Sérgio Queiroz de Medeiros

Utilizando Programação Orientada a Aspectos no Projeto de Sistemas Hardware Desenvolvidos com SystemC

Natal-RN

2006

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

Sérgio Queiroz de Medeiros

Utilizando Programação Orientada a Aspectos no Projeto de Sistemas Hardware Desenvolvidos com SystemC

Orientador:
David Boris Paul Déharbe

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

Natal-RN

2006

Dissertação de Mestrado sob o título *Utilizando Programação Orientada a Aspectos no Projeto de Sistemas Hardware Desenvolvidos com SystemC* defendida por Sérgio Queiroz de Medeiros e aprovada em 03 de março de 2006 em Natal, Rio Grande do Norte, pela banca examinadora constituída pelos professores:

Prof. Dr. David Boris Paul Déharbe
Universidade Federal do Rio Grande do Norte
Orientador

Prof. Dr. Ivan Saraiva Silva
Universidade Federal do Rio Grande do Norte

Profa. Dra. Edna Natividade da Silva Barros
Universidade Federal de Pernambuco
Membro Externo

"Eu não tinha este rosto de hoje,
assim calmo, assim triste, assim magro,
nem estes olhos tão vazios,
nem o lábio amargo.

Eu não tinha estas mãos sem força,
tão paradas e frias e mortas;
eu não tinha este coração
que nem se mostra.

Eu não dei por esta mudança,
tão simples, tão certa, tão fácil:
- Em que espelho ficou perdida
a minha face?"

Cecília Meireles

Agradecimentos

Ao povo lá de casa, o que inclui meu pai, Dudu, que não consegue entender esse povo que tem o terceiro grau, minha mãe, Doninha, minhas duas irmãs, Anna e Regina, e até mesmo a gata Bolotta.

A Janaína, que esteve ao meu lado durante este tempo.

Agradeço a Deus, apesar da pouca fé, a qual eu ainda busco.

A todos os meus amigos que conseguem me aguentar, em especial a Didi, que me ensinou a ser mais alegre, e ao ilustre CCCC.

Agradeço ao meu orientador David, com quem aprendi muito, que me ensinou a converter números binários e decimais, e que teve a paciência de me orientar ao longo de 5 anos. A todos os demais professores do DIMAp, em especial ao professor Ivan, que me ajudou a participar do meu primeiro congresso, e a todos os professores que tive ao longo da vida desde o Instituto Alvorada.

Resumo

Novos paradigmas de linguagens de programação vêem sendo comumente testados e geralmente são incorporados depois por linguagens de descrição de hardware. Recentemente, a programação orientada a aspectos (POA) provou ser útil na tentativa de melhorar a modularidade de linguagens estruturadas e orientadas a objeto tais como Java, C++ e C. Diante de tal fato, podemos esperar que o uso de POA pode melhorar o entendimento de sistemas hardware que estão sendo projetados, bem como tornar seus componentes mais reusáveis e fáceis de manter.

Iremos abordar então o uso de POA em aplicações desenvolvidas utilizando a biblioteca SystemC. Serão apresentados vários exemplos que ilustram o uso de POA juntamente com SystemC, mostrando alternativas e discutindo os seus benefícios.

Abstract

New programming language paradigms have commonly been tested and eventually incorporated into hardware description languages. Recently, aspect-oriented programming (AOP) has shown successful in improving the modularity of object-oriented and structured languages such Java, C++ and C. Thus, one can expect that, using AOP, one can improve the understanding of the hardware systems under design, as well as make its components more reusable and easier to maintain.

We apply AOP in applications developed using the SystemC library. Several examples will be presented illustrating how to combine AOP and SystemC. During the presentation of these examples, the benefits of this new approach will also be discussed.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução **14**

2 Programação Orientada a Aspectos **18**

2.1 Componente $\times$ Aspecto 20

2.2 *Crosscutting Concerns* 20

2.3 Criando Aspectos 21

2.4 Código de Comportamento Transversal 21

2.5 Pontos de Junção 22

2.6 Combinação de Aspectos 22

2.7 AspectC++ 23

2.7.1 Conjunto de Junção 26

2.7.1.1 Expressões de Casamento 26

2.7.2 O ponteiro *thisJoinPoint* 27

2.7.3 Criando um Aspecto 28

2.7.4 Código de Comportamento Transversal 29

2.7.5 Introduções 30

2.7.6 Ordenação de Comportamentos Transversais 33

2.8 O papel da POA na POO 34

3 SystemC **35**

3.1 Elementos Básicos de SystemC . . . 36

3.2 Exemplo de Uma Aplicação SystemC . . . 38

3.3 Interfaces e Canais . . . 40

3.4 Conclusão . . . 42

4 Exemplos da Utilização de Aspectos no Projeto de Sistemas Utilizando SystemC **43**

4.1 Modelagem de Hardware . . . 45

4.1.1 Modelando a Comunicação como um Aspecto . . . 45

Introdução . . . 45

Motivação . . . 45

Implementação . . . 46

Conclusão . . . 48

4.1.2 Política de Troca de Linhas da Cache . . . 49

Introdução . . . 49

Motivação . . . 49

Implementação . . . 49

Conclusão . . . 50

4.1.3 Separação do Controle e do Fluxo de Dados . . . 51

Introdução . . . 51

Motivação . . . 51

Implementação . . . 52

Conclusão . . . 54

4.2 Projeto/Processo . . . 54

4.2.1 Utilizando Aspectos para Realizar a Coleta de Métricas . . . 54

Introdução . . . 54

Motivação . . . 54

Implementação 55
Conclusão 56
4.2.2 Utilizando Projeto por Contrato com POA 57
Introdução 57
Motivação 57
Implementação 57
Conclusão 59
4.2.3 A Biblioteca de Verificação Padrão do SystemC e POA 60
Introdução 60
Motivação 62
Implementação/Conclusão 62
4.2.4 Usando POA com ArchC 63
Introdução 63
Motivação 63
Implementação/Conclusão 64
4.3 Análise de Desempenho 64
4.4 Conclusões 67

5 Conclusões **68**

Referências **71**

Apêndice A – O Protocolo OCP **75**

A.1 Características OCP 75
A.2 Sinais e Codificação 76
A.2.1 Sinais Básicos do Fluxo de Dados 77

Apêndice B – Implementação da Comunicação Utilizando OCP e Aspectos **80**

B.1 specs.h 80
B.2 memory.h 81
B.3 memory.cpp 82
B.4 application.h 84
B.5 application.cpp 85
B.6 slave.h 89
B.7 slave.cpp 90
B.8 master.h 90
B.9 master.cpp 91
B.10 OCP.ah 91
B.11 main.cpp 95

Lista de Figuras

1 Gráfico Ilustrando a Diferença de *time to market* entre Dois Produtos 15

2 Exemplo de Código do Componente misturado com o Código dos Aspectos . . 21

3 Exemplo de Combinação de Aspectos 23

4 Classe Empregado 24

5 Classe Logger 24

6 Classe Empresa 25

7 Arquivo *main* 25

8 Saída do Programa 26

9 Aspecto ALogger 28

10 Aspecto AIdade 31

11 Arquivo *main* modificado 33

12 Saída do Programa 33

13 Exemplo de um Módulo em SystemC 36

14 Exemplo de Relacionamento entre Módulos 37

15 Codificação do Exemplo da Figura 14 38

16 Módulo Produtor 39

17 Módulo Consumidor 40

18 Arquivo *main* do Exemplo Produtor/Consumidor 41

19 Exemplo de Módulos Conectados ao Canal através de uma Interface 42

20 Visão Geral do Fluxo de Projeto no Nível de Sistema 44

21 Conectando os módulos *Master* e *Application* 47

22 Visão Geral do Nosso Sistema Utilizando OCP e Aspectos 47

23 Conectando os módulos *Master* e *Application* 48

24 Aspecto LRU 50

25 Organização dos Módulos que Implementam a FFT 51

26 Nova Organização dos Módulos que Implementam a FFT 52

27 Aspecto FXPT 53

28 Aspecto refererente à Simulação do Sistema 56

29 Exemplo de uma Operação de Soma 58

30 Exemplo de um Contrato 59

31 Exemplo de Código SCV 61

32 Exemplo de Declaração de uma Classe de Restrições 61

33 Exemplo do Uso de uma Classe de Restrições 62

34 Decrição da Instrução *addiu* Utilizando ArchC 64

35 Exemplo de um Sistema Utilizando OCP 76

Lista de Tabelas

1 Desempenho das Aplicações sem usar POA 65

2 Desempenho das Aplicações usando POA 65

3 Desempenho da Aplicação 2 sem Otimização 66

4 Impacto da POA no Tempo de Simulação das Aplicações 66

5 Tamanho do Arquivo de Simulação Sem e Com o uso de POA 66

6 Impacto da POA no Tamanho do Arquivo de Simulação da Aplicações 67

7 Sinais OCP básicos 77

8 Codificação dos Comandos OCP 79

9 Codificação da Resposta do Escravo OCP 79

1 Introdução

Atualmente, a complexidade dos chips sendo projetados é bastante elevada e existe cada vez mais pressão para que um novo produto seja lançado rapidamente no mercado (BERGAMASCHI, 2002). Em virtude desta grande complexidade dos chips atuais, uma empresa que decidisse desenvolver sozinha todos os componentes do seu sistema gastaria um tempo considerável nesta tarefa, tempo este que não seria apropriado para o lançamento do produto no mercado (*time to market*), já que novas tecnologias são consumidas cada vez mais rapidamente, como mostrado pelo caso dos telefones celulares, onde uma nova linha de aparelhos era lançada a cada ano enquanto que hoje em dia já são lançadas duas novas linhas do produto, em média, a cada ano (KORDON; HENKEL, 2003).

Diantes destes fatos, torna-se imprescindível diminuir o tempo entre a concepção de uma idéia e o lançamento do seu produto correspondente no mercado, pois quanto mais lento for este processo, menor será a fatia de mercado destinada ao produto retardatário. A figura 1 ilustra um exemplo deste fenômeno. Nela, a idéia para dois novos produtos de fins similares, *A* e *B*, ocorre ao mesmo tempo e inicia-se o processo de incubação, onde se há um gasto de recursos para fabricar o produto. Acontece que o tempo de incubação do produto *A* é menor do que o do produto *B*, o qual será lançado depois e perderá uma fatia importante do mercado, que logo em seguida alcança o estágio de maturidade. O lucro resultante do produto *A* em virtude de seu rápido lançamento no mercado é indicado na figura.

Em virtude disto, a metodologia de desenvolvimento de sistemas em um único chip (*System-on-Chip* - SoC) vem adotando, cada vez mais, a abordagem de reuso de componentes de propriedade intelectual (*intellectual-property* - IP) pré-projetados (SAVAGE; CHILTON; CAMPOSANO, 2000), visando assim diminuir o intervalo de tempo entre o início do projeto e o lançamento do produto no mercado.

Durante o desenvolvimento de um componente de hardware são utilizadas linguagens de descrição de hardware (*hardware description languages* - HDLs), as quais são uma forma de descrever um sistema digital, como um computador ou um componente de um computador.

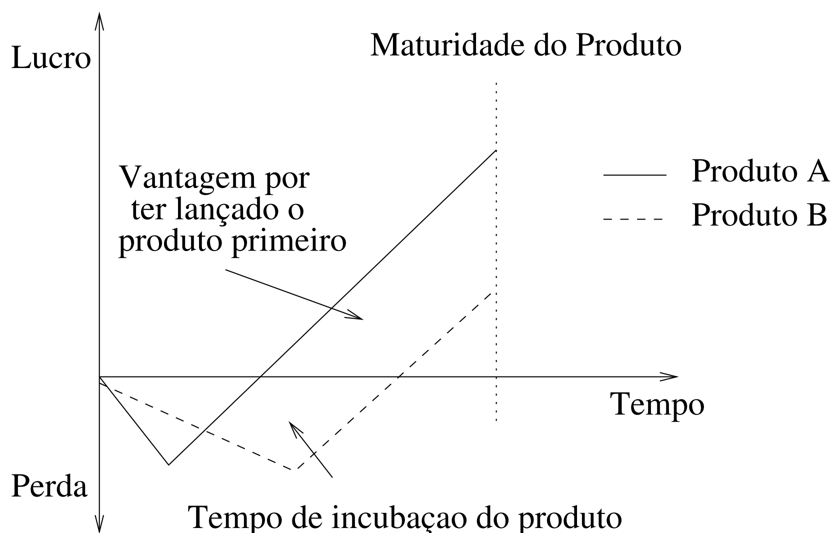

Figura 1: Gráfico Ilustrando a Diferença de *time to market* entre Dois Produtos

Dependendo da sua HDL, você pode descrever o seu sistema em vários níveis, variando de um modelo mais detalhado, descrevendo portas lógicas, até um mais abstrato.

Ao longo da história do desenvolvimento das linguagens de descrição de hardware (CHU et al., 1992; BORRIONE et al., 1992), é possível notar que estas linguagens possuem muitas características em comum com as linguagens de programação e que novos paradigmas utilizados em linguagens de programação são depois incorporados e utilizados com sucesso na área de projeto de hardware. Por exemplo, podemos notar a semelhança entre as linguagens de descrição de hardware **VHDL** e **Verilog** e as linguagens de programação **ADA** e **C**, respectivamente. **SystemC** e **SpecC**, que são baseadas, respectivamente, nas linguagens de programação **C++** e **C** são dois exemplos mais recentes deste fenômeno.

Desta forma, parece ser bastante útil ficar atento a evoluções nas linguagens de programação e ver como estes novos paradigmas que, porventura venham a aparecer, podem ser utilizados na área de projeto de hardware.

Atualmente, uma grande preocupação na área de linguagens de programação é tentar melhorar a expressividade do paradigma orientado a objeto, e um dos mais promissores entre os novos paradigmas que endereçam esta questão é a **Programação Orientada a Aspectos** (POA).

A POA é baseada na idéia de que as metodologias estruturada e orientada a objeto não oferecem meios suficientes para separar de forma clara o código com diferentes propósitos em um sistema, visto que alguns aspectos do sistema (tais como: geração de log, restrições de tempo-real, sincronização, etc.) não podem ser claramente encapsuladas em um único módulo do sistema (e.g. uma classe, um método, etc.). Esta limitação encontrada nos paradigmas

tradicionais leva a uma dificuldade extra para entender, manter e reutilizar o código onde estes aspectos se encontram. Por outro lado, fazendo uso da metodologia POA, um sistema pode ser projetado de tal forma que o código com difererentes propósitos torna-se encapsulado em diferentes unidades específicas, as quais podem ser combinadas depois seguindo algumas regras específicas (KICZALES et al., 1997).

O objetivo deste trabalho é mostrar o uso da POA no contexto do projeto de hardware usando a biblioteca *SystemC* como nossa linguagem de descrição de hardware e a ferramenta *AspectC++*, uma ferramenta que permite o uso da metodologia POA com a linguagem *C++*.

Em virtude de ser possível descrever tanto os componentes de hardware como os componentes de software de um sistema utilizando-se *SystemC*, é de se esperar que a utilização da POA deva trazer benefícios para o desenvolvimento dos componentes de software de um sistema, sendo necessária uma maior investigação para averiguar quais seriam os benefícios da utilização da POA também na modelagem dos componentes de hardware.

As evidências do uso de uma abordagem orientada a aspectos no projeto de hardware ainda são bastante tímidas. Entre as abordagens conhecidas até o momento, temos que (KASUYA; TESFAYE, 2005; VACHHARAJANI; VACHHARAJANI; AUGUST, 2004) requerem a utilização de novas linguagens durante o decorrer do projeto, as quais forneceriam então meios de utilizar, de forma limitida, o paradigma orientado a aspectos. Em uma das propostas (VACHHARAJANI; VACHHARAJANI; AUGUST, 2004), é possível realizar a instrumentação dos modelos construídos usando técnicas de POA. A outra proposta (KASUYA; TESFAYE, 2005) sugere a utilização de POA durante a fase de verificação do sistema.

Uma outra abordagem foi proposta por (CHEN WEIDONG QIU; PENG, 2004), indicando o uso de POA na análise da cobertura de teste de um modelo SystemC. Esta abordagem, contudo, é bastante simples e apenas apresenta o esqueleto de um aspecto que não faz uso da biblioteca SystemC.

Por fim, em (JAKACKI, 2003) é descrita uma abordagem usando aspectos para interceptar chamadas relacionadas com a comunicação entre módulos. Embora sejam apresentados alguns exemplos, a sintaxe de declaração de aspectos é desconhecida e não há nenhuma menção de qual foi a ferramenta utilizada para combinar o código dos aspectos com o código dos módulos SystemC.

A abordagem aqui apresentada não requer a introdução de uma nova linguagem para que se possa fazer uso da metologia de POA, além de não ser restrita a nenhuma fase/atividade particular do projeto do sistema, apresentando várias possibilidades para a utilização da POA

junto com SystemC.

Para ilustrar o uso conjunto de aspectos com SystemC iremos prover vários exemplos tentando modelar aspectos importantes de um sistema, envolvendo questões tais como simulação, comunicação e política de troca de linhas da cache. Abordaremos também o uso de POA durante a fase de verificação de um sistema.

Os capítulos seguintes deste texto encontram-se organizados da seguinte forma:

- O capítulo 2 aborda a programação orientada a aspectos. Neste capítulo é apresentada a teoria sobre a POA, envolvendo alguns conceitos chave desta abordagem. Também será apresentada a ferramenta AspectC++, que nos permitirá utilizar aspectos juntamente com a linguagem de programação C++;

- O capítulo 3 apresenta a biblioteca SystemC, que é utilizada no projeto de sistemas de hardware. Iremos apresentar uma noção geral sobre o funciomento da biblioteca e prover alguns exemplos que ilustrem o seu uso;

- O capítulo 4 mostra exemplos da utilização de aspectos juntamente com SystemC no projeto de sistemas. Serão abordados vários exemplos mostrando como a POA pode ser utilizada no projeto de sistemas hardware/software, além de ser feita uma análise sobre o impacto do uso da POA no tempo de simulação do sistema;

- O capítulo 5 apresenta uma análise dos resultados obtidos e delineia algumas conclusões.

2 *Programação Orientada a Aspectos*

Nas primeiras décadas da computação surgiu um paradigma que vem sendo praticado até os dias de hoje, principalmente entre usuários de linguagens como **C**, que é o paradigma da **Programação Estruturada** (WEINER, 1978). Tal paradigma faz o uso de procedimentos para realizar uma função específica, evitando assim a criação de grandes blocos monolíticos de código, o qual fica agora encapsulado dentro de um procedimento, de acordo com a sua função lógica. Esta abordagem também evita a repetição de código ao longo do programa, já que para ter determinado trecho de código executado basta-se fazer uma chamada ao procedimento correspondente.

Um novo avanço se deu com o surgimento da **Programação Orientada a Objetos** (POO), a qual tornou-se praticamente um padrão na maioria dos projetos, sendo este paradigma suportado por linguagens tais como **Smalltalk**, **C++** e **Java**. O padrão POO pode ser visto facilmente nas metodologias e ferramentas de desenvolvimento de software atuais onde temos: metodologias orientadas a objeto (OO), ferramentas de análise e projeto e linguagens de programação OO (ELRAD; FILMAN; BADER, 2001).

A POO alcançou então um grande sucesso na modelagem de problemas do mundo real, onde uma entidade (do mundo real) é modelada como sendo um objeto, o qual possui dados e métodos capazes de manipular estes dados, e aumentou a taxa de produtividade dos projetos, principalmente quando há uma ênfase no reuso (LEWIS et al., 1991). Devido a este sucesso, algumas vezes considera-se que a expressividade oferecida por este paradigma seja suficiente para representar de maneira adequada as propriedades do sistema sendo desenvolvido.

Porém, em muitas aplicações a metodologia OO não é capaz de isolar de forma adequada o código tratando de propriedades distintas do sistema que está sendo desenvolvido (ELRAD; FILMAN; BADER, 2001). Em tais sistemas, geralmente há questões como: sincronização, qualidade de serviço, segurança, persistência, geração de logs, etc.

