FLEXIBILIDADE E SEPARAÇÃO DE INTERESSES PARA CONCEPÇÃO E EVOLUÇÃO DE SISTEMAS DISTRIBUÍDOS

Alexandre Sztajnberg

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE 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.

____
Prof. Orlando Gomes Loques Filho, PhD.

____
Prof. Otto Carlos Muniz Bandeira Duarte, Dr. Ing.

____
Prof. Julius Cesar Barreto Leite, PhD.

____
Prof. Aloysio de Castro Pinto Pedroza, Dr. Ing.

____
Prof. Joni da Silva Fraga, Dr. Ing.

RIO DE JANEIRO, RJ – BRASIL

MAIO DE 2002

SZTAJNBERG, ALEXANDRE

Flexibilidade e Separação de Interesses para Concepção e Evolução de Sistemas Distribuídos [Rio de Janeiro] 2002

XVI, 304 p. 29,7 cm (COPPE/UFRJ, D.Sc., Engenharia Elétrica, 2002)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Arquiteturas de Software
2. Programação por Configuração
3. Programação de Meta-Nível
4. Sistemas Distribuídos

I. COPPE/UFRJ II. Título (série)

à minha esposa Célia
à minha filha Ana Carina

aos meus pais

aos meus avós (*in memoriam*)

Agradecimentos

Minha jornada não teria sido possível sem a ajuda de muitas pessoas.

Meus agradecimentos especiais à equipe da secretaria da COPPE Elétrica. Sua diligência facilitou minha vida acadêmica. Solange, muito obrigado pela “consultoria”.

Agradeço ao SR/2, DINFO e IME da UERJ pelo apoio financeiro para apresentar trabalhos (SBRC / SBES), pela liberação, em tempo parcial - para obter os créditos do Doutorado, e a liberação integral por 6 meses, no contexto dos programas Procase e Procad - para o Doutorado Sandwich nos EUA.

Obrigado ao IC / UFF pela acolhida em suas instalações e por ceder algumas horas do prof. Orlando Loques, um dos meus orientadores.

Agradeço à CAPES, pelo suporte financeiro na minha empreitada de 6 meses em um Doutorado Sandwich, na Universidade de Illinois em Urbana-Champaign, EUA.

Ao prof. Orlando Loques pela amizade, incentivo, apoio, paciência infinita, orientação segura e por tornar minhas idéias em uma tese, meu muito obrigado.

Ao prof. Jorge Leão, por me receber na COPPE, pelos ensinamentos, discussões frutíferas, orientação e apoio, muito obrigado.

Aos professores Otto, Aloysio e Resende do GTA, ao prof. Julius do IC/UFF, obrigado pelo apoio e competência. Obrigado também pela oportunidade de participar do comitê organizador do SBRC’98 e SBMIDIA’98 (foi uma ótima experiência).

À prof. Stella do IC/UFF, pelo apoio para a apresentação no OOPSLA'00.

Amigos do GTA (COPPE), da DINFO e IME (UERJ), e do IC (UFF), o meu muito obrigado pelo apoio.

Ao grupo de pesquisa do prof. Roy Campbell da UIUC, que tornou minha estada mais produtiva, *thank you*. Agradecimentos especiais aos professores Luiz Cláudio Schara (UFF) e Fábio Kon (USP), que na época eram integrantes deste grupo.

Obrigado a prof. Sarah Sztajnberg (minha tia) pela revisão competente, mesmo no curto espaço de tempo.

Minha esposa Célia e filha Ana Carina, meus pais e irmã, muito obrigado pelo apoio, carinho, paciência e suporte, que me fizeram ir sempre em frente.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

FLEXIBILIDADE E SEPARAÇÃO DE INTERESSES PARA CONCEPÇÃO E EVOLUÇÃO DE SISTEMAS DISTRIBUÍDOS

Alexandre Sztajnberg

Maio / 2002

Orientadores: Jorge Lopes de Souza Leão
Orlando Gomes Loques

Programa: Engenharia Elétrica

Nesta tese apresentamos o *framework* R-RIO, para a descrição, configuração, execução e gerência de aplicações de software. O *framework* R-RIO integra explicitamente as tecnologias de Programação de Meta-Nível e Arquiteturas de Software / Programação por Configuração. Como resultado desta integração, as atividades de descrição e configuração de aplicações em R-RIO, e as aplicações resultantes, gozam das características de reusabilidade, separação de interesses, evolução dinâmica e abrangência. Verifica-se ainda, como benefícios desta integração, a flexibilidade para a programação de aspectos não-funcionais, inerentes ao uso da Programação em Meta-Nível, e a capacidade de composição e verificação de propriedades das Arquiteturas de Software / Programação por Configuração, durante a programação das aplicações. Para validar o *framework* foram desenvolvidas algumas aplicações, e estas foram executadas sobre um protótipo experimental de R-RIO. Os resultados obtidos comprovaram de forma prática as vantagens do *framework*.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

FLEXIBILITY AND SEPARATION OF CONCERNS ON THE DESIGN AND EVOLUTION OF DISTRIBUTED SYSTEMS

Alexandre Sztajnberg

May / 2002

Advisors: Jorge Lopes de Souza Leão
Orlando Gomes Loques

Department: Electric Engineering

In this thesis we present the R-RIO framework, to describe, configure, run and manage software applications. The R-RIO framework integrates Meta-Level Programming and Software Architecture / Configuration Programming technologies. As result of this integration, application description and configuration activities in R-RIO, and the resulting applications, have characteristics such as reusability, separation of concerns, dynamic evolution and large domain applicability. Additionally, due to this integration, R-RIO benefits from the flexibility of non-functional aspects programming, inherent to the use of Meta-Level Programming, and the capability of composition and property verification of the Software Architecture / Configuration Programming technologies, during the application programming. To validate the framework, example applications were developed, and run over an experimental R-RIO prototype. The achieved results prove in a practical way the advantages of the framework.

Sumário

SUMÁRIO IX

LISTA DE FIGURAS XIV

LISTA DE ACRÔNIMOS XVII

CAPÍTULO I - INTRODUÇÃO 19

1.1 INTRODUÇÃO 19
1.2 OBJETIVOS 21
1.3 O *FRAMEWORK* R-RIO 22
1.4 VALIDAÇÃO 24
1.5 CONTRIBUIÇÕES 24
1.6 ESTRUTURA DO TEXTO 25

CAPÍTULO II - CONTEXTO TECNOLÓGICO 29

2.1 INTRODUÇÃO 29
2.2 REQUISITOS BÁSICOS 33
2.3 TECNOLOGIAS E ABORDAGENS PARA O DESENVOLVIMENTO DE APLICAÇÕES 35
2.3.1 Objetos 36
2.3.2 Sistemas Baseados em Componentes e Middleware 41
2.3.3 Programação de Meta-Nível 45
2.3.4 Arquiteturas de Software e Programação por Configuração 52
2.4 COMPARANDO AS TECNOLOGIAS 58
2.5 COMBINANDO AS TECNOLOGIAS 64
2.6 CONSTRUINDO O CONJUNTO 66
2.7 TRABALHOS CORRELATOS 68
2.8 CONCLUSÃO 68

CAPÍTULO III - TRABALHOS CORRELATOS 69

3.1 INTRODUÇÃO 69
3.2 PROGRAMAÇÃO DE META-NÍVEL 69

3.3 SISTEMAS BASEADOS EM COMPONENTES E *MIDDLEWARE* 87

3.4 ARQUITETURAS DE *SOFTWARE* / PROGRAMAÇÃO POR CONFIGURAÇÃO E ADL 93

3.5 UML 98

3.6 CONCLUSÃO 100

CAPÍTULO IV - O *FRAMEWORK* R-RIO 103

4.1 INTRODUÇÃO 103

4.2 MODELO DE COMPONENTES 105

4.2.1 Módulos 105

4.2.2 Portas 108

4.2.3 Conectores 111

4.2.4 Composição de Módulos 115

4.2.5 Composição de Conectores 116

4.3 MODELO DE GERÊNCIA DE CONFIGURAÇÃO 119

4.4 COMBINANDO REFLEXÃO E CONFIGURAÇÃO PARA A ADAPTAÇÃO DE ARQUITETURAS DE *SOFTWARE* 120

4.5 METODOLOGIA PARA A CONFIGURAÇÃO DE ARQUITETURAS DE *SOFTWARE* 124

4.6 CONSIDERAÇÕES SOBRE A INTEGRAÇÃO DE PROGRAMAÇÃO EM META-NÍVEL E ARQUITETURA DE *SOFTWARE* / PROGRAMAÇÃO POR CONFIGURAÇÃO EM R-RIO 127

4.7 EXEMPLO 130

4.8 CONCLUSÃO 133

CAPÍTULO V - ASPECTOS NÃO-FUNCIONAIS 135

5.1 INTRODUÇÃO 135

5.2 ENCAPSULANDO ASPECTOS NÃO-FUNCIONAIS EM CONECTORES 136

5.3 ASPECTOS NÃO-FUNCIONAIS EM R-RIO 137

5.3.1 Aspectos de Interação 138

5.3.2 Aspectos de Distribuição 139

5.3.3 Aspectos de Coordenação 140

5.4 ESPECIFICANDO ASPECTOS NÃO-FUNCIONAIS ATRAVÉS DE CONTRATOS 149

5.5 ASPECTOS DINÂMICOS EM R-RIO 152

5.6 EXEMPLO 154

5.7 CONCLUSÃO 162

CAPÍTULO VI - ASPECTOS DE QUALIDADE DE SERVIÇO 165

6.1 INTRODUÇÃO 165
6.2 QOS EM R-RIO 166
6.2.1 Visão Geral 167
6.2.2 Contratos de QoS em CBabel 169
6.2.3 Configurador de QoS 174
6.2.4 Monitor de QoS 176
6.2.5 Conectores de QoS 177
6.2.6 Avaliação 178
6.3 TRABALHOS CORRELATOS 179
6.4 CONCLUSÃO 183

CAPÍTULO VII - IMPLEMENTAÇÃO 185

7.1 INTRODUÇÃO 185
7.2 O *MIDDLEWARE* PARA O SERVIÇO DE GERÊNCIA DE CONFIGURAÇÃO 185
7.3 MAPEAMENTO 187
7.4 API DE CONFIGURAÇÃO 195
7.5 API DE REFLEXÃO ARQUITETURAL 197
7.6 CONCLUSÃO 200

CAPÍTULO VIII - VALIDAÇÃO 203

8.1 INTRODUÇÃO 203
8.2 CEIA DOS FILÓSOFOS 203
8.3 CHÃO DE FÁBRICA 206
8.4 TICTACTOE 212
8.5 TELEFONE 3 EM 1 219
8.6 CONCLUSÃO 229

CAPÍTULO IX - CONCLUSÃO 231

9.1 INTRODUÇÃO 231
9.2 AVALIAÇÃO DO *FRAMEWORK* R-RIO 231
9.2.1 Reusabilidade 231
9.2.2 Separação de Interesses 232
9.2.3 Evolução Dinâmica 234

9.2.4 Abrangência *234*
9.3 TÓPICOS DE DISCUSSÃO 236
9.3.1 Composição de Interesses *236*
9.3.2 Arquiteturas de Software e Design Patterns *238*
9.3.3 R-RIO como middleware reflexivo *239*
9.3.4 Questões de Desempenho *240*
9.4 PONTOS EM ABERTO E PERSPECTIVAS PARA O FUTURO 241
9.4.1 Reconfigurações seguras *242*
9.4.2 Otimização *242*
9.4.3 Software Architecture Description Loader *243*
9.4.4 Integração com outros ambientes *245*
9.4.5 Integração de ferramentas para verificação *246*
9.5 PRINCIPAIS CONTRIBUIÇÕES DESTA TESE 247
9.6 CONCLUSÃO 248

REFERÊNCIAS BIBLIOGRÁFICAS 251

APÊNDICE A - CBABEL: UMA ADL PARA R-RIO 269

A.1 INTRODUÇÃO 269
A.2 CBABEL - *BUILDING APPLICATIONS BY EVOLUTION WITH CONNECTORS* 269
A.2.1 Referências *270*
A.2.2 Portas *271*
A.2.3 Módulos *272*
A.2.4 Conectores *274*
A.2.5 Mapeamento de componentes para implementação *277*
A.3 CONFIGURAÇÃO DE APLICAÇÃO 280
A.3.1 Instâncias de módulo (instantiate) *280*
A.3.2 Interconexão de módulos e conectores (link) *280*
A.3.3 Visibilidade das portas (export) *283*
A.3.4 Iniciando a execução de um módulo (start) *283*
A.3.5 Bloqueando um módulo (block) *283*
A.3.6 Retomando a execução de um módulo (resume) *284*
A.3.7 Removendo um módulo da arquitetura (remove) *284*
A.3.8 Grupos de módulos *284*

A.4 CONTRATOS PARA ASPECTOS NÃO-FUNCIONAIS 284
A.4.1 Contratos de Interação *285*
A.4.2 Contratos de Distribuição *286*
A.4.3 Contratos de Coordenação *287*
A.5 COMPOSIÇÃO DE MÓDULOS 291
A.6 COMPOSIÇÃO DE CONECTORES 292
A.6.1 Notação compacta *293*
Observação *294*
A.7 HERANÇA 295
A.8 BNF 296
A.9 CONCLUSÃO 300
APÊNDICE B - ASPECTOS FORMAIS 303
B.1 INTRODUÇÃO 303
B.2 MODELO DE BASE 303
B.2.1 Sistema de Transição *304*
B.2.2 Computações *304*
B.2.3 Hipótese do Intercalamento *305*
B.2.4 Sistema de Transição Justo *306*
B.3 MODELO EM REDES DE PETRI 308
B.3.1 Módulos e portas *308*
B.3.2 Conectores *309*
B.3.3 Conectores síncronos e assíncronos *312*
B.4 MODELO PARA OS COMPONENTES 315
B.4.1 Módulo *315*
B.4.2 Portas e Interação *319*
B.4.3 Concorrência *322*
B.5 VERIFICAÇÕES 323
B.5.1 Verificação de Modelo (Model Checking) *323*
B.5.2 Verificação por partes *327*
B.6 CONCLUSÃO 328

Lista de Figuras

FIGURA 1.1 - UMA ARQUITETURA DE *SOFTWARE* EM R-RIO 23
FIGURA 2.1 - REFLEXÃO COMO INTERPOSIÇÃO DE CÓDIGO 46
FIGURA 2.2 - ARQUITETURA DE META-NÍVEL DE UM SISTEMA DE ARQUIVOS DISTRIBUÍDO.......47
FIGURA 2.3 - (A) REFLEXÃO ESTRUTURAL E (B) REFLEXÃO COMPUTACIONAL 48
FIGURA 2.4 - UTILIZANDO REFLEXÃO PARA ADICIONAR CARACTERÍSTICAS NÃO-FUNCIONAIS .51
FIGURA 2.5 - SUPORTE PARA A CONFIGURAÇÃO DE APLICAÇÕES DISTRIBUÍDAS 53
FIGURA 2.6 - CONEXÃO DE DOIS MÓDULOS 55
FIGURA 2.7 - AMBIENTE MULTIPROTOCOLO CONFIGURÁVEL BASEADO EM REFLEXÃO 66
FIGURA 3.1 - *BUFFER* LIMITADO POR REFLEXÃO 70
FIGURA 3.2 - UM OBJETO EM FC..81
FIGURA 3.3 - AOP E O ENTRELAÇADOR (*ASPECT WEAVER*)..84
FIGURA 3.4 - NOTAÇÃO UTILIZADA EM R-RIO PARA A APLICAÇÃO PRODUTOR-CONSUMIDOR 100
FIGURA 4.1 - MÓDULO, PORTA E MÉTODOS..106
FIGURA 4.2 - MAPEAMENTO DE UM MÓDULO..107
FIGURA 4.3 - INTERLIGANDO MÓDULOS ATRAVÉS DE CONECTORES. (A) POR CONECTORES DISTINTOS (B) USANDO UM ÚNICO CONECTOR 112
FIGURA 4.4 - UMA APLICAÇÃO COM MÓDULO COMPOSTO 116
FIGURA 4.5 - UM CONECTOR INTERLIGANDO VÁRIOS SERVIÇOS DO MÓDULO Y ÀS REQUISIÇÕES DO MÓDULO X 117
FIGURA 4.6 - UM CONECTOR COMPOSTO 117
FIGURA 4.7 - CONECTOR COMO ELEMENTO DE META-NÍVEL..121
FIGURA 4.8 - EXEMPLO DO USO DE CONECTORES REFLEXIVOS POR CONTEXTO.......................123
FIGURA 4.9 - A ETAPA DE DESCRIÇÃO E CONFIGURAÇÃO DE ARQUITETURAS DE *SOFTWARE* ...125
FIGURA 4.10 - ARQUITETURA PRODUTOR-CONSUMIDOR-*BUFFER* (EXEMPLO 1) 132
FIGURA 4.11 - CONECTOR CONCEITUALMENTE PRESENTE 132
FIGURA 4.12 - APLICAÇÃO PRODUTOR-CONSUMIDOR 133
FIGURA 5.1 - CONFIGURAÇÃO DE UMA APLICAÇÃO SIMPLES 137
FIGURA 5.2 - MÓDULO COM MÉTODOS SINCRONIZADOS 144
FIGURA 5.3 - UM MÓDULO COM GUARDAS PARA SEUS MÉTODOS 146
FIGURA 5.4 - CONSULTA AO ESTADO DE UM MÓDULO..146
FIGURA 5.5 - DIAGRAMA DE CLASSES PARA O DESIGN PATTERN *SYNCHRONIZER* 147

FIGURA 5.6 - APLICAÇÃO COM MÓDULO-SEMÁFORO 148
FIGURA 5.7 - DIFERENTES VISÕES DA ARQUITETURA DE UMA APLICAÇÃO 151
FIGURA 5.8 - *BUFFER* COM MÉTODOS SINCRONIZADOS 156
FIGURA 5.9 - APLICAÇÃO DISTRIBUÍDA 160
FIGURA 5.10 - APLICAÇÃO COM MÓDULO COMPOSTO 161
FIGURA 6.1 - ESTRUTURA PARA VIABILIZAR ARQUITETURAS DE *SOFTWARE* COM QOS 168
FIGURA 6.2 - QOS PARA PROTOCOLOS DE COMUNICAÇÃO 171
FIGURA 7.1 - ARQUITETURA DO *MIDDLEWARE* DE SUPORTE R-RIO 186
FIGURA 7.2 - FUNCIONAMENTO DO CONECTOR REFLEXIVO POR CONTEXTO 188
FIGURA 8.1 - CONFIGURAÇÃO INICIAL DA FÁBRICA 210
FIGURA 8.2 - RECONFIGURAÇÃO DA FÁBRICA 212
FIGURA 8.3 - CLASSES DA APLICAÇÃO TICTACTOE 213
FIGURA 8.4 - APLICAÇÃO TICTACTOE 215
FIGURA 8.5 - VERSÃO DISTRIBUÍDA DO JOGO 218
FIGURA 8.6 - CONECTOR COBSERVER EM DUAS APLICAÇÕES DIFERENTES 219
FIGURA 8.7 - ARQUITETURA DO TELEFONE 3-EM-1 222
FIGURA 9.1 - (A) FLUXO NORMAL DE REQUISIÇÕES E RESPOSTAS EM UMA CADEIA DE CONECTORES; (B) POSSIBILIDADE DE OTIMIZAÇÃO DO FLUXO DE RESPOSTA 243
FIGURA A.1 - CONTRATO DE INTERAÇÃO *DEFAULT* 285
FIGURA A.2 - UM CONECTOR COM DUAS PORTAS DE ENTRADA E SAÍDA 290
FIGURA A.3 - UM CONECTOR COM UMA PORTA DE SAÍDA COM GUARDA 290
FIGURA A.4 - CADEIA SEQÜENCIAL DE CONECTORES 294
FIGURA A.5 - COMBINANDO COMPOSIÇÃO SEQÜENCIAL E PARALELA 294
FIGURA B.1 - DOIS MÓDULOS, COM UM MÉTODO CADA UM, SEM CONEXÃO 309
FIGURA B.2 - (A) REPRESENTAÇÃO DE UMA PORTA, (B) REPRESENTAÇÃO FUNCIONAL DE UMA PORTA DE SAÍDA INTERAGINDO COM UMA PORTA DE ENTRADA 309
FIGURA B.3 - CONECTOR COMO VISÃO DE META-NÍVEL DE UMA INTERAÇÃO ENTRE MÓDULOS 310
FIGURA B.4 - NO EIXO DOS TEMPOS, O FUNCIONAMENTO DO MÓDULO CLIENTE, SEGUIDO DO CONECTOR E MÓDULO SERVIDOR 310
FIGURA B.5 - VISÃO FUNCIONAL DE UMA INTERAÇÃO SÍNCRONA ENTRE UMA PORTA DE SAÍDA E UMA PORTA DE ENTRADA, (B) A TRANSIÇÃO ÚNICA DE (A) É DIVIDIDA EM 4 TRANSIÇÕES NA VISÃO DE META-NÍVEL 311
FIGURA B.6 - VISÃO FUNCIONAL DA INTERAÇÃO DE MÓDULOS (A) SÍNCRONA (B) ASSÍNCRONA 311
FIGURA B.7 - VISÃO DE META-NÍVEL DA INTERAÇÃO DE MÓDULOS INTERMEDIADOS POR

CONECTORES. (A) SÍNCRONA (B) ASSÍNCRONA (1A VERSÃO)..312
FIGURA B.8 - VISÃO DE META-NÍVEL DA INTERAÇÃO DE MÓDULOS INTERMEDIADOS POR CONECTORES. (A) SÍNCRONA (B) ASSÍNCRONA (2A VERSÃO)..313
FIGURA B.9: REPRESENTAÇÃO DE META-NÍVEL DAS PORTAS COMO UMA SOMA DE INTERAÇÕES SÍNCRONAS E ASSÍNCRONAS. 314
FIGURA B.10 - REDE DE PETRI CONDIÇÃO/AÇÃO DO CONECTOR (A) PORTA DE SAÍDA GPUT E (B) PORTA DE SAÍDA GGET..326

Lista de Acrônimos

ADL *Architecture Description Language* (linguagem de descrição de arquiteturas)

AOP *Aspect-Oriented Programming* (programação orientada a aspectos)

AS Arquitetura de *Software*

BC Baseado em Componentes (*Component-Based*)

LSTS *Labeled State Transition System* (sistema de transições de estados rotulados)

MI *Method Invocation* (invocação de método)

MOP *Meta-Object Protocol* (protocolo de meta-objeto)

ORB *Object Request Broker*

OTS *Off the Shelf* ("da prateleira")

PC Programação por Configuração

PDA *Personal Digital Assistant*

PM-N Programação de Meta-Nível

QoS *Quality of Service* (qualidade de serviço)

RMI *Remote Method Invocation* (invocação de método remoto)

RPC *Remote Procedure Call* (chamada a procedimento remoto)

UML *Universal Modeling Language*

URL *Uniform Resource Locator*

XML *Extensible Markup Language*

Esta página foi intencionalmente deixada em branco

Capítulo I

Introdução

1.1 Introdução

A evolução tecnológica em sistemas de computação vem facilitando o surgimento de aplicações que apresentam grande complexidade e dinamismo. Hoje, já é realidade o uso de computadores pessoais móveis (tais como *palm tops* ou PDAs - *Personal Digital Assistants*) interligados à Internet por *modems celulares* ou *redes ad-hoc*, tornando ubíquo o acesso a serviços distribuídos [1, 2]. Sobre esta estrutura, aplicações podem ser construídas para manipular e coordenar o processamento de informações com origem e natureza diferentes, tais como textos, imagens, áudio, vídeo, ou sinais de sensores. Tanto a criação ou a aquisição destas informações, quanto o seu processamento, podem se dar, simultaneamente, em pontos concentrados - como uma sala - ou dispersos por grandes redes - como a Internet [3, 4, 5]. A tendência de tais aplicações é que sejam distribuídas, heterogêneas e requeiram qualidade de serviço (*quality of service*, QoS) de seus ambientes de suporte. Também é provável que as várias partes destas aplicações possam ser construídas por diversos projetistas e executadas em plataformas computacionais diferentes. Além disso, estas aplicações precisam ser desenvolvidas rapidamente, para atender às demandas de mercado. É necessário que variantes de uma base funcional de um sistema sejam entregues em um espaço curto de tempo, e ainda atendam a requisitos funcionais e não-funcionais específicos.

A fim de que o desenvolvimento de *software* para estas aplicações seja bem sucedido, devem ser consideradas as seguintes premissas:

- modularidade é primordial para construir-se um conjunto de componentes funcionais e, seletivamente, adicionarem-se a este características não-funcionais;
- componentes podem ser projetados independentemente, implementados utilizando-

se diferentes linguagens de programação, e devem ser capazes de executar sobre diferentes plataformas;

- nem todos os requisitos de uma aplicação são conhecidos antes da fase de desenvolvimento. Com freqüência, novas funções precisam ser integradas à aplicação. Além disto, deve ser possível adaptar esta aplicação a ambientes de execução heterogêneos;
- a verificação formal de propriedades deveria garantir a qualidade do *software* produzido.

No contexto delineado, são necessárias metodologias, sistemas de desenvolvimento e ambientes de suporte que possam integrar as premissas, mencionadas anteriormente, em práticas sistemáticas de engenharia de *software*. Observa-se que, no estado da arte, a tecnologia de *hardware* evoluiu rapidamente (máquinas com maior poder computacional e redes de alta velocidade), fazendo frente às demandas das aplicações, mas não foram criadas metodologias e sistemas de suporte totalmente adequados para a concepção do *software* destas aplicações.

Uma das alternativas para contemplar a evolução necessária das aplicações é manter os sistemas de *software* abertos a extensões ou à interação com outros sistemas [6, 7]. É desejável que a evolução das aplicações possa ocorrer por aperfeiçoamentos gradativos. Neste sentido, o desenvolvimento por **composição** e a **separação de interesses** são conceitos importantes para a construção destas aplicações.

A Programação de Meta-Nível (*Meta-Level Programming*), PM-N, permite o arranjo de elementos de *software* em diferentes níveis de interesse. Utilizando-se técnicas reflexivas, o projetista pode isolar o código relacionado com requisitos não-funcionais em um meta-nível e fazer com que a computação do nível-base seja reificada sempre que necessário. Entretanto, a PM-N está geralmente associada a linguagens orientadas a objeto específicas, onde a composição é obtida utilizando-se mecanismos de herança, que escondem a estrutura do *software* dentro de objetos. Esta opacidade da estrutura do *software* dificulta as atividades de verificação e evolução dinâmica.

As abordagens baseadas em componentes (*Component-Based*), BC, permitem a composição de aplicações a partir de elementos de *software* (componentes) construídos de forma independente e heterogênea, provendo reusabilidade e interoperabilidade.

Alguns ambientes de desenvolvimento baseados em componentes oferecem mecanismos para programação de requisitos não-funcionais (por exemplo, interceptores de CORBA [8]), mas lhes faltam conceitos e mecanismos para descrever, configurar e verificar as aplicações de forma sistemática.

1.2 Objetivos

O objetivo central desta tese é demonstrar que a capacidade de abstração, descrição e análise das tecnologias de Arquitetura de *Software* e Programação por Configuração (*Software Architecture / Configuration Programming*), AS/PC, combinada à flexibilidade da reificação e à capacidade intrínseca de separação de interesses de PM-N pode resultar em um *framework* adequado para o desenvolvimento da classe de aplicações mencionada.

AS/PC vai além de BC, permitindo a descrição de sistemas de *software* em um nível abstrato, e considerando componentes funcionais e seus esquemas de interação como interesses separados. Isto facilita ao projetista o entendimento da arquitetura do sistema e permite configurar as aplicações para atenderem a requisitos específicos.

ASs podem ser descritas através de Linguagens de Descrição de Arquiteturas (*Architecture Description Languages*), ADL. Usando uma ADL, um projetista de sistemas pode especificar a composição funcional do sistema, através da seleção de módulos, e associá-los a estilos particulares de interação ou contratos (que no contexto de comunicação de dados são chamados de protocolos), através de portas e conectores. A esta atividade denominamos Programação por Configuração (PC). ADLs também são apropriadas para verificações de propriedades da arquitetura que está sendo descrita. Por exemplo, a exposição explícita da composição de módulos, descrevendo a topologia da arquitetura, facilita as verificações. Esta exposição também facilita a obtenção de um mapeamento natural da arquitetura de *software*, para artefatos de *software* propriamente ditos, e pode, em um estágio posterior, facilitar atividades de reconfiguração dinâmica.

Técnicas de PM-N, como a reflexão, por sua vez, tornam possível o projeto de aplicações em níveis diferentes: um nível-base, onde a computação funcional da aplicação é realizada, e um meta-nível, onde a computação não-funcional (incluída a operacional) pode ser tratada. Além disso, a técnica de reflexão permite que uma

aplicação obtenha informações sobre sua própria estrutura (reflexão estrutural) e comportamento (reflexão comportamental). Com estas informações, é possível, com um suporte adequado, fazer adaptações na aplicação em face a mudanças no seu estado de operação. Por exemplo, no contexto de uma linguagem procedural com suporte à reflexão, determinadas chamadas a procedimento podem ser interceptadas para que o fluxo de controle seja transferido para um código extra, de meta-nível. Isto seria similar à transferência de controle que ocorre entre um módulo e um conector em um contexto de AS. Estas observações nos levam ao caminho da integração destas duas abordagens, AS/PC e PM-N, de forma a se obterem potencialmente os benefícios de cada uma delas.

Mesmo apresentando características apropriadas para o desenvolvimento de aplicações, quando tomadas separadamente, o uso de AS/PC e PM-N apresenta restrições. Ferramentas para PM-N e ambientes de suporte à execução com reflexão não oferecem suporte direto para atividades de configuração e reconfiguração. Neste caso, configurações e reconfigurações são geralmente programadas de forma *ad-hoc* e são dependentes do talento do programador, prejudicando o reuso. A verificação de propriedades também é dificultada porque a estrutura do *software* não fica exposta.

A descrição explícita de aspectos não-funcionais em propostas de AS/PC, por sua vez, não é comum. A maioria das propostas contemporâneas considera apenas aspectos não-funcionais específicos, como a comunicação remota. Combinando AS/PC e PM-N, pode-se, por exemplo, tratar interesses de interconexão e a interação de componentes (não-funcionais) em um meta-nível, e obter-se um *framework* que ofereça as vantagens das duas abordagens. Desta forma, AS/PC pode disciplinar o uso de reflexão, oferecendo um mecanismo simples e sistemático para reificar interações. Adicionalmente, mudanças dinâmicas na aplicação, iniciadas no meta-nível, podem agora ser tratadas com o suporte para reconfiguração provido por AS/PC.

Como resultado da combinação de AS/PC e PM-N, buscamos encontrar uma base para guiar a construção de ferramentas para descrever, configurar, verificar, operar e manter arquiteturas de *software*.

1.3 O *Framework* R-RIO

Buscando atingir os objetivos apontados, investigamos o uso combinado dos

conceitos de AS/PC e PM-N para descrição e execução de aplicações. Como resultado, chegamos ao conceito de um *framework*, R-RIO (*Reflective-Reconfigurable Interconnectable Objects*), que inclui os seguintes elementos:

- um modelo de componentes baseados nos conceitos de AS/PC (figura 1.1): (a) **módulos**, componentes da arquitetura de uma aplicação, que encapsulam basicamente interesses funcionais; (b) **conectores**, componentes da arquitetura, que definem as relações de interação entre os módulos; (c) **portas**, elementos que identificam pontos de acesso, através dos quais os módulos e conectores requisitam e oferecem serviços;

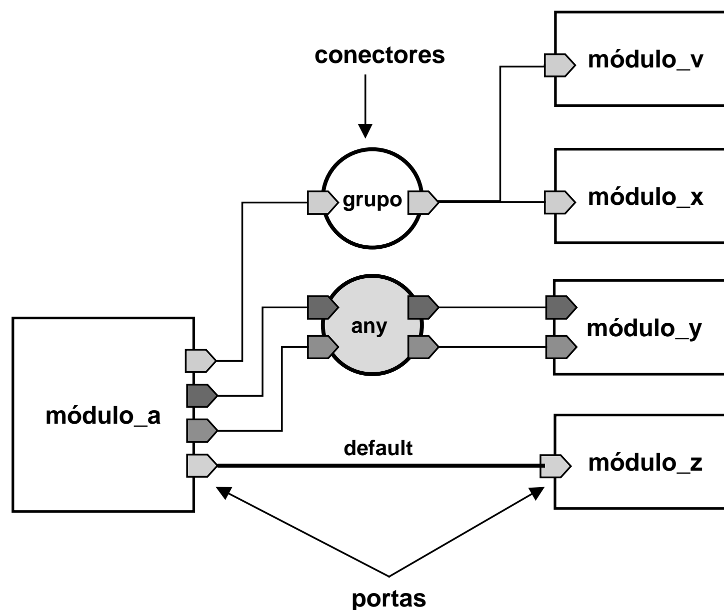


Figura 1.1 - Arquitetura de *software* em R-RIO

- um modelo de gerência de configuração, que permite a criação, ligação, terminação e a reconfiguração dinâmica de componentes;

- CBabel, uma ADL, usada para descrever: (i) componentes de uma aplicação e a estrutura de interligação destes componentes, (ii) contratos especificando interesses não-funcionais, tais como padrões especiais de interação, coordenação, distribuição e qualidade de serviço (QoS);

- um *middleware* reflexivo, que oferece os serviços de gerência, execução e reconfiguração de arquiteturas de *software*;

- uma metodologia para a configuração de arquiteturas de *software*, que estimula o projetista a aderir a uma disciplina simples de programação de meta-nível, onde interesses funcionais são concentrados nos módulos (nível-base) e os interesses não-funcionais são encapsulados em conectores (meta-nível).

1.4 Validação

A partir dos conceitos de R-RIO, foi desenvolvido um protótipo do *middleware* reflexivo implementando um serviço de gerência de configuração. Em conjunto com este protótipo, também foram desenvolvidos e testados alguns conectores, encapsulando aspectos não-funcionais diferentes. Por exemplo, para aplicações distribuídas, foram realizados testes com conectores encapsulando mecanismos de comunicação de propósito geral (*sockets*, RMI, CORBA). Também foram testados conectores para intermediar a interação entre módulos através de estilos específicos, como a difusão seletiva (*multicast*), registro de informações sobre as interações (*log*) e *design patterns* como o *Observer*.

Foi proposta, igualmente, uma técnica para se descreverem requisitos de Qualidade de Serviço (*Quality of Service* - QoS) no nível de arquitetura de *software*, e uma estrutura padronizada para a implantação concreta de QoS em aplicações centradas na abordagem de R-RIO.

A validação das idéias contidas na tese foi completada através da concepção de algumas aplicações, desenvolvidas com as facilidades do *framework* proposto, resultando em sistemas reutilizáveis e facilmente adaptáveis a novos requisitos.

1.5 Contribuições

Uma das contribuições importantes deste trabalho é a proposta de integração dos conceitos de AS/PC e PM-N, como solução potencial para a descrição, configuração e gerência de aplicações. A utilização combinada destas tecnologias possibilita o uso de reflexão para a programação de aspectos não-funcionais, e torna viável a configuração como forma de sistematizar e facilitar o uso da reflexão. Esta combinação se mostrou flexível, apropriada ao desenvolvimento de um amplo espectro de aplicações, e contempla a separação de interesses, o que leva mais facilmente ao reuso do *software* produzido.

Durante nossas investigações para a concepção do *framework*, foram identificados alguns padrões de uso de reflexão. Por exemplo, R-RIO utiliza reflexão computacional para adaptar as interações entre módulos, e a reflexão arquitetural, para permitir que as informações de meta-nível de uma arquitetura de *software* sejam consultados. Em um nível menor de granularidade, R-RIO utiliza reflexão por contexto para a adaptação automática de conectores às interfaces dos módulos.

Uma outra contribuição de nossa proposta é a demonstração das vantagens do emprego de separação de interesses na descrição de arquiteturas de *software*, e durante sua execução. Por exemplo, a ADL CBabel permite que se descrevam separadamente os requisitos funcionais e não-funcionais de uma arquitetura de *software*. A proposta de mapeamento das abstrações do nível de arquitetura, para o nível de implementação, permite a manutenção da separação de interesses durante a fase de operação. Desta forma é possível observar-se a separação de interesses no ciclo de vida de uma aplicação, o que facilita sua evolução dinâmica.

1.6 Estrutura do Texto

O trabalho está estruturado em nove capítulos, incluindo o presente. No segundo capítulo, define-se o contexto tecnológico no qual a tese está inserida (sistemas distribuídos, arquiteturas de *software*, componentes, *middleware*, sistemas reconfiguráveis, etc.). Procurou-se, dentro deste contexto, a definição de um conjunto de requisitos para concepção de aplicações. Ainda neste capítulo analisam-se as tecnologias fundamentais de Objetos, Programação de Meta-Nível (e suas variantes) e Arquitetura de *Software* / Programação por Configuração, que potencialmente podem ser usadas para atender aos requisitos identificados.

No capítulo 3, trabalhos correlatos são avaliados. Foram investigadas propostas, centradas em reflexão computacional [9, 10, 11], programação orientada a aspectos [12], linguagens orientadas a composição de objetos [13] e configuração [14, 15, 16, por exemplo]. Ainda neste capítulo são discutidas algumas soluções contemporâneas e tendências que cercam o contexto delineado, tais como *middleware*, CORBA, Jini e UML. A idéia é desenhar-se um quadro abrangente sobre os esforços feitos na área de engenharia de *software* para a concepção e gerência de aplicações.

O quarto capítulo é destinado à apresentação da proposta do *framework* R-RIO. Introduz-se, primeiramente, o modelo de componentes e o modelo de configuração. Discute-se como os conceitos de AS/PC e PM-N podem ser utilizados de forma integrada, e os benefícios potenciais desta integração em nossa proposta. Em seguida, uma seqüência de exemplos, utilizando a aplicação dos produtores-consumidores com *buffer* limitado, explora as características básicas do *framework*.

O capítulo 5 apresenta a abordagem de R-RIO para a configuração de aspectos não-funcionais e discute como estes são tratados, através do conceito de contratos. Para ilustrar a configuração de aspectos não-funcionais em R-RIO, o exemplo dos produtores-consumidores com *buffer* limitado é novamente utilizado para mostrar como aspectos de sincronização, concorrência e distribuição podem ser introduzidos na aplicação separadamente de seus aspectos funcionais. O capítulo é concluído com uma discussão sobre a capacidade de gerenciar aspectos de uma aplicação de forma dinâmica em R-RIO. Buscou-se, além disso, esclarecer como o emprego da separação de interesses facilita a reconfiguração das aplicações.

Uma abordagem para incluir QoS no nível da arquitetura de *software* é apresentada no capítulo 6. Propõe-se o uso de contratos de QoS em CBabel, e uma estrutura geral para a implantação destes contratos na execução da aplicação em R-RIO.

