvee(G_1, t_k)$. Adicionalmente, dado $t_m \in \prod^*(G_1)$ denomina-se de **conjunto receptor de validação de** t_m ao conjunto de transições $\{t \mid t \in T_1 \wedge \bullet(\bullet t) \subseteq \{t_m\}\}$, e indica-se $\bigvee^*(G_1, t_k)$.

Definition 4.4.4 Dado um STG $G_1 = \langle N_1^*, S_1, \lambda_1 \rangle$ e $t_k \in \coprod(G_1)$, denomina-se de **conjunto fonte de invalidação de** t_k ao conjunto de transições $\{t \mid t \in T_1 \wedge (t\bullet)\bullet \subseteq \{t_k\}\}$, e indica-se $\bigwedge(G_1, t_k)$. Adicionalmente, dado $t_m \in \coprod^*(G_1)$, denomina-se de **conjunto receptor de invalidação de** t_m, e indica-se $\bigwedge^*(G_1, t_m)$, ao conjunto de transições $\{t \mid t \in T_1 \wedge (t\bullet)\bullet \subseteq \{t_m\}\}$.

Uma vez identificados os conjuntos fontes e receptores das relações de causalidade, é necessário gerar as estruturas de rede que interligarão os protocolos. Isto é feito através de uma composição paralela.

Definition 4.4.5 [22]Seja $N_i = \langle P_i, T_i, F_i, Mo_i \rangle, i \in \{1, 2\}$, duas redes de Petri[16] tais que $P_1 \cap P_2 = \emptyset$. A **composição paralela** das redes é definida como : $N_1 \parallel$

[16]A definição formal de uma rede de Petri foi introduzida em 3.1.1.

$N_2 = (P_1 \cup P_2, T_1 \cup T_2, F', Mo_1 \cup Mo_2)$ onde

$$\begin{aligned} F' &= \{(p,t),(t,p) \in F_1 \cup F_2 | t \notin T_1 \cap T_2\} \cup \\ &\quad \{(p_i,t_i),(t_i,p_i) | t_i \in T_1 \cap T_2 \wedge (p_i,t_i),(t_i,p_i) \in F_i\} \end{aligned}$$

No intuito de simplificar as operações, foi estabelecido um procedimento único para identificar os pontos de sincronismo e interligar os dois protocolos. Este procedimento identifica fontes e receptores das relações de causalidade e realiza, de forma implícita, a composição paralela entre as redes. Para facilitar o entendimento, a geração dos pontos de sincronismo será feita, sempre, a partir dos "conjuntos de transição fonte" da relação de causalidade. Dados os dois *STGs* G_A e G_B , a modificação segue os seguintes passos:

1. Calcule os conjuntos $\prod(G_A)$,$\prod(G_B)$,$\prod^*(G_A)$,$\prod^*(G_B)$;
2. Para cada operação descrita nos protocolos faça:
   (a) Identifique a transição $t \in \prod(G_A)$ referente à operação. Indentificar a transição $t^* \in \prod^*(G_B)$ referente à mesma operação;
   (b) Se $\prod(G_A) \neq \emptyset$, escolha apenas uma transição $t_f \in \bigvee(G_A, t)$ e faça:
      i. Para cada transição $t_r^* \in \bigvee^*(G_B, t^*)$, crie um lugar novo p. Crie os arcos que (t_f, p) e (p, t_r^*);
   (c) Repita os passo 2(a) e 2(b) trocando os *STGs* G_A e G_B;
3. Calcule os conjuntos $\coprod(G_A)$,$\coprod(G_B)$,$\coprod^*(G_A)$,$\coprod^*(G_B)$;
4. Para cada operação descrita nos protocolos faça:
   (a) Identifique a transição $t \in \coprod(G_A)$ referente à operação. Indentificar a transição $t^* \in \coprod^*(G_B)$ referente à mesma operação;
   (b) Se $\coprod(G_A) \neq \emptyset$, escolha apenas uma transição $t_r^* \in \bigwedge^*(G_B, t^*)$ e faça:
      i. Para cada transição $t_f \in \bigwedge(G_A, t)$, crie um lugar novo p. Crie os arcos que (t_f, p) e (p, t_r^*);

(c) Repita os passo 4(a) e 4(b) trocando os *STGs* G_A e G_B;

A figura 4.8 ilustra a criação dos pontos de sincronismo, nos dois protocolos exemplo, a partir das transições fontes. O conjunto de fonte de validação do protocolo A é a transição nomeada como (C), e será interligada através de um caminho ao conjunto receptor de validação em B (transição nomeada como (K)). Novos elementos são adicionados a rede como explicado anteriormente. Analogamente, o conjunto fonte de invalidação é a transição nomeada como (N) no protocolo B. Ela será interligada ao conjunto receptor de invalidação de A (transição (G)).

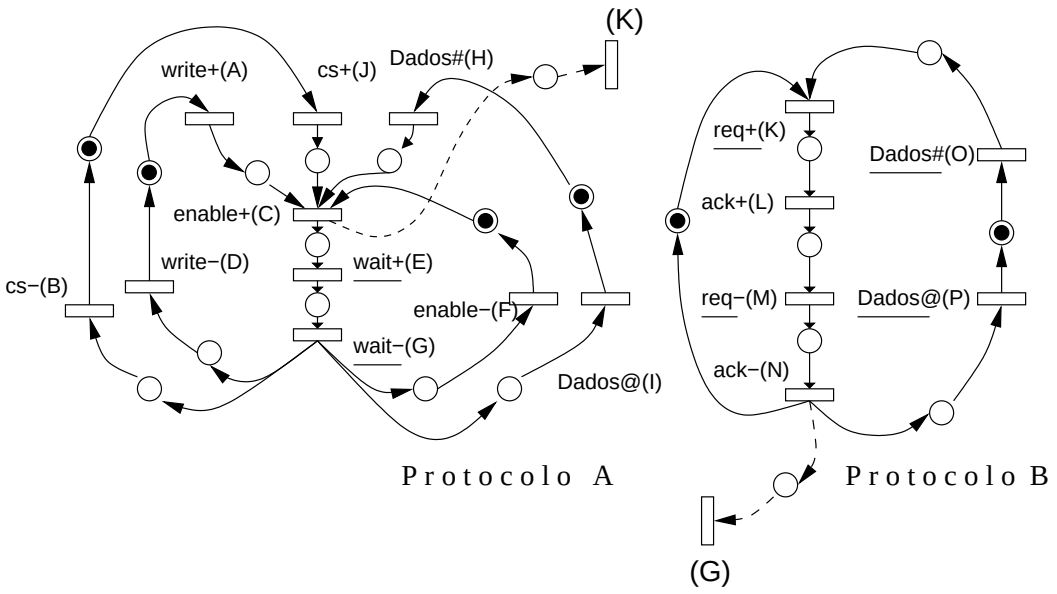


Figura 4.8: Modificações para inserção dos pontos de sincronismo

Note que os conjuntos fonte de validação e receptor de invalidação são vazios para o protocolo B, pois a linha de dados neste protocolo é uma entrada. Da mesma forma, os conjuntos de fonte de invalidação e receptor de validação são vazios para o protocolo A, pois a linha de dados é apenas saída. Isto poderia não ser verdade, por exemplo, se a operação descrita fosse de leitura, e não de escrita.

A inserção automática dos pontos de sincronismo gera (para a maioria dos casos) uma solução implementável, sem maiores ajustes, no entanto, esta solução pode não ser ótima, ou seja, podem existir pontos de sincronismo que levam a construção de um sistema com menor área e/ou maior velocidade.

É preferível, portanto, a indicação das relações de causalidade na especificação

inicial. É importante frisar, no entanto, que as estas indicações prévias não podem violar as propriedades já discutidas anteriormente.

4.4.2 Espelhamento

A operação de espelhamento parte da noção de que, uma vez que o processo de interface será interposto entre os dois módulos, seus sinais de entrada serão sinais de saída dos módulos e, analogamente, seus sinais de saída corresponderão a sinais de entrada nos módulos.

Sendo assim, a operação de espelhamento transforma os sinais marcados como entrada em sinais de saída, e vice-versa. Assim podemos definir:

Definition 4.4.6 [12]Seja um STG $G_1 = \langle N_1^*, S_1, \lambda_1 \rangle$, tal que S_{1I} é o conjunto de sinais de entrada e S_{1O} é o conjunto de saídas de G_1. O **espelhamento** de G_1, denotada por $espelhamento\,(G_1)$ é o mesmo STG mas com S_{1O} como conjunto de sinais de entrada e S_{1I} como conjunto de sinais de saída.

A operação de espelhamento deve ser aplicada após as etapas de renomeação e modificação dos protocolos padronizados. Como exemplo, a figura 4.9 mostra o espelhamento do protocolo (já devidamente trabalhado) do módulo B.

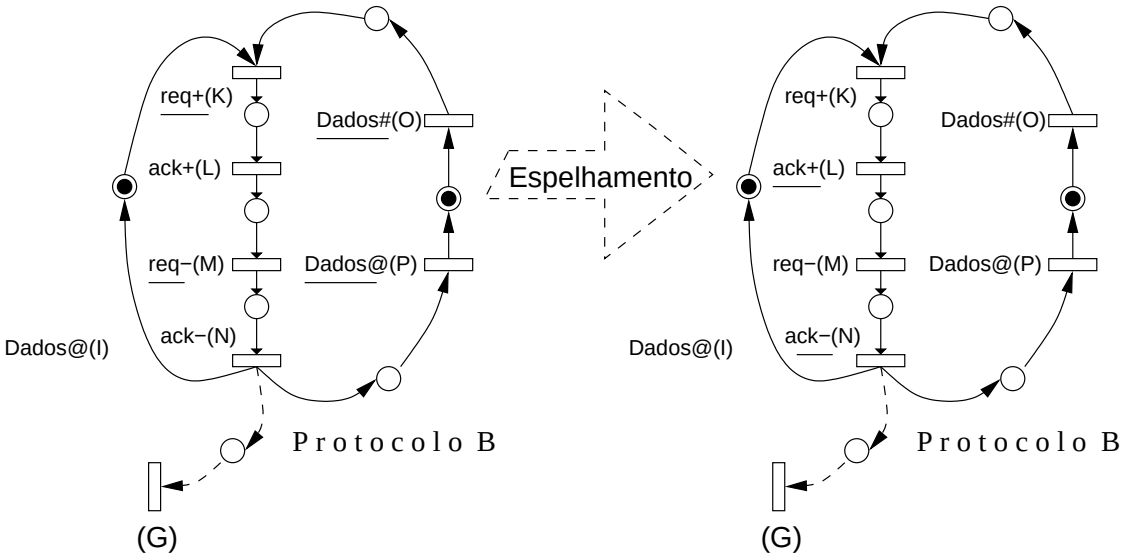


Figura 4.9: Espelhamento

Fizemos referência anteriormente, que operações de escolhas somente poderiam

ser modelados (nos protocolos individuais dos módulos) em sinais de "saída". Ao executar a operação de espelhamento estas escolhas passam, então, a serem resolvidas em sinais de "entrada" do processo de interface. Isto garante a propriedade de *output-persistency*[17] para a rede de Petri que modela o processo de interface.

A figura 4.10 mostra o processo de interface obtido do exemplo da figura 4.2 após o espelhamento. Note que a figura não mostra mais as operações sobre os dados. A razão para isto é simples: o processo de interface não precisa mais das informações sobre os sinais de dados. A eliminação destes caminhos é opcional, contudo, facilita o entendimento e as operações futuras, incorrendo, frequentemente, em soluções melhores.

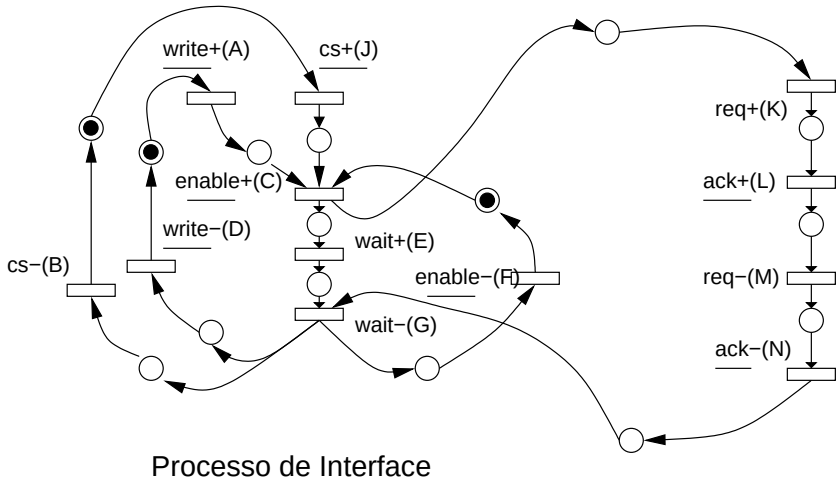


Figura 4.10: Espelhamento do processo obtido

4.4.3 Complete State Encoding

Uma vez modelado, o processo de interface pode ser utilizado para sintetizar o circuito lógico a ser interposto entre os módulos comunicantes, ou ainda, gerar código em uma linguagem de descrição de hardware (VHDL, Verilog) [4] ou de sistema (ex. SystemC).

Antes porém que este passo possa ser efetuado, faz-se necessário resolver os problemas de *complete state encoding*[18] encontrados no grafo de alcançabilidade da

[17] A noção de *output-persistency* foi introduzida na definição 3.2.3.
[18] O problema da codificação completa de estados foi discutido nas seções 3.3 e 3.3.1.

rede que representa o processo de interface. A existência de uma codificação binária completa para os estados é condição necessária para a síntese lógica.

O problema de *CSC* pode ser eficientemente resolvido através do particionamento do conjunto de estados em regiões [58] [14]. Este particionamente é realizado de forma a separar os estados conflitantes. Um sinal adicional é adequadamente inserido de forma a criar uma distinção entre os estados de diferentes regiões. A figura 4.11 ilustra este conceito. Os estados da região A serão diferenciados dos estados da região B, através do sinal *csc* inserido entre estas.

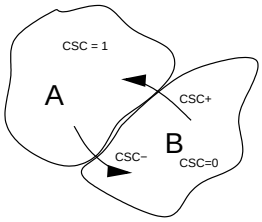


Figura 4.11: O sinal inserido diferencia estados em regiões distintas.

A técnica de inserção de sinais utilizada não altera as propriedades iniciais da rede e preserva "observacionalmente" o seu comportamento, ou seja, a rede obtida após a inserção de um sinal é **observacionalmente equivalente**, considerando os sinais de saída, com a rede anterior. Esta técnica deve ser recursivamente aplicada, até que todos os conflitos estejam resolvidos.

A figura 4.12 mostra a rede final, obtida após a resolução dos problemas de *CSC* no modelo do processo de interface do exemplo proposto.

4.4.4 Suporte à automação

A ferramenta de CAD implementa todos os passos da síntese de processo de interface, incluindo a resolução de problemas de *CSC*. Estas atividades são executadas na seguinte ordem:

1. Os protocolos padronizados são renomeados, segundo detalhado na seção **??**;

2. Se existirem indicações dos eventos em que ocorrerá o sincronismo entre os protocolos, a ferramenta utiliza-os na modificação da rede para estabelecer as

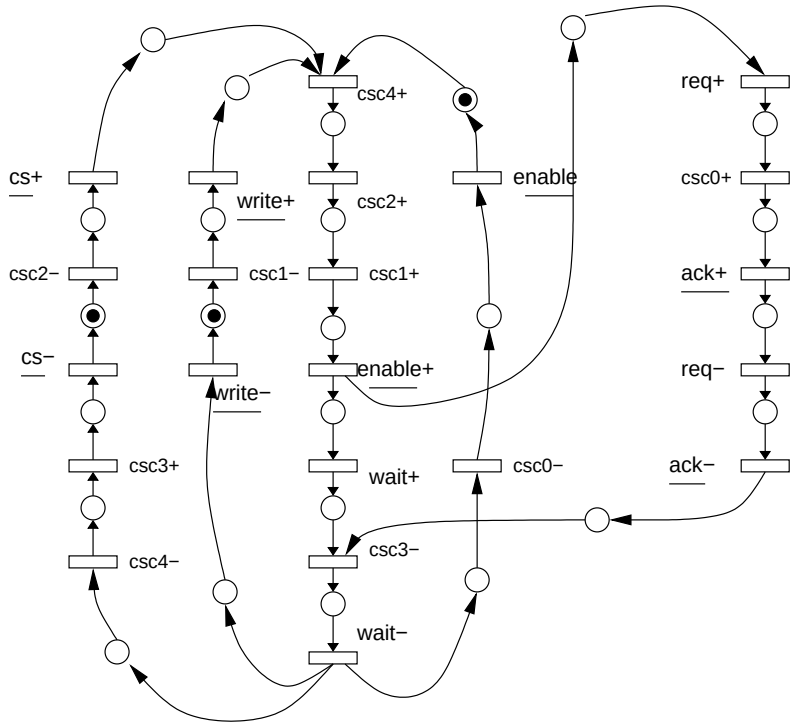


Figura 4.12: Processo de Interface modificado para resolução de *CSC*

relações de causalidade. Caso não existam indicações, o algoritmo utilizado é o do processo automático de identificação destes pontos (seção 4.4.1).