Pode-se notar uma distinção no código de tais sistemas, onde parte dele diz respeito à funcionalidade específica da aplicação sendo desenvolvida (um sistema bancário, um sistema de

reserva de passagens, etc.) e a outra parte não está relacionada com a funcionalidade do sistema em si, mas com um determinado aspecto que é necessário para o funcionamento do sistema de acordo com o desejado (controle do acesso a uma região crítica, sincronização, geração de logs, etc.).

Na POO, estes trechos de código ficam misturados ao longo de todo o sistema, tornando mais difícil o entendimento do mesmo, já que o código tratando desses diversos interesses deve ser entrelaçado.

Este entrelaçamento de interesses não é bom, já que há vários benefícios quando se possui um interesse importante de um sistema (geração de logs, sincronização, etc.) expresso de forma simples e bem localizado em uma única seção de código. Isto possibilita um entendimento mais fácil deste código, visto que ele não está espalhado por todo o sistema, possivelmente em diferentes arquivos, além de que ele não se encontra misturado com outros interesses. Desta forma, este código torna-se mais fácil de analisar, modificar, manter e reusar (CZARNECKI; EISENECKER, 2000).

O primeiro a notar a importância em se lidar com um interesse do sistema de cada vez foi Dijkstra, que intitulou tal abordagem de princípio de separação de interesses (*separation of concerns*) (DIJKSTRA, 1976).

A crescente importância que o software conseguiu a partir da década de 1990, onde objetiva-se reduzir o custo das fases de desenvolvimento de software (definição, desenvolvimento, manutenção) (PRESSMAN, 1996), e a persistência da chamada "crise do software" (JOHNSON, 1996) (dificuldade de escrever software correto e compreensível, necessário para os problemas computacionais cada vez mais complexos que iam surgindo), deram ensejo para o surgimento de várias novas abordagens, as abordagens pós-OO, as quais são uma área de pesquisa muito explorada atualmente (ELRAD; FILMAN; BADER, 2001).

Exemplos de tais abordagens incluem: programação orientada a sujeito (*subject-oriented programming*) (HARRISON; OSSHER, 1993); filtros de composição (*composition filters*) (BERGMANS; AKSIT, 2001); programação adaptativa (*adaptive programming*) ou programação de Demeter (LIEBERHERR, 2000); reflexão (MURPHY; NOTKIN; SULLIVAN, 1995) e metaprogramação (BARTLETT, 2005), etc. Um bom conjunto destas novas abordagens pode ser encontrado em (CZARNECKI; EISENECKER, 2000).

Uma das abordagens pós-OO que mais tem se destacado é a **Programação Orientada a Aspectos (POA)**, ou *Aspect-Oriented Programming* (KICZALES et al., 1997; ELRAD; FILMAN; BADER, 2001), sobre a qual iremos falar neste capítulo.

2.1 Componente × Aspecto

Na POA são definidos dois termos básicos: o **componente**, que engloba as propriedades do sistema cuja implementação pode ser apropriadamente encapsulada como uma entidade específica do sistema; e o **aspecto**, o qual possui propriedades do sistema que não podem ser bem encapsuladas em uma entidade específica utilizando uma abordagem estruturada ou orientada a objeto (KICZALES et al., 1997).

2.2 *Crosscutting Concerns*

Em vários sistemas existem algumas propriedades que não dizem respeito à funcionalidade básica do mesmo, mas que são necessárias para que o sistema seja implementado de forma correta. Tais propriedades são denominadas de requisitos não funcionais do sistema e muitas vezes o código referente à sua implementação encontra-se espalhado e misturado com o código que diz respeito à funcionalidade básica do sistema, o que faz com que o código resultante seja de difícil compreensão e manutenção.

Crosscutting concerns, ou **interesses transversais**, são partes de código que implementam funcionalidades específicas e que afetam diferentes partes do sistema, desta forma, para que um determinado interesse seja implementado, é necessário *"atravessar"* vários módulos do sistema.

A figura 2 ilustra a estrutura de um exemplo de um código onde ocorre *crosscutting concerns*, o que acaba produzindo os seguintes problemas:

- o entendimento do código fica mais difícil, uma vez que temos códigos com propósitos distintos misturados;
- como o código referente a um aspecto não é apropriadamente encapsulado em abordagens como a POO, ele encontra-se espalhado por todo o sistema, provavelmente em diferentes arquivos e classes, desta forma, uma alteração no código do aspecto provocará modificações em vários lugares, tornando com isso difícil a manutenção/evolução do sistema, que é a parte responsável pela maior parcela de custos no processo de desenvolvimento de software (PRESSMAN, 1996);
- visto que o código funcional está misturado com o não funcional, a reusabilidade do componente fica prejudicada caso desejarmos fazer uso da sua funcionalidade básica num contexto diferente;

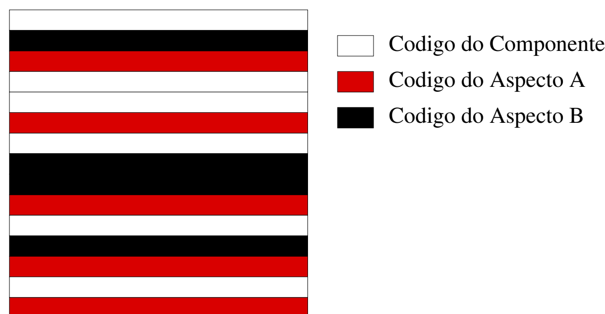

Figura 2: Exemplo de Código do Componente misturado com o Código dos Aspectos

- com o código misturado, o aspecto que desejamos implementar (sincronização, geração de logs, etc.) encontra-se implicitamente representado de uma maneira entrelaçada, quando o mais adequado seria possuirmos uma representação explícita e bem localizada deste aspecto.

2.3 Criando Aspectos

A POA propõe que o código que não diz respeito à funcionalidade básica da aplicação seja implementado separadamente e encapsulado em uma entidade chamada de **aspecto**. Através desta abordagem é possível obter-se uma maior modularidade, bem como pode-se evitar o problema do espalhamento do código referente ao aspecto ao longo de todo o sistema, visto que agora ele está definido em um único lugar.

Após esta separação, o código responsável pela funcionalidade básica da aplicação torna-se mais simples de entender, uma vez que não se encontra misturado com o código dos aspectos, além de tornar-se mais adequado para ser reusado em um contexto diferente.

Deve-se notar que embora os códigos do componente e do aspecto estejam separados/isolados, ainda devem ser fornecidos mecanismos que permitam ao código do aspecto ter acesso a objetos/valores do código do componente.

2.4 Código de Comportamento Transversal

Agora que temos o código do componente e o código do aspecto separados, nós iremos chamar de *código de comportamento transversal* o código do aspecto que adicionará comportamento ao código do componente, ou seja, o código de comportamento transversal é o código do aspecto que irá estender a funcionalidade básica da aplicação, adicionando novas propriedades.

2.5 Pontos de Junção

Para produzirmos o sistema desejado, será necessário em algum momento unir o código do aspecto com o código do componente, obtendo assim o sistema completo. Para tal fim, são definidos pontos de junção (*join points*), que são os locais no código do componente onde os aspectos podem interferir, portanto um ponto de junção é um ponto bem definido no fluxo de execução de um programa.

Um ponto de junção indica um lugar no código do componente onde desejamos inserir um novo comportamento. Exemplos de pontos de junção são: uma chamada a uma função, uma declaração de variável, o construtor de uma classe, etc.

Os aspectos podem adicionar comportamento antes ou depois do ponto de junção, bem como obter o controle completo sobre o ponto de junção, executando o código do aspecto no lugar do código original do ponto de junção.

Devemos observar que para a utilização efetiva da POA a uma linguagem é necessário que se tenha um mecanismo capaz de definir um bom conjunto de pontos de junção, logo, uma ferramenta que possibilite o uso da POA deve ser bastante flexível para permitir a descrição de um vasto conjunto de pontos de junção.

2.6 Combinação de Aspectos

Os pontos de junção definidos serão utilizados por uma ferramenta, chamada de *aspect weaver*, a qual será responsável pela união do código do componente com o código dos aspectos, em um processo chamado de combinação de aspectos (*aspect weaving*), como encontra-se ilustrado na figura 3. Ao final deste processo teremos então o código completo do sistema, com o código dos aspectos sendo inserido no código do componente conforme definido pelos pontos de junção.

O código do aspecto pode ser alterado sempre que se fizer necessário. Após as alterações serem efetuadas, o *aspect weaver* deve ser executado de forma que seja gerada uma nova versão do sistema com os aspectos desejados a partir do código dos componentes e do código dos aspectos a ele fornecidos. Para linguagens interpretadas, há a possibilidade de alterar o conjunto de junção em tempo de execução (FERNANDES, 2004).

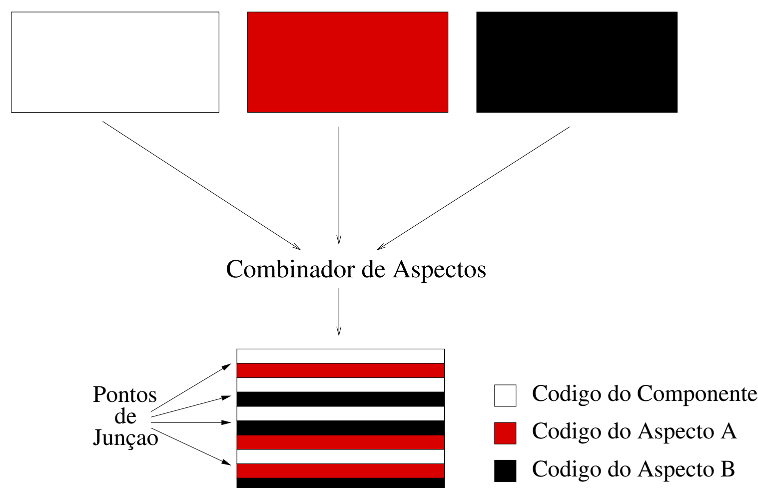

Figura 3: Exemplo de Combinação de Aspectos

2.7 AspectC++

Abordaremos a seguir alguns aspectos fundamentais de AspectC++ (SPINCZYK; GAL; SCHRöDER-PREIKSCHAT, 2002; ASPECTC++, 2006; pure-systems GmbH E OLAF SPINCZYK, 2004), uma extensão orientada a aspectos para a linguagem *C++*.

Os conceitos básicos são bastante semelhantes aos de outra ferramenta, AspectJ (KICZALES et al., 2001; ASPECTJ, 2006), que já se encontra bem mais difundida. Isto é uma tentativa de estimular o uso de AspectC++ por desenvolvedores já acostumados a trabalhar com AspectJ que necessitem trabalhar com aplicações C/C++.

A ferramenta AspectC++ não se encontra ainda, segundo os seus próprios desenvolvedores, em uma implementação suficientemente madura, de modo que a ferramenta não funciona de forma esperada em algumas situações, como é o caso do uso de padrões de classes (*templates*) e de macros, onde a ferramenta consegue lidar apenas com um subconjunto das situações possíveis. AspectC++ evoluiu bastante durante o decorrer do presente trabalho e acredita-se que se encontra perto do lançamento da sua versão *1.0* estável.

Para compilar os exemplos utilizando SystemC e aspectos, iremos utilizar AspectC++ do mesmo modo que usaríamos um compilador C/C++ convencional para compilar um módulo SystemC, a única modificação é que poderemos ou não indicar para AspectC++ qual(is) aspectos irão agir sobre um determinado módulo SystemC.

Durante a apresentação da ferramenta iremos trabalhar com um pequeno exemplo que consiste em uma empresa que contrata empregados e gera um log indicando que um novo empregado foi contratado.

```
1  class Empregado {
2    public:
3
4      Empregado () {}
5
6      void setNome (string n) { nome = n; }
7
8      string getNome () { return nome; }
9
10     int getSalario () { return salario; }
11
12     void setSalario (int s) { salario = s; }
13
14   private:
15     string nome;
16     int salario;
17 };
```

Figura 4: Classe Empregado

Na figura 4 podemos ver a nossa classe *Empregado*, que representa um empregado que possui os atributos *nome* e *salario*, bem como alguns métodos para ler e atribuir valor a estes atributos.

A figura 5 nos mostra a classe *Logger*, a qual possui um método que deverá gerar um log a cada nova contratação de um funcionário pela empresa.

A classe *Empresa* pode ser vista na figura 6. Esta classe possui um vetor de empregados e um método que permite a contratação de novos empregados. A classe *Empresa* possui um objeto do tipo *Logger* para registrar um log toda vez que um novo funcionário é contratado.

Devemos observar que a função de log não está relacionada com a funcionalidade básica da classe *Empresa*, nós poderíamos realizar a contratação de um funcionário e não gerar um log para registrar tal fato, portanto o log é uma característica adicional da classe. Devemos notar também que caso desejássemos gerar um log quando tomássemos outras ações, como por exemplo demitir um funcionário ou dar um aumento a um empregado, os respectivos métodos que

```
1  class Logger {
2    public:
3      Logger () {}
4
5      void logContrata (string s) {
6        cout << "Contratou empregado: " << s <<  endl;
7      }
8  };
```

Figura 5: Classe Logger

```
1  class Empresa {
2    public:
3      Empresa (string n) {
4        nome = n;
5      }
6
7      void contrata (Empregado e) {
8        empregados.insert (empregados.end(), e);
9        logger.logContrata (e.getNome());
10     }
11
12   private:
13     string nome;
14     vector <Empregado> empregados;
15     Logger logger;
16 };
```

Figura 6: Classe Empresa

```
1  int
2  main(int argc, char **argv) {
3    Empresa company ("CCCC");
4    Empregado emp;
5
6    emp.setNome ("Joao");
7    emp.setSalario (300);
8
9    company.contrata (emp);
10
11   emp.setNome ("Reichstul");
12   emp.setSalario (3000);
13
14   company.contrata (emp);
15
16   return 0;
17 }
```

Figura 7: Arquivo *main*

implementassem estas ações também teriam que fazer chamadas a métodos da classe *Logger*. Em tal situação, teríamos várias linhas de código, que não estão associadas com o funcionamento básico da nossa classe *Empresa*, espalhadas pelo nosso programa e caso desejássemos ter uma classe que não gerasse mais log, teríamos que localizar todos estes pontos e apagá-los ou comentá-los para então obter a nossa nova classe *Empresa* sem geração de log. Desse modo, podemos ver que o log é um comportamento transversal e que ele poderia ser modelado como um aspeto.

Por fim, a figura 7 nos mostra o nosso arquivo *main*, onde declaramos um objeto do tipo *Empresa* e contratamos dois empregados. O resultado da execução deste programa pode ser visto na figura 8.

```
Contratou empregado: Joao
Contratou empregado: Reichstul
```

Figura 8: Saída do Programa

2.7.1 Conjunto de Junção

Aspectos em AspectC++ implementam o conceito de interesses transversais de uma maneira modular. Tendo isto em mente, o elemento mais importante da linguagem AspectC++ é o conjunto de junção (*pointcut*), o qual descreve um conjunto de pontos de junção determinando sobre que condições um aspecto deve ter efeito, ou seja, um conjunto de junção descreverá os pontos no fluxo de execução de um programa nos quais o código do aspecto deve ser executado.

Desse modo, cada ponto de junção pode se referir ou a uma função, um atributo, um tipo, uma variável, ou a um ponto a partir do qual um ponto de junção é acessado, de modo que esta condição pode ser, por exemplo, o evento de alcançar um posição de código determinada. Dependendo do tipo do conjunto de junção, ele será avaliado em tempo de compilação ou em tempo de execução.

2.7.1.1 Expressões de Casamento

Para descrever um conjunto de junção, nós fazemos uso de expressões de casamento e de algumas funções pré-definidas. Uma expressão de casamento pode ser entendida como um padrão de busca. Em tal padrão de busca o caractere especial **%** é interpretado como um coringa para nomes ou partes de uma assinatura. A seqüência de caracteres **...** casa qualquer número de parâmetros em uma assinatura de função ou qualquer número de escopos em um nome qualificado. Uma expressão de casamento é uma string entre aspas.

As expressões de casamento podem ser utilizadas tanto para o casamento de funções como para o casamento de tipos, como uma classe por exemplo.

Na hora de realizar o casamento de uma função, a expressão de casamento em questão é internamente decomposta em três partes: uma referente ao tipo, outra referente ao escopo, e uma outra referente ao nome. Supondo que temos a expressão de casamento a seguir:

"const % Company::...::contrata% (string)"

Vamos agora analisar cada parte da expressão de casamento:

- **nome**: o nome da função deve casar com o padrão *contrata%*, ou seja, todos os métodos que começam com a cadeia *contrata*;

- **escopo**: o escopo no qual a função é definida, ou seja, o seu ambiente de nomes (*namespace*), deve casar com *Company::...::*, que pode ser entendido como todas as classes que pertencem ao ambiente de nomes *Company*;

- **tipo**: o tipo da função tem que casar com *const % (string)*, isto quer dizer que o valor de retorno deve ser uma constante e que o argumento do método deve ser uma *string*.

Para classes e outros tipos esta decomposição não é necessária. Por exemplo, a expressão *"Company::Empresa"* é suficiente para descrever uma classe, porque este é o mesmo nome da classe.

A seguir, são apresentados mais alguns exemplos de expressões de casamento:

- *"void %::print ()"*: casa todos os métodos *print*, sem argumentos e sem valor de retorno, de qualquer classe;

- *"void Pessoa::set% (...)"*: casa todos os métodos da classe *Pessoa* com tipo de retorno *void*, com qualquer número de argumentos, cujo nome inicie com *set*;

- *"int funcao (int, %)"*: casa o método *funcao* com exatamente dois argumentos, onde o primeiro deve ser um *int* e o segundo pode ser de qualquer tipo, e o valor de retorno deve ser do tipo *int*.

2.7.2 O ponteiro *thisJoinPoint*

Utilizando AspectC++ é possível obter informação sobre o ponto de junção atual que foi alcançado. Para este fim, podemos utilizar o ponteiro *thisJoinPoint*, ou sua forma abreviada *tjp*. O ponteiro *tjp* possui uma série de funções associadas que nos fornecem informações sobre o ponto de junção atual, abaixo estão listadas algumas destas funções:

- **result ()**
  retorna um ponteiro para a posição de memória designada para ser o valor resultado de uma função, ou zero se a função não possui valor de retorno;

- **arg (int number)**
  retorna um ponteiro para a posição de memória que contém o valor de um argumento da função, onde o índice do argumento da função é indicado pela variável *number*;

```
1  aspect ALogger {
2
3    pointcut contratacao() = "% Empresa::contrata(...)";
4
5    advice execution (contratacao()) : after () {
6      cout << "Contratou empregado: " << ((Empregado *) tjp->arg(0)) -> getNome() << endl;
7    }
8
9  };
```

Figura 9: Aspecto ALogger

- **that ()**
  retorna um ponteiro para o objeto que está iniciando a chamada, ou zero se ele é um método estático ou uma função global;
- **target ()**
  retorna um ponteiro para o objeto que é o alvo da chamada, ou zero se ele é um método estático ou uma função global;
- **proceed ()**
  executa o código original do ponto de junção em um código de comportamento transversal do tipo *around*.

2.7.3 Criando um Aspecto

Agora que já vimos o conceito de conjunto de junção e de expressões de casamento, podemos apresentar a criação de aspectos em AspectC++. A declaração de um aspecto é formulada como uma extensão ao conceito de classe em C++. A estrutura básica de uma declaração de aspecto é exatamente a mesma de uma declaração C++ de uma classe, estrutura ou união, a diferença é que devemos utilizar a palavra-chave *aspect*. Desta forma, aspectos podem ter atributos e métodos, bem como podem herdar de classes e até mesmo de outros aspectos.

Como vimos anteriormente, o registro de logs do nosso sistema poderia ser modelado como um aspecto, de forma que a classe *Empresa* tivesse apenas código relacionado com sua funcionalidade básica e que pudéssesmos reunir o código referente ao registro de log em um único lugar.

Podemos ver a declaração do nosso aspecto *ALogger* na figura 9.

Na linha 1, é declarado o nosso aspecto, da mesma forma como fazemos com uma classe em C++. Na linha 3, declaramos um conjunto de junção, fazemos isso usando a palavra-chave

pointcut, e damos a ele o nome de *contratacao()*. Resolvemos dar um nome ao conjunto de junção pois caso desejemos utilizar o mesmo conjunto de junção novamente basta usar o nome que demos a ele, ao invés da expressão de casamento inteira, que pode vir a ser um pouco longa. A nossa expressão de casamento descreve o método *contrata*, da classe *Empresa*, com qualquer número de parâmetros e com qualquer valor de retorno. A seguir, estudaremos melhor o significados das linhas 5, 6 e 7.

