Universidade Estadual de Campinas
FACULDADE DE ENGENHARIA ELÉTRICA
DEPARTAMENTO DE TELEMÁTICA

UNICAMP

Síntese Comportamental de Circuitos Digitais Utilizando SDL

Renato Jansen de Oliveira Figueiredo

14 de Junho de 1995

Banca examinadora:

Dr. Ivanil Sebastião Bonatti
Dr. Mário Lúcio Cortes
Dr. Furio Damiani

Suplente:

Dr. Carlos Alberto dos Reis Filho

Dissertação apresentada à Faculdade de Engenharia Elétrica da UNICAMP, como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica

Este exemplar corresponde à redação final da tese defendida por Renato Jansen de Oliveira Figueiredo e aprovada pela Comissão Julgadora em 14/06/95.

Orientador

Curriculum Vitae da Banca Examinadora

- **Prof. Dr. Ivanil Sebastião Bonatti** - Presidente
  Doutor em Automática - 1981 - Toulouse - França
  Título da Tese: Gestion de Réseaux de Service: Application au Réseau Telephonique Interurbain
  Local de trabalho: DT - FEE - UNICAMP
- **Prof. Dr. Mário Lúcio Cortes** - Membro
  Ph.D em Engenharia Elétrica - 1987 - U.S.A.
  Título da Tese: Temporary Failures in Digital Circuits: Experimental Results and Fault Modeling.
  Local de trabalho: DCC - IMECC - UNICAMP
- **Prof. Dr. Furio Damiani** - Membro
  Doutor em Engenharia Elétrica - 1983 - UNICAMP
  Título da Tese: Contribuição ao Estudo de Estruturas MOS Implantadas com Cloro.
  Local de trabalho: DSIF - FEE - UNICAMP
- **Prof. Dr. Carlos Alberto dos Reis Filho** - Suplente
  Doutor em Engenharia Elétrica - 1982 - UNICAMP
  Título da Tese: Correção de Curvatura em Fontes de Referência Tipo 'BANDGAP'
  Local de trabalho: DEMIC - FEE - UNICAMP

Dedico este trabalho aos meus pais,
Ferdinando e Marlete. Obrigado,
por tudo.

Agradecimentos

Agradeço ao meu orientador e Professor, Ivanil Bonatti, com quem convivo na Faculdade desde o início do curso de graduação como aluno em disciplinas, iniciação científica e Mestrado, de maneira sempre produtiva e agradável. Sob sua orientação exemplar tive a liberdade e a oportunidade de aprender, discutir, desenvolver, implementar e expor as idéias que resultaram neste trabalho.

Agradeço aos responsáveis pelos Laboratórios do Departamento de Telemática e de Computação Aplicada à Engenharia Elétrica (LCAEe), especialmente aos Professores José Raimundo de Oliveira e Walter da Cunha Borelli, pelo esforço e iniciativa de tornar disponíveis recursos computacionais de última geração, sem os quais este trabalho não seria viável.

Agradeço aos meus familiares mais próximos, meus pais, minha avó Guiomar, meus irmãos Márcio e Luciana (e à pequena sobrinha Marina!) pelo carinho, apoio e confiança que sempre recebi.

Agradeço aos meus amigos e colegas de faculdade, alunos e Professores, de quem sempre obtive incentivo, especialmente ao Professor Furio Damiani, e aos Professores Mário Lúcio Cortes e Carlos Alberto dos Reis Filho pela participação na Banca de Defesa.

Agradeço também ao CNPq pelo apoio financeiro que obtive durante o curso de Mestrado.

Resumo

Metodologias de projeto de sistemas digitais em alto nível têm por objetivos principais permitir a descrição de sistemas cada vez mais complexos, mantendo a visão sistêmica do projeto durante todo o seu ciclo de desenvolvimento, e diminuir o tempo de desenvolvimento de projetos.

Este trabalho apresenta uma abordagem à metodologia de projeto de sistemas digitais através do uso de uma linguagem de especificação (**SDL** - *Specification and Description Language*) para o projeto de circuitos. Um algoritmo de mapeamento de um sub-conjunto da linguagem **SDL** para **VHDL** (*Very High Speed Integrated Circuit Hardware Description Language*) sintetizável é apresentado, juntamente com exemplos de aplicação, e implementado em um programa de domínio público (**Stoht** - *SDL to Hardware Translator*) que realiza este mapeamento de maneira automática.

Abstract

System-level design methodologies allow the description of more complex systems, keeping the view of the system as a whole throughout all its developing cycle, and also allow the shortening of project development schedule.

This work presents an approach to system-level design by using a specification language (**SDL** - *Specification and Description Language*) to hardware design. An algorithm for the translation of a subset of **SDL** to synthesisable **VHDL** (*Very High Speed Integrated Circuit Hardware Description Language*) is presented together with examples of implementation. This algorithm has been implemented in a software translator (**Stoht** - *SDL to Hardware Translator*) that performs the language mapping automatically.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Projeto de Sistemas

1.1 Introdução

Os circuitos digitais têm incorporado sistematicamente os avanços tecnológicos e metodológicos das várias áreas da engenharia desde os seus primórdios. O aumento da capacidade de integração e da velocidade destes circuitos vêm ocorrendo de forma praticamente exponencial com o passar dos anos.

Com o intuito de explorar toda esta potencialidade, diversas metodologias de projeto têm sido propostas, apoiadas principalmente nos crescentes recursos computacionais disponíveis, tanto de hardware quanto de software. Os objetivos principais destas metodologias são o aumento da produtividade e a diminuição do tempo de desenvolvimento dos circuitos.

O desenvolvimento de Linguagens de Descrição de Hardware (**HDL** [1]) que aceitam descrições comportamentais, notadamente **VHDL** [2] [27],[28], e a disponibilidade de ferramentas de automação de projeto (**EDA** [3]) que fazem uso destas linguagens tanto para simulação quanto para síntese têm permitido a descrição de hardware em um nível mais alto de abstração [12],[5],[25].

O objetivo deste trabalho é trazer o projeto de circuitos digitais mais próximo de sua especificação sem perder a capacidade de implementação. Para tal, é usada a linguagem **SDL** [4] [7],[31], desenvolvida pelo **CCITT** [5], como uma ferramenta de captura de projeto gráfica, e a geração automática de código **VHDL** para a implementação final.

Tomando-se como referência a carta Y de Gajski [14] (Fig. 1.1), este trabalho opera no eixo correspondente à visão comportamental e no retângulo correspondente ao nível de arquitetura. A entrada de dados é feita através de uma descrição comportamental em **SDL**, e posteriormente mapeada em uma descrição **VHDL**, também comportamental.

Simultaneamente, o algoritmo proposto neste trabalho promove uma primeira partição estrutural do sistema **SDL** em descrições de componentes **VHDL** para os blocos e processos **SDL**. Os processos **SDL** são particionados em componentes **VHDL** distintos que modelam seu comportamento e sua comunicação com demais processos.

---

[1] Hardware Description Languages
[2] VHSIC Hardware Description Language
[3] Electronic Design Automation
[4] Specification and Description Language
[5] International Telegraph and Telephone Consultative Committee

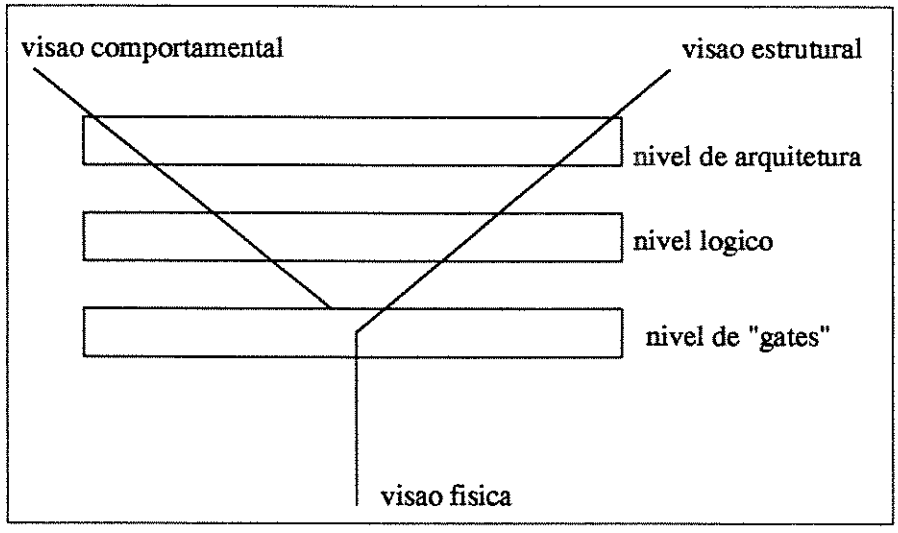


Figura 1.1 Carta Y de gajski relacionando visões e níveis de um projeto digital.

1.2 Metodologia de projeto de sistemas

Metodologias de projeto que permitam o desenvolvimento de circuitos digitais a partir de especificações sistêmicas têm sido abordadas recentemente. O objetivo principal destas metodologias é dar ao projetista uma visão sistêmica do projeto em todo o seu ciclo.

A crescente disponibilidade de recursos computacionais de hardware e software tem permitido o uso de linguagens de programação para a especificação de sistemas (**SDL**, entre outras), para o projeto de hardware (**VHDL**, principalmente) e para o projeto de software. O projeto de sistemas provê uma ponte entre a especificação do sistema e o projeto de hardware e software através do uso destas linguagens formais de especificação e descrição.

Uma proposta de uso de **SDL** para projeto de hardware baseado em **VHDL** foi introduzida em [17] e [16], e posteriormente refinada em [22], cobrindo aspectos de simulação, e [15], cobrindo aspectos de síntese.

Em [21] e [20] foi apresentada uma linguagem intermediária fazendo a interface entre linguagens de especificação e implementação, denominada **SOLAR**. Uma abordagem de projeto conjunto de software e hardware (**co-design**) baseada nesta linguagem foi apresentada em [18]e [19]; outros trabalhos neste tópico foram publicados em [29] e [26].

Em [23] e [24] uma nova linguagem de especificação (**SpecCharts**) foi apresentada para o projeto de sistemas. Esta linguagem é baseada em diagramas de estado concorrentes que contêm declarações VHDL no corpo das transições de estados. Uma característica comum a todos estes trabalhos é a geração de código **VHDL** utilizada para a obtenção de descrições de circuitos.

O projeto de sistemas insere-se em um contexto que engloba avanços em especificação de sistemas, projeto de hardware e projeto de software, resumidos a seguir.

- **Linguagens de Descrição de Hardware (HDLs)**
  Linguagens que oferecem recursos largamente utilizados no desenvolvimento de software estão disponíveis para o projeto de hardware. **VHDL** é o exemplo mais proeminente de tais linguagens; é aceita pela maioria das ferramentas de projeto de circuitos digitais tanto comerciais quanto não-comerciais desde que tornou-se um padrão **IEEE** em 1987 [27].
  VHDL oferece estruturas de dados e controle sequencial de fluxo, como uma linguagem de progra-

mação estruturada (visão comportamental). Ela permite declarações concorrentes, necessárias para modelar circuitos reais (visão *data-flow*). **VHDL** também permite declaração, instanciação e interconexão de componentes, de maneira similar a esquemáticos ou listas de ligações (visão estrutural). Todas estas três visões de projeto podem ser combinadas para a criação de um modelo de hardware, oferecendo uma grande flexibilidade ao projetista.
VHDL ainda possui limitações, mas e' uma linguagem "viva". Portanto, pode incorporar melhoras e corrigir problemas que eventualmente apareçam devido ao seu uso intenso.

- **Linguagens de especificação**: Linguagens de especificação tém sido desenvolvidas para auxiliar projetistas e coordenadores de projeto a representar especificações formalmente. Tal linguagem geralmente oferece entrada de dados gráfica e nível abstrato de descrição (incluindo texto informal). Uma linguagem a ser destacada é **SDL**, suportada pelo **CCITT** e amplamente utilizada em telecomunicações.
  SDL permite a entrada de uma especificação no início do ciclo de projeto, pois aceita texto informal. **SDL** também provê a descrição da estrutura e comportamento do sistema formalmente e gerar código executável através do uso de ferramentas **CASE** [6].
  Embora **SDL** venha sendo usada na maioria das vezes para o projeto de software, sua capacidade de modelamento pode também ser usada para o projeto de hardware, e, mais adiante, para a integração do projeto de software e hardware em um sistema (**co-design**).
  O projetista ou grupo de projetistas pode, portanto, possuir a visão sistêmica do projeto logo na etapa de especificação, e manter esta visão por todo o ciclo de projeto até a implementação final.

- **Projeto** *Top-Down*: A abordagem *top-down* propõe que projetos sejam desenvolvidos a partir do mais alto nível (sistema) e que sejam refinados até que o nível de implementação seja alcançado. Desta maneira, o projetista aproxima-se da especificação do sistema já no início do projeto.
  O projeto de sistemas utiliza o conceito *top-down* em conjunto com a geração automática de código e ferramentas de síntese de alto nível. Ao se fazer esta união, o ciclo de projeto fica mais curto, já que a conexão entre a especificação (sistema) e a implementação pode ser feita de maneira automática por software.

- **Projeto conjunto de hardware/software (co-design)**: A abstração oferecida pela abordagem sistêmica permite o projeto de sistemas independentemente da implementação final, seja ela em software, em hardware ou ambos. Para projetos mistos, a partição do sistema em hardware e software pode ser feita após uma análise das características do sistema. A Fig. 1.2 mostra a metodologia de projeto proposta neste trabalho.

  Ao se usar geração automática de código para ambos os ramos (hardware e software), esta partição pode até mesmo ser alterada sem que a especificação original seja modificada. Uma nova tecnologia disponível, por exemplo, pode permitir a realocação de módulos de software para hardware e melhorar o desempenho do sistema como um todo.
  O objetivo da metodologia de projeto conjunto é integrar as visões de hardware e software em um único ambiente; esta metodologia deve, portanto, prover comunicação transparente entre estas duas visões, assim como síntese automática para ambas, via geração de código.

- **Utilização de ambientes EDA:**
  Ferramentas EDA estão tornando-se cada vez mais complexas e integrando mais aspectos do projeto de circuitos, desde linguagens de descrição (**VHDL**) ao projeto físico (placas de circuitos, circuitos integrados e empacotamento mecânico).

[6] Computer Aided Software Engineering

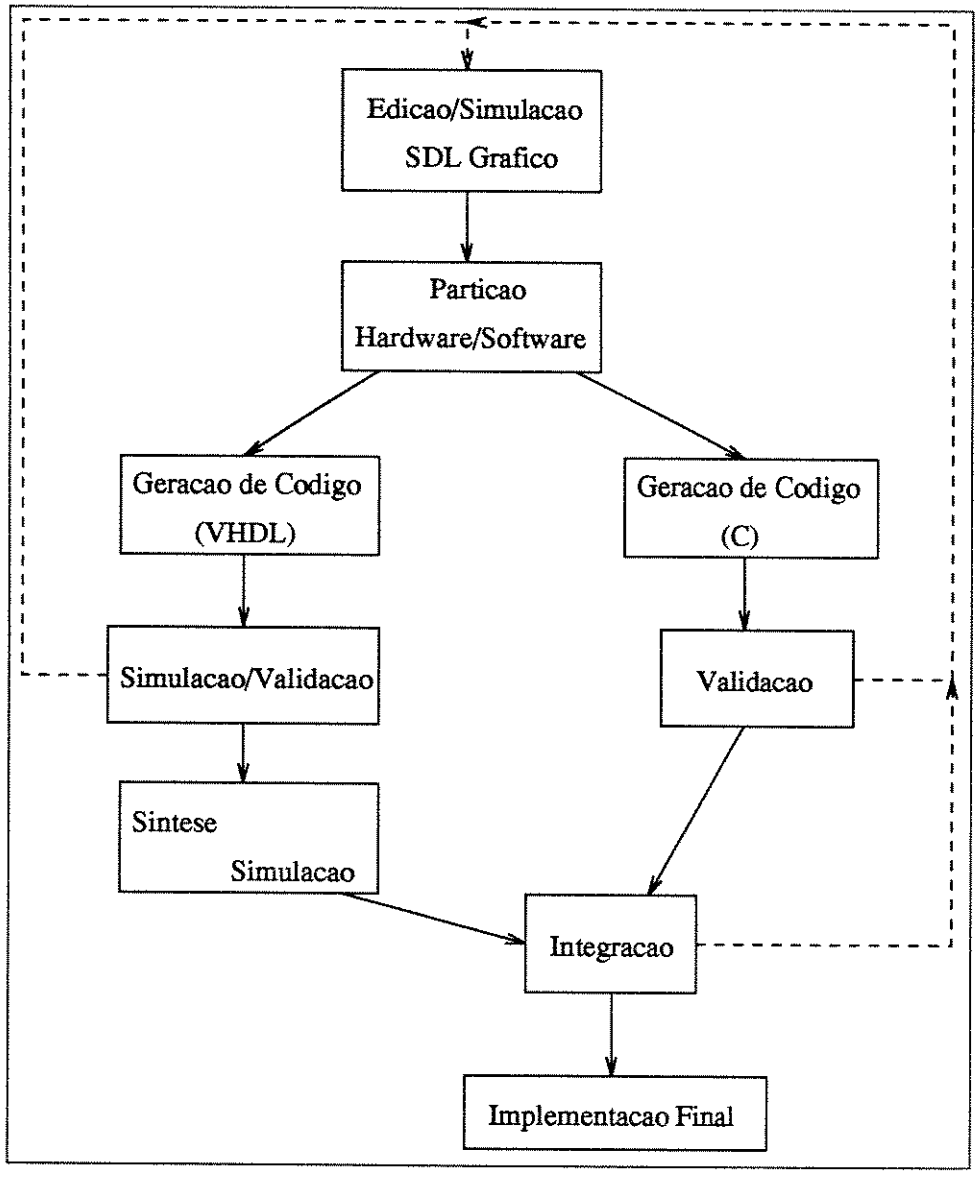


Figura 1.2 Diagrama de blocos: projeto conjunto hardware/software baseado em SDL.

- **Síntese de alto nível:**
  As técnicas de conversão de linguagens de descrição de hardware comportamentais para representações estruturais (**RTL** [7]) representam um grande avanço no projeto de hardware, e são fundamentais para o projeto de sistemas. Linguagens de descrição de hardware podem ser o destino do código gerado a partir de uma linguagem de especificação em um ambiente CASE; podem, então, ser sintetizadas em hardware com o uso de ferramentas de síntese de alto nível disponíveis atualmente em ambientes EDA.

- **Testabilidade**
  A testabilidade deve ser levada em conta no projeto de sistemas. No caso de projeto de circuitos, padrões de teste devem ser mapeados convenientemente da especificação para a simulação. Uma grande parte do trabalho envolvido em um projeto é gasta no teste e validação do circuito, portanto deve-se realizar os padrões de teste automaticamente a partir das suas especificações. Testes iniciais próximos à fase de especificação identificam e geram soluções para problemas que não são propagados para as demais etapas de projeto.
  Uma abordagem comumente usada por projetistas que trabalham com **VHDL** é o projeto tanto do circuito como da "jiga" de testes em **VHDL**; a jiga de teste é tratada como um circuito dedicado para fins de simulação, e pode até vir a ser sintetizada para testes na placa. Esta abordagem pode ser aproveitada pela metodologia de projeto de sistemas, tratando-se a jiga de testes como um subsistema dentro do sistema a ser projetado.

- **Prototipação**
  O projeto de sistemas permite rápida prototipação, ao usar geração de código e ferramentas de síntese. Protótipos são úteis para checar se são satisfeitos os requisitos da especificação e de desempenho, e podem ser usados para realimentar o projeto com informações precisas. Protótipos representam uma versão funcional, real do sistema; não ainda o produto final, mas uma base importante para o seu desenvolvimento.

- **Otimização**
  A geração de código e a síntese automática podem diminuir o desempenho do sistema resultante quando comparado com as soluções "manuais". No caso do hardware, a velocidade tende a diminuir e a área tende a crescer. Não é simples, nem tampouco recomendável, a alteração manual do hardware sintetizado resultante no nível de portas lógicas.
  A tarefa de otimização deve ser feita pelo gerador de código e pelas ferramentas de síntese. Esta tarefa é difícil de ser realizada em ambientes genéricos de síntese; ela obtém sucesso se este ambiente é voltado para alguma aplicação específica. Um ambiente genérico de projeto de sistemas é voltado para casos onde o desempenho não é crítico, para fins educacionais, por exemplo.

- **Integração com componentes disponíveis**
  No contexto do hardware, é desejável que a metodologia de projeto de sistemas permita o uso de componentes "de prateleira". Ferramentas de síntese de alto nível alocam componentes básicos, como multiplexadores e somadores, para gerar uma representação **RTL** a partir de uma descrição comportamental, mas componentes mais complexos, como microprocessadores, não são automaticamente alocados.
  A especificação do sistema deve aceitar tais componentes, e não alterá-los durante o processo de síntese do sistema. Modelos comportamentais para estes componentes podem ser usados para simulação, em um nível mais próximo do hardware (modelos **VHDL**, por exemplo).

- **Documentação**
  Um grande avanço é encontrado na documentação produzida pela metodologia de projeto de sistemas. Ao invés de folhas de esquemático, na maioria das vezes dependente da tecnologia, ou várias linhas

[7] Register Transfer Level

de código **VHDL**, por muitas vezes de difícil leitura e compreensão, a documentação baseada em uma linguagem de especificação, tal como **SDL**, oferece qualidades da representação gráfica e das descrições comportamentais que são independentes da tecnologia a ser usada na implementação final do produto.

Esta dissertação apresenta uma metodologia de projeto de sistemas através do uso de uma linguagem de especificação de sistemas (**SDL**) para o projeto de hardware. Um algoritmo de mapeamento de um sub-conjunto da linguagem **SDL** para **VHDL** sintetizável é apresentado, juntamente com exemplos de aplicação [4].

A abordagem desta dissertação diferencia-se das demais apresentadas na literatura por manter a possibilidade de simulação da descrição ainda no nível da linguagem SDL e por propor uma solução para o modelo de comunicação via troca de sinais da linguagem SDL que seja passível de síntese automática.

Este algoritmo foi implementado em um mapeador **SDL-VHDL**, denominado **Stoht** [8], integrado com a ferramenta **CASE SDT** [9] [3] [1] e com o ambiente **EDA Mentor Graphics** [11] [9] [10].

O programa **Stoht** foi aplicado em dois exemplos e os resultados obtidos foram bastante satisfatórios.

1.3 Conclusões

Propostas de metodologias de projeto auxiliado por computador de sistemas digitais vêm sendo apresentadas na literatura com frequência cada vez maior, buscando oferecer ambientes de desenvolvimento que permitam o projeto de sistemas cada vez mais complexos, envolvendo hardware e software, em prazos cada vez menores. Tais metodologias baseiam-se em paradigmas da engenharia de software e de hardware: utilização de linguagens de alto nível, geração automática de código e síntese de alto nível.

[8] SDL to Hardware Translator
[9] Specification and Description Tool

Capítulo 2

A linguagem SDL

2.1 Origem

SDL é uma linguagem desenvolvida pelo CCITT [1] nos anos 70 com o intuito de auxiliar e padronizar a especificação e descrição de sistemas de comunicação.

Além de poder ser representada na forma de texto, a linguagem SDL possui uma representação dual gráfica, próxima a de um fluxograma ou de um diagrama de fluxo de dados. Esta forma de representação, denominada **GR** [2], facilita a entrada de dados e a compreensão do comportamento do sistema para os profissionais que lidam com SDL. A representação textual (**PR** [3]), por outro lado, é mais adequada para o processamento em compiladores e para a troca de dados entre diferentes plataformas; a linguagem SDL incorpora estas duas qualidades de maneira padronizada. Qualquer sistema SDL pode ser descrito tanto em SDL-GR quanto em SDL-PR. Uma ferramenta CASE [4] que normalmente trabalhe com SDL possui entrada gráfica (SDL-GR) e geração de código SDL-PR.

Desde a primeira recomendação SDL publicada pelo CCITT em 1976, a linguagem vem sendo constantemente revisada e aprimorada, em intervalos de quatro anos, permitindo a inclusão de novas facilidades requeridas pelos seus usuários e tornando-a atualizada. O trabalho apresentado baseia-se na recomendação de 1988 do CCITT [7], implementada no pacote **SDT** [5] [3].

Outras linguagens de especificação similares, como **Estelle** e **Lotos**, são também linguagens padronizadas e de uso difundido, e podem ser utilizadas para a descrição de sistemas. A linguagem **SDL** foi escolhida para a descrição de sistemas neste trabalho devido à existência de ferramentas de edição e simulação **SDL** na Faculdade de Engenharia Elétrica.

2.2 Elementos da linguagem

A linguagem SDL baseia-se em processos concorrentes e independentes contendo máquinas de estados finitas extendidas, e na troca assíncrona de mensagens (sinais) entre estes processos. Filas infinitas de armazenamento de sinais recebidos são alocadas para cada processo concorrente de um sistema SDL.

O modelo da linguagem SDL não leva em consideração implicitamente o tempo entre transições e entre envio e recepção de sinais. Para que intervalos de tempo e atrasos sejam modelados, a linguagem oferece a estrutura **Timer**.

[1] International Telegraph and Telephone Consultative Committee
[2] Graphical Representation
[3] Program Representation
[4] Computer Aided Software Engineering
[5] SDL Tool

A entidade de mais alto nível de uma descrição SDL é o Sistema (**System**). Um sistema contém um ou mais Blocos (**Block**). Blocos, assim como sub-estruturas (**Substructure**), permitem a estruturação da descrição.

Um bloco SDL contém ou sub-estruturas (que, por sua vez, contém um ou mais blocos) ou processos (**Process**). A descrição comportamental na forma de máquinas de estados finitas extendidas são descritas nos processos.

As máquinas de estados extendidas contidas em processos SDL são formadas por estados e transições. As transições são iniciadas com o recebimento de sinais provenientes do ambiente externo ou de outros processos que compõem o sistema. Durante as transições de estado, são permitidas ações sequenciais, como atribuição de valores a variáveis, envio de sinais e desvio do fluxo sequencial de execução (decisões).

Os processos que constituem o sistema comunicam-se através da troca de sinais (**Signal**). Sinais são transportados entre processos de um mesmo bloco através de rotas de sinais (**Signalroute**). Sinais são transportados entre blocos através de canais (**Channel**).

O envio e a recepção de sinais são assíncronos; filas de recepção de sinais são necessárias para permitir esta comunicação assíncrona. Sinais SDL podem carregar parâmetros.

2.3 Exemplo simples de descrição

A Fig. 2.1 mostra um exemplo simples de sistema SDL, contendo um bloco, de nome **MyBlock**, e dois canais, **Channel1** e **Channel2**. O canal **Channel1** transporta o sinal **input_signal** do ambiente externo para o bloco **MyBlock**. O canal **Channel2** transporta o sinal **output_signal** do bloco **MyBlock** para o ambiente externo.

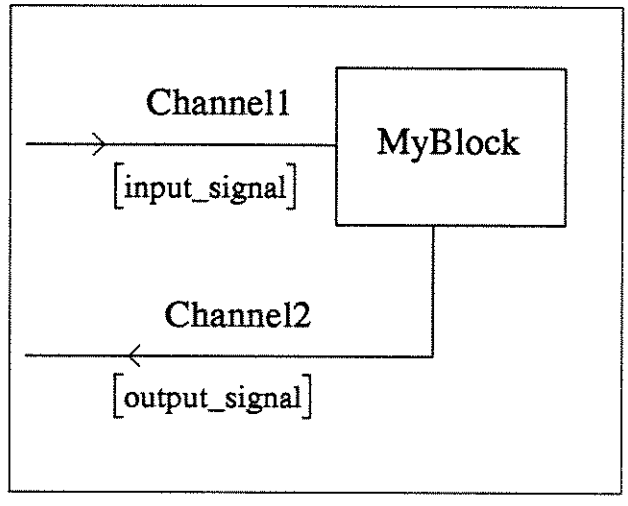


Figura 2.1 Um exemplo simples de sistema SDL

A Fig. 2.2 mostra o bloco **MyBlock**, que consiste de um processo, de nome **MyProcess**, e de duas rotas de sinal, de nomes **Route1** e **Route2**, que transportam os mesmos sinais de **Channel1** e **Channel2**, respectivamente.

A Fig. 2.3 mostra os símbolos SDL usados para a descrição de processos que são usados neste trabalho.

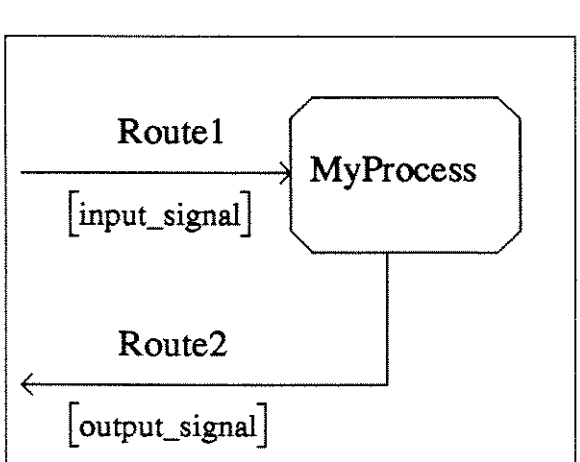


Figura 2.2 Descrição SDL do bloco MyBlock

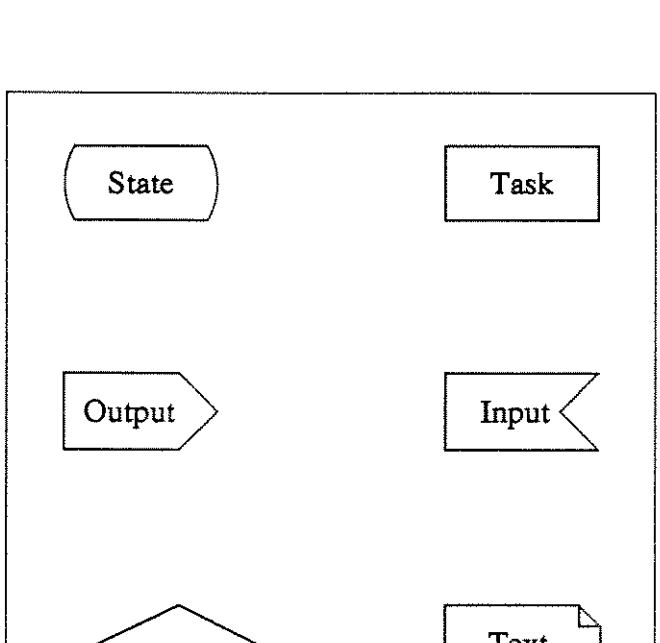


Figura 2.3 Símbolos SDL utilizados para o modelamento de processos

O símbolo de estado (**State**) de uma descrição SDL representa o estado atual em que o processo se encontra. O símbolo de tarefa (**Task**) contém atribuições à variáveis locais do processo. Uma tarefa é executada sequencialmente durante uma transição de estado.

Os símbolos de envio (**Output**) e recepção (**Input**) de sinais representam a comunicação entre processos SDL.

O símbolo de decisão (**Decision**) permite a alteração do fluxo de execução de uma transição baseado no teste do valor de uma variável. O símbolo de texto (**Text**) contém descrições textuais que não possuem símbolo associado, tais como declarações de tipos de dados e de variáveis.

Um exemplo simples de descrição de um processos SDL é mostrado na Fig. 2.4 (processo **MyProcess** declarado na Fig. 2.2).

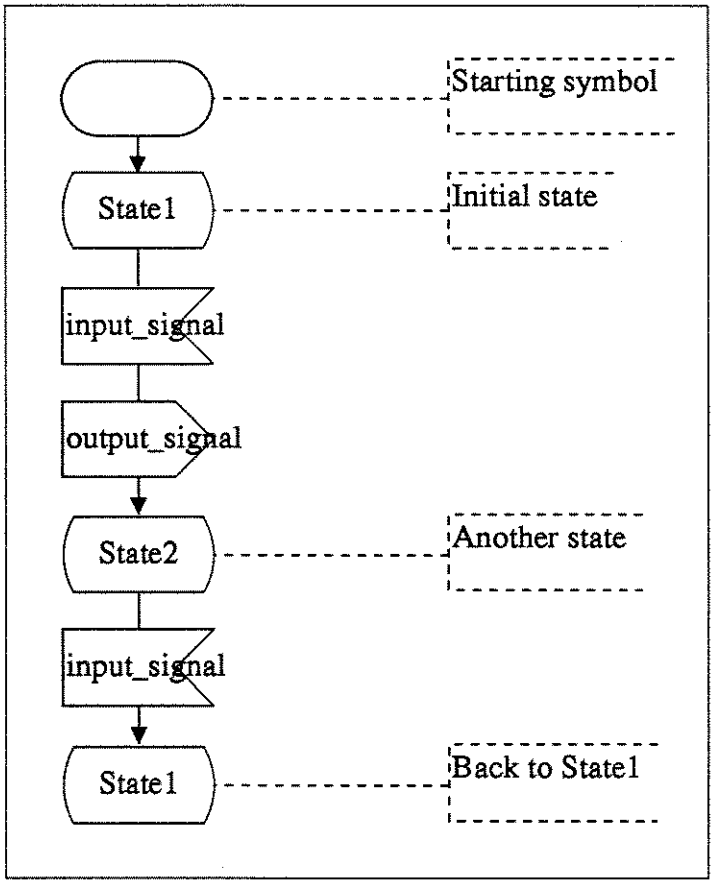


Figura 2.4 Exemplo simples de descrição de um processo SDL

O processo da Fig. 2.4 possui os estados **State1** e **State2**, sendo que o estado **State1** é o estado inicial do processo. Este estado pode realizar uma transição, iniciada pela chegada do sinal **input_signal**. Quando este sinal é recebido, o processo inicia uma transição que realiza um envio do sinal **output_signal** (símbolo **Output**) e uma mudança de estado, para **State2**.

No estado **State2**, o processo novamente inicia uma transição com a chegada do sinal **input_signal**. Entretanto, a transição não inclui um envio de sinal, apenas a mudança de estado para **State1**.

2.4 Estruturação

Um projeto SDL é descrito através do elemento **System**. Este, por sua-vez, pode ser particionado hierarquicamente utilizando as estruturas **Block** e **Substructure**.

Cada bloco pode ser particionado em sub-estruturas (elemento **Substructure**) que, por sua vez, são compostas por blocos de menor hierarquia. Blocos podem conter uma ou mais estruturas do tipo **Process**, que descrevem o comportamento do sistema.

A comunicação entre processos é feita através da estrutura **Signalroute**, enquanto que a comunicação entre blocos é feita através da estrutura **Channel**.

Esta estrutura é exemplificada na Fig. 2.5.

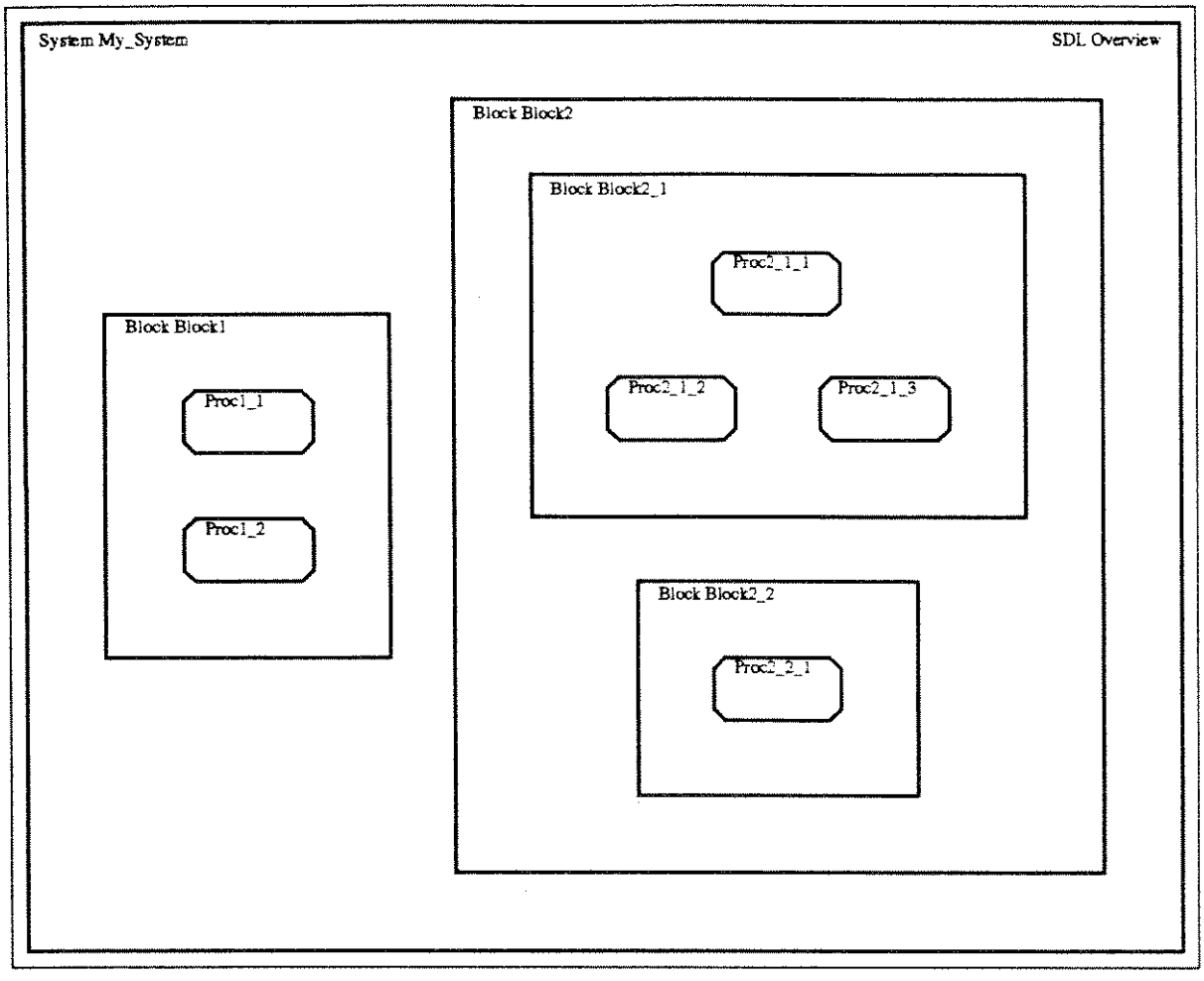


Figura 2.5 Hierarquia de um projeto SDL: Sistema, Blocos e Processos.

Na Fig. 2.5, o sistema **My_System** é composto por dois blocos (**Block_1, Block_2**). O bloco **Block_1** é formado apenas por dois processos (**Proc1_1, Proc1_2**).

O bloco **Block_2** é formado por uma sub-estrutura que contém dois blocos**Block2_1** e **Block2_2**. Esta sub-estrutura não aparece explicitamente na representação gráfica (**SDL-GR**) fornecida pela ferramenta **SDT**; entretanto, ela está presente na descrição textual (**SDL-PR**) do sistema.

O bloco **Block2_1** contém os processos **Proc2_1_1, Proc2_1_2** e **Proc2_1_3**; o bloco **Block2_2**

contém apenas o processo **Proc2_2_1**.

Cada processo contém uma série de estados (**State**) e transições. As transições são causadas pelo recebimento de sinais SDL (**Signal**) e permitem a mudança do estado em que o processo se encontra. Durante uma transição, tarefas (**Task**) podem ser realizadas, decisões (**Decision**) podem ser tomadas, alterando o fluxo de execução da transição, e sinais podem ser enviados a outros processos (**Output**).

No exemplo da Fig. 2.6, o processo representado possui dois estados **flip** e **flop**, ambos sensíveis a dois sinais (**keep** e **toggle**). O recebimento de um sinal **toggle** quando o processo está no estado **flip** faz com que este transite para o estado **flop**, e vice-versa.

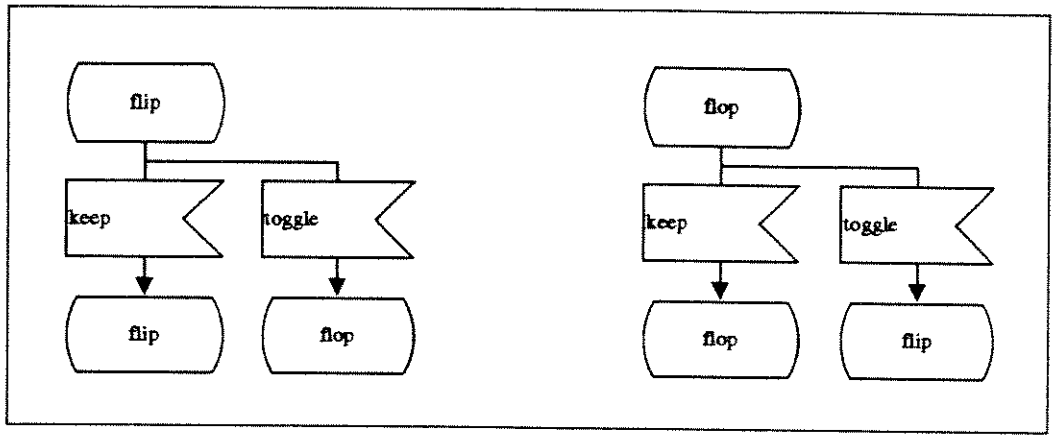


Figura 2.6 Exemplo de máquina de estado finita: representação gráfica em SDL.

Neste exemplo (Fig. 2.6), o corpo da transição é nulo; nenhuma decisão é tomada e nenhuma tarefa é executada. Em estruturas mais elaboradas, o ramo entre os símbolos de recebimento de sinal e próximo estado (representado por uma linha com uma seta na extremidade na Fig. 2.6) poderia compreender ações (tarefas e envio de sinais) dependentes ou não do resultado de decisões.

2.5 Exemplo de processo

A Fig. 2.7 ilustra um processo de uma unidade aritmética simples, contendo um acumulador e operações de soma e subtração.

A declaração da variável **acc** especifica seu tipo (**Integer**) e eventual valor inicial (neste caso, zero). Outra variável é declarada (**param**), que armazenará os dados trazidos por sinais. A declaração de variáveis é feita usando-se a palavra reservada **DCL**, que especifica o nome, o tipo e o eventual valor inicial da variável.

O símbolo de início do processo (**Start**) indica qual a primeira transição que o processo executará; neste caso, a transição significa entrar no estado **espera**.

O estado **espera** é sensível a 5 sinais: **set**, **soma**, **sub**, **leitura** e **halt**. Os três primeiros sinais carregam um parâmetro, de tipo **Integer**.

O processo mantém-se no estado **espera** até que algum destes sinais seja recebido. Quando isto acontece, o processo executa as transições relativas ao sinal que chegou e eventualmente muda de estado.