Aspectos de implementação de R-RIO são discutidos no capítulo 7. Neste capítulo é apresentada uma estrutura para o *middleware*, que possui os elementos necessários para executar e gerenciar as aplicações em R-RIO. Apresentam-se também as APIs para configuração e reflexão arquitetural, que podem ser usadas para reconfigurar dinamicamente as aplicações.

O *framework* é validado no capítulo 8. Aí são apresentados exemplos de aplicação que exploram as diversas possibilidades de nossa proposta. Procuramos evidenciar, quando possível, a flexibilidade, a separação de interesses e os aspectos dinâmicos nestes exemplos.

O capítulo 9 concentra observações sobre a tese e as nossas conclusões. Primeiramente nossa proposta é avaliada em relação aos requisitos delineados no capítulo 2. Em seguida, pontos importantes, tais como a composição de interesses diferentes e otimizações nas arquiteturas são discutidos. Também são destacados alguns tópicos decorrentes de nosso trabalho, os quais podem constituir assunto para novas

pesquisas. Fechamos o capítulo com um sumário sobre as contribuições de nossa pesquisa e as conclusões finais.

Complementando os capítulos apresentados, dois apêndices foram incluídos.

No apêndice A, apresenta-se a ADL CBabel. A sintaxe para a configuração de aplicações, composição de módulo e conectores é discutida. Também é apresentada a descrição dos contratos de aspectos não-funcionais em CBabel. Em seguida discutem-se alguns detalhes sobre a composição de conectores. O apêndice é encerrado com o BNF de CBabel e uma avaliação geral da linguagem.

No apêndice B apresenta-se a proposta de um modelo formal para os componentes de R-RIO. Este modelo ajuda o entendimento dos conceitos de módulos e conectores. Também é capturada a idéia das portas como pontos de interação entre os componentes, e a dos conectores, como elementos de meta-nível em uma arquitetura. Mostra-se neste apêndice, por fim, um dos caminhos possíveis para realizarem-se verificações em uma arquitetura de *software*, a partir do modelo formal proposto.

Esta página foi intencionalmente deixada em branco

Capítulo II

Contexto Tecnológico

2.1 Introdução

No capítulo 1, discutiu-se que uma aplicação, mesmo em operação, pode passar por constantes adaptações. Por exemplo, novas demandas são criadas por usuários desta aplicação, que precisam de uma função não prevista originalmente ou necessitam alterar funções existentes. Juntamente com estas novas demandas funcionais podem surgir requisitos não-funcionais que, de forma geral, não estão associados diretamente às funções da aplicação. Protocolos de comunicação, qualidade de serviço, disponibilidade ou tolerância a falhas são requisitos considerados não-funcionais. Estes requisitos também podem sofrer modificações ao longo da execução das aplicações, em reação a mudanças nas condições de operação da aplicação.

Modelos tradicionais para a concepção de aplicações não oferecem as facilidades necessárias à produção de *software* que se adapte dinamicamente a novos requisitos. Na maioria dos casos a estrutura e a funcionalidade das aplicações é estática e fruto de um grande esforço de programação, resultado de um código complicado, com vários aspectos entrelaçados (*code tangling* [12]). Em tais circunstâncias, é comum um mesmo bloco de código conter instruções que executam a computação intrínseca da aplicação, instruções para a comunicação via *sockets* e a sincronização via semáforos. Esta mistura cria dificuldades para a reutilização de código.

Para facilitar a concepção destas aplicações, uma nova abordagem se faz necessária. Deve-se considerar, por exemplo, que todas as aplicações precisam evoluir dinamicamente em seus aspectos funcionais e não-funcionais, ou mesmo em sua estrutura. Esta nova abordagem deve facilitar a gerência da evolução destas aplicações, mesmo durante sua operação, com o dinamismo e consistência esperados.

O problema do entrelaçamento de código (*Code Tangling*)

O problema do entrelaçamento de código será apresentado através de um exemplo: o uso de *middleware* OTS (*off-the-shelf* - "da prateleira") para a comunicação cliente-servidor em um ambiente distribuído.

Uma das situações que dificultam a adaptação de uma aplicação sobre determinado *middleware* é o entrelaçamento de código que se observa na implementação destas aplicações. Como exemplo deste tipo de *middleware* podemos citar RMI - *Remote Method Invocation* [18] e CORBA - *Common Object Request Brocker Architecture* [8]. Freqüentemente, o programador deve seguir uma "receita" de programação com a finalidade de fazer, por exemplo, uma simples invocação de método em um objeto remoto. Esta prática, algumas vezes, implica a utilização restrita dos serviços do *middleware*, e outras vezes, o uso ineficiente do mesmo. O programador, que deveria pensar apenas na funcionalidade básica da aplicação, está "preocupado com *stubs* e *skeletons*." [19].

A listagem 2.1(a) apresenta um trecho de código, em linguagem de programação Java, onde um componente precisa executar um método (*teste*) de um objeto remoto (*myServer*), utilizando-se de RMI. Antes que seja possível a invocação do método remoto, o cliente precisa consultar um serviço de diretório (implementado pelo *registry*, no caso de RMI), obter a referência de um tratador e, somente depois, fazer a invocação do método. Na listagem 2.1(b) um trecho do lado do servidor é apresentado. Observa-se que uma boa parte do código é associada ao uso de RMI. Observa-se também que a funcionalidade do servidor e do cliente está entrelaçada com o uso de RMI, impedindo a reutilização de código e não observando a separação de interesses.

```
...
try {
   testaRmi myServer =
   (testaRmi) Naming.lookup
           ("rmi://" + host
            + "/" + "testaRMI");
.
   result = myServer.teste(s);
.
}
catch (ConnectException e) {}
catch (java.rmi.NotBoundException e) {}
catch (java.net.MalformedURLException e)
{}
catch (java.rmi.RemoteException e) {}
...
```

```
...
public
class testaImpl extends
           UnicastRemoteObject
             Implements testaRmi {
  public testaImpl (String nome)
          throws RemoteException {
   super();
   try {
     Naming.rebind (nome,this); }
   catch (Exception ex) { }
}
public int teste (String s)
             throws RemoteException
 { return 1; }
}
```

Listagem 2.1 - Exemplo RMI: (a) cliente (b) servidor

Se em determinado momento decidir-se pela utilização de outro *middleware*, como CORBA, apostando-se em maior interoperabilidade, ou por um mecanismo como *sockets* [20], para aumentar a eficiência na comunicação, uma boa parte do código deverá ser refeita. Por exemplo, utilizando-se CORBA, uma rotina de inicialização do ORB e um acesso ao serviço de nomes do CORBA devem ser incluídos. Na listagem 2.2, isto ocupa a maior parte do código, até que a invocação ao método possa realizar-se (em negrito).

```
import crbC.*;              // The package containing our stubs.
import org.omg.CosNaming.*;  // will use the naming service.
import org.omg.CORBA.*;      // All CORBA applications need these classes.

   ...
   Properties props = new Properties();
   props.put("org.omg.CORBA.ORBClass", "com.sun.CORBA.iiop.ORB");
   props.put("org.omg.CORBA.ORBInitialHost", host);
   props.put("org.omg.CORBA.ORBInitialPort", "4900");

   ORB orb = (ORB) ORB.init(args, props);

   org.omg.CORBA.Object objRef =
   (org.omg.CORBA.Object) orb.resolve_initial_references("NameService");
   NamingContext ncRef = NamingContextHelper.narrow(objRef);

   NameComponent nc = new NameComponent("testa CORBA", "");
   NameComponent path[] = {nc};
   corbaC corbaCRef = corbaCHelper.narrow(ncRef.resolve(path));

   String Str_res = corbaCRef.teste(s);
   ...
```

Listagem 2.2 - Exemplo CORBA

Em geral, *middlewares* como RMI e CORBA utilizam a interposição de código ([21] e seção 4.6) entre o cliente e o servidor. No caso do emprego de RMI, dentro da aplicação são feitas indicações de que este mecanismo será usado e, durante a execução, o fluxo de informações passa pelo código interposto chamado *stub* (*stub* cliente e *stub* servidor ou *skeleton*[1]). Os *stubs* são responsáveis pelo empacotamento e desempacotamento de parâmetros em mensagens (procedimento chamado de *marshalling*) e por acionar o transporte, pela rede, das requisições e respostas contidas nestas mensagens. Embora a geração automatizada de *stubs* libere o programador dos detalhes da comunicação, alocação de *buffers* para transmissão e transporte dos dados, ainda é necessário codificar o uso do RMI. Similar a este exemplo, outros poderiam ser

---

[1] A partir da versão 1.2 de Java, um protocolo para *stub* foi introduzido para eliminar a necessidade do *skeleton* (ou *stub* servidor). No lugar, código genérico é usado para executar as mesmas tarefas. Para a interação de objetos executando em JVMs anteriores a 1.2, os *skeletons* ainda são necessários.

dados, como o RPC ou o mecanismo de *socket*, já mencionado. Ou seja, ainda existem algumas lacunas nestes mecanismos para se obter flexibilidade na concepção do *software* de aplicações, mesmo em casos simples como o apresentado.

A primeira parte deste capítulo apresenta uma seleção de requisitos recorrentes na construção de aplicações. Estes requisitos devem contemplar os seguintes pontos:

- componentes podem ser projetados independentemente, podem ser implementados utilizando-se diferentes linguagens de programação. Potencialmente, estes componentes executarão sobre diferentes plataformas;
- deve ser possível a reutilização de componentes e código para construir novas aplicações;
- deve ser possível adicionar, transformar ou atualizar dinamicamente a funcionalidade das aplicações;
- aplicações podem ser adaptadas 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*);
- a qualidade do *software* produzido pode ser melhorada através da verificação de propriedades.

Na segunda parte, analisam-se tecnologias que oferecem contribuição para o atendimento destes requisitos, simplificando o projeto das aplicações. A tecnologia de objetos é apresentada como básica. De uma forma geral as outras tecnologias são baseadas em objetos. A programação de meta-nível (PM-N), sistemas baseados em componentes (BC) e arquiteturas de *software* (AS) são analisadas em relação ao seu potencial em atender aos requisitos selecionados.

A partir desta análise, na terceira parte deste capítulo, é avaliado se a combinação das referidas tecnologias pode levar, mais facilmente, ao atendimento dos requisitos enumerados anteriormente. Com o resultado desta investigação chegamos a proposta de um conjunto de diretivas para guiar o projeto de aplicações, segundo tais requisitos.

2.2 Requisitos básicos

Um conjunto de requisitos para o projeto e implementação de aplicações distribuídas foi selecionado com apoio em trabalhos pesquisados [22, 23, 24, 25, 26, 27]. Nesta seção, este conjunto é examinado de forma a se determinar sua relevância no contexto das discussões que se seguirão.

Reusabilidade. Técnicas empregadas na concepção de aplicações devem permitir que partes de aplicações previamente concebidas sejam reutilizadas de forma a minimizar esforços de desenvolvimento. Em um caso extremo, uma aplicação pode ser criada completamente a partir de partes já existentes [25, 28]. 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 de *software*, onde se deseja construir aplicações interconectando-se componentes disponíveis [29]. Desta forma, esforços poderão concentrar-se na concepção da arquitetura da aplicação e não no desenvolvimento de partes básicas.

Dois conceitos estão associados ao de reusabilidade: modularidade e interoperabilidade. O projeto de componentes modulares está vinculado a dois aspectos: o encapsulamento, em que o comportamento do módulo é auto-contido, e a padronização de interface, em que as regras de acesso à funcionalidade do módulo são claramente definidas. Interoperabilidade é essencial para o suporte de aplicações potencialmente distribuídas, compostas de módulos, oferecidos por provedores independentes, que precisam se comunicar e cooperar [6]. 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 [27].

Separação de Interesses. O conceito de separação de interesses (*separation of concerns*) é considerado fundamental para a concepção de grandes sistemas [30, 31, 13, 7]. A separação de interesses divide uma aplicação em partes funcionais e não-funcionais. Aspectos relacionados com a computação básica de uma aplicação podem idealmente ser tratados independentemente de aspectos não-funcionais, tais como a

coordenação, comunicação e distribuição. Da mesma forma, aspectos não-funcionais como protocolos de comunicação, QoS e gerência de recursos do ambiente de execução, considerados operacionais, também podem ser tratados separadamente. A obtenção da separação de interesses exige, entretanto, uma disciplina do projetista. A arquitetura da aplicação deve ser planejada com tal finalidade, considerando-se a modularidade e interoperabilidade durante este processo. Assim sendo é possível concentrar esforços para a concepção dos módulos funcionais e para a adaptação da aplicação às especificações não-funcionais em momentos diferentes.

Evolução Dinâmica. No escopo deste trabalho, o requisito de evolução dinâmica considera o suporte para a modificação e inclusão de novos componentes em aplicações, que não podem ser completamente paradas para realizar estas operações. Por exemplo, a atualização do *software* de componentes, inclusão de tolerância a falhas, alteração de parâmetros de QoS ou alteração de estilos de comunicação são atividades que podem implicar em mudanças em tempo de execução. O ambiente de execução deve disponibilizar mecanismos que permitam a inclusão de novos módulos e sua ligação a outros componentes da aplicação [32].

No nível da aplicação, o projetista pode ser obrigado a antecipar estas possibilidades, embora em certos casos algum grau de transparência seja possível. Entretanto, alterações dinâmicas podem introduzir problemas não triviais relacionados com segurança. Em certos casos é preciso ter acesso à estrutura interna de algumas partes da aplicação ou do ambiente de execução. Por exemplo, para alterar o comportamento de um módulo pode haver necessidade de consultar-se 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 várias facetas. Primeiramente, deseja-se que um grande número de problemas seja resolvido com uma dada tecnologia, e que para isso ela possua expressividade adequada. Adicionalmente, são observadas a disponibilidade, disseminação e aceitação da tecnologia. Ferramentas facilmente disponíveis e disseminadas facilitam a criação de massa crítica de *experts* e, em conseqüência, são mais aceitas e tendem a 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.

Atentemos que os requisitos enumerados são de natureza diferente e podem estar parcialmente superpostos, ou então não se encontram diretamente relacionados. Entretanto, acreditamos que os mesmos, em conjunto, oferecem uma base consistente para a engenharia de sistemas de *software*. Com este conjunto de requisitos em mente, selecionamos as tecnologias mais relevantes para serem revisadas. Para cada uma destas tecnologias discute-se como estes requisitos são atendidos.

2.3 Tecnologias e abordagens para o desenvolvimento de aplicações

As tecnologias que se apresentam como candidatas para suportar a concepção de aplicações, no cenário descrito, têm como pontos favoráveis a facilidade de composição, modularidade e separação de interesses, em maior ou menor grau. Em nossa pesquisa nós as agrupamos em:

- Sistemas Baseados em Componentes (BC);
- Programação de Meta-Nível (PM-N); e,
- Arquiteturas de *Software* / Programação por Configuração (AS/PC).

Nas seções que se seguem, apresentamos e discutimos estas tecnologias, ressaltando seus pontos fortes e fracos para atender aos requisitos estabelecidos. Estas são comparadas entre si, e as possibilidades de combinação das mesmas também são investigadas. Adicionalmente, a tecnologia de objetos será considerada como básica no contexto deste trabalho, mas em um nível diferente, comparando-se com BC, PM-N e AS/PC. Objetos podem ser considerados como adequados para a programação das unidades de execução ou módulos, e utilizados em AS/PC, PM-N e BC. Entretanto, algumas de suas características podem afetar diretamente a possibilidade de desenvolverem-se aplicações por composição, como por exemplo a herança, e assim também serão discutidas. Como resultado apresentamos algumas diretivas para o uso destas tecnologias na concepção de aplicações, que são empregadas na formulação do *framework* R-RIO.

2.3.1 Objetos

2.3.1.1 Conceitos básicos

A tecnologia de objetos se apóia em dois conceitos: encapsulamento e polimorfismo [3, 4, 33]. 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 contêm código para manipulá-los. Um objeto somente pode ser acessado através da invocação dos métodos, que definem uma interface para o objeto. O polimorfismo permite que em resposta à invocação de um método, o objeto que vai efetivamente executá-lo 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 [3, 34]. 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 o comportamento utilizado para criar objetos. Um objeto é uma instância viva de uma classe, que ocupa espaço em memória e possui um contexto associado. A outra abordagem são as linguagens baseadas em **objetos** ou **protótipos**. Nestas linguagens, 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 [34]. Contudo, neste trabalho, nos concentraremos nos efeitos destas diferenças com relação aos mecanismos de extensão associados à tecnologia de objetos.

2.3.1.2 Herança

No contexto deste trabalho, herança é considerada sob o ponto de vista de um mecanismo para extensão incremental do 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*. A nova classe em construção pode ter funções adicionadas, ou partes

originais da classe herdada podem ser modificadas (*overriding*). Pode-se interpretar a herança 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 tais variações da seguinte maneira:

- herança de interface: reuso da definição original da estrutura de uma classe sem relação estrita com a implementação desta classe. O uso de herança de interface é comum, quando a superclasse é definida como uma classe abstrata (somente sua interface é descrita);
- herança não estrita: extensão do 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: extensão do comportamento de uma classe, criando-se uma classe especializada estritamente baseada na interface da classe pai e adicionando-se novas características, mas sem alteração em 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 as mesmas;
- generalização: criação de uma classe genérica, baseada na união de um conjunto de subclasses particionadas.

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 estado do objeto matriz.

2.3.1.3 Composição e Concatenação

A distinção entre os termos composição e concatenação é difusa [34, 35]. Ambos são usados como referência a mecanismos de extensão com estilos similares. A composição possibilita a criação de novas classes com características mais adequadas

para um dado contexto, através do compartilhamento de código de classes mais simples, em um estilo *caixa-preta*. Em uma composição, as estruturas de duas ou mais classes são copiadas e combinadas para gerar uma outra. Diferentemente da herança ou herança múltipla, é possível implementar a composição 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 devem sofrer modificações.

2.3.1.4 Delegação e Encaminhamento

Delegação é um mecanismo de extensão, alternativo à herança, disponível em algumas linguagens orientadas a objeto. Este mecanismo permite que um objeto delegue a execução de um método a outro objeto. O mecanismo de delegação utiliza um esquema de auto-referência (*self-reference*) ao repassar a invocação do método do objeto original para o objeto delegado. Desta forma, a execução do método 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 [4] delegação e herança são considerados conceitos distintos onde um pode emular o outro. Em [34] a delegação é definida como um mecanismo através do qual a herança pode ser implementada. Neste caso, o objeto delegado seria uma instância da classe, com o papel de classe pai, se a herança fosse utilizada. Uma observação 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 a 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 [34].

2.3.1.5 *Design Patterns*

O termo *design pattern* (padrão de projeto) é utilizado de forma particular em

linguagens e projetos orientados a objetos. Um *design pattern* 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 provavelmente será evitada. Neste caso, não existe mais a necessidade de se redescobrirem ou reinventarem componentes de *software* comuns.

Em projetos que seguem uma metodologia orientada a objetos, *design patterns* identificam, explicam e avaliam sistematicamente estruturas de projeto utilizadas com freqüência. Em [35], *design patterns* são conceituados como descrições de objetos, classes e suas interações, concebidas para resolver um problema genérico de concepção, mas que são voltadas para um contexto particular. Portanto, *design patterns* não são orientados para a solução de uma aplicação específica, mas para prover componentes de *software* mais primitivos e genéricos, que potencialmente podem constituir-se em partes da solução de concepção para um problema maior.

A reutilização de objetos é uma característica importante de um *design pattern*. Assim sendo, os objetos devem ser concebidos segundo técnicas que permitam aos mesmos serem inócuos, flexíveis, e que evitem o problema da fragilidade de classe-base [241]. Por esta razão, "a composição de objetos é preferida em relação ao uso da herança" [23], levando a um estilo de concepção por caixa-preta, escondendo detalhes internos dos objetos. Seguindo esta linha de raciocínio, a técnica de delegação resulta na mais apropriada para este objetivo, tornando a composição tão poderosa, para permitir a reutilização, quanto à herança [35].

2.3.1.6 *Framework*

Um *framework*, na terminologia de objetos, é um esqueleto de implementação de uma aplicação ou subsistema de aplicação de um domínio de problema particular [36, 37]. Ele é composto de classes abstratas e concretas, e é previsto um modelo de interação ou colaboração entre as instâncias das classes definidas. Utiliza-se um *framework* configurando-se objetos através de passagem de parâmetros ou conectando-se classes concretas, a partir das *lacunas* das classes abstratas, para 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 [37, 36, 25].

Para que um *framework* seja realmente usado como um arcabouço a partir do qual projetistas possam obter suas próprias aplicações, ele deve ser simples, consistente e funcional. Neste sentido, é quase obrigatório que se relacionem *frameworks* a *design patterns*. Conforme lembrado em [35], "um *framework* pode ser visto como um conjunto de *design patterns* especializados e interrelacionados para implementar a estrutura básica de determinado domínio de aplicação".

Requisitos atendidos

Uma das vantagens prometidas pela tecnologia de objetos é a 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 tais mecanismos, o mais conhecido é a herança. Entretanto, não existe um consenso a respeito da implementação de herança nos compiladores, nem há uniformidade com relação à semântica associada. Uma possível solução para o problema é a restrição do uso da herança à herança de interface, como sugerido em CORBA. Além disso, o emprego de herança e outros mecanismos de extensão não significa reutilização, e linguagens orientadas a objetos oferecem, normalmente, apenas um deles. O planejamento de objetos para compor uma aplicação deve ser criterioso, a fim de permitir a reutilização e facilidade de interação com outros objetos [38].

Com relação à evolução dinâmica, é desejável que novas classes possam ser concebidas, compiladas e adicionadas livremente em uma aplicação [23] durante sua execução. A tecnologia de objetos dinâmicos (*dynamic objects*) [5] pode solucionar parcialmente este problema. Se esta facilidade não estiver disponível, a alternativa viável é a parar a 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 à afirmação de que "… herança é incompatível com distribuição …" em [3]. 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 levantados de falta de verificação de tipos 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 [4, 7, 13, 23].

2.3.2 Sistemas Baseados em Componentes e *Middleware*

2.3.2.1 Conceitos básicos

Segundo [39], "um componente de *software* é um empacotamento físico e independente de *software* executável, com uma interface bem definida e publicada". A especificação da versão 1.3 da UML [40] complementa este conceito como "uma parte física, auto-contida e cambiável de um sistema, que empacota e provê a implementação de um conjunto de interfaces. Um componente representa um pedaço físico de implementação de um sistema, incluindo código de *software* (fonte, binário ou executável) ou equivalentes tais como *scripts* ou arquivos de comandos".

Componentes são diferenciados de classes por algumas características. Componentes são unidades físicas de um sistema, e classes não são. Em geral, um componente tende a ter maior granularidade que uma classe, e é mais estritamente baseado na descrição de sua interface do que as classes. Classes são modeladas durante a análise, e componentes durante a implementação. Bertrand Meyer [41] conclui ainda que componentes são uma extensão natural para a idéia de objetos, mas não precisam necessariamente ser desenvolvidos com linguagens orientadas a objetos.

A idéia é que um componente ofereça um conjunto de serviços coerentes, e tenha a possibilidade de ser integrado a outros componentes para compor uma aplicação. Componentes podem ser trocados, substituídos ou atualizados durante a concepção de uma aplicação, e, em alguns casos, durante sua execução.

Um bom componente é avaliado em sua adequação para ser usado em sistemas distribuídos, e também em sua modularidade e independência de linguagem e plataforma. Componentes deveriam poder ser desenvolvidos por empresas diferentes e depois integrados. O projetista da aplicação poderia "buscar na prateleira" (*off-the-shelf* - OTS) os componentes necessários para desenvolver sua aplicação. A dependência de um componente em relação a outros componentes deve ser genérica. Assim, qualquer implementação destes componentes consegue satisfazer tal dependência e torna possível a composição.

A tecnologia BC procura isolar alguns problemas encontrados na utilização direta de objetos através de um envoltório (*wrapping*) que permite a configuração dos componentes de forma mais sistemática. Geralmente, um componente implementa um conjunto de interfaces padronizadas (pelo menos em relação a outros componentes do mesmo fabricante) para a configuração de propriedades, introspecção (verificação do estado interno e propriedades válidas), para indicar persistência, entre outras características. Este conjunto de interfaces, que impõe restrições adicionais (ao mesmo tempo que oferece características úteis) ao objeto original, é que dá ao mesmo o *status* de um componente.

2.3.2.2 *Component Bus* e *Middleware*

O conceito, elegante, da tecnologia BC pode esconder a necessidade de suporte para que os componentes possam (i) ser integrados e compostos; (ii) ser referenciados e interagir e (iii) ser carregados para executar. Considerando-se que os componentes possam ser implementados em linguagens, e por fabricantes diferentes, este suporte pode se tornar complexo [42].

Uma abordagem simples (e estática) é a utilização de um padrão para referência e chamada de procedimentos, ao qual o código gerado a partir de uma linguagem de programação pode aderir. Assim, componentes escritos em linguagens diferentes geram módulos de carga binariamente compatíveis e podem ser integrados (*link-editados*) em uma aplicação executável, como se dá com bibliotecas de procedimentos ou classes. VisualBasic e Delphi são exemplos de linguagens que empregam a programação por componentes desta forma.

Em uma abordagem mais flexível, existem sistemas BC que contêm padrões

para a definição e publicação das interfaces dos componentes, e mecanismos padronizados para a passagem de dados entre componentes [43]. Geralmente, estes sistemas admitem aplicações distribuídas e, portanto, os componentes também podem executar em máquinas diferentes. Para isso, regras ou protocolos de interação precisam ser definidos e devem ter a aderência dos componentes. São oferecidos também serviços de apoio, tais como diretório de nomes ou repositório de interfaces, além de suporte para a execução e comunicação remota destes componentes. Tal conjunto de serviços e suporte tem sido chamado genericamente de *software bus* (como se os componentes de *software* pudessem interagir através de um barramento, similar, em abstração, aos barramentos de *hardware*) [44] ou *middleware* (uma camada de serviços que fica entre o *software* e o *hardware*). Assim, a carga, a execução e a interação entre os componentes são intermediadas pelo *middleware*.

Exemplos de *middleware* para componentes distribuídos são o DCOM (*Distributed Component Object Model*) da Microsoft [45, 46], o CORBA (*Common Object Request Brocker Architecture*), um modelo definido pela OMG (*Object Managemet Group*) [8] e o EJB (*Enterprise JavaBeans*), uma infra-estrutura para a execução de componentes definida pela *Sun Microsystems* [47].

De uma forma geral, o suporte para comunicação utilizado nestes casos, que também pode ser usado diretamente como *middleware*, é uma variação da idéia de chamadas a procedimentos remotos (*Remote Procedure Calls* - RPC), originalmente proposta por Birrel e Nelson [48]. RPC preconiza a definição da interface dos serviços a serem oferecidos pelo servidor e utilizados pelo cliente. A partir desta interface, partes de código relacionadas com a comunicação (chamadas de *stubs*) são geradas automaticamente, e o serviço a ser oferecido é publicado. Estes procedimentos oferecem alguma transparência para o programador da aplicação. Por exemplo, EJB utiliza o RMI (*Remote Method Invocation*) de Java, e DCOM, uma implementação POSIX de RPC [49].

As funções dos *stubs* em todas as tecnologias é essencialmente a mesma: serializar e desserializar argumentos, estabelecer canais de comunicação (na maior parte dos casos usando o protocolo TCP/IP através do mecanismo do *sockets*), e resolver diferenças de representação de tipos. Entretanto, a semântica da interação entre as partes cliente e servidor apresenta particularidades em cada caso.

Comentários

Embora os modelos DCOM, CORBA e EJB apresentem características em comum, como, por exemplo, serviços para comunicação entre componentes, a escolha de um ou outro *middleware* é dependente de outros fatores, estratégicos para o usuário, pois existem muitas características particulares:

- DCOM é associada e dependente da plataforma Windows, mesmo com os esforços para integração com outros *middlewares*.
- A especificação de CORBA é independente de plataforma e linguagem, mas deve haver uma implementação para cada linguagem e plataforma de operação (existem atualmente várias implementações para algumas linguagens de programação, como C, C++ e Java, e plataformas de operação como Windows e Unix).
- EJB é concebido para Java, com o suporte da Máquina Virtual Java (Java Virtual Machine, JVM). A independência de plataforma se dá na medida em que exista uma implementação da JVM para a plataforma de operação desejada.

Uma alternativa no sentido de integrar os vários modelos de componentes e *middlewares* talvez seja a utilização de XML (*Extensible Markup Language*) [50] em conjunto com uma definição padronizada e genérica de modelo de componentes. XML pode ser usada como linguagem neutra para definir as interfaces dos componentes e para conter as informações trocadas durante as interações dos componentes. A parte visível do componente seria dependente apenas de XML.

Requisitos atendidos

Em princípio, a utilização de BC leva ao reuso e permite a evolução das aplicações. Componentes que ofereçam um conjunto bem estruturado de serviços podem ser reusados em várias aplicações. Aplicações construídas a partir de componentes, com uma granularidade coerente, podem sofrer modificações localizadas e, desejavelmente, sem efeitos colaterais. Entretanto, BC possui problemas similares à tecnologia de objetos: suporte para evolução dinâmica limitado e a reusabilidade dependente da disciplina de quem projeta os componentes. Por exemplo, CORBA inclui um mecanismo para permitir a interceptação e manipulação de mensagens trocadas entre os objetos. Este mecanismo, chamado *interceptor*, pode ser utilizado pelas

aplicações através de uma API específica, mas que exige conhecimento adicional do programador da aplicação.

2.3.3 Programação de Meta-Nível

A Programação de Meta-Nível (*Meta-Level Programming*), PM-N, propõe uma abordagem para a concepção de *software* em que os elementos de *software* são arranjados em diferentes níveis de interesse. PM-N facilita a separação dos aspectos funcionais básicos da aplicação, dos aspectos relacionados a requisitos não-funcionais da mesma. A programação destes requisitos particulares, desejavelmente ortogonais aos aspectos funcionais, são concentrados em pedaços explícitos de código, que definem uma arquitetura dita de meta-nível da aplicação. Reflexão, filtros de composição e programação orientada a aspectos são propostas que empregam os conceitos de PM-N, sendo a reflexão uma técnica mais geral, apresentada a seguir, e as outras duas, casos particulares, apresentadas no capítulo 3.

2.3.3.1 Reflexão

O termo reflexão dentro de um contexto genérico tem dois sentidos. O primeiro, com o significado de introspecção ou meditação, e o segundo com a acepção de refletir um evento, como a imagem em um espelho, por exemplo. Estes dois sentidos podem ser aplicados à utilização do conceito de reflexão em 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 [51].

Quando a reflexão é utilizada, os sistemas são divididos em um nível-base, onde a computação da função básica é realizada, e em um meta-nível, onde os aspectos não-funcionais são implementados [52, 53].

Existem duas abordagens para a reflexão: estrutural e computacional (ou comportamental) [52]. Na reflexão estrutural (associada 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, por exemplo, oferece reflexão estrutural (seção 3.2.2.1). A reflexão computacional

(associada ao sentido do "espelho") permite que o comportamento das funções básicas de um programa seja monitorado e controlado 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 interceptadas e desviadas para o meta-nível. No meta-nível é possível acrescentar-se alguma computação a estes eventos. Por exemplo, a escrita em um *buffer* genérico pode ser controlada no meta-nível, a fim de que não seja ultrapassada uma certa capacidade de armazenamento, ou para que não se sobreponham novos elementos aos 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, possui 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 2.1). O meta-componente pode ter acesso ao ambiente de execução, permitindo ajustes no componente base em reação a eventos de contexto de baixo nível (por exemplo, um *tick* de relógio).

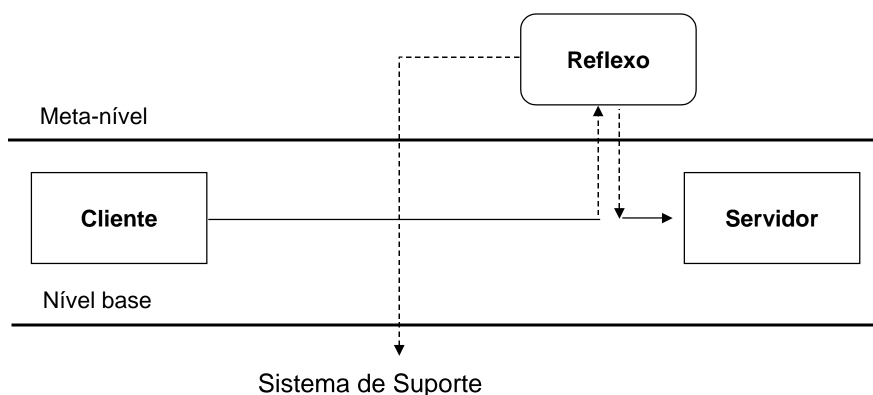


Figura 2.1 - Reflexão como interposição de código

Em certas aplicações cliente/servidor pode ser necessário que tanto o módulo cliente como o módulo servidor sejam refletidos no meta-nível. 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. Deste modo, uma aplicação que solicite uma leitura ou escrita em um arquivo remoto, opera 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. Tal serviço do sistema operacional estaria no meta-nível. O sistema de arquivos da outra estação, por sua vez, recebe o pedido e encaminha o mesmo para o seu próprio sistema local de arquivos. A arquitetura do exemplo citado é ilustrada na figura 2.2.

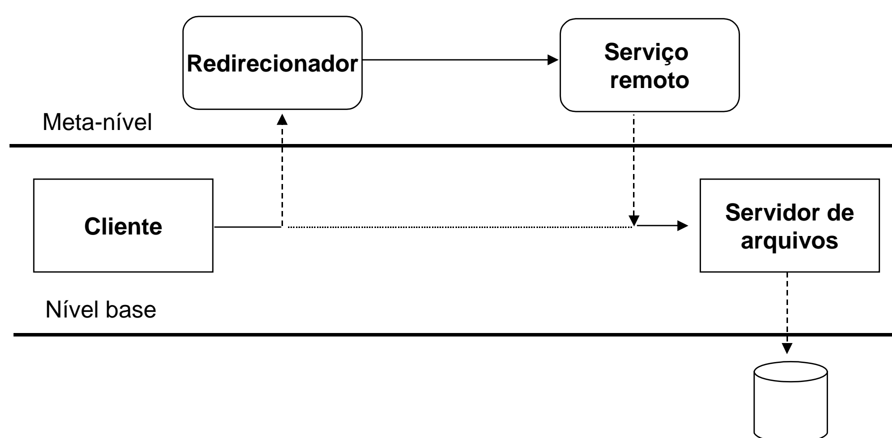


Figura 2.2 - Arquitetura de meta-nível de um sistema de arquivos distribuído

Entre os exemplos de domínios de aplicação que podem se beneficiar da abordagem de reflexão, encontram-se: tolerância a falhas, sistemas de tempo-real, sincronização de sistemas concorrentes e esquemas para tratamento de exceções [54, 55, 56]. Alguns trabalhos na área de sistemas operacionais incluem a reflexão como mecanismo para permitir a extensão e ajuste de funções [57, 58, 59, 11, 60, 61].

Os conceitos de reflexão computacional e estrutural vêm sendo implementados no nível de linguagem de programação, principalmente em linguagens orientadas a objeto. Algumas implementações experimentais, tais como C++[9 e 62], CLOS [10] e MetaJava [63], estão baseadas em extensões especiais de compiladores de linguagens orientadas a objeto. A maioria delas oferece mecanismos para que o fluxo da invocação de um método seja interceptado em pontos-chave (na entrada ou saída, por exemplo) e manipulado. A combinação de características da tecnologia de objetos, mencionadas na seção 2.3.1, e a flexibilidade provida pela reflexão computacional será explorada na próxima subseção.

2.3.3.1.2 Tecnologia de Objetos e Reflexão

Em linguagens orientadas a objeto que oferecem características reflexivas, uma classe, chamada classe-base, pode ser associada a uma meta-classe. A sintaxe da linguagem deve oferecer um meio de se estabelecer uma conexão entre a classe-base e a meta-classe. As regras e sintaxe que descrevem a conexão entre classe-base e meta-classe são chamadas de Protocolos de Meta-Objeto (*Meta-Object Protocol* -MOP).

Na reflexão estrutural, normalmente a linguagem utilizada (em alguns casos, com o suporte de um ambiente de execução) oferece um conjunto de métodos que podem ser usados para obter informações sobre os elementos do próprio programa, como por exemplo: o número de objetos instanciados, a assinatura dos métodos de um objeto específico, o conteúdo de determinada variável de classe, etc. Esta forma de reflexão está ilustrada na figura 2.3(a). De posse destas informações, o meta-objeto pode tomar decisões para alterar a estrutura da própria aplicação, quando permitido pela linguagem de programação. Neste caso, nível-base e meta-nível estão ligados por uma relação de causalidade: quando mudanças forem realizadas no meta-nível, estas são refletidas para o nível-base de forma síncrona.

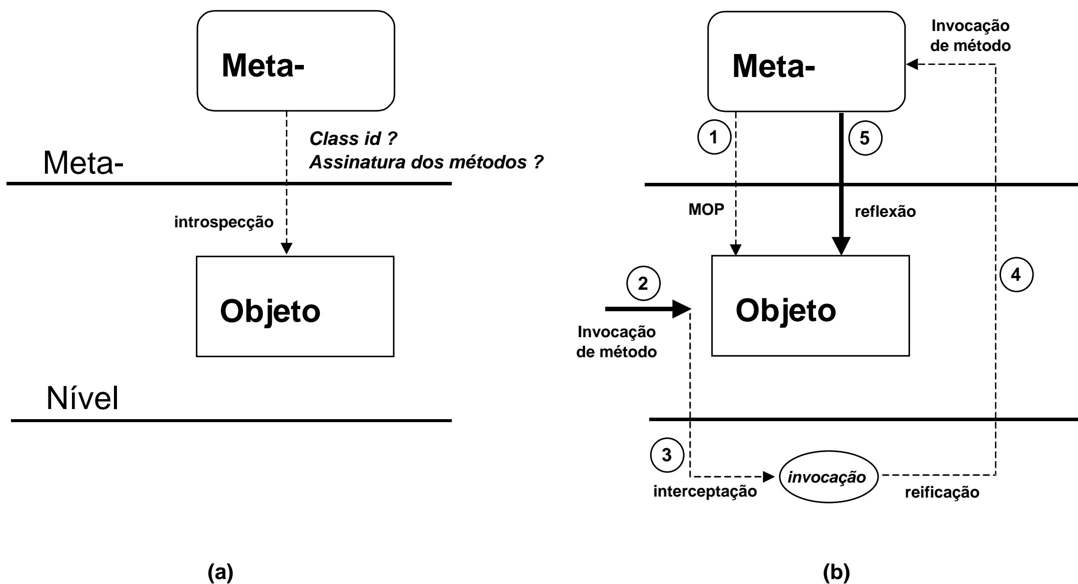


Figura 2.3 - (a) reflexão estrutural e (b) reflexão computacional

Na reflexão computacional, esta conexão é normalmente expressa por construções de reflexão e reificação. Acompanhando a figura 2.3 (b), (1) através do MOP selecionam-se os eventos a serem interceptados e o modo como os mesmos serão tratados pela meta-classe. Quando (2) um evento chega à classe-base (a invocação de

um método, por exemplo), o mesmo é interceptado (3) e *reificado*[2]. A *reificação* (4) materializa o evento redirecionado, tornando disponíveis suas informações relevantes (como variáveis ou argumentos, por exemplo) e invocando um método específico na meta-classe, associado pela construção de reflexão. Na meta-classe as informações do evento original podem ser manipuladas, e o fluxo pode ser reencaminhado (refletido) para a classe-base (5). Neste contexto, a reflexão é dita computacional, em virtude de sua atuação no comportamento e na computação do objeto-base.

