Flexibilidade e Separação de Interesses para Concepção e Evolução de Sistemas Distribuídos

A Abordagem R-RIO

Alexandre Sztajnberg

Exame de tema de tese submetido ao corpo docente da Coordenação dos Programas de Pós-Graduação em Engenharia da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências em Engenharia Elétrica.

RIO DE JANEIRO, RJ - BRASIL

JULHO DE 1999

Flexibilidade e Separação de Interesses para Concepção e Evolução de Sistemas Distribuídos

A Abordagem R-RIO

Alexandre Sztajnberg

Exame de tema de tese submetido ao corpo docente da Coordenação dos Programas de Pós-Graduação em Engenharia da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências em Engenharia Elétrica.

Aprovado por:

_____

Prof. Jorge Lopes de Souza Leão, Dr. Ing.
(GTA/COPPE/UFRJ)

_____

Prof. Orlando Gomes Loques Filho, PhD.
(CAA/IC/UFF)

_____

Prof. Otto Carlos Muniz Bandeira Duarte, Dr. Ing.
(GTA/COPPE/UFRJ)

_____

Prof. Julius Cesar Barreto Leite, PhD.
(CAA/IC/UFF)

RIO DE JANEIRO, RJ - BRASIL

JULHO DE 1999

Sumário

IIISUMÁRIO IV

LISTA DE FIGURAS VI
LISTA DE TABELAS VIII

INTRODUÇÃO 1

MOTIVAÇÃO 1
PROPOSTA 2
OBJETIVO 3
VALIDAÇÃO 3
CONTRIBUIÇÕES 3
ESTADO ATUAL 4
ESTRUTURA DO TEXTO 5
CONVENÇÕES TIPOGRÁFICAS UTILIZADAS 6

CAPÍTULO I - ANÁLISE DE REQUISITOS E TECNOLOGIAS 8

1.1 INTRODUÇÃO 8
1.2 REQUISITOS BÁSICOS 10
1.3 TECNOLOGIAS SELECIONADAS 12
1.3.1 Objetos 12
1.3.2 Reflexão 18
1.3.3 Configuração 25
1.4 INTERRELACIONADO OS CONCEITOS 31
1.4.1 Combinando as Tecnologias 31
1.4.2 Comparando as Tecnologias 33
1.5 CONSTRUINDO O CONJUNTO 37
1.6 TRABALHOS CORRELATOS 38
1.7 CONCLUSÕES 39

CAPÍTULO II - TRABALHOS CORRELATOS 42

2.1 INTRODUÇÃO 42
2.2 REFLEXÃO COMPUTACIONAL 43
2.2.1 Open C++ (reflexão computacional estática) 43
2.2.2 MetaJava (reflexão computacional dinâmica) 47
2.3 FILTROS DE COMPOSIÇÃO (*COMPOSITION-FILTER*) 50
2.4 AOP - *ASPECT ORIENTED PROGRAMMING* 54
2.5 CONFIGURAÇÃO 58
2.5.1 Regis 60
2.5.2 UniCon 63
2.6 CORBA 66
2.7 CONCLUSÃO 69

CAPÍTULO III - CONFIGURAÇÃO ORIENTADA A SEPARAÇÃO DE INTERESSES 71

3.1 INTRODUÇÃO 71
3.2 ELEMENTOS BÁSICOS 72
3.2.1 Módulos 73
3.2.2 Portas 75
3.2.3 Conectores 78
3.2.4 Composição 83
3.3 SEPARAÇÃO DE INTERESSES 84
3.3.1 Configuração de Aspectos em Conectores 86
3.3.2 Aspectos não-funcionais ou operacionais 88
3.3.3 Aspectos de Distribuição 89

3.3.4 Aspectos de Comunicação *90*
3.3.5 Aspectos de Coordenação *93*
3.3.6 Outros Aspectos *102*
3.3.7 Composição de Conectores *105*
3.3.8 Conectores Genéricos *109*
Observação *109*
3.4 AVALIAÇÃO 109
em relação a estrutura das aplicações desenvolvidas *110*
em relação a abrangência *112*
possibilidade de evolução da abordagem *113*
comparação das propostas *115*
3.5 CONCLUSÃO 117

CAPÍTULO IV - A LINGUAGEM DE CONFIGURAÇÃO 119

4.1 INTRODUÇÃO 119
4.2 A LINGUAGEM DE CONFIGURAÇÃO 121
4.2.1 Referências *122*
4.2.2 Portas *123*
4.2.3 Módulos *124*
4.2.4 Conectores *124*
4.2.5 Comandos de Configuração *127*
4.2.6 Sintaxe para Aspectos Não-Funcionais *129*
4.2.7 Sintaxe para os Aspectos de Distribuição *129*
4.2.8 Sintaxe para os Aspectos de Coordenação *130*
4.2.9 Composição de Módulos *134*
4.2.10 Composição de Conectores *135*
4.2.11 Filtros *136*
4.2.12 Herança *137*
Observação *138*
4.3 UM EXEMPLO 138
4.3.1 Primeira Versão *138*
4.3.2 Acrescentando Aspectos Não-Funcionais *140*
4.3.3 Acrescentando Aspectos por Composição *145*
4.3.4 Observação *148*
4.4 AVALIAÇÃO 149
4.4.1 Pontos ainda não avaliados na proposta *152*
4.4.2 Comparação com AOP *153*
4.4.3 Reflexão *154*
4.4.4 Considerações sobre configuração, reflexão e AOP *155*
4.4.5 Otimização *157*
4.5 CONCLUSÃO 159

CAPÍTULO V - CONCLUSÃO 161

5.1 PRÓXIMOS PASSOS 161
5.1.1 Sistema de Tipos para a Linguagem de Configuração *162*
5.1.2 Adicionando Novos Conectores *164*
5.1.3 Alternativas para Implementação do Modelo *167*
5.1.4 Validação *168*
5.2 RESULTADOS PRELIMINARES 168
5.3 PRINCIPAIS CONTRIBUIÇÕES 170
5.4 CONCLUSÕES 171

REFERÊNCIAS BIBLIOGRÁFICAS 173

Lista de Figuras

FIGURA 1.1 - REFLEXÃO COMO INTERPOSIÇÃO DE CÓDIGO....20
FIGURA 1.2 - INTERFACES NECESSÁRIAS EM UM ESQUEMA REFLEXIVO....20
FIGURA 1.3 - OBJETO ORIGINAL E O OBJETO REFLETIDO....22
FIGURA 1.4 - UTILIZANDO REFLEXÃO PARA ADICIONAR CARACTERÍSTICAS NÃO-FUNCIONAIS....24
FIGURA 1.5 - SUPORTE PARA A CONFIGURAÇÃO DE APLICAÇÕES DISTRIBUÍDAS....26
FIGURA 1.6 - CONEXÃO DE DOIS MÓDULOS....27
FIGURA 1.7 - AMBIENTE MULTIPROTOCOLO CONFIGURÁVEL BASEADO EM REFLEXÃO....32
FIGURA 2.1 - BUFFER LIMITADO POR REFLEXÃO....43
FIGURA 2.2 - META-OBJETO BOUNDED EM OPEN C++....45
FIGURA 2.3 - BUFFER LIMITADO EM OPEN C++, UTILIZANDO EXTENSÃO SINTÁTICA....47
FIGURA 2.4 - TRECHO DO CÓDIGO C++ GERADO A PARTIR DO BUFFER DESCRITO EM OPEN C++ E DO PRÉ-PROCESSADOR GERADO COM O META-OBJETO BOUNDED....47
FIGURA 2.5 - A CLASSE BASE BUFFER....48
FIGURA 2.6 - O META-OBJETO BUFFER LIMITADO E SINCRONIZADO....49
FIGURA 2.7 - APLICAÇÃO BUFFER LIMITADO....50
FIGURA 2.8 - UM OBJETO EM CF....52
FIGURA 2.9 - BUFFER LIMITADO EM SINA....53
FIGURA 2.10 - AOP E O ENTRELAÇADOR (*WEAVER*)....55
FIGURA 2.11 - BUF.JCORE: BUFFER EM JCORE....56
FIGURA 2.12 - BUF.COOL: ASPECTOS DE COORDENAÇÃO E SINCRONIZAÇÃO EM COOL....57
FIGURA 2.13 - BUF.JAVA: CÓDIGO JAVA GERADO POR ASPECTJ....58
FIGURA 2.14 - BUFFER LIMITADO DESCRITO EM DARWIN....61
FIGURA 2.15 - ARQUIVO EM C++ GERADO A PARTIR DA APLICAÇÃO DESCRITA EM DARWIN....63
FIGURA 2.16 - BUFFER LIMITADO DESCRITO EM UNICON (UNI-BUF.UNI)....65
FIGURA 2.17 - SAÍDA DO VERIFICADOR DA SINTAXE E COERÊNCIA DA ARQUITETURA DESCRITA EM UNICON 66
FIGURA 2.18 - TRECHO DA LISTAGEM DE PROPRIEDADES DA ARQUITETURA DESCRITA EM UNICON GERADO PELO UPARSE....66
FIGURA 2.19 - DESCRIÇÃO DAS INTERFACES DO BUFFER EM IDL....68
FIGURA 3.1 - MÓDULO, PORTA E MÉTODOS....74
FIGURA 3.2 - MAPEAMENTO DE UM MÓDULO....75
FIGURA 3.3 - CONFIGURAÇÃO INTERLIGANDO MÓDULOS ATRAVÉS DE CONECTORES....80
FIGURA 3.4 - UMA APLICAÇÃO COM MÓDULO COMPOSTO....84
FIGURA 3.5 - CONFIGURAÇÃO DE UMA APLICAÇÃO SIMPLES....87
FIGURA 3.6 - PARTES DO ASPECTO DE COMUNICAÇÃO....91
FIGURA 3.7 - COMUNICAÇÃO DE GRUPO....92
FIGURA 3.8 - MÓDULO COM MÉTODOS SINCRONIZADOS....98
FIGURA 3.9 - UM MÓDULO COM GUARDAS EM SEUS MÉTODOS....99
FIGURA 3.10 - CONSULTA AO ESTADO DE UM MÓDULO....100
FIGURA 3.11 - APLICAÇÃO COM MÓDULO-SEMÁFORO....101
FIGURA 3.12 - FILTROS EM PORTAS DE ENTRADA E SAÍDA EM UM CONECTOR....105
FIGURA 3.13 - UM CONECTOR INTERLIGANDO VÁRIOS SERVIÇOS DO MÓDULO Y ÀS REQUISIÇÕES DO MÓDULO X....106
FIGURA 3.14 - CONECTORES DIFERENTES LIGANDO DOIS MÓDULOS....106
FIGURA 3.15 - UM CONECTOR COMPOSTO....107
FIGURA 4.1 - CONCEPÇÃO DE UMA APLICAÇÃO POR CONFIGURAÇÃO....120
FIGURA 4.2 - UM CONECTOR COM DUAS PORTAS DE ENTRADA E SAÍDA....133
FIGURA 4.3 - UM CONECTOR COM UMA PORTA DE SAÍDA COM GUARDA....133
FIGURA 4.4 - APLICAÇÃO PRODUTOR/CONSUMIDOR....140
FIGURA 4.5 - BUFFER COM MÉTODOS SINCRONIZADOS....142
FIGURA 4.6 - BUFFER LIMITADO E SINCRONIZADO....143
FIGURA 4.7 - APLICAÇÃO COM CONECTOR IMPONDO PRIORIDADE....145
FIGURA 4.8 - APLICAÇÃO COM MÓDULO COMPOSTO....148
FIGURA 4.9 – (A) FLUXO NORMAL DE REQUISIÇÕES E RESPOSTAS EM UMA CADEIA DE CONECTORES; (B) POSSIBILIDADE DE OTIMIZAÇÃO DO FLUXO DE RESPOSTA....158
FIGURA 4.10 - CONECTOR COMPOSTO DESCREVENDO UMA CADEIA DE CONECTORES COM FLUXO OTIMIZADO....159

Lista de Tabelas

TABELA 1.1: COMPARAÇÃO DAS TECNOLOGIAS 37
TABELA 1.2: ONDE APLICAR CADA TECNOLOGIA 38
TABELA 3.1: COMPARAÇÃO DAS PROPOSTAS AVALIADAS 117
TABELA 4.1 - COMANDOS BÁSICOS DE CONFIGURAÇÃO 129
TABELA 4.2 - COMPARAÇÃO DAS PROPOSTAS AVALIADAS 150

Lista de Listagens

LISTAGEM 3.1 78
LISTAGEM 3.2 82
LISTAGEM 3.3 94
LISTAGEM 4.1 123
LISTAGEM 4.2 123
LISTAGEM 4.3 124
LISTAGEM 4.4 125
LISTAGEM 4.5 130
LISTAGEM 4.6 130
LISTAGEM 4.7 131
LISTAGEM 4.8 131
LISTAGEM 4.9 132
LISTAGEM 4.10 132
LISTAGEM 4.11 132
LISTAGEM 4.12 133
LISTAGEM 4.13 133
LISTAGEM 4.14 134
LISTAGEM 4.15 135
LISTAGEM 4.16 135
LISTAGEM 4.17 136
LISTAGEM 4.18 137

RESUMO DO EXAME DE TEMA DE TESE APRESENTADO À COPPE/UFRJ COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS.

Flexibilidade e Separação de Interesses para Concepção e Evolução de Sistemas Distribuídos

A Abordagem R-RIO

Alexandre Sztajnberg

Julho de 1999

ORIENTADORES: Orlando Gomes Loques

Jorge Lopes de Souza Leão

PROGRAMA: Engenharia Elétrica

RESUMO

Aplicações modernas possuem natureza dinâmica e necessitam de suporte para se adaptarem a novas demandas funcionais e não-funcionais. Novas demandas funcionais são criadas por usuários das aplicações, que precisam de novas funções ou necessitam alterar funções existentes. Além disso, durante o ciclo de vida das aplicações, podem surgir novos requisitos não-funcionais que, em geral, não estão associados diretamente com a aplicação. Modelos mais usuais para o desenvolvimento de *software* não oferecem as facilidades necessárias para a produção de aplicações que se adaptem dinamicamente a essas demandas. Portanto, uma abordagem que facilite a concepção dessas aplicações se faz necessária. Apresentamos neste artigo uma nova abordagem, R-RIO, que estende o conceito tradicional de configuração, permitindo que se descreva a arquitetura das aplicações a partir de componentes e suas interações e, adicionalmente, seus aspectos não-funcionais como a distribuição, coordenação e

protocolos de comunicação. A arquitetura de uma aplicação é descrita em alto nível de abstração, com o uso de CBabel, uma linguagem de configuração. A partir da descrição da arquitetura da aplicação, um *middleware* chamado configurador, pode criar instâncias de seus componentes e iniciar a execução da aplicação. O configurador também oferece suporte para adaptação das aplicações que necessitam reagir à mudanças de requisitos ou de condições de operação, inclusive durante a fase de funcionamento.

A metodologia, parte da proposta de R-RIO, utiliza conectores para intermediar as interações entre módulos e também encapsular aspectos não-funcionais das aplicações. Também é proposto um mapeamento do modelo conceitual para o nível de implementação. Este mapeamento poderá ser feito através de ferramentas automáticas e pode ainda contar, eventualmente, com ambientes especializados.

Embora R-RIO tenha sido proposto inicialmente para aplicações distribuídas a abordagem é genérica o suficiente para ser utilizada por outras classes de aplicações. Aspectos que não sejam ligados à distribuição e comunicação também podem ser configurados nas aplicações, desde que, de alguma forma, digam à respeito a interação entre módulos, como mostrado no exemplo apresentado.

A metodologia de R-RIO para a concepção de aplicações foi comparada com protótipos de outras propostas, além de AOP, como *Composition Filters* [Ber 96T], OpenC++ [Chi 96], MetaXa [Gol 97], UniCon [Sha 95] e Conic/Regis/Darwin [Mag 92], através do desenvolvimento de um mesmo conjunto de exemplos. Verificou-se que R-RIO possui expressividade equivalente, e pode ser mais flexível do que as outras abordagens. Adicionalmente, a abordagem utilizada em nosso modelo permite um tratamento integrado do gerenciamento das aplicações desenvolvidas, desde a concepção até a execução e manutenção, diferentemente de outras propostas avaliadas, onde isso é feito de maneira empírica. Nossa proposta de configuração, por exemplo, permite que o mesmo modelo utilizado para conceber a arquitetura de uma aplicação seja utilizado na sua reestruturação, em resposta a novas demandas, mesmo depois desta ser implantada e estar em produção. Esta característica torna R-RIO uma base para o desenvolvimento de aplicações que possam evoluir dinamicamente.

Introdução

Este trabalho descreve uma proposta de tese a ser apresentada como parte dos requisitos ao título de doutorado na COPPE. O conteúdo deste trabalho tem origem nas disciplinas cursadas durante a fase de integralização dos créditos do programa, pesquisas bibliográficas, pesquisas acadêmicas desenvolvidas e trabalhos publicados pelo autor com alguns resultados preliminares [Szt 97, Szt 98A, Lob 98, Loq 99, Szt 99].

Motivação

A evolução tecnológica na área de computação vem criando demanda e facilitando o surgimento de aplicações distribuídas que apresentam grande complexidade e dinamismo. Estas aplicações podem ser construídas para manipular e para coordenar o processamento de informações com origem e natureza diferentes, tais como textos, imagens, áudio, vídeo, sinais de sensores, dados digitados, etc. Tanto a criação ou a aquisição destas informações quanto a sua coordenação podem se dar simultaneamente em pontos dispersos por grandes redes como a Internet [Weg 92, Pou 94 e Lad 97]. Além disso, a tendência destas aplicações é de que sejam heterogêneas. Várias partes destas aplicações poderão ser construídas por diversos projetistas e executadas em plataformas computacionais diferentes. Entretanto, o que se observa é que a tecnologia de hardware evoluiu rapidamente (máquinas com maior poder computacional e redes de alta velocidade), mas não foram criados sistemas de suporte totalmente adequados para a concepção das aplicações distribuídas que vão executar sobre esta tecnologia.

Uma das formas de se acomodar e evolução tecnológica vislumbrada é manter os sistemas e suas partes componentes abertas à extensões ou à interação com outros componentes e sistemas [Weg 96 e Kic 97]. Embora o conceito de sistemas abertos

possa ser utilizado como filosofia, não existem modelos e ferramentas para suportar a concepção e gestão de aplicações distribuídas de grande complexidade, tamanho e diversidade, apenas segundo este conceito. Tais aplicações requerem um modelo que ofereça suporte adequado durante as fases de concepção, desenvolvimento, manutenção e gerenciamento do seu ciclo de vida. O termo *suporte adequado* é bastante amplo e pode ser uma referência a ferramentas para o planejamento da arquitetura das aplicações, concepção e programação de módulos, ferramentas para verificação e testes, um ambiente que permita o controle e a gerência das aplicações, etc. Um outro ponto importante é a questão do dinamismo e evolução destas aplicações. Nem todos os requisitos destas aplicações ou sistemas são conhecidos antes ou durante a fase de desenvolvimento. Não serão pouco freqüentes os casos em que novas funções ou requisitos terão que ser atendidos por uma aplicação distribuída e que estas novas funções sejam desenvolvidas por entidades autônomas ou utilizadas remotamente. Além disto, as aplicações distribuídas deverão estar aptas a se adaptar a diferentes ambientes de execução, como por exemplo, redes com velocidades diferentes, recursos distribuídos com disponibilidades diferentes, etc.

A motivação para o presente trabalho é a busca do entendimento das dimensões de alguns destes problemas levantados, com foco em modelos para a concepção e evolução de grandes aplicações distribuídas e a proposta de uma metodologia para a concepção destas aplicações segundo o modelo mais adequado.

Proposta

Considerando-se as etapas para o projeto e a gerência de aplicações distribuídas, observa-se que não existe uma tecnologia ou ferramenta única que ofereça todo suporte necessário, com um nível de abstração desejado, para todas as etapas. Em busca de uma resposta a este problema apresentamos a proposição de um modelo baseado em configuração, orientado à separação de interesses [Aks 96, Hür 95, Kic 96] e uma metodologia para o desenvolvimento e manutenção de grandes aplicações distribuídas. A idéia central desta abordagem é oferecer uma alternativa para a concepção de aplicações distribuídas em que a arquitetura de uma aplicação é descrita através de seus componentes, interações entre estes componentes, de seus aspectos funcionais e não-funcionais em alto nível de abstração.

Nesta abordagem, aspectos funcionais tipicamente são encapsulados nos componentes da aplicação e aspectos não-funcionais são configurados em elementos de interligação que expressam a interação entre os componentes. O relacionamento entre componentes e elementos de interconexão é mantido com acoplamento fraco, permitindo que estes elementos sejam intercambiados e reutilizados quando necessário. A partir deste modelo podem ser concebidas ferramentas para a construção, verificação e gerência de sistemas distribuídos. Este modelo será caracterizado e avaliado na tese. Será também verificado se ele engloba características desejáveis em sistemas distribuídos como reusabilidade, separação de interesses, capacidade de evolução dinâmica e abrangência.

Objetivo

O objetivo deste trabalho é apresentar uma primeira abordagem de um modelo para a concepção de sistemas distribuídos baseados na tecnologia de configuração e orientado à separação de aspectos. A apresentação do modelo constará da descrição de uma linguagem de configuração e uma metodologia em que aspectos não-funcionais podem ser incluídos na arquitetura de uma aplicação sob construção, de forma separada das funções básicas desta aplicação. Ainda como parte da apresentação, a proposta será comparada com pesquisas similares permitindo uma avaliação preliminar da mesma e demostrando sua relevância. A utilidade do modelo será também comprovada através de exemplos. Além da apresentação do modelo, é nosso objetivo apontar os próximos passos a serem dados para a conclusão das pesquisas.

Validação

A tese será validada através de exemplos desenvolvidos a partir do modelo proposto. Nesta primeira etapa um exemplo simples será utilizado para demonstrar a abordagem de configuração e como esta abordagem deve ser utilizada para facilitar o usuário na obtenção de melhores resultados.

Contribuições

Apresentar um modelo para a construção, evolução e manutenção de sistemas e

aplicações distribuídas. Este modelo apresentará soluções para as fases de projeto da arquitetura das aplicações através da descrição de configurações, um esquema de mapeamento para a geração de código para a programação de componentes a partir da descrição da estrutura da aplicação e uma arquitetura de ambiente de execução que permite o gerenciamento da aplicação e facilita a evolução dinâmica da mesma. O modelo proposto é orientado a separação de aspectos, o que permite que o planejamento das funções da aplicação e das características não-funcionais seja feito separadamente, aumentando as perspectivas de reusabilidade.

Estado Atual

Um estudo aprofundado dos conceitos e tecnologias relacionados ao trabalho foi realizado. Dentre os diversos assuntos pesquisados estão: tecnologia de objetos, reflexão computacional, arquiteturas de software, linguagens de especificação de arquiteturas de software, separação de interesses (*separation of concerns*) como conceito básico de engenharia de software, técnicas de coordenação de processos e questões sobre aspectos formais ligados ao tema.

Uma avaliação de tecnologias candidatas para projeto de sistemas distribuídos está concluída, o que nos permitiu selecionar um conjunto adequado dentre as mesmas e propor algumas diretivas de como empregá-las na concepção de sistemas distribuídos. Durante esta avaliação também foram investigadas propostas correlatas à nossa, centradas em reflexão computacional [Chi 95, Kic 97, Zim 96], programação orientada a aspectos [Lop 97T] e linguagens orientadas a composição de objetos [Ber 96T] e configuração [Kra 90, Bis 96 e Sha 95, entre outros]. Para cada proposta avaliada foram realizadas implementações de exemplos simples para permitir comparações entre cada uma delas e a nossa proposta.

A partir desses estudos fundamentais, uma proposta de um modelo e uma nova linguagem de configuração que integra os paradigmas de configuração e separação de aspectos foi elaborada. A seguir a proposta foi utilizada na programação de diversas versões de uma aplicação distribuída, onde cada versão pôde ser construída a partir da versão anterior por substituição de componentes ou composição. Essa evolução foi obtida a partir de mudanças efetuadas no nível de configuração, afetando minimamente

os módulos funcionais da aplicação, o que permitiu validar preliminarmente a proposta, quanto ao atendimento do requisito de separação de aspectos e evolução dinâmica.

No prosseguimento das pesquisas, estamos ampliando continuamente o levantamento bibliográfico sobre arquiteturas de software, linguagens para descrição de arquiteturas, classificação de aspectos não-funcionais e como estes podem ser introduzidos em nossa abordagem. As abordagens contemporâneas sobre aspectos formais em arquiteturas de software também estão sendo levantadas. Este levantamento visa permitir, em fase posterior a esta proposta, delinear a base formal do modelo proposto.

Estrutura do Texto

O trabalho está estruturado em 5 capítulos. No primeiro capítulo inicia-se uma discussão sobre parâmetros e tecnologias de relevância na concepção de sistemas distribuídos. Um conjunto de requisitos importantes nesta categoria de sistema é levantado e atributos desejáveis para as ferramentas de suporte à concepção de grandes sistemas distribuídos são determinados. Analisa-se ainda neste capítulo algumas tecnologias como objetos, reflexão e configuração, importantes no contexto deste trabalho.

No capítulo seguinte, capítulo 2, apresentam-se alguns trabalhos correlatos e uma análise comparativa destes trabalhos é feita utilizando-se como base o exemplo clássico de um produtor/consumidor com *buffer* limitado.

O terceiro capítulo é destinado à apresentação da proposta do modelo, introduzindo-se os conceitos adotados para módulos, portas e conectores, e apresentando-se uma nova linguagem de configuração, orientada a separação de aspectos. No final deste capítulo uma comparação entre as outras propostas avaliadas e o nosso modelo é desenvolvida. Isto permite o confronto entre as abordagens e destaque dos pontos importantes de nosso modelo.

No capítulo 4 são apresentadas a linguagem de configuração e a sintaxe de aspectos de coordenação, que podem ser programados separadamente das funções da aplicação. Uma seqüência de exemplos, utilizando também a aplicação do Buffer Limitado, explora as características da linguagem como a separação de aspectos, reusabilidade, evolução dinâmica e abrangência. O capítulo é concluído com uma avaliação geral da

linguagem.

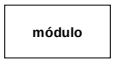
O quinto capítulo agrega tópicos que estão sendo desenvolvidos em nossas pesquisas e outros que ainda merecem mais atenção. Na primeira parte deste capítulo serão apresentadas breves discussões sobre a arquitetura de um ambiente de configuração que possa suportar os conceitos introduzidos no capítulo 3, alternativas para a implementação deste ambiente e possíveis questões sobre modelos formais também são colocadas. Na segunda parte deste capítulo, propostas para a validação do modelo são apresentadas e a direção a ser tomada na continuação das pesquisas é delineada. Finalmente são feitas as últimas considerações sobre a proposta apresentada e algumas conclusões.

Convenções Tipográficas Utilizadas

Os exemplos contendo programas, trechos de programas, textos que representem comandos ou respostas enviadas por um programa, ou máquina, observam as seguintes convenções descritas na tabela abaixo.

Tipo	**Convenção**
`courier`	programas serão escritos com este tipo de letra.
`negrito`	palavras em negrito fazem parte da linguagem ou são símbolos reservados da linguagem que está sendo usada no exemplo.
`itálico`	palavras em itálico devem ser entendidas como nomes de variáveis cujo conteúdo e forma varia de acordo com a sintaxe da linguagem que está sendo usada como exemplo.
<obrigatório>	o nome ou valor entre < e > é de uso obrigatório e o conteúdo varia de acordo com o contexto e a sintaxe da linguagem usada como exemplo.
[opcional]	o nome ou valor entre < e > é de uso opcional e o conteúdo varia de acordo com o contexto e a sintaxe da linguagem usada como exemplo.
/* *comentário* */ ou // *comentário*	frases em itálico iniciadas pelo símbolo // ou entre os símbolos /* e */, devem ser tratadas como comentários inseridos no exemplo.

Para os esquemas e diagramas que representam de uma forma geral um programa, arquitetura ou estrutura de software, a seguinte convenção será usada:

Símbolo	Convenção
	módulos, classes, objetos ou procedimentos, dependendo do contexto do exemplo
	portas, interfaces ou referência para pontos de interação de um módulo. Dependendo do contexto estará associado a um método ou procedimento indicando seu nome ou interface.
	conector. Elemento de interligação entre módulos.
	meta-objeto, meta-classe ou módulo de controle. Elemento que tem função de controle, mas não participa da computação elementar da aplicação.

Alguns exemplos ou diagramas podem não obedecer estas convenções ou podem seguir convenções específicas, como por exemplo diagramas de objetos segundo o modelo OMT [Rum 91] ou UML [Boo 97], que serão associadas ao exemplo no texto ou em legenda apropriada.

Capítulo I

Análise de Requisitos e Tecnologias

1.1 Introdução

Aplicações distribuídas possuem natureza dinâmica e necessitam de suporte para se adaptar a novas demandas funcionais e não-funcionais [Lad 97]. Aplicações tais como vídeo conferência, controle de processos e gerenciamento de *workflow*, possuem funcionalidade complexa e fortes requisitos não-funcionais [Bla 96, Tur 96]. Novas demandas funcionais são criadas por usuários das aplicações, que precisam de novas funções ou necessitam alterar funções existentes. Juntamente com novas demandas funcionais podem surgir novos requisitos não-funcionais que, de uma forma geral, não estão associados diretamente com a aplicação, como protocolos de comunicação, qualidade de serviço, disponibilidade ou tolerância a falhas, por exemplo. Estes requisitos também podem ser modificados ao longo da execução das aplicações por conta dos próprios usuários ou em reação à mudanças nas condições de operação.

Modelos tradicionais para a concepção de *software* não oferecem as facilidades necessárias para a produção de aplicações que se adaptem dinamicamente a novos requisitos. Na maioria dos casos as aplicações são estáticas e fruto de um grande esforço de programação, resultando em um código complicado, com vários aspectos entrelaçados (*code tangling* [Lop 97]). É comum, por exemplo, um mesmo bloco de código conter instruções para a comunicação via *sockets*, sincronização via semáforos, além das instruções que executam a computação intrínseca da aplicação, criando assim uma dificuldade para a reutilização de código. Para facilitar a concepção destas aplicações, uma nova abordagem se faz necessária. Esta abordagem deve considerar que todas aplicações precisam evoluir dinamicamente em seus aspectos funcionais e não-funcionais ou mesmo em sua estrutura. Identifica-se aqui um problema de manejo

da aplicação e do ambiente de execução, para atender adequadamente a estas possíveis mudanças com o dinamismo e consistência esperados.

A primeira parte de nossa investigação tem como objetivo estabelecer um conjunto de características que possam nortear o projeto de aplicações distribuídas em que devem ser contemplados os seguintes pontos:

- utilizar componentes desenvolvidos e operados por entidades independentes;
- reutilizar e reciclar componentes, código (e o *expertise* embutido) para construir novas aplicações;
- adicionar, transformar ou atualizar dinamicamente a funcionalidade das aplicações, afetando a mesma apenas parcialmente, sem a necessidade de paradas;
- adaptar as aplicações dinamicamente para manter a qualidade de serviço esperada pelo usuário e reagir em face a alterações do ambiente de execução (situação de falha, atualização de hardware ou software).

O primeiro passo nesta investigação é a seleção de um elenco de requisitos recorrentes na construção de aplicações distribuídas. O segundo passo é selecionar tecnologias candidatas que possam contribuir para o atendimento destes requisitos, simplificando o projeto das aplicações. Tecnologia de objetos, reflexão computacional, configuração e outras tecnologias prospectivas são analisadas em relação ao seu potencial em atender aos requisitos de projeto selecionados. A partir desta análise, será avaliado se a combinação destas tecnologias permite o atendimento dos requisitos mais facilmente. Com o resultado desta investigação poderemos propor um conjunto de diretivas para facilitar o projeto de aplicações distribuídas e associar estas diretivas a ferramentas e modelos disponíveis.

A primeira parte deste capítulo estabelece um conjunto mínimo de critérios para o projeto aplicações distribuídas abertas. Na segunda parte, seção 1.3, tecnologias de destaque são apresentadas e analisadas baseadas nestes critérios. A parte seguinte, seção 1.4, avalia as possibilidades de utilização destas tecnologias no desenvolvimento de aplicações distribuídas e compara estas tecnologias nesse contexto. Na seqüência, uma série de diretivas sobre como e quando aplicar os parâmetros estabelecidos é proposta. Nas ultimas seções, os conceitos são consolidados e algumas conclusões

apresentadas.

1.2 Requisitos Básicos

Um conjunto de requisitos para o projeto e implementação de aplicações distribuídas foi selecionado com base em outros trabalhos pesquisados [Agh 93, Gue 96, Höl 93, OOP'90, Voa 98 e Bro 96]. Nesta seção este conjunto é apresentado e avaliado de forma a se determinar sua relevância e um contexto para as discussões que se seguirão. Observa-se que os requisitos são de natureza diferente e podem estar superpostos ou não se relacionarem diretamente. Entretanto, acreditamos que os mesmos, em conjunto, oferecem uma base consistente para a engenharia de sistemas distribuídos.

Reusabilidade. Técnicas utilizadas para a concepção de aplicações devem permitir que partes de aplicações previamente concebidas possam ser reutilizadas de forma a minimizar esforços de desenvolvimento. Em um caso extremo, uma aplicação pode ser criada completamente a partir de partes já existentes [OOP'90, Rin 98]. Em um contexto restrito, reusabilidade pode ser observada dentro de uma linguagem de programação, onde tipos de dados e procedimentos já codificados estão disponíveis. No contexto deste trabalho, reusabilidade é considerada em relação a sistemas distribuídos, onde deseja-se construir aplicações utilizando-se componentes disponíveis e interconectando-os através de uma rede [Atk 91]. Desta forma, esforços poderão se concentrar na concepção da arquitetura da aplicação e não no desenvolvimento de partes básicas.

Dois conceitos estão associados a reusabilidade: modularidade e interoperabilidade. O projeto de componentes modulares está vinculado a dois aspectos: encapsulamento, em que o comportamento do módulo é auto-contido, e padronização de interface, em que a as regras de acesso à funcionalidade do módulo são claramente definidas. Interoperabilidade é essencial para o suporte de aplicações distribuídas compostas por módulos, oferecidos por provedores independentes, que precisam se comunicar e cooperar [Weg 96]. Estes módulos podem residir em diferentes nós, usar linguagens, protocolos e formato de dados particulares, levando à necessidade de cuidados especiais para que a interoperabilidade seja obtida. A concepção modular de um componente facilita o caminho para a reusabilidade. Estes dois conceitos formam a

base para a tecnologia de programação baseada em componentes [Bro 96].

Separação de Interesses. A separação de interesses (tradução utilizada para *separation of concerns*) é um paradigma que vem sendo considerado para a concepção de grandes sistemas [Agh 96, Aks 96, Ber 96, Lop 97 e Szt 98C]. A separação de interesses divide o projeto de uma aplicação em especificações de requisitos funcionais, não-funcionais e operacionais. Aspectos relacionados com as funções básicas de uma aplicação podem ser tratados independentemente de aspectos não-funcionais como a coordenação, comunicação e distribuição, ou de aspectos operacionais como protocolos de comunicação, QoS e gerência de recursos do ambiente de execução. Esta estratégia permite que os projetistas possam inicialmente concentrar seus esforços na concepção dos módulos funcionais e, em uma fase posterior, procurar soluções para a adaptação da aplicação às especificações que não estão ligadas diretamente às funções básicas da mesma. Se a modularidade e interoperabilidade forem consideradas durante este processo, componentes do domínio funcional e não-funcional podem ser potencialmente reutilizados.

Evolução Dinâmica. No escopo deste trabalho, evolução dinâmica é considerada um requisito que oferece o suporte para a modificação e inclusão de novos componentes em aplicações distribuídas que não podem ser completamente paradas para realizar estas operações. Por exemplo, a atualização de software (*up-grade*) de componentes, inclusão de tolerância a falhas, alteração parâmetros de QoS, alteração de estilos de comunicação ou restrições temporais, são atividades que podem implicar em mudanças em tempo de execução. No nível da aplicação, o projetista pode ser obrigado a antecipar estas possibilidades, embora em certos casos algum grau de transparência seja possível.

O ambiente de execução deve disponibilizar mecanismos que permitam a inclusão de novos módulos e sua ligação a outros componentes da aplicação. Entretanto, alterações dinâmicas podem introduzir problemas não triviais relacionados com segurança. Em certos casos é preciso se ter acesso à estrutura interna de algumas partes da aplicação ou do ambiente de execução. Por exemplo, para se alterar o comportamento de um módulo pode ser necessário consultar o estado interno do mesmo, garantindo que esta operação só será realizada em determinadas condições. Isto pode acarretar fortes implicações na concepção e implementação dos mecanismos de suporte necessários, tanto no ambiente

de execução como nos compiladores. Entretanto, em algumas situações este é o preço que se deve pagar.

Abrangência. O requisito de abrangência de uma tecnologia apresenta vários aspectos. Primeiramente deseja-se que com uma dada tecnologia um grande numero de problemas possa ser solucionado e que para isso ela tenha expressividade adequada. Adicionalmente são observadas a disponibilidade, disseminação e aceitação das tecnologias. Ferramentas facilmente disponíveis e disseminadas facilitam a criação de massa crítica de *experts* e em conseqüência são mais aceitas e tendem à evoluir. Se, além disso, estas ferramentas apresentarem facilidade de uso, a aceitação pode tornar-se ainda maior. Observa-se, no entanto, que, contrariamente às expectativas, nem sempre as ferramentas mais elegantes tecnologicamente são as mais disseminadas.

Com este conjunto de requisitos em mente, na próxima seção uma seleção das tecnologias mais promissoras serão revisadas e para cada uma delas será verificado como estes requisitos são atendidos.

1.3 Tecnologias Selecionadas

Durante o processo de concepção de aplicações distribuídas e mesmo durante a operação destas aplicações, os projetistas precisam contar com tecnologias e ferramentas apropriadas. É importante que estas ferramentas permitam ao projetista conceber as aplicações atendendo a requisitos dos usuários e requisitos mais gerais, como os estabelecidos na seção anterior, sem esforço desnecessário. Nesta seção reunimos e avaliamos um conjunto de tecnologias que oferecem mecanismos com os quais as aplicações distribuídas podem ser projetadas. Durante esta avaliação são identificados quais requisitos podem ser atendidos e como isso é possível.

1.3.1 Objetos

1.3.1.1 Conceitos Básicos

A tecnologia de objetos é baseada em dois conceitos: encapsulamento e polimorfismo [Weg 93, Rum 91 e Pou 94]. O conceito de encapsulamento é utilizado para assegurar que todas as informações acerca de um objeto estejam definidas dentro de sua própria

estrutura e que o acesso a esta estrutura será controlado. Um objeto é composto por variáveis de dados e procedimentos, conhecidos como *métodos,* que contém código para manipulá-los. Um objeto somente pode ser acessado através da invocação dos métodos, que definem uma interface para o objeto. Polimorfismo permite que em resposta a invocação de um método, o objeto que vai efetivamente executar este método seja chaveado de acordo com a assinatura de parâmetros passados no momento da invocação. Encapsulamento e polimorfismo são mecanismos úteis para o projeto modular de componentes e são de interesse em nosso trabalho.

Existem duas abordagens principais para implementação de linguagens de programação baseadas na tecnologia de objetos [Weg 92 e Tai 96]. Linguagens baseadas em **classes** preconizam que a implementação de um objeto é definida por sua classe. A classe é uma fôrma inativa contendo a descrição da interface e comportamento utilizado para criar objetos deste tipo. Um objeto é uma instância viva de uma classe, ocupando espaço em memória e possuindo um contexto associado. A outra abordagem são as linguagens baseadas em **objetos** ou **protótipos**. Nas linguagens baseadas em protótipos o conceito de classe não é tão forte. Cada elemento é tido como um objeto e objetos não são instâncias de classes, mas clones de outros objetos. Existem diferenças conceituais interessantes entre as duas abordagens [Tai 96], mas, neste trabalho, nos concentraremos nos efeitos destas diferenças com relação aos mecanismos de extensão associados à tecnologia de objetos. Para os nossos propósitos considera-se o uso linguagens baseadas em classe.

Linguagens orientadas a objeto normalmente oferecem mecanismos de composição e extensão. Nas implementações de linguagens orientadas a objetos podem ser geralmente encontrados mecanismos como a herança, composição e a delegação.

1.3.1.2 Herança

No escopo deste trabalho herança é considerada sob o ponto de vista de um mecanismo para extensão incremental da funcionalidade ou código de uma aplicação. Em linguagens baseadas em classe, com o mecanismo de herança uma nova classe pode ser definida utilizando-se como base a estrutura de uma outra classe chamada de pai (*parent*) ou superclasse. Além disso, nova funcionalidade pode ser adicionada à classe em construção ou partes originais da classe herdada podem ser modificadas

(*overriding*). A herança pode ser interpretada como um mecanismo de especialização de uma classe genérica. Objetos instanciados a partir de um hierarquia de classes podem ser utilizados para composição de aplicações. Existem algumas variações na abordagem para a utilização de herança entre as diversas linguagens de programação. Sumarizamos estas variações da seguinte maneira:

- herança de interface: reuso da definição original da estrutura de uma classe sem nenhuma relação estrita com o que esta classe faz;
- herança não estrita: estender o comportamento de uma classe criando-se uma classe especializada com novas funções e alterando-se partes (*overriding*) de uma classe pai;
- herança estrita ou especialização: estender o comportamento de uma classe criando-se uma classe especializada estritamente baseada na interface da classe pai e adicionando-se novas características, mas não alterando-se suas características originais;
- herança múltipla: criação de uma classe especializada baseada em duas ou mais superclasses, herdando-se a interface e o comportamento de todas;
- agregação: criação de uma classe genérica (superclasse) encapsulando-se duas ou mais classes e as indicações de relacionamentos entre estas classes;
- generalização: criação de uma classe genérica baseada na união de um conjunto de subclasses particionadas. Geralmente a superclasse é uma classe abstrata, onde nenhuma implementação é provida, somente a interface, e por isso não pode ser instanciada.

Em linguagens baseadas em protótipo o mecanismo de herança utiliza objetos instanciados na aplicação a partir dos quais clones são criados. Um clone herda a interface, comportamento e o estado do objeto matriz.

1.3.1.3 Composição e Concatenação

A distinção entre os termos composição e concatenação é difusa [Tai 96 e Gam 94]. Os dois termos são usados como referência a mecanismos de extensão com estilos similares. A composição possibilita a criação de novas classes com características mais

completas através do compartilhamento de código de classes mais simples em um estilo *caixa-preta*. Em uma composição a estrutura de duas ou mais classes são copiadas e combinadas para gerar uma outra classe. Diferentemente da herança ou herança múltipla, a composição pode ser implementada sem a necessidade de se quebrar o encapsulamento dos objetos: a interface e a implementação das classes componentes são concatenadas e novas características podem ser adicionadas, mas as partes originais não podem ser modificadas.

1.3.1.4 Delegação e Encaminhamento

Delegação é um mecanismo de extensão disponível em algumas linguagens orientadas a objeto, que permite que um objeto que tenha recebido uma invocação para um de seus métodos, envolva um outro objeto, um objeto delegado, no tratamento e execução deste método. O mecanismo de delegação, utiliza um esquema de auto-referência (*self-reference*) ao delegar o tratamento deste pedido. Assim sendo, a operação é realizada pelo objeto delegado, mas no contexto do objeto que recebeu o pedido originalmente, e sem a necessidade de se incorporar o código do objeto delegado.

Em [Pou 94] delegação e herança são considerados conceitos distintos onde um pode emular o outro. Em [Tai 96] a delegação é considerada como um mecanismo através do qual a herança pode ser implementada. Neste caso, o objeto delegado seria uma instância da classe que seria considerada como classe pai se a herança fosse utilizada. Uma colocação importante é que a herança e a delegação possuem a mesma expressividade.

O mecanismo de encaminhamento (*forwarding*), semelhante à delegação, é usado quando um objeto não sabe como ou não quer tratar um pedido e então repassa este pedido para um outro objeto (um pai ou outro objeto na cadeia). O objeto que se prontifica a tratar a mensagem encaminhada, o faz no seu próprio contexto.

A delegação pode ser obtida em linguagens de programação que não implementam delegação utilizando-se operações de encaminhamento e passando-se a referência do objeto encaminhador como parte do pedido a ser manipulado pelo objeto receptor [Tai 96].

1.3.1.5 Padrões de Projeto (*Design Patterns*)

O termo padrão de projeto é utilizado de forma particular em linguagens e projetos orientados a objetos. Um padrão de projeto é uma forma de se permitir a reutilização de experiências bem sucedidas de projetos de software. Se versões implementadas. depuradas e documentadas das arquiteturas e códigos de software mais utilizados estiverem disponíveis, uma parte do esforço do desenvolvimento de novas aplicações pode ser evitada. Não existe mais, neste caso, a necessidade de se redescobrir ou reinventar componentes de software comuns.

Em projetos que seguem uma metodologia orientada a objetos, padrões de projeto identificam, explicam e avaliam sistematicamente estruturas de projeto utilizadas com freqüência. Em [Gam 94], padrões de projeto são conceituados como descrições de objetos, classes e suas interações, concebidas para resolver um problema de concepção genérico, mas que são customizadas para um contexto particular. Portanto, padrões de projeto não são orientados para a solução de uma aplicação específica, mas para prover componentes de software mais primitivos e genéricos, que potencialmente podem ser partes da solução de concepção de um problema maior.

A reutilização de objetos é uma característica importante dos padrões de projeto. Assim sendo, os objetos devem ser concebidos segundo técnicas que permitem ao objeto ser inócuo, flexível e que evite o problema da fragilidade de classe-base [Pou 94]. Por esta razão, "a composição de objetos é preferida em relação ao uso da herança", levando a um estilo de concepção por caixa-preta, escondendo detalhes internos dos objetos. Seguindo esta linha de raciocínio, a técnica de delegação é a mais apropriada para este objetivo, tornando a composição tão poderosa para permitir a reutilização quanto a herança [Gam 94 e Gue 96].

1.3.1.6 *Frameworks*

Um *framework*, na terminologia de objetos, é um esqueleto de implementação de uma aplicação ou subsistema de aplicação de um domínio de problema particular [Joh 97 e Fay 97]. Ele é composto de classes concretas ou abstratas e é previsto um modelo de interação ou colaboração entre as instâncias das classes definidas. Um framework é utilizado configurando-se ou conectando-se classes concretas para, a partir das *lacunas* das classes abstratas, derivar-se então uma nova aplicação. Um framework é um projeto reusável porque todos os seus usuários compartilham a estrutura de suas classes básicas

e modelo de colaboração [Fay 94, Joh 97, OOP'90].

Para que um framework seja realmente usado como um modelo a partir do qual projetistas possam obter suas próprias aplicações, ele deve ser simples, consistente de funcional. Neste sentido é quase mandatório que se relacionem *frameworks* e *design patterns*. Como lembrado em [Gam 94] um framework pode ser visto como um conjunto de *design patterns* especializados e interrelacionados para implementar a estrutura básica de um determinado domínio de aplicação.

1.3.1.7 Anomalia de Herança

Quando questões de sincronização são consideradas, a herança, normalmente, leva à quebra de encapsulamento. Não existem muitas soluções para se permitir a herança de sincronização [Mat 93].

Como exemplo simples dos impactos da anomalia da herança em relação à sincronização, tomemos uma classe que herda dois métodos de uma superclasse e acrescenta mais um método. Se os métodos da superclasse são sincronizados por um semáforo, por exemplo, a nova classe precisa "saber" desta característica e ter acesso a este semáforo para poder sincronizar o método adicionado com os herdados. Assim sendo, em aplicações que exijam concorrência, sincronização e controle de exclusão mútua, a utilização de herança deve ser feita com cautela.

1.3.1.8 Fragilidade da Classe Base (*Fragile Base Class*)

O problema da fragilidade da classe base ocorre em linguagens orientadas a objeto quando o comportamento de uma classes concebida por herança é alterado em conseqüência de uma modificação na respectiva classe-base. Em [Mik 97], por exemplo, são analisadas várias situações que levam a este problema. Neste mesmo trabalho é sugerido um roteiro para se evitar este tipo de problema. Algumas receitas na programação de classes e utilização adequada de herança são apresentadas.

Em alguns ambientes caseados em objetos como o CORBA [OMG 96A] e DCOM [Bro 98, Wan 98 e Har 97], o mecanismo de herança está disponível de forma limitada. Esta limitação tem o objetivo, em parte, de se evitar o problema da fragilidade de classes e em parte de restringir este mecanismo para herança de interface. Em ambientes como estes, a extensão incremental de objetos é feita, preferencialmente, por composição

através de mecanismos como delegação e encaminhamento.

Requisitos Atendidos

Uma das vantagens prometidas na tecnologia de objetos é a sua habilidade de facilitar a modificação e extensão incremental de código através de mecanismos de extensão embutidos nas linguagens de programação. Estes mecanismos visam permitir a reutilização, combinação e reorganização de objetos para o projeto de novas aplicações. Dentre estes mecanismos, o mais conhecido é a herança. Entretanto, não existe um consenso a respeito da implementação de herança nos compiladores, nem com relação à semântica associada. Uma possível solução para este problema é a restrição do uso da herança à herança de interface, como sugerido em CORBA [OMG 96 A]. Além disso, o uso de herança e outros mecanismos de extensão não são sinônimos de reutilização e linguagens orientadas a objetos oferecem normalmente apenas um deles. O planejamento de objetos para compor uma aplicação deve ser criterioso para permitir a reutilização e facilidade de interação com outros objetos.

Com relação à evolução dinâmica, é desejado que novas classes possam ser concebidas, compiladas e adicionadas livremente em uma aplicação [Gue 96] durante a sua execução. A tecnologia de objetos dinâmicos (*dynamic objects*) [Lad 97], recentemente disponível, pode solucionar parcialmente este problema. Se estas facilidades não estiverem disponíveis, a alternativa viável é a parada da aplicação, recompilar o novo código e reiniciar a aplicação. Esta alternativa é um tanto estática e freqüentemente é necessário o acesso ao código fonte dos módulos a serem estendidos.

Extensões dinâmicas baseadas em mecanismos de herança não são facilmente implementáveis em ambientes distribuídos. Compiladores para linguagens orientadas a objetos baseadas em classes utilizam sistematicamente a verificação estática de tipos na implementação de herança e muitas vezes as relações entre objetos são definidas em tempo de compilação. Esta verificação não pode ser facilmente feita em um sistema distribuído. Isto levou a afirmação de que "… herança é incompatível com distribuição …" em [Weg 92]. No caso da utilização de linguagens orientadas a objeto baseadas em protótipo, a herança dinâmica pode ser obtida dado que o novo objeto será concebido pela clonagem de um objeto em execução e os problemas de falta de verificação de tipos levantados já estão resolvidos nas implementações de linguagens desta natureza.

A inclusão de herança dinâmica está sendo fortemente considerada em linguagens orientadas a objeto de modo a permitir extensibilidade dinâmica [Pou 94, Kic 96, Ber 96].

1.3.2 Reflexão

1.3.2.1 Conceitos Básicos

O termo reflexão dentro de um contexto genérico tem dos sentidos. O primeiro, com o significado de introspeção ou meditação, e o segundo no sentido de refletir um evento, como a imagem em um espelho, por exemplo. Estes dois sentidos podem ser aplicados na utilização do conceito de reflexão na engenharia de software. A idéia da reflexão em sistemas computacionais é permitir que um sistema possa executar algum processamento em benefício próprio, para modificar ou ajustar sua própria estrutura ou comportamento [Str 93].

Quando a reflexão é utilizada os sistemas são divididos em um nível base, onde a computação das funções básicas (ou computação externa) são realizadas, e em um meta-nível, onde os aspectos não-funcionais (ou computação internas) são implementados [Lis 95 e Rub 97].

Na literatura encontramos duas abordagens para a reflexão: estrutural e computacional (ou comportamental) [Lis 95]. Na reflexão estrutural (associado ao sentido de introspecção) o meta-nível pode inspecionar e manipular a estrutura de um programa. A identificação de tipos de dados em linguagens de programação, hierarquia de classes e herança em linguagens orientadas a objeto são exemplos de reflexão estrutural. A linguagem Java [Java A], por exemplo, oferece reflexão estrutural. A reflexão computacional (associado ao sentido do "espelho") permite que o comportamento das funções básicas de um programa sejam monitorados e controlados no meta-nível. Operações como a manipulação de variáveis (leitura e escrita), chamada de funções ou invocações de métodos em linguagens orientadas a objeto, programadas no nível-base podem ser interceptados e desviados para o meta-nível. O meta-nível pode acrescentar alguma computação ao evento e tem a opção de reencaminhar a operação ao curso original ou não. Por exemplo, a escrita em um buffer pode ser controlada no meta-nível para que não seja ultrapassada sua capacidade de armazenamento ou para que não se

sobreponha novos elementos a elementos ainda não consumidos.