Se o processo recebe o sinal **set**, o seu parâmetro é armazenado na variável **acc**. Se o processo recebe o sinal **soma**, o seu parâmetro é armazenado em **param**; em seguida o valor de **param** é acrescido ao valor de **acc**. O comportamento para o sinal **sub** é similar ao do sinal **soma**.

Se o processo recebe o sinal **leitura** (sem parâmetros), ele envia um sinal de nome **retorno** carregando

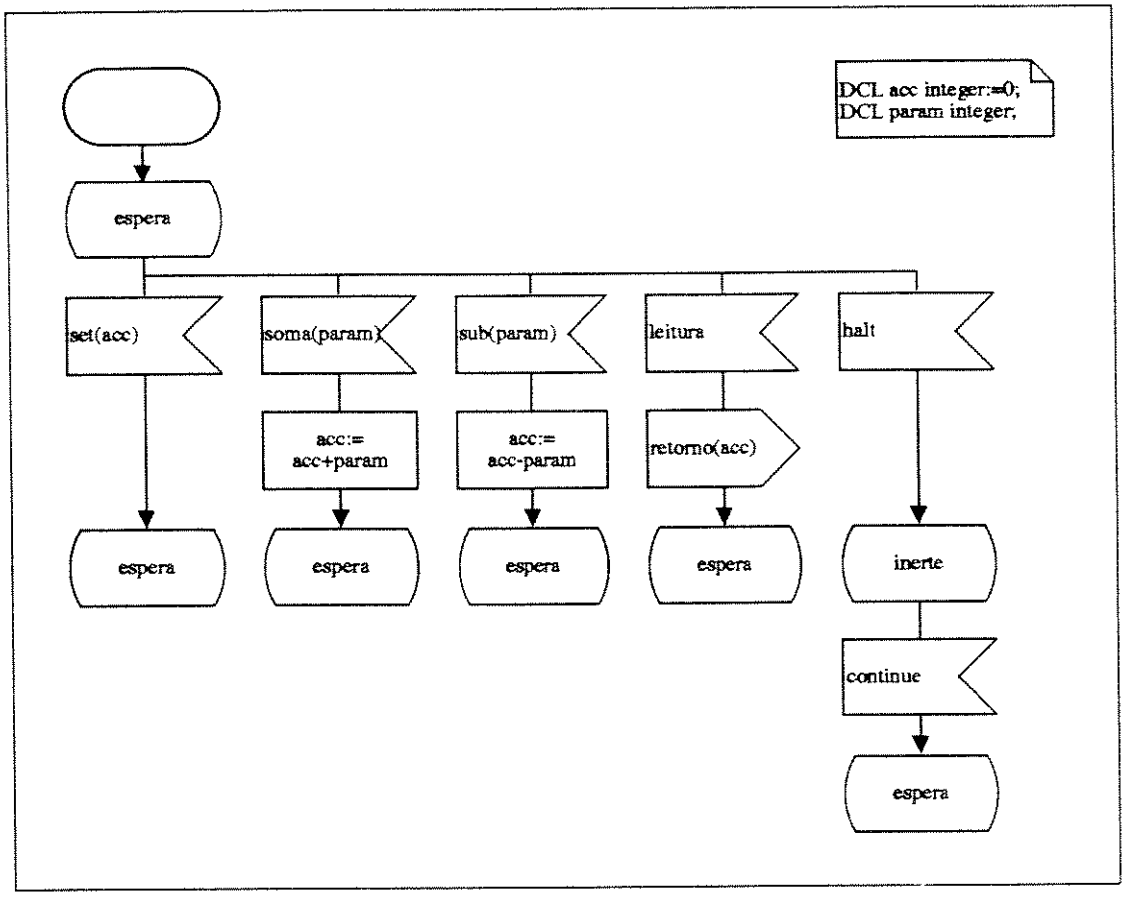


Figura 2.7 Exemplo de processo SDL - unidade aritmética simples.

como parâmetro o acumulador; se um outro processo deseja saber o valor do acumulador ele envia um pedido de leitura e aguarda um sinal de retorno, que contém o valor.

Para os sinais **set**, **soma**, **sub** e **leitura**, o processo continua no estado **espera** após a execução da transição. O recebimento do sinal **halt**, entretanto, leva o processo ao estado **inerte**.

No estado **inerte**, apenas o sinal **continue** é relevante. Qualquer outro sinal que chega ao processo durante este estado é descartado.

2.6 Conclusões

A linguagem **SDL** permite o modelamento de sistemas concorrentes, podendo ser utilizada tanto para o modelamento de software como para o de hardware. Esta linguagem é baseada em um modelo de processos concorrentes que implementam máquinas de estado finitas e comunicam-se através de troca de sinais. A linguagem permite a divisão estrutural do sistema, através da utilização de blocos e sub-estruturas. A representação gráfica da linguagem **SDL** é adequada tanto para a entrada de dados quanto para a documentação.

Capítulo 3

A linguagem VHDL

3.1 Origem

A linguagem VHDL surgiu como um dos resultados do programa VHSIC [1] do Departamento de Defesa Norte-Americano, em meados da década de 1970. É uma linguagem de programação voltada à descrição de circuitos (HDL [2]), contendo características das linguagens de alto nível utilizadas para o desenvolvimento de software, que permitem a descrição comportamental com alto grau de abstração, bem como características de HDL de nível mais baixo, permitindo a descrição estrutural de circuitos próxima da implementação física.

Outras linguagens de descrição de hardware já existiam antes do desenvolvimento do VHDL; entretanto, a especificação do VHDL procurou sanar algumas das deficiências encontradas nestas linguagens:

- Falta de um padrão e de compatibilidade entre as linguagens.
- Baixo nível de abstração.

A linguagem VHDL possui as características desejadas para uma linguagem de descrição de hardware:

- Capacidade de especificação
- Simulação
- Síntese
- Documentação
- Padronização, permitindo a troca de documentos

A linguagem VHDL é uma linguagem de alto nível, com sintaxe semelhante a da linguagem Pascal, permitindo programação estruturada (Loops, decisões do tipo If-Then-Else), estruturas de dados complexas (Records, Pointers, Arrays), tipos de dados abstratos (Integer, Floating-Point), funções, procedimentos, entre outras características encontradas comumente em linguagens de programação destinadas ao desenvolvimento de software. VHDL também permite descrições de nível mais baixo, como por exemplo na descrição das ligações entre componentes.

A linguagem VHDL está se tornando um padrão para a descrição e documentação de circuitos digitais, sendo adotada hoje em dia por quase todos os pacotes de automação de projeto de circuitos EDA [3]. Estes

[1] Very High Speed Integrated Circuit
[2] Hardware Description Language
[3] Electronic Design Automation

pacotes, cada vez mais integrados e poderosos, utilizam VHDL como linguagem de descrição e oferecem ferramentas de simulação e síntese a partir desta linguagem.

A adoção da linguagem VHDL como padrão se deve as suas várias qualidades, tais como:

- **Padronização**: em 1987, a sintaxe VHDL foi padronizada pelo IEEE [4], sob a sigla IEEE-1076 [27], padrão este que é seguido pela quase totalidade dos softwares e publicações relacionadas. As documentações em VHDL-1076 são facilmente intercambiáveis por equipes de projeto, mesmo as que utilizam plataformas de hardware e software distintas.
- **Capacidade de abstração**: A linguagem VHDL permite a descrição comportamental de circuitos, com a adoção de várias construções de alto nível, e permitiu o surgimento de algoritmos de síntese que as mapeam em circuitos realizáveis.
- **Portabilidade**: A descrição do hardware em VHDL é praticamente independente da tecnologia na qual o circuito será mapeado, sendo automática a migração para uma nova tecnologia
- **Suporte à metodologia Top-Down**: a linguagem permite a abordagem partindo do sistema, de hierarquia mais alta, até o nível mais próximo da implementação física que se queira alcançar.

3.2 Descrições em VHDL

A descrição de circuitos em VHDL pode ser abordada de três maneiras:

- **Descrição Comportamental:**
  O comportamento é descrito com os recursos de linguagem de programação estruturada que o VHDL dispõe (tipos de dados, variáveis, loops, decisões).
- **Descrição Estrutural:**
  A descrição é feita através da interconexão entre vários componentes, analogamente ao processo de descrição via captura esquemática.
- **Descrição** *Data-flow*:
  A descrição é feita através de atribuições concorrentes a sinais. Este tipo de descrição não é abordado no contexto deste trabalho.

Um exemplo simples de descrição comportamental em VHDL para um flip-flop RS, e de uma descrição estrutural equivalente é mostrado na Fig. 3.1.

No caso comportamental, o valor lógico '0' é atribuído à saída **q** se a entrada de reset **r** estiver ativa (nível lógico '1'). A outra entrada (**s**) é tratada de maneira análoga, levando **q** ao nível lógico alto. A saída **q_inv** é complementar à **q**.

No caso estrutural, o componente **nand2** possui duas entradas (**in1, in2**) e uma saída (**out1**), e é instanciado duas vezes para formar o flip-flop SR. Às portas dos componentes instanciados (denominados **U1** e **U2** neste exemplo) são associados os sinais **r, s, q** e **q_inv** de maneira semelhante às listas de ligações (*netlists*) de circuitos.

O componente **nand2** pode ser novamente descrito em VHDL (estrutural ou comportamental) até o nível que permita a sua implementação física.

A flexibilidade da linguagem permite a coexistência das três abordagens (comportamental, *data-flow* e estrutural); circuitos podem ser descritos de maneira comportamental em determinados blocos, ao mesmo tempo em que outros blocos são descritos estruturalmente.

Com isso, VHDL torna-se uma linguagem poderosa para a implementação da metodologia *top-down*

[4] Institute of Electrical and Electronic Engineers

```
-- VHDL Comportamental

PROCESS flip_flop_RS (r,s) IS
BEGIN
  IF     (r='1') THEN
    q     <= '0';
    q_inv <= '1';
  ELSIF (s='1') THEN
    q     <= '1';
    q_inv <= '0';
  END IF;
END PROCESS;

-- VHDL Estrutural

U1 : nand2 PORT MAP(
  in1  => r,
  in2  => q_inv,
  out1 => q);
U2 : nand2 PORT MAP(
  in1  => s,
  in2  => q,
  out1 => q_inv);
```

Figura 3.1 Descrições de um flip-flop RS em VHDL Comportamental e em VHDL Estrutural

de projeto; os sistemas são descritos a partir do nível de abstração mais alto e refinados passo-a-passo até que sejam passíveis de implementação física.

Os mais recentes sistemas EDA possuem ferramentas de síntese automática a partir de descrições VHDL comportamentais de alto nível de abstração. Esta automatização, aplicada em conjunto com a metodologia top-down, leva a uma significativa diminuição do tempo de desenvolvimento do projeto de circuitos digitais.

3.3 Fundamentos da Linguagem

3.3.1 Modelo Entidade-Arquitetura

Um sistema modelado em VHDL é descrito através de uma entidade (ENTITY) e uma arquitetura (ARCHITECTURE).

A entidade define a interface deste sistema com o ambiente externo. Portas de entrada e saída do circuito são definidas no escopo da entidade. Uma ENTITY é análoga a um símbolo de um componente utilizado na captura esquemática.

A arquitetura define a funcionalidade do sistema, podendo ser descrita comportamentalmente, estruturalmente ou de maneira híbrida. Definir uma arquitetura é análogo a atribuir um modelo ao símbolo do esquemático.

Como exemplo, uma descrição de uma porta **nand** de duas entradas é apresentada na Fig. 3.2.

```
ENTITY nand2 IS (
  inp1 : IN  BIT,
  inp2 : IN  BIT,
  outp : OUT BIT);
END nand2;

ARCHITECTURE implementacao OF nand2 IS
  SIGNAL interno : BIT;
BEGIN
  outp   <= NOT(interno);
  interno <= (inp1 AND inp2);
END implementacao;
```

Figura 3.2 Descrição completa de uma porta NAND em VHDL Comportamental.

Na declaração da entidade, explicita-se os sinais de entrada e saída do circuito (palavras reservadas IN e OUT) bem como seus tipos (neste caso, tipo BIT).

Na declaração da arquitetura, o nome do modelo e da entidade a ser modelada devem estar presentes. Na Fig. 3.2, a declaração da ARCHITECTURE informa que o modelo **implementacao** da ENTITY **nand2** é descrito. Vários modelos podem ser associados a esta porta **nand2**; cada um deles pode ser descrito em uma arquitetura separada, e a cada modelo deve ser atribuído um nome. Para a simulação ou síntese da entidade, a arquitetura a ser utilizada é referenciada pelo nome.

A área de declarações (*declarative area*) aparece entre a declaração da arquitetura e a palavra reservada BEGIN. Nesta área pode-se definir estruturas de dados, constantes e sinais internos à entidade.

Neste exemplo, definiu-se um sinal (SIGNAL) interno, também do tipo BIT. Este sinal conecta o resultado da operação AND sobre (inp1,inp2) ao NOT que fornece a saída **outp**. O sinal VHDL é análogo a um "fio" de ligação em um circuito, que carrega o valor determinado por uma porta de saída de um componente ou pelo resultado de uma operação lógica aplicada a outros sinais. Denomina-se **driver** o componente ou operação que determina o valor de um SIGNAL VHDL.

No exemplo da Fig. 3.2, o SIGNAL **interno** carrega o valor da operação AND sobre (inp1,inp2); esta operação lógica é o **driver** do SIGNAL **interno**.

Sinais VHDL não podem ter mais de um **driver**. Neste caso há um conflito e deve-se estabelecer regras para sua resolução. Em circuitos digitais, implementações como **wired-or** ou **wired-and** são práticas comuns para a determinação do valor de um sinal com várias fontes. Em VHDL, deve-se definir funções de resolução (**Resolution Functions**) para sinais possuindo mais de um **driver**.

O corpo da arquitetura (entre as palavras reservadas BEGIN e END) implementa o modelo: a saída **outp** é o resultado da operação NOT sobre o resultado do AND entre (inp1,inp2). Os caracteres <= definem uma atribuição; o sinal à esquerda recebe o valor das operações realizadas nos sinais à direita.

3.3.2 Concorrência e Declarações Sequenciais

A linguagem VHDL é concorrente, como o hardware que ela modela; ações podem ser tomadas por diferentes processos em um mesmo instante e independentemente. Deste modo, a ordem em que as atribuições concorrentes de sinal (<=) aparecem no corpo da arquitetura não é relevante.

No exemplo da Fig. 3.2, uma alteração (evento) em qualquer uma das entradas força a análise de (inp1 AND inp2). O resultado desta operação força a atualização do valor da porta de saída **outp**.

Entretanto, construções como *loops*, por exemplo, são sequenciais. A implementação de declarações sequenciais em VHDL é feita internamente aos processos VHDL. A forma de se declarar um processo é ilustrada na Fig. 3.3.

```
PROCESS(signal1,signal2,...,signalN)
BEGIN
  -- corpo do processo
  -- declaracao_sequencial_1
  -- declaracao_sequencial_2
  -- ..
  -- declaracao_sequencial_n
END PROCESS;
```

Figura 3.3 Formato da declaração de Processos em VHDL.

A declaração dos processos se faz dentro do corpo de uma arquitetura.

Um processo é ativado quando um evento ocorre em algum de seus sinais da lista de entrada dada por (signal1,...,signalN), e o seu corpo é executado sequencialmente (na ordem 1,2,..,n no exemplo da Fig. 3.3). A lista de sinais que ativam um processo é denominada *lista de sensibilidade*.

É importante destacar que um processo é concorrente com as demais declarações em VHDL; a diferença

é que sua execução é sequencial, e considera-se que a execução sequencial do processo é instantânea. A arquitetura da porta nand2 pode ser remodelada da maneira ilustrada na Fig. 3.4, utilizando um processo e uma declaração, concorrentes entre si.

```
ARCHITECTURE sequencial_e_concorrente OF nand2 IS
  SIGNAL interno : BIT;
BEGIN
  outp <= NOT(interno);

  PROCESS(inp1,inp2)
  BEGIN
    IF (inp1='1' AND inp2='1') THEN
      interno <= '1';
    ELSE
      interno <= '0';
  END PROCESS;

END sequencial_e_concorrente;
```

Figura 3.4 Modelo da porta NAND utilizando declarações concorrentes e sequenciais.

O processo é ativado concorrentemente quando há um evento em **inp1** ou **inp2**. O corpo do processo (IF-THEN-ELSE) modela a operação AND, e é executado sequencialmente.

As atribuições de sinal (<=) podem ocorrer tanto em escopos sequenciais como concorrentes. Algumas construções (como IF-THEN-ELSE, LOOP) são estritamente sequenciais.

3.4 Sinais e Variáveis

Sinais e variáveis contém informações de mesmo tipo, mas apresentam diferenças básicas:

- Sinais podem ser manipulados tanto em declarações concorrentes como no corpo sequencial de processos. Variáveis só são permitidas em declarações sequenciais.

- Em um contexto sequencial, o valor das variáveis é atualizado imediatamente quando há uma atribuição. A atribuição de um valor a um sinal em um processo somente será efetivada após o término da execução sequencial do processo.
  Considere os processos A e B da Fig. 3.5. No caso do processo A, a atribuição y<=x só será efetivada após a execução de todo o processo; o teste **y='1'** será feito com um valor antigo de **y**, que pode ser diferente de **x**.
  No caso do processo B, a atribuição **y:=x** é processada imediatamente; o valor de **y** é atualizado antes do teste **IF**.
  O uso de sinais em processos deve ser bastante criterioso, para que eventuais inconsistências no modelo sejam evitadas.

- Sinais podem ser visíveis a vários processos. Variáveis são visíveis apenas dentro dos processos onde são declaradas.

- Não existe o conceito de **driver** para variáveis; a analogia com circuitos digitais no caso de variáveis não é a mesma dos sinais. Variáveis são análogas a registros que só são alterados quando ocorre uma atribuição.

```
A : PROCESS(x)                          -- processo A
    BEGIN
      y <= x;                           -- y e x sao sinais
      IF (y='1') THEN
        ...
      END IF;
    END PROCESS;

B : PROCESS(x)                          -- processo B
      VARIABLE y : BIT;
    BEGIN                               -- x = sinal; y =variavel
      y := x;
      IF (y='1') THEN
        ...
      END IF;
    END PROCESS;
```

Figura 3.5 Atribuição de sinais e variáveis dentro de processos

3.5 Sintaxe

O objetivo desta seção é familiarizar o leitor com a sintaxe de um sub-conjunto do VHDL, suficiente para o entendimento dos modelos abordados.

A sintaxe detalhada do VHDL pode ser encontrada em [27].

- Declarações de Sinais e Variáveis

```
SIGNAL sinal_1 : INTEGER;       -- declaracao de sinal
SIGNAL sinal_2 : BIT :='0';     -- decl. de sinal com valor inicial

VARIABLE var_1 : BIT;           -- declaracao de variavel
VARIABLE var_2 : INTEGER := 5;-- decl. de variavel com valor inicial

CONSTANT pi : REAL := 3.1416; -- declaracao de constante
```

- Declarações de Tipos

```
TYPE vetor IS                   -- vetores
  ARRAY (0 TO 3) OF BIT;
TYPE cores IS                   -- tipos enumerados
  (azul, verde, amarelo);
SUBTYPE 5_bits IS               -- sub-tipos
  INTEGER RANGE 0 TO 31;
```

- Atribuições

```
                                -- atribuicoes incondicionais
sinal_a   <= sinal_b;           -- sinais
variavel_a := var_b + 5;        -- variaveis

                                        -- atribuicoes condicionais
out <= '1' WHEN a='1' AND b='1' ELSE    -- WHEN - ELSE
       '0';
```

```
WITH input SELECT                          -- WITH - SELECT
  out <= '0' WHEN '0',
         '1' WHEN OTHERS;
```

- Testes

```
IF a=0 THEN                                -- IF - THEN - ELSE
  -- declaracoes sequenciais
ELSIF a=1 THEN
  -- decl. seq.
ELSE
  -- decl. seq.
END IF;
```

```
CASE a IS                                  -- CASE - WHEN
  WHEN 0 =>
    -- decl. seq.
  WHEN 1 =>
    -- decl. seq.
  WHEN OTHERS => NULL;
END CASE;
```

- Loops

```
FOR a IN 0 TO 10 LOOP                      -- FOR - LOOP
  -- declaracoes sequenciais
END LOOP;
```

- Atributos

```
IF(clock' EVENT AND clock='1') THEN
     ...                                   -- testa borda de subida em "clock"
```

+

3.6 Conclusões

A linguagem **VHDL** vem se tornando um padrão "de fato" para o projeto de circuitos digitais em ambientes de automação de projeto EDA. A linguagem oferece três níveis de representação (comportamental, *data-flow* e estrutural) que podem coexistir em uma descrição e ser usados tanto para a simulação quanto para a síntese de alto nível do projeto. Uma das formas de modelamento comportamental em **VHDL** consiste em processos concorrentes contendo declarações sequenciais.

Capítulo 4

Síntese de Alto Nível e VHDL

4.1 Síntese de alto nível

Síntese consiste no mapeamento de uma representação conveniente ao projetista de circuitos em uma representação de abstração menor, tendo em vista a implementação física.

A síntese de alto nível parte de uma descrição comportamental e fornece uma estrutura (circuito) que implementa este comportamento.

Procurando acompanhar o aumento da capacidade de integração de circuitos e a necessidade do desenvolvimento de sistemas cada vez mais complexos, os algoritmos de síntese vêm constantemente evoluindo. O objetivo da síntese de alto nível (**HLS** [1] [5]) é permitir a especificação de circuitos em um nível comportamental, algorítmico, através do mapeamento automático em uma implementação física equivalente.

A síntese de alto nível trabalha com três níveis de representação [6]:

- Comportamental: Nível mais alto de representação. O circuito é descrito de maneira algorítmica, abstraindo-se de detalhes de implementação.
- Transferência entre registros (**RTL** [2]): O circuito é descrito estruturalmente por registros, multiplexadores, operadores aritméticos e outros componentes.
- Lógico: O circuito é descrito estruturalmente por portas lógicas ou células básicas da tecnologia utilizada na implementação física.

Os algoritmos de síntese de alto nível usualmente envolvem várias tarefas [25]:

- **Compilação**
  Consiste na leitura, interpretação e validação da descrição comportamental, e na elaboração de uma representação interna desta descrição em um formato adequado. Esta representação é usualmente encontrada na forma de grafos para os fluxos de dados e de controle.
- **Escalonamento**
  Uma das tarefas fundamentais da síntese, o escalonamento é responsável pela determinação dos passos de controle para cada operação da descrição comportamental. Como resultado da síntese um circuito de controle é gerado, sendo a máquina de estados deste controlador determinada na etapa de escalonamento.

[1] High Level Synthesis
[2] Register-Transfer Level

- **Alocação**
  Alocação é a etapa que determina quais componentes de hardware são utilizados para realizar cada operação comportamental. A alocação pode ser dividida em três tipos: alocação de unidades funcionais (somadores, comparadores), alocação de registradores (para variáveis), e alocação de interconexões (barramentos e multiplexadores).

Após a síntese de alto nível, uma representação RTL é gerada para a descrição comportamental, contendo geralmente uma parte operativa (*data-path*) e uma parte de controle.

Esta representação RTL é posteriormente mapeada em estruturas lógicas da tecnologia-destino através de ferramentas de síntese lógica.

4.2 Síntese VHDL

A linguagem VHDL é adequada para a síntese de alto nível, pois permite tanto descrições comportamentais em alto nível quanto estruturais e declarações concorrentes.

Considere o seguinte exemplo, composto de uma atribuição em VHDL:

```
-- VHDL comportamental
     x := a + b;
```

Supondo que os operandos são inteiros de 4 bits, uma descrição **RTL** equivalente possível é a alocação de um componente de adição de 4 bits, neste caso denominado **4_bit_adder**:

```
-- VHDL estrutural
     U1 : 4_bit_adder PORT MAP(
             op1=>a,
             op2=>b,
             result=>x);
```

Uma representação lógica resultante possível é o cascateamento de somadores de 1 bit com *carry*, formados por portas XOR, AND e OR. Abaixo estão representadas as portas que geram a soma para o bit mais significativo de **x**:

```
X3a : xor2 PORT MAP(a[3],       b[3],     interno[3]);
X3b : xor2 PORT MAP(interno[3], carry[2], x[3]);
```

Apenas um sub-conjunto da linguagem VHDL é passível de síntese atualmente, pois muitos recursos da linguagem VHDL são orientados à simulação.

4.3 AutoLogic

AutoLogic é a ferramenta de síntese desenvolvida pela *Mentor Graphics Corporation*, parte do seu software integrado de EDA.

O AutoLogic processa descrições em VHDL estrutural, comportamental, *data-flow* ou misto e fornece, após a síntese, listas de ligações ou diagramas esquemáticos com componentes da biblioteca genérica de componentes lógicos adotada pela Mentor Graphics.

Esta biblioteca, denominada **GENLIB**, contém componentes básicos utilizados em circuitos digitais, como portas lógicas (and, not, xor), flip-flops (D, JK), multiplexadores, somadores, entre outros.

O VHDL suportado pelo AutoLogic é um subconjunto do VHDL padrão **IEEE** [27]. Várias primitivas de alto nível são suportadas por esta ferramenta.

Tipo	*Suportado*	*Comentário*
Integer	Sim	
Bit	Sim	
Bit_vector	Sim	
Boolean	Sim	
Arrays	Sim	Uni ou bidimensionais estáticas
Std_Ulogic	Sim	Lógica de 4 valores (0, 1, X, Z) padronizada pelo IEEE
Qsim_state	Sim	Lógica de 4 valores (0, 1, X, Z) fornecida pela Mentor Graphics
Access	Não	
File	Não	
Floating Point	Não	
Real	Não	
Physical	Não	
String	Não	
Record	Não	

Tabela 4.1 Tipos de dados aceitos pelo AutoLogic

4.3.1 Subconjunto AutoLogic-VHDL para Síntese

A linguagem VHDL permite abstrações que não são passíveis de implementação direta em hardware, ou que requerem circuitos extremamente complexos.

Exemplos de estruturas não implementáveis são:

- **Asserts**, que emitem avisos quando uma condição pré-estabelecida não foi satisfeita (um tempo de *setup* de um flip-flop não respeitado, por exemplo). É uma estrutura cuja função é importante para a etapa de simulação.

- Tipos de dados **Access** (apontadores dinâmicos) ou **File** (arquivos do tipo texto).

Aritmética de ponto flutuante é um exemplo de abstração implementável, mas que tem custo de implementação muito alto em termos de número de circuitos.

Os elementos VHDL aceitos pelo AutoLogic como entrada para a síntese de circuitos é apresentada abaixo.

Tipos de Dados

Os tipos de dados com que o AutoLogic trabalha estão sumarizados na Tabela 4.1.

Operadores

Os operadores com que o AutoLogic trabalha estão sumarizados na Tabela 4.2.

Tipo de operadores	*Operadores*	*Suportados*
Unários	+,-,abs	Sim
Binários	*,+,-	Sim
Lógicos	and, or, nand, nor, xor, not	Sim
Deslocamento	sll, srl, sra, rll, rrl	Sim
Relacionais	=,>,>=,<,<=,/ =	Sim
Concatenação	&	Sim
Potenciação	**	Não
Divisão	/, mod, rem	Não

Tabela 4.2 Operadores aceitos pelo AutoLogic

Construções VHDL

As construções aceitas e as não aceitas pelo AutoLogic estão sumarizadss na Tabela 4.3.

Atributos

Os atributos de *arrays* (HIGH, RANGE, LEFT, etc) são todos suportados, bem como os de tipos enumerados (SUCC, HIGH, POS).

Os atributos de **sinais** suportados pelo AutoLogic são:

- EVENT
- LAST_VALUE
- STABLE

Uma restrição para estes atributos é que eles devem ser somente associados a sinais que serão utilizados como **clocks**.

4.3.2 Mapeamento na Biblioteca GENLIB

O resultado da síntese de alto nível de uma descrição em VHDL comportamental é uma descrição estrutural **RTL**, utilizando componentes lógicos básicos da biblioteca GENLIB fornecida pela Mentor Graphics.

Esta biblioteca é independente da tecnologia destino do circuito. Contém componentes básicos de projeto digital, como portas lógicas, registros e multiplexadores, que são passíveis de implementação em qualquer tecnologia.

A otimização a nível de portas lógicas da descrição RTL obtida somente é realizada tendo em vista o mapeamento para uma tecnologia específica. São aproveitadas as características e vantagens de cada tecnologia ao mesmo tempo em que são utilizados algoritmos de minimização de máquinas de estado e funções lógicas. A ferramenta **AutoLogic** realiza tanto a síntese de alto nível quanto a síntese lógica.

4.4 Conclusões

A síntese de alto nível promove a conversão de uma descrição algorítmica de um circuito em uma representação estrutural passível de implementação física. Ferramentas de síntese de alto nível já são disponíveis em ambientes de automação de projetos EDA, na maioria das vezes baseados na linguagem de

Construção	*Suportada*	*Comentário*
AFTER	Sim	Sintaxe aceita mas **ignorada** para efeitos de síntese
ASSERT	Sim	Sintaxe aceita mas **ignorada** para efeitos de síntese
ARCHITECTURE	Sim	
BLOCK	Sim	
CASE	Sim	
COMPONENT	Sim	Declaração e instanciação
CONSTANT	Sim	
ENTITY	Sim	
FUNCTION	Sim	Declaração e chamada
IF	Sim	
LIBRARY	Sim	
LOOP - FOR	Sim	
NEXT	Sim	
PACKAGE	Sim	Declaração e corpo
PROCEDURE	Sim	Declaração e chamada
PROCESS	Sim	
SIGNAL	Sim	Declarações e atribuições concorrentes e sequenciais
SUBPROGRAM	Sim	Declaração e corpo
TYPE	Sim	Declaração e conversão
USE	Sim	
VARIABLE	Sim	Declaração e atribuição
WAIT	Sim	
ALIAS	Não	
CONFIGURATION	Não	
DISCONNECT	Não	
WHILE	Não	
GENERATE	Não	
GENERIC	Não	
LOOP - WHILE	Não	

Tabela 4.3 Construções aceitas pelo AutoLogic

descrição de hardware VHDL. A ferramenta **AutoLogic** é um exemplo de sintetizador que trabalha com descrições comportamentais de um sub-conjunto da linguagem VHDL.

Capítulo 5

Metodologia de descrição comportamental

5.1 Introdução

Neste trabalho propõe-se a descrição comportamental de circuitos digitais realizada a partir da representação gráfica SDL-GR. A partir desta representação, é gerado o código VHDL, padrão IEEE-1076, já contendo restrições de síntese sendo, portanto, passível de implementação direta com o uso de ferramentas de síntese.

A fim de tornar a conversão de SDL para VHDL eficiente e realizável, é preciso delimitar um subconjunto da linguagem SDL e estabelecer restrições no seu uso.

A metodologia proposta neste trabalho utiliza os elementos básicos e mais frequentemente encontrados em descrições SDL, prevendo assim a adaptação natural de usuários de SDL a este subconjunto.

Para propiciar maior facilidade de leitura, ao longo deste capítulo as palavras reservadas SDL são escritas em letras minúsculas, com a inicial em maiúsculas, enquanto as palavras reservadas VHDL são escritas completamente em letras maiúsculas.

5.2 Elementos SDL suportados

O subconjunto da linguagem SDL utilizado para a descrição de circuitos constitue-se de:

- **System**
  A descrição SDL deve partir de um sistema e de sua comunicação com o ambiente externo. Um sistema contém blocos (**Block**), canais (**Channel**), sinais (**Signal**) e definições de dados.

- **Block**
  A partição em blocos permite a divisão estrutural do sistema. Um bloco contém processos (**Process**), sub-estruturas (**Substructure**), rotas de sinais (**Signalroute**), sinais (**Signal**) e definições de dados.

- **Substructure**
  A partição em sub-estruturas também permite a estruturação do sistema. Apenas a sub-estrutura de bloco é suportada pelo algoritmo. Uma sub-estrutura contém blocos, canais, sinais e definições de dados.

- **Process**
  O comportamento do sistema é descrito através de processos SDL. Processos são suportados pelo algoritmo, com a restrição de apenas uma instância por processo; a criação dinâmica de processos não é permitida. Processos contém estados (**State**), recepção e envio de sinais (**Input**, **Output**), sinais contínuos (**Continuous signal**), tarefas (**Task**), decisões (**Decision**), definições de dados e declarações de variáveis (**Variable**).

- **Signal**
  São suportados o envio e recepção de sinais entre processos. A abstração de filas infinitas de chegada para sinais SDL não é suportada pelo algoritmo; filas finitas são utilizadas. Um sinal precisa ser conduzido por canais e rotas de sua fonte até seu destino. Sinais contínuos e condições (**Enabling Conditions**) também são suportadas.

- **Channel**, **Signalroute**
  Devem ser usados para estabelecer caminhos de comunicação entre processos.

- **Variable**
  Variáveis comuns e reveladas (**Revealed**) são suportadas.

- **Tipos de dados** Dados pré-definidos são inteiros (**Integer**) e booleanos (**Boolean**). Faixas de inteiros são definidos com a declaração **Syntype** do SDL. Vetores uni-dimensionais de dados booleanos são definidos com a declaração **Newtype**. Constantes são definidas com a declaração **Synonym**.

- **State**
  Estados fazem parte do modelo de processos máquinas de estado finitas extendidas oferecido pela linguagem SDL.

- **Decision**
  Decisões durante as transições de estado são suportadas.

- **Task**
  As tarefas suportadas permitem atribuições, operações lógicas e operações aritméticas entre variáveis.

5.3 Restrições de síntese

Várias facilidades da representação em SDL têm a implementação em hardware dificultada pela complexidade e consequente aumento no número de componentes do projeto final. Exemplos típicos são a criação em tempo real de instâncias de processos, matemática em ponto flutuante e filas ilimitadas para recebimento de sinais.

Restrições de projeto devem ser estabelecidas para que o resultado seja realizável e eficiente em tamanho e velocidade. Foram adotadas as seguintes restrições de síntese neste trabalho:

- Os sistemas devem ser síncronos.
  Um relógio comum ao sistema é definido automaticamente no processo de mapeamento. As transições de estados SDL contendo tarefas, decisões e envio de sinais são realizadas em um ciclo deste relógio. A recepção de sinais em processos SDL é avaliada na borda de subida do relógio. Sistemas síncronos são menos sensíveis a problemas de temporização da implementação final que sistemas assíncronos.

- As filas de recepção de sinais SDL são limitadas.
  A implementação do envio e recepção de sinais SDL é feita através de protocolos pré-definidos. Estes protocolos são alocados, para cada processo, pelo projetista.

- Processos SDL devem ter apenas uma instância; a criação de novas instâncias de processos (primitiva **Create**) não é suportada
- Aritmética de ponto flutuante não é suportada; os tipos de dados suportados são: inteiros, booleanos e vetores finitos destes dois tipos. Inteiros devem ser limitados no mínimo e máximo valor que podem assumir. Adição e subtração de inteiros são suportadas; multiplicação, divisão, módulo e resto não são suportadas. Operações lógicas são suportadas para dados booleanos.

5.4 Protocolos de comunicação

Uma das mais importantes tarefas necessárias ao mapeamento de SDL em hardware é o modelamento da comunicação SDL, baseada em envio e recepção assíncrono de sinais. A linguagem SDL prevê comunicação assíncrona e filas ilimitadas para sinais recebidos por processos; estas abstrações são dificilmente implementáveis em um circuito real.

Em vários casos, apenas um protocolo simples de comunicação é suficiente para o funcionamento de certas partes de um projeto digital. A implementação mais simples possível não utiliza nenhum tipo de armazenamento para sinais que chegam em um processo SDL.

Outras partes, entretanto, podem requerer esquemas mais complexos de comunicação. O algoritmo de mapeamento proposto é flexível neste sentido, permitindo que o projetista tenha liberdade de utilizar implementações de baixo custo para processos que se comunicam de maneira simples e circuitos complexos para processos que necessitem de recursos de comunicação mais elaborados, como filas de eventos.

Na implementação sugerida neste trabalho, o protocolo de comunicação fica separado dos processos que se comunicam, como representado na Fig. 5.1. Os protocolos possuem uma interface comum com o processo que envia e com os que recebem sinais.

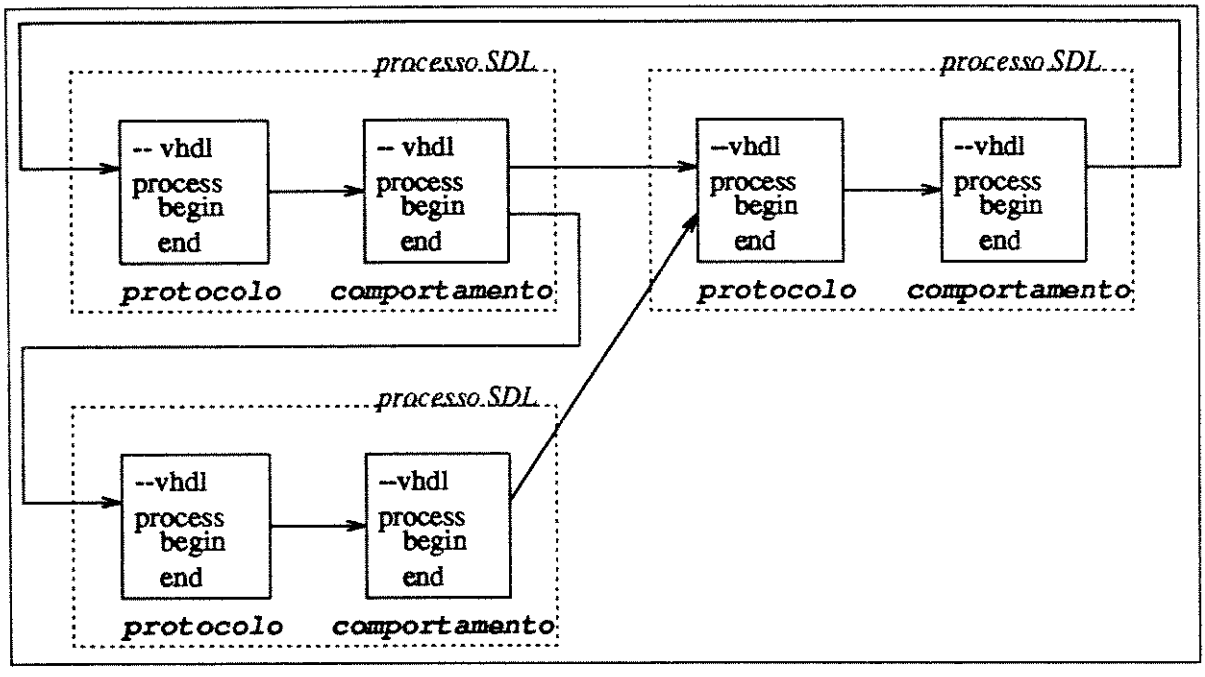


Figura 5.1 Implementação da comunicação inter-processos SDL através do uso de um protocolo.

Procura-se, assim, desvincular a comunicação do comportamento do sistema.

Um protocolo recebe requisições de envio de sinal SDL mapeadas em primitivas VHDL, mais eventuais parâmetros a serem passados pelo sinal, e aciona o processo VHDL recipiente do sinal de acordo com sua política de atuação.

Foram definidos três tipos de protocolo para a síntese da comunicação via troca de sinais SDL. Estes protocolos são detalhados a seguir, usando como referência o processo SDL exemplo da Fig. 5.2. Este processo possui um estado (**stat0**) sensível a três sinais de entrada. Cada um destes sinais habilita uma transição diferente, que acarreta na execução de uma tarefa diferente e na volta ao estado inicial.

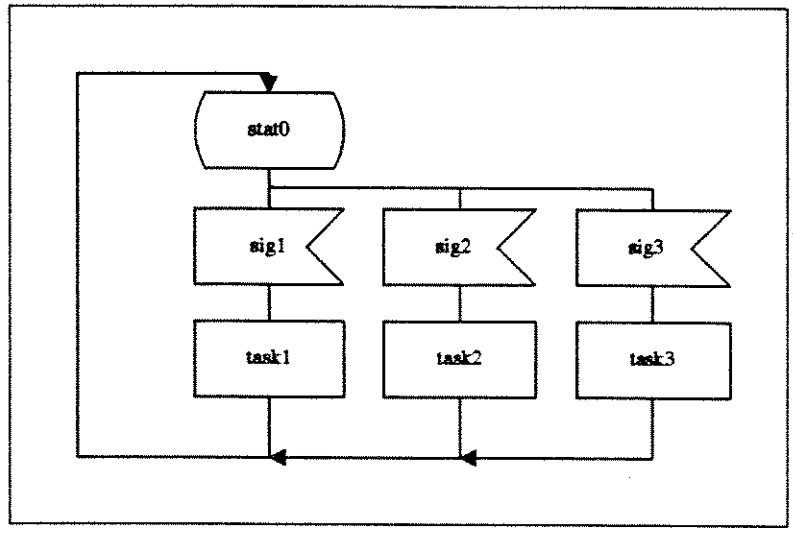


Figura 5.2 Exemplo de recepção de sinais para a ilustração dos protocolos de comunicação.

1. **Comunicação sem fila**
   É o protocolo mais simples e que requer menor área de hardware para ser implementado.
   Neste protocolo, apenas um sinal dentre todos os recebidos é armazenado. Se vários sinais são recebidos simultaneamente, apenas um deles é tratado; os outros são descartados.
   Cabe ressaltar neste ponto que os sinais SDL tratados pelo algoritmo de mapeamento são tornados *síncronos* ao relógio global criado automaticamente na descrição VHDL. Deste modo, a chegada simultânea de sinais acontece quando mais de um sinal é recebido durante um período deste relógio global. A avaliação da recepção de sinais é realizada na borda de subida deste relógio.
   Para o caso da Fig. 5.2, prioridades fixas devem ser atribuídas aos sinais **sig1**, **sig2** e **sig3** de forma que o sinal mais prioritário sempre seja tratado quando da ocorrência de chegadas simultâneas de mais de um sinal.
   Supondo que o sinal **sig1** é declarado de maior prioridade, **sig2** de prioridade intermediária e **sig3** de menor prioridade, a chegada simultânea destes três sinais resulta na execução da tarefa **task1** somente. As tarefas **task2** e **task3** não são executadas pois os sinais **sig2** e **sig3** são descartados.