2.3.3.1.3 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 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++ [9], um pré-processador de C++ que oferece um MOP. Advoga-se que os conceitos de reflexão são utilizados para a elaboração 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 de que as relações entre objetos-base e meta-objetos não podem ser alteradas durante a execução.

---

[2] O termo reificação poderia ser traduzido como **materialização** ou **exposição**. Por reificação quer-se dizer que alguma parte ou evento associado à classe-base pode ser referenciado e manipulado no meta-objeto. Por exemplo, um evento pode ser materializado em um objeto. Isso vale para a invocação de um método, a qual pode ser interceptada pelo meta-objeto, para uma variável, no caso de reflexão computacional, ou mesmo para a própria estrutura da classe-base, no caso de reflexão estrutural.

Entretanto, se considerarmos 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, é viável conceber-se uma biblioteca de meta-objetos com implementações das extensões mais populares e críticas em desempenho [64], e, sempre que necessário, utilizar estes meta-objetos no nível da programação.

A reflexão em tempo de execução permite que as relações entre objetos-base e meta-objetos sejam estabelecidas durante a execução do programa. Na maioria das implementações, 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 [65, 11, 66]. Um suporte mínimo deve permitir também que eventos, como a invocação a um método, sejam interceptados e manipulados no meta-nível antes de serem repassados ao nível-base. Observa-se que este suporte apresenta um *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*) [9, 67, 10]. 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, no caso, se devem à transparência na inclusão de novas funcionalidades agregadas por meta-objetos, e ao conforto sintático da programação. No trecho de código seguinte [9], pode ser observado que no programa sem reflexão, o objeto *Node* precisa herdar as características da classe *PersistentObjetct* para se tornar 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, um objeto que ofereça serviços genéricos, pode tê-los especializados ou diferenciados, através do uso adequado de meta-objetos. Características como a transparência de localização, segurança, tempo-real ou tolerância a falhas [54, 52, 68, 55], podem ser adicionadas por reflexão (figura 2.4). Neste sentido, entendemos que a reflexão amplia as possibilidades de reutilização e extensão das linguagens orientadas a objeto.

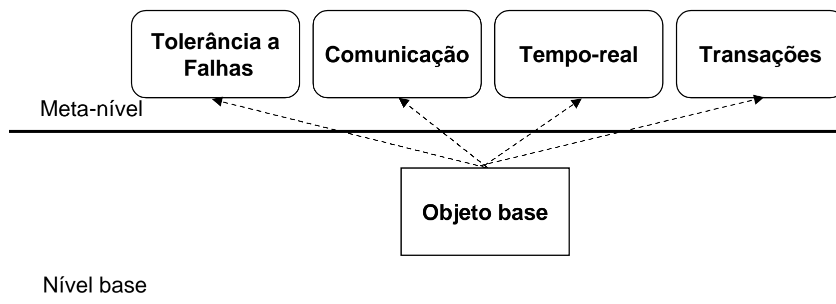


Figura 2.4 - Utilizando reflexão para adicionar características não-funcionais

Requisitos atendidos

Dentre os requisitos para o projeto de aplicações distribuídas, a reflexão atende melhor à 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 de 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, como já discutido, podem ser adicionados por reflexão de objetos-base, contanto que a linguagem de programação em uso tenha características adequadas para tal, e meta-objetos adequados estejam disponíveis. Por outro lado, a

seleção dinâmica de meta-objetos, a ligação dinâmica entre objetos-base e meta-objetos, ou a evolução dinâmica de aplicações baseadas em meta-arquiteturas, são difíceis de se tratar [67]. A maioria dos MOPs têm que ser programados em um estilo estático, em que as classes ou objetos a serem refletidos e seus respectivos meta-objetos sejam 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 [62, 69].

2.3.4 Arquiteturas de *Software* e Programação por Configuração

Uma arquitetura de *software* consiste na composição ou configuração de um sistema em termos de um conjunto de *módulos* (componentes), que desejavelmente encapsulam a computação funcional da aplicação, e um conjunto de *conectores*, que primordialmente descrevem como os módulos são ligados e como eles interagem dentro da arquitetura [70, 17, 71, 72, 73].

Linguagens para descrição de arquiteturas de *software* (*Architecture Description Languages*), ADL, são concebidas para especificação de arquiteturas de *software* de forma textual ou gráfica [75, 76]. Utilizando uma ADL, um projetista pode especificar a composição funcional de um sistema, selecionando um conjunto de módulos, e associar a estes módulos estilos ou contratos de interação particulares (no contexto de comunicação de dados seriam os protocolos), através de conectores. A esta atividade chamamos de programação por configuração (*Configuration Programming*), PC, ou simplesmente configuração.

2.3.4.1 Conceitos básicos

O princípio básico de PC supõe a existência de componentes[3] de *software* que serão selecionados e interligados para construir uma aplicação com características próprias. Em PC, a programação dos componentes de *software* e a estrutura da aplicação são considerados aspectos diferentes [17, 74, 77]. Ao se estabelecer uma configuração, é possível selecionarem-se atributos específicos e formas de interligação

[3] O termo **componente** adotado no contexto de AS/PC é genérico. Para os nossos propósitos um sistema ou aplicação são compostos por componentes. Um componente é um elemento que pode interagir com outros componentes.

dentre um conjunto de opções disponíveis para cada componente (figura 2.5). Adicionalmente, novos componentes e formas de interligação podem ser incluídas.

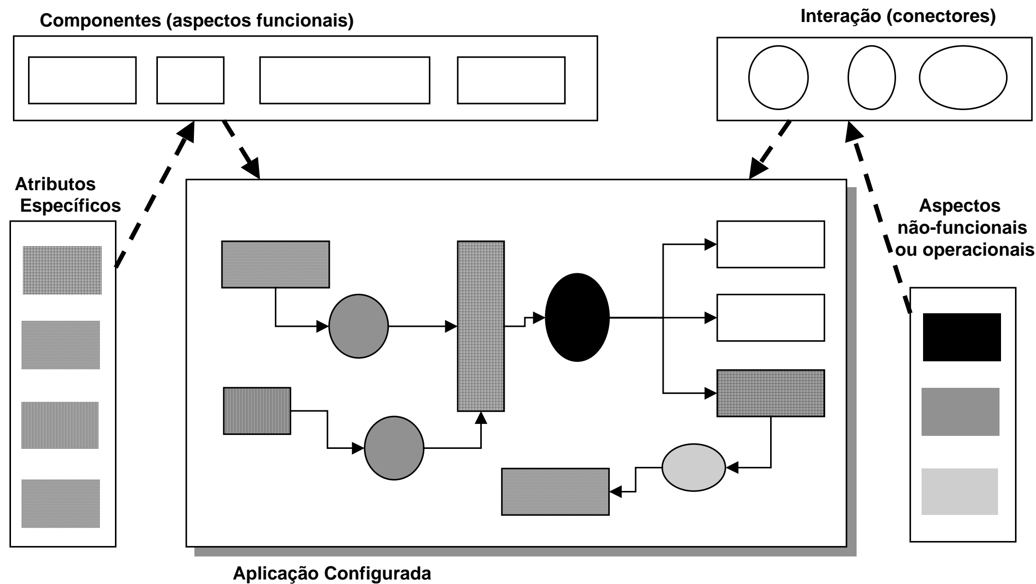


Figura 2.5 - Suporte para a configuração de aplicações distribuídas

A configuração de uma aplicação é descrita em uma linguagem. Através da mesma 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 interconexão de módulos (*module interconnection languages*) (MIL), linguagem de programação por configuração (*configuration programming languages*) (CPL), linguagem de descrição de arquitetura (*architecture description language*) (ADL) [15], ou linguagem de composição (*composition language*) (CL) [78], são encontrados na literatura. Através destas linguagens descreve-se a configuração que espelha a estrutura desejada para a arquitetura das aplicações. Tais configurações podem ser mapeadas para uma linguagem de programação tradicional ou para um forma executável. A utilização de ferramentas gráficas [79], como interfaces *front-end* para as linguagens de configuração, permite que os sistemas sejam descritos de uma forma visual.

O nível de abstração e a expressividade das ADLs tornam as mesmas adequadas para a descrição da arquitetura de *software* de aplicações complexas, o que às vezes é chamado de *programming in-the-large* [80]. ADLs possuem características interessantes para a concepção destas aplicações, tais como:

- permite a composição e construção hierárquica de módulos, e da própria aplicação [75];
- facilita a reusabilidade de módulos que tenham sido concebidos e depurados para outra aplicação [26];
- pode realizar a geração automática de código a partir da configuração descrita [81];
- facilita verificação de propriedades das aplicações descritas [71].

A execução de uma aplicação pode ser tratada de duas formas: (i) estaticamente - quando a descrição de uma configuração é utilizada para gerar uma aplicação que, depois de configurada, executará com o suporte de sistema operacional tradicional, ou (ii) dinamicamente - quando a mesma executará sobre o suporte de um ambiente executivo (como descrito em [82]), o qual permite intervenção de um agente de configuração, externo, através de comandos específicos de configuração. No escopo deste trabalho, consideramos o segundo caso, em que agentes de configuração podem alterar a arquitetura da aplicação, adicionando novos módulos, removendo módulos existentes, alterando a conexão de módulos ou substituindo módulos instanciados por novas versões dos mesmos.

Elementos utilizados para a interligação dos módulos de uma aplicação são chamados genericamente de conectores [15]. O conector é uma abstração utilizada no nível de arquitetura de *software* com dois objetivos: expressar a interação de módulos e oferecer ao programador da arquitetura acesso aos mecanismos nativos do sistema operacional de maneira uniforme. Por exemplo, usando-se o conector, não existe a necessidade de cuidados com detalhes específicos de 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 ou impõe. Podemos ter conectores que implementem estilos de interação diferentes, e neste caso selecionar o mais conveniente para uma dada situação. A figura 2.6 ilustra esta possibilidade.

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 [15, 16]. Na figura 2.6, isto é esquematizado com a possibilidade de selecionar um dentre os

diversos conectores, com diferentes atributos, para atender a um aspecto de comunicação. Além disso, outras características podem ser associadas à ação de se interligarem módulos através de um conector:

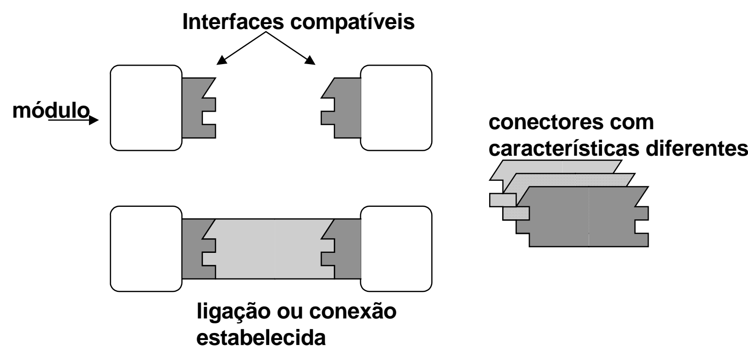


Figura 2.6 - Conexão de dois módulos

- 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 um agente de configuração. Módulos cujas interfaces possuem assinaturas casadas podem ser automaticamente interconectados. Por outro lado, problemas de incompatibilidade de assinaturas podem ser superados com a utilização um conector que implemente uma ponte [6] capaz de fazer a tradução adequada.
- Conector é um elemento onde potencialmente podem ser controlados aspectos não-funcionais da aplicação que envolvam a interação entre componentes. Por exemplo, no caso de uma conexão de módulos que residam em um mesmo nó, a interação pode, em um nível maior de detalhe, ser feita por mecanismos de invocações a métodos, *pipes*, 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.

O conceito de arquitetura de *software* pode ser associado a formalismos em vários níveis [73]. Por exemplo, pode-se definir uma teoria para arquiteturas de *software* [83, 84], clarificando seus conceitos e provendo regras para determinar se uma

dada arquitetura está formada corretamente [85, 86]. O uso de conectores facilita a verificação formal de propriedades de uma arquitetura, pois as informações sobre a topologia e as aspectos não-funcionais ficam disponíveis. Em um nível maior de abstração, dentro dos ambientes de configuração e ADL, os conectores também podem ter o papel de caracterizar o estilo da arquitetura de *software* utilizada pela aplicação, como proposto em [72]. Por exemplo, 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 da configuração de uma aplicação à determinado estilo de arquitetura de *software* [87, 88].

Um dos desafios conhecidos na realização de um ambiente de execução, considerando arquiteturas de *software*, é a dificuldade de se mapearem as descrições do nível de arquitetura em uma implementação com estruturas e contornos bem definidos [15]. A implementação de conectores, e a interface entre módulo e conector podem se apresentar difusas e entrelaçadas com outras partes do código que não digam respeito à interação [89, 31]. Este problema pode ser contornado com a geração de esqueletos de código por ferramentas associadas à ADL, ou pela integração de mecanismos de comunicação dentro das linguagens de programação. Entretanto, um esquema flexível de mapeamento pode evitar a dependência de uma linguagem de programação e pode facilitar a adoção de ambientes de execução heterogêneos.

Comentários

A utilização da tecnologia de AS/PC não limita a linguagem de programação com a qual são concebidos os módulos básicos. É possível, por exemplo, combinar a tecnologia de objetos e AS/PC. Um dos problemas a serem resolvidos é como mapear a abstração de módulos e conectores, de AS/PC em classes ou objetos de uma linguagem de programação orientada a objetos. Para o módulo, uma solução possível é seu mapeamento em uma classe. A interface do módulo é mapeada em métodos da classe. Os conectores, considerados como elementos de primeira classe, possivelmente (mas não necessariamente) também podem ser mapeados em objetos. Nesta solução um objeto, mapeado a partir de um módulo, deve ter uma interface compatível com o objeto implementando o conector. Quando dois objetos precisam ser ligados, um 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.

Requisitos atendidos

AS/PC satisfaz grande parte dos 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 AS/PC e é imposta pelo encapsulamento e pela restrição de todas as interações entre componentes serem 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 em AS/PC. 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 AS/PC oferece a separação de interesses. Um projetista também pode usar módulos especializados, que implementem características não-funcionais de uma aplicação, não relacionadas com a interação dos módulos, para adicionar características necessárias através de composição.

Em um ambiente com suporte à configuração, a evolução dinâmica é atendida com a possibilidade de se agregarem mais elementos ou reconfigurarem elementos existentes, sem a paralisação de todo o sistema. Primitivas de reconfiguração e o suporte adequado de um ambiente de execução devem impor a consistência para a nova configuração construída dinamicamente. Neste contexto, uma seqüência de reconfiguração pode ser iniciada por um agente externo (como por exemplo, um módulo gerenciador de configuração) ou por um módulo que esteja participando da aplicação [14, 90].

Um dos pontos negativos de AS/PC, a despeito de todas as suas vantagens, é o fator disponibilidade. Até o presente momento, não existe um padrão mais aceito, uma ADL mais utilizada ou ambiente de configuração padronizado. Este problema vem sendo superado com a crescente adoção do CORBA da OMG como ambiente para a

implantação dos serviços de suporte de AS/PC. 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 [64, 91, 92, 93, 94].

2.4 Comparando as tecnologias

A tabela 2.1 apresenta uma comparação das tecnologias avaliadas, baseada nas seguintes características:

- Separação explícita de interesses - nível-base e meta: se a tecnologia impõe ou incentiva explicitamente que o *software* seja desenvolvido considerando-se aspectos diferentes separadamente;
- Separação de interesses - em tempo de execução: se os interesses diferentes da aplicação desenvolvida ainda se mantêm localizados e referenciáveis em tempo de execução ou, caso negativo, se os interesses se misturam para formar uma única unidade de execução;
- Representação explícita da estrutura do *software*: se um dos passos necessários para a concepção do *software* é a descrição de sua estrutura, e se esta representação pode ser consultada durante a execução, ou, caso contrário, se esta estrutura fica "escondida" dentro do código;
- Composição: se módulos elementares de *software* podem ser explicitamente compostos para formar uma aplicação, ou se a composição destes elementos se dá apenas por referência dentro do código;
- Suporte explícito para atividades de reconfiguração: se partes da aplicação ou sua estrutura podem ser adaptadas ou modificadas durante sua execução;
- Mecanismos para descrever aspectos não-funcionais: se existe alguma forma para se descreverem aspectos ou propriedades não-funcionais como a comunicação remota, tolerância a falhas, segurança, etc.;
- Mecanismos para programar aspectos não-funcionais: se existem mecanismos para se programarem aspectos não-funcionais (independentemente de existirem mecanismos para descrevê-los).

Tabela 2.1 - Comparação das tecnologias (características técnicas)

	PM-N e reflexão	AS / PC	BC
Separação explícita de interesses (nível-base e meta)	✓	R-RIO	
Separação de interesses (em tempo de execução)		R-RIO	
Representação explícita da estrutura do *software*		✓	
Composição	Herança	✓	✓
Suporte explícito para atividades de reconfiguração		✓	
Mecanismos para descrever aspectos não-funcionais		R-RIO	
Mecanismos para programar aspectos não-funcionais	✓	✓	✓

As características, onde aparece "R-RIO", não são contempladas em outras propostas, e identificam os pontos em que nossa tese oferece contribuição.

A tecnologia PM-N provê explicitamente a separação de interesses funcionais e não funcionais de uma aplicação em níveis diferentes. Em AS/PC, a separação de interesses maior se dá entre a estrutura da aplicação e seus componentes, que devem ser tratados em momentos diferentes. Algumas propostas de AS/PC ainda consideram a comunicação como um aspecto não-funcional separado dos aspectos básicos da aplicação, mas não são a regra. Não se encontram tecnologias que mantenham explicitamente a separação de interesses durante a execução das aplicações. Geralmente considera-se que, uma vez concebido o *software*, suas características só serão alteradas se uma nova versão do programa for feita.

A tecnologia AS/PC com ADL possui a capacidade de descrever e gerenciar explicitamente a composição e a estrutura de sistemas, independentemente de aspectos de programação da aplicação, ajudando, portanto, a separação de interesses no domínio da programação por configuração. Por outro lado, a maioria das abordagens de PM-N e BC não oferecem mecanismos e conceitos claros para suportar adaptação dinâmica, pois a estrutura das aplicações está implícita no código, levando a soluções *ad-hoc.*

Tabela 2.2 - Comparação das tecnologias (requisitos de engenharia de *software*)

Modelo	BC	PM-N	ADL/PC
reusabilidade	Componentes podem ser reutilizados. Reutilização requer planejamento, mas pode ser maximizada com o uso de padrões de projeto (*design patterns*) e padrões de *middleware*.	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 interfaces e estilos de interação. Arquiteturas diferentes podem ser construídas a partir de uma arquitetura básica e agregando-se outros módulos e conectores.
separação de interesses	Não atendido explicitamente	Os conceitos de objeto-base e meta-objeto 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 não-funcionais relacionados com a interação de módulos.
evolução dinâmica	Extensibilidade pode ser obtida por herança de componentes básicos, se considerada uma linguagem orientada a objetos, ou composição. Extensibilidade dinâmica é de difícil obtenção. O suporte à ligação retardada e dinâmica e o uso de *middlewares*, como suporte à execução, podem 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 têm sido propostos para permitir maior dinamismo [69]. Extensibilidade dinâmica de objetos-base deve ser suportada por um ambiente reflexivo especializado.	Obtida pela capacidade de (re)configuração da arquitetura da aplicação, módulos e conectores antes e, em algumas implementações, durante a operação.
abrangência	Sistemas e bibliotecas de componentes são amplamente disponíveis. Falta de padronização nos mecanismos de extensão e *middleware*.	Linguagens de programação ou ambientes de execução com características reflexivas ainda não estão amplamente disponíveis. Não existem MOPs aceitos como padrão.	ADLs e ambientes de configuração estão disponíveis. Não existe uma ADL ou ambiente de suporte à configuração aceitos como padrão.

A capacidade de gerar *software* que atenda a requisitos não-funcionais também está ligada à facilidade para se descreverem estes requisitos. Em AS/PC tal característica não é comum, embora em alguns casos, aspectos específicos, como a comunicação, admitam configuração através de anotações nas ADLs. Por outro lado, as tecnologias avaliadas oferecem mecanismos para programação de requisitos não-funcionais. CORBA, por exemplo, oferece o mecanismo de *interceptors*, no qual pode-se incluir código adicional à funcionalidade básica dos componentes. Em PM-N, no meta-nível, também é possível acrescentar código de aspectos não-funcionais, seja em

meta-objetos (reflexão), filtros de composição (capítulo 3) ou qualquer outro mecanismo específico. Já em AS/PC, os conectores podem ser utilizados com esta finalidade.

A tabela 2.2 sumariza como cada tecnologia mencionada, atende aos requisitos levantados na seção 2.2.

Reusabilidade. As tecnologias selecionadas contemplam a reusabilidade, mas em níveis diferentes. BC pode basear-se em bibliotecas para oferecer componentes úteis. Criam-se componentes especializados por herança ou composição, a partir destes componentes básicos, e por ajustes que o próprio componente deve permitir. A reutilização pode ser maximizada com a adoção das técnicas de padrão de projeto ou *frameworks* [35] em que soluções completas, baseadas em componentes, podem ser reutilizadas.

PM-N permite que objetos-base e meta-objetos sejam reutilizados. Várias aplicações podem se beneficiar da disponibilidade de bibliotecas de meta-objetos e atender a requisitos não-funcionais mais rapidamente. Uma outra possibilidade é a reutilização de módulos de aplicações inteiras no nível-base e a adaptação dos componentes de meta-nível, de forma que o comportamento do sistema original seja adequado para uma nova situação.

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

Separação de Interesses. BC não oferece nenhum mecanismo para atender ao requisito de separação de interesses explicitamente. Mesmo assim, componentes podem ser desenvolvidos para implementar aspectos funcionais e não-funcionais específicos. A modularidade dos componentes depende de uma disciplina de projeto. Da mesma forma, a topologia de uma aplicação é descrita no próprio código. Entretanto, alguns *middlewares* de suporte a BC podem oferecer mecanismos reflexivos para se obter informações sobre a mesma.

PM-N, por sua vez, preconiza a separação de interesses como característica básica. Linguagens de programação reflexivas, por exemplo, permitem uma clara distinção entre o nível de programação de funções e o nível não-funcional. Em cada um destes níveis é possível concentrar preocupações com aspectos diferentes da aplicação.

AS/PC atende à separação de interesses, permitindo que a especificação do comportamento, estilos de interconexão dos módulos e a topologia da aplicação sejam planejados separadamente. Requisitos não-funcionais e operacionais podem ser adicionados externamente, no domínio da configuração, pela composição de componentes e pela seleção adequada do estilo de interação entre estes componentes.

Evolução Dinâmica. De acordo com o que foi 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 em BC com o uso da herança (ou mecanismos similares), se linguagens orientadas a objetos estiverem sendo utilizadas. Objetos-base podem ser estendidos para especializar ou generalizar seus comportamentos. 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 à extensão, mas abordagens diferentes: a herança permite a criação de novas classes, e a reflexão oferece a adaptação de classes com granularidade fina (em [95] a herança é modelada por reflexão e algumas considerações sobre sua expressividade são apresentadas). A abordagem de AS/PC é mais ampla no que diz respeito à 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, que interligam estes módulos, através de comandos de configuração.

Considerando-se isoladamente BC, a obtenção de evolução dinâmica não é trivial. A inclusão ou remoção de componentes em uma aplicação pode ser realizada, se a tecnologia de objetos dinâmicos for utilizada [5]. Entretanto, permitir que novas classes e relações de herança sejam definidas durante a execução de um componente é mais difícil. Problemas tais como ligação dinâmica e retardada (*late binding*) de

componentes e a manutenção da consistência e referências são mais evidentes nestes casos, e agravados em sistemas distribuídos.

Os mesmos problemas identificados em BC podem ocorrer em PM-N. Na maioria dos MOPs disponíveis, os meta-objetos são definidos estaticamente. As classes ou objetos a serem refletidos, e as partes que 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 ocorrer dinamicamente, os sistemas construídos com a tecnologia de objetos e reflexão computacional teriam condição de evoluir dinamicamente também.

Sistemas em AS/PC 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 aplicação, assegurando que as interações dos elementos a serem reconfigurados sejam graciosamente congeladas, levando a uma evolução suave da aplicação [96, 90, 97].

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. Assim, com relação a sistemas distribuídos, é difícil a criação de uma nova classe derivada de uma classe-base definida em uma linguagem diferente, em que a classe-base e a classe derivada podem ainda podem estar em nós diferentes. Linguagens de programação e ambientes de execução com características reflexivas também não são amplamente disponíveis, e ainda não existem características padronizadas aceitas universalmente. AS/PC possui características positivas em relação à abrangência: módulos básicos podem ser escritos em qualquer linguagem de programação, a tecnologia é madura e disponível. Entretanto, ambientes de configuração sofrem do mesmo problema que reflexão computacional: ainda não existem ADLs ou implementações de ambientes de configuração nem padrões de conversão aceitos universalmente e que sejam

interoperáveis [75]. Como se afirmou na seção 2.3.3, 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.

Comparando-se os mecanismos disponíveis nas tecnologias BC, PM-N ou AS/PC para a construção de aplicações verifica-se que eles possuem poder de expressão equivalente. Este fato foi comprovado em [98] na área de tolerância a falhas, comparando-se reflexão e AS/PC. Para produzir uma aplicação, módulos construídos a partir da tecnologia BC 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 no uso das 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 AS/PC, esta mesma estrutura é descrita explicitamente (e separadamente da implementação), através de uma ADL, independentemente da linguagem de programação dos módulos. Vale mencionar que o esforço na programação da associação de objetos-base e meta-objetos é ainda considerável, tendo em vista as linguagens de programação reflexivas disponíveis.

2.5 Combinando as tecnologias

O estudo comparativo das tecnologias, apresentado anteriormente, levou-nos à investigação de técnicas que possam facilitar o uso combinado dos conceitos de AS/PC e PM-N. O objetivo é permitir a construção de ferramentas e ambientes integrados de suporte para projetar, construir, documentar, verificar, operar e manter arquiteturas de *software*.

Partimos da observação principal de que PC, baseado em ADLs, permite que se descrevam várias características estáticas e dinâmicas de sistemas de forma explícita, que podem ser facilmente mapeadas para artefatos de *software* reais. Esta exposição da estrutura em construção facilita ao projetista o entendimento do sistema e a realização das intervenções necessárias para a adaptação do *software*, tanto durante o desenvolvimento da aplicação, quanto durante sua operação.

Uma outra observação nos levou a combinar as tecnologias. A reificação, o processo de chaveamento entre o programa normal (nível-base) e a programação extra (meta-nível), pode ser implementada de diversas formas, dependendo do ambiente PM-N particular sendo considerado. O processo da reificação (como o uso de *interceptors* em BC, por sinal), é bastante similar à transferencia de controle de código que ocorre entre um módulo e um conector no contexto de AS/PC, durante a interação de módulos. Esta observação abre caminho para integrar as tecnologias PM-N e AS/PC, de forma a se obterem os benefícios de suas vantagens intrínsecas.

Como exemplo de utilização de reflexão e configuração de meta-objetos, podemos citar uma aplicação multimídia, como a vídeo conferência, onde os meta-objetos usados 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 interpostos em uma hierarquia de meta-objetos. Na figura 2.7 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 RTP [93]. O projeto do objeto-base é transparente em relação à comunicação e codificação [69, 99, 100, 101].

Na escolha entre PM-N e AS/PC, um compromisso entre eficiência e flexibilidade deve ser buscado. Se uma linguagem orientada a objetos que oferece reflexão em tempo de compilação for utilizada, por exemplo, os objetos-base e os meta-objetos associados serão compilados como um só objeto. É desejável, entretanto, que possamos selecionar meta-objetos e associá-los aos objetos-base, quando necessário. Este seria o caso de ambientes que suportam reflexão em tempo de execução. Uma interface padronizada associada ao objeto-base permitiria que meta-objetos fossem selecionados e instanciados apenas no momento da execução, como ilustra a figura 2.7. Alguns esforços têm sido direcionados para oferecer estas características em linguagens de programação reflexivas, mas estas ainda não são uma completa realidade, pois demandariam que a tecnologia PM-N estivesse associada à AS/PC. Este último ponto reforça nosso interesse especial em investigar a combinação de tais tecnologias.

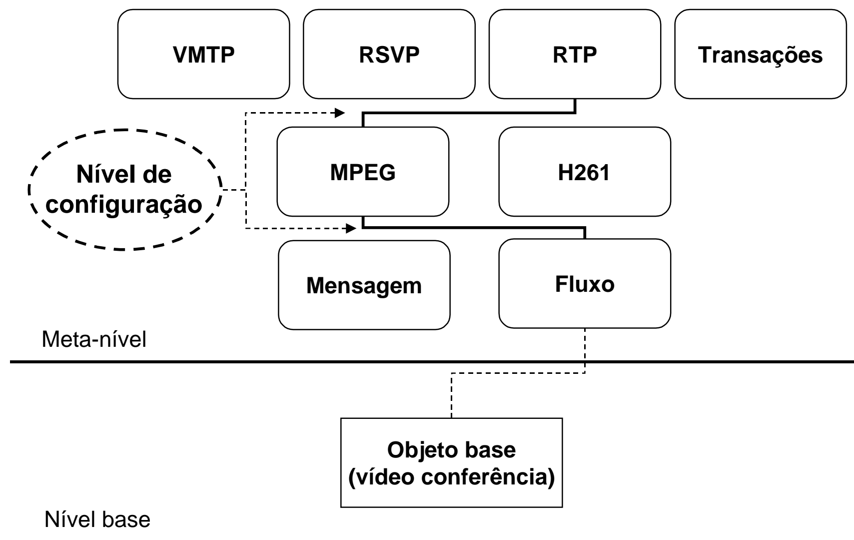


Figura 2.7 - Ambiente multiprotocolo configurável baseado em reflexão

Em [82], 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, Gerente Multiprotocolo (GM), 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 entre módulos. Abordagem similar é utilizada em [13] com a abstração de Filtros de Composição (examinada no capítulo 3).

2.6 Construindo o conjunto

Considerando características positivas de cada tecnologia, pode-se visualizar uma cadeia de decisões em relação à combinação das mesmas para implementar uma aplicação:

1. Para a programação dos módulos básicos das aplicações, a tecnologia BC (com objetos) deveria ser selecionada. Para módulos mais complexos, um padrão de projeto adequado ou *framework* de componentes pode ser utilizado.

2. PM-N pode ser empregada, de forma geral, como conceito / filosofia de projeto, no que se refere à separação de interesses. Se disponível, a reflexão computacional pode ser utilizada no ajuste fino ou extensões não-funcionais na programação de

objetos básicos, em um nível particular de detalhe. Seria desejável que a adoção da reflexão e a seleção de meta-objetos pudesse ser feita dinamicamente e postergada até a execução, ou ativada por comandos de configuração. Entretanto, se um MOP com estas características não estiver disponível, AS/PC poderá ser utilizada com esta finalidade.

3. É possível empregar AS/PC 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 previamente compostos, e com uma arquitetura em mente. Em um nível menor de abstração, AS/PC também pode ser utilizada para descrever módulos simples ou a composição de módulos complexos, antes de se descrever a configuração da aplicação.

A tabela 2.3 consolida as diretivas apresentadas e complementa a tabela 2.2.

Tabela 2.3- Aplicação de cada tecnologia

Modelo	BC	PM-N	ADL/PC
Onde aplicar	Concepção de módulos básicos	Extensão de módulos	Concepção de módulos complexos e construção de arquiteturas de sistemas
Exemplos	Visual Basic, Enterprise Java Beans, CORBA	Open C++, CLOS [10], MetaJava	Babel [102], Write [71], Regis [103], UniCon [16], OMG-STRUDL [104]

As diretivas apresentadas consideram o presente nível de maturidade, evolução e padronização das tecnologias analisadas. Nossa visão é que, de uma forma geral, todas as tecnologias analisadas utilizam a interposição de código (ou um "nível extra de indireção") para adicionar características extras à computação básica de um sistema. Seja a tecnologia BC, PM-N ou AS/PC, o resultado final é a utilização de um elemento extra, no caminho por onde acontecem as interações, que pode mudar a rotina convencional destas interações em benefício da aplicação (comprovado com os testes realizados em diversas implementações destas tecnologias, apresentados no capítulo 3). Assim, podemos afirmar que a essência básica destas tecnologias é a mesma, diferindo no modo como enfocam os requisitos de separação de interesses, reusabilidade, evolução dinâmica e abrangência.

O sucesso da combinação adequada das tecnologias avaliadas em uma solução integrada não é garantido e a combinação não é trivial. Entretanto, as vantagens potenciais desta integração, discutidas na seção 2.5, e as diretivas para realizar esta

integração, discutidas nesta seção, nos leva a este caminho.

2.7 Trabalhos correlatos

Até onde vai nosso conhecimento, as tecnologias avaliadas e os problemas discutidos neste capítulo estão sendo atualmente estudados, isoladamente ou em combinações específicas. Wegner [6] apresenta um painel bastante completo do tipo de evolução aguardada em direção a um consenso no projeto de grandes sistemas distribuídos. [98, 92, 105, 106, 91] apresentam alternativas no uso de configuração com objetos distribuídos sobre ORBs. [59, 57, 61, 54 e 55] apresentam soluções para estender a funcionalidade de sistemas, utilizando reflexão computacional e explorando a flexibilidade do uso de meta-objetos. A abordagem de [13] 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 AS/PC, serão avaliadas no próximo capítulo.

2.8 Conclusão

Neste capítulo foram analisadas as seguintes tecnologias: Baseadas em Componentes, Programação de Meta-Nível e Arquiteturas de *Software* / Programação por Configuração. O objetivo desta análise foi discutir como estas tecnologias atendem aos requisitos de reusabilidade, separação de interesses, evolução dinâmica e abrangência. Estes são, em nossa opinião, requisitos que devem ser preenchidos pelas tecnologias a serem utilizadas no projeto de sistemas modernos. Todas as tecnologias analisadas têm os seus méritos e atendem a alguns dos requisitos enumerados. Em [107, 23 e 6] menciona-se que a tecnologia de objetos, hoje disponível, não apresenta solução para todos os tipos de problemas de *software* e, portanto, outras técnicas precisam ser usadas concomitantemente. Isso nos levou a pesquisar uma combinação adequada entre as tecnologias avaliadas.

No próximo capítulo são analisadas algumas propostas representativas das tecnologias analisadas. As avaliações e os testes feitos com estas propostas têm o propósito de levantar subsídios para validar a abordagem proposta em nossa tese.

Capítulo III

Trabalhos Correlatos

3.1 Introdução

Neste capítulo são discutidas algumas propostas e ferramentas representativas das tecnologias avaliadas no capítulo 2: sistemas Baseados em Componentes (BC), Programação de Meta-Nível (PM-N) e Arquiteturas de *Software*/Programação por Configuração (AS/PC). Adicionalmente, também são examinados alguns pontos de UML, *Universal Modeling Language*, adotado pela OMG como padrão para especificação de sistemas baseados em objetos. Para algumas destas propostas, quando um protótipo estava disponível, foi possível elaborar um exemplo[4]. A aplicação dos produtores-consumidores com um *buffer* limitado é utilizada como base. A partir do exercício de se implementar a referida aplicação com o suporte oferecido em cada proposta, uma comparação com os requisitos determinados no capítulo 2 é realizada. Este exemplo, utilizado nos trabalhos originais destas propostas, facilita uma comparação imediata com a nossa.

O objetivo da discussão deste capítulo é reforçar as observações feitas anteriormente sobre as tecnologias avaliadas, com base em propostas reais e exemplos. Também se objetiva avaliar padrões contemporâneos para o desenvolvimento de *software* a fim de identificar-se onde podemos oferecer contribuição.

3.2 Programação de Meta-Nível

Podemos classificar as várias vertentes de PM-N em reflexão computacional,

---

[4] Uma parte dos exemplos foi desenvolvida durante a proposta de tese. Assim, alguns destes encontram-se resumidos no texto, mas poderão ser consultados em [110].

reflexão estrutural e outras abordagens. Estas últimas não se baseiam explicitamente em reflexão, mas utilizam os conceitos de PM-N ou implementam a reflexão com técnicas particulares, como AOP, por exemplo. A tabela 3.1 aponta as principais características das vertentes de PM-N e enumera alguns exemplos de implementação.

Tabela 3.1 - Quadro comparativo das técnicas de reflexão

	Reflexão computacional		**Reflexão estrutural**		**Outros**	
Característica	OpenC++	MetaJava	Java	Javassist	FC	AOP
Estática	✓					✓
Dinâmica		✓	✓	✓	✓	
Em tempo de compilação	✓					✓
Em tempo de execução		✓	✓	✓	✓	
Ambiente especializado	Pré-processador	JVM modificada	JVM	JVM, reescreve byte codes	Sina	*Weaver*

O exemplo da aplicação produtores-consumidores programada com PM-N utilizará um objeto-base que implementa um *buffer* genérico, sem limitação de tamanho e sem sincronização no seu acesso. Os produtores e consumidores podem ser implementados por simples objetos.

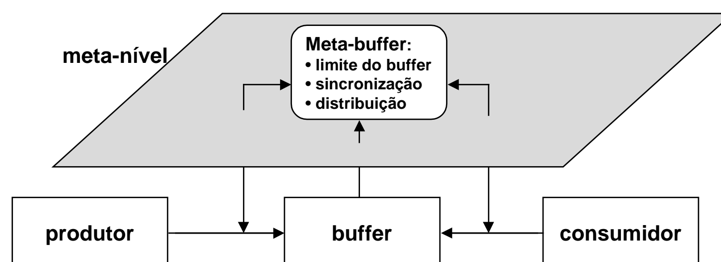


Figura 3.1 - *Buffer* limitado por reflexão

A figura 3.1 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*.

A seguir, discutimos os exemplos da tabela 3.1.

3.2.1 Reflexão Computacional

3.2.1.1 Open C++

Open C++ [9] é uma linguagem de programação, baseada em C++, que oferece reflexão estática. Open C++ oferece um MOP que permite o uso de reflexão comportamental em tempo de compilação.

