UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

JONATHAN TOCZEK SOUZA

TÉCNICAS DE CO-DESIGN APLICADAS AO DESENVOLVIMENTO DE UMA INTERFACE USB

VITÓRIA
2006

JONATHAN TOCZEK SOUZA

TÉCNICAS DE CO-DESIGN APLICADAS AO DESENVOLVIMENTO DE UMA INTERFACE USB

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Mestre em Engenharia Elétrica, na área de concentração em Automação.
Orientador: Prof. Dr. Hans Jörg Andreas Schneebeli

VITÓRIA
2006

JONATHAN TOCZEK

TÉCNICAS DE CO-DESIGN APLICADAS AO DESENVOLVIMENTO DE UMA INTERFACE USB

Dissertação submetida ao programa de Pós-Graduação em Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisição parcial para a obtenção do Grau de Mestre em Engenharia Elétrica - Automação.

Aprovada em 18 de Maio de 2006.

COMISSÃO EXAMINADORA

Prof. Dr. Hans Jörg Andreas Schneebeli
Universidade Federal do Espírito Santo
Orientador

Prof. Dr. Rodolfo Jardim de Azevedo
Universidade Estadual de Campinas

Prof. Dr. Getúlio Vargas Loureiro
Universidade Federal do Espírito Santo

Prof. Dr. Alberto Ferreira de Souza
Universidade Federal do Espírito Santo

Dedico esta dissertação a meus pais,
que mesmo distantes se faziam presentes,
para minha esposa por nunca ter perdido a esperança,
sempre me apoiando em todos os momentos de dificuldade
e para minha filha que trouxe a inspiração de um simples sorriso.

Agradecimentos

Agradeço a todos que direta ou indiretamente colaboraram para a realização deste trabalho.

Ao Conselho Nacional de Pesquisa (CNPq) pelo incentivo financeiro.

A minha familia, que soube compreender os momentos de ausência, acreditando e me apoiando em todos os momentos de dificuldade. Saibam que sem este apoio incondicional, jamais conseguiria terminar este trabalho.

Agradeço especialmente a minha esposa, por ter literalmente me suportado durante todo o tempo de dedicação (quando meu único lazer era a leitura de um livro técnico). Por ter acreditado, quando, mesmo eu, já não encontrava forças. Pelo companheirismo, compreensão, carinho e dedicação, mesmo quando a minha presença era fundamental. Dionízia, saiba que o mérito deste trabalho também é seu.

Aos amigos que me apoiaram, em especial aos colegas do LAI, com os quais, durante o período de pesquisa, tive o privilégio de conviver, aprender e compartilhar experiências: André Ferreira, Érico Lima Machado, Fernando Tello Gamarra, Flávio Garcia Pereira, Jaines Oliveira Bragança, Leonardo Simas, Paulo André, Raquel Frizera Vassallo, Rodrigo de Alvarenga Rosa, Wanderley Cardoso Celeste e a todos os demais membros e alunos do programa de pós graduação.

Gostaria ainda de dedicar meus sinceros agradecimentos ao meu orientador, Prof. Dr. Hans Jörg Andreas Schneebeli, que nos momentos em que me desviei soube com sua experiência indicar o melhor caminho e nos momentos mais difíceis, com poucas palavras, demonstrou a confiança em mim depositada, inspirando e incentivando a realização desta obra.

Rendo homenagens ainda a todos os demais professores do programa, pelo compartilhamento, compreensão e dedicação aos alunos.

“Aprender é a única coisa de que a mente
nunca se cansa, nunca tem medo
e nunca se arrepende.”
Leonardo da Vinci

Resumo

Esta dissertação aborda a metodologia de desenvolvimento conhecida como *hardware/software co-design*, motivada pela complexidade emergente do desenvolvimento de sistemas digitais embarcados e os recentes progressos da tecnologia SoC (*System-on-Chip*).

Nesse contexto, é proposta uma metodologia capaz de atuar num elevado nível de abstração, permitindo aos projetistas: um melhor gerenciamento da complexidade, uma visualização bem definida do processo de desenvolvimento e um refinamento suave entre os componentes do sistema, de maneira que as decisões de projeto e o particionamento entre os componentes de *hardware* e *software* possam ser realizados de maneira simples e natural.

Com esse objetivo foi utilizado a linguagem de modelagem unificada - *UML* (*Unified Modeling Language*), para especificação do sistema em alto nível, e a linguagem SystemC, para a criação de protótipos executáveis e simulações dos vários níveis de abstração definidos pela proposta.

Para demonstração da metodologia, a implementação de uma interface USB (*Universal Serial Bus*), que possui características *co-design* que justificam sua utilização como um exemplo de teste, será especificada e refinada suavemente.

Dessa maneira, considerando a elevada demanda de produção e o tempo de vida relativamente curto destes modernos sistemas que atualmente podem ser encontrados em quase todos os lugares de nosso cotidiano como: carros, celulares, televisores, microondas entre outros, esta dissertação vem auxiliar os esforços metodológicos, em busca do aumento de produtividade, no desenvolvimento destes complexos sistemas.

Abstract

This dissertation is a study about hardware/software co-design, motivated by the increasing complexity on the development of embedded systems and the emergence of the System-on-Chip (SoC) technology.

Considering this context, the work presented here proposes a methodology that provides a high level of abstraction, allowing the designers: a better management of the complexity, a well defined visualization of the development process and a smooth refinement. Thus the project decisions, hardware/software partition and trade-off can be simplified in an easy and natural way.

With this objective, it was used the unified modeling language - UML, for the high level system specification, and the SystemC language, for the creation of executable prototypes and simulation of the different models defined by the proposal.

In order to demonstrate the methodology, the implementation of an USB (Universal Serial Bus) interface will be specified and refined smoothly. The USB has co-design characteristics, which justifies its use as a test case.

In this way, considering the large production demand and the short life time of these modern systems, currently found in almost all places of our daily life, like: cars, cellular phones, televisions, microwaves and others; this dissertation contributes for methodological efforts toward a rising on the productivity upon these complex systems development.

Sumário

Lista de Figuras **vii**

Lista de Tabelas **x**

1 Introdução **1**

1.1 A Metodologia *Co-design* 2

1.2 Elevação dos Níveis de Abstração 3

1.3 Motivação 4

1.4 Definição do Problema 5

1.5 A Metodologia Proposta 6

1.6 Contribuição do Trabalho 7

1.7 Organização do Trabalho 8

1.8 Considerações Finais 9

2 A metodologia co-design **10**

2.1 *Hardware* e *software co-design* 10

2.2 Objetivos do *co-design* 11

2.3 Etapas da metodologia *co-design* 12

2.3.1 Análise de requisitos e restrições 13

2.3.2 Especificação do Sistema 13

2.3.3 Particionamento 14

2.3.4 Co-Simulação 14

2.3.5 Co-Síntese 15

2.4 Domínios de Representações 15

2.5 O nível de sistema 17

2.6 Abordagens de projeto em nível de sistema 17

2.7 Linguagens no nível de sistema 18

2.8 O SystemC 19

2.9 Modelos de abstração do nível de sistema 21

2.10 Considerações Finais 22

3 Processo de Desenvolvimento 23

3.1 Abordagem de desenvolvimento 23

3.1.1 Análise e Levantamento de Requisitos 25

3.1.2 Prototipação do sistema 26

3.1.3 Validação e verificação do sistema 27

3.2 Fluxo de desenvolvimento 28

3.3 Processo de desenvolvimento 30

3.3.1 Modelagem Conceitual 31

3.3.2 Modelagem de Desempenho 32

3.3.3 Modelagem Arquitetural 35

3.3.4 Modelagem Comportamental da Comunicação 36

3.3.5 Modelagem da Implementação 38

3.4 Considerações Finais 40

4 Aplicação da Metodologia Co-design 41

4.1 Introdução ao Problema 41

4.2 Modelagem Conceitual Funcional 43

4.2.1 Análise de Requisitos 44

4.2.2 Prototipação 46

4.2.3 Validação e Verificação 47
4.3 Modelagem Conceitual de Comunicação 48
4.3.1 Análise e Levantamento de Requisitos 49
4.3.2 Prototipação 52
4.3.3 Validação e Verificação 54
4.4 Modelagem de Desempenho 56
4.4.1 Análise Funcional 56
4.4.2 Prototipação 58
4.4.3 Validação e Verificação 60
4.5 Modelagem Arquitetural 63
4.5.1 Prototipação 63
4.5.2 Validação e Verificação 65
4.6 Modelagem Comportamental da Comunicação 66
4.6.1 Análise e Levantamento de Requisitos 67
4.6.2 Prototipação 68
4.6.3 Validação e Verificação 69
4.7 Modelagem de Implementação 71
4.7.1 Análise e Levantamento de Requisitos 71
4.7.2 Prototipação 73
4.7.3 Validação e Verificação 74
4.8 Considerações Finais 75

5 Conclusão **77**

Referências **80**

Apêndice A – Fundamentos do SystemC **84**

A.1 Módulos 85

A.2 *Interfaces*, Portas e Canais 85

A.3 Processos 86

A.4 Eventos 87

A.5 Mecanismo de Simulação 88

Glossário **89**

Lista de Figuras

1 Impacto do atraso do produto no tempo de mercado 2

2 Níveis Clássicos de Abstrações *Hardware/Software* 3

3 Capacidade de integração x produtividade. fonte: (ROWEN, 2002) 4

4 Ambiente unificado de projeto *hardware/software* 6

5 Fluxo Clássico x Fluxo Concorrente 10

6 Etapas da metodologia *co-design* 12

7 A Etapa de Particionamento 14

8 Co-simulação Homogênea x Heterogênea 15

9 A Etapa de Co-síntese 15

10 Gajski´s *Y-chart*. fonte: (GERSTLAUER, 2002) 16

11 Arquitetura do SystemC. fonte: (OSCI, 2002b) 20

12 SLD Tradicional x SLD usando systemC. fonte: (OSCI, 2002b) 21

13 Modelos de Abstração TLM. fonte: (CAI; GAJSKI, 2003b) 22

14 Representação pirâmidal da abordagem de prototipação 24

15 Visualização da base da pirâmide de prototipação 24

16 Exemplo de Modelagem Estrutural SoC. fonte: (UML FOR SOC FORUM, 2004) 26

17 Notação gráfica utilizada para modelagem estrutural do projeto 26

18 Visão geral dos *Testbenches* 27

19 Visão do fluxo do projeto no *Y-chart* 28

20 Ciclo de síntese: Sistema, RTL e Lógica. fonte: (GERSTLAUER, 2002) . . . 29

21 Ciclo de Prototipação 30

22 Modelos definidos no processo de desenvolvimento 31

23 Mapeamento do Modelo Conceitual para o Modelo Arquitetural 36

24 Exemplo do Modelo de Comunicação 37

25 Exemplo do Modelo de Implementação 38

26 Visão simplificada do comportamento USB 41

27 Configuração Típica da Arquitetura USB. fonte: (USB WORK GROUP, 2000) 42

28 Modelo Conceitual 44

29 Diagrama de Estados - Modelo conceitual 45

30 Diagrama de Classes - Modelo conceitual 45

31 Validação do Modelo Conceitual 48

32 Modelo Conceitual da Comunicação (1° Refinamento) 48

33 Modelo Conceitual da Comunicação (2° Refinamento) 49

34 Modelo Lógico USB (Dusb) 50

35 *StateChart* - Comportamento do Equipamento 51

36 *StateChart* - Comportamento do *endpoint0* 51

37 Classe Packet 52

38 Modelo de Testes - 1° Refinamento 54

39 Modelo de Testes - 2° Refinamento 54

40 Validação do Modelo Conceitual - 1° Refinamento 55

41 Microsoft Visual C++ 6.0 *Profile* 57

42 Modelo de Desempenho 58

43 Exemplo do arquivo de saída gerado pelo *testbench* 62

44 Modelo Arquitetural 63

45 Protótipo do Elemento de SW (PeSw) 64

46 Exemplo de falha detectada durante a simulação 65

47 Simulação Arquitetural 65

48 Modelo de Comunicação 67

49 Modelo físico da ligação entre o UTM e o projeto USB 67

50 Refinamento do Canal usando adaptadores 68

51 Gráfico de tempo da Simulação Comportamental 70

52 Modelo de Implementação 71

53 Integração dos adaptadores 72

54 Modelo de Comunicação x Modelo de Implementação 73

55 Modelo de Implementação Completo 74

56 Arquivo de Log - recepção e envio de dados pelo UTMI 75

57 Exemplo de uma ferramenta de comparação de arquivos, utilizada para verificação dos resultados gerados pelos modelos de testes 76

58 Arquitetura do SystemC. fonte: (OSCI, 2002b) 84

Lista de Tabelas

1 Níveis de abstrações x Domínios de Representações 17

2 Requisições Padrão 43

3 Formato dos Pacotes 52

4 Perfil da Especificação Conceitual 57

5 Restrições de tempo do projeto USB 61

1 *Introdução*

O progresso da microeletrônica tem proporcionado o surgimento de componentes digitais cada vez mais sofisticados e complexos, capazes de conter em um único chip, milhões de transistores. Além disso a emergente tecnologia SoC (*System-on-Chip*), que possibilita que todo um sistema seja projetado em um único circuito integrado, modifica completamente a maneira como os sistemas digitais são especificados e implementados, acrescentando diversos novos desafios em todos os estágios do processo de desenvolvimento (EDENFELD et al., 2004).

Um destes desafios é integrar de maneira eficiente as diferentes tecnologias envolvidas nesses modernos sistemas, tipicamente constituídos de elementos de *hardware* e *software*. De fato, isto tem implicado em uma constante necessidade por novas metodologias e abordagens capazes de suprir a demanda do mercado por produtos cada vez mais complexos e com menor tempo de desenvolvimento.

A modificação metodológica necessária, em um primeiro ponto (1) requer um melhor tratamento da complexidade dos sistemas e em um segundo ponto (2) um melhor gerenciamento no mapeamento das funcionalidades em relação aos componentes do sistema, com o objetivo de encontrar o ponto de equilíbrio entre custo e benefício.

O caso (1) envolve a elevação dos níveis de abstração do sistema a ser desenvolvido, enquanto o caso (2) sugere uma metodologia *co-design* que envolve um desenvolvimento combinado entre os componentes *hardware/software* do projeto. Estes pontos são o foco da metodologia definida nesta dissertação e serão a seguir melhor descritos, introduzindo o tema do trabalho.

O restante do capítulo, apresenta a motivação da pesquisa, define o problema, a metodologia utilizada para a realização do trabalho e os objetivos almejados. O capítulo é encerrado descrevendo a estrutura da dissertação.

1.1 A Metodologia *Co-design*

O conjunto de técnicas que permitem em um único fluxo de desenvolvimento combinar componentes de *hardware* e *software* de um projeto é conhecido como metodologia *hardware/software co-design*[1].

Projetos envolvendo componentes de *hardware* e *software* sempre existiram, no entanto, tradicionalmente estes eram desenvolvidos em separado e assim conduzidos até uma etapa posterior onde os componentes deveriam ser integrados. Muitas vezes isso conduzia a um projeto pouco otimizado e sujeito a falhas de integração, gerando retrabalho, consumindo mais tempo e elevando, consequentemente, os custos dos projetos.

Conforme o crescimento das exigências do mercado, juntamente com a demanda por componentes cada vez mais complexos com tempos e custos cada vez mais reduzidos, as técnicas clássicas de desenvolvimento de sistemas digitais embarcados precisaram ser modificadas. Basta comentar que de acordo com alguns estudos, um atraso de 6 meses no projeto implica numa queda de 33% nos lucros (BARROS et al., 2000), conforme mostra o gráfico da figura 1, em uma referência ao conceito de tempo de mercado (*time-to-market*) dos sistemas digitais modernos.

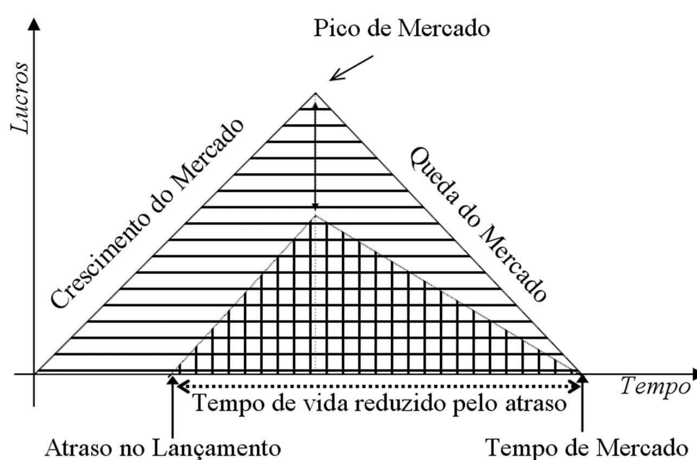

Figura 1: Impacto do atraso do produto no tempo de mercado

De fato, a escolha do *hardware* ao invés do *software* é simplesmente um opção de projeto e deve ser realizado de acordo com as exigências do projeto, isto é, enquanto o *hardware* melhora o desempenho do projeto, seu custo é elevado. O *software*, por sua vez, traz flexibilidade ao projeto em contra partida a uma significativa perda de desempenho.

[1]Por simplicidade utilizaremos somente o termo *co-design*

Assim, a metodologia *co-design*, apoiada em princípios e técnicas, considera esta escolha como uma decisão técnica do processo de desenvolvimento do projeto e não como uma escolha baseada somente na experiência do projetista.

1.2 Elevação dos Níveis de Abstração

Uma maneira clássica de gerenciar a complexidade no desenvolvimento de sistemas é elevar o nível de abstração, de maneira a reduzir os detalhes de implementação e assim facilitar a compreensão do sistema. Conforme o entendimento do sistema aumenta, o projeto deve ser refinado para os níveis mais baixos, agora com maior facilidade, acrescentando mais detalhes a cada nível de implementação.

A figura 2 compara os níveis de abstrações clássicos dos componentes de *hardware* e *software*, utilizados durante décadas como base para o projeto de sistemas embarcados.

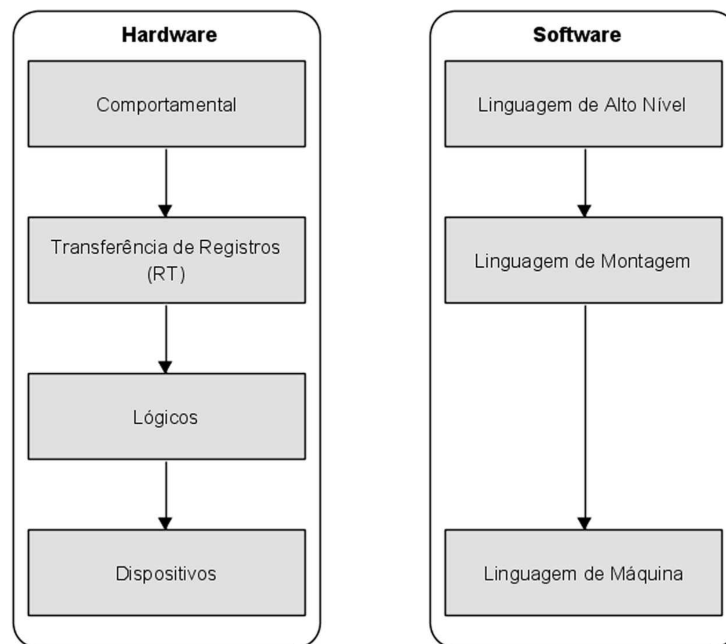

Figura 2: Níveis Clássicos de Abstrações *Hardware/Software*

Os níveis lógicos e de dispositivos para o *hardware* e a equivalente linguagem de máquina utilizada nos primitivos projetos de *software*, são níveis de abstrações já há muito tempo superados, com o auxílio de montadores e ligadores para o *software* e ferramentas de síntese para o *hardware*. Na década de 80 as ferramentas de síntese RT (*Register Transfer*) fizeram o mesmo pelo nível de transferência de registros e somente na década 90, ferramentas de síntese comportamentais começaram a surgir para a implementação dos componentes de *hardware* (VAHID; GIVARGIS, 2002).

Como pode ser observado na figura 2, o nível de linguagem de montagem do *software* equivale ao nível RT do *hardware*. A elevação dos níveis de abstração de *software* para as linguagens de alto nível, equivalente ao nível comportamental do *hardware*, foram realizados através de ferramentas de compilação, popularizadas segundo (VAHID; GIVARGIS, 2002) na década de 60.

No nível de abstração comportamental, as visões de projeto *hardware* e *software*, que no passado eram radicalmente diferentes, começaram a apresentar importantes similariedades, envolvendo em ambos os casos a especificação de um programa sequencial, viabilizando e motivando a utilização de técnicas *co-design*.

1.3 Motivação

Co-design é uma campo relativamente novo, surgindo, após a criação das primeiras ferramentas de síntese comportamental e consequentemente a criação das linguagens de descrição de *hardware*, tais como VHDL e Verilog[2]. De fato, na última década, este campo gerou uma grande gama de pesquisas onde muitos conceitos, técnicas, metodologias, *frameworks* e ferramentas foram criadas.

Desde seu surgimento, a área de pesquisa *co-design*, evoluiu bastante nos últimos anos, atrelada às modificações tecnológicas e avanços no campo da microeletrônica. A capacidade de integração dos *chips*, por exemplo, tem aumentando a uma taxa de 58% ao ano, enquanto a produtividade de projeto tem crescido a uma taxa de 21% ao ano, conforme ilustrado na figura 4.

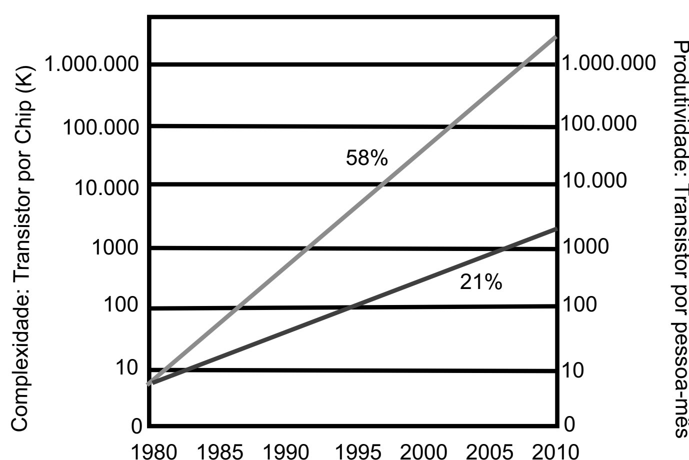

Figura 3: Capacidade de integração x produtividade. fonte: (ROWEN, 2002)

[2]Maiores detalhes sobre estas, ou outras linguagens de descrição de *hardware*, podem ser encontrados em: http://www.eda.org.

Isto implica na necessidade de novas abordagens de projeto de maneira a tentar acompanhar a crescente capacidade de integração dos *chips*, sendo esta a principal motivação da pesquisa realizada.

Além disso, a introdução de projetos SoC modifica em vários pontos as exigências de desenvolvimento (SIEWERT, 2005). A reutilização dos componentes criados, por exemplo, tem sido uma importante forma de gerenciar a complexidade destes sistemas. Ao invés do projetista modelar o projeto como um conjunto de componentes de prateleira, agora o sistema pode ser visualizado como um conjunto de núcleos ou de processadores reutilizáveis, chamados de núcleos de propriedade intelectual ou *IP-Core*.

A reutilização e comercialização destes núcleos cria uma oportunidade sem precedente para os desenvolvedores de sistemas embarcados, de maneira que é imperativo um processo de desenvolvimento bem definido, utilizando uma ferramenta que seja simples e de fácil acesso aos desenvolvedores, demonstrando que o desenvolvimento de complexos projetos *co-design* não estão restritos somente a grandes corporações.

1.4 Definição do Problema

O problema consiste em definir um processo sistemático de desenvolvimento, que possibilite especificar e refinar o projeto, em um elevado nível de abstração, independente de sua natureza arquitetural, ao qual é decidida de maneira natural através dos vários passos de refinamento definidos pela metodologia proposta.

A figura 4, ilustra a foco do problema em relação aos níveis de abstração típicos. Nesse caso, a metodologia proposta pretende criar um ambiente unificado para o projeto de sistemas *co-design*.

Iniciando no nível de sistema unificado, a metodologia deverá refinar o sistema até que possa ser mapeado em componentes de *hardware* e *software* de acordo com os requisitos do projeto. Estes níveis comportamentais ainda podem ser conduzidos de maneira unificada, até que o projeto possa ser implementado em suas respectivas tecnologias. Esse último passo foge do escopo do trabalho, podendo ser realizado por ferramentas automatizadas, tais como Cynthesizer[3] ou CoCentric[4] para os componentes de *hardware* e compiladores para os componentes de *software*.

Para avaliar o processo de desenvolvimento, uma interface USB (*Universal Serial*

[3]Forte Design Systems. Cynthesizer. http://www.forteds.com/

[4]Synopsys Inc. CoCentric System Studio. http://www.synopsys.com/products/cocentric_studio/

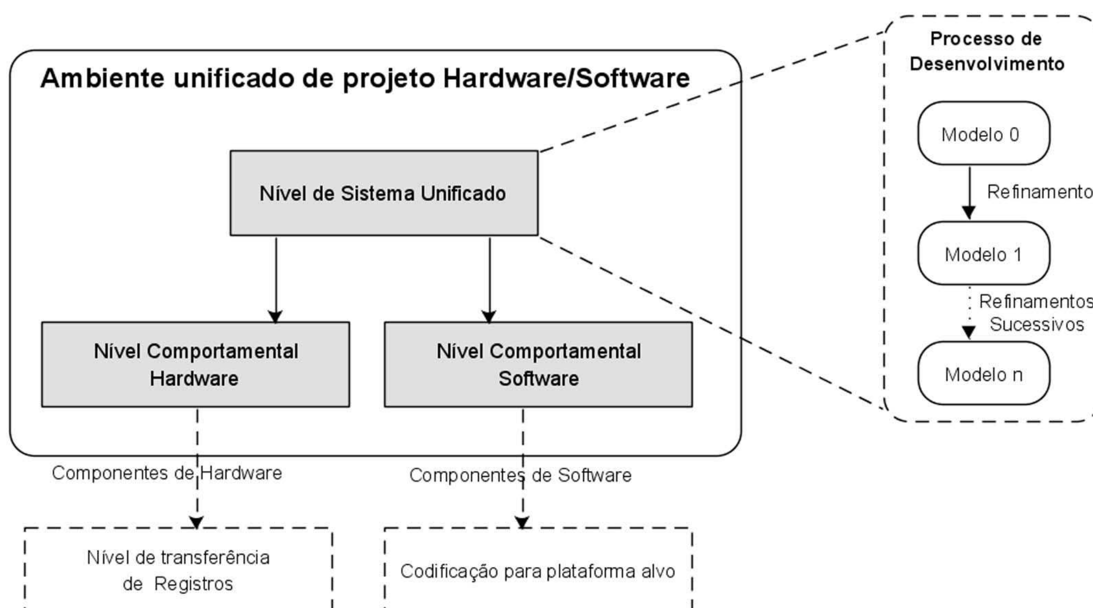

Figura 4: Ambiente unificado de projeto *hardware/software*

Bus) será especificada e refinada suavemente utilizando as técnicas *co-design*, sugeridas no projeto, servindo como inspiração para a definição metodológica realizada e como uma exemplificação prática do processo de desenvolvimento proposto.