2.7.4 Código de Comportamento Transversal

O código de comportamento transversal (*advice code*) pode ser entendido como uma ação ativada por um aspecto quando um ponto de junção correspondente em um programa é alcançado. A ativação do código de comportamento transversal pode acontecer antes, depois, ou antes e depois do ponto de junção que foi alcançado. O elemento da linguagem AspectC++ usado para especificar código de comportamento transversal é a declaração de códigos de comportamento transversal, a qual é introduzida pela palavra-chave *advice* seguida por uma expressão de conjunto de junção definindo onde e sob que condições o código de comportamento transveral deve ser ativado.

Na linha 5 da figura 9 temos uma declaração de código de comportamento transversal. A expressão de conjunto de junção *execution (contratacao())* descreve todas as execuções dos pontos de junção descritos por *contratacao()*, ou seja, descreve todas as execuções do método *contrata* da classe *Empresa*.

A função pré-definida de conjunto de junção *execution* faz com que o fluxo de execução normal do programa seja desviado quando da execução de um método descrito por *contratacao()*, neste caso, quando da execução do método *contrata*. Outra função pré-definida utilizada para interceptar chamadas a métodos é *call*, que faz com que o fluxo de execução seja desviado quando a chamada a um método é realizada, porém antes da execução do mesmo, ou seja, o programa é desviado antes de entrar no contexto de execução do método. Dessa forma, *call* pode ser usado, por exemplo, para casar uma chamada a uma função virtual pura, mas como uma função virtual pura não possui um código executável associado, não é possível usar *execution* para casar tal função.

Ainda na linha 5 do nosso exemplo, o fragmento de código *": after"* após a expressão de conjunto de junção determina que o código de comportamento transversal deve ser ativado exatamente **depois** do código relacionado com o ponto de junção ser alcançado. Existem também as expressões *": before"*, que significa **antes** de alcançar o código do ponto de junção; e *": around"*, que significa que o código de comportamento transversal deve ser executado no lugar

do código descrito pelo ponto de junção.

Em um comportamento transversal do tipo *around* o código do comportamento transversal pode explicitamente executar o código original do programa naquele ponto de junção, de forma que o código de comportamento transversal poderia ser executado **antes** e **depois** do ponto de junção. O código de comportamento transversal não possui permissões de acesso especiais ao código do programa no ponto de junção.

Além do que já foi dito, os conjuntos de junção também podem ter acesso ao valor das variáveis no contexto do ponto de junção, fazendo uso do ponteiro *tjp*. Desta forma, por exemplo, podemos acessar a partir do código de comportamento transversal o valor dos argumentos de uma chamada de função.

É exatamente isto que fazemos na linha 6 do nosso exemplo onde desejamos imprimir, após a mensagem *"Contratou empregado: "*, o nome do empregado que acabou de ser contratado. O empregado que acabou de ser contratado é passado como o primeiro parâmetro da função *contrata* e para acessá-lo usamos o ponteiro *thisJoinPoint* (*tjp*), que é um ponteiro para o ponto de junção que foi ativado. Através de *tjp* podemos utilizar a função *arg*, que retorna um ponteiro para a posição de memória que contém o valor de um argumento da função, de acordo com o índice que fornecemos. Como queremos obter um ponteiro para o primeiro parâmetro da função *contrata*, então iremos passar o valor zero como parâmetro.

Agora que já conseguimos obter o empregado que acabou de ser contratado, iremos imprimir o seu nome. Como já dissemos, o código de comportamento transversal não possui permissões de acesso especiais ao código do programa no ponto de junção, em virtude disso iremos chamar o método público *getNome* da classe *Empregado*, que nos retornará o nome do empregado.

Agora, temos o nosso aspecto *ALogger* responsável pelo log e a classe *Empresa* está livre do código relacionado com o registro de log.

2.7.5 Introduções

O segundo tipo de código de comportamento transversal suportado por AspectC++ são as introduções. Introduções são usadas para estender o código do programa e estruturas de dados em particular. Da mesma forma que uma declaração de comportamento transversal, a introdução é declarada pela palavra-chave *advice*. A seguir, deve vir um conjunto de junção e o *token* ":". A declaração que vem a seguir é então introduzida nas classes e nos ambientes de nomes (*namespaces*) descritos pelo conjunto de junção. O código introduzido pode então ser

```
1  aspect AIdade {
2
3    pointcut empregado() = "Empregado";
4
5    advice empregado () : int idade;
6
7    advice empregado () : public: void setIdade (int i) {
8      idade = i;
9    }
10
11   advice empregado () : public: int getIdade () {
12     return idade;
13   }
14
15   pointcut contratacao () = "% Empresa::contrata (...)";
16
17   advice execution (contratacao()) : around () {
18     Empregado *e = (Empregado *) tjp->arg (0);
19     if (e->getIdade() < idadeMin)
20       return;
21     tjp->proceed ();
22   }
23
24   public:
25     AIdade () { idadeMin = 30; }
26
27   private:
28     int idadeMin;
29 };
```

Figura 10: Aspecto AIdade

usado no código normal do programa como qualquer outra função, atributo, etc.

O código de comportamento transversal em introduções possui as mesmas permissões de acesso do que o código no ponto de junção correspondente, ou seja, um método introduzido em uma classe tem acesso a todos os membros daquela classe, inclusive os privados.

Voltando para o exemplo da nossa empresa, suponha que os diretores resolveram estabelecer uma nova política de contratação de novos funcionários. Esta nova política será baseada na idade dos funcionários e a empresa não contratará ninguém que possua uma idade abaixo de uma idade mínima estipulada, para garantir que todos os seus novos funcionários são profissionais experientes.

Seguindo este princípio, vamos definir um novo aspecto, chamado *AIdade*, que pode ser visto na figura 10.

Na linha 1, assim como no aspecto *ALogger*, declaramos o nosso aspecto *AIdade*, o qual conterá o código relacionado com a nova política de contratação da empresa baseada na idade

dos prováveis novos funcionários.

Na linha 3, é declarado um conjunto de junção que descreve a classe *Empregado*. Visto que um empregado será contratado ou não com base na sua idade, faz-se necessário incluir um novo atributo à classe *Empregado* e isso é feito na linha 5, através do uso de introduções. A declaração que segue os dois pontos, que no caso é *"int idade;"*, será introduzida na classe *Empregado*. Precisamos também de métodos para obter e atribuir o valor da idade do empregado, e novamente usamos introduções nas linhas $7-10$ e $11-13$. Poderíamos também ter introduzido novos métodos para operar sobre quaisquer outros atributos da classe *Empregado* além da *idade*.

A seguir, na linha 17, definimos um conjunto de junção igual ao definido no aspecto *ALogger*, de forma que ele descreve o método *contrata* da classe *Empresa*.

Nas linhas $17-22$, definimos um código de comportamento transversal. Note que a declaração deste código de comportamento transversal se assemelha com aquela declarada no aspecto *ALogger*, observem porém a palavra-chave *around* no fim da linha 17, ela indica que o código que será declarado a seguir deve ser executado no lugar do código original do ponto de junção.

Na linha 18, novamente usamos o ponteiro *tjp* para obter o primeiro argumento do método. Em seguida, testamos se o candidato atual possui uma idade maior do que a idade mínima necessária para poder ser contratado pela empresa. Caso ele não possua idade suficiente, simplesmente terminamos a execução do método. Se o candidato possuir a experiência necessária, fazemos uma chamada ao código original do método que inserirá o novo empregado na lista de empregados e chamará a função de log.

Como dissemos anteriormente, a declaração de um aspecto é semelhante a de uma classe e isto pode ser visto no nosso aspecto *AIdade*, que possui um atributo chamado *idadeMin* e um construtor, que inicializa *idadeMin*.

Em virtude do novo aspecto do nosso sistema o arquivo *main* sofrerá algumas modificações, podemos ver o novo arquivo *main* na figura 11. As modificações foram as chamadas ao método *setIdade* para atribuir uma idade ao empregado. Podemos notar que somente o primeiro empregado possui a idade mínima, portanto só ele será contratado pela empresa. A saída deste programa pode ser vista na figura 12.

O mecanismo de introduções mostrado aqui possui relação com o mecanismo de herança, podendo inclusive ser usado juntamente com este. Com um única introdução, é possível modificar várias classes, inserindo dados ou modificando o comportamente destas classes. Existe

```
1  int
2  main(int argc, char **argv) {
3    Empresa company ("CCCC");
4
5    Empregado emp;
6
7    emp.setNome ("Joao");
8    emp.setSalario (300);
9    emp.setIdade (30);
10
11   company.contrata (emp);
12
13   emp.setNome ("Reichstul");
14   emp.setSalario (3000);
15   emp.setIdade (29);
16
17   company.contrata (emp);
18
19   company.relatorio ();
20
21   return 0;
22 }
```

Figura 11: Arquivo *main* modificado

```
Contratou empregado: Joao
```

Figura 12: Saída do Programa

também um tipo especial de introdução, a introdução de classe base, a qual permite estender a lista de classes base de uma ou mais classes.

2.7.6 Ordenação de Comportamentos Transversais

Se mais de um comportamento transversal afetar o mesmo ponto de junção pode ser necessário definir uma ordem para a execução dos comportamentos transversais caso exista uma dependência entre os códigos dos comportamentos transversais (interação de aspectos). O exemplo a seguir mostra como a precedência de código de comportamento transversal pode ser definida em AspectC++.

- *advice execution("void send(...)") : order("Encrypt", "Log");*
  Se o comportamento transversal de ambos os aspectos, *Encrypt* e *Log*, deve ser executado quando a função *send(...)* é executada, esta declaração de ordem define que o comportamento transversal *Encrypt* possui maior precedência e portanto deve ser executado primeiro.

2.8 O papel da POA na POO

O surgimento da POA não quer dizer que a POO esteja obsoleta e deva ser esquecida. Porém, como já foi enfatizado, o uso da POO sozinha muitas vezes não é uma metologia eficaz, já que produz um modelo insatisfatório do sistema. Dessa forma, o uso da POA vem auxiliar numa melhor separação de interesses do sistema, endereçando alguns dos pontos fracos da POO. Deve-se pensar então, em um uso conjunto destas tecnologias, escolhendo-se a que for mais apropriada para o problema em questão (ELRAD et al., 2001).

Podemos dizer ainda que, da mesma forma que a POO faz uso dos conceitos propostos pela Programação Estruturada, porém adicionando novos elementos, capazes de prover um maior encapsulamento dos dados e reuso do código, a POA baseia-se na POO, provendo mecanismos adicionais que proporcionam uma melhor separação de interesses.

3 SystemC

SystemC (SYSTEMC, 2005; FIN et al., 2001; (OSCI), 2005) é uma biblioteca de classes e padrões de classes implementada em *C++* que permite ao seu usuário utilizar elementos típicos de *hardware*, tais como portas e sinais, dentro do contexto de *C++*. Dessa forma, SystemC permite a utilização de recursos avançados de *C++*, tais como os padrões de classes (*templates*).

Um dos maiores benefícios que o projetista de um sistema pode usufruir com o uso de SystemC é que todo o modelo do sistema pode ser descrito em uma única linguagem, ou seja, SystemC pode ser utilizado tanto para descrever os componentes de *software* como os componentes de *hardware*, o que facilita a co-verificação de sistemas *hardware-software* (LIAO, 2000).

Uma outra vantagem da utilização de SystemC é que podemos construir modelos mais abstratos do sistema, principalmente nas fases iniciais do projeto quando questões específicas de implementação ainda não foram abordadas, e a partir do nosso modelo abstrato do sistema realizar refinamentos para obter um modelo o mais próximo possível da implementação (PANDA, 2001). Desta forma, é possível construir modelos do sistema em diferentes níveis tais como: nível de transferência entre registradores ou *register transfer level* (RTL), nível de comportamento e nível de sistema. Todos estes modelos podem ser construídos utilizando apenas SystemC e podem compartilhar os mesmos bancos de teste (*test benches*), o que agiliza o processo de desenvolvimento de novos componentes.

Além dos benefícios já mencionados, SystemC também trouxe novos mecanismos de abstração e organização de um projeto típicos de *C++*, como polimorfismo e herança, que não possuíam representações equivalentes nas linguagens de descrição de hardware (*hardware description languages*) (HDLs) anteriores e que possibilitaram explorar de forma mais eficiente o espaço de projeto tanto no nível RTL como no nível arquitetural (CHAREST; ABOULHAMID, 2002).

```
SC_MODULE (MeuModulo) {
   sc_in <bool> entrada, clock;
   sc_out <int> saida;
   sc_signal <int> meuSinal;

   void imprime ();
   void atualiza ();

   SC_CTOR (MeuModulo) {
      SC_METHOD (atualiza);
      sensitive << clock;
   }
};
```

Figura 13: Exemplo de um Módulo em SystemC

3.1 Elementos Básicos de SystemC

Os componentes básicos da biblioteca SystemC são os módulos. A descrição de um sistema utilizando SystemC consiste de um conjunto de módulos conectados através de portas de entrada/saída e sinais.

Um módulo pode ser declarado utilizando-se a macro *SC_MODULE* seguida pelo nome do módulo. Podemos comparar um módulo com uma classe ou estrutura padrão em *C++*, com o módulo possuindo atributos e métodos, pois a macro *SC_MODULE* será depois expandida para a declaração de uma classe que herda da classe *sc_module*, a qual é a classe base de todas as entidades estruturais do SystemC. Na figura 13 podemos ver um exemplo de módulo em SystemC.

Neste exemplo nós declaramos um módulo, chamado *MeuModulo*, que possui as seguintes variáveis:

- **entrada**: é uma porta de entrada, como indicado pelo padrão de classe *sc_in*, que assume valores booleanos;
- **clock**: é uma porta de entrada que deverá ser conectada com o relógio do sistema;
- **saida**: é uma porta de saída, como indica o padrão de classe *sc_out*, do tipo *int*;
- **meuSinal**: é um sinal, como indica o padrão de classe *sc_signal*, do tipo *int*.

Além destas variáveis nosso módulo possui os métodos *imprime* e *atualiza*. A seguir nós declaramos o construtor do nosso módulo utilizando a macro *SC_CTOR*. Toda vez que declaramos uma instância de um objeto em SystemC devemos passar um nome como argumento do

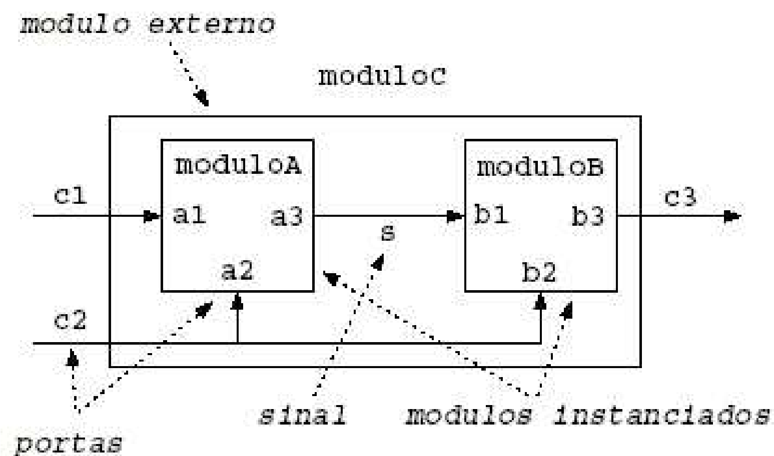

Figura 14: Exemplo de Relacionamento entre Módulos

construtor, de modo que cada instância possui um nome único. A macro *SC_CTOR* será então expandida de forma a possibilitar o gerenciamento dos nomes dos objetos e a sua hierarquia.

No construtor utilizamos a macro *SC_METHOD*, a qual indica que um método deve ser ativado quando ocorrer uma mudança em algum dos elementos aos quais ele é sensível. Em seguida, informamos que o método *atualiza* é sensível à variável *clock*, de modo que quando esta variável tiver o seu valor alterado o método *atualiza* será executado.

O módulo *MeuModulo* pode se comunicar com outros módulos através de suas portas de entrada/saída que devem estar conectadas através de sinais aos módulos com os quais o módulo *MeuModulo* deseja se comunicar. Um módulo também pode conter instâncias de outros módulos, construindo assim uma hierarquia entre os módulos. A figura 14 mostra um possível exemplo do relacionamento entre módulos.

Nesta figura temos três módulos:

- **Módulo A**: que possui as portas de entrada *a1* e *a2* e a porta de saída *a3*;
- **Módulo B**: que possui as portas de entrada *b1* e *b2* e a porta de saída *b3*;
- **Módulo C**: que possui as portas de entrada *c1* e *c2* e a porta de saída *c3*.

No exemplo da figura 13, o módulo *C* possui uma instância dos módulos *A* e *B*. A porta de entrada *c1* do módulo *C* está conectada com a porta de entrada *a1* do módulo *A*, e a porta de entrada *c2* do módulo *C* está conectada com as portas de entrada *a2* e *b2* dos módulos *A* e *B*, respectivamente, ao passo que a porta de saída *a3* do módulo *A* está conectada com a porta

```
SC_MODULE (A) {
  sc_in <bool> a1;
  sc_in <bool> a2;
  sc_out <bool> a3;

  ...
};

SC_MODULE (B) {
  sc_in <bool> b1;
  sc_in <bool> b2;
  sc_out <bool> b3;

  ...
};

SC_MODULE (C) {
  sc_in <bool> c1;
  sc_in <bool> c2;
  sc_out <bool> c3;

  sc_signal <bool> s;

  A *moduloA;
  B *moduloB;

  SC_CTOR (C) {
    moduloA = new A ("moduloA");
    moduloA->a1 (c1);  moduloA->a2 (c2);  moduloA->a3 (s);
    moduloB = new B ("moduloB");
    moduloB->b1 (s);  moduloB->b2 (c2);  moduloB->b3 (c3);
  }
};
```

Figura 15: Codificação do Exemplo da Figura 14

de entrada *b1* do módulo *B* através do sinal *s*. Por fim, a porta de saída *b3* do módulo *B* está conectada com a porta de saída *c3* do módulo *C*.

Um exemplo de codificação do exemplo anterior é mostrado na figura 15.

3.2 Exemplo de Uma Aplicação SystemC

Iremos agora ilustrar o uso de SystemC em uma aplicação do tipo produtor/consumidor. Na nossa aplicação o produtor irá produzir itens que serão imediatamente consumidos pelo consumidor, sem o uso de *buffers*.

Na figura 16 podemos ver o módulo *Produtor*. Este módulo possui uma porta de saída, denominada *saida*, a qual conterá o dado produzido, e uma porta de entrada, chamada *clock*,

```
#ifndef PRODUTOR_H
#define PRODUTOR_H

#include <systemc.h>

SC_MODULE (Produtor) {
   sc_out <int> saida;
   sc_in <bool> clock;
   int dado;

   void produz () {
      saida = ++dado;
   }

   SC_CTOR (Produtor) {
      SC_METHOD (produz);
      sensitive << clock;
      dado = 0;
   }
};

#endif //PRODUTOR_H
```

Figura 16: Módulo Produtor

que será utilizada para se conectar com o relógio do sistema. No construtor, nós colocamos o método *produz* como sensível à mudança do valor da variável *clock*.

Na figura 17 encontra-se descrito o módulo *Consumidor*. Este módulo possui duas portas de entrada. A porta denominada *clock* será utilizada para conectar o módulo ao relógio do sistema, enquanto a porta *entrada* deverá ser conectada com a porta *saida* do módulo *Produtor* de modo a receber o novo dado produzido. Este módulo possui um método chamado *consome* que é sensível a mudanças na porta de entrada denominada *entrada*, ou seja, sempre que o valor desta porta for alterado, o que quer dizer que um novo dado foi recebido, o método *consome* será executado.