2. **Comunicação com fila de uma posição para cada sinal**
   Este protocolo resolve o conflito apresentado acima com a utilização de uma fila de uma posição para cada sinal que possa ser recebido pelo processo. No caso anterior, uma posição era compartilhada por todos os sinais.
   Supondo a chegada simultânea dos três sinais do exemplo da Fig. 5.2, com este protocolo consegue-se a execução das três tarefas em sequência (uma para cada período do relógio do sistema).
   A prioridade determinada para cada sinal indicará a sequência de execução das tarefas. Considerando

as mesmas prioridades definidas no item anterior, **task1** é executada no primeiro ciclo de relógio, **task2** no segundo e **task3** no terceiro.
Novamente, o sinal de chegada mais recente é o que ocupa a fila. Se no exemplo anterior ocorre a chegada de mais um sinal **sig3** durante a execução da tarefa **task2** iniciada por **sig2**, o sinal **sig3** mais antigo é descartado.

3. **Comunicação com fila finita, 'n' posições**
Este protocolo permite a armazenagem de **n** sinais pendentes em uma fila ao invés de apenas um. O valor de **n** deve ser atribuído pelo projetista levando em conta dados de simulação, como quantidade de sinais trocados com o processo em questão em situação de pior caso.
As **n** posições da fila podem ser preenchidas por sinais de tipos distintos ou por sinais de mesmo tipo. Considerando ainda o exemplo da Fig. 5.2 e uma fila de tamanho **n**=3, a chegada simultânea dos sinais **sig1**, **sig2** e **sig3** leva ao tratamento destes três sinais na sequência **sig1**, **sig2**, **sig3** (ordem de prioridade definida pelo projetista). Caso um outro sinal **sig3** chegue durante a execução de **task2**, ele ocupa o lugar na fila deixado pelo sinal **sig1**, pois **sig1** já foi tratado.
Nota-se a diferença em relação ao protocolo 2 através deste exemplo. No caso do protocolo 2, a segunda ocorrência do sinal **sig3** retirava a primeira ocorrência não tratada deste sinal. Em ambos os casos existem 3 posições na fila, mas no caso do protocolo 2 as posições não podem ser compartilhadas por sinais de tipos diferentes.

O esquema do funcionamento dos protocolos 1, 2 e 3 é ilustrado na Fig. 5.3.

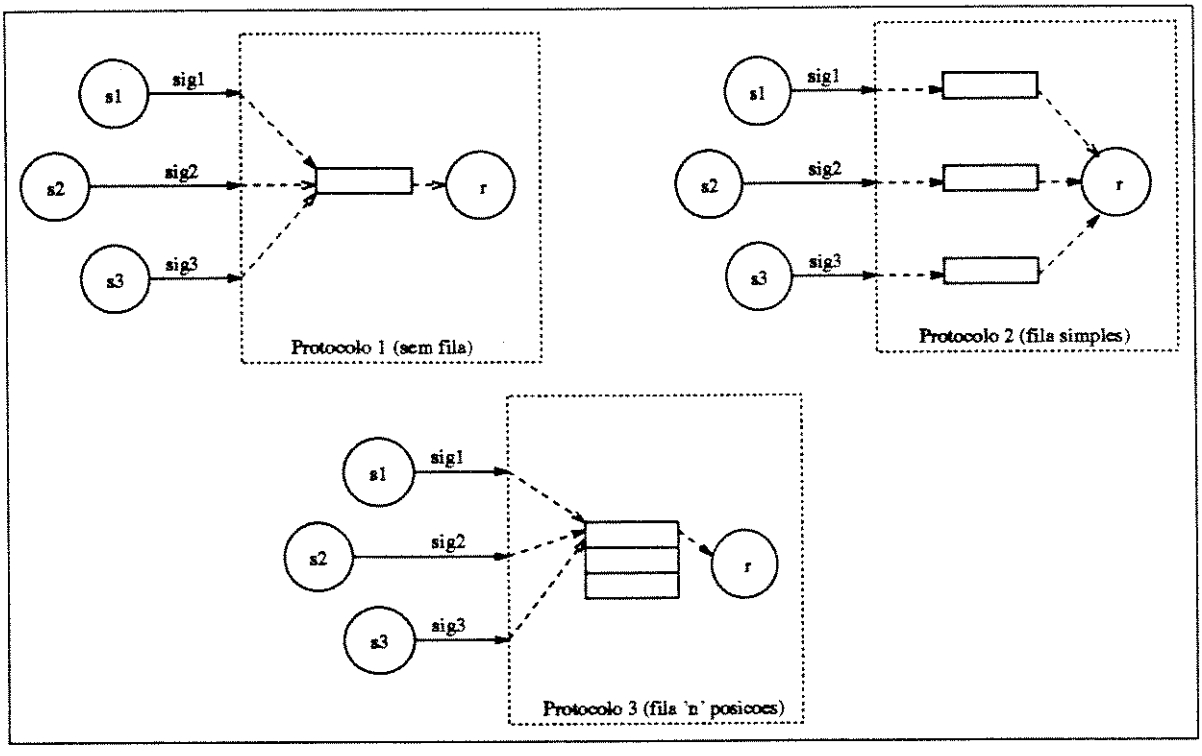


Figura 5.3 Esquema de funcionamento dos protocolos de comunicação: Sem fila (1), Fila Simples (2) Fila Finita (3).

O nível de complexidade dos circuitos utilizados para a comunicação inter-processos depende do protocolo escolhido; o protocolo mais simples (1) é implementado com apenas um flip-flop e uma porta XOR

descritos em VHDL estrutural para cada sinal SDL, descrevendo um conversor de transição (inversão lógica) para pulso como mostra a Fig. 5.4, enquanto que o mais complexo (3) utiliza vetores de registros e contadores descritos em VHDL comportamental. O protocolo 2 tem complexidade intermediária, e também é descrito em VHDL comportamental.

Como existe uma separação entre protocolos de comunicação e processos, e como as interfaces dos pro-

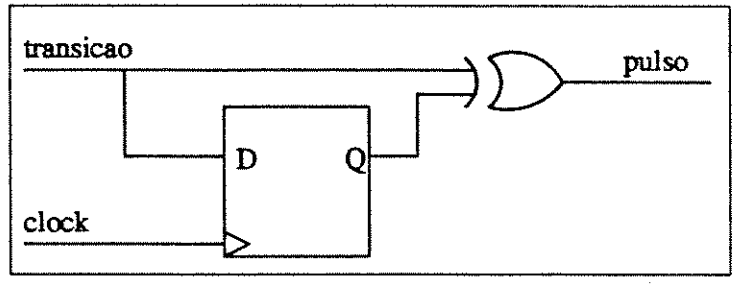


Figura 5.4 Conversor transição/pulso implementado no protocolo 1

tocolos são idênticas, novos protocolos podem ser adicionados ao mapeador SDL-VHDL com facilidade. Implementações otimizadas (em VHDL estrutural a nível de portas lógicas, por exemplo) podem substituir implementações mais caras.

5.5 Interface protocolo-processo

Um protocolo é associado a cada processo SDL que recebe sinais durante o mapeamento. Este protocolo é mapeado em um componente VHDL (ENTITY e ARCHITECTURE) que tem sua interface (portas VHDL **IN/OUT**) padronizada com os processos envolvidos na comunicação.

Uma ENTITY associada a um protocolo de comunicação apresenta as seguintes portas de entrada e saída:

- Portas **clock, reset**
  Relógio e reset do sistema, respectivamente.

- Portas de entrada de prefixo **rec_**
  Para cada sinal recebido pelo processo SDL há uma porta de entrada VHDL correspondente com este prefixo seguido do nome do sinal.
  Estas portas são utilizadas para representar, em VHDL, a ação de recepção de sinal SDL.

- Portas de entrada de prefixo **rec_parN_**
  Para cada sinal SDL que carrega $N > 1$ parâmetros são alocadas N portas VHDL correspondentes, com este prefixo seguido do nome do sinal. A numeração dos parâmetros (N) é feita automaticamente pelo mapeador, de acordo com a ordem de declaração dos parâmetros carregados pelo sinal.
  Estas portas são utilizadas para representar os dados contidos em um sinal SDL.
  As portas VHDL de prefixo **rec_** e **rec_parN_** são portas de entrada do protocolo. Estas portas são escritas por um processo VHDL quando há um envio de sinal (primitiva SDL **Output**).

- Portas de saída de prefixo **p_send_**
  É alocado o mesmo número de portas VHDL encontrado para o caso do prefixo **rec_**. Entretanto, as portas de prefixo **p_send_** são portas VHDL de saída, com destino no componente VHDL que representa o comportamento do processo SDL.

- Portas de saída de prefixo **p_send_parN_**
  É alocado o mesmo número de portas VHDL encontrado para o caso do prefixo **rec_parN_** Entretanto, as portas de prefixo **p_send_parN_** são portas VHDL de saída, com destino no componente VHDL que representa o comportamento do processo SDL.
  Os prefixos **rec_** e **p_send_** indicam, respectivamente, recepção e envio de sinais do ponto de vista do componente VHDL que implementa o protocolo de comunicação. As portas de prefixo **rec_** conectam-se ao componente VHDL que pretende enviar um sinal SDL; as portas de prefixo **p_send_** conectam-se ao componente VHDL que recebe tal sinal. O protocolo implementa a interface entre estes dois componentes.

Do ponto de vista dos componentes VHDL que representam os processos SDL que estão trocando sinais, os seguintes prefixos são utilizados: a porta de **send_** representa a requisição de envio de sinal no processo transmissor, sendo conectada à porta de prefixo **rec_** do protocolo; **p_rec_** representa a recepção de sinal no processo receptor, sendo conectada à porta de prefixo **p_send_** do protocolo. A Fig. 5.5 ilustra a convenção da nomenclatura utilizada para as portas VHDL de entrada e saída geradas pelo algoritmo para dois processos SDL que se comunicam via troca de sinais.

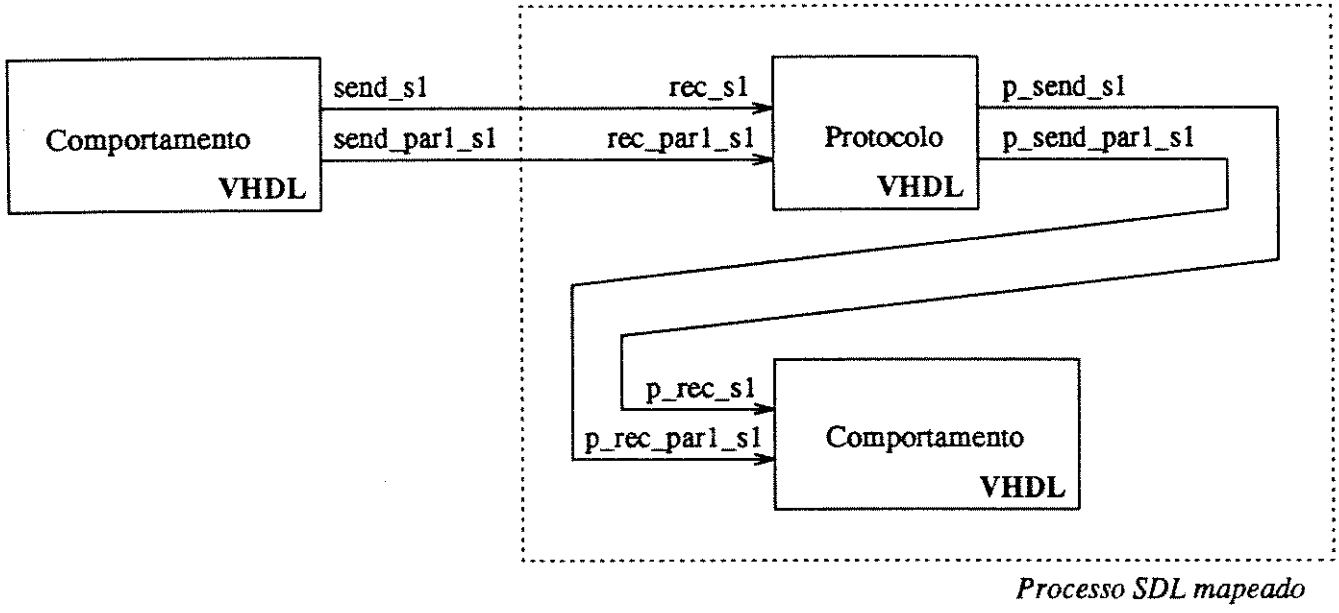


Figura 5.5 Convenção dos nomes de portas VHDL de entrada e saída utilizadas na comunicação inter-processos

Um componente VHDL que pretende enviar um sinal SDL faz uma requisição de envio aos protocolos dos processos recipientes deste sinal com as seguintes ações:

- Atribuição dos eventuais valores dos parâmetros que deseja passar às portas de prefixo **send_parN_**;
- Inversão do estado lógico da porta de prefixo **send_** correspondente ao sinal que deseja enviar.

Um processo é avisado da ocorrência de um determinado sinal através da presença de um pulso de duração igual a um período do relógio do sistema na porta **p_rec_** correspondente. Os eventuais parâmetros passados estão armazenados nas portas **p_rec_parN_** durante este período.
Tanto o envio quanto a recepção de sinais são sincronizados pela borda de subida do relógio do sistema.

Existe, portanto, uma atraso de um período de relógio entre o envio e a recepção de um sinal.

A simulação do funcionamento do protocolo 3 para o caso do exemplo da Fig. 5.2 é mostrada na Fig. 5.6. Nota-se a requisição simultânea do envio dos três sinais na primeira borda de relógio, representada pelas inversões dos três sinais **send_**, e uma nova requisição de envio de **sig3** na segunda borda do relógio. A saída mostra pulsos correspondentes à presença de **sig1**, **sig2** e duas ocorrências de **sig3**.

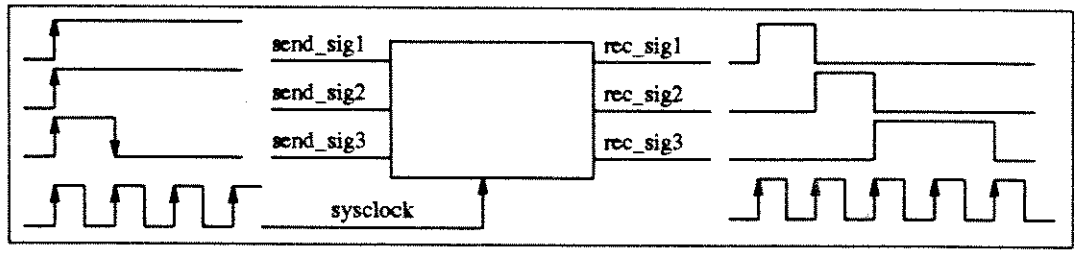


Figura 5.6 Representação do envio de sinais SDL em VHDL.

5.6 Algoritmo de mapeamento em VHDL

O algoritmo de mapeamento analisa os seguintes aspectos de uma descrição em SDL: estrutura, definição de dados, comunicação entre processos, comportamento de processos, definição de variáveis e inicialização do sistema.

- **Estrutura**
  O sistema (**System**) SDL gera uma ENTITY (e correspondente ARCHITECTURE) VHDL cujo nome é dado pelo prefixo **system_** seguido do nome original do sistema. Esta ENTITY contém uma instanciação de componente para cada bloco SDL contido no sistema SDL. As portas de entrada e saída desta ENTITY são obtidas dos canais que ligam o sistema SDL ao ambiente externo.
  A Fig. 5.7 mostra um exemplo de descrição de um sistema SDL. A Fig. 5.8 mostra o correspondente código VHDL para este sistema.

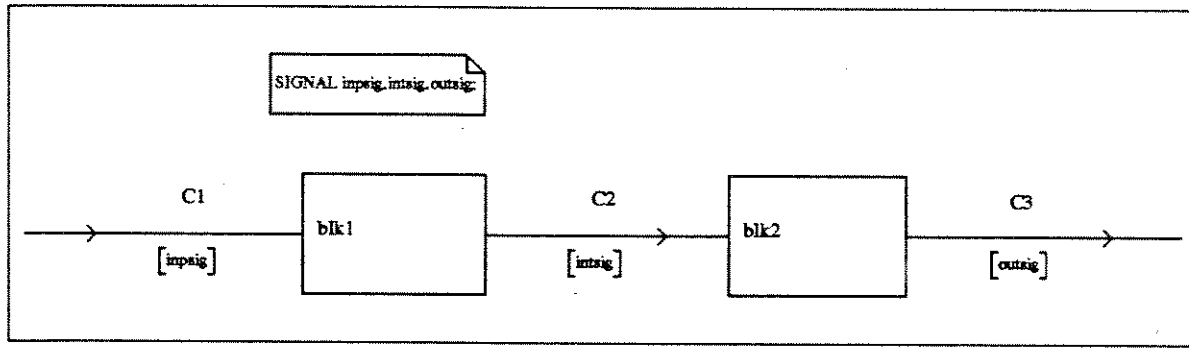


Figura 5.7 Definição de um sistema SDL.

Cada bloco (**Block**) SDL gera uma ENTITY (e correspondente ARCHITECTURE) VHDL cujo nome é dado pelo prefixo **block_** seguido do nome original do bloco. Esta ENTITY contém uma instanciação de componente para cada processo e sub-estrutura contidos no bloco SDL. As portas de entrada e saída desta ENTITY são obtidas dos canais que ligam o bloco SDL ao sistema ou

```
ENTITY system_sys IS PORT(
  clock : IN std_ulogic;
  reset : IN std_ulogic;
  rec_inpsig : IN std_ulogic;
  send_outsig : OUT std_ulogic);
END system_sys;

ARCHITECTURE stoht OF system_sys IS
  COMPONENT block_blk2 PORT(
    clock : IN std_ulogic;
    reset : IN std_ulogic;
    rec_intsig : IN std_ulogic;
    send_outsig : OUT std_ulogic);
  END COMPONENT;
  COMPONENT block_blk1 PORT(
    clock : IN std_ulogic;
    reset : IN std_ulogic;
    rec_inpsig : IN std_ulogic;
    send_intsig : OUT std_ulogic);
  END COMPONENT;
  SIGNAL send_intsig : std_ulogic;
  SIGNAL rec_intsig : std_ulogic;

BEGIN
  blk2 : block_blk2 PORT MAP(
    clock,
    reset,
    rec_intsig,
    send_outsig);
  blk1 : block_blk1 PORT MAP(
    clock,
    reset,
    rec_inpsig,
    send_intsig);

  -- **** Channel connections ****
  rec_intsig <= send_intsig;

END stoht;
```

Figura 5.8 Mapeamento do sistema SDL em primitivas VHDL.

sub-estrutura que o contém.
A Fig. 5.9 mostra um exemplo de descrição de um bloco SDL. Este bloco contém um processo, de nome **p2**. Os números entre parenteses no símbolo deste processo indicam que ele é iniciado com uma instância e continua com uma instância permanentemente. A Fig. 5.10 mostra o correspondente código VHDL para este bloco.

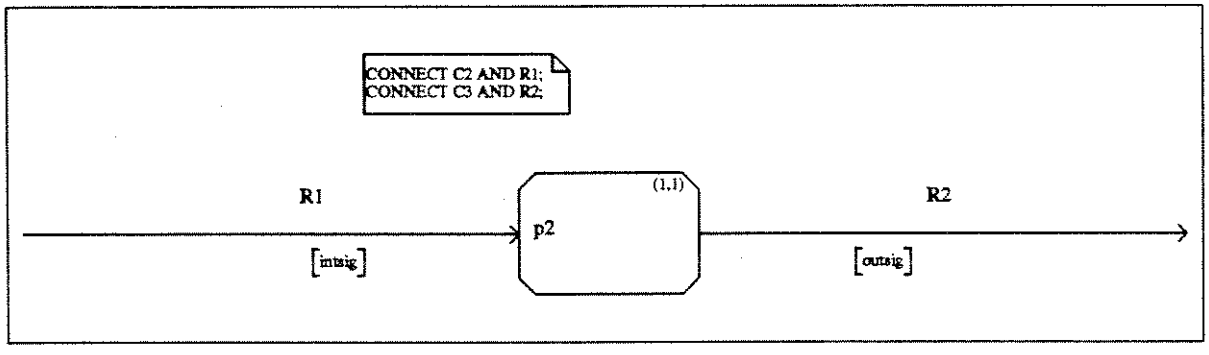


Figura 5.9 Definição de um bloco SDL.

Cada sub-estrutura (**Substructure**) SDL gera uma ENTITY (e correspondente ARCHITECTURE) VHDL cujo nome é dado pelo prefixo **substructure_** seguido do nome original da sub-estrutura. Esta ENTITY contém uma instanciação de componente para cada bloco contido na sub-estrutura SDL. As portas de entrada e saída desta ENTITY são obtidas dos canais que ligam a sub-estructura SDL ao bloco que a contém. O código VHDL gerado para uma declaração de sub-estrutura é similar ao gerado para uma declaração de sistema.

```
ENTITY block_blk2 IS PORT(
  clock : IN std_ulogic;
  reset : IN std_ulogic;
  rec_intsig : IN std_ulogic;
  send_outsig : OUT std_ulogic);
END block_blk2;

ARCHITECTURE stoht OF block_blk2 IS
  COMPONENT prot_p2 PORT(
    clock : IN std_ulogic;
    reset : IN std_ulogic;
    rec_intsig : IN std_ulogic;
    p_send_intsig : OUT std_ulogic);
  END COMPONENT;
  COMPONENT proc_p2 PORT(
    clock : IN std_ulogic;
    reset : IN std_ulogic;
    p_rec_intsig : IN std_ulogic;
    send_outsig : OUT std_ulogic);
  END COMPONENT;

  SIGNAL p_rec_intsig : std_ulogic;
  SIGNAL p_send_intsig : std_ulogic;

BEGIN
  p_p2 : prot_p2 PORT MAP(
    clock,
    reset,
    rec_intsig,
    p_send_intsig);
  p2 : proc_p2 PORT MAP(
    clock,
    reset,
    p_rec_intsig,
    send_outsig);

  p_rec_intsig <= p_send_intsig;

END stoht;
```

Figura 5.10 Mapeamento do bloco SDL em primitivas VHDL.

Cada processo (**Process**) SDL gera dois componentes distintos: uma ENTITY (e correspondente ARCHITECTURE) VHDL, cujo nome é dado pelo prefixo **proc_** seguido do nome original do processo, que modela seu comportamento; e uma ENTITY (e correspondente ARCHITECTURE) VHDL, cujo nome é dado pelo prefixo **prot_** seguido do nome original do processo, que modela o protocolo de comunicação para este processo.
As portas de entrada e saída da ENTITY que implementa o protocolo de comunicação são obtidas dos sinais que chegam ao processo através de rotas, seguindo a nomenclatura convencionada na seção 5.5. As portas de entrada e saída da ENTITY que implementa o comportamento do processo são obtidas dos sinais que chegam e partem do processo através de rotas.
A Fig. 5.11 mostra um exemplo de descrição de um processo SDL. A Fig. 5.12 mostra o correspondente código VHDL para o protocolo de comunicação, e a Fig. 5.13 mostra o código VHDL que representa o comportamento deste processo.

Canais (**Channels**) e rotas (**Signalroutes**) são mapeados em ligações entre os componentes VHDL gerados acima (sistema, blocos, sub-estruturas e processos).

- **Definição de dados**
  Tipos podem ser derivados dos tipos básicos aceitos (**integer, boolean**) com o uso das declarações SDL **Syntype** e **Newtype**. Ambas declarações são mapeadas em VHDL através da declaração de tipo (**TYPE**). Sinônimos (**Synonym**) SDL são mapeados em constantes VHDL.
  Tipos definidos em sistema, bloco ou sub-estrutura SDL são colocados em PACKAGEs VHDL. Para cada escopo é gerado um PACKAGE, lido pelos escopos de hierarquia menor.
  Tipos definidos em processos SDL não geram PACKAGEVHDL, mas são definidos internamente ao componente VHDL gerado para este processo.
  A Fig. 5.14 mostra um exemplo de declaração de tipos em SDL (**Newtype**, **Syntypee Synonym**).
  A Fig. 5.15 mostra o correspondente código VHDL encapsulado em um PACKAGE.

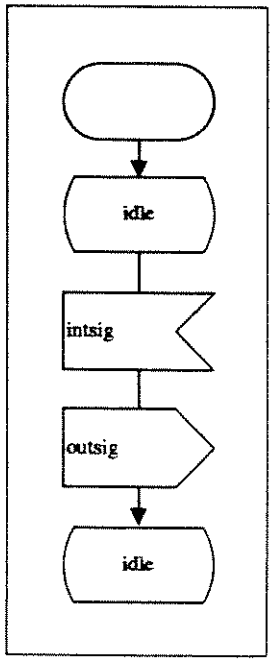


Figura 5.11 Definição de um processo SDL.

```
ENTITY prot_p2 IS PORT(
  clock : IN std_ulogic;
  reset : IN std_ulogic;
  rec_intsig : IN std_ulogic;
  p_send_intsig : OUT std_ulogic);
END prot_p2;

ARCHITECTURE stoht OF prot_p2 IS
  SIGNAL
    int_intsig : std_ulogic;
BEGIN
  latch : PROCESS(clock)
  BEGIN
    IF (clock'EVENT AND clock='1' AND clock'LAST_VALUE='0') THEN
      int_intsig <= rec_intsig;
    END IF;
  END PROCESS latch;

  p_send_intsig <= rec_intsig XOR int_intsig;
END stoht;
```

Figura 5.12 Mapeamento do processo SDL em primitivas VHDL (protocolo)

```
ENTITY proc_p2 IS PORT(
  clock : IN std_ulogic;
  reset : IN std_ulogic;
  p_rec_intsig : IN std_ulogic;
  send_outsig : OUT std_ulogic);
END proc_p2;

ARCHITECTURE stoht OF proc_p2 IS

  SIGNAL
    internal_send_outsig : std_ulogic;
  TYPE states_p2 IS (stoht_start_state,idle);
  SIGNAL cstate_p2 : states_p2;
BEGIN
  send_outsig <= internal_send_outsig;

  p_p2 : PROCESS(clock,reset)
  BEGIN
    IF (reset='1') THEN
      internal_send_outsig <= '0';
      cstate_p2 <= stoht_start_state;
    ELSIF (clock'EVENT AND clock='1' AND clock'LAST_VALUE='0') THEN
      CASE cstate_p2 IS
        WHEN stoht_start_state =>
          cstate_p2 <= idle;
        WHEN idle =>
          IF (p_rec_intsig='1') THEN
            internal_send_outsig <= NOT(internal_send_outsig);
            cstate_p2 <= idle;
          END IF;
      END CASE;
    END IF;
  END PROCESS p_p2;
END stoht;
```

Figura 5.13 Mapeamento do processo SDL em primitivas VHDL (comportamento)

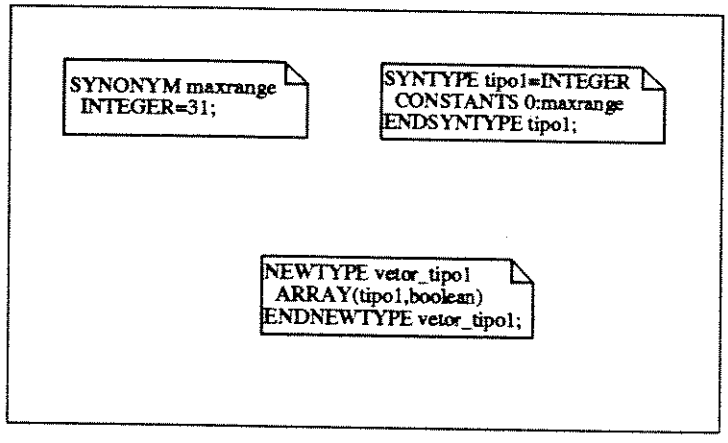


Figura 5.14 Declaração de tipos em SDL.

- **Comunicação entre processos**
  A comunicação via troca de sinais é feita por componentes VHDL que implementam um dos três protocolos descritos na seção 5.4.

- **Comportamento de processos**
  O comportamento de um processo SDL é modelado por um componente VHDL correspondente. Este componente possui um processo VHDL sensível aos sinais de **clock** e **reset** alocados automaticamente pelo algoritmo.
  A Fig. 5.16 mostra uma descrição de um processo SDL que será usada para ilustrar o algoritmo de mapeamento. A Fig. 5.17 mostra o código VHDL que representa o comportamento deste processo.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

-- **** System-level type definitions ****

PACKAGE types_system_my_system IS
  CONSTANT maxrange : integer := 31;
  SUBTYPE tipo1 IS integer RANGE 0 TO maxrange;
  TYPE vetor_tipo1 IS ARRAY(tipo1) OF std_ulogic;
END types_system_my_system;
```

Figura 5.15 PACKAGE contendo declarações de tipo.

O estado do processo é armazenado, em VHDL, em um sinal local a este processo, cujo nome é dado pelo prefixo **cstate_** seguido do nome do processo. Este sinal VHDL é do tipo enumerado, contendo os mesmos nomes dos estados originais SDL acrescido de um estado inicial utilizado na inicialização do processo. Um exemplo de declaração de estados é mostrado na Fig. 5.17, linhas 14-15.
Transições de estado são modeladas pela estrutura CASE-WHEN do VHDL. Cada estado possui uma entrada correspondente em uma cláusula WHEN (Fig. 5.17, linhas 25, 26, 29 e 42).
A presença de sinal, que habilita a transição de estados, é testada com cláusulas IF-ELSIF-THEN. A ordem na qual os sinais são testados é dada pela prioridade fornecida pelo projetista para cada sinal (Fig. 5.17, linhas 30, 32 e 43). O corpo destes testes contém as ações (envio de sinais, atribuições, operações lógico-aritméticas) associadas à esta transição de estados.
Decisões (**Decision**) são mapeadas em cláusulas IF-ELSE-THEN (Fig. 5.17, linhas 33 e 36).
Atribuições em tarefas (**Task**) são mapeadas em atribuições sequenciais em VHDL (Fig. 5.17, linhas 34, 37 e 44).
Envio de sinais (**Output**) é mapeado na requisição de envio ao protocolo do processo receptor: atribuição dos parâmetros a serem passados aos sinais de prefixo **send_parN_** e inversão do sinal de prefixo **send_**

- **Definição de variáveis**
  Variáveis declaradas em um processo SDL são mapeadas em variáveis VHDL.
  Variáveis SDL reveladas (primitiva SDL **Revealed**) são tornadas visíveis a todos os outros processos do mesmo bloco, com a criação de uma porta VHDL de saída do processo VHDL. Processos que lêem uma variável revelada (primitiva SDL **View**) possuem uma porta VHDL de entrada conectada com a porta de saída do processo que a revela.
  A Fig. 5.17 mostra uma variável, de nome **counter**, declarada **Revealed**. A linha 19 do código VHDL mostra sua declaração; uma porta de saída da ENTITY VHDL correspondente ao processo SDL (**revealed_counter**) recebe uma cópia desta variável sempre que seu valor é alterado (Fig 5.17. linhas 28, 35, 38 e 45).
  Como o algoritmo não prevê a criação dinâmica de objetos, o tipo SDL **PId** não é implementado. Portanto, expressões **View** são resolvidas procurando, em processos contidos no mesmo bloco, por uma variável homônima declarada **Revealed**. Esta procura por variáveis reveladas em outros processos do mesmo bloco é uma característica da revisão de 1992 da linguagem SDL [8].

- **Inicialização do sistema**
  O sinal de **reset** alocado pelo algoritmo de mapeamento realiza as seguintes ações no circuito gerado:

  - Atribuição dos valores iniciais às variáves de cada processo (Fig. 5.17, linha 22)

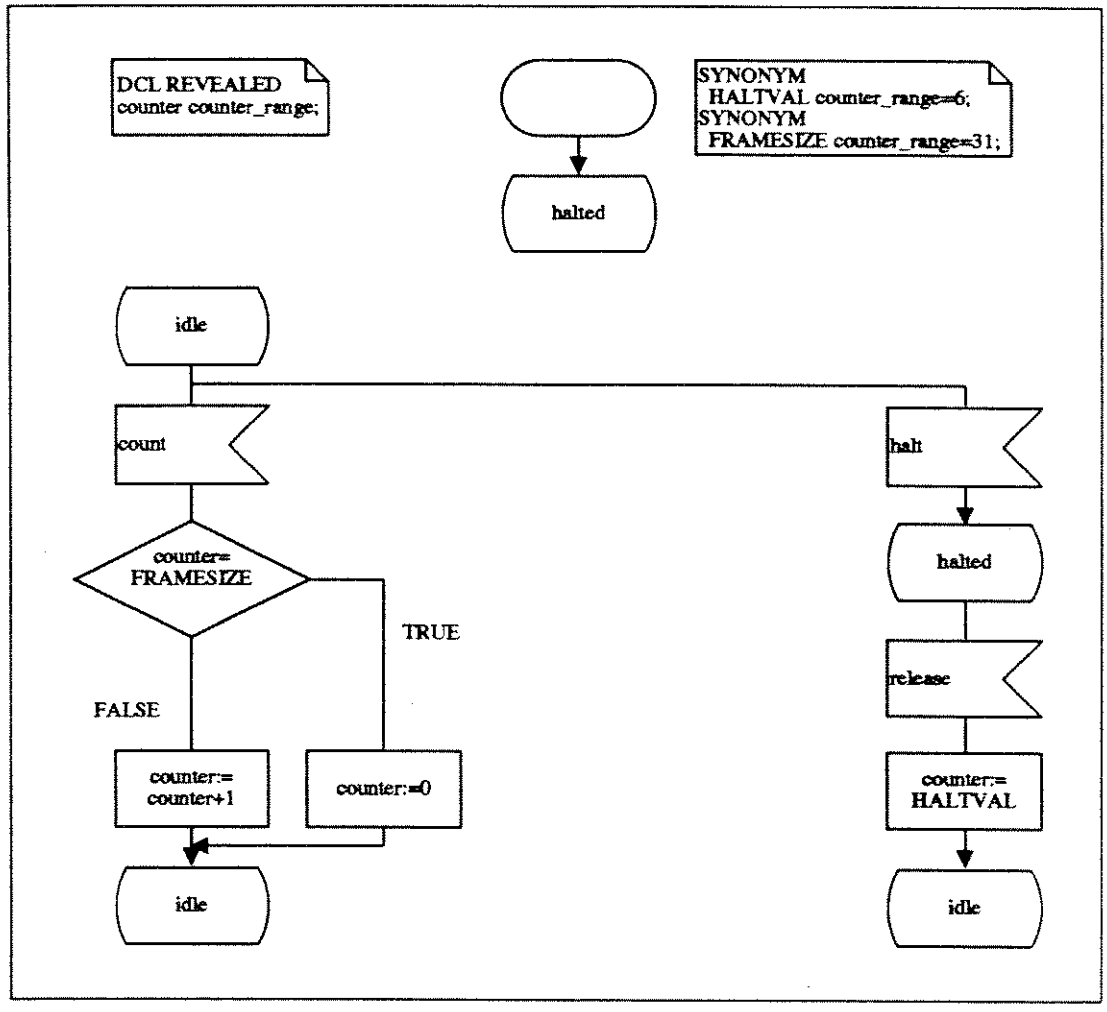


Figura 5.16 Definição de um processo SDL.