3. Espelhamento (seção 4.4.2).

4. Composição paralela e simplificação da rede (seção **??**).

5. Resolução dos problemas de *CSC*. A ferramenta de CAD pode não encontrar soluções para todos os conflitos se houverem eventos nos sinais envolvidos em uma escolha. Estes eventos (transições do *STG*) não podem ser utilizados para inserção de novos sinais, ou haveria violação da propriedade de *output-persistency*. No entanto, são raros os casos em que os conflitos não podem ser resolvidos, e adicionalmente, o algoritmo sempre converge para uma solução, caso não existam escolhas no protocolo.

Desta forma, a ferramenta de CAD provê automação para estas atividades, que podem ser altamente cansativas e susceptíveis a erros se elaboradas manualmente.

4.5 Síntese de Código VHDL

A etapa final desta metodologia é a geração de código sintetizável VHDL, a partir do modelo do processo de interface. Para o objetivo de incorporação rápida de módulos ao sistema, a geração de código sintetizável de alto nível é mais interessante pois:

- **É independente da plataforma de implementação.** De fato, gerar a *netlist* de um circuito lógico específico implicaria em incluir detalhes da arquitetura alvo do dispositivo onde o sistema seria implementado. A abordagem de uma linguagem em alto nível torna a descrição deste circuito independente, sendo responsabilidade da ferramenta de síntese, realizar o mapeamento tecnológico.

- **É de fácil incorporação em fluxos de projeto.** O código VHDL gerado pode ser utilizado para conectar módulos em uma abordagem estrutural do sistema. Neste sentido, o projetista pode, por exemplo, unir blocos com diferentes níveis de abstração (ex.: *firmware_ cores* com um componente encriptado).

A geração de código VHDL, no entanto, não pode seguir à risca os métodos tradicionais de síntese lógica a partir de STG [41] [40] [59], pois estes, frequentemente visam operações de minimização booleana. Da mesma forma, os resultados obtidos através de traduções estruturais da rede [60], podem levar a soluções de elevado custo em termos de área e tempo.

Introduzi, portanto, um novo método, baseado na síntese de circuitos assíncronos da classe *speed-independent* [41], mas que não visa o mapeamento tecnológico e sim um modelo estrutural. O código VHDL gerado a partir deste método descreve um modelo composto de três blocos de circuito, conforme ilustrado na figura 4.13. O primeiro bloco (a) mapeia o estado atual do sistema (composto pelos sinais de entrada, sinais de saída e sinais internos) em um vetor binário baseado na rede de Petri que descreve o processo de interface. Este vetor binário é entrada para bloco de lógica com dois níveis (soma de produtos) que realiza as funções de excitação (*set* e *reset*) de *latches*. Os *latches*, por sua vez, constituem a terceira camada, ou bloco, e suas saídas são os sinais de saída e internos do processo de interface.

Antes de prosseguir, convém definir os seguintes conceitos:

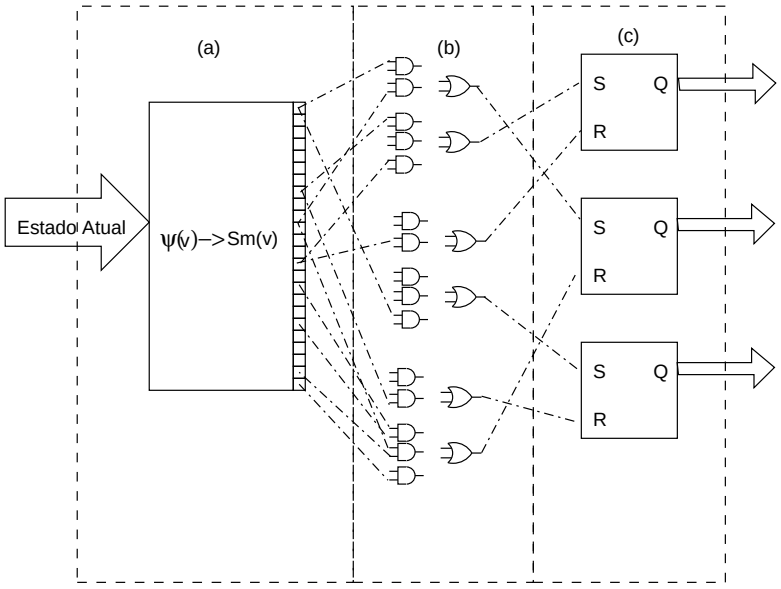


Figura 4.13: Modelo do circuito descrito em VHDL

Definition 4.5.1 Dado um *STG* $G_1 = \langle N_1^*, S_1, \lambda_1 \rangle$, denomina-se **grafo de alcançabilidade codificado,** à tupla $A(G_1) = \langle V, E, \psi \rangle$, onde V é o conjunto de estados do grafo de alcançabilidade de G_1; E é o conjunto de transições entre os estados do grafo; e $\psi : V \rightarrow \{0,1\}^{\sharp s}$, é uma função que atribui a cada estado $v \in V$ um vetor binário, de tamanho $\sharp s$, que representa o valor dos sinais de entrada, saída e internos, naquele estado.

Definition 4.5.2 Dado um STG $G_1 = \langle N_1^*, S_1, \lambda_1 \rangle$ e seu grafo de alcançabilidade, denomina-se **função de marcação dos estado,** à função $S_m : V \rightarrow [Mo >$ que mapeia cada possível estado do grafo de volta à sua marcação equivalente M no *STG*.

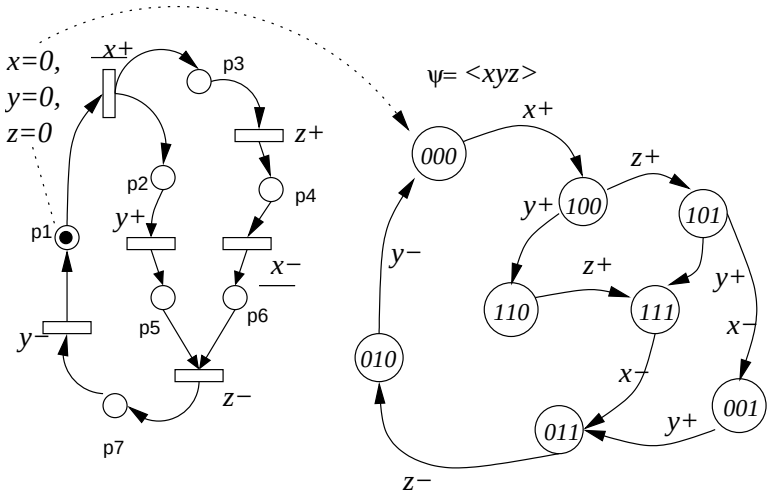
A idéia do primeiro bloco é transformar a codificação binária do grafo de alcançabilidade codificado em outro vetor binário que representa a marcação da rede de Petri. Isto é realizado, a partir do grafo de alcançabilidade, da seguinte forma:

1. Transformar o grafo de alcançabilidade em um grafo codificado, calculando para cada estado $v \in V$, a função $\psi\,(v)$. Note que os problemas de codificação de estado não podem mais existir. Este procedimento é similar ao que foi detalhado na seção 3.3.

2. Calcular, para cada estado $v \in V$ do grafo de alcançabilidade codificado, a função de marcação deste estado $S_m(v)$, representando cada marcação como um vetor binário. Cada posição deste vetor representa um lugar da rede de Petri, e recebe 1 se e somente se o número de marcas neste lugar é diferente de zero. E recebe zero, caso contrário.

3. Construir um mapeamento tal que $\psi(v) \rightarrow S_m(v)$.

Este mapeamento é facilmente codificado em VHDL através de um sinal selecionado. A figura 4.14 exemplifica a construção do primeiro bloco para a rede modelada no exemplo dado no capítulo anterior, figura 3.16.



ψ= <xyz>	Sm = <p1p2p3p4p5p6p7>
000	1000000
100	0110000
101	0101000
110	0010100
111	0001100
001	0100010
011	0000110
010	0000001

```
with      select
  Sm  <=     "    1000000" when "000"
             "0110000" when "100",
             "0101000" when "101",
             "0010100" when "110",
             "0001100" when "111",
             "0100010" when "001",
             "0000110" when "011",
             "0000001" when "010",
             "0000000" when others;
```

Figura 4.14: Mapeamento $\psi(v) \rightarrow S_m(v)$ e descrição em VHDL

O segundo bloco implementa as funções de excitação dos *latches* de saída, a partir do vetor de marcação de estado (PSV) gerado pelo primeiro bloco. Pela semântica das redes de Petri, sabemos que cada transição (e consequentemente o evento atribuído a ela) somente estará habilitado a ocorrer quando todos os lugares

de entrada contiverem ao menos uma marca. Desta forma, o segundo bloco pode ser montado da seguinte maneira:

1. Para cada transição nomeada com um sinal de saída ou interno, fazer um *AND* lógico dos bits do *PSV* que representam os lugares de entrada desta transição.

2. Para cada evento $x*$ em um sinal de saída ou interno, fazer um *OU* lógico dos sinais obtidos no passo anterior que representam transições nomeadas como $x*$.

Ao final deste passo, para cada sinal x de saída ou interno, temos duas funções C_{set} e C_{reset} que realizam o seguinte comportamento:

$$C_{set}(x) = \begin{cases} 1 & se\ o\ evento\ x+\ esta\ habilitado \\ 0 & caso\ contrario \end{cases}$$

$$C_{reset}(x) = \begin{cases} 1 & se\ o\ evento\ x-\ esta\ habilitado \\ 0 & caso\ contrario \end{cases}$$

A terceira e última camada é implementada somente com *latches* ou alternativamente com elementos C, especiais para sistemas assíncronos. Os *latches* devem ser assíncronos e com *delays* internos desprezíveis. Assim, podemos entendê-los como componentes atômicos e livres de *hazards*. Adicionalmente, caso sejam excitados com os sinais *set=1* e *reset=1* simultaneamente, devem apresentar a saída igual a zero (*reset-dominant latch*) ou igual a um (*set dominant latch*) bem definidos, não havendo instabilidade.

Para cada sinal x de saída ou interno é atribuído um *latch*, e as funções $C_{set}(x)$ e $C_{reset}(x)$ são ligadas de forma a excitar suas entradas de *set* e *reset* respectivamente. A saída deste *latch* é o próprio sinal de saída x.

Se corretamente construído, o processo de interface modela um sistema onde a condição $C_{set} = C_{reset} = 1$ nunca ocorre. Para cada *latch* de saída, os possíveis estados de excitação são $\{C_{set} = C_{reset} = 0\} \vee \{C_{set} = 0, C_{reset} = 1\} \vee \{C_{set} = 1, C_{reset} = 0\}$. Mais ainda, as transições de estado de excitação

$$\{C_{set} = 1, C_{reset} = 0\} \leftrightarrow \{C_{set} = 0, C_{reset} = 1\}$$

são proibidas, ou seja, somente são permitidas as transições $\{C_{set} = 1, C_{reset} = 0\} \leftrightarrow \{C_{set} = 0, C_{reset} = 0\}$ ou $\{C_{set} = 0, C_{reset} = 0\} \leftrightarrow \{C_{set} = 0, C_{reset} = 1\}$.

A figura 4.15 ilustra a construção conjunta das três camadas.

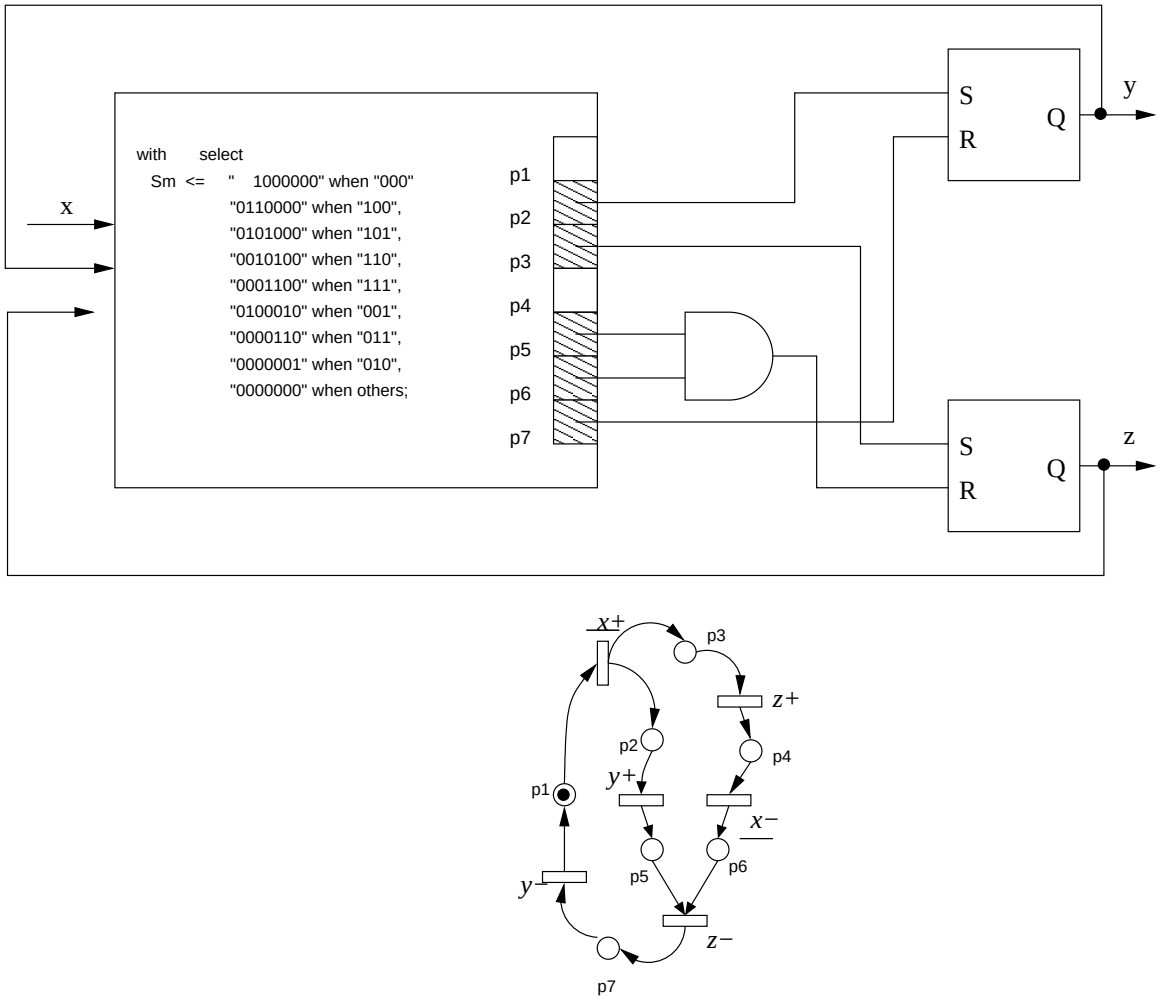


Figura 4.15: Modelo final do circuito a ser descrito em VHDL para a rede *xyz*.

O código VHDL gerado para o exemplo proposto neste capítulo, figura 4.2, pode ser visto no apêndice A.