A interface de comunicação USB possui requisitos bem definidos de desempenho e uma ampla possibilidade de escolhas entre componentes *hardware* e *software*, motivando a sua escolha como um estudo de caso. Além disso, trata-se de um projeto bastante atrativo, em parte por sua ampla aceitação no mercado de computadores pessoais, industriais e de comunicação embarcada e, em parte, por se tratar de uma iniciativa aberta para tentar solucionar inúmeros problemas existente nas outras interfaces, possuindo assim, documentação de fácil acesso, capaz de propiciar o levantamento inicial para o desenvolvimento do projeto.

1.5 A Metodologia Proposta

A metodologia proposta, conforme ilustrado na figura 4, se baseia na abordagem de projeto *top-down*, através de uma seqüência de passos suaves de refinamentos, considerando um processo iterativo envolvendo modelagem, prototipação e validação.

A base deste processo é permitir a rápida criação de protótipos executáveis, capazes de serem simulados e avaliados através de modelos de testes reutilizáveis, comumente chamados de *testbenches*.

Cada passo de refinamento proporcionará um melhor entendimento do sistema, capacitando o projetista a avaliar de maneira simples e direta, as várias decisões de projeto, incluindo o particionamento *hardware/software*. Mesmo neste caso, o projeto é refinado seguindo um mesmo formalismo, através de linguagens capazes de atuar no nível de sistema.

Entre estas linguagens, conhecidas como SLDL (*System Level Design Language*), o SystemC[5] foi escolhido para a metodologia proposta. O mesmo é inteiramente baseado no C++ e pode ser obtido livremente sem custos. Aliado a isso, a capacidade de orientação a objetos do C++ e a grande aceitação da ferramenta motivaram a sua escolha (Um resumo dos fundamentos do SystemC se encontra no apêndice A).

A orientação a objetos SystemC permitiu a utilização direta da UML (*Unified Modeling Language*), solidificando ainda mais a união das disciplinas de desenvolvimento *hardware/software*. De fato, projetos complexos como o USB são difíceis de serem especificados sem uma ferramenta capaz de auxiliar a análise e levantamento de requisitos do projeto. A forma de integrar o UML no processo de desenvolvimento é apresentado no capítulo 3.

Todos os passos definidos no processo de desenvolvimento são baseados no estado da arte atual do campo *co-design*, assim, os princípios de modelagem em nível de transações, conhecidos como TLM (*Transaction Level Modeling*)[6], são utilizadas desde o início da metodologia, com a preocupação de manter os detalhes da comunicação separado dos detalhes funcionais.

1.6 Contribuição do Trabalho

Apesar de atualmente a área de pesquisa *co-design* estar mais estável com alguns métodos e ferramentas já bem solidificados e aceitos no meio acadêmico, sua área de atuação é muito vasta e complexa, com vários problemas em aberto.

Uma metodologia *co-design* completa deve cercar várias etapas, desde a concepção do sistema até sua síntese, passando por passos complexos como o particionamento entre *hardware* e *software*, co-simulação e a co-verificação do sistema.

Apesar de cada uma destas etapas já terem sido bastante discutidas com particularidades e problemas específicos, a proposta defendida neste trabalho as aplica através

[5]www.systemc.org

[6]O TLM bem como o estado da arte co-design são comentadas no capítulo 2

de um processo de refinamento sucessivo, baseado no ciclo de prototipação de sistemas, envolvendo os passos de análise e levantamento de requisitos, prototipação e validação.

A proposta não se prende aos passos tradicionais de uma metodologia *co-design* (BARROS et al., 2000), ao invés disso, se baseia em modelos de refinamento intermediários, com base em princípios e técnicas envolvendo o estado da arte de desenvolvimento co-design e o nível de sistema.

Os modelos utilizados foram inspirados nos modelos introduzidos pela metodologia SpecC (DÖMER, 2002) e nos modelos sugeridos por (GROTKER, 2002) para o SystemC. Os modelos de refinamento definidos pela proposta, se diferenciam principalmente para atender às características do SystemC e ao ciclo de refinamento definido pela metodologia.

Enquanto a metodologia SpecC possui um padrão de desenvolvimento consolidado, bem documentado e comprovado, não foi encontrado uma metodologia similar para o SystemC. Nesse caso, a metodologia definida neste trabalho se equipara ao SpecC em diversos pontos, se destacando em algumas características:

1. A Linguagem de Projeto em Nível de Sistema utilizada pela metodologia, o SystemC, se destaca por sua grande aceitação, orientação a objetos e por ser de fácil acesso.
2. A metodologia criada possui um ciclo de desenvolvimento formalizado, representado por um ciclo de prototipação pirâmidal.
3. A etapa de particionamento, uma das mais complexas etapas de uma metodologia co-design, é realizada em dois passos de refinamento, permitindo um melhor gerenciamento da complexidade e uma maior flexibilidade na mudança dos parâmetros.
4. Utilização da modelagem gráfica UML incorporada ao processo de desenvolvimento.

1.7 Organização do Trabalho

As pesquisas realizadas sobre o tema *co-design* são resumidas no capítulo 2, fornecendo a base conceitual para o projeto.

No capítulo 3, o processo de desenvolvimento proposto é descrito, relacionando as ferramenta selecionadas e sugeridas para a aplicação da metodologia.

No capítulo 4, a metodologia aplicada a um caso de estudo é descrita, provendo um guia prático de sua aplicação. Com isso será possível constatar os reais benefícios dos

conceitos e técnicas levantados, de maneira a poder realizar a conclusão do trabalho no capítulo seguinte.

Assim, a conclusão descrita no capítulo 5, resume os benefícios alcançados pela metodologia e os problemas visualizados ainda em aberto, que podem ser alcançados em futuros projetos.

1.8 Considerações Finais

Este capítulo, apresentou de maneira resumida o problema pesquisado, descrevendo a metodologia, objetivos e, ainda que superficialmente, os conceitos chaves utilizados na dissertação. Estes serão, melhor detalhados no capítulo seguinte de maneira a fornecer o referencial teórico necessário para a realização do trabalho.

2 A metodologia *co-design*

Este capítulo apresenta o estado da arte da metodologia *co-design*, procurando situar o trabalho em relação as recentes pesquisas sobre o tema.

2.1 *Hardware* e *software co-design*

Uma definição clássica para *hardware/software co-design* foi dada por Micheli e Gupta (MICHELI; GUPTA, 1997), onde a mesma se caracteriza como uma metodologia que procura atender os requisitos do sistema, explorando a sinergia existente entre o *hardware* e o *software* através de um projeto concorrente.

Essa sinergia representa a possibilidade de cooperação entre estes componentes, visto que atualmente não existe uma diferença fundamental do que pode ser implementado em *hardware* em relação ao que que pode ser implementado em *software*. De fato, esta exploração só é possível através de um projeto concorrente, onde o desenvolvimento segue em um fluxo de *hardware/software* unificado, sem uma separação distinta entre os componentes. Somente em etapas posteriores do projeto, quando se possui informações suficientes, esta separação é realizada. Isto difere do fluxo de projeto tradicional, conforme ilustra a figura 5.

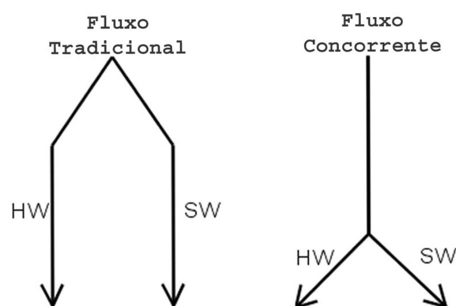

Figura 5: Fluxo Clássico x Fluxo Concorrente

No fluxo clássico a divisão entre os componentes é realizada ainda na concepção do projeto, geralmente, baseado na experiência do projetista.

Em resumo, os seguintes pontos podem ser considerados como fatores evolutivos associados à metodologia *co-design*:

- **A grande maioria dos sistemas digitais são programáveis e incluem componentes em *hardware* e *software***: Tipicamente constituído por uma plataforma de *hardware* capaz de executar aplicações de *software*.
- **A utilização do *software* sugere um meio de diferenciação de produtos baseado numa mesma arquitetura de *hardware***: Com o aumento dos custos de produção, convém aproveitar ao máximo a arquitetura do *hardware* e, nesse caso, o *software* agrega um atrativo importante para flexibilidade do projeto.
- **A complexidade inerente dos modernos sistemas:** Com o aumento da complexidade é improvável que o projetista por mais experiente que seja, possa otimizar todos os pontos do projeto baseando-se somente em sua habilidade. Mesmo se o fizer, o tempo será elevado e com alta probabilidade de erros.
- **Diversidade de aplicações e a demanda do mercado**: A realimentação do mercado em relação à tecnologia faz com que estes sistemas tenham um tempo de vida de mercado relativamente curto, sendo cada vez mais necessário que os sistemas sejam desenvolvidos em curtos espaços de tempo.

Estes fatores sugerem o uso da metodologia *co-design* realizada de acordo com os objetivos do projeto, como: qualidade, custo de projeto e produção, flexibilidade, tolerância a falhas, entre outros que dependem justamente de como o *hardware* e *software* são projetados.

2.2 Objetivos do *co-design*

Uma metodologia *co-design* define vários passos de projeto que, de maneira resumida, deve ter os seguintes objetivos abaixo relacionados:

- Proporcionar um desenvolvimento concorrente, integrado e eficiente dos componentes de *hardware/software*;
- Gerenciar a complexidade de sistemas heterogêneos;
- Diminuir o tempo de projeto;

2.3 Etapas da metodologia *co-design*

Normalmente, uma metodologia *co-design* completa deve ser capaz de, partindo da concepção do sistema, promover um projeto unificado até o momento em que o projeto possa ser particionado em componentes distintos. Durante este processo a metodologia passa por várias etapas de refinamentos, conforme esboça a figura 6 (BARROS et al., 2000).

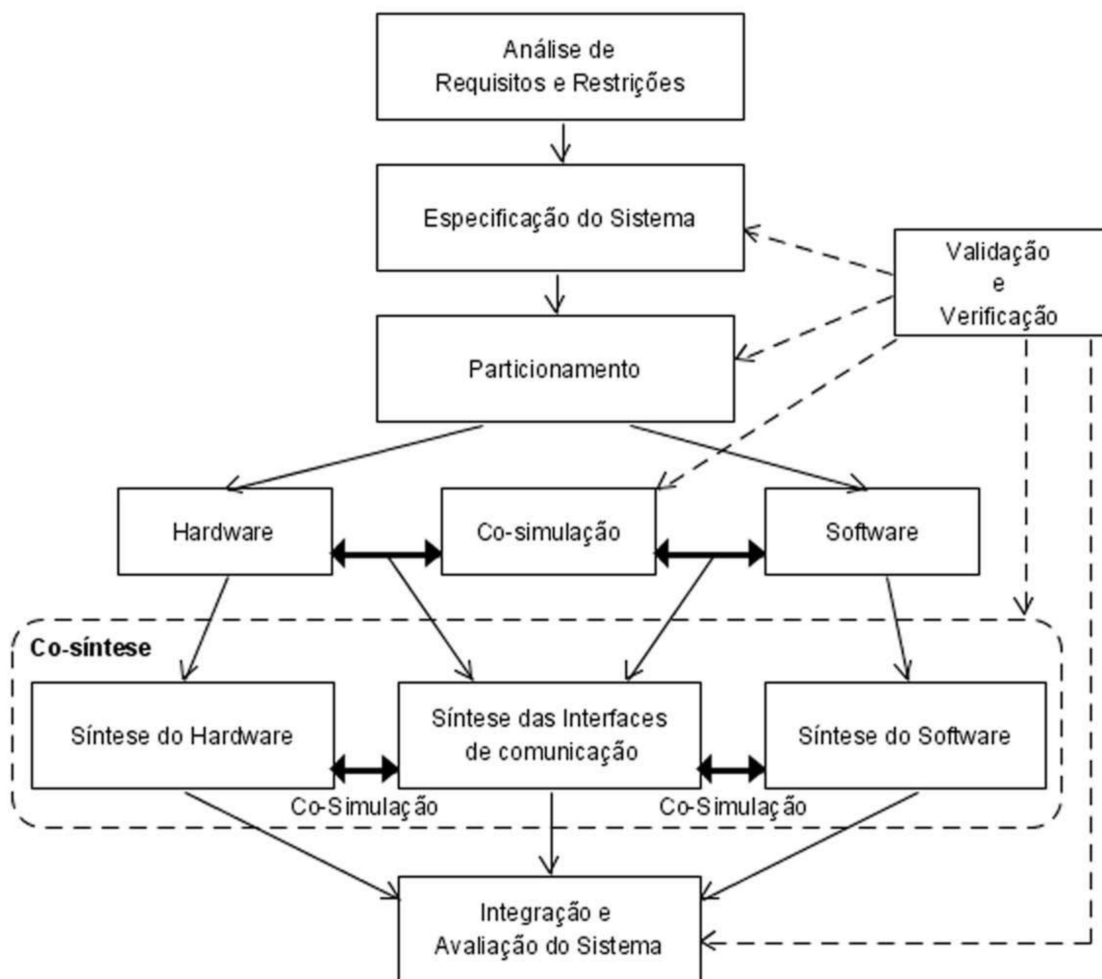

Figura 6: Etapas da metodologia *co-design*

Assim de acordo com (BARROS et al., 2000), a metodologia *co-design* compreende as seguintes etapas, explicadas nas próximas subseções:

- Análise de Requisitos e Restrições
- Especificação do Sistema
- Particionamento do Sistema

- Co-simulação
- Co-síntese
- Avaliação do Sistema

2.3.1 Análise de requisitos e restrições

Compreende a etapa padrão de levantamento de requisitos, onde são definidas as características do sistema, servindo como base para a etapa de especificação.

As principais características a serem capturadas são:

- Requerimentos de tempo real;
- Tecnologia de realização;
- Programabilidade;
- Consumo de Potência;
- Tamanho do Produto;
- Custo de Desenvolvimento e Produção;
- Ambiente de utilização do produto;
- Confiabilidade, Manutenção e Evolução do Projeto.

2.3.2 Especificação do Sistema

Compreende a modelagem do sistema, preferencialmente através de modelos executáveis e simuláveis. Esta modelagem é realizada independente arquitetura alvo, onde a própria arquitetura deverá ser definida com base no modelo.

Em resumo, a especificação do sistema *co-design* pode ser situada em relação aos seguintes pontos:

- Independência da Arquitetura alvo;
- Altos níveis de Abstração;
- Formalismo homogêneo durante a especificação;
- Capacidade de Verificação e Validação através de uma especificação executável.

2.3.3 Particionamento

Consiste em subdividir a especificação e decidir quais partes serão mapeadas em *hardware* e quais serão mapeadas em *software*. Esta etapa é uma das mais importantes da metodologia *co-design*, envolvendo duas atividades básicas: seleção dos componentes da arquitetura alvo e o particionamento do sistema entre estes componentes (BARROS et al., 2000).

O problema do particionamento segundo (BARROS et al., 2000) é um típico problema NP-COMPLETO devido as inúmeras possibilidades de divisões e de escolhas de componentes para arquitetura alvo, conforme ilustra a figura 7.

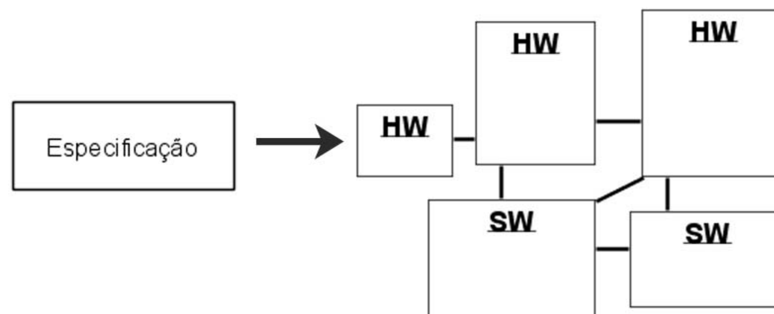

Figura 7: A Etapa de Particionamento

2.3.4 Co-Simulação

Mesmo após o particionamento, os passos seguintes de refinamento devem ser realizados de maneira integrada sob pena de adição de erros nos refinamentos sucessivos. A co-simulação nesse caso fornece o suporte a esta verificação integrada.

Dependendo da metodologia *co-design* adotada, a co-simulação pode se tornar bastante complexa, como o caso da co-simulação para a abordagem heterogênea, que nesse caso envolveria a integração entre diferentes linguagens e simuladores.

A figura 8 exibe a comparação da co-simulação em relação as abordagens de desenvolvimento homogênea e heterogênea.

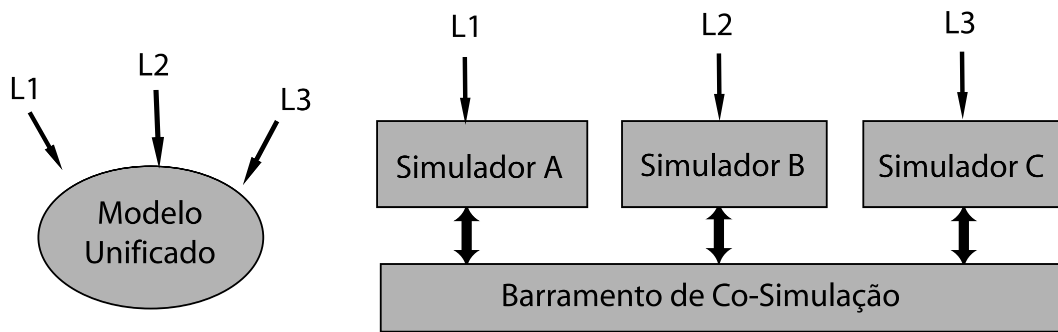

Figura 8: Co-simulação Homogênea x Heterogênea

2.3.5 Co-Síntese

A co-síntese, consiste no mapeamento do modelo particionado em uma arquitetura alvo real.

Após o particionamento, o projeto é em geral descrito como um conjunto de módulos de *hardware/software*, fornecendo uma visão arquitetural intermediária do projeto, conhecida como protótipo virtual (BARROS et al., 2000), nesse caso, cada um destes módulos são sintetizados em seus respectivos os componentes, incluindo a síntese da interface de comunicação, conforme exibido na figura 9.

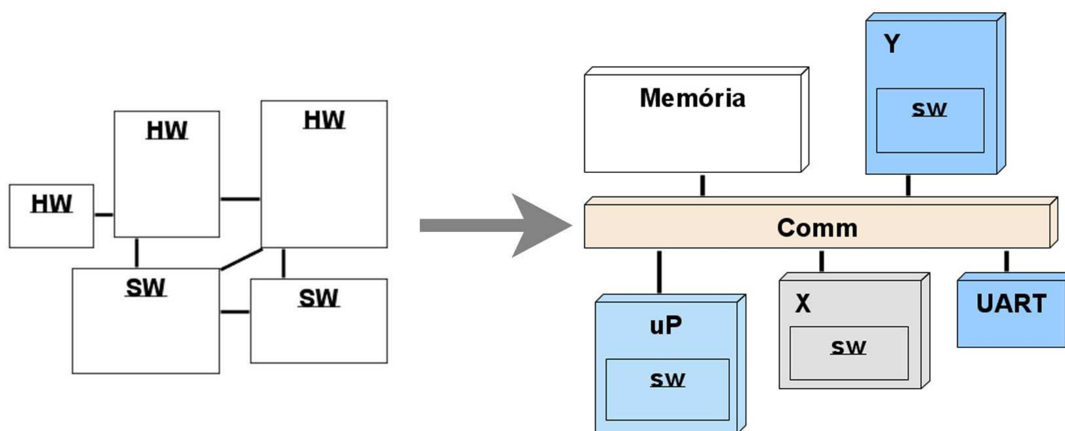

Figura 9: A Etapa de Co-síntese

2.4 Domínios de Representações

Conforme anteriormente comentado, uma maneira de gerenciar a complexidade é a elevação dos níveis de abstrações. Tradicionalmente pode-se considerar os seguintes níveis de abstrações (WALKER; THOMAS, 1985):

- Nível de Sistema;
- Nível de Transferência de Registros;
- Nível de Portas Lógicas;
- Nível de Dispositivos.

Cada um destes níveis podem ser descritos em três diferentes visões, conforme figura 10, que ilustra o gráfico em "Y" definido em (GAJSKI; KUHN, 1983).

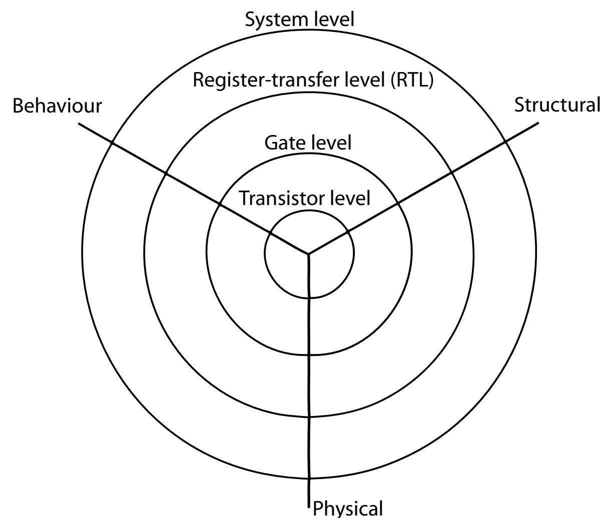

Figura 10: Gajski´s *Y-chart*. fonte: (GERSTLAUER, 2002)

O gráfico consiste de três eixos, *behavioral*, *structural* e *physical* que são os domínios de representações definidos como segue (GERSTLAUER, 2002):

Representação funcional (*behavioral*): O sistema é descrito em termos de suas funcionalidades, desconsiderando como estas funcionalidades são implementadas internamente.

Representação estrutural (*structural*): Descreve internamente as funcionalidades, como um conjunto de componentes e suas conectividades.

Representação física (*physical*): Descreve o layout espacial dos componentes, tratando os detalhes das interconexões e roteamento do chip.

Nesse caso, o interesse deste trabalho recai sobre as visões funcional e estrutural. Assim, o projetista inicia o projeto como uma especificação em alto nível até um modelo estrutural, refletindo as escolhas arquiteturais.

2.5 O nível de sistema

O nível de sistema corresponde ao estado da arte dos atuais projetos embarcados e SoC. Na tabela 1 (WALKER; THOMAS, 1985) pode-se observar o objetivo deste nível em relação aos outros níveis de abstração.

Níveis de Abstração	Domínios de Representações		
	Funcional	Estrutural	Físico
Sistema	Algorítmo	Processadores, Memórias	Circuitos, Macro-células
Transferência de Registros	Fluxo de Dados	Registros, ALUs, MUXs	Disposição dos blocos sobre o silício
Lógico	Equações booleana, Tabelas verdades	Portas Lógicas, Flip-flops	Células
Dispositivo	Equações elétricas, funções de transferência	Transístores, capacitores e resistores	Geometria sobre o silício

Tabela 1: Níveis de abstrações x Domínios de Representações

A especificação no nível de sistema provê a base para o entendimento do projeto inteiro. Para que o projetista possa refina-lo conforme o aumento do nível de conhecimento sobre o projeto. Nesse caso a validação e a verificação da especificação ainda no nível de sistema é crucial para que o projetista possa seguir o processo de desenvolvimento de maneira segura.

Dessa maneira, este trabalho se concentra no nível de sistema, mais especificamente (conforme comentado na seção anterior) nos domínios de representação funcional e estrutural, onde os componentes são constituídos por memórias, barramentos e elementos de processamento ou simplesmente PEs (*processing elements*).

2.6 Abordagens de projeto em nível de sistema

Pode-se distinguir três abordagens para o projeto em nível de sistema: síntese de sistema (*system-synthesis*), baseado em plataformas (*platform based*) e baseado em componentes (*component based*) (CAI, 2004), definidas a seguir:

Projeto de síntese de sistema (*system-synthesis design*): Consiste em um fluxo de projeto *top-down*. O projeto nesse caso é iniciado com uma representação funcional e é refinado em vários passos de síntese até seu modelo estrutural, onde sua representação arquitetural é definida (CAI, 2004; ABDI et al., 2003).

Projeto Baseado em Componentes (*component based design*): Consiste em um fluxo de projeto *bottom-up*, onde os componentes estruturais da arquitetura são selecionados e modelados para, enfim, atender as funcionalidades do sistema. Nesse caso o projeto parte da descrição estrutural para a funcional (ARATO; MANN; ORBAN, 2004).

Projeto baseado em plataformas (*platform based design*): Consiste no meio termo entre o fluxo de projeto *top-down* e *bottom-up*, conhecida como *meet-in-the-middle*, onde a arquitetura é pré-definida ao invés de ser gerada através da especificação funcional do projeto. Nesse caso, a abordagem mapeia o modelo funcional na plataforma selecionada (VINCENTELLI, 2002). Uma descrição dos benefícios e desafios da abordagem pode ser consultada em (VINCENTELLI et al., 2004).

O projeto desta dissertação, nesse caso, se concentra na abordagem de síntese de sistema, iniciado pela especificação em nível de sistema. Normalmente a especificação é representada em uma forma abstrata, onde o sistema deve ter condições de ser simulado e validado em um elevado nível de abstração. É importante que esta abordagem seja baseada em modelos formais de computação para que não ocorra interpretações erradas no processo de desenvolvimento.

Os modelos de computação mais utilizados em sistemas embarcados, citando: (1) Eventos Discretos, (2) Máquina de Estados Finitos, (3) Síncronos ou Reativos e (4) Fluxo de Dados entre outros são comparados em (EDWARDS et al., 1997) e (CORTÉS; ELES; PENG, 1999).

2.7 Linguagens no nível de sistema

Para que o sistema possa ser especificado e simulado é necessário uma linguagem de especificação formal. As linguagens no nível de sistema ou SLDL (*System Language Design Level*), devem permitir que o sistema seja especificado livre de detalhes de implementação. Assim, idealmente, devem possuir as seguintes características:

1. Linguagem Mista;

2. Suporte a vários níveis de abstração;

3. Suporte a vários modelos de computação;

4. Possa ser simulado;

5. Modelagem hierárquica.

Por linguagem mista, deseja-se que a linguagem possa ser utilizada tanto para refinar um sistema de *hardware* como de *software*, sendo importante nesse caso, que o refinamento e os níveis de abstrações utilizem a mesma semântica utilizada na especificação. Nesse caso os modelos de testes (*testbenches*), utilizados para simulação e validação, podem também ser aproveitados em cada nível de abstração, facilitando como um todo o refinamento do projeto.

O critério de modelagem hierárquica é um requisito de desenvolvimento básico para sistemas digitais em elevados níveis de abstrações, permitindo que cada módulo seja quebrado em outros e assim por diante.

Entre as modernas linguagens do nível de sistema, é possível destacar: SystemC (OSCI, 2002a), SpecC (GAJSKI et al., 2001) e System-Verilog (ACELLERA, 2004).

SystemC e SpecC satisfazem as restrições definidas e são baseadas na linguagem C++ e C respectivamente. O SystemC é uma biblioteca de classe C++ adicionando propriedades necessárias para a modelagem de *hardware* enquanto o SpecC é um super conjunto de extensão do ANSI-C (CAI; VERMA; GAJSKI, 2003). Já o System-Verilog estende o verilog-2001 para atuar no nível de sistema (SUTHERLAND, 2003).

Nesse caso, o SystemC foi escolhido para a metodologia definida no trabalho. A capacidade de orientação a objeto do C++, propiciando uma modelagem orientada a objetos, sua grande aceitação e sua alta disponibilidade, podendo ser obtida com facilidade, inclusive com acesso ao código fonte (*open-source*), justificaram a escolha da mesma.

2.8 O SystemC

A linguagem SystemC, surgiu da idéia de se utilizar uma linguagem que fosse bem conhecida tanto por projetista de *software* como de *hardware*. Desta iniciativa um consórcio

chamado OSCI - *Open SystemC Iniciative*[1] foi formado para definir as características que deveriam ser criadas, para que a linguagem C++ pudesse ser utilizada para a modelagem de *hardware*. Assim, as bibliotecas para serem utilizadas sobre o núcleo (*Kernel*) da linguagem C++, foram implementadas de acordo com a arquitetura exibida na figura 11.