Na figura 18 temos o arquivo *main* do exemplo do produtor/consumidor. A biblioteca SystemC reconhece a função denominada *sc_main* como sendo a função principal que deve ser executada. Nesta função nós declaramos um sinal, chamado de *barramento*, que será utilizado para ligar a porta de saída, denominada *saida*, do módulo *Produtor* com a porta de entrada, denominada *entrada*, do módulo *Consumidor*.

Em seguida declaramos instâncias dos módulos *Produtor* e *Consumidor* e conectamos as suas respectivas portas de entrada/saída. Como dito anteriormente, quando cada instância é criada devemos fornecer um nome único como argumento do seu construtor. Depois chamamos a função *sc_start* do SystemC para inicializar o escalonador do SystemC e entramos em um

```
#ifndef CONSUMIDOR_H
#define CONSUMIDOR_H

#include <systemc.h>
#include <cstdio>

SC_MODULE (Consumidor) {
   sc_in <int> entrada;
   sc_in <bool> clock;

   void consome () {
      printf ("Dado = %d\n", entrada.read());
   }

   SC_CTOR (Consumidor) {
      SC_METHOD (consome);
      sensitive << entrada;
   }
};

#endif //CONSUMIDOR_H
```

Figura 17: Módulo Consumidor

laço para simular o nosso sistema.

3.3 Interfaces e Canais

Vimos que é possível fazer a comunicação entre módulos através da utilização de sinais para realizar a conexão entre as portas dos módulos, porém esta é uma abordagem de baixo nível e talvez não seja a mais adequada quando estamos no nível de sistema e desejamos trabalhar com um modelo mais abstrato e sofisticado de comunição. Uma abordagem mais apropriada para a comunicação no nível de sistema pode ser obtida utilizando-se canais.

Podemos dizer que um projeto SystemC descrito no nível de sistema consiste de um conjunto de módulos e canais, onde nos módulos residiria a implementação das funcionalidades do sistema, enquanto que os canais estariam encarregados dos aspectos relacionados com a comunicação.

Canais podem ser bastante genéricos e conter dentro deles a implementação de algoritmos complexos necessários para que ocorra a comunicação, como um protocolo de barramento com árbitro, por exemplo. Além disto, canais podem ter uma estrutura hierárquica, assim como os módulos.

Interfaces provêem um meio de acessar canais e consistem de classes abstratas que declaram

```
#include "produtor.h"
#include "consumidor.h"

int
sc_main (int argc, char **argv) {
   sc_signal <int> barramento;
   sc_signal <bool> clock;
   sc_time t (20, SC_NS);

   Produtor produtor ("produtor");
   produtor.saida (barramento);
   produtor.clock (clock);

   Consumidor consumidor ("consumidor");
   consumidor.entrada (barramento);
   consumidor.clock (clock);

   sc_start ();

   while (1) {
      clock.write (1);
      sc_cycle (t);

      clock.write (0);
      sc_cycle (t);
   }

   return 0;
}
```

Figura 18: Arquivo *main* do Exemplo Produtor/Consumidor

um conjunto de funções virtuais puras (os métodos de interface). Dizemos então que um canal implementa uma interface quando ele define todos os métodos declarados pela interface. O objetivo das interfaces é explorar a abordagem OO de forma que canais possam ser redefinidos independentemente dos módulos que fazem uso deles.

Os métodos definidos em um canal são tipicamente chamados via uma interface. Um canal pode implementar mais de uma interface, e uma única interface pode ser implementada por mais de um canal. Na figura 19 vemos o exemplo de uma comunicação entre módulos via interfaces e canais, os métodos implementados pelo canal são visíveis para a porta, que é conectada ao canal através de uma interface, mas não os detalhes de implementação dos métodos. Isto faz com que a implementação do módulo seja independente do modelo de comunicação.

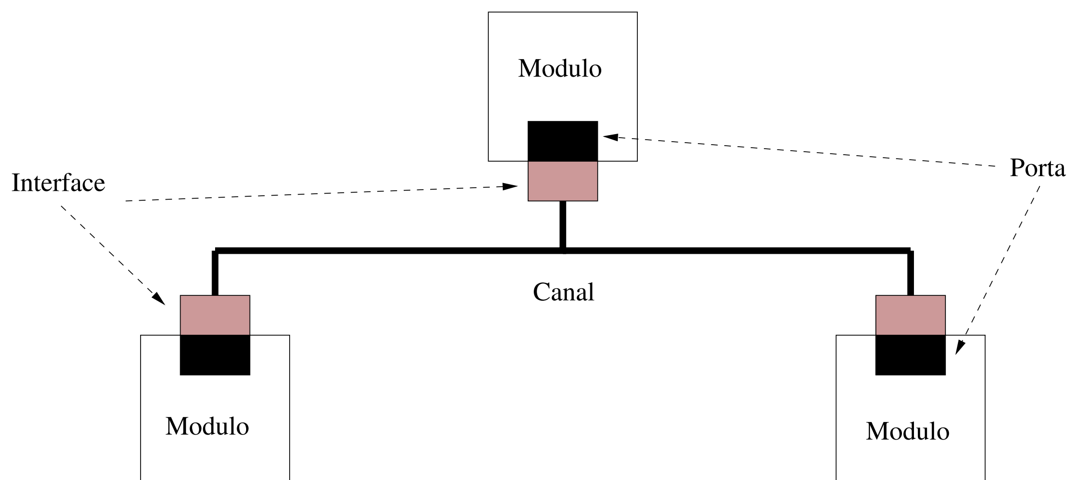

Figura 19: Exemplo de Módulos Conectados ao Canal através de uma Interface

3.4 Conclusão

Neste capítulo nós apresentamos SystemC, uma HDL que vem ganhando cada vez mais adeptos, pois permite descrever o sistema sendo projetado utilizando diferentes níveis de abstração. Mostramos os elementos básicos de SystemC, os módulos, e vimos duas maneiras de realizar a conexão entre módulos: uma utilizando uma abordagem de mais baixo nível, portas e sinais; e outra utilizando uma abordagem de mais alto nível, interfaces e canais.

Existem também várias características de SystemC que não foram abordadas aqui, como os diversos tipos de dados que o SystemC oferece aos seus usuários, tornando possível a modelagem de vários conceitos específicos hardware, e os processos *THREAD*, os quais possuem um fluxo de execução próprio, que pode ser suspenso e depois reativado.

No próximo capítulo, iremos abordar através de exemplos como a POA pode ser utilizada no desenvolvimento de aplicações SystemC, proporcionando assim uma melhor separação de interesses.

4 Exemplos da Utilização de Aspectos no Projeto de Sistemas Utilizando SystemC

Neste capítulo iremos abordar alguns exemplos da utilização da metodologia de desenvolvimento de software orientada a aspectos no projeto de sistemas que utilizem a biblioteca SystemC. Decidimos por uma abordagem prática pois nos parece ser esta a melhor forma de tentar convencer a comunidade de circuitos integrados da viabilidade da nossa estratégia, sendo assim, faremos uma prova do nosso conceito através de exemplos.

Para realizar este estudo foram analisadas várias aplicações SystemC. Algumas destas aplicações já apresentavam interesses transversais, de modo que uma abordagem orientada a aspectos iria naturalmente proporcionar uma melhor separação dos interesses transversais, tornando mais fácil de entender e de manter o código dos módulos onde tais interesses encontravam-se entrelaçados. Exemplos desta situação encontram-se descritos nas seções 4.2.1, que aborda a questão da coleta de métricas, e 4.2.4, que mostra como a POA poderia ser utilizada para gerar informação relacionada com a depuração do sistema.

Outras vezes, não existia um interesse transversal evidente no sistema, porém o uso de POA parecia adequado, pois permitiria encapsular uma caracterísica do sistema em uma unidade específica, bem como tornaria fácil a exploração do espaço de projeto. Um exemplo desta situação pode ser conferido na seção 4.1.2, que trata da política de troca de blocos da cache.

Também ocorreram situações onde não existia nenhuma aplicação SystemC disponível, sendo necessário a construção de uma aplicação específica para em seguida poder ser avaliado o uso da metodologia POA juntamente com a aplicação desenvolvida. O exemplo da seção 4.2.2, abordando uma estratégia de verificação de sistemas hardware/software, foi elaborado desta maneira.

Começaremos abordando o uso de aspectos na separação de interesses transversais em aplicações relacionadas com a modelagem de hardware, tais como: implementação da política de

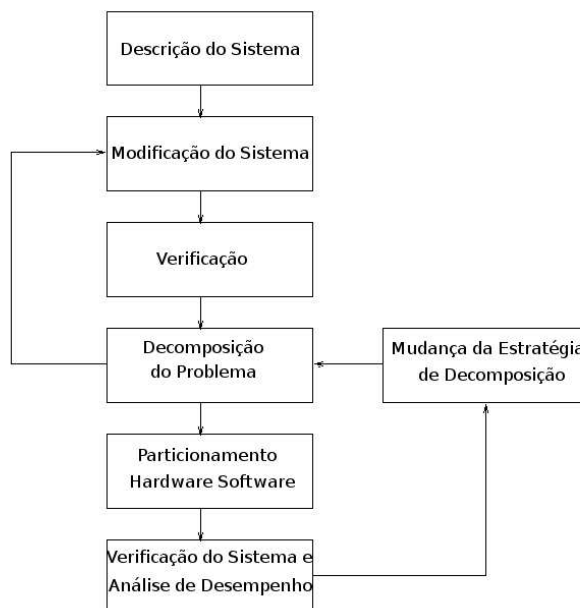

Figura 20: Visão Geral do Fluxo de Projeto no Nível de Sistema

troca de blocos da cache, descrição de um protocolo de comunicação e separação entre controle e dados.

Em seguida, abordaremos o uso de aspectos em atividades relacionadas com o projeto/processo de sistemas hardware/software. Serão abordadas aplicações tais como: coleta de métricas referentes ao resultado da simulação de uma aplicação, verificação de um sistema sendo projetado e geração de informação para ser utilizada na depuração do sistema.

A figura 20 nos dá uma visão geral do fluxo de projeto no nível de sistema usando uma linguagem como SystemC. Como iremos mostrar ao longo dos exemplos deste capítulo, podemos fazer uso de técnicas de POA em várias etapas do projeto. Podemos usar aspectos nas etapas iniciais do projeto, quando estamos iniciando a descrição do sistema, podemos decidir quais funcionalidades representam interesses transversais e poderiam ser melhor encapsuladas como um aspecto, como apresentado nas seções 4.1.1 e 4.1.3. Em seguida, podemos usar aspectos tanto na especificação como na verificação do sistema, como mostrado nas seções 4.2.2 e 4.2.3, e também podemos usar aspectos em etapas posteriores onde estamos interessados em como o sistema se comporta quando simulado, como podemos ver na seção 4.2.1.

4.1 Modelagem de Hardware

4.1.1 Modelando a Comunicação como um Aspecto

Introdução

O *Open Core Protocol* (OCP) (OCP-IP ASSOCIATION, 2003) define uma interface de alto desempenho, independente de barramento, entre dois componentes IP, o que reduz o tempo e os riscos do projeto de SoCs.

Um componente IP pode ser desde um simples periférico até um microprocessador de alta performance e através do uso do OCP torna-se mais fácil a reutilização de componentes IP, uma vez que o OCP torna os componentes IP independentes da arquitetura e do projeto do sistema no qual eles são utilizados.

Isto é feito através da especificação de um *wrapper* para o barramento, provendo assim uma interface de protocolo de transação independente de barramento para os componentes IP. Informações adicionais sobre o protolo OCP podem ser encontradas no apêndice A.

Motivação

Iremos agora demonstrar como aspectos podem ser usados para modelar a comunicação de um sistema utilizando como protocolo de comunicação o OCP.

Suponha que temos dois módulos SystemC, cada um com seus próprios sinais e portas de entrada/saída, entre os quais desejamos estabelecer uma comunicação. Existem várias abordagens possíveis para este problema, como conectar diretamente suas portas e sinais ou o uso de um protocolo de comunicação. Ao escolher fazer uso de um protolo de comunicação, podemos escolher uma abordagem de mais alto nível de abstração, usando interfaces e canais, ou elaborar uma solução mais detalhada, utilizando portas e sinais. Iremos optar então pelo uso do protocolo de comunicação OCP, mantendo um certo nível de detalhe e optando pela utilização de portas e sinais.

Um problema quando da utilização de um protocolo de comunicação usando um baixo nível de abstração ocorre quando adicionamos o código relativo à comunicação entre os módulos ao código original dos módulos, ou seja, misturamos o código do aspecto de comunicação com o código dos componentes. Desta forma, se quisermos usar estes módulos novamente com um protocolo de comunicação diferente será necessário reescrever nossos módulos e analisar quais partes deles dizem respeito às suas funcionalidades básicas, e portanto podem ser reutilizadas,

e quais partes não são, e necessitam ser mudadas para que se utilize um novo protocolo. Como pode ser visto, a reusabilidade destes módulos em um contexto diferente torna-se complicada se nós apenas misturamos o código do componente e o código da implementação do protocolo de comunicação. Se nós pretendemos tornar o nosso modelo um componente IP, a reusabilidade deste componente fica prejudicada pois há uma restrição quanto ao protocolo de comunicação que deve ser utilizado.

Um solução para este problema é descrever o protocolo de comunicação como um aspecto, de forma que é possível preservar os módulos originais. Feito isto, nosso módulos tornam-se mais reutilizáveis e fáceis de entender e de manter, visto que eles não possuem códigos com propósitos diferentes misturados ao longo deles. Dessa forma, iremos construir uma aplicação que contém um bom nível de detalhamento do sistema, com um baixo nível de abstração, e que ao mesmo tempo encontra-se bem modularizada, o que dificilmente seria alcançado se não utilizássemos a POA.

Implementação

O próximo passo é implementar os sinais básicos do OCP com a ajuda da POA. No exemplo em questão existem dois módulos: *Application*, que atua como um mestre OCP, e *Memory*, que atua como um escravo OCP. Em virtude do OCP usar *wrappers* como interfaces para os módulos, iremos então definir um *wrapper* bem simples para nossos modelos, contendo essencialmente apenas os sinais básicos do OCP e um método sensível ao relógio do sistema. Visto este ser um *wrapper* genérico, ele pode ser reutilizado toda vez que seja necessário usar o OCP para fazer a comunicação entre módulos.

Agora é necessário conectar os módulos *Application* e *Memory* com suas interfaces OCP, usando para isto aspectos. Como visto na seção 2.7.5, com o uso de aspectos é possível estender o código do programa e suas estruturas de dados, então nós iremos inserir alguns sinais nas nossas interfaces OCP para conectá-las aos seus respectivos módulos. Na figura 21, podemos ver parte do código do nosso aspecto, o qual irá alterar apenas nossos módulos *wrapper*, não realizando nenhuma modificação nos módulos *Application* e *Memory*, uma visão completa do sistema é apresentada na figura 22. Na linha 1, é declarado um conjunto de junção chamado *master()*. Nas linhas 3 e 4, são declarados dois sinais que serão adicionados à lista de variáveis do módulo chamado *Master*, o qual é nossa interface mestre do OCP. A razão de fazer isto é que o módulo *Application* possui portas chamadas *DataIn* e *DataOut*, então é necessário criar sinais para conectar estas portas ao módulo *Master*. Na linha 6, é inserida uma instância da classe *Application*. Agora precisamos inicializar as novas variáveis adicionadas ao módulo

```
1    pointcut master () = "Master";
2
3    advice master() : sc_signal <sc_int<WORD> > sig_DataIn;
4    advice master() : sc_signal <sc_int<WORD> > sig_DataOut;
5
6    advice master() : Application *application;
7
8    advice construction (master()) : after () {
9       Master *master = (Master *) (tjp->target());
10      master->application = new Application ("application");
11      master->application->DataIn (mw->sig_DataIn);
12      master->application->DataOut (mw->sig_DataOut);
13   }
```

Figura 21: Conectando os módulos *Master* e *Application*

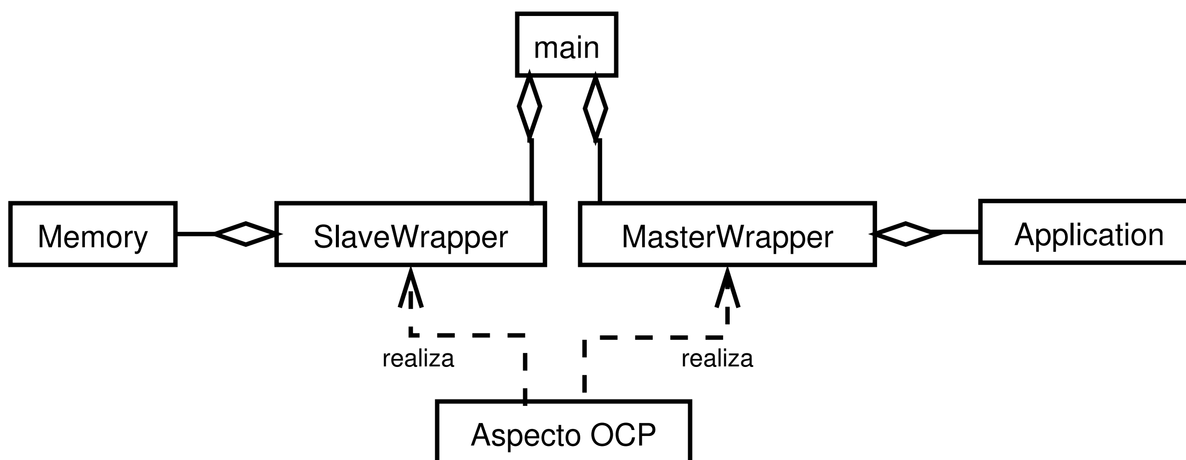

Figura 22: Visão Geral do Nosso Sistema Utilizando OCP e Aspectos

Master. Para isto, usamos a diretiva *construction* na linha 8, a qual serve para inserir código de comportamento transversal no construtor de uma classe. Na linha 9, é declarado um ponteiro para a instância do módulo *Master* que ativou este ponto de junção. Em seguida, é instanciada a variável *application* e suas portas *DataIn* e *DataOut* são conectadas aos sinais *sig_DataIn* e *sig_DataOut* do módulo *Master*. Uma abordagem similar é usada para conectar os módulos *Memory* e *Slave*.

Neste exemplo, o módulo *Application*, que atua como um mestre OCP, escreve dados para o módulo *Memory* ou lê dados do mesmo. Toda vez que um mestre OCP deve realizar uma operação, ele envia um comando para um escravo e aguarda pela resposta. Na figura 23, está descrito uma parte do aspecto de comunicação relacionado com o mestre OCP quando ele está esperando pela resposta de um escravo. Depois da composição de aspectos, este código será inserido no módulo *Master*. Na linha 2, o módulo *Master* verifica se há algum dado para ler da memória; se há, ele então envia um comando *RD*, o endereço de memória correspondente

```
1    if (waitingSCmdAccept) {
2       if (master->application->getState () == READING) {
3          master->MCmd.write (RD);
4          master->MAddr.write (master->application->addr.read ());
5          master->sig_availableIn.write (0);
6       } else if (master->application->getState () == WRITING) {
7          master->MCmd.write (WR);
8          master->MData.write (master->application->DataOut.read ());
9          master->MAddr.write (master->application->addr.read ());
10         master->sig_writingDone.write (0);
11      } else {
12         master->MCmd.write (IDLE);
13         master->MDataValid.write (0);
14      }
15      if (master->SCmdAccept.read()) {
16         if (master->MCmd.read () == WR) {
17            master->MDataValid.write (1);
18         }
19         master->MCmd.write (IDLE);
20         master->MRespAccept.write (0);
21         waitingSCmdAccept = false;
22      }
23   }
```

Figura 23: Conectando os módulos *Master* e *Application*

e informa que não há dado disponível na porta *DataIn* do mestre. Na linha 6, testa-se se o módulo *Application* possui algum dado para escrever na memória; caso tenha, o comando *WR* é enviado, junto com o dado correspondente e o endereço de memória onde ele deve ser escrito. Também informamos que o processo de escrita não foi concluído ainda. Na linha 15, é feito um teste para verificar se a resposta do escravo OCP chegou. Se o comando atual é *WR*, então o mestre OCP já colocou um dado válido no campo *MData*.