```
1 - ENTITY proc_counter IS PORT(
2 -   clock : IN std_ulogic;
3 -   reset : IN std_ulogic;
4 -   p_rec_halt : IN std_ulogic;
5 -   p_rec_release : IN std_ulogic;
6 -   p_rec_count : IN std_ulogic;
7 -   revealed_counter : OUT counter_range);
8 - END proc_counter;
9 -
10- ARCHITECTURE stoht OF proc_counter IS
11-
12-   CONSTANT haltval : counter_range := 6;
13-   CONSTANT framesize : counter_range := 31;
14-   TYPE states_counter IS (stoht_start_state,idle,halted);
15-   SIGNAL cstate_counter : states_counter;
16- BEGIN
17-
18-   p_counter : PROCESS(clock,reset)
19-     VARIABLE counter : counter_range;
20-   BEGIN
21-     IF (reset='1') THEN
22-       counter := 0;
23-       cstate_counter <= stoht_start_state;
24-     ELSIF (clock'EVENT AND clock='1' AND clock'LAST_VALUE='0') THEN
25-       CASE cstate_counter IS
26-         WHEN stoht_start_state =>
27-           cstate_counter <= halted;
28-           revealed_counter <= counter;
29-         WHEN idle =>
30-           IF (p_rec_halt='1') THEN
31-             cstate_counter <= halted;
32-           ELSIF (p_rec_count='1') THEN
33-             IF NOT(counter=framesize) THEN
34-               counter := counter+1;
35-               revealed_counter <= counter;
36-             ELSIF (counter=framesize) THEN
37-               counter := 0;
38-               revealed_counter <= counter;
39-             END IF;
40-             cstate_counter <= idle;
41-           END IF;
42-         WHEN halted =>
43-           IF (p_rec_release='1') THEN
44-             counter := haltval;
45-             revealed_counter <= counter;
46-             cstate_counter <= idle;
47-           END IF;
48-       END CASE;
49-     END IF;
50-   END PROCESS p_counter;
51- END stoht;
```

Figura 5.17 Mapeamento do processo SDL em primitivas VHDL (comportamento)

- Atribuição do estado inicial, de nome **stoht_start_state**, a cada processo (Fig. 5.17, linha 23). Este estado representa o símbolo **Start** do SDL; no primeiro ciclo de relógio o processo realiza as transições alocadas para este símbolo.

As regras de mapeamento VHDL-SDL estão sumarizadas na Tabela 5.1, incluindo a relação entre os nomes da descrição em VHDL e a descrição SDL original. Os prefixos determinados nesta tabela são completados com o nome original declarado em SDL.

5.7 Metodologia de projeto

A metodologia de projeto de circuitos digitais (inicialmente apresentada na Fig. 1.2, capítulo 1) desenvolvida durante o trabalho de tese está apresentada na Fig. 5.18.

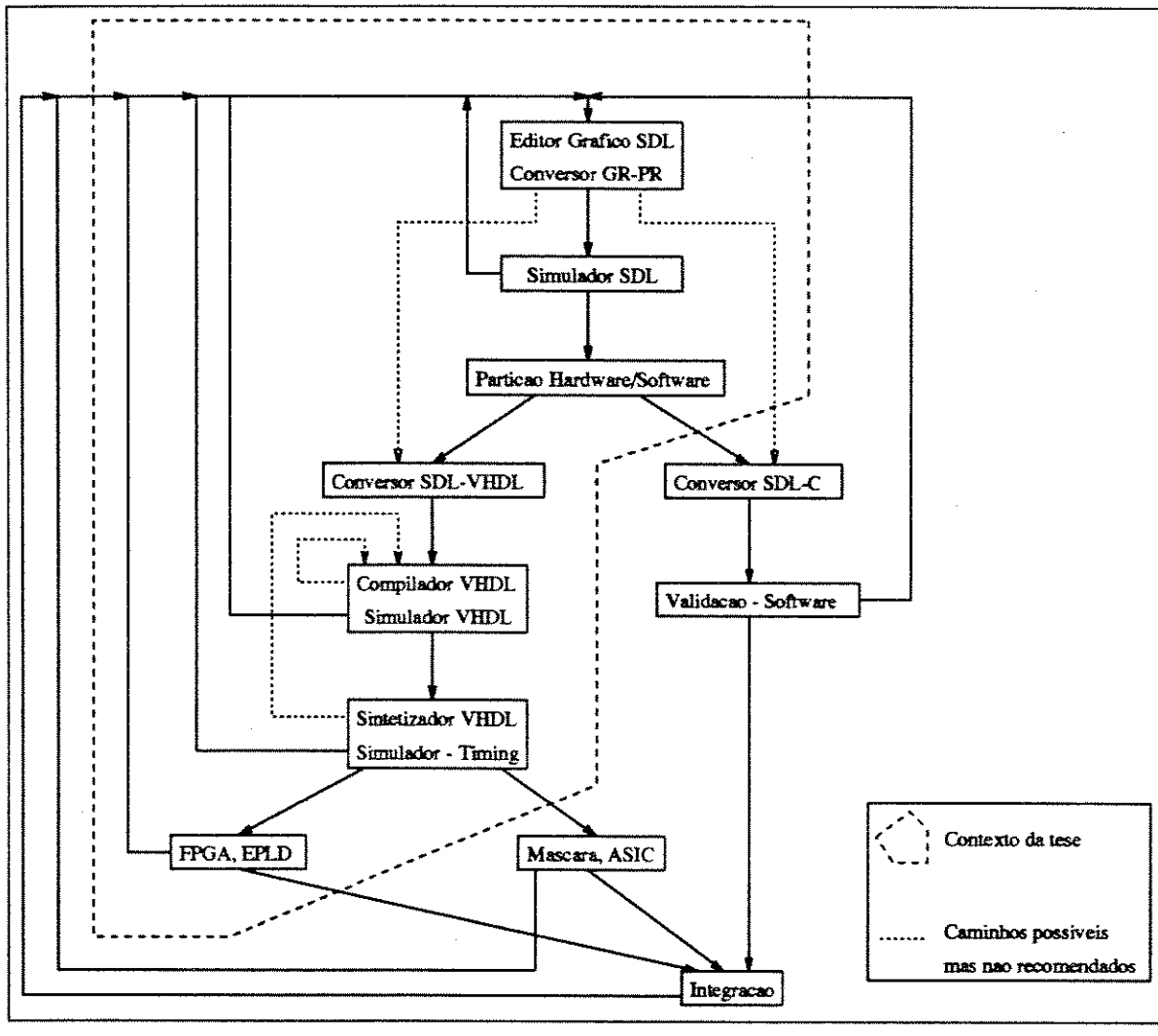


Figura 5.18 Diagrama de blocos da metodologia de projeto proposta.

Sintaxe SDL	*Sintaxe VHDL*
System	-**ENTITY, prefixo system_ + ARCHITECTURE** contendo instanciações de componentes -**PACKAGE**, prefixo **types_system_**, contendo declarações de tipo
Block	-**ENTITY, prefixo block_ + ARCHITECTURE** contendo instanciações de componentes -**PACKAGE**, prefixo **types_block_**, contendo declarações de tipo
Substructure	-**ENTITY, prefixo substructure_ + ARCHITECTURE** contendo instanciações de componentes -**PACKAGE**, prefixo **types_substructure_**, contendo declarações de tipo
Process	**ENTITY** de prefixo **proc_** + ARCHITECTURE contendo um **PROCESS** VHDL correspondente ao comportamento do processo; **ENTITY** de prefixo **prot_** + ARCHITECTURE contendo um **PROCESS** VHDL correspondente ao protocolo determinado para este processo
Newtype	**TYPE**, declarado no **PACKAGE** correspondente a seu escopo
Syntype	**SUBTYPE**, declarado no **PACKAGE** correspondente a seu escopo
Synonym	**CONSTANT**, declarado no **PACKAGE** correspondente ao seu escopo
Output	Atribuição dos parâmetros de prefixo **send_parN_** Inversão do PORT de prefixo **send_**
Input	Teste do valor do PORT de prefixo **rec_** na estrutura IF-ELSIF-THEN de recepção de sinais Atribuição dos parâmetros recebidos nos sinais **rec_parN_**
State	Tipo enumerado de prefixo **states_** contendo todos os estados do processo mais o estado inicial **stoht_start_state**
Dcl	Declaração de **VARIABLE** homônimo no escopo VHDL correspondente
Revealed	Porta de saída de prefixo **revealed_** no componente VHDL correspondente
Viewed	Porta de entrada de prefixo **viewed_** no componente VHDL correspondente
Decision	Estrutura IF-ELSE-THEN
Task	Atribuição sequencial em VHDL

Tabela 5.1 Correspondência entre as sintaxes SDL e VHDL

O projeto inicia-se com a entrada, via editor gráfico, da descrição do sistema em SDL. O sistema deve ser descrito da forma mais abrangente possível nesta etapa. Esta descrição gráfica é convertida em descrição textual (SDL-PR) e simulada ainda no domínio do SDL.

Tendo alcançado um resultado comportamental satisfatório nesta simulação, passa-se para a etapa de partição. Partes do sistema são alocadas para a implementação em hardware enquanto outras partes são implementadas em software.

Os módulos alocados para a implementação em hardware são mapeados em VHDL automaticamente, respeitando as restrições de síntese mencionadas neste capítulo, e posteriormente compilados e simulados.

Tendo a simulação em VHDL comportamental sido validada, esta descrição passa pela etapa de síntese e mapeamento na tecnologia-destino. O circuito é novamente simulado, já contendo informações de temporização da tecnologia escolhida.

Após a validação desta simulação são gerados protótipos dos componentes, que podem então ser integrados com os módulos direcionados para a implementação em software.

Problemas encontrados durante estas etapas devem ser corrigidos no topo da hierarquia do projeto, ou seja, na descrição gráfica em SDL. A metodologia suporta modificações a nível de VHDL comportamental, já que as interfaces propostas pelo algoritmo de mapeamento são padronizadas. Entretanto, estas modificações causam dificuldades na documentação e devem ser evitadas.

O trabalho de tese englobou o contexto de projeto de hardware (região delimitada pelas linhas tracejadas na Fig. 5.18). A integração hardware-software é um tópico importante que será abordado em trabalhos futuros.

Foram utilizadas as seguintes ferramentas para a aplicação desta metodologia:

- **Editor SDL, conversor GR-PR, simulador SDL:** SDT, Telelogic [3]
- **Conversor SDL-VHDL:** STOHT, desenvolvido durante a tese [13]
- **Compilador VHDL:** System1076, Mentor Graphics Corp. [11]
- **Simulador VHDL/Timing:** Quicksim, Mentor Graphics Corp. [10]
- **Sintetizador VHDL:** Autologic, Mentor Graphics Corp. [9]
- **Implementação em lógica programável:** MaxPlusII, Altera Inc.

5.8 Conclusões

A metodologia de projeto de circuitos digitais proposta neste trabalho baseia-se no uso da linguagem de especificação de sistemas **SDL** e na geração automática de código **VHDL** sintetizável para a implementação física. Um sub-conjunto da linguagem **SDL** foi isolado para o projeto de circuitos; neste capítulo os elementos deste sub-conjunto são identificados, e uma proposta de mapeamento para a linguagem **VHDL** foi apresentada, levando-se em conta aspectos de estruturação, comunicação e comportamento. A metodologia prevê a integração do projeto de hardware/software através da geração de código a partir da linguagem **SDL**.

Capítulo 6

Stoht - Mapeador SDL-VHDL

6.1 Introdução

Para validar o algoritmo proposto neste trabalho foi desenvolvido um programa de conversão SDL-VHDL que o implementa, denominado **Stoht** (**S**DL-**to**-**H**ardware **T**ranslator). A estrutura e o funcionamento deste programa são abordados neste capítulo.

O objetivo de tornar o programa compatível com várias plataformas que suportam ambientes **EDA** levou à escolha do sistema operacional **UNIX**. A linguagem utilizada foi **C++**, padrão **ANSI**, compilado com o pacote de domínio público **GNU-C** da Free Software Foundation, Inc.

Procurou-se com estas escolhas tornar o programa disponível em uma vasta gama de plataformas, como **SunOS** e **Solaris** para arquitetura Sun, **Linux** e **NetBSD** para arquiteturas Intel-386/486/Pentium. **AIX** para arquitetura IBM/RISC, entre outros. A maioria dos centros de pesquisa e ensino possuem sistemas que são capazes de suportar a execução do programa **Stoht**.

O programa trabalha com arquivos de entrada do tipo texto contendo descrições SDL-PR, e gera arquivos texto contendo o código VHDL obtido após o mapeamento.

O programa **Stoht** e sua documentação estão disponíveis na rede Internet através do servidor *World Wide Web* do Departamento de Telemática da Faculdade de Engenharia Elétrica da Unicamp, no endereço: **http://dt.fee.unicamp.br**.

6.2 Estrutura

O programa **Stoht** é dividido da seguinte maneira:

- Analisador léxico
  Lê o arquivo de entrada, separa *tokens* e palavras reservadas da linguagem SDL, fornecendo-os para o analisador sintático.
- Analisador sintático
  Monta uma estrutura de dados (*parse tree*) a partir dos tokens fornecidos pelo analisador léxico e das regras de sintaxe do SDL.
- Analisador semântico
  Valida a semântica da descrição, percorrendo a árvore montada pelo analisador sintático, já levando em conta restrições de síntese que o algoritmo introduz. Produz uma base de dados formada por uma tabela de símbolos e por objetos que representam os elementos da linguagem SDL.

- Gerador de código
  Percorre a base de dados gerada pelo analisador semântico, gerando código VHDL de acordo com o algoritmo de mapeamento.

As análises léxica e sintática são realizadas com o auxílio das ferramentas **flex** e **bison**, também distribuídas pelo projeto GNU da Free Software Foundation. Estes programas são variantes dos programas **lex** e **yacc** comumente encontrados em vários sistemas UNIX.

As especificações léxicas e sintáticas utilizadas são as divulgadas em 1992 pelo CCITT [8], traduzidas para uma sintaxe próxima à reconhecida pelo programas **flex** e **bison** por B. Noerbaek [30].

A vantagem de se utilizar um analisador sintático que interprete a gramática completa da linguagem ao invés de um que interprete apenas o sub-conjunto determinado para a síntese de circuitos é a possibilidade de identificar e apontar com maior clareza para o projetista eventuais elementos não suportados pelo algoritmo.

A Fig. 6.1 mostra a descrição léxica do SDL, no formato aceito pelo programa **flex**; a Fig. 6.2 mostra um fragmento da descrição da gramática do SDL, no formato aceito pelo programa **bison**.

As descrições das Figs. 6.1 e 6.2 são processadas pelos programas **flex** e **yacc**, que geram código **C** ANSI. O código **C** é ligado ao restante do programa descrito em **C++**.

O analisador sintático gerado pela descrição da Fig. 6.2 realiza operações de deslocamento (**shift**) e redução (**reduce**) nos tokens fornecidos pelo analisador léxico, gerando nós da árvore de "parse" a cada redução. Os algoritmos de análise léxica e sintática são descritos detalhadamente em [2].

6.3 Estruturas de dados

A primeira estrutura de dados manipulada pelo programa **Stoht** é a árvore de "parse" gerada durante a análise sintática. Esta árvore é formada por estruturas do tipo **NODE**. Cada nó do tipo **NODE** contém dados, como a linha do arquivo correspondente a este nó e a cadeia de caracteres correspondente a um identificador lido, e apontadores para eventuais nós filhos. A Fig. 6.3 ilustra a árvore de "parse" gerada pelo programa.

A etapa de análise semântica gera outras duas estruturas de dados:

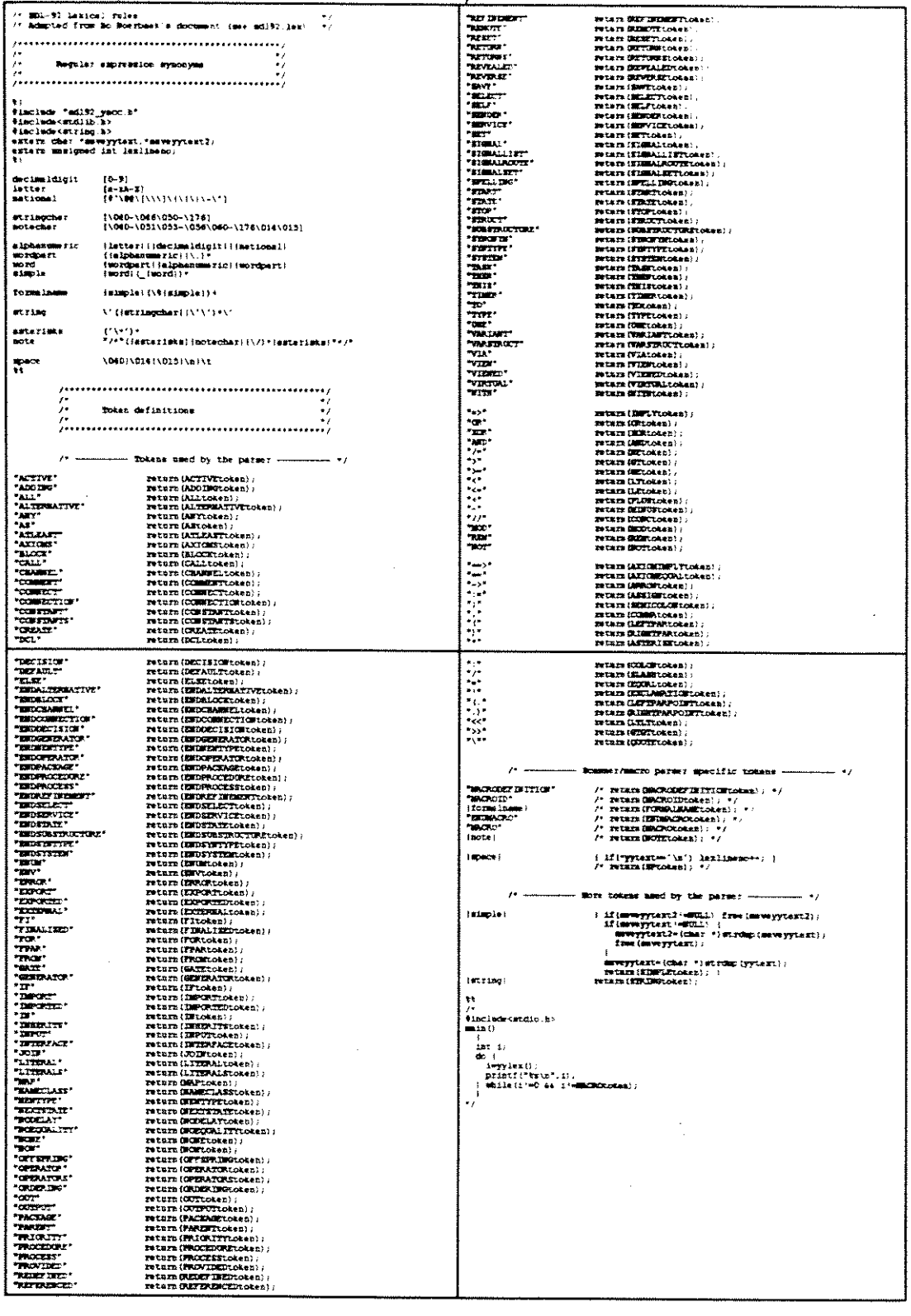
- **Tabela de símbolos:** contém os nomes dos diversos símbolos obtidos na análise léxica, como variáveis ou sinais. Cada símbolo possui uma entrada nesta tabela, e é acessado por um identificador representado por um número inteiro. Cada símbolo possui também seu escopo armazenado nesta tabela.

- **Base de dados SDL:** formada por objetos que representam os elementos da linguagem SDL utilizados pelo algoritmo, como **System**, **Block**, etc. Cada objeto possui dados que o caracterizam, como o identificador correspondente na tabela de símbolos, e possui também referências a objetos que ele contém. Um objeto **sdl_system** contém, por exemplo, o identificador do nome deste sistema e listas de blocos, canais e declarações de dados. Todas as referências entre os objetos desta base de dados são feitas via identificadores inteiros armazenados na tabela de símbolos.

A Fig. 6.4 ilustra a estrutura de dados gerada durante a análise semântica.

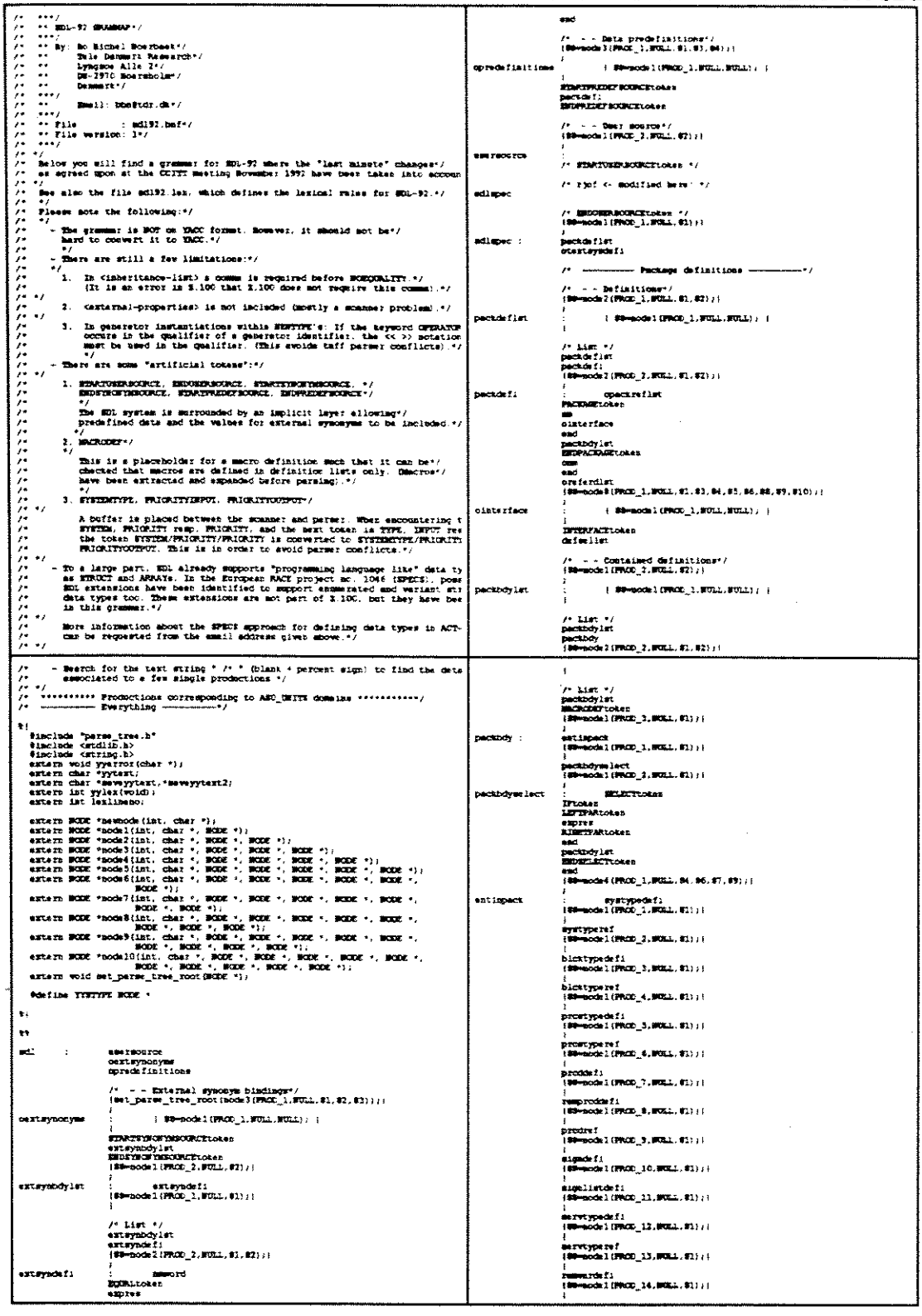
6.4 Integração com ferramentas EDA/CASE

O programa **Stoht** está atualmente integrado com a ferramenta **SDT** para a entrada de dados e com o compilador **System1076** para a saída de dados.



```
/* SDL-92 Lexical Rules                                    */
/* Adapted from Bo Boerbaek's document (sw: sdl92.lex)     */

/*****************************************/
/*                                       */
/*      Regular expression synonyms      */
/*                                       */
/*****************************************/

%{
#include "sdl92_yacc.h"
#include<stdlib.h>
#include<string.h>
extern char *saveyytext, *saveyytext2;
extern unsigned int lexlineno;
%}

decimaldigit    [0-9]
letter          [a-zA-Z]
national        [@'\?@\[\\\]\^\{\|\}\~\"]

stringchar      [\040-\046\050-\176]
notechar        [\040-\051\053-\056\060-\176\014\013]

alphanumeric    {letter}|{decimaldigit}|{national}
wordpart        {alphanumeric}|\_}*
word            {wordpart}|{alphanumeric}{wordpart}
simple          {word}|_{word}}*

formalname      {simple}{\%{simple}}*

string          \'{stringchar}|{\'\'}*\'

asterisks       {\*}*
note            "/*"({asterisks}|{notechar}|{\/}*|{asterisks}"*/"

space           \040|\014|\013|\n|\t
%%

        /*****************************************/
        /*                                       */
        /*      Token definitions                */
        /*                                       */
        /*****************************************/


        /* ---------- Tokens used by the parser ---------- */

"ACTIVE"                return(ACTIVEtoken);
"ADDING"                return(ADDINGtoken);
"ALL"                   return(ALLtoken);
"ALTERNATIVE"           return(ALTERNATIVEtoken);
"AMV"                   return(AMVtoken);
"AS"                    return(AStoken);
"ATLEAST"               return(ATLEASTtoken);
"AXIOMS"                return(AXIOMStoken);
"BLOCK"                 return(BLOCKtoken);
"CALL"                  return(CALLtoken);
"CHANNEL"               return(CHANNELtoken);
"COMMENT"               return(COMMENTtoken);
"CONNECT"               return(CONNECTtoken);
"CONNECTION"            return(CONNECTIONtoken);
"CONSTANT"              return(CONSTANTtoken);
"CONSTANTS"             return(CONSTANTStoken);
"CREATE"                return(CREATEtoken);
"DCL"                   return(DCLtoken);

"DECISION"              return(DECISIONtoken);
"DEFAULT"               return(DEFAULTtoken);
"ELSE"                  return(ELSEtoken);
"ENDALTERNATIVE"        return(ENDALTERNATIVEtoken);
"ENDBLOCK"              return(ENDBLOCKtoken);
"ENDCHANNEL"            return(ENDCHANNELtoken);
"ENDCONNECTION"         return(ENDCONNECTIONtoken);
"ENDDECISION"           return(ENDDECISIONtoken);
"ENDGENERATOR"          return(ENDGENERATORtoken);
"ENDNEWTYPE"            return(ENDNEWTYPEtoken);
"ENDOPERATOR"           return(ENDOPERATORtoken);
"ENDPACKAGE"            return(ENDPACKAGEtoken);
"ENDPROCEDURE"          return(ENDPROCEDUREtoken);
"ENDPROCESS"            return(ENDPROCESStoken);
"ENDREFINEMENT"         return(ENDREFINEMENTtoken);
"ENDSELECT"             return(ENDSELECTtoken);
"ENDSERVICE"            return(ENDSERVICEtoken);
"ENDSTATE"              return(ENDSTATEtoken);
"ENDSUBSTRUCTURE"       return(ENDSUBSTRUCTUREtoken);
"ENDSYNTYPE"            return(ENDSYNTYPEtoken);
"ENDSYSTEM"             return(ENDSYSTEMtoken);
"ENDUNION"              return(ENDUNIONtoken);
"ENV"                   return(ENVtoken);
"ERROR"                 return(ERRORtoken);
"EXPORT"                return(EXPORTtoken);
"EXPORTED"              return(EXPORTEDtoken);
"EXTERNAL"              return(EXTERNALtoken);
"FI"                    return(FItoken);
"FINALIZED"             return(FINALIZEDtoken);
"FOR"                   return(FORtoken);
"FPAR"                  return(FPARtoken);
"FROM"                  return(FROMtoken);
"GATE"                  return(GATEtoken);
"GENERATOR"             return(GENERATORtoken);
"IF"                    return(IFtoken);
"IMPORT"                return(IMPORTtoken);
"IMPORTED"              return(IMPORTEDtoken);
"IN"                    return(INtoken);
"INHERITS"              return(INHERITStoken);
"INPUT"                 return(INPUTtoken);
"INTERFACE"             return(INTERFACEtoken);
"JOIN"                  return(JOINtoken);
"LITERAL"               return(LITERALtoken);
"LITERALS"              return(LITERALStoken);
"MAP"                   return(MAPtoken);
"NAMECLASS"             return(NAMECLASStoken);
"NEWTYPE"               return(NEWTYPEtoken);
"OBJECTGATE"            return(OBJECTGATEtoken);
"MODELAY"               return(MODELAYtoken);
"MODQUALITY"            return(MODQUALITYtoken);
"NOW"                   return(NOWtoken);
"NOT"                   return(NOTtoken);
"OFFSPRING"             return(OFFSPRINGtoken);
"OPERATOR"              return(OPERATORtoken);
"OPERATORS"             return(OPERATORStoken);
"ORDERING"              return(ORDERINGtoken);
"OUT"                   return(OUTtoken);
"OUTPUT"                return(OUTPUTtoken);
"PACKAGE"               return(PACKAGEtoken);
"PARENT"                return(PARENTtoken);
"PRIORITY"              return(PRIORITYtoken);
"PROCEDURE"             return(PROCEDUREtoken);
"PROCESS"               return(PROCESStoken);
"PROVIDED"              return(PROVIDEDtoken);
"REDEFINED"             return(REDEFINEDtoken);
"REFERENCED"            return(REFERENCEDtoken);

"REFINEMENT"            return(REFINEMENTtoken);
"REMOTE"                return(REMOTEtoken);
"RESET"                 return(RESETtoken);
"RETURN"                return(RETURNtoken);
"REVEALED"              return(REVEALEDtoken);
"REVERSE"               return(REVERSEtoken);
"SAVE"                  return(SAVEtoken);
"SELECT"                return(SELECTtoken);
"SELF"                  return(SELFtoken);
"SERVER"                return(SERVERtoken);
"SERVICE"               return(SERVICEtoken);
"SET"                   return(SETtoken);
"SIGNAL"                return(SIGNALtoken);
"SIGNALLIST"            return(SIGNALLISTtoken);
"SIGNALROUTE"           return(SIGNALROUTEtoken);
"SIGNALSET"             return(SIGNALSETtoken);
"SPELLING"              return(SPELLINGtoken);
"START"                 return(STARTtoken);
"STATE"                 return(STATEtoken);
"STOP"                  return(STOPtoken);
"STRUCT"                return(STRUCTtoken);
"SUBSTRUCTURE"          return(SUBSTRUCTUREtoken);
"SYNONYM"               return(SYNONYMtoken);
"SYNTYPE"               return(SYNTYPEtoken);
"SYSTEM"                return(SYSTEMtoken);
"TASK"                  return(TASKtoken);
"THEN"                  return(THENtoken);
"THIS"                  return(THIStoken);
"TIMER"                 return(TIMERtoken);
"TO"                    return(TOtoken);
"TYPE"                  return(TYPEtoken);
"USE"                   return(USEtoken);
"VARIANT"               return(VARIANTtoken);
"VARSTRUCT"             return(VARSTRUCTtoken);
"VIA"                   return(VIAtoken);
"VIEW"                  return(VIEWtoken);
"VIEWED"                return(VIEWEDtoken);
"VIRTUAL"               return(VIRTUALtoken);
"WITH"                  return(WITHtoken);

"=>"                    return(INFLYtoken);
"OR"                    return(ORtoken);
"XOR"                   return(XORtoken);
"AND"                   return(ANDtoken);
"/="                    return(NEtoken);
">"                     return(GTtoken);
">="                    return(GEtoken);
"<"                     return(LTtoken);
"<="                    return(LEtoken);
"+"                     return(PLUStoken);
"-"                     return(MINUStoken);
"//"                    return(CONCtoken);
"MOD"                   return(MODtoken);
"REM"                   return(REMtoken);
"NOT"                   return(NOTtoken);

"=>>"                   return(AXIOMINFLYtoken);
"=>>*"                  return(AXIOMEQUALtoken);
"*=>"                   return(ARROWtoken);
"*:="                   return(ASSIGNtoken);
":"                     return(COLONtoken);
"*"                     return(STARtoken);
"*(*"                   return(LEFTPARtoken);
"*)*"                   return(RIGHTPARtoken);
"*\""                   return(QUOTEtoken);

"*;*"                   return(SEMICOLONtoken);
"*/*"                   return(SLASHtoken);
"*=*"                   return(EQUALtoken);
"*!*"                   return(EXCLAMATIONtoken);
"*(.*"                  return(LEFTPARQUOTEtoken);
"*.)*"                  return(RIGHTPARQUOTEtoken);
"*<<"                   return(LITLTtoken);
"*>>*"                  return(GTGTtoken);
*\""                    return(QUOTEtoken);


        /* ---------- Some tokens used by the parser: ---------- */

"MACRODEFINITION"       /* return(DMACRODEFINITIONtoken); */
"MACROID"               /* return(DMACROIDtoken); */
|formalname|            /* return(FORMALNAMEtoken); */
"ENDMACRO"              /* return(ENDMACROtoken); */
"MACRO"                 /* return(DMACROtoken); */
|note|                  /* return(NOTEtoken); */

|space|                 { if(*yytext=='\n') lexlineno++; }
                        /* return(SPtoken); */


        /* ---------- More tokens used by the parser: ---------- */

|simple|                { if(saveyytext2!=NULL) free(saveyytext2);
                          if(saveyytext!=NULL) {
                            saveyytext2=(char *)strdup(saveyytext);
                            free(saveyytext);
                          }
                          saveyytext=(char *)strdup(yytext);
                          return(SIMPLEtoken); }
|string|                return(STRINGtoken);

%%
/*
#include<stdio.h>
main()
  {
    int i;
    do {
      i=yylex();
      printf("%s\n",i);
    } while(i!=0 && i!=MACROtoken);
  }
*/
```

Figura 6.1 Descrição léxica SDL utilizada pelo **Stoht**.



```
/*   ***/
/*   ** SDL-92 GRAMMAR*/
/*   ***/
/*   ** By: Bo Michel Beerbaek*/
/*   **     Tele Danmark Research*/
/*   **     Lyngsoe Alle 7*/
/*   **     DK-2970 Hoersholm*/
/*   **     Denmark*/
/*   ***/
/*   **     Email: bbm@tdr.dk*/
/*   ***/
/*   ** File        : sdl92.bnf*/
/*   ** File version: 1*/
/*   ***/
/* */
/*   Below you will find a grammar for SDL-92 where the "last minute" changes*/
/*   as agreed upon at the CCITT meeting November 1992 have been taken into account*/
/* */
/*   See also the file sdl92.lex, which defines the lexical rules for SDL-92.*/
/* */
/*   Please note the following:*/
/* */
/*     - The grammar is NOT in YACC format. However, it should not be*/
/*       hard to convert it to YACC.*/
/*       */
/*     - There are still a few limitations:*/
/*       */
/*       1.  In <inheritance-list> a comma is required before NONEQUALITY.*/
/*           (It is an error in X.100 that X.100 does not require this comma).*/
/* */
/*       2.  <external-properties> is not included (mostly a scanner problem).*/
/* */
/*       3.  In generator instantiations within NEWTYPE's: If the keyword OPERATORS*/
/*           occurs in the qualifier of a generator identifier, the << >> notation*/
/*           must be used in the qualifier. (This avoids tuff parser conflicts).*/
/*           */
/*     - There are some "artificial tokens":*/
/* */
/*       1.  STARTUSERSOURCE, ENDUSERSOURCE, STARTSYNTYPESOURCE, */
/*           ENDSYNTYPESOURCE, STARTPREDEFSOURCE, ENDPREDEFSOURCE*/
/*           */
/*           The SDL system is surrounded by an implicit layer allowing*/
/*           predefined data and the values for external synonyms to be included.*/
/*           */
/*       2.  MACRODEF*/
/*           */
/*           This is a placeholder for a macro definition such that it can be*/
/*           checked that macros are defined in definition lists only. (Macros*/
/*           have been extracted and expanded before parsing).*/
/*           */
/*       3.  SYSTEMTYPE, PRIORITYINPUT, PRIORITYOUTPUT*/
/* */
/*           A buffer is placed between the scanner and parser. When encountering t*/
/*           SYSTEM, PRIORITY resp. PRIORITY, and the next token is TYPE, INPUT res*/
/*           the token SYSTEM/PRIORITY/PRIORITY is converted to SYSTEMTYPE/PRIORITY*/
/*           PRIORITYOUTPUT. This is in order to avoid parser conflicts.*/
/* */
/*     - To a large part, SDL already supports "programming language like" data ty*/
/*       as STRUCT and ARRAYs. In the European RACE project ac. 1046 (SPECS), poss*/
/*       SDL extensions have been identified to support enumerated and variant str*/
/*       data types too. These extensions are not part of X.100, but they have bee*/
/*       in this grammar.*/
/* */
/*       More information about the SPECS approach for defining data types in ACT-*/
/*       can be requested from the email address given above.*/
/* */

/*     - Search for the text string " /* " (blank + percent sign) to find the data*/
/*       associated to a few single productions */
/* */
/*  ********** Productions corresponding to ASC_UNITS domains **********/
/*  ---------- Everything ----------*/

%{
  #include "parse_tree.h"
  #include <stdlib.h>
  #include <string.h>
  extern void yyerror(char *);
  extern char *yytext;
  extern char *mewyytext,*mewyytext2;
  extern int yylex(void);
  extern int lexlineno;

  extern NODE *newnode(int, char *);
  extern NODE *node1(int, char *, NODE *);
  extern NODE *node2(int, char *, NODE *, NODE *);
  extern NODE *node3(int, char *, NODE *, NODE *, NODE *);
  extern NODE *node4(int, char *, NODE *, NODE *, NODE *, NODE *);
  extern NODE *node5(int, char *, NODE *, NODE *, NODE *, NODE *, NODE *);
  extern NODE *node6(int, char *, NODE *, NODE *, NODE *, NODE *, NODE *,
                     NODE *);
  extern NODE *node7(int, char *, NODE *, NODE *, NODE *, NODE *, NODE *,
                     NODE *, NODE *);
  extern NODE *node8(int, char *, NODE *, NODE *, NODE *, NODE *, NODE *,
                     NODE *, NODE *, NODE *);
  extern NODE *node9(int, char *, NODE *, NODE *, NODE *, NODE *, NODE *,
                     NODE *, NODE *, NODE *, NODE *);
  extern NODE *node10(int, char *, NODE *, NODE *, NODE *, NODE *, NODE *,
                     NODE *, NODE *, NODE *, NODE *, NODE *);
  extern void set_parse_tree_root(NODE *);

  #define YYSTYPE NODE *

%}

%%

sdl     :       sdlresource
                sdlsynonyms
                sdlpredefinitions

                /* - - External synonym bindings*/
                {set_parse_tree_root(node3(PROD_1,NULL,$1,$2,$3));}
                ;
sdlsynonyms     :       | $$=node1(PROD_1,NULL,NULL); |
                ;
                STARTSYNONYMSOURCEtoken
                sdlsynbodylist
                ENDSYNONYMSOURCEtoken
                {$$=node1(PROD_2,NULL,$2);}
                ;
sdlsynbodylist  :       sdlsyndefi
                {$$=node1(PROD_1,NULL,$1);}
                |

                /* List */
                sdlsynbodylist
                sdlsyndefi
                {$$=node2(PROD_2,NULL,$1,$2);}
                ;
sdlsyndefi      :       newword
                EQUALtoken
                sdlpred
                ;

                end

                /* - - Data predefinitions*/
                {$$=node3(PROD_1,NULL,$1,$3,$4);}
                ;
sdlpredefinitions       :       | $$=node1(PROD_1,NULL,NULL); |
                ;
                STARTPREDEFSOURCEtoken
                packdefi
                ENDPREDEFSOURCEtoken

                /* - - User source*/
                {$$=node1(PROD_2,NULL,$2);}
                ;
sdlresource     :
                /* STARTUSERSOURCEtoken */

                /* typf <- modified here */
sdlspec

                /* ENDUSERSOURCEtoken */
                {$$=node1(PROD_1,NULL,$1);}
                ;
sdlspec :       packdeflist
                otextsyndefi

                /* ---------- Package definitions ----------*/

                /* - - Definitions*/
                {$$=node2(PROD_1,NULL,$1,$2);}
                ;
packdeflist     :       | $$=node1(PROD_1,NULL,NULL); |
                ;

                /* List */
                packdeflist
                packdefi
                {$$=node2(PROD_2,NULL,$1,$2);}
                ;
packdefi        :       opackrefist
                PACKAGEtoken
                ##
                ointerface
                end
                packbdylist
                ENDPACKAGEtoken
                ###
                end
                orefердlist
                {$$=node8(PROD_1,NULL,$1,$3,$4,$5,$6,$8,$9,$10);}
                ;
ointerface      :       | $$=node1(PROD_1,NULL,NULL); |
                ;
                INTERFACEtoken
                definelist

                /* - - Contained definitions*/
                {$$=node1(PROD_2,NULL,$2);}
                ;
packbdylist     :       | $$=node1(PROD_1,NULL,NULL); |
                ;

                /* List */
                packbdylist
                packbdy
                {$$=node2(PROD_2,NULL,$1,$2);}
                ;