Aplicações construídas a partir da abordagem de reflexão computacional podem utilizar módulos compostos por um componente base e um meta-componente. O meta-componente, ou reflexo, tem uma interface com o componente base através da qual pode obter informações internas do componente base ou interceptar chamadas que deveriam ir diretamente para o mesmo. Uma invocação interceptada e manipulada pelo meta-componente, pode ou não ser reencaminhada para o componente original (figura 1.1). O meta-componente pode ser concebido para ter acesso ao ambiente de execução que hospeda a aplicação, permitindo ajustes no componente base em reação à eventos de contexto de baixo nível (por exemplo, um *tick* de relógio).

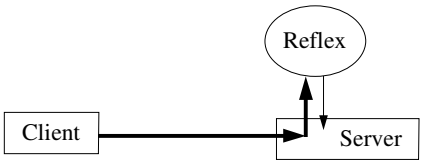


Figura 1.1 - Reflexão como interposição de código

Em certas aplicações cliente/servidor pode ser necessário que tanto o módulo cliente como o módulo servidor sejam refletidos. Por exemplo, em um sistema distribuído de arquivos tanto a parte do cliente local como o serviço remoto de arquivos devem compreender o protocolo de *redirecionamento*. Este protocolo pode ser implementado por reflexão computacional. Neste caso, uma aplicação que solicite uma leitura ou escrita em um arquivo remoto, o faz como se o arquivo fosse local. Este seria o nível base. O próprio sistema operacional intercepta e redireciona o pedido para o sistema remoto de arquivos. Este serviço do sistema operacional estaria no meta-nível. O sistema de arquivos da outra estação, por sua vez, recebe o pedido e reencaminha o mesmo para o seu próprio sistema local de arquivos. A arquitetura deste exemplo é ilustrada na figura 1.2.

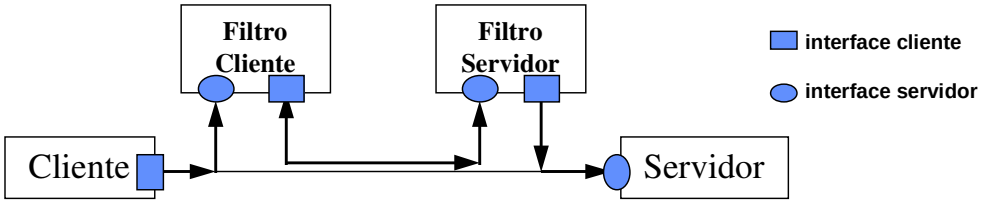


Figura 1.2 - Interfaces necessárias em um esquema reflexivo

Dentre exemplos de domínios de aplicação que podem se beneficiar do modelo de reflexão e que estão sendo considerados em alguns trabalhos encontram-se: tolerância a falhas, sistemas de tempo-real, sincronização de sistemas concorrentes e esquemas para tratamento de exceções [Fab 96, Fra 97 e Gol 97]. Alguns trabalhos na área de sistemas operacionais incluem a reflexão como mecanismo para permitir a extensão e ajuste de funções [Mit 97, Chi 97, Gow 95, Zim 96, Wu 98 e Sin 97].

O conceito de reflexão computacional vem sendo implementado no nível de linguagem de programação, principalmente em linguagens orientadas a objeto. Algumas implementações experimentais, tais como C++[Chi 95 e Gow 96], CLOS [Kic 97] e MetaJava (MetaXa) [Kle 96], estão concentradas em extensões especiais de compiladores de linguagens orientadas a objeto. A combinação de características da tecnologia de objetos mencionadas na seção 1.3.1 e a flexibilidade provida pela reflexão computacional será explorada na próxima subseção.

1.3.2.2 Tecnologia de Objetos e Reflexão

Em linguagens orientadas a objeto que oferecem características reflexivas, uma classe, agora chamada classe base, pode ser associada a uma meta-classe. A sintaxe da linguagem deve oferecer uma forma de se estabelecer uma conexão entre a classe base e a meta-classe. Esta conexão é normalmente expressa por construções de reificação e reflexão. A *reificação*[1] é utilizada para indicar que um evento direcionado para uma classe base deve ser tratado pela respectiva meta-classe e para tornar informações relevantes, internas a classe base, disponíveis para a meta-classe. A construção de

---

[1] O termo reificação poderia ser traduzido como **materialização** ou **exposição**. Por reificação quer se dizer que alguma parte ou evento associado a classe base é passível de ser referenciada e manipulada no meta-objeto como um objeto, ou seja, um evento pode ser materializado, por exemplo, em um objeto. Isso pode valer para, por exemplo, a invocação de um método, que pode ser interceptada pelo meta-objeto, para uma variável, no caso de reflexão computacional e até mesmo à própria estrutura da classe

reflexão efetivamente intercepta os eventos e trata os mesmos no contexto da meta-classe. As regras e sintaxe que descrevem a conexão entre classe case e meta-classe são chamadas de Protocolos de Meta-Objeto (*Meta-Object Protocol (MOP)*). Neste contexto, a reflexão é dita computacional devido à sua atuação no comportamento e na computação do objeto base.

O paradigma da reflexão ganhou aspectos particulares em sistemas construídos com linguagens orientadas a objetos devido a aptidão natural destas linguagens em facilitar a especificação de objetos com características de outros objetos, mais primitivos, através da *herança*. Com a utilização de herança um objeto-base pode ser utilizado para a criação de seu meta-objeto, que herda, por exemplo, a interface do objeto-base e seus métodos básicos, e altera os procedimentos de atendimento dos métodos necessários para dotar o objeto-base com as novas funcionalidades. Deve ficar claro, entretanto, que o meta-objeto não depende apenas de herança para ser implementado: são necessários mecanismos que permitam a interceptação das mensagens, estabelecendo uma conexão entre a instância do objeto-base e seu meta-objeto, e que tornem disponíveis para o meta-objeto as informações relevantes do objeto-base.

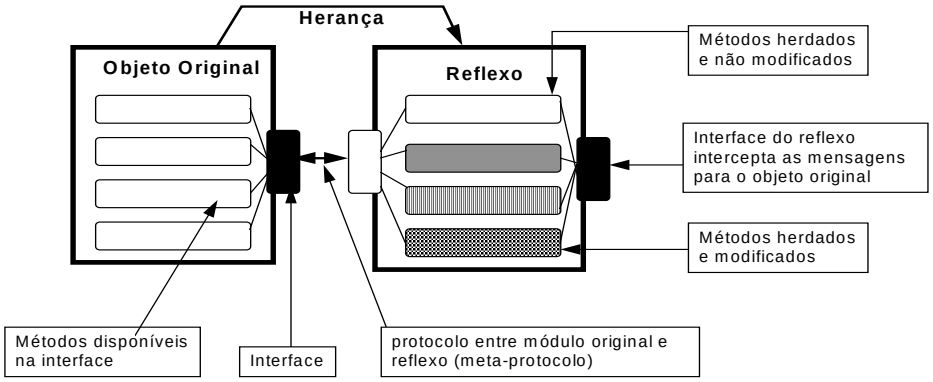


Figura 1.3 - Objeto original e o objeto refletido

1.3.2.3 Implementações

As implementações conhecidas de linguagens que oferecem reflexão são classificadas de acordo com a forma de prover a interação entre nível base e meta-nível. A reflexão

---

base no caso de reflexão estrutural.

pode ser oferecida em tempo de compilação, carga ou execução.

As vantagens potenciais do uso da reflexão, em tempo de compilação, são limitadas pela tecnologia dos compiladores. A implementação do conceito de meta-objetos em linguagens orientadas a objetos, como C++, não parece ser trivial. Um exemplo de implementação é o Open C++ [Chi 95], um pré-processador de C++ que oferece um MOP. É advogado que os conceitos de reflexão são utilizados para a geração de código mais facilmente, permitindo que um sistema seja programado em seus níveis funcionais e não-funcionais separadamente, antes da compilação. Após a compilação objetos base e meta-objetos são ligados no mesmo módulo executável, não existindo a necessidade de um sistema executivo para suportar o meta-nível.

Na reflexão em tempo de carga, a relação entre objetos-base e meta-objetos é estabelecida durante a carga da aplicação. Neste momento, módulos compilados para admitir ligação dinâmica são conectados a módulos de um meta-programa, formando uma unidade executável. A reflexão em tempo de carga também é estática, no sentido que as relações entre objetos base e meta-objetos não podem ser alteradas durante a execução.

Entretanto, se considerarmos que em sistemas distribuídos existe a necessidade de mudanças dinâmicas na operação da aplicação, a solução de um sistema compilado como um módulo único não pode atender a todas as situações. Mesmo assim, pode-se conceber uma biblioteca de meta-objetos com implementações das extensões mais populares e críticas em desempenho [Loq 96] e utilizar-se destes meta-objetos no nível da programação quando necessário.

A reflexão em tempo de execução permite que as relações entre objetos base e meta-objetos seja estabelecida durante a execução do programa. Na maioria das implementações deste tipo de linguagem, meta-objetos também podem ser instanciados dinamicamente. Nesta abordagem, geralmente utiliza-se o suporte de um sistema executivo ou interpretador que gerencia a instanciação de objetos base e meta-objetos e garante a coerência dos sistemas criados baseados nestes elementos [Cam 96, Zim 96, Gol 97]. Um suporte mínimo nestes interpretadores deve permitir também que eventos como a invocação a um método ou a escrita de uma variável no nível base sejam interceptados e manipulados no meta-nível antes de serem repassados ao nível-base.

Este suporte, entretanto, apresenta a desvantagem do *overhead* inerente ao monitoramento dos eventos e gerenciamento do repasse das operações para o meta-nível.

Uma possibilidade adicional para linguagens reflexivas é a utilização do MOP para prover extensibilidade nas construções sintáticas e semânticas da própria linguagem de programação, chamada de linguagem aberta (*open language*) [Chi 95, Lee 95, Kic 97]. Este esquema pode tornar o uso da reflexão computacional mais fácil, se uma sintaxe apropriada puder ser construída para a programação da associação entre objetos base e meta. As vantagens adicionais do uso da reflexão para o programador da aplicação, neste caso, se devem à transparência na inclusão de novas funcionalidades agregadas por meta-objetos e no conforto sintático da programação. No trecho de código seguinte [Chi 95] pode ser observado que no programa sem reflexão, o objeto *Node* precisa herdar as características da classe *PersistentObjetct* para se tornar um objeto persistente, enquanto que no programa utilizando reflexão em um linguagem aberta, a persistência é implementada como uma extensão sintática da linguagem C++.

```
class Node: public PersistentObject
{
   public:
      Node *next;
      doublevalue;
};
```

```
persistent class Node
{
   public:
      Node *next;
      doublevalue;
};
```

Comentários

Em uma aplicação distribuída, um objeto que ofereça serviços genéricos, pode ter estes serviços especializados ou diferenciados através do uso adequando de meta-objetos. Características como a transparência de localização, segurança, tempo-real ou tolerância a falhas [Fab 95, Lis 95, Fra 97], podem ser adicionadas por reflexão (figura 1.4). Neste sentido, podemos afirmar que a reflexão amplia as possibilidades de reutilização e extensão das linguagens orientadas a objeto.

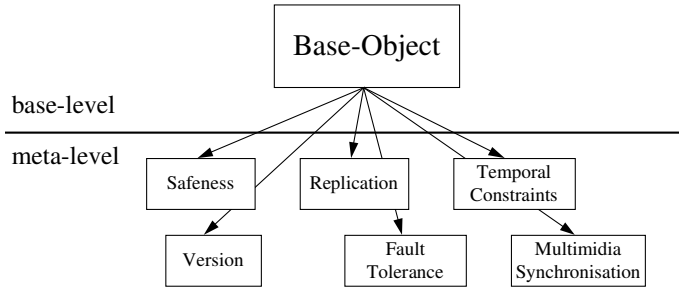


Figura 1.4 - utilizando reflexão para adicionar características não-funcionais

Requisitos Atendidos

Dentre os requisitos para o projeto de aplicações distribuídas, a reflexão atende melhor a separação de interesses, através da distinção de níveis base e meta. De uma forma geral, a metodologia de programação utilizada em conjunto com a reflexão leva à concentração de aspectos funcionais no nível base e ao tratamento aspectos não-funcionais no meta-nível.

Como se dá com a tecnologia de objetos, a reflexão também não é um instrumento direto para a obtenção de modularidade e reusabilidade. Estas serão preocupações do programador da aplicação. Entretanto, o requisito de reusabilidade pode ser obtido se os meta-objetos forem cuidadosamente concebidos para serem utilizados em várias situações e armazenados em uma biblioteca de meta-objetos.

Aspectos dinâmicos da reflexão são desafiadores. Por um lado, aspectos importantes para um sistema distribuído, como já relatado, podem ser adicionados por reflexão de objetos base, dado que a linguagem de programação em uso tenha características adequadas para tal e meta-objetos adequados estejam disponíveis. Por um outro lado, a seleção dinâmica de meta-objetos, a ligação dinâmica entre objetos base meta-objetos ou a evolução dinâmica de aplicações baseadas em meta-arquiteturas são difíceis de se tratar [Lee 95]. A maioria dos MOPs tem que ser programados em um estilo estático, em que as classes ou objetos a serem refletidos e seus respectivos meta-objetos são selecionados no código e então compilados. Depois disto nada mais pode ser feito. Ambientes que contemplem a seleção dinâmica de meta-objetos ainda não estão amplamente disponíveis [Gow 96, Oli 98].

Os benefícios oferecidos pela reflexão computacional para a engenharia de software ainda não são evidentes porque a programação dos MOPs não é exatamente fácil e

linguagens de programação reflexivas ainda não estão amplamente disponíveis ou disseminadas e são especializadas. Reflexão pode ser utilizada mais facilmente por um programador de aplicação se uma biblioteca de meta-objetos estiver disponível, mas um programador de meta-nível é necessário para construir a biblioteca.

1.3.3 Configuração

1.3.3.1 Conceitos Básicos

O princípio básico da tecnologia de configuração supõe a existência de componentes[2] de software que serão selecionados e interligados para construir uma aplicação com características próprias. Nesta tecnologia a programação dos componentes de software e a estrutura da aplicação são considerados aspectos diferentes [Sha 95 e Mag 96]. Ao se estabelecer uma configuração é possível selecionar-se atributos específicos e formas de interligação dentre um conjunto de opções disponíveis para cada componente (figura 1.5). Adicionalmente, novos componentes e formas de interligação podem ser incluídas.

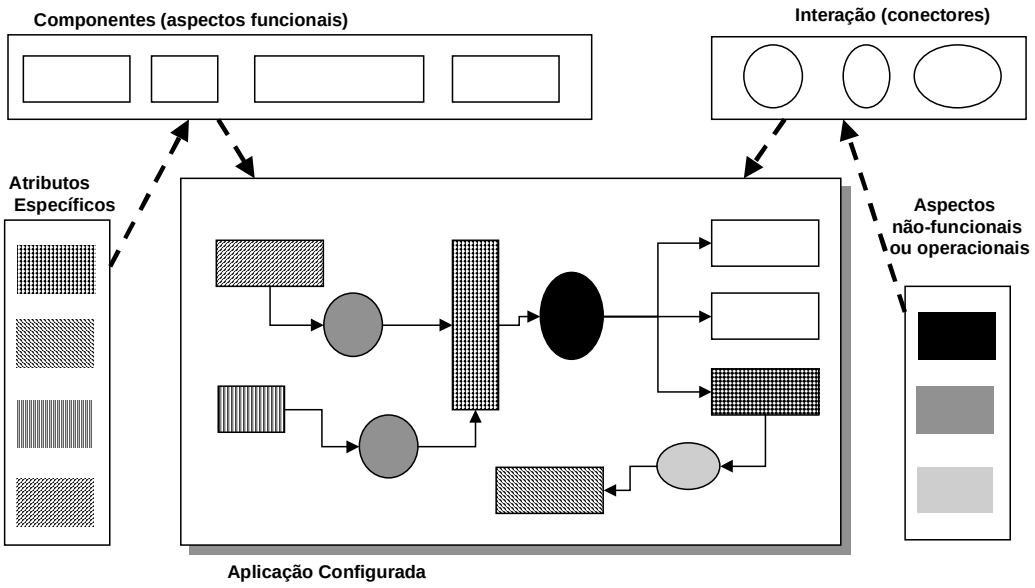


Figura 1.5 - suporte para a configuração de aplicações distribuídas

[2] O termo **componente** possui conotações diferentes dependendo do contexto. O significado adotado neste texto é o mais genérico. Para os nossos propósitos um sistema ou aplicação são compostos por componentes. Um componente é um elemento que pode interagir com outros componentes.

Tradicionalmente, a configuração de uma aplicação é descrita em uma linguagem. Através desta linguagem descreve-se a estrutura desejada para a aplicação, com os módulos utilizados, interconexões destes módulos e um estado inicial. Dependendo do nível de abstração pretendido os termos: linguagem de programação de configuração (*configuration programming languages*) (CPL), linguagem de descrição de arquitetura (*architectural description language*) (ADL) [Bis 96], linguagem de interconexão de módulos (*module interconnection languages*) (MIL) ou linguagem de composição (*composition language*) (CL) [Iss 96] são encontradas na literatura. Através destas linguagens descrevem-se a um roteiro de configuração que espelha a arquitetura das aplicações. Ester roteiros podem ser traduzidos para uma linguagem de programação tradicional ou para um forma executável. A utilização de ferramentas gráficas [Wer 95] como interfaces *front-end* para as linguagens de configuração permite que os sistemas sejam descritos de uma forma visual e mais convenientemente. As linguagens de configuração possuem características interessantes para a concepção de aplicações, tais como:

- composição e construção hierárquica de módulos e da própria aplicação [Med 98];
- reusabilidade de módulos que tenham sido concebidos e depurados para outra aplicação [Voa 98];
- geração automática de código a partir da configuração descrita [Gar 94];
- verificação de propriedades das aplicações descritas [All 94].

A execução de uma aplicação pode ser tratada de duas formas: estaticamente, quando a descrição de uma configuração é utilizada para gerar uma aplicação que, uma vez configurada, executará com o suporte de sistema operacional tradicional, ou dinamicamente, quando a mesma executará sobre o suporte de um ambiente executivo (com descrito em [Szt 95]) que permite intervenção de um agente de configuração, externo, através de comandos de configuração. No escopo deste trabalho, é considerado o segundo caso. Agentes de configuração podem alterar a estrutura da aplicação, adicionando novos módulos, removendo módulos existentes, alterando a conexão de módulos ou modificando o comportamento de elementos.

Elementos utilizados para a interligação dos módulos de uma aplicação são chamados genericamente de conectores [Bis 96]. O conector é uma abstração utilizada no nível de

configuração com dois objetivos: expressar a interação de módulos e oferecer, ao programador da arquitetura, acesso aos mecanismos de comunicação nativos do sistema operacional de uma maneira uniforme. Com o uso do conector não existe a necessidade de cuidados com detalhes específicos com aspectos de comunicação durante o desenvolvimento dos módulos funcionais. Um módulo é projetado de tal forma que a interface através da qual ele pode oferecer ou usar serviços de outros módulos é sempre interligada por conectores. Uma outra característica do conector é o estilo de interação que ele suporta. Podemos ter conectores que implementem estilos de interação diferentes e selecionar o mais conveniente para uma dada situação. A figura 1.6 ilustra este mecanismo.

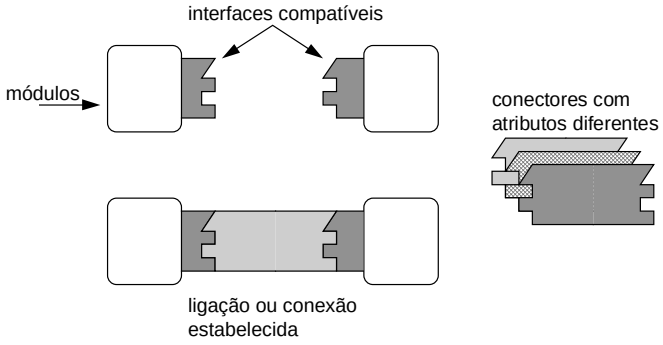


Figura 1.6 - Conexão de dois módulos

Uma das vantagens de se explicitar a conexão entre dois módulos é a clareza com que os requisitos funcionais e não-funcionais podem ser programados [Bis 96 e Sha 95]. Na figura 1.6 isto é esquematizado com a possibilidade de se selecionar um dentre os diversos conectores com diferentes atributos para atender a um aspecto de comunicação. Além disso, outras características podem ser associados à ação de se interligar módulos através de um conector:

- garantir que a assinatura das interfaces dos componentes interligados sejam compatíveis. Esta verificação poderá ser feita estaticamente analisando-se o *script* de configuração ou dinamicamente quando a interligação de dois módulos for realizada por uma agente de configuração. Módulos com interfaces com assinaturas casadas podem ser automaticamente interconectados por conectores genéricos [Lob 99 A]. Por outro lado, problemas de incompatibilidade de assinaturas podem ser superados

com a utilização um conector que implemente uma ponte [Weg 96] que faça a tradução adequada.

- o conector é um elemento onde potencialmente podem ser controlados aspectos não-funcionais da aplicação que envolvam a interação entre componentes. Por exemplo, na caso de uma conexão de módulos que residem em um mesmo nó, a interação pode, em um nível maior de detalhe, ser feita por mecanismos de invocações a métodos, *sockets*, *pipes*, *streams* ou memória compartilhada. No caso em que os módulos estejam distribuídos, o aspecto do protocolo de comunicação pode ser introduzido. Por exemplo, TCP, UDP ou RTP podem ser selecionados para esta conexão. O suporte à configuração, em conjunto com o ambiente de execução, deve permitir a seleção dos mecanismos a serem utilizados e pode sugerir a melhor opção.

Dentro de um ambiente de configurador ou ADL os conectores também podem ter o papel de caracterizar o estilo da arquitetura de software utilizada pela aplicação, como proposto em [All 97T]. Um conector pode ser concebido para implementar, sob o ponto de vista da arquitetura de software, um *pipe*, um barramento de software ou um gerenciador de eventos. Esta funcionalidade adicional dos conectores, permite, ainda, a derivação de informações importantes sobre a aderência ou corretude da configuração de uma aplicação à determinado estilo de arquitetura de software.

Um dos desafios conhecidos na realização de um ambiente com o esquema de conectores é a dificuldade de se mapear as descrições do nível de configuração em uma implementação com estruturas e contornos bem definidos [Bis 96]. A implementação tradicional de conectores e as interfaces entre módulos e conectores pode se apresentar difusa e entrelaçada com outras partes do código que não dizem respeito à interação [Hür 95 e Aks 96]. Este problema tem sido contornado com a geração de esqueletos de código por ferramentas associadas à linguagem de configuração ou pela integração de mecanismos de comunicação dentro das linguagens de programação.

Comentários

O desenvolvimento aplicações distribuídas complexas tem o seu tempo reduzido se houver o suporte de um ambiente de configuração. Inicialmente o projetista concebe a arquitetura da aplicação e seus requisitos funcionais. Somente depois desta fase o projetista precisa se preocupar com aspectos não-funcionais da aplicação. Estes

aspectos podem ser contemplados através da seleção de conectores adequados ou através da utilização de módulos contendo funcionalidade especializada. Dentre os aspectos não-funcionais podemos citar a tolerância a falhas, que pode ser seletivamente configurada para cada módulo [Loq 97], protocolos de comunicação utilizados na comunicação remota e a reserva de recursos, tipicamente em aplicações multimídias [Flo 96, Fro 98].

A utilização da tecnologia de configuração não limita a linguagem de programação com a qual são concebidos os módulos básicos. É possível assim que a tecnologia de objetos seja utilizada na configuração. Existem duas abordagens para combinar estas tecnologias. Um dos problemas a serem resolvidos é como mapear a abstração de módulos e conectores, da configuração, em classes ou objetos de uma linguagem de programação orientada a objetos. Para o módulo, uma solução possível é o seu mapeamento em uma classe. A interface do módulo é mapeada em métodos da classe. Um agente externo de configuração liga cada instrução de invocação de um objeto chamador a um método do objeto chamado, resolvendo as referências de endereçamento que não eram conhecidas pelo chamador antes da ligação. Em um outro estilo de abordagem, os conectores são considerados como elementos de primeira classe, possivelmente (mas não necessariamente) também mapeados em objetos. Nesta solução, um objeto mapeado a partir de um módulo descrito no nível da configuração, deve ter uma interface compatível com o objeto implementando o conector. Quando dois objetos precisam ser ligados, o agente de configuração instancia um objeto conector e interliga ao mesmo cada um dos dois objetos. A flexibilidade desta abordagem é atraente. O objeto conector instanciado pelo agente de configuração pode ser selecionado pelo usuário e adaptado para incorporar características desejadas. Uma outra vantagem desta abordagem é que o conector pode ser preparado para traduzir diferentes padrões de interação e comunicação, permitindo que objetos de ambientes diferentes possam ser conectados.

Requisitos Atendidos

A tecnologia de configuração satisfaz grande parte dos os requisitos estabelecidos na primeira seção. A reutilização é possível na medida em que uma aplicação pode ser construída por composição e adaptação de componentes disponíveis. Modularidade é

essencial para a tecnologia de configuração e é imposta pelo encapsulamento e pela restrição de que todas as interações entre componentes sejam feitas através de interfaces padronizadas. O requisito de extensibilidade é atendido permitindo-se que módulos complexos ou especializados sejam construídos e adaptados através da composição de módulos mais primitivos.

A separação de interesses é também contemplada na configuração. A seleção adequada dos módulos básicos, e o comportamento e estilos de interação adaptáveis que definem as características não-funcionais de uma aplicação, são as formas com que a tecnologia de configuração oferece a separação de interesses. Em um outro nível, além de descrever os estilos de interconexão e o comportamento específico das interações dos módulos, um projetista pode usar módulos especializados que implementem características não-funcionais de uma aplicação, que não se relacionam com a interação dos módulos, para adicionar características necessárias através de composição.

Em um ambiente de configuração, a evolução dinâmica é atendida com a possibilidade de se agregar mais módulos ou reconfigurar módulos existentes, sem a paralisação de todo sistema. Primitivas de reconfiguração e o suporte adequado de um ambiente de execução devem impor a consistência para a nova configuração construída dinamicamente. Pode ser observado que não é incomum que uma reconfiguração possa, além de ser iniciada por um agente externo (como por exemplo, um módulo gerenciador de configuração), ser iniciada por um módulo que está participando da aplicação [Kra 90, End 92].

Um dos pontos negativos da configuração, a despeito de todas as suas vantagens, é o fator disponibilidade. Até o presente momento, não existe um padrão mais aceito, uma linguagem mais utilizada ou ambientes de configuração padronizados. Este problema vem sendo superado com a crescente adoção do CORBA da OMG como ambiente para o desenvolvimento e suporte de sistemas distribuídos. A implantação de mecanismos de configuração sobre CORBA é factível e se beneficia de todas as vantagens subjacentes da tecnologia de objetos e do esforço de padronização da OMG [Loq 96, Iss 96, Cra 97 e Szt 97].

1.4 Interrelacionado os Conceitos

Neste ponto temos um conjunto de requisitos desejáveis em modelos, ferramentas e metodologias voltadas para a concepção de sistemas distribuídos. Um conjunto de tecnologias candidatas para atender a estes requisitos foi analisado. O próximo passo será dado relacionando-se estas tecnologias. Primeiro, as combinações possíveis serão discutidas. Em seguida, uma comparação das diferenças de como cada uma delas atende aos requisitos básicos conclui esta subseção. Com estas informações uma boa medida da adequabilidade de cada tecnologia pode ser obtida e um conjunto de diretivas para o projeto de sistemas distribuídos é, então, derivado.

1.4.1 Combinando as Tecnologias

Admitimos que a tecnologia de objetos é adequada para a programação de módulos básicos e, portanto, será envolvida, por definição, em todas as combinações das tecnologias de reflexão computacional e configuração discutidas nesta seção.

Como exemplo de utilização de reflexão e configuração de meta-objetos podemos utilizar uma aplicação multimídia, como a vídeo conferência onde os meta-objetos utilizados são compostos por configuração. Na configuração desta aplicação, uma série de parâmetros indicando requisitos de QoS determinam a seleção adequada de meta-objetos a serem interpostas em uma hierarquia de meta-objetos. Na figura 1.7 o objeto base, aplicação vídeo conferência, utiliza um meta-objeto que implementa um mecanismo de fluxo contínuo codificado para realizar a comunicação, que por sua vez utiliza o protocolo RSVP [Szt 97]. O projeto do objeto base é transparente em relação à comunicação e codificação [Oli 98, Rav 98 e Wel 98].

Na escolha entre reflexão e configuração, um compromisso entre eficiência e flexibilidade deve ser buscado. Se uma linguagem orientada a objetos que oferece reflexão em tempo de compilação for utilizada, os objetos base e os meta-objetos associados serão compilados como um só objeto. É desejável, entretanto, poder selecionar-se meta-objetos e associá-los aos objetos base quando necessário. Este seria o caso de ambientes que suportam reflexão em tempo de execução e para isso, suporte em tempo de execução e técnicas de ligação dinâmicas devem estar disponíveis. Uma interface padronizada associada ao objeto base permitiria que meta-objetos fossem selecionados e instanciados apenas no momento da execução, como ilustra a figura 1.7. Alguns esforços tem sido direcionados para oferecer estas características em linguagens

de programação reflexivas, mas estas ainda não são uma completa realidade, pois demandariam que a reflexão computacional estivesse associada à abordagem da configuração. Este último ponto é de interesse especial em nossa investigação.

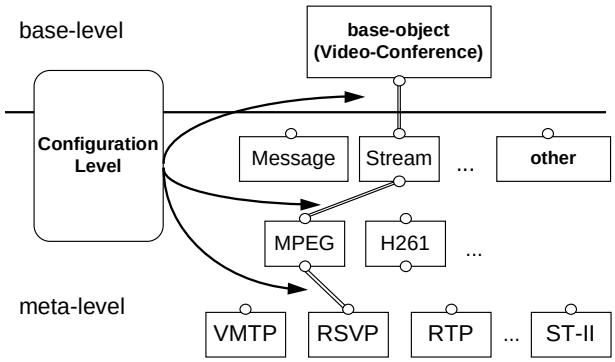


Figura 1.7 - Ambiente multiprotocolo configurável baseado em reflexão

Utilizando-se um sistema reflexivo, a técnica de compressão a ser utilizada, no exemplo da figura 1.7, para transmitir os dados de vídeo deve ser selecionada em tempo de programação, indicando-se a associação entre o objeto base e o meta-objeto selecionado. Como alternativa, os conceitos de reflexão poderiam ser utilizados adotando-se, na descrição do sistema, a tecnologia de configuração. A relação entre os módulos-base e os meta-módulos, neste caso, poderia ser implementada conectando-se dois módulos que implementam respectivamente as funções de um objeto-base e um meta-objeto, com o cuidado de selecionar-se um meta-módulo para as necessidades específicas da aplicação, e com vantagem adicional de que isto pode ser feito dinamicamente durante a execução.

No ambiente RIO [Szt 95] uma abordagem conjunta de configuração e reflexão computacional foi utilizada para oferecer flexibilidade ao subsistema de comunicação para suportar a seleção dinâmica de características específicas nos protocolos. Neste ambiente, um módulo de meta-nível GM (Gerente Multiprotocolo) tem a missão de efetivar interações entre módulos da aplicação e o subsistema de comunicação, habilitando a operação correta do protocolo selecionado. Módulos GM implementando estilos de comunicação diferentes podem ser configurados e substituídos para atender a requisitos específicos de comunicação de módulos compondo uma aplicação. Uma abordagem similar é utilizada em [Pan 93] com um sistema baseado em Atores (*Actors*)

e em [Ber 96T] com a abstração de Filtros de Composição (que será revista em detalhes no capítulo 2).

1.4.2 Comparando as Tecnologias

A tabela 1.1 sumariza as diferenças e as características semelhantes de cada tecnologia mencionada. Em alguns itens avaliados existe uma clara distinção em porque e quando aplicar cada uma. Discute-se agora alguns pontos desta tabela.

Reusabilidade. Todas as tecnologias selecionadas contemplam a reusabilidade, mas em níveis diferentes. A tecnologia de objetos pode se basear em bibliotecas para oferecer classes úteis. Classes especializadas podem ser criadas por herança, delegação ou agregação a partir destas classes básicas. A reutilização pode ser maximizada com a adoção das técnicas de padrão de projeto ou frameworks [Gam 94] em que soluções completas baseadas em objetos podem ser reutilizadas.

A tecnologia de reflexão permite que objetos base e meta-objetos sejam reutilizados. Várias aplicações podem se beneficiar da disponibilidade de bibliotecas de meta-objetos e atender requisitos não-funcionais mais rapidamente. Uma outra possibilidade é a reutilização de módulos de aplicações inteiras no nível base e *customizar* os componentes de meta-nível, obtendo-se um comportamento inteiramente novo para o sistema original.

Ambientes baseados em configuração permitem a reutilização de módulos primitivos na composição de módulos complexos, ou as próprias aplicações em si. A abordagem de configuração é dirigida à reutilização, na medida em que os módulos precisam considerar uma certa padronização de interface e podem ser isolados dos estilos de interação, facilitando a conexão com outros módulos. Ainda assim, os módulos em um ambiente de configuração também devem ser planejados para serem reutilizáveis.

Separação de Interesses. A tecnologia de objetos não oferece nenhum mecanismo para atender ao requisito de separação de interesses explicitamente. Mesmo assim, a tecnologia de objetos pode ser utilizada para implementar aplicações onde uma separação de aspectos diferentes na programação é requerida de forma limitada, contando com o gabarito dos projetistas.

A reflexão computacional, por sua vez, preconiza a separação de interesses como

característica básica. Linguagens de programação reflexivas permitem uma clara distinção entre o nível de programação de funções e o nível não-funcional. Cada um destes níveis pode concentrar as preocupações com aspectos diferentes das aplicações.

A configuração atende à separação de interesses permitindo que o projeto, a especificação do comportamento e estilos de interconexão dos módulos sejam planejados separadamente. Requisitos não-funcionais e operacionais podem ser implantados externamente, no domínio da configuração, pela composição de componentes e pela seleção adequada do estilo de interação entre estes componentes. Para obter características equivalentes, a tecnologia de reflexão depende de um ambiente de execução especializado.

Evolução Dinâmica. Como discutido nas duas primeira seções, o requisito de evolução dinâmica engloba dois aspectos: a necessidade de extensibilidade, permitindo que as aplicações evoluam a partir de uma configuração inicial e a necessidade desta evolução acontecer dinamicamente.

A extensibilidade é obtida na tecnologia de objetos pelo uso da herança e mecanismos similares. Objetos base podem ser estendidos para especializar ou generalizar seu comportamento. Com linguagens reflexivas, meta-objetos podem prover extensões para objetos base, modificando ou adicionando características ao comportamento do objeto base original. Herança e reflexão têm objetivos similares com relação a extensão, mas abordagens diferentes: a herança permite a criação de novas classes e a reflexão oferece a adaptação de classes com granularidade fina (em [Jag 96] a herança é modelada por reflexão e algumas considerações sobre sua expressividade são apresentadas). A abordagem da configuração é mais ampla no que diz respeito a extensibilidade: aplicações podem ser estendidas pela substituição de módulos, adição de novos módulos e reconfiguração da estrutura presente. Um comportamento diferente pode ser imposto a módulos selecionados alterando-se características não-funcionais nos conectores interligando estes módulos através de comandos de configuração.

Considerando-se isoladamente orientação a objetos, a obtenção de evolução dinâmica é um tanto difícil. A inclusão ou remoção de objetos em uma aplicação pode ser obtida se a tecnologia de objetos dinâmicos for utilizada [Lad 96]. Entretanto, permitir que novas classes e herança sejam definidas durante a execução é mais difícil. Problemas tais

como ligação dinâmica e retardada (*late binding*) de objetos e a manutenção da consistência e referências são mais evidentes nestes casos e agravadas em sistemas distribuídos.

Os mesmos problemas identificados na tecnologia de objetos podem ocorrer na tecnologia de reflexão. Na maioria dos MOPs disponíveis, os meta-objetos são definidos estaticamente. As classes ou objetos a serem refletidos e que partes devem ser reificadas são definidas no código fonte. A extensão ou adição de meta-objetos normalmente é feita modificando-se o código. Se estas operações pudessem acontecer dinamicamente, os sistemas construídos com a tecnologia de objetos e reflexão computacional poderiam evoluir dinamicamente.

Sistemas configuráveis podem evoluir dinamicamente por ações de reconfiguração, ocasião em que várias características da aplicação podem ser modificadas. Ambientes que suportam configuração dinâmica e agentes externos de configuração geralmente possuem mecanismos para garantir a reconfiguração consistente da estrutura da aplicação, assegurando que as interações dos elementos que estarão sofrendo reconfiguração sejam graciosamente congeladas, levando a uma evolução suave [Mag 92, End 92].

Abrangência. O requisito de abrangência não é totalmente atendido pelas tecnologias avaliadas, principalmente pela falta de uniformidade e padrões. Por exemplo, enquanto a reutilização é normalmente obtida nas tecnologias analisadas, implementações diferentes da mesma tecnologia geralmente não conseguem interoperar facilmente. Por exemplo, linguagens orientadas a objetos são amplamente disponíveis mas alguns pontos relacionados com as partes internas da implementação dos mecanismos embutidos, como a herança, estão ainda em aberto. Portanto, com relação à sistemas distribuídos, a criação de uma nova classe derivada de uma classe-base definida em uma linguagem diferente, e onde a classe-base e a classe derivada podem estar em nós diferentes, é de difícil obtenção. Linguagens de programação e ambientes de execução com características reflexivas também não são amplamente disponíveis e ainda não existem características padronizadas aceitas universalmente. A tecnologia de configuração possui características positivas em relação à abrangência: módulos básicos podem ser escritos em qualquer linguagem de programação, a tecnologia é madura, comprovadamente útil e amplamente disponível. Entretanto, ambientes de

configuração sofrem do mesmo problema que reflexão computacional: ainda não existem linguagens de configuração ou implementações de ambientes de configuração nem padrões de conversão aceitos universalmente e que sejam interoperáveis [Med 97]. Como colocado na seção 1.3.3, CORBA pode tornar a solução para a padronização mais próxima em relação às tecnologias analisadas, agregando as mesmas em torno do núcleo de sua arquitetura de objetos distribuídos.

Modelo	Objetos	Reflexão	Configuração
reusabilidade	Classes e objetos podem ser reutilizados; Objetos especializados podem ser criados por herança, delegação ou composição; Reutilização requer planejamento, mas pode ser maximizada com o uso de padrões de projeto (*design patterns*) e *frameworks*.	Objetos-base e meta-objetos com interfaces similares podem ser reutilizados. Conjunto de meta-objetos formando meta-arquiteturas para solucionar determinados domínios de interesse podem maximizar a reutilização.	Módulos e conectores podem ser reutilizados na composição de módulos complexos; Requer padronização de interface e estilos de interação; Conectores podem adaptar módulos com interfaces dissimilares; Arquiteturas diferentes podem ser construídas a partir de uma arquitetura básica e ajustando-se módulos e conectores existentes.
separação de interesses	Não atendido explicitamente	O conceito de objetos-base e meta-objetos facilitam a separação de interesses no desenvolvimento de programas.	Módulos básicos podem ter seu comportamento , estilos de interação e aspectos não-funcionais planejados e implantados separadamente; Conectores podem ser responsáveis por aspectos relacionados com a interação de módulos.
evolução dinâmica	Extensibilidade pode ser obtida por herança, delegação ou composição, mas extensibilidade dinâmica é de difícil obtenção. O suporte à ligação retardada e dinâmica pode facilitar a concepção de mecanismos que permitam evolução dinâmica.	Extensibilidade é provida por meta-objetos; Na maioria dos MOPs disponíveis, meta-objetos são definidos estaticamente. MOPs mais flexíveis tem sido propostos para permitir maior dinamismo [Oli 98]. Extensibilidade dinâmica de objetos-base deve ser suportada por um ambiente reflexivo especializado.	Obtido pela capacidade de (re)configuração da arquitetura da aplicação, módulos e conectores simples ou compostos antes e durante a operação.
abrangência	Linguagens orientadas a objeto são amplamente disponíveis; Falta de padronização nos mecanismos de extensibilidade.	Linguagens de programação ou ambientes de execução com características reflexivas ainda não estão amplamente disponíveis; Não existem características padronizadas aceitas universalmente.	ADLs e ambientes de configuração estão disponíveis; Não existem padrões ou implementações específicas aceitas universalmente.

Tabela 1.1: comparação das tecnologias

Comparando-se os mecanismos disponíveis nas tecnologias de objetos, reflexão ou configuração para a construção de aplicações pode-se verificar que eles possuem poder de expressão equivalentes. Este fato foi comprovado em [Loq 97] na área de tolerância a falhas, comparando-se reflexão e configuração. Para produzir uma aplicação, módulos construídos a partir da tecnologia de objetos devem ser programados em alguma linguagem e então combinados. Esta combinação poderia ser feita utilizando-se reflexão ou configuração, dependendo dos fatores apontados nas últimas seções. O que difere as duas técnicas é a abordagem para se descrever esta combinação. No caso da reflexão, a estrutura da aplicação é descrita em parte implicitamente dentro dos objetos e em parte através de uma linguagem de programação especializada. No caso de utilização da abordagem de configuração esta mesma estrutura é descrita explicitamente (e separadamente da implementação) através de uma linguagem de configuração, independentemente da linguagem de programação dos módulos. Um outro ponto que vale ser mencionado é que o esforço na concepção de meta-objetos e na associação de objetos base e meta-objetos é ainda grande, dadas as linguagens de programação reflexivas disponíveis.

1.5 Construindo o Conjunto

Nossa intenção não é afirmar que o projetista de uma aplicação deva utilizar todas as tecnologias avaliadas neste capítulo para ser bem sucedido, mas seria bastante confortável tê-las disponíveis e poder escolher os mecanismos mais adequados para solucionar o problema em mãos. Considerando características positivas de cada tecnologia pode-se visualizar uma cadeia de decisões em relação a quais mecanismos serão adotados para implementar uma aplicação:

1. Para a programação do nível base das aplicações a tecnologia de objetos deveria ser selecionada. Para módulos mais complexos, um padrão de projeto adequado ou framework pode ser utilizado.
2. Reflexão computacional pode ser utilizada na programação de objetos básicos onde ajuste fino ou extensões não-funcionais são necessárias, em um nível particular de detalhe. Seria desejável que a adoção de reflexão ou seleção de meta-objetos fosse

feita dinamicamente e postergada até a execução ou ativada por comandos de configuração. Entretanto, se um MOP não estiver disponível para a programação dos ajustes necessários na aplicação, a abordagem de configuração poderá ser utilizada.

3. A tecnologia de configuração pode ser utilizada em vários níveis de granularidade. Em um nível maior de abstração, a configuração pode ser usada para a construção de grandes sistemas distribuídos baseados em módulos compostos previamente e com uma arquitetura em mente. Em um outro nível, esta mesma tecnologia pode ser utilizada para compor módulos complexos ou refinar módulos muito abstratos em um primeiro momento e depois para estruturar a aplicação.

A tabela 1.2 consolida estas diretivas e complementa a tabela 1.1

Modelo	Objetos	Reflexão	Configuração
Onde aplicar	concepção de módulos básicos	extensão de módulos	concepção de módulos complexos e construção de arquiteturas de sistemas
Exemplos	C++, Smaltalk e Java	Open C++, CLOS [Kic 96], MetaJava	Babel [Mal 93], Write [All 94], Regis [Kra 96], UniCon [Sha 95], OMG-STRUDL [OMG 97]

Tabela 1.2: onde aplicar cada tecnologia

As diretivas apresentadas consideram o presente nível de maturidade, evolução e padronização das tecnologias analisadas. Existe ainda um espaço a ser preenchido antes que possa ser delineado um modelo definitivo para todos os pontos mencionados nas últimas seções.

1.6 Trabalhos Correlatos

Todas as tecnologias apresentadas e problemas discutidos neste capítulo estão sendo atualmente estudados por em várias linhas de pesquisa. Até onde vai nosso conhecimento os aspectos mencionados nestes trabalhos estão sendo estudados isoladamente ou em combinações específicas. Wegner [Weg 96] apresenta um painel bastante completo do tipo de evolução que está sendo esperada em direção a um consenso no projeto de grandes sistemas distribuídos. [Loq 97, Cra 97, Mag 97, Pry 97 e Iss 96] apresentam alternativas no uso de configuração com objetos distribuídos sobre ORBs. [Gow 95, Mit 97, Sin 97, Fab 95 e Fra 97] apresentam soluções para estender a funcionalidade de sistemas utilizando reflexão computacional e explorando os aspectos

da flexibilidade do uso de meta-objetos. A abordagem de [Ber 96T] combina objetos, composição e reflexão limitada, sendo que a principal característica deste trabalho trata da troca de mensagens entre objetos. Algumas destas propostas e propostas baseadas em configuração serão avaliadas no próximo capítulo.

Esta primeira parte de nossa pesquisa complementa outros trabalhos na medida em que estamos identificando requisitos, como reusabilidade, evolução dinâmica, separação de interesses e abrangência, em um sentido amplo, e mecanismos úteis em um conjunto grande de tecnologias, como a reflexão e configuração, que possam, empregadas adequadamente, oferecer os meios para o projeto de grandes sistemas distribuídos. Esta primeira fase é concentrada na obtenção de um modelo derivado da combinação adequada destas ferramentas. Especificamente, nossos esforços estão concentrados para a utilização de configuração como tecnologia principal para o projeto de grandes sistemas distribuídos, aliada à flexibilidade da reflexão computacional com uma perspectiva direcionada para a separação de interesses.

1.7 Conclusões

Neste capítulo foram analisadas as tecnologias de objetos, reflexão computacional e configuração e como estas atendem aos requisitos de reusabilidade, separação de interesses, evolução dinâmica e abrangência. Estes são, em nossa opinião, requisitos que devem ser preenchidos pelas tecnologias que serão utilizadas para o projeto de grandes sistemas distribuídos. Todas as tecnologias analisadas tem os seus méritos e atendem a alguns dos requisitos. Em [Agh 96, Gue 96 e Weg 96] é mencionado que as tecnologias de objeto, hoje disponíveis, não apresentam solução para todos os tipos de problemas de software e, portanto, outras técnicas precisam ser utilizadas concomitantemente. Isso nos levou a pesquisar uma combinação adequada entre as tecnologias de configuração e reflexão computacional.

Com a tecnologia de objetos, a herança, delegação e composição de granularidade fina são mecanismos a serem utilizados na programação de objetos básicos. A tecnologia de objetos e os mecanismos de extensão associados não devem, sozinhos, serem utilizados para a concepção de grandes estruturas ou sistemas. A tecnologia de objetos deve ser encarada com um tecnologia de fundação.

As tecnologias de configuração e reflexão podem ser utilizadas para a construção de sistemas complexos. Em princípio, para projetar a aplicação em um nível de arquitetura ou estrutura, recomenda-se a configuração, enquanto a reflexão computacional pode ser usado no nível de linguagem de programação. O uso de reflexão em combinação com linguagens orientadas a objetos se mostra promissor na construção de módulos completos com diferentes requisitos operacionais, enquanto que as vantagens do uso desta técnica no projeto de grandes sistemas distribuídos, se comparadas com as vantagens obtidas com técnicas de configuração, não são evidentes. A reflexão oferece flexibilidade na construção de componentes básicos, mas não permite facilmente a interferência externa para o ajuste do comportamento da aplicação. Este ajuste pode ser previsto, no entanto, durante a programação da aplicação. A solução de reflexão não é, na maioria das implementações atualmente disponíveis, afeita a evoluções dinâmicas dos elementos de meta-nível. Entretanto, sistemas reflexivos, baseados em um suporte especializado, propostos mais recentemente, como [Oli 98], permitem a substituição de meta-objetos similares em tempo de execução. A configuração, por sua vez, permite que módulos componentes sejam adaptados e interconectados sob demanda e a interação entre os elementos sejam ajustados durante a execução da aplicação. O preenchimento de requisitos não-funcionais com a configuração é simples e dinâmico.

A tecnologia de AOP - Programação Orientada a Aspectos (*Aspect Oriented Programming*) não foi envolvida nas seções de avaliação e comparação por ser a mesma fruto de um novo paradigma. AOP parte da idéia de que sistemas extremamente complexos não podem ser construídos usando o estilo tradicional de modularidade [Kic 96] por conter código de aspectos diferentes entrelaçados. Em AOP, por exemplo, módulos devem encapsular apenas um aspecto da aplicação, não misturando, por exemplo, a funcionalidade básica da mesma com aspectos de comunicação ou coordenação. AOP permite que os projetistas pensem a respeito de aspectos naturais de um sistema, que são capturados separadamente, possivelmente com uma representação diferente, que então são interconectados automaticamente, gerando um sistema resultante final. [Kic 96] e [Lop 97T] apresentam e avaliam um sistema distribuído projetado a partir do paradigma de AOP. Em AOP, os requisitos de separação de interesses e reutilização são suportados, mas em relação aos outros requisitos nada de concreto pode ser dito ainda. Esta tecnologia será analisada no capítulo 2.

Outros requisitos importantes, não considerados diretamente neste trabalho, mas igualmente relevantes, são a **eficiência** e **modelos formais**. Tecnologias com respaldo de modelos formais são mais facilmente aceitas [Med 97]. A tecnologia de configuração associada com a abstração de conectores, por exemplo, é reconhecidamente bem formalizada. A abstração do conector facilita a verificação de propriedades formais como a ausência de *deadlock*, corretude e terminação [Mor 95] e propriedades operacionais (como por exemplo, a coerência e viabilidade de parâmetros de QoS requisitados para cada módulo ou conexão em sistemas multimídia, que podem auxiliar os mecanismos de admissão na rede [Flo 96 e Fro 98]). A eficiência, por sua vez, é um requisito implícito em se tratando de sistemas distribuídos. Por isso, é esperado que todas as tecnologias mencionadas ofereçam desempenho adequado.

No próximo capítulo são analisadas algumas propostas representantes das tecnologias analisadas que se prestam à construção de aplicações distribuídas ou são relacionadas com alguns aspectos desta classe de aplicações como a coordenação, comunicação. Um dos pontos em comum destas propostas diz respeito à abordagem para a construção do *software* onde a separação de interesses está presente de alguma forma. As avaliações e os testes feitos com estes exemplos têm o propósito de levantar subsídios para avaliar a abordagem proposta desta tese.

Capítulo II

Trabalhos Correlatos

2.1 Introdução

Neste capítulo são avaliados alguns trabalhos contemporâneos que apresentam abordagens particulares para a construção de aplicações distribuídas, mas que são correlatos em vários aspectos no que diz respeito ao atendimento dos requisitos estabelecidos no capítulo 1. A avaliação está dividida em duas partes. Na primeira são analisadas propostas que consideram a separação de aspectos, modularidade, reusabilidade e abrangência, como importantes para o desenvolvimento de sistemas distribuídos, mas com abordagens diferentes. Estas propostas estão relacionados com as tecnologias abordadas no capítulo 1, e entre elas estão linguagens de programação com reflexão (estáticas e dinâmicas), Filtros de Composição e Programação Orientada a Aspectos [Lop 97T, Ber 94T e Gol 97T].

Na segunda parte são avaliados alguns trabalhos que consideram a configuração e a programação no nível de arquitetura ou estrutura como sendo importantes para a concepção de aplicações distribuídas.

Para cada proposta selecionada, são levantadas as características mais importantes e uma aplicação produtor/consumidor com um *buffer* limitado é utilizada como base para avaliações. A partir do exercício de se implementar o *buffer* limitado com o suporte oferecido por cada proposta, uma comparação com os padrões determinados no capítulo 1 é realizada. Este exemplo tem sido utilizado como básico nos trabalhos originais destas propostas, o que facilita uma comparação imediata com nossa. Este mesmo exemplo também será, portanto, utilizado no capítulo 3 e 4 onde as comparações realizadas neste capítulo serão estendidas para envolver a nossa proposta.

2.2 Reflexão Computacional

A versão da aplicação produtor/consumidor programada com linguagens reflexivas utilizará um objeto-base que implementa um buffer genérico, sem limitação (podendo ter sua capacidade ultrapassada em certas circunstâncias) e sem sincronização no seu acesso. Os produtores e consumidores podem ser implementados por simples objetos.