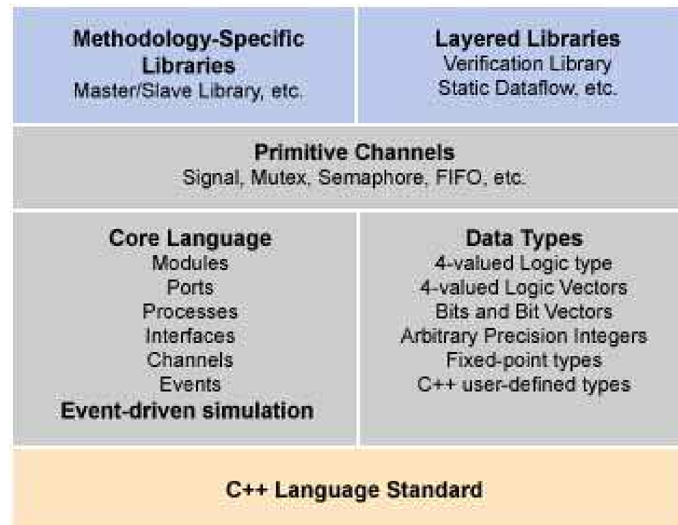

Figura 11: Arquitetura do SystemC. fonte: (OSCI, 2002b)

Nesse caso, é possível destacar os seguintes aspectos implementados em C++, para formar o SystemC:

1. Noção de Tempo;
2. Concorrência;
3. Tipos de Dados de *Hardware*.

Essas características permitiram a criação de construtores similares aos utilizados em linguagens de descrições de *hardware* como VHDL e Verilog, com a utilização de módulos, portas, sinais e tipos específicos de *hardware* como *bits*, vetores de *bits* entre outros (OSCI, 2002a).

O SystemC assim manteve o projeto de *hardware* bem próximo de padrões já estabelecidos, tais como: VHDL, Verilog, entre outras. Permitindo ao desenvolvedor uma facilidade de aprendizado da estrutura utilizada pelo SystemC, podendo ser empregado tanto em um desenvolvimento estrutural com o uso de módulo hierárquicos bem como em módulos comportamentais com o uso de processos concorrentes, que nesse caso funcionam de maneira similar aos processos utilizados em VHDL (Maiores detalhes sobre os fundamentos do SystemC foram colocados no apêndice A).

[1]www.systemc.org

2.9 Modelos de abstração do nível de sistema

As linguagens de especificação em nível de sistema, como o SystemC, permitem um mesmo formalismo durante todo o projeto do sistema. A figura 12 ilustra estas características, em relação ao métodos anteriores de projetos em nível de sistema (OSCI, 2002b).

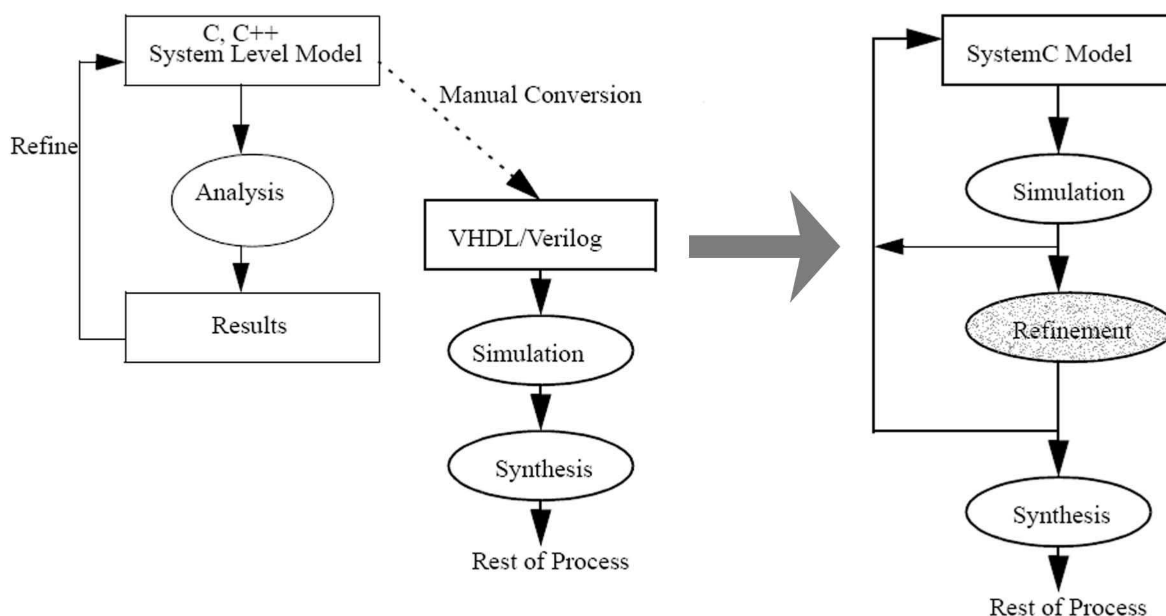

Figura 12: SLD Tradicional x SLD usando systemC. fonte: (OSCI, 2002b)

Tradicionalmente as empresas analisavam o sistema em linguagens de sistema, como C ou C++, porém a tradução destes para as linguagens de descrição de *hardware*, tais como VHDL ou Verilog, eram manuais e, conseqüentemente, bastante propensa a erros. A linguagem SystemC, ao contrário, permite especificar o projeto tanto no nível de sistema, quanto nos níveis de abstração abaixo, sendo possível realizar inclusive códigos sintetizáveis em RTL.

Durante o processo de refinamento hachurado na figura 12, vários modelos de projeto intermediários devem ser definidos, de maneira a reduzir a lacuna de abstração, existente entre o nível de sistema e o nível de implementação RTL. Isto caracteriza a definição de um processo de desenvolvimento, que é o objetivo desta dissertação.

O modelo conhecido como nível de transação ou TLM (*Transaction Level Modeling*) (CAI; GAJSKI, 2003a), tem sido recentemente bastante comentado, como uma forma de reduzir esta lacuna de abstração, através da separação entre a especificação da funcionalidade e suas respectivas comunicações. Este princípio de projeto, propicia, entre vários benefícios, a integração simplificada de componentes reutilizáveis ou *IP-Core* (Núcleos de Propriedades Intelectuais) (VANTHOURNOUT; GOOSSENS; KOGEL, 2005).

A figura 13, mostra vários possíveis modelos para os níveis de abstrações no nível de sistema (CAI; GAJSKI, 2003b). Os modelos são dispostos convenientemente em dois eixos perpendiculares representando os refinamentos ortogonais da computação funcional e da comunicação (KEUTZER et al., 2000).

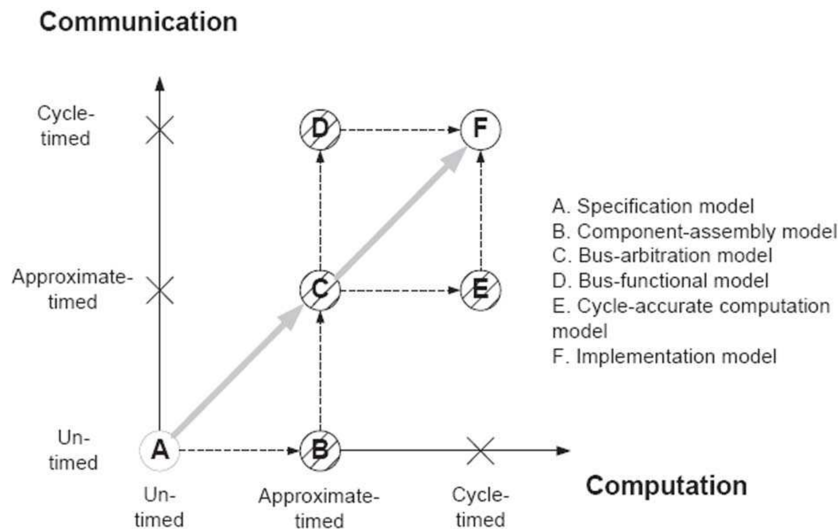

Figura 13: Modelos de Abstração TLM. fonte: (CAI; GAJSKI, 2003b)

Cada metodologia poderia optar por uma caminho diferenciado de refinamento, conforme as setas da figura 13 sugerem. Assim, este trabalho utiliza os modelos representados na figura 13, como referência para a definição do processo de desenvolvimento. Maiores detalhes sobre estes modelos podem ser encontrados também em (CAI, 2004).

2.10 Considerações Finais

Este capítulo apresentou os conceitos necessários ao desenvolvimento e utilização de uma metodologia *co-design*. Estes conceitos correspondem ao levantamento bibliográfico necessário para definir e situar o trabalho em relação ao estado da arte atual das pesquisas relacionadas ao desenvolvimento no nível de sistema e *co-design*.

O próximo capítulo, especifica o processo de desenvolvimento a ser utilizado para, em seguida, aplicá-lo no desenvolvimento de uma interface USB.

3 Processo de Desenvolvimento

O capítulo anterior forneceu uma base de apoio sobre os conceitos *co-design*. Conforme foi comentado, uma metodologia *co-design* compreende várias etapas capazes de conduzir um projeto desde a concepção até sua implementação.

Este capítulo apresenta o processo de desenvolvimento da proposta, definindo vários modelos de refinamento intermediários, com os quais as atividades do processo de desenvolvimento serão aplicadas, estas atividades são detalhadas, explorando as características da linguagem SystemC (Vide apêndice A).

3.1 Abordagem de desenvolvimento

A abordagem proposta consiste em uma adaptação do modelo de prototipação de sistemas (PRESSMAN, 2001; SOMMERVILLE, 2001), envolvendo um processo de iteração entre análise e levantamento de requisitos e a especificação do sistema. A cada iteração, uma nova especificação executável é refinada.

A figura 14 ilustra bem a proposta. Inicialmente a base da pirâmide é especificada com uma visão conceitual em alto nível do sistema, servindo como alicerce para os próximos passos de refinamento. Cada camada fornece um melhor entendimento do sistema, de seus comportamentos e de seus requisitos. Assim que cada camada é validada, a camada acima pode ser iniciada, acrescentando, a cada passo, maiores detalhes da complexidade do sistema.

Os três lados da pirâmide representam as principais ações realizadas durante o refinamento: análise dos requisitos, prototipação e validação. A etapa de análise e levantamento de requisitos produz um modelo de especificação do sistema, a etapa de prototipação traduz o modelo em um protótipo executável do sistema e a etapa de validação realiza a verificação do protótipo obtido. Isto pode ser melhor visualizado na figura 15, que exibe somente a base da pirâmide.

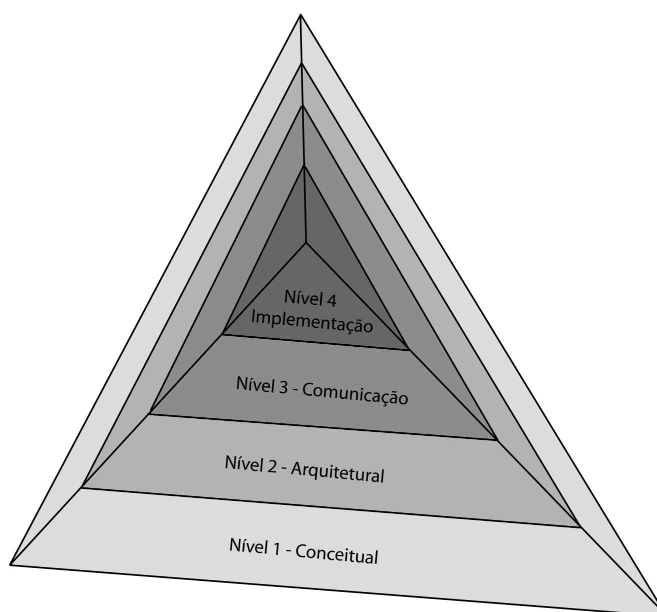

Figura 14: Representação pirâmidal da abordagem de prototipação

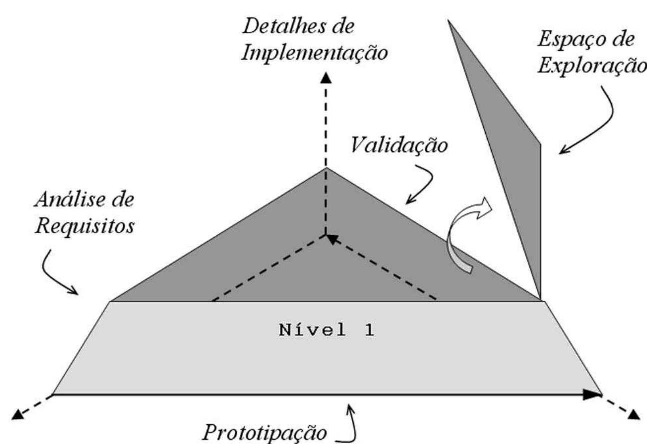

Figura 15: Visualização da base da pirâmide de prototipação

Conforme ilustrado na figura 15, a exploração espacial do projeto pode ser observado pela extensão de cada camada na pirâmide, motivando o uso da representação pirâmidal para descrever o processo de desenvolvimento. Assim, a cada refinamento, o espaço de exploração do projeto vai se reduzindo.

Apesar da figura 14 exibir somente quatro modelos de refinamento, pode-se ter tantos modelos quantos forem necessários. Nesse caso, os modelos definidos no projeto são especificados na próxima seção.

Nas subseções que seguem, as três atividades do processo de desenvolvimento representadas pelos eixos da pirâmide, são melhor definidas.

3.1.1 Análise e Levantamento de Requisitos

Esta etapa envolve o levantamento das necessidades do sistema, como o sistema deverá se comportar, o que deverá ser desenvolvido para atender a estas necessidades e a correta interpretação de requisitos de projeto, como desempenho, custo e diversos outros parâmetros.

O principal produto desta etapa são os requisitos funcionais. De fato, o levantamento destes requisitos em sistemas complexos como o caso do USB, não é uma tarefa trivial, dado a quantidade de informação que se deve administrar. Até mesmo um especialista em sistemas USB teria dificuldades.

Lidar com esta complexidade é possível através de modelos capazes de representar o sistema, nesse contexto, uma tendência que se tem observado é a possibilidade de utilizar UML (RUMBAUGH; JACOBSON; BOOCH, 1999).

Apesar de ter sido criada inicialmente para representar modelos de *software*, a UML possui uma especificação evolutiva, capaz de se adaptar a novos conceitos e necessidades conforme (KUKKALA et al., 2005). Dessa maneira, extensões à UML foram criadas para propiciar projetos de sistemas embarcados e SoC (OMG, 2005; UML FOR SOC FORUM, 2004; SELIC, 1998).

A utilização da UML para auxiliar o processo de análise dos requisitos funcionais do sistema é natural e vai na linha da tendência emergente de união entre as disciplinas de *software* e *hardware* que envolve o conceito de *co-design*. Além disso, a modelagem gráfica facilita a especificação executável do sistema.

Particularmente, para a modelagem estrutural, a notação foi baseada em uma extensão UML para projetos SoC (*UML Extension Profile for SoC*) (UML FOR SOC FORUM, 2004), criado pelo consórcio: *UML for SoC Forum*, que reúne várias empresas de SoC, com o objetivo centrado na utilização da UML em seus projetos (HASEGAWA, 2004). A figura 16, ilustra um exemplo deste modelo, com um módulo pai (bModule) conectado a seu módulo filho (aModule) através de suas portas. A linha cinza representa a conexão entre um módulo com um canal abstrato, enquanto a linha preta representa uma conexão física (canal primitivo).

No caso do projeto, este modelo foi adaptado utilizando uma notação similar ao modelo de colaboração UML, podendo ser utilizado com facilidade em qualquer ferramenta UML. Todas as portas, inclusive as interfaces dos canais abstratos, são representados da mesma forma, conforme pode ser observado na figura 17, que ilustra um exemplo da

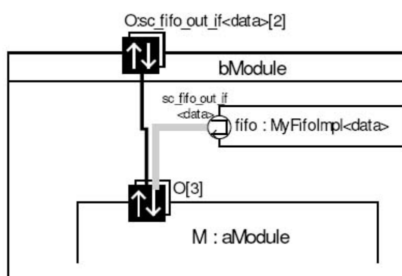

Figura 16: Exemplo de Modelagem Estrutural SoC. fonte: (UML FOR SOC FORUM, 2004)

notação empregada no projeto. Esta mesma notação pode ser utilizada em todos os níveis de abstração.

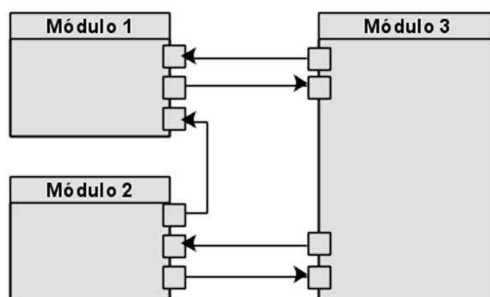

Figura 17: Notação gráfica utilizada para modelagem estrutural do projeto

3.1.2 Prototipação do sistema

Esta etapa compreende a tradução dos modelos UML em um modelo executável. A linguagem utilizada para especificação do projeto é a linguagem SystemC. Os principais conceitos do SystemC para um melhor entendimento do trabalho foram descritos no apêndice A.

Com uma metodologia bem delineada, a especificação em SystemC não deverá adicionar grandes dificuldades, uma vez que apenas refletirá a análise em alto nível da notação UML. De qualquer maneira, cada etapa do projeto possui diferentes necessidades que serão melhor detalhadas durante a descrição das mesmas (seção 3.3).

Particularmente, a etapa de especificação inicial do projeto merece uma atenção especial, visto que é o ponto de partida para os sucessivos refinamentos do projeto. Nesse caso, a descrição dos comportamentos serão baseadas na modelagem Statechart (HAREL, 1987) que, além de ser padronizada pela UML, é comumente utilizada na descrição de sistemas embarcados, como por exemplo em (GERSTLAUER et al., 1999).

3.1.3 Validação e verificação do sistema

A validação e verificação é uma importante atividade no projeto de sistemas e, devido aos elevados custos de produção e desenvolvimento, sua importância tem sido cada vez maior.

De fato, a linguagem SystemC é sobretudo uma linguagem de simulação, possuindo um escalonador (*SystemC scheduler*), capaz de controlar: o tempo, a ordem de execução dos processos, a notificação de eventos e as requisições dos canais de comunicação (Maiores detalhes, vide apêndice A).

O princípio do processo de validação e verificação é garantir que o comportamento do sistema condiz com o esperado. Nesse caso, um modelo de testes (*testbench*) é projetado para fornecer estímulos (*stimuli generators*) ao modelo e capturar as respostas (*response checkers*) ou registrar os eventos característicos do projeto. A figura 18 ilustra a utilização do modelo de testes, onde o projeto sob testes é chamado de DUT (*Device Under Test*).

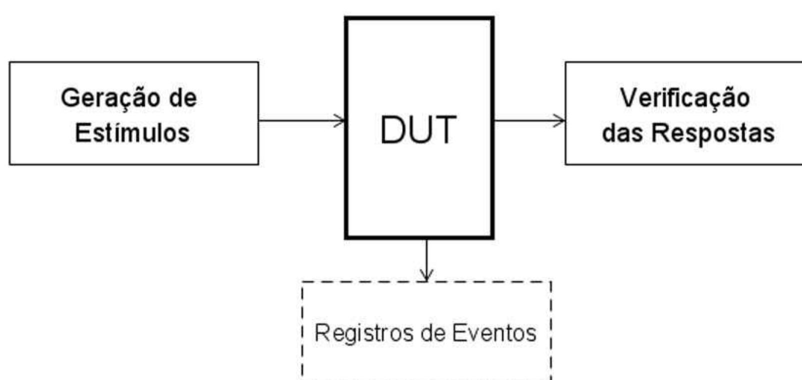

Figura 18: Visão geral dos *Testbenches*

Através deste modelo de verificação é possível se implementar diversas técnicas de testes, como geração de alertas durante a simulação e registros de variações dos sinais no tempo (utilizado para esboçar os formatos de onda em um gráfico de tempo).

A especificação dos modelos de testes, muitas vezes, são tão ou mais trabalhosas quanto à especificação do próprio sistema, tornando importante uma metodologia capaz de facilitar a reutilização dos modelos de testes.

3.2 Fluxo de desenvolvimento

O fluxo de desenvolvimento pode ser visualizado através do gráfico em Y exibido na figura 19. Uma linha adicional tracejada foi adicionada ao gráfico para representar a modelagem UML.

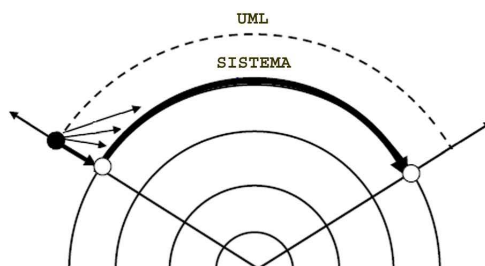

Figura 19: Visão do fluxo do projeto no *Y-chart*

A direção da seta no gráfico da figura 19, representa o fluxo do projeto onde o processo de desenvolvimento será aplicado. Durante este fluxo, os seguintes modelos serão representados:

1. **Modelo Conceitual:** Consiste no primeiro modelo do fluxo, onde a especificação é realizada livre dos detalhes de implementação. O modelo não considera o tempo para realização da computação e da comunicação, assim, a principal preocupação desta etapa reside no validação funcional do projeto.

2. **Modelo de Desempenho:** Este modelo é similar ao modelo conceitual. Porém, os tempos de computação são estimados para prover uma avaliação de desempenho em relação as possíveis decisões de *hardware* e *software* do projeto. As considerações temporais nesse caso são aproximadas (*Aproximate Timed*) e devem permitir uma fácil modificação de critérios de maneira a proporcionar rápidas avaliações de desempenho.

3. **Modelo Arquitetural:** As decisões de projeto realizadas com base no modelo de desempenho, são refletidas neste modelo, onde as características estruturais do sistema já podem ser esboçadas em um modelo de arquitetura intermediária. Dessa forma, esta etapa corresponde a metade do fluxo de projeto (conforme pode ser observado na figura 20), onde os elementos de comunicação e computação têm temporização aproximada (*Aproximate Timed*).

4. **Modelo de Comunicação:** Neste modelo, as decisões de projeto sobre as interfaces de comunicação deverão ser definidas. As interconexões entre os elementos de computação são refinados para o ciclo de relógio (*Cycle timed*) e modelados com exatidão de pinos (*Pin Accurated*). Os elementos de computação permanecem com temporização aproximada e independentes da comunicação, assim, o refinamento realizado nesta etapa não deve provocar modificações nos elementos de computação. Isto é possível através dos princípios de modelagem TLM.

5. **Modelo de Implementação:** Este modelo reflete todas as características estruturais do projeto em nível de sistema, onde tanto os elementos de comunicação quanto de computação são refinados e integrados no ciclo de relógio (*Cycle timed*). A partir deste modelo, os elementos podem ser sintetizados de acordo com a arquitetura alvo selecionada. A figura 20 esboça os próximos passos de síntese realizados para os componentes de *hardware*.

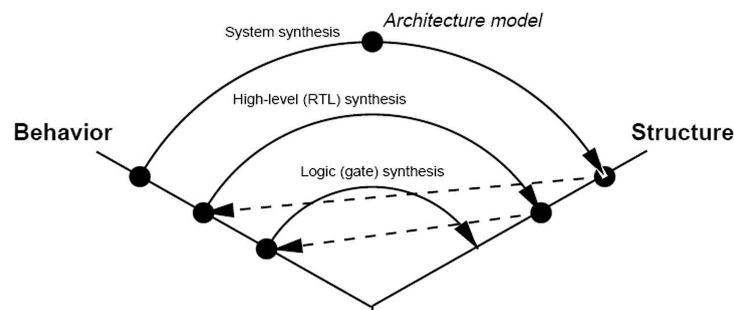

Figura 20: Ciclo de síntese: Sistema, RTL e Lógica. fonte: (GERSTLAUER, 2002)

Enquanto a síntese em nível de sistema especifica a estrutura do projeto em função de elementos de processamento (PEs) tais como, Microcontroladores, Memórias, Barramentos, *Hardware* Customizados e Núcleos de Propriedade Intelectual. O nível RTL, por sua vez, descreve uma microarquitetura estrutural para cada PE, composto por unidades de Controle (UC) e unidades de execução (UE).

No nível RTL, o processo de síntese é conhecido como *high-level synthesis* ou síntese de alto nível, conforme pode ser observado na figura 20, que esboça o ciclo de síntese utilizado em projeto SoC, assim, a síntese lógica corresponde a implementação em portas-lógicas (*gate netlist*), para as descrições dos componentes estruturais em RTL (GERSTLAUER, 2002). Como estes dois últimos processos de síntese, podem ser realizados de maneira automatizada, não serão considerados neste trabalho.

Os modelos definidos podem ser comparados com os modelos definidos nos trabalhos de Cai e Gajski (CAI; GAJSKI, 2003b), conforme comentado no capítulo passado.

As atividades de análise, prototipação e validação (definidas na seção 3.1), são executadas na transição dos modelos definidos no fluxo do projeto. A figura 21 ilustra este processo.

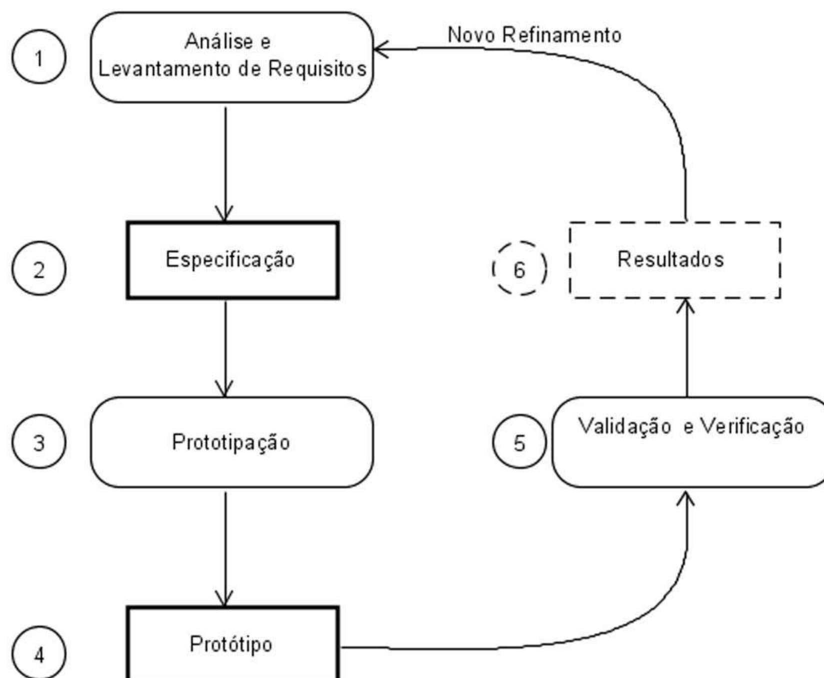

Figura 21: Ciclo de Prototipação

As atividades de Análise e levantamento de requisitos, Prototipação e Validação e Verificação já foram introduzidas na seção 3.1, suas saídas geram, respectivamente, a especificação através de modelos UMLs, protótipos executáveis dos modelos e os resultados dos testes de validação e verificação, ao qual serão consistidos para os novos passos de refinamentos. Cada protótipo devidamente validado deverá condizer com cada um dos níveis de modelagem definidos pela metodologia.