Conclusão

Através do uso desta abordagem, foi possível realizar uma implementação mais detalhada do nosso modelo e encapsular o código relacionado com o aspecto de comunicação em uma única unidade, ao invés de deixá-lo misturado com o código relativo às funcionalidades básicas dos módulos, o que provavelmente ocorreria se não utilizássemos técnicas de POA. O código completo deste exemplo pode ser visto no apêndice B.

4.1.2 Política de Troca de Linhas da Cache

Introdução

Este experimento tenta mostrar como a POA pode ser empregada para explorar o espaço de projeto. No exemplo aqui apresentado, a POA irá auxiliar a determinar qual é a política de troca de linhas da cache mais adequada para a nossa aplicação.

Motivação

A política de troca de linhas é um dos fatores chave para determinar a eficiência de uma memória cache. Os processadores atuais utilizam várias destas políticas, tais como *Random*, Bloco Menos Recentemente Usado (*Least Recently Used* - LRU), e *First-In-First-Out* (FIFO), o que indica que não há um consenso geral sobre qual é a melhor política de troca de linhas a ser utilizada (AL-ZOUBI; MILENKOVIC; MILENKOVIC, 2004). Em virtude disto, é importante explorar o espaço de projeto para encontrar a política de troca de linhas da cache mais apropriada.

Isto pode ser conseguido através do uso da POA para determinar a melhor política de cache de uma aplicação. Usando técnicas de POA, iremos representar cada política de cache como um aspecto. Desta forma, poderemos escolher que política de cache iremos utilizar para cada módulo apenas selecionando o aspecto correspondente, sem precisar realizar nenhuma mudança no código fonte dos módulos.

Implementação

A abordagem proposta na seção anterior foi testada com um processador desenvolvido localmente, denominado de GPOP (SOARES, 2006), e com as políticas de cache Random, LRU e FIFO. Neste processador, os módulos *DCache* e *ICache* são os módulos responsáveis pela implementação da memória cache. Para este exemplo escolhemos utilizar uma política LRU de troca de linhas da cache e vamos representar isto através de um aspecto. A figura 24 nos mostra então o esqueleto do aspecto *LRUBlockReplacement*, que vai atuar sobre os módulos *DCache* e *ICache* para implementar a política LRU. A linha 4 declara então um conjunto de junção que descreve todos os métodos, de ambos os módulos, que realizam uma operação de escrita na cache. Na linha 7 é declarado outro conjunto de junção para os métodos que realizam operações de leitura na cache.

Nas linhas 16 a 19 é declarado um código de comportamento transversal que irá manter

```
1 aspect LRULineReplacement {
2
3   public:
4   pointcut memWrite() = "% DCache::writeData(...)"
5                    || "% ICache::writeData(...)";
6
7   pointcut memRead() = "% DCache::readData(...)"
8                   || "% ICache::readData(...)";
9
10  pointcut replacement() = "% DCache::replacement()"
11                       || "% ICache::replacement()";
12
13  pointcut reset() = "% DCache::reset(...)"
14                || "% ICache::reset (...)";
15
16  advice execution (memWrite() || memRead())
17                   : after () {
18    //atualiza a lista de linhas acessadas
19  }
20
21  advice execution (replacement()) : around() {
22    //retorna a linha que deve ser descartada
23  }
24
25  advice execution (reset()) : before() {
26    //inicializa a lista de linhas accessadas
27  }
28
29  private:  list <int> lru;
30};
```

Figura 24: Aspecto LRU

atualizada a lista das linhas mais recentemente utilizadas. Nas linhas 21 a 23 temos o código de comportamento transversal que indica qual é a linha que deve ser descartada. E nas linhas 25 a 27 é realizada a rotina de inicialização da lista de linhas acessadas. Por fim, na linha 29 é declarada uma varíavel do tipo *list* para manter a nossa lista de linhas acessadas.

Conclusão

O exemplo aqui apresentado mostrou que é possível representar o código relacionado com a política de troca de linhas da cache como um aspecto. Esta solução, que foi construída a partir de um implementação já existente, nos pareceu a mais adequada para alterar o sistema em questão. O fato de poder representar a política de troca de linhas da cache como um aspecto nos permite concluir que é possível separar um componente do sistema de hardware do restante do sistema. Sendo assim, diferentes alternativas de hardware podem ser testadas através da escolha de um componente diferente e da sua conexão com o sistema principal, avaliando-se então qual é o desempenho do sistema com este novo componente.

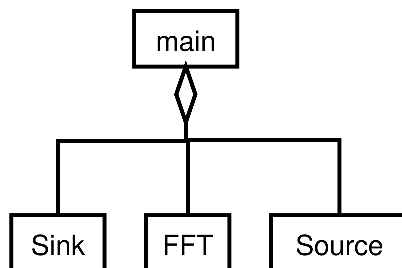

Figura 25: Organização dos Módulos que Implementam a FFT

Uma vez que a melhor política de cache tenha sido determinada, pode-se utilizar o combinador de aspectos AspectC++ para produzir uma descrição do sistema contendo somente código SystemC/C++, de forma que outras ferramentas que lidam com SystemC podem utilizar o modelo gerado neste estágio.

4.1.3 Separação do Controle e do Fluxo de Dados

Introdução

O exemplo que será apresentado agora é baseado na implementação da Transformada Rápida de Fourier (*Fast Fourier Transform* - FFT), disponibilizada no arquivo de exemplos do SystemC.

O objetivo da implementação original é fornecer uma implementação da FFT para ser usada com números inteiros e uma outra para ser utilizada com números de ponto flutuante. Para alcançar este objetivo a implementação foi dividida em dois diretórios e seu respectivo conjunto de arquivos. A figura 25 nos mostra a estrutura dos módulos, que é a mesma para as duas implementações (o componente *main* não é verdadeiramente um módulo SystemC, mas representa a função principal do sistema, i.e., *sc_main*).

Motivação

Embora as duas implementações da FFT compartilhem a mesma estrutura, a escolha da transformada rápida de Fourier afeta bastante os módulos do sistema, de forma que *todos* os módulos do sistema precisam ser modificados para que se passe a utilizar uma FFT ao invés da outra. Podemos observar que a mudança de implementação da FFT altera o caminho de dados dos módulos, porém, os sinais/portas relacionados com o controle dos mesmo, permanecem inalterados, não sofrendo nenhuma modificação. Sendo assim, podemos tratar este exemplo

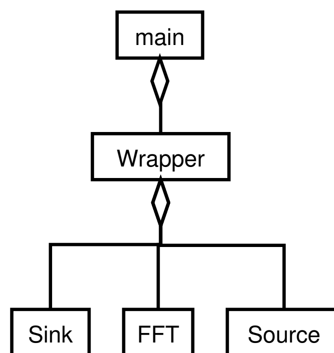

Figura 26: Nova Organização dos Módulos que Implementam a FFT

como um problema de separação entre controle e dados (THEKKATH; LEVY; LAZOWSKA, 1994; YANG; LI, 2004).

A idéia então é utilizar POA para encapsular em um aspecto este código relativo à implementação da transformada rápida de Fourier que se espalha por vários módulos do sistema e que eventualmente será modificado se desejarmos mudar a nossa implementação. Tal código consiste de diversas portas de entrada e saída, além de funções que realizam o algoritmo da FFT.

Iremos também propor uma nova estrutura de módulos, inserindo o módulo *Wrapper*, que conterá o código responsável por conectar os módulos *Sink*, *Source* e *FFT*, de forma que o módulo *main* não sofrerá mais nenhuma influência desses módulos e nem será afetado pelo nosso aspecto. A nova estrutura dos módulos pode ser vista na figura 26.

Implementação

Com a nova estrutura elaborada, podemos ver como fica o esqueleto do nosso aspecto, o qual está representado na figura 27. Serão necessários dois aspectos, cada um será responsável por uma implementação diferente da FFT, de forma que poderemos escolher qual deles iremos utilizar apenas mudando uma opção de compilação. O aspecto apresentado na figura 27 é responsável pela implementação da FFT com números inteiros.

Na linha 1 nós declaramos o aspecto *FXPT* e em seguida, na linha 3, declaramos o conjunto de junção *ptwrapper*. Em seguida (linhas $5-8$), introduzimos variáveis relacionadas com implementação da FFT no módulo *Wrapper*, de forma a adaptá-lo para o algoritmo em questão. Uma abordagem similar é realizada para os módulos *Sink* (linhas $10-13$), *Source* (linhas $15-18$) e *FFT* (linhas $20-25$).

```
1  aspect FXPT {
2
3    pointcut ptwrapper () = "Wrapper";
4
5    advice ptwrapper () : public: sc_signal <sc_int <WORD> > in_real;
6    advice ptwrapper () : public: sc_signal <sc_int <WORD> > in_imag;
7    advice ptwrapper () : public: sc_signal <sc_int <WORD> > out_real;
8    advice ptwrapper () : public: sc_signal <sc_int <WORD> > out_imag;
9
10   pointcut ptsink () = "Sink";
11
12   advice ptsink () : public: sc_in <sc_int<WORD> > in_real;
13   advice ptsink () : public: sc_in <sc_int<WORD> > in_imag;
14
15   pointcut ptsource () = "Source";
16
17   advice ptsource () : public: sc_out<sc_int<WORD> > out_real;
18   advice ptsource () : public: sc_out<sc_int<WORD> > out_imag;
19
20   pointcut ptfft () = "FFT";
21
22   advice ptfft () : public: sc_in<sc_int<WORD> > in_real;
23   advice ptfft () : public: sc_in<sc_int<WORD> > in_imag;
24   advice ptfft () : public: sc_out<sc_int<WORD> > out_real;
25   advice ptfft () : public: sc_out<sc_int<WORD> > out_imag;
26
27   advice ptfft () : public: void func_butterfly ( ... )  { ... };
28
29   advice execution ("void FFT::entry()") && that (ff): around (FFT& ff)
30   { ... }
31
32 };
```

Figura 27: Aspecto FXPT

O módulo *FFT* requer uma atenção especial, pois além de inserirmos um novo conjunto de portas de entrada/saída, devemos também modificar a sua função que realiza a transformada rápida de Fourier, bem como introduzir alguma outra função auxiliar. Na linha 27, adicionamos ao módulo *FFT* a função *func_butterfly*, que aqui teve a sua lista de parâmetros omitida e o seu corpo omitidos, para realizar a computação borboleta (*butterfly*).

Finalmente, nas linhas $29-30$, introduzimos o método *entry* que será responsável por realiazar a transformada rápida de Fourier, este método também teve o seu corpo omitido. Devemos notar, na linha 29, a chamada da função *that* para se obter um ponteiro para o objeto do tipo *FFT* que alcançou este ponto de junção, ao qual demos o nome de *ff*.

Conclusão

Com a abordagem aqui apresentada, o código relacionado com a transformada rápida de Fourier para números inteiros, que antes encontrava-se espalhado por todos os módulos do nosso sistema, agora está bem localizado no aspecto *FXPT*, não afetando mais o restante do sistema. Esta abordagem também propiciou uma boa separação entre o código relacionado com fluxo de dados e o código relacionado com o fluxo de controle.

Uma abordagem utilizando somente técnicas de OO talvez não conseguisse uma boa separação de interesses, principalmente porque o código relacionado com a FFT ainda ficaria espalhado por diversos módulos do sistema, o que tornaria difícil a modificação do mesmo.

4.2 Projeto/Processo

4.2.1 Utilizando Aspectos para Realizar a Coleta de Métricas

Introdução

A simulação de uma aplicação SystemC é um estágio muito importante no desenvolvimento de um projeto, pois nesta etapa os projetistas do sistema podem observar e aperfeiçoar o comportamento da aplicação que está sendo desenvolvida. Quando se está desenvolvendo um sistema com SystemC, freqüentemente é inserido no código funcional do sistema algum código extra para que se possa coletar informações a respeito da simulação do sistema.

Motivação

Como podemos notar, o código extra que é inserido, relacionado com a coleta de métricas, não faz parte do código do componente, e poderia ser encapsulado em um aspecto, trazendo vários benefícios tais como:

- melhoria da legibilidade do código do componente;
- o código responsável por coletar métricas da simulação que encontrava-se espalhado pelo sistema agora está bem localizado em um único lugar;
- visto que agora temos um aspecto responsável por coletar métricas, podemos utilizá-lo também no projeto de outros sistemas.

Implementação

O exemplo que será apresentado, foi extraído de uma aplicação que implementa uma Rede em Chip (*Network on Chip* - NoC) (JANTSCH; TENHUNEN, 2003). Nesta aplicação, cada nó da rede possui portas de entrada e saída, sendo que cada porta possui um número de canais virtuais associado. Estes canais virtuais são implementados atráves de módulos que armazenam os pacotes em uma estrutura de dados do tipo FIFO, onde os pacotes são armazenados na ordem em que chegam. O funcionamento destes módulos foi modelado através de um máquina de estados, de forma que o comportamento do módulo será determinado a partir do seu estado atual. Deseja-se saber então, qual foi o comportamento dos nossos canais virtuais ao longo da simulação. Para isto, vamos usar um aspecto para coletar informação sobre o estado dos nossos módulos ao longo da simulação, averiguando quantas vezes um determinado módulo do sistema passou por cada estado possível.

Pensando nisto, foram introduzidas variáveis para contar quantos ciclos um módulo passou em cada estado possível; estas variáveis encontram-se descritas a seguir:

- **idle_cycles**: conta o número de ciclos que o módulo passou no estado *IDLE*;
- **requesting_cycles**: conta o número de ciclos que o módulo passou no estado *REQUESTING*;
- **pushing_cycles**: usada para contar o número de ciclos que o módulo ficou no estado *PUSH*;
- **popping_cycles**: conta o número de ciclos que o módulo passou no estado *POP*;
- **cyles**: conta o número total de ciclos.

Na figura 28, podemos ver uma versão simplificada do nosso aspecto. Na linha 3, são declaradas as variáveis para contar o número de ciclos gastos em cada estado. Na linha 4, declaramos um código de comportamento transversal, a palavra chave *around* indica que este código de comportamento transversal deve substituir o código original associado com o ponto de junção em questão.

Este código de comportamento transversal chama o método *init()* do aspecto para inicializar as variáveis e em seguida chama o método *proceed()* do ponteiro *tjp*. Como dito na seção 2.7.4, o ponteiro *tjp* contém informações relacionadas com o ponto de junção que foi alcançado, e ao fazermos uma chamada ao seu método *proceed()* estamos fazendo uma chamada ao código original do ponto de junção, que neste caso é a função *main*.

```
1  aspect Simulation {
2    enum State = {RESET, IDLE, PUSHING, REQUESTING, POPPING};
3    int cycles, idle_cycles, requesting_cycles, pushing_cycles, popping_cycles;
4    advice execution("int main(...)") : around() {
5      cout << "Simulation Started" << endl;
6      init();
7      tjp->proceed();
8      show_results();
9    }
10   advice execution("void MyModule::mainFunction(...)") : after() {
11     MyModule *mm = (MyModule *) (tjp->target());
12     ++cycles;
13     switch(mm->state) {
14       case (RESET): init();
15                     break;
16       case (IDLE):  ++idle_cycles;
17                     break;
18       case (PUSHING): ++pushing_cycles;
19                       break;
20       case (REQUESTING): ++requesting_cycles;
21                          break;
22       case (POPPING): ++popping_cycles;
23                       break;
24     }
25   }
26   public:
27     void init() { ... }
28     void show_results() { ... }
29 };
```

Figura 28: Aspecto refererente à Simulação do Sistema

Em seguida, fazemos uma chamada ao método *show_results()*, o qual deve imprimir algumas informações sobre o resultado final da simulação. A implementação deste método e do método *init()* foram omitidas aqui, visto que não eram necessárias para o entendimento deste exemplo.

Na linha 10, é declarado um outro código de comportamento transversal que deve ser executado *depois* de cada execução da função *mainFunction* do módulo *MyModule*. Neste código de comportamento transversal, procuramos descobrir qual é o estado atual do módulo e atualizamos a variável contadora correspondente.

Conclusão

Como foi visto na seção anterior, temos agora o código relativo à simulação agrupado em um aspecto chamado de *Simulation* e o módulo relacionado com as funcionalidades básicas do sistema torna-se mais claro, pois os códigos com propósitos diferentes não estão mais misturados. Para realizar uma simulação sem o aspecto que definimos, basta compilar o(s) módulo(s)

que irá(ão) interagir com o aspecto novamente sem habilitar a combinação de aspectos, ou então usando um compilador convencional.

4.2.2 Utilizando Projeto por Contrato com POA

Introdução

Projeto por Contrato (*Design by Contract* - DBC) (MEYER, 2000) é uma técnica baseada em asserções cujo objetivo é assegurar a qualidade, a confiabilidade e a reusabilidade do software sendo produzido e que já foi aplicada com sucesso na área de sistemas embarcados (BRUNEL et al., 2004). No Projeto por Contrato, cada componente (método ou classe) é associado a um contrato que especifica o seu comportamento. Geralmente, o contrato é expresso usando uma lógica de primeira ordem.

Além de ser uma forma de documentação do sistema, o contrato também pode ser usado para verificar que o componente é corretamente empregado, e que ele comporta-se como esperado.

Para verificar se uma aplicação está de acordo com um contrato, iremos fazer uso de asserções. Uma asserção é uma expressão booleana usada para testar uma condição de entrada (ou pré-condição), uma condição de saída (ou pós-condição), ou um invariante, i.e., uma condição a qual se aplica a todo o conjunto de variáveis que define um módulo/componente. A ocorrência de uma falha ao tentar cumprir as exigências de um contrato é tipicamente o sinal de que há um erro no hardware/software.

Motivação

A POA facilita o uso da metodologia DBC em virtude da sua flexibilidade, transparência e capacidade de lidar com separação de interesses (DIOTALEVI, 2004). O resultado é que abordagem DBC, combinada com a metodologia POA, é bastante apropriada para realizar a verificação de um sistema.

Implementação

Vamos ilustrar o uso de DBC juntamente com POA tomando por base o exemplo da figura 29, a qual mostra uma operação de soma sobre dois valores. O número de bits usados para a representação binária das variáveis *a*, *b* e *res* é indicado por *WORD*. É possível também acessar o valor de um único bit de cada uma das variáveis, de forma que uma expressão do tipo

```
1 sc_int <WORD>
2 Company::add (sc_int<WORD> a, sc_int<WORD> b) {
3   sc_int <WORD> res = a + b;
4   if (a[WORD-1] == b[WORD-1] && res[WORD-1] != a[WORD-1]) {
5     //trata Overflow/Underflow
6   }
7   return res;
8 }
```

Figura 29: Exemplo de uma Operação de Soma

$a[WORD-1]$ indica qual é o bit mais significativo da variável a.

Na metodologia DBC, o módulo que implementa esta operação de soma é definido como o fornecedor, ao passo que um módulo que venha a usar esta operação é definido como um cliente. Algumas vezes, o cliente não usa as funcionalidades oferecidas pelo fornecedor de maneira apropriada, de forma que ele pode passar, por exemplo, um parâmetro inválido para uma função, quebrando assim o contrato estabelecido entre ambos. Da mesma forma, o fornecedor pode descumprir o contrato ao não realizar corretamente a função que se espera dele.

No exemplo da figura 29, um possível problema seria um valor de retorno errado da operação *add*, que não representasse de forma correta a adição dos valores *a* e *b*. Através do uso de DBC propõe-se detectar se tal caso ocorre e verificar que o cliente usa a interface do fornecedor de maneira apropriada, bem como que o fornecedor produz os resultados esperados.

A figura 30 mostra o exemplo de um contrato declarado como um aspecto. A linha 5 declara um código de comportamento transversal do tipo *around*, de modo que o código contido nesta declaração será executado no lugar do código original da função *sc_main*. Este código de comportamento transversal coloca o corpo da função principal dentro de um bloco *try/catch*, de forma que agora é possível tratar as exceções disparadas a partir de *sc_main*.