O objetivo da analise é verificar as facilidades disponíveis para a concepção da aplicação e acréscimos de características que possam ser introduzidas no buffer, como a limitação e sincronização, através de meta-objetos, sem necessidade de se alterar o objeto-base. O aspecto de distribuição não será considerado nesta tecnologia pois este, como será concluído, depende de um suporte que deve ser provido à parte, onde questões de localização, obtenção de referências remotas e instanciação em nós distintos devem estar resolvidas.

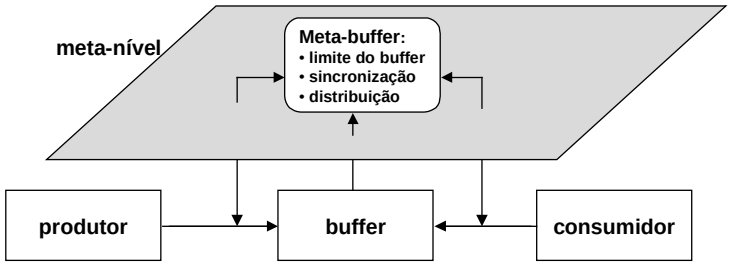


Figura 2.1 - Buffer Limitado por Reflexão

A figura 2.1 esquematiza a aplicação. Observa-se que apenas o buffer será controlado por um meta-objeto. A seta que parte do buffer para o meta-buffer indica o relacionamento entre objeto e meta-objeto e as linhas com dupla orientação indicam que as invocações que chegam ao buffer são interceptadas pelo meta-buffer e podem ser devolvidas ao buffer.

Para implementar esta aplicação, selecionamos o OpenC++ e MetaJava, duas linguagens orientadas a objeto com características reflexivas.

2.2.1 Open C++ (reflexão computacional estática)

Open C++ [Chi 95] é uma linguagem de programação, baseada em C++, que oferece reflexão em tempo de compilação. Open C++ oferece um MOP bastante completo,

permitindo que as aplicações se utilizem de reflexão estrutural e comportamental.

Além da reflexão, Open C++ utiliza o MOP para permitir que o programador controle a compilação, pois a arvore de *parsing* fica completamente disponível para o programador. O programador pode, então, escrever código que toma decisões não triviais a partir das informações disponíveis no meta-nível que não seriam factíveis para um otimizador de código. A partir daí, a sintaxe original de C++ pode ser estendida. Portanto, Open C++ também pode ser considerada uma linguagem aberta.

O código do buffer limitado da figura 2.2 foi desenvolvido a partir dos exemplos do manual do usuário da versão 2.0 [Chi 96].

O primeiro passo para se desenvolver uma aplicação em Open C++ é planejar e desenvolver os meta-objetos necessários. O MOP de OpenC++ oferece uma interface para se executar um método ***before*** associado a um outro método do objeto base. Por exemplo, sempre que o método *Put* for invocado, o método *before_Put* é executado antes do método Put. Em nosso exemplo, o método before_Put tem o objetivo de suspender a execução enquanto o buffer estiver cheio até um dado limite. Para simplificar a implementação, apenas o aspecto de limitação foi considerado. Assim, uma meta-classe *BoundedClass* é desenvolvida com dois objetivos: oferecer a característica de limitação ao *buffer* e estender a sintaxe do C++ permitindo que se declare um buffer limitado simplesmente acrescentando o lexema **bounded** na declaração da classe.

Na criação da meta-classe *BoundedClass* foram utilizados dois métodos do MOP:

- **TranslateMemberCall** que permite a reconstrução dos métodos originais da classe base (linhas 8, 9, 39 e 67);
- **BeforeFunction** que implementa o esquema de chamar um método *before* no objeto base associado a um outro método (linha 12);
- **Initialize** (linha 10) que, por sua vez chama a função **RegisterMetaclass** para instalar mais uma palavra reservada na linguagem (linha 17).

```
1    /*   Bounded extension */
2
3    #include "mop.h"
4
5    class BoundedClass : public Class {
6    public:
7        BoundedClass(Ptree* d, Ptree* m) : Class(d, m) {}
```

```
8       Ptree* TranslateMemberCall(Environment*, Ptree*, Ptree*, Ptree*, Ptree*);
9       Ptree* TranslateMemberCall(Environment*, Ptree*, Ptree*);
10      static BOOL Initialize();
11  private:
12      Ptree* BeforeFunction(Ptree* name);
13  };
14
15  BOOL BoundedClass::Initialize()
16  {
17      RegisterMetaclass("bounded", "BoundedClass");
18      return Class::Initialize();
19  }
20
21  class TempVarName : public Environment::ClientData {
22  public:
23      TempVarName(Ptree* p) { varname = p; }
24      Ptree* varname;
25  };
26
27  // BeforeFunction() receives a member NAME and looks for a before_NAME()
28  // member function.  If it finds, it returns before_NAME.
29
30  Ptree* BoundedClass::BeforeFunction(Ptree* name)
31  {
32      Ptree* before = Ptree::Make("before_%p", name);
33      if(IsMember(before) != 0)
34          return before;
35      else
36          return nil;
37  }
38
39  Ptree* BoundedClass::TranslateMemberCall(Environment* env, Ptree* object,
40                                           Ptree* op, Ptree* member,
41                                           Ptree* arglist)
42  {
43      Ptree* varname;
44      Ptree* before_func = BeforeFunction(member);
45
46      if(before_func == nil)
47          return Class::TranslateMemberCall(env, object, op, member, arglist);
48      else{
49          Ptree* class_name = Name();
50          TempVarName* tmpvar
51              = (TempVarName*)env->LookupClientData(this, class_name);
52          if(tmpvar != nil)
53              varname = tmpvar->varname;
54          else{
55              varname = Ptree::GenSym();
56              tmpvar = new TempVarName(varname);
57              Ptree* decl = Ptree::MakeStatement("%p* %p;",
58                                                 class_name, varname);
59              env->InsertDeclaration(decl, this, class_name, tmpvar);
60          }
61         return Ptree::Make("((%p=%c%p)->%p(), %p->%p%p)",
62                            varname, (op->Eq('.') ? '&' : ' '), object,
63                            before_func, varname, member, arglist);
64      }
65  }
66
67  Ptree* BoundedClass::TranslateMemberCall(Environment* env, Ptree* member,
68                                           Ptree* arglist)
69  {
70      Ptree* before = BeforeFunction(member);
71      Ptree* expr = Class::TranslateMemberCall(env, member, arglist);
72      if(before == nil)
73          return expr;
74      else
75          return Ptree::Make("(%p(), %p)", before, expr);
76  }
```

Figura 2.2 - Meta-objeto bounded em Open C++

A declaração desta meta-classe dá origem a um pré-processador executável. Este pré-processador consegue interpretar o código da aplicação descrita em OpenC++ que utiliza a reflexão e as extensões sintáticas recém-criadas. O resultado deste pré-processamento é a tradução do código, originalmente escrito em OpenC++, convertido para C++ com a funcionalidade equivalente aos objetos base e meta-objetos, que então pode ser compilado normalmente por um compilador C++.

No próximo trecho de código (figura 2.3) apresentamos a aplicação. Observa-se de início o conforto sintático para o programador da aplicação que pode introduzir facilmente na classe *Buffer* as características embutidas no meta-objeto. Observa-se, ainda, que ao se declarar a classe *Buffer* como sendo *bounded*, também declara-se os métodos *before_Put* e *before_Get* (linhas 6, 7, 19 e 31). Estes dois métodos podem ser considerados com implementações de guardas ou pré-condições para os métodos *Put* e *Get*, respectivamente. Para aumentar a modularidade a classe *Buffer* poderia ter sido concebida por herança a partir de uma classe mais genérica, e apenas os métodos before_Put e before_Get teriam sido acrescentados.

```
1   /* Buffer Limitado com reflexao */
2
3   bounded class Buffer {
4   public :
5      Buffer (int j) { max_itens = j; putPtr = 0; getPtr = 0; n_itens = 0; }
6      int Put(int value);    void before_Put();
7      int Get (int i);       void before_Get ();
8
9   private:
10     int putPtr; int getPtr; int n_itens; int max_itens; int itens[50];
11  };
12
13  int Buffer::Put(int value) {
14      printf("Buffer: armazenando %d\n ",value);
15      itens[putPtr] = value; putPtr = (putPtr + 1) % max_itens;  ++n_itens;
16      return (0);
17  }
18
19  void Buffer::before_Put() {
20      printf ("before_Put() esta sendo invocado. (n_itens = %d)", n_itens);
21      if (n_itens >= 5) puts("operacao suspensa: buffer esta cheio.");
22  }
23
24  int Buffer::Get(int i) {
25      int it;
26      printf("Buffer: retirando\n");
27      it = itens [getPtr]; getPtr = (getPtr + 1) % max_itens; --n_itens;
28      return (it);
29  }
30
31  void Buffer::before_Get() {
32      printf("before_Get() esta sendo invocado.(n_itens = %d)", n_itens);
33      if (n_itens <= 1) puts("operacao suspensa: buffer esta vazio.");
34  }
35
36  main()
37  {
38     Buffer BBuf(5);
```

```
39      BBuf.Put (10);  BBuf.Put (5);  BBuf.Put (32);  BBuf.Put (4);
40      BBuf.Put (6);   BBuf.Put (15); BBuf.Put (41);  BBuf.Put (57);
41      BBuf.Get (1);   BBuf.Get (2);
42   }
```

Figura 2.3 - Buffer Limitado em Open C++, utilizando extensão sintática

Na listagem da figura 2.4 é apresentado um trecho do código C++ resultante. Observa-se, por exemplo que uma simples invocação de método como `BBuf.Put (10)` é rescrita como `((_sym20184_1=&BBuf)->before_Put(), _sym20184_1->Put(10))`, forçando o método before_Put ser chamado antes do método Put.

```
1    main() {
2    # 1395 "buffer.ii"
3    Buffer* _sym20184_1;
4    # 88 "buffer.cc"
5    Buffer BBuf(10);
6    ((_sym20184_1=&BBuf)->before_Put(), _sym20184_1->Put ( 10 ) ) ;
7    ((_sym20184_1=&BBuf)->before_Put(), _sym20184_1->Put ( 5 ) ) ;
8    ...
9    ((_sym20184_1=&BBuf)->before_Get(), _sym20184_1->Get ( 1 ) ) ) ;
10   ((_sym20184_1=&BBuf)->before_Get(), _sym20184_1->Get ( 2 ) ) ) ;
11   # 101 "buffer.cc"
12   }
```

Figura 2.4 - Trecho do código C++ gerado a partir do buffer descrito em Open C++ e do pré-processador gerado com o meta-objeto Bounded

A reflexão em OpenC++ está limitada a classes. A ligação de uma classe-base e sua meta-classe só pode ser feita em tempo de compilação. Não há possibilidade de se estabelecer estes vínculos dinamicamente ou em relação a objetos ou métodos de um objeto. Também não é possível a utilização de classes reflexivas a partir de código pré-compilado, pois as referências à reificação são transformadas em código C++, perdendo-se as informações de meta-nível no código gerado. Por outro lado, o fato dos níveis base e meta darem origem a um único nível de código, evita o *overhead* do controle de um sistema executivo ou interpretador que teriam o papel de coordenar a execução dos dois níveis em tempo de execução.

2.2.2 MetaJava (reflexão computacional dinâmica)

MetaJava [Kle 96 e Gol 97T] é uma extensão da Máquina Virtual Java (JVM) que suporta reflexão estrutural e computacional, adicionando características de reflexão não encontradas em Java originalmente, e admite a criação e a ligação de meta-obetos com objetos-base dinamicamente. O modelo de objeto para os objetos de nível base é o modelo original de Java. No meta-nível novas características, como a persistência, replicação ou a transformação em objetos ativos, podem ser adicionadas aos objetos do

nível base associando-os à meta-objetos concebidos com estas finalidades.

A implementação de reflexão dinâmica em MetaJava foi possibilitada e facilitada porque:

- programas em Java executam sobre um interpretador - Java Virtual Machine (JVM) o que simplifica a extensão do ambiente de execução com características de reflexão mais dinâmicas;
- JVM oferece algumas características de reflexão estrutural (permite a identificação de classes, métodos e parâmetros destes métodos em um contexto diferente dos objetos criados a partir destas classes);
- JVM permite o controle de alguns eventos associados a objetos, como a carga destes objetos através de uma classe especial chamada *ClassLoader*.

As figuras 2.5 a 2.7 identificam o processo de criação da aplicação produtor/consumidor com buffer limitado através de reflexão utilizando MetaJava. Na figura 2.5 é apresentada a classe-base Buffer, implementada por um vetor grande (60 itens, para o nosso exemplo, pode ser considerado um buffer infinito - linha 4) e dois métodos: Put (linha 7) e Get (linha 13). Nada é dito a respeito da sincronização destes métodos.

```
1    public class Buffer {
2      private int putPtr = 0, getPtr = 0;
3      private int n_itens = 0;
4      private static final int max_itens=60;
5      private int itens[] = new int[max_itens];
6
7      public int Put(int value) {
8        itens[putPtr] = value;
9        putPtr = (putPtr + 1) % max_itens;   ++n_itens;
10       return 0;
11     }
12
13     public int Get (int i) {
14       int it = itens [getPtr];
15       getPtr = (getPtr + 1) % max_itens;  --n_itens;
16       return it;
17     }
18   }
```

Figura 2.5 - A classe base Buffer

Na figura 2.6 apresentamos a meta-classe MetaBuffer que herda as características da classe MetaObject definida pelo MOP de MetaJava (linha 3). A classe MetaObject possui métodos que são invocados quando o interpretador intercepta eventos que ocorrem em uma classe base. Observe que no construtor de MetaBuffer é estabelecida a

ligação entre o objeto base (passado como parâmetros para o construtor) e o meta-objeto (linhas 10 e 11). Em seguida, ainda no construtor, uma lista com os métodos cujas invocações devem ser interceptadas é registrada.

Sempre que um método com tipo de retorno ***int*** for invocada no método base, o método **eventMethodEnterInt** (linha 14) é invocado no meta-objeto, antes da invocação original chegar ao objeto-base. Na continuação, os métodos ***MetaPut*** ou ***MetaGet*** são ativados (linhas 16 e 17 respectivamente), em função do método do objeto-base interceptado. Estes métodos são sincronizados (em Java a referência ***synchonized*** em um método se encarrega disto) e a variável ***rn_itens***, juntamente com um esquema de sinais (guardas que não permitem operações em um buffer cheio ou vazio para os métodos de Put e Get, respectivamente) completam a implementação do buffer sincronizado e limitado. Em seguida, os métodos Put e Get do objeto-base são reativados pela função ***continueExecutionInt*** e prosseguem normalmente sua execução (linhas 25 e 34).

```
1    import meta.*;
2
3    public class MetaBuffer extends MetaObject{
4      private int rn_itens;
5      private int META_MAX = 10; // o novo limite para o Buffer
6
7      public MetaBuffer (Object obj, String MetaArgs []) {
8        rn_itens = 0;
9
10       attachObject(obj); // estabelece a ligação entre os níveis base e meta
11       registerEventMethodCall(obj , methodNotificationHandler, MetaArgs);
12     }
13
14     public int eventMethodEnterInt(Object o, EventDescMethodCall event) {
15       int ret;
16       if (event.methodname.compareTo ("Put")==0) ret = this.MetaPut (o, event);
17       else  ret = this.MetaGet (o, event);
18       return ret;
19     }
20
21     public  synchronized  int MetaPut (Object o, EventDescMethodCall event) {
22       while (rn_itens >= META_MAX)
23          try { wait (); } catch (InterruptedException e) {}
24
25       int ret = continueExecutionInt(o, event);  ++rn_itens;
26       notifyAll ();
27       return ret;
28     }
29
30     public synchronized  int MetaGet (Object o, EventDescMethodCall event) {
31       while (rn_itens <= 0)
32          try { wait (); } catch (InterruptedException e) {}
33
34       int ret = continueExecutionInt(o, event);  --rn_itens;
35       notifyAll ();
36       return ret;
37     }
38   }
```

Figura 2.6 - O meta-objeto Buffer Limitado e Sincronizado

A figura 2.7 apresenta a aplicação. A separação de aspectos se torna evidente pois somente na implementação da aplicação, o programador instancia um objeto da classe Buffer e depois estabelece um vínculo com uma instância da classe MetaBuffer dotando o objeto-base com as outras características. Os aspectos de sincronização e limite do *buffer* são separados em relação a implementação do *buffer* propriamente dito. O vínculo é estabelecido, instanciando-se um objeto MetaBuffer (***new MetaBuffer*** - linha 17) e passando-se como parâmetros o nome do objeto base e a lista de métodos a serem reificados.

Neste exemplo, vários produtores e consumidores são instanciados (e implementados como *threads*) para comprovar a funcionalidade do buffer limitado em situações de concorrência.

```
1    /**
2     * Usando reificação das chamadas de métodos
3     */
4    public class BoundBuffer {
5      public static void main(String argv[]) {
6        /* Instancia os objetos classe-base*/
7        Buffer Buffer  = new Buffer();
8        Producer prod1 = new Producer (Buffer, 1);
9        Consumer cons1 = new Consumer (Buffer, 1);
10       Consumer cons2 = new Consumer (Buffer, 2);
11       Producer prod2 = new Producer (Buffer, 2);
12
13       String MetaArgs [] = new String[2];   // métodos reificados
14       MetaArgs[0] = "Put(I)I";
15       MetaArgs[1] = "Get(I)I";
16
17       new MetaBuffer(Buffer, MetaArgs); // Instancia o meta-objeto MetaBuffer
18
19       /* Ativa as threads nos produtores e consumidores */
20       prod1.start();  cons1.start();  cons2.start();  prod2.start();
21     }
22   }
```

Figura 2.7 - Aplicação Buffer Limitado

Um exemplo na mesma linha foi proposto em [Oli 98] com algumas vantagens: o desenvolvedor do nível base da aplicação não precisa indicar o que vai ser refletido para o meta-nível. Para isso ser possível, o meta nível tem acesso irrestrito ao nível base, mas pode impor um esquema de segurança para que a aplicação não possa ultrapassar certos limites no uso de recursos [Gow 95]. Neste caso, no meta-nível é possível a configuração dos meta-objetos utilizados em tempo de execução.

2.3 Filtros de Composição (*Composition-Filter*)

O modelo de Filtros de Composição (*CF*) [Ber 96T] apresenta alguns conceitos que têm

interseção com a proposta de nosso trabalho. CF é baseado em uma extensão do modelo tradicional de objetos. Uma característica desta extensão é a imposição da separação explícita da especificação da interface do objeto e seu comportamento.

O elemento central neste modelo é composto de um núcleo de objeto e uma interface. O núcleo possui a semântica de um objeto tradicional, baseado em protótipos. A interface gerencia o acesso ao núcleo do objeto. A comunicação entre objetos é sempre feita através de mensagens simples do tipo request/reply e estas são tratadas como objetos de primeira-classe. As mensagens carregam informações que atuam como seletores do comportamento dos objetos.

A parte externamente visível da interface de um objeto é completamente controlada por filtros. Estes filtros podem inspecionar, manipular as mensagens de entrada a saída dos objetos e reagir de acordo com o conteúdo das mesmas e com o tipo do filtro, oferecendo assim características reflexivas (limitadas) ao modelo, no sentido em que estas são refletidas para o filtro antes de seguirem para o núcleo. As decisões tomadas pelos filtros também consideram variáveis de condição, visíveis na interface, que expõem informações internas ao núcleo do objeto na forma de expressões booleanas. O esquema de filtros permite que aspectos não-funcionais ou operacionais sejam adicionados ao comportamento original do objeto de uma forma dinâmica.

Uma aplicação é programada pela composição de objetos. A composição é especificada declarando-se na interface de um objeto, referências à instâncias de outros objetos, chamados de objetos internos. O relacionamento entre um objeto e seus objetos internos é ditada pela programação dos filtros. Este modelo oferece a possibilidade de se utilizar mecanismos de herança e delegação, e ainda permite que estas relações de herança e delegação entre objetos sejam dinamicamente habilitadas e desabilitadas. A configuração desta relação é obtida através da programação adequada de filtros.

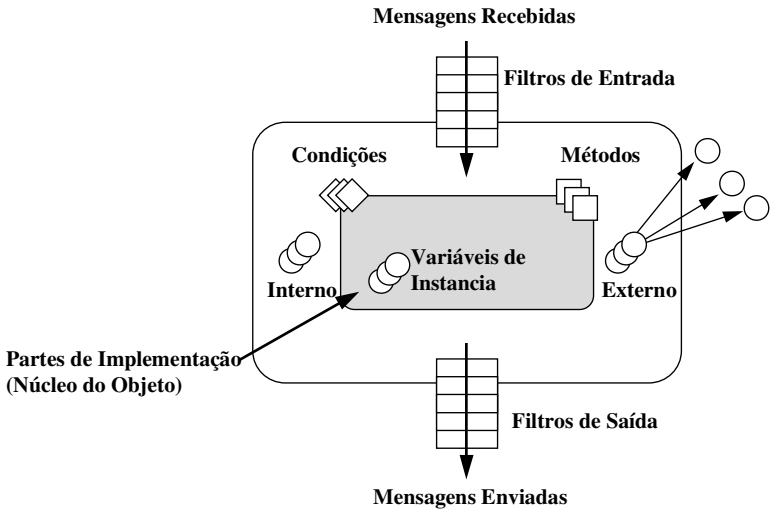


Figura 2.8 - um objeto em CF

Com este esquema, segundo os autores, problemas de domínios diferentes podem ser resolvidos através da composição e programação de filtros. A composição de objetos com a habilidade de resolver estes problemas pode ser feita de forma ortogonal e sem efeitos colaterais entre os diversos domínios de problemas. Entre os domínios de problemas estudados com o modelo estão tempo-real, programação concorrente e especificação de restrições de sincronismo, reusabilidade e extensibilidade.

A existência dos filtros para gerenciar o acesso ao núcleo do objeto implica em uma forma de evolução e dinamismo no comportamento do objeto. A atuação dos filtros é dependente do estado atual do objeto. Portanto, clientes acionando um método de determinado objeto podem obter resultados diferentes, associados à evolução do estado do mesmo. Existem filtros de tipos diferentes. Cada filtro pode executar algumas ações padronizadas, como rejeitar uma mensagem ou repassar uma mensagens sem nenhum processamento, e ações especiais relacionados com o tipo, como por exemplo travar o repasse de uma mensagem até que determinada condição seja satisfeita no estado do objeto.

A herança é utilizada em CF para oferecer extensibilidade. Os filtros, entretanto, são especificados separadamente e não considerados na herança. Apenas a assinatura do objeto (métodos e variáveis declaradas) é utilizada para implementar a herança. Estas características, segundo os autores [Ber 96T], isentam o modelo do problema da anomalia da herança. Por exemplo, no que diz respeito a coordenação, toda a

especificação de sincronização é feita através dos filtros. Como estes são separados da interface do objeto, a informação de sincronização não é propagada.

Sina [Sch 96] é uma linguagem orientada a objetos que implementa os conceitos de CF. A implementação de filtros é interna à linguagem, que oferece alguns tipos de filtros, cujo comportamento deve ser conhecido para que estes sejam utilizados segundo a semântica desejada. Na versão analisada, três filtros estão disponíveis: *Dispatch* – apenas encaminha a mensagem, *Wait* – utilizado para sincronização dos métodos de um objeto (funcionando de forma semelhante a um guarda), *Error* – rejeita mensagens que não atendem determinadas condições. Existe também uma seqüência de ações que podem ser programadas em cada filtro, de forma que mesmo o filtro mais simples (neste caso, o *Dispatch*) pode manipular as mensagens com alguma liberdade. Além disso, um filtro pode estar programado para encaminhar as mensagens para um outro filtro, ao invés de encaminhá-las para o núcleo do objeto.

O próximo exemplo implementa o Buffer Limitado utilizando Sina, onde pode ser observado o estilo de sua sintaxe.

```
1    class BoundedBuffer (limit: Integer) interface
2           comment "Objeto - Buffer limitado - sincronizado e com guardas";
3           conditions
4               Full, Empty, Partial;
5           methods
6               put (Any) returns Nil;
7               get returns Any;
8           inputfilters
9               BufferSync : Wait = {{Empty, Partial}=>put, {partial, Full}=>get };
10              disp : Dispatch = {inner.*};
11              ;
12          end // class BoundedBuffer interface
13
14   class BoundedBuffer implementation
15          instvars
16              store: Array (limit);
17              hear, tail: Integer;
18          conditions
19              Empty begin return head=tail end;
20              Partial begin return (inner.Empty.not and inner.Full.not) end;
21              Full begin return (limit+tail-head).mod(limit)=1; end;
22          initial
23              begin head :=  1; tail := 1; end;
24          methods
25              put (elem: Any) returns Nil
26                 begin store.atPut(head. elem); head := head.mod(limit)+1; end;
27              get returns Any
28                 begin return store.at(tail); tail := tail.mod(limit) + 1; end;
29   end
```

Figura 2.9 - Buffer Limitado em Sina

Neste exemplo foram utilizados os filtros *Dispatch* e *Wait* (linhas 9 e 10). Devido a

simplicidade do exemplo, não foi necessária a utilização dos mecanismos de relacionamento entre objetos.

Observa-se que Sina obriga o programador descrever explicitamente a interface de uma classe (linhas de 1 a 12), o que facilita a reutilização. Sina parece ser adequada para expressar aspectos de coordenação. Entretanto, o relacionamento de delegação ou herança entre classes precisa estar embutido na interface, através da declaração de filtros do tipo Dispatch o que pode não permitir que este objeto seja reutilizado em muitas situações.

Um filtro do tipo Meta também está disponível em Sina. O comportamento do filtro Meta é similar ao Dispatch. Entretanto, se a mensagem recebida é aceita pelo filtro Meta, a mensagem é reificada como um objeto da classe *Message* e passada como parâmetro para um objeto da classe especial *ACT*. Este objeto pode investigar ou modificar o conteúdo da mensagem original e em seguida converter o objeto *Message* de volta para a execução da mensagem. O filtro *Meta* dota a linguagem Sina, assim, com características de reflexão, através da interceptação de mensagens, limitadas a invocações à métodos.

Outros aspectos como tempo-real foram adaptados ao modelo de CF [Aks 92]. Não foi possível avaliar, entretanto, se este modelo permite a integração de outros aspectos. Nas referencias consultadas apenas os filtros Dispatch, Wait, Error e Send estavam disponíveis sendo que dois deles foram utilizados no exemplo. A inclusão de novos filtros requer, a principio, a modificação da linguagem.

2.4 AOP - *Aspect Oriented Programming*

A Programação Orientada a Aspectos (AOP - *Aspect-Oriented Programming*) é um paradigma recentemente proposto [Kic 97] como solução para a concepção de sistemas complexos, que utiliza uma extensão do conceito de modularidade. AOP considera que, além da decomposição em módulos, uma aplicação pode ser concebida mais facilmente se decomposta em seus aspectos funcionais, não-funcionais e operacionais. Neste contexto, um módulo deve encapsular, então, apenas um aspecto do sistema. Cada aspecto da uma aplicação é tratado e programado separadamente. Em uma etapa seguinte, os programas relativos aos diversos aspectos são combinados para gerar um

único programa final.

A decomposição de um sistema em aspectos é ortogonal à decomposição em módulos. Considerando, por exemplo, sistemas distribuídos, determinado problema pode ser decomposto em funções básicas (que abrangeriam os aspectos funcionais) e aspectos não-funcionais ou operacionais de localização, comunicação e sincronização. Cada um dos aspectos deve ser descrito separadamente, mas o relacionamento entre eles deve estar claramente identificado. No modelo tradicional de modularidade, os vários módulos componentes do sistema descreveriam os aspectos funcionais e trariam embutidos os aspectos não funcionais ou operacionais, todos entrelaçados.

A descrição dos aspectos funcionais e não-funcionais de uma aplicação poderá ser feita em linguagens de programação diferentes ou por notações criadas especificamente com a finalidade de descrever um determinado aspecto. Uma vez que todos os aspectos de um sistema estejam descritos e a inter-relação entre estes aspectos esteja claramente visível, uma ferramenta chamada "entrelaçador de aspectos" (chamado de *Aspect Weaver* pelos autores de AOP) é responsável por combinar e misturar todos os aspectos em uma forma executável. Esta forma executável poderá apresentar um código emaranhado, mas eficiente, dependendo da complexidade do problema e do "entrelaçador". Observa-se que o entrelaçador deve ser capaz de receber as informações descrevendo os aspectos e gerar como resultado alguma representação da aplicação que possa ser executada.

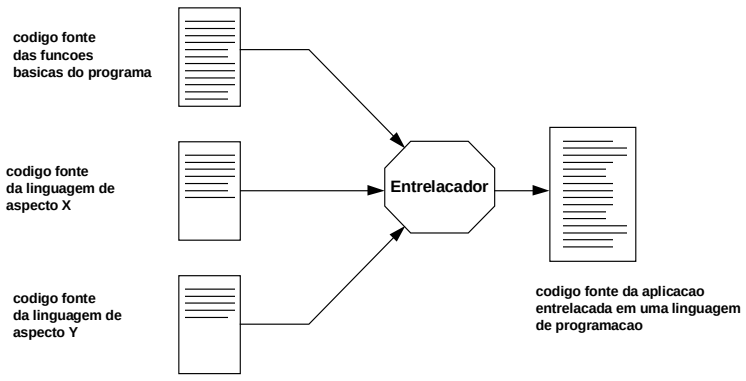


Figura 2.10 - AOP e o Entrelaçador (*Weaver*)

Vantagens imediatas de AOP, além da separação de aspectos, são a legibilidade de código (antes do entrelaçamento) e a potencialidade de reutilização de módulos dos

aspectos funcionais básicos, bem como outros aspectos não-funcionais.

Um exemplo de implementação de em AOP, “D”, é descrito em [Lop 97]. D é um conjunto de ferramentas orientadas a objeto que utiliza AOP para permitir a concepção de sistemas distribuídos. Em D, a funcionalidade básica de uma aplicação distribuída é descrita através de uma linguagem orientada a objetos, chamada de JCore (um subconjunto de Java). Aspectos de coordenação e sincronização são descritos através de uma linguagem chamada Cool o mesmo acontecendo com os aspectos de distribuição que são descritos com Ridl. Tanto Cool como Ridl são simples e podem ser classificadas como linguagens declarativas.

Um outro componente de D, o “*Aspect Weaver*” combina as descrições em JCore, Cool e Ridl e produz código Java que pode ser compilado e gerar um programa executável contemplando o comportamento especificado nos aspectos não-funcionais.

A seguir é apresentado o exemplo do Buffer Limitado em D (adaptado de [Lop 97T]) implementado para a primeira versão de AspectJ [AspJ 98B], versão veiculada pela Xerox, onde o projeto de AOP está atualmente hospedado. Na figura 2.11 é programado a classe BoundedBuffer em JCore. Não existe no código nenhuma referência à limitação do *buffer* nem ao aspecto de sincronização. Entretanto, é necessária a contagem da ocupação do *buffer* (variável **`usedSlots`**, linha 3) para permitir que a linguagem de coordenação obtenha esta informação a partir da classe.

```
1     public class BoundedBuffer {
2           protected int putPtr = O, takePtr = 0;
3           int usedSlots = 0; int size;
4           private Object array [];
5
6           public BoundedBuffer(int capacity) {
7               size = capacity;
8               array = new Object[capacity];
9           }
10          public void put(Object o) {
11              array[putPtr] = o;
12              putPtr = (putPtr + 1) % array.length;
13              usedSlots++;
14          }
15          public Object take() {
16              Object o = array[takePtr];
17              takePtr = (takePtr + 1) % array.length;
18              usedSlots--;
19              return o;
20          }
21    }
```

Figura 2.11 - Buf.jcore: Buffer em JCore

Na figura 2.12 a coordenação do *buffer* (limitação e sincronização) é programada. O

aspecto de limitação é obtido através de guardas (linhas 6 e 12) que monitoram o estado do *buffer* e somente permitem a execução dos métodos sob sua responsabilidade (*put* e *get*) se determinadas condições forem satisfeitas. Na saída, as condições monitoradas são atualizadas. A sincronização é obtida indicando-se os métodos que serão executados em exclusão mútua (linhas 2 e 3), segundo um esquema de monitor.

```
1    coordinator BoundedBuffer {
2       selfex put, take;
3       mutex {put, take};
4       condition empty = true;
5       condition full  = false;
6       guard put:
7            requires (!full);
8            onexit {
9              empty = false;
10             if (usedSlots == size) full = true;
11           }
12      guard take:
13           requires (!empty);
14           onexit {
15             full = false;
16             if (usedSlots == 0) empty = true;
17           }
18    }
```

Figura 2.12 - Buf.cool: Aspectos de Coordenação e Sincronização em Cool

Comparando-se o código apresentado nas figuras 2.11 e 2.12 observa-se que os aspectos de coordenação estão desvinculados dos aspectos funcionais da aplicação. Neste exemplo apenas os aspectos de coordenação foram contemplados. A programação do aspecto de distribuição de elementos não estava disponível na ferramenta utilizada. Embora este aspecto seja essencial para a concepção de sistemas distribuídos, não é o único aspecto presente nesta categoria. Por exemplo, aspectos específicos de comunicação podem ser necessários. Assim sendo, as linguagens que permitem a descrição de aspectos devem ser ampliadas ou novas linguagens de descrição de aspectos devem ser oferecidas para que o programador possa indicar estilos e protocolos de comunicação. Neste caso, o *aspect weaver* também precisa absorver estas extensões. A seguir é apresentado um trecho da aplicação entrelaçada em código Java em que pode ser observado que o método *Put* passa a conter código para sincronização e limitação do *buffer* (linha 26, por exemplo).

```
1    // Woven output code produced by AspectJ Weaver v0.1.0.
2    //
3    // Do not edit this code!
4    // All edits will be lost when the weaver is run again.
5
6    // from file "Buf.jcore"
7    // starting at line 1
8    public class BoundedBuffer {
9       protected int putPtr = 0, takePtr = 0;
```

```
10      int usedSlots = 0;
11      int size;
12      private Object array[];
13      public BoundedBuffer (int capacity){
14         size = capacity;
15         array = new Object[capacity];
16      }
17
18     // produced by "Cool Aspect Weaver"
19     // from file "Buf.cool"
20     // starting at line 1
21
22     public
23     // from file "Buf.jcore"
24     // starting at line 10
25     void put(Object o){
26        synchronized(getLock()){
27           while (!(selfex0.testEx("BoundedBuffer.put") &&
28                    mutex2.testEx("BoundedBuffer.put") &&
29                  (!
30            // from file "Buf.cool"
31            // starting at line 9
32                       full)))
33           {
34              try { getLock().wait(); }
35              catch(InterruptedException e){ }
36           }
37           selfex0.enterEx("BoundedBuffer.put");
38           mutex2.enterEx("BoundedBuffer.put");
39           getLock().notifyAll();
40        }
41        try{
42           innerput(
43           // from file "Buf.jcore"
44           // starting at line 10
45           o);
46        }
47
48        finally {
49           synchronized(getLock()){
50
51           // from file "Buf.cool"
52           // starting at line 11
53              empty = false;
54              if(usedSlots == size){ full = true; };
55              ;
56              selfex0.exitEx("BoundedBuffer.put");
57              mutex2.exitEx("BoundedBuffer.put");
58              getLock().notifyAll();
59           }
60        }
61     }
62   ...
```

Figura 2.13 - Buf.java: Código Java gerado por AspectJ

Uma das possibilidades para permitir a extensão de ferramentas como AspectJ é o uso de reflexão para tornar as linguagens de aspectos abertas para novos acréscimos. Alternativamente, sem a necessidade de uma linguagem aberta, estamos propondo a utilização de configuração com propriedades equivalentes a AOP, como será apresentado no capítulo 3.

2.5 Configuração

Além das tecnologias avaliadas no capítulo 1 e das propostas avaliadas na primeira parte deste capítulo, são apresentadas nesta seção algumas propostas que consideram a descrição da estrutura ou arquitetura de aplicações distribuídas, independentemente da implementação, como sendo uma etapa importante na construção destes sistemas. Consideramos o termo configuração adequado para tratar de linguagens ou ambientes que permitem a descrição de sistemas explicitando seus componentes e a forma com que são interconectados. Entretanto, como mencionado na seção 1.3.3, alguns autores classificam os ambientes de configuração levando em consideração outras características como a linguagem de configuração, aspectos formais e o nível de abstração.

Robert Allen em sua tese [All 97T] apresenta uma compilação sobre linguagens de configuração sob o ponto de vista de aspectos formais e sob o conceito da arquitetura de aplicações. Neste trabalho, as linguagens são classificadas de acordo com os seguintes requisitos: (1) capacidade de descrever configurações de arquiteturas, (2) descrição de estilos de arquitetura, (3) análise de propriedades de interesse e (4) aplicação prática em sistemas reais. As linguagens avaliadas neste trabalho são divididas em 2 grupos: ADLs, em um deles e PLs, MILs e IDLs em outro. A grande diferença entre os dois grupos, no contexto deste trabalho, é que as ADLs devem suportar o conceito de arquitetura e o outro grupo possui apenas características que facilitam, de alguma forma, a descrição da estrutura de aplicações mas não contemplam diretamente o conceito de arquitetura de software.

Em [Bis 96], Judi Bishop e Roberto Faria classificam linguagens de configuração em estágios de abstração e de acordo com a abordagem utilizada pelas mesmas para conectores. Neste trabalho Linguagens de Programação como C++ e Ada, por exemplo, são classificadas em um estágio inicial e ADLs como UniCon e Wright estariam no estágio final, oferecendo alto nível de abstração onde uma aplicação pode ser concebida em termos da sua arquitetura.

Cada um destes trabalhos e outros que não mencionamos diretamente (como [Agh 93, All 94, Luc 95 e Med 97] acrescentam informações importantes ao entendimento dos conceitos de configuração que serão utilizados em nossa proposta. A seguir, apresentamos uma avaliação e uma implementação da aplicação produtor/consumidor com *buffer* limitado sobre dois projetos na área arquiteturas de software: Regis, baseado

em uma linguagem e um ambiente especial de execução com suporte à configuração, e UniCon, que possui um compilador para a linguagem de configuração de mesmo nome que gera código executável a partir de módulos e descrições de arquiteturas. Além destes, outros projetos importantes como Wright [All 94], que se concentra em oferecer uma base formal para especificação de interações de componentes através de conectores e estilos de arquiteturas e Rapide [Luc 95], que enfatiza a especificação do comportamento e a simulação de arquiteturas de software para gerar refinamentos sobre estas, podem ser mencionados.

2.5.1 Regis

Regis [Reg 95, Mag 96 e Kra 90] é um ambiente desenvolvido no Imperial College of London que utiliza configuração para descrever aplicações em termos de seus componentes e a interligação dos mesmos. Regis possui as seguintes características:

- separação entre computação e comunicação de componentes;
- suporta programas com estruturas dinâmicas;
- primitivas de configuração para instanciação retardada e dinâmica de componentes;
- o acesso ao sistema operacional nativo é feito através de um sistema executivo especializado;
- componentes podem interagir através de primitivas de comunicação especificadas pelo usuário.

Uma aplicação em Regis é descrita através da linguagem Darwin [Dar 97]. Darwin é uma linguagem descritiva que permite a estruturação hierárquica de componentes, facilitando a concepção de sistemas com grande número de componentes. Componentes em Darwin se referem as objetos que realizam a computação da aplicação e objetos que são utilizados para a comunicação de outros componentes. Um componente em Darwin se comunica com outros componentes através de *portas* que podem ser providas para outros componentes ou requeridas por outros componentes. As portas podem ser de entrada ou saída. A seguir apresenta-se a aplicação produtor/consumidor com *buffer* limitado, implementada em Darwin, adaptada de [Reg 95].

```
1    component Buffer (int max) {
2          provide
```

```
3               GetReq <port " portref<int> ">;
4               Put    <port " portref<int> ">;
5           require
6               OK     <port " portref<int> ">;
7               GetRep <port " portref<int> ">;}
8
9   component Producer  {
10          require
11              Put    <port " portref<int> ">;
12          provide
13              OK     <port " portref<int> ">;}
14
15  component Consumer  {
16          require
17              GetReq    <port " portref<int> ">;
18          provide
19              GetRep    <port " portref<int> ">;}
20
21  component BB (int n) {
22          inst B: Buffer (n);
23          inst P: Producer;
24          inst C: Consumer;
25
26          bind
27              P.Put -- B.Put;
28              B.OK -- P.OK;
29              C.GetReq -- B.GetReq;
30              B.GetRep -- C.GetRep;}
```

Figura 2.14 - Buffer Limitado descrito em Darwin

Observa-se que Darwin não explicita conectores. O símbolo "--" (linhas 26 a 30) é usado para indicar que um componente está ligando sua porta de saída – *required* a uma porta de entrada - *provided* por outro componente. As características da comunicação das portas está implícita no código do componente. Regis oferece uma biblioteca de classes C++ que, juntamente com o código gerado a partir de uma descrição em Darwin, implementam as abstrações, como as portas, e a associação destas com mecanismos de comunicação. Por exemplo, se as portas tivessem que ser implementadas por *sockets* para permitir a comunicação de componentes distribuídos isso seria passado como um parâmetro para o gerador de código. A seguinte alteração seria feita nas portas do *buffer* do exemplo em Darwin:

```
provides Put <port "portref <int>" >;
```

O código gerado relacionado com as portas conduz o programador ao uso de primitivas do tipo ***send*** e ***receive*** que são operadas como métodos de um objeto do tipo port.

A partir da descrição em Darwin um gerador de código cria alguns arquivos com a descrição das classes necessárias em C++ e código, incluindo as chamadas ao suporte para a abstração de portas. A figura 2.15 apresenta um trecho do código em C++ gerado a partir da configuração descrita em Darwin.

```
1   #include <dw.h>
2
3   class Buffer : public process {
4   public:
5          port< portref<int> > GetReq;
6          port< portref<int> > Put;
7          portref< portref<int> > OK;
8          portref< portref<int> > GetRep;
9          Buffer(int max);
10  };
11
12  ...
13
14  #ifndef DWCLEAN
15  static void Buffer_IN(char *dw_buf);
16
17  void _Buffer(int dw_proc, refIF dw_ref, int max) {
18         int dw_len;
19         if (dw_proc & WORKINST) {
20             faceP SELF = work_if(Buffer_IN, 4, 0, 0, 0, "pprr"
21                ");
22             dw_send(dw_ref, SELF);
23             return;
24         }
25         if (dw_proc & LAZYINST) {
26             faceP SELF = lazy_if(4, 0, 0, 0, "pprr"
27                ");
28             SELF -> inst.call=mpack(0, dw_len, "iiri", dw_proc & ~LAZYINST,
29                Buffer_IN, &dw_ref, max);
30             dw_send(dw_ref, SELF);
31             return;
32         }
33         if (dw_proc != thisnode) {
34             char *dw_buf=mpack(0, dw_len, "iiri", dw_proc, Buffer_IN,
35                &dw_ref, max);
36             rpc_call(dw_proc, dw_buf, dw_len);
37             free(dw_buf);
38             return;
39         }
40
41         refIF dw_r = dw_ref;
42         rtprocname="Buffer";
43         Buffer *DW = new Buffer(max);
44         faceP SELF = cc_if((void *) DW, 4, 0, 0, 0);
45         cc_provide (SELF, 1, &DW->GetReq);
46         cc_provide (SELF, 2, &DW->Put);
47         cc_require (SELF, 3, &DW->OK);
48         cc_require (SELF, 4, &DW->GetRep);
49         dw_send(dw_r, SELF);
50  }
51  #endif
52  /// fim de bbuffer.h
53
54  /// inicio bbuffer.cd
55
56  #include <stdlib.h>
57  #include <dw.h>
58  static char *DWX="already instantiated";
59
60  extern void _Buffer(int dw_proc, refIF dw_ref, int max);
61  extern void _Producer(int dw_proc, refIF dw_ref);
62  extern void _Consumer(int dw_proc, refIF dw_ref);
63
64  class BB : public process {
65  public:
66         BB(refIF *dw_ref, int n);
67  };
68
69  #ifndef DWCLEAN
70  static void BB_IN(char *dw_buf);
71
72  void _BB(int dw_proc, refIF dw_ref, int n) {
73         int dw_len;
```

```
74          if (dw_proc & WORKINST) {
75              faceP SELF = work_if(BB_IN, 0, 0, 0, 0, "");
76              dw_send(dw_ref, SELF);
77              return;
78          }
79          if (dw_proc & LAZYINST) {
80              faceP SELF = lazy_if(0, 0, 0, 0, "");
81              SELF -> inst.call=mpack(0, dw_len, "iiri", dw_proc & ~LAZYINST,
82                 BB_IN, &dw_ref, n);
83              dw_send(dw_ref, SELF);
84              return;
85          }
86          if (dw_proc != thisnode) {
87              char *dw_buf=mpack(0, dw_len, "iiri", dw_proc, BB_IN,
88                 &dw_ref, n);
89              rpc_call(dw_proc, dw_buf, dw_len);
90              free(dw_buf);
91              return;
92          }
93
94          refIF *dw_ref2 = new refIF;
95          *dw_ref2 = dw_ref;
96          rtprocname="BB";
97          (void) new BB(dw_ref2, n);
98    }
99
100
101   // ------------------------------------------------------------------------
102
103
104   main(int argc, char *argv[]) {
105     if (dw_start(argc, argv, 0)) {
106       faceP SELF;
107       portIF mn_port;
108       refIF mn_ref(mn_port);
109
110       if (--argc < 1) dw_fatal("More args needed");
111       switch (argc) {
112       default: _BB(0, mn_ref, atoi(argv[1]));
113       }
114       mn_port.in(SELF);
115       free(SELF);
116     }
117     dw_end();
118   }
119
120
```

Figura 2.15 - Arquivo em C++ gerado a partir da aplicação descrita em Darwin

No contexto de Darwin, a adição de características não-funcionais não é possível no nível da configuração, a menos que componentes específicos com esta finalidade sejam concebidos. Neste caso, componentes implementando aspectos não-funcionais não serão distinguidos de componentes implementando aspectos funcionais. Por exemplo, na configuração da figura 2.14 nada pode ser dito sobre a sincronização das portas do componente *Buffer*. O tamanho do *Buffer* foi passado como parâmetro e este precisou ser concebido para prever o limite. A solução para a sincronização pode seguir a proposta apresentada nos exemplo da seção do capítulo 4.

2.5.2 UniCon

UniCon [Sha 95] é um projeto do Departamento de Ciências da Computação da Universidade de Carnegie Mellon baseado em uma ADL (que também é chamada de UniCon). O destaque em UniCon é a utilização explícita de conectores. UniCon oferece uma ferramenta para construção de aplicações executáveis a partir da descrição de suas arquiteturas. UniCon é baseado no conceito de tipos e implementações de componentes, e conectores que contém a "inteligência de interligação". Os conectores ligam componentes uns aos outros segundo o comportamento associado a esta "inteligência". Conectores de UniCon são importantes nas arquiteturas de software onde a maneira de como a interação entre os componentes se dará, seja relevantes. Nós acreditamos que a declaração explícita do estilo de interconexão de componentes facilita sob vários aspectos a concepção de sistemas distribuídos.

Em UniCon a descrição de componentes e conectores deve ser explicitada em detalhes (linha 45, por exemplo). Informações completas sobre o tipo, a interface e a implementação devem ser explicitamente colocadas na declaração de cada elemento.

```
1   COMPONENT Buffer
2     INTERFACE IS
3       TYPE Module
4       PLAYER Get IS RoutineDef
5         SIGNATURE ("char * *"; "void")
6       END Get
7       PLAYER Put IS RoutineDef
8         SIGNATURE ("char *"; "void")
9       END Put
10    END INTERFACE
11    IMPLEMENTATION IS
12      VARIANT Buffer IN "buffer.c"
13        IMPLTYPE (Source)
14      END Buffer
15    END IMPLEMENTATION
16  END Buffer
17
18  COMPONENT Producer
19    INTERFACE IS
20      TYPE Module
21      PLAYER Put  IS RoutineCall
22        SIGNATURE ("char *";"void")
23      END Put
24    END INTERFACE
25    IMPLEMENTATION IS
26      VARIANT Producer IN "prod.c"
27        IMPLTYPE (Source)
28      END Producer
29    END IMPLEMENTATION
30  END Producer
31
32  COMPONENT Consumer
33    INTERFACE IS
34      TYPE Module
35      PLAYER Get  IS RoutineCall
36        SIGNATURE ("char * *"; "void")
37      END Get
38    END INTERFACE
39
40
```

```
41      IMPLEMENTATION IS
42        VARIANT Consumer IN "cons.c"
43          IMPLTYPE (Source)
44        END Consumer
45      END IMPLEMENTATION
46    END Consumer
47
48    CONNECTOR C-proc-call
49      PROTOCOL IS
50        TYPE ProcedureCall
51          ROLE definer IS definer
52          ROLE caller IS caller
53      END PROTOCOL
54      IMPLEMENTATION IS
55        BUILTIN
56      END IMPLEMENTATION
57    END C-proc-call
58
59    COMPONENT BoundedBuffer
60      INTERFACE IS
61        TYPE Filter
62      END INTERFACE
63      IMPLEMENTATION IS
64        USES Prod INTERFACE Producer
65        USES Cons INTERFACE Consumer
66        USES Buff INTERFACE Buffer
67        USES put-prod PROTOCOL C-proc-call
68        USES get-prod PROTOCOL C-proc-call
69        CONNECT Prod.Put TO put-prod.caller
70        CONNECT Buff.Put TO put-prod.definer
71        CONNECT Cons.Get TO get-prod.caller
          CONNECT Buff.Get TO get-prod.definer
      END IMPLEMENTATION
    END BoundedBuffer
```

Figura 2.16 - Buffer Limitado descrito em UniCon (uni-buf.uni)

Uma ferramenta (chamada *uparse*, na versão da implementação testada) faz uma análise sintática e semântica da configuração descrita. Se o conector utiliza algum tipo de inteligência especial, o analisador deve ser preparado para suportar as características adicionais [Zel 97].

```
1   Checking syntax...
2
3   -> Component 'Buffer' syntactically correct
4   -> Component 'Producer' syntactically correct
5   -> Component 'Consumer' syntactically correct
6   -> Connector 'C-proc-call' syntactically correct
7   -> Component 'BoundedBuffer' syntactically correct
8
9   Synthesizing attributes...
10
11  Checking semantics...
12
13  -> Component 'Buffer' semantically correct
14  -> Component 'Producer' semantically correct
15  -> Component 'Consumer' semantically correct
16  -> Connector 'C-proc-call' semantically correct
17  -> Component 'BoundedBuffer' (locally) semantically correct
18
19  Checking global semantics for component 'BoundedBuffer'...
20
21  Checking connections for component 'BoundedBuffer'...
22
23  The following system was analyzed:
24
```

```
25          BoundedBuffer
```

Figura 2.17 - Saída do verificador da sintaxe e coerência da arquitetura descrita em UniCon

Se o projetista desejar, uma analise mais detalhada de propriedades também pode ser feita.

```
1   Component Type BoundedBuffer Extends UniCon-filter With {
2     Representation {
3       System BoundedBuffer = {
4         Component Prod : UniCon-module = new Producer;
5         Component Cons : UniCon-module = new Consumer;
6         Component Buff : UniCon-module = new Buffer;
7         Connector put-prod : UniCon-procedurecall = new C-proc-call;
8         Connector get-prod : UniCon-procedurecall = new C-proc-call;
9         attachments {
10          Prod.Put to put-prod.caller;
11          Buff.Put to put-prod.definer;
12          Cons.Get to get-prod.caller;
13          Buff.Get to get-prod.definer;
14        };
15      };
16    };
17  };
18
19  System UniCon-Top-Level = {
20    Component BoundedBuffer : UniCon-filter = new BoundedBuffer;
21  };
```

Figura 2.18 - trecho da listagem de propriedades da arquitetura descrita em UniCon gerado pelo uparse

A versão de UniCon avaliada apresenta por volta de 7 tipos de conectores embutidos na linguagem, dentre eles *pipe*, RPC e caixa-postal. Em [Zel 97] é apresentado um roteiro para a adição de novos tipos de conectores ao UniCon, mas o próprio autor ressalta que não é tarefa trivial.