Além da reflexão, o MOP de Open C++ admite que o programador controle a compilação, disponibilizando a árvore de *parsing* para consulta e, opcionalmente, alterações. O programador pode, então, escrever um código que toma decisões bem específicas, a partir das informações disponíveis no meta-nível. Estas decisões, que dependem de informações de meta-nível, não seriam possíveis para um otimizador de código. Com esta técnica, a sintaxe original de C++ pode ser estendida. Portanto, Open C++ também pode ser considerada uma linguagem aberta.

Um exemplo

O código do *buffer* limitado foi desenvolvido a partir dos exemplos do manual do usuário da versão 2.0 de Open C++ [108].

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. Para simplificar a implementação, apenas o aspecto de limitação foi considerado. Assim, uma meta-classe *BoundedClass* é desenvolvida com dois objetivos: impor a característica de limitação ao *buffer* e estender a sintaxe do C++, permitindo que se declare um *buffer* limitado simplesmente acrescentando o lexema ***bounded*** à declaração da classe.

A declaração da meta-classe *BoundedClass* dá origem a um pré-processador executável. Este 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++, para C++, com a funcionalidade equivalente aos objetos-base e meta-objetos, que então pode ser

compilado por um compilador C++ tradicional.

No próximo trecho de código (listagem 3.1) apresentamos a aplicação. Observa-se, de início, o conforto sintático para o programador da aplicação, que pode introduzir na classe *Buffer* as características embutidas no meta-objeto. Observa-se, ainda, que ao se declarar a classe *Buffer* como sendo *bounded*, também devem ser declarados os métodos *before_Put* e *before_Get* (linhas 6, 7, 19 e 31). Estes dois métodos podem ser considerados guardas ou pré-condições para os métodos *Put* e *Get*, respectivamente.

```
 1  /* Buffer Limitado com reflexão */
 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  }
```

Listagem 3.1 - *Buffer* limitado em Open C++, utilizando extensão sintática

Comentários

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

estabelecerem 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 teria o papel de coordenar a execução dos dois níveis em tempo de execução.

3.2.1.2 MetaJava

MetaJava [63, 56] é uma extensão da Máquina Virtual Java (*Java Virtual Machine* - JVM) que suporta reflexão computacional, adicionando características de reflexão não encontradas em Java originalmente, e admite a criação e a ligação de meta-objetos 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 a meta-objetos concebidos com estas finalidades.

A implementação de reflexão dinâmica em MetaJava foi possibilitada porque:

- programas em Java executam sobre um interpretador (JVM), o que simplifica a extensão do ambiente de execução com características de reflexão mais dinâmicas;
- a 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);
- a JVM permite o controle de alguns eventos associados a objetos, como a carga destes objetos, através de uma classe especial chamada *ClassLoader*.

Um exemplo

Na listagem 3.2 é apresentada a classe-base *Buffer*, implementada por um vetor grande (60 itens, para o nosso exemplo, este 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 últimos.

```
 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  }
```

Listagem 3.2 - Classe-base Buffer

Na listagem 3.3 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-se que no construtor de *MetaBuffer* é estabelecida a ligação entre o objeto-base (passado como parâmetro para o construtor) e o meta-objeto (linhas 8 e 9). Em seguida, ainda no construtor, uma lista com os métodos, cujas invocações devem ser interceptadas, é registrada.

```
 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    public MetaBuffer (Object obj, String MetaArgs []) {
 7      rn_itens = 0;
 8      attachObject(obj); // estabelece a ligação entre os níveis base e meta
 9      registerEventMethodCall(obj , methodNotificationHandler, MetaArgs);
10    }
11    public int eventMethodEnterInt(Object o, EventDescMethodCall event) {
12      int ret;
13      if (event.methodname.compareTo("Put")==0) ret=this.MetaPut(o,event);
14      else  ret = this.MetaGet (o, event);
15      return ret;
16    }
17    public synchronized int MetaPut (Object o, EventDescMethodCall event) {
18      while (rn_itens >= META_MAX)
19         try { wait (); } catch (InterruptedException e) {}
20      int ret = continueExecutionInt(o, event);  ++rn_itens;
21      notifyAll ();
22      return ret;
23    }
24    public synchronized int MetaGet (Object o, EventDescMethodCall event) {
25      while (rn_itens <= 0)
26         try { wait (); } catch (InterruptedException e) {}
27      int ret = continueExecutionInt(o, event);  --rn_itens;
28      notifyAll ();
29      return ret;
30    }
31  }
```

Listagem 3.3 - Meta-objeto *buffer* limitado e sincronizado

Sempre que um método com tipo de retorno ***int*** for invocado no método base o método **eventMethodEnterInt** (linha 11) é invocado no meta-objeto, antes da invocação original chegar ao objeto-base. Na continuação, os métodos ***MetaPut*** ou ***MetaGet*** são ativados (linhas 13 e 14 respectivamente), em função do método do objeto-base interceptado. Estes métodos são sincronizados (em Java a referência ***synchronized*** 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 20 e 27).

A listagem 3.4 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 à 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.

```
 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                                          // Instancia o meta-objeto MetaBuffer
17      new MetaBuffer(Buffer, MetaArgs);
18
19      /* Ativa as threads nos produtores e consumidores */
20      prod1.start();  cons1.start();  cons2.start();  prod2.start();
21    }
22  }
```

Listagem 3.4 - Aplicação *buffer* limitado

Comentários

Guaraná [69], um sistema semelhante a MetaJava, foi proposto com características adicionais: o desenvolvedor do nível-base da aplicação não precisa indicar o que vai ser refletido para o meta-nível. Para que isso aconteça, o meta-nível tem acesso irrestrito ao nível-base, mas pode-se impor um esquema de segurança para que a aplicação não consiga ultrapassar certos limites no uso de recursos. Adicionalmente, em Guaraná é possível a configuração dos meta-objetos utilizados em tempo de execução.

3.2.2 Reflexão Estrutural

3.2.2.1 Java

A API de reflexão estrutural de Java representa, ou reflete, as classes, interfaces, e objetos em uma JVM (*Java Virtual Machine*) em operação. Segundo o tutorial sobre a API de reflexão estrutural de Java [109], "você vai querer utilizar a API de reflexão se você está desenvolvendo ferramentas tais como depuradores (*debuggers*), visualizadores de classes e construtores de interfaces gráficas, por exemplo". Para isso, a API de reflexão de Java permite:

- determinar a classe de um objeto, se a referência para o objeto for conhecida;
- obter informações sobre uma classe, tais como modificadores, campos, métodos, construtores e superclasses;
- descobrir quais são as constantes e declarações de métodos de uma interface;
- criar uma instância de uma classe, cujo nome só será conhecido em tempo de execução;
- obter e atribuir valores a campos de um objeto, mesmo que o nome do campo só seja conhecido em tempo de execução;
- invocar um método em um objeto, mesmo que o nome do método e a referência ao objeto sejam obtidas apenas em tempo de execução, ou mesmo conhecendo-se apenas o nome da classe.

Ao manipular uma classe, por exemplo, para inspecionar seu funcionamento, é necessário obter a referência desta classe, a partir do seu nome. Isto é possível por

reflexão. Em seguida pode-se conseguir informações sobre seus métodos e campos. Para obter esta informação é necessário, antes, dispor de uma referência ao objeto **Class** que reflete esta classe. Isto também é possível por reflexão.

Para cada classe carregada na JVM, o ambiente Java de execução (*Java Runtime Environment* - JRE) mantém um objeto *Class*, imutável, que contém informações sobre a classe. Um objeto *Class* reflete a estrutura da classe associada. Com a API de reflexão, é possível invocar os métodos de um objeto *Class* para que este retorne uma referência aos objetos das classes *Constructor*, *Method* e *Field*, que contém detalhes da classe associada. Em seguida, os mencionados objetos podem ser consultados para obter informações sobre os construtores, métodos e campos correspondentes definidos nesta classe.

Comentários

O tipo de reflexão estrutural oferecido em Java é dinâmico. Isto é factível, pois:

- existe a infra-estrutura da JVM que carrega e "executa" classes, e pode ser consultada durante a execução de um programa;
- o código gerado pelo compilador Java para cada classe é uma representação binária, padronizada, chamada *ByteCode*, que é interpretada pela JVM. Assim sendo, qualquer sistema que possua uma implementação da JVM pode interpretar o *ByteCode* de uma classe;
- o compilador Java também gera código para alimentar a JVM com as meta-informações de cada elemento (por exemplo, o objeto *Class*, associado a uma classe carregada).

Além da API de reflexão, a JVM tem a capacidade de carregar classes e criar instâncias das mesmas, durante sua operação. A JVM possui um carregador (*ClassLoader*) *default*, mas uma outra API de Java permite que o programador crie uma especialização do carregador para controlar a carga das classes. Por exemplo, durante a operação de carga, é possível substituir o nome original de uma classe. Isto deixa espaço para que se combine a API de reflexão com um carregador especializado, na tentativa de adaptar a funcionalidade das aplicações.

A reflexão em Java possui limitações, entretanto. Permite-se consultar as meta-informações, disponibilizadas através da API, mas não é possível alterar as mesmas (por

exemplo acrescentar um método a uma classe). Assim sendo, a evolução dinâmica das aplicações em Java fica comprometida. Outra limitação neste contexto (mas não diretamente relacionada à reflexão), está no fato de não se poder sobrepor a carga de uma classe à sua imagem carregada anteriormente. Também não é possível apagar-se explicitamente uma classe carregada. Isto limita a carga de uma nova versão de uma classe em tempo de execução.

3.2.2.2 Javassist

Javassist [111] é uma extensão à API de reflexão de Java, que permite alterações no comportamento das aplicações, em adição a consultas às meta-informações desta estrutura, como originalmente disponível em Java. Diferentemente de outras extensões, que facilitam mudanças no comportamento de aplicações em Java, Javassist não altera a JVM ou o compilador Java, e nem trabalha interceptando o fluxo de operações nos objetos. Assim sendo, e para não incorrer em problemas de desempenho, a reflexão estrutural em Javassist só é permitida antes que as classes sejam carregadas.

Javassist permite que se altere a estrutura de uma aplicação através de mudanças na definição de uma classe, função ou registro, sob-demanda. A idéia central da implementação de Javassist é oferecer reflexão estrutural, através de transformações nos *ByteCodes* em tempo de compilação ou em tempo de carga, técnica chamada de BCA - *ByteCode Adaptation*. Os autores de Javassist argumentam que realizando as transformações, antes da execução dos *ByteCodes*, não se impõe trabalho adicional à JVM e são evitados problemas de desempenho.

Os programadores que desejam utilizar Javassist não precisam, em princípio, conhecer a estrutura de *ByteCodes*: é proposta uma abstração de programação que facilita o uso dos mecanismos de Javassist na própria linguagem Java. Javassist, por outro lado, impõe ao programador a inclusão de uma classe que deve estender a classe *ClassLoader* para carregar os *ByteCodes*. Um exemplo deste código seria:

```
class MyLoader extends ClassLoader {
public Class loadClass(String name) {
byte[] bytecode = readClassFile(name);
return resolveClass(defineClass(bytecode));
}
private byte[] readClassFile(String name) {
// ler um arquivo contendo a classe
}
```

Para usufruir da API de Javassist, a carga das classes também deve seguir um roteiro específico. O *ByteCode* original é lido pelo construtor de uma classe chamada *CtClass*, que armazena as informações simbólicas para futuras consultas. Neste ponto, o programador pode usar a API de Javassist para fazer modificações na classe lida. Em seguida, invoca-se o método *toBytecode()* para converter a classe alterada em *ByteCod*e compatível com a JVM. Finalmente, a classe pode ser carregada na JVM através do método *load (),* retornando um objeto da classe *Class*, original de Java.

A API de Javassist oferece as seguintes categorias de funções:

- introspecção: com possibilidades adicionais em relação à API original de Java
- alteração: métodos como setName(String name), para alterar o nome de uma classe e addMethod(...), para adicionar um método novo a uma classe, estão disponíveis. Também é possível alterar um método existente, e até mesmo expressões dentro de um método existente.

A introspecção estará disponível, desde que o programador tenha incluído o seu *ClassLoader*, e siga o roteiro para a carga de classes de Javassist. Entretanto, a alteração não é trivial. Por exemplo, o método *addMethod()* tem a seguinte assinatura:

```
void addMethod(CtMethod m, String name, ClassMap map)
```

O nome do método a ser acrescentado é especificado em *name*. O corpo do método é copiado de um dado método *m*, pré-compilado. *map* é uma *HashTable* obtida em dois passos derivada da classe original e usada como referência para fazer a inclusão do novo método. A alteração de métodos é ainda mais complexa e não será exemplificada.

Comentários

As transformações são operadas diretamente sobre *ByteCodes*, não sendo necessário que o código-fonte esteja disponível. É até mesmo possível, com a API da Javassist, criar uma nova classe sem utilizar um compilador, em tempo de execução, contanto que se tenha os *ByteCodes* do conjunto de métodos desta classe. Isto pode ser útil para a criação de classes *proxy* sob-demanda (ver a seção 7.3 do capítulo 7 para um caso de uso).

Javassist oferece ao programador um nível de controle sobre o programa que Java não oferece originalmente. A aplicação pode ser programada para evoluir dinamicamente. No código do programa, as classes a serem carregadas por Javassist são selecionadas e alteradas sob demanda. Entretanto, este controle só está disponível antes de cada classe ser carregada. Uma aplicação com seus componentes já em execução não admite alterações.

A proposta de Javassist é interessante para obter a separação de interesses e evolução dinâmica, e é uma alternativa a outras propostas dos mesmos autores, como o OpenC++, discutido anteriormente. Entretanto, Javassist manipula *ByteCodes* para realizar sua tarefa, o que é geralmente considerado problemático. A manipulação de *ByteCodes* de Java, por outro lado, tem se tornado uma prática aceitável, e é utilizada com objetivos semelhantes em [112, 101 e 113].

3.2.3 Outros

3.2.3.1 Filtros de Composição (*Composition-Filter*)

O modelo de Filtros de Composição (FC) [13] apresenta alguns conceitos que têm interseção com a proposta de nosso trabalho. FC é baseado em uma extensão do modelo tradicional de objetos. Uma característica desta extensão é a imposição da separação explícita entre a interface de um objeto e o 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 pedido/resposta, e estas são tratadas como objetos de primeira-classe. Cada mensagem carrega informações que atuam como seletores do comportamento dos objetos (figura 3.2).

A parte externamente visível da interface de um objeto é controlada por filtros. Estes podem inspecionar e manipular as mensagens de entrada/saída dos objetos, e reagir de acordo com o conteúdo das mesmas. Assim, são oferecidas 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 em FC é programada pela composição de objetos. Uma composição é especificada declarando-se, na interface de um objeto, referências a instâncias de outros objetos, chamados de objetos internos. O relacionamento entre um objeto e seus objetos internos é configurado através da programação adequada dos filtros. Este modelo oferece a possibilidade de se utilizarem mecanismos de herança e delegação para estabelecer o relacionamento entre objetos, e ainda permite que tais relações sejam dinamicamente habilitadas e desabilitadas.

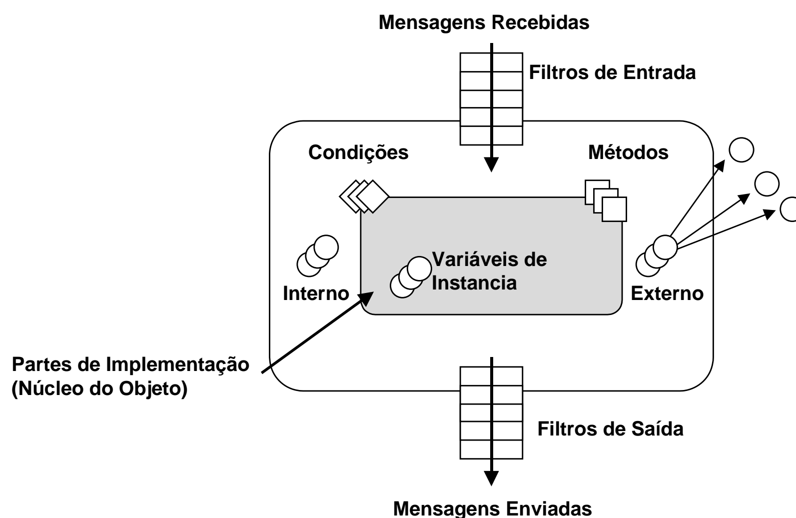


Figura 3.2 - Objeto em FC

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, poderia ser feita de forma ortogonal e sem efeitos colaterais entre os diversos domínios de problemas. Entre os domínios de problemas estudados pelos autores, 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 ou repassar mensagens sem nenhum processamento, e ações especiais relacionadas com o tipo como, por exemplo, travar o repasse de uma mensagem até que determinada condição seja satisfeita no estado do objeto.

Sina [114] é uma linguagem orientada a objetos que implementa os conceitos de FC. 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 a determinadas condições. Existe também uma seqüência de ações possíveis de serem 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 ser programado para encaminhar as mensagens para um outro filtro, ao invés de encaminhá-las para o núcleo do objeto.

Um exemplo

O próximo exemplo implementa um *buffer* limitado utilizando Sina.

```
 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  class BoundedBuffer implementation
14     instvars
15         store: Array (limit);
16         hear, tail: Integer;
17     conditions
18         Empty begin return head=tail end;
19         Partial begin return (inner.Empty.not and inner.Full.not) end;
20         Full begin return (limit+tail-head).mod(limit)=1; end;
21     initial
22         begin head :=  1; tail := 1; end;
23     methods
24         put (elem: Any) returns Nil
25            begin store.atPut(head. elem); head := head.mod(limit)+1; end;
26         get returns Any
27            begin return store.at(tail); tail := tail.mod(limit) + 1; end;
28  end
```

Listagem 3.5 - *Buffer* limitado em Sina

Neste exemplo (listagem 3.5) foram utilizados os filtros *Dispatch* e *Wait* (linhas 9 e 10). Devido à simplicidade do exemplo, não foi necessária a utilização dos mecanismos de relacionamento entre objetos.

Observa-se que Sina obriga o programador a descrever explicitamente a interface de uma classe (linhas de 1 a 12), o que facilita a sua reutilização. Sina nos pareceu 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.* Em muitas situações, isto limita a reutilização destas classes.

Comentários

Um filtro do tipo Meta também está disponível em Sina. O comportamento do filtro Meta é similar ao *Dispatch*. Entretanto, no filtro Meta, uma mensagem recebida 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 com características de reflexão, através da interceptação de mensagens.

Outros aspectos, como tempo-real, foram adaptados ao modelo de FC pelos autores [115]. Nas referências 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 princípio, a modificação da linguagem.

3.2.3.2 AOP - *Aspect Oriented Programming*

A Programação Orientada a Aspectos (*Aspect-Oriented Programming*, AOP) é um paradigma proposto como solução para a concepção de sistemas complexos, que oferece separação de interesses, caracterizados por aspectos da aplicação neste contexto [116]. AOP propõe que a implementação de uma aplicação em uma linguagem de programação particular consiste em:

- uma linguagem para a programação de componentes e uma ou mais linguagens para programar aspectos ou interesses não-funcionais;
- um entrelaçador ou costurador (*weaver*) para o conjunto destas linguagens;

- um programa que implementa os componentes funcionais utilizando a linguagem de componentes; e,
- um ou mais programas de aspectos que implementam os aspectos não-funcionais utilizando as linguagens de aspectos.

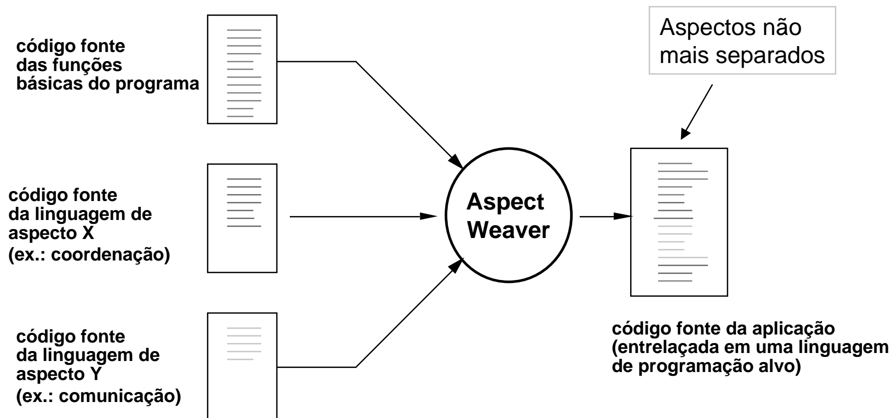


Figura 3.3 - AOP e o Entrelaçador (*Aspect Weaver*)

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 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" é responsável por receber as informações descrevendo os aspectos e gerar como resultado uma representação da aplicação que possa ser executada. Esta forma executável poderá apresentar um código emaranhado, mas eficiente, dependendo da complexidade do problema e do "entrelaçador" (figura 3.3).

"D", um exemplo de implementação de AOP, é descrito em [12]. *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 quanto Ridl são simples e podem ser classificadas como linguagens declarativas.

Um outro componente de *D*, chamado "*Aspect Weaver*", combina as descrições em JCore, Cool e Ridl, e produz código Java. O código Java pode, então, ser compilado gerando um programa executável, que contempla o comportamento especificado nos aspectos não-funcionais.

Um exemplo

A seguir apresentamos o exemplo do *buffer* limitado em *D* (adaptado de [12]), implementado para a primeira versão de AspectJ [117], uma versão da linguagem veiculada pela Xerox, onde o projeto de AOP está atualmente hospedado [118]. Na listagem 3.6, a classe *BoundedBuffer*, programada em JCore, é apresentada. 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 = 0, 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   }
```

Listagem 3.6 - Classe *BoundedBuffer* em JCore

Na listagem 3.7, 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).

```
 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    }
```

Listagem 3.7 - Aspectos de coordenação e sincronização em Cool

Comparando o código apresentado nas listagens 3.6 e 3.7, observa-se que os aspectos de coordenação estão desvinculados dos aspectos funcionais da aplicação. A programação do aspecto de distribuição não estava disponível na ferramenta utilizada. Embora este 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, 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.

Comentários

Além da separação de aspectos, as vantagens imediatas de AOP são a legibilidade do código (antes do entrelaçamento) e a potencialidade de reutilização de módulos dos aspectos funcionais básicos.

O entrelaçador funciona como um compilador especial, que combina o código dos aspectos funcionais e não-funcionais para implementar um único programa executável. Analisando a literatura de AOP, dois tipos de transformação de código, feitos pelo entrelaçador, podem ser identificados. O primeiro, não muito comum, consiste em otimizações guiadas por diretivas para descrição de aspectos não-funcionais especiais, que levam a códigos mais eficientes, ou com melhor desempenho [119]. Os autores argumentam que nesta classe de aplicações, a análise dos programas e o raciocínio envolvido é tão significativo, que não se pode confiar em compiladores para fazer estas otimizações de forma confiável. Isto ocorreria porque os pontos de junção entre os níveis base e meta não poderiam ser tratados separadamente [120].

Consideramos que este tipo de otimização é na maioria dos casos ortogonal à arquitetura dos sistemas. No contexto de AS/PC, os programas descritos com linguagens de aspecto, o entrelaçador e o ambiente de suporte à execução podem ser conectados a módulos específicos ou subsistemas, através de anotações introduzidas no nível da ADL. Em [121] é sugerido este tipo de anotação para selecionar o protocolo de coerência de dados, requerido para suportar interações de módulos no contexto de memória compartilhada-distribuída (*distributed-shared memory* - DSM).

Um segundo tipo de transformação de código, que parece ser mais comum, ocorre quando os pontos de junção de código podem ser tratados separadamente. Isto é, em essência, similar ao que ocorre entre nível-base e meta-nível, em compiladores para linguagens reflexivas. De fato, como apontado em [122], "reflexão parece ser um mecanismo suficiente para AOP, mas muitos a consideram poderosa demais: qualquer coisa pode ser feita com reflexão, incluindo AOP". Neste contexto, uma vantagem de AOP seria sua habilidade para disciplinar a reflexão através de linguagens de programação de aspectos especiais. Por exemplo, AspectJ, baseada em Java, provê uma cláusula *advise*, que é usada para especificar os métodos a serem logicamente interceptados e associados a um código de meta-nível quando executados (veja exemplo do capítulo 7). O papel desta cláusula é similar àquele das ligações em ADLs, e isto pode ser uma ponte entre AOP e AS/PC.

3.3 Sistemas Baseados em Componentes e *Middleware*

O conceito básico de componente, como módulo de computação autônomo e reutilizável, parece ser adequado para a construção de sistemas. Um dos diferenciais entre um sistema BC e outro é o *middleware* que provê a infra-estrutura para os componentes. Por exemplo, determinado *middleware* pode oferecer serviços mais convenientes para determinado tipo de aplicação, ou pode ser utilizado em um maior numero de plataformas. Dentre os diversos *middlewares* disponíveis, selecionamos CORBA e Jini para serem avaliados. CORBA e Jini são projetos hospedados em grandes empresas, mas contêm contribuições de vários segmentos da indústria e da academia, seja através do consórcio administrado pela OMG, no caso de CORBA, ou como projeto aberto, no caso de Jini.

3.3.1 CORBA

CORBA [8] é um padrão para a implementação do modelo de gerência de objetos, *Object Management Architecture* - OMA, também da OMG. Argumenta-se que CORBA possui características consideradas facilitadoras para o projeto de sistemas:

- 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 à linguagem de programação;
- oferece soluções adequadas para a semântica de objetos distribuídos;
- possui suporte adequado para sistemas distribuídos (chamado de *facilidades comuns* - *common facilities* ou *CORBAservices*);
- possui um conjunto de *frameworks* de propósito específico (chamados de *facilidades verticais* - *vertical/domain CORBAfacilities*) 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 diferentes, como o DCE da OSF e o COM da Microsoft, com o CORBA. A OMG possui um compromisso com a padronização e o mapeamento entre CORBA e outros padrões como OSF-ODP, ANSA e TINA;
- a interoperabilidade com outros padrões e 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;

Um exemplo

Apresenta-se, a seguir, o exemplo dos produtores-consumidores implementado por uma descrição em OMG-IDL 2.0. Observa-se que na listagem 3.8 apenas a interface do módulo *buffer* é descrita (linha 15), porque este elemento irá oferecer um serviço através de seus métodos. Embora o aspecto de distribuição esteja automaticamente contemplado pela infra-estrutura de suporte ao CORBA, não é possível descrever aspectos como a sincronização ou exclusão mútua.

```
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    void Put (in item item_info) raises (NO_EMPTY_SLOTS);
17    short Get (void) raises (NO_SLOT_IN_USE);
18    sreg Status (void);
19 };
```

Listagem 3.8 - Descrição das interfaces do *buffer* em IDL

No exemplo, nada é mencionado a respeito dos clientes, produtores e consumidores. Entretanto, na descrição da interface pode ser previsto o tratamento de exceções. Por exemplo, na listagem 3.8, 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*.

Comentários

Observa-se que faltam em CORBA mecanismos para a descrição da arquitetura das aplicações, sendo apenas permitida a descrição das interfaces a serem implementadas por objetos servidores e utilizadas por clientes. Além disso, não existem meios de se descreverem ou comporem aspectos não-funcionais e operacionais das aplicações, o que leva a soluções *ad-hoc*. O código de um cliente CORBA para Java, por exemplo, assemelha-se ao da listagem 2.2- capítulo 2, onde se verifica grande entrelaçamento de código dos aspectos básicos da aplicação com o uso da API de CORBA.

A versão 2.0 de CORBA somente admite o modelo implícito de interação entre objetos. Aspectos não-funcionais relacionados com a comunicação, reserva de recursos ou QoS não podem ser descritos. Existem propostas de extensões aos estilos de comunicação oferecidos [123, 120], para que sejam aceitas a comunicação de grupo e conexões orientadas a fluxos contínuos. Algumas RFPs (*Request for Proposal*) estão em aberto na OMG, com relação a padronização destes pontos [124], bem como para a padronização de extensões de tempo-real, chamadas RT-CORBA [125].

No final de 1998, uma nova versão de CORBA, 3 [126], foi anunciada. Esta versão inclui algumas características que aproximam um pouco CORBA de PM-N e AS/PC. As inovações são agrupadas em 3 categorias: (i) maior integração com a Internet, (ii) controle de QoS e (iii) um modelo de componentes.

Por exemplo, o modelo de componentes de CORBA 3 inclui facilidades para o programador de aplicações reutilizar *software*. Uma das inovações é o emprego da técnica de *wrapping* para envolver um objeto CORBA em uma casca de componente:

- um padrão de *container* foi desenvolvido, para encapsular interações em transações, segurança, persistência e uma interface para a resolução de eventos;
- integração com EJB (*Enterprise JavaBeans*) e outras linguagens de programação, que são encapsuladas nos *containers*;
- um formato de distribuição de *software* multiplataforma e ferramentas auxiliares baseadas em XML; e,
- um mapeamento para linguagens de *script* já estabelecidas (Perl, por exemplo).

Não obstante CORBA 3 incorporar características úteis para a concepção de grandes sistemas, acreditamos que ainda existam pontos em aberto. Por exemplo, embora seja possível a configuração de características de QoS ou de tempo-real, isto é feito diretamente no código do cliente e objeto servidor (com a exceção da configuração de características do protocolo de comunicação, como o TCP, que pode ser descrita em uma *struct* IDL, separadamente das interfaces dos objetos). Não existe uma forma para se definirem tais características através de OMG-IDL. Além disso, a definição da estrutura da aplicação se encontra embutida no código dos objetos e clientes. Assim, embora tenham sido ampliados os mecanismos para a programação de aspectos não-funcionais em CORBA, a separação de interesses ainda não foi contemplada. Este ponto de vista é corroborado por [127], observando que "… embora OMG-IDL seja adequada para especificar infra-estruturas para computação distribuída, ela não permite a especificação do comportamento ou relacionamento entre classes, além do que é permitido por generalização (com adaptações isto também é aplicável a DCOM-IDL). Consequentemente, uma IDL pode ser utilizada para se especificarem as operações associadas com uma interface, mas não pode definir métodos, casos de uso ou vários tipos de relacionamentos tipicamente associados a componentes reais de *software*".

3.3.2 Jini

Jini [128, 129] é um *middleware* proposto pela Sun Microsystems para confederar dispositivos, componentes de *software* e serviços interligados por redes. Telefones celulares, PDAs, impressoras, *scanners*, fotocopiadoras, e serviços de *software*, podem interagir apenas *plugando* sua interface no ambiente oferecido por Jini.

A tecnologia Jini é baseada em Java e utiliza intensamente recursos da linguagem, tais como: a serialização (e mobilidade) de objetos, *sockets*, RMI, carga dinâmica de *ByteCodes* e segurança.

Existem dois componentes principais em um ambiente Jini:

- Serviços: uma impressora, uma unidade de disco, um sistema de pedido eletrônico, etc.
- Clientes: utilizam os serviços Jini disponíveis, conhecendo apenas suas interfaces.

Os serviços básicos de suporte de um ambiente Jini são uma combinação de:

- mobilidade dos objetos de serviço, incluindo seu código, (ou o seu *proxy*), para o cliente do serviço, sob demanda;
- identificação dos serviços por sua assinatura de tipos em Java;
- um serviço de localização (*Lookup Service*), para o registro de serviços e consulta aos serviços disponíveis pelos clientes;
- um protocolo (*Discovery Protocol*), que descobre servidores de *Lookup* disponíveis na rede, utilizando *multicast*.

A especificação Jini é independente de protocolo de rede, mas até o momento as únicas implementações são baseadas em TCP/IP. A serialização de objetos e *sockets* Java é utilizada para mover os objetos ao redor da rede, entre os componentes. A movimentação de objetos se dá principalmente com o código de serviços, ou o seu *proxy*, para as JVMs onde se encontram os clientes. RMI é utilizado para a interação entre clientes e serviços que se encontram em JVMs distribuídas.

Um exemplo

Na listagem 3.9 apresentam-se alguns trechos da classe *Prod*, que inicia um objeto produtor. Neste trecho de código, apenas as rotinas para o uso dos serviços do

ambiente Jini estão presentes, representando uma "casca", envolvendo o objeto que implementa o produtor, necessária para o funcionamento em um ambiente Jini. A implementação do funcionamento básico do produtor foi omitida (esta foi a parte simples da implementação).

Observa-se que é necessária a inclusão de bibliotecas de classes e rotinas a fim de que o objeto se identifique para o ambiente, e em seguida tentam-se algumas alternativas para se registrar em um *Lookup Server*. Existe também a necessidade de se tratarem as propriedades de segurança em Jini. Parte deste tratamento, na realidade, é relativo ao uso de RMI.

```
 1  import ProdCons;
 2  import java.security.Permission;
 3  import java.rmi.RMISecurityManager;
 4  import java.rmi.RemoteException;
 5  import net.jini.discovery.LookupDiscovery;
 6  import net.jini.discovery.DiscoveryListener;
 7  import net.jini.discovery.DiscoveryEvent;
 8  import net.jini.core.lookup.ServiceRegistrar;
 9  import net.jini.core.lookup.ServiceTemplate;
10
11  public class Prod implements DiscoveryListener
12  {
13     public Prod()
14     {
15        System.setSecurityManager(new RMISecurityManager());
16        LookupDiscovery  discover = null;
17
18        discover=new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
19
20        discover.addDiscoveryListener(this);
21     }
22
23     public void discovered(DiscoveryEvent evt)
24     {
25        ServiceRegistrar[] registrars = evt.getRegistrars();
26        Class[] classes = new Class[]{Prod.class};
27        Prod prod = null;
28        ServiceTemplate template=new ServiceTemplate(null, classes, null);
29
30        ServiceRegistrar registrar=registrars[n];
31        prod = (Prod) registrar.lookup(template);
32
33        prod.MakeUI(); // aqui a interface que implementa a funcionalidade básica é instanciada.
34     }
35  }
36
37  class MyRMISecurityManager extends RMISecurityManager
38  {
39     public void checkPermission(Permission perm)
40     {
41        super.checkPermission(perm);
42     }
43  }
```

Listagem 3.9 - Preparando um produtor para operar em um ambiente Jini

Comentários

Embora a descrição das interfaces dos serviços oferecidos seja obrigatória, em Jini não existe um esquema para a descrição explícita da estrutura das aplicações. Estas se parecem com um teia de serviços e clientes que se forma durante a execução, da mesma forma que ocorre com CORBA ou DCOM. Também não existe suporte para a configuração dos estilos de interação dos objetos de uma forma simples. A comunicação entre objetos é feita preferencialmente por RMI.

Comparando com outros *middlewares* da categoria, como CORBA ou DCOM, Jini parece tornar mais transparente o uso de serviços de repositório de interfaces e a geração de *stubs*. Isto é sistematizado através da geração dos *proxies* dinâmicos e o uso automático de invocação dinâmica na interação entre os objetos. Mesmo assim, o usuário não fica totalmente livre de programar aspectos relacionados com o uso do próprio Jini. Por exemplo, o código de uma aplicação-cliente deve conter as rotinas para acessar o localizador de serviços, através da API de *Discovery*, e indagar ao localizador a referência dos serviços que se deseja utilizar. Somente depois destes passos as invocações para o serviço podem ser montadas.

3.4 Arquiteturas de *Software* / Programação por Configuração e ADL

Propostas fundamentadas em AS/PC e ADL, de uma forma geral, consideram a descrição da arquitetura de uma aplicação, independentemente de como esta será implementada, como sendo uma etapa importante na construção de sistemas. Entretanto, como mencionado na seção 2.3.4, alguns autores enfatizam outras características, como aspectos formais e o nível de abstração da ADL.

Robert Allen apresenta, em sua tese [72], uma compilação de linguagens de configuração. 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. Neste contexto, a grande diferença entre os dois grupos, é que as ADLs devem suportar o conceito de arquiteturas de *software*. O outro grupo possui apenas características que

facilitam, de alguma forma, a descrição da topologia das aplicações, mas não contemplam diretamente a descrição de outras características importantes em uma arquitetura de *software* (seção 2.3.4).

Em [15], Judi Bishop e Roberto Faria classificam linguagens de configuração em níveis de abstração, e de acordo com a abordagem utilizada pelas mesmas para tratar conectores. Neste trabalho, Linguagens de Programação como C++ e Ada, por exemplo, são classificadas em um estágio inicial. ADLs como UniCon e Wright estariam no estágio final, oferecendo alto nível de abstração onde uma aplicação pode ser descrita em termos da sua arquitetura.

Nas próximas subseções, apresenta-se a avaliação de duas propostas na área de arquitetura de *software*: (a) Regis, baseada em uma linguagem e um ambiente especial de execução com suporte à configuração, e (b) UniCon, que possui um compilador para a linguagem de configuração, que gera código executável a partir de módulos e descrições de arquiteturas. Além destes, outros projetos importantes podem ser mencionados: (i) Wright [71], que se concentra em oferecer uma base formal para especificação de interações de componentes, através de conectores e estilos de arquiteturas, e (ii) Rapide [130], que enfatiza a especificação do comportamento e a simulação de arquiteturas de *software* para gerar refinamentos sobre as mesmas.

3.4.1 Regis

Regis [131, 74 e 14] é um ambiente desenvolvido no Imperial College, no qual se emprega uma linguagem de 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 topologias de interconexão de componentes dinâmicas;
- primitivas de configuração para instanciação retardada e dinâmica de componentes;
- acesso ao sistema operacional nativo, feito através de um sistema executivo especializado;
- componentes interagem através de primitivas de comunicação especificadas pelo usuário.

Uma aplicação em Regis é descrita através da linguagem Darwin [132], uma linguagem descritiva que permite a estruturação hierárquica de componentes, facilitando a concepção de sistemas com grande número de elementos. Em Darwin, componentes se referem a objetos que realizam a computação da aplicação e a objetos que são utilizados para a comunicação de outros componentes. Componentes se comunicam através de *portas.* Um componente pode prover portas para outros componentes, e/ou pode requerer portas de outros componentes.

Um Exemplo

Na listagem 3.10 apresenta-se a aplicação produtor-consumidor com *buffer* limitado, implementada em Darwin, adaptada de [131].

```
 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;}
```

Listagem 3.10 - *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 entre portas estão implícitas no código do componente. Regis oferece uma biblioteca de classes C++ que, juntamente com o código gerado a partir de uma

descrição em Darwin, implementa as abstrações, como as portas, e a associação destas com mecanismos de comunicação. Por exemplo, se as portas tiverem que ser implementadas por *sockets,* isso seria passado como um parâmetro para o gerador de código. Neste caso, 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 o código que inclui as chamadas ao suporte para a abstração de portas.

Comentários

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 listagem 3.10 nada pode ser dito sobre a sincronização das portas do componente *Buffer*. O tamanho do *Buffer* foi passado como parâmetro e o código foi concebido para prever o limite.

3.4.2 UniCon

UniCon [16] é um projeto do Departamento de Ciências da Computação da Universidade de Carnegie Mellon, baseado em uma ADL, chamada de UniCon. O destaque em UniCon é a utilização explícita de conectores. UniCon é fundamentado 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, como propostos em UniCon, são importantes em arquiteturas de *software* nas quais a maneira como a interação entre os componentes se realiza é relevante.

Um Exemplo

Em UniCon a descrição de componentes e conectores deve ser especificada em detalhes (linha 4 e 5, da listagem 3.11, por exemplo). Informações completas sobre o tipo, a interface e a implementação devem ser explicitamente colocadas na declaração de cada elemento. Estas informações são utilizadas para realizar verificações sobre a arquitetura descrita, e para a geração de código e *scripts* que facilitam o mapeamento da mesma em uma implementação.