4.5.1 Geração Automática de Código VHDL

A ferramenta de CAD implementada neste trabalho realiza o procedimento exposto na seção 4.5 de forma automática e sintetiza por completo o arquivo com o código VHDL (extensão ".vhd"). Este arquivo pode ser utilizado para interligar, de forma estrutural, os módulos comunicantes.

Os *latches* são descritos, neste arquivo, como componentes fixos de uma biblioteca, facilitando à ferramenta de síntese futura, identificá-lo como um componente atômico. É possível que sejam necessárias mudanças nesta descrição dependentes das necessidades da ferramenta de síntese a ser utilizada posteriormente.

O código VHDL gerado segue padrão 87 {ref].

4.6 Resumo

Neste capítulo, foi apresentada a metodologia para síntese do processo de interface, desenvolvida em quatro etapas principais, ilustrada através de um exemplo. A primeira etapa diz respeito à especificação dos protocolos individuais dos módulos a serem integrados. Esta especificação deve ser refinada até descrever o protocolo como um conjunto de eventos sobre os sinais da interface. Mostru-se como fazer isto para diagramas temporais, atribuindo a estes marcas que expressam relações de causalidade entre os eventos.

A segunda etapa é responsável pela tradução destes protocolos para um formato intermediário em Redes de Petri, largamente conhecido como *STG*. Com base neste formato intermediário podem ser verificadas condições necessárias e suficientes para a síntese do processo de interface.

Na terceira etapa o procedimento de síntese em si é realizado. Esta fase tem como entrada dois protocolos padronizados, descritos com *STGs* e tem como saída uma única rede de Petri que descreve o comportamento do processo de interface.

A quarta etapa descreve como o modelo do processo de interface é traduzido para uma codificação em VHDL.

A seguir, os detalhes de implementação da ferramenta de CAD, fruto deste trabalho, são apresentados, bem como um estudo de caso que valida as idéias expostas aqui.

Capítulo 5

A ferramenta de CAD - CoreBond

Este capítulo detalha o trabalho de implementação da ferramenta de CAD CoreBond para geração automática do processo de interface de *IP-Cores* em arquiteturas SoC. No capítulo 4 foram discutidas algumas funcionalidades implementadas na ferramenta com base nas diversas etapas de síntese da interface. Neste, será detalhada sua interface com usuário, formatos dos arquivos de entrada e saída e estrutura interna.

5.1 Interface com o usuário

A interface com o usuário da ferramenta CoreBond visa o acompanhamento progressivo, por parte do projetista, das atividades propostas na metodologia. O projetista pode, desta forma, identificar com facilidade em que etapa encontra-se o processo de síntese, bem como sanar eventuais problemas.

A figura 5.1 mostra a tela principal da ferramenta. A interface com o usuário é dividida em dois blocos principais: o painel de atividades e o painel de mensagens.

No painel de atividades está ilustrado o fluxo de atividades a serem desenvolvidas durante o processo de síntese da interface. Inicialmente, todos os ícones que formam o painel estão desabilitados, com uma coloração opaca. A única excessão são os ícones de "Protocolos iniciais", que encontram-se ativos(coloração viva). Quando desabilitados, os ícones não apresentam reatividade, ou seja, um clique de *mouse* nestes ícones não dispara nenhuma ação.

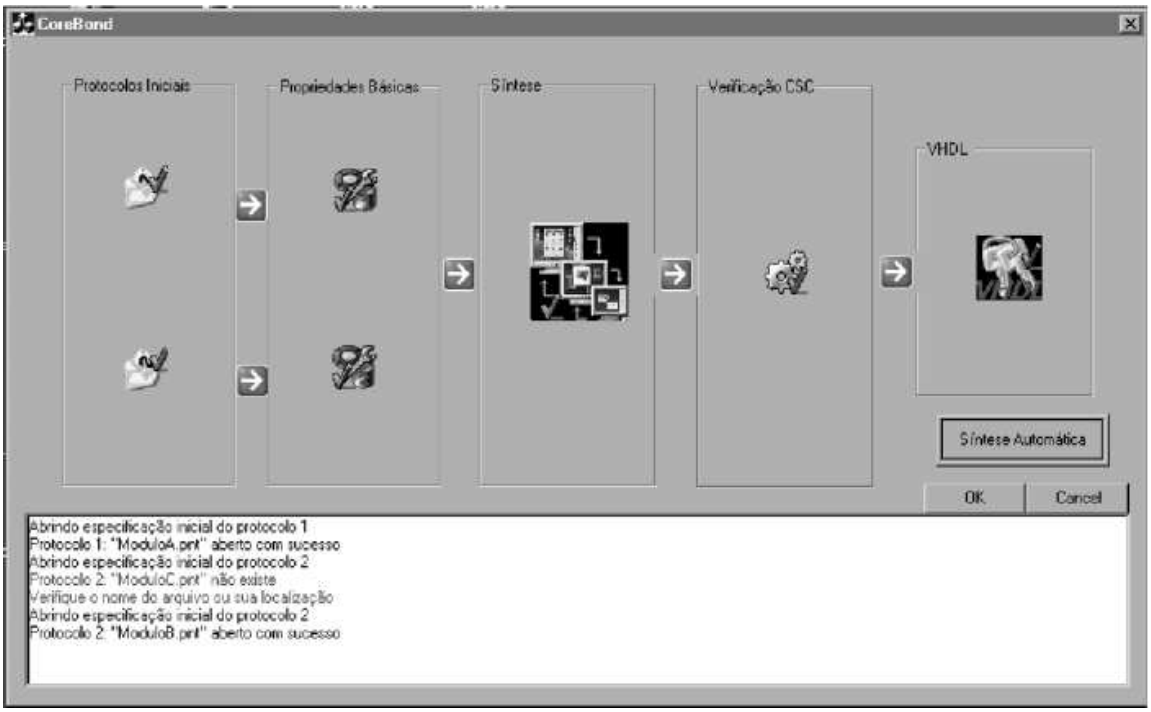


Figura 5.1: Tela do programa CoreBond

O painel de mensagens encontra-se imediatamente abaixo do painel de atividades. Sua função é prover informações sobre os procedimentos internos da ferramenta, em uma linguagem acessível ao usuário. À medida que o processo de síntese é realizado, resultados de testes, verificações, ações e status do sistema são informados no painel de mensagens.

Adicionalmente, um *menu* é oferecido na parte superior esquerda da tela, para as ações mais básicas de abertura e salvamento de arquivos.

5.1.1 O Painel de Atividades

O painel de atividades é o meio através do qual o usuário do CoreBond interage com o sistema, abrindo arquivos, iniciando ações e requisitando informações sobre o processo de síntese. No painel de atividades está desenhado um fluxograma das etapas envolvidas no processo de síntese. Este fluxograma é feito de pequenos ícones que guiarão o usuário através das operações.

Inicialmente, apenas os dois ícones de "Protocolos iniciais" estão habilitados (em cores vivas). Um clique de *mouse* sobre eles ativará a janela de abertura dos arquivos

de entrada do sistema. Uma vez abertos, e validados os dois arquivos de entrada, o sistema mostra o ícone de verificação de propriedades ativo[1] (figura 5.2). Adicionalmente, uma marca $\surd$ aparece ao lado dos ícones cuja operação já foi completada com sucesso.

Caso os arquivos lidos não sejam válidos (não formatados corretamente, por exemplo), mensagens de erro serão emitidas no painel de mensagens e possíveis encaminhamentos para o contorno adequado do problema.

De forma similar, à medida que cada operação é realizada, novos ícones começam a ser habilitados. Esta abordagem progressiva é interessante pois orienta o projetista nos passos a serem desenvolvidos, além de fornecer uma noção da metodologia por trás da ferramenta. O uso repetitivo deste procedimento, no entanto, poderia ser enfadonho, uma vez que todas estas atividades podem ser feitas automaticamente. Para tal caso, o usuário pode utilizar o ícone "Síntese Automática" no canto inferior direito do painel de atividades.

Este ícone dispara automaticamente, e em ordem adequada, todos os outros. Se erros ocorrerem no processo, este pára as atividades e espera por uma ação do usuário.

5.1.2 O Painel de Mensagens

O painel de mensagens é o meio através do qual o programa CoreBond informa ao usuário todos os procedimentos que estão sendo realizados internamente. No painel de mensagens são informados ainda mensagens de erros, status do sistema e orientações para procedimentos posteriores.

Desta forma existem quatro categorias de mensagens e o CoreBond adota um sistema de cores para cada tipo de mensagem:

Azul São mensagens de procedimento. Indicam atividades que estão ocorrendo na ferramenta de CAD.

[1]Note também que, no painel de mensagens, diversas indicações desta operação estão aparecendo.

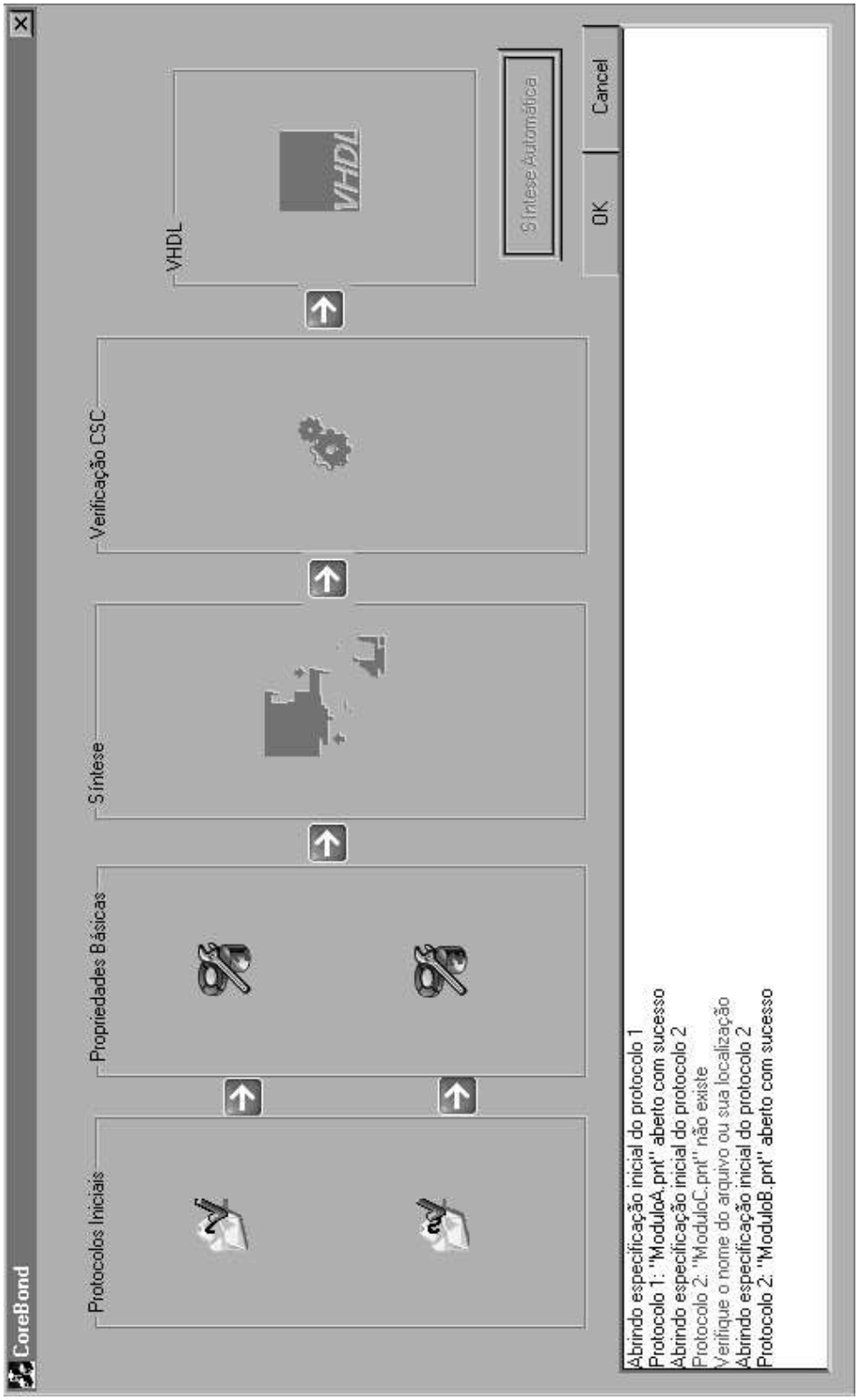


Figura 5.2: CoreBond. Atividades de abertura de arquivos completadas

Preto São mensagens de status. Indicam a finalização de tarefas e o resultado

destas.

Vermelho São mensagens de erro. Indicam que uma falha foi detectada no procedimento. Exemplos são: arquivos de entradas inválidos, propriedades necessárias não verificadas, entre outros.

Verde São mensagens de orientação. Indicam ao usuário o que fazer em caso de erros ou qual o próximo procedimento a ser tomado.

É possível, a qualquer momento, salvar o conteúdo do painel de mensagens. Para tal, basta clicar com o botão direito do mouse no painel e selecionar a opção "Salvar". O mesmo para limpar o conteúdo do painel.

Figura 5.3: Painel de Mensagens.

5.2 Formato dos arquivos de entrada e saída

Os arquivos de entrada do sistema descrevem os protocolos individuais dos módulos a serem interligados como um *STG*. O arquivo de saída do sistema é um arquivo escrito em VHDL, pronto para o processo de integração dos módulos.

O *STG,* descrito no arquivo de entrada, deve ser formatado como uma rede de Petri segundo um padrão INA modificado. O INA [61] (*Integrated Net Analyser*) é um programa para manipulação de redes de Petri do tipo *place-transition* bastante popular na comunidade acadêmica. Os arquivos de entrada do INA seguem um formato próprio, bastante simples, e que podem ser escritos em qualquer editor de texto comum. O uso do formato INA modificado oferece, ainda, outras vantagens: primeiro, os arquivos no novo formato ainda podem ser utilizados pelo INA, permitindo ao projetista fazer outros tipos de teste na rede de Petri do protocolo. Segundo, existem diversos editores gráficos de rede de Petri que exportam as redes

graficamente descritas para o formato INA, facilitando a edição dos protocolos pelo projetista.

Para descrever os eventos atribuídos a cada transição do *STG*, contudo, foi necessário propor uma pequena modificação nestes formatos. A alteração consistiu na inserção de uma nova seção que descreve os sinais do *STG*, seus nomes, natureza, direção e estado inicial. A descrição final do INA modificado está exposto no apêndice B1, no formato *Backus-Nauer Form* extendido (EBNF).

O arquivo de saída da ferramenta é escrito em código VHDL'93 [62] [3] [63] composto de componentes descritos de forma comportamental e estrutural. A parte comportamental do código descreve os blocos de conversão do estado atual para a marcação da rede de Petri[2] e as funções de excitação dos *latches* de saída. O componente estrutural do código VHDL final representa os *latches* de cada sinal de saída e interno. Pode ser interessante manter esta camada descrita de forma estrutural com o intuito de facilitar, à ferramenta de síntese, uma identificação deste elemento, como discutido anteriormente na seção 4.5.1.

Um exemplo completo do arquivo VHDL gerado pode ser observado no apêndice A. O código deste apêndice refere-se ao processo de interface gerado ao longo do exemplo no capítulo 4.

O arquivo de saída pode ser utilizado, sem necessidade de modificações adicionais, para integrar os dois módulos comunicantes, de uma forma estrutural. Durante a integração, no entanto, é necessário ainda ligar cada sinal de saída e interno, presente nos *ports* do processo de interface, a seu sinal de realimentação na entrada. Esta realimentação é necessária pois o estado futuro é função também das saídas e sinais internos atuais. Estes sinais de realimentação (entradas) são facilmente reconhecidos pois são homônimos ao sinal de saída correspondente, mais com um prefixo "f_" .

A seguir, será explicado com maior detalhes, a organização interna da ferramenta de CAD, quais seus componentes principais e como interagem entre si.

[2]Ver capítulo 4.

5.3 Estrutura Interna da ferramenta de CAD

A ferramenta de CAD foi implementada utilizando a linguagem C++ [64] em uma abordagem orientada a objetos [65]. O programa é dividido em pacotes de classes, onde cada pacote implementa um conjunto de conceitos necessários à execução da metodologia proposta. Posteriormente, estes pacotes são organizados em uma estrura de camadas que coordena o disparo de tarefas e realiza a interface com o usuário.