2.6 CORBA

Vários trabalhos na área de projeto de sistemas distribuídos têm considerado o CORBA da OMG [OMG 96A] como um possível padrão de infra-estrutura (alguns deles são referenciados no capítulo 1, seção 1.6). Argumenta-se que CORBA possui características consideradas facilitadoras para o projeto de grandes sistemas distribuídos:

- ele é baseado em objetos e a OMG-IDL pode mapear as descrições de interface em linguagens de programação como C, C++, Smaltalk e ADA, provendo independência quanto a linguagem de programação;
- ele permite a programação distribuída e possui suporte adequado para sistemas

distribuídos (chamado de facilidades comuns [*common facilities*]);

- oferece soluções adequadas para a semântica de objetos distribuídos e impõe restrições semânticas quando necessário;
- a proposta para a programação de meta-classes está em estudos pela OMG [OMG 97B]. A programação de meta-nível permitirá a extensão flexível de objetos de aplicação;
- características de composicionalidade também estão sendo propostas recentemente em [OMG 97]. Nesta proposta, a IDL foi ampliada para incorporar comandos de configuração resultado em uma Linguagem de definição Estrutural (*Structural Definition Language*), OMG-STRUDL. A possibilidade de um executivo para configuração externa não foi explicitamente prevista neste proposta, mas pode ser adicionada como uma facilidade comum horizontal (*common horizontal facility*);
- ele possui um conjunto de *frameworks* de propósito geral (chamados de Facilidades Verticais Comuns [*Common Vertical Facilities*]) com os quais aplicações de domínios bem definidos e aceitos na indústria podem ser rapidamente construídos (saúde, finanças, aviônica, etc.);
- conversores e pontes de software estão disponíveis para a integração de arquiteturas como o DCE da OSF e o COM da Microsoft com o CORBA. A OMG possui um compromisso com a padronização e possibilidades de mapeamento entre CORBA e outros padrões como OSF-ODP, ANSA e TINA;
- a interoperabilidade com outros padrões e outros ORBs diferentes é provida através do IIOP, *Internet Inter-ORB Protocol*, um *gateway* entre ORBs, permitindo o uso da tecnologia de CORBA sobre grandes redes como a Internet;
- existem frameworks propostos para a migração de objetos, visando a disponibilidade de recursos, tolerância a falhas e permitindo a implantação do paradigma de agentes móveis autônomos sobre CORBA;
- estão sendo estudadas extensões nos estilos de comunicação oferecidos [Her 96, Maf 95, IONA-stream] para que sejam aceitas a comunicação de grupo e conexões orientadas a fluxos contínuos. Existe uma RFP (*Request for Proposal*) em aberto, solicitada pela OMG com relação a estes pontos, bem como para extensões de

tempo-real. Com estas extensões, aplicações com mídias de vídeo e áudio podem ser projetadas sobre CORBA e o suporte a requisitos de QoS podem ser atendidos em seguida, se estas experiências forem bem sucedidas.

Observa-se entretanto, que falta em CORBA mecanismos para a descrição da estrutura completa das aplicações, sendo apenas permitida a descrição da interface de objetos servidores. Além disso não existem meios de se descrever ou compor aspectos não-funcionais e operacionais das aplicações.

Embora tenhamos colocado CORBA como um suporte básico sobre o qual um esquema baseado em configuração possa ser concebido, apresentamos o exemplo do produtor/consumidor implementado por uma descrição em CORBA IDL 2.0, de forma a permitir a comparação do maior número de propostas.

Observa-se que na figura 2.19 apenas a interface do módulo *buffer* é descrita (linha 15), porque este elemento irá oferecer um serviço através de seus métodos. Nada é mencionado a respeito dos clientes, produtores e consumidores no caso. Embora o aspecto de distribuição esteja automaticamente contemplado pela infra-estrutura de suporte ao CORBA, não se pode descrever os aspectos de coordenação como a sincronização ou exclusão mútua, mas pode-se prever o tratamento de exceções.

```
1    struct item {
2       short Serial;
3       string Info;
4    };
5    struct StatReg {
6       short  Status;
7       string Reason;
8    };
9    typedef StatReg sreg;
10   typedef sequence<item> itemSequence;
11
12   exception NO_EMPTY_SLOTS {};
13   exception NO_SLOT_IN_USE {};
14
15   interface buffer {
16      short Put (in item item_info) raises (NO_EMPTY_SLOTS);
17      short Get (out book book_info) raises (NO_SLOT_IN_USE);
18      sreg Status (out short num_itens);
19   };
```

Figura 2.19 - descrição das interfaces do buffer em IDL

No exemplo da figura 2.19, os métodos Put e Get (linha 16 e 17) descritos na interface do objeto *Buffer* utilizam as exceções como forma alternativa para contemplar o aspecto de limitação do *buffer*. Nada é descrito sobre a sincronização destes métodos. Uma outra observação a ser feita é o fato de que nada pode ser dito sobre os objetos

clientes que vão acionar os métodos da interface. A ligação que existe entre a descrição IDL e a aplicação é a geração dos *stubs* e esqueleto de código necessários para desenvolver estes clientes. Com este código disponível o programador deve incluir outros aspectos necessários nos objetos servidores e clientes.

2.7 Conclusão

Os trabalhos analisados apresentam características comuns adequadas ao propósito de se desenvolver aplicações distribuídas. AOP e CF apresentam como pontos fortes a modularidade e a separação de interesses. CF obtém estas características através da abstração de filtros e a separação explícita entre interface e implementação, enquanto AOP, por sua vez utiliza linguagens especializadas para descrição de aspectos. Metajava e OpenC++ oferecem recursos para separação dos aspectos funcionais e não-funcionais no nível da programação, separando a programação de nível básico e meta-nível.

Das ferramentas analisadas, MetaJava se mostrou interessante porque estava facilmente disponível, foi facilmente integrada a um ambiente já utilizado e apresentou características que suportam o dinamismo desejado em aplicações distribuídas. Foi possível construir a aplicação exemplo e explorar várias possibilidades de implementação. Estas características são importantes em uma ferramenta para a concepção de sistemas distribuídos.

As ferramentas e ambientes avaliados na primeira parte não mostraram, entretanto, adequabilidade para descrição de arquiteturas ou estruturas de aplicações. O suporte oferecido está no nível da programação. Além disto, informações relevantes sobre o possível comportamento da aplicação que está em desenvolvimento não podem ser extraídas diretamente nestes ambientes. Ferramentas adicionais são necessárias para complementar as funções oferecidas.

Estas observações levam ao reforço das conclusões preliminares estabelecidas no capítulo 1, que as tecnologias de objeto, reflexão e AOP, embora apresentando características desejáveis para a construção de aplicações distribuídas, não podem ser utilizadas como instrumentos autônomos. Elas podem ser utilizados como fundação para modelos mais abrangentes, como por exemplo, ambientes de configuração.

Dois exemplos de sistemas baseados em configuração, Regis (e a linguagem de configuração Darwin) e UniCon foram avaliados. Considerações sobre código executável ou a sua praticidade não foram feitas, apenas o nível da configuração foi avaliado. De uma forma geral, foi possível descrever a aplicação exemplo através de seus componentes e a interligação destes componentes. A aparência de uma configuração em Darwin parece ser mais "limpa" do que em UniCon. Entretanto, UniCon permite que se explicite características sobre a interligação dos componentes e alguns aspectos não-funcionais. No exemplo desenvolvido isto se deu através da utilização de um conector oferecendo chamadas a procedimentos embutidos na própria ferramenta de compilação e geração de código (*BUILTIN*). Este grau de liberdade não é suportado diretamente em Darwin.

Um exemplo baseado em CORBA também foi desenvolvido para permitir uma comparação mais ampla. Uma avaliação mais abrangente sobre os sistemas analisados será feita nos capítulos 4 e 5 após a apresentação de nossa proposta.

No próximo capítulo apresentamos nossa abordagem para o projeto de grandes sistemas distribuídos. Esta abordagem utiliza o conceito de arquiteturas de software para descrever uma aplicação. Isto é feito em alto nível de abstração, que pode ser refinada até que os módulos funcionais possam ser mapeados em uma implementação. A formulação desta abordagem procura atender ao conjunto de requisitos construído na capítulo 1 e para isso se vale de características úteis das diversas tecnologias estudadas como, o encapsulamento da tecnologia de objetos, a flexibilidade da programação em meta-nível e da abrangência da configuração. Dentro da descrição de uma arquitetura, aspectos funcionais e não-funcionais podem ser descritos, permitindo que o projetista possa conceber a aplicação em todos os requisitos em alto nível, evitando que aspectos não-funcionais tenham que ser contemplados apenas na fase de implementação, de uma forma *ad-hoc*.

Capítulo III

Configuração Orientada a Separação de Interesses

3.1 Introdução

A presente proposta oferece uma nova alternativa para a construção de aplicações distribuídas baseadas em configuração. Esta proposta difere de outras existentes por integrar o paradigma de separação de interesses à tecnologia de configuração através de mecanismos de reflexão e a programação de aspectos. Nela também estão contemplados requisitos como a facilidade para reutilização de elementos e o suporte à evolução dinâmica. Além disso, ela apresenta significativa abrangência; qualidade relevantes discutidas no primeiro capítulo.

No escopo de nossa proposta, o conceito de configuração está baseado em:

- uma metodologia para a concepção de aplicações, por composição, a partir de módulos componentes, atendendo a aspectos funcionais básicos das mesmas;
- uma linguagem de configuração, que permite:

1. descrever a interconexão de módulos para formar a estrutura das aplicações ou módulos de maior complexidade.
2. descrever características ou aspectos não-funcionais das aplicações, como por exemplo, a sincronização, distribuição, tolerância a falhas ou parâmetros de qualidade de serviço.

- um sistema de suporte, chamado configurador, que execute as ações necessárias para criar, instanciar, ligar ou remover componentes das aplicações e faça o mapeamento

entre o nível da configuração e os sistemas de execução nativos.

Durante o desenvolvimento desta proposta, a tecnologia de objetos e técnicas reflexivas são empregadas em várias partes. A concepção do modelo básico e a correspondente estrutura de suporte, utiliza conceitos da tecnologia de objetos como encapsulamento e visibilidade de interface, por exemplo. A reflexão computacional e estrutural são empregadas na construção do próprio sistema de suporte à configuração e também estão presentes nos mecanismos para a composição das aplicações. Adicionalmente os conceitos da reflexão computacional são utilizados para propiciar que a separação de interesses seja observada durante a concepção das aplicações. Estes pontos serão discutidos neste capítulo.

As próximas seções estão organizadas da seguinte forma: inicialmente apresentam-se, em forma preliminar, os conceitos básicos do modelo de configuração adotado. Em seguida, destaca-se a forma com que a separação de interesses é tratada na proposta através da análise de alguns aspectos não-funcionais como a distribuição, comunicação e a coordenação. Sugere-se também uma metodologia para que o modelo seja utilizado com o objetivo de se obter a separação de interesses. O capítulo é concluído com uma discussão sobre o modelo e sua aplicabilidade, enquadrando-o segundo os requisitos identificados no capítulo 1 e através de comparações com as propostas avaliadas no capítulo 2 .

3.2 Elementos Básicos

A nossa proposta está baseada nos seguintes elementos:

1. **módulos**, que são os componentes de uma aplicação, que basicamente, encapsulam sua funcionalidade;
2. **conectores**, usados, no nível de configuração para interligar e selecionar a forma de interação de módulos. No nível de operação os conectores intermediam e executam a interação de módulos de acordo com o que foi configurado;
3. **portas**, que identificam os pontos de acesso pelos quais módulos e conectores oferecem ou solicitam serviços.

O primeiro passo para a construção de uma aplicação é a descrição de sua arquitetura.

Uma arquitetura é descrita através de uma configuração de módulos, que irão compor a funcionalidade básica da mesma e conectores, que interligarão e intermediarão a interação destes módulos. Em seguida, o relacionamento destes elementos é determinado descrevendo-se as ligações entre eles. Adicionalmente, durante o estabelecimento da configuração podem ser descritos aspectos não-funcionais da aplicação, permitindo que a composição de módulos e conectores seja ajustada em diferentes aspectos.

3.2.1 Módulos

Um módulo é um elemento que potencialmente encapsula a funcionalidade de uma aplicação (ou parte dela). Elementos funcionais claramente distintos da aplicação dão origem a tipos de módulos distintos [Sha 95], como por exemplo clientes, servidores, geradores e tratadores de eventos. Elementos funcionais claramente distintos da aplicação dão origem a tipos de módulos distintos. Servidores de arquivos, banco de dados ou servidores HTTP, etc., são facilmente representados por um módulo. Outros recursos como arquivos ou interfaces de E/S também podem ser representadas por módulos no nível da configuração.

Uma aplicação é composta por instâncias de módulos. Observe-se a figura 1. O módulo é o lugar onde estão contidas estruturas de dados e métodos[3]. Em uma instância de módulo as estruturas de dados armazenam o estado do módulo (representado por Var1 e Var2 na figura 3.1). Os métodos de um módulo, similarmente ao conceito de métodos na tecnologia de objetos, implementam a funcionalidade do módulo realizando computações ou manipulando o estado do módulo.

As interações de um módulo com outros elementos são realizadas através de pontos de acesso chamados de **portas**. Os métodos de um módulo podem estar associados a uma porta de entrada, através da qual outros módulos podem solicitar serviços (métodos 2, 3 e 4 da figura 3.1). Desta forma, estes métodos passam a ter visibilidade externa. Se um método não estiver associado a uma porta de entrada este será considerado privativo do

---

[3] Optamos por utilizar o termo método, tradicionalmente utilizado pela tecnologia de objetos, para fazer referência a procedimentos ou funções que implementam a funcionalidade de um módulo.

módulo e não poderá ser invocado diretamente por outros módulos (método 1 da figura 3.1). Portas de saída de um módulo podem ser utilizadas por quaisquer métodos deste módulo para solicitar serviços a outros módulos. O conjunto de portas de entrada e saída de um módulo podem ser genericamente referenciado como **interface** ou **assinatura** do módulo.

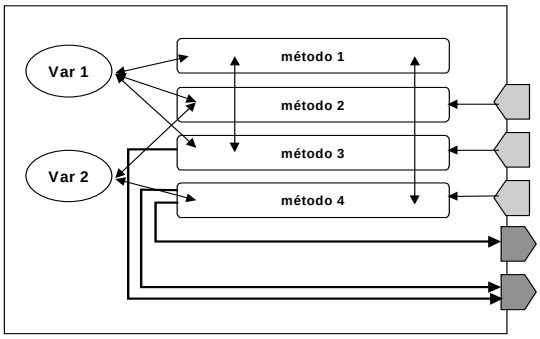


Figura 3.1 - Módulo, porta e métodos

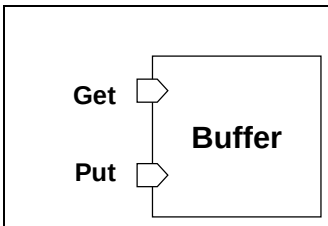
No modelo hora proposto, todos os módulos componentes de uma aplicação potencialmente executam concorrentemente. O mesmo ocorre em relação aos métodos de um módulo, que, a princípio, executam de forma concorrente entre si.

Os módulos não precisam ser concebidos especificamente para prever reconfiguração ou evolução dinâmica, aspectos relevantes em nossa proposta. O suporte básico para configuração e reconfiguração é provido através de comandos de configuração, que podem instanciar, bloquear, desbloquear ou terminar um módulo em situações específicas, como será detalhado nas seções seguintes. Reconfigurações dependentes da preservação da consistência de estado do sistema são fortemente dependentes da aplicação, requerendo suporte específico fornecido pelo programador.

A proposta de configuração é, em princípio, independente de linguagens de programação ou ambientes de execução especializados. É importante que uma aplicação descrita através da linguagem de configuração possa ser mapeada em programas em uma linguagem de programação qualquer. A ponte entre os conceitos do modelo e a implementação é simples.

Um módulo pode ser mapeado em uma estrutura de implementação como um objeto em C++, Java ou Smaltalk, ou em estruturas mais abstratas como descrições em IDL-

CORBA ou DCOM, por exemplo. O casamento entre um módulo e um objeto é bastante direto. Um objeto possui métodos e um estado. De forma análoga ao que ocorre nos módulos no nível da configuração, os métodos de um objeto são invocados utilizando-se uma referência ao objeto e o nome dos métodos, juntamente com os valores dos tipos dos parâmetros necessários. O seguinte exemplo em Java ilustra este ponto para um módulo *Buffer* com duas portas de entrada, *Put* e *Get*. O mapeamento do módulo em Java é descrito através de uma classe *Buffer* (linha 1), contendo dois métodos, *Put* e *Get* (linhas 6 e 7).




```
1   class Buffer {
2      private int putPtr; int getPtr;
3      private int max_itens;
4      private int itens[50];
5      public Buffer (int j);
6      public int Put(int value);
7      public int Get (int i);
8      ...
9   };
```

Figura 3.2 - mapeamento de um módulo

Nos casos mais simples ou nos casos em que a linguagem de programação não suporta o conceito de objeto, um módulo pode ser mapeado em um procedimento ou função, disponíveis nas linguagens de programação tradicionais como C ou Pascal. Isto eqüivaleria a um módulo com apenas uma porta (e um método). Em caso de necessidade, esta limitação pode ser contornada usando-se alguns artifícios. Por exemplo, é possível reservar um parâmetro na assinatura de um método (que, neste caso, seria programado através de um procedimento ou função) para selecionar um trecho de código a ser executado. Quando uma requisição chega a este método, apenas uma parte é executada de acordo com o valor utilizado para este parâmetro. Neste caso, o módulo de onde parte a requisição determina o serviço específico a ser executado pela valoração adequada do parâmetro de seleção e não pela referência ao nome do método.

3.2.2 Portas

Portas são elementos auxiliares do modelo de configuração que representam pontos de interação de um módulo com outros módulos de uma aplicação. As portas podem ser de entrada ou de saída, como discutido na seção anterior. Na linguagem de configuração a

porta é utilizada como referência na ligação de dois componentes. Por exemplo, ao ligarmos uma porta de saída de um módulo cliente a uma porta de entrada de um módulo servidor, estamos ligando e resolvendo estas referências, permitindo a interação entre estes componentes. As portas também são elos importantes entre o nível de configuração e a implementação, como será discutido nos próximos parágrafos.

Seguindo os exemplos de servidores representados por módulos, um módulo servidor de arquivos pode ser concebido com portas de entrada distintas para oferecer serviços como abertura, leitura e escrita de arquivos. Um módulo cliente deste servidor de arquivos utiliza portas de saída, também distintas, neste caso, para solicitar os serviços desejados.

Uma porta é caracterizada, inicialmente, por um nome, que a identifica, um conjunto de tipos de parâmetros e, adicionalmente, um tipo de resultado. Este conjunto de características pode ser chamado de **assinatura da porta**.

Todo método com visibilidade externa é representado por uma porta de entrada. Na declaração desta porta são relacionados os parâmetros com seus respectivos tipos básicos e um tipo para o resultado, se o método necessitar apresentar algum. No nível da implementação, um mapeamento imediato para portas de entrada é o método em linguagens de programação orientadas a objetos. A assinatura de uma porta é diretamente mapeada para a assinatura de um método. O nome e o conjunto de tipos de parâmetros da porta são mapeados no nome e na lista de parâmetros do método de um objeto. Como alternativa, se uma linguagem orientada a objetos não for disponível, uma porta de entrada pode ser mapeada no nome e respectiva assinatura de uma função ou procedimento, oferecidos por uma linguagem de programação como C ou Pascal.

Quando um módulo precisa solicitar serviços a outros módulos, ele utilizará portas de saída configuradas para esta finalidade. Uma requisição de serviço a um método de outro módulo deve ser feita com um conjunto de argumentos contendo valores de tipos definidos pelos parâmetros declarados na porta de saída. Se algum resultado é esperado, seu tipo também deve ser declarado na porta de saída e uma variável com tipo correspondente deve ser prevista para receber este resultado. Portas de saída são mapeadas, de forma simétrica à portas de entrada em invocações à métodos ou chamadas a procedimentos. O nome da porta é mapeado no nome do método a ser

invocado e o conjunto de tipos definidos nos parâmetros declarados na porta são mapeados nos argumentos passados na invocação do método. No nível da implementação a referência ao objeto destino da invocação precisa apenas ser compatível com a classe deste objeto, não importando o nome da instância, uma vez que esta referência só será conhecida no momento em que as respectivas portas forem ligadas no nível da configuração (seção 3.2.3).

Quanto à direção do fluxo de informações, as portas podem ter um único sentido, apropriadas para interações que não envolvam retorno de resultados ou ter sentido duplo, apropriadas para interações em que haja espera/retorno de resultados, permitindo que se caracterize interações como síncronas ou assíncronas.

Para que a interação entre dois módulos ocorra é necessário que as portas de entrada e saída envolvidas estejam interligadas através de um conector. A interação entre módulos tipicamente ocorre enviando-se uma requisição com os valores do conjunto de argumentos por uma porta de saída de um módulo fonte que, por intermédio de um conector, chega a uma porta de entrada do módulo destino onde um método determinado é acionado. No módulo destino, o tratamento de uma requisição por um método é conseqüência da funcionalidade programada no mesmo, dos valores dos argumentos instanciados no módulo origem passados na requisição e do estado módulo destino.

De uma forma geral, uma porta de saída deve ter a assinatura compatível com a porta de entrada com a qual ela pode potencialmente interagir. Assim, é garantido que os tipos do conjunto de argumentos na porta de saída e o conjunto de parâmetros na porta de entrada correspondente sejam os mesmos, e que a requisição do método será feita corretamente.

A opção de se introduzir o conceito de porta no modelo, foi influenciada pelo fato de que as portas de saída permitem que se tornem explícitos todos os tipos de interações que partem de um módulo. Isso não seria possível se apenas os pontos de entrada da interface de um módulo, descritos pelas portas de entrada, fossem representados. Esta opção permite que se obtenha informações mais completas sobre a configuração sendo descrita. Por exemplo, a descrição de uma classe abstrata em C++ ou a descrição de uma interface em CORBA-IDL contemplam apenas os métodos implementados pelo

objeto (seria o equivalente às portas de entrada). Não existe, nestes exemplos, uma forma de se indicar, na interface, as interações que partem de um objeto.

Embora conceitualmente importantes na descrição da arquitetura das aplicações, as portas podem aparecer de maneira não explícita no nível da programação interna dos módulos. Essa opção depende de decisões relativas à implementação a ser adotada para os conceitos aqui expostos.

O trecho de programa a seguir, apresenta um módulo cliente e um módulo servidor, implementados por classes em Java (linhas 8 e 2, respectivamente). O módulo servidor disponibiliza dois serviços, *Service1* e *Service2*, através de portas de entrada, mapeadas em métodos da classe servidora em Java (linhas 4 e 5). O módulo cliente envia requisições para um módulo servidor através de suas porta de saída, mapeadas em invocações a método (linhas 13 e 15).

```
1    // servidor
2    public class Servidor {
3    private int n_itens = 0;
4    public int Service1(int i) {...};
5    public int Service2(int i) {...};
6    }

7    // cliente
8    public class Cliente {
9      public void ativar_cliente () {
10       int i;
11       Servidor serv = new Servidor();
12   ...
13       serv.Service1(i);
14   ...
15       serv.Service2(i);
16   }}
```

Listagem 3.1

É importante notar que existem outras alternativas para o mapeamento das portas. Por exemplo, em Regis, avaliado na seção 2.5.1, as portas da linguagem de configuração Darwin, são mapeadas explicitamente no código na forma de objetos instanciados a partir de classes especiais *port* e *portref*. Em RIO [Wer 95], as portas são mapeadas como estruturas de dados de um ambiente especializado, que devem ser instanciadas como variáveis no código que implementa os módulos. Nestes dois casos, é necessária uma disciplina por parte do programador, que deve se preocupar explicitamente com a programação das portas e em implementar o código do módulo para receber e enviar pedidos de interação usando primitivas do tipo *send* e *receive*. Em nossa proposta procura-se tornar o uso de portas transparente no nível da implementação não

obrigando o programador utilizar elementos estranhos à funcionalidade dos módulos.

3.2.3 Conectores

O conector é um elemento que oferece a abstração necessária para determinar-se, no nível da configuração, a interação de módulos, a ser efetivada no nível da operação. Conectores são responsáveis, primariamente, por encaminhar requisições que devem ir de uma porta de saída de um módulo para uma porta de entrada de outro módulo e o retorno dos resultados no sentido contrário, quando necessário. Este encaminhamento é executado segundo um padrão de comportamento para o qual o conector foi concebido.

Para realizar a interconexão de módulos, os conectores desempenham outro papel importante, que é o casamento de referências das portas de entrada e saída. Na seção anterior caracterizou-se a porta como referência a pontos de interação nos módulos. Observa-se, entretanto, que os módulos são concebidos como entidades autônomas, sem ligação implícita ou explícita com outros módulos. O único indício de como um módulo irá interagir com outros módulos é o seu conjunto de portas. Não se sabe, *a priori*, com que tipos de módulo esta interação se dará, apenas se sabe quais informações transitarão durante as interações, por conta do conjunto de tipos dos parâmetros configurados nas portas. Em nosso modelo, o conector é exatamente o elemento que acopla estas referências, no nível da configuração, interligando portas de entrada a portas de saída.

Conceitualmente, os conectores também possuem portas de entrada e saída. O papel das portas nos conectores é similar ao dos módulos, são referências a pontos de interação e também possuem uma assinatura. Observa-se que nos conectores as portas são geralmente configuradas aos pares, de forma que as requisições possam ser encaminhadas de uma porta de para uma porta de saída entrada do conector.

As interconexões de módulos são indicadas explicitamente durante a configuração através de comandos de *ligação*. Através destes comandos as referências a portas de entrada e saída dos módulos envolvidos passam a ser conhecidas e podem ser acopladas pelo conector. Em um comando de ligação são indicados os módulos a serem interconectados e o tipo de conector a ser utilizado.

A sintaxe do comando de ligação, a ser apresentada no próximo capítulo, possui

variações que permitem que se faça referência aos módulos sendo interconectados, ou às portas de saída e entrada sendo interconectadas. No primeiro caso, todas as portas dos módulos envolvidos são automaticamente ligadas pelo conector e, no segundo caso, a ligação é feita apenas entre as portas referenciadas. Em qualquer caso, durante uma ligação, portas de saída dos módulos são ligadas a portas de entrada do conector e portas de saída do conector são ligadas a portas de entrada dos módulos. Uma ligação entre uma porta de um módulo e uma porta de um conector só poderá ocorrer se as assinaturas destas portas forem compatíveis. Não é necessário que os nomes das portas sejam os mesmos para garantir esta compatibilidade, mas isso poderá facilitar ligações automáticas baseadas em contexto. O suporte a configuração pode facilmente renomear os identificadores de portas, se for conveniente.

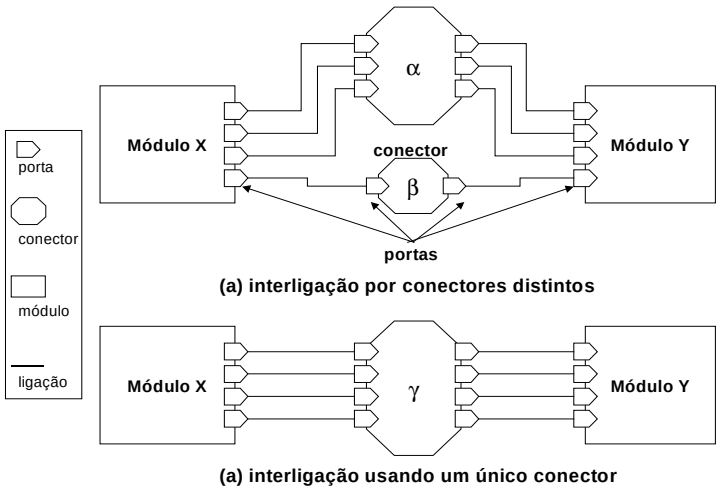


Figura 3.3 - configuração interligando módulos através de conectores

As figuras 3.3.a e 3.3.b apresentam duas configurações simples. Na figura 3.3.a a ligação dos módulos *X* e *Y* foi feita utilizando-se a referências de porta com conectores diferentes (α e β) e na figura 3.3.b todas as portas dos módulos foram ligadas pelo mesmo conector (γ). A utilização de um ou outro estilo de ligação depende da conveniência para o projetista da aplicação sendo concebida.

O conector pode operar, em algumas configurações, simplesmente repassando as requisições, por exemplo, de uma porta de saída de um módulo *X* a uma porta de entrada de um módulo *Y* sem fazer qualquer manipulação sobre os dados transportados. Este seria o caso de uma simples invocação a método. Em outras situações, além do

encaminhamento de requisições, o conector pode atuar manipulando os valores dos argumentos das requisições que passam através dele ou impondo comportamentos diferentes à forma com a qual o encaminhamento das requisições é feito ou adicionar novas características à interação dos módulos. Por exemplo, no caso de módulos distribuídos, o protocolo de comunicação que transportará as requisições e respostas, pode ser configurado selecionando-se um conector adequado. Adotando-se esta metodologia consegue-se o efeito de se adicionar características não-funcionais ou operacionais aos módulos que o conector interconecta, definindo-se, com isso, uma arquitetura diferente para a aplicação. Este ponto será aprofundado na seção 3.3.

Os conectores têm papel significativo na estruturação das aplicações, entretanto não participam diretamente das funções básicas das mesmas. Conectores interligam módulos determinando uma composição e identificando módulos que podem interagir uns com os outros. Além disso, conectores podem controlar as interações entre módulos interceptando e manipulando requisições e respostas. Esta atuação ocorre em um meta-nível e torna evidente a reflexividade deste elemento e o seu papel no comportamento da arquitetura descrita, atuando no ajuste de aspectos não-funcionais.

O mapeamento de conectores para o nível da implementação não é rígido. Um conector pode assumir, por exemplo, a forma de uma consulta a uma tabela para resolver referências de uma chamada, iniciada em um módulo, para um procedimento que é oferecido por outro módulo. É possível também que um conector tenha que prover acesso a recursos do sistema de suporte, como mecanismos de comunicação, no caso em que os módulos estejam distribuídos e a interação entre eles tenha que ser feita através de uma rede de comunicação.

No exemplo da *Listagem 3.1*, o conector que interliga o módulo cliente ao módulo servidor é mapeado simplesmente criando-se, no corpo do objeto cliente, uma instância do objeto servidor (linha 11) e utilizando-se esta instância e os nomes dos métodos para construir as invocações. O mapeamento do conector em uma implementação, neste caso, é exatamente a referência à instância do objeto servidor, criada no corpo do módulo cliente, e a invocação de um método da instância deste objeto feita corretamente. Neste caso, a resolução de referências é feita pelo compilador ou interpretador.

A implementação dos conectores pode, também, seguir a mesma linha dos módulos. Um conector pode assumir a forma de um objeto. Este objeto será o intermediário entre outros objetos, que implementam módulos. Tipicamente um objeto implementando um conector deverá ser concebido de forma que ele ofereça um conjunto de métodos compatíveis com os métodos invocados pelo módulo cliente (que contém portas de saída) e deve invocar métodos com parâmetros compatíveis com o módulo servidor (que contém portas de entrada). O trecho de código da *Listagem 3.2* ilustra este ponto, estendendo o exemplo da *Listagem 3.1*. Na classe *Connector* são oferecidos dois métodos compatíveis com os métodos invocados pela classe *Cliente* (linhas 4 e 8) que por sua vez invocam os métodos da classe *Servidor* e repassam o retorno de volta ao cliente (linhas 6 e 10). Observa-se que o código do conector contém pares de portas de entrada-saída associadas a cada porta de saída e de entrada dos módulos que ele interliga.

```
1    // conector
2    public class Connector {
3       private server = new Servidor ();
4       public int Service1(int i) {
5          // código de aspectos não-funcionais
6          return server.Service1 (int i);
     ...};
7       public int Service2(int i) {
8          // código de aspectos não-funcionais
9          return server.Service2 (int i);
10   ...};
11   }
```

Listagem 3.2

Os módulos *Cliente* e *Servidor* (definidos na listagem 3.1) e conector *Connector* (definido na listagem 3.2) podem ser utilizados em um exemplo de aplicação mais completo. Na descrição da configuração deste exemplo os módulos devem ser instanciados e em seguida interligados pelo conector. Um suporte para configuração é responsável por resolver e encaixar as referências contidas nos módulos e no conector, no nível da implementação, para que esta espelhe a aplicação descrita. Uma das formas para se realizar este encaixe é a renomeação de algumas referências na implementação do conector, como descrito em [Lob 98], de forma que este processo seja transparente à implementação dos módulos.

Ainda com relação ao exemplo acima no corpo de cada método implementado no conector é possível adicionar-se código para implementar alguns aspectos não previstos nos módulos. Por exemplo, seria simples introduzir-se, entre as linhas 4 e 6, uma

pequena rotina que verificasse os limites do parâmetro *i* no método *Service1*.

No caso de conectores que interligam módulos distribuídos, a implementação destes pode ter uma solução também distribuída. Assim, um conector pode ser implementado por sub-partes que vão estar distribuídas, provavelmente nas mesmas estações onde estiverem instanciados os módulos que ele interliga. A parte do conector que interage com um módulo cliente será chamada de conector-cliente e a parte que interage com um módulo servidor será chamada de conector-servidor. Basicamente, é necessário que o conector-cliente receba as requisições do módulo cliente e as repasse para o conector-servidor. A comunicação entre as partes do conector se dará através de algum mecanismo de comunicação disponível, como *sockets* por exemplo. O conector-servidor, por sua vez, repassa as requisições para o módulo servidor, completando a interação entre os módulos. Se a interação utilizar uma resposta, o caminho inverso é percorrido. Em [Szt 98A] este estilo de mapeamento é apresentado com mais detalhes. Observa-se que a mecânica utilizada neste mapeamento é semelhante ao RPC, RMI e CORBA. Com efeito, conectores também podem ser implementados utilizando-se estes mecanismos se necessário.

3.2.4 Composição

Ao descrever-se a configuração de uma aplicação, essencialmente descreve-se uma composição de módulos. A concepção de uma aplicação é iniciada com a seleção e a descrição dos módulos e conectores a serem utilizados, que têm alguma correspondência com os requisitos levantados junto ao usuário. Isto feito, a aplicação é composta descrevendo-se como os módulos e conectores são interconectados. Sob esta perspectiva toda aplicação pode ser considerada uma composição de módulos.

A linguagem de configuração estende a perspectiva de composição, permitindo a descrição de módulos compostos. Um modulo composto é formado por módulos primitivos, isto é, que possuem um mapeamento para uma implementação, ou outros módulos compostos. Um módulo composto também possui um nome, que o identifica, e um conjunto de portas, compondo sua interface. O conjunto de portas de um modulo composto é formado pelo conjunto, ou um subconjunto, das portas dos módulos componentes. Este subconjunto deve ser selecionado e marcado como tendo visibilidade externa com relação ao modulo composto. Para o resto da aplicação, um

módulo composto tem a mesma apresentação de um módulo primitivo e somente se relaciona com a mesma através de suas portas.

A figura 3.4 apresenta um exemplo de uma aplicação com um modulo composto (MC1) formado por três módulos primitivos (Mp1, Mp2 e Mp3). Nesta figura também estão identificadas duas portas selecionadas para ter visibilidade externa. Os outros módulos da aplicação têm acesso ao módulo composto apenas através destas duas portas.

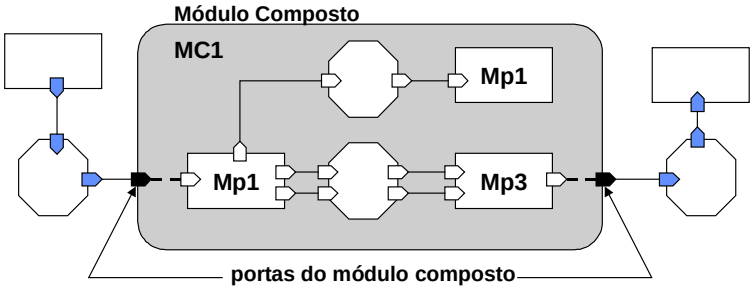


Figura 3.4 - uma aplicação com módulo composto

A composição de módulos pode ser usada para a construção de módulos com função agregada ou mais especializados. Agregando-se adequadamente, por exemplo, um *modulo de áudio* e um *módulo de vídeo* pode-se construir um *módulo de mídia* composto. O *módulo de áudio* do mesmo exemplo pode ser, por sua vez, um módulo especializado, composto por *um módulo fonte de áudio* e um *módulo codificador*. A composição de módulos também pode ser usada para a criação de conjuntos ou bibliotecas de componentes especializados ou *frameworks*. Assim, os benefícios da estruturação da aplicação com módulos compostos, são ampliados com a possibilidade de reutilização destes módulos, no nível da configuração, em outras aplicações. Deve ser observado ainda que os módulos compostos só existem no nível da configuração e não precisam de mapeamento para uma linguagem de programação, uma vez que este mapeamento já está definido para os módulos primitivos.

3.3 Separação de Interesses

No capítulo 1, as vantagens da concepção de aplicações distribuídas considerando-se o requisito *separação de interesses* foram levantadas. Segundo este requisito, uma aplicação distribuída deve ser concebida por partes logicamente distintas que serão

integradas posteriormente. Por exemplo, a parte funcional da aplicação pode ser concebida em primeiro lugar. Em seguida, as interfaces com o usuário e as necessidades de comunicação são projetadas. Desta forma, pontos de interesse razoavelmente distintos da aplicação são abordados separadamente e o projetista pode concentrar seus esforços no projeto de cada ponto em momentos diferentes. Chamamos a cada um destes pontos de aspecto.

Um aspecto se refere a um conjunto de características, propriedades ou comportamentos, com alguma afinidade, relacionados com a aplicação. Aspectos associados com a funcionalidade da aplicação são chamados aspectos funcionais. Aspectos que não fazem parte da funcionalidade básica da aplicação são chamados aspectos não-funcionais. Normalmente, estes aspectos são relacionados com detalhes ou características subjetivas do comportamento da aplicação. Aspectos não-funcionais relacionados com o ambiente de operação da aplicação são também chamados de **operacionais**. Usos semelhantes do termo *aspecto* são encontrados em [Kic 96 e Lop 97] e o termo interesse (*concern*) também é utilizado com sentido semelhante em [Aks 96], [Hür 95] e [Ber 96T].

A identificação de propriedades e características de sistemas de software que podem ser consideradas como um aspecto ainda é objeto de pesquisas [Lop 97T, And 97, Mat 97, Agh 98]. Ao se determinar um aspecto, deve ser observada uma certa coerência para que algumas propriedades não estejam sendo utilizadas de forma repetida em outros aspectos. É desejável também que um aspecto possa ser configurado separadamente de outros aspectos e que as configurações de aspectos diferentes possam ser combinadas sem efeitos colaterais ou inconsistências [Jac98]. O primeiro passo em direção a esta realidade seria garantir a ortogonalidade entre os aspectos. Entretanto, nem sempre uma separação total entre aspectos é possível ou factível [Men 97]. Este problema será investigado nos desdobramentos de nossa pesquisa para ser aplicado na composição de conectores (seção 3.3.7).

No contexto desta proposta a separação de interesses é atendida das seguintes formas:

- selecionando-se módulos para compor a estrutura funcional da aplicação e conectores para intermediar a interação destes módulos, em uma primeira fase;
- concebendo-se a arquitetura da aplicação pela interconexão dos módulos e

conectores para atender os requisitos de interação demandados pela aplicação;

- configurando-se aspectos não-funcionais através de ajustes nos conectores ou por composição de módulos e conectores adaptados, a partir de módulos e conectores mais simples.

Na seção 3.2.1, mencionou-se que os aspectos funcionais de uma aplicação podem ser potencialmente encapsulados em módulos separados e depois combinados em uma mesma aplicação. Módulos compostos, encapsulando aspectos funcionais mais especializados, podem ser concebidos pela composição de módulos provendo funções básicas e módulos que ofereçam características não-funcionais que sejam relevantes à funcionalidade do módulos composto. Na seção 3.2.2, salientou-se que os conectores, além de interligar os módulos, por outro lado, podem especializar ou acrescentar características às interações que ocorrem entre os módulos, devido a capacidade reflexiva dos mesmos. Aspectos relacionados com a interação e comunicação de módulos, por exemplo, são facilmente identificados e embutidos em conectores. Com estas possibilidades oferecidas por nossa abordagem, o projetista poderá usar uma metodologia na concepção das aplicações para que os módulos sejam responsáveis pelos aspectos funcionais e os conectores pelos aspectos não-funcionais. Esta disciplina, embora não obrigatória, potencializa a reusabilidade dos módulos e conectores.

3.3.1 Configuração de Aspectos em Conectores

Ao longo de experiências preliminares com o desenvolvimento de ambientes de configuração [Wer 95 e Szt 95], esquemas para a adição e gerenciamento de alguns requisitos não-funcionais e operacionais como a comunicação, seleção de protocolos e tolerância a falhas foram implementados e sua factibilidade comprovada. Nestes trabalhos, a idéia de um elemento exclusivo para interligar e intermediar a interação entre módulos ainda não existia. O suporte para a configuração de aspectos era, então, provido em conjunto pelos módulos envolvidos e por um sistema de suporte especializado. Na presente proposta, requisitos não-funcionais são tratados nos conectores que podem implementar diversos aspectos relacionados a estes de uma maneira mais uniforme sob o ponto de vista de utilização pelos módulos. Os conectores possuem um conjunto de características que os tornam especialmente atraentes para

tratar aspectos relacionados com a interação e a comunicação entre componentes [Bis 96]. Dentre estas características, já relacionadas no texto, destacam-se:

- o conhecimento da interface e as referências dos módulos por eles interligados;
- a capacidade de examinar e manipular o conteúdo das requisições e respostas que passam por eles;
- a possibilidade de controlar a forma da interação dos módulos.

Estas características oferecem ao projetista a liberdade para adequar o comportamento dos conectores às necessidades especificas da aplicação. Adicionalmente, ao mapear um conector em uma implementação, o projetista ainda pode utilizar mecanismos auxiliares como primitivas de sincronização, protocolos de comunicação, entre outros, oferecidos pelo sistema de suporte nativo, para realizar o controle deste comportamento. Assim sendo, outros aspectos, além da comunicação, também podem ser configurados a partir dos conectores. Por exemplo, aspectos como a sincronização podem ser associados aos conectores, tornando, de certa forma, a programação dos módulos independente destes aspectos não-funcionais.

Para exemplificar a configuração de aspectos a partir de um conector, considere uma aplicação composta por dois módulos clientes utilizando um conector para interagir com um módulo servidor (figura 3.5). Na versão da figura 3.5.a determinou-se que cada módulo será instanciado em uma estação distinta. Sendo assim, durante a configuração, seleciona-se um conector, dentre um conjunto disponível, que contemple os aspectos de comunicação necessários e configura-se este conector com portas compatíveis com os módulos a serem interligados.

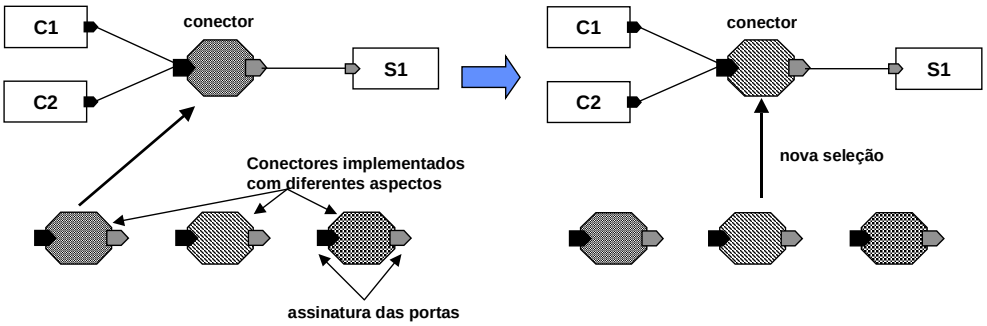


Figura 3.5 - configuração de uma aplicação simples

Suponha que por uma razão operacional, após um período de funcionamento, esta aplicação necessite que as mensagens que circulam entre módulos clientes e servidor sejam criptografadas. Para atender a este novo aspecto, um outro conector, com a mesma assinatura, e que contemple, adicionalmente aos aspectos de comunicação, a criptografia, é selecionado para substituir o conector original (figura 3.5.b). Na seção 3.3.7 argumenta-se que este novo conector pode ser concebido por composição. Neste caso não existe a necessidade de um único conector encapsulando todos os aspectos desejados.

Vale observar que ajustes durante a operação podem ser feitos sob controle da própria aplicação pela reconfiguração de conectores. Uma opção é concentrar-se as tomadas de decisão sobre as reconfigurações em módulos especiais, utilizando-se inclusive linguagens especializadas, que interagem com o configurador para adaptar a aplicação [Gar 94 e Agh 98].

3.3.2 Aspectos não-funcionais ou operacionais

A configuração de cada aspecto é feita através da linguagem de configuração que dispõe de uma sintaxe adequada para que os conectores possam ser descritos com esta finalidade. Por exemplo, para o aspecto de coordenação, estão disponíveis os mecanismos para se indicar no conector os requisitos de concorrência e sincronização dos módulos por ele interligados (ver a seção 3.3.5).

Dentre os aspectos não funcionais que podem ser necessários na concepção de aplicações distribuídas e que são, em princípio, realizáveis nos conectores, destacam-se:

- **Suporte de comunicação** - Permite a configuração de mecanismos e parâmetros relacionados com a comunicação entre os módulos da aplicação;
- **Replicação e tolerância a falhas** - Permite a configuração de módulos replicados e a seleção de esquemas de tolerância a falhas;
- **Persistência -** Permite configurar aspectos de persistência para os módulos. Este aspecto pode ser utilizado em conjunto com o aspecto de tolerância a falhas para facilitar a implementação de esquemas de recuperação de módulos que falharam;
- **Tempo Real** - Características relacionadas com restrições temporais ou o

escalonamento de módulos associado a políticas associadas com o tempo podem ser especificados através do uso de conectores que ofereçam suporte para tempo-real;

- **Coordenação -** Os aspectos relacionados com a coordenação de módulos, como a concorrência entre instâncias de módulos ou entre métodos de uma mesma instância, e a sincronização no acesso destes métodos podem ser configurados nos conectores associados aos mesmos.

Para esta primeira abordagem do modelo serão detalhados apenas alguns dentre os aspectos elencados, mas que ilustram o potencial de expressividade dos conectores:

- **distribuição**, que explora a possibilidade dos conectores poderem ser concebidos por sub-partes que cooperam para realizar suas funções;
- **comunicação**, que explora o fato de que o conector pode controlar a comunicação entre os módulos, e assim sendo pode tratar o transporte das requisições e respostas segundo uma configuração imposta;
- **coordenação**, que se vale do fato de que o conector conhece a interface dos módulos que ele interconecta e assim pode controlar o comportamento dos métodos destes módulos imediatamente antes ou imediatamente depois das requisições lhes serem entregues.

Observa-se que a programação destes aspectos é, em grande parte, ortogonal à concepção dos módulos, contemplando a separação de interesses e possibilitando encapsular os mesmos em conectores. Entretanto outros aspectos podem não levar a uma solução completamente separada. Assim sendo, o modelo não proíbe o programador de agregar aspectos funcionais e não-funcionais no mesmo componente, embora isso deva ser evitado, e nem de utilizar qualquer recurso disponível nas linguagens de programação.

3.3.3 Aspectos de Distribuição

O modelo proposto permite que os módulos de uma aplicação possam ser instanciados em qualquer ponto de um sistema distribuído. A dosagem correta da distribuição de módulos da aplicação pode ter implicações no seu desempenho. O projetista pode usar uma metodologia, como a apresentada em [Pur 98], por exemplo, para planejar a

distribuição de objetos, da melhor forma possível, considerando-se fatores como o tráfego gerado pela interação de objetos e a hierarquia de classes. Na implementação dos módulos, não é necessária nenhuma referência quanto a localização do módulo ou qualquer outro suporte para a interação de módulos distribuídos. A localização de uma instância de um módulo é transparente em relação a sua funcionalidade.

Na linguagem de configuração, a instanciação de um módulo pode vir acompanhada da referência associada à localização onde o módulo deve ficar. Este é o único momento em que se associa um módulo à sua localização (ver seção 4.2.4).

O aspecto de distribuição precisa ser suportado, de alguma forma, pelo ambiente de configuração. Mecanismos para instanciar módulos em pontos distribuídos e gerenciar as referências destas instâncias devem estar disponíveis. Quando um módulo é ligado a outro módulo através de um conector, as referências necessárias para localizar estes elementos são instaladas em repositórios e administradas pelo ambiente de configuração de forma que este esquema seja transparente aos módulos.

O ambiente de configuração também pode otimizar a escolha da localização das instâncias de módulos se nada for informado na configuração da aplicação, por exemplo segundo um critério de menor carga. O esquema específico deste suporte não é relevante para o modelo. Este suporte poderá ser provido por um sistema de nomes ou algum outro mecanismo que ofereça os serviços de localização e referência necessários para viabilizar o aspecto de distribuição. Por exemplo, os sistemas de identificação e localização disponíveis em ORBs (*Object Request Broker*) ou os serviços de mapeamento existentes nos mecanismos de RPC (*Remote Procedure Call*) e RMI (*Remote Method Invocation*), podem ser adaptados para o ambiente de configuração.

3.3.4 Aspectos de Comunicação

Em uma aplicação distribuída a interação entre os módulos é feita através de um sistema de comunicação. Tipicamente, as requisições e respostas que devem circular entre os módulos são encapsuladas em mensagens e transportadas por este sistema de comunicação. É provável também que a interação entre os diversos módulos não possa obedecer a um único estilo de comunicação. Tanto na concepção quanto durante a operação de uma aplicação distribuída, ajustes na comunicação entre os módulos

podem ser necessários, como por exemplo a seleção do mecanismo a ser utilizado, o protocolo de comunicação mais adequado ou a determinação de parâmetros de qualidade de serviço (QoS). Estes pontos são tratados nos aspectos de comunicação.

Nos conectores a configuração de aspectos de comunicação torna-se natural. Como a comunicação entre os módulos de uma aplicação é intermediada por conectores, detalhes operacionais na forma desta comunicação podem ser ajustados através destes. Para isso, durante a implementação de um conector o programador pode fazer uso de facilidades, disponíveis no sistema nativo, para realizar a comunicação. Por exemplo, o programador poderá se valer de mecanismos de comunicação como sockets, RPC ou RMI. Muitas das interfaces de programação destes mecanismos também permitem que se determine parâmetros de QoS, complementando as possibilidades de ajuste nos aspectos de comunicação que se pode oferecer através de um conector.

No nível da configuração os aspectos de comunicação serão indicados por seleção de conectores ou através de sintaxe própria. A linguagem de configuração pode permitir que anotações sobre a comunicação sejam declaradas nos conectores de maneira simples. Existe, assim, a vantagem para o projetista em poder configurar detalhes sobre os aspectos de comunicação sem a preocupação imediata de como serão implementados.

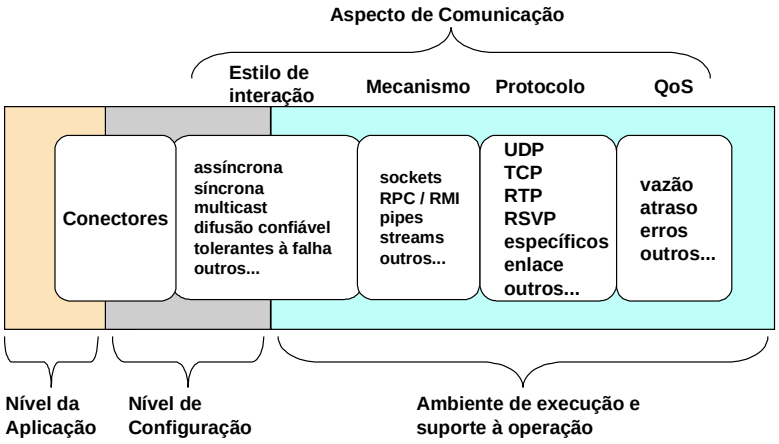


Figura 3.6 - partes do aspecto de comunicação

Para facilitar o projeto das aplicações é útil dispor de um repositório de conectores implementados com as combinações de estilos de comunicação, mecanismos e protocolos mais comumente usados. Caso contrário, o novo conector requerido deverá

ser implementado ou obtido por composição (ver seção 2.2.5). Em qualquer caso, durante a implementação de um conector, o programador pode fazer uso de facilidades disponíveis no sistema nativo. Por exemplo, o programador poderá se valer de mecanismos de comunicação como *sockets* ou RPC. Muitas das interfaces de programação destes mecanismos também permitem que se determinem parâmetros de QoS, complementando as possibilidades de ajuste nos aspectos de comunicação que se pode oferecer através de conectores. A figura 3.6 esquematiza as características que podem ser potencialmente configuradas no aspecto de comunicação. De uma forma geral, através da linguagem de configuração podemos selecionar conectores indicando:

- o estilo de interação entre módulos (síncrono, assíncrono, assíncrono com reconhecimento, etc.);
- estilos não triviais de comunicação em difusão, difusão seletiva e difusão seletiva confiável [Szt 94] de forma transparente para um grupo de módulos (figura 3.7);
- o mecanismo utilizado para realizar a comunicação, que pode ser selecionado por fatores como a disponibilidade dos mesmos, eficiência, etc. Como exemplos podemos listar:

1. *streams*, *pipes*, *sockets* encontrados em sistemas como o UNIX;
2. facilidades de RPC, disponíveis em vários sistemas, e RMI disponível em ambientes utilizando Java;
3. *middleware* do tipo CORBA que oferecem, entre outros serviços, a comunicação entre objetos remotos;

- o protocolo de comunicação a ser utilizado, como o UDP, TCP, RTP, etc.;
- o tratamento de incompatibilidades entre portas implementando pontes (*bridges*) de software. Um conector pode se responsabilizar pela conversão de representação de tipos, utilizando XDR, por exemplo, na comunicação de módulos residentes em máquinas com arquiteturas diferentes;
- parâmetros de qualidade de operação (QoS), permitindo a reserva de recursos, sincronização do fluxo de dados, etc.