                /* List */
                packbdylist
                ###OKORDEFtoken
                {$$=node1(PROD_3,NULL,$1);}
                ;
packbdy :       entispack
                {$$=node1(PROD_1,NULL,$1);}
                ;
                packbdyselect
                {$$=node1(PROD_2,NULL,$1);}
                ;
packbdyselect   :       SELECTtoken
                IFtoken
                LEFTPARtoken
                expres
                RIGHTPARtoken
                end
                packbdylist
                ENDSELECTtoken
                end
                {$$=node4(PROD_1,NULL,$4,$6,$7,$9);}
                ;
entispack       :       sysstypedefi
                {$$=node1(PROD_1,NULL,$1);}
                |
                sysstyperef
                {$$=node1(PROD_2,NULL,$1);}
                |
                blkstypedefi
                {$$=node1(PROD_3,NULL,$1);}
                |
                blkstyperef
                {$$=node1(PROD_4,NULL,$1);}
                |
                prcstypedefi
                {$$=node1(PROD_5,NULL,$1);}
                |
                prcstyperef
                {$$=node1(PROD_6,NULL,$1);}
                |
                prcdefi
                {$$=node1(PROD_7,NULL,$1);}
                |
                remprcdefi
                {$$=node1(PROD_8,NULL,$1);}
                |
                prcdref
                {$$=node1(PROD_9,NULL,$1);}
                |
                sigdefi
                {$$=node1(PROD_10,NULL,$1);}
                |
                sigdeflistdefi
                {$$=node1(PROD_11,NULL,$1);}
                |
                servtypedefi
                {$$=node1(PROD_12,NULL,$1);}
                |
                servtyperef
                {$$=node1(PROD_13,NULL,$1);}
                |
                remsrvdefi
                {$$=node1(PROD_14,NULL,$1);}
                ;
```

Figura 6.2 Descrição da gramática SDL utilizada pelo **Stoht**.

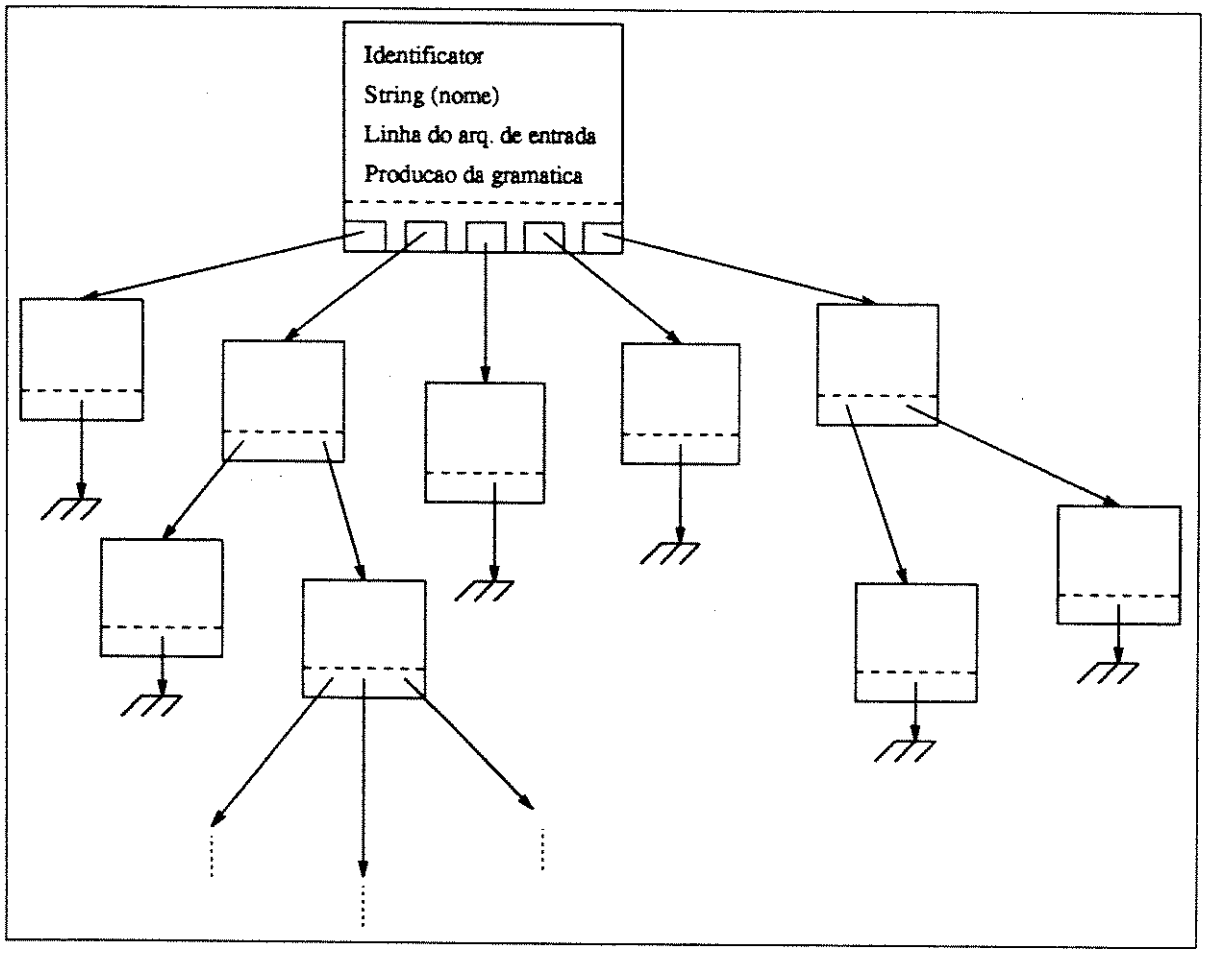


Figura 6.3 Ilustração da árvore de "parse" gerada pelo **Stoht**.

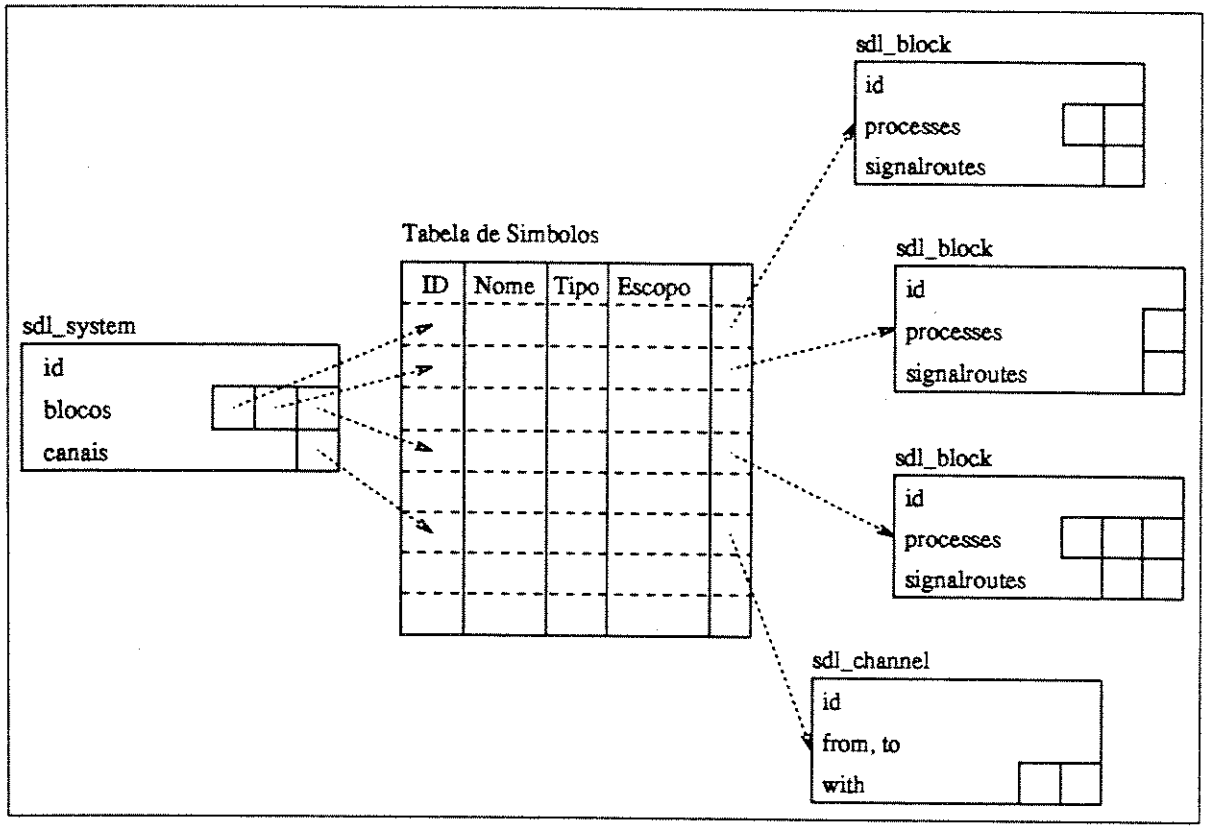


Figura 6.4 Ilustração da base de dados SDL e da tabela de símbolos.

A saída de dados inclui arquivos VHDL, arquivos de mapeamento de biblioteca e um *script* UNIX que realiza a compilação completa do projeto através de chamadas ao compilador **System1076**.

Os arquivos VHDL são gerados em sub-diretórios que mantém a estruturação original da descrição SDL. A partir do diretório de chamada do programa é criado um sub-diretório que armazena o sistema SDL. Dentro deste sub-diretório são criados outros sub-diretórios para blocos, e assim sucessivamente para os demais blocos, sub-estruturas e processos que compõem o sistema.

No caso específico do ambiente **Mentor Graphics**, os arquivos contendo descrições VHDL possuem extensão **.hdl** e são acompanhados de arquivos de mosmo nome e extensão **.lmf**, que fazem o mapeamento das diversas bibliotecas criadas pelo conversor **Stoht**.

Deste modo, a compilação completa do sistema pode ser realizada apenas com a chamada do *script* também gerado pelo conversor.

A documentação e listagem dos módulos que compõem o mapeador **Stoht** estão disponíveis em um relatório interno do Departamento de Telemática da Faculdade de engenharia Elétrica.

6.5 Conclusões

O mapeador **SDL-VHDL** desenvolvido durante a tese integra as ferramentas disponíveis na Faculdade de Engenharia Elétrica para a descrição de sistemas em SDL (**SDT**) e para a automação de projeto de circuitos (**Mentor Graphics**). A sua utilização em projetos-exemplo desenvolvidos durante a tese serviu para auxiliar o desenvolvimento do algoritmo de mapeamento e para a sua validação. A codificação do programa utilizou recursos de geração de código para a definição do compilador **SDL**, que reconhece a última recomendação **CCITT** para esta linguagem, divulgada em 1992.

Capítulo 7

Exemplos de aplicação da metodologia

7.1 Detector de Sincronismo

7.1.1 Introdução

Para exemplificar o método de descrição de circuitos apresentado, foi escolhido o projeto de um sistema simples de detecção e recuperação de sincronismo, frequentemente utilizado na transmissão digital em sistemas de telecomunicações.

A partir da especificação do comportamento do sistema, foi obtida uma descrição em SDL e posteriormente foi realizado o mapeamento em VHDL sintetizável, que permitiu a realização física do circuito de forma automática.

7.1.2 Conceitos - Transmissão Digital

A transmissão digital consiste na codificação binária da informação que se deseja transmitir, de tal forma que a informação codificada seja utilizada na comunicação.

Esta forma de comunicação permite um ganho de qualidade na transmissão quando comparada à analógica, principalmente quando as distâncias envolvidas são grandes, ao mesmo tempo em que aumenta a flexibilidade do sistema de comunicação, fazendo uso dos recursos crescentes de hardware e software disponíveis.

Uma técnica largamente utilizada em telefonia para otimização do uso dos meios digitais de comunicação é a Multiplexação por Divisão Temporal (TDM [1]). Esta consiste no compartilhamento do acesso a um único meio de alta capacidade de transmissão (medida em bits/s) por vários canais, independentes, de taxas de transmissão menores.

Os canais que acessam este meio são ordenados e agrupados em uma estrutura lógica denominada **quadro**, como esquematizado na Fig. 7.1.

Neste esquema tem-se um quadro formado por C canais independentes, sendo que cada canal comporta B bits. O número de canais por quadro, o número de bits por canal e a posição do canal em relação ao início do quadro são fixos. A transmissão é sincronizada por uma base de tempo, denominada relógio a nível de bit.

A divisão do meio de transmissão em quadros e canais é apenas lógica; não existem marcações explícitas que delimitam estas estruturas, existe apenas uma sequência de bits. Torna-se necessária portanto, na

[1] Time Division Multiplex

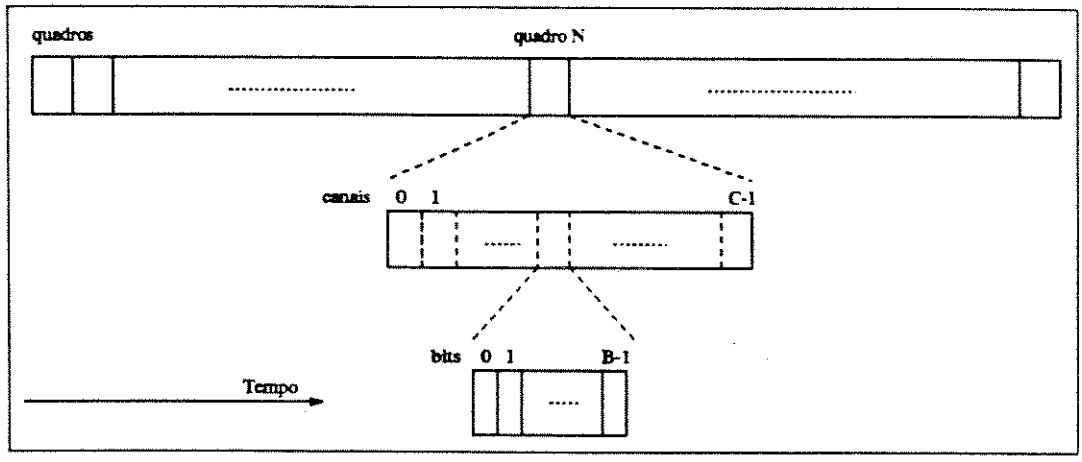


Figura 7.1 Representação da estrutura lógica de quadro e de canal em TDM (Time Division Multiplex)

recepção dos dados, a recuperação da divisão lógica produzida na transmissão.

Sendo o número de canais C e de bits B os mesmos na transmissão e recepção, basta o conhecimento do instante de início de quadro para se recuperar a estrutura lógica na recepção.

Isto é conseguido com a alocação de um dos canais para a transmissão de informação de sincronismo. Este canal (normalmente o canal zero) não transmite dados, mas apenas um código de sincronismo.

A função do Detector de Sincronismo é identificar este canal, assegurando a recuperação correta das informações enviadas nos demais canais. Para tal, deve constantemente comparar o conteúdo do canal de sincronismo com o padrão esperado e decidir se o sistema mantém-se sincronizado; caso decida que o sincronismo foi perdido, deve ser capaz de recuperá-lo no menor número de quadros possível.

7.1.3 Especificação

O sistema de sincronismo desenvolvido segue o algoritmo de transição abrupta (Fig. 7.2), considerando um quadro formado por **C** canais de **B** bits cada, transmitidos com um relógio **CLK**; a palavra de sincronismo é denominada **W**.

Este algoritmo tem quatro estados no detector:

1. Sincronismo: a recepção está sincronizada com a estrutura lógica de quadro da transmissão.
2. Pré-alarme: a recepção estava em sincronismo mas, por erro na transmissão ou "escorregamento" do quadro, perdeu-a.
3. Fora de sincronismo: a recepção não consegue identificar a estrutura de quadro enviada pelo transmissor e passa a buscar a palavra de sincronismo bit a bit.
4. Pré-sincronismo: a recepção procura, quadro a quadro, a confirmação da palavra de sincronismo.

Os estados **2** e **3** podem ser divididos em vários níveis de pré-alarme (**N** >= 1) e pré-sincronismo (**M** >= 1).

O comportamento do sistema é determinado de acordo com seu estado atual e de suas entradas, da seguinte forma:

- Quando em sincronismo:
  O canal de sincronismo é verificado a cada quadro. Se contiver a palavra **W**, o sistema continua sincronizado; caso contrário, entra em pré-alarme, nível 0.

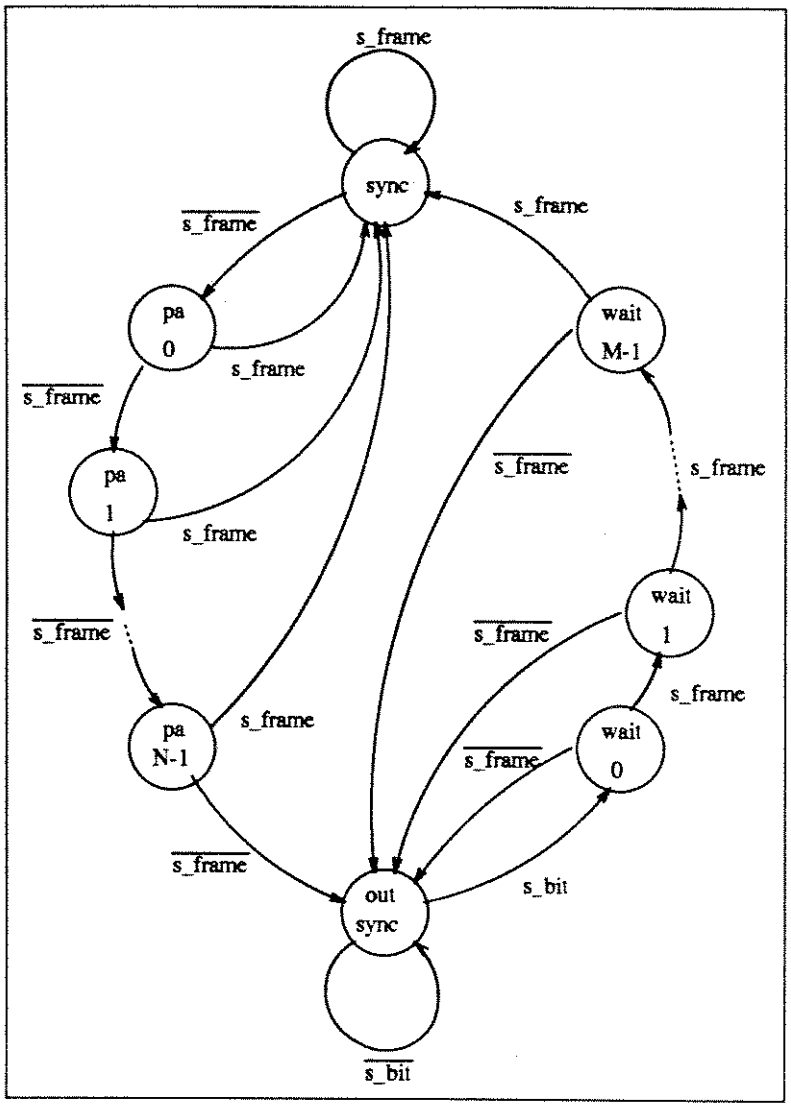


Figura 7.2 Máquina de Estados do Detector de Sincronismo : N pré-alarmes, M pré-sincronismos.

- Quando em pré-alarme, nível **n**:
  O canal de sincronismo é verificado a cada quadro. Se contiver a palavra **W**, o sistema volta a situação de sincronismo; caso contrário, avança um nível (**n**+1) de pré-alarme. Caso este nível alcance o limiar **N**, o sistema passa a estar fora de sincronismo.

- Quando fora de sincronismo:
  O detector testa, bit a bit, a presença da palavra **W** nos últimos **B** bits lidos. Ao encontrá-la, assume que este é o canal de sincronismo e entra em pré-sincronismo, nível 0.

- Quando em pré-sincronismo, nível **m**:
  O canal de sincronismo é verificado a cada quadro. Se contiver a palavra **W**, o sistema avança um nível (**m**+1) de pré-sincronismo; caso este nível alcance o limiar **M**, o sistema passa a operar em sincronismo. Caso a palavra lida no canal de sincronismo não seja **W**, o sistema volta a estar fora de sincronismo.

7.1.4 Descrição Comportamental em SDL

O detector de sincronismo foi realizado com 1 estado de pré-alarme e 1 estado de pré-sincronismo (N=M=1). A máquina de estados resultante é mostrada na Fig. 7.3.

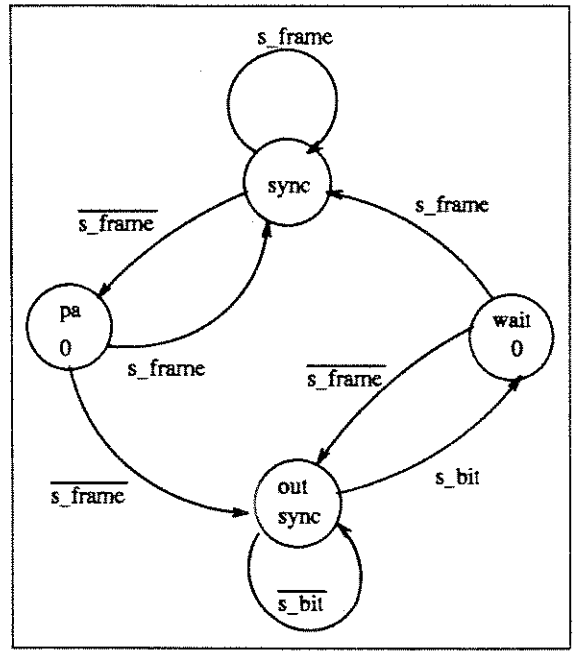


Figura 7.3 Máquina de Estados do Detector de Sincronismo para N=M=1.

A Fig. 7.4 mostra o nível mais alto de abstração do sistema (**System** SDL), contendo um bloco (**mainblk**) que interage com o ambiente externo através de um sinal de entrada (**serial**) e um de saída (**syncstat**). O sinal **serial** representa a cadeia de bits de entrada do detector; a chegada de um bit é representada por um evento neste sinal, e o valor deste bit é representado pelo parâmetro que este sinal carrega. O sinal de saída informa se o sistema está ou não em sincronismo.

A Fig. 7.5 mostra o bloco **mainblk** decomposto em: processos (**counter**, **shifter** e **state_machine**), que descrevem o comportamento do detector de sincronismo; rotas (**R1**, **R2**, etc), que permitem comunicação entre processos via troca de sinais; e em declarações de sinais (**Signal**) e de tipos (**Newtype**, **Syntype**) utilizadas dentro dos processos.

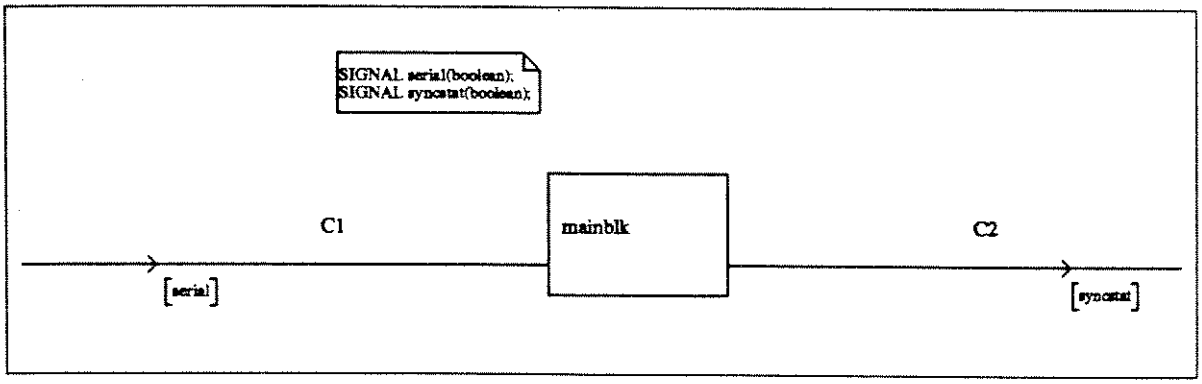


Figura 7.4 Circuito detector de sincronismo descrito em seu nível SDL mais alto.

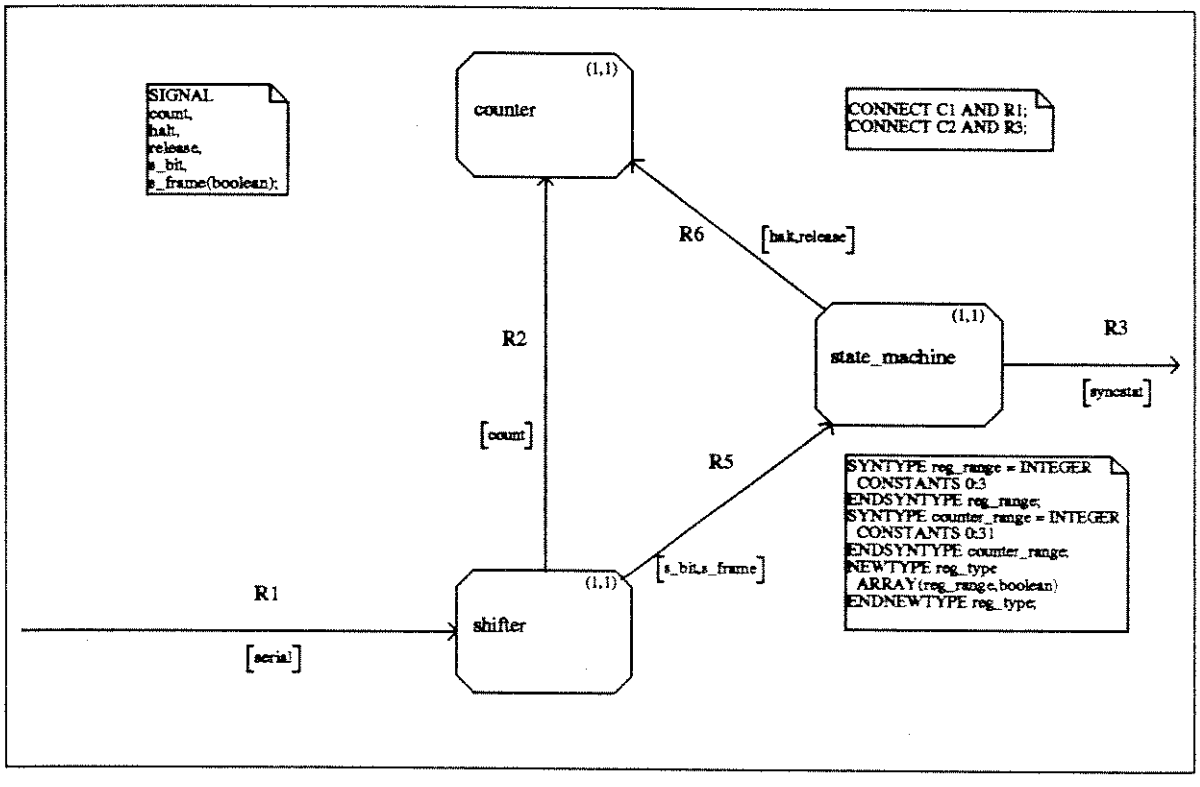


Figura 7.5 **mainblk**: Único bloco do sistema **sqta** com seus processos componentes.

A Fig. 7.6 mostra a máquina de estados relativa ao processo **state_machine**. A máquina possui quatro estados (**outsync**, **sync**, **pa_0** e **wait_0**), sendo que o estado inicial é fora de sincronismo (**outsync**). A máquina realiza transições de estado através de eventos nos sinais **s_bit** (sincronismo de bit) e **s_frame** (sincronismo de quadro). Nota-se que apenas o estado **outsync** é sensível ao sincronismo de bit, enquanto os demais estados somente realizam transições de quadro em quadro.

O processo **state_machine**, através dos sinais **halt** e **release**, controla o processo **counter** (Fig. 7.7), que armazena a informação da posição de cada bit recebido em relação à estrutura lógica do quadro.

A Fig. 7.7 mostra o processo **counter**, contador que armazena a posição do bit de chegada na estrutura de quadro. Este processo possui dois estados, **halted** e **idle**. O primeiro é alcançado quando o detector está fora de sincronismo; neste caso, a informação do quadro lógico é perdida e a posição relativa do bit de chegada é irrelevante. O estado **idle** e a transição iniciada pelo sinal **count** representam a operação usual do contador. Note que, durante esta transição, o contador eventualmente envia um sinal **frame** para o processo **shifter** (Fig. 7.8), se os últimos bits lidos estiverem na posição do canal de sincronismo (isto é, se counter=CSYNC).

A Fig. 7.8 mostra o processo **shifter**, registrador de deslocamento que armazena os últimos **B** bits lidos da entrada serial. Este processo tem apenas um estado (**idle**) e duas transições, iniciadas pelos sinais **serial** e **frame**. A transição iniciada por **serial** leva ao armazenamento do último bit recebido (parâmetro **val**) e ao teste da presença da palavra de sincronismo no canal lido; em caso afirmativo, o sinal **s_bit** é enviado ao processo **state_machine** (Fig. 7.6). Note que a recepção do sinal **s_bit** pelo processo **state_machine** só é relevante quando este processo encontra-se fora de sincronismo (estado **outsync**); nos demais estados, este sinal é descartado.

A transição iniciada pelo sinal **frame** (proveniente do processo **counter**, Fig. 7.7) resulta na avaliação do conteúdo do canal lido e no envio do sinal **s_frame** para a máquina de estados do processo **state_machine** (Fig. 7.6). Note que o sinal **s_frame** contém um parâmetro que indica a presença ou não da palavra de sincronismo no canal de sincronismo; este parâmetro é avaliado no processo **state_machine** durante a transição de estados.

7.1.5 Conversão e síntese

A aplicação das regras de conversão SDL-VHDL sintetizável descritas na seção 5.6 nos diagramas SDL do detector de sincronismo produziu o código VHDL mostrado nas Figs. 7.9 a 7.13. O mapeamento foi automático, realizado pelo programa **Stoht** desenvolvido durante o trabalho de tese.

A Fig. 7.9 mostra a ENTITY VHDL correspondente ao **System** SDL, contendo declaração e instanciação do componente relativos ao bloco que o sistema contém (**mainblk**).

A descrição VHDL correspondente ao bloco SDL **mainblk** contém declarações e instanciações de componentes relativos aos processos SDL declarados neste bloco, e seus respectivos protocolos de comunicação. Estes componentes, também descritos em VHDL, são mostrados nas Figs. 7.11 (processo **counter**), 7.12 (processo **shifter**) e 7.13 (processo **state_machine**).

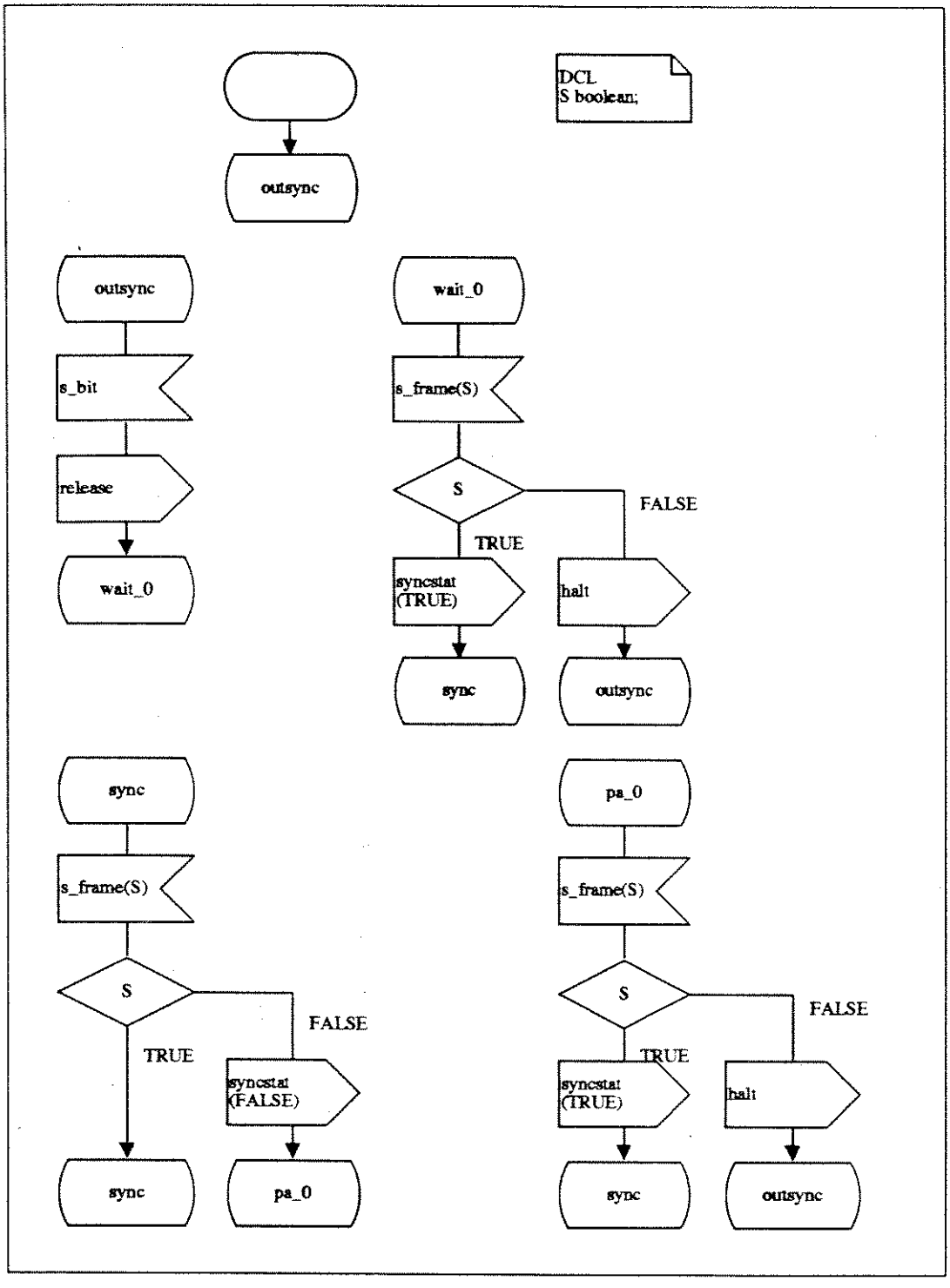


Figura 7.6 **state_machine**: Processo Máquina de Estado do detector de sincronismo.

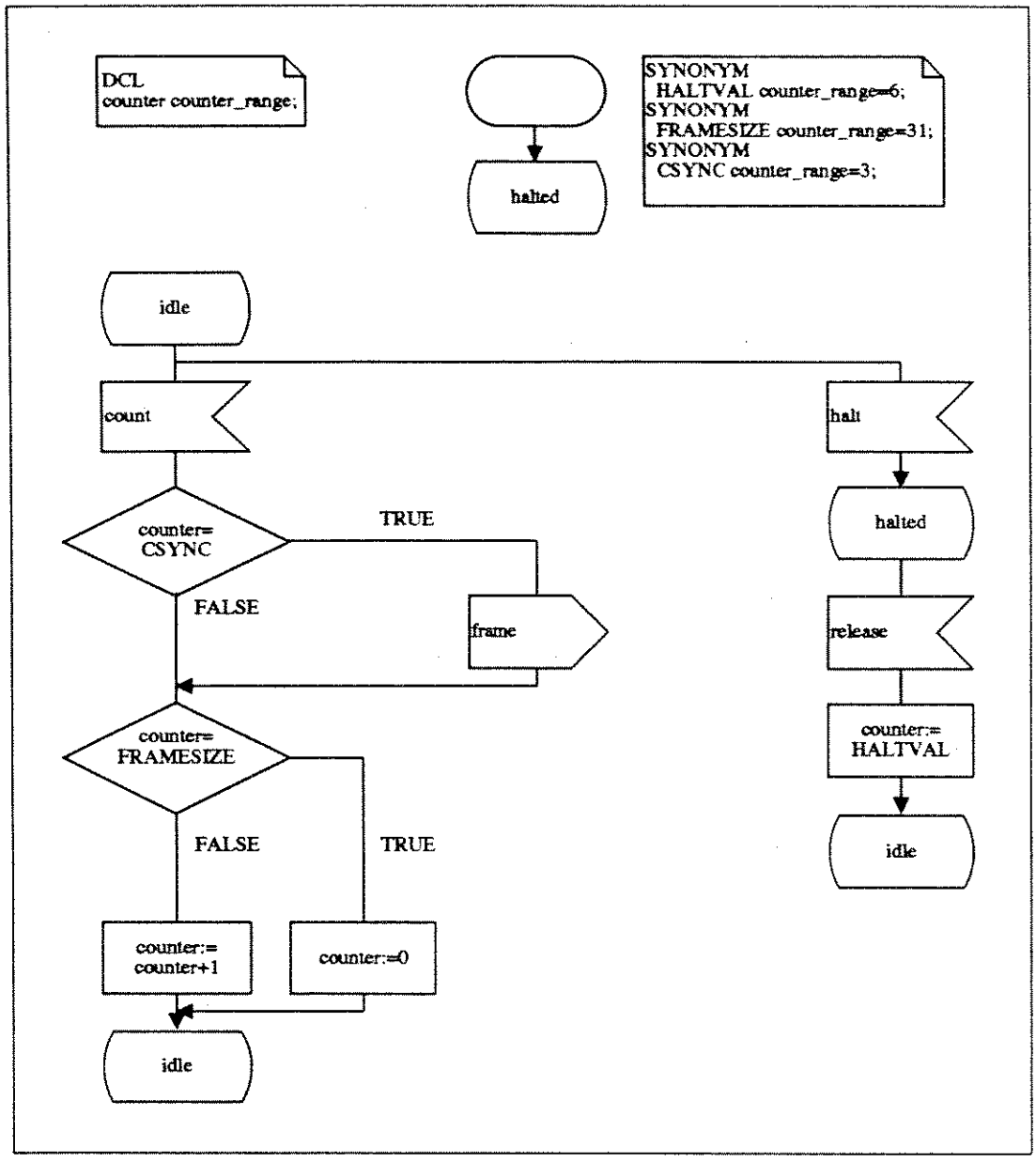


Figura 7.7 **counter**: Processo contador.

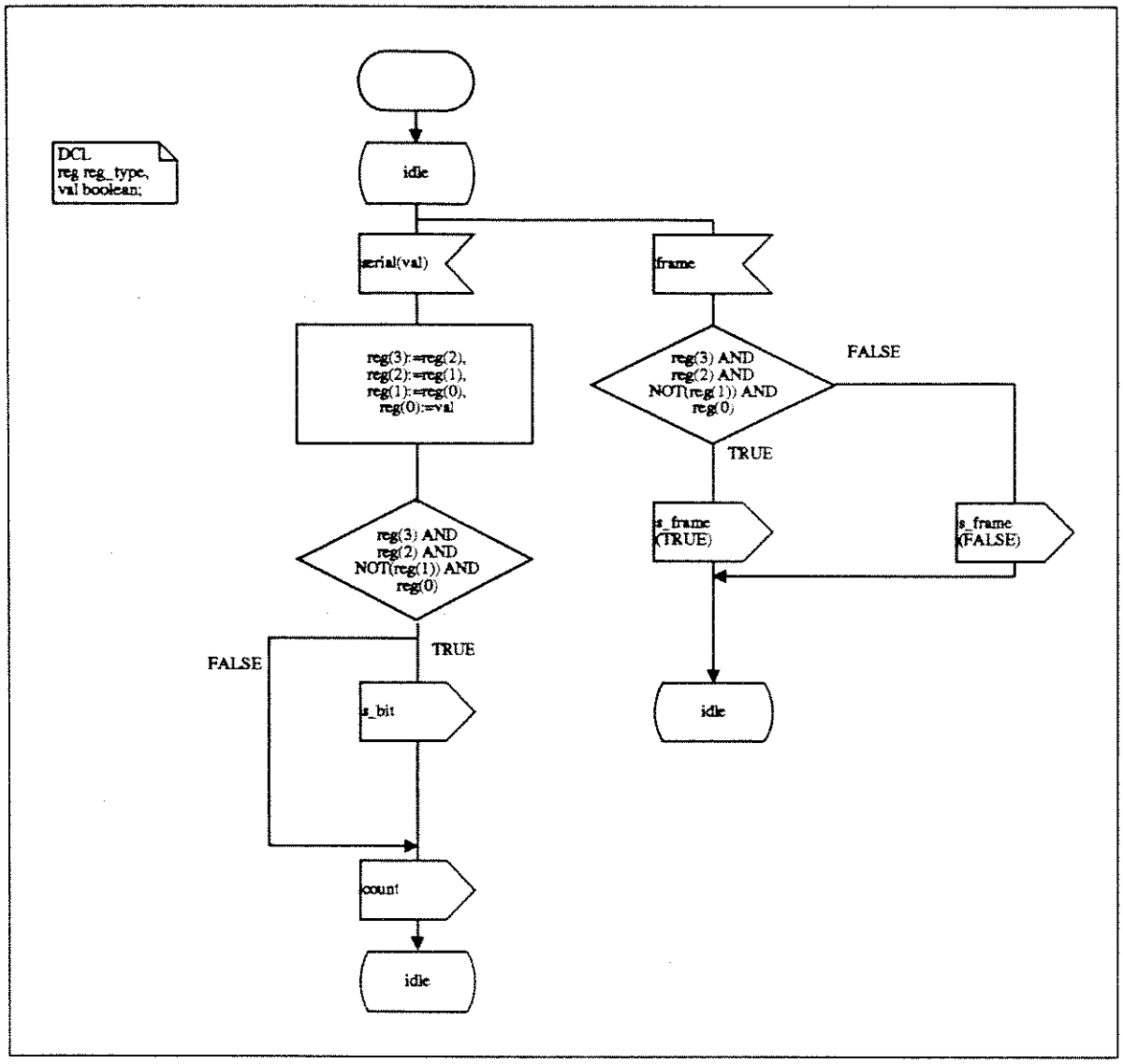
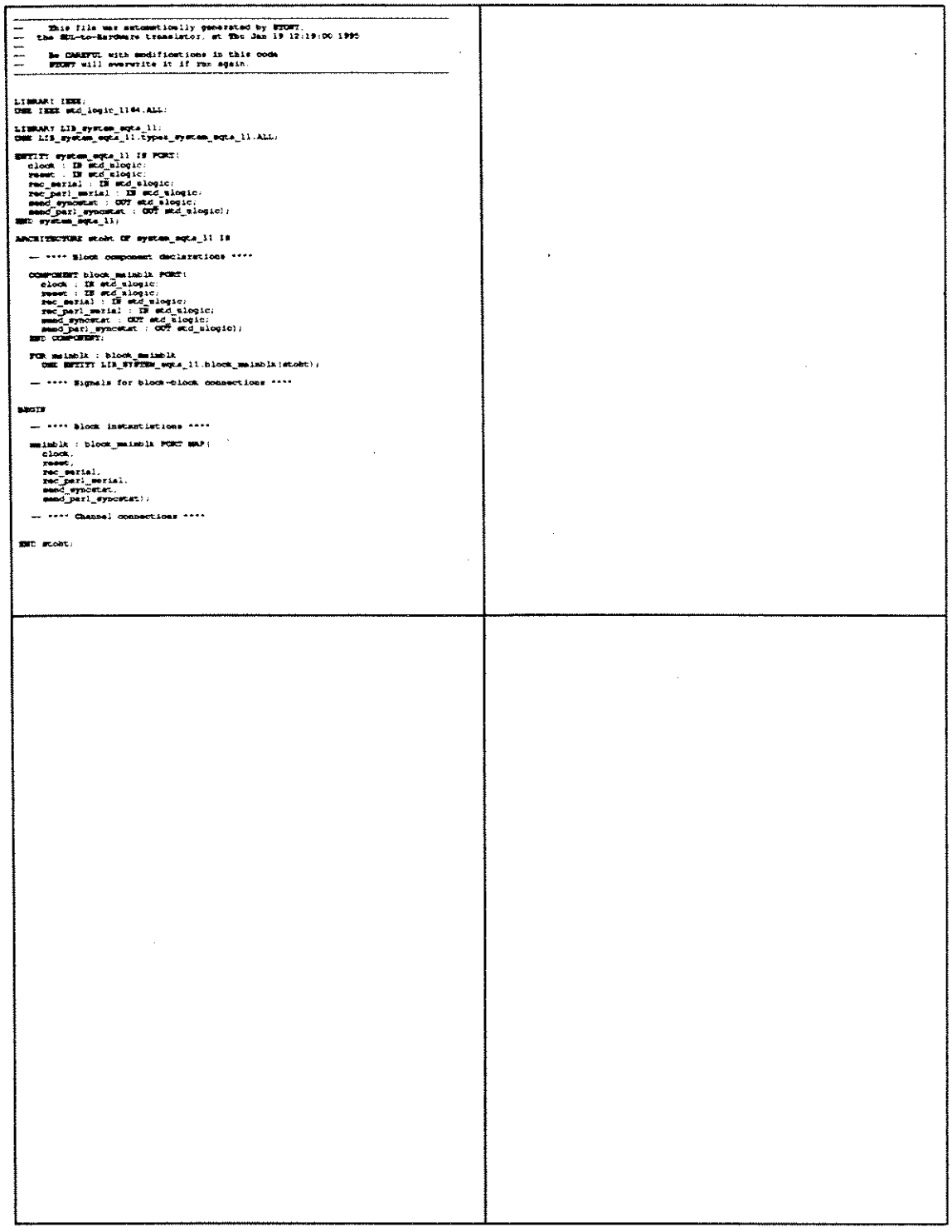
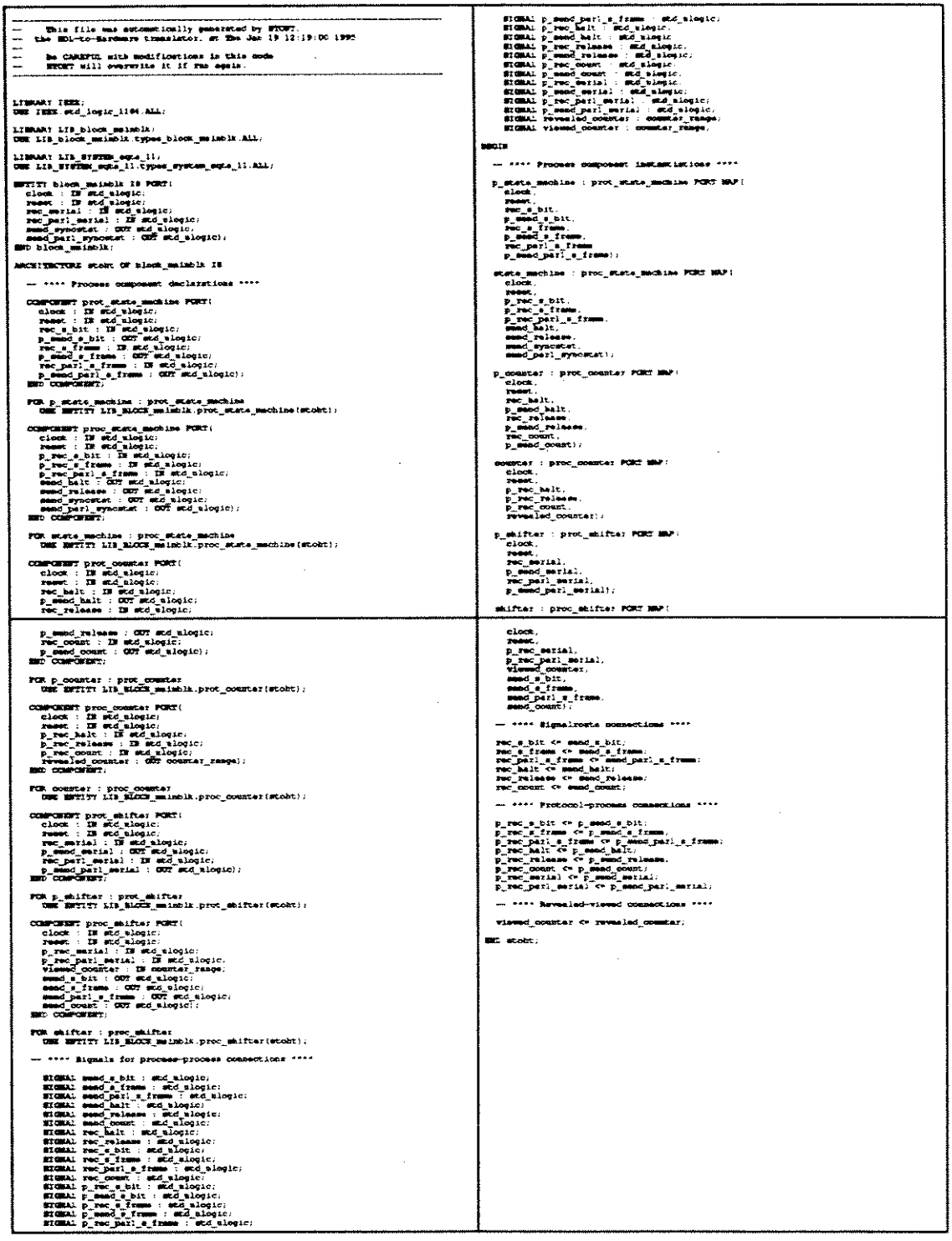


Figura 7.8 **shifter**: Processo do registrador de deslocamento.



```
--------------------------------------------------------------------------------
--     This file was automatically generated by STOHT,
--   the SDL-to-Hardware translator, at Thu Jan 19 12:19:00 1995
--
--     Be CAREFUL with modifications in this code
--     STOHT will overwrite it if run again.
--------------------------------------------------------------------------------


LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

LIBRARY LIB_system_sqts_11;
USE LIB_system_sqts_11.types_system_sqts_11.ALL;

ENTITY system_sqts_11 IS PORT(
   clock : IN std_ulogic;
   reset : IN std_ulogic;
   rec_serial : IN std_ulogic;
   rec_par1_serial : IN std_ulogic;
   send_syncstat : OUT std_ulogic;
   send_par1_syncstat : OUT std_ulogic);
END system_sqts_11;

ARCHITECTURE stoht OF system_sqts_11 IS

   -- **** Block component declarations ****

   COMPONENT block_mainblk PORT(
      clock : IN std_ulogic;
      reset : IN std_ulogic;
      rec_serial : IN std_ulogic;
      rec_par1_serial : IN std_ulogic;
      send_syncstat : OUT std_ulogic;
      send_par1_syncstat : OUT std_ulogic);
   END COMPONENT;

   FOR mainblk : block_mainblk
      USE ENTITY LIB_SYSTEM_sqts_11.block_mainblk(stoht);

   -- **** Signals for block-block connections ****

BEGIN

   -- **** Block instantiations ****

   mainblk : block_mainblk PORT MAP(
      clock,
      reset,
      rec_serial,
      rec_par1_serial,
      send_syncstat,
      send_par1_syncstat);

   -- **** Channel connections ****

END stoht;
```

Figura 7.9 Descrição VHDL do detector de sincronismo obtida a partir do sistema SDL.