As classes do programa foram divididas nos seguintes pacotes:

PNKernelC++ Descreve uma estrutura de classes que modela todos os conceitos de uma rede de Petri. Lugares, transições, sinais, *tokens* e semântica de disparo são apenas alguns exemplos de conceitos modelados por este conjunto de classes.

NetActions São classes que implementam ações e transformações sobre uma rede de Petri ou um *STG*.

InaParser Um parser para os arquivos de entrada da ferramenta, capaz de ler arquivos ".stg" e convertê-los no modelo interno com classes do pacote PNKernelC++ e vice-versa.

VhdFile Um montador de código VHDL, capaz de traduzir redes de Petri em formato *STG* para a linguagem de descrição de hardware.

CoverGraph Possui conceitos para implementação, operação e extração de resultados das árvores de cobertura e grafos de alcançabilidade. Trabalha em conjunto com o pacote NetActions, verificando propriedades das redes.

CSC Classes para resolução dos problemas de codificação completa de estados. Este pacote implementa os conceitos de regiões e inserção de sinais entre estas.

Signals Conjunto de classes que constitui o modelo para sinais de um *STG*. São extensões a serem acopladas às classes do PNKernelC++ para implementar o conceito de *STG*.

A estrutura de cada pacote pode ser observada no apêndice B2, como diagramas de classes em formato padrão *Universal Modelling Language* (UML) [65]. Neste apêndice é possível observar as relações de herança, associação, composição, etc. entre as classes de cada pacote. A seguir são descritos aspectos importantes para cada um deles.

5.3.1 Pacote PNKernelC++

O pacote **PNKernelC++** é uma tradução para o C++ do *Petri Net Kernel* [66] [67], um conjunto de classes, originalmente escrito em JAVA na Universidade de Berlin, Alemanha. A inteção principal do *Petri Net Kernel* é criar uma infra-estrutura básica padrão para implementação de aplicações baseados em redes de Petri. Para tal, o pacote original inclui um conjunto de classes que modelam os conceitos básicos como lugares, transições, arcos, entre outros. A este conjunto são adicionadas operações básicas, porém bastante flexíveis, sobre as redes ou seus elementos, como por exemplo, mecanismos para descrever a semântica de disparo.

Todo este conjunto foi elaborado para representar, de forma o mais genérico possível, redes de Petri. Em outras palavras, é independente do tipo de rede utilizado. Esta dissociação do tipo de rede é possível através do conceito de extensão. Segundo este conceito, cada elemento da rede (lugares e transições, por exemplo) contém um mecanismo para receber extensões. Estas extensões são classes definidas pelo usuário para modelar novos conceitos, e são adequadamente anexadas aos elementos já existentes no conjunto original.

A proposta do *Petri Net Kernel* então, é tornar-se uma biblioteca padrão e largamente utilizada para o desenvolvimento de aplicativos baseados em redes de Petri. Neste trabalho, as partes fundamentais do *Petri Net Kernel* foram traduzidas para o C++, faltando apenas suas interfaces gráficas. Ressaltamos a tradução desta biblioteca como uma contribuição secundária deste trabalho, uma vez que, a partir destas bibliotecas traduzidas, futuros trabalhos poderão ser desenvolvidos baseando-se no mesmo modelo, mesmo que não trabalhem com *STGs* especificamente.

O conceito de extensão foi então utilizado nas transições da rede, e na rede em si, para a introdução do conceito de sinais, presentes nos *STGs*. A figura 5.4 ilustra esta

relação. Os *tokens* são uma extensão natural dos lugares, pois os lugares "contém" *tokens*. Da mesma forma, para o modelo de *STG*, as redes contém sinais. Sobre estes sinais incidem ações, estas por sua vez associadas às transições.

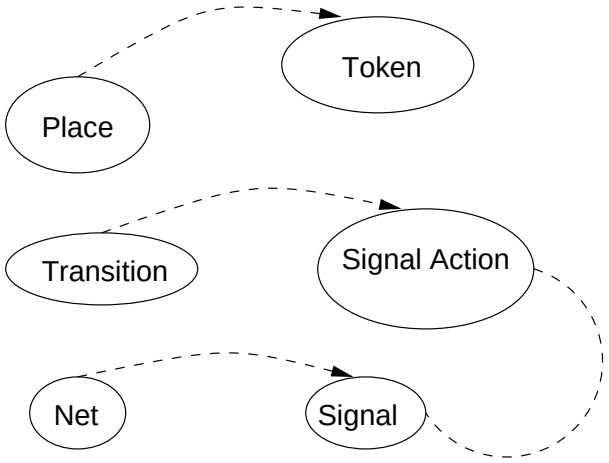


Figura 5.4: Extensão das transições para modelagem do conceito de sinais

5.3.2 NetActions

O pacote **NetActions** implementa ações completas realizadas em uma rede de Petri e que resultam, comumente, em outra rede, com uma nova configuração. São exemplos de ações sobre a rede: o disparo de um uma transição habilitada, renomeações de elementos da rede, verificação de propriedades estruturais[3] e composição paralela entre redes, entre outros.

As classes deste pacote recebem, no momento em que são instanciadas, uma referência à rede com a qual irão operar. Após a execução da ação descrita no código da classe esta rede estará adequadamente modificada. Um sistema de log padrão provê informações sobre as atividades desenvolvidas sobre a rede, detalhando possíveis erros ou resultados.

[3]Pode ser verificado, por exemplo, se uma rede é da classe Free-Choice somente observando sua estrutura.

5.3.3 InaParser

Como o próprio nome sugere, o pacote **InaParser** realiza a tradução dos arquivos de entrada, em formato INA modificado (ver seção 5.2) para o formato interno constituído de elementos do pacote PNKernelC++. Adicionalmente, este pacote também faz a tradução inversa, sendo possível, a qualquer momento do processo, gravar a rede atual em formato similar ao de entrada.

Esta é uma característica interessante, em especial para trabalhos de pesquisa futuros. O projetista poderia, por exemplo, gravar a rede obtida logo após a síntese do processo de interface, ainda com problemas de CSC, para estudar melhores soluções. Poderia ainda, por exemplo, a partir da rede final do processo (imediatamente antes de transcrevê-la em VHDL) simular o comportamento do processo de interface em um programa para simulação de redes de Petri.

5.3.4 VhdFile

O pacote **VhdFile** contém os conceitos de tradução do formato intermediário em redes de Petri (*STG*) para VHDL'93. Suas classes são organizadas de forma a montar a estrutura do arquivo de saída em seções. Desta forma, existem classes para a declaração da entidade, declaração de sinais, declaração de componentes, arquitetura, entre outras.

O arquivo VHDL gerado admite a existência de um componente externo, arbitrariamente chamado de *speedIndependent_latch*. Este componente deve ser um latch atômico e o projetista deve indicar à ferramenta de síntese, se necessário, esta informação. Para a plataforma do QUARTUS II [68], da Altera, utilizado neste trabalho, foi criado um componente externo com o mesmo nome (*speedIndependent_latch*) através de recursos do próprio programa (*megafunctions*). No processo de síntese, o QUARTUS infere este componente como um latch atômico e o constrói a partir de um *flip-flop* com entradas assíncronas de *set* e *reset* já presentes na arquitetura do dispositivo alvo.

5.3.5 CoverGraph

A montagem das árvore de cobertura e grafos de alcançabilidade, bem como os testes de verificação de propriedades são feitas nas classes do pacote **CoverGraph**, em conjunto com o pagote NetActions. A construção da árvore de cobertura segue o algoritmo explicado em [44], bem como a verificação de propriedades na rede.

5.3.6 CSC

Todo o procedimento para resolução dos problemas de CSC são encaminhados pelas classes do pacote **CSC**. As classes implementam a resolução de CSC através da teoria de regiões abordada em [58] e [42], já largamente discutida anteriormente.

5.3.7 Signals

O pacote **Signals** é complementar ao PNKernelC++, extendendo-o para representar o conceito de sinais e eventos. Estes conceitos são importantes para modelar *STGs* como redes de Petri. O disparo de transições, por exemplo, foi extendido para implementar também um evento sobre um sinal. Este evento modifica o estado do sinal.

A rede ganha, então, uma segunda codificação, que atribui a cada marcação da rede um vetor binário que expressa o estado dos sinais naquela marcação.

5.4 Resumo

Neste capítulo foram expostos os detalhes de implementação da ferramenta de CAD - CoreBond, sua interface gráfica com o usuário, os formatos dos arquivos de entrada e saída e a estrutura interna dos pacotes que compõem a ferramenta.

Capítulo 6

Incorporando um *MAC* AMBA no sistema NIOS - Estudo de Caso

No intuito de validar as idéias expostas ao longo deste trabalho, elaboramos como estudo de caso a incorporação de um módulo de *IP-Core* a um ambiente SoC baseado no processador NIOS [15] [69]. O *IP-Core* a ser incorporado é um circuito *Multiply-And-Accumulate* (*MAC*), projetado com uma interface preparada para o barramento padrão AMBA [17], denominamos então como *MAC-AMBA*. O sistema alvo, onde o *MAC-AMBA* será incorporado, é um sistema processador/periféricos completo baseado na plataforma Excalibur Nios, da Altera, e que é montado em um barramento padrão AVALON [70].

Este estudo de caso é centrado, especialmente, nas interfaces a serem incorporadas e não na aplicação final que será atribuída ao *MAC*. De fato, a funcionalidade dos módulos comunicantes não são importantes, *per si*, para este trabalho. Os padrões de interface a serem interligados são largamente utilizados no mercado e pode-se afirmar que exibem um nível de "razoável" complexidade, devido ao número de sinais envolvidos, às suas características de alta performance e especificações temporais rígidas. Pode-se então defender que este estudo de caso é significativo para os objetivos deste trabalho, pois:

- Os padrões de barramento AVALON e AMBA estão estabelecidos entre a

comunidade científica e industrial[1], sendo reconhecidos como exemplos típicos de barramentos para ambientes SoC. Esta característica está de acordo com o objetivo deste trabalho à medida que este visa a incorporação rápida de módulos neste tipo de ambiente.

- Estes padrões têm ainda características próprias de sistemas de alta velocidade, com relativa complexidade em suas interfaces. Este aspecto é particularmente importante quando compara-se este trabalho com outros com a mesma finalidade [11] [22] [10], e que exemplificam a interligação de protocolos bastante elementares, como os protocolos assíncronos tipo *NRZ* e *RZ*.

- O uso de um *MAC* como periférico somente faz sentido se os custos de comunicação forem baixos. Este aspecto enriquece a solução alcançada à medida em que esta mostra não comprometer o uso deste módulo. Um aumento elevado do custo de comunicação, introduzido pelo processo de interface, pode comprometer o uso de determinado *IP-Core* e dificilmente seria aceitável para este caso.

Nas **seções** seguintes detalhamos o *IP-Core* do *MAC* a ser incorporado, os padrões de barramentos AVALON e AMBA, o sistema NIOS, onde o módulo será integrado e, por fim, os resultados obtidos. É importante ressaltar ainda algumas dificuldades encontradas na incorporação, em especial àquelas referentes à implementação da solução em um ambiente reconfigurável.

6.1 O módulo *Multiply-And-Accumulate*

Como já citado anteriormente, o módulo a ser incorporado ao sistema neste estudo de caso é um *Multiply-And-Accumulate (MAC)*. Este, por sua vez, é um circuito bastante simples, mas que encontra aplicações em diversos sistemas práticos, como tratamento de sinais e imagens, médias ponderadas, circuitos de redes neurais em hardware, multiplicação vetorial, entre outros.

[1]Exemplos: Virtual IP Group (http://www.virtualipgroup.com), European Space Agency (http://www.estec.esa.int/microelectronics), inSilicon Inc. (http://www.insilicon.com), entre outros.

Em muitos sistemas o *MAC* é disponibilizado como um recurso do processador (ou co-processador). Neste trabalho, ele será integrado ao sistema como um periférico diretamente ligado ao processador principal. Sua estrutura interna pode ser vista na figura 6.1.

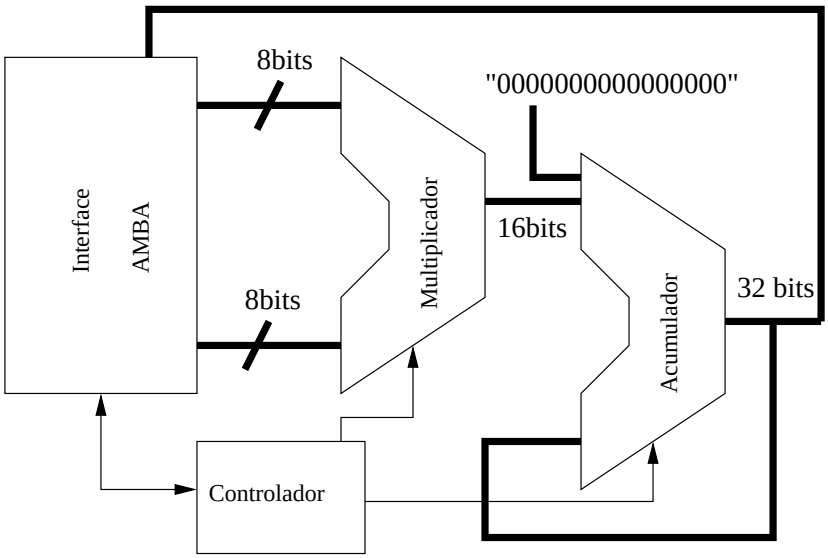


Figura 6.1: Estrutura Interna do *MAC*

O módulo recebe como entrada dois operandos de 8 bits, realiza o produto entre eles e soma o resultado com o valor já acumulado em um registrador de 32 bits. Para uma operação de escrita, o processador envia uma palavra de 32 bits, onde os dois bytes menos significativos representam os dois operandos de entrada. Para uma operaçao de leitura o processador captura o conteúdo do registrador-acumulador de 32 bits.

A interface do *MAC* segue o padrão AMBA, com os sinais necessários para executar as operações de leitura e escrita, selecionar o módulo e indicar a validade do dado no barramento. As operações realizadas devem ser atômicas, ou seja, apenas um dado é trafegado por operação, não sendo implementado o modo de transmissão contínua, por motivos já discutidos.

A figura 6.2 mostra os sinais presentes na interface padrão AMBA do *MAC*.

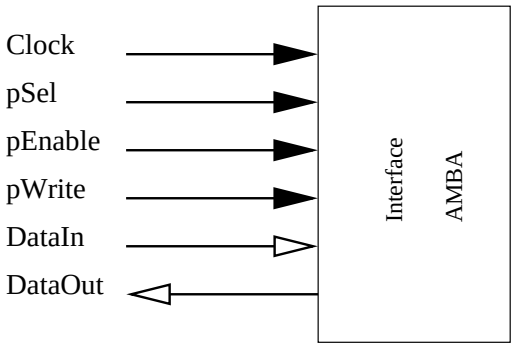


Figura 6.2: Interface AMBA utilizada no *MAC*

6.2 O padrão de barramento AMBA

O padrão de barramento AMBA [17] foi desenvolvido pela ARM, e sua sigla significa *Advanced Microcontroller Bus Architecture*. Em verdade, consiste em uma proposta de arquitetura de barramento composta de dois níveis: o *Advanced High-Performance Bus(AHB)* e o *Advanced Peripheral Bus*. Seu esquema geral pode ser visto na figura 6.3.