Outro código de comportamento transversal do tipo *around* é declarado na linha 13, de modo que ele será executado no lugar do método *add* da classe *Company*. Neste código de comportamento transversal, a primeira ação tomada é salvar o valor dos parâmetros do método *add*, o que é feito nas linhas 15 e 17, usando a função *arg* do ponteiro *tjp*, a qual retorna um argumento da função (por questão de simplicidade, assumimos que o tipo *int* possui um tamanho semelhante ao de *WORD*). O próximo passo é verificar as pré-condições e os invariantes. Um exemplo de pré-condição poderia ser que os parâmetros da função fossem números não negativos. É importante notar que a verificação dos invariantes é realizada duas vezes, antes e depois da execução da função, para garantir que a execução da função não afetou o valor de alguns campos externos.

```
1 aspect Contract {
2
3   pointcut target () = "% Company::add(...)";
4
5   advice execution("int sc_main(...)"): around(){
6     try {
7       tjp -> proceed ();
8     } catch (ContractException e) {
9       //trata exceção
10    }
11  }
12
13  advice execution (target()) : around () {
14
15    int olda = ((sc_int<WORD> *) tjp->arg(0))->to_int();
16
17    int oldb = ((sc_int<WORD> *) tjp->arg(1))->to_int();
18
19    //verifica invariantes se houver
20    //verificar pré-condições se houver
21
22    tjp->proceed();
23
24    int res = ((sc_int<WORD> *) tjp->result()) -> to_int();
25
26    if (res != olda + oldb) {
27      throw new Exception (POSTCONDITION, "Wrong result value");
28    }
29
30    //verifica invariantes se houver
31  }
32};
```

Figura 30: Exemplo de um Contrato

Iremos nos concentrar somente na verificação da corretude da pós-condição, i.e., se o resultado da função é correta e representa a soma de *a* e *b*. Na linha 22, é feita uma chamada explícita ao código original da função *add*, através do método *proceed*, e na linha 24 é obtido o valor de retorno desta chamada usando o método *result* do ponteiro *tjp*. Na linha 26 é realizado o teste para verificar se o valor de retorno está correto e corresponde à soma de *olda* e *oldb*, e caso o valor de retorno esteja errado uma exceção é disparada indicando isto.

Conclusão

A utilização de técnicas de POA tornou mais fácil a verificação do sistema usando DBC, visto que podemos estabelecer vários contratos diferentes para um mesmo componente, e selecioná-los através da escolha do aspecto correspondente, sem precisar, contudo, realizar mudanças no código do componente.

4.2.3 A Biblioteca de Verificação Padrão do SystemC e POA

Introdução

A Biblioteca de Verificação Padrão do SystemC (*SystemC Verification Standard* - SCV) (SYSTEMC, 2003), fornece um padrão para o desenvolvimento de bancos de teste e de verificação de IPs para o projeto de SoCs. SCV provê uma Interface de Programação de Aplicação (*Application Programming Interface* - API), para realizar tarefas como:

- verificação baseada em transação (*transaction-based verification*);
- geração aleatória de valores de acordo com restrições e com pesos;
- manipulação de exceções;
- utilização de um banco de testes *SystemC* para projetos escritos usando *VHDL* ou *Verilog*.

Além das características acima mencionadas, SCV também fornece um mecanismo de introspecção de dados (*data introspection*) que permite manipular tipos de dados arbitrários de uma maneira consistente, incluindo os tipos de dados básicos de *C/C++*, os tipos de dados fornecidos por *SystemC*, os tipos de dados compostos definidos pelo usuário e as enumerações definidas pelo usuário.

Para se realizar a verificação de um modelo, deve ser usado algum banco de teste que pode ser visto como um conjunto de restrições sobre os possíveis valores da simulação, escrito utilizando SCV, além do código de teste propriamente dito e de algum código SystemC auxiliar.

Na figura 31, temos o exemplo de um código utilizando SCV que expressa um conjunto de restrições sobre os valores que uma variável pode vir a assumir durante a simulação. Na linha 1, é declarado um ponteiro esperto, denominado *data*, que será usado para gerar valores aleatórios de inteiros durante a simulação. Na linha 2, é aplicada uma restrição sobre o conjunto de valores que este tipo pode assumir, somente valores entre 0 e 15. Em seguida, na linha 3, é feita uma nova restrição, onde são descartados os valores que se encontram entre 6 e 9, de modo que os possíveis valores que *data* pode gerar encontram-se agora na faixa $0-5$ e $10-15$. Por fim, na linha 4, um novo valor aleatório, respeitando as restrições impostas, é gerado utilizando-se o método *next*.

Como foi dito anteriormente, SCV pode ser utilizado sobre tipos arbitrários de dados, basta somente fazer uma espécie de cadastro dos tipos definidos pelo usuário que vão fazer uso de SCV e quais são os seus campos. No caso de um classe, seria necessário então indicar quais são as variáveis que são membros daquela classe.

```
1  scv_smart_ptr <int> data;
2  data->keep_only (0, 15);
3  data->keep_out (6, 9);
4  data->next ();
```

Figura 31: Exemplo de Código SCV

```
1  class write_constraint : virtual public scv_constraint_base {
2    public:
3    scv_smart_ptr<write_t> write;
4    SCV_CONSTRAINT_CTOR(write_constraint) {
5      SCV_CONSTRAINT( write->addr() <= ram_size );
6      SCV_CONSTRAINT( write->addr() != write->data() );
7    }
8  };
```

Figura 32: Exemplo de Declaração de uma Classe de Restrições

Também é possível definir um conjunto de restrições no estilo de uma classe, de modo que seja possível utilizá-lo depois sobre um objeto de um determinado tipo. Um exemplo disto pode ser visto na figura 32. Na linha 1, é declarada a nossa classe de restrições denominada de *write_constraint*, que possui como classe base *scv_constraint_base*. Em seguida, na linha 3, declaramos um ponteiro esperto, do tipo *write_t*, denominado *write*. Depois, na linha 4, temos o construtor da nossa classe e, nas linhas 5 e 6, as restrições que devem ser aplicadas sobre a nossa variável *write* quando for ser gerado um novo conjunto de valores para uma instância da mesma. Todas as restrições devem ser informadas no construtor da classe *write_constraint*, não podendo ser adicionadas restrições posteriormente.

Uma possível aplicação do conjunto de restrições descrito na classe *write_constraint* pode ser visto na figura 33. Na linha 1, é declarado um ponteiro esperto do tipo *write_t*, denominado *w*. Depois, na linha 3, é gerado um novo conjunto de valores para as variáveis de *w*. Como não foi definida nenhuma restrição e sabendo que o ponteiro esperto *w* é do tipo da classe *write_t*, a qual possui dois campos inteiros como membros (*addr* e *data*), as variáveis membro de *w* podem assumir então qualquer valor dentro da faixa de valores possíveis que um inteiro pode assumir. Na linha 4, um valor da classe *write_t* é gerado pela aplicação do método *get_instance* ao nosso ponteiro esperto, e é fornecido a um objeto *transactor* através do método *write*.

Resolvemos então declarar uma classe de restrições na linha 6. Em seguida, na linha 7, aplicamos esta classe de restrições ao nosso ponteiro esperto *w*, de forma que as restrições que se aplicavam ao campo *write* da nossa classe *write_constraint*, agora se aplicam também ao ponteiro esperto *w* quando ele tentar gerar novos valores para os seus campos, como é feito na linha 8. Devemos notar que a restrição ainda se aplicaria a *w*, mesmo que saíssemos do escopo

```
1   scv_smart_ptr<write_t> w;
2
3   w->next ();
4   transactor->write (w->get_instance () );
5
6   write_constraint wc ("wc");
7   w->use_constraint (wc.write);
8   w->next();
9   transactor->write( w->get_instance() );
10
11  ...
12
13  other_constraint oc ("oc");
14  w->use_constraint (oc.write);
15  w->next ();
```

Figura 33: Exemplo do Uso de uma Classe de Restrições

da variável *wc*, supondo que *w* possuísse um escopo maior. Por fim, na linha 13, declaramos um novo conjunto de restrições, do tipo *other_constraint*, e o aplicamos à variável *w*, agora, ao gerarmos novos valores para os campos de *w*, as restrições da classe *other_contraint* devem ser respeitadas ao invés daquelas contidas na classe *write_constraint*.

Motivação

Com base no que foi apresentado, poderíamos então criar um aspecto responsável pela verificação do sistema. Tal aspecto poderia conter, por exemplo:

- introduções/código de comportamento transversal que agissem sobre as classes de restrições, adaptando-as de maneira a criar um banco de teste específico para um determinado projeto;
- o código de teste, que poderia ficar espalhado por vários módulos do projeto, e que agora estaria bem encapsulado em uma única unidade;
- código capaz de definir, a partir do contexto de execução do ponto de junção, qual seria o conjunto de restrições a ser utilizado na geração de novos valores.

Implementação/Conclusão

O uso de POA com SCV, contudo, não é possível ainda, senão de maneira bastante limitada. Isto se deve ao fato de que a ferramenta AspectC++ não consegue manipular de forma adequada descrições de pontos de junção que envolvam macros e *templates*, e que se fazem necessários

para se possa usar POA juntamente com SCV. Deste modo, dentre as idéias aqui apresentadas, a única que pode ser implementada é a que diz respeito ao agrupamento do código relacionado com o teste em um aspecto, não podendo as demais idéias serem implementadas no estágio atual de desenvolvimento de AspectC++.

4.2.4 Usando POA com ArchC

Introdução

ArchC (RIGO et al., 2004) é uma linguagem de descrição de arquiteturas baseada em SystemC. Utilizando ArchC, é possível descrever tanto a arquitetura de um processador como de uma hierarquia de memória, sempre seguindo o estilo de sintaxe do SystemC.

O objetivo principal de ArchC é proporcionar informação suficiente, no nível de abstração adequado, de forma a permitir aos usuários a exploração e a verificação de uma nova arquitetura através da geração automática de ferramentas de software tais como simuladores e *assemblers*.

A descrição de uma arquitetura utilizando ArchC é composta de duas partes: o Conjunto de Instruções da Arquitetura (*Instruction Set Architecture* - AC_ISA), e os Recursos da Arquitetura, (*Architecture Resources* - AC_ARCH). A descrição *AC_ISA* deve fornecer as informações sobre o formato das instruções, o seu tamanho, seu nome, além de informações relacionadas com a decodificação e com o comportamento das instruções. Já a descrição *AC_ARCH*, deve informar o ArchC sobre dispositivos de armazenamento, estrutura de pipeline, etc. Com base nestas duas descrições, ArchC pode gerar simuladores interpretados (usando SystemC), simuladores compilados e assemblers (usando um conjunto de ferramentas denominado *Binutils*).

Motivação

Nós iremos nos concentrar agora sobre um exemplo que ilustra como a POA pode ser usada para tratar de interesses transversais presentes em um modelo ArchC que descreve o conjunto de instruções de uma arquitetura **MIPS** (HENNESSY et al., 1982).

Na figura 34, é possível ver a descrição da instrução *addiu* (adição imediata para números sem sinal) usando ArchC. Pode-se notar que além do código necessário para a realização da instrução, que está localizado na linha 4, existe também um código de interesse transversal relacionado com a depuração do modelo, linhas 3 e 5, de modo que temos códigos com propósitos distintos misturados, o que poderia ser evitado se usássemos POA para criar um aspecto relacionado com a depuração. Dessa forma, no arquivo que descreve as instruções da arquitetura

```
1  void ac_behavior( addiu )
2  {
3    dbg_printf("addiu r%d, r%d, %d\n", rt, rs, imm & 0xFFFF);
4    RB[rt] = RB[rs] + imm;
5    dbg_printf("Result = %#x\n", RB[rt]);
6  };
```

Figura 34: Decrição da Instrução *addiu* Utilizando ArchC

MIPS, teríamos agora somente o código necessário para o correto funcionamento do modelo sendo descrito, não existindo mais a necessidade de se preocupar com o interesse relacionado com a depuração do sistema.

Implementação/Conclusão

Visto que a geração de um simulador baseado em SystemC a partir da descrição de uma arquitetura utilizando ArchC foi implementada apoiando-se no processamento de macros, e tendo em vista que a ferramenta AspectC++ não é capaz ainda de lidar de maneira adequada com o código gerado a partir de macros, não é possível inserir nos pontos de junção o código de comportamento transversal necessário ao nosso aspecto de depuração, de forma que o exemplo aqui proposto da utilização da POA juntamente com ArchC é apenas uma idéia, não possuindo uma implementação viável ainda.

4.3 Análise de Desempenho

Para analisar o impacto da utilização da POA no desempenho do sistema sendo construído, resolvemos medir o tempo médio de execução de alguns dos exemplos anteriormente apresentados. Foram selecionadas então três aplicações:

1. A aplicação que realiza a transformada rápida de Fourier, a qual foi abordada na seção 4.1.3. Será usada a versão da aplicação que lida com números inteiros, cujo aspecto correspondente encontra-se descrito em linhas gerais na figura 27;

2. A aplicação que realiza a comunicação entre dois módulos, usando para isto o protocolo de comunicação OCP, a qual foi descrita na seção 4.1.1;

3. A terceira aplicação selecionada consiste da política de troca de blocos da cache do processador GPOP e foi descrita na seção 4.1.2. Para medir o desempenho desta aplicação tomaremos como nosso modelo a política de troca de blocos FIFO.

Exemplo	Ciclos	Tempo Médio
FFT para Números Inteiros	450	0.172s
Comunicação usando OCP	1000000	0.848s
Política de Cache FIFO	300000	298.672s

Tabela 1: Desempenho das Aplicações sem usar POA

Exemplo	Ciclos	Tempo Médio
FFT para Números Inteiros	450	0.184s
Comunicação usando OCP	1000000	0.839s
Política de Cache FIFO	300000	300.280s

Tabela 2: Desempenho das Aplicações usando POA

Deve-se notar que a adoção de técnicas de POA também requer que se gaste um pouco mais de tempo no processo de compilação dos módulos, visto que eles terão que ser combinados com os aspectos agora.

Antes de apresentarmos os resultados obtidos, vale ressaltar que o impacto da POA no tempo de simulação de uma aplicação depende da quantidade de informação acessada em cada ponto de junção pelo código de comportamento transversal, bem como da capacidade do compilador final de tornar o código *inline*.

O tempo médio das simulações foi medido utilizando-se a seguinte abordagem:

- Utilizou-se um computador com sistema operacional Linux;
- Existia somente um único usuário conectado que estava utilizando uma sessão de modo texto;
- Cada programa foi executado 10 vezes, tendo seu tempo de execução medido pelo aplicativo *time*;
- O tempo médio de simulação foi determinado pela soma do tempo de todas as simulações daquela aplicação dividida por 10.

Na tabela 1, podemos ver o tempo médio de simulação das aplicações selecionadas sem o uso de aspectos, enquanto que a tabela 2 nos fornece o tempo médio das aplicações utilizando aspectos.

Podemos notar que ocorreu um pequeno aumento no tempo de simulação das aplicações quando foi utilizada POA. Porém, a aplicação número 2 (Comunicação usando OCP) apresen-

Exemplo	Ciclos	Tempo Médio
Comunicação usando OCP sem POA	1000000	1.460s
Comunicação usando OCP com POA	1000000	1.495s

Tabela 3: Desempenho da Aplicação 2 sem Otimização

Exemplo	Impacto no Tempo de Simulação
FFT para Números Inteiros	$+7\%$
Comunicação usando OCP com $-O3$	-1%
Política de Cache FIFO	$+1\%$
Comunicação usando OCP com $-O0$	$+2\%$

Tabela 4: Impacto da POA no Tempo de Simulação das Aplicações

tou uma diminuição no seu tempo da simulação quando foram utilizados aspectos, provavelmente devido a questões de otimização do compilador, já que os arquivos foram compilados no compilador $g{+}{+}$ usando a opção de otimização $-O3$. Em virtude disto, este exemplo foi avaliado novamente, desta vez com a opção de otimização $-O0$. O resultado desta nova avaliação pode ser visto na tabela 3. Desta vez, a versão da aplicação que não utiliza aspectos apresentou um tempo de simulação menor, como era esperado.

A tabela 4 apresenta em termos percentuais qual foi o impacto do uso da POA no tempo de simulação das aplicações escolhidas. Um percentual negativo indica que houve uma diminuição no tempo de simulação e um percentual positivo indica que ocorreu um aumento no tempo de simulação. Como pode ser visto, não ocorreu nenhuma alteração significativa no tempo de simulação das aplicações em virtude do uso da POA, o que mostra, para os exemplos apresentados aqui, que o uso da POA não deve ser restringido por questões de desempenho.

Também comparamos qual foi o tamanho do arquivo SystemC executável gerado em cada aplicação quando foi utilizada a POA. Na tabela 5, podemos ver então qual foi o tamanho, em bytes, do arquivos executáveis gerados.

A tabela 6 informa, em termos percentuais, qual foi o impacto da utilização da POA nos arquivos executáveis gerados. Podemos notar que o impacto causado pela POA no tamanho do

Exemplo	Impacto no Arquivo de Simulação	Tamanho com POA
FFT para Números Inteiros	838813	839530
Comunicação usando OCP com	2290990	2352625
Política de Cache FIFO	1312486	1312712

Tabela 5: Tamanho do Arquivo de Simulação Sem e Com o uso de POA

Exemplo	Impacto no Tamanho do Arquivo Executável
FFT para Números Inteiros	0%
Comunicação usando OCP com	+3%
Política de Cache FIFO	0%

Tabela 6: Impacto da POA no Tamanho do Arquivo de Simulação da Aplicações

arquivo executável SystemC gerado é bem pequena. Devemos notar ainda que com o uso de POA, podemos chegar a obter inclusive um arquivo de simulação de tamanho menor, visto que é possível escolher que aspectos devem atuar sobre o sistema, não sendo necessário gerar um modelo com todas as propriedades disponíveis e que possuiria um tamanho maior.

4.4 Conclusões

As aplicações mostradas neste capítulo mostraram como a POA pode auxiliar no projeto de sistemas desenvolvidos com SystemC. A POA pode ser utilizada desde as primeiras fases do desenvolvimento do sistema, onde deve-se decidir que propriedades do sistema poderiam se encapsuladas como um aspecto, melhorando assim a separação de interesses do sistema, auxiliando a modificação de propriedades do mesmo e facilitando a exploração do espaço de projeto.

A POA também pode ser utilizada juntamente com técnicas de especificação/verificação formal, ajudando na aplicação destas técnicas, representando contratos como aspectos ou determinando quais restrições se aplicam a um determinado objeto.

Por fim, podemos utilizar aspectos também na hora de simular o sistema, através da coleta de informações ou da geração de informações de log. Vimos também que o impacto da POA no tempo de simulação da aplicação é bastante reduzido, bem como o seu impacto sobre o tamanho do modelo executável do sistema gerado, de modo que estes não são fatores que devam limitar o uso da POA juntamente com SystemC.

5 Conclusões

A programação orientada a aspectos é uma das abordagens de maior sucesso entre os mecanismos pós-OO, pois propicia uma melhora no entendimento, na manutenção e na reusabilidade do código do sistema que é projetado fazendo uso deste novo paradigma.

Baseando-se em uma tendência das linguagens de descrição de hardware de adotarem novos mecanismos que surjam nas linguagens de programação, decidimos então estudar a relevância do paradigma POA no projeto de sistemas hardware. Para tal fim, usamos SystemC como nossa linguagem de descrição de hardware e a ferramenta AspectC++ para tornar possível a utilização da POA juntamente com a linguagem de programação C++.

A partir desta idéia, foram desenvolvidos vários exemplos de aplicações utilizando POA juntamente com SystemC, abordando questões como: política de troca de linhas da cache, verificação do sistema, simulação, comunicação etc. Estes exemplos procuraram mostrar a viabilidade do uso de POA no projeto de sistemas utilizando SystemC, tentando construir aplicações que pudessem se beneficiar do uso da POA de diferentes maneiras.

Desta forma, foi possível separar interesses do sistemas que se encontravam entrelaçados ao longo do mesmo, bem como encontravam-se espalhados por vários arquivos. Alguns destes interesses foram então representados como aspectos, concentrando-se assim todo o código que dizia respeito a um determinado interesse em um único lugar, como foi apresentado no exemplo da coleta de métricas.

A POA também mostrou-se útil para realizar uma rápida exploração do espaço de projeto, ao possibilitar a avaliação de diferentes implementações de uma política de troca de linhas da cache, bastando para isso somente a alteração de uma diretiva de compilação, sem a necessidade de fazer modificações no código do sistema principal.

Esperamos que os vários exemplos apresentados neste trabalho sejam úteis e possam vir a incentivar desenvolvedores SystemC a incluir a metologia POA nos seus projetos, utilizando-os como um ponto de partida ou adaptando-os para as suas próprias necessidades.