```
-------------------------------------------------------------------------------
--     This file was automatically generated by STOHT,
--   the HDL-to-Hardware translator, on Thu Jan 19 12:19:00 1995
--
--     Be CAREFUL with modifications in this code
--     STOHT will overwrite it if run again.
-------------------------------------------------------------------------------


LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

LIBRARY LIB_block_mainblk;
USE LIB_block_mainblk.types_block_mainblk.ALL;

LIBRARY LIB_SYSTEM_sqts_11;
USE LIB_SYSTEM_sqts_11.types_system_sqts_11.ALL;

ENTITY block_mainblk IS PORT(
  clock : IN std_ulogic;
  reset : IN std_ulogic;
  rec_serial : IN std_ulogic;
  rec_par1_serial : IN std_ulogic;
  send_syncstat : OUT std_ulogic;
  send_par1_syncstat : OUT std_ulogic);
END block_mainblk;

ARCHITECTURE stoht OF block_mainblk IS

  -- **** Process component declarations ****

  COMPONENT prot_state_machine PORT(
    clock : IN std_ulogic;
    reset : IN std_ulogic;
    rec_s_bit : IN std_ulogic;
    p_send_s_bit : OUT std_ulogic;
    rec_s_frame : IN std_ulogic;
    p_send_s_frame : OUT std_ulogic;
    rec_par1_s_frame : IN std_ulogic;
    p_send_par1_s_frame : OUT std_ulogic);
  END COMPONENT;

  FOR p_state_machine : prot_state_machine
    USE ENTITY LIB_BLOCK_mainblk.prot_state_machine(stoht);

  COMPONENT proc_state_machine PORT(
    clock : IN std_ulogic;
    reset : IN std_ulogic;
    p_rec_s_bit : IN std_ulogic;
    p_rec_s_frame : IN std_ulogic;
    p_rec_par1_s_frame : IN std_ulogic;
    send_halt : OUT std_ulogic;
    send_release : OUT std_ulogic;
    send_syncstat : OUT std_ulogic;
    send_par1_syncstat : OUT std_ulogic);
  END COMPONENT;

  FOR state_machine : proc_state_machine
    USE ENTITY LIB_BLOCK_mainblk.proc_state_machine(stoht);

  COMPONENT prot_counter PORT(
    clock : IN std_ulogic;
    reset : IN std_ulogic;
    rec_halt : IN std_ulogic;
    p_send_halt : OUT std_ulogic;
    rec_release : IN std_ulogic;
    p_send_release : OUT std_ulogic;
    rec_count : IN std_ulogic;
    p_send_count : OUT std_ulogic);
  END COMPONENT;

  FOR p_counter : prot_counter
    USE ENTITY LIB_BLOCK_mainblk.prot_counter(stoht);

  COMPONENT proc_counter PORT(
    clock : IN std_ulogic;
    reset : IN std_ulogic;
    p_rec_halt : IN std_ulogic;
    p_rec_release : IN std_ulogic;
    p_rec_count : IN std_ulogic;
    revealed_counter : OUT counter_range);
  END COMPONENT;

  FOR counter : proc_counter
    USE ENTITY LIB_BLOCK_mainblk.proc_counter(stoht);

  COMPONENT prot_shifter PORT(
    clock : IN std_ulogic;
    reset : IN std_ulogic;
    rec_serial : IN std_ulogic;
    p_send_serial : OUT std_ulogic;
    rec_par1_serial : IN std_ulogic;
    p_send_par1_serial : OUT std_ulogic);
  END COMPONENT;

  FOR p_shifter : prot_shifter
    USE ENTITY LIB_BLOCK_mainblk.prot_shifter(stoht);

  COMPONENT proc_shifter PORT(
    clock : IN std_ulogic;
    reset : IN std_ulogic;
    p_rec_serial : IN std_ulogic;
    p_rec_par1_serial : IN std_ulogic;
    viewed_counter : IN counter_range;
    send_s_bit : OUT std_ulogic;
    send_s_frame : OUT std_ulogic;
    send_par1_s_frame : OUT std_ulogic;
    send_count : OUT std_ulogic);
  END COMPONENT;

  FOR shifter : proc_shifter
    USE ENTITY LIB_BLOCK_mainblk.proc_shifter(stoht);

  -- **** Signals for process-process connections ****

  SIGNAL send_s_bit : std_ulogic;
  SIGNAL send_s_frame : std_ulogic;
  SIGNAL send_par1_s_frame : std_ulogic;
  SIGNAL send_halt : std_ulogic;
  SIGNAL send_release : std_ulogic;
  SIGNAL send_count : std_ulogic;
  SIGNAL rec_halt : std_ulogic;
  SIGNAL rec_release : std_ulogic;
  SIGNAL rec_s_bit : std_ulogic;
  SIGNAL rec_s_frame : std_ulogic;
  SIGNAL rec_par1_s_frame : std_ulogic;
  SIGNAL rec_count : std_ulogic;
  SIGNAL p_rec_s_bit : std_ulogic;
  SIGNAL p_send_s_bit : std_ulogic;
  SIGNAL p_rec_s_frame : std_ulogic;
  SIGNAL p_send_s_frame : std_ulogic;
  SIGNAL p_rec_par1_s_frame : std_ulogic;
  SIGNAL p_send_par1_s_frame : std_ulogic;
  SIGNAL p_rec_halt : std_ulogic;
  SIGNAL p_send_halt : std_ulogic;
  SIGNAL p_rec_release : std_ulogic;
  SIGNAL p_send_release : std_ulogic;
  SIGNAL p_rec_count : std_ulogic;
  SIGNAL p_send_count : std_ulogic;
  SIGNAL p_rec_serial : std_ulogic;
  SIGNAL p_send_serial : std_ulogic;
  SIGNAL p_rec_par1_serial : std_ulogic;
  SIGNAL p_send_par1_serial : std_ulogic;
  SIGNAL revealed_counter : counter_range;
  SIGNAL viewed_counter : counter_range;

BEGIN

  -- **** Process component instantiations ****

  p_state_machine : prot_state_machine PORT MAP(
    clock,
    reset,
    rec_s_bit,
    p_send_s_bit,
    rec_s_frame,
    p_send_s_frame,
    rec_par1_s_frame,
    p_send_par1_s_frame);

  state_machine : proc_state_machine PORT MAP(
    clock,
    reset,
    p_rec_s_bit,
    p_rec_s_frame,
    p_rec_par1_s_frame,
    send_halt,
    send_release,
    send_syncstat,
    send_par1_syncstat);

  p_counter : prot_counter PORT MAP(
    clock,
    reset,
    rec_halt,
    p_send_halt,
    rec_release,
    p_send_release,
    rec_count,
    p_send_count);

  counter : proc_counter PORT MAP(
    clock,
    reset,
    p_rec_halt,
    p_rec_release,
    p_rec_count,
    revealed_counter);

  p_shifter : prot_shifter PORT MAP(
    clock,
    reset,
    rec_serial,
    p_send_serial,
    rec_par1_serial,
    p_send_par1_serial);

  shifter : proc_shifter PORT MAP(
    clock,
    reset,
    p_rec_serial,
    p_rec_par1_serial,
    viewed_counter,
    send_s_bit,
    send_s_frame,
    send_par1_s_frame,
    send_count);

  -- **** Signalroute connections ****

  rec_s_bit <= send_s_bit;
  rec_s_frame <= send_s_frame;
  rec_par1_s_frame <= send_par1_s_frame;
  rec_halt <= send_halt;
  rec_release <= send_release;
  rec_count <= send_count;

  -- **** Protocol-process connections ****

  p_rec_s_bit <= p_send_s_bit;
  p_rec_s_frame <= p_send_s_frame;
  p_rec_par1_s_frame <= p_send_par1_s_frame;
  p_rec_halt <= p_send_halt;
  p_rec_release <= p_send_release;
  p_rec_count <= p_send_count;
  p_rec_serial <= p_send_serial;
  p_rec_par1_serial <= p_send_par1_serial;

  -- **** Revealed-viewed connections ****

  viewed_counter <= revealed_counter;

END stoht;
```

Figura 7.10 Descrição VHDL do bloco **mainblk**.

A síntese, a partir deste código VHDL, utilizando a ferramenta AutoLogic (Mentor Graphics Corp.) particionou o sistema em três níveis:

- Sistema, contendo um componente representando o bloco **mainblk**
- Bloco, contendo seis símbolos, correspondentes aos processos **counter**, **shifter**, **state_machine** e seus respectivos protocolos
- Processo, contendo componentes da biblioteca genérica (portas lógicas, flip-flops, multiplexadores) resultantes da síntese

O diagrama esquemático que representa o **System** SDL do detector de sincronismo, composto do símbolo correspondente ao único bloco do sistema (**mainblk**) é mostrado na Fig. 7.14.

O diagrama esquemático que representa o **Block** SDL **mainblk** composto de símbolos representando os três processos e seus respectivos protocolos é mostrado na Fig. 7.15.

Nas figuras 7.16 a 7.21 são mostrados os esquemáticos resultantes dos processos SDL: máquina de estado do detector de sincronismo (processo **state_machine**, Fig. 7.16) e respectivo protocolo (Fig. 7.17); contador (processo **counter**, Fig. 7.18) e respectivo protocolo (Fig. 7.19); registrador de deslocamento (processo **shifter**, Fig. 7.20) e respectivo protocolo (Fig. 7.21).

7.1.6 Validação

Tanto a descrição em VHDL quanto o circuito sintetizado foram simulados com a ferramenta integrada de simulação da Mentor Graphics (*QuickSim*).

Foi gerado um padrão de excitações nas entradas destes circuitos, através da definição de estímulos nos sinais VHDL **clk**, **rec_serial**, **rec_par1_serial** e **reset**. Este padrão foi utilizado tanto para a simulação VHDL quanto para a simulação do circuito sintetizado a nível de portas lógicas.

- O relógio **clk** foi definido com período de 50ns, simétrico (25ns em nível baixo e 25ns em nível alto).
- O sinal de reset assíncrono, ativo alto, é aplicado nos primeiros 100ns da simulação, ficando inativo no restante do tempo.
- Os sinais de entrada serial (**rec_serial** e **rec_par1_serial**) foram manipulados para levar o sistema de sua condição inicial "fora de sincronismo" para os demais estados.

Na Fig. 7.22 é ilustrado o início da simulação da descrição VHDL.

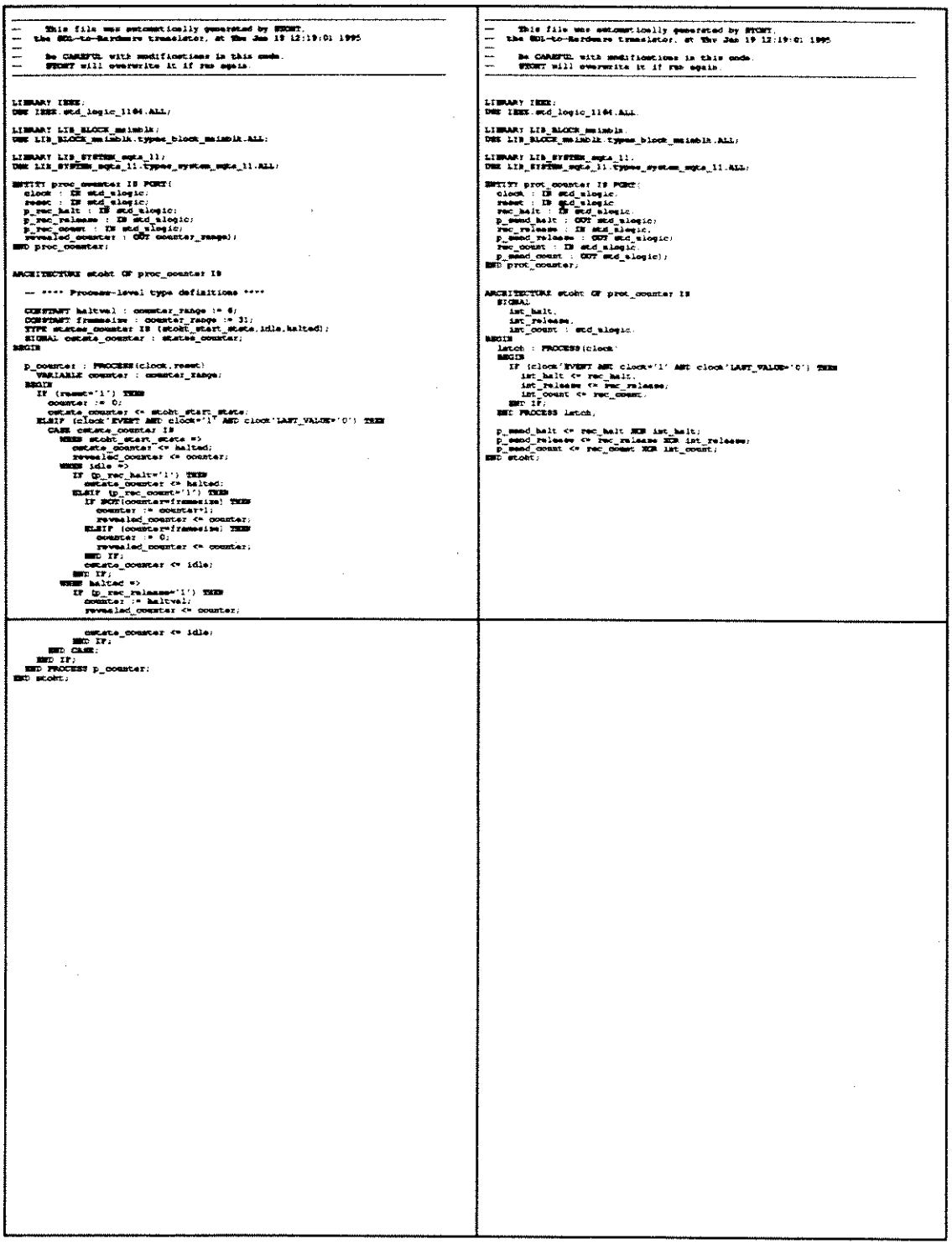
Nota-se que o pulso de **reset** leva os processos SDL às suas condições iniciais (estado **stoht_start_state**) bem como os sinais VHDL que são utilizados para representar o envio de sinais SDL (sinais de prefixo **send_**) são forçados ao zero lógico. Após o primeiro pulso de **clock**, os processos **state_machine** e **counter** transitam para, respectivamente, **outsync** e **halted**.

Entre os instantes 200ns e 400ns é aplicada a palavra de sincronismo **W=1101** no sinal VHDL de entrada (**rec_par1_serial**). O processo **shifter**, ao reconhecer esta palavra, inverte o sinal VHDL **send_s_bit**; o protocolo **proc_shifter** é acionado e gera um pulso de largura de um período de relógio no sinal VHDL **p_send_s_bit**. Com este pulso, o processo **shifter** sinaliza ao protocolo do processo **state_machine** a presença de uma palavra de sincronismo nos últimos quatro bits lidos.

O processo **state_machine** recebe o sinal VHDL **p_send_s_bit** de seu protocolo, e muda do estado **outsync** para **wait_0**.

Durante a transição de estados, o processo **state_machine** envia o sinal SDL **release** para o contador (invertendo o sinal VHDL **send_release**).

O processo **counter** recebe este sinal, transita de **halted** para **idle** e coloca o valor inicial (constante

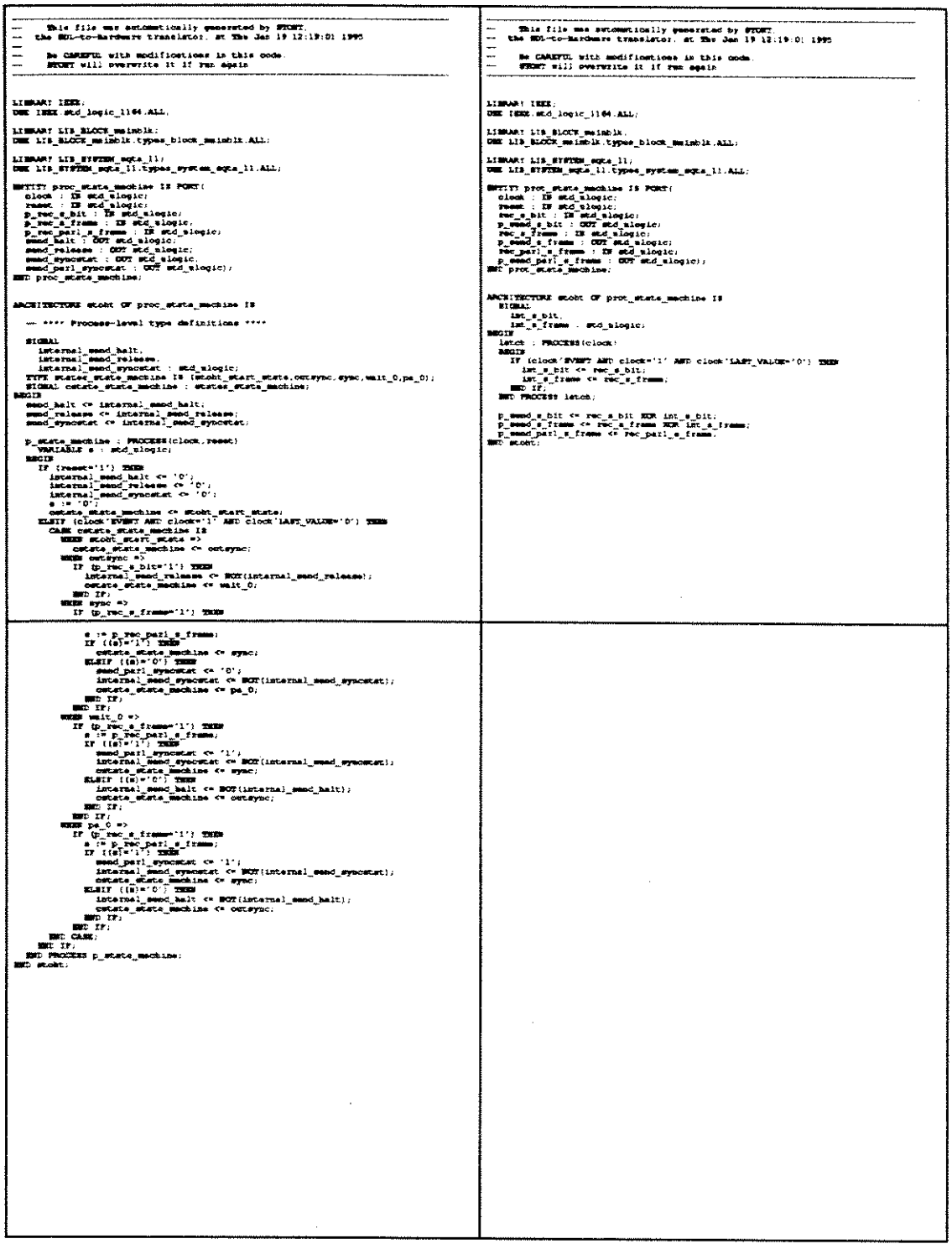


```
--------------------------------------------------------------------------------
--    This file was automatically generated by ESCHT,
--  the SDL-to-Hardware translator, at Thu Jan 19 12:19:01 1995
--
--    Be CAREFUL with modifications in this code.
--    ESCHT will overwrite it if run again.
--------------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

LIBRARY LIB_BLOCK_assimblk;
USE LIB_BLOCK_assimblk.types_block_assimblk.ALL;

LIBRARY LIB_SYSTEM_sqta_11;
USE LIB_SYSTEM_sqta_11.types_system_sqta_11.ALL;

ENTITY proc_counter IS PORT(
  clock : IN std_ulogic;
  reset : IN std_ulogic;
  p_rec_halt : IN std_ulogic;
  p_rec_release : IN std_ulogic;
  p_rec_count : IN std_ulogic;
  revealed_counter : OUT counter_range);
END proc_counter;

ARCHITECTURE stoht OF proc_counter IS

  -- **** Process-level type definitions ****

  CONSTANT haltval : counter_range := 6;
  CONSTANT framesize : counter_range := 31;
  TYPE states_counter IS (stoht_start_state,idle,halted);
  SIGNAL cstate_counter : states_counter;
BEGIN

  p_counter : PROCESS(clock,reset)
    VARIABLE counter : counter_range;
  BEGIN
    IF (reset='1') THEN
      counter := 0;
      cstate_counter <= stoht_start_state;
    ELSIF (clock'EVENT AND clock='1' AND clock'LAST_VALUE='0') THEN
      CASE cstate_counter IS
        WHEN stoht_start_state =>
          cstate_counter <= halted;
          revealed_counter <= counter;
        WHEN idle =>
          IF (p_rec_halt='1') THEN
            cstate_counter <= halted;
          ELSIF (p_rec_count='1') THEN
            IF NOT(counter=framesize) THEN
              counter := counter+1;
              revealed_counter <= counter;
            ELSIF (counter=framesize) THEN
              counter := 0;
              revealed_counter <= counter;
            END IF;
            cstate_counter <= idle;
          END IF;
        WHEN haltac =>
          IF (p_rec_release='1') THEN
            counter := haltval;
            revealed_counter <= counter;

            cstate_counter <= idle;
          END IF;
      END CASE;
    END IF;
  END PROCESS p_counter;
END stoht;
```

```
--------------------------------------------------------------------------------
--    This file was automatically generated by ESCHT,
--  the SDL-to-Hardware translator, at Thu Jan 19 12:19:0: 1995
--
--    Be CAREFUL with modifications in this code.
--    ESCHT will overwrite it if run again.
--------------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

LIBRARY LIB_BLOCK_assimblk;
USE LIB_BLOCK_assimblk.types_block_assimblk.ALL;

LIBRARY LIB_SYSTEM_sqta_11;
USE LIB_SYSTEM_sqta_11.types_system_sqta_11.ALL;

ENTITY prot_counter IS PORT(
  clock : IN std_ulogic;
  reset : IN std_ulogic;
  rec_halt : IN std_ulogic;
  p_send_halt : OUT std_ulogic;
  rec_release : IN std_ulogic;
  p_send_release : OUT std_ulogic;
  rec_count : IN std_ulogic;
  p_send_count : OUT std_ulogic);
END prot_counter;

ARCHITECTURE stoht OF prot_counter IS
  SIGNAL
    int_halt,
    int_release,
    int_count : std_ulogic;
BEGIN
  latch : PROCESS(clock)
  BEGIN
    IF (clock'EVENT AND clock='1' AND clock'LAST_VALUE='0') THEN
      int_halt <= rec_halt;
      int_release <= rec_release;
      int_count <= rec_count;
    END IF;
  END PROCESS latch;

  p_send_halt <= rec_halt XOR int_halt;
  p_send_release <= rec_release XOR int_release;
  p_send_count <= rec_count XOR int_count;
END stoht;
```

Figura 7.11 Descrições VHDL do processo **counter** e de seu respectivo protocolo.

Figura 7.12 Descrições VHDL do processo **shifter** e de seu respectivo protocolo.



```
--  ------------------------------------------------------------------
--    This file was automatically generated by STATE.
--  the HDL-to-Hardware translator, at Thu Jan 19 12:19:01 1995
--
--    Be CAREFUL with modifications in this code.
--    STATE will overwrite it if run again
--  ------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

LIBRARY LIB_BLOCK_mainblk;
USE LIB_BLOCK_mainblk.types_block_mainblk.ALL;

LIBRARY LIB_SYSTEM_sqta_11;
USE LIB_SYSTEM_sqta_11.types_system_sqta_11.ALL;

ENTITY proc_state_machine IS PORT(
  clock : IN std_ulogic;
  reset : IN std_ulogic;
  p_rec_s_bit : IN std_ulogic;
  p_rec_s_frame : IN std_ulogic;
  p_rec_par1_s_frame : IN std_ulogic;
  smod_halt : OUT std_ulogic;
  smod_release : OUT std_ulogic;
  smod_syncstat : OUT std_ulogic;
  smod_par1_syncstat : OUT std_ulogic);
END proc_state_machine;

ARCHITECTURE stobt OF proc_state_machine IS

  -- **** Process-level type definitions ****

  SIGNAL
    internal_smod_halt,
    internal_smod_release,
    internal_smod_syncstat : std_ulogic;
  TYPE states_state_machine IS (stobt_start_state,outsync,sync,wait_0,ps_0);
  SIGNAL cstate_state_machine : states_state_machine;
BEGIN
  smod_halt <= internal_smod_halt;
  smod_release <= internal_smod_release;
  smod_syncstat <= internal_smod_syncstat;

  p_state_machine : PROCESS(clock,reset)
    VARIABLE s : std_ulogic;
  BEGIN
    IF (reset='1') THEN
      internal_smod_halt <= '0';
      internal_smod_release <= '0';
      internal_smod_syncstat <= '0';
      s := '0';
      cstate_state_machine <= stobt_start_state;
    ELSIF (clock'EVENT AND clock='1' AND clock'LAST_VALUE='0') THEN
      CASE cstate_state_machine IS
        WHEN stobt_start_state =>
          cstate_state_machine <= outsync;
        WHEN outsync =>
          IF (p_rec_s_bit='1') THEN
            internal_smod_release <= NOT(internal_smod_release);
            cstate_state_machine <= wait_0;
          END IF;
        WHEN sync =>
          IF (p_rec_s_frame='1') THEN
            s := p_rec_par1_s_frame;
            IF ((s)='1') THEN
              cstate_state_machine <= sync;
            ELSIF ((s)='0') THEN
              smod_par1_syncstat <= '0';
              internal_smod_syncstat <= NOT(internal_smod_syncstat);
              cstate_state_machine <= ps_0;
            END IF;
          END IF;
        WHEN wait_0 =>
          IF (p_rec_s_frame='1') THEN
            s := p_rec_par1_s_frame;
            IF ((s)='1') THEN
              smod_par1_syncstat <= '1';
              internal_smod_syncstat <= NOT(internal_smod_syncstat);
              cstate_state_machine <= sync;
            ELSIF ((s)='0') THEN
              internal_smod_halt <= NOT(internal_smod_halt);
              cstate_state_machine <= outsync;
            END IF;
          END IF;
        WHEN ps_0 =>
          IF (p_rec_s_frame='1') THEN
            s := p_rec_par1_s_frame;
            IF ((s)='1') THEN
              smod_par1_syncstat <= '1';
              internal_smod_syncstat <= NOT(internal_smod_syncstat);
              cstate_state_machine <= sync;
            ELSIF ((s)='0') THEN
              internal_smod_halt <= NOT(internal_smod_halt);
              cstate_state_machine <= outsync;
            END IF;
          END IF;
      END CASE;
    END IF;
  END PROCESS p_state_machine;
END stobt;
```

```
--  ------------------------------------------------------------------
--    This file was automatically generated by STATE.
--  the HDL-to-Hardware translator, at Thu Jan 19 12:19:01 1995
--
--    Be CAREFUL with modifications in this code.
--    STATE will overwrite it if run again
--  ------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

LIBRARY LIB_BLOCK_mainblk;
USE LIB_BLOCK_mainblk.types_block_mainblk.ALL;

LIBRARY LIB_SYSTEM_sqta_11;
USE LIB_SYSTEM_sqta_11.types_system_sqta_11.ALL;

ENTITY prot_state_machine IS PORT(
  clock : IN std_ulogic;
  reset : IN std_ulogic;
  rec_s_bit : IN std_ulogic;
  p_smod_s_bit : OUT std_ulogic;
  rec_s_frame : IN std_ulogic;
  p_smod_s_frame : OUT std_ulogic;
  rec_par1_s_frame : IN std_ulogic;
  p_smod_par1_s_frame : OUT std_ulogic);
END prot_state_machine;

ARCHITECTURE stobt OF prot_state_machine IS
  SIGNAL
    int_s_bit,
    int_s_frame : std_ulogic;
BEGIN
  latch : PROCESS(clock)
  BEGIN
    IF (clock'EVENT AND clock='1' AND clock'LAST_VALUE='0') THEN
      int_s_bit <= rec_s_bit;
      int_s_frame <= rec_s_frame;
    END IF;
  END PROCESS latch;