```
 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  COMPONENT Producer
18    INTERFACE IS
19      TYPE Module
20      PLAYER Put  IS RoutineCall
21        SIGNATURE ("char *";"void")
22      END Put
23    END INTERFACE
24    IMPLEMENTATION IS
25      VARIANT Producer IN "prod.c"
26        IMPLTYPE (Source)
27      END Producer
28    END IMPLEMENTATION
29  END Producer
30  COMPONENT Consumer
31    INTERFACE IS
32      TYPE Module
33      PLAYER Get  IS RoutineCall
34        SIGNATURE ("char * *"; "void")
35      END Get
36    END INTERFACE
37    IMPLEMENTATION IS
38      VARIANT Consumer IN "cons.c"
39        IMPLTYPE (Source)
40      END Consumer
41    END IMPLEMENTATION
42  END Consumer
43  CONNECTOR C-proc-call
44    PROTOCOL IS
45      TYPE ProcedureCall
46        ROLE definer IS definer
47        ROLE caller IS caller
48    END PROTOCOL
49    IMPLEMENTATION IS
50      BUILTIN
51    END IMPLEMENTATION
52  END C-proc-call
```

```
53 COMPONENT BoundedBuffer
54   INTERFACE IS
55     TYPE Filter
56   END INTERFACE
57   IMPLEMENTATION IS
58     USES Prod INTERFACE Producer
59     USES Cons INTERFACE Consumer
60     USES Buff INTERFACE Buffer
61     USES put-prod PROTOCOL C-proc-call
62     USES get-prod PROTOCOL C-proc-call
63     CONNECT Prod.Put TO put-prod.caller
64     CONNECT Buff.Put TO put-prod.definer
65     CONNECT Cons.Get TO get-prod.caller
66     CONNECT Buff.Get TO get-prod.definer
67   END IMPLEMENTATION
68 END BoundedBuffer
```

Listagem 3.11 - *Buffer* limitado descrito em UniCon

UniCon oferece ferramenta (chamada *uparse*, na versão da implementação testada) que faz uma análise sintática e semântica da configuração descrita. Isto permite a verificação de descrições em UniCon antes de se gerar código. Uma outra ferramenta facilita a construção de aplicações executáveis a partir da descrição de uma arquitetura. Se o conector utiliza algum tipo de inteligência especial, o analisador deve ser preparado para suportar as características adicionais [133].

Comentários

A versão de UniCon avaliada apresenta por volta de 7 tipos de conectores embutidos na linguagem. Entre eles estão: *pipe*, RPC e caixa-postal. Em [133] apresenta-se um roteiro para a adição de novos tipos de conectores ao UniCon, mas o próprio autor ressalta que não é tarefa trivial.

3.5 UML

UML, *Unified Modeling Language* [134], é resultado da convergência e integração de vários métodos de modelagem de sistemas baseados em objetos, como o método de Booch (proposto por Grady Booch), o método OMT - *Object Modeling Technique* (de James Rambaugh) e o método OOSE - *Object-Oriented Software Engineering* (de Ivar Jacobson). Atualmente, UML está incorporada aos padrões da OMG [40, 135].

UML é uma linguagem de modelagem para especificar, documentar, visualizar e construir sistemas, que pode ser usada durante as várias fases do processo de desenvolvimento dos mesmos. UML é independente das tecnologias utilizadas na

implementação dos sistemas modelados. Por isso, UML admite extensões ao modelo, para que se possa adequá-lo aos sistemas em construção e às diferentes tecnologias empregadas.

UML define vários diagramas, tais como: diagramas de classe, diagrama de estados, diagrama de interação, etc. Cada diagrama de UML tem seu contexto de utilização, e geralmente um sistema é modelado através de um subconjunto destes diagramas, cada qual apresentando uma faceta do mesmo. No contexto desta tese, diagramas de UML podem ser eventualmente úteis. Entretanto, o diagrama de componentes e o diagrama de implantação têm maior importância por sua aplicabilidade no projeto e descrição do *software* das aplicações, ao contrário dos outros diagramas, que normalmente são utilizados na etapa de análise.

UML também define a *Object Constraint Language* - OCL [136] uma linguagem para a definição de contratos, que restringem o uso dos métodos através de assertivas. OCL segue o modelo de contratos da linguagem Small Talk, através do qual podem ser especificadas pré-condições, que devem ser atendidas para que o método seja invocado, e pós-condições, indicando restrições no resultado da execução do método ou no estado final do objeto invocado.

Comentários

A nosso ver, UML ainda não oferece uma forma simples para descrever (i) aspectos não-funcionais das aplicações, (ii) características não-funcionais das interações entre elementos e (iii) a topologia de interconexão das aplicações. Por exemplo, nos diagramas de classes e de implantação não se consegue representar facilmente os aspectos de concorrência e sincronização da classe *Buffer,* nem como se dará explicitamente a comunicação entre os componentes.

Em [137], nossa opinião é reforçada, quando se afirma que "… o arquiteto de *software* deve ser capaz de organizar sistemas de *software* e tomar decisões estratégicas de projeto, para atingir os objetivos relacionados com a disponibilidade do sistema, segurança, escalabilidade, sobrevivência, flexibilidade, granularidade, qualidade e manutenção de dados, mecanismos de empacotamento e entrega. Estes pontos não são comumente explorados por UML ou outras abordagens populares de modelagem. Entretanto, estes pontos são críticos para o sucesso de projetos modernos de *software*".

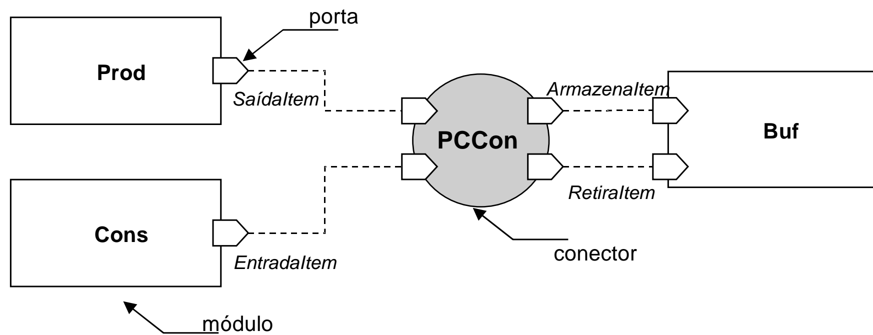


Figura 3.4 - Notação utilizada em R-RIO para a aplicação produtor-consumidor

Em nossa proposta utilizamos uma notação visual, que não pretende ser completa ou constituir um novo padrão, mas permite representar explicitamente a estrutura das aplicações no contexto de ASs. A figura 3.4 apresenta a aplicação produtor-consumidor nesta notação. Embora não apresente vantagens específicas, a notação supre a falta de alguns elementos em UML, como o conector, que é fundamental, de acordo com nossa abordagem, na descrição de aplicações.

Basicamente acrescentou-se uma representação para o conector e explicitou-se cada método implementado ou requisitado por um componente, através de portas de entrada e saída. Os detalhes do modelo de componentes de R-RIO e da notação visual serão apresentados a partir do capítulo 4.

3.6 Conclusão

As avaliações das diversas implementações e ferramentas confirmam o que foi discutido sobre as tecnologias no capítulo 2.

As implementações baseadas em PM-N oferecem separação de interesses na implementação do *software* de aplicações, via reflexão. Entre as ferramentas analisadas, MetaJava mostrou-se abrangente porque estava facilmente disponível, foi integrada a um ambiente já utilizado, Java, 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.

Observou-se que o uso mais adequado da reflexão é no ajuste fino, ou em detalhes, da programação de aspectos não-funcionais das aplicações, *programming **in-the-small*** [80]. Entretanto, a reflexão não facilita a estruturação de aplicações. As

reconfigurações, que devem ocorrer no meta-nível, também não têm suporte explícito e dependem de soluções *ad-hoc*. Verificou-se, ainda, que a disponibilidade de reflexão é dependente de linguagem de programação e /ou ambiente de execução.

Os *middlewares* avaliados apresentam características úteis para a concepção de grandes sistemas distribuídos baseados em objeto. A possibilidade de reutilização de componentes e a facilidade para a comunicação remota dos mesmos são exemplos destas características. Entretanto, a transparência e a separação de interesses para o programador da aplicação, no nível de maturidade atual, ainda são restritas.

Dois exemplos de sistemas representativos de AS/PC, Regis e UniCon, foram avaliados. A avaliação considerou a descrição de arquiteturas e a facilidade para descrever aspectos não-funcionais. Também foram testadas as ferramentas disponíveis para verificar as descrições de arquiteturas. De uma forma geral, foi possível descrever a aplicação-exemplo através de seus componentes e da interligação destes. A "aparência" de uma configuração descrita em Darwin parece ser mais "limpa", se comparada com a descrição em UniCon. Entretanto, UniCon permite que se explicitem características sobre a interligação dos componentes e alguns aspectos não-funcionais. No exemplo desenvolvido, isto se deu através da utilização de um conector, que oferece chamadas a procedimentos, embutido na própria ferramenta de compilação e geração de código (*BUILTIN*). Este grau de liberdade não é suportado diretamente em Darwin.

De forma geral, a utilização mais indicada para as implementações baseadas em AS/PC é a construção de grandes sistemas, em que a estrutura da aplicação é importante. Embora os exemplos implementados não indiquem isso diretamente, em tais sistemas o passo inicial é o projeto da aplicação em termos de grandes blocos, *programming* ***in-the-large***, com o auxílio de uma ADL. Algumas propostas ainda auxiliam no refinamento do projeto, podendo-se chegar, através da ADL, à descrição de módulos funcionais mais próximos de um componente de *software*, segundo a tecnologia BC. Observou-se entretanto, que faltam nas ADLs avaliadas a capacidade de se descreverem aspectos não-funcionais mais claramente e um roteiro para mapear as descrições em implementações concretas.

No capítulo seguinte iniciamos a apresentação de nossa proposta, R-RIO, para a descrição, configuração, execução e manutenção da arquitetura de *software* de aplicações.

Esta página foi intencionalmente deixada em branco

Capítulo IV

O *Framework* R-RIO

4.1 Introdução

Por muitos anos têm ocorrido desencontros entre a abordagem acadêmica e as forças de mercado, no que diz respeito ao desenvolvimento de *software*. O mercado demanda a produção rápida e barata de *software*. Pelo lado acadêmico, modelos para a produção de *software* normalmente obedecem a passos bem definidos, em que cada etapa só é iniciada quando a anterior já foi concluída. Mesmo quando a prototipagem ou o desenvolvimento interativo é permitido, as premissas básicas não mudam: os requisitos e o projeto devem estar bem definidos e documentados antes de se iniciar a implementação.

Aqueles que já se envolveram com desenvolvimento e manutenção sabem que *software* real nem sempre pode ser criado desta forma. Algumas vezes existe a noção do que se quer fazer, mas vários passos errados são dados antes de se apontar para a direção mais acertada. Embora esta realidade não seja universal, ela já tem sido observada há algum tempo. Em *The Mytical Man-Month* [138] é sugerido que se torne prática geral, simplesmente, "jogar fora" a primeira versão de qualquer *software* produzido. Segundo Highsmith [139], "projetos complexos de *software* precisam de uma meta, algumas linhas mestras e, possivelmente, também de algumas regras, mas o sucesso - para o desgosto dos adeptos de processos rigorosos - muitas vezes resulta da efetiva improvisação". As aplicações desenvolvidas para o contexto da Internet, com estruturas e requisitos altamente dinâmicos, são exemplos desta classe de *software*.

Considerando-se os pensamentos acima, e os problemas levantados no capítulo 1, conclui-se que ainda existem lacunas no processo de desenvolvimento de *software*. A proposta apresentada em nossa tese objetiva diminuir tais lacunas.

Neste capítulo, apresentamos R-RIO - *Reflective-Reconfigurable Interconnectable Objects*, um *framework*[5] para a descrição, configuração e manutenção de aplicações. R-RIO difere de outras propostas por integrar os conceitos de Arquitetura de *Software* / Programação por Configuração e Programação de Meta-Nível em um único *framework*. Com esta integração, uma aplicação de *software* é beneficiada pela separação de interesses em várias fases do seu ciclo-de-vida. Em nossa proposta também estão contemplados requisitos, como a facilidade para reutilização de elementos de *software* e o suporte à evolução dinâmica das aplicações. Além disso, o *framework* apresenta significativa abrangência no que diz respeito aos domínios de aplicação que podem ser desenvolvidos, qualidade relevantes discutida no primeiro capítulo.

O *framework* R-RIO é constituído dos seguintes elementos:

1. um modelo de componentes;
2. um modelo de gerência de configuração, que permite a criação, conexão e remoção dinâmica de componentes das aplicações;
3. uma linguagem para descrição de arquiteturas de *software* (ADL), chamada CBabel;
4. um sistema de suporte, que executa o serviço de gerência de configuração;
5. uma metodologia para a configuração de arquiteturas de *software*, que estimula a separação de aspectos funcionais e não-funcionais em componentes diferentes da arquitetura.

As próximas seções estão organizadas da seguinte forma[6]. Inicialmente, os conceitos básicos e elementos do *framework* são apresentados. Destacam-se, neste contexto, algumas características importantes de R-RIO, como a facilidade de composição de componentes, e a integração dos conceitos de programação de meta-nível e programação por configuração. O modelo de configuração é apresentado na seqüência. Discute-se como a separação de interesses é tratada em nossa proposta, e

---

[5] O termo *framework*, usado aqui, tem uma conotação mais abrangente em relação a sua aplicação estrita no contexto da tecnologia de objetos (seção 2.3.1.6 - capítulo 2). No contexto de nossa proposta, um *framework* é um conjunto de elementos que podem ser utilizados separadamente ou em conjunto para a realização de um objetivo.

[6] A ADL CBabel é apresentada, separadamente, no apêndice A.

como o *framework* facilita a configuração de aspectos funcionais e não-funcionais, separadamente. Em seguida, a proposta é situada no contexto do desenvolvimento de aplicações. Sugere-se também uma metodologia para a configuração de aplicações, com o objetivo de se obter a separação de interesses. Um primeiro exemplo é, então, apresentado. O capítulo é concluído com uma discussão sobre a aplicabilidade do *framework*.

4.2 Modelo de Componentes

O modelo de componentes de R-RIO está fundamentado nos seguintes elementos:

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

A arquitetura de *software* de uma aplicação é descrita em R-RIO, através da ADL CBabel, utilizando-se os três elementos citados. Uma descrição de arquitetura típica contém uma seleção de módulos, que irá compor a funcionalidade básica da aplicação, e um conjunto de conectores, para interligar e intermediar a interação dos módulos. A estrutura da aplicação é determinada descrevendo-se, separadamente, as ligações entre os módulos e conectores. Adicionalmente, podem ser descritos aspectos não-funcionais, permitindo que a configuração da aplicação seja ajustada em diferentes aspectos.

4.2.1 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, como por exemplo: clientes, servidores, geradores e tratadores de eventos. Servidores de arquivos, banco de dados ou servidores

HTTP são facilmente representados por um módulo. Outros recursos, como arquivos ou dispositivos de E/S, também podem ser representados por módulos no nível da arquitetura de *software*.

Uma aplicação é composta por instâncias de módulos. Observe-se a figura 4.1. O módulo é o lugar onde estão contidas estruturas de dados e métodos[7]. Em uma instância de módulo, as estruturas de dados locais podem armazenar o estado do mesmo (representado por Var1 e Var2 na figura 4.1). Os métodos de um módulo, similarmente ao conceito de métodos na tecnologia de objetos, implementam sua funcionalidade realizando computações ou manipulando o estado do módulo.

A interação de um módulo com outros componentes é realizada através de pontos de interação chamados de ***portas***. Um método de um módulo pode estar associado a uma porta de entrada, através da qual outros módulos podem solicitar o serviço sendo oferecido (métodos 2, 3 e 4 da figura 4.1). Desta forma, o método passa a ter visibilidade externa. Se um método não estiver associado a uma porta de entrada, será considerado privativo do módulo e não poderá ser invocado diretamente por outros módulos (método 1 da figura 4.1). Uma porta de saída de um módulo pode ser utilizada por qualquer método deste, para solicitar serviços a outros módulos. O conjunto de portas de entrada e saída de um módulo pode ser genericamente referenciado como **interface** ou **assinatura** do módulo.

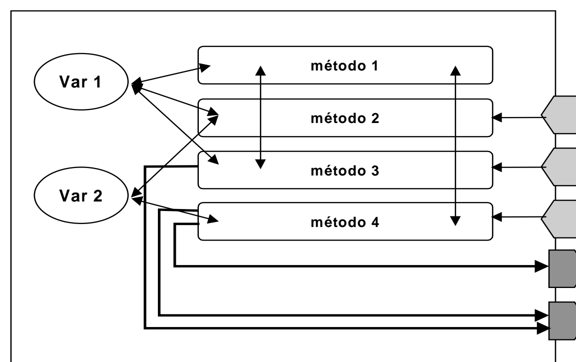


Figura 4.1 - Módulo, porta e métodos

[7] 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.

A princípio, um módulo não precisa ser concebido especificamente, prevendo-se reconfiguração ou evolução dinâmica. O serviço de gerência de configuração, provido separadamente, pode instanciar, bloquear, desbloquear ou terminar um módulo em situações específicas, como será detalhado nas seções seguintes. Entretanto, reconfigurações que precisam preservar a consistência de estado do sistema são fortemente dependentes da aplicação, requerendo, normalmente, suporte específico fornecido pelo programador.

No modelo de componentes de R-RIO, potencialmente, os módulos componentes de uma aplicação executam concorrentemente. O mesmo ocorre em relação aos métodos de um módulo, que, por definição, executam de forma concorrente entre si.

Implementação

O modelo de componentes é, em princípio, independente de linguagens de programação ou ambientes de execução especializados. É importante, entretanto, que uma aplicação descrita através de CBabel possa ser mapeada em programas descritos em uma linguagem de programação.

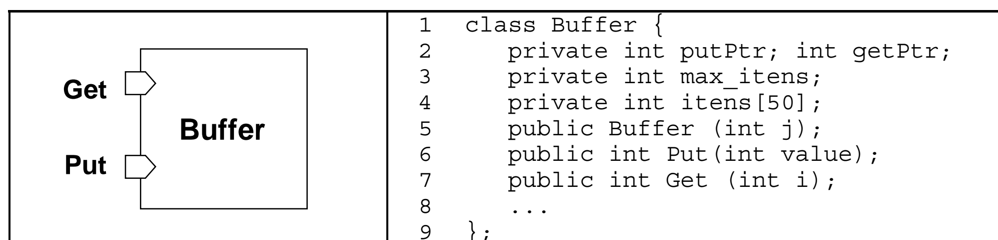


Figura 4.2 - Mapeamento de um módulo

O mapeamento entre o modelo de componentes e a implementação em uma linguagem de programação é conceitualmente simples. Por exemplo, 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 OMG-IDL. O casamento entre um módulo e um objeto, por exemplo, é bastante direto. Um objeto possui métodos e um estado. De forma análoga ao que ocorre nos módulos no nível da arquitetura, os métodos de um objeto são invocados, utilizando-se uma referência ao objeto e o nome dos métodos, juntamente com os valores dos argumentos. A figura 4.2 ilustra este ponto para um módulo *Buffer* com duas portas de entrada, *Put* e *Get*. A

classe *Buffer* descrita em Java (linha 1), contendo dois métodos, *Put* e *Get* (linhas 6 e 7), corresponde a uma possível implementação do módulo *Buffer*.

Tendo em vista a plataforma de execução, a implementação de um módulo pode resultar em formas diferentes. Por exemplo, se executado diretamente sobre um sistema operacional Unix, um módulo pode corresponder a um *shared object* [140]. Da mesma forma, um módulo pode tomar a forma de uma *DLL* em uma plataforma Microsoft Windows, ou um *ByteCode*, se uma *Máquina Virtual Java* for usada como suporte. Os sistemas de suporte atuais permitem a carga e ligação dinâmica de unidades básicas de execução. Isto facilita tanto o mapeamento dos módulos, como a gerência da configuração das arquiteturas de *software* e a evolução dinâmica das mesmas durante sua execução.

4.2.2 Portas

Uma ***porta*** é um elemento auxiliar do modelo de componentes, que representa um ponto de interação de um módulo com outros módulos de uma aplicação. Uma porta pode ser de entrada ou de saída, como elucidado na seção anterior. Em CBabel, 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 arquitetura de *software* e a implementação, como será discutido nos próximos parágrafos.

Seguindo o exemplo 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, o tipo do resultado ou retorno. 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 for necessário apresentar algum.

Quando um módulo precisa solicitar serviços a outros módulos, ele utiliza 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.

Quanto ao fluxo de informações, uma porta pode ter um único sentido, apropriada para interações que não envolvam retorno de resultados, ou ter sentido duplo, apropriada para interações em que haja retorno de resultados. Isto caracteriza a interação através de uma porta como assíncrona ou síncrona.

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 passados na requisição pelo módulo origem e do estado do módulo destino.

Geralmente, uma porta de saída deve ter a assinatura compatível com a porta de entrada com a qual vai interagir. Assim, garante-se 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 portas de saída tornarem 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 a obtenção de informações mais completas sobre a arquitetura sendo descrita. Por exemplo, a descrição de uma classe abstrata em C++, ou a descrição de uma interface em OMG-IDL contempla apenas os métodos implementados pelo objeto (seria o equivalente às portas de entrada). Não existe, nestes exemplos, uma forma de se indicarem explicitamente, na interface, as interações que partem de um objeto.

Implementação

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

Em linguagens de programação orientadas a objetos, um mapeamento imediato para a porta de entrada é o método. 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.

Uma porta de saída é mapeada, de forma simétrica à porta de entrada, em invocações a métodos ou chamadas a procedimentos. O nome da porta é mapeado na referência a um método a ser invocado. 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 ser compatível com a classe do mesmo. A referência para instância deste objeto não é importante neste contexto, pois esta só será conhecida quando as respectivas portas forem ligadas durante a configuração. Neste momento, as referências das portas são cruzadas e possivelmente renomeadas pelo suporte à configuração (seção 4.3).

O trecho de programa a seguir (listagem 4.1) 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 portas 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); }
```

Listagem 4.1 - Módulos cliente e servidor em Java

No caso em que a linguagem de programação não suporta o conceito de objeto, uma porta pode ser mapeada em um de procedimento ou função, disponíveis nas linguagens de programação tradicionais como C ou Pascal.

É importante notar que existem outras alternativas para o mapeamento das portas. Estas alternativas geralmente requerem a participação explícita do programador. Por exemplo, em Regis, avaliado na seção 3.4.1 - capítulo 3, as portas da linguagem de configuração Darwin, são mapeadas explicitamente no código, na forma de objetos instanciados a partir das classes especiais *port* e *portref.* Em [79], as portas são mapeadas como estruturas de dados de um ambiente especializado, as quais 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 com a programação das portas e o uso de primitivas do tipo *send* e *receive* para realizar a interação entre os módulos.

4.2.3 Conectores

Um conector é um elemento que oferece a abstração necessária para determinar-se, no nível da arquitetura de *software*, a topologia de interligação de módulos. Conectores são responsáveis, primariamente, por encaminhar requisições que devam ir de uma porta de saída de um módulo para uma porta de entrada de outro módulo, e pelo retorno dos resultados no sentido contrário, quando necessário. Adicionalmente, conectores podem encapsular aspectos não-funcionais, a serem efetivados quando em operação.

Para realizar a interligação de módulos, o conector desempenha outro papel importante, que é o casamento de referências das portas de entrada e saída no nível da arquitetura. Na seção anterior, caracterizou-se a porta como referência a pontos de interação nos módulos. Observa-se, entretanto, que os módulos são entidades autônomas. A única informação de como um módulo irá interagir com outros módulos é o seu conjunto de portas. Não se sabe, *a priori*, com que classes de módulo estas interações se darão. Sabe-se, apenas, quais são os tipos dos argumentos que transitarão durante as interações, por conta do conjunto de tipos dos parâmetros definidos nas portas. Em nosso modelo, o conector é o elemento que acopla estas referências, no nível da arquitetura, interligando portas de entrada a portas de saída.

Conceitualmente, um conector também possui portas de entrada e saída. O papel das portas em relação ao conector é similar ao destas em relação ao módulo: são referências a pontos de interação. Observa-se que, no conector, as portas são geralmente configuradas aos pares, de forma que as requisições possam ser encaminhadas de uma porta de entrada para uma porta de saída do mesmo.

As interligações de módulos são descritas explicitamente em CBabel, através de declarações de *ligação*. Em uma declaração de ligação são indicados os módulos que precisam interagir e o conector a ser utilizado para interligá-los. Com isto, as referências a portas de entrada e saída dos módulos envolvidos passam a ser conhecidas e podem ser acopladas pelo conector.

Em uma declaração de ligação pode-se fazer referência aos módulos, ou às portas de saída e entrada sendo interconectadas (detalhes da sintaxe em CBabel encontram-se no apêndice A). 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 situação, ao se estabelecer uma ligação, portas de saída de um módulo cliente são ligadas a portas de entrada de um conector, e as portas de saída deste conector são ligadas a portas de entrada de um módulo servidor. Não é necessário que os nomes das portas sendo ligadas sejam os mesmos, mas isso poderá facilitar ligações automáticas baseadas no contexto dos módulos envolvidos. O suporte à configuração também pode realizar ligações baseadas no tipo das portas e pode facilmente renomear os identificadores de portas, se for conveniente.

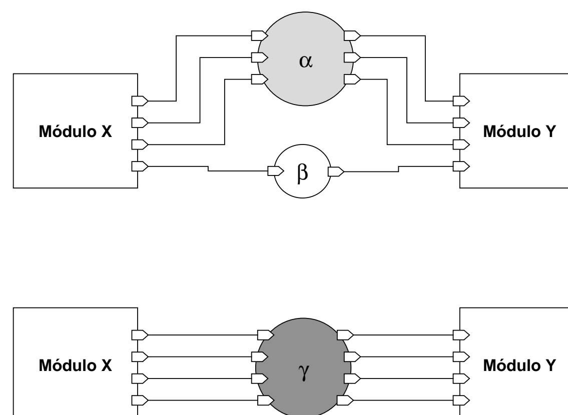


Figura 4.3 - Interligando módulos (a) por conectores distintos (b) usando um único conector

As figuras 4.3(a) e 4.3(b) apresentam duas configurações simples. Na figura 4.3(a), a ligação dos módulos *X* e *Y* foi feita, utilizando-se as referências de porta com conectores diferentes (α e β), e na figura 4.3(b), todas as portas destes módulos foram ligadas pelo mesmo conector (γ). A utilização de um ou outro estilo de ligação depende da conveniência para o projetista da aplicação.

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, o conector pode controlar as interações entre módulos, interceptando e manipulando requisições e respostas. Com isso, é possível impor um comportamento diferente ao encaminhamento das requisições. 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. Ao adotar-se esta metodologia obtém-se a capacidade de se adicionar novas características aos módulos que o conector interconecta, e a possibilidade de atuar no ajuste da arquitetura da própria aplicação. Estas questões serão aprofundadas na seção 4.5 e ao longo do capítulo 5.

Os conectores têm papel significativo na estruturação das aplicações, entretanto não participam obrigatoriamente das funções básicas das mesmas. Observa-se que a atuação de um conector ocorre em um meta-nível e torna evidente a reflexividade deste elemento. Um conector pode ser considerado como um módulo especializado em determinar uma composição de módulos e em promover a interação entre os mesmos. Entretanto, a separação dos conceitos de módulos e conectores se justifica com (i) a possibilidade de se associarem formalismos para a verificação automática de propriedades das aplicações sendo configuradas [141], e (ii) pelo tratamento diferenciado dado, para cada elemento, pelo suporte à configuração (ver seção 7.2, e discussão na seção A.2.4.2 - apêndice A).

Implementação

O mapeamento de um conector para o nível da implementação não é único. Um conector pode ser mapeado, por exemplo, em uma estrutura de dados, um procedimento ou um objeto. Assim, um conector pode assumir a forma de uma simples consulta a uma

tabela, para resolver uma referência, seguida de uma solicitação de reserva de recursos. É possível, também, que um conector tenha que prover acesso a mecanismos do sistema de suporte, como canais de comunicação, para o caso em que os módulos estejam distribuídos por uma rede.

Na listagem 4.1, no contexto de uma linguagem de programação orientada a objeto, o conector que interliga o módulo cliente ao módulo servidor é implementado (i) criando-se no corpo do objeto cliente uma instância do objeto servidor (linha 11), e (ii) utilizando-se esta instância e as assinaturas de seus métodos para construir as invocações. Neste caso, o mapeamento do conector em uma implementação é a referência à instância do objeto servidor, criada no corpo do módulo cliente, e a resolução de referências é feita pelo compilador ou suporte em tempo de execução.

A implementação de um conector também pode assumir a forma de um objeto independente. Este será um intermediário entre outros objetos, que implementam módulos. Tipicamente, um objeto implementando um conector deverá ser concebido de forma que (i) ofereça um conjunto de métodos compatíveis com os aqueles invocados pelo módulo cliente (que contém portas de saída) e, (ii) invoque métodos com parâmetros compatíveis com os módulos servidores (que contêm portas de entrada). O trecho de código da listagem 4.2 ilustra este ponto, estendendo o exemplo da listagem 4.1. Na classe *Connector* são oferecidos dois métodos compatíveis com os métodos invocados pela classe *Cliente* (linhas 4 e 7) que, por sua vez, invocam os métodos da classe *Servidor*, e repassam o retorno de volta ao cliente (linhas 6 e 9). 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 4.2 - Código do conector em Java

Os módulos *Cliente* e *Servidor* (definidos na listagem 4.1) e o conector *Connector* (definido na listagem 4.2) podem ser utilizados em um exemplo de aplicação

mais completo. Na descrição da configuração deste exemplo os módulos seriam instanciados e, em seguida, interligados pelo conector. Um *middleware* de suporte à configuração é responsável por resolver e ligar as referências contidas nestes componentes, no nível da implementação, para que se imponha a arquitetura da aplicação descrita. Uma das formas de se realizar esta ligação é a renomeação de algumas referências na implementação do conector, como descrito em [142], de modo que o processo seja transparente à implementação dos módulos.

Ainda com relação ao exemplo acima, no corpo de cada método do conector é possível adicionar código para implementar alguns aspectos não previstos nos módulos. Por exemplo, seria simples introduzir, entre as linhas 4 e 6, uma pequena rotina para verificar os limites do parâmetro *i* no método *Service1*.

Se um conector interliga módulos instanciados em nós distribuídos, é necessário que ele seja implementado por partes diferentes, as quais também serão executadas em nós distribuídos. Além disso, para que a comunicação entre as partes do conector possa ocorrer, é necessário utilizar algum mecanismo de comunicação, geralmente disponível no sistema de suporte. Detalhes sobre a o mapeamento de conectores para uma implementação podem ser consultados nas seção 7.3.1 - capítulo 7, e na seção A.2.5 - apêndice A.

4.2.4 Composição de Módulos

O *framework* R-RIO integra o conceito de módulo composto. Um módulo composto é formado por módulos primitivos, isto é, que possuem um mapeamento para uma implementação, ou por 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 é formado pelo conjunto, ou um subconjunto, das portas dos módulos componentes. Na descrição do módulo composto, este subconjunto deve ser selecionado e marcado como tendo visibilidade externa. Para o resto da aplicação, um módulo composto tem a mesma apresentação de um módulo primitivo, e somente se relaciona com este através de suas portas. Sob tal perspectiva, a arquitetura de uma aplicação também é considerada um módulo composto.

A figura 4.4 apresenta um exemplo de aplicação contendo um módulo composto (MC1) formado por três módulos primitivos (Mp1, Mp2 e Mp3). Na figura também

estão identificadas duas portas selecionadas para terem visibilidade externa. Os outros módulos da aplicação têm acesso ao módulo composto apenas através destas duas portas.

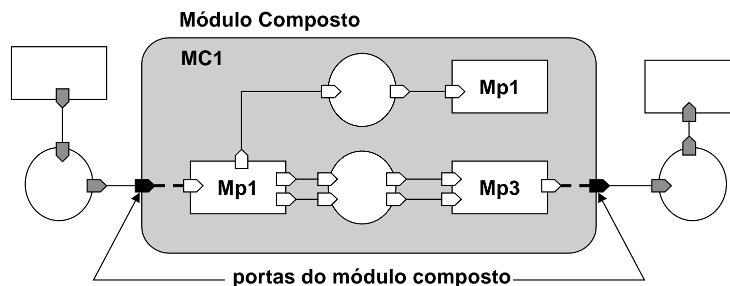


Figura 4.4 - Aplicação com módulo composto

A composição de módulos pode ser usada para a construção de módulos com função agregada ou mais especializados. Agregando-se adequadamente, por exemplo, um *módulo de áudio* e um *módulo de vídeo*, pode-se construir um *módulo de mídia* composto. No mesmo exemplo, o *módulo de áudio* pode constituir um módulo especializado, composto por um *módulo fonte de áudio* e um *módulo codificador* [93]. A composição de módulos também pode ser usada para a criação de conjuntos ou bibliotecas de componentes especializados, que sejam reutilizáveis em outras aplicações.

4.2.5 Composição de Conectores

A configuração de um conector composto, através da combinação de conectores mais simples, também é possível. Uma composição de conectores é similar a uma composição de módulos. A principal diferença é que estruturalmente um conector composto pode ser formado apenas pela ligação de conectores primitivos, como será explicado adiante.

Recorremos novamente ao exemplo de uma aplicação com dois módulos e um conector. Na figura 4.5, 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 portas 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 interações entre os dois módulos.

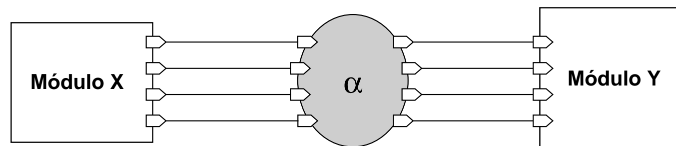


Figura 4.5 - Conector interligando vários serviços do módulo Y às requisições do módulo X

Se ajustes finos se fizerem necessários, individualmente, para cada par de portas que vão interagir, o conector α da figura 4.5 pode ser substituído por um novo conector implementado especificamente com este objetivo. Entretanto, este novo conector poderia ser obtido pela composição de conectores elementares que, em conjunto, promovam os ajustes desejados. Tal conector é exemplificado na figura 4.6.

Observa-se, na figura 4.6, que às portas de saída do conector α, original, foram ligadas aos conectores β, χ, δ e ε. Cada um dos conectores selecionados pode encapsular aspectos não-funcionais diferentes. Por exemplo, o conector β poderia utilizar o protocolo UDP e o conector χ, o protocolo RTP. Ligado ao conector χ, encontra-se o conector ϕ. Este último, por exemplo, poderia comprimir o fluxo de informações. O resultado, portanto, é um conector, referenciado como um único componente, adaptado exatamente às necessidades da aplicação, construído a partir de outros conectores disponíveis.

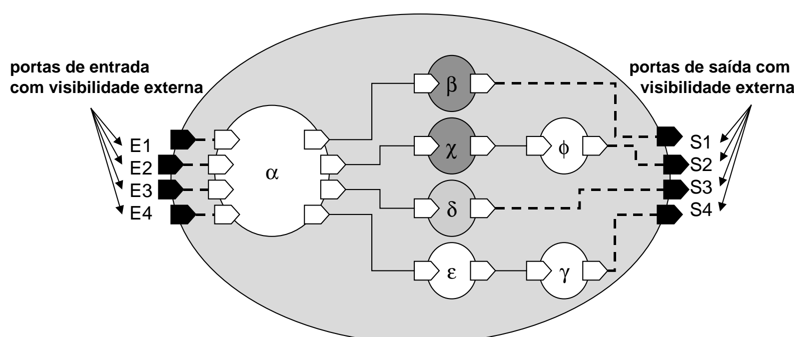


Figura 4.6 - Conector composto

Em CBabel, a declaração de uma composição de conectores é similar à composição de módulos. Na descrição de um conector composto, declaram-se instâncias de conectores mais elementares, 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.

Embora o conceito da composição de conectores seja simples, existem algumas questões que devem ser consideradas:

- é necessário manter a ortogonalidade, ausência de conflitos ou interações indesejadas entre os aspectos sendo compostos, para garantir a coerência do resultado [36, 143, 144];
- deve-se verificar se os aspectos configurados, tomados em composição, têm suporte no sistema onde a aplicação será executada;
- seria interessante dispor de ferramentas para facilitar a composição automática de conectores, a partir de características descritas na configuração.

Além destas questões, existem, também, considerações sobre a eficiência. Conceitualmente, o comportamento de um conector composto é idêntico ao de um único conector que contenha as mesmas características reunidas. Entretanto, dependendo do mapeamento de cada conector componente para uma implementação, podem ser identificadas algumas fontes de ineficiência. Por exemplo, supondo-se que os conectores componentes sejam implementados por objetos, haverá tempo consumido com invocações de métodos, transferências de dados entre espaços de endereçamento diferentes e escalonamento não coordenado. Por outro lado, é possível prototipar inicialmente uma aplicação com conectores compostos. Em seguida, caso o resultado seja satisfatório, pode-se desenvolver um único conector com todas as caraterísticas desejadas.

A composição e a compactação de conectores poderiam ser, potencialmente, automatizadas por uma ferramenta especializada ([145], por exemplo). Entretanto, a implementação de tal ferramenta não seria trivial. Verifica-se, porém, que é possível a realização de algumas composições, sem aparentes problemas. Por exemplo, a composição automática de um conector que encapsule aspectos de coordenação e de um outro, configurado para realizar comunicação, não oferece dificuldades. O mesmo pode-

se dizer para a compactação destes conectores. Outras questões relativas à composição de conectores são aprofundadas na seção A.6, apêndice A.

4.3 Modelo de Gerência de Configuração

Em nossa proposta, a atividade de configuração é empregada em dois momentos. No primeiro, utilizando-se a ADL CBabel, módulos são selecionados e ligados para interagir através de conectores. Chamamos a esta atividade de Programação por Configuração (PC), como já comentado nos primeiros capítulos, pois o projetista configura uma arquitetura de *software* para uma aplicação e para atender determinados requisitos de operação. Em um segundo momento, também se diz que a aplicação está sendo configurada, ao colocar a arquitetura de *software* em execução. Durante esta atividade, os módulos e conectores da arquitetura são carregados e iniciados.