3.3 Processo de desenvolvimento

O modelo completo do processo de desenvolvimento pode ser visualizado na figura 22, onde cada modelo é refinado pelo ciclo de prototipação.

A modelagem UML é o ponto de partida para o refinamento de cada modelo, para a modelagem funcional foi utilizado modelos StateCharts e para a modelagem estrutural foi utilizado uma simplificação da *UML Extension Profile for SoC*, conforme comentado na seção 3.1.1, além disso, cada módulo teve seu modelo estático representado por diagramas de classe.

A seguir cada uma das atividades de refinamento, serão detalhadas, servindo de guia para o processo de desenvolvimento do projeto.

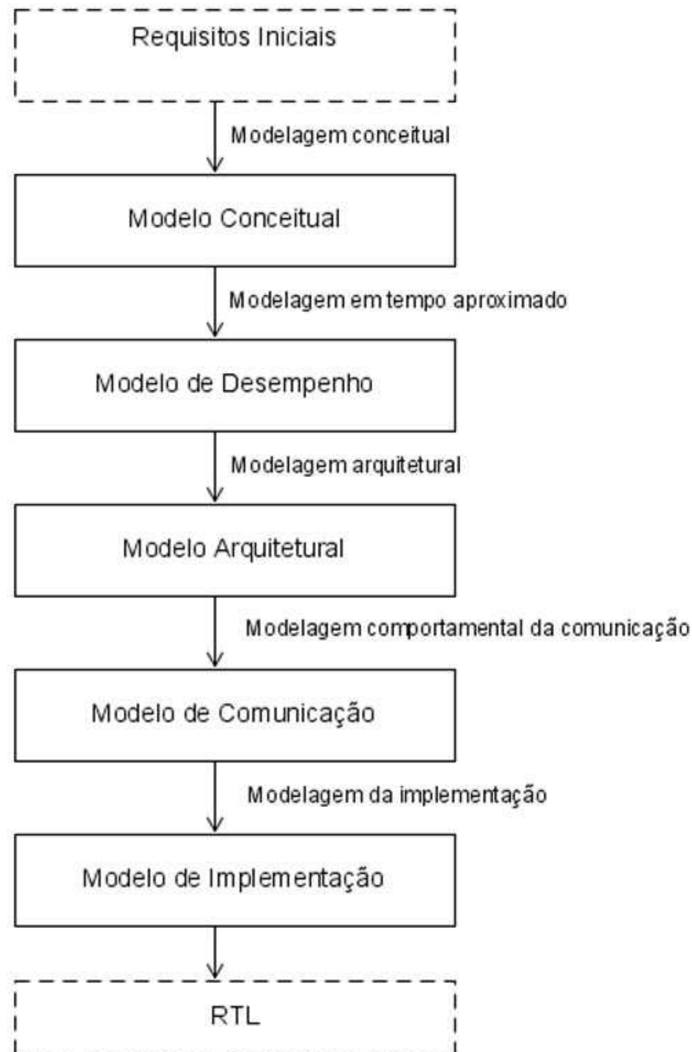

Figura 22: Modelos definidos no processo de desenvolvimento

3.3.1 Modelagem Conceitual

Na modelagem conceitual, somente os aspectos funcionais do projeto devem ser capturados. Os requisitos de tempo, desempenho e de comunicação serão tratados nas etapas futuras do refinamento.

A análise conceitual do projeto pode ser iniciada através de modelos Statecharts. Apesar de ser possível especificar o comportamento do projeto inteiro em um único Statechart, o entendimento funcional do sistema pode ficar comprometido. Assim, cada módulo do projeto deverá modelar um Statechart, a quantidade de módulos necessários é baseada na habilidade do projetista. Porém, muitas vezes o próprio modelo Statechart pode fornecer

algum indício da necessidade de novos módulos ou sub-módulos, tais como a complexidade do Statechart ou o uso de estados paralelos.

Como a funcionalidade dos módulos são representadas por seus processos e considerando que cada módulo pode conter, além de vários processos, outros sub-módulos, a visão estrutural do projeto pode ficar confusa, assim, o projetista deve se esforçar para manter um processo por módulo e evitar utilizar módulos hierárquicos misturados com processos, isto é, só os módulos folhas deverão ter comportamentos.

Contudo, algumas vezes, mais de um processo se faz necessário. Esses casos são justificáveis na necessidade de processos paralelos que auxiliam o comportamento principal do módulo. Assim o Statechart corresponderá, senão o único, ao principal processo do módulo.

Em relação as interconexões dos módulos e sub-módulos e da interface externa de comunicação do sistema, a modelagem é realizada utilizando canais de comunicação primitivos do SystemC, preferencialmente canais *fifo's* (*sc_fifo*), dessa maneira, a sincronização e transferência de dados são simplificadas através dos métodos bloqueáveis read() e write() da fila.

Para tirar proveito dessas vantagens, os processos são modelados utilizando a macro *sc_thread* que permite a utilização de bloqueios, tais como os métodos da *fifo* ou funções de wait(). A macro *sc_thread* possibilita assim recursos de alto nível, permitindo uma sincronização dinâmica, através de eventos de notificação associados à função wait(). É importante ter o cuidado de manter o *thread* em *loop* e com pelo menos alguma forma de bloqueio, caso contrário a simulação não irá avançar no tempo.

Uma simplificação importante nesta etapa é a possibilidade de se definir estruturas de dados na comunicação entre as funcionalidades, isto acrescenta características de alto nível, facilitando como um todo as etapas de análise, prototipação e verificação do sistema. Estas estruturas de dados definidas pelo usuário podem ser utilizadas facilmente no canal primitivo *sc_fifo* do SystemC (SYSTEMC VERIFICATION WORKING GROUP, 2003).

3.3.2 Modelagem de Desempenho

A modelagem de desempenho não deverá alterar os comportamentos da especificação conceitual, ao contrário, ele se baseia no modelo anterior para poder analisar e levantar possíveis dados que motivem a escolha entre elementos de *hardware* e *software*.

Nesse contexto, a análise funcional da especificação permite inferir a freqüência de utilização e a complexidade computacional dos módulos. O cruzamento destas informações fornecem um indício das possibilidades de *hardware/software* do projeto.

Normalmente a decisão é realizada com base em uma biblioteca de componentes, onde o projetista pode comparar os valores obtidos, com as estimativas dos componentes, incluindo, nesse caso, custo e desempenho. A viabilidade da escolha é comprovada através da simulação funcional da especificação, levando em conta os aspectos da arquitetura alvo em relação aos requisitos temporais desejados.

Nesse caso, o processo de refinamento para o modelo de desempenho deverá incluir as atividades de alocação e mapeamento em um processo de refinamento iterativo. Existem diversas estratégias para permitir uma alocação eficiente de elementos tentando maximizar o desempenho e minimizar o custo. A maioria das estratégias consideram o paralelismo existente da especificação inicial para predizer uma quantidade máxima de elementos com a qual o projeto poderá ter seu desempenho elevado (CAI; GAJSKI, 2002).

Os elementos a serem alocados nesta etapa, em geral, são elementos de processamento ou simplesmente PEs (*processing elements*) conforme definido em (GAJSKI et al., 1999). Estes elementos são constituídos por componentes programáveis como DSPs (digital signal processing) ou processadores de propósito geral e componentes não programáveis como ASIC ou componentes IP de funcionalidade específica.

A alocação dos componentes deste nível é utilizada apenas para permitir a realização das estimativas de desempenho. Assim que estes componentes são definidos, a atividade de mapeamento é realizada para decidir quais módulos serão mapeados em cada um dos PEs alocados.

Mesmo que a alocação limite a quantidade de PEs, os módulos ainda devem se manter estruturalmente indepedentes, assim, nenhuma modificação estrutural é realizada nesta etapa. De fato, isto representa uma simplificação do processo de particionamento, justificada pela flexibilidade de exploração das diferentes decisões arquiteturais, dado a iteração inerente desta etapa.

Dessa maneira, ao elevar o nível de abstração com uma metodologia bem definida, as atividades de alocação, mapeamento e particionamento poderão ser beneficiadas através de rápidas estimativas, capazes de propiciar uma iteração eficiente do processo de refinamento. Nesse caso, tanto o projeto quanto suas estimativas poderão ser afinadas à medida que as decisões de projeto vão sendo ajustadas.

Contudo as estimativas de tempo em nível de sistema ainda são um problema em aberto. Nas palavras de Kogel et al. "(...) a anotação inicial representa um educado palpite (...)"(KOGEL et al., 2004) exigindo bastante experiência profissional na realização das estimativas.

Normalmente é preciso um refinamento até os modelos baseados no ciclo de relógio, para que se possa obter estimativas confiáveis. Isto elevaria o tempo de exploração arquitetural a um ponto proibitivo, visto a iteração necessária neste processo. Nesse contexto pode-se sugerir algumas formas de melhorias, conforme resumido abaixo:

1. A arquitetura de *hardware* e *software* são pré-fixadas, de maneira que a estimativa de tempo de um dado comportamento mapeado em *hardware* ou *software* pode ser obtido com facilidade. Mesmo que o projeto não seja mapeado na arquitetura a noção de desempenho e custo são relevantes na escolha do modelo arquitetural (KOGEL et al., 2004);

2. Estimar pesos relacionando os tipos de operação e os possíveis componentes de *hardware* e *software* (CAI, 2004);

3. Refinar parte do comportamento até o nível RTL (se *hardware*) ou Linguagem de Máquina (se *software*) e assim se basear nestes para estimar os demais comportamentos (BAGHDADI et al., 2000).

Uma vez que as estimativas são realizadas, elas são traduzidas no protótipo executável, através da inserção de *waits* no código em SystemC, esta é uma prática bastante utilizada, conhecida como Anotação de Atraso (*Delay Annotation*) ou Anotação de Tempo (*Timing Annotation*) (CAI; GAJSKI, 2003a; POSADAS et al., 2004).

A verificação também desempenha um papel chave no processo de desenvolvimento realizado nesta etapa, visto que é através desta que as decisões arquiteturais serão validadas. Nesse caso, os *testbenches* deverão ser refinados para considerar os aspectos de tempo do projeto, esta técnica é endereçada como co-simulação temporal.

Estruturalmente, os modelos de testes não deverão sofrer modificações, porém as restrições de tempo devem ser anotadas no modelo. Assim, de maneira similar à anotação de atraso, os requisitos são inseridos no gerador de estímulos através de *waits*, que representam os tempos limites de carga que o DUT deverá suportar.

Na verificação dos resultados, as respostas geradas e os principais eventos deverão ser registrados para permitir a verificação dos tempos de respostas. Dependendo da quantidade de informações, é importante projetar a verificação para gerar alertas, em relação aos tempos críticos de projeto.

3.3.3 Modelagem Arquitetural

De maneira geral, a modelagem arquitetural concluirá as decisões arquiteturais realizadas na etapa anterior. O resultado final deste refinamento equivale ao protótipo virtual definido no capítulo 2 (seção 2.3.5).

Desse modo, as atividades de alocação e mapeamento são concluídas através do refinamento estrutural do projeto, associando cada módulo ao seu respectivo PE. Nesse processo, cada elemento deverá originar um módulo independente.

A modelagem hierárquica, com um módulo contendo outros sub-módulos, é uma forma simples e direta de representar os módulos mapeados em um mesmo componente de *hardware*, contudo, esta decisão é deixada a cargo do projetista, dependendo sobretudo das características de cada projeto.

Já os módulos mapeados em um mesmo componente de *software* deverão ser serializados no módulo do PE, isto é realizado movendo somente os processos dos módulos mapeados para os PEs correspondentes. Neste processo, alguns módulos deixarão de existir, bem como outros deverão ser criados, isso incorre na necessidade de substituição das interconexões utilizadas nos antigos módulos pelo barramento de comunicação necessário aos atuais PEs.

Os barramentos são modelados usando canais abstratos do SystemC que, diferente dos canais primitivos utilizados até agora, são implementados pelo desenvolvedor de acordo com as necessidades do projeto.

No caso da modelagem deste etapa, os canais abstratos são projetados para agrupar todas as necessidades de transferência entre os PEs. Assim, a comunicação que nas etapas anteriores eram realizadas usando filas, são agrupadas em um único barramento de comunicação. Este processo, pode ser observado na figura 23, que ilustra um exemplo simplificado, onde dois módulos de softwares são serializados em um único componente de software, destacando o mapeamento de cada elemento do modelo conceitual para o modelo arquitetural.

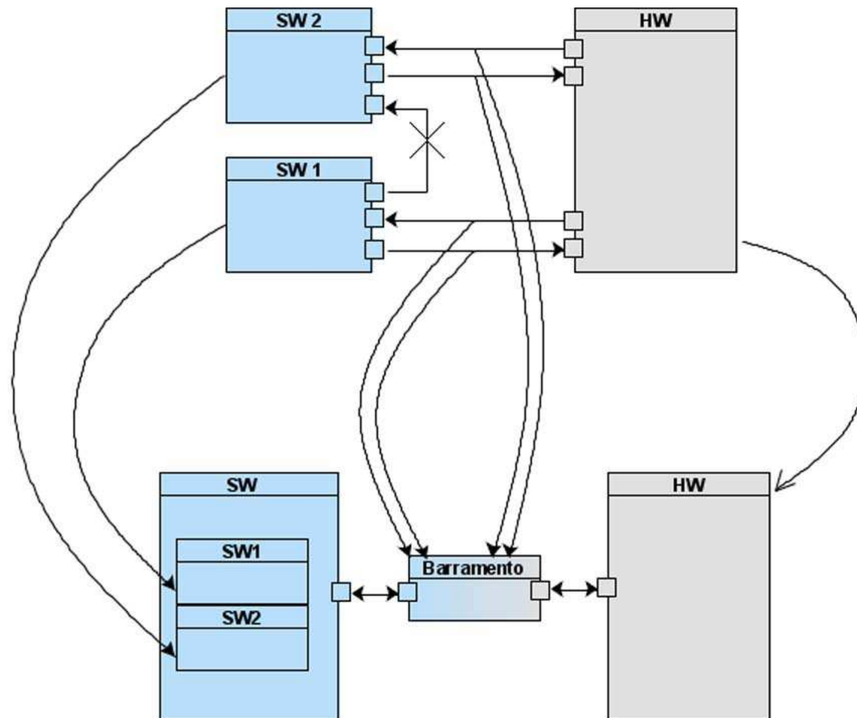

Figura 23: Mapeamento do Modelo Conceitual para o Modelo Arquitetural

O elevado nível de abstração da comunicação deverá ser mantido. Dessa forma, as interfaces de acesso ao canal serão implementadas utilizando as mesmas características de bloqueio utilizadas pelo canal primitivo *sc_fifo*.

3.3.4 Modelagem Comportamental da Comunicação

Na modelagem comportamental da comunicação, os aspectos de comunicação serão refinados para um modelo baseado no ciclo do relógio e com precisão dos sinais de interface.

Visto que os barramentos definidos na etapa anterior fornecem uma visão geral das necessidades de transferência de dados entre os PEs, eles são o ponto de partida para o refinamento desta etapa, que nesse caso se concentra especificamente sobre estes canais abstratos.

A utilização de canais abstratos é uma técnica endereçada pela abordagem TLM (CAI; GAJSKI, 2003a) e sua implementação pode ser realizada em diferentes níveis de abstrações. No caso da modelagem realizada nesta etapa os barramentos são refinados para o nível de abstração comportamental, onde os detalhes de comunicação são definidos e implementados.

De fato, este detalhamento necessita de decisões de projeto, tais como arbitração, sincronismo, *handshakes*, protocolos de comunicação, que eram até agora evitadas em virtude das técnicas *co-design*. No entanto, considerando que nesta etapa o projetista já possui um protótipo virtual estável, estas decisões de projeto podem ser realizadas com maior segurança. Normalmente é importante a adoção de algum padrão de interface bem aceito, tal como o padrão Wishbone (OPENCORES, 2002).

Durante o processo de refinamento do canal, nenhuma modificação deverá ser realizada nos PEs. Nesse caso, as interfaces de comunicação entre os PEs e seus respectivos barramentos são mantidas sem alteração, isto propicia que os elementos de comunicação e computação tenha um refinamento independente, a motivação para isso é baseada nos conceitos de ortogonalização definidos em (KEUTZER et al., 2000) e nos princípios de modelagem TLM (CAI; GAJSKI, 2003a).

Assim, deve-se ter a preocupação de não alterar as interfaces de acesso ao barramento. Isto pode ser realizado utilizando módulos adaptadores, que substituem a implementação do canal. Neste caso, as interfaces do barramento são mantidas nos adaptadores, porém, a comunicação entre os adaptadores é refinada para o ciclo de relógio, conforme pode ser visualizado na figura 24, que ilustra o refinamento do exemplo anterior (figura 23) para o modelo de comunicação.

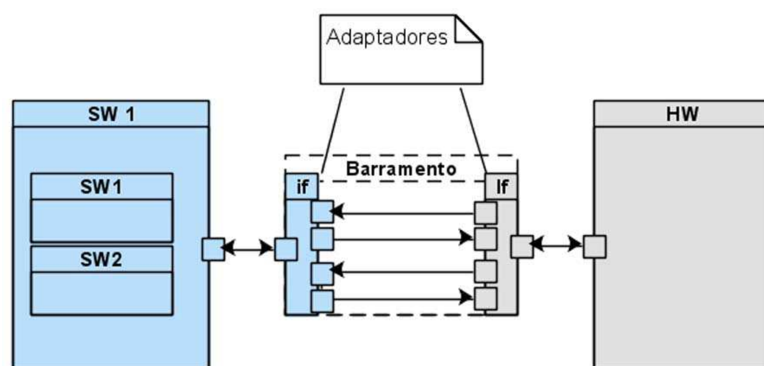

Figura 24: Exemplo do Modelo de Comunicação

Um dos benefícios, inclui a reutilização dos mesmos modelos de testes, que podem ser utilizados durante o refinamento desta etapa. Contudo, ao final do processo convém atualizar o *testbench* para registrar as variações dos sinais em baixo nível e assim permitir uma visualização do formato de onda em um gráfico de tempo.

3.3.5 Modelagem da Implementação

Nesta etapa, os elementos de computação e comunicação são finalmente integrados para serem refinados em conjunto num modelo de implementação em que todos os elementos estruturais são independentes, completos e bem definidos, conforme pode ser observado na figura 25, que ilustra o modelo do exemplo anterior (figura 24) refinado para o modelo de implementação.

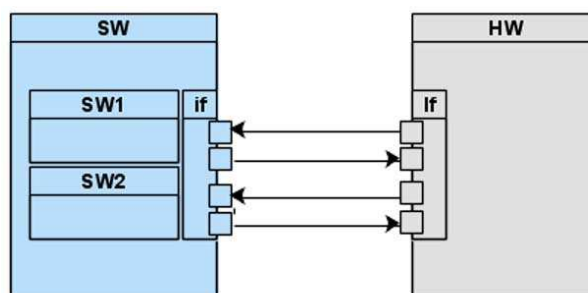

Figura 25: Exemplo do Modelo de Implementação

Nos próximos passos de síntese, cada um dos PEs modelados podem ser internamente tratados com o objetivo de obter um modelo sintetízavel em RTL. Este último passo foge ao escopo do trabalho, uma vez que, para se obter um modelo sintetizável é necessário obedecer tanto as necessidades da arquitetura alvo como das ferramentas de síntese utilizadas na síntese de alto nível e nas posteriores etapas de síntese do projeto, conforme a seqüencia do ciclo de síntese necessária ao projeto (vide seção 3.2, figura 20).

De uma maneira geral, os componentes de *hardware* em nível comportamental podem ser obtidos utilizando as primitivas de *hardware* do SystemC (GROTKER, 2002), enquanto os componentes de *software* são dependentes dos elementos de *hardware* ao qual estarão embarcados.

Normalmente os componentes de *software* são embarcados em um microcontrolador ou DSP, neste caso, existem duas formas de manter a unificação *hardware/software*:

1. Obter um *IP-Core* do microcontrolador ou DSP selecionado, ao qual o código em SystemC poderia ser mapeado;

2. Implementar o *hardware* do controlador baseado nas funcionalidades ou necessidades do projeto.

A primeira alternativa é uma solução conveniente para a maioria dos projetos, porém tem se observado várias pesquisas atuando na automatização da segunda opção. Em

ambos os casos o código em SystemC dos módulos de *software* já possuem toda a descrição que deverá ser embarcada, compreendendo a maior parte da especificação de *software*.

Observe que, na modelagem clássica de sistemas embarcados, muitas vezes o projeto de *software* só era iniciado quando já se tinha um modelo de *hardware* disponível em RTL, implicando em um atraso significativo do projeto, sem considerar a possibilidade do projeto do *software* acabar sendo inviabilizado com a plataforma de *hardware* especificada.

Existe uma tendência de comercialização de núcleos IP (HAVERINEN et al., 2002) em nível de sistema, de maneira que cada um dos módulos independentes poderiam ser separadamente comercializados ou até mesmo em conjunto. No caso USB, por exemplo, um núcleo padronizado, chamado de UTM (USB Transceiver Macrocell), atuando na sinalização em baixo nível, pode ser encontrado comerciamente como um *IP-Core* distinto das funcionalidades USB. No caso de estudo USB, definido no próximo capítulo, será projetado o núcleo da funcionalidade USB, ao qual deverá prover uma interface em comum com o UTM, conhecida como UTMI (*USB Transceiver Macrocell Interface)* (INTEL, 2001).

Dependendo da complexidade do projeto e como o mesmo foi projetado, esta etapa pode ocasionar severas modificações no modelo, entre as quais, todas as estruturas de dados que normalmente são definidas na modelagem conceitual (seção 3.3.1), se ainda existirem, devem ser agora traduzidas para o nível de sinalização, isto implica em adicionais modificações na comunicação.

As características de bloqueio, utilizadas nos canais primitivos *sc_fifo*, que foram eventualmente mantidas no refinamento da comunicação e na criação dos canais abstratos, dependendo das características do projeto, precisam ser alteradas, isto porque o conceito de bloqueio é uma simplificação que muitas vezes não reflete a utilização real do módulo no nível de implementação.

Não existe uma solução única para estes problemas, visto que dependem diretamente do projeto em questão, de qualquer maneira, as várias etapas de refinamento anteriores permitem uma visualização direta destes e outros problemas, propiciando ao desenvolvedor uma forma mais segura de decidir sobre a melhor solução. Isto será melhor visualizado no estudo de caso do próximo capítulo.

3.4 Considerações Finais

Este capítulo ilustrou a metodologia proposta no trabalho servindo como modelo de guia para o processo de desenvolvimento definido.

A estratégia adotada para a especificação do processo de desenvolvimento foi baseada nas características de modelagem do SystemC e nas dificuldades encontradas na modelagem de uma *interface* USB. Contudo, a metodologia apresentada neste capítulo é suficientemente abrangente e baseada nos recentes avanços metodológicos, podendo ser adotada na especificação de diversos outros tipos de projetos.

Duas referências que demonstram a tendência e a utilização de modernos conceitos de modelagem são os trabalhos envolvendo a metodologia SpecC (DÖMER, 2002) e a referência (GROTKER, 2002), ambos consideram o nível de sistema e os princípios de modelagem TLM como uma maneira de gerenciar a complexidade no projeto de sistemas embarcados e SoC.

No capítulo seguinte, a modelagem da *interface* USB será apresentada para validação e demonstração da metodologia definida neste capítulo.

4 Aplicação da Metodologia Co-design

O processo de desenvolvimento definido no último capítulo será aplicado no estudo de caso USB, em conformidade a metodologia *co-design* especificada. Nesse caso, a maior importância deste capítulo consiste na prova de conceito do processo de desenvolvimento definido.

4.1 Introdução ao Problema

De maneira simplificada, o barramento USB pode ser visto como um equipamento USB conectado a um *Host* USB, conforme mostrado na figura 26.

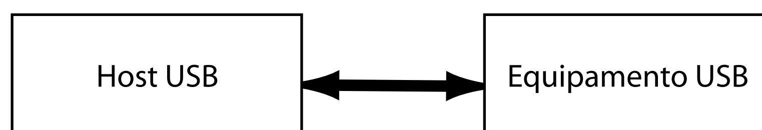

Figura 26: Visão simplificada do comportamento USB

A estrutura de comunicação é mestre escravo. O *host* é o mestre da comunicação, sendo responsável pelo controle do fluxo de dados no barramento, requisitando serviços aos equipamentos e determinando quando e qual equipamento pode transmitir seus dados em cada intervalo de tempo.

O equipamento USB é o escravo da comunicação e é o alvo do estudo de caso desenvolvido neste capítulo. O *host* é modelado para os testes, gerando requisições de acordo com a especificação da interface USB, permitindo validar o desenvolvimento do projeto.

O *host* USB pode ser visualizado como um computador, permitindo que vários periféricos e equipamentos USB sejam conectados, conforme pode ser visualizado na figura 27, que exibe uma típica configuração da arquitetura USB, com diversos equipamentos USB: Monitor de Vídeo, Caixas de Som, Teclado, Mouse, Planilha Digital, Telefone e

Fone de ouvido. Cada um destes equipamentos, possui alguma funcionalidade específica, e são configuradas pelo *host*, num processo conhecido como enumeração (*enumeration*) (USB WORK GROUP, 2000).

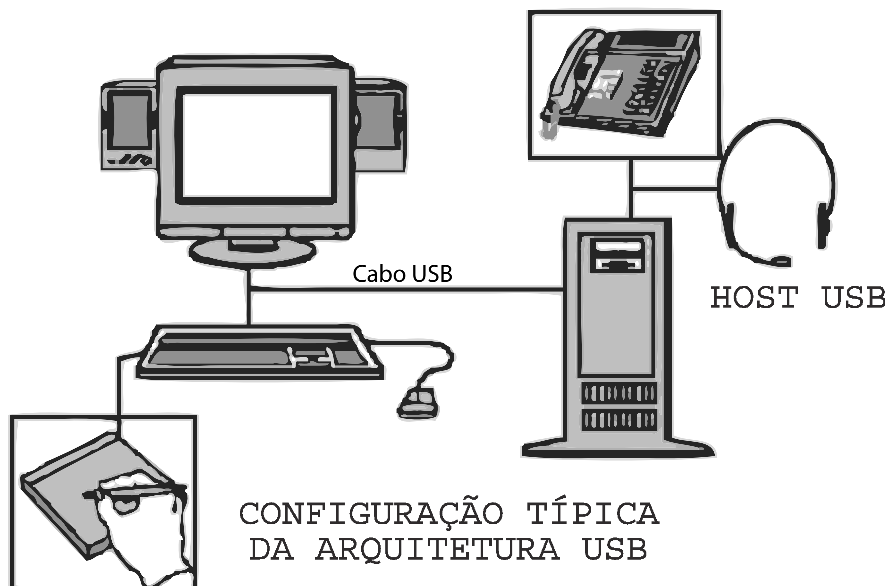

Figura 27: Configuração Típica da Arquitetura USB. fonte: (USB WORK GROUP, 2000)

Para completar a enumeração o *host* realiza uma seqüencia de requisições ao equipamento. A tabela 2 exibe as requisições padrões, ao qual o equipamento deverá responder (USB WORK GROUP, 2000). O equipamento, por sua vez, deverá aceitar e responder a estas requisições, fornecendo as informações necessárias para sua correta configuração. Ao final do processo, um endereço de comunicação é atribuído e o equipamento é configurado para a função necessária.

Toda a comunicação incide sobre os *endpoints* do equipamento USB. "Um *endpoint* é uma porção unicamente identificável do equipamento USB, correspondendo ao terminador do fluxo de comunicação entre o *host* e o *device*" (USB WORK GROUP, 2000).