  p_smod_s_bit <= rec_s_bit XOR int_s_bit;
  p_smod_s_frame <= rec_s_frame XOR int_s_frame;
  p_smod_par1_s_frame <= rec_par1_s_frame;
END stobt;
```

Figura 7.13 Descrições VHDL do processo **state_machine** e de seu respectivo protocolo.

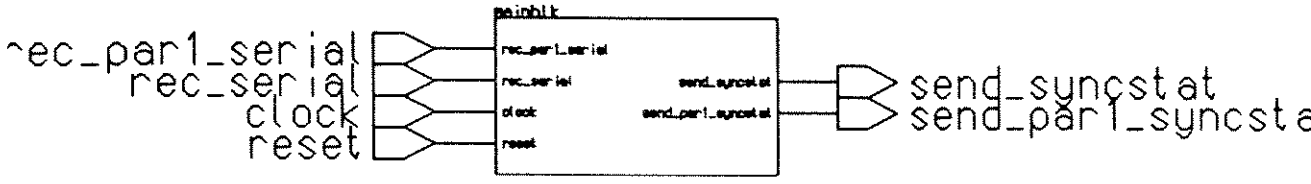


Figura 7.14 Resultado da síntese : ENTITY VHDL obtida, como resultado da síntese utilizando a ferramenta **AutoLogic**, da descrição SDL do sistema **sqta**.

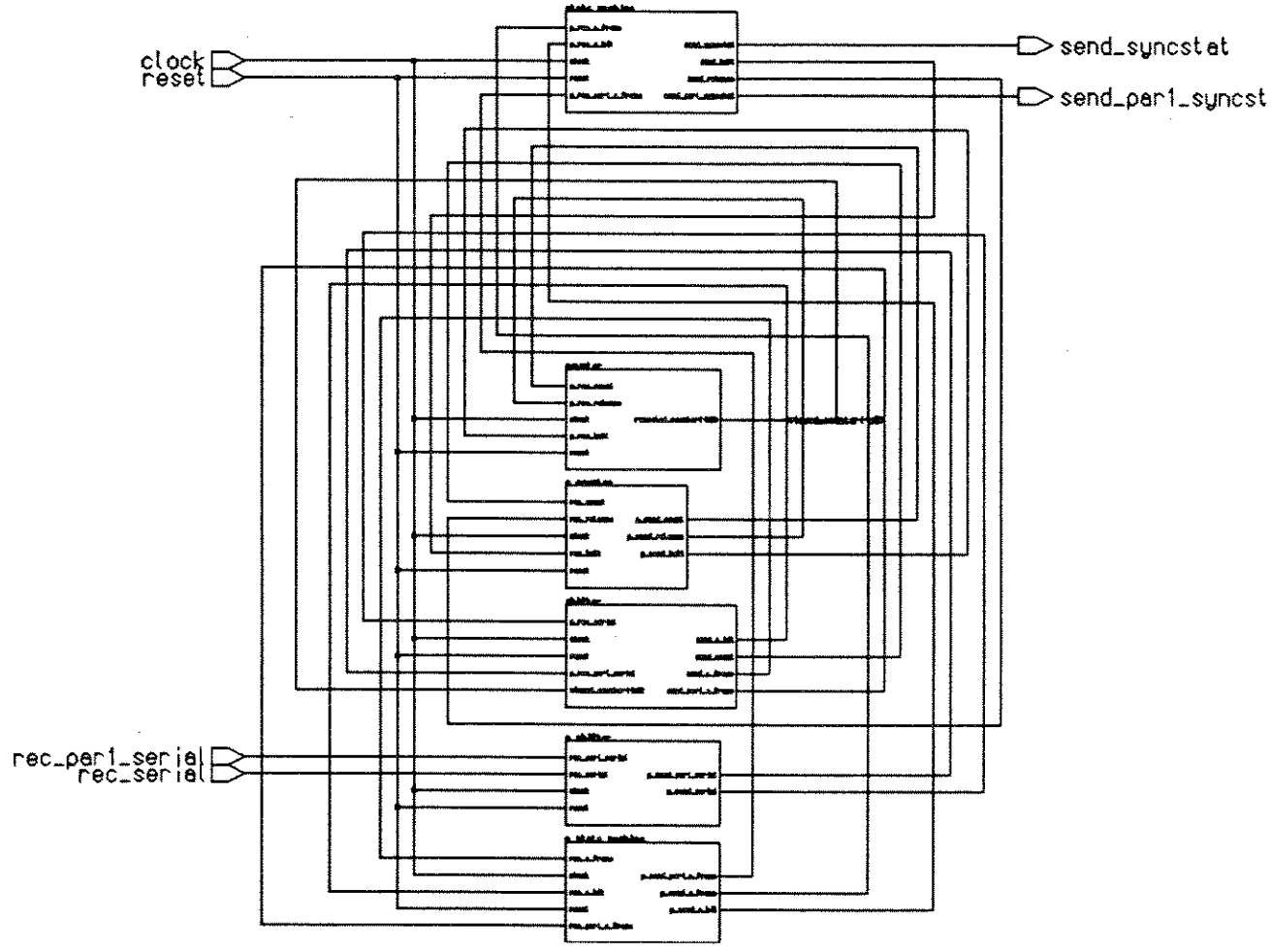


Figura 7.15 Circuito VHDL correspondente ao bloco SDL **mainblk**.

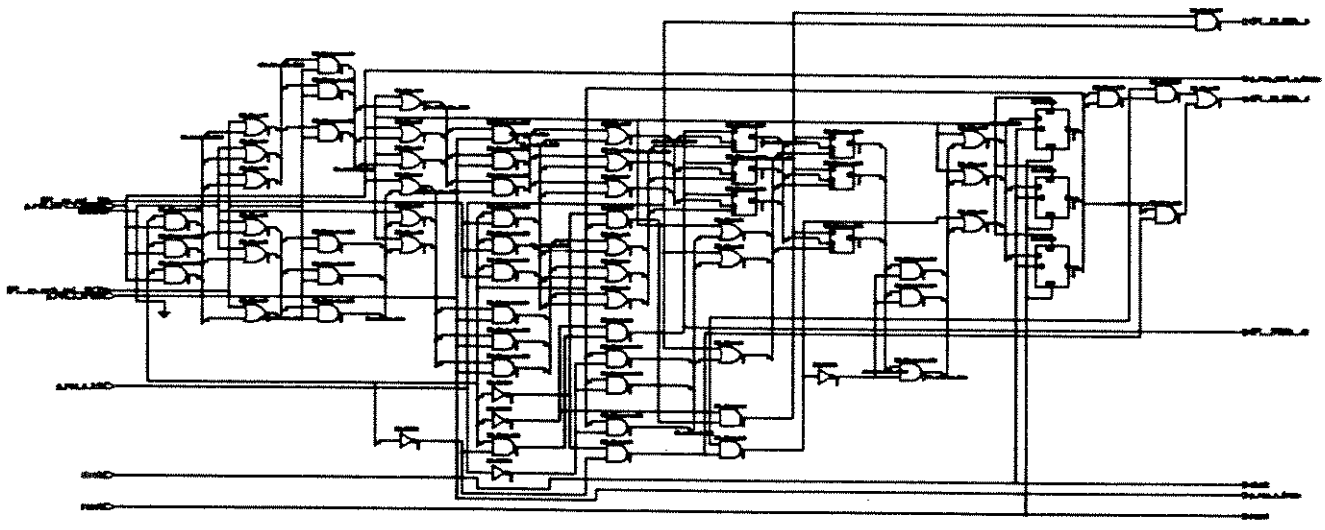


Figura 7.16 Circuito VHDL correspondente ao processo SDL **state_machine**.

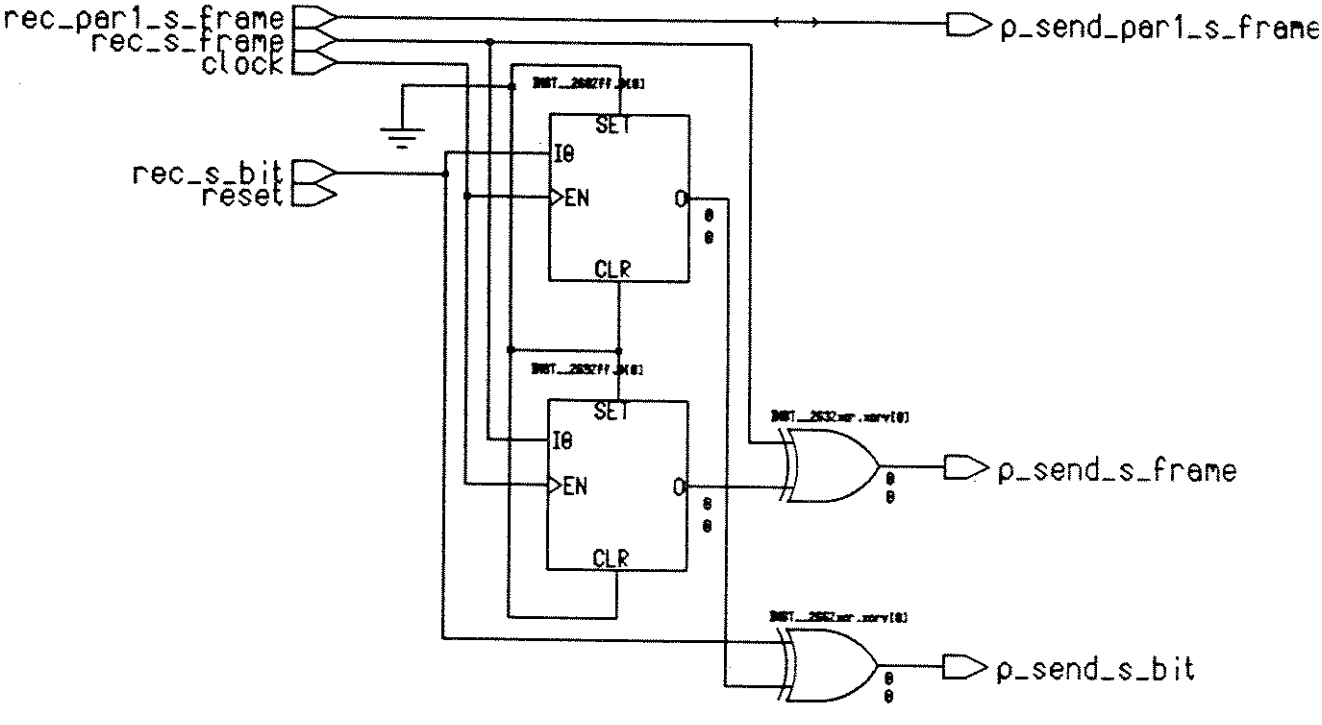


Figura 7.17 Circuito VHDL correspondente ao protocolo de comunicação do processo SDL **state_machine**.

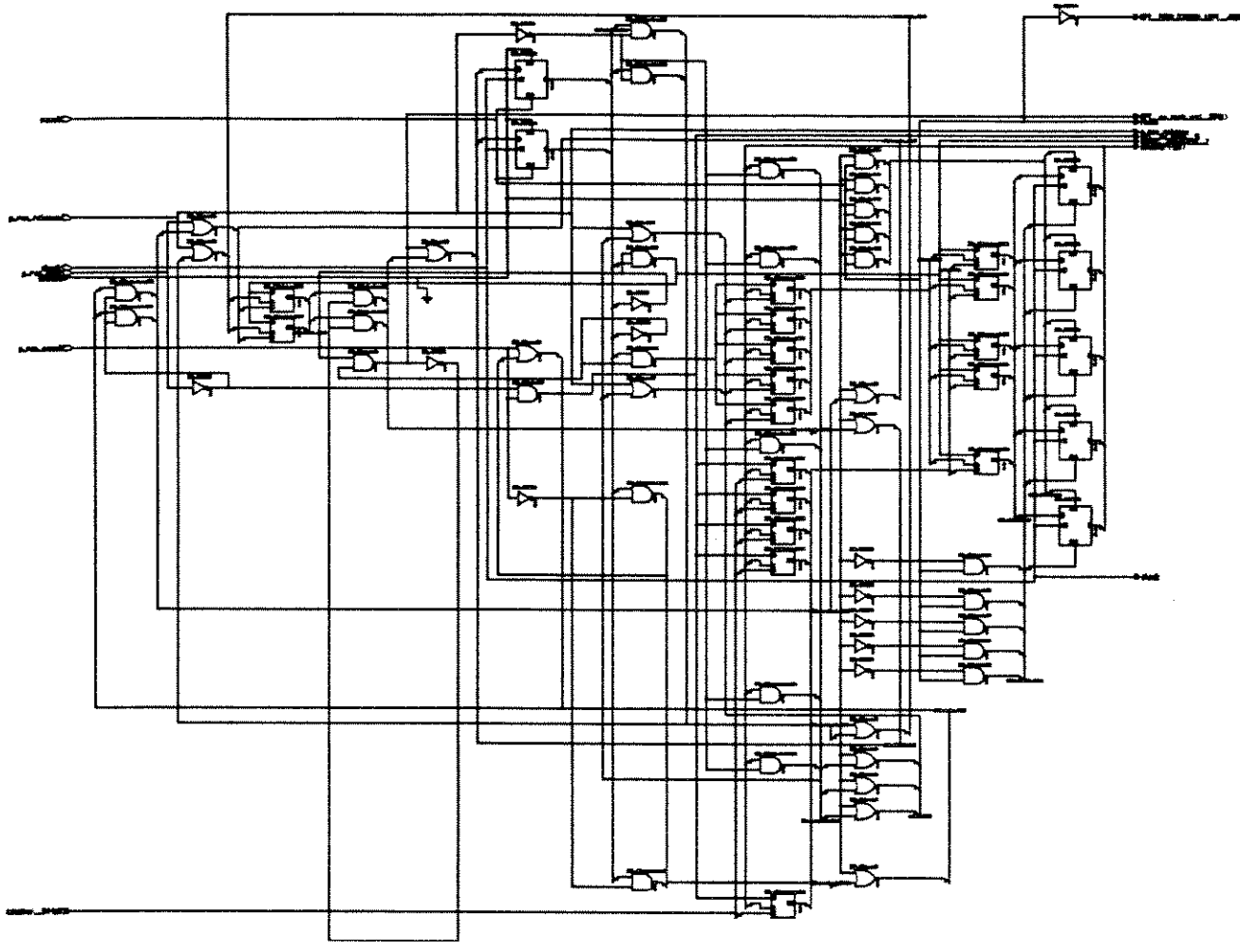


Figura 7.18 Circuito VHDL correspondente ao processo SDL **counter**.

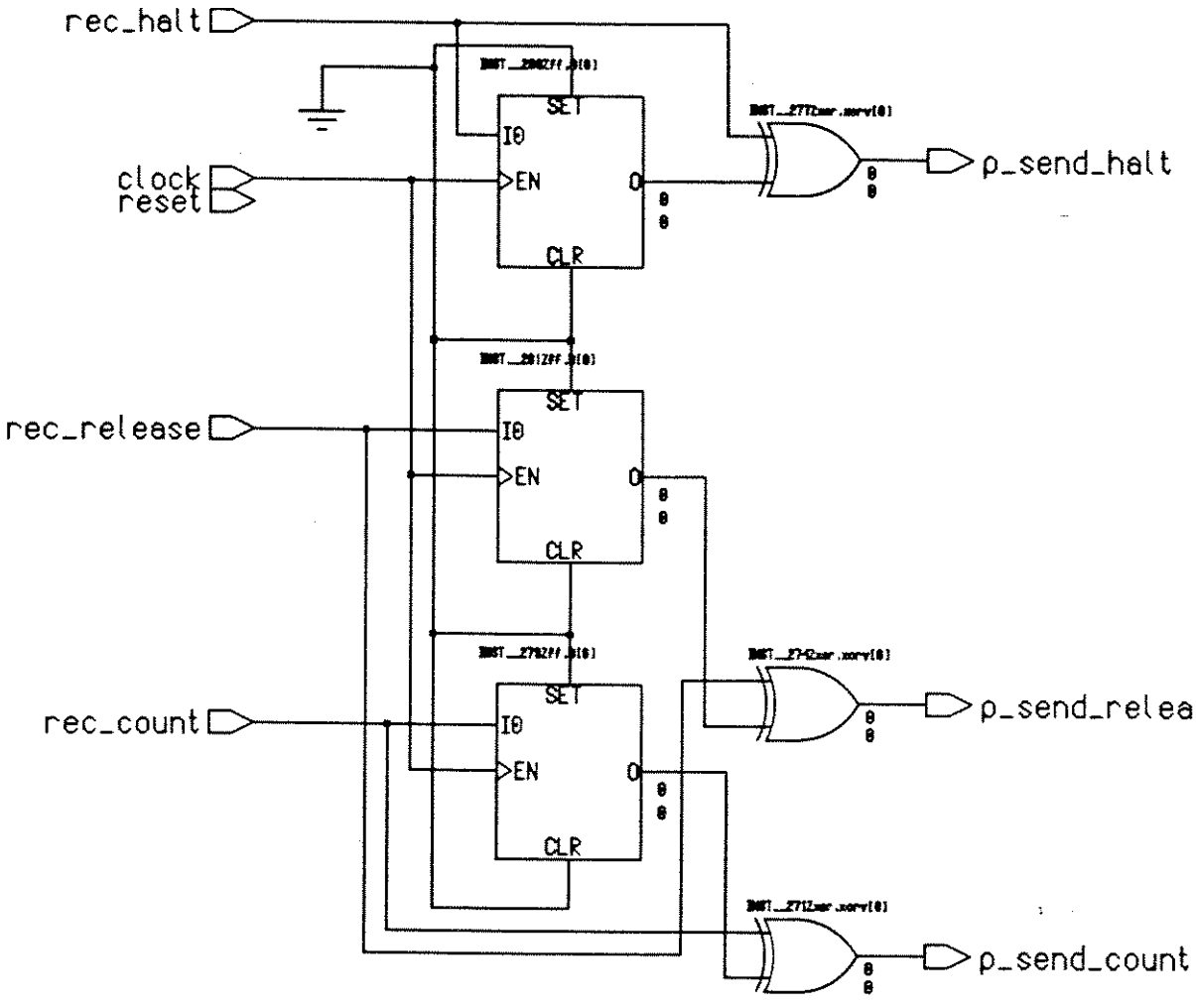


Figura 7.19 Circuito VHDL correspondente ao protocolo de comunicação do processo SDL **counter**.

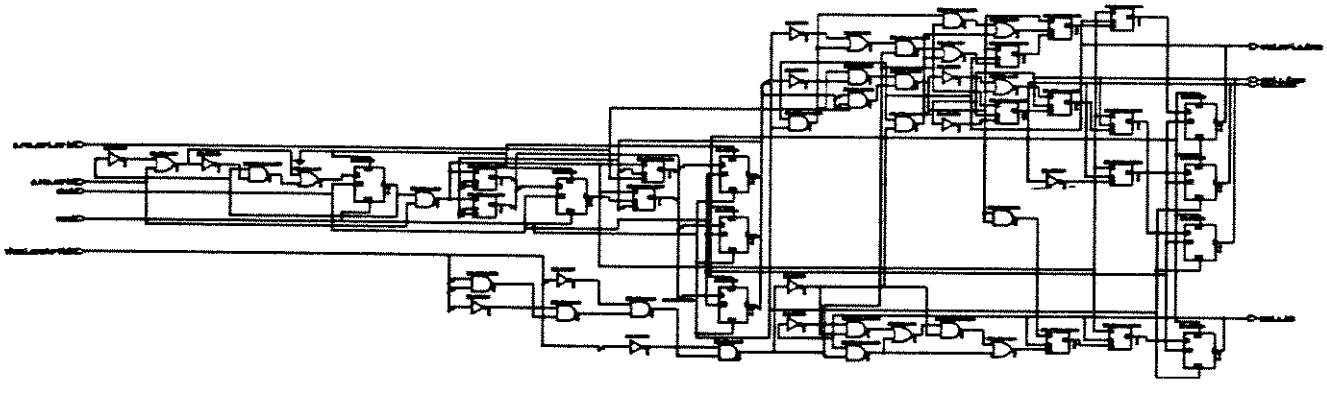


Figura 7.20 Circuito VHDL correspondente ao processo SDL **shifter**.

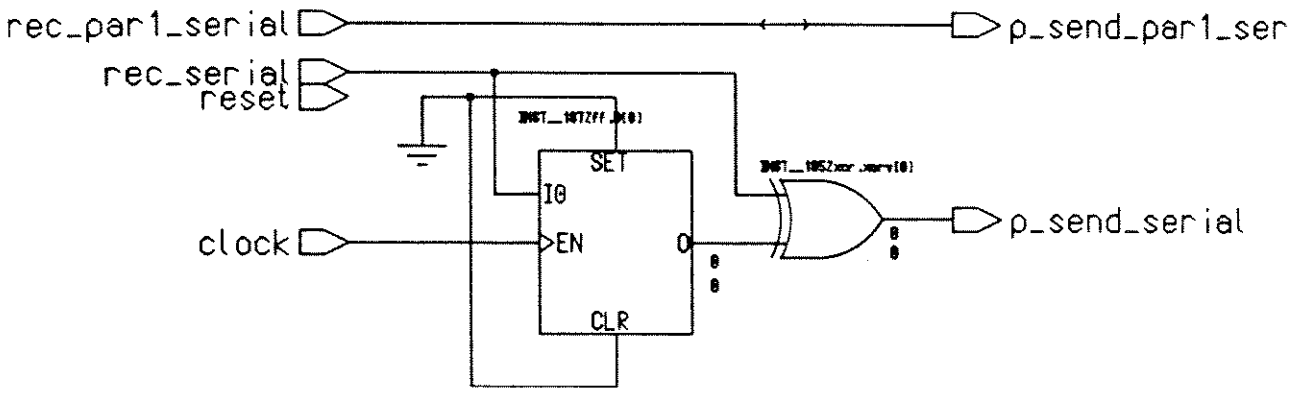


Figura 7.21 Circuito VHDL correspondente ao protocolo de comunicação do processo SDL **shifter**.

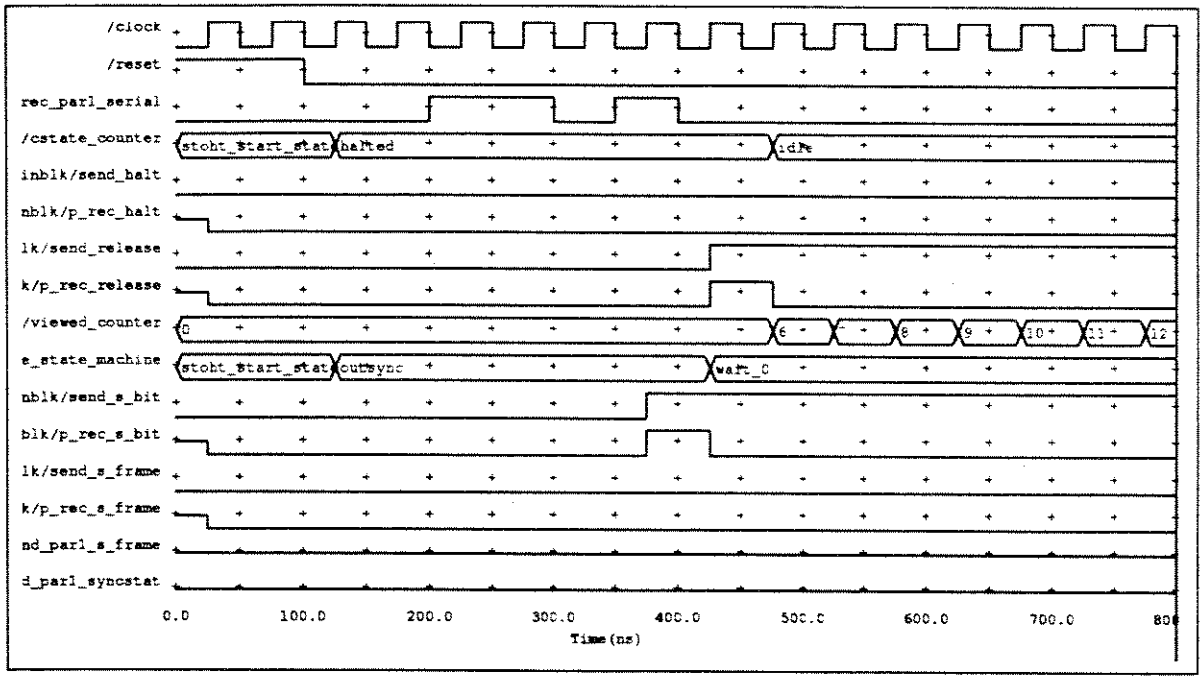


Figura 7.22 Simulação da descrição VHDL (condições iniciais).

HALTVAL) no contador. Nota-se que este valor foi obtido levando-se em conta os atrasos no envio e recepção de sinais.

Na Fig. 7.23 é ilustrada a simulação completa de vários quadros.

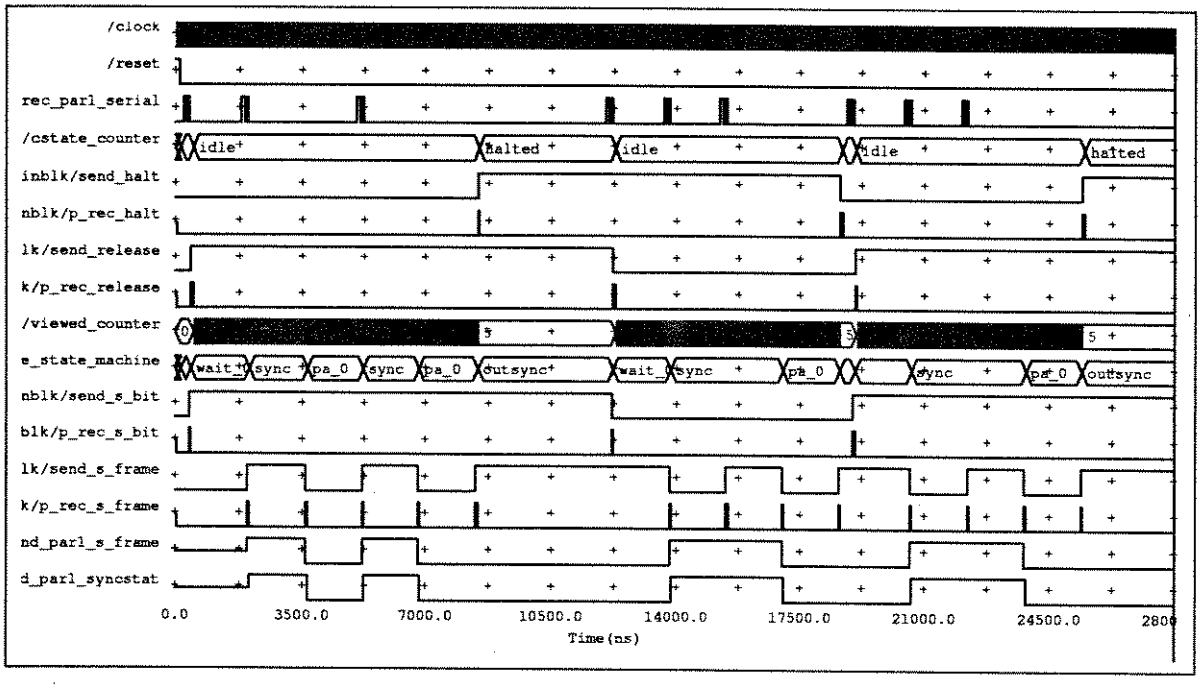


Figura 7.23 Simulação da descrição VHDL (completa).

O sinal **rec_par1_serial** é mantido em zero durante a maior parte da simulação, para maior facilidade de visualização. A palavra de sincronismo **W=1101** é inserida em seis pontos da simulação: nos instantes 200ns, 1800ns, 5000ns, 12000ns, 13600ns e 15200ns.

Dado que o quadro é composto de 8 canais de 4 bits, e que o relógio de bit é de 50ns, tem-se que o período do quadro lógico é de $4*8*50 = 1600ns$. Com o reconhecimento da primeira ocorrência da palavra de sincronismo **W** em 200ns, o sistema passa para o estado de pré-sincronismo **wait_0** e marca este canal como sendo o de sincronismo (canal zero).

Passado 1 quadro (200ns+1600ns), o sistema novamente lê a palavra de sincronismo no canal zero e passa para o estado **sync**. No próximo quadro (3400ns), a palavra de sincronismo não está presente e o sistema entra em pré-alarme (**pa_0**). No quadro seguinte (5000ns), a palavra **W** está presente, e o sistema volta ao sincronismo (**sync**).

Nos dois quadros seguintes, a palavra **W** não está presente no canal zero, e o sistema sofre a transição **sync - pa_0 - outsync**. Nota-se que o contador inibe sua contagem (estado **halted**) quando o sistema sai de sincronismo. É interessante notar a retomada de sincronismo iniciada em 12000ns, concretizada em 13600ns e mantida em 15200ns, por mais um quadro.

Nesta simulação também estão presentes os sinais VHDL que representam envio (prefixo **send_**) e recepção (prefixo **rec_**) de sinais SDL, mostrando o comportamento dinâmico dos sinais no circuito.

Os pulsos presentes nos sinais VHDL de prefixo **rec_** correspondem aos instantes de envio do sinais SDL correspondentes.

As figuras 7.24 e 7.25 mostradas a seguir representam a aplicação deste mesmo padrão no circuito sintetizado a partir da descrição VHDL. A simulação para o circuito sintetizado ocorre no nível de portas lógicas; são mostrados os mesmos intervalos de tempo representados nas figuras 7.22 e 7.23, permitindo

a comparação dos resultados.

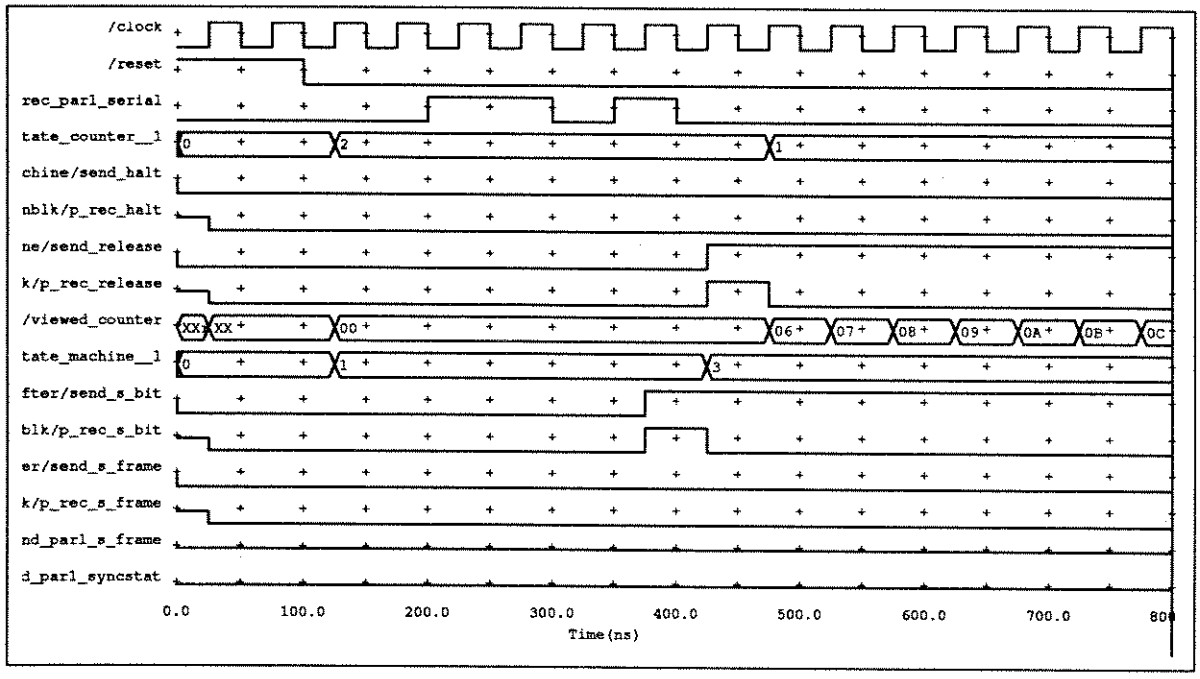


Figura 7.24 Simulação do circuito sintetizado (condições iniciais)

Nota-se que o algoritmo de síntese associou o valor **0** ao estado **stoht_start_state**, **1** ao estado **outsync**, **2** ao estado **sync**, **3** ao estado **wait_0** e **4** ao estado **pa_0**. O comportamento do circuito foi idêntico ao obtido com a descrição VHDL.

7.2 Máquina de venda

7.2.1 Introdução

Para exemplificar a aplicação da metodologia proposta, foi desenvolvido o projeto de uma máquina de venda de produtos automática (*Vending Machine*). Esta máquina foi originalmente usada como exemplo por Perry em [28] para ilustrar os passos de um projeto Top-Down em VHDL.

No exemplo original [28], são realizadas várias decomposições do projeto até que sua representação atinja o nível de portas lógicas. Aplicando a metodologia proposta neste trabalho, a conversão de SDL comportamental para o circuito digital foi automática, via ferramentas de conversão e de síntese.

7.2.2 Especificação

A máquina recebe dinheiro do comprador através de moedas de quatro valores distintos: 5 centavos (**nickel**), 10 centavos (**dime**), 25 centavos (**quarter**) e 50 centavos (**half**).

A máquina comporta quatro tipos de produto para venda: biscoitos tipo **cookie**, custando 55 centavos, salgadinhos tipo **pretzel**, custando 50 centavos, batata frita (**chip**), custando 45 centavos, e doces do tipo **doughnut**, custando 60 centavos.

A máquina tem dois tipos de entrada disponíveis para o consumidor: um orifício para cada tipo de

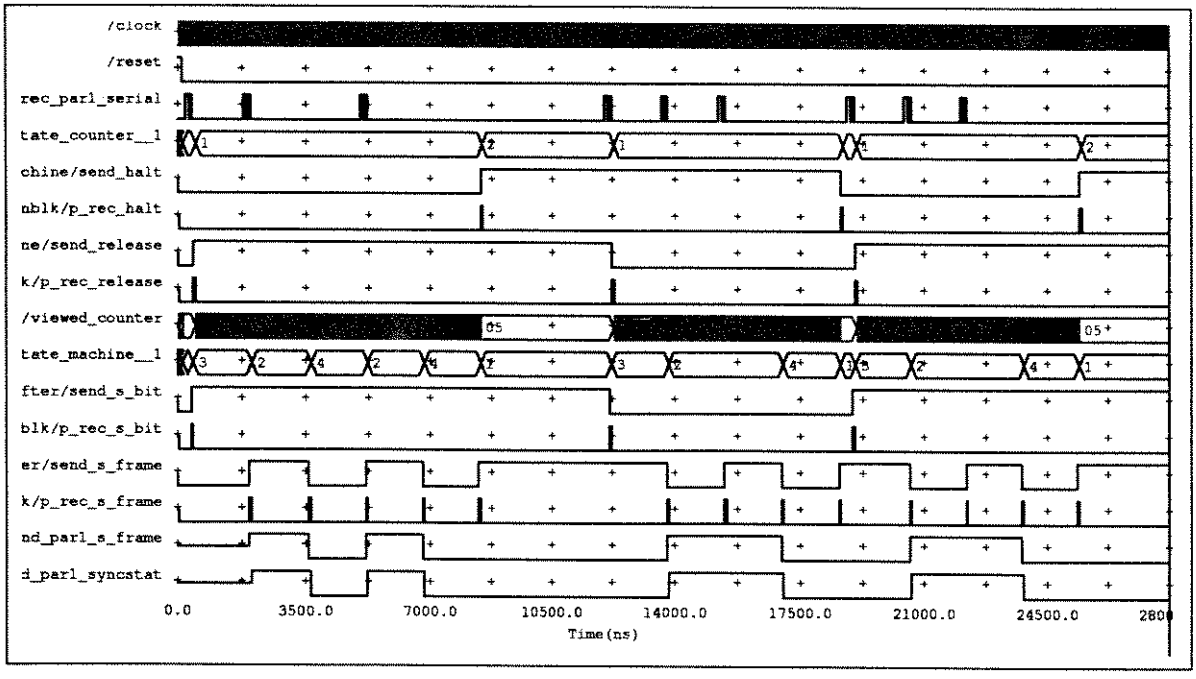


Figura 7.25 Simulação do circuito sintetizado (completa)

moeda e quatro botões, correspondentes ao pedido de compra de cada um dos produtos. As saídas da máquina são: um compartimento para o produto comprado, outro para as moedas do troco, um *display* decimal (7 segmentos) para indicar a quantidade de dinheiro inserida, quatro leds indicando a eventual falta de um ou mais produtos, e um led indicando a ausência de troco na máquina.

Existem ainda entradas para a recarga de um determinado produto e um botão de *reset* da máquina, disponíveis apenas para o encarregado de sua manutenção.

O comprador coloca moedas de vários valores na máquina até completar ou ultrapassar o valor do produto que deseja comprar. Em seguida, pressiona o botão de compra correspondente a este produto. A máquina despacha este produto e o eventual troco para o consumidor.

O controlador desta máquina recebe sinais digitais provenientes de sensores que indicam a entrada de um determinado tipo de moeda e a requisição de um determinado produto. O controlador fornece como saída sinais digitais que comandam atuadores para o fornecimento de produtos ou de troco, controle dos leds e do *display* de 7 segmentos.

A máquina não deve aceitar mais moedas caso o montante de dinheiro já entrado seja maior ou igual ao valor do produto mais caro. Neste caso, qualquer moeda nova inserida será ignorada e devolvida ao consumidor.

7.2.3 Descrição comportamental em SDL

O controlador digital da máquina foi completamente descrito em SDL, seguindo as restrições de síntese descritas no capítulo 5. A descrição SDL foi convertida em uma descrição VHDL sintetizável através do conversor **Stoht**. O modelo VHDL foi simulado, validado, e em seguida sintetizado na tecnologia de lógica programável **EPLD** da Altera.

O circuito sintetizado foi novamente simulado nesta tecnologia, levando em conta informações de temporização, e novamente validado. O circuito sintetizado completo necessitou de apenas um componente

da família **FLEX** deste fabricante para sua implementação.

A Fig. 7.26 mostra o sistema SDL correspondente ao controlador da máquina.

Os canais de entrada (**Ccoins** e **Crequest**) carregam sinais provenientes dos sensores da máquina. Os canais de saída (**CspitPurchased**, **CamountDisplay**, **Creject**, **CemptyDisplay**, **CexactDisplay** e **CspitChange**) carregam os sinais que controlam os acionadores de saída.

Os blocos estruturam o sistema em três partes: controle da entrada de moedas (**CoinInterface**), controle dos pedidos de compra (**DecodeRequests**) e controle do troco (**ChangeInterface**). Canais internos interconectam estes blocos.

No escopo do sistema também são definidos dois tipos de dados utilizados em processos: tipo **int**, que armazena quantidade de dinheiro, e tipo **items**, que armazena quantidade de produtos.

A Fig. 7.27 mostra o bloco **CoinInterface**.

O bloco da Fig. 7.27 contém um processo (**CoinHandle**) e rotas conectadas aos canais definidos no escopo do sistema. Neste bloco ainda são definidos os valores de cada tipo de moeda, através da declaração de **Synonym**.

O processo **CoinHandle** é mostrado na Fig. 7.28.

O processo **CoinHandle** (Fig. 7.28) está normalmente à espera de moedas (estado **wait_for_coins**); O valor correspondente ao tipo de moeda inserida é enviado como parâmetro do sinal **add**. Se o consumidor inseriu dinheiro suficiente para comprar o produto mais caro, este processo transita para o estado **rej_coins**, rejeitando qualquer nova moeda inserida através do envio do sinal **reject_coin** para um atuador da máquina.

Na Fig. 7.29 é mostrado o bloco **DecodeRequests**, responsável pelo controle dos pedidos de compra do consumidor.

O bloco **DecodeRequests** (Fig. 7.29) contém um processo para cada tipo de produto, um processo de controle do dinheiro atual da máquina, rotas conectadas a canais do escopo do sistema, rotas internas e sinônimos representando os valores de cada tipo de produto.

Na Fig. 7.30 é mostrado o processo **AmountHandler**, responsável pelo controle da quantidade de dinheiro entrada pelo consumidor.

O processo **AmountHandler** (Fig. 7.30) recebe requisições de adição (sinal **add** proveniente de **CoinHandle**, Fig. 7.28) quando uma moeda é inserida, ou de subtração (sinal **sub** proveniente dos processos de controle dos 4 tipos de produtos) quando um produto é comprado.

Quando a quantidade de dinheiro inserida equivale ou ultrapassa o item mais caro (**PMAX**), este processo envia o sinal **rej_further_coins** para o processo **CoinHandle**, evitando que demais moedas sejam inseridas.

Quando a compra de um produto é requisitada (sinal **sub**), o processo **amountHandler** envia um sinal para o controlador de troco, carregando um parâmetro correspondente à diferença entre o valor inserido e o proço do produto.

Em cada mudança do valor da quantidade de dinheiro inserida (variável **current**), o *display* de saída é atualizado através do envio do sinal **amount_entered**. A variável **current** é revelada, permitindo que os outros processos do bloco **DecodeRequests** tenham acesso ao valor inserido pelo consumidor.

A Fig. 7.31 mostra um dos quatro processos de controle de produtos, **PretzelHandler**. Os outros três processos são idênticos, a menos de valores de constantes.

O processo da Fig. 7.31 recebe requisições de compra (sinal **pur_pretzel**) e testa se a quantidade de dinheiro atual é suficiente para comprar tal produto, através da leitura da variável revelada **current**. Se o dinheiro é suficiente, envia o sinal **sub** com o valor correspondente do produto para o processo **AmountHandler** (Fig. 7.30) e o sinal **spit_pretzel** para o atuador de saída de produtos. Se este item terminar, o processo transita para o estado **empty** e espera pela recarga deste item.

Na Fig. 7.32 é mostrado o bloco **ChangeInterface** de controle da emissão de troco.

O bloco **ChangeInterface** contém apenas um processo, **ChangeHandler**. Os sinônimos **NCHANGE** e **MINCHANGE** representam, respectivamente, o número de moedas para troco inicialmente pre-

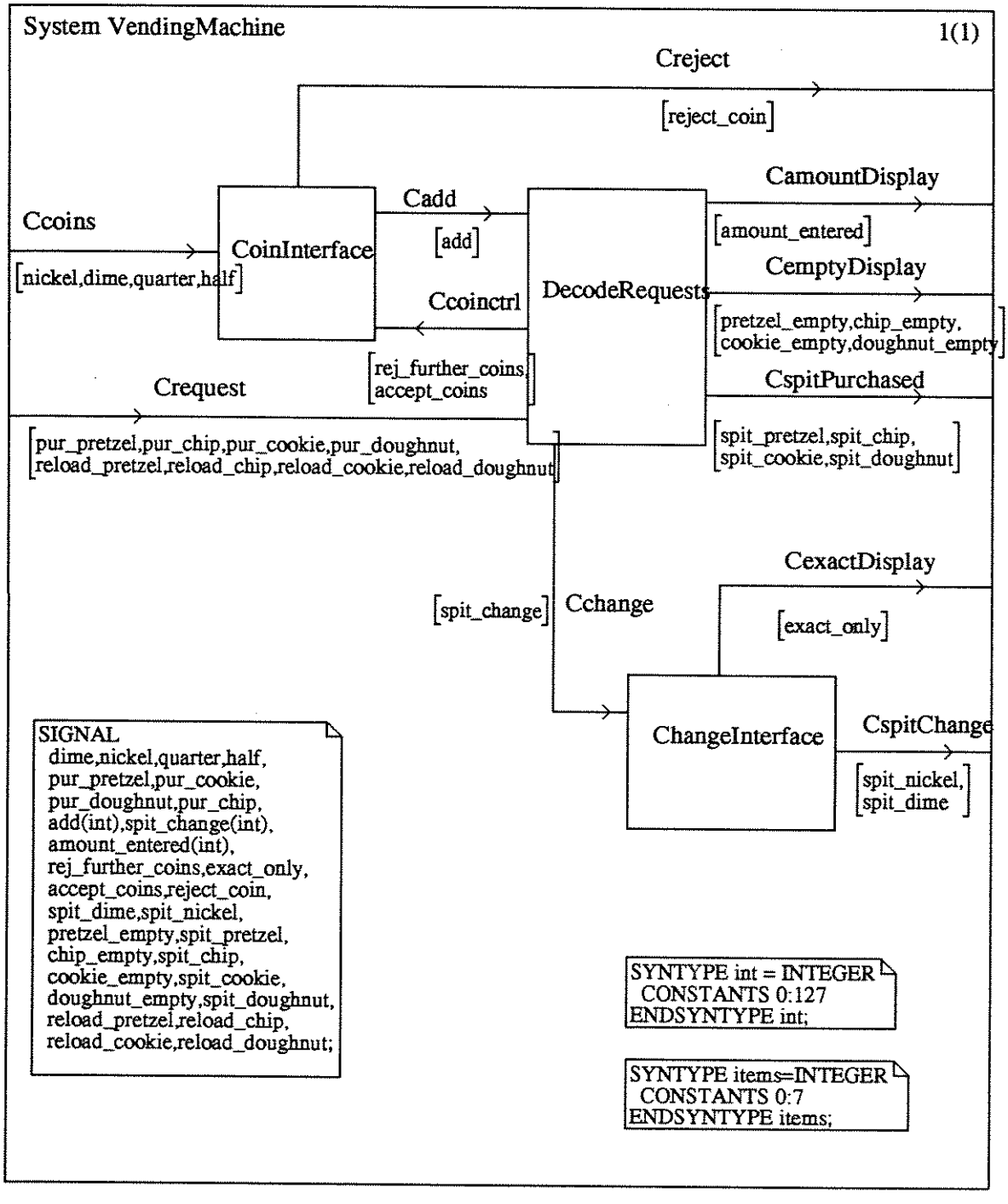


Figura 7.26 Sistema SDL do controlador da máquina de venda.

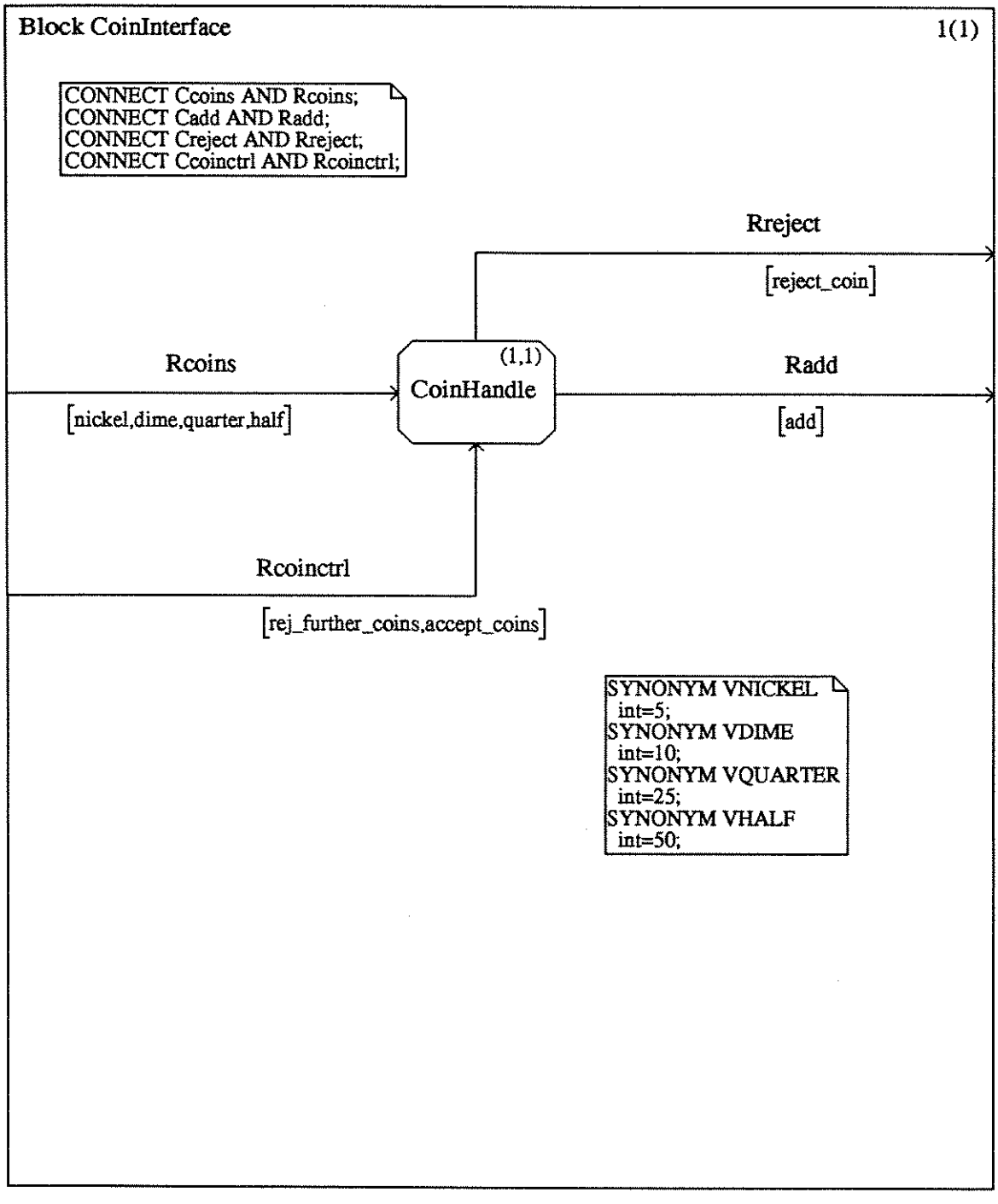


Figura 7.27 Bloco SDL de controle da entrada de moedas.

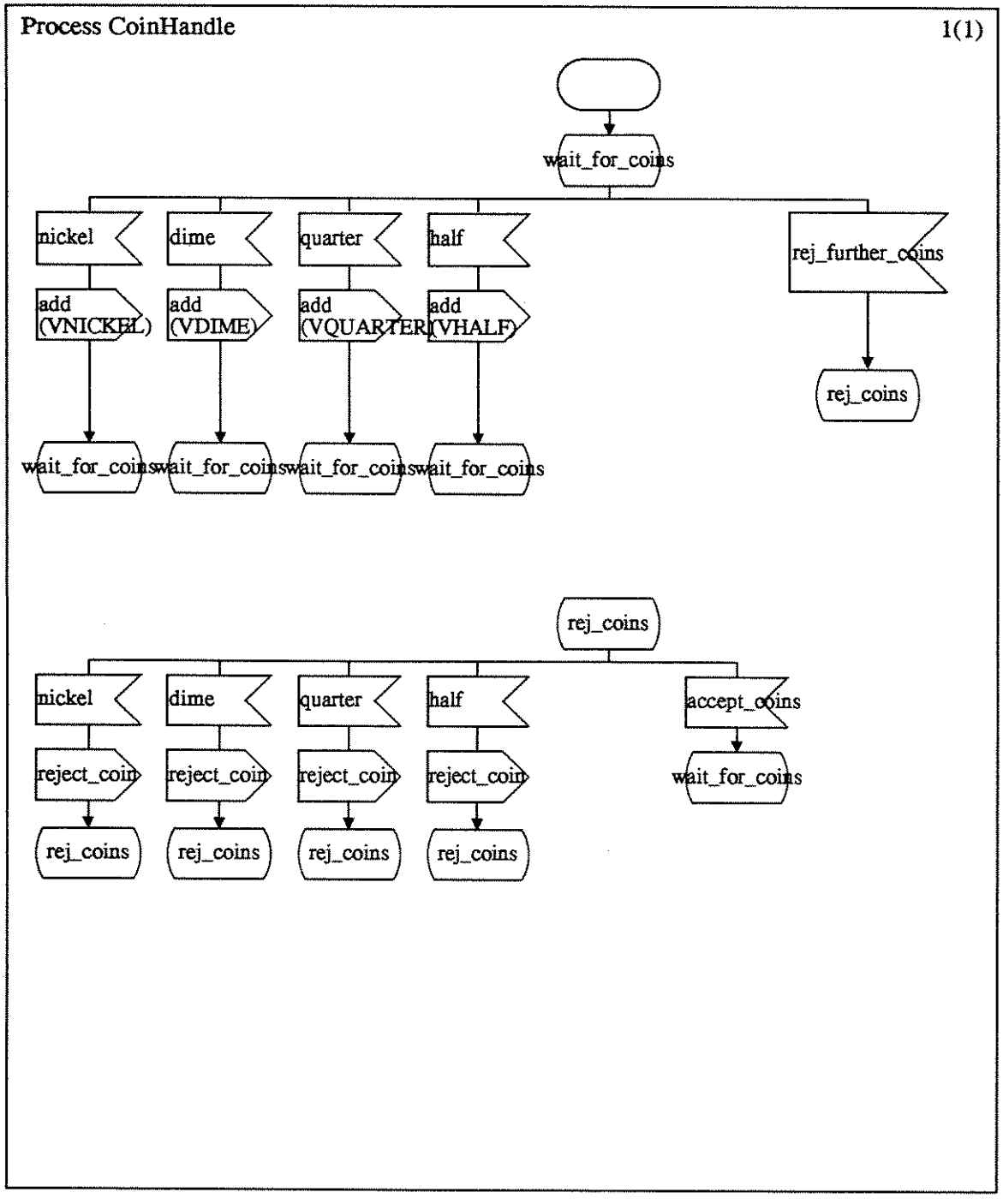


Figura 7.28 Processo SDL de controle da entrada de moedas.

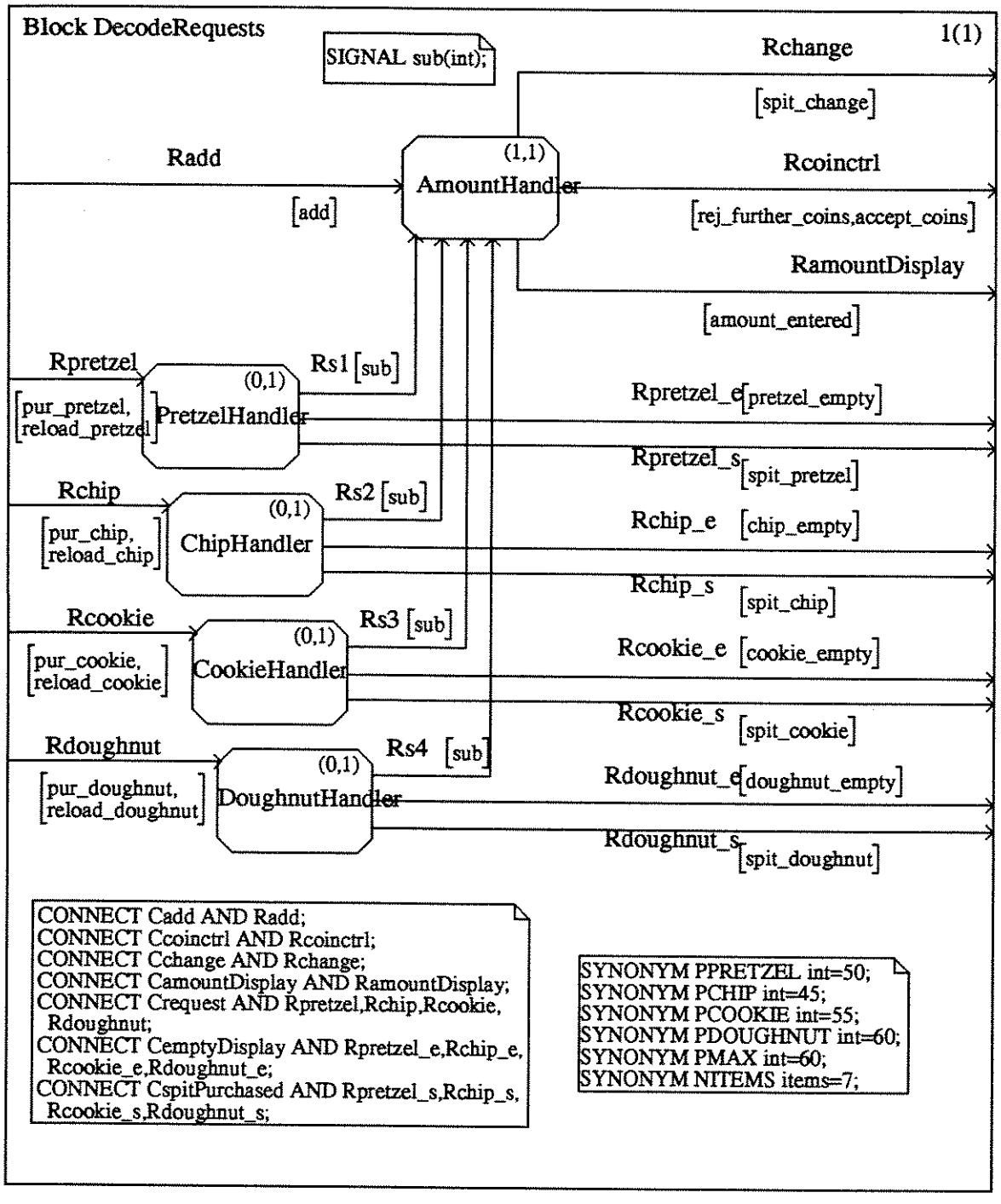


Figura 7.29 Bloco SDL de controle de pedidos de compra.

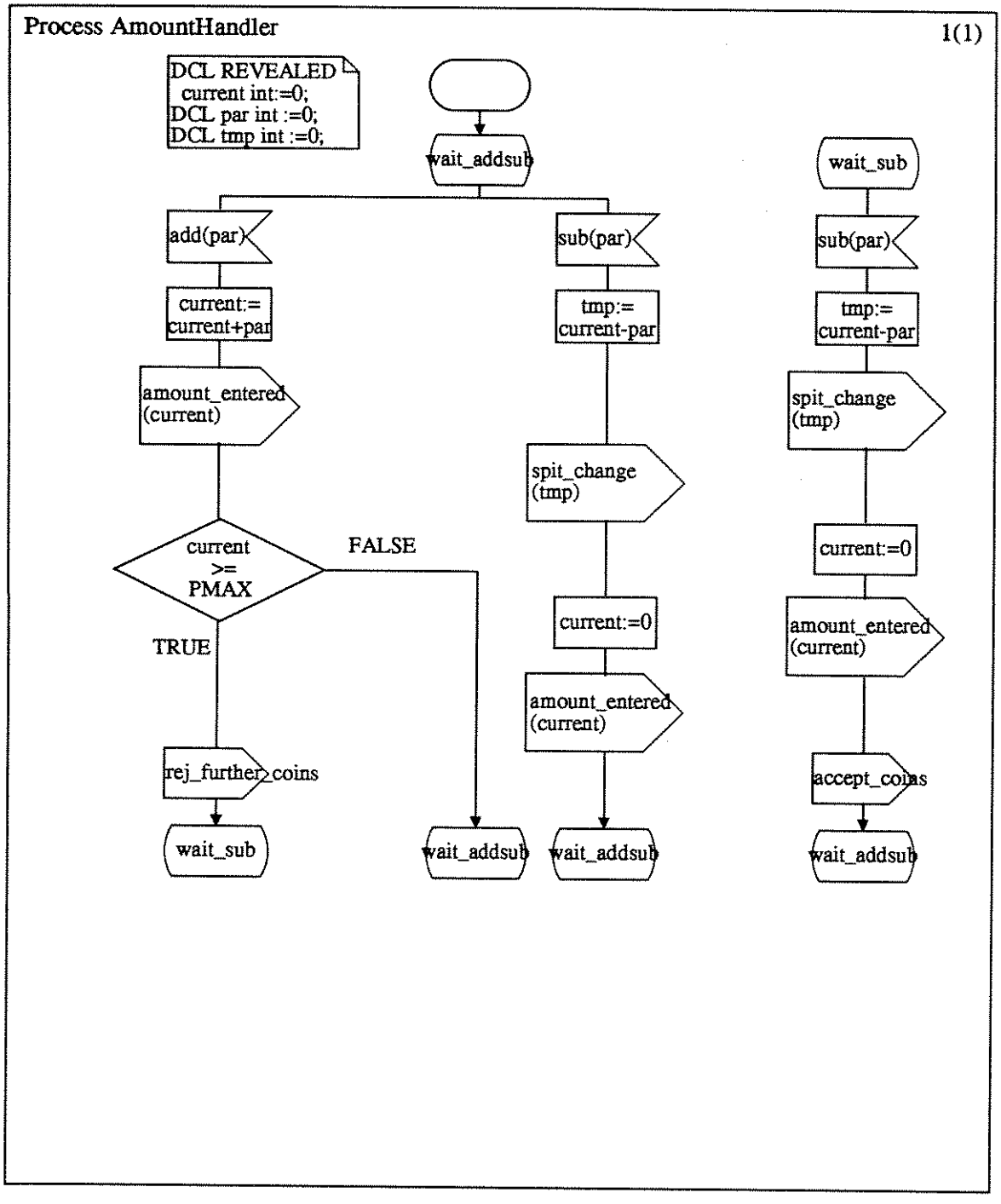


Figura 7.30 Processo SDL de controle da quantidade de dinheiro na máquina.

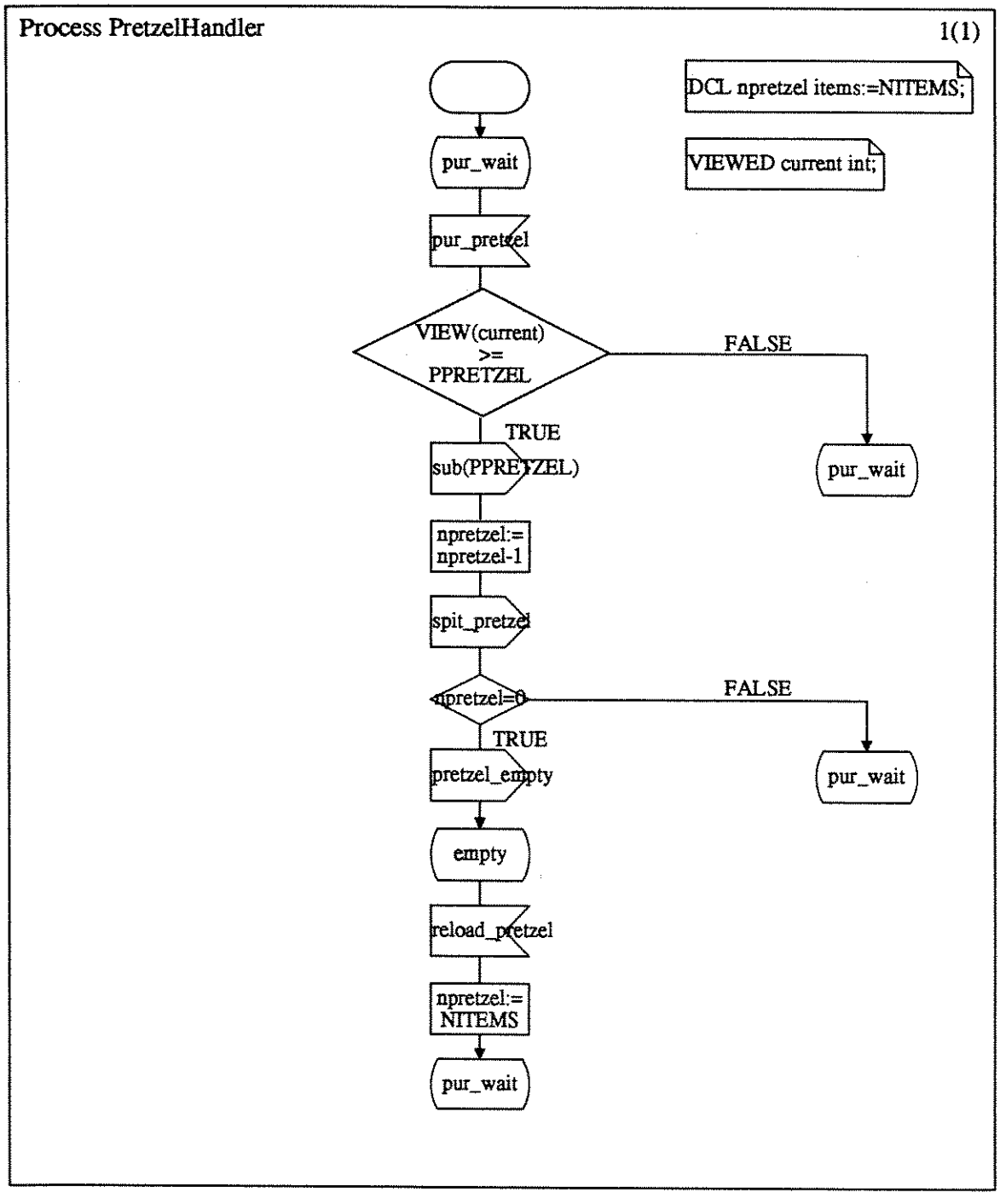


Figura 7.31 Processo SDL de controle do item **pretzel**.

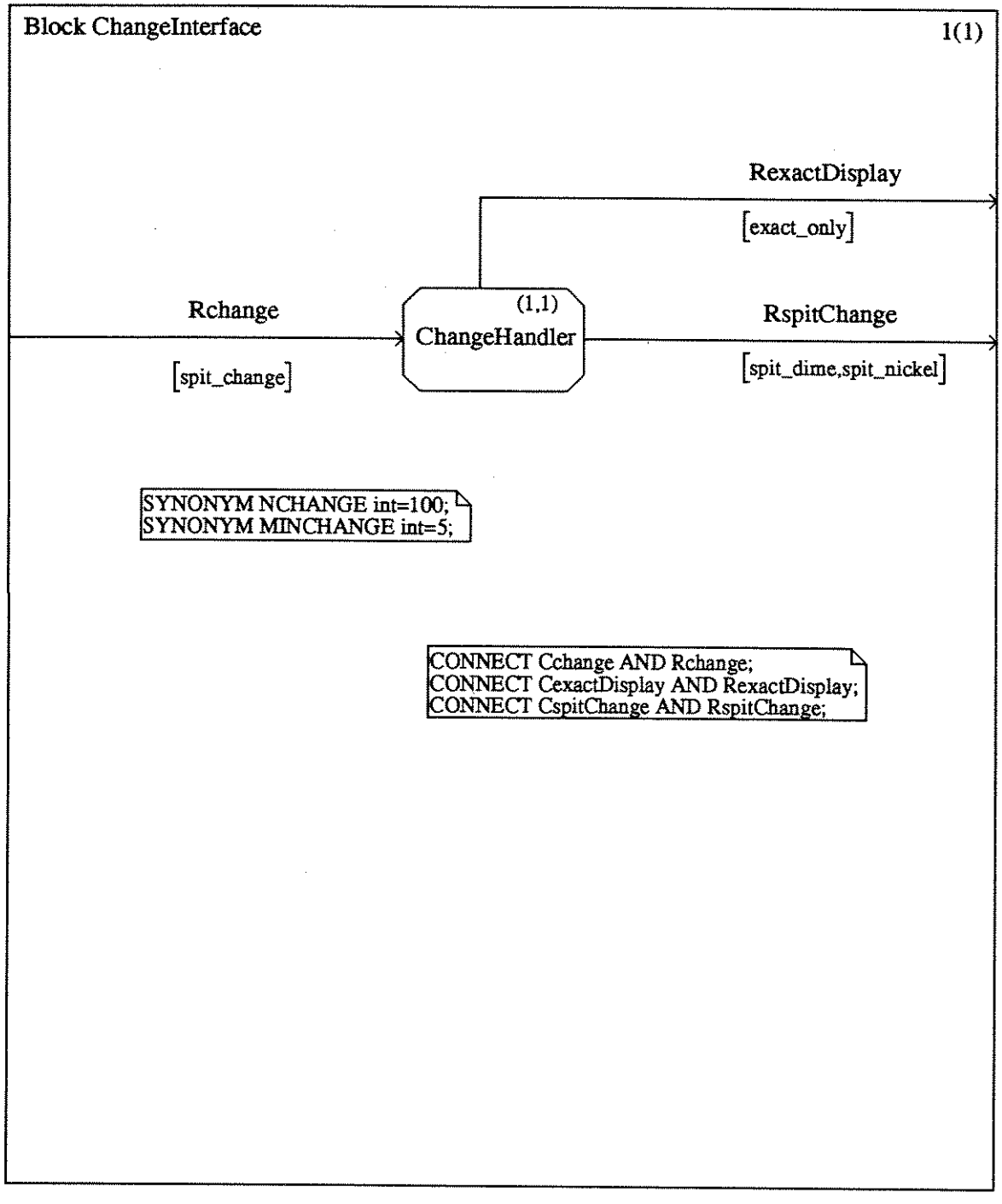


Figura 7.32 Bloco SDL de controle da emissão de troco.

sente na máquina e o número mínimo de moedas para troco que a máquina deve possuir. Se este valor mínimo for atingido, cabe à máquina avisar ao consumidor que deve ser inserido o valor exato do produto apenas.

Na Fig. 7.33 é mostrado o processo **ChangeHandler**, que realiza a emissão de troco para o consumidor.