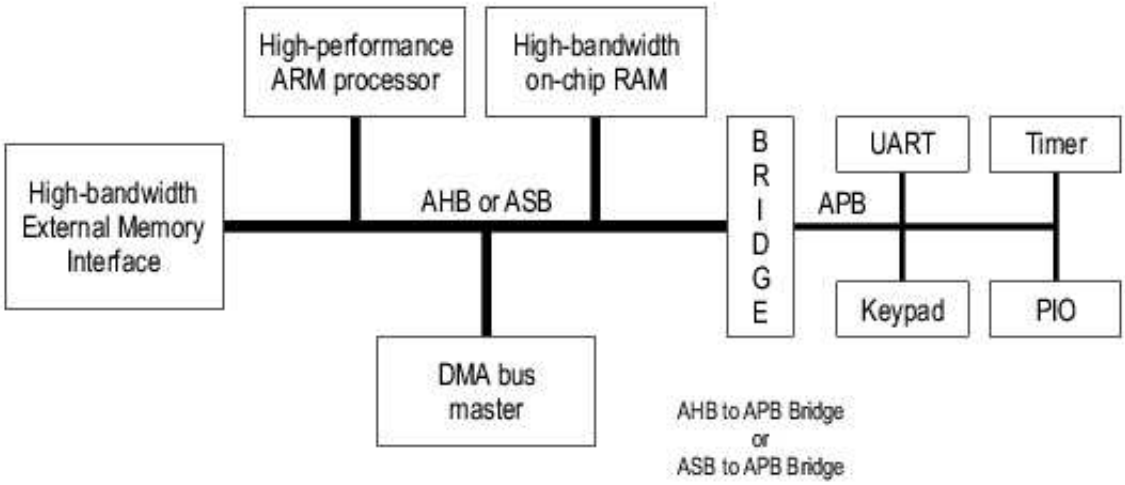


Figura 6.3: Arquitetura do barramento AMBA

O primeiro nível (*AHB)* é destinado a módulos de alta performance e com grande tráfego de dados com o processador, como por exemplo, memórias e DMAs. O segundo nível (*APB*) tem como finalidade a incorporação de módulos periféricos de mais baixa velocidade. Apresenta, no entanto, vantagens de consumo de energia em relação ao *AHB*, além de ter uma versão mais simples de interface. A finalidade do

padrão AMBA é [17]:

- Facilitar o desenvolvimento de sistemas *on-chip*;
- Ser independente da tecnologia alvo de implementação, garantindo o máximo de reusabilidade para barramento e periféricos ao longo de diversos processos de fabricação de circuito integrado;
- Encorajar o projeto de sistemas modulares e enfocados em reusabilidade;
- Minimizar a infra estrutura de silício requerida para suportar uma comunicação eficiente em sistemas *on-chip*;

Neste sentido, a especificação do AMBA e seus diagramas temporais não fazem menção a tempos absolutos para os eventos nos sinais. Alternativamente, a especificação deixa claro apenas as relações de causalidade e precedência, como explicados ao longo deste trabalho. Esta é uma tendência cada vez mais presente em especificações que pretendem ser independentes da tecnologia de implementação.

A figura 6.4 ilustra os diagramas temporais para operações de escrita e leitura, utilizados neste trabalho. Estes diagramas temporais foram marcados, estabelecendo explicitamente a relação de causalidade entre os eventos e posteriormente traduzidos para o modelo *STG*. Este modelo constitui o protocolo padronizado de entrada para o módulo *MAC* a ser integrado, e pode ser visto na figura 6.5.

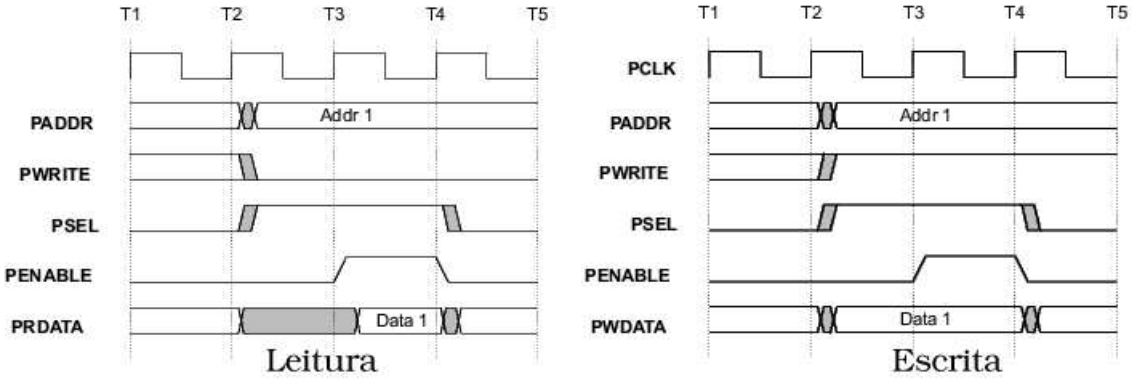


Figura 6.4: Operações de Leitura e Escrita AMBA

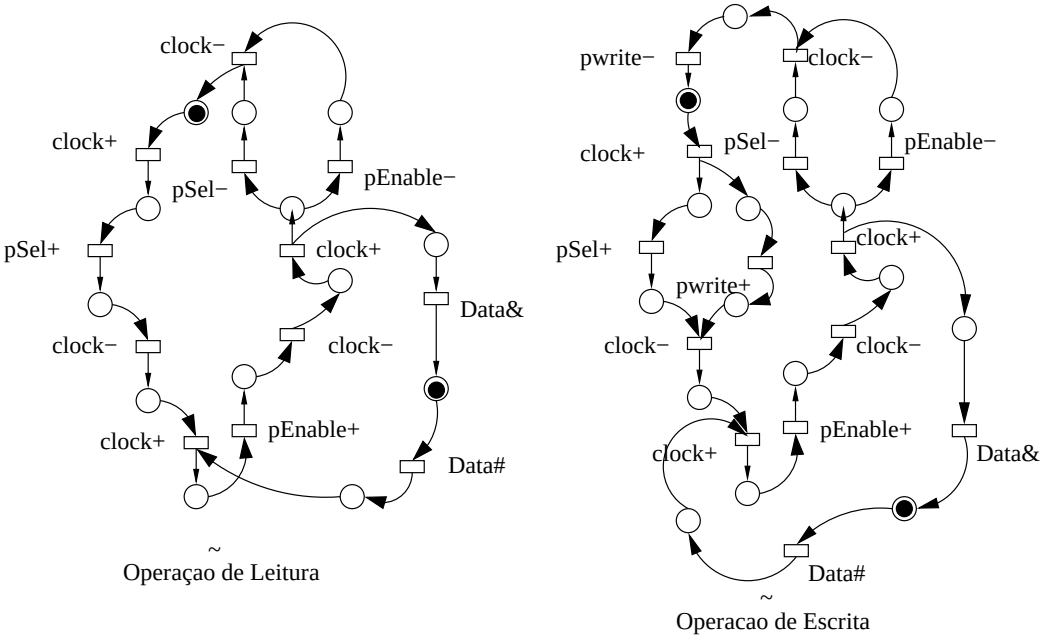


Figura 6.5: Protocolo padrão AMBA

6.3 O padrão de barramento AVALON

A exemplo da especificação do AMBA, o AVALON é uma arquitetura de barramento proposta visando a interconexão de módulos em um sistema processador/periféricos *on-chip*. Adicionalmente, a arquitetura AVALON preocupa-se com a implementação em ambientes SoC reconfiguráveis, ou seja, em hardware reconfigurável, como FPGAs e CPLDs, por exemplo.

Os principais objetivos desta arquitetura [70] são:

Simplicidade: em protocolos e procedimentos;

Otimização lógica: em especial para arquiteturas dos dispositivos reconfiguráveis de mercado.

Operação síncrona: facilitando análise temporal e adequação a plataformas reconfiguráveis.

Além destes aspectos, o AVALON tem sido largamente difundido na comunidade acadêmica e na indústria de sistemas digitais, devido a ser o barramento nativo para o processador-*Core* da Altera, o NIOS. Algumas características desta arquitetura são :

- Transferência de dados com tamanhos variáveis (8, 16 ou 32);
- Permite periféricos com latência;
- Múltiplos mestres de barramento. Em verdade o NIOS tem um interessante sistema de arbitragem, chamado *slave_ arbitration.*
- Baseado em multiplexadores e não tri-state buffers, otimizando a implementação em dispositivos configuráveis.

A figura 6.6 ilustra os diagramas temporais para operações de escrita e leitura para o barramento AVALON, utilizados neste trabalho. Estes diagramas temporais foram marcados, estabelecendo explicitamente a relação de causalidade entre os eventos e posteriormente traduzidos para o modelo *STG*. Este modelo constitui o protocolo padronizado de entrada para o barramento AVALON ao qual o *MAC* será incorporado, e pode ser visto na figura 6.7.

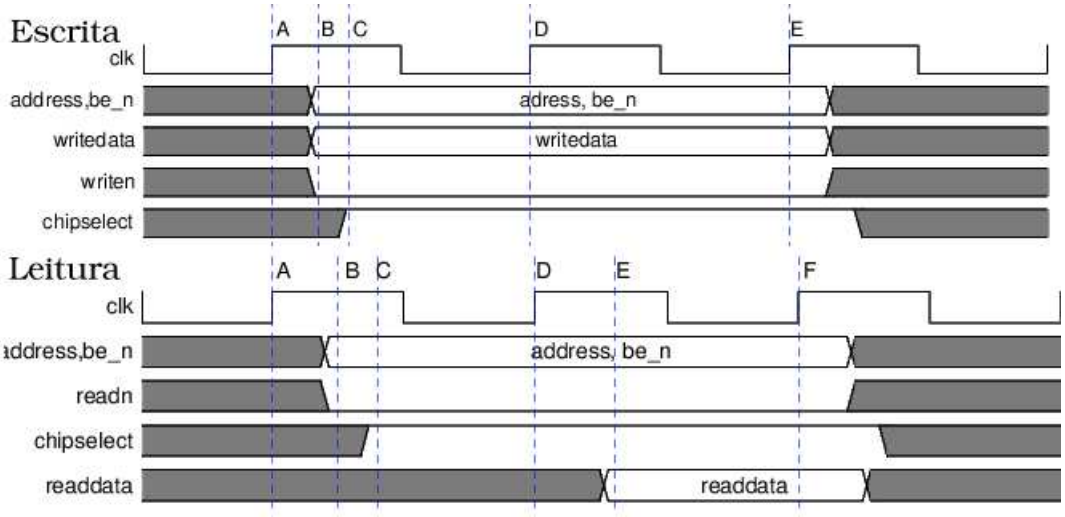


Figura 6.6: Operações de Leitura e Escrita AVALON

6.4 O Sistema Excalibur NIOS

O sistema Excalibur-NIOS [15] [69] [71] [72] é um sistema integrado[2] composto por *soft-cores* de hardware e módulos de software que se destinam a geração de SoCs.

[2]Desenvolvido pela Altera.

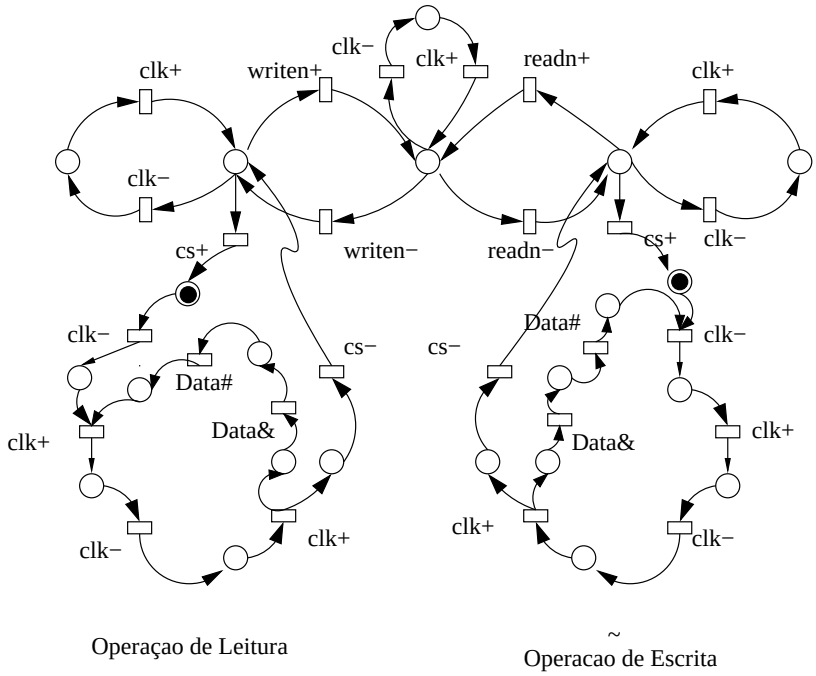


Figura 6.7: Protocolo padrão gerado para o AVALON

O componente central desta arquitetura é um processador NIOS fornecido como um *soft-core* customizável. Desta forma é possível criar uma versão do processador do sistema com diversas características: 16 ou 32 bits, instruções customizadas pelo usuário, bancos de registradores com diversos tamanhos, entre outros.

O processador é então interligado a diversos periféricos padrões [73] ou projetados pelo usuário (este é o caso do *MAC*) através do barramento AVALON [70]. Todo este sistema pode ser rapidamente montado devido ao suporte de uma ferramenta de CAD chamada SoPC Builder [74]. Adicionalmente, um compilador C é customizado para a arquitetura imaginada pelo projetista.

O sistema Excalibur-NIOS é um típico exemplo de ambiente SoC. Uma vez projetado, este sistema pode ser implementado em um placa de prototipação rápida fornecida com o kit do Excalibur-NIOS. Para este trabalho, utilizamos um sistema NIOS com todos os periféricos integrados a uma versão de 32 bits do processador principal. O processador não possui uma unidade de *MAC,* esta por sua vez, foi integrada como um novo periférico.

O periférico do *MAC*, como já explicado, foi preparado com uma interface padrão AMBA, e não poderia ser integrado diretamente ao sistema (utilizando o programa

SoPC Builder) por causa de incompatibilidades entre suas interfaces. Tornou-se necessário a geração de um processo de interface entre o módulo do *MAC* e o barramento AVALON. A próxima seção explica como a metodologia proposta nesta dissertação foi utilizada para rapidamente incorporar este módulo ao sistema.

6.5 Incorporando o *MAC* ao sistema NIOS

Para que o processo de interface entre o módulo *MAC* e o barramento AVALON fosse criado, foram executadas todas as fases propostas neste trabalho.

Inicialmente, os diagramas temporais para operações de leitura e escrita foram marcados para explicitar as relações de causalidade e precedência, considerando o barramento AVALON como mestre e o *MAC* como escravo das operações. Neste processo, os sinais *writen* e *readn* da interface do barramento AVALON foram utilizados numa operação de escolha entre as duas operações. Da mesma forma o sinal *cs* da mesma interface foi utilizado como escolha para indicar o início de uma operação. Estes diagramas marcados foram adequadamente traduzidos para uma descrição em rede de Petri, em um modelo *STG*. Estas redes podem ser observadas nas figuras 6.5 e 6.7, descrevendo as operações na interface do módulo do *MAC* e do AVALON, respectivamente.

Todas as propriedades iniciais foram observadas:

- As redes são da classe *Live and Safe Free-Choice Petri Nets*;
- As redes são persistentes em relação aos sinais de "saída", em especial aos sinais de escolha *writen*, *readn* e *cs*;
- As redes são válidas;

Os protocolos individuais, uma vez no formato padronizado, foram utilizados para prosseguir com a geração do processo de interface.

A síntese do processo de interface começou com a identificação dos pontos de sincronismo entre os protocolos. Seguindo o método automático, através dos conjuntos fonte e receptor (de validação e invalidação) atinge-se um sistema com maior

grau de paralelismo, devido aos poucos pontos de sincronismo encontrados. Esta, no entanto, não é uma solução interessante, por dois motivos:

- Os dois protocolos envolvidos são síncronos, sendo que os dados somente podem ser avaliados na transição de subida do sinal de relógio. Isto indica que será interessante estabelecer estes eventos também como fontes de sincronismo;
- Uma vez que a plataforma alvo de implementação é uma arquitetura reconfigurável, é interessante tornar o sistema o mais seqüencial possível, diminuindo assim a incidência de possíveis *hazards*.

Os pontos de sincronismo foram então indicados pelo projetista e o procedimento para conexão destes pontos de sincronismo, bem como o espelhamento do sistema, foram executados. A nova rede sintetizada denota agora o comportamento do processo de interface e pode ser vista na figura 6.8 . Note que foram eliminadas, para efeitos de simplificação, os lugares e transições que denotam eventos em sinais de dados (estes não serão mais necessários daqui em diante). Esta rede é da classe *LSFCPN*, é persistente com relação aos sinais de "entrada" (condição necessária para implementação) e válida. Não possui, porém codificação completa de estados em seu grafo de alcançabilidade.