Em nossa proposta, o modelo de gerência de configuração pressupõe que todos os componentes (módulos e conectores) podem ser gerenciados por comandos de (i) criação e remoção de instâncias; (ii) ligação entre componentes, e (iii) início, bloqueio e retomada de execução de um componente. O modelo também pressupõe que o efeito produzido por um comando de configuração é atômico e consistente (não leva a aplicação ao *deadlock* ou a estados não desejáveis).

O *framework* R-RIO possui um serviço de gerência de configuração que é utilizado para criar imagens executáveis de arquiteturas de *software* e reconfigurar arquiteturas em execução. Este serviço apresenta um arquitetura em dois níveis: um **executivo** e um **gerente**. O executivo executa efetivamente os comandos de configuração, oferecendo os métodos básicos para o controle de configuração. O gerente de configuração executa funções de configuração de mais alto nível. O gerente é baseado em um interpretador, que recebe especificações de arquiteturas e gera invocações ao executivo para a execução de comandos de configuração. O gerente também pode interpretar *scripts* de reconfiguração, para alterar a estrutura de uma arquitetura durante sua execução. Como resultado, as instâncias dos componentes da arquitetura são criadas, ativadas ou reconfiguradas.

A arquitetura e um protótipo do *middleware*, que oferece o serviço de gerência

de configuração, são apresentados no capítulo 7. A proposta de uma API para o acesso a este serviço também é discutida neste capítulo.

4.4 Combinando Reflexão e Configuração para a adaptação de Arquiteturas de *Software*

No decorrer da elaboração de nossa proposta, constatou-se que alguns conceitos das tecnologias de AS/PC e PM-N possuem interseções e se complementam, o que facilita a combinação das mesmas. A tabela 4.1 sumariza as correspondências entre as abstrações respectivamente utilizadas em reflexão computacional e no modelo de componentes de R-RIO.

Tabela 4.1 - Reflexão e o modelo de componentes de R-RIO

Reflexão	AS/PC	Correspondência
Pontos de reificação e reflexão	Portas	Portas definem pontos de reificação para os conectores e de reflexão para os módulos
Componentes de nível-base	Módulos	Módulos encapsulam interesses básicos de uma arquitetura
Componentes de meta-nível	Conectores	Conectores funcionam como elementos de meta-nível de uma arquitetura

Integrando-se as duas tecnologias, soma-se (i) a capacidade de descrever e verificar arquiteturas de *software*, e tratar reconfigurações dinâmicas, de AS/PC, à (ii) flexibilidade para tratar separadamente diferentes aspectos da aplicação, de PM-N, como acontece no mecanismo de reflexão. Isto ajuda a obtenção de separação de interesses e pode facilitar a reutilização de *software*.

A correspondência e a integração das abstrações de reflexão computacional e do modelo de componentes de R-RIO são utilizadas para a **adaptação de arquiteturas de *software***. Em nossa proposta, durante a configuração de uma arquitetura de *software*, o projetista pode selecionar um conector específico para adaptar a arquitetura. Este conector pode ser programado para atuar sobre o fluxo das requisições, em um meta-nível, inserindo características não-funcionais, sem interferir diretamente sobre a computação dos módulos. O uso da reflexão computacional, desta forma, é disciplinado e facilitado através da programação por configuração.

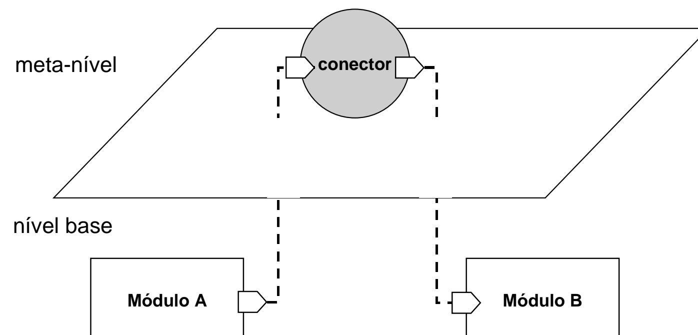


Figura 4.7 - Conector como elemento de meta-nível

A arquitetura da figura 4.7 auxilia o entendimento do parágrafo anterior. Nesta arquitetura, os módulos, considerados componentes funcionais, estão no nível-base. O conector, considerado componente não-funcional, está no meta-nível. Sob o ponto de vista do nível-base, o módulo *A* interage com o módulo *B* diretamente. Entretanto, no início de uma interação, o fluxo de informações pode ser interceptado, ao passar pela porta de saída do módulo *A*, e redirecionado para o meta-nível, representado pelo conector. Este pode inspecionar e manipular as informações contidas no fluxo de uma interação. Tais informações estão disponíveis ao conector, por definição do modelo de componentes de R-RIO. O fluxo de informações, reificado para o conector, pode ser refletido de volta para o nível-base através da porta de entrada do módulo *B*. A transição do fluxo de informações entre os dois níveis não requer nenhuma programação adicional. A configuração da arquitetura define automaticamente os pontos onde estas transições ocorrem, tornando a programação de meta-nível mais simples.

Aspectos não-funcionais específicos podem ser introduzidos na aplicação através desta técnica. Por exemplo, é possível configurar guardas de sincronização (seção 5.3.3, capítulo 5) dentro de um conector, para adicionar restrições de coordenação aos métodos que estão sendo invocados durante a interação dos módulos. A configuração de aspectos não-funcionais nos conectores será discutida no capítulo 5.

4.4.1 Outros usos de Reflexão combinada à Configuração

Além da adaptação de arquiteturas, empregamos em nossa proposta o mecanismo de reflexão, combinado à configuração, de outras formas:

- a reflexão estrutural é utilizada para a gerência de configuração. Esta forma, chamada de **arquitetural** em R-RIO, permite a manutenção, no meta-nível, de

informações sobre as arquiteturas de *software* instanciadas. Isto facilita a adaptação dinâmica das mesmas para atender a novos requisitos.

- em uma outra forma, a reflexão estrutural é utilizada com o fim de adaptar dinamicamente a implementação de um conector ao contexto das instâncias dos módulos funcionais, cujas interações ele deve mediar. Isto facilita o reuso e a adoção de *middlewares* "de prateleira" (*off-the-shelf* - OTS). Embora seja uma aplicação de reflexão em um outro nível de R-RIO, comparando-se com a anterior, esta forma, chamada de **contextual**, também é facilitada pela correspondência entre as abstrações da reflexão e o modelo de componentes.

4.4.1.1 Reflexão Arquitetural

Em nossa proposta, uma modalidade de reflexão estrutural, que chamamos de reflexão arquitetural, facilita o monitoramento e a reconfiguração de uma arquitetura de *software* em execução[8]. Informações de meta-nível, mantidas disponíveis para consulta, refletem o estado atual da arquitetura e sua estrutura de módulos e conectores. Através dos comandos de configuração esta arquitetura pode ser modificada, e as informações de meta-nível sobre a nova arquitetura são dinamicamente atualizadas.

A reflexão arquitetural disponibiliza as informações necessárias para que decisões sobre a alteração da arquitetura possam ser tomadas. As mudanças que precisam ocorrer na arquitetura, durante uma reconfiguração, são efetivamente executadas pelo serviço de gerência de configuração. Esta conjugação de facilidades (o acesso a informações do que adaptar, e os mecanismos para realizar tal adaptação) não é encontrada facilmente em outras propostas. No caso de propostas que oferecem apenas o suporte à reflexão, realizar alterações na aplicação em execução pode se tornar tarefa difícil e geralmente são codificadas dentro dos objetos. Por outro lado, no caso de propostas que oferecem apenas um sistema de suporte à configuração, as informações do estado da arquitetura precisam ser mantidas e disponibilizadas pela própria aplicação, o que pode representar tarefa complexa para o programador. A técnica de uso combinado de reflexão e programação por configuração, facilita, assim, a evolução dinâmica das aplicações.

---

[8] O termo reflexão arquitetural é empregado em [180] em um contexto similar.

4.4.1.2 Conectores reflexivos por contexto

Como parte das pesquisas desta tese, investigou-se uma técnica para padronizar a implementação de conectores. Foi observado que a maior parte das funções realizadas por um conector são recorrentes e podem ser modularizadas. Como resultado, propôs-se uma técnica que torna os conectores, no *framework* R-RIO, genéricos e reflexivos por contexto.

Quando *plugado* entre dois módulos que precisam interagir, um conector pode ser adaptado, automática e dinamicamente, para mediar a interação entre portas com assinaturas compatíveis. Esta capacidade é obtida utilizando-se a reflexão arquitetural para consultar as assinaturas definidas no conjunto de módulos que ele vai mediar. No momento da configuração, a interface do conector é adaptada, segundo o contexto das informações consultadas. Isto permite que os aspectos não-funcionais encapsulados em um conector sejam programados sem dependências de interface, tornando-os genéricos. Com esta abordagem, não é necessária a geração estática ou o armazenamento de *stubs*, nem a codificação de um conector para cada par de portas que possam eventualmente ser conectadas.

A figura 4.8 ilustra a utilização de um conector genérico para mediar a interação entre dois pares de módulos com interfaces diferentes.

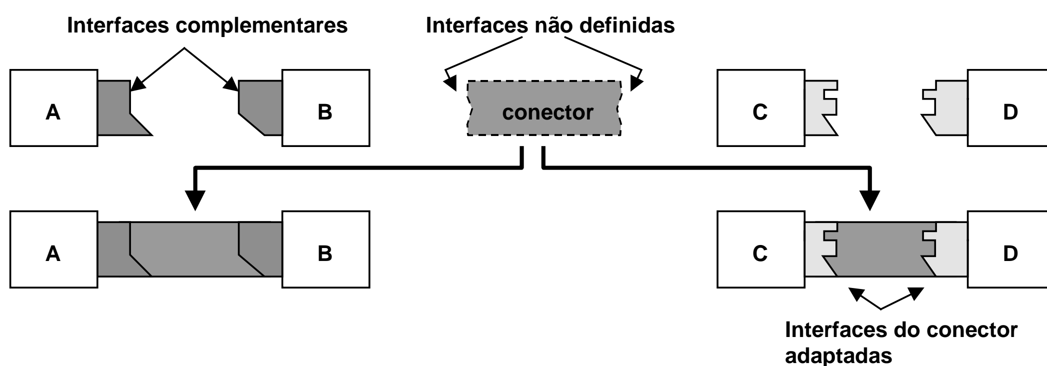


Figura 4.8 - Exemplo do uso de conectores reflexivos por contexto

Em nossos experimentos, tomando por base uma mesma aplicação distribuída, foi possível testar conectores genéricos que suportam mecanismos de comunicação, tais como: RMI, *sockets*, comunicação CORBA e *multicast*. Da mesma forma, cada um dos conectores genéricos foi utilizado em aplicações diferentes, sem a necessidade de qualquer codificação extra. A técnica de conectores reflexivos por contexto ainda pode

ser utilizada no desenvolvimento de conectores genéricos, para encapsular mecanismos que suportem funções de criptografia/decriptografia ou compressão/descompressão, por exemplo.

Em um outro experimento [146], foi desenvolvido um conector genérico que encapsula o *design pattern Observer*. Parte deste experimento é discutido na seção 8.4 - capítulo 8. Em [147], o mesmo experimento foi ampliado para outros *design patterns* de interação como *memento* e *chain-of-responsibilities*.

Um tipo de conector reflexivo por contexto é também proposto, em [148], para suportar evolução de arquiteturas de *software*. Neste projeto, o conector é concebido para ligar componentes, utilizando um estilo de arquitetura específico, chamado C2, que impõe uma estrutura particular para a aplicação. As principais funções deste conector são o roteamento e a difusão de mensagens. As mensagens podem ser filtradas internamente pelo conector, de acordo com algumas políticas específicas. Além disso, a implementação proposta para estes conectores não suporta alterações dinâmicas em uma configuração.

Em [149] identificaram-se alguns *design patterns* para a interação de componentes em sistemas distribuídos, no contexto do projeto Regis. O foco destes *patterns* está nos mecanismos de localização e referência para o suporte de interações, mas a introdução de aspectos não-funcionais não é contemplada.

4.5 Metodologia para a configuração de arquiteturas de *software*

Os conceitos básicos na área de arquiteturas de *software* estão sendo propostos há algum tempo [150, 14, 70, 17, 73]. Entretanto, apenas recentemente começaram a ser empregados no processo de desenvolvimento de sistemas. Arquiteturas de *software* também passaram a ser consideradas na engenharia de requisitos [151], em conjunto com a tecnologia de objetos. Com isto diminui-se a lacuna existente entre o modelo conceitual de uma aplicação, o conjunto de classes a ser utilizado na implementação da aplicação, e o efetivo mapeamento das classes em uma implementação.

Em nossa proposta, consideramos que as atividades de descrição e configuração da arquitetura de *software* (seção 4.3) constituem uma etapa importante no ciclo-de-vida de uma aplicação. Levamos em conta, também, que a arquitetura de uma aplicação é passível de evolução e reconfiguração dinâmicas. Assim, a etapa de descrição /

configuração pode ser revisitada sempre que necessário. A figura 4.9 ilustra como isto ocorre, e ajuda a elucidar os próximos parágrafos.

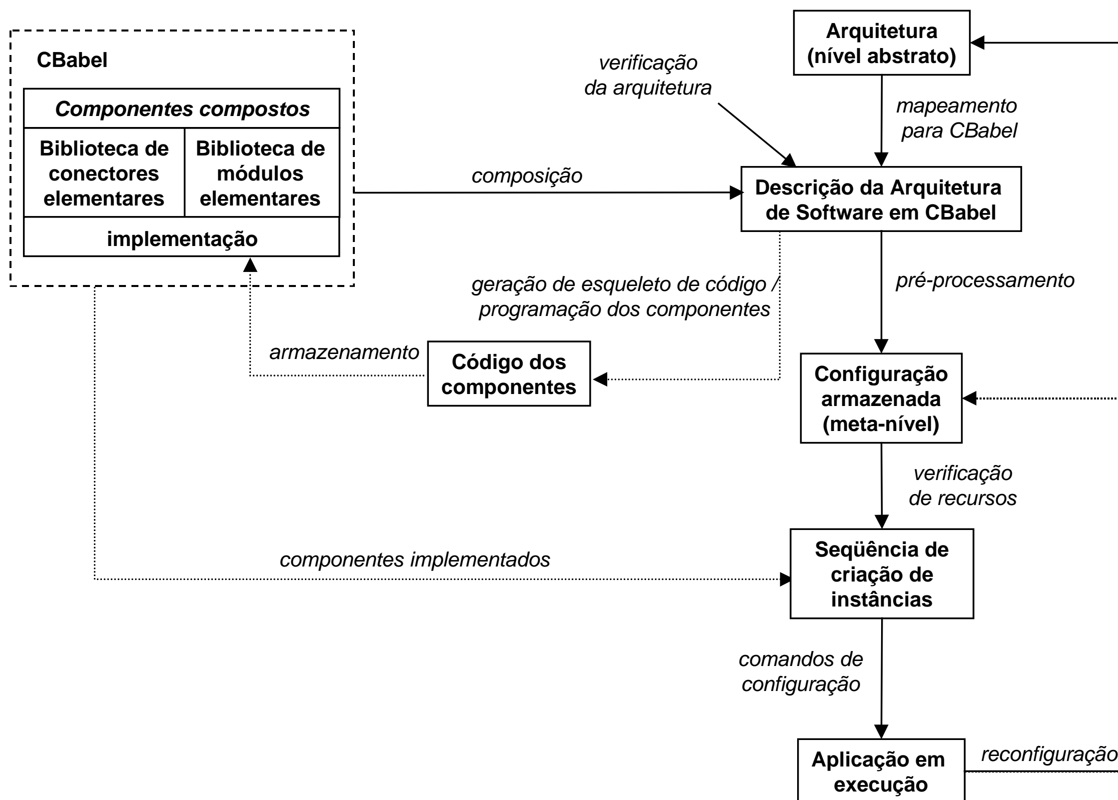


Figura 4.9 - Descrição e configuração de arquiteturas de *software*

A demanda pela concepção de uma aplicação é originada por um usuário potencial. Inicialmente, a aplicação é apenas uma idéia. Antes de iniciar a descrição da arquitetura de *software* de uma aplicação, o projetista precisa ter uma visão global da aplicação a ser projetada.

Em seguida, um levantamento de requisitos pode ser feito, com o auxílio de técnicas disponíveis [152], e um modelo abstrato da aplicação pode ser elaborado. Este poderá ser refinado, resultando em uma arquitetura para o *software* da aplicação. Assim, o modelo da aplicação é traduzido em um conjunto de módulos e conectores, e em uma topologia de interligação destes elementos. A configuração da arquitetura da aplicação é descrita através de CBabel. Para compor a arquitetura, o projetista poderá reutilizar módulos e conectores previamente descritos e implementados, se disponíveis em bibliotecas.

A descrição em CBabel pode ser compilada e verificada. Isto permite obterem-se informações sobre algumas propriedades da aplicação em um estágio inicial do seu

ciclo-de-vida [153].

No caso em que os componentes utilizados na configuração ainda não tenham sido implementados, é possível gerar esqueletos de código em uma linguagem de programação selecionada. A partir dos esqueletos de código, o código dos componentes deve ser programado. Se uma ferramenta para a geração automática de esqueletos de código não estiver disponível, a implementação de componentes pode ser feita inteiramente de forma manual, seguindo-se o modelo de componentes de R-RIO.

O carregamento de uma aplicação para a sua execução se dá em etapas:

- a descrição da arquitetura em CBabel é pré-processada para ser armazenada no repositório de meta-nível do serviço de gerência de configuração. Nesta etapa também poderia ser verificado se os recursos necessários para instanciar a arquitetura descrita estão disponíveis.
- em seguida, ocorre a criação das instâncias dos módulos e conectores, carregando-se as implementações dos componentes de acordo com o mapeamento descrito em CBabel. Quando esta etapa for concluída, os módulos estão prontos para iniciar sua execução.

Durante a execução de uma aplicação é possível que sua arquitetura necessite ser reconfigurada para entrar em um modo de operação específico, ou para adaptar-se a novos requisitos. Neste caso, uma nova etapa de descrição/configuração é realizada. Durante a operação de reconfiguração, componentes disponíveis na biblioteca podem ser utilizados, caso contrário, a implementação de novos componentes pode ser necessária. A arquitetura resultante de uma reconfiguração deve ser refletida no repositório de meta-nível.

As etapas de implementação, teste e manutenção do ciclo-de-vida de uma aplicação serão facilitadas se sua arquitetura de *software* for configurada atendendo-se ao princípio da separação de interesses. CBabel facilita a descrição de arquiteturas de *software*, segundo este princípio, mas não impõe a separação de interesses por si só. O mesmo se dá com a evolução dinâmica e o reuso. Assim, é necessário que haja uma disciplina no uso do *framework* para se obterem as vantagens potencialmente oferecidas. Neste contexto, propomos uma metodologia para a configuração de arquiteturas de *software* que incentiva (i) a modularidade na concepção dos elementos

básicos, e (ii) a separação explicita entre a criação de componentes e as interações dos mesmos, fundamentos para se obter a separação de interesses. Esta metodologia é apresentada através de um conjunto de recomendações (algumas destas já introduzidas nas seções anteriores), enumeradas a seguir:

1. Identificar os aspectos funcionais e não-funcionais da aplicação e verificar se estes podem ser separados.
2. Tanto quanto possível, concentrar os aspectos funcionais em módulos.
    - Verificar se existem módulos mais elementares que, compostos, podem oferecer a funcionalidade desejada.
3. Tanto quanto possível, concentrar aspectos não-funcionais em conectores.
    - Verificar se existem conectores mais elementares que, compostos, podem oferecer as características ou estilo de interação desejado.

Em princípio, seguindo as recomendações citadas, consegue-se retirar dos módulos a responsabilidade de implementar e mediar as interações, concentrando tais aspectos nos conectores. Esta separação de interesses aumenta a possibilidade de reuso dos módulos funcionais e facilita a adequação dos conectores para diferentes aplicações.

A aderência à metodologia apresentada é necessária, mas o projetista tem a liberdade de decidir como selecionar módulos e conectores, como formar composições a partir destes elementos, e como separar aspectos funcionais de não-funcionais. Existe a flexibilidade para a tomada destas decisões de projeto, mesmo porque, aspectos funcionais em uma aplicação podem ser considerados não-funcionais em outra. O uso desta metodologia, embora não obrigatória, potencializa a reusabilidade dos módulos e conectores, além de ajudar a obter o melhor proveito do *framework*.

Algumas implicações de ordem prática no emprego desta metodologia são discutidas no capítulo 5, e reforçadas nos exemplos do capítulo 8.

4.6 Considerações sobre a integração de PM-N e AS/PC em R-RIO

A implementação de linguagens com suporte à reflexão computacional normalmente utiliza uma das seguintes alternativas:

- na primeira, existe o controle sobre o ambiente de execução. Esta alternativa é normalmente adotada quando se deseja oferecer reflexão dinâmica. Neste caso, o ambiente deve ser especializado para interceptar eventos, tais como a chamada de métodos, criação de objetos ou leitura e escrita em variáveis. Em seguida, o evento ocorrido precisa ser reificado e encaminhado a um meta-componente. O uso destes mecanismos é programado através de APIs que acessam o ambiente de execução. MetaJava oferece estas possibilidades, por exemplo (seção 3.2.1).
- na segunda, o código-fonte das aplicações é acrescido de anotações específicas, as quais permitem identificar as partes de nível-base e de meta-nível. Baseado nestas anotações, um compilador especial mistura as partes, resultando em um único código. Isto também caracteriza reflexão estática. É o caso de OpenC++, por exemplo.

Como visto na seção 3.2.3.2 - capítulo 3, implementações de AOP, como D e AspectJ, operam sobre código-fonte, tanto para a especificação funcional, como para a descrição de aspectos. O resultado do entrelaçamento da descrições é, também, um código, contendo todos os aspectos misturados. A separação de aspectos se perde neste momento. Conclui-se, assim, que a reflexão estática e AOP são parecidos, ou, de outro modo, podemos dizer que AOP é uma forma de MOP. Em [154], entretanto, é advogado que "aspectos não deveriam morrer". O sentido da afirmação é o de que a granularidade das unidades de código reconfiguráveis durante a execução é diminuída, o que torna difícil o suporte de requisitos de evolução dinâmica para algumas aplicações. Se os aspectos entrelaçados mantivessem seu contorno durante a execução, seria possível pensar em adaptar uma aplicação durante sua execução, sem a necessidade de códigos-fonte ou a geração de novos módulos de execução.

Por outro lado, as vantagens da separação de interesses oferecidas pela reflexão computacional, mesmo não aplicadas para evolução dinâmica, podem ser obtidas em etapas iniciais do ciclo de vida de uma aplicação, e não necessariamente em sua manutenção. Por exemplo, o uso de um nível-base e um meta-nível tem sido considerado também durante a análise de requisitos [155]. Neste caso, a evolução e a reusabilidade das aplicações são de igual modo facilitadas, embora não obrigatoriamente de forma dinâmica.

A utilização de reflexão computacional ou AOP, em sistemas onde sejam

disponíveis apenas componentes já compilados, não é trivial. Não existem mecanismos que definam os pontos de junção onde a reificação e a reflexão devem ocorrer, ou como entrelaçar aspectos em códigos executáveis. No caso da tecnologia de componentes, este problema tem sido contornando com o desenvolvimento destes em duas camadas [27]. Uma camada implementa a funcionalidade prometida do componente. 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 pode ser utilizada durante a execução, para que o componente interaja com o ambiente de suporte, capturando e emitindo eventos, ou para receber pedidos de adaptação de algum sistema de controle da aplicação.

Para o problema apontado no parágrafo anterior, pode haver uma solução em um outro nível, se um ambiente de suporte, que ofereça reflexão estrutural, estiver disponível, e se as unidades executáveis, correspondentes dos componentes, sejam padronizadas em um nível de baixa granularidade. Assim, seria possível inspecionar a estrutura da aplicação e alterar partes de um componente, mesmo sem acesso ao seu código-fonte. Esta solução tem sido empregada em propostas baseadas em Java ([111], avaliada na seção 3.2.2, e [112]). Neste contexto, é possível alterar o comportamento de um objeto utilizando uma combinação de reflexão estrutural, disponível na JVM, e a manipulação de *ByteCodes*. A reflexão estrutural é utilizada para obter as referências aos elementos que se deseja alterar, e a manipulação do *ByteCode*, efetivamente, modifica seu conteúdo. A dificuldade do uso desta técnica está em se manipular diretamente um código binário com segurança.

Nas demais situações, em que o acesso à informação estrutural do ambiente de execução, ou a códigos-fonte, não seja possível, o comportamento de um módulo de determinada aplicação pode ser adaptado inserindo-se um intermediário em sua interação com outros módulos. Esta técnica tem sido utilizada através de *wrappers* [156, 24], filtros de composição (seção 3.2.3.1) e conectores. Cada uma destas soluções utiliza a interceptação das requisições e respostas entre os módulos, e executa algum tratamento sobre as mesmas, afetando o comportamento destes módulos. O mecanismo de *interceptors* de CORBA 2.0 pode ser citado como exemplo. Em tempo de execução, um *interceptor* pode inspecionar e alterar as mensagens trocadas por um objeto em um ORB, modificando o seu comportamento.

Em nossa proposta, a utilização de AS/PC e PM-N com 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 intermedeiam a interação entre módulos e são flexíveis o suficiente para manipular estas interações, em um nível diferente dos componentes envolvidos (um meta-nível). O chaveamento entre nível-base e meta-nível é feito naturalmente, quando as requisições e respostas são interceptas nas portas, durante as interações entre componentes (seção 4.4). Adicionado a isso, a reflexão arquitetural combinada com o serviço de gerência de configuração, permite a substituição de conectores ou a reestruturação da aplicação, sem a necessidade de acesso à implementação dos módulos.

4.7 Exemplo

Como primeiro exemplo de aplicação no *framework* R-RIO, será utilizado o problema dos produtores-consumidores com *buffer* limitado. Na versão inicial, basicamente descrevem os componentes da aplicação e uma configuração simples. Nas versões posteriores, serão "impostos" novos requisitos e, em resposta, a aplicação será reconfigurada para atendê-los. A cada nova versão, apenas as diferenças entre a configuração anterior e a atual serão discutidas. A descrição de cada configuração é feita através da ADL CBabel.

4.7.1 Primeira versão

Nesta primeira versão, um sistema produtor produz iterativamente um item e envia uma requisição para o *buffer*, solicitando-lhe o armazenamento do mesmo. 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 arquitetura desta aplicação é 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.

Inicialmente, define-se os tipos de portas, as classes de módulos e conectores

que serão utilizadas na arquitetura. Estas definições iniciais têm escopo global, ou seja, poderão ser utilizadas por qualquer aplicação. Estas definições poderiam estar armazenadas em um repositório ou biblioteca para uso posterior (veja figura 4.9).

```
port int PutT (String Item);
port String GetT (void);
```

A assinatura dos tipos de portas usados na aplicação é declarada.

```
module  BufferC {
  in port PutT;
  in port GetT;
  map CLASS Java "example.buffer";
}

module  ProducerC {
  out port PutT;
  map  CLASS Java "example.producer";
}

module  ConsumerC {
  out port GetT;
}
```

As classes de módulos são declaradas, indicando-se as portas do módulo e uma referência de como o mesmo é implementado. Por exemplo, os módulos da classe *BufferC* são implementados pela classe *example.buffer.class* programada em Java.

Observe que os modificadores **in** e **out** são utilizados para indicar o sentido assumido pela instância da porta

O próximo passo descreve a arquitetura da aplicação.

```
module  BufferApplClass  {

  module  ConsumerC {
    map  CLASS Java "example.consumer";
  } Cons;

  instantiate  BufferC  as  Buff;
  instantiate  ProducerC  as  Prod;
  instantiate  Cons;

  link  Prod  to  Buff;
  link  Cons  to  Buff;
}
```

Primeiramente uma instância *Cons*, da classe *ConsumerC* é definida, e um mapeamento para uma implementação é indicado.

Em seguida, a estrutura da aplicação é definida, declarando-se as instâncias de módulos e a ligação entre eles.

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

```
module  BufferApplClass  BA;
instantiate  BA;
start BA;
```

Neste ponto, a instância completa da aplicação BA foi criada e sua execução iniciada. O esquema da figura 4.10 apresenta o resultado.

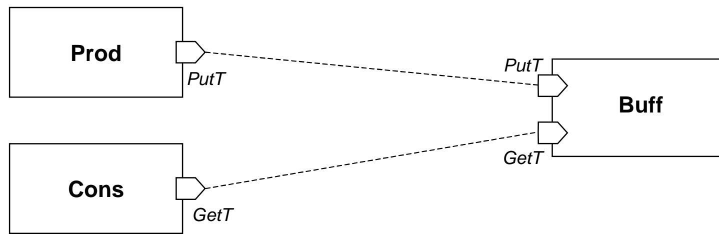


Figura 4.10 - Arquitetura produtor-consumidor-*buffer* (exemplo 1)

Simplificou-se, propositadamente, a descrição da aplicação. Não foram declaradas instâncias de portas nos módulos; apenas indicou-se o tipo que seria utilizado e o seu sentido. Assim, todas as ligações serão verificadas em tempo de compilação e realizadas em tempo de execução, por contexto. Observe-se, ainda, que nada foi descrito a respeito do conector. Neste caso, espera-se que o Gerente de Configuração crie uma instância de um conector *default*, invocação de método (MI), para interligar os módulos, uma vez que todos eles estão no mesmo nó. Pode-se abstrair este conector *default* e imaginar a existência do mesmo, como na figura 4.11.

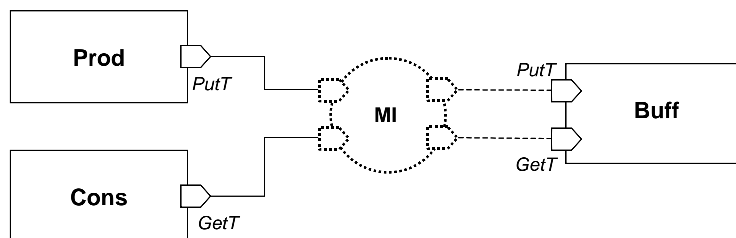


Figura 4.11 - Conector conceitualmente presente

Supondo-se, agora, que seja necessário registrar todas as interações entre o módulo consumidor *Cons* e o módulo *buffer Buff*. Como solução, ele se utiliza de um conector genérico, que, além de intermediar interações, registra as mesmas em um arquivo. Assim sendo, as seguintes alterações são feitas para a nova configuração da aplicação:

```
  ...

  connector (String Herald) {
     map Class Java "connectors.log";
  } Log;
```

Para este caso, declarou-se diretamente uma instância de conector, *Log*, com um mapeamento para a implementação do conector genérico com *log*.

```
  ...

  link Cons to Buff by Log("Prod: ");
}
```

Em seguida os módulos *Cons* e *Buff* são ligados pelo conector *Log*. No momento da criação da instância do conector *Log* foi passado um

parâmetro que, neste caso, é do tipo *string*.

O esquema da figura 4.12 apresenta a nova da arquitetura descrita. O módulo *Prod* continua ligado ao módulo *Buff* por um conector MI.

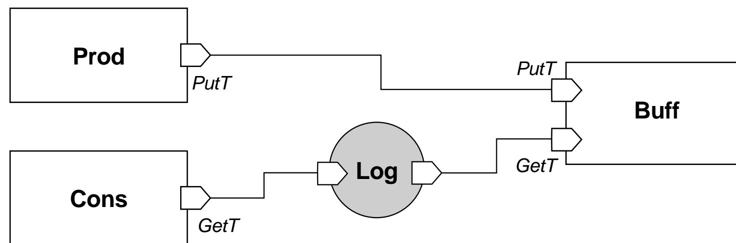


Figura 4.12 - Aplicação produtor-consumidor

Este exemplo inicial permite formar-se uma idéia da flexibilidade de R-RIO. Não foram necessárias grandes modificações na arquitetura da aplicação para agregar novas características. Não houve necessidade de modificação nos módulos da aplicação, pois os aspectos funcionais estão separados dos não-funcionais (no caso, o registro das interações).

4.8 Conclusão

Neste capítulo, os principais elementos do *framework* R-RIO foram apresentados. Dentre estes elementos, o modelo de componentes, o modelo de configuração e o serviço de gerência de configuração formam a base de nossa proposta, para a configuração, execução e manutenção da arquitetura de *software* de aplicações, baseadas em componentes. A arquitetura de *software* de uma aplicação é descrita através de uma ADL, CBabel. A integração das tecnologias de AS/PC e PM-N, utilizada como fundamento em nossa proposta, facilita a obtenção de características importantes para as aplicações: separação de interesses, reuso, evolução dinâmica e abrangência.

Com a finalidade de orientar o projetista no uso do *framework* R-RIO, propusemos um conjunto de recomendações para a configuração de arquiteturas de *software*. Estas recomendações estimulam o projetista a aderir a uma simples disciplina de programação em meta-nível, na qual aspectos funcionais são concentrados em módulos (nível-base) e aspectos não-funcionais são encapsulados em conectores (meta-nível).

No capítulo seguinte discute-se como aspectos não-funcionais são tratados no *framework* R-RIO, e como estes são descritos em CBabel. O uso de contratos é central em nossa proposta, como base para a especificação de aspectos não-funcionais. Também serão abordados, no próximo capítulo, os aspectos dinâmicos no *framework* R-RIO.

Capítulo V

Aspectos Não-Funcionais

5.1 Introdução

Segundo o princípio da *separação de interesses*, discutido no capítulo 2, interesses funcionais e não-funcionais de uma aplicação podem ser concentrados em partes logicamente distintas. É possível ao projetista direcionar seus esforços para cada ponto de interesse, em momentos diferentes, e, posteriormente, integrar os resultados obtidos. Além disso, pontos específicos de interesse, como a comunicação ou a segurança, por exemplo, podem ficar sob responsabilidade de projetistas especializados. Algumas das vantagens da separação de interesses foram demonstradas no capítulo 4.

Em nossa proposta, um ponto de interesse, ou *aspecto*, refere-se a um conjunto de características ou propriedades com alguma afinidade, relacionado com a aplicação. Aspectos associados com a funcionalidade específica da aplicação são denominados aspectos *funcionais*. Aspectos que não fazem parte da funcionalidade específica da aplicação são chamados de *não-funcionais*. Aspectos não-funcionais, relacionados com o ambiente de operação da aplicação, são também chamados de *operacionais*. Uso semelhante do termo *aspecto* é encontrado em [116 e 12], e o termo interesse (*concern*) também é utilizado com sentido semelhante em [31, 89 e 13].

A identificação de aspectos não-funcionais em arquiteturas de *software* ainda é objeto de pesquisas [12, 157, 154, 141]. Interesses distintos de uma aplicação podem ser identificados, com alto nível de abstração, ainda na fase de análise de requisitos [155]. Ao se determinar um aspecto não-funcional deve ser observada coerência, de forma que este possa ser descrito separadamente de outros aspectos, e que as configurações de aspectos diferentes possam ser combinadas, sem efeitos colaterais ou inconsistências [144]. Entretanto, nem sempre uma separação total entre aspectos é possível [122]. A

granularidade das características de um aspecto também pode variar, de acordo com o domínio da aplicação ou mesmo com o estilo de projeto. Como exemplos de aspectos não-funcionais eventualmente requeridos em uma aplicação podemos destacar: concorrência e sincronização, comunicação e qualidade de serviço (QoS), tais como replicação, tolerância a falhas ou restrições temporais.

Neste capítulo, discute-se como os aspectos não-funcionais são tratados em nossa proposta, no nível de arquitetura de *software*. Discute-se, também, como o conceito de contrato é utilizado para a descrição de aspectos não-funcionais. Aspectos dinâmicos em R-RIO são examinados na seqüência. Fechando o capítulo, retoma-se o exemplo dos produtores-consumidores e acrescentam-se requisitos não-funcionais a esta aplicação.

5.2 Encapsulando aspectos não-funcionais em conectores

Ao longo de experiências preliminares com o desenvolvimento de ambientes de configuração [79, 82], foram propostas e implementadas técnicas para a adição e gerenciamento de alguns requisitos não-funcionais, tais como: a comunicação, seleção de protocolos e tolerância a falhas. No contexto destes trabalhos, a idéia de um elemento exclusivo para interligar e intermediar a interação entre módulos não existia. O suporte para a configuração de aspectos era, então, provido por um sistema de suporte especializado. Um módulo funcional, por sua vez, deveria indicar explicitamente o uso deste suporte para a interação com outros módulos.

Em nossa proposta, aspectos não-funcionais são tratados nos conectores. Em princípio, a inclusão de aspectos não-funcionais em um conector é transparente, sob o ponto de vista dos módulos que ele interliga. Os conectores reúnem características que os tornam especialmente adequados para tratar aspectos não-funcionais [15, 158]. Entre estas características, já relacionadas no texto, destacam-se:

- o conhecimento das interfaces providas e requeridas pelos 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.

Vale lembrar que o mapeamento de um conector para uma implementação pode resultar em um objeto ou conjunto de objetos. Estes objetos teriam o código necessário ao atendimento dos aspectos não-funcionais. Entretanto, para encapsular um aspecto não-funcional em um conector, pode ser necessário combinar, em sua implementação, rotinas para reservas de recursos, execução de procedimentos específicos, uso de mecanismos disponíveis no sistema de suporte nativo e, também, a criação de objetos.

Exemplificando a configuração de aspectos, a partir de um conector, considere-se uma aplicação composta por dois módulos clientes utilizando um conector para interagir com um módulo servidor (figura 5.1). Na versão da figura 5.1(a) determinou-se que cada módulo será instanciado em um nó distinto. Sendo assim, durante a configuração, seleciona-se um conector, de um conjunto disponível, que contemple os aspectos de comunicação necessários.

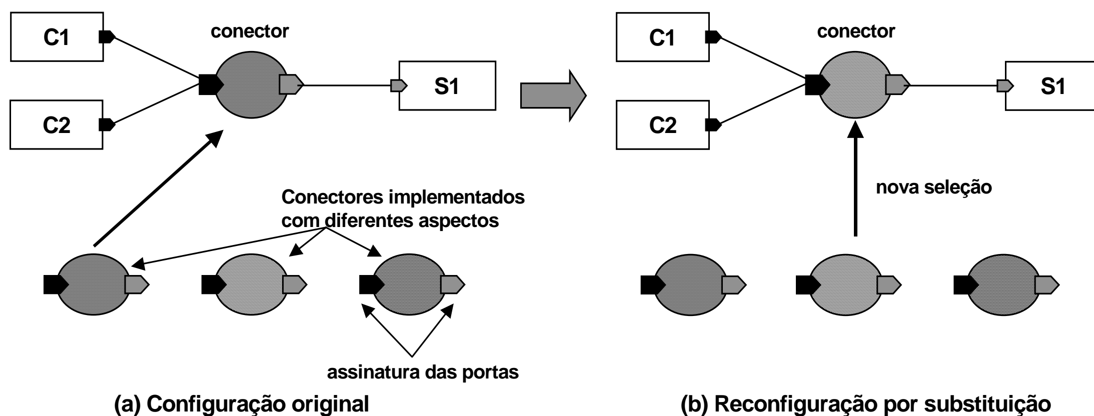


Figura 5.1 - Configuração de uma aplicação simples

Suponhamos que por uma razão operacional, após um período de funcionamento, esta aplicação necessite que as mensagens que transitam 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 5.1(b)).