Durante o presente trabalho, uma dificuldade que surgiu várias vezes foi a utilização da ferramenta AspectC++, a qual evoluiu muito durante o decorrer do trabalho, para realizar a combinação dos aspectos. Inicialmente, não era possível gerar um modelo SystemC executável do sistema, devido a um problema durante a ligação dos arquivos objeto. Depois, notou-se que algumas versões da ferramenta não apresentavam características descritas no manual de referência da mesma, de forma que foi necessário trabalhar com várias versões da ferramenta, às vezes com versões mais antigas, mas que ofereciam uma funcionalidade que não estava presente na versão mais atual. Por fim, AspectC++, na sua versão 1.0*pre*1, apresentou um tempo de compilação muito elevado para os arquivos SystemC, o que tornava lento e tedioso o trabalho de recompilação de arquivos.

Estes problemas foram resolvidos através da troca de mensagens com os autores da ferramenta, que na sua versão atual, 1.0*pre*2, não apresenta estes problemas. Porém, AspectC++ não é capaz de lidar muito bem ainda com *templates* e macros, o que impossibilitou a implementação de algumas possíveis aplicações.

Como foi dito anteriormente, o impacto do uso da POA depende de cada aplicação e da quantidade de informação que o código do aspecto acessa em cada ponto de junção. Os exemplos aqui apresentados, os quais possuíam aspectos que acessavam informação dos pontos de junção a cada ciclo, mostraram que não houve muito impacto no tempo de simulação da aplicação devido ao uso de aspectos, o que é uma boa indicação.

O exemplo relacionado com a política de cache foi testado no projeto de um processador paralelo, denominado GPOP, composto de mais de 30 módulos SystemC e com mais de 11000 linhas de código, o que indica que é possível a utilização da metologia orientada a aspectos em um projeto de maior porte.

A realização deste trabalho também auxiliou no desenvolvimento da ferramenta AspectC++, uma vez que vários problemas da ferramenta foram detectados após tentativas de se utilizar AspectC++ com aplicações SystemC.

Para uma possível continuação deste trabalho, seria interessante um maior contato com desenvolvedores de aplicações SystemC, os quais pudessem fornercer vários exemplos de aplicações SystemC reais, bem como pudessem auxiliar na avaliação das soluções propostas utilizando POA, de forma que elas atendessem às suas necessidades e pudessem ser testadas no decorrer de futuros projetos, de forma que tais soluções pudessem ser validadas, rejeitadas ou modificadas para se adequar melhor a um contexto.

Como trabalho futuro, poderiam ser estudados outros paradigmas de software que, assim

como o DBC, são aplicados na área de projeto de hardware e que têm sua utilização facilitada/melhorada com a ajuda da POA. Um exemplo de tais paradigmas são os Padrões de Projeto (GAMMA et al., 1995), um paradigma de software que também é aplicado no projeto de hardware (RINCON et al., 2005; DAMASEVICIUS; MAJAUSKAS; STUIKYS, 2003), e que pode ser melhorado com a utilização de técnicas de POA (GARCIA et al., 2005), propiciando assim um aperfeiçoamento nos sistemas nos quais os Padrões de Projeto são aplicados.

Esperamos assim, com este trabalho, ter contribuído com uma nova proposta de desenvolvimento de sistemas hardware/software que possibilite melhorias no projeto dos mesmos.

Referências

AL-ZOUBI, H.; MILENKOVIC, A.; MILENKOVIC, M. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In: *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*. New York, NY, EUA: ACM Press, 2004. p. 267–272. ISBN 1-58113-870-9.

ASPECTC++. *The Home of AspectC++*. 2006. Disponível em http://www.aspectc.org. Acesso em Janeiro de 2006.

ASPECTJ. *aspectj - crosscutting objects for better modularity*. 2006. Disponível em http://eclipse.org/aspectj. Acesso em janeiro de 2006.

BARTLETT, J. *The art of metaprogramming*. 2005. Disponível em http://www-128.ibm.com/developerworks/linux/library/l-metaprog1.html. Acesso em Janeiro de 2006.

BERGAMASCHI, R. A. The A to Z of SoCs. In: *International Conference on Computer-Aided Design*. San Jose, California: IEEE, 2002. p. 791–798.

BERGMANS, L.; AKSIT, M. Composing crosscutting concerns using composition filters. *Communications of the ACM*, ACM Press, New York, NY, EUA, v. 44, n. 10, p. 51–57, 2001. ISSN 0001-0782.

BORRIONE, D. et al. Three Decades of HDLs: Part II, Conlan Through Verilog. *IEEE Design & Test*, IEEE Computer Society Press, v. 9, n. 3, 1992.

BRUNEL, J.-Y. et al. Softcontract: an Assertion-Based Software Development Process that Enables Design-by-Contract. In: *DATE '04: Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, EUA: IEEE Computer Society, 2004. p. 10358. ISBN 0-7695-2085-5-1.

CHAREST, L.; ABOULHAMID, E. M. A VHDL/SystemC Comparison in Handling Design Reuse. In: *International Workshop on System-on-Chip for Real-Time Applications*. Canada: [s.n.], 2002. p. 79–85.

CHEN WEIDONG QIU, B. Z. Y.; PENG, C. An Automatic Test Coverage Analysis for SystemC Description Using Aspect-Oriented Programming. In: *The 8th International Conference on Computer Supported Cooperative Work in Design Proceedings*. [S.l.]: IEEE Computer Society, 2004. p. 632–636.

CHU, Y. et al. Three Decades of HDLs: Part I, CDL Through TI-HDL. *IEEE Design & Test*, IEEE Computer Society Press, v. 9, n. 2, 1992.

CZARNECKI, K.; EISENECKER, U. *Generative Programming: Methods, Tools and Applications*. [S.l.]: Addison-Wesley Professional, 2000.

DAMASEVICIUS, R.; MAJAUSKAS, G.; STUIKYS, V. Application of design patterns for hardware design. In: *DAC '03: Proceedings of the 40th conference on Design automation*. New York, NY, EUA: ACM Press, 2003. p. 48–53. ISBN 1-58113-688-9.

DIJKSTRA, E. W. *A Discipline of Programming*. Upper Saddle River, NJ, EUA: Prentice Hall, 1976.

DIOTALEVI, F. *Contract enforcement with AOP*. 2004. Disponível em http://www-128.ibm.com/developerworks/library/j-ceaop/. Acesso em Janeiro de 2006.

ELRAD, T. et al. Discussing aspects of AOP. *Communications of the ACM*, ACM Press, v. 44, n. 10, 2001.

ELRAD, T.; FILMAN, R. E.; BADER, A. Aspect-oriented programming: Introduction. *Communications of the ACM*, ACM Press, v. 44, n. 10, 2001.

FERNANDES, F. *Combinando Aspectos e Componentes: uma abordagem interpretada*. Dissertação (Mestrado) — UFRN, 2004.

FIN, A. et al. Systemc-A Homogeneous Environment to Test Embedded Systems. In: *9th International Symposium on Hardware-Software Codesign*. Denmark: ACM Press, 2001. p. 17–22.

GAMMA, E. et al. *Design patterns: elements of reusable object-oriented software*. Boston, MA, EUA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.

GARCIA, A. et al. Modularizing design patterns with aspects: a quantitative study. In: *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*. New York, NY, EUA: ACM Press, 2005. p. 3–14. ISBN 1-59593-042-6.

HARRISON, W.; OSSHER, H. Subject-oriented programming: a critique of pure objects. In: *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*. New York, NY, EUA: ACM Press, 1993. p. 411–428. ISBN 0-89791-587-9.

HENNESSY, J. et al. MIPS: A microprocessor architecture. In: *MICRO 15: Proceedings of the 15th annual workshop on Microprogramming*. Piscataway, NJ, EUA: IEEE Press, 1982. p. 17–22.

JAKACKI, G. Aspect-Oriented Techniques for Extraction of Communication Models from Systemc Designs. In: *Proceedings of the 5th International Conference on ASIC*. [S.l.]: IEEE Computer Society, 2003. p. 262–265.

JANTSCH, A.; TENHUNEN, H. (Ed.). *Networks on chip*. Hingham, MA, EUA: Kluwer Academic Publishers, 2003. ISBN 1-4020-7392-5.

JOHNSON, D. M. The systems engineer and the software crisis. *SIGSOFT Software Engineering Notes*, ACM Press, New York, NY, EUA, v. 21, n. 2, p. 64–73, 1996. ISSN 0163-5948.

KASUYA, E. H. A.; TESFAYE, T. *Jeda Tutorial*. [S.l.], 2005. Acesso em Janeiro de 2006.

KICZALES, G. et al. Getting-Started wit Aspectj. *Communications of the ACM*, ACM Press, v. 44, n. 10, 2001.

KICZALES, G. et al. Aspect-Oriented Programming. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Finland: Springer-Verlag, 1997. (Lecture Notes in Computer Science, 1241).

KORDON, F.; HENKEL, J. An Overview of Rapid System Prototyping Today. *Design Automation for embedded Systems*, Kluwer Academic Publishers, v. 8, n. 4, p. 275–282, 2003.

LEWIS, J. A. et al. An empirical study of the object-oriented paradigm and software reuse. In: *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*. New York, NY, EUA: ACM Press, 1991. p. 184–196. ISBN 0-201-55417-8.

LIAO, S. Y. Towards a new standard for system-level design. In: *International Conference on Hardware Software Codesign*. United States: ACM Press, 2000. p. 2–6.

LIEBERHERR, K. J. Demeter/adaptive programming. *SIGSOFT Software Engineering Notes*, ACM Press, New York, NY, EUA, v. 25, n. 1, p. 100–101, 2000. ISSN 0163-5948.

MEYER, B. *Object-Oriented Software Construction*. [S.l.]: Prentice Hall PTR, 2000.

MURPHY, G. C.; NOTKIN, D.; SULLIVAN, K. Software reflexion models: bridging the gap between source and high-level models. In: *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*. New York, NY, EUA: ACM Press, 1995. p. 18–28. ISBN 0-89791-716-2.

OCP-IP ASSOCIATION. *Open Core Protocol Specification*. [S.l.], 2003.

(OSCI) Open SystemC Initiative. *Draft Standard SystemC Language Manual*. [S.l.], 2005.

PANDA, P. R. SystemC: a modeling platform supporting multiple design abstractions. In: *International Symposium on Systems Synthesis*. Canada: ACM Press, 2001. p. 75–80.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. [S.l.]: Mcgraw-Hill College, 1996.

pure-systems GmbH E OLAF SPINCZYK. *AspectC++ Language Reference*. [S.l.], 2004.

RIGO, S. et al. ArchC: A SystemC-Based Architecture Description Language. In: *SBAC '04: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*. [S.l.]: IEEE Computer Society, 2004. p. 66–73.

RINCON, F. et al. Model Reuse through Hardware Design Patterns. In: *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, EUA: IEEE Computer Society, 2005. p. 324–329. ISBN 0-7695-2288-2.

SAVAGE, W.; CHILTON, J.; CAMPOSANO, R. Ip reuse in the system on a chip era. In: *ISSS '00: Proceedings of the 13th international symposium on System synthesis*. Washington, DC, EUA: IEEE Computer Society, 2000. p. 2–7. ISBN 1080-1082.

SOARES, R. *Projeto e Implementação de uma Plataforma MP-SoC usando SystemC*. Dissertação (Mestrado) — UFRN, 2006.

SPINCZYK, O.; GAL, A.; SCHRöDER-PREIKSCHAT, W. Aspectc++: An Aspect-Oriented Extension to the C++ Programming Language. In: NOBLE, J.; POTTER, J. (Ed.). *40th International Conference on Techonology of Object-Oriented Languages and Systems (TOOLS) Pacific 2002*. Australia: Australian Computer Society, 2002. v. 10, p. 53–60.

SYSTEMC. *SystemC Verification Library*. 2003. Disponível em http://www.systemc.org. Acesso em Janeiro de 2006.

SYSTEMC. *The Open SystemC Initiative*. 2005. Disponível em http://www.systemc.org. Acesso em Janeiro de 2006.

THEKKATH, C. A.; LEVY, H. M.; LAZOWSKA, E. D. Separating data and control transfer in distributed operating systems. *SIGPLAN Not.*, ACM Press, New York, NY, EUA, v. 29, n. 11, p. 2–11, 1994. ISSN 0362-1340.

VACHHARAJANI, M.; VACHHARAJANI, N.; AUGUST, D. I. The liberty structural specification language: a high-level modeling language for component reuse. In: *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. New York, NY, EUA: ACM Press, 2004. p. 195–206. ISBN 1-58113-807-5.

WEINER, L. H. The roots of structured programming. In: *Papers of the SIGCSE/CSA technical symposium on Computer science education*. New York, NY, EUA: ACM Press, 1978. p. 243–254.

YANG, Y.; LI, D. Separating data and control: support for adaptable consistency protocols in collaborative systems. In: *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*. New York, NY, EUA: ACM Press, 2004. p. 11–20. ISBN 1-58113-810-5.

APÊNDICE A – O Protocolo OCP

A.1 Características OCP

O OCP define uma interface ponto-a-ponto entre duas entidades que estão se comunicando, tais como componentes IP e módulos de interface de barramento (*wrappers* de barramento).

Para que duas entidades realizem uma comunicação ponto-a-ponto é necessário que existam duas instâncias do OCP conectando-as — uma onde a primeira entidade é um mestre, e uma onde a primeira entidade é um escravo.

Somente o mestre pode enviar comandos, portanto, o mestre é a entidade que controla a comunicação. O escravo responde aos comandos enviados a ele, ou aceitando o dado enviado pelo mestre, ou enviando dados ao mestre.

As características do componente IP irão determinar se o componente necessitirá agir como mestre OCP, escravo OCP ou como ambos. Os módulos da interface *wrapper* devem agir então como o lado complementar do OCP para cada entidade conectada.

Um componente IP que tome a iniciativa da comunicação (agindo como um mestre OCP) envia comandos e dados para o escravo ao qual está conectado (um módulo da interface *wrapper* do barramento). O módulo da interface leva a requisição através do sistema de barramento *on-chip*. O OCP não especifica a funcionalidade do barramento embarcado. Ao invés disso, o projetista da interface converte a requisição OCP em uma transferência do barramento embarcado. Assim, ao enviar uma requisição o mestre OCP converte a operação do barramento embarcado em um comando OCP legal. O sistema alvo (o escravo OCP) então recebe o comando e realiza a ação requisitada. Na figura A.1 podemos ver o exemplo de um sistema utilizando OCP. Na figura, os componentes que agem como um mestre OCP são os *Iniciadores do Sistema*, enquanto os componentes que agem como um escravo OCP são os *Alvos do Sistema*. Um componente utilizando OCP pode ser conectado a qualquer barramento, então dois componentes que usem OCP, sendo um o mestre e o outro o escravo, podem se comunicar sem a necessidade de um barramento *on-chip*.

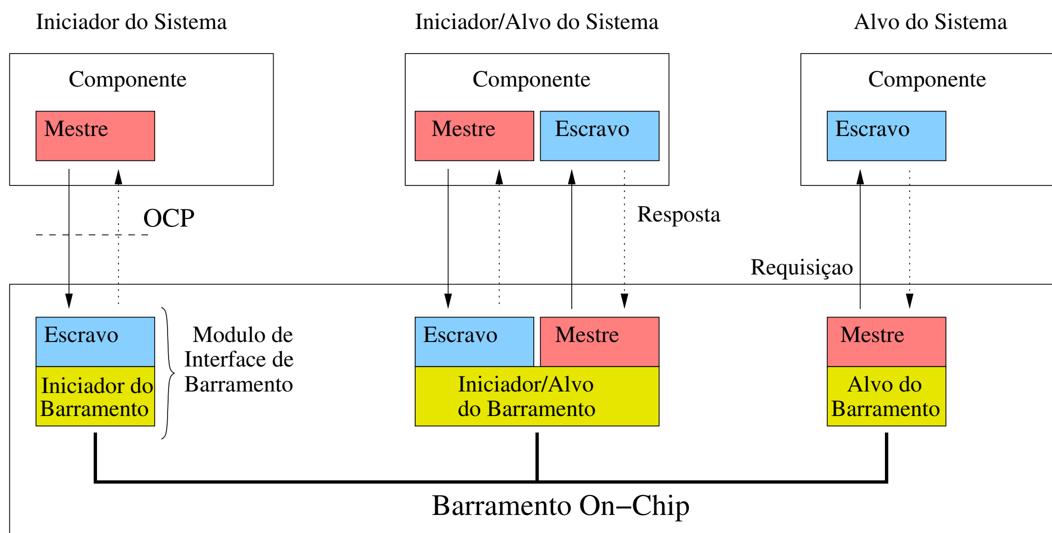

Figura 35: Exemplo de um Sistema Utilizando OCP

Cada instância do OCP é configurada (através da escolha do conjunto de sinais que ela deve implementar ou do comprimento em bits de um sinal particular) com base nos requerimentos das entidades conectadas e a configuração de cada instância do OCP é independente das outras.

Sistemas iniciadores podem, por exemplo, requererem mais endereços de bits em suas instâncias OCP do que sistemas alvo; os endereços de bits adicionais podem vir a ser utilizados pelo barramento embarcado para selecionar qual barramento alvo está sendo endereçado pelo sistema iniciador.

A.2 Sinais e Codificação

A interface OCP é composta por um conjunto de sinais OCP os quais são agrupados em três grupos:

- Sinais de fluxo de dados: são os sinais relacionados com a transferência de dados entre os componentes OCP;
- Sinais de *sideband*: são todos os sinais do OCP que não estão diretamente relacionados com o fluxo de dados, tais como sinais de controle ou que indiquem eventos do sistema;
- Sinais de teste: são sinais adicionais que oferecem suporte a características como varredura e controle de *clock*

Nome	Comprimento	Emissor	Função
Clk	1	varia	OCP clock
MAddr	configurável	mestre	Endereço de transferência
MCmd	3	mestre	Comando de transferência
MData	configurável	mestre	Dado a escrever
MDataValid	1	mestre	Dado a escrever válido
MRespAccept	1	mestre	Mestre aceita a resposta
SCmdAccept	1	escravo	Escravo aceita a transferência
SData	configurável	escravo	Dado a ler
SDataAccept	1	escravo	Escravo aceita dado a escrever
SResp	2	escravo	Resposta da transferência

Tabela 7: Sinais OCP básicos

Os sinais de fluxo de dados são divididos em: sinais básicos, extensões simples, extensões de rajada e extensões de *thread*. Um pequeno conjunto dos sinais básicos relacionados ao fluxo de dados é necessário em todas as configurações OCP. Sinais opcionais podem ser configurados para suportar requerimentos de comunicação adicionais dos componentes. Todos os sinais de teste e de *sideband* são opcionais e não serão abordados neste documento.

O OCP é uma interface síncrona com um único sinal de *clock*. Todos os sinais OCP são dirigidos com respeito à, e amostrados pela, subida do *clock* OCP. Exceto para o *clock*, todos os sinais OCP são estritamente ponto-a-ponto e unidirecionais. Iremos apresentar a seguir, os sinais básicos relacionados com o fluxo de dados.

A.2.1 Sinais Básicos do Fluxo de Dados

Os sinais de fluxo de dados consistem de um pequeno conjunto de sinais necessários e um número de sinais opcionais que podem ser configurados para suportar requerimentos de comunicação adicionais dos componentes. Os sinais de fluxo de dados são agrupados em: sinais básicos (os quais veremos mais adiante), extensões simples (que adicionam opções tais como informação *in-band*), extensões de rajada (as quais adicionam suporte à comunicação por rajada), e extensões de thread (que adicionam suporte a várias threads).

As convenções de nome para sinais de fluxo de dados usam o prefixo **M** para sinais que são dirigidos pelo mestre OCP e o prefixo **S** para sinais que são dirigidos pelo escravo OCP.

Na tabela 7 vemos a lista dos sinais OCP básicos. Somente **Clk** e **MCmd** são necessários. Os sinais OCP restantes são opcionais.

- **Clk**: o sinal *clock* para o OCP. Todos os sinais de interface são sincronizados com a

subida de *Clk*. *Clk* é emitido por uma terceira entidade e serve como entrada tanto para o mestre como para o escravo.