Foi necessária a introdução de mais três sinais ($csc0$, $csc1$ e $csc2$) para resolução dos problemas de *CSC*. Após a inserção dos sinais para resolução do *CSC*, a rede resultante foi utilizada para síntese do código VHDL que descreve o processo de interface. O código gerado pode ser visto no apêndice C.

6.6 Resultados

Os testes executados com o código VHDL gerado seguiram três etapas. Inicialmente foram elaborados simulações simplesmente funcionais do processo de interface, para validar o comportamento do circuito. Na segunda etapa foram consideradas simulações temporais, com base nos recursos disponibilizados em um FPGA da família APEX20K. Por fim, o circuito foi utilizado para incoporar o módulo do *MAC* no

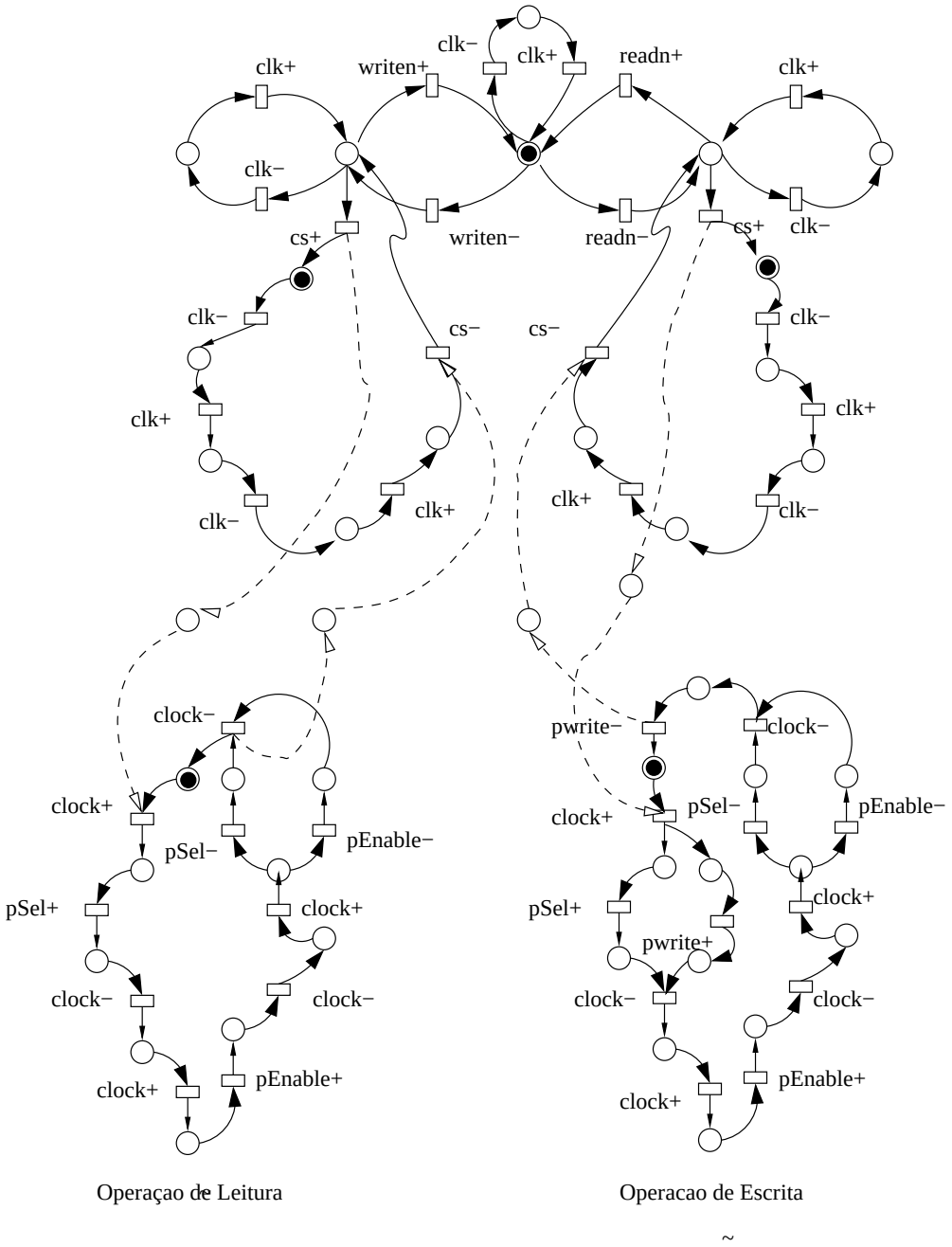


Figura 6.8: Processo de Interface gerado para o estudo de caso. Com problemas de CSC

sistema NIOS implementado na placa de prototipação rápida que compõe o kit de desenvolvimento do NIOS-Excalibur.

A figura 6.9 mostra a simulação funcional do código VHDL gerado para o processo de interface. Esta simulação foi executada no programa QUARTUS II [68] e demonstra o correto comportamento do circuito sintetizado. As duas operações, leitura e escrita, estão demonstradas.

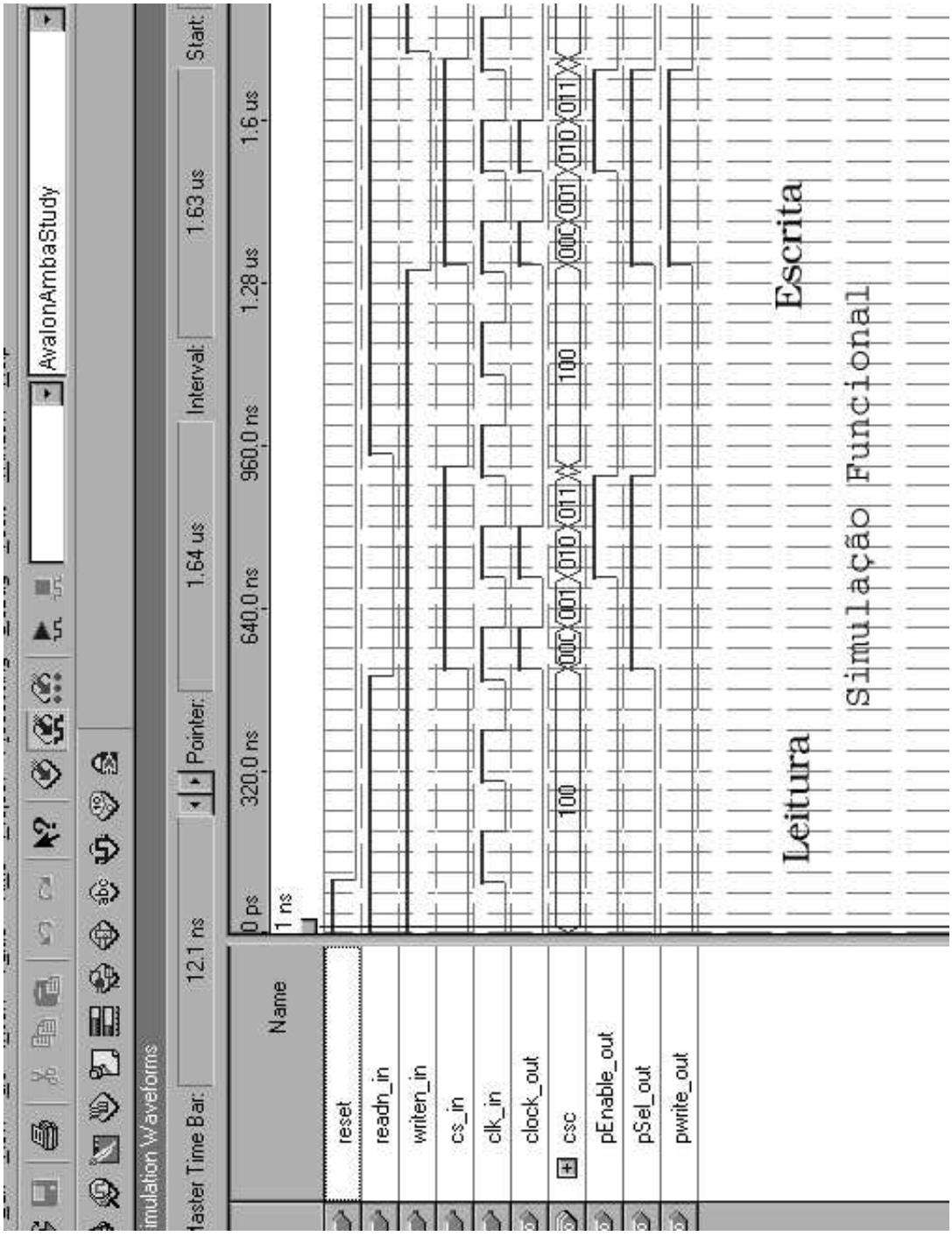


Figura 6.9: Simulação Funcional do Estudo de Caso

A figura 6.10 mostra a simulação temporal do código VHDL do processo de interface. Esta simulação leva em consideração os parâmetros temporais de um dispositivo FPGA APEX20K200E . A arquitetura de dispositivos FPGA não contempla a implementação de sistemas assíncronos de forma eficiente [13] [59], por este motivo, a freqüência de clock utilizada na simulação para o sinal *clk* , proveniente da interface

do AVALON, foi diminuída até que o sistema apresentasse estabilidade e comportamento correto. Este procedimento demonstrou que para frequências abaixo de 20Mhz o sistema apresenta comportamento conforme esperado. Acima deste valor, foram observados instabilidades (*hazards*) nos sinais de saída.

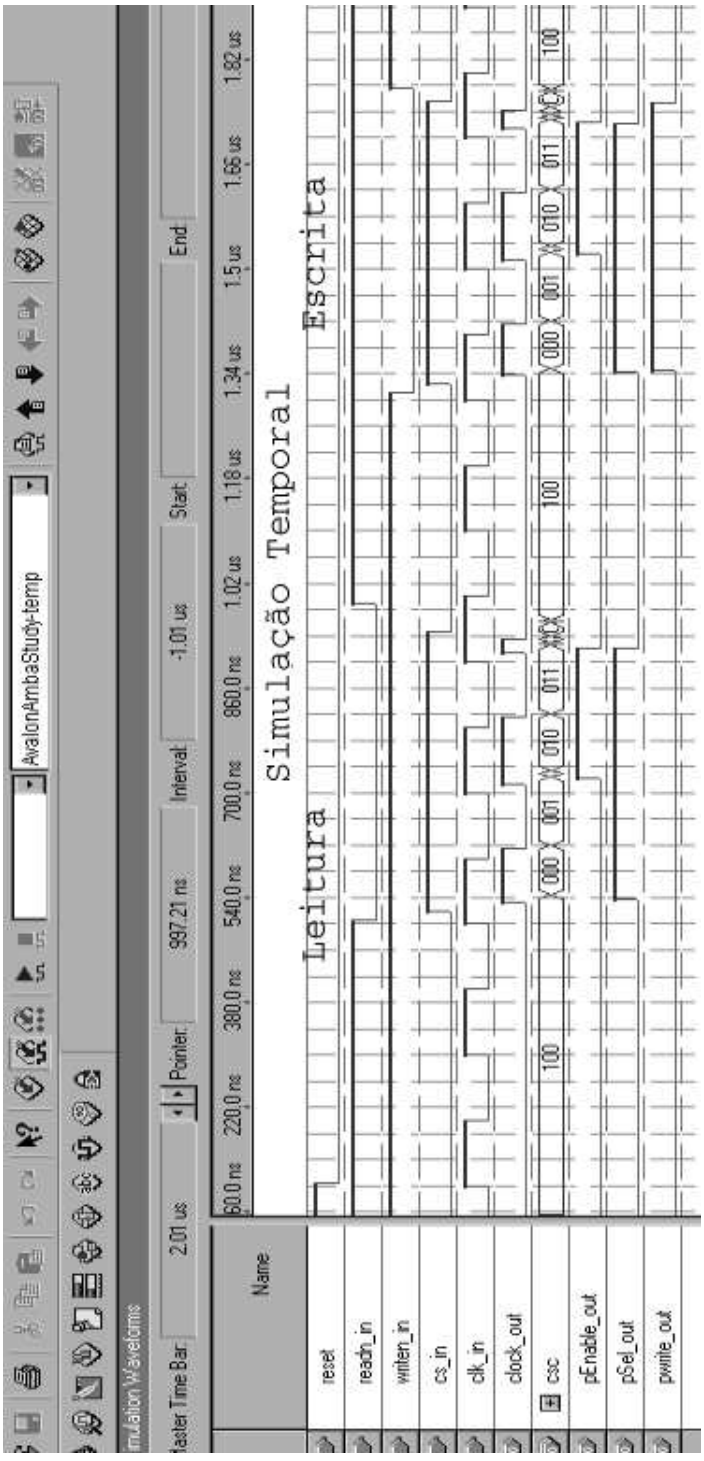


Figura 6.10: Simulação Temporal do Estudo de Caso

As instabilidades temporais ocorrem devido a violações nas condições assumidas como válidas para implementação de circuitos segundo o modelo *speed-independent*. Em FPGAs, por exemplo, não é possível garantir que o tempo de propagação nas vias utilizadas para rotear os sinais será muito menor que o tempo de propagação das portas lógicas.

Na terceira etapa dos testes, o *MAC* foi incorporado ao sistema NIOS. Inicialmente o processo de interface foi acoplado ao *MAC* através de um arquivo VHDL estrutural. O novo módulo formado consiste então do *MAC* com uma interface AMBA/AVALON, que foi integrada ao sistema NIOS através do programa SoPC Builder [74]. A tabela 6.12 demonstra algumas informações de área e tempo do processo de interface utilizado neste experimento.

Dispositivo	EP20K200EFC484-2x
Elementos Lógicos	94 / 8320 (1%)
ESBs	0 / 52 (0%)
fan-out máximo	30
fan-out médio	3.2
f_max	70Mhz
Delay máximo	44,251ns
Delay mínimo	11,841ns
Latches	7

Tabela 6.1: Dados sobre o circuito do Processo de Interface

As dificuldades encontradas neste experimento referem-se a implementação em uma plataforma reconfigurável. Detectamos a dificuldade de implementação de alguns requisitos necessários ao funcionamento do circuito do processo de interface. A arquitetura do FPGA, por exemplo, não é adequada para implementação de circuitos assíncronos com o modelo de *speed-independent*. No intuito de forçar o sistema a implementar o *latch* de cada sinal de saída, como um elemento atômico, foi necessário descrever este *latch* através de *megafunctions,* uma espécie *template* sugerido no programa do QUARTUS II. Uma vez descrito como uma *megafunction* o QUARTUS II infere o circuito a ser implementado de forma mais fácil e o mapeia diretamente em estruturas adequadas [68].

6.7 Resumo

Neste capítulo foi descrito o experimento utilizado para validar as idéias expostas no presente trabalho. Foram descritas as características dos módulos comunicantes a serem interligados, um *IP-Core* de um *MAC* e o módulo de barramento padrão AVALON. Os protocolos individuais foram marcados e transformados em uma descrição de rede de Petri, modelo *STG*. Os protocolos foram utilizados para realizar a síntese do processo de interface. Problemas de *CSC* foram eliminados e o código VHDL para o circuito foi sintetizado.

O circuito do processo de interface foi submetido a simulação funcional e temporal, mostrando funcionamento correto e validando as idéias da dissertação. Por fim, o módulo *MAC* foi integrado ao sistema NIOS e implementado na plataforma EXCALIBUR-NIOS.

Embora apenas um estudo de caso completo tenha sido apresentado, observamos que a metodologia para geração de interface mostra-se ser capaz de gerar as interfaces de forma eficiente, "correta por construção", e com baixo custo.

Capítulo 7

Conclusões

Este trabalho estabeleceu uma promissora metodologia para incorporação rápida de módulos de *hardware* no projeto de sistemas *on-chip*. Esta incorporação é possível graças à automação da síntese do processo de interface, um circuito interposto entre dois módulos comunicantes de forma a compatibilizar seus protocolos de comunicação.

Além disso, foi implementada uma ferramenta de CAD, capaz de realizar os passos estabelecidos pela metodologia, transformando especificações das interfaces individuais dos módulos, em código VHDL sintetizável, o qual descreve o processo de interface. A ferramenta de CAD automatiza uma etapa do projeto de sistemas digitais altamente susceptível a erros e normalmente executada de forma manual.