Cada *endpoint* tem características que determinam o tipo de transferência necessário. A combinação do endereço do equipamento (atribuído em tempo de conexão), o número do *endpoint* (atribuído em tempo de desenvolvimento) e a direção do fluxo de dados do *endpoint* (determinado pelo equipamento) permitem a identificação única de um *endpoint*.

Estas e diversas outras características são armazenadas em estruturas de dados conhecidas como descritores (*descriptors*). Isto permite diferenciar, dependendo das necessidades e da funcionalidade desejada, os requerimentos de comunicação e o fluxo de dados, entre o *host* e os *endpoints* no equipamento USB. Especificamente, a interface USB define 4 tipos de transferências:

Código	Requisição	Descrição Resumida
0	GET_STATUS	Retorna o status do equipamento, interface ou *endpoint*.
1	CLEAR_FEATURE	Apaga ou desabilita uma característica específica
3	SET_FEATURE	Seta ou habilita uma característica específica
5	SET_ADDRESS	Seta o endereço para os futuros acessos ao equipamento
6	GET_DESCRIPTOR	Retorna um *descriptor* específico
7	SET_DESCRIPTOR	Pode ser utilizado para atualizar ou adicionar um novo *descriptor*
8	GET_CONFIGURATION	Retorna a configuração corrente do equipamento
9	SET_CONFIGURATION	Seleciona a configuração desejada para o equipamento
10	GET_INTERFACE	Retorna a configuração alternativa para uma interface específica
11	SET_INTERFACE	Seleciona a configuração alternativa para uma interface específica
12	SYNCH_FRAME	Seta e reporta o frame de sincronização do *endpoint*

Tabela 2: Requisições Padrão

- Transferência de Controle (*Control Transfer*);
- Transferência de Volume (*Bulk Transfer*);
- Transferência de Interrupção (*Interrupt Transfer*);
- Transferência Isócrona (*Isochrounous Transfer*).

Maiores detalhes sobre a utilização e características destas transferências podem ser obtidas no documento de especificação USB (USB WORK GROUP, 2000). De fato, todo equipamento USB deverá obrigatoriamente definir um *endpoint* com número 0 (*endpoint0*), que implementa a transferência de controle, necessária para manipulação e configuração do equipamento através do *host*.

4.2 Modelagem Conceitual Funcional

Dada a complexidade do projeto descrito anteriormente, o mesmo será, a seguir, especificado num elevado nível de abstração, onde a preocupação é definir o funcionamento do sistema eliminando detalhes como a comunicação dos elementos e aspectos da arquitetura alvo.

4.2.1 Análise de Requisitos

Apesar da comunicação entre *host* e equipamento ser realizada através de um conjunto de regras que definem o protocolo de comunicação USB (USB WORK GROUP, 2000), do ponto de vista funcional, tem-se um *software* cliente no *host* se comunicando com a aplicação do equipamento, conforme pode ser observado na figura 28.

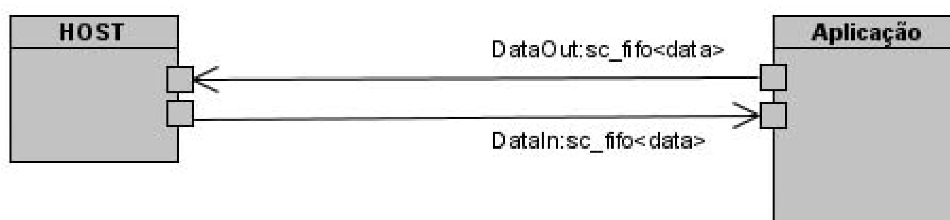

Figura 28: Modelo Conceitual

Partindo deste modelo, o protocolo USB pode ser visto como um facilitador da comunicação entre *host* e aplicação, de maneira que seus detalhes serão considerados somente em etapas posteriores, ou seja, nesse ponto é somente importante entender o comportamento esperado da aplicação. Nesse caso, foram definidos os seguintes serviços para a aplicação:

SetMemo: Armazena os dados enviados ao equipamento no endereço especificado.

GetMemo: Recupera os dados armazenados no ou a partir do endereço espcificado.

SetName: Registra o nome do serviço ou do usuário do equipamento.

GetName: Recupera os dados armazenados pelo serviço anterior.

Apesar da simplicidade da funcionalidade, estes serviços são suficientes para validar o processo de desenvolvimento e entender os principais conceitos.

A figura 29 mostra o diagrama de estados para o módulo de aplicação e a figura 30 o diagrama de classes.

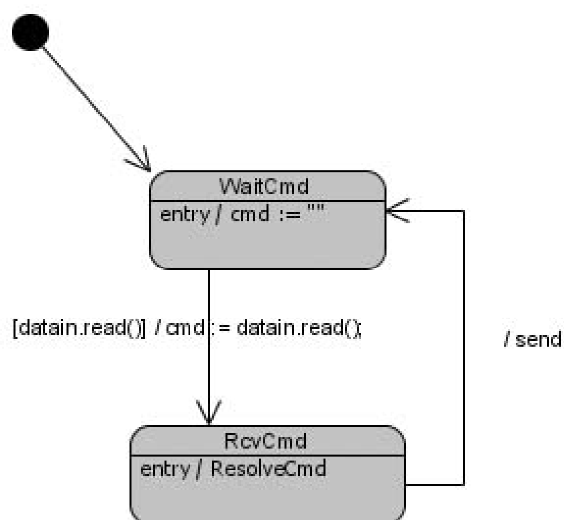

Figura 29: Diagrama de Estados - Modelo conceitual

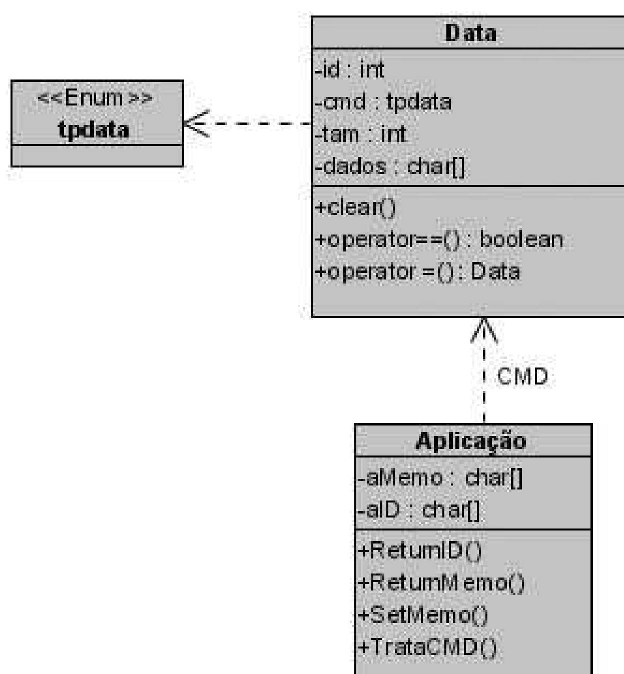

Figura 30: Diagrama de Classes - Modelo conceitual

4.2.2 Prototipação

Pela simplicidade do modelo, o mesmo pode ser facilmente convertido em uma especificação executável. Observando o diagrama de classes da figura 30, percebe-se que o módulo aplicação representado como uma classe possui uma relação de dependência com a classe Data, que define a estrutura dos comandos que serão utilizados pela aplicação, esta característica permite uma facilidade de alto-nível para a prototipação e verificação do sistema (SYSTEMC VERIFICATION WORKING GROUP, 2003).

A título de exemplificação, a listagem 4.1 corresponde a declaração em SystemC do módulo de aplicação.

Listagem 4.1: Especificação do Modelo Conceitual

```
SC_MODULE(aplicacao) {
    sc_port<sc_fifo_in_if<data> >  iData;
    sc_port<sc_fifo_out_if<data> > oData;

    char     aMemo[100];
    char     aId[10];

    void mWatchCmd(void);
    void mSetMemo(data& cmd);
    void mGetMemo(data& cmd);
    void mSetName(data& cmd);
    void mGetName(data& cmd);

    SC_CTOR(aplicacao) {
        strcpy(aId,"USB-UFES");
        SC_THREAD(mWatchCmd);
    }
};
```

A comunicação entre *host* e equipamento é realizada utilizando o canal abstrato *fifo* do SystemC, em conformidade a metodologia, onde, no nível conceitual, os canais de comunicação são preferencialmente canais primitivos ou abstratos que serão refinados em etapas consecutivas.

4.2.3 Validação e Verificação

Para validar o módulo de aplicação, o módulo *host* foi implementado para solicitar uma sequência de serviços ao módulo de aplicação. Cada solicitação sentida na aplicação foi registrada e sua resposta capturada no *host*.

A listagem 4.2 exibe a construção do *testbench*, pode-se observar a criação do canal e a instanciação dos módulos do *host* e aplicação.

Listagem 4.2: *TestBench* do Modelo Conceitual

```
int sc_main(int argc, char *argv[]) {
    sc_fifo<data> fifo_data1(1);
    sc_fifo<data> fifo_data2(1);

    host i_host("HST");
    aplicacao i_aplicacao("APL");

    i_host.DataIn(fifo_data1);
    i_host.DataOut(fifo_data2);
    i_aplicacao.iData(fifo_data2);
    i_aplicacao.oData(fifo_data1);

    sc_start();
    return 0;
}
```

Inicialmente, somente as validações básicas são realizadas. Posteriormente, cada vez mais características vão sendo incorporadas ao módulo de testes, até que todas as características pertinentes do protótipo possam ser validadas. A figura 31 exemplifica uma parte inicial da simulação realizada pelo *testbench*, ilustrando as transações recepcionadas no equipamento (módulo aplicação) e no *host*.

Os dados exibidos na tela de simulação da figura 31 são, respectivamente: o módulo ("hst-rcv" para o *host* e "dev-rcv" para a aplicação), numeração sequencial da transação, quantidade de bytes e o código da requisição, seguido pela identificação textual da requisição, os *bytes* recebidos (em formato hexadecimal, separando cada *byte* por uma trena "|") e sua codificação ASCII correspondente.

A preocupação é validar a funcionalidade não sendo necessário considerar requisitos temporais, visto que a própria comunicação ainda precisa ser refinada.

```
              SystemC 2.0.1 --- Apr  8 2005 22:58:16
        Copyright (c) 1996-2002 by all Contributors
                     ALL RIGHTS RESERVED
   dev-rcv:0:3:1(setMemo)|5|0|6a(j)
   dev-rcv:1:0:4(getName)()
   hst-rcv:1:8:4(getName)|55|53|42|2d|55|46|45|53(USB-UFES)
   dev-rcv:2:3:2(getMemo)|5|0|1()
   hst-rcv:2:1:2(getMemo)|6a(j)
   dev-rcv:3:4:1(setMemo)|6|0|74|73(ts)
   dev-rcv:4:3:2(getMemo)|5|0|3()
   hst-rcv:4:3:2(getMemo)|6a|74|73(jts)
...
```

Figura 31: Validação do Modelo Conceitual

4.3 Modelagem Conceitual de Comunicação

O primeiro refinamento do projeto corresponde ao tratamento do canal de comunicação que, nesse caso, define a implementação do protocolo USB. O objetivo é definir conceitualmente a interface de comunicação necessária para a implementação da aplicação, aproximando a especificação funcional do modelo final.

Dessa maneira, esta etapa define conceitualmente o protocolo USB, sendo assim, a etapa mais complexa do nível conceitual. Para tanto o modelo de comunicação foi realizado em dois passos de refinamento conforme exibe as figuras 32 e 33.

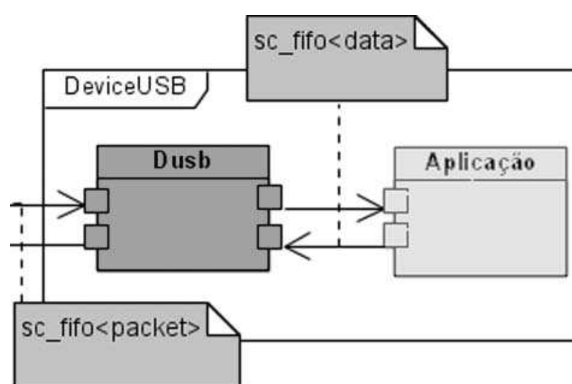

Figura 32: Modelo Conceitual da Comunicação (1° Refinamento)

O primeiro refinamento acrescenta o módulo Dusb que define a lógica do protocolo, enquanto o segundo refinamento adiciona o módulo Dsie que se encarrega de reconhecer os pacotes e verificar o CRC (*Cyclic Redundancy Check*).

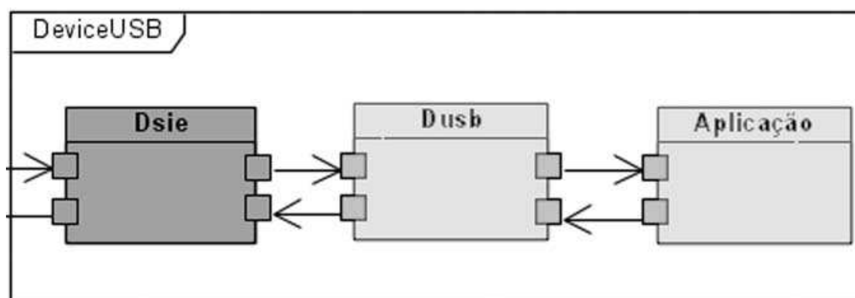

Figura 33: Modelo Conceitual da Comunicação (2° Refinamento)

Ao quebrar o processo de desenvolvimento em dois passos de refinamento, com cada um adicionando particularidades ao projeto, faz com que o mesmo seja conduzido com maior facilidade, sendo claramente uma boa prática de projeto.

Inicialmente foi especificado somente o módulo Dusb. Depois que o protótipo foi devidamente validado, foi necessário refinar o modelo, uma vez que o primeiro refinamento não continha todos os aspectos conceituais necessários. O módulo Dsie dessa maneira possui um escopo diferenciado atuando nos *bytes* enviados ao invés de pacotes, assim ele interpreta os dados recebidos do *host*, checa o CRC e monta o pacote que será repassado para o módulo Dusb, deixando a cargo do Dusb questões como endereçamento, consistência e ordem das transações.

4.3.1 Análise e Levantamento de Requisitos

O módulo Dusb controlará os serviços requisitados à aplicação, impondo a noção de transação USB, interpretando os pacotes, executando e respondendo as requisições solicitadas.

Uma transação completa USB equivale aos pacotes *Token*, *Data* e *Handshake* sendo que, dependendo da transação realizada, o pacote *Data* não é obrigatório. A transação é sempre iniciada pelo *host* através da transmissão de um *token*, indicando assim o tipo de transação: (USB WORK GROUP, 2000)

- SOF - *Start of Frame*
- OUT - Transmissão de dados do *host* para o equipamento
- IN - Transmissão de dados do equipamento para o *host*
- SETUP - Transmissão de comandos do *host* para o equipamento

Além do *endpoint0*, que é responsável pela configuração do equipamento, será utilizado o *endpoint* de número 1 (*endpoint1*) para a execução dos serviços definidos pelo módulo aplicação. Considerando que as requisições da aplicação se comportam como comandos, o tipo de transferência especificado foi a transferência de controle. A figura 34 exibe a visão estrutural do modelo.

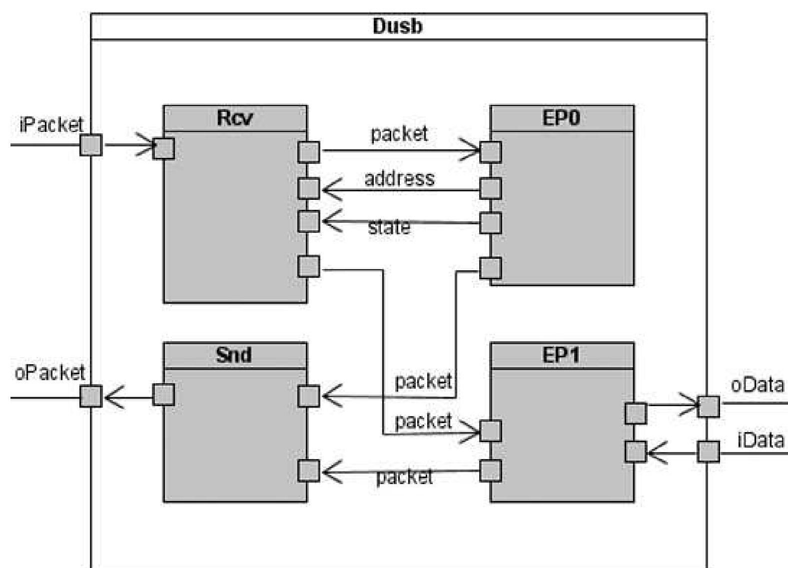

Figura 34: Modelo Lógico USB (Dusb)

Os sub-módulos RCV, SND, EP0, EP1 da figura 34 modelam, respectivamente, a recepção de pacotes, o envio de pacotes, os serviços do *endpoint0* e o tratamento do *endpoint1*. Cada um destes sub-módulos tiveram seus comportamentos modelados através de Statecharts. As figuras 35 e 36 definem os comportamentos dos módulos Rcv e EP0, respectivamente.

O comportamento do EP1 é similar ao EP0, uma vez que os dois utilizam o mesmo tipo de transferência, a divergência entre os dois é funcional, onde o EP0 deverá atender as requisições padrão, conforme a tabela 2, enquanto o EP1 é responsável por traduzir as requisições definidas a nível de projeto (*vendor-request*) (USB WORK GROUP, 2000) para o módulo de aplicação.

Para facilitar o compreensão e a estrutura do texto, somente os modelos chaves para o entendimento do processo estão sendo exibidos.

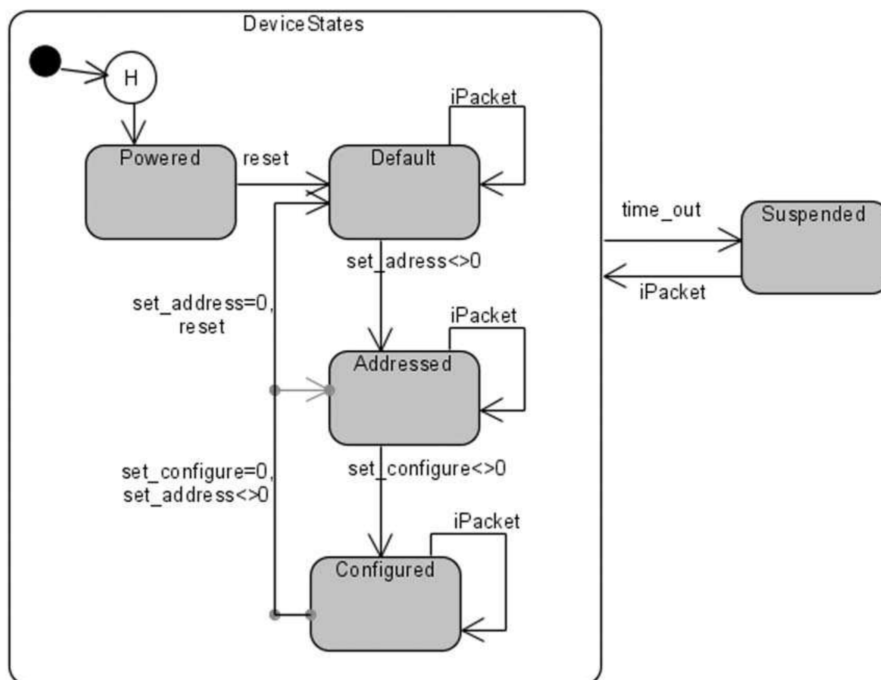

Figura 35: *StateChart* - Comportamento do Equipamento

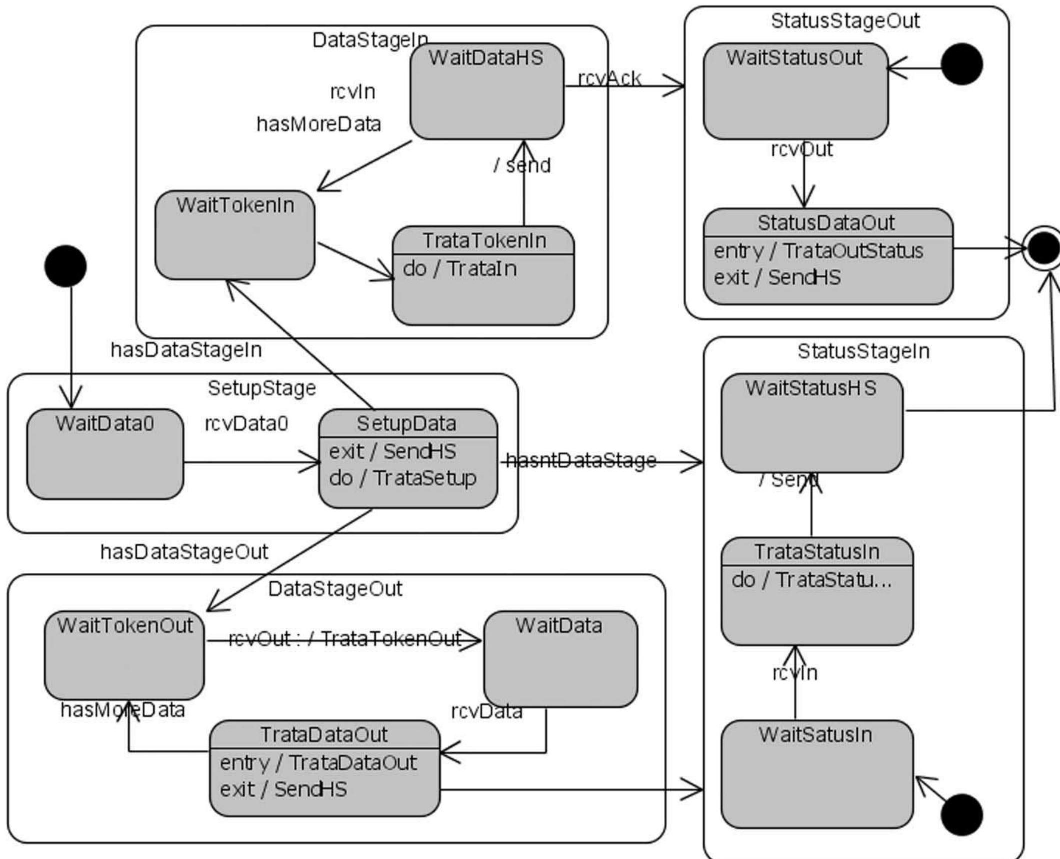

Figura 36: *StateChart* - Comportamento do *endpoint0*

4.3.2 Prototipação

Seguindo a mesma linha da prototipação conceitual funcional da seção anterior, os modelos foram traduzidos para SystemC fornecendo uma especificação executável em nível de sistema.

O tipo de dados *packet* utilizado pelo módulo Dusb corresponde a uma classe utilizada para a composição dos pacotes, conforme exibido na figura 37. Sua estrutura é bem similar a classe Data utilizada na prototipação funcional.

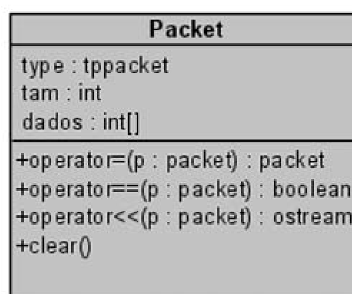

Figura 37: Classe Packet

O pacote é identificado pelo atributo type, que corresponde a uma codificação enumerada de todos os possíveis PIDs (*Packet Identifications*). O restante dos dados que compõem o pacote são inseridos no atributo dados da classe. A tabela 3 exibe os formatos para os pacotes tratados no projeto. (USB WORK GROUP, 2000)

Tokens =	PID	ADDR	ENDP	CRC5	
Data =	PID	DADO N	...	DADO 0	CRC16
Handshake =	PID				

Tabela 3: Formato dos Pacotes

Nesse caso é importante observar que o campo *CRC* não precisa ser incluído no *packet*, visto que o mesmo é tratado e verificado diretamente pelo módulo Dsie do segundo refinamento. Os restantes dos campos com exceção do *PID* são colocados no atributo dados do *packet*.

A listagem 4.3 exibe o protótipo executável em SystemC do módulo Dusb, referente ao primeiro refinamento do projeto de comunicação conceitual.

Listagem 4.3: Protótipo do Módulo Dusb

```
SC_MODULE(Dusb) {
    sc_port<sc_fifo_in_if<packet> >  iPacket;
    sc_port<sc_fifo_out_if<packet> > oPacket;
    sc_port<sc_fifo_out_if<data> > oData;
    sc_port<sc_fifo_in_if<data> > iData;

    //canais de comunicação
    sc_signal<int> address;
    sc_signal<bool> SetupConfig;
    sc_signal<bool> SetupDone;
    sc_fifo<packet> ep0PacketA;
    sc_fifo<packet> ep0PacketB;
    sc_fifo<packet> ep1PacketA;
    sc_fifo<packet> ep1PacketB;
    sc_fifo<int> ch;

    //instancias dos módulos
    usbRcv      i_UsbRcv;
    usbSnd      i_UsbSnd;
    ep0         i_Ep0;
    ep1         i_Ep1;

    SC_CTOR(usb):i_UsbRcv("iRCV"),i_Ep0("iEP0"),
    i_Ep1("iEP1"),i_UsbSnd("iSND")
    {
        i_UsbRcv.iPacket(iPacket);
        i_UsbRcv.oEp0Packet(ep0PacketA);
        i_UsbRcv.oEp1Packet(ep1PacketA);
        i_UsbRcv.address(address);
        i_UsbRcv.iDeviceState(ch);

        i_UsbSnd.oPacket(oPacket);
        i_UsbSnd.iEp0Packet(ep0PacketB);
        i_UsbSnd.iEp1Packet(ep1PacketB);

        i_Ep0.iPacket(ep0PacketA);
        i_Ep0.oPacket(ep0PacketB);
        i_Ep0.address(address);
        i_Ep0.oDeviceState(ch);

        i_Ep1.iPacket(ep1PacketA);
        i_Ep1.oPacket(ep1PacketB);
        i_Ep1.iData(iData);
        i_Ep1.oData(oData);
    }
};
```

4.3.3 Validação e Verificação

Para realizar a validação, o modelo de testes do *host* também foi refinado, possibilitando a construção de um ambiente de simulação modular, de maneira que o módulo de testes definido no nível funcional é reutilizado sem nenhuma alteração. Esta é uma prática altamente recomendada de maneira a maximizar a utilização dos módulos de testes (HAVERINEN et al., 2002).

As figuras 38 e 39 mostram os modelo de *testbenches* utilizados no primeiro e segundo refinamento respectivamente.

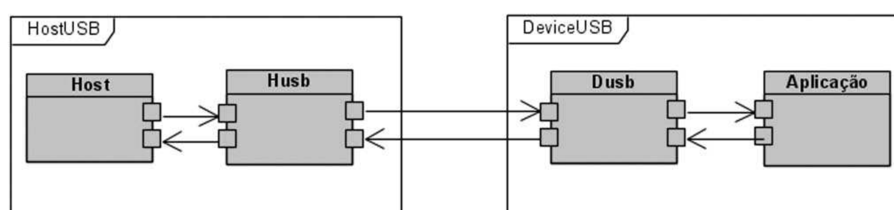

Figura 38: Modelo de Testes - 1° Refinamento

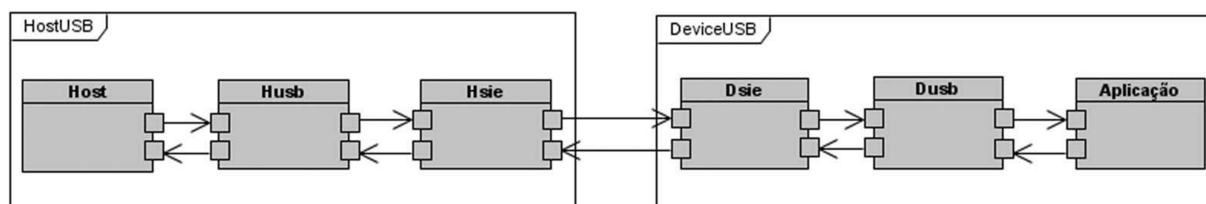

Figura 39: Modelo de Testes - 2° Refinamento

Certamente, um *testbench* mais completo envolveria especificar o modelo de testes com o rigor de um *host* real, que traria um acréscimo elevado no tempo de desenvolvimento. Como no caso do projeto, o foco é na demonstração metodológica, o modelo de testes implementado possui um escopo limitado à demonstração dos aspectos pertinentes da especificação USB.