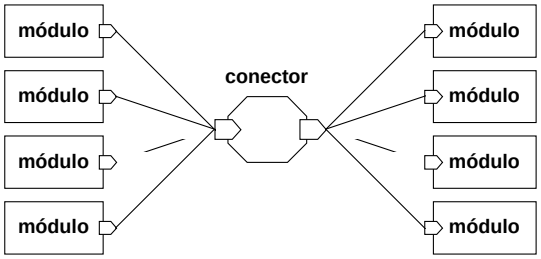


Figura 3.7 - Comunicação de grupo

Conceitualmente, características ortogonais relacionadas com a comunicação, configuradas em um conector não estão *amarradas* umas as outras. Pode-se ter uma aplicação, por exemplo, com dois módulos ligados por um conector que ofereça um estilo de comunicação do tipo *fluxo* (um canal com fluxo de dados contínuo) e especificar que o mesmo utilizará o protocolo UDP, TCP ou algum outro. Da mesma forma, não existe, em princípio, rigidez na escolha dos mecanismos para a implementação do conector. No mesmo exemplo, o conector selecionado poderia ser implementado com mecanismo de RPC, dependendo da situação específica em que a aplicação fosse executada.

Para que esta escolha *livre* de características possa ser feita, a consistência dos parâmetros configurados no aspecto de comunicação deve ser verificada, impedindo que uma seleção de um conector inadequado ou ineficiente seja instanciada. Existe também a necessidade de que os parâmetros selecionados sejam efetivamente suportados. Por exemplo, parâmetros de QoS indicados na configuração precisam ser mapeados de uma forma que faça algum sentido para o sistema de comunicação e que o mesmo possa ser ajustado segundo estes parâmetros.

Deve ser observado, entretanto, que nem todas as combinações fazem sentido ou são factíveis. Existem duas alternativas para a disponibilizar a configuração aspectos de comunicação em conectores, considerando-se o problema da combinação de características. A primeira seria utilizar um conjunto de conectores onde as implementações das combinações mais tradicionais destas características estejam disponíveis e que algumas propriedades, como QoS, por exemplo, possam ser definidas através de parâmetros. Uma outra alternativa seria a combinação de conectores diferentes, cada qual implementando características ortogonais relacionadas com a comunicação, para se compor um conector com as características desejadas, a exemplo

do que é feito em sistemas como [Hut 91, Med 97].

A composição de conectores pode ampliar as possibilidades de configuração de aspectos não-funcionais, mas implica em complexidade adicional para a linguagem de configuração e para o suporte à configuração. Por exemplo, para permitir a composição automática de conectores, a ortogonalidade das características sendo compostas e a consistência do conector composto resultante devem ser verificadas também de forma automática. Isto se aplicaria não somente a características distintas do aspecto de comunicação, mas também a outros aspectos não-funcionais ou a combinação de aspectos diferentes. A composição automática de conectores não estará completamente equacionada nesta proposta (ver seção 3.3.7), mas será objeto de estudos futuros.

3.3.5 Aspectos de Coordenação

Durante a concepção de aplicações concorrentes, questões sobre a execução autônoma de módulos, a disciplina na interação entre módulos e entre métodos de um mesmo módulo, devem ser encaradas. Tipicamente estas questões referem-se à concorrência, exclusão mútua e sincronização. O programador destas aplicações, muitas vezes, precisa embutir comandos ou primitivas de concorrência e sincronização no código para, por exemplo, acessar recursos compartilhados em exclusão mútua, valendo-se de mecanismos como *locks* ou semáforos [Ben 90].

O trecho de código da listagem 3.3 apresenta uma implementação possível para o método ***put*** de uma classe *BoundedBuffer* em Java, que segue a linha do exemplo usado no capítulo 2. Este método deve ser acessar o *buffer* com exclusão mútua em relação a outras chamadas ao método *put* e ao método *get* (não representado na listagem). Além disso a condição *não-cheio* (! full, linha 5) também deve ser verdadeira. Observa-se que dentro do código do método *put*, quase todas as linhas dizem respeito ao aspectos de concorrência e sincronização (com exceção da linha 11, que efetivamente executa o método) e para isso várias primitivas relacionadas com monitores e semáforos, segundo a semântica adotada em Java, são utilizadas (linhas 4, 5, 7, 8, 14, 18 e 20) . O código resultante é confuso e ofusca as partes que realmente dizem respeito ao problema original. Este problema é chamado de **entrelaçamento de código** (*code tangling*) em [Lop 97] que também identifica outras fontes deste problema, como a comunicação.

```
1    Public class BoundedBuffer {
2    ...
3       public void put(Object o){
4          synchronized(getLock()){
5             while (! (selfex.testEx() && mutex.testEx() && (! full))
6                getLock().wait();
7             selfex.enterEx(); mutex.enterEx();
8             getLock().notifyAll();
9          }
10
11         executa-o-put(o);
12
13         finally {
14            synchronized(getLock()){
15               empty = false;
16               if(usedSlots == size) full = true;
17
18              selfex.exitEx(); mutex.exitEx();
19              getLock().notifyAll();
20            }
21         }
22      }
23   ...
24   }
```

Listagem 3.3

Existem algumas opções disponíveis para incorporar-se requisitos de coordenação nas aplicações. Alguns sistemas oferecem mecanismos para a programação de concorrência e sincronização pelo uso de funções primitivas disponíveis através de bibliotecas. Sistemas operacionais como o UNIX oferecem a abstração de processos como unidade de concorrência e normalmente incorporam bibliotecas, como a IPC [Bac 87], que permitem a programação de mecanismos como semáforos, mensagens, *pipes*, etc. Existem também bibliotecas, como a pthreads [DCE 93], que oferecem mecanismos como *threads*, monitores, *mutexes*, etc. Estas bibliotecas podem ser utilizadas em programas escritos em linguagens de programação como o C e C++ e muitas vezes tem o suporte do próprio sistema operacional [Pow 91].

Algumas linguagens orientadas a objeto incorporam mecanismos para a programação de concorrência e sincronização, já inseridos na sintaxe da própria linguagem. Java é um exemplo. Várias destas linguagens foram concebidas como extensões das versões lineares das linguagens originais [Lea 93, Ach 93 e Mey 93]. Existem também algumas variações propostas na abordagem, como [Löh 93] que propõe o uso de anotações, como se fossem comentários ao programa principal, para expressar concorrência e sincronização.

Em experiências anteriores [Szt 95], com ambientes baseados em configuração, os aspectos de concorrência entre os métodos de um módulo eram determinados pela

forma de implementação adotada, que impunha uma política compulsória. Por exemplo, todas as mensagens que chegassem a um módulo eram enfileiradas e tratadas serialmente. Neste caso, a unidade de concorrência era o módulo e ajustes na coordenação interna dos módulos não eram possíveis de forma explícita no nível da configuração. Entretanto, nem toda aplicação possui uma estrutura que se encaixe naturalmente nesse modelo de concorrência.

Os problemas levantados nos parágrafos anteriores são tratados, em nosso modelo, através da configuração do aspecto de coordenação. Compartilhamos da idéia que a separação entre a concepção e os esquema de coordenação a ser adotado para os módulos de uma aplicação, oferece um nível de abstração adequado ao projetista no momento de planejar a arquitetura da mesma, além de aumentar as possibilidades de reuso [Agh 93].

A idéia de se separar os aspectos de coordenação da implementação da computação principal da aplicação aparece em algumas propostas pesquisadas. AOP e Composition Filters, avaliadas no capítulo 2 propõem respectivamente uma linguagem de coordenação e um mecanismo que estende o modelo tradicional de objetos, para tratar dos aspectos de coordenação separadamente dos aspectos básicos da aplicação. Outras propostas, anteriores a estas, como [Mat 93], já propunham soluções para separar a coordenação das classes de uma linguagem orientada a objetos, utilizando o conceito de *sincronizadores* e *transições*, extensões da abstração de classes, similares a anotações. DRAGOON [Atk 91], um outro exemplo, separa a abstração de classes em duas partes, *funcional* e *comportamental*. A parte comportamental de uma classe, descrita por uma linguagem específica, diferente da parte funcional, é responsável pela coordenação.

Em nossa proposta, os aspectos de coordenação de uma aplicação são descritos através da linguagem de configuração. Estes aspectos são especificados em alto nível de abstração sem a necessidade de prevê-los na concepção ou implementação dos módulos. A rigor, instâncias diferentes de um mesmo tipo de módulo podem estar submetidas a esquemas de coordenação distintos. O modelo para o tratamento do aspecto de coordenação prevê três situações: concorrência entre módulos de uma aplicação, concorrência e exclusão mútua entre métodos de um mesmo módulo, e a sincronização entre métodos de um mesmo módulo.

Concorrência e sincronização entre módulos de uma aplicação

Em nosso modelo todos os módulos de uma aplicação são executados potencialmente de forma concorrente. Cada módulo pode ser considerado uma unidade autônoma que pode concorrer pelo processamento com outros módulos ou podem executar paralelamente se estiverem em ambientes paralelos ou distribuídos. Este modelo é adotado em propostas semelhantes, como em [Agh 93 e Lop 97], em que cada objeto desempenha um papel independente na aplicação como se fosse um "ator". A modelagem de uma aplicação com módulos concorrentes captura naturalmente as situações de concorrência encontradas no mundo real. Por exemplo, em um sistema bancário temos vários clientes, caixas e contas-correntes, que são elementos autônomos, com vida própria. Uma aplicação modelada a partir deste sistema, segundo nosso modelo, pode ser concebida mapeando-se cada cliente, caixa e conta-corrente em uma instância de módulo.

O grau de concorrência dos módulos dependerá de como um módulo é implementado e da política de escalonamento do ambiente de suporte. Uma instância de módulo poderá assumir algumas formas diferentes. Por exemplo, se Java e a correspondente *Máquina Virtual Java (JVM)* forem utilizadas, um módulo será mapeado em uma classe com seu respectivo *bytecode* [Java A] que poderá conter uma ou mais *threads*. No caso de um ambiente Unix, um módulo será mapeado em um processo, ou ainda em uma ou mais *threads*, se houver suporte para este mecanismo [Pou 91]. O escalonamento, por sua vez, também associado ao ambiente de suporte utilizado, poderá ter seu funcionamento configurado rigidamente, sem possibilidade de adaptação, ou poderá dispor de uma interface através da qual alguns ajustes sejam possíveis [Kon 98]. Neste caso, o aspecto de controle de concorrência entre módulos também pode ser configurado através da linguagem de configuração utilizando-se anotações adequadas e encapsulado em conectores.

A sincronização entre módulos é caracterizada pela sincronização dos respectivos métodos que estão interagindo, interligados por conectores. Portanto, a sincronização entre métodos de módulos distintos que irão interagir, é definida pela seleção do estilo de interação a ser provido pelo conector que permitirá esta interação. Se o estilo síncrono de interação for selecionado, por exemplo, a execução do método que receberá a requisição é bloqueada até que esta tenha chegado, e o método que iniciou a

requisição é bloqueado até que uma resposta ou aviso de exceção tenha chegado. Se o estilo assíncrono fosse selecionado, por outro lado, a execução do método que iniciou a requisição continuaria sem o seu bloqueio a espera de uma resposta.

Concorrência entre métodos de um módulo (intra-módulo)

Um outro ponto do aspecto de coordenação a ser modelado é a concorrência entre os métodos de um mesmo módulo. Em determinados casos, pode ser natural que os métodos de um módulo sejam executados concorrentemente, em outros pode ser necessário que um esquema de exclusão mútua ou atomicidade na execução destes métodos seja imposto. Uma forma clássica de sincronização intra-módulo é a implementação do módulo como um monitor [Bur 95]. Neste caso, todos os métodos de um módulo são executados sob exclusão mútua. Neste esquema, um método só pode iniciar sua execução apenas quando:

- o módulo esteja ocioso,
- a execução de um método invocado anteriormente tenha sido completada,
- o método anterior foi bloqueado para esperar a ocorrência de uma condição ou evento.

Em nosso modelo, adotamos uma solução similar a apresentada em [Lop97] onde permite-se indicar subconjuntos dos métodos de um módulo que devem executar em exclusão mútua, permitindo aumentar o nível de concorrência intra-módulo. A unidade de concorrência utilizada é o método. Por exemplo, em um módulo com seis métodos, podemos configurar três métodos que devem executar em exclusão mútua entre si, outros dois métodos de forma semelhante e o método restante não será marcado. Assim, até três métodos, sendo um da cada grupo, podem ser executados concorrentemente em um dado instante. Esta especificação de configuração pode ser visualizada na figura 3.8.

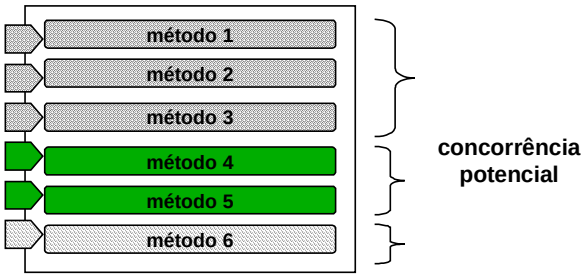


Figura 3.8 - módulo com métodos sincronizados

Chamadas concorrentes para um mesmo método podem ser tratadas de duas formas, ou são serializadas em uma fila para tratamento seqüencial pelo mesmo método, ou encarnações diferentes do mesmo método são criadas para atender diversas chamadas concomitantemente. Este comportamento pode ser configurado para cada método.

Chamadas concorrentes para um mesmo método podem ser úteis na construção de determinados servidores concorrentes em que um mesmo tipo processamento deve ser realizado, mas sobre recursos diferentes ou que não precisam ser acessados em exclusão mútua. Os métodos que precisam utilizar recursos comuns devem estar no mesmo grupo de sincronização. Métodos que não dependam de acessos a recursos compartilhados podem estar fora do grupo de sincronização. Por exemplo, em uma determinada aplicação, os métodos que atualizam uma base de dados (alteração e exclusão) seriam sincronizados. Os métodos de consulta também seriam executados em exclusão mutua com os métodos de atualização, mas poderiam executar concorrentemente entre si.

Sincronização entre métodos de um módulo (intra-módulo)

Além do gerenciamento da concorrência e exclusão mútua entre métodos de um módulo, algumas situações requerem que a execução de determinados métodos seja condicionada ao estado atual do módulo. Em nossa proposta, utilizamos o conceito de guarda de forma similar a [ADA A]. Um guarda pode controlar a admissão de requisições à porta de entrada de um método, retardando o tratamento destas requisições, até que um conjunto de condições, associadas a este guarda, seja satisfeito. Podemos ilustrar o funcionamento de um guarda como uma chave que só pode assumir os estados aberto ou fechado, um valor booleano. O estado do guarda resulta da avaliação de uma expressão que envolva variáveis que indiquem uma dada condição

relacionada com o estado atual do módulo e resulte em verdadeiro, abrindo o guarda, ou falso, fechando o guarda.

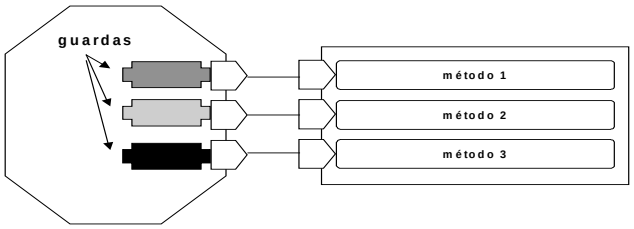
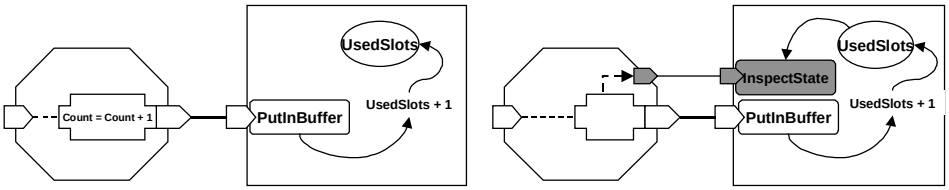


Figura 3.9 - um módulo com guardas em seus métodos

A declaração de um guarda se encaixa naturalmente na declaração de uma porta de entrada, no nível da linguagem de configuração: uma porta de entrada pode ter um guarda. Observa-se entretanto que, para manter a metodologia de retirar-se dos módulos a responsabilidade por aspectos não-funcionais, os guardas estarão contidos nos conectores, mais especificamente nas portas de saída que estarão ligadas diretamente às portas de entrada dos módulos. A adoção deste esquema visa principalmente não se interferir na estrutura dos módulos (figura 3.9). A única exceção diz respeito às variáveis internas ao módulo que precisam entrar na expressão que avalia a condição do guarda.



(a) estado mantido fora do módulo **(b) estado consultado ao módulo**

Figura 3.10 - consulta ao estado de um módulo

Em algumas situações é necessário o conhecimento do valor de algumas variáveis internas a um módulo para se obter o seu estado. Em outras situações é possível manter-se este estado externamente ao módulo guardado, mesmo que a custa de redundâncias. Por exemplo, no caso em que o número de vezes que um dado método tenha sido invocado seja importante na expressão que avalia o guarda, uma contagem pode ser feita externamente ao módulo, pelo conector que interliga a porta de entrada

do método, ao invés de se inspecionar uma variável do módulo com a mesma finalidade (figura 3.10.a). Em nossa proposta, um conector com guardas tem autonomia para manter o estado do módulo associado (figura 3.10.a) ou poderá solicitar ao módulo o valor de algumas variáveis quando for avaliar o estado do guarda através de métodos especiais de acesso, seguindo o *design pattern* ***memento*** [Gam 94] (figura 3.10.b). Detalhes sobre a sintaxe para a declaração de guardas serão apresentados na seção 4.2.7.

Observação

Nesta primeira abordagem, para o aspecto de coordenação, a unidade de concorrência e sincronização em um módulo é o método. Esquemas de sincronização com menor granularidade ainda não estão previstos. Por exemplo, ainda não é possível definir-se uma região crítica específica dentro do método que deva ser sincronizada ou executada em exclusão mútua, o que poderia aumentar um pouco o grau de concorrência. Para isso ser possível outros pontos de interação, além das portas, precisariam ser definidos, de forma que mecanismos de sincronização, como semáforos, por exemplo, pudessem ser *inseridos* corretamente no módulo.

Para contornar esta situação, um semáforo pode ser concebido como um módulo composto, utilizando-se um módulo primitivo que faria uma contagem e um conector com guardas de entrada. Esta seria tipicamente uma implementação de semáforos a partir de monitores [Ben 90]. Mesmo assim, nos parece que a utilização deste tipo de mecanismo obrigaria o projetista a abrir mão do alto nível de abstração e não traria benefícios significativos durante a concepção das aplicações. O projetista seria obrigado a definir portas de saída adicionais para fazer requisições aos métodos *wait* e *signal* no "módulo-semáforo" (figura 3 11). Uma vantagem desta solução seria a possibilidade de sincronizar métodos de módulos diferentes. O módulo-semáforo poderia residir em um nó diferente dos outros módulos da aplicação, por exemplo, e adicionalmente poderia ser configurado como tolerante a falhas.

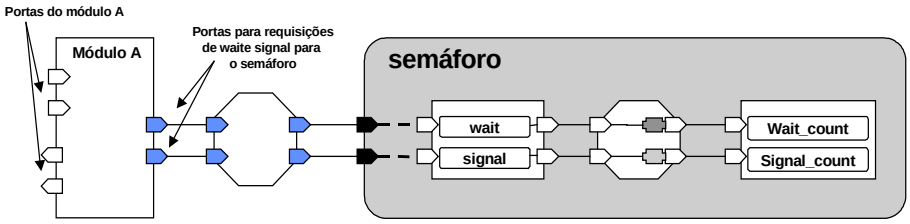


Figura 3.11 - aplicação com módulo-semáforo

Solução semelhante foi adotada, por exemplo, na versão 0.2 de AspectJ [AspJ 98C] em que pode-se introduzir primitivas de sincronização em pontos bem definidos de uma classe em Java, como na execução do método construtor, destrutor, exceções, antes e depois da execução de um método, etc. Além disso, em uma linguagem reflexiva com um MOP que permita inspecionar eventos como a leitura e escrita em variáveis, como é o caso de MetaJava avaliada no capítulo 2, seria possível controlar a sincronização no meta-nível com uma granularidade ainda mais fina. Entretanto, neste caso, o programador deverá definir e programar os pontos de inspeção e os mecanismos de sincronização a serem utilizados, a partir do meta-nível, durante a programação do objeto. Este esquema não parece ser vantajoso durante o processo de concepção dos aspectos de coordenação da aplicação, no estilo em que estamos propondo, pois o projetista teria que se preocupar com detalhes distantes do nível da arquitetura.

Em nosso modelo, o programador de um módulo não está impedido de utilizar diretamente qualquer mecanismo de sincronização disponível pela linguagem de implementação, sem passar pelo nível de configuração, mas deverá estar ciente de que estes não serão *enxergados* neste nível. Todavia, com base nas primeiras experiências com o ambiente RIO, linguagens como ADA [ADA A] e propostas similares [Ber 96T e Lop 97T], acreditamos que o esquema proposto para o tratamento do aspecto de coordenação no nível da configuração deverá atender a uma grande classe de aplicações, com a vantagem da separação de interesses e facilidade de estruturação.

3.3.6 Outros Aspectos

Além dos aspectos de distribuição, comunicação e coordenação, planejados para esta primeira versão do modelo, outros aspectos identificados na seção 3.2.1, freqüentemente utilizados na concepção de aplicações distribuídas podem ser

associados a conectores. A seguir apresentamos considerações sobre alguns deles.

Replicação e Tolerância a falhas. Em aplicações de missão crítica, são utilizadas algumas técnicas para que os serviços sejam oferecidos por um conjunto de módulos replicados, que operam com algum grau de redundância para se obter alta disponibilidade, mesmo na ocorrência de falhas em algumas réplicas. Estes serviços são ditos tolerantes a falhas. A adição de tolerância a falhas pode ser viabilizada pela configuração dos aspectos de replicação de módulos e seleção de protocolos de tolerância a falhas [Bri 95]. Para este esquema funcionar de forma transparente é necessário que o sistema de suporte ofereça uma série de funções tais mecanismos para gerenciamento de réplicas, e esquemas de comunicação em grupo.

Em [Bri 95] vários protocolos de tolerância a falhas foram avaliados no ambiente RIO. A tolerância a falhas foi obtida por configuração, através da composição dos módulos replicados da aplicação, com módulos especializados em manter o estado das réplicas e implementar as ações especificadas pelo protocolo de tolerância a falhas. Além disso, o suporte à configuração oferecia difusão confiável atômica [Szt 94], facilitando a manutenção da consistência de réplicas.

Em nossa proposta, é possível adicionarmos à linguagem de configuração as extensões necessárias para indicar os módulos que devem ser replicados e o esquema de tolerância a falhas a ser usado. Por exemplo, a indicação da replicação de um módulo seria feita no momento de instanciá-lo, quando o número de réplicas e a localização destas réplicas seriam informados. Além da indicação do módulo a ser replicado, é necessário um conector que ofereça o suporte adequado para interligar o grupo de réplicas aos clientes deste grupo. Este conector deve apresentar algumas características desejáveis para que a tolerância a falhas também possa ser configurada. Por exemplo, características como a difusão confiável para a comunicação do grupo de réplicas, protocolos de manutenção do estado de consistência das réplicas poderiam estar disponíveis neste conector. Além disto, este conector deve estar programado para disponibilizar esquemas de gerenciamento da tolerância a falhas, bem como o esquema de chaveamento entre réplicas ativas e passivas [Bri 95]. Assim, ao se conceber a aplicação, bastaria que a combinação de características mais apropriadas fosse selecionada na configuração deste conector, de forma que os aspectos de tolerância a falhas pudessem operar como desejado. Em [Loq 97] este esquema para configurar

tolerância a falhas na aplicação, através de elementos de conexão suportando tolerância a falhas foi implementando. Este experimento utilizou como sistema de suporte uma implementação de CORBA com extensões para comunicação de grupo [Maf 96].

Prioridades. A abrangência do aspecto de coordenação pode ser ampliada oferecendo-se um esquema para configurar prioridades para as portas de entrada de um modulo. Este esquema segue a mesma linha de concepção estabelecida para os guardas. Com uma extensão da linguagem de configuração, pode-se indicar um peso ou índice de prioridade para cada porta de entrada de um modulo. No caso de mensagens chegando concorrentemente para portas diferentes deste modulo, a prioridade de cada porta indicará qual delas terá preferência para repassar a requisição para o respectivo método.

Aplicações Paralelas. As aplicações paralelas apresentam aspectos de comunicação e coordenação específicos [Car 98], cujas estruturas também podem ser descritas através da linguagem de configuração. Assim, no aspecto de comunicação podem ser definidos estilos de interação como *scatter-gather*, *all-gather*, entre outros, utilizados em várias classes de aplicações paralelas. Da mesma forma, esquemas de sincronização, típicos de aplicações paralelas, como *barriers*, podem ser introduzidos no aspecto de coordenação.

Em sistemas que oferecem suporte para memória compartilhada, conectores especializados para a interação por memória compartilhada distribuída podem ser desenvolvidos. Uma das vantagens é a possibilidade de se configurar, no conector, o protocolo para manutenção de consistência, ou a política de escrita a ser usada pela aplicação. Outra possibilidade é a indicação, no nível da configuração, de outros ajustes para este mecanismo, como a seleção de implementações diferentes dos protocolos de manutenção de consistência, dependendo da arquitetura utilizada. Por exemplo, uma aplicação que executará em um *cluster* de estações ligadas por rede local pode utilizar um conector diferente de uma aplicação que executará em máquinas paralelas, ainda que as duas utilizem memória compartilhada [Loq 98].

Persistência. Permite configurar aspectos de persistência para os módulos. Este aspecto pode ser utilizado em conjunto com o aspecto de tolerância a falhas para facilitar a implementação de esquemas de recuperação de módulos que falharam e em sistemas de suporte à transações [Iss 96];

Tempo Real. Características relacionadas com restrições temporais ou o escalonamento de módulos [Fra 96] podem ser especificados através do uso de conectores.

Filtros. Em determinadas situações, pode ocorrer que o conjunto de conectores disponíveis, já desenvolvidos, não consiga atender a certos aspectos de uma aplicação ou que não seja possível desenvolver novos conectores que atendam a estes aspectos. Por exemplo, em aplicações em que seja necessário fazer críticas e verificações nos dados que estão sendo passados para os módulos servidores antes que estes sejam utilizados pelos métodos, a introdução das rotinas de verificação no conector que interliga estes módulos solucionaria o problema. Exemplos de aspectos semelhantes a este, como a criptografia, compressão ou conversão de representação de tipos, que operam sobre o conteúdo das requisições sendo enviadas de um módulo a outro, poderiam ser introduzidos nos conectores sem que houvesse a necessidade de se desenvolver um novo conector para contemplar estes novos aspectos. Para permitir estas adaptações em conectores já desenvolvidos, introduzimos o conceito de **filtro**.

Filtros podem ser configurados em portas de entrada ou saída de um conector. Um filtro funciona interceptando as informações que estão entrando ou saindo da porta e realizando alguma operação com estas informações. Cada filtro é constituído por um conjunto de expressões que manipulam os as instâncias dos tipos dos parâmetros declarados na porta. Em princípio, são permitidas operações matemáticas, booleanas, atribuições e expressões condicionais (if-then-else). Estas expressões somente podem envolver o conjunto de parâmetros declarados na porta onde o filtro está sendo configurado. Entretanto, se houver o suporte adequado, é possível que as expressões declaradas na configuração dos filtros possam conter trechos de código em uma linguagem de programação. O esquema dos filtros é representado na figura 3.12.

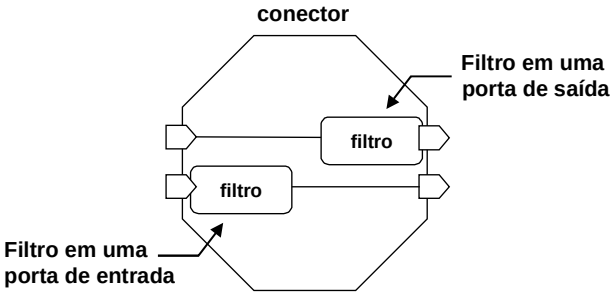


Figura 3.12 - filtros em portas de entrada e saída em um conector

Deve-se considerar o uso de filtro como uma alternativa ao uso de conectores implementados especificamente para atender aos aspectos necessários Esta opção pode se dar quando o esforço para realização destes conectores não seja compensatória ou no caso em que as características desejadas não representem propriamente um aspecto trivial, o que levaria a baixa reusabilidade. Na próxima seção uma outra técnica, a composição de conectores, é discutida como alternativa para a descrição de aspectos não-funcionais. A composição de conectores pode ser usada em situações onde os aspectos necessários estão disponíveis em conectores diferentes ou em aplicações que necessitam de ajustes não usuais.

3.3.7 Composição de Conectores

Conectores podem ser considerados como módulos especializados em promover a interação entre outros módulos. Entretanto, a separação dos conceitos de módulos e conectores se justifica com a possibilidade de se associar formalismos para a verificação automática de propriedades das aplicações sendo configuradas [Agh 98]. Assim sendo, a concepção de um conector composto através da composição de conectores mais simples também é possível. A mecânica da composição de conectores é similar à composição de módulos. A diferença principal é que estruturalmente um conector composto é formado apenas pela ligação de conectores, como será explicado adiante.

Recorremos novamente ao exemplo simples de uma aplicação com dois módulos e um conector. Na figura 3.13, existe um módulo Y, com vários métodos que oferecem serviços através de suas respectivas portas de entrada. Um módulo X requisita os serviços do módulo Y através de portas de saída. Um conector α interliga os pares de porta de entrada-saída compatíveis dos dois módulos. Com o conector α todas as portas compatíveis dos módulos X e Y podem ser ligadas automaticamente. Em conseqüência, as características definidas neste conector afetam igualmente todas as portas de saída e entrada, o que é refletido no comportamento das diversas interações entre os dois módulos.

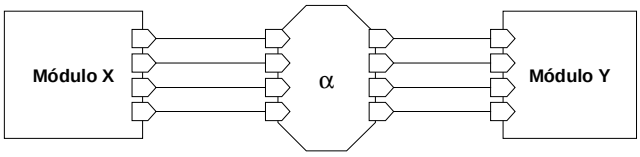


Figura 3.13 - um conector interligando vários serviços do módulo Y às requisições do módulo X

Uma possibilidade imediata para aumentar a granularidade no ajuste das conexões é desmembrar um conector como o da figura 3.13 em vários conectores independentes. Cada par de portas de entrada-saída de X e Y é interligado por um conector independente. Em cada conector podemos configurar aspectos diferentes ou selecionar conectores diferentes para cada interconexão.

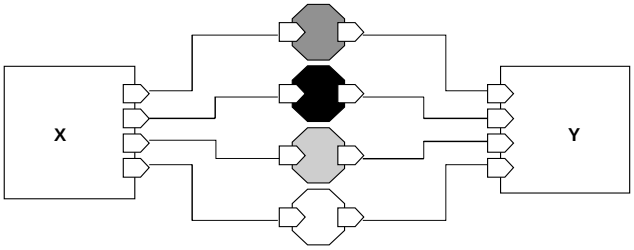


Figura 3.14 - conectores diferentes ligando dois módulos

Uma alternativa ao desmembramento de um conector com várias portas é a introdução de filtros, como sugerido na seção 3.3.6, nas portas do conector onde ajustes se fizerem necessários. Entretanto, o mecanismo de filtros não é adequado, em princípio, para a implementação de aspectos inteiramente novos ou de grande complexidade.

Supondo-se, agora, que para a mesma aplicação da figura 3.13 seja exigido um ajuste fino nos aspectos das interações entre módulos e que os ajustes devam ser feitos de maneira muito diversa para cada fluxo de interação. O uso de filtros possivelmente estaria descartado devido a complexidade dos ajustes. Uma solução possível é o desenvolvimento de um novo conector para atender aos requisitos, mas isso poderia não ser produtivo no momento. Supondo-se então, que alguns conectores que atendam à parte dos requisitos individualmente estejam disponíveis. Neste caso, seria possível selecionar um conector que atendesse mais proximamente aos requisitos da aplicação para cada conexão, mas a solução estaria incompleta. Entretanto, se os conectores disponíveis, em conjunto, pudessem de alguma forma solucionar integralmente o

problema, a composição de conectores poderia ser a solução mais adequada para esta situação.

A figura 3.15 apresenta um exemplo de um conector composto que poderia substituir o conector α da figura 3.13. O conector composto, entretanto, possui características adicionais. Observa-se às portas de saída do conector α foram ligados os conectores β, χ, δ e ε. Cada um destes conectores apresenta aspectos diferentes. Por exemplo, o conector β utiliza o protocolo UDP e o conector χ utiliza o protocolo RTP. Ligado ao conector χ, encontra-se o conector ϕ. Este conector, por exemplo , comprime o fluxo de informações. O resultado, portanto, é um conector adaptado exatamente às necessidades da aplicação, construído a partir de outros conectores disponíveis.

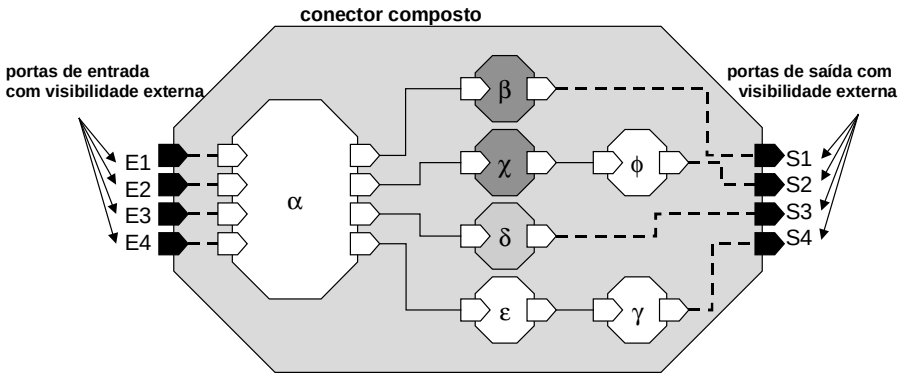


Figura 3.15 - um conector composto

Na linguagem de configuração a sintaxe utilizada para declarar a composição de conectores é similar a composição de módulos, com a diferença de que na composição de conectores somente conectores são utilizados, sem a presença de módulos. Na descrição de um conector composto, aninham-se instâncias de conectores, estabelecem-se as ligações entre estas instâncias e exportam-se as portas que terão visibilidade externa. O mesmo princípio de reusabilidade aplicado a módulos compostos é aplicado para conectores compostos. Uma outra vantagem é a possibilidade de se testar o comportamento de uma aplicação com conectores composto para em seguida, em caso satisfatório, desenvolver um único conector com todas as caraterísticas desejadas à título de eficiência.

Embora a idéia da composição de conectores seja simples, existem algumas

implicações que devem ser levantadas. Alguns dos problemas que podem advir da composição de conectores foram mencionados no final da seção 3.3.4, quando foi sugerido que a composição de conectores poderia ser usada para o aspecto de comunicação. Basicamente os problemas da composição de conectores são:

- verificar a ortogonalidade, ausência de conflitos ou interações indesejadas dos aspectos sendo compostos para garantir a coerência do resultado [Joh 97, Mon 97, Jac 98];
- dispor de ferramentas para facilitar a composição automática de conectores, a partir de características descritas na configuração;
- verificar se os aspectos configurados de forma composta têm suporte no sistema onde a aplicação será executada.

Além destes pontos existem considerações sobre a eficiência. Conceitualmente, o comportamento de um conector composto é idêntico ao de um único conector que contenha as mesmas características. Entretanto, dependendo do mapeamento de um conector em uma implementação, no caso do conector composto, podem ser identificados algumas fontes de ineficiência, tais como, transferências de dados entre espaços de endereçamento diferentes e escalonamento, entre outras. Por outro lado, pode-se prototipar uma aplicação com conectores compostos para em seguida desenvolver um único conector com todas as caraterísticas desejadas. Isto pode ser automatizado por um compilador especializado, que faria otimizações a partir da aplicação sendo descrita.

A solução para estes problemas não parece ser trivial, mas verifica-se que algumas composições podem ser feitas sem aparentes problemas. Por exemplo, a composição de um conector onde aspectos de coordenação tenham sido definidos e um conector onde aspectos de comunicação tenham sido declarados não oferece dificuldades. Questões relativas a composição de conectores serão aprofundadas futuramente.

3.3.8 Conectores Genéricos

Tradicionalmente, um conector é concebido para permitir a interação entre módulos, que possuem um conjunto de portas de entrada e saída, cada qual com sua assinatura. Assim, existe uma dependência de interfaces, e o conector deve conhecê-las para poder

gerenciar a interações. Um conector RMI, por exemplo, só servirá para o conjunto de portas para o qual ele foi preparado. Para aumentar a possibilidade de reutilização, R-RIO utiliza o conceito de conector genérico Esta alternativa permite, para certos tipos de conectores, maior flexibilidade no desenvolvimento de aplicações, uma vez que um conector é concebido independentemente do conjunto de portas dos módulos que ele irá conectar. Por exemplo, uma vez implementado um conector que faz comunicação utilizando RMI, esse mesmo conector pode ser utilizado para ligar quaisquer pares de módulos, de um lado sem a necessidade do uso de bibliotecas por parte dos módulos envolvidos na comunicação, e do outro sem a necessidade de declaração de interface e geração de *proxys* e *stubs*.

Observação

Até este ponto apresentamos as bases de nossa proposta. Destacamos o modelo básico e alguns aspectos não-funcionais que podem ser utilizados na concepção de uma aplicação. A discussão que se segue, avalia e compara o modelo proposto segundo os requisitos levantados no capítulo 1 e com as outras propostas estudadas no capítulo 2. Esta discussão tem por objetivo verificar os pontos relevantes de nossa proposta e criar subsídios para a avaliação da linguagem de configuração e exemplos apresentados no próximo capítulo.

3.4 Avaliação

No capítulo anterior foram selecionadas e avaliadas algumas propostas que representam de alguma forma as tecnologias candidatas para o desenvolvimento de sistemas distribuídos e possuem correlação com a nossa proposta. Metajava foi escolhida como representante de um ambiente implementando reflexão computacional dinâmica e OpenC++ como linguagem oferecendo reflexão estática. Os Filtros de Composição, em uma abordagem diferente, se enquadra em um modelo proposto para a utilização de tecnologia de objetos, oferecendo alguns recursos para a obtenção da separação de aspectos através de reflexão restrita e utilizando a composição para a montagem de aplicações. Em seguida foi analisada a linguagem D, uma implementação da idéia do AOP. Durante os testes, a linguagem disponível para avaliação foi a AspectJ 0.1, uma

adaptação do framework D, disponibilizadas como versão Beta e implementada na no centro de pesquisas da Xerox. Por último, CORBA-IDL também foi utilizada na avaliação para ampliar o espectro das escolhas. Duas linguagens representantes de propostas de configuração também foram avaliadas. Darwin e UniCon foram selecionadas para que a comparação com a nossa proposta pudesse ser enriquecida. A escolha destas propostas foi proposital visando estabelecer elementos que permitissem uma avaliação comparativa. Esta avaliação leva em consideração os aspectos conceituais e práticos destas propostas e será complementada no capítulo 4 com o exemplo baseado na aplicação do *buffer* limitado desenvolvido segundo nossa proposta. Com isso, as seguintes observações podem ser feitas:

em relação a estrutura das aplicações desenvolvidas

Nas conclusões dos capítulos 1 e 2 salientou-se a importância de que as ferramentas utilizadas para a concepção de aplicações distribuídas devem oferecer mecanismos para se descrever a estrutura das aplicações. Reflexão, CF e AOP não contribuem diretamente para facilitar a descrição da estrutura de uma aplicação. A utilização de MOPs, a programação de filtros e a descrição de aspectos em AOP deve ser cuidadosamente planejada no nível da programação para espelhar a estrutura de uma aplicação. Embora estas propostas valorizem a separação de aspectos, elas não facilitam diretamente o projeto de grandes aplicações.

Sistemas mais complexos que utilizam reflexão como técnica de suporte, devem ser planejados através de meta-arquiteturas, que normalmente utilizam algum padrão de projeto (*design pattern*) para implementar e integrar os meta-objetos. Isso não invalida a reflexão computacional como instrumento de extensão e desenvolvimento baseado na separação de interesses. Se a meta-arquitetura for genérica o suficiente para ser associada a sistemas comuns e dotá-los com as características nela contidas, as possibilidades de utilização de reflexão serão ampliadas. A proposta de configuração segue, em termos, esta última idéia: o sistema de suporte à configuração pode ser concebido como uma arquitetura de meta-nível. Por exemplo, os comandos de configuração para instanciar e ligar os módulos ou a seleção de um dado conector para interligar os módulos, atuam sobre a estrutura e o comportamento da aplicação em um nível diferente da funcionalidade da aplicação.

Embora nossa proposta de configuração não tenha relação direta com o modelo de objetos adotado em CF, as duas abordagens tem em comum a utilização de composição para a descrição da estrutura de uma aplicação. Em Sina, por exemplo, é possível indicar explicitamente para cada classe, objetos com os quais existe relacionamento por delegação e, em conjunto com o esquema de filtros, também programados para a classe, emular outros tipos de relacionamento. As aplicações, então, são compostas por instâncias de objetos que interagem segundo as indicações de relacionamento programadas nas respectivas classes. Podemos comparar, de certa forma, a combinação de filtros envolvidos na comunicação entre dois objetos com o conector de nossa proposta de configuração. Os dois mecanismos procuram expressar o relacionamento de duas entidades segundo especificações de interfaces e aspectos não-funcionais relacionados com a interação entre componentes.

Uma das diferenças entre CF e a nossa proposta é o fato de que na configuração o relacionamento entre módulos e o estilo com o qual este relacionamento ocorre não é indicado na programação do módulo. De acordo com o conceito de portas, apenas a assinatura e o sentido básico das requisições (de entrada ou saída) tem que ser conhecido pelo módulo. Aspectos específicos da interação entre módulos são completamente definidos nos conectores, que são adicionados externamente ao módulo independentemente de considerações no nível da programação. Em [Ber 96T – seção 4.4] é sugerido um método para a projeto de software que utiliza composição e procura separar aspectos de sincronização da funcionalidade dos objetos, para complementar o modelo de objetos proposto por CF. Com o resultado deste método a aplicação composta é traduzida para Sina, onde os aspectos de sincronização terminam por ser embutidos no código através dos filtros.

A abordagem utilizada para separação de interesses permite, em princípio, que aspectos funcionais e não-funcionais sejam isolados no nível da programação interna dos elementos. Entretanto, aspectos funcionais e não-funcionais fazem parte de uma estrutura maior que é a aplicação. Na concepção da linguagem de configuração, optamos por não fazer a descrição destes aspectos isoladamente como em AOP, onde cada aspecto é programado através de uma linguagem especializada e a integração entre as linguagens é feita somente por referências externas (observe-se o exemplo da seção 2.4 no capítulo 2 em que na programação de guardas é necessário uma referência ao

método sob sua supervisão). Assim, com a linguagem de configuração é possível a descrição de aspectos funcionais e não-funcionais mantendo-se a visão da estrutura da aplicação como um todo.

em relação a abrangência

Um dos problemas fundamentais na utilização das propostas estudadas é relacionado com a abrangência. A proposta de Metajava é baseada em uma extensão do interpretador Java e uma biblioteca de classes que oferece o acesso ao MOP. Esta solução restringe o desenvolvimento da aplicação à utilização da linguagem Java e à execução com o interpretador estendido. Não é possível, por exemplo, o desenvolvimento de parte de uma aplicação em uma linguagem de programação diferente de Java. Em geral linguagens e ambientes de execução que apresentam características reflexivas sofrem deste mesmo problema. O problema, em princípio, não está na tecnologia, mas na falta de padronização.

AOP possui restrições ainda mais acentuadas. A linguagem e o *entrelaçador* utilizados na avaliação são intimamente associados a Java. Tanto a linguagem para a programação da parte funcional da aplicação (JCore), quanto as linguagens para programação dos aspectos de sincronização (COOL) e distribuição (RIDL) foram concebidas visando uma arquitetura alvo, no caso, Java. Embora a idéia de AOP possa ser implantada potencialmente em qualquer ambiente, não foi considerada sua utilização para integrar partes de uma aplicação, programadas em diferentes linguagens, e entrelaçá-las com os códigos de outros aspectos. O *weaver* entrelaça código fonte e gera código fonte. Se o entrelaçamento pudesse ser feito em um outro nível, em que houvesse alguma padronização (por exemplo, entrelaçando módulos ou partes de programa compilados como *shared objects* ou *DLLs*, que são padronizados), seria possível reduzir as restrições ao uso do AOP.

O paradigma de configuração potencialmente permite integrar módulos desenvolvidos em qualquer linguagem de programação, desde que a interface deste módulo seja bem definida. Isto é possível a partir de um mapeamento da arquitetura para uma linguagem de programação, como descrito na seção 3.2. Esta integração é completada pelos conectores que também podem ser construídos a partir de diversas tecnologias, desde que estes conectores também apresentem interfaces bem definidas. A definição da

configuração de uma aplicação é independente da linguagem de programação com a qual seus elementos serão implementados.

De uma forma geral todas as propostas avaliadas possuem os recursos necessários para o desenvolvimento de aplicações distribuídas. Aspectos como distribuição e comunicação estão disponíveis ou podem ser implantados em todas elas. Aspectos de coordenação não estão disponíveis, entretanto, nas propostas baseadas em configuração avaliadas. A combinação da configuração com a descrição de aspectos não-funcionais permite que as aplicações sejam concebidas mais facilmente. Em nossa proposta, aliamos a esta combinação, semânticas convenientes para os aspectos disponibilizados na linguagem de configuração, resultadas de pesquisas e experimentos preliminares.

possibilidade de evolução da abordagem

Entende-se evolução da abordagem como a possibilidade de determinada proposta permitir facilmente que novos aspectos sejam incorporados a mesma. Com relação a esta questão todos as tecnologias estudadas se apresentam deficiências. Um dos conceitos a serem investigados e possivelmente contemplado em nosso ambiente de configuração é a elaboração de uma solução para a integração de novos conectores, como apresentaremos oportunamente.

No caso de Metajava, a criação de novos meta-objetos é relativamente simples. O programador precisa conhecer o MOP oferecido e desenvolver o código da meta-classe, gerando um classe em Java que será interpretada pela Máquina Virtual Java estendida para oferecer reflexão. Entretanto, como já avaliado, a extensão de meta-objetos que já estejam instanciados em uma aplicação não é trivial. Seria necessário salvar o estado do meta-objeto, destruí-lo e instanciar uma nova versão. Em Java isto não é possível, inicialmente, pois não existem meios para se destruir um objeto (em contrapartida à primitiva *new*). Além disso, a evolução de um MOP é impossível de ser realizada neste contexto. O MOP normalmente oferece um conjunto de opções para se estabelecer a ligação entre o nível base e o meta-nível. Como visto no exemplo da seção 2.2.2 a classe *Metaobject* de MetaJava oferece um método para a interceptação de chamadas aos métodos da classe base ao qual ela tenha sido ligada. Se um outro tipo de vínculo, que não exista na implementação da linguagem, for necessário, nada poderá ser feito por um programador de aplicação. O interpretador, neste caso, deverá ser modificado.

Uma das soluções para este problema é a utilização de linguagens abertas. Na seção 2.2.1 verificou-se que OpenC++ permite que extensões sintáticas sejam criadas a partir de C++ com o MOP oferecido. OpenC++ pode ser considerada uma linguagem aberta para a introdução de extensões que permitem a evolução da própria linguagem de programação, mas sua utilização não é trivial.

Em AOP, a integração de novos aspectos também não é trivial. Por exemplo, se um aspecto de tolerância a falhas tiver que ser integrado, uma nova linguagem para descrever este aspecto deverá ser criada, e o *weaver* terá que ser re-escrito para entrelaçar de forma correta mais este aspecto. Em [And 97] é sugerido que se utilize uma abordagem similar às linguagens de programação que admitem reflexão estrutural ou linguagens abertas para se introduzir extensões sintáticas e semânticas no entrelaçador. Esta característica parece estar sendo integrada nas versões mais recentes de AspectJ [AspJ 98C]. A tendência parece ser a utilização de uma única linguagem para programar aspectos, em contrapartida à filosofia original de contar uma linguagem diferente para cada aspecto. Neste novo esquema, todos os aspectos são programados através de uma única linguagem e o programador fica encarregado de indicar os pontos de junção dos diversos aspectos no código base. Fazendo uma rápida análise deste novo esquema para AOP identificam-se muitas semelhanças com linguagens reflexivas.

Embora CF ofereça características de composição, as aplicações só podem utilizar os tipos de filtros disponíveis na linguagem para implementar aspectos não-funcionais. No caso de Sina, a linguagem de programação orientada a objetos que implementa os conceitos de CF, apenas 3 tipos de filtros foram implementados. Embora vários problemas possam ser resolvidos com estes recursos disponíveis, pode haver necessidade de maior expressividade. Em [Bur 97T] foi proposta a implementação de CF sobre CORBA. Entendemos que uma arquitetura baseada nos conceitos de CF implantada sobre CORBA tende a ser mais flexível e aberta a adição de novos tipos de filtro e poderá ter um futuro promissor.

De forma ainda pouco rigorosa, em nossa proposta a integração de novos aspectos é contemplada pela utilização de novos conectores. Conectores com as características necessárias para uma aplicação específica podem ser implementados e disponibilizados para configuração. Isso é possível porque a interface deste conector estará bem definida pelas portas e porque existe uma referência no nível de configuração mapeando a

implementação do conector (cláusula ***map*** declarada nos conectores e módulos). Entretanto, esta solução, embora permita a descrição da estrutura da aplicação com novos conectores, impossibilita que os outros benefícios de um ambiente de configuração sejam usufruídos, como por exemplo a verificação de corretude semântica da configuração, a verificação de propriedades com relação a sincronização, restrições temporais ou viabilidade de recursos a serem utilizados pelo conector. Neste caso, o conector é opaco para o ambiente de configuração que nada pode *falar* sobre ele. No capítulo 5 esta discussão será ampliada e algumas soluções acadêmicas, como apresentadas em [Zel 97] serão analisadas antes de apresentarmos a nossa abordagem.

comparação das propostas

Para finalizar as considerações apresentadas nesta subseção, as propostas avaliadas no capítulo 2 e a proposta apresentada neste capítulo são comparadas resumidamente segundo os parâmetros do capítulo 1 na tabela 3.1.