- **MAddr**: especifica o endereço relacionado com o escravo que possui o recurso almejado pela transferência atual.

- **MCmd**: é o comando de transferência que indica o tipo de transferência OCP que o mestre está requisitando. A codificação dos comandos é mostrada na tabela 8. Cada comando diferente de *IDLE* é uma requisição de leitura ou de escrita, dependendo da direção do fluxo de dados.

  Há dois comandos básicos, *READ* e *WRITE*, e cinco extensões de comando. Os comandos *WriteNonPost* e *Broadcast* possuem uma semântica similiar àquela do comando *Write*. Um comando *WriteNonPost* explicitamente instrui o escravo OCP a não enviar uma notificação de resposta após um comando de escrita oriundo do mestre. Com o comando *Broadcast* o mestre indica que está tentando escrever para vários ou para todos os dispositivos alvo remotos que estão conectados ao outro lado de um escravo. Desta forma, o comando *Broadcast* geralmente é útil somente para se comunicar com escravos que agem como um mestre OCP quando se comunicam em um outro meio (como em um barramento ligado a ele por exemplo).

  As outras extensões de comando, *ReadExclusive*, *ReadLinked* e *WriteConditional*, são usadas para sincronização entre sistemas iniciadores. *ReadExclusive* faz par com *Write* ou com *WriteNonPost*, e tem semântica bloqueante. *ReadLinked*, usado juntamente com *WriteConditional*, tem semântica não bloqueante preguiçosa. Estas primitivas de sincronização correspondem àquelas disponíveis nativamente nos conjuntos de instruções de diferentes processadores.

  O uso de comandos pode ser limitado através do uso de variáveis que habilitam/desabilitam comandos.

- **MData**: este sinal carrega o dado a escrever do mestre para o escravo. O seu comprimento pode ser configurado, não sendo restrito a múltiplos de 8.

- **MDataValid**: quando em 1, este bit indica que o dado no sinal *MData* é válido.

- **MRespAccept**: o mestre indica que aceita a resposta atual do escravo colocando o valor 1 neste sinal.

- **SCmdAccept**: quando este sinal possui o valor 1 isto indica que o escravo aceita a requisição de transferência do mestre.

MCmd$[2:0]$	**Comando**	**Mnemônico**	**Tipo de Requisição**
0 0 0	Idle	IDLE	nenhuma
0 0 1	Write	WR	escrita
0 1 0	Read	RD	leitura
0 1 1	ReadEx	RDEX	leitura
1 0 0	ReadLinked	RDL	leitura
1 0 1	WriteNonPost	WRNP	escrita
1 1 0	WriteConditional	WRC	escrita
1 1 1	Broadcast	BCST	escrita

Tabela 8: Codificação dos Comandos OCP

SResp$[1:0]$	**Resposta**	**Mnemônico**
0 0	Sem reposta	NULL
0 1	Data válido/aceito	DVA
1 0	Falha na requisição	FAIL
1 1	Erro de resposta	ERR

Tabela 9: Codificação da Resposta do Escravo OCP

- **SData**: este sinal carrega o dado a ler requisitado pelo mestre. O comprimento deste sinal pode ser configurado, não sendo restrito a múltiplos de 8.

- **SDataAccept**: o escravo indica que ele aceita o dado do mestre colocado no *pipeline* através de um valor 1 neste sinal. Este sinal possui significado somente quando o parâmetro *datahandshake* está em uso.

- **SResp**: é o sinal de resposta do escravo para uma requisição de transferência do mestre. A codificação da resposta é mostrada na tabela 9.

APÊNDICE B – Implementação da Comunicação Utilizando OCP e Aspectos

B.1 specs.h

```
#ifndef SPECS_H
#define SPECS_H

#define WORD  8
#define ADDR 8
#define MEMORYSIZE 256
#define MCMDSIZE 3
#define SRESPSIZE 2
#define NUMOP 3
#define OK 1
#define ERROR 2

/* OCP Mnemonics */

/* MCmd (3) */
#define IDLE 0
#define WR 1
#define RD 2
#define RDEX 3
#define RDL 4
#define WRNP 5
#define WRC 6
#define BCST 7

/* SResp (2) */
//#define NULL 0
#define DVA 1
#define FAIL 2
#define ERR 3
```

```
enum Operation {READ, WRITE, SKIP};
enum State {FREE, READING, WRITING, RESET};

#endif //SPECS_H
```

B.2 memory.h

```
#ifndef MEMORY_H
#define MEMORY_H

#include "systemc.h"

#include "specs.h"
#include <iostream.h>

SC_MODULE (Memory) {
   sc_in <sc_int<WORD> > data_in;
   sc_in <sc_int<ADDR> > addr_in;
   sc_in <bool> clk, rst;

   sc_out <sc_int<WORD> > data_out;
   sc_out <int> available_out;

   private:
   bool read, write;

   public:
   void doit();
   void reset();
   void print(int);
   void setRead(bool b) {read = b;}
   bool getRead() {return read;}
   void setWrite(bool b) {write = b;}
   bool getWrite() {return write;}
   State getState() {return currentState;}

   sc_int <WORD> memory[MEMORYSIZE];

   SC_CTOR (Memory) {
```

```
      currentState = RESET;

      setRead (false);
      setWrite (false);

      SC_METHOD(doit);
      sensitive_pos << clk;
   }

   State currentState;

   private:
      State nextState;

};

#endif //MEMORY_H
```

B.3 memory.cpp

```
#include <iostream.h>

#include "memory.h"

void
Memory::doit() {

   print(0);

   switch (currentState) {

      case (RESET):
         {
            reset();
            print(1);
            nextState = FREE;
            break;
         }

      case (FREE):
         {
            if (read) {
```

```
            nextState = READING;
         } else if (write) {
            nextState = WRITING;
         }
         break;
      }

   case (WRITING):
      {
         cout << "Escrevi na memoria\n\n";
         if (write == false) {
            nextState = FREE;
         }
         break;
      }

   case (READING):
      {
         cout << "Li da memoria\n\n";
         if (read == false) {
            nextState = FREE;
         }
         break;
      }

   default:
      break;
   }

   currentState = nextState;

}

void
Memory::print(int mode) {
   cout << "Memory State: ";
   switch (currentState) {
      case FREE:
         cout << "IDLE\n";
         break;

      case READING:
         cout << "READING\n";
```

```
         break;

      case WRITING:
         cout << "WRITING\n";
         break;

      case RESET:
         cout << "RESET\n";
         break;
   }

   cout << "\n";

   if (mode > 0) {
      int i = 0;
      while (i < MEMORYSIZE/2 - 1) {
         for (int j=0; j<30; ++j, ++i) {
            cout << memory[i] << ' ';
            if (i == MEMORYSIZE/2 - 1) {
               ++i;
               break;
            }
         }
         cout << endl;
      }
      cout << endl << endl;
   }
}


void
Memory::reset() {
   for(int i=0; i<MEMORYSIZE; ++i) {
      memory[i] = i;
   }

}
```

B.4 application.h

```
#ifndef APPLICATION_H
#define APPLICATION_H
```

```
#include "systemc.h"

#include "specs.h"

SC_MODULE (Application) {
   sc_in <sc_int<WORD> > DataIn;
   sc_in <bool> clk, rst;
   sc_in <int> availableIn;
   sc_in <int> writingDone;

   sc_out <sc_int<WORD> > DataOut;
   sc_out <sc_int <ADDR> > addr;

   void doit();
   void reset();
   void print(int);
   State getState();

   sc_int <WORD> memory[MEMORYSIZE];

   SC_CTOR (Application) {
      currentState = RESET;

      SC_METHOD (doit);
      sensitive_pos << clk;
   }

   State currentState, nextState;
   int localAddr, remoteAddr;

   private:
      Operation op;

};
#endif //APPLICATION_H
```

B.5 application.cpp

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
```

```
#include "application.h"

void
Application::doit() {
   if (rst.read() == 1) {
      reset();
   }

   print(0);

   switch (currentState)
   {
      case (RESET):
         {
            reset();
            print(1);
            nextState = FREE;
            break;
         }

      case (FREE):
         {
            op = (Operation) (rand() % NUMOP);
            localAddr = rand() % (MEMORYSIZE / 2);
            remoteAddr = rand() % (MEMORYSIZE / 2);

            if (op == READ) {  //read a value from addr2 and store it into addr1
               cout << "READ:\n";
               addr.write (remoteAddr);
               nextState = READING;
            }
            else if (op == WRITE) {  //write the value of addr1 into addr2
               cout << "WRITE:\n";
               DataOut.write (memory[localAddr]);
               addr.write(remoteAddr);
               nextState = WRITING;
            }

            else if (op == SKIP) {
               cout << "SKIP:\n\n";
               nextState = FREE;
            }
```

```
            break;
         }

      case (READING):
         {
            if (availableIn.read() != 0) {
               if (availableIn.read() == OK) {
                  memory [localAddr] = DataIn.read();
               }
               else if (availableIn.read() == ERROR) {
                  cout << "Error READING" << endl;
               }
               nextState = FREE;
            } else {
               nextState = READING;
            }
            break;
         }

      case (WRITING):
         {
            if (writingDone.read () == OK) {
               nextState = FREE;
            } else if (writingDone.read () == ERROR) {
               cout << "Error WRITING" << endl;
               nextState = FREE;
            } else {
               nextState = WRITING;
            }
            break;

         }

      default:
         break;
   }

   currentState = nextState;

}

State
```

```
Application::getState() {
   return currentState;
}

void
Application::reset() {
   srand (time(0));

   currentState = RESET;

   for (int i=0; i<MEMORYSIZE; ++i)
      memory[i] = i;
}

void
Application::print(int mode) {
   cout << "Application State: ";
   switch (currentState) {
      case FREE:
         cout << "IDLE\n";
         break;

      case READING:
         cout << "READING1\n";
         break;

      case WRITING:
         cout << "WRITING1\n";
         break;

      case RESET:
         cout << "RESET\n";
         break;

      default:
         cout << currentState << endl;
         break;
   }

   cout << "\n";

   if (mode > 0) {
      int i = 0;
```

```
      while (i < MEMORYSIZE/2 - 1) {
         for (int j=0; j<30; ++j, ++i) {
            cout << memory[i] << ' ';
            if (i == MEMORYSIZE/2 - 1) {
               ++i;
               break;
            }
         }
         cout << endl;
      }
      cout << endl << endl;
   }
}
```

B.6 slave.h

```
#ifndef SLAVE_H
#define SLAVE_H

#include "systemc.h"

#include "specs.h"
#include "memory.h"

SC_MODULE (Slave) {
   //OCP PORTS

   //Master
   sc_in <sc_int<ADDR> > MAddr, MData;
   sc_in <sc_uint<MCMDSIZE> > MCmd;
   sc_in <bool> MDataValid, MRespAccept, Clk;

   //Slave
   sc_out <sc_int<WORD> > SData;
   sc_out <sc_uint<SRESPSIZE> > SResp;
   sc_out <bool> SCmdAccept, SDataAccept;

   void doit();
   void print(int);

   SC_CTOR (Slave) {
```

```
      SC_METHOD(doit);
      sensitive_pos << Clk;

   }
};

#endif //SLAVE_H
```

B.7 slave.cpp

```
#include "slave.h"

void Slave::doit() {

}

void
Slave::print(int d) {
   memory->print (d);
}
```

B.8 master.h

```
#ifndef MASTER_H
#define MASTER_H

#include "systemc.h"

#include "specs.h"

SC_MODULE (Master) {

   //OCP PORTS

   //Master
   sc_out <sc_int<ADDR> > MAddr, MData;
   sc_out <sc_uint<MCMDSIZE> > MCmd;
   sc_out <bool> MDataValid, MRespAccept;

   //Slave
   sc_in <sc_int<WORD> > SData;
```

```
   sc_in <sc_uint<SRESPSIZE> > SResp;
   sc_in <bool> Clk, SCmdAccept, SDataAccept;

   void doit();
   void print(int);

   SC_CTOR (Master) {

      SC_METHOD (doit);
      sensitive_pos << Clk;

   }

};

#endif //MASTER_H
```

B.9 master.cpp

```
#include "master.h"

void Master::doit() {

}

void Master::print (int d) {
   application->print (d);
}
```

B.10 OCP.ah

```
#ifndef OCP_AH
#define OCP_AH

#include <iostream.h>
#include "systemc.h"
#include "specs.h"
#include "application.h"
#include "memory.h"
#include "master.h"
#include "slave.h"
```

```
aspect OCP {

  public:

  pointcut master() = "Master";

  advice master() : sc_signal <sc_int<WORD> > sig_DataIn;
  advice master() : sc_signal <sc_int<WORD> > sig_DataOut;
  advice master() : sc_signal <sc_int<ADDR> > sig_addr;
  advice master() : sc_signal <bool> sig_rst;
  advice master() : sc_signal <int> sig_availableIn;
  advice master() : sc_signal <int> sig_writingDone;

  advice master() : Application *application;

  advice construction (master()) : after () {
    Master *mw = (Master *) (tjp->target());
    mw->application = new Application ("application");
    mw->application->DataIn (mw->sig_DataIn);
    mw->application->DataOut (mw->sig_DataOut);
    mw->application->addr (mw->sig_addr);
    mw->application->rst (mw->sig_rst);
    mw->application->availableIn (mw->sig_availableIn);
    mw->application->writingDone (mw->sig_writingDone);
    mw->application->clk (mw->Clk);
  }


  pointcut slave() = "Slave";

  advice slave() : sc_signal <sc_int<WORD> > sig_data_in;
  advice slave() : sc_signal <sc_int<WORD> > sig_data_out;
  advice slave() : sc_signal <sc_int<ADDR> > sig_addr_in;
  advice slave() : sc_signal <bool> sig_rst;
  advice slave() : sc_signal <int> sig_available_out;

  advice slave() : Memory *memory;

  advice construction (slave()) : after () {
    Slave *sw = (Slave *) (tjp->target());
    sw->memory = new Memory ("memory");
    sw->memory->data_in (sw->sig_data_in);
```

```
  sw->memory->data_out (sw->sig_data_out);
  sw->memory->addr_in (sw->sig_addr_in);
  sw->memory->rst (sw->sig_rst);
  sw->memory->available_out (sw->sig_available_out);
  sw->memory->clk (sw->Clk);
}


public:

advice execution ("% Master::doit(...)") : after () {
  Master *master = (Master *) (tjp->target());
  static bool waitingSCmdAccept = true;

  if (waitingSCmdAccept) {

    if (master->application->getState () == READING) {
      master->MCmd.write (RD);
      master->MAddr.write (master->application->addr.read ());
      master->sig_availableIn.write (0);
    }
    else if (master->application->getState () == WRITING) {
      master->MCmd.write (WR);
      master->MData.write (master->application->DataOut.read ());
      master->MAddr.write (master->application->addr.read ());
      master->sig_writingDone.write (0);
    }
    else {
      master->MCmd.write (IDLE);
      master->MDataValid.write (0);
    }

    if (master->SCmdAccept.read()) {
      if (master->MCmd.read () == WR) {
        master->MDataValid.write (1);
      }
      master->MCmd.write (IDLE);
      master->MRespAccept.write (0);
      waitingSCmdAccept = false;
    }
  }
  else {
```

```
    if (master->SResp.read() == DVA) {
      if (master->application->getState () == READING) {
        master->sig_DataIn.write (master->SData.read ());
        master->sig_availableIn.write (OK);
      }
      else if (master->application->getState () == WRITING) {
        master->sig_writingDone.write (OK);
      }
      master->MRespAccept.write (1);
      waitingSCmdAccept = true;
    }
    else if (master->SResp.read() == FAIL || master->SResp.read() == ERR) {
      if (master->application->getState () == READING) {
        master->sig_availableIn.write (ERROR);
      } else if (master->application->getState () == WRITING) {
        master->sig_writingDone.write (ERROR);
      }
      master->MRespAccept.write (1);
      waitingSCmdAccept = true;
    }
  }
}

advice execution ("% Slave::doit(...)") : before () {
  Slave *slave = (Slave *) (tjp->target());

  if (slave->memory->getState () == FREE) {
    if (slave->MCmd.read() == IDLE) {
      slave->SCmdAccept.write (0);
      slave->SResp.write (0);
    }
    else if (slave->MCmd.read() == RD || slave->MCmd.read() == WR) {
      slave->SCmdAccept.write (1);
      slave->SResp.write (0);
      slave->SDataAccept.write (0);

      if (slave->MCmd.read() == RD) {
        slave->memory->setRead (true);
      } else if (slave->MCmd.read() == WR) {
        slave->memory->setWrite (true);
      }
    }
  }
```

```
    else if (slave->memory->getState () == READING || slave->memory->getState () == WRITING) {

      if (slave->MAddr.read() >= MEMORYSIZE) {
        slave->SResp.write (ERR);
        if (slave->memory->getState () == READING) {
          slave->memory->setRead (false);
        }
        else if (slave->memory->getState () == WRITING) {
          slave->memory->setWrite (false);
        }
      }
      else {
        slave->SResp.write (DVA);
        if (slave->memory->getState () == READING) {
          slave->SData.write (slave->memory->memory[slave->MAddr.read()]);
          slave->memory->setRead (false);
        } else if (slave->memory->getState () == WRITING && slave->MDataValid.read ()) {
          slave->memory->memory[slave->MAddr.read()] = slave->MData.read();
          slave->SDataAccept.write (1);
          slave->memory->setWrite (false);
        }
      }

      slave->SCmdAccept.write (0);
    }

  }

};

#endif //OCP_AH
```

B.11 main.cpp

```
#include <iostream.h>

#include "master.h"
#include "slave.h"

extern "C" int sc_main(int argc, char **argv);

int
```

```
sc_main(int argc, char **argv) {
  sc_signal <sc_int<WORD> > sig_MData, sig_SData;
  sc_signal <sc_int<ADDR> > sig_MAddr;
  sc_signal <sc_uint<MCMDSIZE> > sig_MCmd;
  sc_signal <sc_uint<SRESPSIZE> > sig_SResp;
  sc_signal <bool> sig_MDataValid, sig_MRespAccept, sig_SCmdAccept, sig_SDataAccept;
  sc_signal <bool> clock;

  //OCP signals


  sc_time t (20, SC_NS);

  //OCP connections
  Master master ("master");
  master.MAddr (sig_MAddr);
  master.MData (sig_MData);
  master.MCmd (sig_MCmd);
  master.MDataValid (sig_MDataValid);
  master.MRespAccept (sig_MRespAccept);

  master.SData (sig_SData);
  master.SResp (sig_SResp);
  master.SCmdAccept (sig_SCmdAccept);
  master.SDataAccept (sig_SDataAccept);

  master.Clk (clock);


  //OCP connections
  Slave slave ("slave");
  slave.MAddr (sig_MAddr);
  slave.MData (sig_MData);
  slave.MCmd (sig_MCmd);
  slave.MDataValid (sig_MDataValid);
  slave.MRespAccept (sig_MRespAccept);

  slave.SData (sig_SData);
  slave.SResp (sig_SResp);
  slave.SCmdAccept   (sig_SCmdAccept);
  slave.SDataAccept (sig_SDataAccept);

  slave.Clk (clock);
```

```
  sc_start ();

  int i = 0, ciclos;

  cin >> ciclos;

  while (i < ciclos) {

    clock.write(1);
    sc_cycle(t);

    clock.write(0);
    sc_cycle(t);

    ++i;
  }

  slave.print(1);

  master.print(1);

  return 0;
}
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

[Baixar livros de Literatura]
[Baixar livros de Literatura de Cordel]
[Baixar livros de Literatura Infantil]
[Baixar livros de Matemática]
[Baixar livros de Medicina]
[Baixar livros de Medicina Veterinária]
[Baixar livros de Meio Ambiente]
[Baixar livros de Meteorologia]
[Baixar Monografias e TCC]
[Baixar livros Multidisciplinar]
[Baixar livros de Música]
[Baixar livros de Psicologia]
[Baixar livros de Química]
[Baixar livros de Saúde Coletiva]
[Baixar livros de Serviço Social]
[Baixar livros de Sociologia]
[Baixar livros de Teologia]
[Baixar livros de Trabalho]
[Baixar livros de Turismo]