De qualquer maneira, é importante observar a importância de uma validação mais rigorosa, visto que as empresas tem gasto cada vez mais tempo na modelagem dos testes, por isso a necessidade de reaproveitar os módulos de testes durante todo o ciclo do projeto.

A figura 40 ilustra uma parte da simulação do *testbench* referente ao primeiro refinamento e baseado na validação iniciada na etapa anterior (figura 31). Podem-se observar o início e o fim do processo de *enumeração* nas linhas 6 e 49 respectivamente. (As requisições intermediárias do processo de enumeração foram omitidas)

```
2               SystemC 2.0.1 --- Apr  8 2005 22:58:16
3          Copyright (c) 1996-2002 by all Contributors
4                      ALL RIGHTS RESERVED
5  Dusb-rcv:10(rst)
6  GET DEVICE DESCIPTOR
7  -SETUP STAGE
8    Dusb-rcv:2(stp)|0|0
9    Dusb-rcv:5(dt0)|40|6|1|0|0|0|0|0(@)
10   Husb-rcv:7(ack)
11 -DATA STAGE
12   Dusb-rcv:3(in )|0|0
13   Husb-rcv:6(dt1)|12|1|0|2|ff|0|ff|40|ff|ff|1|0|1|0|1|2|3|1(@)
14   Dusb-rcv:7(ack)
15 -STATUS STAGE
16   Dusb-rcv:4(out)
17   Dusb-rcv:6(dt1)()
18   Husb-rcv:7(ack)
19 SET ADDRESS
20 ...
41 SET CONFIGURATION
42 -SETUP STAGE
43   Dusb-rcv:2(stp)|1|0
44   Dusb-rcv:5(dt0)|0|9|1|0|0|0|0|0()
45   Husb-rcv:7(ack)
46 -STATUS STAGE
47   Dusb-rcv:3(in )|1|0
48   Husb-rcv:5(dt0)()
49   Dusb-rcv:7(ack)
50 SET MEMO
51 -SETUP STAGE
52   Dusb-rcv:2(stp)|1|1
53   Dusb-rcv:5(dt0)|40|1|5|0|0|6a|0|1(@j)
54    apl-rcv:0:3:1(setMemo)|5|0|6a(j)
55   Husb-rcv:7(ack)
56 -STATUS STAGE
57   Dusb-rcv:3(in )|1|1
58   Husb-rcv:5(dt0)()
59   Dusb-rcv:7(ack)
60 GET NAME
61 -SETUP STAGE
62   Dusb-rcv:2(stp)|1|1
63   Dusb-rcv:5(dt0)|40|4|0|0|1|0|0|0(@)
64    apl-rcv:1:0:4(getName)()
65   Husb-rcv:7(ack)
66 -DATA STAGE
67   Dusb-rcv:3(in )|1|1
68   Husb-rcv:6(dt1)|4|1|8|55|53|42|2d|55|46|45|53(USB-UFES)
69    hst-rcv:1:8:4(getName)|55|53|42|2d|55|46|45|53(USB-UFES)
70   Dusb-rcv:7(ack)
71 -STATUS STAGE
72   Dusb-rcv:4(out)|1|1
73   Dusb-rcv:6(dt1)()
74   Husb-rcv:7(ack)
75 ...
```

Figura 40: Validação do Modelo Conceitual - 1° Refinamento

Os dados recebidos tanto pelo *host* como equipamento são exibidos na tela de simulação identificando o nome do módulo e o tipo de dados. No caso do módulo Dusb é exibido o tipo do pacote seguido por todas as informações contidas no atributo dados da classe packet em formato hexadecimal e separados por uma trena "|", assim pode-se observar os vários estágios da transferência de controle, conforme a modelagem do Statechart da figura 36.

Repare que os dados recebidos pelo módulo Aplicação e *Host*, identificados na simulação como "apl" e "hst" (linhas 54, 64 e 69 da figura 40) são idênticos aos exibidos na simulação da etapa anterior (figura 31).

Apesar do modelo simplificado, os *testbenches* criados são suficientes para comprovar a funcionalidade implementada, forçando possíveis anormalidades como a transmissão de pacotes fora de ordem, transações com endereços inválidos e demora excessiva na transmissão do *host* para o equipamento.

4.4 Modelagem de Desempenho

Até esse ponto não foi considerada nenhuma característica arquitetural, de maneira que o projeto possui uma independência total em relação a arquitetura alvo.

A decisão dos elementos arquiteturais a serem utilizados são analisados com base no modelo conceitual. O ponto central deste processo são as estimativas realizadas em contraste com os requisitos do projeto. No caso em questão, pretende-se obter um projeto flexível, atendendo os requisitos temporais mínimos da interface USB com o menor custo possível.

4.4.1 Análise Funcional

A atividade de analisar a especificação executável da etapa anterior de forma a levantar algumas características do projeto é normalmente conhecida como *profile*.

Esta análise fornece dados independente da arquitetura, servindo como entrada para diversas técnicas de exploração, particionamento e alocação de componentes. Existem algumas ferramentas de *profile* capazes de auxiliar e automatizar a captura de diversos dados. A ferramenta SCE (ABDI et al., 2003) por exemplo realiza uma análise abrangente da especificação em SpecC, porém, não tomou-se conhecimento de uma ferramenta específica para o SystemC.

Nesse caso, alguns dados foram obtidos usando o *profile* do Microsoft Visual C++ 6.0, conforme figura 41, que exibe as informações do *profile* para o *testbench* da etapa anterior.

```
Count    %   Line
------------------
  121  52,8 (dsie.cpp:195)
   33  14,4 (dusbrcv.cpp:37)
   20   8,7 (ep0.cpp:15)
   14   6,1 (dusbsnd.cpp:23)
   10   4,4 (ep1.cpp:32)
   10   4,4 (packet.h:37)
    2   0,9 (aplic.cpp:19)
```

Figura 41: Microsoft Visual C++ 6.0 *Profile*

Para gerar os dados da figura 41, algumas precauções foram necessárias visto não se tratar de uma ferramenta específica para SystemC, por exemplo, os tempos de execução dos processos acabam sendo contabilizados mesmo quando um *wait* é executado, bem como a quantidade de vezes que os processos são chamados, acabam sendo iguais para todos os comportamentos, devido a característica do simulador SystemC que implicitamente realiza as chamadas aos processos um a um. Nesse caso, foi implementado um perfil por linha do código *(line count profiling)*, contabilizando quantas vezes foram executadas as linhas ou trechos de códigos previamente selecionados.

Assim, com uma seqüencia de testes foi possível obter uma estimativa para os valores, conforme exibido na tabela 4. Os dados são os valores médios da seqüencia de simulações realizadas, capazes de fornecer um indicador simplificado para as decisões *hardware/-software*.

Módulo	Complexidade	Taxa de Acesso
Aplicação	Baixa	1
Dusb-EP1	Média	5
Dusb-EP0	Média	5
Dusb-Snd	Baixa	7
Dusb-Rcv	Baixa	10
Dsie	Alta	40

Tabela 4: Perfil da Especificação Conceitual

Esta simplificação foi inspirada na técnica utilizada em (GERSTLAUER et al., 1999). A complexidade e taxa de acesso exibidos na tabela 4 são valores comparativos, tomando o módulo aplicação como origem comparativa. Assim o Dusb-Rcv é acessado em média 10 vezes mais que o módulo aplicação, isso pode ser facilmente observado pela simulação da figura 40, além disso ele possui uma complexidade similar ao da aplicação, isto é

constatado pela quantidade de operações realizadas. Por outro lado o Dsie possui uma complexidade computacional elevada devida aos cálculos de CRC realizados.

4.4.2 Prototipação

Com base no perfil segue-se a atividade de alocação e mapeamento. Estas atividades são um complexo problema devido à natureza heterogênea do *hardware/sofware co-design* envolvendo inúmeras possibilidades de combinação. Para o caso do projeto em questão, a alocação e mapeamento desta fase são mostrados na figura 42.

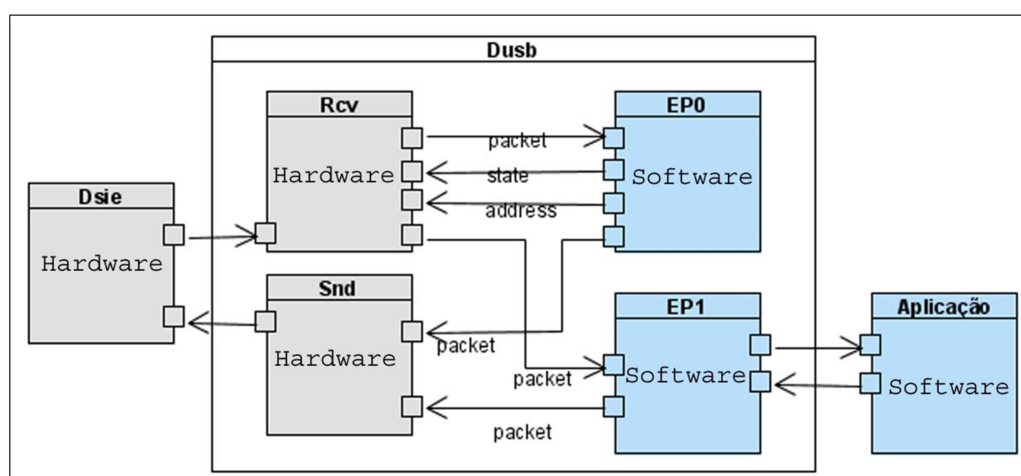

Figura 42: Modelo de Desempenho

Assim os módulos Dsie e Rcv e Snd são mapeados em componentes de *hardware* distintos enquanto os módulos EP0, EP1 e Aplicação foram mapeados em um único componente de *software*. A freqüência de utilização principalmente do módulo Dsie, justificam a decisão do mapeamento. No caso dos módulos EP0 e EP1, o componente de *software* se apresenta como uma alternativa de custo e flexibilidade para o projeto, onde outros *endpoints* poderiam ser futuramente implementados sem necessidade de novos componentes de *hardware*, além disso, conforme tabela 4, estes módulos não estão no caminho crítico para o desempenho do projeto.

A etapa de prototipação é encerrada refinando o modelo executável aproximadamente em relação aos componentes selecionados. Isto é realizado inserindo *waits* no código, baseado na estimativa de execução das funcionalidades em seus respectivos elementos.

No caso do projeto exemplo aqui descrito, as estimativas de tempo dos componentes de *hardware* foram baseados no CPLD CY7C371-143 da Cypress (CYPRESS, 2004). A motivação para a escolha foi baseado em sua simplificada especificação de tempo (SKAHILL, 1996) e pela disponibilidade deste modelo no laboratório de automação inteligente da

UFES, podendo ser utilizado como plataforma para a síntese real do projeto numa possível continuidade do trabalho.

Inspirado em (POSADAS et al., 2004), os tempos computacionais foram inseridos em segmentos do código, estimando a quantidade de ciclos em cada segmento de acordo com a arquitetura escolhida (*hardware/software*). O trecho de código da listagem 4.4 ilustra este processo, observe as anotações de tempo sublinhadas.

Listagem 4.4: Exemplo de Anotação de Tempo no módulo Dsie

```
void Dsie::mRcvCtrl(void) { sbyte sb; while (true) {
    sb = iData->read();
    din = sb.dado;
    wait(tck);
    switch (aSieState) {
    case PID:
        pout.clear();
        switch(din) {
        case 0x1E:
            pout.type = out;
            aSieState = TOKEN1;
            wait(tck);
            break;
        case 0x96:
            pout.type = in;
            aSieState = TOKEN1;
            wait(tck);
            break;
...
    case TOKEN1:
        crc5 = 0x1F;
        pout.dados[0] = (din & 0x7F) >> 1;
        crcdata = pout.dados[0];
        for (i=0;i<=6;i++) {
            ax = (crcdata & 0x01) << 4;
            bx = (crc5 & 0x10);
            crc5 <<= 1;
            crcdata >>= 1;
            if (ax^bx) crc5 ^= poly5;
            wait(tck);
        }
        ax = 0x01 & din ;
        aSieState = TOKEN2;
        wait(tck);
...
```

É importante observar que a estrutura do código não foi modificada, bem como o projetista tem amplas possibilidades de modificações para verificar alternativas e outras possibilidades através de simulações, conforme comentado na próxima subseção.

4.4.3 Validação e Verificação

Esta técnica é endereçada como co-simulação temporal (*timed-cosimulação*), levando em consideração as restrições e requisitos do projeto.

Estruturalmente o *testbench* é o mesmo já definido anteriormente (vide figura 39). No entanto, as restrições de tempo do projeto devem ser anotadas nos módulos de testes. A listagem 4.5 ilustra as restrições de tempo inseridas no módulo Husb em um trecho do processo de enumeração (observe os anotações sublinhadas).

Listagem 4.5: Anotação das Restrições de tempo no modelo de testes

```
void Husb::enumeration(void) {
...
    GetDeviceDescriptor();
    wait(frame);
    SetAddress();
    wait(Taddr);
...
}
void Husb::GetDeviceDescriptor(void) {
...
    //TOKEN PHASE (SETUP STAGE) ----------
        p.type= setup;
        p.dados[0] = 0x00; //address
        p.dados[1] = 0x00; //endpoint
        p.tam = 2;
        oPacket->write(p);
    //----------------------------------------
        wait(Trspipd);

    //DATA PHASE (SETUP STAGE) ----------
        p.type= data0;
...
        oPacket->write(p);
        TimeWaitingStart = sc_simulation_time();

    //HANDSHAKE PHASE (SETUP STAGE) ---
        wait(evRcv);
        if (pin.type != ack) {
            cout << "***erro -> esperava-se ack (Husb.cpp -ln119)\n";
        }
        if (sc_simulation_time()-TimeWaitingStart > Trspipd)
            fprintf(f->flog,"* \n"); //alerta
        else
            fprintf(f->flog,"\n"); //ok!
    //----------------------------------------
        wait(Tdrqcmpl);

    //TOKEN PHASE (DATA STAGE) ----------
        p.type = in;
...
```

Os requisitos temporais observados na listagem 4.5 são definidos idealmente em um arquivo de configuração do projeto. A tabela 5 lista os requisitos de tempo considerados no projeto, comparando as restrições para as três velocidades comportadas pela interface USB (USB WORK GROUP, 2000).

Símbolo	Descrição	*LowSpeed*	*FullSpeed*	*HighSpeed*
Bit-Time	Período de um *Bit*	666,66 nS	83,33 nS	2,0833 nS
(μ)*frame*	Intervalo entre os *frames* ou μ*frames*	1 mS	1 mS	125 μS
T_{IPD}	Tempo mínimo entre pacotes (*Inter Packet Delay*)	2 *bit-times*	2 *bit-times*	88 *bit-times*
T_{RSPIPD}	Tempo mínimo para resposta do equipamento	7,5 *bit-times*	7,5 *bit-times*	192 *bit-times*
$T_{DRQCMPL}$	Tempo para completar uma requisição (sem retornar dados)	(max)50 ms	(max)50 ms	(max)50 ms
$T_{DRETDT1}$	Tempo para completar uma requisição (retornando dados)	(max)500 ms	(max)500 ms	(max)500 ms
T_{ADDR}	Tempo para que um endereço atribuído passe a vigorar	(max)2 ms	(max)2 ms	(max)2 ms

Tabela 5: Restrições de tempo do projeto USB

A co-simulação temporal, diferente das etapas anteriores de validação, deverá conter os aspectos temporais do projeto. Dessa maneira, a saída da simulação foi gerada em arquivo, fornecendo ao projetista os instantes de início e tempos de latência durante a execução de cada evento do projeto, conforme pode ser observado na figura 43, que mostra uma trecho do arquivo de saída gerado.

É importante que as funcionalidades dos módulos de testes permaneçam sem inserções de tempo (*Untimed*), de maneira a não influenciar o resultado das simulações de tempo. O arquivo de simulação, conforme exemplificado na figura 43, exibe a esquerda o tempo da ocorrência do evento em μS, seguido por uma identificação textual e, ser for o caso, a quantidade de mensagens trafegadas e o tempo transcorrido (em nS) na execução do evento. Isto foi muito importante para identificar pontos críticos e falhas, até mesmo durante o refinamento realizado nesta etapa.

Os pontos críticos mapeados durante a simulação podem ser alvo de um refinamento da estimativa. Normalmente o projetista traduz o segmento crítico do código para a arquitetura alvo de maneira a avaliar mais precisamente a estimativa inserida.

```
73  ...
74  *******************************************************************************************
75  *         REQUEST SET ADDRESS                                                             *
76  *******************************************************************************************
77  [  6009.823]SETUP STAGE
78  [  6009.823]tkn phase — snd setup
79  [  6009.823]    Dsie Receving...d2
80  [  6009.990]dta phase — snd data0
81  [  6010.372]    Dsie Receive Finished                              (qt: 3 te:    548)Dsie
82  [  6010.392]    Dsie Receving...3c
83  [  6010.392]            Dusb Received...2(stp)
84  [  6010.559]            Dusb Receive Finished                      (qt: 1 te:    167)Dusb
85  [  6010.580]                    EP0 Receiving...2(stp)
86  [  6011.080]                    EP0 Tratamento Concluido           (qt: 1 te:    500)Dep0
87  [  6012.996]    Dsie Receive Finished                              (qt:11 te:   2604)Dsie
88  [  6013.017]            Dusb Received...5(dt0)
89  [  6013.683]            Dusb Receive Finished                      (qt: 1 te:    667)Dusb
90  [  6013.704]                    EP0 Receiving...5(dt0)
91  [  6014.288]            Dusb Sending...7(ack)
92  [  6014.371]            Dusb Send Finished                         (qt: 1 te:     83)Dusb
93  [  6014.392]    Dsie Sending...7(ack)
94  [  6014.413]hsk phase — rcv ack                                    (qt: 1 te:   4423)Husb*
95  [  6014.413]SETUP STAGE CONCLUDED                                  (qt: 2 te:   4589)Husb
96  [  6014.433]    Dsie Send Finished                                 (qt: 0 te:     42)Dsie
97  [  6015.017]                    EP0 Tratamento Concluido           (qt: 1 te:   1312)Dep0
98  [  8014.413]STATUS STAGE
99  [  8014.413]tkn phase — snd in
100 [  8014.413]    Dsie Receving...96
101 [  8014.967]    Dsie Receive Finished                              (qt: 3 te:    555)Dsie
102 [  8014.988]            Dusb Received...3(in )
103 [  8015.155]            Dusb Receive Finished                      (qt: 1 te:    167)Dusb
104 [  8015.176]                    EP0 Receiving...3(in )
105 [  8015.509]            Dusb Sending...5(dt0)
106 [  8015.592]            Dusb Send Finished                         (qt: 1 te:     83)Dusb
107 [  8015.613]    Dsie Sending...5(dt0)
108 [  8015.634]dta phase — rcv data                                   (qt: 1 te:   1221)Husb*
109 [  8015.634]hsk phase — snd ack
110 [  8015.634]STATUS STAGE CONCLUDED                                 (qt: 2 te:   1221)Husb
111 *******************************************************************************************
112 *         REQUEST CONCLUDED                                        (qt: 0 te:2005811)Husb
113 *******************************************************************************************
114 ...
```

Figura 43: Exemplo do arquivo de saída gerado pelo *testbench*

Devido a quantidade de informações, é comum a utilização de ferramentas para análise dos resultados, técnicas de análise formal ou validações em tempo de simulação. No caso do USB, isto foi realizado em relação ao requisito temporal T_{RSPIPD} em comparação ao tempo de resposta. Caso a restrição tenha sido ultrapassada, é impresso um asterisco a direita do evento. Na figura 43 por exemplo, pode-se observar esta restrição atingida em dois pontos: linhas 94 e 108 (se for considerado o caso de um projeto USB *full-speed*. Visto que o tempo mínimo para resposta é aproximadamente 622 nS, conforme T_{RSPIPD} na tabela 5).

Nesse caso é importante ressaltar que no caso de um projeto em *full-speed* ou *high-speed*, a arquitetura de *hardware/software* selecionada deveria ser revista, em razão do não atendimento da restrição T_{RSPIPD} observada acima. Contudo, a arquitetura foi mantida sem modificações, uma vez que, em *low-speed*, todos os requisitos são satisfeitos (T_{RSPIPD} $\sim 5\ \mu$S).

4.5 Modelagem Arquitetural

Na etapa funcional realizada na última seção, a modificação da especificação inicial foi mínima, porém ainda é necessário modificar o modelo para refletir as decisões de projeto.

Dessa maneira os módulos EP0, EP1 e Aplicação que, de acordo com a figura 42, serão mapeados em *software*, necessitam ser serializados em um único módulo, bem como os canais de comunicação precisam ser modificados para satisfazer a nova estrutura.

Assim, a figura 44 exibe modelo arquitetural, onde é possível visualizar os 4 PEs se comunicando através dos canais de comunicação agupados que, nesse caso, representam o barramento da arquitetura.

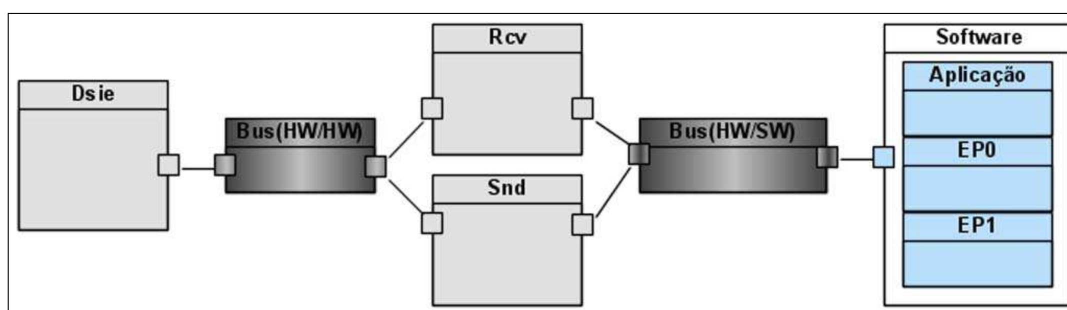

Figura 44: Modelo Arquitetural

4.5.1 Prototipação

A maior parte do refinamento realizado foi na criação do barramento de comunicação entre os PEs. O restante do refinamento foi no contexto estrutural, de maneira a associar cada PE a um módulo. Assim foi criado um módulo para o *software*, chamado de “PeSw”, em contrapartida o módulo Dusb foi eliminado. Conforme exibido na figura 44.

O pouco das modificações realizadas nos comportamentos funcionais, foi para adequar as chamadas de comunicação ao barramento criado, agregando simplicidade aos comportamentos. Com esse propósito, alguns processos puderam ser, inclusive, eliminados. A listagem 4.6 exemplifica a criação do canal de comunicação e a figura 45 exemplifica algumas partes importantes do módulo “PeSw” ilustrando as chamadas aos canais.

Vale ressaltar que o refinamento pode ser realizado em vários passos, dependendo da complexidade do projeto e da quantidade de modificações pertinentes da etapa. No caso do USB esta etapa foi realizada em 4 pequenos passos de refinamento de maneira a proporcionar modificações simples e suaves. Assim, cada pequena modificação realizada foi validada utilizando os mesmos *testbenches* já criados, evitando a inserção de erros.

Listagem 4.6: Protótipo do barramento HwSwBus

```
class SwIf : virtual public sc_interface { public:
    virtual packet ReadPacketRcv(short &) = 0;
    virtual void WriteState(int) = 0;
    virtual void WriteAddress(int) = 0;
    virtual void WritePacketSnd(packet) = 0;
};

class DusbRcvIfHwSw : virtual public sc_interface {
public:
    virtual void WritePacketRcv(packet,short) = 0;
    virtual int ReadState() = 0;
    virtual int ReadAddress() = 0;
};

class DusbSndIfHwSw : virtual public sc_interface {
public:
    virtual packet ReadPacketSnd() = 0;
};

class HwSwBus:  public sc_channel,
                public DusbRcvIfHwSw,
                public DusbSndIfHwSw,
                public SwIf{
...
```

PeSw.h

```
   :::
15 SC_MODULE(PeSw) {
16
17     sc_port<SwIf>  HwSwBus;
   :::
47     void ep1_CtrlSetup(packet p);
48     void ep1_TrataSetup(packet &p);
49     void ep1_TrataIn();
50     void ep0_CtrlSetup(packet p);
51     void ep0_TrataSetup(packet &p);
52     void ep0_GetDescriptor(packet &p);
53     void ep0_TrataIn();
54     void aplSetMemo(data& cmd);
55     void aplGetMemo(data& cmd);
56     void aplSetName(data& cmd);
57     void aplGetName(data& cmd);
58     void aplCtrl(data d);
59
60     void main();
61
62     SC_CTOR(PeSw) {
   :::
81         SC_THREAD(main);
82     }
83 };
```

PeSw.cpp

```
    ...
442 void PeSw::main() {
443 short ep;
444 packet p;
445     while (true) {
446         p = HwSwBus->ReadPacketRcv(ep);
447         if (ep==0)
448             ep0_CtrlSetup(p);
449         else if (ep==1)
450             ep1_CtrlSetup(p);
451         else
452             cout << " packet ignored - ep" <
453     }
454 }
    :::
```

Figura 45: Protótipo do Elemento de SW (PeSw)

4.5.2 Validação e Verificação

A validação tem um fator fundamental nesta etapa, visto a quantidade de modificações necessárias quando comparado com as etapas conceituais. Nesse caso, todos os modelos de teste podem ser aproveitados sem nenhuma modificação.

Dessa maneira, possíveis inserções de erros podem ser verificadas de maneira rápida e simples. Como exemplo, a figura 46 ilustra uma parte da simulação onde uma falha foi identificada durante o processo de refinamento. O alerta observado na figura (linha 284), foi emitido porque o *Host* recebeu um pacote *Data* no instante em que aguardava um pacote *Ack*, e aconteceu devido a uma pequena falha de codificação que acabava transmitindo duas vezes o pacote *Data*.

```
2               SystemC 2.0.1 --- Apr  8 2005 22:58:16
3          Copyright (c) 1996-2002 by all Contributors
4                      ALL RIGHTS RESERVED
5   ...
275   Hsie-rcv:42
276   Hsie-rcv:2d
277   Hsie-rcv:55
278   Hsie-rcv:46
279   Hsie-rcv:45
280   Hsie-rcv:53
281   Hsie-rcv:ba
282   Hsie-rcv:1d
283    Husb-rcv:6(dt1)|4|1|8|55|53|42|2d|55|46|45|53(USB-UFES)
284 ***erro -> esperava-se ack (Husb.cpp -ln119)
285   Hsie-rcv:2d
286    Husb-rcv:7(ack)
287 ...
```

Figura 46: Exemplo de falha detectada durante a simulação

Ao final do processo, o *testbench* é refinado com a criação de um canal de comunicação abstrato, ligando os modelos de teste do host com o projeto sobre desenvolvimento, conforme a figura 47.