	Reusabilidade
OpenC++	É possível a reutilização das extensões sintáticas e meta-classes se forem planejadas de forma modular. O código base pode ser reutilizado em outras aplicações e adaptado por meta-objetos. O código fonte no nível base deve estar disponível ou uma biblioteca com as classes do nível-base devem ser oferecidas. Se os objetos de meta-nível forem os mesmos, o pré-processador com sintaxe estendida pode ser reutilizado.
MetaJava	Permite reusar nível-base e meta-nível.
CF	Alguns tipos de filtros são disponíveis. Instâncias de filtros não são entidades independentes, portanto não podem ser reutilizados. O corpo de cada filtro deve ser reescrito para cada objeto.
AOP	A reutilização de código em AOP é equivalente a da tecnologia de objetos. Os objetos relativos aos aspectos funcionais de uma aplicação podem ser reutilizados. Entretanto a reutilização da aplicação com seus aspectos funcionais e não-funcionais já entrelaçados não é possível. A reutilização da definição de aspectos poderá se feita se estes forem suficientemente genéricos.
Regis	Componentes podem ser reutilizados.
UniCon	Componentes e conectores podem ser reutilizados.
	Separação de Interesses
OpenC++	Sim. É obtida na medida em que aspectos funcionais e não-funcionais são separados em objetos e meta-objetos.
MetaJava	Sim. Aspectos funcionais e não-funcionais são separados em objetos e meta-objetos.
CF	Sim. O núcleo do objeto implementa a funcionalidade da aplicação. Filtros implementam aspectos não-funcionais. Filtros podem ser encadeados, permitindo a ação de filtros consecutivos para compor um comportamento específico. Aspectos não-funcionais só

	podem ser programados se filtros de tipos adequados estiverem disponíveis na linguagem.
AOP	Sim. Aspectos não funcionais são definidos através de linguagens separadas
Regis	Não. Aspectos devem ser programados nos componentes
UniCon	Limitado. UniCon permite a configuração de propriedades associadas estaticamente a módulos, conectores e portas pela linguagem de configuração que possui alguns tipos de anotações e parâmetros para esta finalidade.
	Evolução Dinâmica
OpenC++	Parcial. A associação dos níveis base e meta são feitas durante a fase de pré-processamento. Novas associações não podem ser feitas durante a execução da aplicação. Portanto, a aplicação não pode evoluir dinamicamente. Entretanto, a linguagem de programação pode ter sua sintaxe estendida, o que representa uma evolução em um outro contexto, mas não evolução dinâmica da aplicação.
MetaJava	Permite. Adaptando-se os meta-objetos e refazendo-se a ligação entre objeto base e meta-objeto em tempo de execução.
CF	Parcialmente. Se o objeto deve ser adaptado, sua interface tem que ser modificada. Novos filtros não podem ser acrescentados dinamicamente à interface do objeto. Entretanto os filtros permitem que o comportamento do objeto seja adaptado segundo as mensagens que chegam `a interface e segundo o comportamento dos filtros e estado do próprio objeto.
AOP	Não permite. A aplicação terá que ser reprogramada para admitir nova funcionalidade.
Regis	Sim. Através de reconfiguração da descrição da configuração ou durante a execução da aplicação.
UniCon	Não. Após a geração de código executável, a estrutura de componentes e conectores passa a não existir, ao menos na versão avaliada. Entretanto, uma configuração original pode ser alterada para uma nova configuração, mas novo código executável deve ser gerado.
	Abrangência
OpenC++	Linguagem especializada. Somente para C++. O programador deve conhecer técnicas de geração de código para tirar melhor proveito da ferramenta.
MetaJava	Tem a vantagem de ser compatível com Java. Entretanto, o interpretador teve que ser modificado. Terá que se adaptar a cada nova versão de Java ou a cada possibilidade a ser acrescentada ao MOP.
CF	Pouca abrangência. A aplicação das idéias demanda uma linguagem especializada e o acréscimo de novos filtros não é factível por desenvolvedores de aplicação. Entretanto as idéias foram adaptadas para CORBA, utilizando-se as características proprietárias de filtros do Orbix [IONA 92A], uma implementação comercial de CORBA em [Bur 97T], aumentando as possibilidades de abrangência.
AOP	Ainda em estudo. Linguagens particulares para cada aspecto. Para o acréscimo de novos aspectos, novas linguagens deve ser concebidas e o *weaver* modificado. A versão de AspectJ, a implementação patrocinada pela Xerox, atualmente disponível, tende a unificar todas as linguagens de aspectos e não especializar a sintaxe para cada aspecto.
Regis	Permite a concepção de uma grande gama de aplicações. Linguagem de configuração particular (Darwin). Os conceitos de Regis e Darwin, estão sendo adaptado para CORBA [ROI 98].
UniCon	Permite a concepção de uma grande gama de aplicações. Linguagem de configuração particular (UniCon). Tipos de conectores pré-definidos na linguagem permitem o ajuste de parâmetros e a utilização de ferramentas para a extração de propriedades da aplicação. Entretanto, o acréscimo de novos conectores não é trivial.

Tabela 3.1: comparação das propostas avaliadas

No próximo capítulo, após a apresentação da linguagem de configuração e alguns exemplos, a tabela 3.1 será complementada com as observações a cerca de nossa proposta.

3.5 Conclusão

Neste capítulo foi apresentado o modelo de configuração orientado à separação de interesses, objeto central deste trabalho. Dentro deste modelo, procurou-se ressaltar os conceitos de módulos, conectores e portas, adotados para a construção de aplicações distribuídas. Uma metodologia para a concepção de arquiteturas de aplicações também foi proposta. Esta metodologia permite que se descrevam separadamente aspectos funcionais e não-funcionais que vão integrar a arquitetura da aplicação

Três aspectos não-funcionais, comunicação, coordenação e distribuição foram apresentados como parte da abordagem. Através de exemplos simples, os conceitos do modelo e a configuração destes aspectos não-funcionais foram aplicados. Observou-se que a possibilidade de seleção e composição de elementos na descrição da estrutura de uma aplicação e a facilidade de configuração de aspectos não-funcionais favorece a reutilização de componentes e a concepção de aplicações customizadas para necessidades específicas.

A imposição da modularidade na concepção de elementos funcionais e o encapsulamento de aspectos não-funcionais importantes nos conectores, permite que módulos e conectores possam ser utilizados na composição de vários componentes. O fato dos módulos poderem ser concebidos sem entrelaçamento com aspectos não-funcionais aumenta as possibilidades de reutilização dos mesmos em aplicações diferentes, pois estes não estarão *amarrados* a características de operação específicas.

Módulos e conectores concebidos em algum momento podem ser armazenados para uso futuro. Estes elementos podem ser reutilizados na construção de aplicações distintas ou na construção de módulos compostos (e de funcionalidade mais complexa) através da interconexão de módulos básicos selecionados.

Um módulo especializado poderá ser projetado, por composição, a partir de módulos mais simples disponíveis ou pela interconexão de módulos disponíveis por conectores com as características necessárias. Em uma aplicação multimídia, podemos construir

um módulo *fluxo de vídeo codificado* se pudermos dispor de um módulo *fonte de vídeo*, um módulo *codificador* e um conector que implemente comunicação por fluxo [Szt 97].

O reuso de aplicações inteiras ou partes de aplicações também é permitido. Podemos imaginar o exemplo de uma pequena aplicação chamada "célula-de-manufatura" configurada com alguns módulos suficientes para implementar uma célula de manufatura. Esta aplicação poderia ser reutilizada para se construir uma aplicação "chão-de-fábrica" com células de manufatura interligadas por conectores que representariam, na arquitetura da aplicação, o fluxo do chão de fábrica.

O próximo capítulo apresenta uma proposta para a linguagem de configuração através da qual descreve-se a arquitetura de uma aplicação. Logo em seguida uma série de exemplos são apresentados. Durante a discussão da linguagem de configuração, na seção 4.2 do capítulo 4, os conceitos do modelo passam a ser materializados com a introdução da sintaxe para a declaração de módulos, portas e conectores. Da mesma forma, serão apresentadas as soluções inseridas na linguagem de configuração para a declaração de alguns aspectos não-funcionais e detalhes da semântica destas declarações serão também discutidos. Ainda nesta seção, a apresentação dos comandos para a composição da estrutura das aplicações fecha o ciclo necessário para podermos desenvolver os exemplos da seção seguinte.

Os exemplos desenvolvidos na seção 4.3 exploram as possibilidades da linguagem, através da composição de módulos e da utilização dos aspectos de distribuição e coordenação. Estes exemplos giram em torno do problema produtor-consumidor com *buffer* limitado, em várias versões, onde novos aspectos vão sendo adicionados gradativamente. Em cada versão algumas facetas da linguagem de configuração são exploradas para atender aos requisitos introduzidos. Com isso, as discussões apresentadas neste capítulo serão ampliadas, incorporando-se os exemplos.

Capítulo IV

A Linguagem de Configuração

4.1 Introdução

Antes de iniciar a descrição de uma aplicação através da linguagem de configuração, algumas etapas preliminares devem ser concluídas. O projetista precisa ter uma visão global da mecânica de projeto utilizada na abordagem proposta. Esta mecânica está esquematizada na figura 4.1. Segundo este esquema, a concepção de uma aplicação passa por cinco etapas:

1. A demanda pela concepção de uma aplicação é originada por um usuário potencial ou situação de necessidade diagnosticada. Inicialmente a aplicação é apenas uma idéia, geralmente obtida a partir de uma análise de requisitos.

O levantamento de requisitos pode ser feito com o auxílio de técnicas disponíveis e, um modelo da concepção da aplicação pode ser elaborado através de ferramentas como a UML [Boo 97]. Este modelo poderá passar por várias etapas de refinamento e resultará em uma arquitetura para aplicação. No projeto desta arquitetura, o modelo conceitual da aplicação é traduzindo para uma estrutura composta de módulos, conectores, a interligação destes elementos e aspectos não-funcionais. Neste ponto a aplicação pode ser configurada.

2. Através da linguagem de configuração, toda arquitetura da aplicação é, então, descrita. O programador poderá utilizar módulos e conectores previamente concebidos e disponíveis em bibliotecas para projetar a aplicação. O projeto da aplicação poderá ser acelerado a partir de *frameworks*, se disponíveis, valendo-se dos recursos de composição e ajustando aspectos não-funcionais.

No caso em que os componentes utilizados na configuração ainda não tenham sido

implementados, esqueletos de código são gerados automaticamente em uma linguagem de programação selecionada. Se as ferramentas para geração de código não estiverem disponíveis ou se estas não puderem gerar código para determinada linguagem de programação, a implementação de componentes pode ser feita manualmente, seguindo-se as convenções e receitas do modelo para a geração de módulos e conectores.

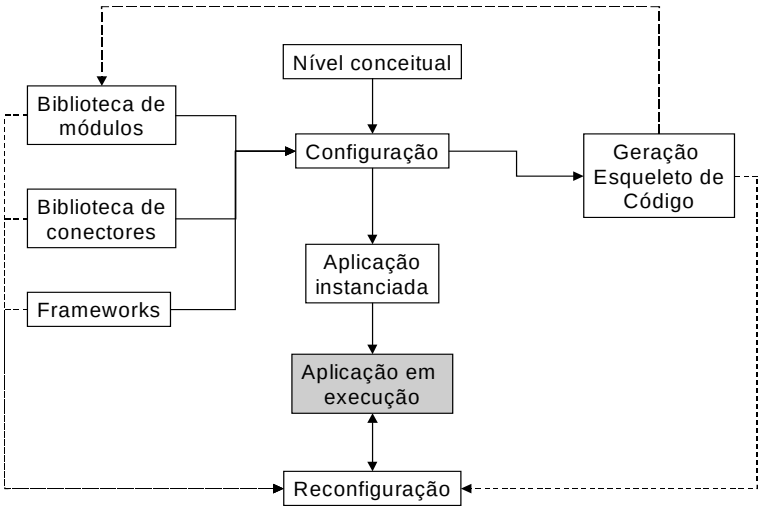


Figura 4.1 - concepção de uma aplicação por configuração

Uma vez que a implementação de todos os componentes estiver concluída, a aplicação poderá ser instanciada.

3. Na etapa de instanciação da aplicação, a descrição de sua configuração será consultada pelo suporte à configuração, para a criação das instancias dos módulos e conectores. A interligação dos mesmos é providenciada a seguir. Quando esta etapa for concluída, a aplicação poderá ser posta em execução.

4. Durante a execução de uma aplicação, é possível que novos requisitos se apresentem, ou que a aplicação deva ser adaptada para algum modo de operação específico. Neste caso, a aplicação deverá sofrer reconfigurações.

5. Uma operação de reconfiguração poderá envolver a paralisação temporária de partes da aplicação, a criação ou remoção de instâncias de componentes e alterações de aspectos não-funcionais ou de sua estrutura. Para a realização de uma reconfiguração, componentes disponíveis em uma biblioteca também podem ser utilizados ou a implementação de novos componentes poderá ser necessária.

Neste capítulo, a linguagem de configuração é apresentada. Na seção 4.2, os conceitos do modelo passam a ser materializados com a apresentação da sintaxe para a declaração de módulos, portas e conectores. Em seguida, são apresentadas as soluções inseridas na linguagem de configuração para a declaração de alguns aspectos não-funcionais e detalhes da semântica destas declarações são também discutidos. Ainda nesta seção, a apresentação dos comandos para a composição da estrutura das aplicações fecham o ciclo necessário para podermos desenvolver os exemplos da seção seguinte.

Na seção 4.3, alguns exemplos são utilizados para demonstrar a aplicabilidade do modelo de configuração proposto e para permitir uma comparação mais ampla com outras propostas, que foram avaliadas no capítulo 2. Os exemplos desenvolvidos nesta seção exploram as possibilidades da linguagem, através da composição de módulos e da utilização dos aspectos de distribuição e coordenação. Estes exemplos giram em torno do problema produtor/consumidor com buffer limitado, em várias versões, onde novos aspectos vão sendo adicionados gradativamente. Em cada versão, algumas facetas da linguagem de configuração são exploradas para atender aos requisitos introduzidos.

O capítulo é concluído com uma discussão sobre o modelo, enquadrando-o de acordo com os requisitos identificados no capítulo 1 e através de comparações com as propostas avaliadas no capítulo 2, segundo as perspectivas da linguagem de configuração apresentada.

4.2 A linguagem de Configuração

A concepção de uma aplicação distribuída é efetivamente iniciada pela descrição de sua arquitetura, através de uma linguagem de configuração. A linguagem de configuração que estamos propondo permite, como outras linguagens de configuração, a descrição, o armazenamento e a recuperação da descrição de módulos, conectores e aplicações. Em adição aos recursos tradicionais das linguagens de configuração estamos oferecendo recursos para a configuração de aspectos não-funcionais, como os descritos na seção 3.3.

A linguagem de configuração possui declarações para a especificação de classes de módulos, tipos de conectores e portas. Para produzir efetivamente a configuração de uma aplicação a linguagem possui comandos para instanciar, interligar e acionar

módulos e conectores. As declarações da linguagem de configuração possuem sintaxe semelhante à declaração de módulos e interfaces da linguagem IDL da OMG [OMG 96A], porém com algumas diferenças semânticas. Além disso, o sistema de tipos e o esquema de visibilidade da IDL também é válido.

A linguagem de configuração aumenta a expressividade da IDL, pois esta permite apenas que se descreva a interface de objetos que oferecem serviços através de seus métodos. Em contrapartida, a linguagem de configuração permite a descrição explicita de toda a aplicação, incluindo módulos clientes e servidores, bem como a forma de interligação destes elementos. A ligação entre clientes e servidores está implícita nas descrições de interface em IDL mas, até o presente momento, não podem ser configuradas. Assim sendo, podemos somar às vantagens na declaração das portas de entrada e de saída apresentadas na seção 3.2.2, o fato das interações que partem de um módulo ficarem expostas e, com isso, a reutilização de módulos pode ser feita com menor chance de ocorrência de problemas como a *fragilidade da classe base* (este problema será comentado no capítulo 5), que podem acontecer com o uso de herança em linguagens orientadas a objetos. Além disso este esquema permite que se faça verificações sobre a arquitetura sendo descrita.

A linguagem proposta é batizada de CBabel (*Building Applications by Evolution with Conectors*). Os fundamentos de Babel, uma versão anterior da linguagem de configuração, que não utiliza o conceito de conectores, são discutidos em [Mal 96]. Como a maioria das CLs, CBabel é essencialmente declarativa. Uma versão preliminar da linguagem e do formato das declarações é apresentado a seguir.

4.2.1 Referências

Na linguagem de configuração, as referências à classes de portas, módulos e conectores ou referências a instâncias destes elementos são representadas por nomes. Um nome é constituído por uma cadeia de caracteres, no mesmo estilo adotado pela IDL-CORBA. As referências tem o seu significado regido pela escopo e visibilidade de onde estão sendo declaradas (módulos ou conectores compostos, veja seção 4.2.9) e onde estão sendo referenciadas. Uma referência pode usar implicitamente o seu escopo como informação de visibilidade ou uma referência parcial e pode ainda utilizar explicitamente a informação de toda a localização, uma referência completa.

Referencias completas são representados por nomes separados por pontos em um esquema hierárquico e também depende do escopo. Na listagem 4.1 estas variações são apresentadas.

```
1    string                    // uma referência a um elemento qualquer (módulo, porta ou conector)
2    módulo.porta              // uma referência a porta de um módulo
3    estação.módulo.porta      // uma referência a uma porta de um módulo em uma estação
4    módulo at porta           // uma referência a um módulo em uma estação
5    módulo.porta at estação   // uma referência a uma porta de um módulo em uma estação
```

Listagem 4.1

Nas próximas subseções, onde estiver indicado `<nome-tipo-porta>`, `<nome-tipo-módulo>`, `<nome-tipo-conector>` ou `[instância]`, podemos substituir por um nome como *PGenerica*, *MTradutorT ou Modulo1*.

4.2.2 Portas

A declaração de um tipo de porta ou instância de porta é apresentada abaixo. A indicação do sentido da porta (*in* ou *out*) tem o objetivo de explicitar se o método associado será primariamente um servidor ou cliente e pode ser declarado apenas em uma instância de porta. *parms* é usado para se declarar os tipos e instâncias de parâmetros. O sentido destes parâmetros (entrada ou saída) está associado ao sentido declarado na instância da porta. *return* é utilizado para especificar o tipo do parâmetro de retorno. Os parâmetros podem ter os tipos disponíveis na IDL-CORBA.

```
1    [in | out] port <nome-tipo-porta> {
2       parms (lista de parâmetros);
3      [return (parâmetro de retorno);]
4    } [instâncias;]
```

Listagem 4.2

Observações:

- um tipo de porta pode ser declarado separadamente de um módulo e este tipo poderá ser instanciado em vários módulos diferentes.
- um tipo de porta pode ser declarado sem o sentido. Na declaração de instâncias desta porta, na interface de um módulo, os modificadores ***in*** e ***out*** vão configurar o sentido da mesma e indicar o sentido dos parâmetros;
- uma lista de parâmetros poderá conter uma série de tipos de parâmetros separados por vírgulas. Os tipos de parâmetros seguem os tipos primitivos de IDL-CORBA.

Por exemplo,

```
parms (limite int, cont int, rótulo string);
```

contém uma lista com 3 parâmetros. Cada parâmetro possui um tipo e pode possuir um nome. Os parâmetros *limite* e *cont* são do tipo **int** e *rótulo* é do tipo **string**. Se um nome for declarado para o parâmetro, este poderá ser referenciado explicitamente nos comandos de configuração. Caso contrário o parâmetro será utilizado apenas na verificação da compatibilidade de assinaturas.

4.2.3 Módulos

A declaração de um tipo ou instância de módulo é feita através de uma construção **module**. Dentro da declaração de um módulo, as portas de entrada e saída associadas aos métodos deste módulo devem ser instanciadas e possuir um nome. Se for necessário é permitido a sobreposição de atributos declarados nos tipos de portas na declaração de instâncias destas portas no módulo, como por exemplo a alteração do tipo do parâmetro de retorno. Entretanto, a sobreposição de atributos será mais utilizada na declaração de guardas, como veremos adiante. A declaração **map** que indica o mapeamento entre a abstração do módulo e sua implementação será também comentada adiante.

```
1   module <nome-tipo-módulo> {
2      port <nome-tipo-porta> [{sobreposição de atributos}] <nome-porta>;
3      map <CODE | SOURCE> <LANGUAGE | CLASS | SO> <arquivo>;
4   } [instâncias;]
```

Listagem 4.3

Uma aplicação, descrita pela composição de módulos, também é considerada um módulo. Assim sendo, para se descrever uma aplicação, basta que se declare um módulo, composto por módulos mais elementares, configurados adequadamente, que descrevam a arquitetura desta aplicação.

4.2.4 Conectores

Tipos ou instâncias de conectores são declaradas através da construção **connector**. No corpo de um conector devemos declarar instâncias das portas de entrada e saída utilizadas. Geralmente as portas são declaradas aos pares nos conectores. Uma porta de entrada de um conector encaminha o fluxo recebido para uma porta de entrada. Exemplos deste tipo de declaração são apresentados na seção 4.3. Não é permitida a

sobreposição ou extensão da assinatura das portas dos conectores como acontece nos módulos, embora outros tipos de extensão sejam permitidos. Além disso, o nome da instâncias de portas não é obrigatório.

```
1    connector <nome-tipo-conector> {
2       in | out port <nome-tipo-porta> [nome-porta];
3       map <UNIX-PIPE | SOCKET | RPC | CODE | SOURCE> [<arquivo>];
4    }[instâncias;]
```

Listagem 4.4

Deve ser observado que na declaração de módulos e conectores, o escopo dos identificadores é interno, não possuindo sentido fora do elemento onde o identificador foi usado. O esquema de visibilidade adotado por IDL-CORBA vale para a linguagem de configuração.

map: Mapeamento da Implementação

Os módulos e conectores declarados podem possuir uma indicação de seu mapeamento no nível da implementação , características de um aspecto não-funcional ou recurso necessário que deve estar disponível no ambiente de execução. Por exemplo, podemos declarar que determinado módulo é implementado por um programa cujo código está em um arquivo externo, da seguinte forma:

```
map CODE C++ "arquivo" ;
```

A declaração ***map*** *CODE C++ "arquivo";* faz a associação entre a linguagem de configuração e o nível de implementação. Neste caso, está sendo declarado que o código do módulo está em linguagem C++, é externo a esta declaração e está armazenado em *arquivo*. Por outro lado, esta mesma declaração pode ser utilizada por uma ferramenta geradora de código, se este código não existir.

A forma geral desta cláusula é:

```
map <tipo> <linguagem> <referência>
```

onde:

tipo: dependendo do tipo de elemento ele pode indicar se a referência é:

- no caso de um módulo: um código fonte, código executável, uma classe (em Java, ou C++, por exemplo) ou um elemento que pode ser ligado dinamicamente (objeto compartilhado - *shared object* encontrado em

sistemas Unix, ou biblioteca compartilhada como as DLLs disponíveis no ambiente Windows).

- no caso de um conector: um mecanismo de implementação como *pipe*, *streams*, *sockets*, RPC, um estilo de conector, ou alguma alternativa correspondente ao caso do módulo.

linguagem : indica a linguagem que está sendo utilizada pela referência. Este argumento só é válido nos casos de código fonte.

referência: nome do arquivo ou URL [URL A] com a referência do elemento que está sendo utilizado. Com a utilização de uma URL esta referência poderá estar localizada de forma distribuída.

O interpretador da linguagem deve utilizar as informações das declarações ***map*** para gerar código automaticamente ou verificar a consistência da configuração. Por outro lado, cabe ao suporte à configuração identificar a forma adequada de combinar estas informações para instanciar o módulo ou conector. As informações contidas na declaração ***map*** podem ser utilizadas pelo suporte à configuração na tomada de decisões de alocação de recursos, verificação de parâmetros de qualidade de serviço e na própria distribuição dos módulos por estações diferentes. Em sistemas sensíveis à arquitetura ou QoS (*architecture and QoS aware*) um suporte deste tipo é importante [Kon 98 e Iss 98]. Através das informações contidas nas cláusulas ***map***, o configurador fica à par dos requisitos dos módulos e poderá interagir com o sistema de execução para atender a estes requisitos. Adicionalmente, o configurador pode ser programado para alterar dinamicamente os módulos e conectores de uma aplicação em reação à mudanças do sistema de execução.

O esquema mapeamento de módulos em suas respectivas implementações cria também facilidades para que a aplicação possa ser programada por diversas linguagens de programação. É possível que cada módulo seja implementado em uma linguagem de programação diferente. Pode-se selecionar uma linguagem de programação que possua determinadas características que facilitarão a programação de certo módulo. A *cola* que permitirá a interação entre módulos programados em linguagens diferentes é o conector. O conector pode ser concebido para traduzir as incompatibilidades de código entre módulos. Além de intermediar a interação entre os módulos, o conector poderá

corrigir diferenças de representação ou estilos de chamada de procedimento, por exemplo. Se o código dos módulos foi gerado automaticamente a interação entre eles poderá ocorrer sem a intervenção de conectores preparados especificamente para esta tarefa, desde que o código gerado para as linguagens diferentes, resultem em módulos de execução compatíveis. Uma possibilidade é o mapeamento dos módulos no nível de configuração em objetos CORBA. O código gerado a partir da descrição dos módulos da uma arquitetura pode ser IDL-CORBA traduzida para uma das linguagens que possuam mapeamento para IDL. Desta forma módulos programados em linguagens diferentes podem ser comunicar através de conectores com estrutura suportada por um ORB.

A cláusula *map* poderá ser utilizada, nas futuras versões da linguagem de configuração, para a inclusão de novos tipos de conectores ou novas características à própria linguagem. Por exemplo, anotações para aspectos que precisam ser contemplados na linguagem podem ser adicionadas. Estas possibilidade serão investigadas futuramente, ficando também em aberto a concepção de um mecanismo para facilitar a introdução sistemática destes acréscimos.

4.2.5 Comandos de Configuração

Após a declaração de módulos e conectores, estes elementos precisam ser interligados e instanciados. A linguagem de configuração oferece um conjunto de primitivas de configuração para ser utilizado na descrição da composição dos elementos em uma aplicação. O ambiente de suporte à configuração utiliza estas primitivas para materializar a arquitetura através de seus respectivos comandos, quando uma aplicação é iniciada ou durante manobras de reconfiguração. Na tabela seguinte, são apresentados comandos básicos de configuração necessários para descrever-se uma aplicação.

Comando	Descrição

instantiate	Cria uma instância de um módulo ou conector. Este comando ativa as rotinas de iniciação do módulo (por exemplo, no caso de utilização de C++ a rotina de iniciação do objeto é executado), aloca os espaços de memória necessários e registra a existência do módulo no ambiente de execução.
	
	Sintaxe:
	
	`instantiate <module-type> as <module-name> [at <station>]`
	`instantiate <station> at <ip address>`
	`instantiate <connector-type> as <connector-name>`

link	O comando ***link*** permite efetuar as interconexões dos componentes de uma aplicação para formar sua estrutura. No nível da configuração este comando liga portas de pares de módulos através de um conector. No nível da execução a ligação de dois módulos pode desencadear algumas ações como a resolução de referências a endereços, criação de recursos como *sockets* ou formação de canais de comunicação. As portas envolvidas devem ser complementares. Se uma porta ***out*** possui um ***parm*** com assinatura ***int*** o módulo parceiro também deve ter uma porta ***in*** com ***parm*** int para que possa haver a ligação, como no exemplo abaixo:
	
	`...`
	`// declarado no módulo cliente`
	`out port {`
	`  parms (int request_output);`
	`  return (string answer);`
	`} pedido;`
	`...`
	`// declarado no módulo servidor`
	`in port {`
	`  parms (int request_input);`
	`  return (string answer);`
	`} resposta;`
	
	O conector utilizado para a ligação deve ter portas complementares para todos os módulos aos quais ele conecta. No caso dos módulos acima, o conector teria pelo menos duas portas.
	
	Em um caso mais simples pode-se declarar a conexão das portas de dois módulos sem a referência de conectores intermediários, embora ele sempre exista conceitualmente. Uma outra possibilidade é a ligação entre dois módulos sem a referência das portas. Neste caso, todas as portas de tipos complementares são conectadas automaticamente pelos conectores utilizados.
	
	Sintaxe:
	
	`link <module-name> to <module-name>`
	`link <module-name> to <module-name> by <connector-name>`
	`link <module>.<port> to <module>.<port> by <connector-name>`

export	Liga uma porta de um módulo composto (seção 4.2.9) a uma porta de um módulo componente. Este comando é utilizado para indicar-se a visibilidade das portas de um módulo ou conector composto por outros módulos da aplicação. A ligação deve ser efetuada sempre do módulo ou conector componente para o módulo composto. Exemplos de sua utilização serão apresentados na seção 4.2.9 e 4.2.10. Sintaxe: **export** <port>

Tabela 4.1 - Comandos Básicos de Configuração

4.2.6 Sintaxe para Aspectos Não-Funcionais

A declaração de aspectos não-funcionais na linguagem de configuração tem a forma de extensões na sintaxe da declaração de módulos, portas e conectores. Nesta etapa inicial da proposta da linguagem de configuração serão apresentadas apenas as sintaxes para aspectos de sincronização e distribuição, mas a inclusão de aspectos de comunicação ou tolerância a falhas seria facilmente suportada [Szt 94, Bri 95] na linguagem. Algumas anotações da cláusula *map* podem ser indicações de aspectos não-funcionais relacionados com os mecanismos utilizados na comunicação, mas no momento, não vamos considerá-las para as configurações de aspectos.

4.2.7 Sintaxe para os Aspectos de Distribuição

Durante a execução de uma aplicação módulos componentes podem ser instanciados em estações distribuídas. A sintaxe para a configuração do aspecto de distribuição é bastante simplificada. Na realidade foi apresentada juntamente com o comando de instanciação de módulos.

Em princípio, o ambiente de execução poderá ser preparado para operar segundo as seguintes alternativas em relação a distribuição dos módulos se nada for informado:

- os módulos são instanciados na mesma estação, possivelmente na estação onde a configuração está sendo descrita, ou
- os módulos serão distribuídos por estações disponíveis , segundo algum critério (estações com menor carga de processamento, por exemplo),

Se a aplicação precisar, de uma forma explícita, indicar onde os módulos devem ser instanciados, basta que ser declare os comandos de instanciação com os parâmetros ***at*** *<estação>* indicando-se a estação onde a referida instância deverá ser criada.

Observa-se que a localização dos módulos é transparente para o mesmo, que não precisa ser concebido antecipando-se este aspecto. O controle das referências dos módulos instanciados distribuidamente também não requer intervenção da aplicação. Isto fica por conta do suporte à configuração.

4.2.8 Sintaxe para os Aspectos de Coordenação

Para oferecer acesso à configuração de aspectos de coordenação segundo o modelo proposto na seção 3.3.5, a linguagem possui extensões que permitem indicar o esquema de concorrência e sincronização de métodos e a associação de guardas à portas de um módulo.

exclusive - indica que o grupo de métodos deve ser executado em exclusão mútua

```
1    exclusive {
2       <portas-de-uma-classe> |
3       <portas-de-uma-instância>
4    }
```

Listagem 4.5

selfexclusive - indica se um método não pode ser executado concorrentemente. Se uma porta não for marcada como selfexclusive, várias instâncias deste método podem executar concorrentemente para atender a requisições diferentes.

```
1    selfexclusive {
2       <portas-de-uma-classe> |
3       <portas-de-uma-instância>
4    }
```

Listagem 4.6

concurrent - indica que o grupo de métodos pode ser executado concorrentemente. Este é o esquema *default*: se nada for indicado, todos os métodos de um módulo são executados concorrentemente. Este modelo segue, em outro nível, o modelo de objeto adotado em Java, por exemplo, em que os métodos são concorrentes se o modificador ***synchronized*** não for utilizado.

```
1    concurrent {
2       <portas-de-uma-classe> |
3       <portas-de-uma-instância>
4    }
```

Listagem 4.7

guards - guardas são definidos nas portas de saída de um conector, pois estas serão interligadas à portas de entrada de um módulo, onde o guarda precisa efetivamente atuar. Para a configuração de guardas foram acrescentadas extensões na declaração das portas dos conectores. Se um módulo for utilizar guardas para um ou mais de seus métodos, o conector que interliga estes métodos deve declarar uma lista de guardas a serem usados.

A sintaxe geral para se definir guardas em um conector é a seguinte:

- no conector onde serão programados os guardas é necessário definir-se uma lista de **variáveis de condição** que serão usadas para avaliar se um guarda está aberto ou fechado. É necessário, também, definir-se que variáveis, declaradas no módulo onde os guardas devem atuar, serão inspecionadas. Isso permitirá, na etapa de implementação, mapear código automaticamente para o módulo, definindo um método de inspeção com os dados necessários, por exemplo. Deve ser de conhecimento do programador quais são as variáveis a serem usadas. Isto é feito através das seguintes declarações:

```
1   connector {
2      condition <condição1> [, <condicao2>, <condiçãoN>] ;
3      saterequired <tipo variável1>[, <tipo variável2>, <tipo variável N>];
4   ...
5   }
```

Listagem 4.8

- cada porta de saída do conector pode definir um guarda de entrada e saída. Um guarda de entrada tem efeito antes da requisição ser tratada pelo respetivo método no módulo sob proteção e o guarda de saída tem efeito após a execução do método. A definição de um guarda possui 3 partes:

1. uma expressão booleana envolvendo as variáveis de condição declaradas, cujo resultado indica se o guarda está aberto (resultado verdadeiro) ou fechado (resultado falso);

2. uma expressão composta que pode conter as variáveis de condição e variáveis declaradas na expressão *staterequired*, que será avaliada antes da execução do método (cláusula *before*);

3. uma expressão composta que pode conter as variáveis de condição e variáveis declaradas na expressão *staterequired*, que será avaliada após a execução do método (cláusula *after*);

```
1    out port {   // guardas são declarados em portas de saída de conectores
2
3        // declara um guarda para a  entrada de um método
4        inguard (<expressão-variáveis-de-condição>) {
5           before {<expressão-composta>};
6           after {<expressão-composta>};
7        }
8
9        // declara um guarda para a saída de um método
10       outguard (<expressão-variáveis-de-condição>) {
11          before {<expressão-composta>};
12          after {<expressão-composta>};
13       }
14   }
```

Listagem 4.9

- um guarda de entrada ou de saída também pode ser configurado adicionalmente com um prazo máximo de espera para que uma condição seja satisfeita. Uma porta alternativa também pode ser configurada para que a requisição que estava bloqueada no guarda seja enviada no caso (1) do tempo expirar ou no caso (2) do guarda estar fechado e nenhuma cláusula de temporização ter sido declarada. Este esquema permite tratar situações em que determinada condição não possa ser esperada indefinidamente e facilitar um tratamento alternativo para as requisições ou exceções.

```
1    inguard (<expressão-variáveis-de-condição>) {
2       timeout (<prazo de espera>);
3       alternative (<porta declarada no mesmo conector>);
4       before {<expressão-composta>};
5       after {<expressão-composta>};
6    }
```

Listagem 4.10

Por exemplo, se um guarda monitora a ocupação de um recurso, uma nova requisição que chega para ser tratada, poderá ser encaminhada para um outro módulo como alternativa de tratamento.

O próximo exemplo ilustra o uso de guardas. Seja o modulo abaixo em que um de seus métodos deva ser guardado (conforme comentário no código) …

```
1    module Buffer {
2       ...
3       in port Put; // Put deve ser guardado
4       in port Get;
5    } BB;
```

Listagem 4.11

… e um conector que interliga este modulo a outros módulos

```
1    connector Request {
2       ...
3       in port  entrada1;
4       in port  entrada2;
5       out port saida1;  // que interliga Put
6       out port saida2;  // que interliga Get
7       ...
8    } RQ;
```

Listagem 4.12

A figura 4.2 esquematiza esta configuração.

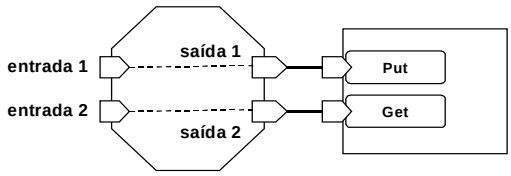


Figura 4.2 - um conector com duas portas de entrada e saída

Supondo que o método *Put* do módulo *BB* necessite de um guarda, isso será feito no conector da que interliga a porta correspondente a este método, da seguinte forma:

```
1    connector GuardedRequest {
2    ...
3       condition OkToPut = true;     // declara as variáveis de condição e  inicializa
4       saterequired (int UsedSlots);
5       out port {
6
7          inguard OkToPut {          // guarda bloqueante de entrada
8             before {expressão};     //
9             after  {expressão};     //
10         }
11      } saida1;                     // porta com guarda
12
13      out port saida2;             // porta não guardada
     } GRQ;
```

Listagem 4.13

A figura 4.3 esquematiza a nova situação.

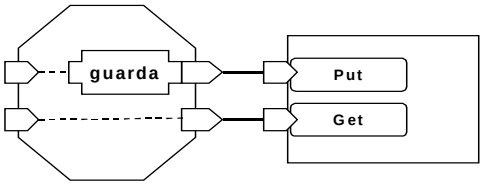


Figura 4.3 - um conector com uma porta de saída com guarda

Vale lembrar que tanto a declaração ***inguard*** como ***outguard*** são seguidas de uma expressão booleana contendo somente variáveis de condição. As declarações ***before*** e ***after*** são seguidas de uma ou mais cláusulas que podem conter atribuições, expressões condicionais e expressões que resultem em valores booleanos, envolvendo variáveis de condição, variáveis declaradas no conector ou variáveis declaradas na cláusula ***staterequired***. Observe o exemplo abaixo:

```
1    connector {
2       int MAX = 10;
3       condition Vazio = true, Ativo = false;
4       staterequired int UsedSlots;
5       port {
6          OUT
7          inguard (Vazio AND Ativo) {
8             after {
9                if (UsedSlots = MAX) Vazio = false;
10            }
11         }
12      } Put;
13   ...
14   }
```

Listagem 4.14

O guarda da porta *Put* estará aberto se as condições **Vazio** e **Ativo** forem verdadeiras e, neste caso, após a execução do método no módulo interligado por este conector, a expressão declarada na cláusula *after* é avaliada. Esta expressão utiliza a variável de condição *Vazio* e a variável inteira *UsedSlots* declarada na cláusula *staterequired* que é inspecionada no módulo.

4.2.9 Composição de Módulos

A descrição de aplicações e módulos compostos na linguagem de configuração é feita de forma simples e algumas variações de estilo são permitidas. O exemplo da figura 3.4, no capítulo 3, pode ser descrito, pelo menos, de duas formas, como apresentado a seguir nas listagens 4.15 e 4.16. Na primeira forma, instâncias dos módulos básicos e conectores são declaradas diretamente dentro do módulo composto.

```
1    module MC1 {
2       …
3       module {…} Mp1;              // cria instância Mp1
4       module {…} Mp2;              // idem para Mp2 e Mp3
5       module {…} Mp3;
6       instantiate Mp1;             // instancia Mp1, Mp2 e Mp3
7       instantiate Mp2;
8       instantiate Mp3;
9       link Mp1 to Mp2 by C1;       // interliga os módulos internos
10      link Mp1 to Mp3 by C2;
11      export Mp1.porta1;             // exporta as portas com visibilidade externa
12      export Mp3.porta3;
13   }
```

Listagem 4.15

A visibilidade de Mp1, Mp2, Mp3 é interna a MC1, pois estes módulos foram declarados dentro do corpo deste. A classe dos módulos Mp1, Mp2 e Mp3 também não estão disponíveis, pois simplesmente foram criadas instâncias diretamente. Um outro estilo de programação possível é a declaração das classes de módulos e conectores fora do escopo de qualquer outro módulo, possibilitando que instâncias destas classes sejam criadas em qualquer ponto. Logicamente, a visibilidade da instância depende de onde a mesma foi criada.

```
1    module T1 {}                 // declara-se classes de módulos com visibilidade global
2    module T2 {}
3    module T3 {}
4
5    module MC1 {
6       instantiate T1 as Mp1;       // instancia Mp1, Mp2 e Mp3
7       instantiate T2 as Mp2;
8       instantiate T3 as Mp3;
9       link Mp1 to Mp2 by C1;       // interliga os módulos internos
10      link Mp1 to Mp3 by C2;
11      export Mp1.porta1;           // exporta as portas com visibilidade externa
12      export Mp3.porta3;
13   }
```

Listagem 4.16

4.2.10 Composição de Conectores

A descrição de um conector composto através da linguagem de configuração é bastante semelhante a de um módulo composto. O exemplo da seção 3.3.7, capítulo 3 será utilizado para ilustrar como a composição de conectores é descrita na linguagem. Observa-se que para interligar conectores entre si foi introduzido o comando ***bind***. Em princípio a introdução de mais um comando seria desnecessária. Bastaria o comando link para indicar a ligação de portas de entrada e saída de conectores. Entretanto, fazendo-se esta diferenciação simplifica-se a implementação de um interpretador da

linguagem e permite-se que verificações da estrutura sendo descrita sejam mais eficientes em distinguir erros de configuração como a interligação de dois módulos sem o uso de um conector.

```
1   connector Ccomposto {
2      connector { } α;              // declara-se as instâncias dos conecotres componentes
3      connector { } β;
4      connector { } χ;
5      connector { } δ;
6      connector { } ε;
7      connector { } ϕ;
8      connector { } γ;
9
10     BIND α.portaS1 to β.portaE1; // bind permite ligação entre portas de conectores
11     BIND α.portaS2 to χ.portaE1; // interliga-se as portas dos conectores
12     BIND α.portaS3 to δ.portaE1;
13     BIND α.portaS4 to ε.portaE1;
14     BIND χ.portaS1 to ϕ.portaE1;
15     BIND ε.portaS1 to γ.portaE1;
16
17     export α.portaE1;             // indica a visibilidade das portas do conector composto
18     export α.portaE2;
19     export α.portaE3;
20     export α.portaE4;
21     export β.portaS1;
22     export ϕ.portaS1;
23     export δ.portaS1;
24     export γ.portaS1;
25
26  } CC;
27
28  instantiate CC; // ao se instanciar um conector composto, todas as instâncias
29                  // de conectores componentes são automaticamente criadas.
```

Listagem 4.17

Na listagem 4.17 uma classe de conector composto, *Ccomposto*, e uma instância desta classe, *CC*, são criadas. Da linha 2 até a linha 9 as instâncias dos conectores componentes são declaradas. Em seguida, são interligadas as portas dos conectores através de comandos bind. Os nomes das portas utilizadas são ilustrativos, já que a declaração dos conectores não apresentaram, nesta listagem, este nível de detalhe. O próximo passo é indicar quais são as portas dos conectores componentes que terão visibilidade externa. Este conector composto apresenta quatro portas de entrada e quatro de saída. Por último, uma instância para este conector é declarada. Neste ponto, o conector poderá ser utilizado pela aplicação. Assim, através do comando instantiate, uma instância do conector CC é criada. Dentro do conector composto não foram criadas as instâncias dos conectores declarados. Quando o comando para instanciar o conector composto for executado, o suporte à configuração se encarregará de criar as

instâncias dos conectores componentes, na ordem em que foram declarados.

4.2.11 Filtros

Deve-se considerar o uso de filtro como uma alternativa ao uso de conectores que atenderiam especificamente aos aspectos necessários ou a composição de conectores, como abordado na seção anterior. Esta opção pode se dar quando o esforço para realização destes conectores não seja compensatória ou no caso em que as características desejadas não representem propriamente um aspecto, o que levaria a baixa reusabilidade.

Na linguagem de configuração os filtros são declarados no corpo das portas através de uma cláusula *filter*. A sintaxe é apresentada no trecho de código a seguir.

```
1    connector <nome-tipo-conector> {
2    ...
3       out port <nome-tipo-porta> {
4          parms (int par1, int par2);
5          return (int ret);
6          filter {                                  // declara o filtro na porta
7             if (par1 > 10) par2 = par2 + 20;       // expressões envolvendo parâmetros
8             par2 = par2 / 2;                       // declarados nas portas
9          }
10      } FPort;
11   ...
12   }[instâncias];
```

Listagem 4.18

4.2.12 Herança

A algumas das construções permitidas na linguagem de configuração podem ser classificadas como herança. Tanto na descrição de módulos quanto na descrição de conectores, pode-se utilizar uma classe e estender a sua assinatura ou pode-se utilizar a definição da interface de uma classe de conector e associar a uma implementação diferente da original. Este esquema de herança se resume à herança de interface, como tratado no capítulo 1. Por exemplo, conectores mais simples podem ser estendidos para incorporar aspectos de coordenação, utilizando-se para isso a sintaxe oferecida pela linguagem de configuração ou através de filtros. Não existe na linguagem de configuração, entretanto, uma equivalência completa com a semântica utilizada na herança de linguagens como C++ ou Java, justificado pela discussão apresentada na seção 1.3.1 que trata da tecnologia de objetos .

No que diz respeito à herança como mecanismo de extensão existe uma outra interpretação para a linguagem de configuração. Pode-se estender a funcionalidade de módulos e conectores através de composição, como mencionado na subseção anterior, criando-se um conector mais especializado ou que agregue maior funcionalidade. A composição ou agregação são considerados mecanismos legítimos para implementar a herança [Tai 96]. Uma das vantagens de uso de composição em relação aos esquemas tradicionais de herança é que não se recai com facilidade nos problemas de anomalia de herança e fragilidade de classe base (seção 1.3). Usando-se a composição como mecanismo para implementar herança nos conectores, por exemplo, procura-se preservar o encapsulamento e a ortogonalidade dos aspectos implementados em cada conector componente, e maximizar a separação de interesses. Exemplos práticos sobre composição e extensão de interfaces são apresentados na próxima seção.

Observação

Neste texto apresentamos uma versão inicial da linguagem de configuração. Nem todos os aspectos levantados, como de comunicação ou tolerância a falhas, foram contemplados, neste momento, na linguagem. Optamos por avaliar a linguagem de configuração a partir de um subconjunto de suas possibilidades ao invés de apresentar exemplos muito complexos e com aspectos muito diferentes. Detalhes sobre a semântica da linguagem serão apresentados na continuação deste trabalho.

Com a apresentação desta proposta inicial da linguagem de configuração podemos testar o modelo em exemplos concretos para demonstrar sua aplicabilidade e comparar a solução com as propostas relacionadas no capítulo 2, de forma complementar ao que foi feito no final do capítulo 3.

4.3 Um Exemplo

Nesta seção o uso dos conceitos de nosso modelo é exemplificado através de uma aplicação que evolui de modo a atender a uma série de requisitos. O exemplo é baseado na aplicação produtor/consumidor que foi utilizada no capítulo 2. Na versão inicial, os requisitos da aplicação são simplificados. Nas versões posteriores, serão impostos novos requisitos e, em resposta, alguns aspectos serão configurados na aplicação para

atendê-los. Por exemplo, durante o desenvolvimento da arquitetura da aplicação, os aspectos de sincronização e distribuição são configurados gradualmente, na medida em que os novos requisitos são impostos a uma configuração inicial básica. Nestas novas versões apenas as diferenças entre a configuração anterior e a atual serão discutidas.

4.3.1 Primeira Versão

Nesta primeira versão (figura 4.4), um sistema produtor produz iterativamente um item e envia uma requisição para o buffer, solicitando-o a armazenar este item. Um sistema consumidor iterativamente envia uma requisição para o buffer, solicitando a retirada de um item armazenado. O buffer é implementado para receber requisições de um produtor para armazenar um item e requisições de um consumidor para retirar um item. Nenhum requisito adicional é imposto.

A solução para a estrutura desta aplicação será composta de 3 módulos: um produtor, um consumidor e um buffer. Por decisão do projetista, a comunicação entre os módulos é síncrona: conectores com estilo pedido/resposta são utilizados. A configuração é descrita a seguir com comentários destacando as principais características utilizadas.

module BufferApplicationClass {
port PutT {
parms (String Item);
return (int Ack);
}
port GetT {
return (String Item);
}
	Para criar essa aplicação, define-se inicialmente os tipos de portas, os tipos de conectores e as classes de módulos que serão utilizados.
	A assinatura das portas é declarada aqui.

module BufferT { **in port** PutT ArmazenaItem; **in port** GetT RetiraItem; **map** CLASS Java “buffer” } Buffer;	Os módulos são declarados indicando-se instâncias das portas deste módulo e uma referência de como este módulo é implementado (neste caso o módulo *Buffer* é implementado pela classe *buffer.class* programada em Java).

module ProducerT {
out port PutT SaidaItem;
map CLASS Java "producer";
} Producer;
 	
module ConsumerT {
out port GetT EntradaItem;
map CLASS Java "consumer";
} Consumer;
	De forma semelhante os outros módulos são configurados.
	Observe que os modificadores **in** e **out** são utilizados para indicar o sentido assumido pelo instância da porta

connector { **in port** GetT; **out port** GetT; } ConsumeItem; **connector** {	Em seguida os conectores que interligarão os módulos são definidos. Verifica-se também que o nome das instâncias já foi atribuído para módulos e conectores,

`in port PutT;`
`out port PutT;`
`} ProduceItem;`
	mas poderia ser deixado para o momento da instanciação.

`instantiate Buffer`
`instantiate Producer;`
`instantiate Consumer;`
 	
`instantiate ProduceItem;`
`instantiate ConsumeItem;`
	Neste ponto os módulos e conectores são instanciados.

`link Producer.SaidaItem to`
`Buffer.ArmazenaItem by ProduceItem;`
`link Consumer.EntradaItem to`
`Buffer.RetiraItem by ConsumeItem;`
`}`
	Concluída a configuração os módulos são interligados através dos conectores. Observa-se que a referência a porta sendo interligada foi utilizada, mas não seria imprescindível neste caso. Neste ponto a aplicação está configurada, como apresentado na figura 4.4.

Quando a criação da instância de aplicação é solicitada, são criadas todas as instâncias dos módulos componentes e realizadas as conexões especificadas. Uma aplicação é tratada de forma similar a outro módulo qualquer. Portanto, cria-se uma aplicação como instância de BufferAplicationClass da seguinte forma:

```
module  BufferAplicationClass  BA;
instantiate  BA;
```

Neste ponto a instância completa da aplicação BA foi criada e sua execução iniciada.

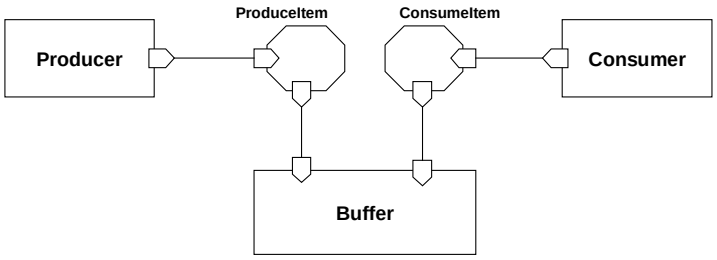


Figura 4.4 - Aplicação Produtor/Consumidor

4.3.2 Acrescentando Aspectos Não-Funcionais

Observa-se que na primeira versão da especificação nada foi dito a respeito da concorrência ou sincronização entre os módulos. A única observação sobre este aspecto é que, por definição do modelo, todos os módulos são executados concorrentemente, e o mesmo ocorrendo nos métodos de cada módulo. A coordenação entre os módulos se dá pela natureza síncrona das interações entre eles. Assim, sendo nenhum controle no acesso ao buffer foi imposto. Desta forma, uma vez que não se pode prever a ordem ou

o intercalamento (como especificado na seção 4.2, os métodos de um módulo são concorrentes por *default*) da execução dos módulos e métodos, podem acontecer inconsistências tais como dois produtores escrevendo na mesma área do buffer ou um produtor e um consumidor tentando acessar a mesma área do buffer.

Para a segunda versão da aplicação acrescentam-se, então, os seguintes requisitos:

- a aplicação deve permitir o acesso ao *buffer* por vários produtores e vários consumidores de forma concorrente;
- o acesso ao *buffer* deve ser feito em exclusão mútua: somente um produtor ou um consumidor podem ter acesso ao *buffer* e este acesso deve ser iniciado e terminado sem a interferência de outros acessos (ou seja, atomicamente).

Certamente, o módulo *Buffer* poderia ter sido programado para satisfazer este requisito, mas para isso o programador teria que ter antecipado esta necessidade, já que nada havia sido dito. Por outro ângulo, supondo que haja um repositório onde um módulo do tipo BufferClass genérico (sem qualquer outro aspecto programado além de sua funcionalidade básica), esteja disponível, no modelo proposto não existe a necessidade de se alterar este módulo ou de se rescrever um novo módulo englobando as novas características. Além disso, no primeiro caso, o programador deve estar treinado em programação concorrente.

Estes novos aspectos são contemplados na aplicação substituindo-se os conectores da primeira versão por um conector com as portas de saída marcadas como *exclusive.* O *Buffer* não será tocado porque é possível encapsular o aspecto de coordenação no conector. O requisito para a inclusão de vários produtores e vários consumidores concorrentes é facilmente configurado na aplicação, bastando criarem-se novas instâncias de cada módulo e interligá-las através do conector já instanciado. O código a seguir mostra os acréscimos necessários na configuração original da aplicação e a figura 4.5 esquematiza o resultado.

```
module SincBufferApplClass  {

  connector {
    in port GetT;
    in port PutT;
    exclusive {
      out port GetT;
```

Observa-se que por definição todos os módulos instanciados executam concorrentemente.