5.3 Aspectos não-funcionais em R-RIO

Durante nossas pesquisas, identificamos alguns aspectos não-funcionais

recorrentes para vários domínios de aplicação: interação, distribuição, coordenação e QoS. Este conjunto de aspectos pode ser adequadamente encapsulado em conectores. Desta forma, optamos por tratar primariamente os aspectos identificados, os quais respectivamente, permitem:

- **Interação -** a configuração de características das interações entre os módulos;
- **Distribuição -** a localização de um módulo em nó distribuído, independentemente de sua implementação (aqui se incluem os mecanismos para manutenção das referências dos módulos distribuídos e a comunicação entre estes módulos);
- **Coordenação -** o controle da concorrência e da sincronização no encaminhamento das requisições para os módulos;
- **QoS -** o controle da qualidade de serviço de uma aplicação. Em R-RIO, consideram-se QoS as características ou propriedades ligadas à operação das arquiteturas de *software*, tais como tolerância a falhas, restrições temporais, ou parâmetros dos protocolos de comunicação. Aspectos de QoS são discutidos mais amplamente no capítulo 6.

Observa-se que a programação dos aspectos não-funcionais acima mencionados pode ser, em grande parte, ortogonal à programação dos módulos. Isto facilita a descrição destes aspectos e o encapsulamento dos mesmos em conectores. Entretanto, nem sempre é possível separar completamente os diferentes aspectos em determinadas aplicações, ou ainda, aspectos não-funcionais específicos podem ser requeridos. Nestes casos, o projetista não está proibido de fazer uso de outros aspectos não-funcionais, além dos identificados anteriormente, em sua aplicação. Também não é vedada a agregação de aspectos funcionais e não-funcionais no mesmo componente, embora isso deva ser evitado.

5.3.1 Aspectos de Interação

Uma das características importantes em nossa proposta é a separação entre: (i) os módulos funcionais da arquitetura, (ii) a topologia de interligação que estes irão formar e (iii) como estes irão interagir. Ou seja, além de se definirem os módulos que irão interagir, é possível especificar como se dará esta interação através de conectores.

Em R-RIO, os aspectos de interação permitem descrever:

- os grupos de módulos que vão interagir de alguma forma;
- qual o estilo de interação entre portas de saída e entrada, síncrono, assíncrono ou difusão;
- como o conector deve tratar o encaminhamento das interações. Isto pode ser um simples repasse de dados entre portas compatíveis do conector, ou uma forma mais sofisticada de interação (veja a seção A.4.1 - apêndice A e o exemplo da seção 8.4).

5.3.2 Aspectos de Distribuição

O modelo de componentes de R-RIO permite que os módulos de uma aplicação sejam instanciados em qualquer nó de um sistema distribuído. Na implementação de um módulo, não é necessária nenhuma referência quanto à localização do mesmo, nem qualquer outro suporte para a interação de módulos distribuídos. Mais especificamente, a localização de uma instância de um módulo é transparente em relação à sua funcionalidade. Problemas de localização são resolvidos pelo suporte à configuração.

A interação entre módulos distribuídos é realizada através de um conector que ofereça alguma forma de encaminhar as requisições e respostas por uma rede. Este conector pode ser composto por partes distribuídas que cooperam através de algum mecanismo de comunicação. Por exemplo, um conector que encapsule o mecanismo de RMI ou que utilize comunicação por *sockets* pode ser utilizado para intermediar a interação de módulos distribuídos.

A configuração das aplicações com módulos distribuídos pode ser facilitada com o uso de uma biblioteca de conectores, implementados com as combinações de estilos de comunicação, mecanismos de suporte e protocolos mais comumente usados. Assim, ao se descrever a arquitetura da aplicação em CBabel, pode-se selecionar um conjunto de classes de conectores mais adequado para a mesma. Caso contrário, os conectores requeridos deverão ser implementados ou obtidos por composição.

A distribuição escolhida para os módulos da aplicação pode ter implicações no seu desempenho. Em CBabel, a instanciação de um módulo pode vir acompanhada de uma referência associada à sua localização (ver seção A.4, apêndice A). Por outro lado, o suporte à configuração também pode otimizar, no meta-nível, a escolha da localização das instâncias de módulos, segundo uma política definida pelo projetista. Uma

metodologia, tal qual a apresentada em [159], para planejar a distribuição de objetos, poderia ser utilizada. Nesta distribuição, podem ser considerados fatores como o tráfego gerado pela interação dos módulos, a composição destes módulos, ou a carga dos nós.

O suporte específico para implementar uma política de escolha de localização de módulos não é relevante para o modelo de componentes de R-RIO. Este suporte poderá ser provido por um sistema de nomes ou diretório. Por exemplo, os serviços de identificação, localização e corretagem, disponíveis em ORBs, ou os serviços de mapeamento existentes nos mecanismos de RPC e RMI, podem ser utilizados pelo suporte à configuração.

5.3.3 Aspectos de Coordenação

Durante a concepção de uma aplicação, podem surgir questões sobre a concorrência, exclusão mútua e sincronização. O programador da aplicação, muitas vezes, precisa embutir no código comandos ou primitivas para tratar estes aspectos. Para acessar recursos compartilhados em exclusão mútua, por exemplo, podem ser utilizados mecanismos como *locks* ou semáforos [160].

Existem algumas opções que podem ser adotadas para incorporar-se requisitos de coordenação nas aplicações. Certos sistemas oferecem mecanismos para a programação de concorrência e sincronização pelo uso de funções primitivas disponíveis através de bibliotecas. Sistemas operacionais, tais como o UNIX, oferecem a abstração de processos como unidade de concorrência e, normalmente, incorporam bibliotecas para a comunicação entre processos (*Inter-Process Communication* - IPC) [140], que permitem a programação de mecanismos como semáforos, mensagens, *pipes*, etc. Existem também bibliotecas, como a *pthreads* [49], que oferecem acesso à programação de mecanismos como *threads*, monitores, *mutexes*, etc. Estas bibliotecas podem ser integradas a programas escritos em linguagens de programação como o C e C++ e, muitas vezes, têm o suporte do próprio sistema operacional [161].

Algumas linguagens orientadas a objeto incorporam mecanismos para a programação de concorrência e sincronização, já inseridos na sintaxe da própria linguagem (Java, por exemplo). Várias destas linguagens foram concebidas como extensões das versões seqüenciais das linguagens originais [162, 163, 164, 165]. Existem também algumas variações nesta abordagem, como [166] que propõe o uso de

anotações, com a função de comentários ao programa principal, para expressar concorrência e sincronização.

O trecho de código da listagem 5.1 apresenta uma implementação possível para o método ***put*** de uma classe *BoundedBuffer* em Java, que segue a linha do exemplo usado no capítulo 3. Este método deve 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 aos aspectos de concorrência e sincronização (com exceção da linha 11, que efetivamente executa o método), e para isso são utilizadas várias primitivas relacionadas com monitores e semáforos, segundo a semântica adotada em Java (linhas 4, 5, 7, 8, 14, 18 e 20). O código resultante é confuso e obscurece as partes que realmente dizem respeito ao problema original. Este é mais um exemplo de problema que decorre do entrelaçamento de código mencionado no início do capítulo 2.

```
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 5.1 - implementação da classe BoundedBuffer

Em experiências anteriores com ambientes baseados em configuração [82], os aspectos de concorrência entre os métodos de um módulo eram determinados pela forma de implementação adotada, que impunha uma política compulsória. Por exemplo, toda mensagem, ao chegar a um módulo, era enfileirada e consumida 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 encaixa naturalmente nesse modelo de concorrência.

A idéia de se separarem os aspectos de coordenação e a implementação da computação principal da aplicação aparece em algumas propostas mais recentes. AOP, avaliada no capítulo 3, sugere uma linguagem especial de coordenação. Filtros de Composição, também avaliada no capítulo 3, propõe um mecanismo que estende o modelo tradicional de objetos, tratando os aspectos de coordenação em filtros, separadamente dos aspectos básicos da aplicação. Trabalhos anteriores a estes, como [167], 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 [29], um outro exemplo, divide 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.

5.3.3.1 Modelo de Coordenação em R-RIO

Em nossa proposta, os aspectos de coordenação de uma aplicação são tratados no nível da arquitetura de *software*. Estes aspectos são especificados em CBabel, em alto nível de abstração, sem a necessidade de prevê-los na concepção ou implementação dos módulos. A rigor, instâncias diferentes de uma mesma classe de módulo podem estar submetidas a esquemas de coordenação distintos.

O modelo para o tratamento do aspecto de coordenação, em R-RIO, prevê três situações: (i) coordenação entre módulos de uma aplicação, (ii) concorrência entre métodos de um mesmo módulo e (iii) sincronização entre métodos de um mesmo módulo.

5.3.3.2 Coordenação entre módulos

No modelo de componentes de R-RIO, cada módulo pode ser considerado uma unidade autônoma. Os módulos de uma aplicação são executados potencialmente de forma concorrente. Este modelo é adotado em propostas semelhantes, como em [30 e 12], em que cada objeto desempenha um papel autônomo na aplicação. A modelagem de uma aplicação com módulos concorrentes captura naturalmente as situações de

concomitância encontradas no mundo real. Em um sistema bancário, por exemplo, temos vários clientes, caixas e contas correntes, que são elementos autônomos, com vida própria. Uma aplicação modelada a partir deste esquema, segundo nosso modelo, pode ser concebida mapeando-se cada cliente, caixa e conta corrente em uma instância de módulo.

O tipo 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 um ambiente Java for utilizado, um módulo será mapeado em uma classe, com seu respectivo *ByteCode* [168], e 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 [161]. Em cada um destes ambientes, a concorrência entre módulos poderá ser diferente.

O escalonamento, por sua vez, é também associado ao ambiente de suporte utilizado. Em alguns ambientes, o escalonamento pode ter seu funcionamento configurado rigidamente, sem possibilidade de adaptação. Em outros, é possível dispor-se de uma interface através da qual alguns ajustes sejam admitidos [61]. Por exemplo, o padrão POSIX para *threads*, utilizado em alguns sistemas operacionais, possui uma API para configurar políticas e parâmetros de escalonamento para um grupo de *threads*. Em outro nível, algumas implementações de CORBA oferecem uma API para configuração do escalonamento na execução de objetos [169].

Em R-RIO, as políticas de escalonamento para os módulos de uma aplicação também poderiam ser configuradas como parte do aspecto de coordenação. Para descrever-se uma política de escalonamento, anotações adequadas seriam introduzidas na sintaxe do aspecto de coordenação. Um conector, implementado para suportar a configuração de políticas de escalonamento, faria o acesso aos mecanismos do sistema de suporte para efetivamente adaptar o escalonamento, de acordo com a descrição feita no nível da arquitetura. Uma outra possibilidade seria considerar o escalonamento como um requisito de QoS. Neste caso, a solução, descrita no capítulo 6, poderia ser utilizada.

A sincronização entre módulos é caracterizada pelo estilo de interação entre os mesmos. Logo, a sincronização entre métodos de módulos distintos que irão interagir, é definida pelo tipo das portas de entrada e saída que estão ligadas, que seleciona o estilo

de interação a ser provido pelo conector. Se o estilo síncrono de interação for configurado, a execução do método que oferece um serviço é bloqueada até que uma requisição tenha chegado. Por sua vez, o método que iniciou a requisição é bloqueado até que uma resposta seja retornada. Por outro lado, se o estilo assíncrono for selecionado, a execução do método que iniciou a requisição continua sem bloqueio.

5.3.3.3 Concorrência entre métodos de um módulo

Um outro ponto do aspecto de coordenação é a concorrência entre os métodos de um mesmo módulo. Em determinados casos, parece natural que os métodos de um módulo sejam executados concorrentemente, em outros, talvez se torne necessária a imposição de um esquema de exclusão mútua ou atomicidade na execução destes métodos. Uma forma clássica de sincronização intra-módulo é a implementação do módulo como um monitor [170]. 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 está ocioso;
- a execução de um método invocado anteriormente foi completada;
- um método anterior foi bloqueado para esperar a ocorrência de uma condição ou evento.

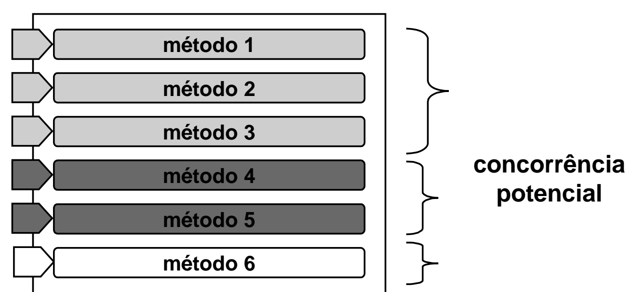


Figura 5.2 - Módulo com métodos sincronizados

Em nosso modelo, adotamos uma solução similar à apresentada em [12], a qual permite indicar subconjuntos de métodos de um módulo, que devem executar em exclusão mútua, permitindo configurar o nível de concorrência intra-módulo. A unidade de concorrência utilizada é o método. Desta maneira, em um módulo com seis métodos,

podemos configurar três métodos que devem executar em exclusão mútua, outros dois métodos de forma semelhante. O método restante não será marcado. Assim, até três métodos, sendo um da cada grupo, podem ser executados concomitantemente. Esta especificação de configuração pode ser visualizada na figura 5.2.

Chamadas concorrentes a um mesmo método admitem duas formas de tratamento: ou são serializadas em uma fila para tratamento seqüencial pelo método, ou encarnações diferentes do método são criadas para atender cada chamada, concomitantemente. É possível configurar uma destas alternativas de comportamento para cada método.

Chamadas concorrentes a um mesmo método podem ser úteis na construção de servidores concorrentes, em que um mesmo tipo de processamento deve ser realizado, sobre recursos diferentes, ou que não precisam ser acessados sob exclusão mútua. Os métodos que precisam fazer uso de recursos compartilhados devem estar no mesmo 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 mútua com os métodos de atualização, mas poderiam executar concorrentemente entre si.

5.3.3.4 Sincronização entre métodos de um 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 [171]. Um guarda pode controlar a admissão de requisições a uma porta de entrada de um módulo, retardando o tratamento destas requisições, até que a expressão lógica associada a este guarda seja satisfeita. Podemos ilustrar o funcionamento de um guarda comparando-o a uma chave que só pode assumir os estados aberto ou fechado. O estado do guarda resulta da avaliação de uma expressão que envolva variáveis que representem uma dada condição relacionada ao estado atual do módulo, e resulte em "verdadeiro", abrindo o guarda, ou "falso", fechando o guarda.

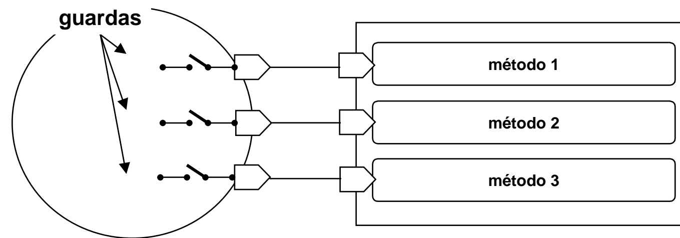
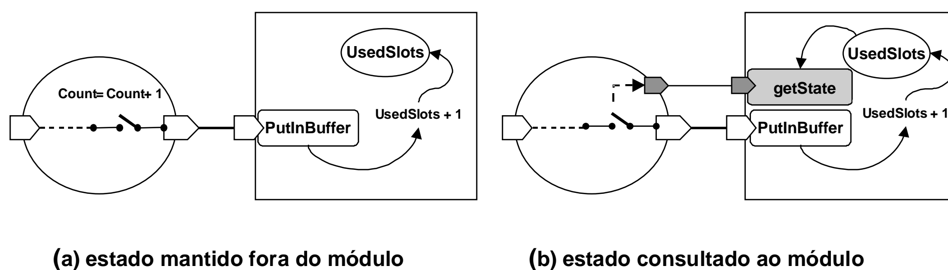


Figura 5.3 - Módulo com guardas para seus métodos

Guardas são encapsulados nos conectores, mais especificamente nas portas de saída do conector, que estarão ligadas diretamente às portas de entrada do módulo, que dão acesso aos métodos sob guarda (figura 5.3). Desta forma, retira-se dos módulos a responsabilidade por aspectos de sincronização. Esta opção está em conformidade com o modelo formal para R-RIO apresentado no apêndice B. Segundo este modelo, o "posicionamento" do guarda, antes ou depois das portas do conector/módulo, não representa, em princípio, diferença na execução da aplicação. Entretanto, sob o ponto de vista de operação, e nos casos em que a arquitetura requer a composição de aspectos de coordenação com outros aspectos não-funcionais, os guardas devem atuar próximos do módulo coordenado.

Observa-se, entretanto, que a avaliação da expressão da condição de um guarda pode depender dos módulos. Em algumas situações é necessário o conhecimento do valor de certas variáveis internas de um módulo para se obter o seu estado. Em outras situações é possível manter-se este estado externamente ao módulo guardado, mesmo que à 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. Isto substitui a inspeção de uma variável do módulo com a mesma finalidade.



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

Figura 5.4 - Consulta ao estado de um módulo

Em R-RIO, um conector tem autonomia para manter informações de estado (figura 5.4(a)) ou pode solicitar ao módulo interligado o valor de algumas variáveis (figura 5.4(b)). Estas informações podem ser utilizadas no momento de avaliar o estado do guarda. A consulta à variáveis de estado de um módulo será feita através de um método de acesso, fornecido pelo programador. Este método, que deve estar presente no nível da implementação, é descrito implicitamente na arquitetura ao se declarar variáveis que podem ser consultadas através do mesmo. Deste modo, declarando-se uma variável *UsedSlots* na descrição de uma classe de módulo, admite-se a existência de uma porta de entrada *getState()* que retorna seu valor (e de outras variáveis também declaradas), e um método associado a esta porta (veja seção A.4.3, apêndice A). Solução similar também é utilizada na OMG-IDL, através da declaração de atributos (ex.: *readonly attribute <tipo>*) das interfaces, e em outros *middlewares*, tais como [123].

Em [172] uma estratégia semelhante para programação de sincronização é documentada como um *design pattern* chamado *Synchronizer*. Neste *pattern*, são destacadas as vantagens em se separarem as políticas de sincronização dos objetos a serem coordenados, reforçando nossa tese. Sugere-se a interceptação do fluxo que chega a um objeto, antes de entregá-lo ao mesmo, desviando-o para um objeto "sincronizador", que implementa a política de sincronização desejada. Isto deve ser feito através de um mecanismo de herança (figura 5.5).

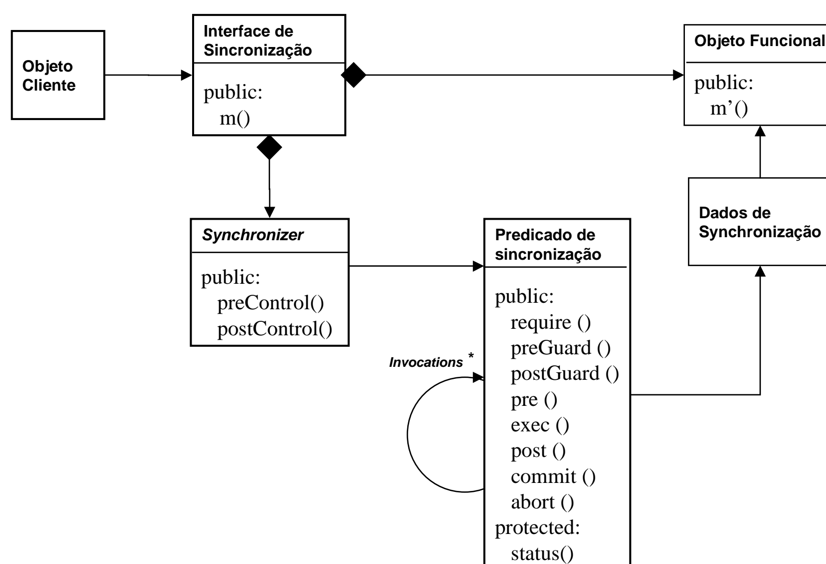


Figura 5.5 - Diagrama de classes para o design pattern *Synchronizer*

No *pattern* citado não se prevê, entretanto, uma forma para se descrever a política de sincronização. Além disso, não existe um suporte, como o conector em R-RIO, com o fim de encapsular o código de coordenação. É necessária a codificação dos relacionamentos entre os objetos sincronizadores e o objeto funcional, através de herança e invocações explícitas de método. Também não se sugere como fazer a interceptação das invocações, sem programá-la explicitamente. Como alternativa, são documentadas e sugeridas algumas políticas de sincronização (leitores-escritores, pessimista e otimista, e produtor-consumidor), utilizando-se um esquema de guardas (através dos métodos do objeto *Synchronization Predicate* - figura 5.5) similar a R-RIO.

5.3.3.5 Granularidade de sincronização

Em nossa proposta, consideramos o método como a unidade de concorrência e sincronização com menor granularidade. Por exemplo, no nível da arquitetura, não é possível definir-se uma região crítica específica, dentro de um método, que deva ser sincronizada ou executada em exclusão mútua. Para tornar viável tal procedimento, outros pontos de interação, além das portas, precisariam ser definidos, de forma que mecanismos de sincronização, como semáforos, por exemplo, pudessem ser *inseridos* corretamente.

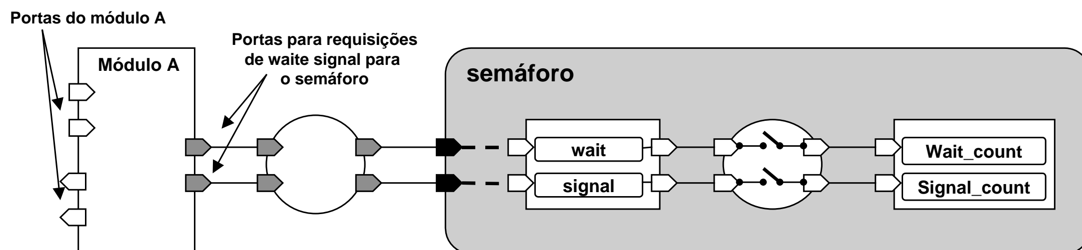


Figura 5.6 - Aplicação com módulo-semáforo

Se uma granularidade menor de concorrência for estritamente necessária, 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 [160]. O módulo-semáforo poderia residir em um nó diferente dos outros módulos da aplicação. Adicionalmente, o conector poderia ser configurado com tolerância a falhas. Por outro lado, haveria a necessidade de se definir, no módulo, portas de saída adicionais para fazer requisições aos métodos *wait* e *signal* no "módulo-semáforo" (figura 5.6). Parece-

nos que o uso deste tipo de mecanismo obrigaria o projetista a abrir mão do alto nível de abstração, além do que não traria benefícios significativos para a configuração das aplicações.

Uma solução com granularidade semelhante foi adotada, por exemplo, na versão 0.2 de AspectJ [117]. Na linguagem AspectJ, podem-se introduzir primitivas de sincronização em pontos definidos de uma classe em Java, como antes ou depois da execução de um método, incluindo o método construtor e destrutor, ou exceções. Em uma linguagem reflexiva, com um MOP que permita inspecionar eventos como a leitura e escrita em variáveis, qual sucede em 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. Tal esquema não nos parece vantajoso durante o processo de concepção dos aspectos de coordenação da aplicação, com a abstração que estamos propondo, pois o projetista teria de preocupar-se com detalhes muito distantes do nível da arquitetura.

Partindo de nosso modelo, o programador de um módulo não fica impedido de utilizar diretamente quaisquer mecanismos de sincronização disponíveis na linguagem de implementação, sem passar pelo nível de configuração, mas deverá estar ciente de que estes não serão *enxergados* neste nível. Todavia, observando-se linguagens como ADA e Java, e propostas similares a R-RIO [79, 13 e 12], acreditamos que o esquema proposto para o tratamento do aspecto de coordenação, no nível da arquitetura de *software*, deverá atender a uma determinada classe de aplicações, com a vantagem da separação de interesses e facilidade de estruturação.

5.4 Especificando aspectos não-funcionais através de contratos

Empregamos, nesta proposta, o conceito de contratos para especificar os aspectos não-funcionais, discutidos nas seções anteriores, no nível da arquitetura de *software*.

O conceito de contratos, proposto inicialmente para a tecnologia de objetos [173, 174, 175], incentiva o programador de um objeto a indicar o comportamento

externo esperado e as restrições associadas a este objeto, quando um de seus métodos é invocado. Os usuários, por sua vez, expressam sua disposição em usar o objeto, com o comportamento indicado e suas restrições. Desta forma, objeto-cliente e objeto-servidor estabelecem um contrato. De um modo geral, o uso de contratos permite que o projetista consiga utilizar objetos de forma mais adequada para a aplicação, sem com isso quebrar o encapsulamento dos mesmos.

O uso de contratos também pode ser aplicado em outros níveis de *software*, além de objetos. Sistemas baseados em componentes geralmente admitem a especificação de restrições e limites para a interação de componentes e invocação de métodos. Por exemplo, em EJB - *Enterprise JavaBeans*, as classes de componentes apresentam um padrão para consulta e utilização de suas interfaces. Em outro contexto, OCL - *Object Constraint Language* [136] permite a declaração de restrições para objetos descritos através de diagramas UML (veja discussão na seção 3.5, capítulo 3), e KDL - *Kind Description Language* [176] propõe a declaração de contratos, através de reflexão, para aplicações baseadas em componentes.

No nível da arquitetura de *software*, um módulo é conhecido pela assinatura de sua interface. Uma descrição textual complementa as informações sobre o mesmo com alguns aspectos de comportamento. O projetista geralmente conta apenas com estas informações para a utilização deste módulo. O modelo citado é usualmente chamado de caixa-preta [7], porque as partes funcionais internas dos módulos estão escondidas, expondo-se apenas a parte superficial, de forma suficiente à sua utilização. Propõe-se que, com o uso da abstração de contrato, aspectos não-funcionais possam ser descritos no nível da arquitetura, facilitando a reutilização de módulos em aplicações com requisitos não-funcionais específicos.

Em R-RIO, contratos de aspectos não-funcionais são tratados como visões distintas da aplicação (figura 5.7) [177, 175, 178]. Cada aspecto não-funcional discutido na seção 5.3 pode ser especificado por um contrato específico.

Um contrato usualmente compreende: definição, aderência, imposição e gerência (incluindo o monitoramento, o tratamento de exceções e uma possível rescisão) [174, 173, 175]. Em R-RIO, a gerência de um contrato é suportada por diversos mecanismos, pelos seguintes elementos:

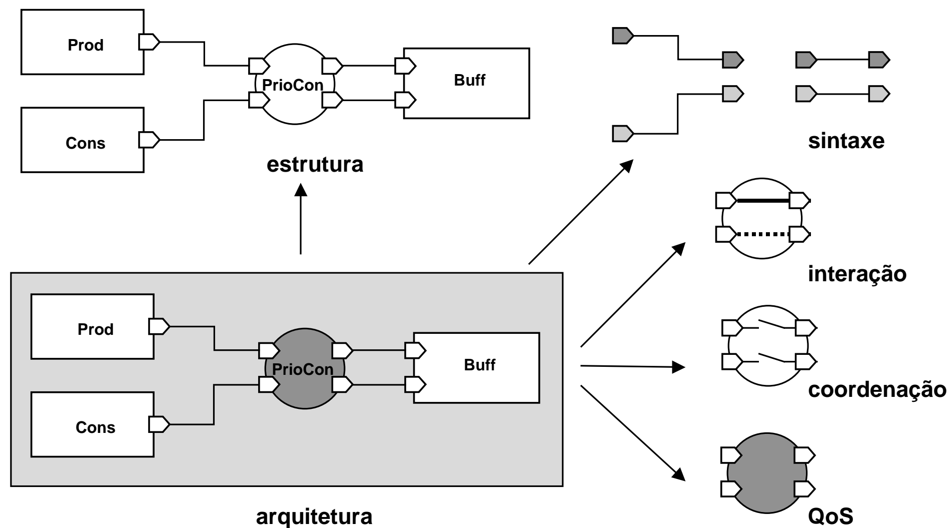


Figura 5.7 - Diferentes visões da arquitetura de uma aplicação

- CBabel. Um contrato é definido no nível da arquitetura através de CBabel. A descrição de contratos para cada aspecto não-funcional possui uma sintaxe específica. Os contratos de interação, distribuição e coordenação são inteiramente descritos em CBabel. A sintaxe destes contratos pode ser consultada na seção A.4, apêndice A. Para a descrição de contratos de QoS, optou-se por uma sintaxe que permite descrever, de uma forma aberta, as propriedades sendo consideradas. Assim, contratos de QoS não são restritos a características fixas. Em geral, indica-se, inicialmente, o conjunto de propriedades que um módulo servidor pode oferecer a outros módulos, e o conjunto de propriedades que um módulo cliente deseja requisitar de um módulo servidor. Em seguida, define-se, no conector que os interliga, os termos do contrato. O capítulo 6 apresenta mais detalhes sobre contratos de QoS.

  É possível associar-se CBabel a ferramentas de verificação, que assegurem a aderência a um contrato por parte de todos os módulos envolvidos. Neste nível, a aderência será verificada estaticamente, a partir da definição de um contrato. A descrição de um contrato, por sua vez, pode resultar em geração de código. Pode haver, também, o envio de comandos de configuração para o Gerente de Configuração inicializar ou reservar recursos relacionados com os contratos.

- Conectores. Um contrato definido em um conector, no nível da arquitetura, pode ser imposto, na fase de execução da aplicação, por uma implementação que contenha a infra-estrutura necessária para tornar o contrato operacional. De acordo com o que já

foi discutido, a implementação de um conector pode utilizar mecanismos ou serviços do ambiente de execução nativo para viabilizar um contrato. Outra possibilidade, é a de combinar conectores mais elementares, para que estes, configurados adequadamente, formem um conector composto que ofereça a infra-estrutura necessária à imposição de um contrato. Esta habilidade pode ser também explorada para adaptar um conector composto, reconfigurando-se seus componentes, com o objetivo de manter um contrato.

- Suporte à configuração. O suporte à configuração oferece a infra-estrutura necessária para implantar um contrato. Este suporte é oferecido através das APIs de configuração e reflexão arquitetural (seções 4.3 e 4.4 - capítulo 4). Para gerenciar um contrato, módulos especiais podem ser providos como parte da aplicação. Tais módulos podem ser programados para conhecer a natureza de um dado contrato, acessando informações de meta-nível da arquitetura, e interagir com o suporte à configuração, quando necessário. Desta forma, estes módulos tornam-se responsáveis por: (i) monitorar e decidir quando um contrato deve ser renegociado ou terminado; (ii) compor dinamicamente um conector adequado que possa fazer valer o contrato e (iii) efetivamente promover a reconfiguração de um conector, se a configuração original não mais puder fazer valer o contrato. O uso deste esquema para gerenciar um contrato é mais amplamente discutido no capítulo 6.

5.5 Aspectos dinâmicos em R-RIO

A arquitetura de uma aplicação pode ser modificada durante a operação, para atender a mudanças planejadas ou *ad hoc*. A combinação de AS/PC e PM-N, em R-RIO, permite que a separação de interesses seja mantida em casos de reconfiguração. A API de reflexão arquitetural oferece acesso às informações da arquitetura a ser manipulada, e a API de configuração oferece os meios para que as mudanças se efetuem. Por exemplo, uma seqüência de comandos para instanciação, bloqueio, terminação e ligação, se executada em ordem adequada, resulta na substituição de um módulo ou conector de uma aplicação. Além de efetivamente mudar a arquitetura da aplicação, as alterações promovidas por uma reconfiguração são refletidas na representação interna da arquitetura em execução, no meta-nível.

5.5.1 Reconfigurações planejadas

Reconfigurações planejadas são previstas em tempo de configuração. A linguagem de descrição de arquiteturas de R-RIO pode suportar a programação de reconfigurações, através de construções sintáticas básicas para esta finalidade.

Alguns tipos de reconfigurações planejadas poderiam ser previstas nos contratos de CBabel. Por exemplo, através de uma cláusula especial, poder-se-iam indicar ações a serem tomadas, se um contrato não estivesse mais sendo honrado. O início da execução destas ações se faria externamente quando isso fosse solicitado ao Gerente de Configuração. Adicionalmente, módulos especializados poderiam ser adicionados ao suporte à configuração, para garantir a consistência da arquitetura após a reconfiguração.

Outras propostas estendem as ADLs, provendo-as com a capacidade de expressar mecanismos de monitoramento e políticas de gerenciamento de configuração, separadamente, definidas por construções especiais [90, 74, 179, 180]. Embora tal capacidade possa ser incluída em CBabel, optou-se por conservá-la concisa [181], evitando-se adicionar muitas regras de disparo de reconfigurações baseadas em eventos e estado. R-RIO suporta uma versão pragmática dos conceitos descritos em [180], que adere à idéia de se prover uma ADL compacta, na qual se podem somar detalhes adicionais, quando necessário. A tomada de decisão sobre as reconfigurações ficam concentradas em módulos especiais, que impõem ao Gerente de Configuração a adaptação da aplicação. No capítulo 6, isto é exemplificado no decorrer da discussão sobre reconfiguração planejada para contratos de QoS.

5.5.2 Reconfigurações não-planejadas (*ad hoc*)

R-RIO também oferece suporte para reconfigurações não planejadas (*ad hoc*), que podem ser executadas através da API de configuração. Um módulo especial, provido na própria aplicação, ou externo a ela, pode iniciar uma reconfiguração, reagindo a eventos não previstos. Este tipo de reconfiguração não é descrito na arquitetura da aplicação, e deve ser codificado diretamente neste módulo especial. É possível, entretanto, a análise e a validação das mudanças pretendidas em uma reconfiguração, utilizando-se a reflexão arquitetural, antes de se proceder efetivamente às mudanças.

Outros esquemas de programação adaptativa, tais como o utilizado no sistema Darts [99], oferecem um conjunto limitado de políticas de adaptação. Além disso, a implementação do mecanismo de chaveamento destas políticas geralmente está embutida no ambiente de suporte. O programador fica restringido a adaptar a configuração do *software* em execução, através de diretivas chamadas a partir dos módulos da própria aplicação, durante a operação da mesma. O esquema de R-RIO oferece flexibilidade e não proíbe o uso do suporte à configuração dinâmica, oferecida por sistemas de suporte, como Darts.

5.5.3 Consistência de estados

Em geral, reconfigurações devem considerar de que modo o estado interno dos componentes e o estado das interações em curso podem ser afetados. Reconfigurações *ad hoc* devem ser realizadas apenas em situações tais que não tragam inconsistências para a aplicação, ou quando são inevitáveis, mesmo que causando inconsistência. Isto demanda um suporte que mantenha atualizado e disponível o estado dos componentes instanciados e das interações entre componentes que ainda estão sendo realizadas. As técnicas descritas em [14, 182, 74, 183] podem ser utilizadas para garantir a consistência das aplicações durante atividades de reconfiguração. Com estas técnicas, leva-se a aplicação a um estado *quiescente*, em que as mudanças serão feitas sem problemas. Opcionalmente, o programador pode deixar *preparado* um esquema para facilitar a inspeção e alteração do estado interno do componente, através de uma porta especial de manutenção. Tal esquema pode ser padronizado para todos componentes de uma aplicação. Como visto anteriormente, em CBabel, a inspeção do estado de um componente pode ser prevista, declarando-se as variáveis a serem inspecionadas. Entretanto, um serviço para a alteração do estado de um componente deve ser declarado através de portas específicas para esta finalidade.

5.6 Exemplo

Contando, agora, com a possibilidade de especificar aspectos de interação, distribuição e coordenação em R-RIO, a aplicação produtor-consumidor é retomada, introduzindo-se requisitos não-funcionais.

5.6.1 Tratando a concorrência com múltiplos produtores e consumidores

Na primeira versão da aplicação, seção 4.7, capítulo 4, não foram especificados aspectos não-funcionais. Observou-se apenas que, por definição do modelo de componentes, todos os módulos da aplicação são executados concorrentemente, o mesmo ocorrendo com os métodos de cada módulo. Também pôde ser observado que a sincronização entre os módulos se dá pela natureza (síncrona, no exemplo) das interações entre eles.

Uma vez que é impossível prever a ordem de execução dos módulos e métodos da aplicação, é provável que aconteçam inconsistências, tais como dois produtores escrevendo na mesma área do *buffer*, ou um produtor e um consumidor tentando acessá-la.

A esta nova versão da aplicação acrescentam-se, então, os seguintes requisitos:

- A aplicação deve permitir o acesso concorrente ao *buffer* por vários produtores e vários consumidores.
- O acesso ao *buffer* deve ser feito em exclusão mútua: somente um produtor ou um consumidor pode fazê-lo. Este acesso deve ser iniciado e terminado sem a interferência de outros acessos (ou seja, atomicamente).

Certamente, a classe de módulo *BufferC*, descrita no exemplo do capítulo 4, poderia ter sido programada para satisfazer a tais requisitos, mas, para tanto, caberia ao programador antecipar estas necessidades, já que nada havia sido especificado. Por outro ângulo, vamos supor que uma classe de módulo *BufferClass,* genérica, sem qualquer outro aspecto programado, além de sua funcionalidade básica, esteja disponível. Em R-RIO, esta classe de módulo pode ser utilizada, sem a necessidade de alterações para contemplar a exclusão mútua. Além disso, separando-se este aspecto da funcionalidade básica, é possível reconfigurar a aplicação com outro esquema de concorrência, sem alterar-se o módulo *buffer*.

Os requisitos de concorrência e exclusão mútua são contemplados no exemplo, descrevendo-se um conector com um contrato de coordenação. Neste contrato, especifica-se a exclusão mútua, marcando-se as portas de saída do conector como *exclusive.* Observa-se que este conector deve interligar todos os produtores e consumidores ao módulo *Buff*, a fim de que a exclusão mútua seja garantida.

O requisito para a inclusão de vários produtores e vários consumidores concorrentes é facilmente configurado na aplicação, bastando-se criar 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 5.8 esquematiza o resultado.