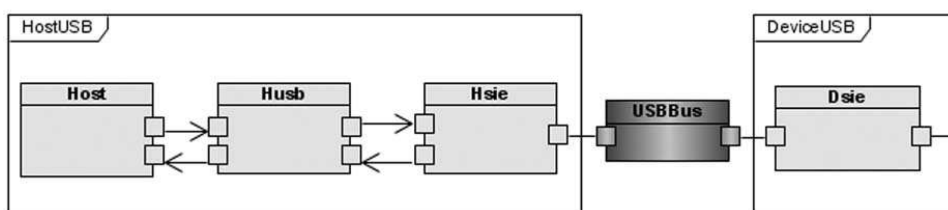

Figura 47: Simulação Arquitetural

A listagem 4.7 exibe os ajustes realizados no modelo, mostrando a instanciação dos barramentos e a ligação dos módulos aos barramentos. Os resultados das simulações devem permanecer inalterados, indicando que não foram inseridos erros durante o refinamento da etapa anterior.

Listagem 4.7: *Testbench* do Modelo Arquitetural

```
int sc_main(int argc, char *argv[]) {
...
    HwSwBus     i_HwSwBus("HSBUS");
    HwHwBus     i_HwHwBus("HHBUS");
    UsbBus      i_UsbBus("USBUS");

    //ligação dos módulos
...
    i_Hsie.iPacket(oFifoHusb);
    i_Hsie.oPacket(iFifoHusb);
    i_Hsie.UsbBus(i_UsbBus);

    i_Dsie.UsbBus(i_UsbBus);
    i_Dsie.HwHwBus(i_HwHwBus);

    i_UsbRcv.HwHwBus(i_HwHwBus);
    i_UsbRcv.HwSwBus(i_HwSwBus);

    i_UsbSnd.HwHwBus(i_HwHwBus);
    i_UsbSnd.HwSwBus(i_HwSwBus);

    i_PeSw.HwSwBus(i_HwSwBus);

    sc_start();
    return 0;
}
```

4.6 Modelagem Comportamental da Comunicação

Nesta etapa, a comunicação é refinada, mantendo a independência entre a comunicação e a funcionalidade dos módulos. Este é um conceito importante na criação de componentes reutilizáveis (KEUTZER et al., 2000).

Assim, a comunicação é especificada em um nível de abstração com tempo e pinagem exatas, enquanto os módulos funcionais permanecem nos níveis de abstrações conceituais. Isto é possível, através das técnicas de adaptadores e empacotamento (GROTKER, 2002). Devido a essas características, este nível representa o ponto intermediário entre o nível TLM e o nível comportamental.

A figura 48, exibe o modelo definido nesta etapa, as interfaces representam os adaptadores. Assim os módulos acessam os adaptadores da mesma forma que acessavam o barramento da etapa arquitetural.

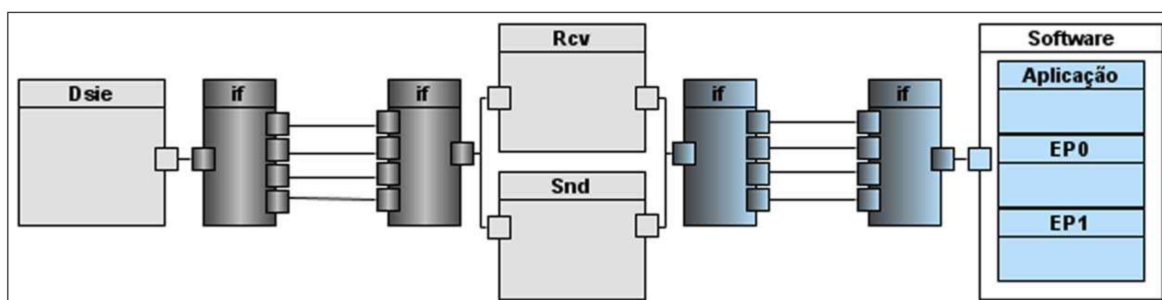

Figura 48: Modelo de Comunicação

4.6.1 Análise e Levantamento de Requisitos

Qualquer decisão de projeto referente a comunicação interna e externa entre os elementos de processamento devem ser definidos nesta etapa. Assim, foi utilizado o padrão Wishbone (OPENCORES, 2002) para a comunicação interna entre os PEs e para manter os modelos sem alterações funcionais (conforme definido pela metodologia), a comunicação é realizada em blocos.

Na comunicação externa do projeto é necessário considerar um módulo adicional, entre a linha de comunicação USB e o módulo Dsie. Para isso o USB-IF[1] *(USB-Implementers Forum)*, define uma macrocélula padrão, conhecida como UTM (*USB Transceiver Macrocell*) atuando na sinalização de baixo nível do protocolo USB. A figura 49 exibe a ligação física do UTM com a estrutura atual o projeto.

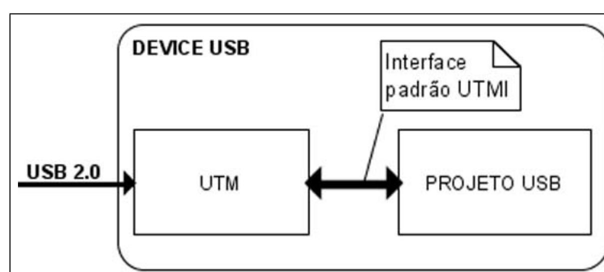

Figura 49: Modelo físico da ligação entre o UTM e o projeto USB

O UTM é agora necessário, devido a definição exata da pinagem de comunicação entre o equipamento e a linha USB. Além disso, como o UTM é um componente bem definido e um padrão utilizado nos projetos USB, não se fazia necessário sua especificação conceitual, nem mesmo existia dúvida quanto sua natureza (*hardware/software*).

Nesse caso a interface externa de comunicação do projeto seguirá o padrão de interface especificado pelo bloco UTM, chamada de UTMI (*USB Transceiver Macrocell Interface*) (INTEL, 2001).

[1] www.usb.org

4.6.2 Prototipação

O refinamento do protótipo desta etapa consiste em substituir os barramentos especificados na última etapa por modelos com exatidão de pinos e de ciclo de relógio.

Isto é realizado através de adaptadores conforme demonstrado na figura 50, que exibe o refinamento realizado para barramento HwHwBus.

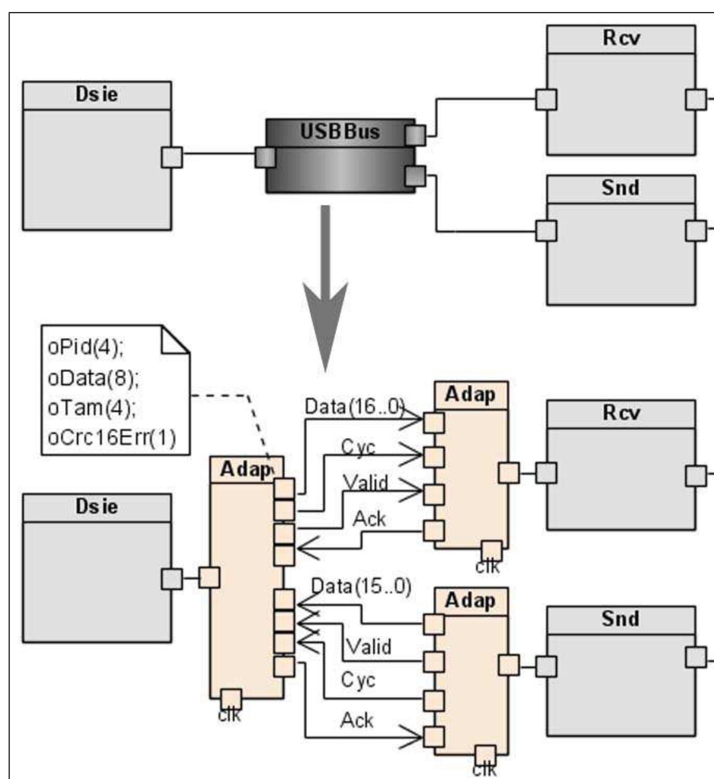

Figura 50: Refinamento do Canal usando adaptadores

Os sinais definidos nas interconexão dos adaptadores da figura 50 foram baseadas na especificação do Wishbone (OPENCORES, 2002). Isto pode ser melhor visualizado na implementação dos adaptadores em SystemC, conforme exibido na listagem 4.8.

A figura 4.8 exibe uma parte da implementação do adaptador utilizado pelo Dsie. Observe que as interfaces entre os PEs e os adaptadores não são modificadas. No caso da figura, a interface DsieIfHwHw implementada pelo adaptador, possui as mesmas funções da etapa anterior, como é o caso da função WritePacketRcv implementada agora em nível comportamental.

Listagem 4.8: Exemplo de código em SystemC para o refinamento dos Canais

```
DsieIfHwHwBus:       public sc_channel ,
                           public DsieIfHwHw{
public:
                                         //WISHBONE CROSS REFERENCE//
    sc_in_clk              clk;          // CLK_I                  //
                                         //———MASTER INTERFACE————//
    sc_out<sc_uint<4> > oPid;            //                        //
    sc_out<sc_uint<8> > oData;           // DAT_O(21..0)           //
    sc_out<sc_uint<4> > oTam;            //                        //
    sc_out<bool>           oCrc16Err;    //                        //
    sc_out<bool>           oValid;       // STB_O                  //
    sc_out<bool>           oCyc;         // CYC_O                  //
    sc_in<bool>            iAck;         // ACK_I                  //
                                         //———SLAVE INTERFACE——— //
    sc_in<sc_uint<4> >  iPid;            //                        //
    sc_in<sc_uint<8> >  iData;           // DAT_O(21..0)           //
    sc_in<sc_uint<4> >  iTam;            //                        //
    sc_in<bool>            iValid;       // STB_O                  //
    sc_out<bool>           iCyc;         // CYC_I                  //
    sc_out<bool>           oAck;         // ACK_I                  //
                                         ////////////////////////////

//Dados Recebidos no Barramento USB e enviados para DusbRcv
    void WritePacketRcv(packet p){
    int i;
        oPid = p.type;
        oCrc16Err = false;
        oTam = p.tam;
...
```

4.6.3 Validação e Verificação

Na validação realizada nesta etapa, as simulações são realizadas no ciclo de relógio (*clock accurated*), nesse caso, o tempo de simulação é significativamente maior que nas etapas anteriores.

Por outro lado, é possível registrar as variações dos sinais em nível comportamental e, assim, gerar arquivos que possam ser exibidos graficamente. Os formatos padrões mais comuns como VCD (*Value Change Dump*), WIF (*Waveform Intermediate Format*) e ISDB (*Integrated Signal Data Base*) são suportados pelo SystemC.

A listagem 4.9 demonstra a utilização do *tracing* do SystemC para geração de um arquivo de simulação no padrão VCD.

Os sinais registrados no arquivo TraceHwHw, são referentes a interconexão entre os PEs Dsie, Rcv e Snd (vide a figura 50) onde os sinais Rx são os sinais que são enviados para o Rcv e os sinais Tx são os sinais que o Snd envia para o Dsie.

Listagem 4.9: *Testbench* usando *tracing*

```
...
        sc_trace_file* tf1 = sc_create_vcd_trace_file("traceHwHw");
        sc_trace(tf1,clk,"clk");
        sc_trace(tf1,rxPid,"rxPid");
        sc_trace(tf1,rxData,"rxData");
        sc_trace(tf1,rxTam,"rxTam");

        sc_trace(tf1,rxValid,"rxValid");
        sc_trace(tf1,rxCyc,"rxCyc");
        sc_trace(tf1,rxAck,"rxAck");

        sc_trace(tf1,txPid,"txPid");
        sc_trace(tf1,txData,"txData");
        sc_trace(tf1,txTam,"txTam");
        sc_trace(tf1,txValid,"txValid");
        sc_trace(tf1,txCyc,"txCyc");
        sc_trace(tf1,txAck,"txAck");
...
        sc_start(sc_time(18,SC_MS));

        sc_close_vcd_trace_file(tf1);
...
```

A figura 51 exibe um trecho do gráfico de tempo para o arquivo de simulação gerado. Apesar do gráfico de tempo ser de difícil interpretação manual, eles são importantes para entender o comportamento do sistema e para o diagnóstico de falhas.

Os outros arquivos de saída já produzidos pelos modelos de testes podem auxiliar a identificação dos pontos importantes no gráfico. Nesse caso, o gráfico do tempo exibido na figura 51 retrata justamente o processo de endereçamento, que pode ser comparado com o arquivo gerado na seção 4.4.3, figura 43.

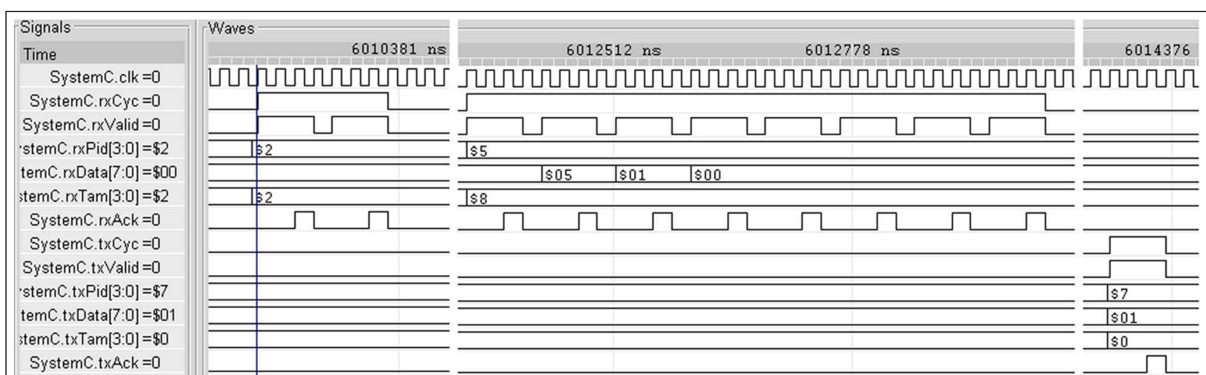

Figura 51: Gráfico de tempo da Simulação Comportamental

A primeira parte do gráfico corresponde à transmissão do *token setup*, a segunda parte aos dados da requisição que, nesse caso, contém o endereço que será atribuído ao equipamento e a última parte corresponde ao estágio de *status*.

4.7 Modelagem de Implementação

No modelo obtido na etapa anterior, somente os elementos de comunicação foram refinados, esta flexibilidade permitiu validar o contexto de comunicação sem nenhuma alteração nos módulos PEs, os quais são agora alterados de forma a integrar a comunicação, conforme pode ser observado na figura 52, que exemplifica o modelo obtido nesta etapa.

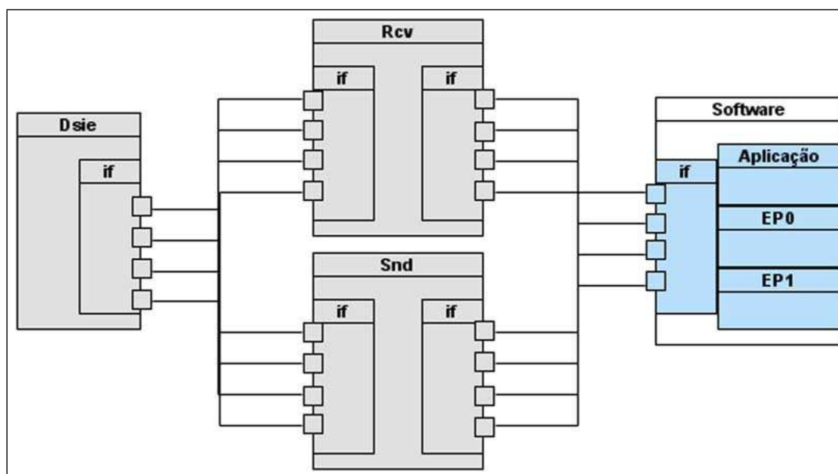

Figura 52: Modelo de Implementação

4.7.1 Análise e Levantamento de Requisitos

Para tirar proveito do refinamento sucessivo definido no trabalho, a modelagem desta etapa é realizada em três passos de refinamento. O primeiro passo consiste em integrar a comunicação ao PE sem alterar o comportamento interno dos PEs, isto envolve mover as funções definidas nos adaptadores para os módulos funcionais, conforme demonstrado na figura 53, para os módulos Dsie, Rcv e Snd (que modelam o barramento dos componentes de *hardware* exibido na seção 4.6.2, figura 50).

O segundo passo consiste no refinamento das funcionalidades em circunstância da comunicação integrada. Os modelos são refinados para o nível comportamental, os tipos de dados definidos conceitualmente, tais como "packet", "sbyte" e "data" são eliminados, implicando em alterações adicionais no modelo de comunicação. Este é um passo delicado e trabalhoso do processo de refinamento e deve ser realizado em sucessivos passos, auxiliado pelos modelos de testes.

Nesse caso, as entradas e saídas dos PEs que até então foram projetadas através de transações, agora são realizadas de acordo com as necessidades do modelo final. Os pacotes (*packet*), por exemplo, eram inteiramente montados antes de serem tratados ou

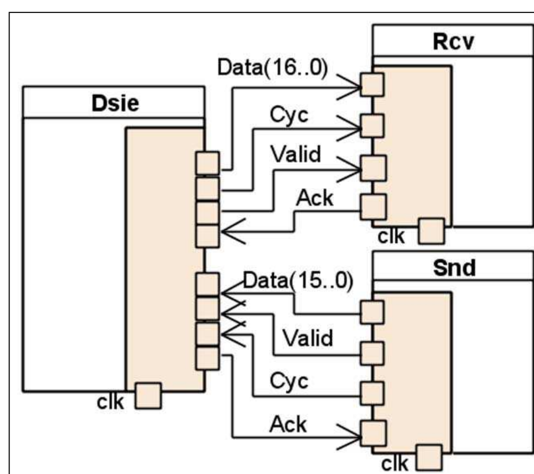

Figura 53: Integração dos adaptadores

repassados para os módulos seguintes, nesse caso, eles podem ser tratados tão logo os primeiros *bytes* sejam recepcionados.

Deve ser observado que os pacotes foram definidos desde o modelo conceitual, com base na especificação de pacotes USB (USB WORK GROUP, 2000), facilitando o tratamento e o entendimento das transações do protocolo USB. Enquanto os pacotes de *token* possuem poucos *bytes* capazes de identificar o endereço e o *endpoint* de destino, os pacotes de dados, podem conter, dependendo da interface, até 1024 *bytes*.

Mesmo que os *endpoints* especificados tenham sido projetados com o máximo de 64 *bytes*, as características analisadas acima tornam evidente as modificações necessárias, tanto na recepção quanto na transmissão dos PEs. Imagine, por exemplo, o Dsie recebendo um pacote inteiro de 64 *bytes* proveniente do Dsnd, ou seja, antes de iniciar a transmissão USB, o mesmo permanece ocioso vários ciclos de clock, até que os 64 *bytes* sejam disponibilizados para, só então, iniciar a montagem do CRC e transmissão dos mesmos para o *transceiver*.

Por fim, o terceiro passo resolve as rígidas exigências em relação a taxa de recepção e transmissão da interface USB, isto é, a interface externa, que até agora foi baseada nas propriedades de bloqueio da *fifo*, será modificada para atender os requisitos da especificação UTMI (INTEL, 2001). Nesse caso, ao invés do canal abstrato ser integrado ao Dsie, o mesmo é projetado como um novo módulo de *hardware*, capaz de recepcionar e armazenar os dados internamente, em paralelo ao comportamento do Dsie. Assim, enquanto o módulo Dsie trata o primeiro *byte* recepcionado, mais *bytes* vão sendo armazenados.

4.7.2 Prototipação

A integração dos adaptadores é realizada movendo as funções implementadas nos adaptadores para os PEs. A figura 54 demonstra o processo realizado para o elemento Rcv, a esquerda é ilustrado o módulo antes do refinamento e a direita após a integração da comunicação.

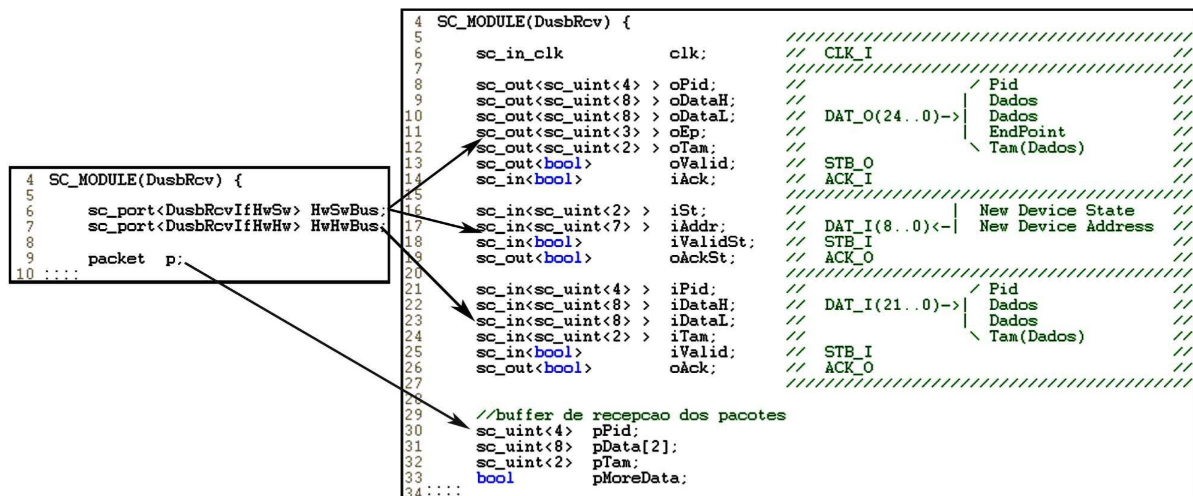

Figura 54: Modelo de Comunicação x Modelo de Implementação

Assim, as interfaces de comunicação interna entre os PEs, foram alteradas para trafegar: 4 bits para identificação do pacote, 16 bits de dados e 2 bits de tamanho. No sincronismo o sinal CYC (definido no modelo de comunicação) se tornou desnecessário, visto que agora a transmissão não é mais realizada em blocos. No caso dos pacotes de *tokens* os 2 *bytes* contém, respectivamente, o endereço e o *endpoint*.

A figura 54, exibe ainda, o *buffer* de dados em substituição ao tipo de dados *packet*, assim, as funções que foram mapeadas do canal abstrato são atualizadas para considerar estes sinais.

Todo o processo de refinamento realizado nesta etapa foi facilitado pela completa reutilização dos modelos de testes. Ao final da prototipação, o módulo Snd foi integrado ao Dsie, assim, a figura 55 exibe o modelo completo, resultante do trabalho realizado nesta etapa.

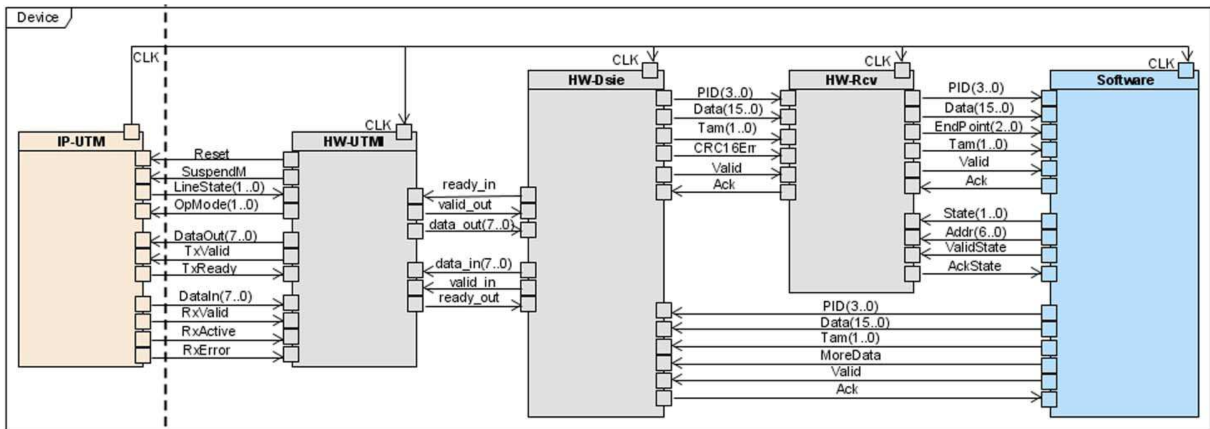

Figura 55: Modelo de Implementação Completo

4.7.3 Validação e Verificação

O modelo de teste foi atualizado para simular as solicitações de acordo com a interface padrão UTMI, nesse caso, para testar a capacidade e o desempenho do projeto, o modelo de teste foi especificado considerando a maior taxa de sinalização.

A maior carga de transmissão ao qual o projeto USB esta sujeito, acontece quando não ocorrem *stuff-bits*. Nessa situação, a cada 32 pulsos de *clock* um *byte* é sinalizado (INTEL, 2001).

Adicionalmente, foram capturados os sinais UTMI no ciclo de relógio, o qual foi analisado através do gráfico do tempo e comparado com a especificação. Também teve-se a preocupação de inserir alertas no módulo UTMI, para cercar decisões de projeto, como o tamanho da fila ou alguma situação de erro.

Além disso, foi gerado um terceiro arquivo de saída, especificamente para analisar as requisições chegando no elemento UTMI, as respectivas respostas e o tempo total que o projeto demorou para responde-las. A figura 56 mostra um trecho deste arquivo de saída, referente ao endereçamento (o mesmo trecho apresentado nas etapas anteriores).

O arquivo exibe à esquerda, o tempo de ocorrência do evento em μS, seguido pelo *byte* recebido/enviado e, para os casos em que o equipamento deve responder, é impresso o tempo transcorrido (em nS) desde a recepção completa da requisição até o início da resposta. Este é um dos tempos críticos do projeto, conforme comentado na seção 4.4.3 (vide tabela 5).

```
 72 [   5032.820] txUTMI(final)
 73 [   6033.514] rxUTMI(inicio)
 74 [   6033.514]    setup
 75 [   6033.514]     d2
 76 [   6034.180]     0
 77 [   6034.847]     8
 78 [   6034.868] rxUTMI(final)
 79 [   6035.534] rxUTMI(inicio)
 80 [   6035.534]    data0
 81 [   6035.534]     3c
 82 [   6036.201]     0
 83 [   6036.867]     5
 84 [   6037.534]     1
 85 [   6038.200]     0
 86 [   6038.867]     0
 87 [   6039.534]     0
 88 [   6040.200]     0
 89 [   6040.867]     0
 90 [   6041.533]     d7
 91 [   6042.200]     a4
 92 [   6042.221] rxUTMI(final)
 93 [   6043.054] txUTMI(inicio) :     833
 94 [   6043.054]    ack
 95 [   6043.054]     2d
 96 [   6043.762] txUTMI(final)
 97 [   8044.463] rxUTMI(inicio)
 98 [   8044.463]    in
 99 [   8044.463]     96
100 [   8045.129]     0
101 [   8045.796]     8
102 [   8045.817] rxUTMI(final)
103 [   8046.733] txUTMI(inicio) :     917
104 [   8046.733]    data0
105 [   8046.733]     3c
106 [   8047.441] txUTMI(final)
107 [   8048.150] rxUTMI(inicio)
108 [   8048.150]    ack
109 [   8048.150]     2d
110 [   8048.170] rxUTMI(final)
111 [  10048.142] rxUTMI(inicio)
```

Figura 56: Arquivo de Log - recepção e envio de dados pelo UTMI

4.8 Considerações Finais

Este capítulo demonstrou a aplicação de uma metodologia bem delineada sobre um caso de teste complexo o suficiente para justificar o uso de modernas técnicas de modelagem *co-design*. Com esse propósito, uma interface USB foi especificada iniciando de um elevado nível de abstração até o nível de implementação comportamental.