Assim, a única alteração é a substituição dos conectores e o acréscimo do atributo **exclusive** às portas de saída do conector, que

`out port PutT;`
`}`
`} SincPCcon;`
	liga produtores e consumidores ao módulo *Buffer*, para sincronizar os métodos do *buffer*. Observe-se que as portas deste conector são instancias de tipos de portas já definidas.

`instantiate ProducerT as Prod1;`
`instantiate ConsumerT as Cons1;`
`...`
`instantiate ProducerT as ProdN;`
`instantiate ConsumerT as ConsN;`
	A diferença na criação de instâncias, com relação a configuração anterior, é a criação de várias instâncias de produtores e consumidores. Todas as instâncias serão interligadas pela mesma instância de conector.

`link Prod1 to Buffer by SincPCcon;`
`link Cons1 to Buffer by SincPCcon;`
`...`
`link ProdN to Buffer by SincPCcon;`
`link ConsN to Buffer by SincPCcon;`
`...`
`}`
	A última modificação na configuração da aplicação é a ligação das várias instâncias de produtores e consumidores ao buffer. Observa-se que nestas ligações não foram feitas referências às portas sendo ligadas. Portanto, portas compatíveis foram ligadas automaticamente.

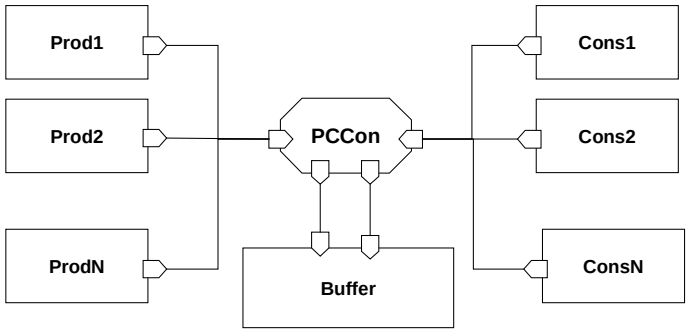


Figura 4.5 - Buffer com métodos sincronizados

4.3.2.1 Limitando e corrigindo o Buffer

Na versão anterior é detectado que não existe controle no acesso ao buffer. Por exemplo, um consumidor pode tentar consumir um item que ainda não foi produzido ou um produtor poderá tentar armazenar um produto produzido sem que haja espaço disponível (superpondo este novo item no lugar de um ainda não consumido).

Assim sendo, novos requisitos são impostos para a aplicação. Uma requisição de inclusão de item só deve chegar ao *buffer* se houver lugar disponível para armazená-lo. O mesmo vale para as mensagens do consumidor em relação a itens existentes: uma requisição pedindo a retirada de um item só poderá chegar ao *buffer* se houver algum item já armazenado. Como requisito adicional, independentemente da organização e do tamanho do *buffer* implementado no módulo Buffer, deseja-se limitar a capacidade do *buffer* da aplicação em 10 itens.

Este novo aspecto pode ser configurado na aplicação através do uso de guardas para habilitar o tratamento das requisições apenas quando as condições necessárias forem satisfeitas. Na solução apresentada, um único conector com 4 portas (2 de entrada e 2 de saída) interliga produtores e consumidores ao módulo *buffer*. Admite-se que a variável *n_itens* tenha sido declarada internamente, na implementação do módulo, ou seja, *n_itens* representa um estado interno do módulo. Neste caso, não é necessário o acesso ao código fonte do módulo, mas estamos admitindo que existe uma variável que armazena a informação do estado a ser consultado e um método de acesso a esta informação que será utilizado pelo conector.

module BoundedSincBufferApplClass {
...
connector {
condition vazio = true, cheio = false;
staterequired int n_intes;
int MAX_ITENS = 10;
exclusive {
	Primeiramente declara-se a lista de variáveis de condição a serem usadas pelos guardas e uma lista de variáveis de estado a serem consultadas no módulo *Buffer*. Em seguida, é definida uma variável, local ao conector, para limitar o tamanho do *buffer*.

out port PutT {
inguard (vazio) {
after {
cheio = true;
if (n_itens == MAX_ITENS) {
vazio = false;
} } }
} GPut;
	A requisição não chegará ao método *ArmazenaItem* do módulo *Buffer* se o guarda **vazio** for falso.
	Ao completar o método e antes de enviar a confirmação, o guarda **vazio** recebe o valor false se *n_itens* for MAX_ITENS.

out port GetT {
inguard (cheio) {
after {
vazio = true;
if (n_itens == 0) {
cheio = false;
} } }
} GGet;
}
	De forma análoga, o método RetiraItem do Buffer não será executado enquanto o guarda **cheio** não for verdadeiro.
	Ao completar o método e antes de retornar o item retirado do *buffer* o guarda **cheio** recebe falso se o espaço do *buffer* esta completamente vazio.

in port GetT; **in port** PutT; } GSincPCcon; ... }	Nenhuma outra modificação é necessária para acomodar estes aspectos de coordenação além de ligar Produtores e Consumidores ao *Buffer* através do conector **GSincPCcon**.

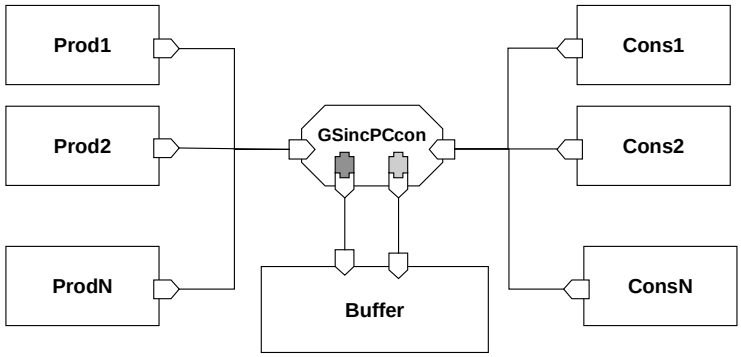


Figura 4.6 - Buffer limitado e sincronizado

4.3.2.2 Distribuindo os Módulos

Na versão anterior da aplicação os aspectos de sincronização e coordenação foram configurados e a aplicação atende aos requisitos impostos. Neste ponto, introduz-se mais um novo aspecto: os módulos da aplicação devem ser instanciados em pontos distribuídos.

Na realidade este aspecto está contemplado na definição do ambiente de configuração que tem por *default* que todos os módulos podem ser instanciados em pontos distribuídos. O ambiente de configuração pode ser preparado, inclusive, para distribuir a instanciação de módulos segundo um critério de balanceamento de carga.

Mesmo assim, para ilustrar os conceitos, a configuração anterior é alterada de forma que cada módulo seja instanciado explicitamente em uma estação diferente e por decisão de projeto os conectores devem oferecer a comunicação implementadas por RPC. Se esta última característica não fosse explicitada, o ambiente de execução selecionaria um tipo de conector adequado para realizar a comunicação dos módulos a partir de uma análise das referências dos endereços das estações onde as instâncias foram criadas e com base em um tipo de conector *defaul*t para realizar esta comunicação (um conector que utilize *sockets* para a comunicação, por exemplo). Suponha-se um módulo A instanciado na estação est1 e um módulo B em uma estação est2. Agora, suponha-se que um conector genérico tenha sido utilizado para interligar estes módulos na descrição da configuração. De forma automática, o sistema pode interligar estes módulos, utilizando o endereço IP das estações est1 e est2, e selecionando um conector que encapsule um subsistema de comunicação implementado por mecanismo de sockets ou RPC, por exemplo.

module DistribBoundedSincBufferApplClass { ...	
connector { ... **map** RPC; } RPCGSincPCcon; ...	Para explicitar que o conector implementa a interligação de módulos por RPC a cláusula **map** RPC é acrescentada.
instantiate Buffer **at** Station1 **instantiate** Producer **at** Station2; **instantiate** Client **at** Station3; ... }	A única diferença na configuração da aplicação além do conector é a indicação da estação onde o módulo deve ser instanciado. Esta referência pode ser, por exemplo, o nome DNS da máquina.

4.3.2.3 Impondo prioridade para os produtores

A flexibilidade no mapeamento de um conector para uma implementação, permite que o mesmo ofereça outras características além das definidas diretamente pela linguagem de configuração. Uma possibilidade é capacidade de filtrar as requisições que passam por ele, como mencionado na seção 3.3.1. Outras possibilidades podem ser exploradas na implementação de um conector através de uma linguagem de programação com recursos adequados e disponibilizando-se este conector para ser instanciados em uma aplicação.

No exemplo seguinte, a aplicação produtor/consumidor é modificada para impor um esquema de prioridade dos produtores em relação aos consumidores. Um novo conector, contendo a funcionalidade desejada é implementado por uma classe em Java que, internamente é programada para dar prioridade aos produtores. Os detalhes da programação deste conector não são apresentados. O objetivo deste exemplo é ilustrar a capacidade do modelo em acrescentar novos aspectos a uma aplicação, mesmo na ausência de extensões específicas na linguagem de configuração. As alterações em relação a aplicação original são apresentadas a seguir.

module PrioBoundedBufferApplClass { **connector** {	
map CODE Java “priocon.class”; } PrioCon;	A implementação deste conector é feita em Java e se encontra no arquivo *priocon.java*
instantiate ProiCon;	Apenas este conector é instanciado
link Produtor **to** Buffer **by** PrioCon; **link** Consumidor **to** Buffer **by** PrioCon; }	Neste ponto as ligações entre produtor, consumidor e buffer são feitas.

	Observa-se, mais uma vez, que não foram feitas referencias a nomes de instancias das portas dos módulos pois as ligações são feitas corretamente pela verificação dos tipos de portas envolvidas.

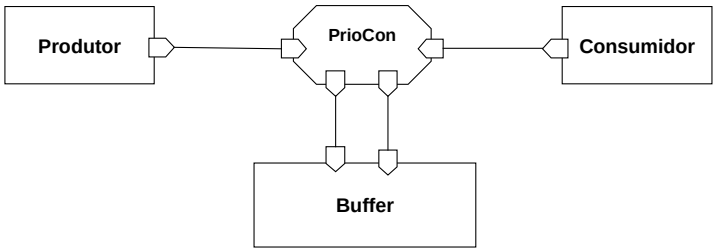


Figura 4.7 - Aplicação com conector impondo prioridade

4.3.3 Acrescentando Aspectos por Composição

Nos exemplos anteriores, os acréscimos de aspectos de sincronização e distribuição foram realizados por alterações incrementais na estrutura que descreve a configuração da aplicação. Uma alternativa disponível no modelo é a configuração de uma aplicação através de módulos compostos, ou seja, nem a configuração nem a implementação do módulo são substancialmente alteradas. Na solução por composição um módulo com a nova funcionalidade é concebido e agregado à aplicação original.

Uma das alternativas, no caso de nosso o exemplo é a construção de um módulo com o objetivo de receber os pedidos de clientes e servidores, sincronizar e coordenar este pedido e depois repassá-lo para o módulo Buffer. Vamos chamar este novo módulo de *CoposedBoundedBuffer*. Vamos utilizar, ainda, um módulo especializado em verificar a consistência e a prioridade das requisições antes de repassá-las para o módulo Buffer. Observa-se que neste caso, o projetista fez a opção por colocar características não-funcionais em um módulo e utilizá-lo na composição de um módulo mais complexo. Esta é uma possibilidade do modelo.

`module ComposedBoundedBufferClass {`
`  connector SincPCcon {`
`  condition vazio = true, cheio = false;`
`  staterequired int n_intes;`
`  int MAX_ITENS = 10;`
`  exclusive {`
`  out port PutT {`
`  inguard (vazio) {`
	O conector usado no módulo composto é semelhante ao do exemplo anterior.
	
	Foram acrescentados, apenas, os nomes das instâncias das portas de entrada e saída para

`after {`
`cheio = true;`
`if (n_itens == MAX_ITENS) {`
`vazio = false;`
`} } }`
`} RepassaPut;`
`out port GetT {`
`inguard (cheio) {`
`after {`
`vazio = true;`
`if (n_itens == 0) {`
`cheio = false;`
`} } }`
`} RepassaGet;`
`}`
	melhorar a legibilidade.

`in port PutT RecebePut;`
`in port GetT RecebeGet;`
`map RPC;`
`}`	

`module VerifClass (int Prio){`
`in port GetT RetiraItem;`
`out port GetT ForwardRetiraItem;`
 	
`in port PutT ArmazenaItem;`
`out port PutT ForwardArmazenaItem;`
 	
`map CLASS Java “verifier.class”`
`}`
	Neste módulo duas portas adicionais são incluídas para fazer o repasse dos pedidos vindos dos produtores e consumidores. Portanto, os métodos recebem as requisições pelas portas de entrada e as repassam pela porta de saída, depois de verificar e priorizar as requisições.
	
	Observa-se ainda que as portas de entrada deste módulo não serão sincronizadas. A prioridade dos consumidores será indicada por passagem de parâmetro no momento de sua criação.

`instantiate Buffer` `instantiate VeriClass as Verifier(4);`	As instâncias dos módulos Buffer e Verifier são criadas (a prioridade dos consumidores é 4).
`instantiate SincPCcon as Conect;`	O conector que vai interligar os dois módulos também é instanciados e ...
`link Verifier to Buffer by Conect;` `link Verifier to Buffer by Conect;`	os módulos são interconectados.

`export Verifier.ArmazenaItem;`
`export Verifier.RetiraItem;`
`} // ComposedBoundedBufferClass`
	Para finalizar a configuração deste módulo, as portas que devem ser *enxergadas* pela aplicação são exportadas.

`module CBBApplClass{`
`instantiate ProdItemT as ProduceItem;`
`instantiate ConsItemT as ConsumeItem;`
	A partir deste ponto a aplicação começa a ser configurada. Observa-se que com a disponibilidade dos elementos em uma biblioteca, a configuração da aplicação torna-se bastante simples.

`instantiate Producer;`
`instantiate Client;`
`instantiate CBBApplClass as CBBuf;`
	Os módulos necessários são instanciados.

`link Produtor.SaidaItem to`
`CBBuf.ArmazenaItem by ProduceItem;`
`link Consumidor.EntradaItem to`
`CBBuf.RetiraItem by ConsumeItem;`
`} // CBBApplClass`
	Em seguida os módulos são interconectados. A partir deste ponto, basta ser criada a instância da aplicação, como feito no primeiro exemplo. A figura 4.8 apresenta a aplicação configurada.

Neste exemplo foi possível explorar parte da capacidade de composição e reutilização

de módulos da linguagem de configuração. A idéia é ter disponível o um tipo de módulo Buffer genérico e, sem alterar sua natureza, estender sua funcionalidade compondo com outros módulos um elemento de funcionalidade mais refinada. Este novo módulo, por sua vez, também poderá ser disponibilizado no repositório e também poderá ser usado na composição de outras aplicações. Observa-se que para explorar melhor as possibilidades do modelo, o projetista deve utilizar a metodologia que está se sugerindo na proposta, procurando-se encapsular aspectos funcionais nos módulos e aspectos não-funcionais nos conectores, sempre que possível.

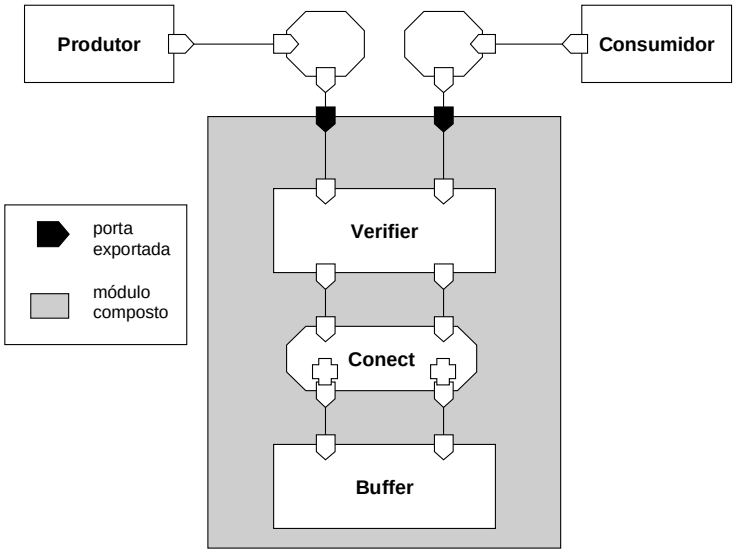


Figura 4.8 - Aplicação com módulo composto

4.3.4 Observação

Antes da prosseguirmos para a conclusão, uma observação deve ser feita com relação às versões do exemplo anterior: entre a execução de uma versão e de outra imediatamente posterior, o arquivo de configuração sofre as alterações necessárias, a aplicação em execução é terminada e a nova versão da aplicação reconfigurada é então, instanciada e iniciada. Esta seqüência de passos não é estritamente obrigatória. Uma das possibilidades admitidas no modelo é a execução das alterações na configuração de uma aplicação com a mesma em atividade. Não existe, a princípio, impedimento para que os comandos de configuração apresentados na seção 4.2.4 sejam executados por um agente externo à aplicação durante sua execução.

Na aplicação apresentada, o módulo *buffer* poderia ser submetido a uma seqüência de comandos de configuração (partindo de uma console de configuração, por exemplo) como *parar* e *terminar*, em seguida um outro módulo *buffer*, com as novas características configuradas seria *instanciado* e *ligado*. Isso poderia implicar na perda dos dados que estivessem armazenados no *buffer* original, problemas de consistência e em alguns problemas de comunicação.

Em geral, a reconfiguração de um sistema real, sem parada de operação, exige cuidados especiais de modo a não causar inconsistências na aplicação. Deve ser ressaltado que reconfigurações durante a operação não são incompatíveis com nossa proposta. Pelo contrário, a existência de uma descrição explícita da configuração facilita a implantação de diversas políticas e mecanismos de suporte à reconfiguração. Entretanto, questões sobre reconfiguração não serão tratadas como parte direta da proposta pois envolvem questões como a manutenção da coerência de estado da aplicação durante reconfigurações, que fogem um pouco das questões principais sendo tratadas para a concepção de aplicações distribuídas. Existem técnicas que permitem superar esses entraves possibilitando reconfigurar transparentemente um conjunto de módulos durante a operação, sem incorrer em perda nos serviços em execução [End 92]. Aspectos como persistência e atomicidade poderiam ser inseridos na arquitetura da aplicação, com esta finalidade.

4.4 Avaliação

Com a apresentação da linguagem de programação e analisando-se os exemplos apresentados é possível traçarmos um perfil mais amplo das características da proposta apresentada. Será, também, possível concluir a comparação, iniciada no fim do capítulo 3, entre a nossa proposta e os trabalhos avaliados no capítulo 2. Completamos, agora, a tabela 3.2 salientando como os requisitos de reusabilidade, separação de interesses, evolução dinâmica e abrangência são atendidos em nossa proposta.

Reusabilidade
Aplicações podem ser rapidamente prototipadas ou construídas a partir de módulos e conectores. Conectores podem encapsular padrões de interação entre módulos no mesmo sentido que é feito em *design patterns*. Módulos e conectores podem ser reutilizados para a

composição de módulos mais complexos. A composição de conectores permite a obtenção de conectores com novas características. Módulos compostos podem encapsular soluções de *software* para determinada área de aplicação, como fazem os *frameworks* no modelo de objetos.
Separação de Interesses
Separação de interesses é obtida com o projeto separado de aspectos diferentes da aplicação, integrados através da linguagem de configuração. Módulos representam, potencialmente, os componentes funcionais da aplicação. Conectores são utilizados para a interligação de módulos que vão interagir e a configuração de requisitos não-funcionais da aplicação. Em princípio, aspectos não-funcionais não interferem na concepção dos módulos. Estilos diferentes de interação dos componentes são planejados e implantados separadamente nos conectores. Existe a possibilidade do uso da reflexividade intrínseca aos conectores em ajustes finos das conexões dos módulos das aplicações. A composição de conectores para combinar diferentes aspectos também é possível.
Evolução Dinâmica
A evolução dinâmica é obtida pela capacidade de reconfiguração da aplicação durante sua operação. O esquema de mapeamento entre os elementos do modelo e a implementação permite que o encapsulamento de cada elemento seja preservado durante a execução. O suporte do configurador permite a realização de manobras na aplicação, como a substituição de componentes ou alterações nas ligações entre os módulos. Aliado a isso, a possibilidade de verificações de propriedades da arquitetura sendo manipulada permite que estas manobras sejam feitas com segurança.
Abrangência
A metodologia proposta oferece flexibilidade para o desenvolvimento de aplicações distribuídas. A facilidade de composição de elementos permite a combinação de funcionalidades para atender aos diversos requisitos das aplicações. Em princípio, o desenvolvimento de módulos e conectores pode ser feito em qualquer linguagem de programação e não existem restrições quanto aos recursos utilizados do sistema operacional hospedeiro.

Tabela 4.2 - comparação das propostas avaliadas

A linguagem de configuração foi concebida com dois grupos de declarações. O primeiro grupo é responsável pela declaração dos tipos e instâncias dos elementos a serem usados na aplicação. Neste grupo estão as declarações *module*, *port* e *connector*. O segundo grupo permite que se descreva as interligações destes elementos e o

momento de instanciá-los. *instantiate*, *link* e *start* estão neste grupo. Embora um comando de criação de uma instância possa ser considerado um comando imperativo, no contexto da linguagem de configuração ele apenas indica a ordem em que as instâncias devem ser criadas. Por outro lado, no contexto da operação, os mesmos comandos deste segundo grupo são efetivamente utilizados como comandos imperativos para o suporte à configuração. Este esquema foi preparado intencionalmente. O suporte à operação permite tanto que a aplicação seja instanciada para a execução, como reconfigurar-se a aplicação, através destes mesmos comandos. Isto facilita a preparação de roteiros de comandos de reconfiguração, que pode raciocinar em acordo com a linguagem de configuração e depois interagir diretamente com o suporte à configuração.

Os exemplos da seção 4.3 permitem observar que, à medida em que novos tipos de módulos, portas e conectores tornam-se disponíveis, a configuração de uma aplicação fica mais simples e menor. Esta facilidade poderá ser ampliada com a criação de elementos genéricos, realmente úteis e com a reunião destes elementos em módulos compostos e *frameworks*. Para ampliar esta capacidade de reutilização podemos introduzir um esquema de parametrização. Por exemplo, um conector pré-concebido com guardas genéricos pode receber as informações sobre as variáveis de condição ou pós-condições dos guardas através de parâmetros. Na realidade quando os atributos de um conector são descritos na linguagem de configuração eles podem funcionar como parâmetros para um gerador de código, se o conector ainda não foi implementado, ou para o suporte à configuração que adapta o conector [Lob 98].

Nossa linguagem possui algumas semelhanças com a linguagem de UniCon, no que se refere descrição de módulos e conectores. Ela se diferencia, entretanto, em vários pontos. O mais evidente é a configuração de aspectos. Não existe, em UniCon, o conceito de aspecto, apenas algumas características podem ser parametrizadas. Em UniCon alguns aspectos de interação entre componentes são programados através da seleção de um tipo de conector e através de anotações neste conector. Em UniCon alguns aspectos também podem ser configurados nos módulos através de parâmetros e anotações. Em nossa linguagem, além da configuração de aspectos disponíveis na linguagem, novos tipos de conectores podem ser introduzidos através da cláusula *map*. Uma outra diferença diz respeito à mecânica do modelo. Em UniCon, após a geração de

código executável, as informações sobre a arquitetura da aplicação deixam de existir. Podemos comparar este esquema de UniCon com o *Aspect Weaver* de AOP, onde os diversos aspectos programados são entrelaçados em um único código. Neste particular nossa proposta é semelhante à Regis. As estruturas das aplicações descritas em Darwin mantém o seu contorno durante a operação das mesmas, facilitando com isso a reconfiguração dinâmica. Entretanto, em Darwin, o programador é obrigado a usar recursos do ambiente de execução que implementa a comunicação entre componentes e gerencia a ligação dos mesmos. Em nosso esquema o mapeamento entre elementos do nível de configuração e suas respectivas implementações é transparente para o programador. Nenhum recurso primitivo é de uso obrigatório. Todo suporte necessário para a operação é fornecida pelo suporte à configuração.

Uma outra observação sobre a linguagem de configuração é sua semelhança com OMG-IDL. Embora, a sintaxe não seja exatamente a mesma, uma aplicação descrita através da linguagem de configuração pode ser facilmente convertida para IDL para, que sejam gerados *stubs* e *skeletons* de CORBA. De fato, nosso modelo admite conectores CORBA, bastando apenas que se indique esta característica através de uma cláusula *map* nos conectores. Além disso, o conjunto de tipos e o esquema de visibilidade da linguagem de configuração é o mesmo da IDL.

4.4.1 Pontos ainda não avaliados na proposta

O presente capítulo e o capítulo anterior não esgotaram os assuntos em torno da proposta. Esta é uma primeira abordagem. Alguns pontos importantes, ainda não trabalhados, serão discutidos brevemente em seções apropriadas no capítulo 5. Dentre estes pontos enumeramos alguns:

formalização - o modelo proposto pode ser complementado com uma abordagem formal. Vários partes do modelo como os elementos básicos (módulos, portas e conectores), a composição de módulos e conectores e a configuração de aspectos não funcionais podem ser revestidos de formalismo. A partir deste formalismo e a definição da linguagem de configuração, é possível a obtenção de informações sobre propriedades das aplicações construídas a partir do modelo, antes de se concretizar sua implementação. Pode ser estudada a possibilidade de se adaptar à nossa abordagem formalismos já propostos em outros trabalhos [All 96, Inv 95, Luc 95, Mor 95 e Nau

97].

aspectos de implementação - evitou-se nesta proposta a discussão sobre detalhes de implementação. Apenas o mapeamento dos elementos básicos do modelo de configuração para o nível de implementação foi abordado. Um levantamento das possíveis alternativas implementações para o suporte à configuração, serão apresentadas oportunamente.

evolução do modelo - um assunto importante neste trabalho é o estudo de alternativas para permitir a extensão da abrangência do modelo. Especificamente, deseja-se obter uma forma para acrescentar novos tipos de conectores e novos aspectos que possam ampliar a gama de problemas que possam ser concebidos baseados no modelo. Existem algumas alternativas possíveis que serão investigadas:

- a descrição do comportamento do conector através de algum formalismo e que esta descrição faça parte do conector;
- criação de um sistema de tipos de conectores que admita a inclusão de novos tipos, possibilitando também a inclusão de informações sobre o comportamento destes novos conectores. Estas informações seriam utilizadas pela própria linguagem de configuração;
- aumentar o comportamento reflexivo do conector através de algum mecanismo que permita interpor código em um conector existente para atender a novos requisitos, como proposto pelo mecanismo de filtros na seção 3.3;
- permitir a composição de conectores, como foi discutido na seção 3.4.

Dentre as alternativas apresentadas, as duas últimas parecem promissoras por permitir ao máximo a reutilização de componentes e por serem realizáveis com certa facilidade.

4.4.2 Comparação com AOP

A idéia de AOP evoluiu de sua primeira abordagem, contendo uma linguagem base e várias linguagens de aspectos, para um esquema com uma linguagem base e apenas uma linguagem de genérica para aspectos. Ainda existe o papel do *weaver*, um compilador especializado que mistura o código do programa base com os vários aspectos (ou *weaves*, como são chamados nas ultimas versões de AspectJ [AspJ 98A])

em um único programa. Esta linguagem de aspectos, mais genérica, permite dois tipos de construção. No primeiro o programador pode interpor código em pontos bem definidos no programa base, como antes de uma chamada a método ou depois da ocorrência de uma exceção. No segundo o programador pode introduzir novo código sem alterar código já existente (por exemplo, introduzir um método de inspeção do estado de variáveis em uma classe).

Esta nova abordagem da linguagem de aspectos de AOP é próxima da idéia de conectores em nossa proposta. O conector pode, como os *weaves*, alterar ou introduzir novas características aos módulos em pontos bem definidos, mais especificamente nos pontos onde existe interação entre os mesmos. Mesmo em caso de aspectos muito específicos, estes podem ser introduzidos nos conectores através do uso de filtros como apresentado na seção 3.3.6. Novas características também podem ser introduzidas, nos módulos por composição, sem a necessidade de quebra de encapsulamento, em nossa proposta. O programador também tem nas mãos, em nossa abordagem, estas duas alternativas.

Em nosso sistema, o *weaving* ocorre em duas situações:

- na interpretação da arquitetura descrita pela linguagem de configuração. Nesta fase pode haver geração de código a partir dos módulos e conectores descritos ou a geração de comandos de configuração para que a aplicação seja instanciada pela composição e ligação dos módulos e conectores selecionados.
- na execução dos comandos de configuração pelo configurador. O configurador faz um *weaving* em tempo de execução da aplicação efetivamente entrelaçando os componentes para criar a aplicação.

As verificações feitas pelo configurador para garantir a coerência dos comandos de configuração são equivalentes às feitas por um *weaver* AOP para programas fonte. No caso de nossa proposta o entrelaçamento de aspectos é feito também no nível da arquitetura indicada pela composição da estrutura da aplicação. Além disso, a própria relação entre os módulos é descrita separadamente e depois entrelaçada, definindo assim um outro aspecto, que podemos chamar de *arquitetural*.

Em nossos testes foi possível reproduzir utilizando-se a configuração, os exemplos apresentados em [AspJ 98A], levando-nos a concluir que além das equivalências

conceituais, o poder de expressividade das duas abordagens também é semelhante.

4.4.3 Reflexão

Nossa proposta possui características reflexivas. Podemos afirmar, inicialmente, que as interações entre módulos são interceptadas e reificadas para o conector que funciona como reflexo. Isto significa que as requisições e respostas intermediadas podem ser inspecionadas e manipuladas, em um meta-nível, dentro do conector, antes de serem repassadas para os módulos. O MOP disponível para o programador de aplicações inclui os pontos de junção de invocação de métodos, o retorno de respostas e avisos de ocorrência de exceções. Não é necessário a inserção de anotações no código dos módulos para avisar o uso de reflexão. Vale notar que o MOP utilizado na configuração já incorpora a substituição dinâmica de meta-objetos (os conectores, no caso). Isto permite que as aplicações evoluam dinamicamente substituindo componentes do meta-nível através de comandos de reconfiguração.

Em nossa proposta, a principio, não é oferecida a reflexão estrutural ou MOPs para reificação de eventos como a leitura e escrita de variáveis, a menos que isto ocorra através de métodos de acesso. Módulos e conectores não podem simplesmente investigar a estrutura interna de outros módulos ou conectores. A visibilidade de um módulo é restrita à sua interface. Nem mesmo no caso de módulos ou conectores compostos, elementos componentes não têm como fazer inspeção em suas estruturas. Por outro lado, no nível da estrutura da aplicação, a linguagem de configuração permite que o programador tenha controle sobre os módulos, incluindo suas assinaturas, para tomar decisões de projeto.

A solução utilizada para os conectores com guardas é a inclusão de métodos especiais que permitam o acesso ao estado do modulo para consulta pelo conector. Este método considera que o acesso ao estado deve ser feito em exclusão mútua para oferecer informações consistentes. Métodos de acesso ao estado, neste caso, podem ser inseridos automaticamente por um gerador de código ou pré-processador. Uma solução genérica para controle e acesso do estado dos módulos por conectores será investigado na solução deste trabalho. Esta solução deverá considerar problemas que podem surgir a partir de esquemas não triviais de concorrência entre os métodos de um módulo e deve ser genérica para permitir tanto a consulta como a alteração do estado dos módulos. A

implementação de conectores contemplando aspectos como a persistência e tolerância a falhas poderá ser automatizada se este ponto estiver equacionado.

4.4.4 Considerações sobre configuração, reflexão e AOP

Implementações de AOP, como AspectJ, operam sobre códigos fonte, tanto para a especificação funcional, como para a descrição de aspectos. O resultado do *weaving* é um programa contendo todos os aspectos de forma entrelaçada. A separação de aspectos se perde neste momento. Em [Mat 97], entretanto é advogado os aspectos "não deveriam morrer". O que se quis afirmar é que se os aspectos descritos pudessem manter o seu contorno mesmo após a geração do código final ou em tempo de execução, seria possível se pensar em adaptar estes aspectos durante a execução, sem a necessidade de códigos fonte ou a geração de novos módulos de execução.

A implementação de linguagens reflexivas normalmente utiliza duas alternativas: ou existe o controle sobre o ambiente de execução (interpretador ou sistema operacional), normalmente adotada quando se deseja oferecer reflexão dinâmica, ou a reificação e reflexão operam sobre programas fonte, caracterizando-se a reflexão estática. No primeiro caso, o ambiente deve ser especializado para interceptar eventos como a chamada de métodos, criação de objetos ou leitura e escrita em variáveis (MetaJava permite isso, por exemplo), reificar este evento e encaminhá-lo para o meta-componente correspondente. No segundo caso, usamos os fontes para misturar as partes descritas no meta-nível com as partes do código base de acordo com anotações apropriadas introduzidas no código (este é o caso de OpenC++, por exemplo). Neste segundo caso, identifica-se que a reflexão e AOP são muito parecidos, ou de outra forma, podemos dizer que AOP é uma forma de MOP .

A utilização de reflexão ou AOP em sistemas com módulos ou componentes já compilados não é trivial. Não existem mecanismos para que se informe os pontos de junção onde a reificação e reflexão devem ocorrer ou como entrelaçar aspectos em códigos executáveis. No caso da tecnologia de componentes [Bro 96], este problema tem sido contornando com o desenvolvimento destes em duas camadas. Uma camada implementa a funcionalidade prometida do componente. Uma outra camada, chamada de interface de meta-nível, permite que, em tempo de programação e projeto, se parametrize o componente para maior adequação. Parte desta interface de meta-nível

poderá ser utilizada durante a execução, de modo que o componente interaja com o ambiente de execução, capturando e emitindo eventos, ou possa receber pedidos de adaptação de algum sistema de controle da aplicação.

Nos outros casos em que não existe acesso ao ambiente de execução ou aos códigos fontes dos módulos, uma forma de se adaptar o comportamento de um modulo de uma aplicação é intermediando explicitamente sua interação com outros módulos. Esta técnica tem sido utilizada através de *wrappers* [Bir 98, Höl 93], filtros de composição e conectores, como estamos propondo. Cada uma destas soluções intercepta de alguma forma as requisições e respostas entre os módulos e executa algum tratamento sobre as mesmas, afetando o comportamento deste módulos. De uma forma geral estas técnicas não precisam dispor do código fonte dos módulos ou de privilégios de acesso ao ambiente de execução para permitir a adaptação da aplicação.

A utilização de configuração e conectores oferece a possibilidade de adaptação da aplicação em um meta-nível e procura afetar minimamente os componentes da aplicação e o ambiente de execução. Conectores intermediam toda interação entre módulos e são flexíveis o suficiente para manipular estas interações, em um nível diferente dos componentes envolvidos (um meta-nível), adicionando aspectos como sincronização, comunicação, tolerância a falhas ou crítica de dados. Não existe a necessidade de anotações internas nos módulos e a interceptação dos pontos de junção é feita naturalmente quando ocorrem eventos como a invocação de métodos. Existe entretanto a necessidade de se adicionar ao ambiente de execução alguma forma para se armazenar e manipular meta-informações a respeito dos componentes das aplicações. Em nossa proposta isto é feito pelo **configurador**, que pode ser considerado um módulo especial, que é responsável por instanciar, ligar ou terminar componentes, ao mesmo tempo que armazena informações importantes sobre estes componentes para futuras consultas. Com estas informações por exemplo pode-se substituir um componente ou reestruturar-se a aplicação, independentemente de como os módulos foram implementados ou do sistema de execução.

4.4.5 Otimização

Um dos recursos que pode ser agregado à linguagem de configuração é a capacidade de otimização da arquitetura descrita. Da mesma forma que um compilador de uma

linguagem de programação tradicional pode otimizar o código de máquina a ser gerado, a linguagem de configuração pode levar a uma otimização da arquitetura da aplicação sendo descrita. É possível por exemplo, que seja detectada a necessidade de se fundir conectores básicos de um conector composto, ou dividir um modulo para minimizar o fluxo de requisições.

No exemplo da figura 4.9a temos uma aplicação em que a interação entre os módulos cliente e servidor é intermediada por uma cadeia de 3 conectores, A, B e C. Independentemente dos aspectos implementados por cada conector, o fluxo de informação da requisição do módulo cliente é tratada pelos três conectores, o mesmo ocorrendo com a resposta devolvida pelo servidor. Pode ocorrer, entretanto, que a resposta não tenha a necessidade de ser manipulada por estes conectores. Por exemplo, considere-se que o conector A faça criptografia, o conector B faça um registro de transação e o conector C faça a decriptografia. Considere-se também que a resposta seja simplesmente um código de reconhecimento avisando ao cliente que a requisição foi completada. Neste caso, não existe a necessidade da resposta passar por A, B e C, bastando um conector para intermediar o encaminhamento da informação (conector *return* da figura (b)).

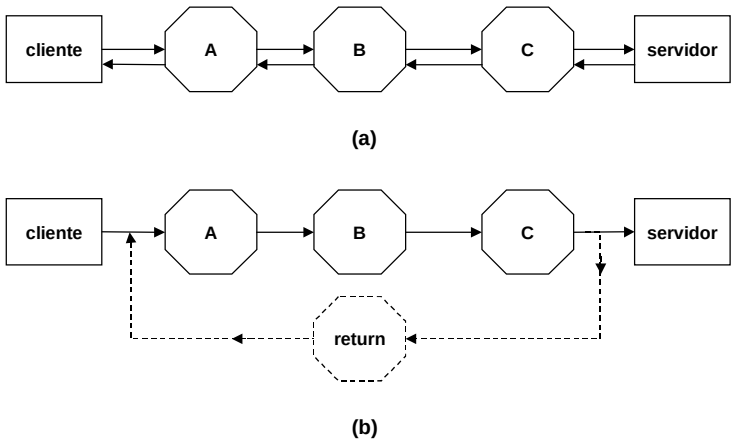


Figura 4.9 – (a) fluxo normal de requisições e respostas em uma cadeia de conectores; (b) possibilidade de otimização do fluxo de resposta

Um interpretador para a linguagem de configuração pode ser capaz de otimizar esta estrutura criando um conector otimizado, como ilustrado na figura 4.10. Neste exemplo, adaptam-se os conectores A, B e C para que os mesmos não fiquem

bloqueados aguardando um retorno e acrescenta-se um conector de retorno que encaminhará as respostas. Para que o estilo de comunicação síncrona do arranjo inicial seja preservado, um par de conectores adicionais são instalados mais próximos aos módulos, adaptando o estilo da interação quando necessário.

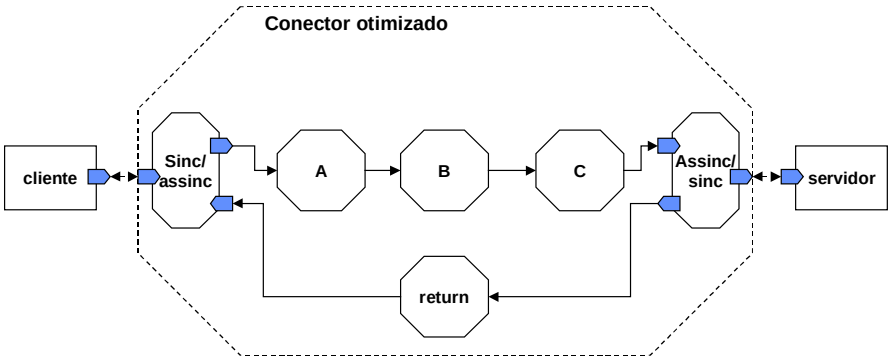


Figura 4.10 - conector composto descrevendo uma cadeia de conectores com fluxo otimizado

A inteligência para realizar estas otimizações deve ser inserida no interpretador ou em uma ferramenta auxiliar. A abordagem utilizada nesta proposta e as técnicas de verificação formal disponíveis na literatura (discutidas brevemente no próximo capítulo) facilita a construção destas ferramentas.

4.5 Conclusão

Uma primeira versão da linguagem de configuração foi apresentada. Esta linguagem é essencialmente declarativa e permite descrever-se a arquitetura de aplicações através de módulos e conectores. A sintaxe para a configuração de dois aspectos não-funcionais, coordenação e distribuição foi integrada à linguagem de configuração. Através de exemplos, os conceitos do modelo e a configuração de aspectos não-funcionais foram aplicados. Observou-se que a possibilidade de seleção e composição de elementos na descrição da estrutura de uma aplicação e a facilidade de configuração de aspectos não-funcionais favorece a reutilização de componentes e a concepção de aplicações customizadas para necessidades específicas.

Uma análise destes exemplos permite situar preliminarmente a proposta no que tange ao atendimento dos requisitos enumerados no capítulo 1. O modelo apresentado possui as seguintes características:

- permite a reutilização de componentes funcionais e não-funcionais, facilitada pela imposição da modularidade na descrição da estrutura de aplicações;
- é orientado a separação de interesses oferecendo a possibilidade de se programar os aspectos funcionais nos módulos e os aspectos funcionais nos conectores. Com isso, o projetista pode adaptar com bastante conforto a aplicação sendo concebida para vários ambientes de operação. Esta característica por sua vez amplia as possibilidades de reutilização de componentes.
- tanto no nível de configuração, quanto no nível de operação, a aplicação pode ser adaptada para atender a novos requisitos. O modelo de configuração, a linguagem de configuração e uma metodologia adequada facilitam a evolução dinâmica das aplicações pela possibilidade de seleção e substituição de componentes funcionais e não-funcionais, e pela possibilidade de adaptar a estrutura de ligação destes componentes.
- permite o desenvolvimento de um grande número aplicações distribuídas, que pode ser aumentado com a inclusão de novos conectores. A linguagem de configuração não está associada a uma linguagem específica de programação, facilitando sua aplicabilidade.

O próximo capítulo, será destinado à discussão de alguns detalhes da proposta, à observações e conclusões. O objetivo é indicar os rumos a serem tomados na continuação das pesquisas. Dentre os assuntos abordados são apresentados os pontos que devem ser aprofundados como a semântica da linguagem de configuração e as possibilidades para se formalizar o modelo. No prosseguimento são feitas considerações sobre arquiteturas para implementação de um protótipo de ambiente de configuração baseado no modelo e problemas esperados para sua aplicação prática. Finalmente são feitas algumas observações sobre o modelo proposto, sobre o estado das pesquisas e as contribuições esperadas no prosseguimento das mesmas.

Capítulo V

Conclusão

Neste capítulo final completamos a apresentação de nossa proposta de tese com uma discussão a cerca de seus resultados e pontos de aprimoramento. Esta discussão está dividida em duas partes. Na primeira parte apresentam-se as perspectivas de continuação das pesquisas para a versão final da tese. Na parte seguinte concluímos este trabalho com algumas considerações sobre os resultados obtidos com a nossa proposta até o presente momento e situando a mesma em relação aos resultados esperados.

5.1 Próximos Passos

Nesta seção discutimos os assuntos que continuarão sendo investigados no prosseguimento deste trabalho, aprimorando a proposta apresentada. Dentre os principais pontos a serem abordados enumeramos:

- consolidação da sintaxe e semântica da linguagem de configuração;
- refinamento dos mecanismos para a composição de conectores como fator de reusabilidade e expressividade da proposta. Este assunto vem sendo pesquisado em trabalhos correlatos como [Wel 98, Agh 98 e Oli 98];
- generalização do conceito de conector para possibilitar a sua utilização na implementação de outros aspectos não-funcionais, além dos discutidos no capítulo 3, e ampliar a abrangência da proposta para qualquer tipo de aplicação;
- discutir mais profundamente como a linguagem de configuração pode ser utilizada para "colar" módulos e conectores desenvolvidos em linguagens de programação diferentes ou mesmo através de *middlewares* disponíveis comercialmente como CORBA e DCOM;

- estudo de alternativas possíveis para implementar as idéias contidas em nossa proposta, como por exemplo um interpretador para a linguagem de configuração e um ambiente de execução;
- validação da tese através de exemplos.

Nas próximas subseções é iniciada uma discussão sobre alguns temas que se enquadram dentro dos pontos relacionados e que se destacam por sua importância. Sabemos, entretanto, que estes pontos deverão ser estudados com maior profundidade na seqüência deste trabalho.

5.1.1 Sistema de Tipos para a Linguagem de Configuração

Na concepção da linguagem de configuração a necessidade de um sistema de tipos foi avaliada preliminarmente. A definição das assinaturas das portas, por exemplo, deve permitir que sejam descritos conjuntos de tipos de parâmetros. Com esta finalidade o conjunto de tipos da OMG-IDL está sendo utilizado. Assim sendo, todos os tipos admitidos na IDL também serão admitidos na linguagem de configuração.

Ficam em aberto dois pontos importantes: tipos ou classes de módulos e conectores. A utilização formal de tipos ou classes de módulos na linguagem de configuração deverá ser analisada mais amplamente. No estado atual da proposta estamos utilizando o conceito de classes de módulos e conectores como solução para se fazer referencia a *fôrmas* básicas a partir das quais instâncias são criadas. Como os módulos encapsulam, basicamente, aspectos funcionais da aplicação, nenhum suporte adicional parece ser necessário na linguagem para estes elementos no que tange a linguagem de configuração. Não é objetivo principal da linguagem de configuração interferir na semântica da aplicação. Entretanto, o estudo desenvolvido no capítulo 3 para o mapeamento de módulos para implementações em linguagens de programação poderá ser ampliado, como sugerido no início desta seção, levando a uma definição mais específica para tipos ou classes de módulos na linguagem de configuração.

O uso de tipos ou classes de conectores apresenta implicações diferentes em comparação com os módulos. Podemos tratar, em uma primeira abordagem, os conectores de forma semelhante aos módulos. Na linguagem de configuração podem ser definidas classes diferentes de conectores. Uma classe de conector é caracterizada

inicialmente por seu nome e pela assinatura das portas declaradas. O comportamento deste conector está associado com as anotações e cláusulas caracterizando aspectos não-funcionais, incluídas na sua declaração, e um mapeamento para uma implementação. Entretanto, não existe, nesta primeira abordagem da linguagem de configuração, uma verificação de propriedades da estrutura da aplicação sendo descrita, que considere o comportamento (ou semântica) dos conectores sendo usados. Apenas as diferenças e características sintáticas dos elementos descritos serão consideradas. Inicialmente, em nossa proposta, conectores possuem um comportamento genérico.

Alguns trabalhos correlatos dirigem seus esforços para o desenvolvimento de um conjunto de tipos de conectores que possa ser utilizado na descrição formal de arquiteturas de software, como é o caso de UniCon. Tipos de conectores especializados fazem parte da linguagem de configuração e cada tipo pode implementar um estilo de interação distinto. Em UniCon, a sintaxe para a definição de um conector apresenta uma cláusula do tipo **CONNECTOR_TYPE**, com a qual o tipo de conector é indicado (vide capítulo 2). O objetivo, neste caso, é fazer verificações sobre as descrições de arquiteturas, com base nas características implícitas em cada tipo de conector. Assim, além das características sintáticas, o comportamento que cada tipo de conector encerra é relevante para linguagem.

Temos a perspectiva de que durante a fase de refinamento do modelo e na implementação de exemplos, as implicações de um sistema de tipos para conectores seja amadurecida. O que está claro, até o momento, é que podemos utilizar conectores já implementados ou novos conectores, desde que suas interfaces estejam claramente identificadas (o que é o caso, em conseqüência do uso explícito do conceito de portas). Isso já nos permite a verificação da compatibilidade das portas dos módulos que são interligados através de um dado conector. Não existem, tampouco, grandes dificuldades para se utilizar um novo conector, bastando apenas que se faça a associação com a implementação correspondente quando da declaração do mesmo. O problema surge quando verificações mais complexas precisam ser feitas a partir de uma descrição de arquitetura. Nestes casos, a semântica dos conectores deve ser conhecida pela ferramenta que vai executar a verificação e isto pode ser obtido por um sistema de tipos específico para conectores. Uma solução direta seria a introdução deste conhecimento, na linguagem de configuração, para um conjunto de tipos populares de conectores. Para

o caso da introdução de novos conectores outras soluções devem ser consideradas, como discutido na próxima seção.

5.1.2 Adicionando Novos Conectores

A adição de novos conectores, que ofereçam características de interação diferentes (e possivelmente novos aspectos) das oferecidas pelo conjunto de conectores conhecidos pelo configurador, está ligada à questão de tipos de conectores. No que diz respeito à nossa proposta, podemos investigar a adição de novos tipos de conectores, como um problema de se introduzir informações ou regras sobre o comportamento e a semântica destes elementos na linguagem de configuração e nas ferramentas de verificação que possam estar associadas a mesma.

A extensão sintática da linguagem para permitir a descrição de novos tipos de não é muito difícil. Existem algumas alternativas, como por exemplo, parametrizar a sintaxe que declara conectores para facilitar esta tarefa. Esta alternativa pode não ser elegante, mas deve funcionar. Uma segunda alternativa é a geração de uma nova versão da linguagem de configuração e ferramentas associadas, contendo as extensões sintáticas necessárias para o tratamento dos novos conectores. Esta alternativa significa alterar-se compiladores e outras ferramentas, o que não é conveniente para o usuário na maioria dos casos.

Pelo ponto de vista do configurador que tem o papel de efetuar comandos de configuração, não existem maiores dificuldades em se instanciar novos tipos de conectores ou configurar novos aspectos, desde que os mesmos tenham sido implementados e o mapeamento necessário para esta implementação tenha sido descrito. Para o suporte à configuração, os módulos e conectores podem representar unidades de execução, com uma identificação e parâmetros mantidos em um repositório. Isto é suficiente, em princípio, para que o configurador possa instanciá-los e interconectá-los. O comportamento específico destes elementos não tem muita relevância neste nível. Em [Szt 95], por exemplo, utilizando-se um esquema com objetos compartilhados e ligação dinâmica, um módulo poderia ser terminado e uma outra instância do mesmo poderia ser criada e interligada à aplicação a partir de uma nova versão de sua implementação.

Entretanto, se o objetivo é da extensão é permitir que a partir da linguagem de configuração e possíveis ferramentas de verificação agregadas à linguagem, se obtenha informações sobre o comportamento, desempenho ou aderência a padrões de projeto, a extensão sintática não é suficiente. As verificações neste contexto são feitas com base nas informações da semântica dos elementos. Por exemplo, o comportamento de um novo conector ou seus requisitos para operação são importantes para a apuração de problemas potenciais de operação de uma aplicação. Assim sendo, a inclusão de novos tipos de conectores deve contemplar a inclusão das informações sobre a sintaxe e semântica dos mesmos.

Uma outra alternativa para permitir a extensão da linguagem de configuração, mencionada no capítulo 1, é a utilização do conceito de linguagem aberta. Linguagens abertas tem como objetivo facilitar alterações e extensões da própria linguagem. Em [Kic 97], [Chi 95] e [Duc 97] esta alternativa é utilizada como mecanismo para introduzir reflexividade nas linguagens. Nos parece, entretanto, que mesmo neste caso, a tarefa de incluir novos elementos não é trivial (ver capítulo 2). Não seria uma tarefa própria para o projetista de uma aplicação, que deveria estar concentrado na concepção da estrutura da mesma, se preocupar com extensões da linguagem de configuração. Além disso, abrir a linguagem de configuração pode acarretar problemas potenciais de segurança se mal utilizados. As ferramentas para verificação formal de arquiteturas descritas por configuração, devem evoluir paralelamente às inovações (novos conectores) introduzidas na linguagem de configuração para continuarem sendo consideradas como base para comparações e verificações das aplicações produzidas ou devem avisar que as verificações serão limitadas ao conhecimento existente.

No capítulo 3, a utilização de filtros e a composição de conectores é sugerida como alternativa imediata para ampliar a funcionalidade de conectores. Através dos filtros podemos acrescentar funcionalidade ao conector, introduzindo-se código ao mesmo. A composição de conectores permite que um novo conector seja concebido, combinando-se conectores disponíveis. Embora estes mecanismos sejam efetivos para a criação de novos conectores, as questões de verificação de consistência do novo conector ou da aplicação onde ele é utilizado ainda persistem. Quando o mecanismo de filtros é utilizado, a nova funcionalidade é opaca à linguagem de configuração. Se isto não pode ser admitido, algum mecanismo formal para indicar à que o filtro se propõe deve estar

disponível. A composição de conectores, suportada pela linguagem de configuração, também poderia ser complementada com algum mecanismo para verificar-se automaticamente as propriedades do conector resultante. Podemos recair, assim, no problema inicial identificado no início desta subseção, que é a introdução de informações sobre a semântica de novos elementos.