O processo **ChangeHandler** espera por uma requisição de envio de troco no estado **wait**. Quando uma requisição chega, o processo entra em *loop* no estado **spitting** até que o troco completo seja fornecido, voltando então ao estado **wait**. Uma transição ocorre no estado **spitting** quando um dos sinais contínuos que testam o valor da variável **change** é verdadeiro.

Como a prioridade do sinal contínuo **change>10** é a mais alta, a máquina iniciará o fornecimento de troco por moedas de 10 centavos. Eventualmente a máquina envia uma moeda de 5 centavos, e quando não há mais troco a ser fornecido (sinal contínuo **change=0**), o processo volta a esperar por demais requisições no estado **wait**.

Como todas as transições de estado são sincronizadas pelo relógio global inserido pelo algoritmo, cada moeda de troco é fornecida em um ciclo deste relógio.

7.2.4 Implementação

A descrição em SDL foi mapeada em VHDL seguindo o algoritmo de mapeamento proposto e implementado no programa **Stoht**, simulada e sintetizada na tecnologia de lógica programável Altera. A Fig. 7.34 mostra o resultado da síntese em seu nível hierárquico mais alto.

Nota-se que a partição em blocos do sistema SDL original (Fig. 7.26) é mantida no esquemático do circuito sintetizado (Fig. 7.34). Os sinais trocados com o ambiente são portas de entrada e saída do sistema sintetizado.

A Fig. 7.35 mostra o resultado da síntese para o bloco **DecodeRequests** (Fig. 7.29).

Nota-se que os 5 processos encotrados originalmente no bloco **DecodeRequests** são mapeados em 10 componentes ao final da síntese, já que cada processo SDL gera dois componentes VHDL (comportamento e protocolo).

A Fig. 7.36 mostra o resultado da síntese do protocolo tipo 1 alocado para o processo **PretzelHandler**. O protocolo mais simples aloca apenas um flip-flop e uma porta ou-exclusivo para cada sinal que o processo recebe.

A Fig. 7.37 mostra a síntese do comportamento do processo **PretzelHandler**.

A Fig. 7.38 mostra o resultado de uma das simulações feitas com a descrição VHDL da máquina de vendas. Nesta simulação são exercitados diversos pedidos de compra (sinais de prefixo **pur_**) e combinações de entrada de moedas **quarter** e **half**. As saídas de produtos são observadas (sinais de prefixo **spit_**) bem como os sinais de envio de troco (**spit_dime, spit_nickel**) e o valor decimal do *display* de saída.

7.3 Conclusões

Os exemplos do detetor de sincronismo e da máquina de venda foram apresentados para ilustrar a capacidade de modelamento da metodologia e para a sua validação. Através destes exemplos, o algoritmo de mapeamento proposto neste trabalho foi ilustrado. Os projetos seguiram a metodologia de projeto proposta: foram descritos em **SDL**, mapeados em **VHDL**, simulados e sintetizados em tecnologia de lógica programável.

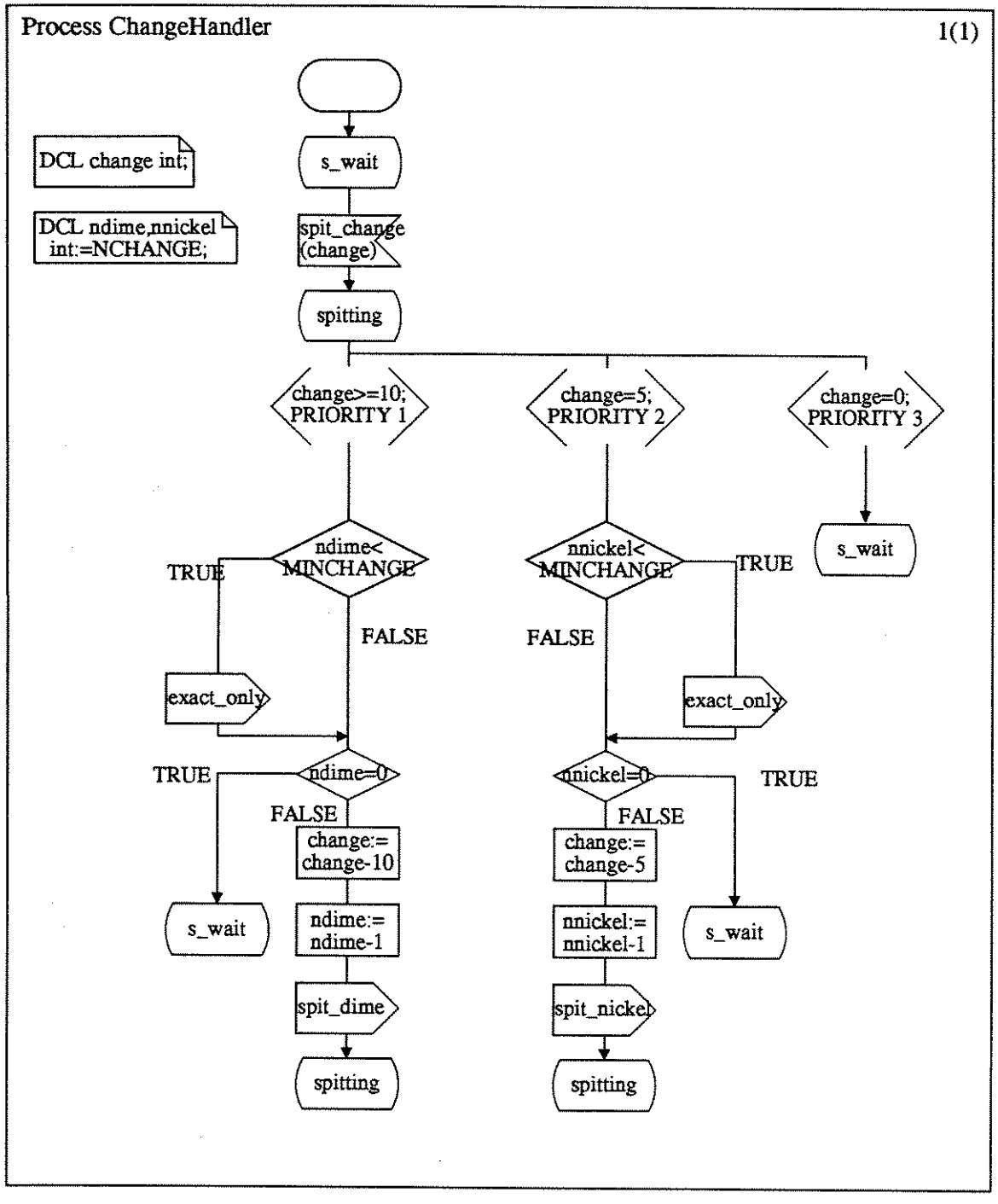


Figura 7.33 Processo SDL que realiza a emissão de troco.

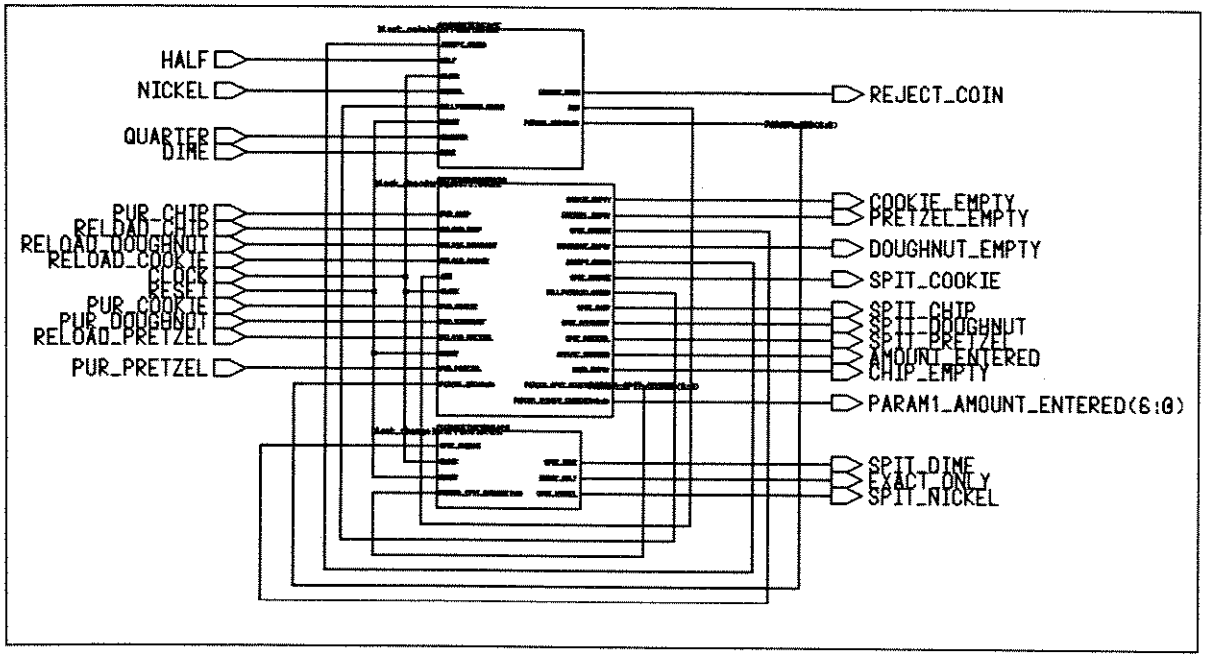


Figura 7.34 Circuito sintetizado correspondente ao sistema **VendingMachine**.

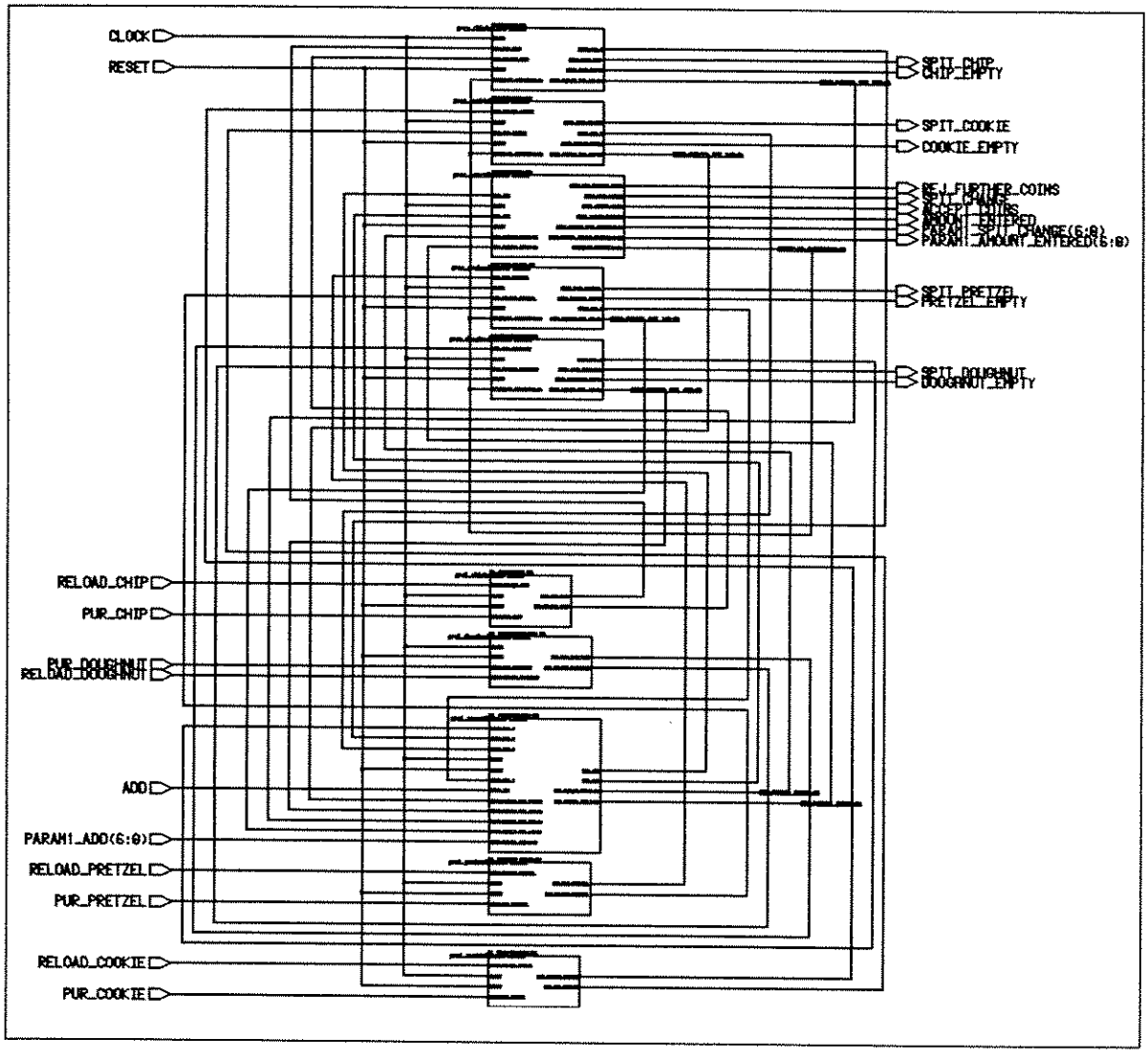


Figura 7.35 Circuito sintetizado correspondente ao bloco **DecodeRequests**.

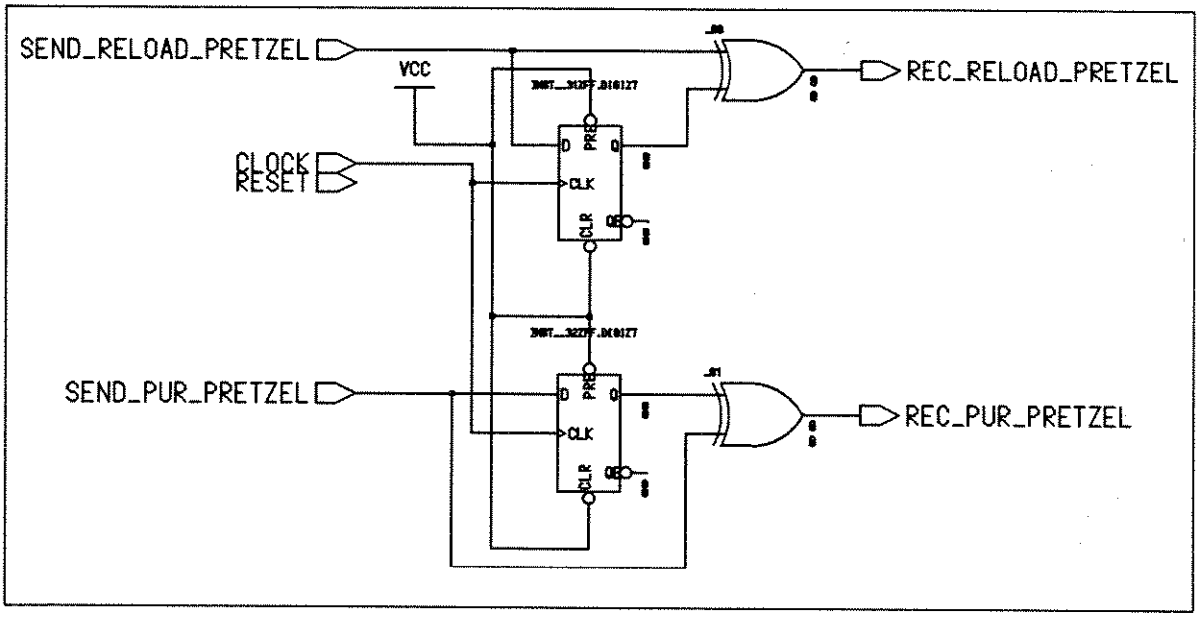


Figura 7.36 Circuito sintetizado correspondente ao protocolo do processo **PretzelHandler**.

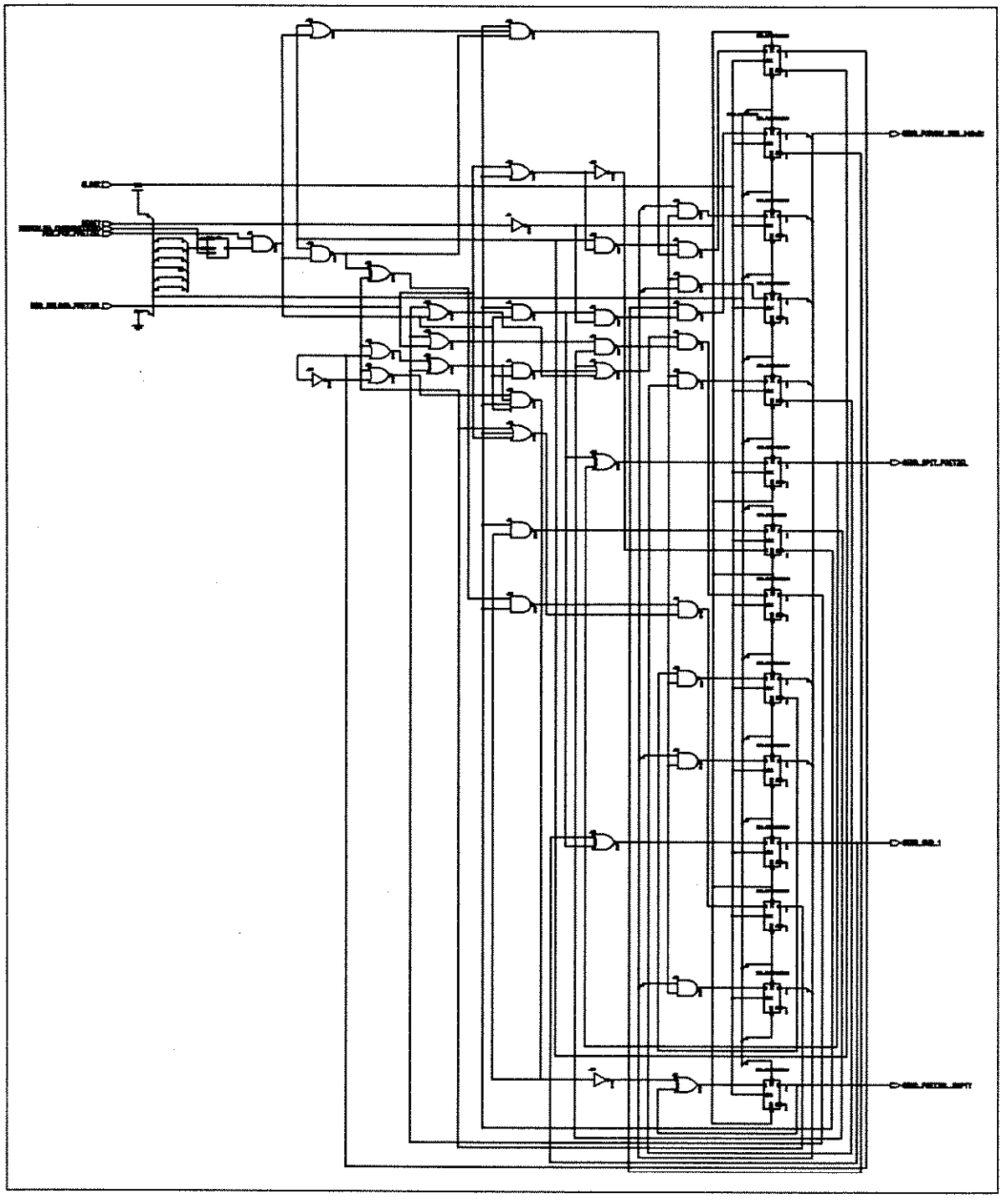


Figura 7.37 Circuito sintetizado correspondente ao comportamento do processo **PretzelHandler**.

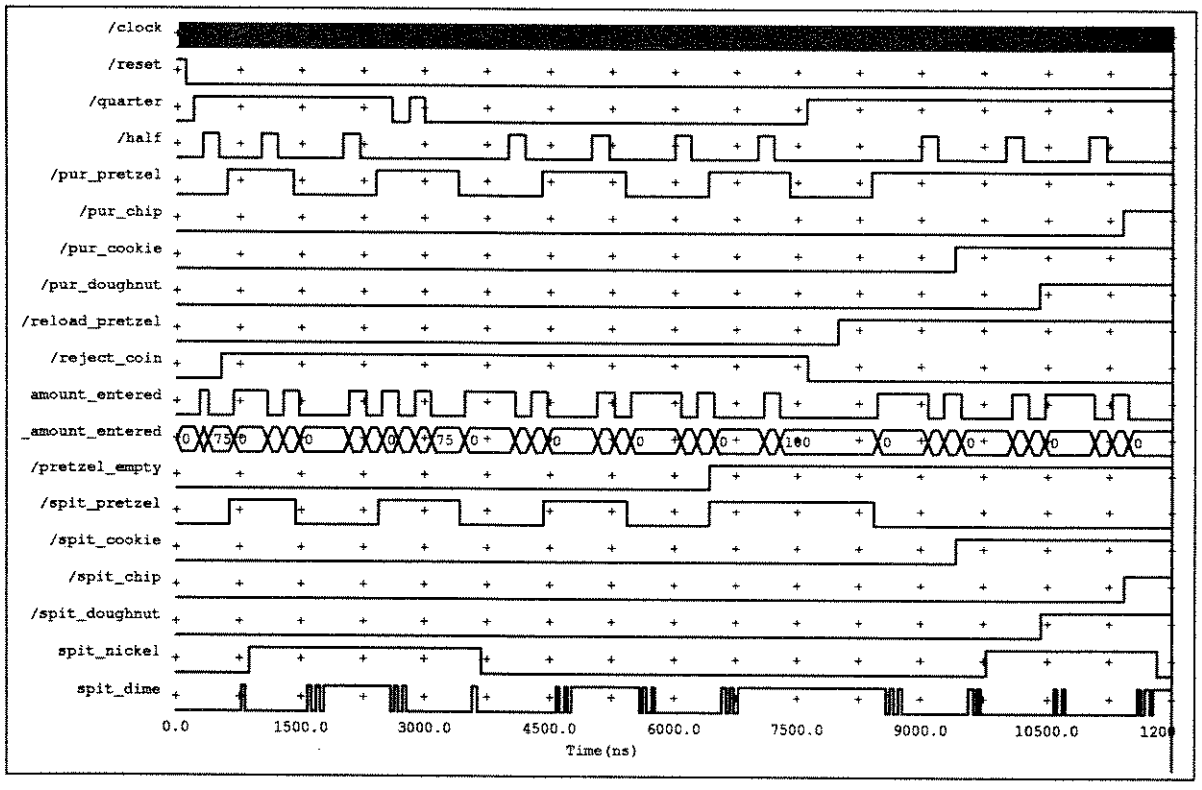


Figura 7.38 Simulação da descrição VHDL gerada para o sistema **VendingMachine**.

Capítulo 8

Conclusões

A linguagem de especificação e descrição de sistemas **SDL**, aplicada usualmente para o desenvolvimento de software, mostrou-se também adequada para o modelamento de circuitos digitais. A utilização desta linguagem como entrada (interface homem-máquina) para uma metodologia de projeto de sistemas permite o uso de descrições gráficas comportamentais independentes de tecnologia, objetivando atender a demanda por sistemas cada vez mais complexos e com ciclos de desenvolvimento mais curtos.

A disponibilidade de ferramentas de síntese de alto nível baseadas na linguagem **VHDL**, adotada como padrão "de fato" por pacotes de automação de projeto, torna viável a adoção da geração de código para a implementação de hardware.

A delimitação de um sub-conjunto da linguagem **SDL** para projeto de circuitos, necessária para permitir uma implementação física eficiente, e a geração de código **VHDL** sintetizável tornam automático o mapeamento da descrição comportamental **SDL** em uma descrição estrutural a nível de portas lógicas através do uso de ferramentas **EDA**. O mapeador **Stoht**, desenvolvido durante este trabalho, comprovou a viabilidade do algoritmo de mapeamento para o projeto de circuitos a partir de descrições **SDL**.

Os exemplos de aplicação *Detector de Sincronismo* e *Máquina de Venda* foram desenvolvidos em poucas horas, em um intervalo de tempo bastante inferior ao esperado para um projeto realizado manualmente. Constatou-se que a abstração oferecida pela entrada de dados via **SDL** gráfico permite este aumento de produtividade, ao aproximar o projeto da sua especificação.

8.1 Trabalhos Futuros

Tópicos que não foram abordados neste trabalho mas que constituem sua extensão natural:

- Integração com projeto de software (co-design). Particionamento do sistema em módulos de software e módulos de hardware.
- Geração automática de código para software e alocação de arquiteturas que o suportem (microprocessadores, memória, etc).
- Mapeamento de primitivas de comunicação via troca de sinais do SDL em protocolos de comunicação entre hardware e software.
- Extensão do sub-conjunto SDL aceito pelo algoritmo, incluindo *Procedures* e *Macros*, entre outros.
- Implementação de síntese condicional: blocos ou processos existentes não são sintetizados. Definição de protocolos de comunicação via troca de sinais com estas estruturas. Incorporação de código VHDL na descrição SDL.

Capítulo 9

Glossário

- **CASE** - *Computer Aided Software Engineering.*
  Ferramentas de software que objetivam a automatização do desenvolvimento de software.
- **Co-design**.
  Projeto integrado de hardware e software.
- **CCITT** - *International Telegraph and Telephone Consultative Committee.*
- **EDA** - *Electronic Design Automation.*
  Ferramentas de software que têm por objetivo automatizar o projeto de circuitos em todas as suas etapas.
- **EPLD** - *Erasable Programmable Logic Device.*
  Tecnologia de lógica programável desenvolvida pelo fabricante de componentes **Altera**.
- **FLEX** Família de componentes de lógica programável da **Altera**.
- **GENLIB** - *Generic Library.*
  Biblioteca *Mentor Graphics* de componentes digitais básicos (portas lógicas, registros) utilizada para representar os componentes resultantes da síntese lógica.
- **HDL** - *Hardware Description Language.*
  Classificação dada às linguagens de programação aplicadas diretamente no projeto de hardware.
- **HLS** - *High Level Synthesis.*
  Síntese de circuitos partindo de descrições comportamentais (algorítmicas).
- **IEEE** - *Institute of Electrical and Electronic Engineers.*
- **RTL** - *Register Transfer Logic.*
  Representação estrutural obtida após a síntese de VHDL comportamental.
- **SDL** - *Specification and Description Language.*
  Linguagem desenvolvida pelo *CCITT* voltada para a especificação e descrição de sistemas de comunicação digital (hardware e software).
- **SDL-GR** - *Specification and Description Language - Graphical Representation.*
  Representação gráfica do SDL.
- **SDL-PR** - *Specification and Description Language - Program Representation.*
  Representação textual do SDL.

- **SDT** - *Specification and Description Tool.*
  Ferramenta computacional utilizada para descrição e simulação de sistemas SDL.

- **TDM** - *Time Division Multiplex.*
  Método de compartilhamento temporal do meio de transmissão digital através da divisão em janelas de tempo.

- **TOP-DOWN**
  Metodologia de projeto de circuitos que parte da especificação do comportamento global do sistema e desce até o nível de implementação física.

- **VHDL** - *Very High Speed Integrated Circuit Hardware Description Language.*
  Linguagem proposta pelo programa VHSIC e adotada como padrão pelo *IEEE*.

- **VHSIC** - *Very High Speed Integrated Circuit.*
  Projeto de pesquisa do Departamento de Defesa Norte-Americano na área de circuitos digitais de alta capacidade.

Bibliografia

[1] TeleLOGIC Malmö AB. SDT User's Guide, 1993.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compiler: Principles, Techniques and Tools.* Addison-Wesley, 1988.

[3] Michael Atlevi. SDT The SDL Design Tool. In *Formal Description Techniques - Proc. of the First Int. Conf.*, pages 55–59, 1989.

[4] I.S. Bonatti and R.J.O. Figueiredo. An Algorithm for the Translation of SDL to Synthesisable VHDL. *The Current Issues in Electronic Modeling - Issue 3*, June (to be published) 1995.

[5] R. Camposano. High-Level Synthesis. In *Proceedings of the II Brazilian Microelectronics School*, 1992.

[6] S. Carlson and E. Girczyc. Understanding Synthesis Begins with Knowing the Terminology, 1992.

[7] CCITT. *Recommendation Z.100: Specification and Description Language SDL*, volume X.1-X.5. CCITT, 1988.

[8] CCITT. *Recommendation Z.100: Specification and Description Language SDL*, volume X-R25-X-R32. CCITT, 1992.

[9] Mentor Graphics Corporation. AutoLogic VHDL Synthesis Guide, V8.2. Mentor Graphics Online Documentation, 1993.

[10] Mentor Graphics Corporation. Quicksim User Guide, V8.2. Mentor Graphics Online Documentation, 1993.

[11] Mentor Graphics Corporation. System-1076 User Guide, V8.2. Mentor Graphics Online Documentation, 1993.

[12] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[13] Renato J. O. Figueiredo. Stoht - The SDL to Hardware Translator. Documentação distribuída com o programa, 1995.

[14] D. Gajski. *Silicon Compilers.* Addison-Wesley, 1987.

[15] W. Glunz, T. Kruse, T. Rössel, and D. Monjau. Integrating SDL and VHDL for System-Level Hardware Design. In *CHDL 93 - Computer Hardware Description Languages and their Application*, Ottawa, Canada, April 1993.

[16] W. Glunz and G. Venzl. Hardware Design using CASE Tools. In *Proceedings of the IFIP VLSI'91 Conference Edinburgh*, pages 237–246, 1991.

[17] W. Glunz and G. Venzl. Using SDL for Hardware Design. In *Proceedings of the Fifth SDL Forum*, Glasgow, 1991.

[18] T.B. Ismail, M. Abid, and A. Jerraya. COSMOS: A Codesign Approach for Communicating Systems. In *Co-Design, Computer Aided Hardware/Software Engineering*. IEEE Press, 1994.

[19] T.B Ismail and A.A. Jerraya. Synthesis Steps and Design Models for Codesign. In *Computer*, pages 44–52. IEEE Press, 1995.

[20] A.A. Jerraya and K. O'Brien. SOLAR: An Intermediate Format for System-Level Modelling and Synthesis. In *Co-Design, Computer Aided Hardware/Software Engineering*. IEEE Press, 1994.

[21] A.A. Jerraya, K. O'Brien, and T. Ben-Ismail. Linking System Design Tools and Hardware Design Tools. In *Proc. CHDL'93*, pages 345–351, Ottawa, Canada, April 1993.

[22] B. Lutter, W. Glunz, and F.J. Rammig. Using VHDL for Simulation of SDL Specifications . In *Proceedings of the EURO-VHDL*, 1992.

[23] S. Narayan, F. Vahid, and D.D. Gajski. Translating System Specifications to VHDL. In *Proc. Second European Design Automation Conf.*, pages 390–394. IEEE CS Press, 1991.

[24] S. Narayan, F. Vahid, and D.D. Gajski. System Specification with the SpecCharts Language. *IEEE Design and Test of Computers*, pages 6–13, December 1992.

[25] F.A.M. Nascimento and T.S. Weber. Algoritmos e Procedimentos para a Síntese Automática de Alto Nível com VHDL. *Revista Brasileira de Microeletrônica*, 1:21–45, Ano II.

[26] K. O'Brien, T.B. Ismail, and A.A. Jerraya. A Flexible Communication Modelling Paradigm for System-Level Synthesis. In *Intl. Workshop on Hardware-Software Co-Design*, 1993.

[27] The Institute of Electrical and Electronic Engineers. *IEEE Standard VHDL Language Reference Manual*. IEEE, 1987.

[28] D.L. Perry. *VHDL*. McGraw-Hill Inc, 1991.

[29] O. Pulkkinen and K. Kronlöf. Integration of SDL and VHDL for High-Level Digital Design. In *Proceedings of the EURO-VHDL*, 1992.

[30] Bo Bichel Noerbaek Tele Danmark Research. SDL-92 Grammar and Lexical Description. Arquivos de domínio público obtidos na rede Internet.

[31] R. Saracco, J.R. Smith, and R. Reed. *Telecommunication Systems Engineering Using SDL*. Elsevier Science Publishers B.V., 1989.