Este trabalho contribui, portanto, para o aumento da produtividade no segmento de projetos de sistemas digitais ao (1)permitir ao projetista trabalhar em um nível mais alto de abstração, ponto fundamental para abordar a complexidade dos projetos atuais; e (2)introduzir uma ferramenta importante para o aumento da reusabilidade de módulos.

Podemos, então, dividir as conclusões deste trabalho em duas partes: as contribuições principais e os trabalhos e atividades que podem ser realizadas no futuro para melhorar os resultados obtidos.

7.1 Contribuições

As principais contribuições estabelecidas ao longo deste trabalho, são:

- A metodologia estabelecida **une os trabalhos de síntese de interface a partir de redes de Petri [12, 21, 22], com técnicas de síntese lógica a partir de *STGs* [14, 42, 59]** em um único *framework*; Desta forma, a proposta desta tese é baseada em métodos já estabelecidos na literatura, e que foram unidos aqui.

- Este trabalho introduz, ao melhor de nosso conhecimento, **uma nova técnica de síntese de código VHDL a partir de redes de Petri**. Esta nova técnica está focada na descrição de uma máquina de estados assíncrona que relaciona o estado atual do sistema com a marcação correspondente na rede de Petri. A síntese de circuitos é realizada, desta forma, utilizando técnicas de minimização lógica unidas a técnicas de tradução estrutural da rede.

- Uma **ferramenta de CAD foi implementada**, capaz de automatizar a etapa de geração do processo de interface. A automação total, no entanto, é opcional. O procedimento pode, também, ser interativo, permitindo ao projetista inferir sobre a qualidade do resultado obtido.

Podemos, ainda, ressaltar as contribuições secundárias deste trabalho:

- A tradução para C++ de um conjunto de classes (PetriNet Kernel [ref]) para modelagem computacional de Redes de Petri, a partir de JAVA. Este conjunto de classes tem uma visão generalizada do conceito de redes de Petri, sendo facilmente utilizada para descrever os diferentes modelos semânticos e estruturais destas.

- A verificação automática de propriedades necessárias e/ou suficientes ao longo de todo o processo. Este fato entrega maior confiabilidade ao resultado obtido;

- O formato intermediário em redes de Petri, facilita não somente o entendimento dos conceitos envolvidos na síntese, mas permite o rápido desenvolvimento desta metodologia para incorporar novos aspectos;

- O arquivo VHDL gerado permite uma conexão estrutural em alto nível dos módulos comunicantes, facilitando a incorporação de módulos descritos em diferentes níveis de abstração; assim, por exemplo, é possível ligar facilmente *soft cores* (descritos normalmente em alto nível) com *hard cores* (distribuídos como uma *netlist* ou arquivo criptografado).

Desta forma, esperamos ter contribuído significativamente para os conhecidos problemas do *gap* de produtividade e do tempo total de projeto. E além disso, facilitado a atividade de projeto, retirando das mão do projetista a tarefa de lidar com centenas de pinos, sinais, características elétricas, etc., em um ambiente de complexidade crescente.

7.2 Trabalhos Futuros

Esta tese abre muitos caminhos para atividades e trabalhos que visem ampliar os conhecimentos estabelecidos aqui. A ferramenta de CAD, por exemplo, foi estruturada de forma a facilitar a introdução de novos módulos de análise e verificação, uma rica área para novos trabalhos.

Expomos, para cada etapa da metodologia, os pontos que acreditamos serem de grande interesse, não só para a expansão deste trabalho, mas como contribuições à área de projeto de sistemas digitais. São eles:

- A pesquisa de **formatos de especificação em alto nível e técnicas gradativas de refinamento**. Uma vez que a especificação dos protocolos individuais é o *front-end* desta metodologia, seria interessante a possibilidade de realizá-la a partir de diferentes bases. Neste sentido, sugerimos o desenvolvimento de formatos em SystemC [5], em linguagens no estilo *CSP,* ou mesmo em VHDL;

  - Ainda neste mesmo sentido, uma vez que as especificações devem ser refinadas até o nível de sinais, podem ser estudados **métodos que extraiam a especificação a partir das formas de onda geradas na simulação de um módulo**. Ou seja, o projetista não precisa mais preocupar-se em

especificar a interface, esta especificação seria automaticamente deduzida a partir de simulações funcionais (ou temporais) dos módulos.

- Incorporar à metodologia, técnicas para resolução de problemas como: compatibilidade de tipos de dados (converter, por exemplo, dados de 8 bits em formatos maiores, ou vice-versa); decodificação de endereços; e modos de transferência contínuas (*burst*) de dados. Estas novas possibilidades aumentariam o espaço de problemas resolvidos pela metodologia.

- **Novas heurísticas podem ser propostas para encontrar os pontos de sincronism**o entre os protocolos. Isto pode melhorar o resultado final obtido. Adicionalmente, técnicas de simplificação e otimização podem otimizar o processo em termos de área, explorando o paralelismo entre os sinais.

- O procedimento de síntese do código VHDL, em particular, é bastante novo. **Técnicas para otimizar a síntese de código VHDL** podem ser desenvolvidas. Isto pode ser feito, por exemplo, eliminando-se lugares desnecessário ou marcações redundantes.

- Pesquisar uma versão síncrona do **processo de interface**, que pudesse atingir velocidades mais altas **em plataformas reconfiguráveis** (*FPGAs*). Ou ainda mais interessantes, desenvolver técnicas para implementação de circuitos assíncronos nestas plataformas.

7.3 Considerações finais

O trabalho desenvolvido ao longo de dois anos no Centro de Informática, sob a orientação do professor Manoel Eusébio, foi gratificante, não somente pelo engrandecimento cultural e científico trazido pela pesquisa, mas pelo amadurecimento pessoal. Neste período, descobri novos aspectos e conceitos sobre a pesquisa científica, e de que forma ela pode contribuir na construção de um mundo mais humano.

Um agradecimento sincero a toda a família CIN, e em especial aos componentes do GRECO (Grupo de Engenharia da Computação), pelas ricas interações científicas e pessoais.

25 de março de 2003

Referências Bibliográficas

[1] H. Chang, L. Code, M. Hunt, G. Martin, A. J. McNelly, and L. Todd, *Surviving the SOC revolution: a guide to platform-based design.* Kluwer Academic Publishers, 1 ed., 1999.

[2] E. Barros and A. Sampaio, "Towards probably correct hardware/software partitioning using occam," *International Workshop on Hardware/Software Codesign*, pp. 210–217, 1994.

[3] D. Pellegrin and D. Taylor, *VHDL Made Easy.* Upper Saddle River, New Jersey: Prentice Hall PTR, 1997.

[4] A. Y. et. al., *Hardware and Petri Nets.* Kluwer Academic Publishers, 1st edition ed., 2000. Netherlands.

[5] *SystemC User's Guide*, version 2.0 ed., 2001.

[6] R. Domer and D. D. Gajski, "Reuse and protection of intellectual property in the specc system," *Proceedings of Asia and South Pacific Design Automation Conference*, pp. 49–54, January 2000. Yokohama, Japan.

[7] P. R. M. Maciel, E. Barros, and W. Rosenstiel, "A petri net model for hardware/software codesign.," *Design Automation of Embedded Systems*, vol. 4, no. 4, pp. 243–310, 1999. Boston - USA.

[8] C. Valderrama, M. E. de Lima, S. Cavalcante, and E. Barros, *Hardware/Software co-design: projetando hardware e software concorrentemente.* Escola de Computação, 2000.

[9] VSI Alliance, *VSI Alliance Architecture Document*, 1997. Version 1.0.

[10] D. D. G. Sanjiv Narayan, "Interfacing incompatible protocols using interface process generation," *Proceedings of the 32nd Design Automation Conference*, pp. 468–473, June 1995. San Francisco, CA.

[11] R. Passerone and J. A. Rowson, "Automatic synthesis of interfaces between incompatible protocols," *Proceedings of the Design Automation Conference*, pp. 8 – 15, June 1998. San Francisco, CA, USA.

[12] B. Lin and S. Vercauteren, "Synthesis of concurrent system interface modules with automatic protocol conversion generation," *Proceedings of the International Conference on Computer-Aided Design*, pp. 101–109, november 1994.

[13] A. Yakovlev and A. M. Koelmans, *Petri Nets and Digital Hardware Design*, vol. 1492 of *Lecture notes in Computer Science*. Springer-Verlag, 1998.

[14] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Methodology and tools for state encodin in asynchronous circuit synthesys," *Proceedings of the 33rd Design and Automation Conference*, 1996.

[15] Altera Inc., 101 Inovation Drive, San Jose CA, *Nios 2.0 CPU*, 2.0 ed., January 2002. http://www.altera.com.

[16] M. O'Nills, "Communication in hardware/software embedded systems," tech. rep., Royal Institute of Technology, 1997. Stocolm, Sweden.

[17] ARM Inc., *AMBA 2.0 Specification*, 2.0 ed., March 1999. http://www.arm.com.

[18] G. Borriello and R. Katz, "Synthesis and op0timization of interface transducer logic," *Proceedings of the International Conference on Computer Aided Design*, November 1987.

[19] G. Borriello, *A New Interface Specification Methodology and its Applications to Transducer Synthesis*. PhD thesis, University of California, Berkley, 1988.

[20] K.-S. Chung, R. K. Gupta, and C. L. Liu, "An algorithm for the synthesis of system-level interface circuits," *Proceedings of the Internacional Conference on Computer Aided Design*, 1996.

[21] K. J. Lin and C. S. Lin, "Automatic synthesis of asynchronous circuits," *Proceedings of the IEEE Designa Automation Conference*, pp. 296–301, 1991.

[22] G. G. Jong and B. Lin, "A communication petri net model for the design of concurrent asynchronous modules," *Proceedings of the 31st IEEE Design Automation Conference*, pp. 49–55, 1994.

[23] J. L. Peterson, *Petri Net Theory and the Modelling of Systems.* Prentice Hall Inc., 1981.

[24] C. Y. Couvreur, P. Vanbekbergen, and B. Lin, "Assassin: An asynchronous i/o interface synthesis system." Tutorial and Reference Manual IMEC Lab, October 1993.

[25] M. Einsering and J. Teich, "Interfacing hardware and software," *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications*, pp. 520–524, August 1998. Tallin, Estonia.

[26] M. Einsenring, M. Platzer, and L. Thiele, "Communication synthesis for reconfigurable embedded systems," *Proceedings of Field-Programmable Logic and Applications Conference*, pp. 205–214, August/September 1999. Glasgow UK.

[27] C. Araújo, *InterfPISH.* PhD thesis, Universidade Federal de Pernambuco, January 2001.

[28] L. Tauro and F. Vahid, "Message-based hardware/software communication in hdl/c environments," *Proceedings of the Asian-South Pacific Conference on Hardware Description Languages*, August 1997.

[29] K. Lahiri, A. Raghunathan, and S. Dey, "Efficient exploration of the soc communication architecture design space," *Proceedings of the International Conference on Computer Aided Design*, pp. 424–430, November 2000.

[30] F. Vahid and T. Givargis, "Incorporating cores into system-level specification," *Proccedings of the International Syposium on System Synthesis*, pp. 43–48, December 1998.

[31] "Opencores.org." http://www.opencores.org/.

[32] K. Kuçukçakar, "Analysis of emerging core-based design lifecycle," *Proceedings of International Conference on Computer Aided Design*, pp. 445–449, 1998. San Jose, CA, USA.

[33] M. Meerwein, C. Baumgartner, and W. Glauert, "Linking codesign and reuse in embedded systems design," *Proceedings of CODES 2000*, pp. 93–97, 2000. San Diego, CA, USA.

[34] J. C. Palma, F. Moraes, and N. Calazans, "Métodos para desenvolvimento e distruibuição de ip cores," *Seminário de Computação Reconfigurável*, Agosto 2001. Belo Horizonte, MG.

[35] R. Bergamaschi and W. R. Lee, "Designing systems-on-chip using cores," *Proceedings of the Design and Automation Conference*, pp. 420–425, 2000. Los Angeles, CA.

[36] R. L. Lyseck, F. Vahid, and T. Givargis, "Experiments with the peripheral virtual component interface," *Proceedings of the 13th International Symposium on System Synthesis*, September 2000. Madrid, Spain.

[37] S. M. Nowick, *Automatic Synthesis of burst-mode asynchronous controllers.* PhD thesis, Stanford University, March 1993.

[38] A. Davis, B. Coates, and K. Stevens, "Automatic synthesis of fast compact self-timed control circuits," *In Procceedins of IFIP Working Conference on Asynchronous Design Methodologies*, 1993.

[39] K. Y. Yun and D. L. Dill, "Automatic synthesis of 3d asynchronous finite-state machine," *In Proceedings of the International Conference on Computer Aided Design*, November 1992.

[40] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev, "Technology mapping for speed-independent circuits: decomposition and resynthesis," *Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1997. Eindhoven.

25 de março de 2003

[41] M. Kishinevsky, J. Cortadella, and M. Kondratyev, "Asynchronous interface specification," *Proceedings of Design Automation Conference*, pp. 2–7, June 1998.

[42] J. Cortadella, M. Kishinevsky, A. Kondratyev, and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *Proceedings of the 11th Conference in Design of Integrated Circuits and Systems*, pp. 205–210, November 1996. Barcelona, Spain.

[43] C. A. Petri, "Communication with automata," RADC-TR-65-377 1, Grifiss Airforce Base, 1966. New York.

[44] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, pp. 541–580, April 1989.

[45] J. Desel, *Lectures on Petri Nets I, Basic Models*, vol. 1491 of *Lecture Notes In Computer Science*, ch. Basic Linear Algebraic Techniques for Place/Transition Nets, pp. 257–308. Springer Verlag, 1998.

[46] E. Best, "Structural theory of petri nets: The free choice hiatus," *Lecture Notes in Computer Science*, vol. 254, pp. 168–206, 1987. Springer-Verlag.

[47] P. A. Beerel and T. H. Y. Meng, "Automatic gate-level synthesis of speed-independent circuits," *Proceedings of the International Conference on Computer Aided Design*, pp. 322–325, 1991.

[48] S. T. Jung and C. S. Jhon, "Direct synthesis of efficient speed-independent circuits from deterministic signal transition graphs,"

[49] J. Desel and J. Esparza, *Free Choice Petri Nets.* Cambridge University Press, 1995. Cambridge, Great Britain.

[50] T. A. Chu, *Synthesis of Self-timed VLSI Circuits form Graph Theoretic Specifications.* PhD thesis, Department of EECS, Massachussets Institute of Technology, September 1987.

[51] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev, "Logic decomposition of speed-independent circuits," *Proceedings of the IEEE*, vol. 87, pp. 347–362, February 1999.

[52] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vicentelli, "Algorithms for synthesis of hazard-free asynchronous circuits," *Proceedings of IEEE Design Automation Conference*, pp. 302–308, 1991.

[53] A. Semenov, A. Yakovlev, E. Pastor, M. A. Pena, and J. Cortadella, "Synthesis of speed-independent circuits from stg-unfolding segment," *Proceedings of Design and Automation Conference*, 1997. Anaheim, CA, USA.

[54] J. Carmona, J. Cortadella, and E. Pastor, "A structural encoding technique for the synthesis of asynchronous circuits," *In the Proccedings of the IEEE Second International Conference on Application of Concurrency to System Design*, 2001. New Castle Upon Tyne, UK.

[55] R. Millner, *A calculus of communication Systems*, vol. 92. Springer-Verlag, 1982.

[56] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, E. Pastor, and A. Yakovlev, "Decomposition and technology mapping of spped-independent circuits using boolean relations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 1221–1236, September 1999.

[57] L. Lavagno, C. W. Moon, R. K. Brayton, and A. Snafiovanni-Vicentelli, "Solving the state assignment problem for signal transition graphs," *Proceedings of the 29th IEEE Design Automation Conference*, pp. 568–572, 1992.

[58] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "A region-based theory for state assignment in speed-independent circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 793–812, August 1997.

[59] M. Kishinevsky, M. Kondratyev, A. Taubin, and V. Varshavsky, "Concurrent hardware: The theory and practice of self-timed design," Series in Parallel Computing, 1994.

[60] M. Uzam, M. K. Yalçin, and M. Avci, "Digital hardware implementation of petri net based specifications: Direct translation form safe automation petri nets to circuit elements," *Proceedings of the International Workshop on Discrete Event Systems Design*, pp. 25–33, June 2001. Poland.