Contratos

Uma das formas possíveis para solucionar a falta de uma notação para o comportamento dos conectores é o uso do conceito de contratos [Hel 90], também chamadas de *roles & glue* (papéis e cola) em [All 97 T]. O conector se apresenta como lugar ideal para se expressar um contrato. Um contrato identifica formalmente como se dará a interação entre duas entidades. Para isso é necessário que se conheçam as entidades envolvidas na interação, suas interfaces e como devem ocorrer a trocas de informações através delas. Uma vez determinados todos estes itens estabelece-se um contrato, que deve ser respeitado por todos os envolvidos. A forma com que se expressam os contratos podem variar, dependendo do ambiente ou linguagem de programação [Hel 95, Mal 96]. Normalmente uma notação com bases formais é utilizada, garantindo a clareza do que se espera do contrato. Por outro lado, existem casos em que determinados contratos não podem ser expressos através de qualquer formalismo.

Em nossa proposta, um contrato pode seu usado para expressar o comportamento de um conector. A implantação desta idéia nos conectores é factível, uma vez que o conector conhece a interface dos módulos por ele conectados e já inclui alguma forma de expressar o comportamento e aspectos da interação destes módulos através de anotações e cláusulas de mapeamento. O uso de contratos adicionaria um formalismo para se expressar este comportamento, permitindo-se obter informações que poderiam ser traduzidas como ortogonalidade ou coerência, quando conectores fossem compostos. Nos conectores poderíamos estabelecer contratos para os diversos aspectos das aplicações. Assim, teríamos um contrato para comunicação, sincronização, exclusão mútua e assim por diante. Como os contratos são investidos de caráter formal, a solução para permitir a composição automática de conectores estaria mais próxima.

Os pontos levantados aqui também serão parte importante na continuação das pesquisas

para a tese.

5.1.3 Alternativas para Implementação do Modelo

Uma das etapas da seqüência deste trabalho inclui o estudo de alternativas para a implementação dos conceitos da proposta. Uma parte importante nesta tarefa, já abordada na seção 3.2.5 é o mapeamento de módulos e conectores no nível da implementação. Este mapeamento é importante para a construção de ferramentas como compiladores para a linguagem de configuração e geradores automáticos da código. A arquitetura de um ambiente de configuração contendo ferramentas úteis pode ser concebida, utilizando-se o modelo e a metodologia de configuração propostas. Isto validaria, de certa forma, o próprio modelo. Dentre as partes desta arquitetura pode-se prever:

- um compilador ou interpretador para a linguagem de configuração. Esta ferramenta poderá permitir que as descrições das estruturas de aplicações sejam convertidas automaticamente em módulos executáveis ou resultem em comandos de configuração para um ambiente de configuração;
- um verificador de propriedades. Esta ferramenta poderá ser integrada ao compilador, permitindo-se extrair algumas propriedades tais como, a coerência de módulos ou conectores compostos e da estrutura descrita, a viabilidade da aplicação ser executada atendendo à configuração solicitada para os aspectos não-funcionais, entre outras;
- um ambiente de suporte à configuração, que materialize os conceitos de configuração e permita a execução de comandos de configuração para que as aplicações possam ser instanciadas e executadas. Adicionalmente, este ambiente de configuração pode ser concebido de forma que o contorno de módulos e conectores sejam mantidos mesmo durante a execução, possibilitando que a mesma possa ser reconfigurada;
- uma ferramenta gráfica de gerência que facilite os usuários nas tarefas de concepção das arquiteturas e gerência das aplicações.

A implementação de uma ferramenta não precisa passar necessariamente pela construção de todas as outras listadas. A verificação dos conceitos e a implementação

de exemplos pode ser feita com o mapeamento manual da aplicação em unidades de programação em uma linguagem como Java, por exemplo. Será necessário, entretanto, estender o ambiente de execução para que os comandos de configuração possam ser utilizados. Esta ultima fase já esta sendo desenvolvida, e os primeiros resultados podem ser apreciados em [Lob 98A e Lob 98B].

5.1.4 Validação

Os conceitos apresentados em nossa proposta serão validados através de exemplos. Com este objetivo apresentaremos a construção de uma série de aplicações distribuídas e não distribuídas, de naturezas diferentes, utilizando o modelo de configuração e a metodologia propostas. Em cada exemplo, a arquitetura da aplicação será descrita na linguagem de configuração com seus módulos e conectores. O mapeamento para uma linguagem de configuração também será discutido e implementado. Alguns cenários serão testados, onde a aplicação tenha que ser reconfigurada. Sempre que possível, comparações serão feitas entre a solução concebida por configuração e soluções alternativas utilizando outras propostas. Além disso, para cada exemplo serão verificadas como as características de reusabilidade, separação de aspectos e evolução dinâmica são contempladas. Acreditamos que com a preparação destes exemplos, poderemos obter subsídios para aperfeiçoar a proposta e comparar sua aplicabilidade.

5.2 Resultados Preliminares

Na primeira parte do trabalho procurou-se avaliar quais são as necessidades e requisitos durante o ciclo de vida de aplicações distribuídas de grande tamanho, desde a fase de concepção até a fase de operação. A partir desta avaliação chegou-se a um conjunto de requisitos, que consideramos essenciais, relacionados a este tipo de aplicação: separação de interesses, reusabilidade, evolução dinâmica e abrangência. Este conjunto de requisitos é genérico o bastante para englobar outros requisitos importantes, como por exemplo, a modularidade, intrínseca à reusabilidade, e específico o suficiente para caracterizar aplicações distribuídas.

A partir deste conjunto de requisitos, foram avaliadas as tecnologias de objetos, reflexão computacional, programação orientada a aspectos e configuração,

consideradas como alicerces para o desenvolvimento de grandes sistemas. Esta avaliação objetivou verificar de que forma estas tecnologias podem contribuir para a concepção de aplicações distribuídas e como os requisitos enunciados anteriormente seriam atendidos. Como resultado foi listada uma serie de parâmetros para a concepção de sistemas distribuídos a partir destas tecnologias. Concluímos que a tecnologia de objetos pode ser utilizada como fundamento em qualquer aplicação onde a modularidade e encapsulamento de aspectos seja atrativos. Reflexão computacional pode ser utilizada em um nível fino de granularidade para estender a funcionalidade dos objetos. Entretanto, esta tecnologia é melhor utilizada como uma técnica de projeto do que como mecanismo de programação em situações que não seja possível utilizar-se linguagens ou ambientes reflexivos especializados. Programação orientada a aspectos é uma solução interessante para a separação de interesses, mas exige a inteligência de um *weaver* para entrelaçar os vários aspectos em um só bloco de código, perdendo-se a partir dai as informações sobre os diferentes aspetos programados, o que não é satisfatório para a evolução dinâmica. Verificamos, então, que a tecnologia de configuração reúne características necessárias para a concepção de aplicações distribuídas, agregando várias características das outras tecnologias, e pode ser usada em vários níveis de granularidade, tanto para a concepção de módulos complexos, como para grandes aplicações. Como contribuição adicional, observamos que após a avaliação das tecnologias, o conjunto de requisitos selecionados para esta avaliação mostrou-se bastante adequado.

Concluída esta pesquisa inicial, formulamos uma proposta que integra a abordagem de configuração à reflexão e programação de aspectos. Esta abordagem permite a descrição de várias classes de aplicações distribuídas ou não. Nossa proposta permite que se descreva a arquitetura de uma aplicação mantendo-se a separação de aspectos funcionais e não-funcionais. Esta descrição é feita a partir de componentes e a interligação destes componentes. Neste modelo são admitidos dois tipos de componentes: módulos, que podem encapsular aspectos funcionais, e conectores que intermediam a interação de módulos e podem encapsular aspectos não-funcionais. Além disso, este modelo permite manter-se um acoplamento fraco entre os diferentes aspectos e ainda assim manter a coesão da estrutura sendo descrita. Isto é possível pela descrição explícita da interconexão dos elementos que encapsulam estes aspectos e

utilizando-se uma metodologia que favoreça este esquema. O conceito de módulos e conectores admite a idéia de composição, permitindo que módulos e conectores complexos sejam concebidos a partir de elementos mais simples. Isto facilita reutilização destes componentes, maximizando a produtividade no projeto das aplicações.

5.3 Principais Contribuições

A primeira parte de nosso trabalho, o estudo de requisitos e tecnologias para a concepção de grandes sistemas distribuídos, levou a um conjunto de diretivas de como aplicar estas tecnologias e que requisitos elas atendiam. Como contribuição maior, entretanto, este estudo comparativo nos levou a uma visão integrada das tecnologias de objetos e reflexão computacional. Esta visão nos permitiu selecionar as melhores características de cada tecnologia e identificar as interseções dos seus conceitos. Propostas correlatas como AOP - *Aspect Oriented Programming* [Kic 96] e linguagens que oferecem reflexão computacional [Gol 97T], além de outros ambientes de configuração [All 94, Mag 92 e Sha 95], foram analisadas. Essas propostas utilizam alguma forma de interceptação do fluxo de execução e interposição de código [Joh 93] para obter a separação de interesses. AOP, por exemplo, possui uma ferramenta chamada *weaver*, que entrelaça o código de aspectos programados separadamente em um único bloco [Lop 97T]. Linguagens de programação com características reflexivas normalmente utilizam uma combinação de identificação e interceptação do fluxo de execução do nível base (*reificação*), para disparar a execução de um código interposto no meta-nível (*reflexão*). Nos dois casos, existe uma ferramenta que administra a interceptação e interposição de código. Identificamos também que a tecnologia de configuração pode ter características semelhantes a cada uma das tecnologias estudadas, e oferece explicitamente um esquema para descrever a arquitetura das aplicações através de composição e a possibilidade de verificações de propriedades dessas arquiteturas. Isto facilita a concepção de aplicações em suas várias fases.

Com isto em mente, uma nova abordagem para a construção de aplicações distribuídas foi proposta. Nesta abordagem, baseada em configuração, utiliza-se conectores para intermediar as interações entre módulos e também encapsular aspectos não-funcionais das aplicações. A arquitetura de uma aplicação é descrita através de uma linguagem de

configuração e os vários aspectos envolvidos podem ser tratados separadamente. As características que destacam esta metodologia são:

- uso de módulos e conectores para descrever aspectos funcionais e não-funcionais de aplicações, além do uso tradicional destes elementos, em ambientes de configuração, para descrever a arquitetura e estrutura das aplicações;
- uso de conectores, um elemento intermediador de interações entre módulos, como elemento de para possibilitar a evolução dinâmica de aspectos não-funcionais das aplicações;
- possibilidade do uso da reflexividade intrínseca aos conectores na concepção e em ajustes finos das interações dos módulos das aplicações;
- composição de módulos funcionais e conectores como mecanismo para maximizar a abrangência do modelo e reusabilidade de elementos.

5.4 Conclusões

Neste trabalho introduziu-se uma alternativa para a concepção de aplicações distribuídas através de um modelo baseado em configuração, utilizando-se também os conceitos de reflexão e separação de aspectos. Neste modelo, aspectos funcionais das aplicações são encapsulados em módulos que são interligados entre si por conectores. Os conectores são elementos com vocação para intermediar as interações entre módulos e também encapsular aspectos não-funcionais das aplicações. A arquitetura de uma aplicação, neste modelo, é descrita através de uma linguagem de configuração. O modelo facilita a concepção de aplicações distribuídas na medida em que os vários aspectos envolvidos podem ser abordados separadamente. Módulos já implementados podem ser reutilizados e a estrutura da aplicação rapidamente testada.

Sabemos que a implementação destes conceitos não é trivial se o objetivo é oferecer a separação de aspectos durante todo o ciclo de vida de uma aplicação, inclusive durante a execução. Assim sendo, também foi proposto um mapeamento do modelo conceitual para o nível de implementação. Este mapeamento poderá ser feito através de ferramentas automáticas e pode ainda contar, eventualmente, com ambientes especializados.

Amadurecemos a conclusão de que o modelo de configuração proposto utiliza reflexão comportamental para inserir aspectos não-funcionais através dos conectores. Entretanto o esquema reflexivo usado difere do que se tem feito tradicionalmente através de linguagens especializadas ou ambientes de suporte a reflexão dinâmica. O que ocorre no modelo proposto é que os conectores podem controlar a interação entre os módulos, uma vez que eles interceptam toda comunicação, caracterizando assim a reflexão computacional.

Ficamos convencidos ao longo da elaboração desta proposta que o modelo é genérico o suficiente para ser utilizado em sistemas que não sejam necessariamente distribuídos. A rigor podemos admitir que sistemas não-distribuídos são casos particulares de sistemas distribuídos. Aspectos que não sejam ligados à distribuição e comunicação também podem ser configurados nas aplicações, desde que, de alguma forma, digam à respeito a interação entre módulos. Desta forma, um conector pode conter a expressividade, por exemplo de um *design pattern* do tipo ***Observer*** [Gam 94], que trata da geração de eventos e da reação esperada de *observadores*. Efeito semelhante foi proposto em [AspJ 98C] concentrando a implementação do protocolo de geração e tratamento de eventos na descrição de um aspecto. Outros aspectos, além deste, como a validação do conteúdo dos parâmetros de requisições, controle de versões ou a persistência dos módulos, que não são específicos de sistemas distribuídos, também podem ser providos por configuração.

A experiência com a utilização do modelo durante a concepção dos exemplos do capítulo 4 foi satisfatória. Cada exemplo foi selecionado para explorar as potencialidades da proposta e da linguagem de configuração. Pela comparação do mesmo conjunto de exemplos desenvolvidos a partir da nossa abordagem e com protótipos de outras propostas [Szt 97, Szt 99], verificou-se que R-RIO possui expressividade equivalente, e pode ser mais flexível do que outras abordagens. Adicionalmente, a abordagem utilizada em nosso modelo permite um tratamento integrado do gerenciamento das aplicações desenvolvidas, desde a concepção até a execução e manutenção, diferentemente das outras propostas avaliadas, onde isso é feito de maneira *ad-hoc*. Nossa proposta de configuração, por exemplo, permite que a mesma metodologia utilizada para conceber a arquitetura de uma aplicação seja utilizada na sua reestruturação, em resposta a novas demandas, mesmo depois desta ser

implantada e estar em produção. Esta característica torna R-RIO uma base para o desenvolvimento de aplicações que possam evoluir dinamicamente.

Por fim, os próximos passos a serem dados neste trabalho foram delineados. O modelo será aperfeiçoado em vários pontos. A possibilidade da inclusão de novos aspectos será experimentada e as linhas principais para a implementação de um protótipo em que se possa testar as idéias contidas no modelo serão discutidas. A capacidade de composição e reflexão do modelo serão exploradas mais profundamente para investigar sua abrangência a domínios diferentes de aplicações distribuídas e não distribuídas. A avaliação preliminar do modelo nos oferece as indicações para prosseguirmos nas direções traçadas. Acreditamos que, concluída a tese, as contribuições para a concepção de grandes aplicações distribuídas estarão ainda mais evidentes.

Referências Bibliográficas

[Ach 93] Achauer, B., "The DOWL Distributed Object-Oriented Language", Communications of the ACM, Vol. 36, No. 9, pp. 48-54, Setembro, 1993.

[ADA A] Ledgard, H., "ADA: An Introduction", Springer-Verlag, USA, 1983.

[Agh 93] Agha, G., Wegner, P. e Yonezawa, A. “Research Directions in Concurrent Object-Oriented Programming”, The MIT Press, 1993.

[Agh 93] Agha, G., Frolund, S. *et alli*, “Abstraction and Modularity Mechanisms for Concurrent Computing”, IEEE Parallel and Distributed Technology, Systems and Applications, IEEE Computer Society, Maio, 1993.

[Agh 96] Agha, G., “Linguistic Paradigms for Programming Complex Distributed Systems”, ACM Computing Surveys, Vol. 28, No 2, pp. 295-296, Junho, 1996.

[Agh 98] Astley, M. e Agha, G., “Customizations and Composition of Distributed Objects: Middleware Abstractions for Policy Management”, ACM SIGSOFT'98 - 6th. International Syposium on Foundations of Software Engineering, Lake BuenaVista, Florida, pp. 1-9, Novembro, 1998.

[Aks 92] Aksit, M., Bergmans, L. e Vural, S., "An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach,"' Proceedings ECOOP '92, Ole Lehrmann Madsen (Ed.), LNCS 615, Springer, pp. 372-395,June 1992.

[Aks 96] Aksit, M. "Composition and Separation of Concerns in the Object-Oriented Model", ACM Computing Surveys, 28A (4), Dezembro, 1996.

[All 94] Allen, R. e Garlan, D., “Beyond Definition/Use: Architectural Interconnection”, ACM SIGPLAN Notices, IDL Workshop, Vol. 29, No. 8, pp. 35-44, Agosto, 1994.

[All 97T] Alllen, Robert J.; “A Formal Approach to Software Architecture”, Doctor of Philosophy Thesis, School of Computer Science, Cernegie Mellon University, USA, Maio, 1997.

[Ama 98] Amano, N e Watanabe, T., "LEAD++: An Object-Oriented Reflective Language for Dynamically Adaptable Software", OOPSLA'98, Workshop on Reflective Programming in C++ and Java (W13), Vancouver, Canada, Outubro, 1998

[And 97] Andreson, K. “Thoughts on AOP”, 1997. http://openmap.bbn.com/~kanderson/aop/AOP_thoughts.htm

[Arc 96] ____, "IEEE Software – Letters: Defining Design Terminology", IEEE Software, Vol. 13, No. 3, pp. 8-20, Maio, 1996.

[AspJ 98A] ____, "AspectJ™ Specification", Xerox, Paulo Alto Center, Março, 1998.

[AspJ 98B] ____, "AspectJ™: User's Guide and Primer ", Xerox, Paulo Alto Center, Março, 1998.

[AspJ 98C] ____, "AspectJ™: 0.2 Beta", Apresentação PowerPoint, http://www.parc.xeox.com/aop, Xerox, Paulo Alto Center, Março, 1998.

[Atk 91] Atkinson, C., "Object-Oriented Reuse, Concurrency and Distribution: an Ada-based approach", Addison-Wesley, 1991.

[Bac 87] Bach, M. J., “The Design of the UNIX Operation System”, Englewood Cliffs, NJ, Prentice-Hall, 1987.

[Ben 90] Ben-Ari, M. “Principles of Concurrent and Distributed Programming”, Englewood Cliffs, NJ, Prentice-Hall, 1990.

[Ber 96T] Bergmans, M. J. L., “Applying Compositional-Filters for the Development and Reuse of Concurrent Object-Oriented Programs”, PhD. Thesis, ISBN 90 9007359-0, Universiteit Twente, NL, Junho, 1996.

[Bir 84] Birrell, A. e Nelson, B. J., "Implementing Remote Procedure Calls", ACM Transactions on Computer Systems, Vol. 2, No. 1, pp. 39-59, Fevereiro, 1984.

[Bir 98] Birman, K. P., "Building Secure and Reliable Network Applications",. Manning Publications Co., USA, 1996.

[Bis 96] Bishop, J. e Faria, R, “Connectors in Configuration Programming Languages: are They Necessary?”, IEEE Third International Conference on Configurable Distributed Systems, Annapolis, Maryland, Maio, 1996.

[Bla 96] Blakowski, G. e Steinmetz, R., “A Media Synchronization Survey: Reference Model, Specification, and Case Studies, IEEE JSAC, Vol 14, No 1, pág. 5-35, janeiro de 1996.

[Boh 98] Bohrer, K., Johnson, V., Nilsson, A. e Rubin, B. "Business Process Components for Distributed Object Applications", Communications of the ACM, Vol. 41, No. 6 , pp. 43-48, Junho, 1998.

[Boo 97] Booch, G., Jacobson, I. e Rumbaugh, J., "Unified Modeling Language", Versão 1.0, Rational Software Corp., 1997.

[Bri 95] Brito, O. F. G., "Tolerância a Falhas no ambiente RIO", Dissertação de Mestrado, Departamento de Engenharia Elétrica - PUC/RJ, Março, 1995.

[Bro 96] Brown, A. W., e Wallnau, K. C., “Engineering of Component-Based Systems”, Component-Based Software Engineering: Selected Papers from the Software Engineering Institute, IEEE Press, 1996.

[Bro 98] Brown, N. e Kindel, C., "Distributed Component Object Model Protocol - DCOM/1.0", Microsoft Corp., http://www.microsoft.com/com, 1998.

[Buh 95] Buhr, P. A., Fortier, M., e Coffin, M. H, "Monitor Classification", ACM

Computing Surveys, Vol. 27, No. 1, pp. 63-108, Março de 1995.

[Bur 97T] Burggraaf, Arjan; “Solving modelling problems of CORBA unignng composition-filters”, M. Sc. Thesis, Dep. od Computer Science, Univeristy of Twente, Finlandia, Aug. 1997.

[Cam 96] Campo, M., e Price, N.H., “Meta-Object Manager: A framework for Customizable Meta-Objetc Support for Smaltalk 80”, I Simpósio Brasileiro de Linguagens de Programação, Belo Horizonte, MG, Setembro, 1996.

[Car 96] Carrera, E. V, e Loques, O G., "Integrando Memória Compartilhada, Troca de Mensagens e Configuracao",

[Chi 95] Chiba, S., “A Metaobject Protocol for C++”, OOPSLA 95, 10th Conference on Objetc-Oriented Programming Systems, Languages and Applications, SIGPLAN Notices, Vol. 30, No. 10, pp. 285-299, Austin, Texas, EUA, Outubro, 1995.

[Chi 96] Chiba, S., "Open C++ Programmer's Guide for Version 2", Xerox PARC, Technical Report SPL-96-024, Xerox PARC, Maio,1996.

[Chi 97] Chiba, S. *et alli*, “Weak Protection for Reflective Operating Systems”, ECOOP’97, Workshop on Reflective Real-time Object-Oriented Programming and Systems (W3), Jyvaskyla, Finland, Junho, 1997.

[Cra 95] Crane, S., Magee, J. e Pryce, N., "Design Patterns for Binding in Distributed Systems", Department of Computing, Imperial College, Setembro, 1995.

[Cra 96] Crane, S., "A Framework for Distributed Interaction", International Workshop on Development and Evolution of Software Architectures for Product Families, Madrid, Espanha, Novembro, 1996.

[Cra 97] Crane, S. and Dulay, N., “A Configurable Protocol Architecture for CORBA Environments”, ISADS'97 - International Symposium for Autonomous Decentralized Systems”, Berlin, Alemanha, Abril, 1997.

[Dar 97] _____, "The Darwin Language – Version 3c", Department of Computing, Imperial College of Science, Technology and Medicine, Março, 1997.

[DCE 93] _____, "DCE 1.2.2 Introduction to OSF DCE", Open Group Product Documentation, F201 ISBN 1-85912-182-9, 259 páginas, Novembro, 1997.

[Dea 95] Dean, T. R. e Cordy, J. R., "A Syntactic Theory of Software Architecture", IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 302-313, Abril, 1995.

[Der 97T] Deri, Luca; “A Component-based Architecture for Open, Independently Extensible Distribute Systems”, Doctoral Thesis, Universität Bern, Instituto de Informática, Junho, 1997.

[Duc 97] Ducasse, S. e Richner, T., "Executable Connectors: Towards Reusable Design Elements", Software Engineering Notes, Vol. 22, No. 6, pp. 60-76, ACM SIGSOFT, Novembro, 1997.

[Dud 98] Vogel, A. e Duddy, K., "JAVA Programming with CORBA", Second Edition, Wiley Computer Publishing, USA, 1998.

[End 92] Endler, M. e Wei. J., "Programming Generic dynamic reconfigurations for distributed applications", Int. Workshop on Configurable Distributed Systems, pp. 68-79, IEE, Março, 1992.

[EST 89] International Organization for Standardization, "ESTELLE: A Formal Description Technique Based on the Extended State Transition Model", Techical Report IS 9074, ISSO, 1989.

[Fab 95] Fabre, J.C.; Perennou, T. e Blain, L., “Meta-Object Protocols for Implementing Reliable and Secure Distributed Applications”, LAAS REPORT 95037, CNRS, February, 1995.

[Fab 98] Fabre, J.C.; Perennou, T. e Blain, L., “A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach”, IEEE Transactions on Computers, Vol. 47, No. 1, pp. 78-95, Janeiro, 1998.

[Fay 97] Fayad, M. E. e Schmidt, D. C., Guest Editors, "Object-Oriented Application Frameworks", Communications of the ACM, Vol. 40, No. 10, pp. 32-38, Outubro, 1997.

[Flo 96] Florissi, P. GS., e Yemini, Y. “Management of Application Quality of Service”, Distributed Computing and Communications Lab, Computer Science Dep. Columbia University, NY, 1996.

[Fra 96] Fraga, J., Farines, J., Furtado, O., Siqueira, F., “Programação de Aplicações Distribuídas Tempo-Real em Sistemas Abertos, XXIII Seminário Integrado de Software e Hardware, Recife, PE, Agosto, 1996.

[Fra 97] Fraga, J., Maziero, C., Lung, L.C., e Loques, O. G., “Implementing replicated services in open systems using a reflective approach”, Third International Symposium on Autonomous Decentralized Systems, Berlin, Germany, Abril, 1997.

[FraM 97] Franz, M. , "Dynamic Linking of Software Components", IEEE Computer, pp. 74-81, Março, 1997.

[Fro 98] Frolund, S. e Koistinen, J. "Quality-of-service specification in distributed object systems", Distributed Systems Engineering 5, The British Computer Society, pp. 179-202, 1998.

[Gam 94] Gamma, E. *et. alli*, “Design Patterns - Elements of Reusable Object-Oriented Software”, Addison-Wesley Publishing Co., USA, 1995.

[Gar 94] Garlan, D., Allen, R. e Ockerbloom, J., “Exploiting Style in Architectural Design Environments”, SIGSOFT'94: 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 175-188, New Orleans LA, USA, Dezembro, 1994.

[Gib 91] Gibbs S., “Composite Multimedia and Active Objects”, OOPSLA’91, SIGPLAN Notices, Vol. 26, Nº 11, pág. 97-112, novembro de 1991.

[Gol 97] Golm, M. e Kleinöder, J., “Implementing Real-Time Actors with MetaJava”, ECOOP’97 Workshop on Reflective Real-time Object-Oriented Programming and Systems (W3), Jyvaskyla, Finland, Junho, 1997.

[Gol 97T] Golm, Michael; “Design and Implementation of a Meta Architecture for Java”, M.Sc. Thesis, Instituto de Matemática, Erlanngen-Nüremberg University, Germany, Janeiro, 1997.

[Gol 98] Golm, M. e Kleinöder, J., "metaXa and the Future of Reflection", OOPSLA'98, Workshop on Reflective Programming in C++ and Java (W13), Vancouver, Canada, Outubro, 1998.

[Gow 95] Gowing, B. e Cahill, V., “Making Meta-Object Protocols Practical for Operating Systems”, Forth International Workshop on Object Orientation in Operating Systems, pp. 52-55, 1995.

[Gow 96] Gowing, B. e Cahill, V. “Meta-Object Protocols for C++: The Iguana Approach”, Reflection’96, San Francisco, USA, Abril, 1996.

[Gra 98] Gray, D. N, Hotchkiss, J., LaForge, S., Shalit, A. e Weinberg, T., "Modern Languages and Microsoft's Component Object Model", Communications of the ACM, Vol. 41, No. 5, pp. 55-65, Maio, 1998.

[Gue 96] Guerraqui, R. *et alli*, "Strategic Directions in Object-Oriented Programming", ACM Computing Surveys, Vol. 28, No 4, pp. 691-700, Dezembro, 1996.

[Har 97] Harrision, T. H., Levine, D. L., Schmidt, D. C. "The Design and Performance of a Real-time CORBA Event Service", OOPSLA'97, ACM SIGPLAN Conference On Object-Oriented Programming Systems, Languages and Applications, Atlanta, Georgia, USA, Outubro, 1997.

[Har 97] Harich, Jack. "The trick to using a basic Java 1.1 network and file class loader". JavaWorld, outubro de 1997. Documento eletrônico: http://www.javaworld.com/javaworld/javatips/jw-javatip39.html.

[Hel 90] Helm, R., Holland, I.M. e Gangopadhyay, D., "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", European Conference on Object-oriented Programming - ECOOP'90, pp. 169-180, Ottawa, Outubro, 1990.

[Her 96] Herbert, A., "CORBA Extensions for Real-Time and Multimedia", APM.1311.02, ANSA Phase III, 12 pages, 1996.

[Hoa 85] Hoare, C. A. R., "Communicating Sequencial Processes", Prentice-Hall International, first edition, 1985.

[Höl 93] Hölzle, U., "Integrating Independently-Developed Components in Object-Oriented Languages", ECOOP'93 - Object-Oriented Programming, 7th European Conference, Springer Verlag Lecture Notes on Computer Science, pp. 36-56, Kaiserslautern, Germany, Julho, 1993.

[Hol 97] Holmes, D., Noble, J., Potter, J., "Aspects os Synchronization", ECOOP'97, Workshop on Aspect-Oriented Programming (W15), Jyvaskyla, Finland, Junho, 1997.

[Hür 95] Hürsch, W. L e Lopes, C. V., "Separation of Concerns", 1995.

[Hut 91] Hutchinson, N.C, e Peterson, L.L, "The x-Kernel: AN Architecture for Implementing Network Protocols, IEEE Transactions on Software Engineering, Vol. 17, No. 1, Janeiro de 1991.

[Inv 95] Inverardi, P. e Wolf, A. L. , "Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Model", IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 373-386, Abril, 1995.

[IONA 92A] _____, "Orbix 2.0 Programming Guide", IONA Technologies Ltd., Outubro, 1992.

[IONA 92B] _____, "Orbix 2.0 Reference Guide", IONA Technologies Ltd., Outubro, 1992.

[Iss 96] Issarny, V. e Bidan, C., "Aster: A CORBA-based Software Interconnection System Supporting Distributed System Customization", 3rd. International Conference on Configurable Distributed Systems, pp. 586-593, Annapolis, USA, Maio, 1996.

[Iss 96] Issarny, V., Bidan, C., "Aster: A Framework for Sound Customization of Distributed Runtime Systems", 16th IEEE International Conference on Distributed Computing Systems, pp. 586-593, Hong-Kong, Maio 1996.

[Iss 98] Issarny, V., Bidan, C. e Saridakis, T., "Achieving Middleware Customization in a Configuration-Based Development Environment: Experience with Aster

Prototype", 4th International Conference on Configurable Distributed Systems, pp. 207-214, , Annapolis, Maryland, USA, Maio 1998.

[Jac 98] Jackson, M, e Zave, P., "Distributed Feature Composition: A Virtual Architecture for Telecommunications Services", IEEE Trasaction on Software Engineering, Vol. 24, No. 10, pp. 831-847, Outubro, 1998.

[Jag 92] Jagannathan, S. e Agha, G., “A Reflective Model of Inheritance”, European Conference on Obbject-Oriented Programming, Utrecht, The Nederlands, pp. 350-371, Junho, 1992.

[Java A] Arnold, K. e Gosling, J., "The Java Programming Language", Second Edition, Sun Microsystems, Addison Wesley, USA, 1998.

[JavaCC 97] _____, "Description of the JavaCC Grammar File", Sun Microsystems, 1997.

[Joh 97] Johnson, R. E., "Frameworks = (Components + Patterns)", Communications of the ACM, Vol. 40, No. 10, pp. 39-42, Outubro, 1997.

[Jon 93] Jones, M. B., “Interposition Agents: Transparently Interposing User Code at the System Interface”, 14th ACM Symposium on Operating Systems Principles, Asheville, EUA, 1993.

[Kic 96] Kiczales, G. e Irwing, J. *et alli*, “Aspect-Oriented Programming”, Position Paper for the ACM Workshop on Strategic Directions in Computing Research, MIT, Junho, 1996. (http://www.parc.xerox.com/spl/projects/aop)

[Kic 97] Kiczales, G., Lamping, J., Lopes, C. V., Maeda, C., Mendhekar, A., "Open Implementations Design Guidelines", 19th International Conference on Software Engineering, ACM Press, 1997.

[Kic 97] Kiczales, G., Rivières, J. e Bobrow, D.G., “The Art of the Metaobject Protocol”, MIT Press, 1991.

[Kim 95] Kim, W.Y. e Agha, G., “Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages”, Supercomputing’95, San Diego, USA, Dezembro, 1995.

[Kir 98] Krieger, D. e Adler, R. M., "The Emergence of Distributed Component Platforms", IEEE Computer, pp. 43-53, Março, 1998.

[Kle 96] Kleinöder, J. e Golm, M., “MetaJava: na Efficient Run-Time Meta Architecture for Java”, International Workshop on Object Orientation in Operating Systems, IEEE, Seattle, Washington, Outubro, 1996.

[Kon 98] Kon, F., Campbell, R. H. *et alli*, "2K: A Reflective, Component-Based Operation System for Rapidly Changing Environments", 1998.

[Kra 90] Kramer, J., “Configuration Programming – A Framework for Development of Distributed Systems”, IEEE International Conference on Computer Systems and Software Engineering, Tel Aviv, Israel, Maio, 1990.

[Lad 97] Laddaga, R. e Veitch, J., “Dynamic Object Technology”, Communications of the ACM, Vol. 40, No. 5, pp. 37-38, Maio, 1997.

[Lea 93] Lea, R., Jacquemot, C. e Pillevesse, E., "COOL: System Support for Distributed Programming", Communications of the ACM, Vol. 36, No. 9, pp. 37-47, Setembro, 1993.

[Lee 95] Lee, A. H. e Zachary, J. L., “Reflections on Metaprogramming”, IEEE Transactions on Software Engineering, Vol. 21, No. 11, pp. 883-893, Novembro, 1995.

[Lew 98] Lewandowski, S. M., "Frameworks for Component-Based Client/Server Computing", ACM Computing Surveys, Vol. 30, No. 1, pp. 5-27, Março, 1998.

[Lis 95] Lisbôa, M. L., “Reflexão Computacional no Modelo de Orientação a Objetos”, 25º Jornadas Argentinas de Informatica y Investigacion Operativa, Buenos Aires, Argentina, Setembro, 1995.

[Lob 98] Lobosco, M., Loques, O. G. e Sztajnberg, A. "Configuração de Sistemas Distribuídos baseada em Java", I Simpósio de Sistemas de Simulação e Controle, 41-48, IPqM, Rio de Janeiro, Setembro, 1998.

[Lob 99] Lobosco, M., "R-RIO: Um Ambiente para Suporte à Construção e Evolução de Sistemas", dissertação de mestrado, CAA/UFF, Março, 1999.

[Löh 93] Löhr, K-P., "Concurrency Annotations for Reusable Software", Communications of the ACM, Vol. 36, No. 9, pp. 81-89, Setembro, 1993.

[Lop 97] Lopes, C. V. e Kiczales, G., “D: A Language Framework for Distributed Programming”, Xerox Palo Alto Research Center, 1997, (http://www.parc.xerox.com/spl/projects/aop)

[Lop 97T] Lopes, Cristina I. V.; “D: A Language Framework for Distributed Programming”, Doctor of Philosophy Thesis, College of Computer Science, Northeastern University, USA, Novembro, 1997.

[Loq 96] Loques, O., Leite, J., Botafogo, R. and Lobosco, M., “Configuração, Reflexão e CORBA: Algumas Considerações”, III Workshop do Projeto ASAP, UFCE, Fortaleza, Ceará, 1996.

[Loq 97] Loques, O., Botafogo, R. e Leite, J., “A Configuration Approach for Distributed Object-Oriented System Customization”, IEEE Third International Workshop on Object-oriented Real-time Dependable Systems, Newport Beach, California, Fevereiro, 1997.

[Loq 98] Loques, O. G., Leite, J. e Carrera, E. V, "P-RIO: A Modular Parallel-Programming Environment", IEEE Concurrency, Vol.6, No. 1, pp. 47-57, Janeiro-Março, 1998.

[Loq 99] Loques, O. G., Leite, J., Sztajnberg, A., Lobosco, M. "Towards Integrating Meta-Level Programming and Configuration Programming", submetido para Conference on Reflection, 1999.

[LOT 89] ISSO IS 8807, "LOTOS - a Formal Description Techinique based on temporal ordering of observational behavior", Techical Report, ISSO, 1989.

[Luc 95] Luckham, D.C., *et alli*, “Specification and Analysis of System Architecture Using Rapide”, IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 336-355, Abril, 1995.

[Luc 96] Lucena, C. J. P. e Alencar, P. S. C., "A Formal Description of Evolving Software Systems Architectures", Anais do I Prêmio COMPAC/ Uniemp, pp. 3-16, São Paulo, 1996.

[Lun 98] Lunau, C. P., "Is Composition of Metaobjects = Aspect Oriented Programming, ECOOP'98, Workshop on Aspect-Oriented Programming (W15), Bruchelas, Belgica, Julho, 1998.

[Maf 95] Maffeis, S. “Adding group communication and fault-tolerance to CORBA”, Proceedings of the 1995 USENIX Conference on Object Oriented Technologies, Monterey, CA, EUA, Junho, 1995.

[Maf 96] Maffeis, S., “Electra 2.0b Programer’s Manual”, Dept. of Computer Science,

Cornell University, February, 1996.

[Mag 92] Magee, J., Dulay, N. e Kramer, J., "Structuring parallel and distributed programs", Int. Workshop on Configurable Distributed Systems, IEE, pp. 102-117, Março, 1992.

[Mag 94] Magee, J., Dulay, N. e Kramer, J., "Regis: A Constructive Development Environment for Parallel and Distributed Programs", IEE/IOP/BCS Distributed Systems Journal, 1(5), PP. 304-312, Setembro, 1994.

[Mag 96] Magee, J., Kramer, J., "Dynamic Structure in Software Architectures", ACM SIGSOFT'96, Fourth Symposium on the Foundations of Software Engineering, pp. 3-14, San-Francisco, California, EUA, Outubro, 1996.

[Mag 97] Magee, J. e Kramer, J., "Composing Distributed Objects in CORBA", ISADS'97, Third International Symposium on Autonomous Decentralized Systems, Berlin, Germany, Abril, 1997.

[Mal 93] Malucelli, V. V, "RIO – Linguagem de Configuração – Tolerância à Falhas", Nota Interna, DEE / PUC-RJ, Fevereiro, 1993.

[Mal 96] Malucelli, V. V, "Babel - Contruindo Aplicações por Evolução ", dissertação de mestrado, DEE / PUC-RJ, Fevereiro, 1996.

[Mat 93] Matsuoka, Satoshi, "Analysis of inheritance anomaly in object-oriented concurrent programming languages", Research Directions in Concurrent Object-Oriented Programming, The MIT Press, pp. 107-150, 1993.

[Mat 97] Matthijs, F., *et alli*, "Aspects should not die", ECOOP'97, Workshop on Aspect-Oriented Programming (W15), Jyvaskyla, Finland, Junho, 1997.

[McC 98] McCann, P. J. e Roman, G. C., "Compositional Programming Abstractions for Mobile Computing", IEEE Transactions on Software Engineering, Vol. 24, No. 3, pp. 97-111, Fevereiro, 1998.

[Med 97] Medvidovic, N. e Taylor, R. N., "A Framework for Classifying and Comparing Architecture Description Languages", Software Engineering Notes, SIGSOFT, ACM, pp. 60-75, Vol. 22, No. 6, Novembro, 1997.

[Men 97] Mens, K., Lopes, C., Tekinerdogan, B. e Kiczales, G., "Aspect-Oriented Programming Workshop Report", ECOOP'97, Workshop on Aspect-Oriented Programming (W15), Jyvaskyla, Finland, Junho, 1997.

[Mey 93] Meyer, B., "Systematic Concurrent Object-Oriented Programming", Communications of the ACM, Vol. 36, No. 9, pp. 56-80, Setembro, 1993.

[Mey 98] Meyer, B. "Tell Less, Say More: The Power of Implicitness", IEEE Computer, pp. 97-98, Julho, 1998.

[Mik 97] Mikhajlov, Leonid e Sekerinski, Emil; "The Fragile Base Class Problem and Its Solution", TUCS Technical Report No 117, Turku Center of Computer Science, University of Turku, Finlandia, Junho, 1997.

[Mil 89] Milner, R., "Communication and Concurrency", Prentice-Hall International, first edition, 1989.

[Mit 97] Mitchell, S. E., *et alli*, "Adaptive Scheduling using Reflection", ECOOP'97 Workshop on Reflective Real-time Object-Oriented Programming and Systems (W3), Jyvaskyla, Finland, Junho, 1997.

[Mon 97] Monroe, R. T., Kompanek, A., Melton, R. e Garlan, D., "Architectural Styles, Design Patterns and Objects", IEEE Software, Vol. 14, No. 1, pp. 43-52, Janeiro

de 1997.

[Mor 95] Moriconi, M., Qian, X. e Riemenschneider, R.A, “Correct Architecture Refinement”, IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 356-372, Abril, 1995.

[Nau 97] Naumovich, G. *et alli*, "Applying Static Analysis to Software Architecture", Software Engineering Notes, SIGSOFT, ACM, Vol. 22, No. 6, pp. 77-93, Novembro, 1997.

[Neb 98] Nebbe, R. D., "Coordination and Composition: The Two Paradigms Underlying AOP ?", ECOOP'98, Workshop on Aspect-Oriented Programming (W15), Bruchelas, Belgica, Julho, 1998.

[Oli 98] Oliva, A., e Buzato, L. E., "Composition of Meta-Objects in Guaraná", OOPSLA'98, Workshop on Reflective Programming in C++ and Java (W13), Vancouver, Canada, Outubro, 1998.

[OMG 96A] OMG, “The Common Object Request Brocker: Architecture and Specification”, Revision 2.0, Julho, 1996.

[OMG 96B] OMG RPF Revised Joint Submission, “Joint Meta Object Facility, Revision 1.0”, OMG CFTF, RPF-5, Janeiro, 1997.

[OMG 97] OMG RPF Revised Joint Submission, “Multiple Interfaces and Composition, OMG TC Document orbos/97-05-17, Version 4.2.7, Junho, 1997.

[OOP’90] ___, “Designing Reusable Designs: Experiences Designing Object-Oriented Frameworks”, OOPSLA/ECOOP’90, Addendum to the Proceedings, pp. 19-24, Ottawa, Canada, Outubro, 1990.

[OpD 96] _____, "OpenDoc Shaping Tomorrow's Software – White Paper", Apple Computer, Inc., 1996.

[Pan 95] Pancake, C. M., "The Promisse and Cost of Object Technology: A Five-Year Forecast", Communications of the ACM, Vol. 38, No. 10, Outubro, 1995.

[Pou 94] Pountain, D. and Szyperski, C., “Extensible Software Systems”, Byte Magazine, pp. 57-62, Maio, 1994.

[Pow 91] Powell, M.L. *et alli*, "SunOS 5.0 Multithread Architecture", Procedings of Usenix Winter Technical Conference, pp. 65-79, Dalas, TX, 1991.

[Pru 92] Pruy, R., "Cooperative Concurrency Control", TR-14-18-92, Univ. Erlangen-Nürnberg, Alemanha, Novembro, 1992.

[Pry 96] Pryce, N. e Crane, S., ”A Uniform Approach to Configuration and Communication on Distributed Systems”, 3rd International Conference on Configurable Distributed Systems, IEEE Press, Annapolis, USA, Maio, 1996.

[Pry 98] Pryce, N., "Component Interaction in Distributed Systems", 4th International Conference on Configurable Distributed Systems, Annapolis, Maryland, USA, Maio, 1998.

[Pur 98] Purao, S., Jain, H. e Nazareth, D. "Effective Distribution of Object-Oriented Applications", Communications of the ACM, Vol. 41, No. 8, pp. 100-108, Agosto, 1998.

[Rav 98] Raverdy, P-G., Gong, H. L. V. e Lea, R., "DART: A Reflective Middleware for Adaptive Applications", OOPSLA'98, Workshop on Reflective Programming in C++ and Java (W13), Vancouver, Canada, Outubro, 1998.

[Reg 95] Crane, S., Magee, J., Dulay, N. e Twidle, K., "Regis Programmer's Manual", The

Regis Implementation Group, Release 0.6.1, Imperial College of London, Agosto de 1995.

[Rin 92] Rine, D. C. e Bhargava, B., "Guest Editors' Introduction: Object-Oriented Computing", Computer, IEEE, pp. 6-10, Outubro, 1992.

[Rin 98] Rine, D. C., "Supporting Reuse with Object Technology", IEEE Computer, Vol. 30, No. 10, pp. 43-45, Outubro, 1997.

[ROI 97] Crane, S., "ROI - a Regis ORB Implementation, Abril, 1997. http://www.cs.ic.ac.uk/roi (URL)

[Rov 98] Rover, D. T., Waheed, A. *et alli*, "Software Tools for Complex Distributed Systems: Toward Integrated Tool Environments", IEEE Concurrency, Vol. 6, No. 2, pp 38-54, Abril-Junho, 1998.

[Rub 94] Rubira, C. M. F., "Structuring Fault-Tolerant OO Systems Using Inheritance and Delegation", Ph.D. Thesis, Dep. of Computer Science, University of Newcastle, Outubro, 1994.

[Rub 97] Rubira, C. M. F., Corrêa, S. L. e Buzato, L. E., "Um Framework Orientado a Objetos Reflexivo para a Construção de Software Tolerante a Falhas", VII Simpósio de Computadores Tolerantes a Falhas, pp. 75-89, Campina Grande, Paraíba, Julho, 1997.

[Rum 91] Rumbaugh, J., *et alli*., "Object-Oriented Modeling and Design", Prentice-Hall Inc., 1991.

[Sal 97] Salvador, L. N. e Sato, L. M., "Anomalia da Herança em Programação Concorrente Orientada a Objetos: Uma Proposta de Tratamento", Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho - SBAC-PAD'97, pp.431-446, 1997.

[Sch 96] Jean-Guy Schneider, "Sina and the Composition Filters Object Model", http://www.iam.unibe.ch/~scg/Research/ComponentModels/sina.html, Setembro, 1996.

[SDL 92] CCITT Com X-R 17-E, "Rec Z100 Functional Specification and Description Language (SDL)", Techical Report, CCITT, 1992.

[Sha 95] Shaw, M., DeLine, R., Klein, D. e Zelesnik, G., "Abstractions for software architecture and tools to support them", IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 314-335, Abril, 1995.

[Sha 96] Shaw, M., DeLine, .R. e Zelesnik, G. "Abstractions and Implementations for Architectural Connections", 3rd Intl. Conference on Configurable Distributed Systems (ICCDS'96), 1996.

[Sin 97] Singhai, A., Sane, A. e Campbell, R., "Reflective ORBs: Supporting Robust, Time-critical Distribution", ECOOP'97, Workshop on Reflective Real-time Object-Oriented Programming and Systems (W3), Jyvaskyla, Finland, Junho, 1997.

[Siq 98] Siqueira, F. "A Framework for Distributed Multimedia Applications based on CORBA and Integrated Services Networks", Distributed Systems Group, Department of Computer Science, Trinity College, Dublin, 1998.

[Ste 90] Stevens, W. R., "Unix Network Programming", Prentice-Hall, Englewood Cliffs, NJ, 1993.

[Str 93] Strouds, R., "Transparency and Reflection in Distributed Systems", ACM Operating Systems Review, Vol. 22, No. 2, Abril, 1993.

[Str 93] Stroustrup, B e Ellis, M. A., "The Annotated C++ Reference Manual", Addison-Wesley, USA, 1993.

[Szt 93] Sztajnberg, A. e Loques, O. G., "O Sistema de Comunicação Multi-Protocolo do Ambiente RIO", V Simpósio de Computadores Tolerante a Falhas, São José dos Campos, pp. 1-17, Outubro, 1993.

[Szt 94] Sztajnberg, A, Loques, O. G. e Leite, J., "Implementação e Teste do Sistema de Comunicação do Ambiente RIO", XII Simpósio Brasileiro de Redes de Computadores, pp. 53-62, Curitiba , Maio, 1994.

[Szt 95] Sztajnberg, A e Loques, O. G., "Ambiente RIO: Estado do Projeto", XXII Seminário Integrado de Software e Hardware, pp. 1089-1100, XV Congresso da SBC, Canela, RS, Agosto, 1995.

[Szt 97] Sztajnberg, A. e Loques, O. G., "Reflexão Computacional e Multimídia Distribuida", XV Simpósio Brasileiro de Redes de Computadores, pp. 417-431, São Carlos, SP, Maio, 1997.

[Szt 98A] Sztajnberg, A., Lobosco, M., Loques, O. G., "Uma Plataforma para Suporte à Evolução de Sistemas Distribuídos", II Workshop em Sistemas Distribuídos, pp.25-32, Curitiba, Maio, 1998.

[Szt 98B] Sztajnberg, A., Lobosco, M., Loques, O. G., "Separação de Interesses em Sistemas Distribuídos", I Simpósio de Sistemas de Simulação e Controle, pp.33-40, IPqM, Rio de Janeiro, Setembro, 1998.

[Szt 99] Sztajnberg, A., Lobosco, M., Loques, O. G., " A abordagem R-RIO para concepção de aplicações distribuídas", Simpósio Brasileiro de Redes de Computadores - SBRC'99, Salvador , Maio, 1999.

[Tai 96] Taivalsaari, A., "On the Notion of Inheritance", ACM Computing Surveys, Vol. 28, No. 3, pp. 438-479, Setembro, 1996.

[Thr 90] _____, "Threads Extension for Portable Operating Systems", P1003.4ª, Draft 4, Technical Committee on Operating Systems of the IEEE C S Press, Los Alamitos, CA, 1990.

[Tic 97T] Tichelaar, Sander; "A Coordination Component Framework for Open Distributed Systems", M.Sc Thesis, Faculty of Mathematics and Natural Science, University of Groningen, Suiça, Maio, 1997.

[Tur 96] Turletti, T. e Huitema, C., "Videoconferencing on the Internet", IEEE/ACM Transactions on Networking, Vol. 4, Num 3, pp. 640-350, junho de 1996.

[URL A] Berners-Lee, T., Masinter, L., e McCahill, M. " Uniform Resource Locators (URL)", Request for Comments: 1738, Network Working Group, Category: Standards Track, Dezembro, 1994.

[Voa 98] Voas, J. M., "The Challenges of Using COTS Software in Component-Based Development", IEEE Computer, Vol. 31, No. 8, pp. 44-45, Junho, 1998.

[Wan 98] Wang, Y-M. e Chung, P-Y. E., "Customization of distributed systems using COM", IEEE Concurrency, pp. 8-12, Julho-Setembro, 1998.

[Way 96] Wayner, P. "Better Java Programming", Byte Magazine, pp. 63-64, Setembro, 1996.

[Wec 98] Weck, W., "Inheritance Using Contracts & Object Composition", ECOOP '97 Second International Workshop of Component-Oriented Programming (WCOP '97), Springer-Verlag Berlin, LNCS 1357, ISBN 3-540-64039-8, pp. 384-388, Jyväskylä, Finland, Junho, 1998.

[Weg 92] Wegner, P. "Dimensions of Object-Oriented Modeling", IEEE Computer, pp. 12-20, Outubro, 1992.

[Weg 96] Wegner, P., “Interoperability”, ACM Computing Surveys, Vol. 28, No. 1, pp. 285-287, Março, 1996.

[Wel 98] Welch, Ian; Stroud, Robert. "Dalang - A Reflective Java Extension", OOPSLA'98, Workshop on Reflective Programming in C++ and Java (W13), Vancouver, Canada, Outubro, 1998

[Wer 95] Werner, J. A. V., "Metodologia e Suporte para Sistemas Distribuídos Configuráveis", dissertação de mestrado, DEE / PUC-RJ, Março, 1995.

[Wer 98] Wermelinger, M. e Fiadero, J. L., "Connectors for Mobile Programs", IEEE Transactions on Software Engineering, Vol. 24, No. 5, pp. 331-342, Maio, 1998.

[Wu 98] Wu, Zhixue, "Reflective Java and A Reflective Component-Based Transaction Architecture", OOPSLA'98, Workshop on Reflective Programming in C++ and Java (W13), Vancouver, Canada, Outubro, 1998.

[Yan 96] Yang, Z e Duddy, K., “CORBA: A Platform for Distributed Object Computing”, ACM Operation Systems Review, Vol. 30, No. 2. pp. 3-41, Abril, 1996.

[Zav 97] Zave, Pamela, "Classification of Research Efforts in Requirements Engineering", ACM Computing Surveys, Vol. 29, No. 4, pp. 315-321, Dezembro, 1997.

[Zel 97] Zelesnik, G., "Adding Support for Connector Abstractions in the UniCon Compiler", http://www.cs.cmu.edu/afs/cs/project/vit/www/unicon/adding-connectors/expert-creation.html, Department of Computing, Carnegie Mellon University, 1997.

[Zim 96] Zimmermann, C., Cahill, V. “It’s Your Choice - On the Design and Implementation of a Flexible Metalevel Architecture”, 3rd International Conference on Configurable Distributed Systems, IEEE Press, Maryland, EUA, Maio, 1996.