Todas as estimativas realizadas atendem aos requisitos necessários para a operação do projeto em *low-speed*. Apesar de ser válido obter estimativas mais confiáveis utilizando SystemC, este propósito foge ao foco do trabalho.

Durante o refinamento, foi constatada a importância dos modelos de testes para avaliação de cada etapa do projeto, uma vez que, várias falhas de projeto foram detectadas e diagnosticadas. Durante cada refinamento, o *testbench* era executado e comparado com

a saída gerada antes do refinamento, esta comparação foi realizada através de ferramentas de comparação de arquivos, tal como ilustrado na figura 57. Somente se acontecesse alguma divergência era necessário uma verificação manual.

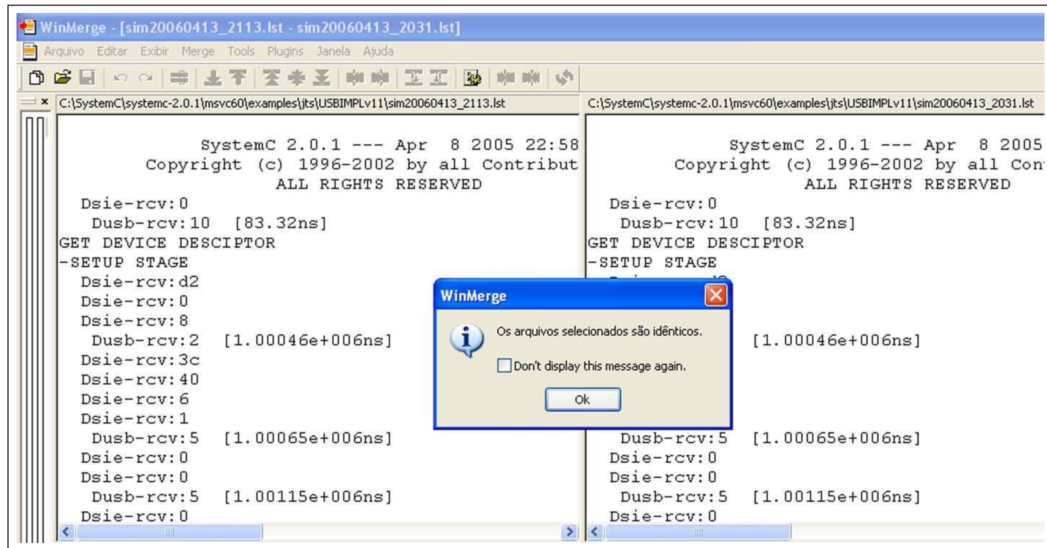

Figura 57: Exemplo de uma ferramenta de comparação de arquivos, utilizada para verificação dos resultados gerados pelos modelos de testes

De fato, os *testbenches*, as ferramentas para projetos em nível de sistema, tais como o SystemC e o uso ferramentas de análise de sistemas em alto nível, como UML, são de vitais importância para o desenvolvimento de sistema complexos, como o USB.

5 *Conclusão*

Este trabalho apresentou uma metodologia *co-design*, baseada em um refinamento sucessivo em nível de sistema, demonstrando o potencial dos novos conceitos de modelagem e a união das áreas da engenharia de *software* e *hardware* que até pouco tempo eram consideradas completamente independentes.

Os resultados obtidos durante o refinamento de uma interface USB foram promissores, visto a simplificação evidente do processo de desenvolvimento, permitindo uma sistematização bem definida, suave e natural para o desenvolvimento de projetos embarcados e SoC.

Com o processo de desenvolvimento definido, foi possível refinar o modelo a partir do nível de sistema, até um nível em que as decisões arquiteturais do projeto possam ser definidas com maior segurança e validadas em um ambiente unificado de *hardware/software*, propiciando facilidade na busca por alternativas de projeto.

Um dos fatores proeminentes, foi a possibilidade de se utilizar uma linguagem de modelagem padrão como UML, isto vai no sentido da própria integração das duas disciplinas de desenvolvimento: *hardware* e *software*. Neste mesmo caminho seguem as ferramentas de especificação em nível de sistema como o SystemC, que permitem esta unificação e potencializam metodologias de desenvolvimento como OO (Orientação a Objetos). Isto pôde ser evidenciado no decorrer da dissertação.

Assim, conforme demonstrado, os seguintes pontos podem ser destacados pela metodologia definida:

1. Capacidade de atuar num elevado nível de abstração, onde nenhum aspecto arquitetural precisa ser definido.

2. Um fluxo de refinamento bem definido, propiciando um projeto suave entre os diversos níveis de abstração definidos.

3. Facilidade de simulação e validação em cada passo da especificação, através de *testbenches* modulares e reutilizáveis.

4. Capacidade de fornecer informações e resultados que indiquem possíveis alternativas para exploração espacial do projeto, ou até mesmo gerar dados que poderiam servir de entrada para técnicas de particionamento e mapeamento.

5. Facilidades para o particionamento e mapeamento do modelo de forma simples e suave.

6. Capacidade de modelagens de alto nível usando UML, auxiliando o esforço de análise e documentação do sistema.

7. Capacidade de integração transparente entre os componentes de *hardware* e *software*.

Sobretudo, é importante mencionar a busca por ferramentas gratuitas e de baixo custo para a pesquisa realizada, desse modo, a linguagem SystemC, uma ferramenta *open-source* construída sobre a linguagem C++, foi o alvo da estratégia de modelagem empregada nesta dissertação.

O SystemC foi recentemente padronizado pelo IEEE, sendo oficialmente chamado de IEEE Std 1666-2005, IEEE *Standard* SystemC *Language Reference Manual* (IEEE, 2006). Apesar disso, não foi localizado uma metodologia bem definida para o desenvolvimento de projetos co-design. Esta carência foi o alvo da exploração realizada neste trabalho. O SpecC por exemplo, possui uma metodologia bem difundida, com vários casos de estudo, resultados e constatações práticas de seus benefícios, tanto, que algumas de suas propriedades, foram incorporados à implementação do SystemC.

Apesar da similaridade entre ambas ferramentas, a metodologia SpecC não mostrou bons resultados quando aplicada com o SystemC (CAI; VERMA; GAJSKI, 2003). De maneira contrária, a metodologia definida nesta dissertação explorou efetivamente os benefícios do SystemC, apoiado no estado da arte do projeto de sistema digitais em nível de sistema(conforme definido no capítulo 2), incluindo importantes conceitos da metodologia definida para o SpecC.

No decorrer do trabalho, foi possível evidenciar os benefícios da disciplina *co-design*, bem como a possibilidade de sua utilização sem necessidade de investir altos custos em ferramentas especializadas, algumas, custando milhares de dólares (GOERING, 2006), apoiado pela promissora fase da tecnologia SoC.

Considerando que atualmente a grande maioria de *IP-Core´s* disponíveis estão em linguagens de descrição de *hardware* como VHDL e Verilog, uma definição bem aceita dos níveis de abstrações permitiria a incorporação destes componentes IP em níveis de abstrações acima do RTL. Um projetista de *IP-Core*, por exemplo, poderia fornecer o mesmo produto em vários níveis de abstração, o que permitira sua adequação nas primeiras fases da especificação de um projeto. Este é um ponto que poderia ser explorado em um futuro trabalho. Existem várias pesquisas envolvendo este tema, de maneira que estas poderiam ser investigadas para incorporar a proposta de metodologia descrita.

Um problema prático verificado durante a exploração espacial do projeto foi a geração de estimativas. Para o caso de teste USB, as estimativas foram realizadas com base na experiência pessoal envolvendo o CPLD CY7C371-143, em alguns casos, uma parte da especificação foi traduzida para obtenção das estimativas de tempo, usando a ferramenta Warp da Cypress. Apesar do capítulo 3 apontar algumas sugestões para facilitar a obtenção destes parâmetros, as mesmas não foram exploradas, de maneira que seria fruto de futuras pesquisas envolvendo o trabalho, outros tipos de estimativas também poderiam ser analisadas, tais como, custo, área e potência. Este é um tema interessante e ainda em aberto.

No capítulo 4, a utilização de ferramentas de *profile* foi discutida, e como não se encontrou uma ferramenta específica para SystemC, a criação desta poderia ser alvo de um futuro trabalho.

O processo de desenvolvimento descrito no trabalho pode ainda ser aproveitado para a criação de ferramentas de particionamento e mapeamento automatizadas, técnicas de verificação formal e ferramentas gráficas de modelagem, capazes de gerar protótipos em SystemC baseados nos modelos UMLs.

Ainda como sugestão de futuros trabalhos, a implementação do projeto USB em uma plataforma real, poderia concretizar e validar os resultados alcançados.

Por fim, seria importante validar a proposta metodológica em outros projetos, de maneira a agregar resultados, elevar o escopo metodológico e refinar ainda mais a metodologia. Além disso, novas características do SystemC recentemente disponibilizadas poderiam ser alvo de futuros refinamentos da metodologia, com destaque para as novas alternativas de verificação.

Referências

ABDI, S. et al. System-on-chip environment: Sce version 2.2.0 beta. *Center for Embedded Computer Systems, University of California, Irvine*, Irvine, CA, USA, n. 03-41, 2003. Tutorial.

ACELLERA. *SystemVerilog 3.1a Languagem Reference Manual Accellera Extensions to Verilog*. Napa, CA, USA, 2004.

ARATO, P.; MANN, Z. A.; ORBAN, A. Component-based hardware/software co-design. In: *Proceedings of the 17th International Conference on Architecture of Computing Systems*. Augsburg, Germany: [s.n.], 2004.

BAGHDADI, A. et al. Design space exploration for hardware/software codesign of multiprocessor systems. In: *IEEE International Workshop on Rapid System Prototyping*. [s.n.], 2000. p. 8–13. Disponível em: <citeseer.ist.psu.edu/298739.html>.

BARROS, E. et al. *Hardware/Software co-design: projetando hardware e software concorrentemente*. São Paulo, Brasil: Escola de Computação, 2000.

CAI, L. *Estimation and Exploration Automation of System Level Design*. Tese (Doutorado) — University of California, 2004.

CAI, L.; GAJSKI, D. System level design using specc profiler. *Center for Embedded Computer Systems, University of California, Irvine*, Irvine, CA, USA, n. 02-08, 2002. Technical Report.

CAI, L.; GAJSKI, D. Transaction level modeling: an overview. In: *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM Press, 2003a. p. 19–24. ISBN 1-58113-742-7.

CAI, L.; GAJSKI, D. Transaction level modeling in system level design. *Center for Embedded Computer Systems, University of California, Irvine*, Irvine, CA, USA, n. 02-08, 2003b. Technical Report.

CAI, L.; VERMA, S.; GAJSKI, D. Comparison of specc and systemc languages for system design. *Center for Embedded Computer Systems, University of California, Irvine*, Irvine, CA, USA, n. 03-11, 2003. Technical Report. Disponível em: <citeseer.ist.psu.edu/cai03comparison.html>.

CORTÉS, L. A.; ELES, P.; PENG, Z. A survey on hardware/-software codesign representation models. *SAVE*, 1999. Disponível em: <http://www.ida.liu.se/labs/eslab/publications/pap/db/SAVE99.pdf>.

CYPRESS. *CY7C371i-UltraLogic 32-Macrocell Flash CPLD*. Document: 38-03032 rev. a. San Jose, CA, USA, 2004. Datasheet.

DÖMER, R. The specc system-level design language and methodology. In: *Embedded Systems Conference*. San Francisco: Center for Embedded Computer Systems, 2002.

EDENFELD, D. et al. 2003 technology roadmap for semiconductors. *IEEE Computer*, v. 37, n. 1, p. 47–56, 2004.

EDWARDS, S. et al. Design of embedded systems: Formal models, validation, and synthesis. *Proc. of the IEEE*, v. 85, n. 3, year 1997. Disponível em: <citeseer.ist.psu.edu/edwards97design.html>.

GAJSKI, D. et al. *System Design: A Practical Guide With SpecC*. Norwell, MA, USA: Kluwer Academic Publishers, 2001.

GAJSKI, D.; KUHN, R. H. New vlsi tools - guest editors' introduction. *IEEE Computer*, v. 16, n. 12, p. 11–14, 1983.

GAJSKI, D. et al. The specc methodology. *Department of Information and Computer Science, University of California, Irvine*, Irvine, CA, USA, n. 99-56, 1999. Technical Report.

GERSTLAUER, A. Specc modeling guidelines. *Center for Embedded Computer Systems, University of California, Irvine*, Irvine, CA, USA, n. 02-16, 2002. Technical Report.

GERSTLAUER, A. et al. Design of a gsm vocoder using specc methodology. *Department of Information and Computer Science, University of California, Irvine*, Irvine, CA, USA, n. 99-11, 1999. Technical Report.

GOERING, R. Tools ease transaction-level modeling. *EDA News of EETimes*, 2006. Disponível em: <http://www.eetimes.com/news/design/>.

GROTKER, T. *System Design with SystemC*. Norwell, MA, USA: Kluwer Academic Publishers, 2002. ISBN 1402070721.

HAREL, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, v. 8, n. 3, p. 231–274, June 1987. Disponível em: <citeseer.ist.psu.edu/harel87statecharts.html>.

HASEGAWA, T. Uml profile for soc - extension for soc design. In: UML-SOC WORKSHOP IN DAC 2004. [S.l.], 2004.

HAVERINEN, A. et al. Systemc based soc communication modeling for the ocp protocol. 2002. White paper. Disponível em: <http://www.ocpip.org>.

IEEE. Standard systemc® language reference manual. In: *Embedded Systems Conference*. New York, USA: IEEE Computer Society, 2006.

INTEL. *USB 2.0 Transceiver Macrocell Interface (UTMI)*. Version 1.05. [S.l.], 2001. Specification.

KEUTZER, K. et al. System level design: Orthogonalization of concerns and platform-based design. *IEEE Trans. on CAD*, 2000.

KOGEL, T. et al. Virtual architecture mapping: A systemc based methodology for architectural exploration of system-on-chip designs. In: *Computer Systems: Architectures, Modeling, and Simulation, Third and Fourth International Workshops, SAMOS 2003 and SAMOS 2004*. Greece: Springer, 2004. (Lecture Notes in Computer Science, v. 3133), p. 138–148. ISBN 3-540-22377-0.

KUKKALA, P. et al. Uml 2.0 profile for embedded system design. In: *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005. p. 710–715. ISBN 0-7695-2288-2.

MICHELI, G. D.; GUPTA, R. K. Hardware/sofware co-design. *IEEE*, v. 85, n. 3, 1997.

OMG. Uml profile for schedulability,performance, and time specification. *Object Management Group*, n. Version 1.1, 2005. Disponível em: <http://www.omg.org/docs/realtime/05-01-02.pdf>.

OPENCORES. *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. Revision b.3. [S.l.], 2002. Specification.

OSCI. Open systemc initiative. Functional Specification for SystemC 2.0. 2002a. Disponível em: <www.systemc.org>.

OSCI. Open systemc initiative. SystemC User´s Guide. 2002b. Disponível em: <www.systemc.org>.

POSADAS, H. et al. System-level performance analysis in systemc. In: *DATE '04: Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2004. p. 10378. ISBN 0-7695-2085-5-1.

PRESSMAN, R. *Software engineering : a practitioner's approach*. 5th. ed. New York, NY, USA: McGraw-Hill, 2001.

ROWEN, C. Reducing soc simulation and development time. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 35, n. 12, p. 29–34, 2002. ISSN 0018-9162.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. *The Unified Modeling Language Reference Manual*. United States of America: Addison-Wesley, 1999. ISBN 0-201-30998-X.

SELIC, B. Using uml for modeling complex real-time systems. In: *LCTES*. Montreal, Canada: Springer, 1998. (Lecture Notes in Computer Science, v. 1474), p. 250–260. ISBN 3-540-65075-X.

SIEWERT, S. Soc drawer: Function allocation and specification. *IBM developerWorks*, 2005.

SKAHILL, K. *VHDL for Programmable Logic*. USA e Canada: Addison-Wesley, 1996.

SOMMERVILLE, I. *Software Engineering*. 6th. ed. LANCASTER UK: Addison-Wesley, 2001.

SUTHERLAND, S. An overview of systemverilog 3.1. *EEdesign.com*, 2003.

SYSTEMC VERIFICATION WORKING GROUP. Systemc verification standard specification. In: *Submission to SystemC Steering Group*. [S.l.], 2003. Version 1.0e.

UML FOR SOC FORUM. Profile for soc - extension for soc design. In: . 2004. Disponível em: <http://www.omg.org/docs/realtime/04-10-11.pdf>.

USB WORK GROUP. *Universal Serial Bus Specification*. Revision 2.0. [S.l.], 2000. Specification.

VAHID, F.; GIVARGIS, T. *Embedded System Design: A Unified Hardware/Software Introduction*. NY - USA: John Wiley & Sons, Inc, 2002.

VANTHOURNOUT, B.; GOOSSENS, S.; KOGEL, T. Developing transaction-level models in systemc. *CoWare whitepaper*, 2005.

VINCENTELLI, A. S. Defining platform-based design. *EEDesign of EETimes*, February 2002. Disponível em: <http://www.gigascale.org/pubs/141.html>.

VINCENTELLI, A. S. et al. Benefits and challenges for platform-based design. In: *DAC '04: Proceedings of the 41st annual conference on Design automation*. New York, NY, USA: ACM Press, 2004. p. 409–414. ISBN 1-58113-828-8.

WALKER, R. A.; THOMAS, D. E. A model of design representation and synthesis. In: *DAC '85: Proceedings of the 22nd ACM/IEEE conference on Design automation*. [S.l.]: ACM Press, 1985. p. 453–459. ISBN 0-8186-0635-5.

APÊNDICE A – Fundamentos do SystemC

Um dos principais objetivos do SystemC é possibilitar a especificação dos sistemas acima dos nível de abstração de transferência de registros (RTL) e adicionalmente permitir seu refinamento até este nível (OSCI, 2002a).

Através de uma arquitetura em camadas, o SystemC permite que um conjunto extenso de modelos de desenvolvimento, níveis de abstrações e metodologias sejam utilizadas. O núcleo do SystemC se encontra na base desta arquitetura, construído inteiramente sobre a linguagem C++.

A figura 58, ilustra as várias camadas do SystemC. Pode-se observar o núcleo da linguagem (*Core Language*), onde seu principal elemento é um mecanismo de simulação, chamado de *Event-Driven Simulation*.

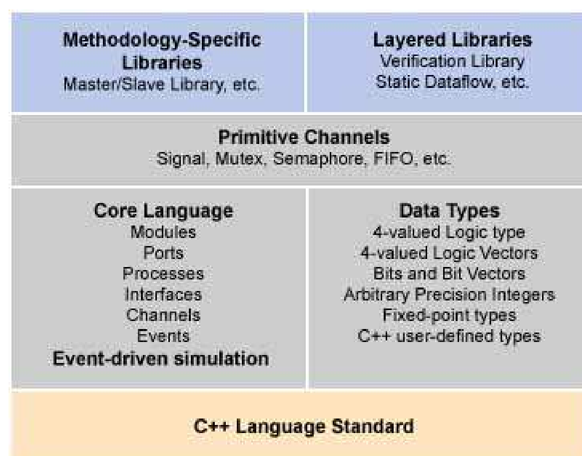

Figura 58: Arquitetura do SystemC. fonte: (OSCI, 2002b)

O núcleo de simulação trabalha com eventos e processos de uma maneira abstrata, não necessitando saber o que os eventos representam ou o que os processos fazem (GROTKER, 2002). Os outros elementos do núcleo incluem: módulos (*modules*), *interfaces*, canais (*channels*) e portas (*Ports*).

Estes elementos, juntamente com os canais de comunicação básicos (*Elementary Channels*), formam os fundamentos do SystemC, que serão a seguir melhor definidos. Acima destes se encontram modelos específicos e modelos de canais abstratos, que podem ser úteis ao desenvolvimento. Os tipos de dados são ortogonais ao núcleo e são de fundamental importância para definição de tipos específicos para modelagem de *hardware*.

A.1 Módulos

Os módulos são os principais elementos do SystemC, contendo os componentes físicos de processamento capazes de definir sua funcionalidade. O módulo é derivado da classe sc_module, geralmente, utilizando a macro SC_MODULE:

```
SC_MODULE(ModuloExemplo) {
  //Declaração de portas, corpo de processos, dados, etc
  SC_CTOR(ModuloExemplo) {
      //Corpo do Processo : Inicialização dos elementos
      //Declaração dos processos
  }
}
```

A macro SC_CTOR é utilizada para declaração do método construtor do módulo, nele são declarados os processos e o código necessário na inicialização do módulo, como o estado inicial do módulo ou a inicialização de outros dados internos. Um módulo tipicamente contém, conforme (GROTKER, 2002):

- Portas, com o qual o módulo se comunica com o ambiente;
- Processos, que descrevem a funcionalidade do módulo;
- Dados internos ou canais, mantendo os estados internos do módulo;
- Outros módulos, ou seja, capacidade hierárquica.

A.2 *Interfaces*, Portas e Canais

Por padrão, uma *interface* representa um conjunto de operações, sem especificar como estas são implementadas. Em SystemC as *interfaces* são derivadas da classe sc_interface.

As portas são utilizadas pelo módulo para exteriorizar sua funcionalidade, transmitindo ou recebendo dados de outros módulos. O SystemC possui três tipos básicos de portas: sc_in, sc_out e sc_inout. Cada uma destas portas são derivadas da classe base sc_port e possuem um conjunto de *interfaces*, tais como read() e write().

Os canais implementam as *interfaces*, ou seja, enquanto as *interfaces* e portas descrevem as funcionalidades disponíveis, os canais definem como estas funções são executadas. No caso dos tipos de portas básicas (sc_in, sc_out e sc_inout), estas são utilizadas pelos canais que implementam as *interfaces* read() e write().

Outros tipos de canais podem precisar de diferentes métodos, sendo necessários, outras *interfaces* e, conseqüentemente, diferentes portas de acesso. Os canais básicos, sc_signal e sc_fifo são dois exemplos de canais utilizados com freqüência na metodologia definida no projeto.

O sc_signal representa um sinal físico de *hardware* de maneira similar ao *signal* utilizado em VHDL, o mesmo implementa a *interface* sc_signal_inout_if e pode ser utilizada, juntamente com os tipos de portas básicas.

O sc_fifo é um canal de fila, implementando as *interfaces* sc_fifo_in_if e sc_fifo_out_if, estas *interfaces* possuem tanto métodos bloqueáveis e não bloqueáveis. Estes métodos bloqueantes foram utilizados com freqüência pela abordagem do projeto como uma maneira de facilitar o sincronismo dos módulos, isto é, se a fila está vazia, uma leitura suspenderia o processo, até que se tenha algum dado na fila. O mesmo aconteceria ao tentar escrever em uma fila cheia.

A listagem abaixo, exibe um exemplo de declaração de portas para um módulo de exemplo:

```
SC_MODULE(ModuloExemplo) {
   sc_in<bool> read;
   sc_port<sc_fifo_out_if<int> > dados;
   //restante do módulo não exibido
}
```

A.3 Processos

Os processos descrevem as funcionalidades dos módulos. Internamente são implementados de maneira similar a uma função típica do C++, porém, através do mecanismo de

simulação do SystemC, estas funções suportam concorrência de ações, isto é, execuções de processos em paralelo, onde mais de uma ação podem ser executadas ao mesmo tempo. Isto é fundamental para o projeto de componentes de *hardware*.

O SystemC possui três macros, utilizadas para fazer com que uma função se comporte como um processo: sc_method (*method process*), sc_thread (*thread process*) e sc_cthread (*clocked thread process*).

O sc_method é executado com base em uma lista de sensibilidade definida na declaração do mesmo, sempre que algum evento da lista de sensibilidade é notificado, o método é executado do início ao fim. Nesse caso, estes tipos de processos não podem suspender sua execução, não suportando métodos bloqueáveis tal como os utilizados pelos canais *fifo's*.

O sc_thread, ao contrário, pode suspender sua execução através de funções *wait()* ou qualquer outra forma de suspensão, nesse caso, quando o processo retorna à sua execução, ele continua do ponto onde havia parado.

O sc_cthread é uma variação do sc_thread onde a lista de sensibilidade do processo é restrito ao sinal de *clock*, nesse caso, o processo sempre será ativado ou avaliado na transição do *clock*.

A.4 Eventos

Os processos utilizando *threads*, além de serem mais abrangentes e com maior poder de expressão, podem ser sensíveis a eventos, o que os acrescenta, uma capacidade de sensibilidade dinâmica, isto é, os processos utilizando sc_threads, não estão restritos a uma lista de sensibilidade estática definida na declaração do processo.

Os eventos são utilizados em conjunto com a função wait(), por exemplo: (OSCI, 2002a)

```
...
// espera até que o evento e1 seja notificado
wait( e1 );
...
// espera até que o evento e1 ou o evento e2 seja notificado
wait( e1 | e2 );
...
```

Quando a função wait() é chamada sem argumentos o processo é suspenso até que um dos sinais em sua lista estática de sensibilidade seja notificado. A função wait() também aceita o tempo como argumento, por exemplo: (fonte: (GROTKER, 2002))

```
wait(100,SC_NS);
```

ou o equivalente:

```
sc_time t(100, SC_NS);
wait(t);
```

A.5 Mecanismo de Simulação

A linguagem SystemC é sobretudo uma linguagem de simulação, possuindo um escalonador (*SystemC scheduler*), capaz de controlar: o tempo, a ordem de execução dos processos, a notificação de eventos e as requisições dos canais de comunicação. Permitindo assim a noção de concorrência entre os processos.

Dessa maneira a simulação é executada em uma seqüencia de passos realizados pelo escalonador, conforme resumido abaixo (OSCI, 2002b):

1. **Inicialização:** Os processos são um a um inicializados.
2. **Avaliação:** Seleciona e executa um a um os processos que estejam prontos para execução, este passo é repetido enquanto existir processos prontos para execução.
3. **Atualização:** Qualquer chamada de atualização (*update()*) pendente é executada.
4. **Eventos Imediatos:** Se existir notificações imediatas, o escalonador identifica e sinaliza os processos para execução, retornando em seguida para o segundo passo.
5. **Eventos Futuros:** Se existir notificações no tempo, o escalonador atualiza o tempo corrente da simulação para a próxima notificação pendente, identifica e sinaliza os processos para execução e retornando para o segundo passo.
6. **Finalização:** A simulação é encerrada.

Glossário

ASIC	*Application Specific Integrated Circuit*
CPLD	*Complex Programmable Logic Device*
CRC	*Cyclic Redundancy Check*
DSP	*Digital Signal Processing*
DUT	*Device Under Test*
IP	Propriedade Intelectual
IP-Core	Núcleo de Propriedade Intelectual
ISDB	*Integrated Signal Data Base*
OO	Orientação a Objetos
OSCI	*Open SystemC Initiative*
PE	*Processing Element*
PID	*Packet Identification*
RT	*Register Transfer*
RTL	*Register Transfer Level*
SLDL	*System Level Design Language*
SoC	*System-on-Chip*
TLM	*Transaction Level Modeling*
UML	*Unified Modeling Language*
USB	*Universal Serial Bus*
UTM	*USB Transceiver Macrocell*

UTMI *USB Transceiver Macrocell Interface*

VCD *Value Change Dump*

WIF *Waveform Intermediate Format*