[61] S. Roch and P. H. Starke, *INA - Integrated Net Analyzer Version 2.2*. Humbolt University, 1999.

[62] S. Mazor and P. Langstraat, *A Guide to VHDL*. Kluwer Academic Publisher, 1993. USA.

[63] Institute of Electrical and Electronics Engineers, *IEEE Standard VHDL Language Reference Manual*, ieee std 1076-1993 ed., 1994. New York USA.

[64] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley Pub Co, 3rd edition ed., February 2000.

[65] J. W. Satzinger and T. U. Orvik, *The Object-Oriented Approach: Concepts, system development and Modeling with UML*. Course Technology, 2nd edition ed., January 2001.

[66] E. Kindler and M. Weber, *The Petri Net Kernel: Documentation of the application interface*. Humbolt University Berlin, September 1998. Revision 1.1.

[67] "The petri net kernel," 1998.

[68] Altera Inc., *Quartus II Manual*, 1.2 ed., 2002. http://www.altera.com.

[69] Altera Inc, 101 Inovation Drive , San Jose, CA, *Nios Embedded Processor: 32-bit programmer's reference manual*, 2.0 ed., January 2002. http://www.altera.com.

[70] Altera Inc., 101 Inovation Drive, San Jose, CA, *AVALON Bus Specification - Reference Manual*, 2.0 ed., January 2002. http://www.altera.com.

[71] Altera Inc., 101, Inovation Drive, San Jose, CA, *Nios Embedded Processor - Getting Started*, 2.0 ed., January 2002. http://www.altera.com.

[72] Altera Inc., 101, Inovation Drive, San Jose, CA, *Nios Tutorial*, 2.0 ed., January 2002. http://www.altera.com.

[73] Altera Inc, 101, Inovation Drive, San Jose, CA, *Nios Embedded Processor - Peripheral Reference Manual*, 2.0 ed., January 2002. gttp://www.altera.com.

[74] Altera Inc., 101, Inovation Drive, San Jose, CA, *SoPC Builder - Data Sheet*, 2.0 ed., January 2002. http://www.altera.com.

Apêndice A

Este apêndice mostra o arquivo VHDL gerado para o exemplo desenvolvido ao longo do capítulo 4.

```
_************************************************************************
–
– Universidade Federal de Pernambuco
– Centro de Informática
– GRECO - Grupo de Engenharia da Computação
– Programa de Pós Graduação - Mestrado em Ciência da Computação
–
– Federal University of Pernambuco
– Centre for Informatics
– GRECO - Computing Engineering Group
– Post Graduation program - Masters on Computing Science
–
– Developed by Julio Alexandrino de Oliveria Filho , Msc Student
– Under orientation of Dr. Prof. Manoel Eusébio de Lima
– A CAD Tool for Interface Automatic Generation
– Copyrigths reserved
–
–
–
– Universidade Federal de Pernambuco
– Av. Professor Luiz Freire, S/N
– Centro de Informática
– Julio Alexandrino de Oliveira Filho
– jaof@cin.ufpe.br
– Manoel Eusébio de Lima
– mel@cin.ufpe.br
–
_************************************************************************
– Project default libraries.
LIBRARY IEEE;

USE IEEE.STD_LOGIC_1164.ALL;

USE IEEE.STD_LOGIC_ARITH.ALL;

– Entity declaration.
ENTITY netTese IS – Input/Output port declarations.
  PORT (
    f_csc0 : IN STD_LOGIC;
    f_csc1 : IN STD_LOGIC;
    f_csc2 : IN STD_LOGIC;
```

```
    f_csc3 : IN STD_LOGIC;
    f_csc4 : IN STD_LOGIC;
    f_req : IN STD_LOGIC;
    f_wait : IN STD_LOGIC;
    ack_in : IN STD_LOGIC;
    cs_in : IN STD_LOGIC;
    csc0_out : OUT STD_LOGIC;
    csc1_out : OUT STD_LOGIC;
    csc2_out : OUT STD_LOGIC;
    csc3_out : OUT STD_LOGIC;
    csc4_out : OUT STD_LOGIC;
    enable_in : IN STD_LOGIC;
    req_out : OUT STD_LOGIC;
    wait_out : OUT STD_LOGIC;
    reset_wrapper : IN STD_LOGIC;
    write_in : IN STD_LOGIC;
  ); END netTese; -- Architecture declaration.

ARCHITECTURE behavioural_interface of netTese IS
  SIGNAL wait_set : STD_LOGIC;
  SIGNAL wait_reset : STD_LOGIC;
  SIGNAL req_set : STD_LOGIC;
  SIGNAL req_reset : STD_LOGIC;
  SIGNAL ack_set : STD_LOGIC;
  SIGNAL csc0_set : STD_LOGIC;
  SIGNAL csc0_reset : STD_LOGIC;
  SIGNAL csc1_set : STD_LOGIC;
  SIGNAL csc1_reset : STD_LOGIC;
  SIGNAL csc2_set : STD_LOGIC;
  SIGNAL csc2_reset : STD_LOGIC;
  SIGNAL csc3_set : STD_LOGIC;
  SIGNAL csc3_reset : STD_LOGIC;
  SIGNAL csc4_set : STD_LOGIC;
  SIGNAL csc4_reset : STD_LOGIC;
  SIGNAL StateVector : STD_LOGIC_VECTOR(1 to 11);
  SIGNAL PSV : STD_LOGIC_VECTOR(1 to 25);
  SIGNAL PSVaux : STD_LOGIC_VECTOR(1 to 25);
  SIGNAL GND : STD_LOGIC;

-- Component declaration.
COMPONENT speedIndependent_latch IS -- Input/Output port declarations.
  PORT (
    aclr : IN STD_LOGIC;
    aset : IN STD_LOGIC;
    data : IN STD_LOGIC;
    gate : IN STD_LOGIC;
    q : OUT STD_LOGIC );
  END COMPONENT;
BEGIN
  U_wait : speedIndependent_latch PORT MAP (wait_reset ,wait_set ,GND ,GND ,wait_out);
  U_req : speedIndependent_latch PORT MAP (req_reset ,req_set ,GND ,GND ,req_out);
  U_csc0 : speedIndependent_latch PORT MAP (csc0_reset ,csc0_set ,GND ,GND ,csc0_out);
  U_csc1 : speedIndependent_latch PORT MAP (csc1_reset ,csc1_set ,GND ,GND ,csc1_out);
  U_csc2 : speedIndependent_latch PORT MAP (csc2_reset ,csc2_set ,GND ,GND ,csc2_out);
  U_csc3 : speedIndependent_latch PORT MAP (csc3_reset ,csc3_set ,GND ,GND ,csc3_out);
```

```
U_csc4 : speedIndependent_latch PORT MAP (csc4_reset ,csc4_set ,GND ,GND ,csc4_out);

 StateVector(1) <= write_in;
 StateVector(2) <= cs_in;
 StateVector(3) <= enable_in;
 StateVector(4) <= f_wait;
 StateVector(5) <= f_req;
 StateVector(6) <= ack_in;
 StateVector(7) <= f_csc0;
 StateVector(8) <= f_csc1;
 StateVector(9) <= f_csc2;
 StateVector(10) <= f_csc3;
 StateVector(11) <= f_csc4;

 PSV <= "000001001000010000000100111" when reset_wrapper = '1' else PSVaux;
WITH StateVector SELECT PSVaux <= "000000000010000000001010000" when "00000000010",
   "001000000100000000001000000" when "00000000110",
   "100000000100000000010000" when "00000001010",
   "101000000100000000000000" when "00000001110",
   "000000000000000000101010000" when "00100000010",
   "001000000000000000101000000" when "00100000110",
   "100000000000000000100010000" when "00100001010",
   "101000000000000000100000000" when "00100001110",
   "000000001000000000001010000" when "00100010010",
   "001000001000000000001000000" when "00100010110",
   "100000001000000000010000" when "00100011010",
   "101000001000000000000000" when "00100011110",
   "000100000100000000001000000" when "01000000010",
   "000000000100000000001000001" when "01000000100",
   "000001000100000000001000000" when "01000000101",
   "000000000100000000001001000" when "01000000110",
   "100100000100000000000000" when "01000001010",
   "100000000100000000000001" when "01000001100",
   "100001000100000000000000" when "01000001101",
   "100000000100000000001000" when "01000001110",
   "000100000000000000101000000" when "01100000010",
   "000000000000000000101000001" when "01100000100",
   "000001000000000000101000000" when "01100000101",
   "000000000000000000101001000" when "01100000110",
   "100100000000000000100000000" when "01100001010",
   "100000000000000000100000001" when "01100001100",
   "100001000000000000100000000" when "01100001101",
   "100000000000000000100001000" when "01100001110",
   "000100001000000000001000000" when "01100010010",
   "000000001000000000001000001" when "01100010100",
   "000001001000000000001000000" when "01100010101",
   "000000001000000000001001000" when "01100010110",
   "100100001000000000000000" when "01100011010",
   "100000001000000000000001" when "01100011100",
   "100001001000000000000000" when "01100011101",
   "100000001000000000001000" when "01100011110",
   "010000000100000000010000" when "10000000010",
   "011000000100000000000000" when "10000000110",
   "000010000100000000010000" when "10000001010",
   "001010000100000000000000" when "10000001110",
   "010000000000000100010000" when "10100000010",
```

```
"011000000000000100000000" when "10100000110",
"000010000000000100010000" when "10100001010",
"001010000000000100000000" when "10100001110",
"010000001000000000010000" when "10100010010",
"011000001000000000000000" when "10100010110",
"000010001000000000010000" when "10100011010",
"001010001000000000000000" when "10100011110",
"010100000100000000000000" when "11000000010",
"000000000000000000000010" when "11000000011",
"010000000100000000000001" when "11000000100",
"010001000100000000000000" when "11000000101",
"010000000100000000001000" when "11000000110",
"000000000000000000100000" when "11000000111",
"000110000100000000000000" when "11000001010",
"000010000100000000000001" when "11000001100",
"000011000100000000000000" when "11000001101",
"000010000100000000001000" when "11000001110",
"000000000000000010000000" when "11000001111",
"010100000000000100000000" when "11100000010",
"010000000000000100000001" when "11100000100",
"010001000000000100000000" when "11100000101",
"010000000000000100001000" when "11100000110",
"000110000000000100000000" when "11100001010",
"000010000000000100000001" when "11100001100",
"000011000000000100000000" when "11100001101",
"000010000000000100001000" when "11100001110",
"000000100010000000000000" when "11100001111",
"010100001000000000000000" when "11100010010",
"010000001000000000000001" when "11100010100",
"010001001000000000000000" when "11100010101",
"010000001000000000001000" when "11100010110",
"000110001000000000000000" when "11100011010",
"000010001000000000000001" when "11100011100",
"000011001000000000000000" when "11100011101",
"000010001000000000001000" when "11100011110",
"000000100000010000000000" when "11100011111",
"000000100000100000000000" when "11100111111",
"000000100001000000000000" when "11101001111",
"000000100000001000000000" when "11101011111",
"000000100000100000000000" when "11101111111",
"000000010010000000000000" when "11110001111",
"000000000000000000000100" when "11110011101",
"000000010000010000000000" when "11110011111",
"000000010000100000000000" when "11110111111",
"000000010001000000000000" when "11111001111",
"000000010000001000000000" when "11111011111",
"000000010000100000000000" when "11111111111",
"000000000000000000000000" when others;

wait_reset <= PSV(23); wait_set <= PSV(7);
req_reset <= PSV(13); req_set <= PSV(11);
csc0_reset <= PSV(9); csc0_set <= PSV(12);
csc1_reset <= PSV(1); csc1_set <= PSV(20);
csc2_reset <= PSV(3); csc2_set <= PSV(24);
csc3_reset <= PSV(8) and PSV(15); csc3_set <= PSV(25);
csc4_reset <= PSV(6); csc4_set <= PSV(4) and PSV(10);
```

```
        PSV_out <= PSV;

    END behavioural_interface;
```

Apêndice B1

Neste apêndice, estão descritoso os arquivos de entrada da ferramenta de CAD implementada, em um formato *Bakus-Nauer Form* extendido. Os símbolos [ ] denotam opções, { } denotam repetições, | uma alternativa, "" uma string, < > uma unidade sintática e ::= é a substituição de um símbolo. A única particularidade é a string "<cr>", que deve ser interpretada como fim-de-linha.

O arquivo consiste em quatroseções separadas pelo símbolo @. A primeira seção contém informações sobre a estrutura da rede, a segunda e a terceira seção sobre os lugares e transições, respectivamente. A quartaseção contém informações sobre os sinais do *STG* e consiste na modificação que foi realizada nete trabalho. Os arquivos de entrada, formatados desta forma, devem receber a extensão ".stg". A EBNF que define o formato dos arquivos ".stg" é:

```
<pnt-file> ::= <netheader> ''<cr>''
               <netstruct> ''<cr>''
               ''@<cr>''
               <placedata> ''<cr>''
               ''@<cr>''
               <transdata> ''<cr>''
               ''@<cr>''
               <signaldata>''<cr>''
               ''@<cr>''
<netheader> ::= ''P   M   PRE,POST   NETZ'' <netid>
    <netid> ::= <netnr> [ '':''<netname>]
    <netnr> ::= <number>
  <netname> ::= <name>
```

```
<netstruct> ::= <placedef> {“<cr>”<placedef>}
 <placedef> ::= {”  “<placenr>” “<tokens>”  “
          [<prelist>][”,”<postlist>]}
  <placenr> ::= <number>
   <tokens> ::= <number>
  <prelist> ::= {<transnr> [ “: “ <arcmult> ] “ “}
 <postlist> ::= {<transnr> [”: “ <arcmult> ] “ “}
  <transnr> ::= <number>
  <arcmult> ::= <number>
<placedata> ::= “place nr.        name capacity time”
          {”<cr>     “<placenr> “: “ <placename> “    “
           <capacity> }
<placename> ::= <name>
 <capacity> ::= “ ”<number> | “oo”
<transdata> ::= “trans nr.        name signal action”
          {”<cr>    “<transnr>”: “ <transname> “    “
             <signalnr> <signalaction> }
<transname> ::= <name>
 <signalnr> ::= <number>
<signalaction> ::= “+”| “-” | “#”| “&”
  <signaldata> ::= “signal nr.        name nature direction default_stat
          {”<cr>    “ <signalnr> “: “ <signalname> “  “
          <signalnature> “  “ <signaldirection> “ “
          <signaldefault> }
  <signalname> ::= <name>
 <signalnature> ::= “SIMPLE” | “DATA”
<signaldirection> ::= “IN” | “OUT” | “INTERNAL”
  <signaldefault> ::= <number>
```

A figura 1 mostra uma pequena rede, *STG*, e o código equivalente em seguida.

```
P   M    PRE,POST NETZ 0:Signalnets
```

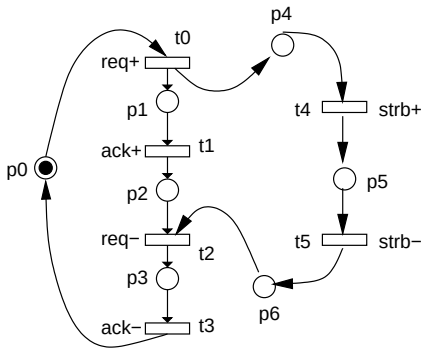


Figura 1: Rede exemplo para formato de entrada

```
  0 1     3, 0
  1 0     0, 1
  2 0     1, 2
  3 0     2, 3
  4 0     0, 4
  5 0     4, 5
  6 0     5, 2
@
place nr. name capacity time
  0: p0          oo  0
  1: p1          oo  0
  2: p2          oo  0
  3: p3          oo  0
  4: p4          oo  0
  5: p5          oo  0
  6: p6          oo  0
@ trans nr. name signalNr. action
  0: t0      0  +
  1: t1      1  +
  2: t2      0  -
  3: t3      1  -
  4: t4      2  +
  5: t5      2  -
```

```
@
signal nr. name nature direction default_state
  0: req     SIMPLE  OUT  0
  1: ack     SIMPLE  IN   0
  2: strb    SIMPLE  OUT  0
@
```

.