```
module ConcurBufferApplClass  {

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

No conector é inserido um contrato de exclusão mútua, identificado através do atributo **exclusive.** Observe-se que as portas deste conector são instâncias de tipos de portas já definidos.

```
  Instantiate ProducerC  as Prod1;
  Instantiate ConsumerC  as Cons1;
  ...
  instantiate ProducerC as ProdN;
  instantiate ConsumerC as ConsN;
```

A diferença na criação de instâncias, com relação à configuração anterior, é a criação de várias instâncias de produtores e consumidores.

```
  Link Prod1 to Buff by MutexPCcon;
  Link Cons1 to Buff by MutexPCcon;
  ...
  link ProdN to Buff by MutexPCcon;
  link ConsN to Buff by MutexPCcon;
  ...
}
```

As várias instâncias de produtores e consumidores são ligadas ao *buffer Buff*. Observa-se que nestas ligações não são feitas referências às portas. Portas compatíveis são ligadas automaticamente. Todas as instâncias são interligadas pela mesma instância de conector.

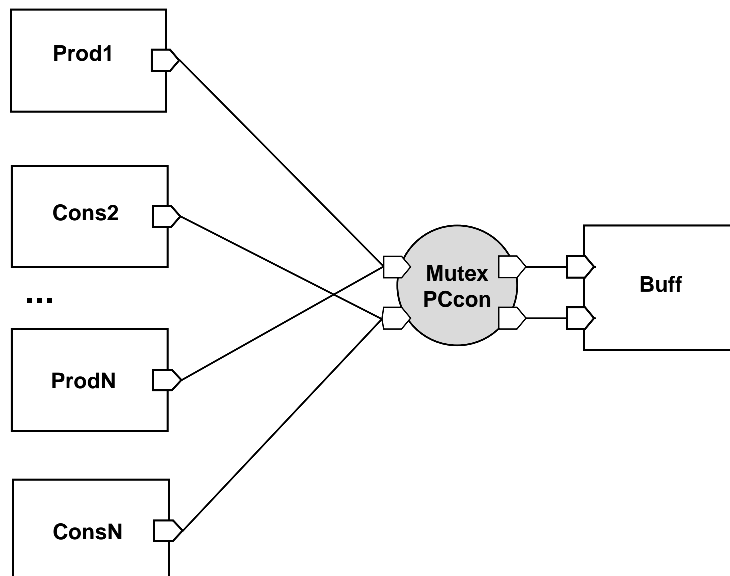


Figura 5.8 - *Buffer* com métodos sincronizados

Alternativamente, pode-se usar uma notação mais compacta para a especificação das instâncias dos módulos e a estrutura da aplicação, utilizando-se vetores:

```
  For (i = 1 to N) {
    Instantiate ProducerC  as Prod[N];
    Instantiate ConsumerC  as Cons[N];
  }
  ...

  For (i = 1 to N) {
    Link Prod[N] to Buff by MutexPCcon;
    Link Cons[N] to Buff by MutexPCcon;
  }
  ...
}
```

... ou ainda, como as ligações estão sendo feitas por contexto, é possível compactar-se as declarações de ligação:

```
  For (i = 1 to N) {
    Link Prod[N], Cons[N] to Buff by MutexPCcon;
  }
```

5.6.2 Limitando e sincronizando o acesso ao *buffer*

Observa-se na versão anterior que, embora cada acesso ao *buffer* seja feito em exclusão mútua, não existe um controle no uso do *buffer*. Por exemplo, um consumidor pode tentar consumir um item que ainda não tenha sido produzido, ou um produtor poderá tentar armazenar um item produzido sem que haja espaço disponível, superpondo este novo item no lugar de outro ainda não consumido.

Assim sendo, requisitos de sincronização são impostos à aplicação. Uma requisição de inclusão de item só deve chegar ao *buffer* se houver lugar disponível para armazená-lo. Da mesma forma, um item só poderá ser retirado do *buffer* se houver algum item armazenado. Como requisito adicional, independentemente da organização e do tamanho do *buffer* implementado no módulo *Buff*, deseja-se limitar, em 10 itens, a capacidade do *buffer* da aplicação.

Este novos requisitos podem ser configurados na aplicação, através do uso de guardas, para habilitar o tratamento das requisições pelo *buffer*, apenas quando as condições necessárias forem satisfeitas.

No conector desta solução, cada porta de acesso ao *buffer* é configurada com um guarda. Para a avaliação dos guardas é necessário que uma variável *n_itens* tenha sido declarada no módulo *Buff*. Esta variável representa o estado interno do módulo que será inspecionado pelo conector. No conector, duas variáveis de condição, *vazio* e *cheio*,

indicam se os guardas estão abertos ou fechados. A variável MAX_ITENS indica o tamanho máximo do *buffer*.

```
Module BoundedSincConcurBufferApplClass  {

module  BufferC {
  int n_itens;
...
  map CLASS Java "example.Sbuffer";
}
```

Na classe *BufferC*, declara-se a variável *n_itens* e indica-se o mapeamento para uma implementação.

```
...
  connector {
    condition vazio = true, cheio = false;
    staterequired int n_itens;
    int MAX_ITENS = 10;
    exclusive {
```

No conector, declara-se a lista de variáveis de condição (*condition*) e a lista de variáveis de estado (*staterequired*). Em seguida, declara-se a variável local MAX_ITENS.

```
      out port PutT {
        guard  (vazio) {
          after {
            cheio = true;
            if (n_itens == MAX_ITENS) {
              vazio = false;
        } } }
      } GPut;
```

Na porta *GPut*, uma requisição só será repassada se o guarda **vazio** for verdadeiro (indicando que existe lugar disponível no *buffer*).

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 {
        guard  (cheio) {
          after {
            vazio = true;
            if (n_itens == 0) {
              cheio = false;
        } } }
      } GGet;
    }
```

De forma análoga, na porta *GGet*, a requisição só será repassada se o guarda **cheio** for verdadeiro (indicando que existe pelo menos um item no *buffer*).

Ao completar o método, e antes de retornar o item retirado do *buffer*, o guarda **cheio** recebe falso, se o espaço do *buffer* estiver completamente vazio.

```
    in port GetT Get;
    in port PutT Put;
  } SincMutexPCcon;
  ...
}
```

Os módulos produtores e consumidores passam a ser ligados ao módulo *Buff* através do novo conector, *SincMutexPCcon*.

5.6.3 Distribuindo os módulos e acrescentando segurança

Em uma nova versão, introduzem-se mais dois requisitos: os módulos da aplicação devem ser instanciados em pontos distribuídos, e as informações devem ser armazenadas criptografadas no *buffer*.

A configuração anterior é alterada, de forma que cada módulo seja instanciado explicitamente em um nó diferente. Por decisão de projeto, os conectores devem

oferecer a comunicação implementada por *socket*. Se um conector *socket* não fosse explicitado, o ambiente de execução selecionaria um tipo de conector adequado para realizar a comunicação dos módulos. Tal seleção seria feita a partir de uma análise das referências dos endereços dos nós onde as instâncias foram criadas, e com base em uma classe de conector *defaul* (no protótipo em Java - capítulo 7 - optou-se por utilizar o mecanismo de RMI neste conector).

Para atender aos requisitos de criptografia, são selecionados conectores que encapsulam um mecanismo de criptografia e decriptografia, de forma que as informações a serem armazenadas no *buffer* sejam transportadas pela rede de forma segura. Ressalta-se que, neste exemplo, o programador precisa conhecer como o conector selecionado funciona. É necessário, por exemplo, saber que o conector que encapsula a função de criptografar age sobre o fluxo de informações que segue para o *buffer*, bem como que o conector que faz a decriptografia age sobre o fluxo de informações que retorna do *buffer*. Vale observar que aspectos não funcionais, como a segurança, podem ser descritos através de contratos de QoS (capítulo 6).

A topologia da aplicação é configurada, indicando-se a ordem adequada para a ligação dos conectores. O conector de coordenação, usado na versão anterior da aplicação, é reutilizado, e sua instância será criada no nó onde o módulo *Buff* está localizado.

```
Module
DistriCryptSincConcurBufferApplClass  {
  ...
```

```
  Connector Socket {
    map CLASS Java "conn.distrib.socket";
  }
  ...
  connector Crypt {
    map CLASS Java "conn.crypt";
  }

  connector Decrypt{
    map CLASS Java "conn.decrypt";
  }
```

Os conectores acrescentados possuem indicação do seu mapeamento para a implementação, através da cláusula **map**.

Propositadamente não foram declaradas instâncias para tais conectores neste ponto.

```
  Instantiate Buff at No1
  Instantiate ProducerC as Prod_x at No2;
  Instantiate ProducerC as Prod_y at No3;
  Instantiate ConsumerC as Cons_z at No4;

  link Prod_x to Buff by
       Crypt > Socket > SincMutexPCcon;
  link Prod_y to Buff by
       Crypt > Socket > SincMutexPCcon;
  link Cons_z to Buff by
       Decrypt > Socket > SincMutexPCcon;
}
```

Nas cláusulas de instanciação, indica-se o nó onde o módulo deve ser instanciado.

A ligação dos módulos é feita através da composição serial dos conectores, de acordo com a notação compacta discutida na seção A.6.1, apêndice A. A figura 5.9 apresenta a nova configuração.

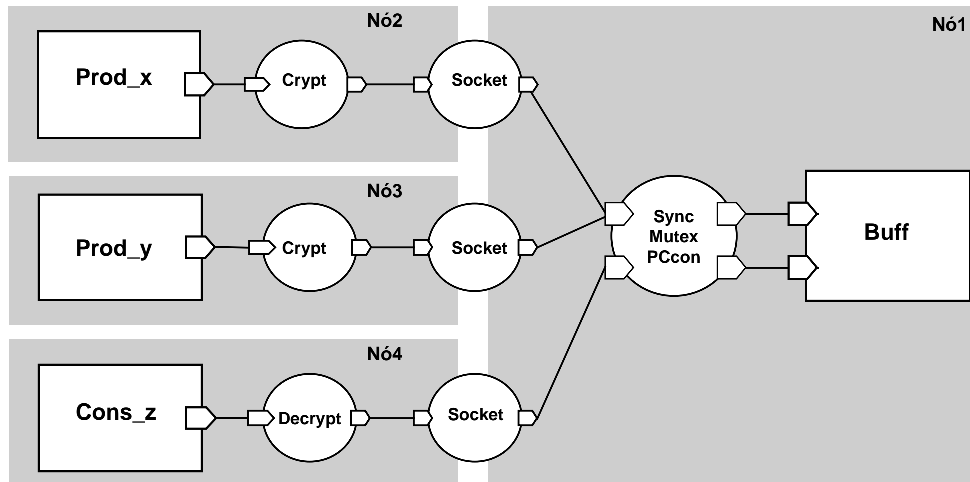


Figura 5.9 - Aplicação distribuída

Neste exemplo, fica claro que o programador da aplicação precisa conhecer os aspectos funcionais e não-funcionais da mesma, para poder fazer a seleção dos módulos e conectores e, em seguida, configurar sua arquitetura. Por exemplo, o conector *SyncMutexPCcon* precisa estar localizado junto ao módulo *Buff* para impor o contrato de coordenação. R-RIO, entretanto, facilita esta tarefa, provendo os mecanismos para descrever a arquitetura das aplicações e executar as mesmas.

5.6.4 Acrescentando aspectos por composição

Nas versões anteriores, os acréscimos de aspectos de sincronização e distribuição foram realizados por alterações incrementais na configuração que descreve a arquitetura da aplicação. Todos os componentes foram descritos sob o escopo direto da aplicação.

Uma alternativa disponível em CBabel, para compactar a descrição da aplicação, é a utilização de composição para descreverem-se elementos com funcionalidade agregada.

No caso de nosso exemplo, podemos ter um módulo composto, com o objetivo de receber os pedidos de produtores e consumidores, sincronizar e coordenar estes pedidos e depois repassá-los para o módulo *buffer*. Denominaremos esta nova classe de módulo de *CompBoundBuffClass*.

Código	Descrição
`module SbufferC {...}`	Uma classe de módulo *SBufferC* está disponível para ser reutilizada (através de uma declaração fora do escopo de outro módulo, ou através de biblioteca).
`connector PCGuard {...}`	O mesmo ocorre para o conector, que encapsula os aspectos de coordenação.
`Module CompBoundBuffClass {`	
`  Instantiate SBufferC as Buff;`	A instância *Buff* da classe de módulo *SbufferC* tem escopo limitado à classe composta *CompBoundBuffClass*.
`  link Buff to PCGuard;`	Uma instância do conector *PCGuard* é criada automaticamente (o nome da referência da instância é conhecido pelo suporte) e ligada por contexto ao módulo *Buff*.

`  export  PCGuard.Put;`
`  export  PCGuard.Get;`
`} // CompBoundBuffClass`
	As portas que devem ter visibilidade externa são exportadas.

`module  PCCBBApplClass{`
`  instantiate ProducerT  as Prod;`
`  instantiate ConsumerT  as Cons;`
`  instantiate CompBoundBuffClass as Buff;`
`  ...`
	A descrição da aplicação fica compacta. Os módulos são instanciados, observando-se que o nome da instância do módulo composto também é "Buff".

`  link Prod to Buff;`
`  link Cons to Buff;`
`  ...`
`}`
	Em seguida os módulos são interligados (através de um conector *default*). A figura 5.10 apresenta o resultado.

Através deste exemplo explora-se a capacidade de composição e reutilização de módulos do *framework* R-RIO. A idéia é ter disponível uma classe de módulo *buffer* genérica 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, bem como ser usado na composição de outras aplicações.

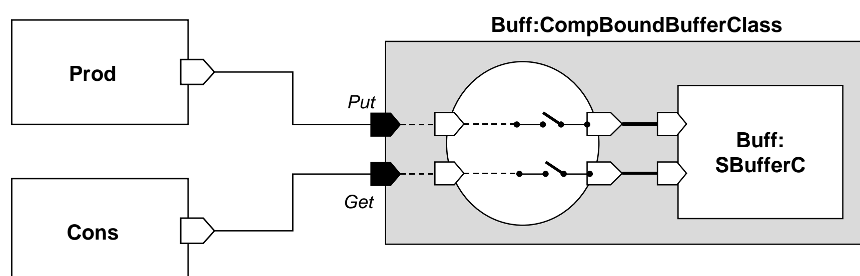


Figura 5.10 - Aplicação com módulo composto

Atualização das versões

Uma observação deve ser feita com relação à seqüência dos exemplos. Entende-se que, entre a execução de uma versão e de outra imediatamente posterior, a descrição da arquitetura 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, entretanto, não é estritamente obrigatória. Como discutido na seção 5.5, uma das possibilidades admitidas em R-RIO consiste na execução de alterações na configuração de uma aplicação, com a mesma em atividade. Por exemplo, o acréscimo de mais um produtor ou consumidor à configuração da seção 5.6.1 pode ser feito sem a necessidade de interferência direta na execução dos outros módulos.

Ainda no exemplo apresentado, o módulo *Buff* 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 módulo *buffer*, com as novas características configuradas, seria instanciado e ligado aos demais módulos da aplicação. Esta mudança poderia implicar na perda dos dados que estivessem armazenados no *buffer* original, em problemas de consistência, e em alguns problemas de comunicação. Assim, o projetista teria que aplicar as técnicas adequadas à manutenção da consistência, como discutido na seção 5.5.3.

5.7 Conclusão

Neste capítulo apresentamos nossa abordagem para aspectos não-funcionais em arquiteturas de *software*, tais como a interação, distribuição e coordenação, no contexto do *framework* R-RIO. Discutiu-se como tais aspectos são tratados, em nossa proposta, através de contratos. Examinou-se, também, quais são os elementos do *framework* R-RIO que oferecem suporte a tais contratos. Por exemplo, os contratos de aspectos não-funcionais podem ser encapsulados em conectores. Observou-se que a possibilidade de seleção e composição dos elementos da arquitetura de uma aplicação, aliada à facilidade de configuração de aspectos não-funcionais nos conectores, favorece a reutilização de componentes e a concepção de aplicações adaptadas para necessidades específicas.

Através da seqüência de exemplos da seção 5.6, mostrou-se como contratos de aspectos não-funcionais podem ser representados em ADLs, como CBabel. Na arquitetura da seção 5.6.3, aspectos de comunicação e segurança foram contemplados,

simplesmente selecionando-se os conectores adequados e configurando-os em uma estrutura coerente. Este exemplo também nos leva a observar que, à medida em que novas classes de módulos e conectores tornam-se disponíveis, a configuração de uma aplicação fica mais simples e compacta. É possível ampliar tal facilidade com a criação de elementos genéricos, de ampla utilização, e com a reunião destes elementos em módulos compostos.

O próximo capítulo é reservado à discussão de nossa abordagem para a especificação e configuração de contratos para aspectos de qualidade de serviço (QoS). Encaixam-se nesta categoria diversos aspectos não-funcionais e operacionais, tais como características da comunicação entre módulos em arquiteturas distribuídas, tolerância a falhas e segurança. Nesta abordagem, uma estrutura geral é proposta para o suporte de aplicações que requeiram QoS, e nela estão incluídos módulos especiais para o gerenciamento de contratos destes aspectos.

Esta página foi intencionalmente deixada em branco

Capítulo VI

Aspectos de Qualidade de Serviço

6.1 Introdução

Aspectos de qualidade de serviço (QoS), em geral, não fazem parte da funcionalidade básica das aplicações. A introdução de QoS no projeto de uma aplicação apresenta desafios, na medida em que podem estar envolvidos recursos de *hardware* e *software*, potencialmente distribuídos em uma rede. Além disso, a introdução de aspectos de QoS normalmente requer a programação de mecanismos específicos dos sistemas de suporte.

Como apontado no capítulo 5, uma aplicação pode ter requisitos de QoS tão diversificados quanto a comunicação, segurança ou tolerância a falhas. O programador desta aplicação teria que incluir, entre as suas tarefas, a programação de ações concretas sobre o sistema de suporte ou rede de comunicação, para impor e monitorar QoS. Por exemplo, em uma aplicação de vídeo-sob-demanda, além dos seus requisitos funcionais, um módulo de exibição necessita de um fluxo de 30 quadros/segundo chegando ao seu *buffer*, para se obter uma apresentação de qualidade. Outros requisitos de QoS, como atraso máximo, ou a taxa de erros aceitável, também são comuns neste tipo de aplicação [184]. De que modo o programador deve tratar requisitos tão diversos? Estes deveriam estar embutidos no código da aplicação?

É desejável que aspectos de QoS de uma aplicação sejam especificados, implementados e configurados separadamente dos aspectos funcionais. Além disso, se espera que seja possível sistematizar o mapeamento entre a especificação de requisitos de QoS e uma possível implementação.

Neste capítulo, apresentamos uma proposta para contemplar aspectos de QoS, no nível da arquitetura de *software*. Discute-se como a visão de contratos de QoS,

introduzida no capítulo 5, pode ser descrita em ADLs como CBabel. Discute-se, também, a infra-estrutura necessária para configurar, executar e monitorar uma aplicação com QoS, em um ambiente de suporte à configuração, como R-RIO.

O capítulo é organizando da seguinte forma: inicialmente, apresenta-se uma estrutura geral para a configuração de arquiteturas de *software* com QoS; mostra-se, em seguida, como contratos de QoS podem ser descritos através de CBabel. O exemplo do produtor-consumidor é utilizado para ilustrar este ponto. Discute-se, então, a funcionalidade dos elementos de infra-estrutura que possibilitam o mapeamento de especificações de QoS para a estrutura proposta. Antes da conclusão, alguns trabalhos correlatos são enumerados e comentados.

6.2 QoS em R-RIO

O planejamento de aspectos de QoS de uma aplicação, durante as primeiras fases de projeto, tende a facilitar a detecção de problemas, tais como a indisponibilidade de recursos, em uma fase inicial da concepção desta aplicação [185, 186]. Adicionalmente, para facilitar a evolução dinâmica da mesma, seria conveniente que a separação de interesses fosse mantida, de forma que apenas os elementos de meta-nível e a infra-estrutura de suporte precisassem incorporar funcionalidades para impor QoS. Considere-se, por exemplo, a necessidade de se mudar a implementação de um módulo, mantendo os requisitos de QoS, ou mesmo usando requisitos de QoS diferentes, para instâncias de módulos diferentes de uma mesma classe. É comum, também, que parâmetros de QoS precisem ser negociados. Se não fosse observada a separação de interesses, a negociação teria de ser feita, ou pelo menos iniciada, pelos próprios módulos funcionais da aplicação. Outros pesquisadores suportam esta argumentação [187, 188].

A inclusão de QoS em nossa proposta objetiva mostrar que é factível e potencialmente vantajoso considerar os aspectos de QoS de uma aplicação ainda na etapa de descrição da arquitetura de *software*. Objetiva-se também propor uma estrutura que suporte a execução de aplicações com QoS e a gerência de aspectos de QoS no nível da arquitetura de *software*. No contexto desta proposta, QoS será considerado um aspecto relacionado a características não-funcionais ou operacionais das aplicações, sem restrições para um domínio particular de características.

6.2.1 Visão Geral

Nossa abordagem para configurar arquiteturas de *software* com aspectos de QoS é composta dos seguintes elementos:

- **Módulos da Aplicação**: demandam características de QoS, mas não são programados para garantir estas características;
- **Conectores de QoS**: conectores, selecionados e configurados para atender às características de QoS descritas em um contrato de QoS;
- **Contratos de QoS**: descrevem as características de QoS oferecidas e requeridas pelos módulos da aplicação. Uma ADL, como CBabel, pode integrar a sintaxe necessária para descrever contratos de QoS;
- **Configurador de QoS**: módulo especial com a capacidade de selecionar e compor conectores de QoS para satisfazer a um contrato; e
- **Monitor de QoS**: módulo especial que monitora se as características de QoS de determinado contrato estão sendo satisfeitas.

A figura 6.1 apresenta a visão geral de uma configuração de arquitetura de *software* com aspectos de QoS, que utiliza os elementos descritos anteriormente. No diagrama são representados os componentes de uma pequena aplicação cliente-servidor e os elementos de suporte à configuração. Representam-se, também, os elementos de infra-estrutura de QoS, consistindo do contrato de QoS e dos módulos *Configurador* e *Monitor* de QoS.

A aplicação consiste de um módulo *Cliente* que faz requisições a um módulo *Servidor*. A interação entre estes módulos apresenta requisitos de QoS. Estes requisitos devem ser descritos através de contratos de QoS, os quais ficam disponíveis para consulta, como informações de meta-nível. Um conector, chamado *Conector de QoS*, é selecionado para intermediar a interação entre os módulos *Cliente* e *Servidor* e, adicionalmente, atender os requisitos de QoS descritos no contrato.

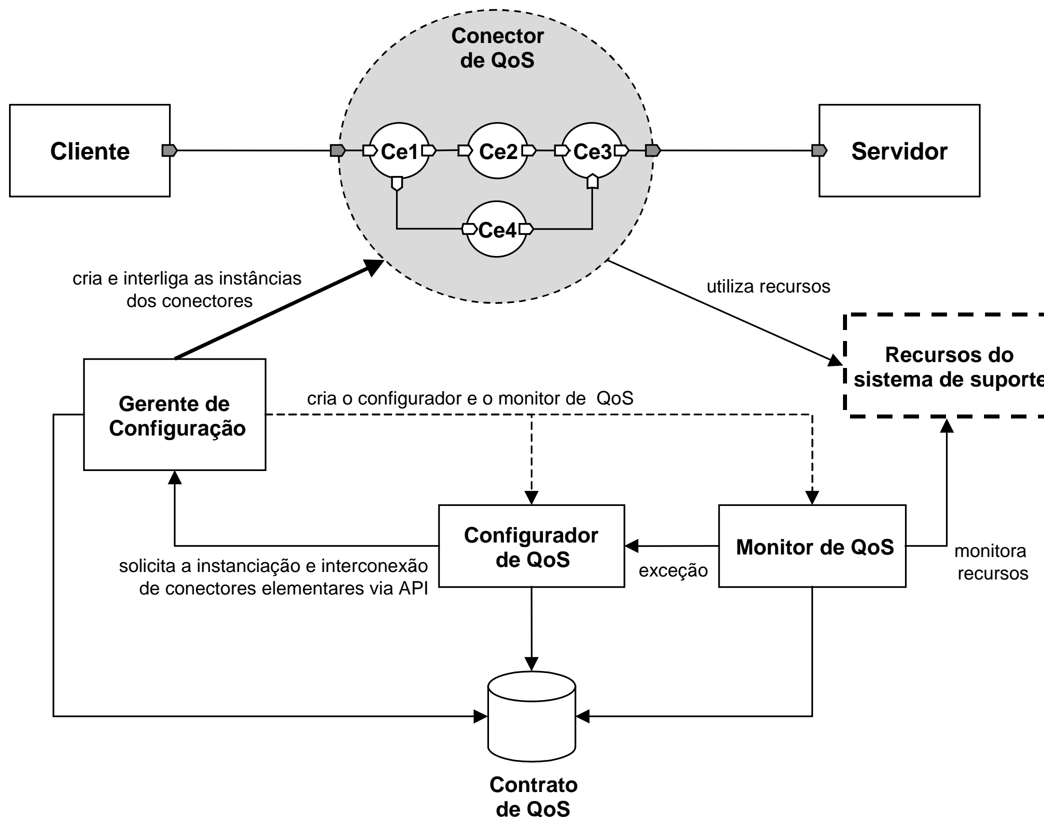


Figura 6.1 - Estrutura para viabilizar arquiteturas de *software* com QoS

Ao configurar uma aplicação para executar, o *Gerente de Configuração* tem, entre as suas tarefas, a de criar as instâncias dos conectores que interligarão os módulos. Entretanto, se existe um contrato de QoS descrito, o *Gerente de Configuração* instancia módulos especiais, chamados *Configurador* e *Monitor de QoS*. Estes módulos, em conjunto, têm a função de gerenciar os aspectos de QoS especificados no contrato.

Um módulo *Configurador de QoS* é programado para selecionar / compor um *Conector de QoS*, que possa atender os requisitos de QoS descritos no contrato. Para realizar esta tarefa, este módulo deve ter acesso ao contrato de QoS e a uma biblioteca de conectores elementares. Uma vez obtido o *Conector de QoS*, o *Configurador de QoS* solicita a instanciação do mesmo ao *Gerente de Configuração*.

Um módulo *Monitor de QoS* é programado com o fim de monitorar um contrato de QoS. Para fazê-lo, este módulo tem acesso ao contrato de QoS e aos recursos de suporte que devem ser monitorados. Ao detectar alguma alteração nas condições dos recursos monitorados, o *Monitor* informa o fato ao *Configurador de QoS*. Este pode, então, baseado no contrato de QoS, reconfigurar o conector para manter válido o referido contrato.

Em [187, 195, 196, 199], são apresentadas estruturas para controle de contratos de QoS, semelhantes ao sistema formado pelo *Configurador* e *Monitor de QoS*. Em R-RIO, entretanto, o suporte explícito para configuração facilita a programação das operações necessárias para a gerência de QoS.

Nas subseções seguintes, apresentamos mais detalhes sobre os elementos de nossa abordagem para QoS.

6.2.2 Contratos de QoS em CBabel

Para descrever contratos de QoS, poderíamos desenvolver uma extensão de CBabel ou adaptar uma das linguagens de QoS descritas na literatura. Optamos por utilizar os conceitos de *categorias*, *propriedades* e *perfis* de QoS descritos em [185], que possibilitam descrições abstratas de QoS. Estes conceitos foram adaptados para o contexto de arquiteturas de *software*. O objetivo é permitir, em um primeiro passo, a descrição das características de QoS utilizadas em uma aplicação. Em seguida, descreve-se como um módulo servidor está disposto a prover tais características, e como um módulo cliente requer as mesmas. Em um último passo, as demandas e ofertas de QoS dos módulos que vão interagir são unidas em um contrato. Nesta contexto, utilizamos as seguintes abstrações:

- ***Categoria.*** Uma categoria de QoS define um aspecto particular de QoS. Uma categoria de QoS tem um nome e um conjunto de propriedades. Por exemplo, aspectos relacionados com a comunicação, podem ser agrupados em uma categoria de QoS.

- ***Propriedade.*** Uma propriedade de determinada categoria de QoS possui um nome e um domínio de valores. Três tipos de domínio podem ser utilizados em uma propriedade: *sets* (conjuntos), *enumerated* (enumerado) e *numeric* (numéricos). Os valores de uma propriedade quantificam-na ou a restringem. Assim, como propriedades da categoria de QoS de comunicação, podemos ter: protocolo, controle de fluxo e controle de erro.

- ***Perfil.*** Um módulo pode ser associado a um perfil de QoS. Um perfil de QoS descreve como o módulo provê ou demanda QoS. Um perfil de QoS é formado por um conjunto de instâncias de categorias de QoS. Na descrição de um perfil de QoS são especificados os pontos de junção entre as categorias de QoS, utilizadas neste

perfil, e a interface do módulo. As categorias de QoS de um perfil podem ter escopo de interface ou podem ser associadas a uma porta individual. Perfis de QoS são referenciados nos contratos de QoS.

- ***Contrato.*** Um contrato de QoS liga dois ou mais perfis de QoS de módulos que vão interagir. A ligação é possível quando um dos módulos oferece um conjunto de categorias de QoS em seu perfil, e o outro módulo requisita um conjunto compatível de categorias de QoS.
- ***Política de Exceção.*** Em um contrato de QoS, pode-se especificar uma política a ser usada no caso de quebra do mesmo. Por exemplo, pode se optar por receber um aviso de exceção e terminar a ligação entre os módulos, ou deixar o sistema de suporte renegociar o contrato e manter esta ligação com o QoS contratado.

Na próxima seção ilustramos uma possível utilização de contratos de QoS em CBabel, através do exemplo produtor-consumidor.

6.2.2.1 Exemplo de contrato de QoS

Tomamos o exemplo do produtor-consumidor e adicionamos, como requisitos de QoS, a especificação de alguns parâmetros do protocolo de comunicação que suporta a interação entre produtor e *buffer*. A figura 6.2 ilustra que a interação do módulo *Prod* com o módulo *Buff* requer certas propriedades do protocolo TCP, para funcionar adequadamente. Cada módulo possui um perfil de QoS. Neste perfil são descritas as propriedades relacionadas à comunicação. No perfil do módulo *Buff*, são listadas as propriedades disponíveis para serem utilizadas na comunicação com outros módulos. No perfil do módulo *Prod*, apresentam-se as propriedades desejadas para a comunicação. Ao estabelecer-se um contrato, um conector adequado é selecionado para realizar a comunicação, atendendo aos requisitos do contrato.

Inicia-se a especificação dos aspectos de QoS em CBabel, descrevendo-se a categoria de QoS *Communication* e, em seu corpo, suas propriedades (listagem 6.1). Observa-se que cada propriedade é qualificada. Por exemplo, *protocol* (linha 2) é uma lista (*enum*) de protocolos possíveis, os quais admitem seleção para uma dada interação. A propriedade *MSS* (*maximum segment size*) é qualificada como um valor numérico (linha 6) e os valores preferidos (que significam melhor qualidade) são os maiores (*increasing*). As outras propriedades são qualificadas de forma semelhante.

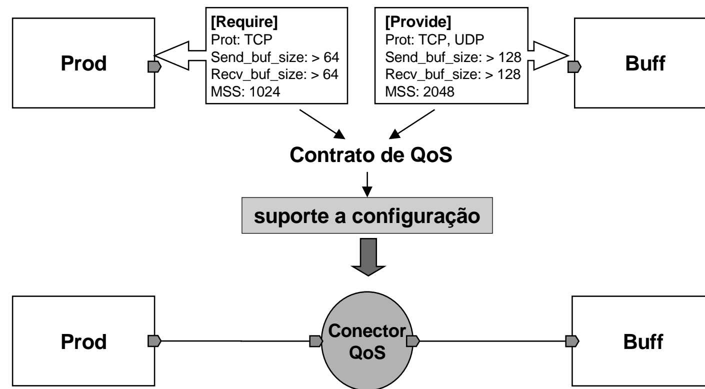


Figura 6.2 - QoS para protocolos de comunicação

```
1  QoScategory  Communication {
2     protocol: enum {UDP, TCP, RTP} ;
3     slidingWindowSize: increasing numeric ;
4     send_buf-size: increasing numeric ;
5     recv_buf_size: increasing numeric ;
6     MSS: increasing numeric ;
7     errorDetection: increasing enum {parity, crc16, crc32}
8                    with order {parity < crc16 < crc32}
9  }
```

Listagem 6.1 - Categoria de QoS para aspectos de comunicação

É importante notar que a categoria de QoS *Communication* ainda não está relacionada aos módulos *Prod* e *Buff*. Uma categoria de QoS pode ser utilizada na definição do perfil de QoS para uma classe de módulo ou para uma dada instância de módulo. No caso de o perfil de QoS ser associado à classe de módulo, toda instância de módulo derivada desta classe será associada a este perfil. No outro caso, apenas a instância de módulo será associada. Para o exemplo, o perfil do módulo *Buff* é apresentado na listagem 6.2, linhas 1 a 8, e o perfil do módulo *Prod*, entre as linhas 10 e 17.

```
1   QoSprofile ComProfProv for Buff {
2      provide Communication {
3         protocol: {UDP, TCP} ;
4         send_buf-size: <= 128 ;
5         recv_buf_size: <= 128 ;
6         MSS: 2048 ;
7      }
8   }
9
10  QoSprofile ComProfReq for Prod {
11     require Communication {
12       protocol: TCP ;
13        send_buf-size: > 64 ;
14        recv_buf_size: > 64 ;
15        MSS: 1024 ;
16     }
17  }
```

Listagem 6.2 - Perfil de QoS para o módulo *Buff* e *Prod*

Em um perfil de QoS, ao se associar uma categoria de QoS a um ponto de junção da interface de um módulo, especifica-se também se o módulo está oferecendo (***provide***) ou se ele está requerendo (***require***) as propriedades desta categoria. No exemplo, o perfil de QoS do módulo *Buff* indica que a categoria *Communication* será utilizada (listagem 6.2, linha 2) para todo módulo. Neste caso, todas as portas do módulo podem interagir através dos protocolos TCP e UDP (linha 3), com um *buffer* de transmissão (linha 4) e recepção (linha 5) de até 128 bytes, e um tamanho máximo de segmento de 2048 (linha 6). O perfil de QoS do produtor indica os requisitos de QoS deste módulo: o protocolo TCP deve ser usado (linha 12), com um *buffer* de transmissão (linha 13) e recepção (linha 14), pelo menos maiores que 64 bytes, com um tamanho máximo de segmento de 1024 (linha 15).

Após a definição das categorias de QoS e a associação dos módulos a perfis de QoS, a configuração da aplicação com QoS é, então, definida. Um contrato de QoS é associado ao conector que vai mediar as interações entre os módulos. A cláusula *QoS contract* vincula os perfis de QoS dos módulos que irão interagir. No exemplo dado, a configuração da aplicação é essencialmente a mesma, descrita no capítulo 5, em sua versão com módulos distribuídos e *buffer* limitado. Entretanto, para a nova versão, incluímos um contrato de QoS no conector que interliga os módulos *Prod* e *Buff*. Na listagem 6.3, apresenta-se o conector *QoSDistrib_Coord*, que contém este contrato. Observa-se que uma política para tratar exceções também pode ser especificada. No exemplo, o parâmetro ***renegotiate***, linha 4, indica que, se ocorrer uma violação no contrato, o suporte à QoS deve iniciar um procedimento de renegociação, seguido de uma reconfiguração.

```
1  connector QoSDistrib_Coord{
2  ...
3      QOS contract {
4       (Prod, Buff) renegotiate;
5      }
6  }
7
8  link Prod to  Buff  by  QoSDistrib_Coord;
```

Listagem 6.3 - Contrato de QoS para entre os módulos *Prod* e *Buff*

No exemplo anterior, não existem requisitos de QoS para o módulo consumidor. Assim sendo, a configuração associada ao módulo consumidor não sofreu alterações.

A descrição de um contrato de QoS pode ser submetida a uma verificação de coerência e consistência. A verificação realizada neste contexto é estática. Um perfil

mais *forte* (com maiores restrições nos valores das propriedades) é conforme a um perfil mais *fraco* (menos restritivo nos valores das propriedades). No mesmo exemplo, o módulo *Prod* não pode requerer ao módulo *Buff* um nível de QoS, que este não possa oferecer. Deste modo, se no perfil de QoS do produtor fosse especificado *MMS: 4096* (ao invés de *MMS: 1024*), o contrato não poderia ser realizado, pois a qualidade máxima oferecida pelo módulo *buffer* é *MMS: 2048*. É possível a um projetista fazer experiências com QoS, utilizando-se desta verificação, antes de qualquer outro passo na implementação da aplicação. O procedimento de verificação também pode ser automatizado por um compilador.

6.2.2.2 Outros exemplos

Além dos aspectos de comunicação, apresentados na aplicação produtor-consumidor, outras categorias de QoS podem ser descritas, segundo nossa proposta. Por exemplo, replicação e tolerância a falhas são considerados aspectos de QoS, pois são relacionados a características operacionais de uma aplicação [189]. A descrição de tais aspectos em CBabel seria simples. Por exemplo, a descrição de uma categoria de QoS *Replication*, para replicação de módulos, admitiria a forma apresentada na listagem 6.4.

```
1  QoS category  Replication {
2     numberOfReplicas: increasing numeric ;
3     replicationPolicy: enum {PassiveColdStandBy, PassiveHotStandBy,
4                              SimpleActive, ActiveCompetitive, ActiveCiclic}
5     faultTolerance: increasing enum {restrict, byzantine}
6                    with order {restrict < byzantine }
7  }
```

Listagem 6.4 - Categoria de QoS para aspectos de replicação

Três propriedades são descritas na categoria de QoS *Replication*. A primeira, *numberOfReplicas* (linha 2) indica o número de réplicas para um dado módulo servidor (ou requerido por um módulo cliente). Em seguida, aparece *replicationPolicy* (linha 3), que declara qual política de replicação pode ser utilizada. Como não existe uma forma numérica de comparação entre estas opções, elas não são ordenadas, apenas enumeradas. A terceira propriedade, *faultTolerance* (linha 5), está relacionada com os tipos de falhas suportados.

O aspecto de replicação e tolerância a falhas poderia ser usado em uma aplicação, configurando a categoria de QoS *Replication* adequadamente no perfil dos módulos. Um módulo servidor teria, em seu perfil de QoS, a categoria *Replication*

configurada como sendo provida. Um módulo cliente poderia ter, em seu perfil de QoS, a categoria de QoS *Replication* descrita como sendo requerida.

Um outro exemplo ilustrativo seria uma aplicação com um servidor e um cliente SSH - *Secure Shell*, implementando um terminal virtual seguro. Antes de se estabelecer uma sessão, o módulo cliente SSH deve selecionar alguns parâmetros de operação. Tais parâmetros são, em sua maioria, relacionados à segurança, e às características do protocolo de comunicação. Idealmente estes aspectos poderiam ser encapsulados em um *Conector de QoS*. Os módulos funcionais implementariam apenas a função de um terminal virtual. A listagem 6.5 apresenta uma possível descrição para a categoria de QoS *SSH*, contendo as propriedades para configurar a qualidade de serviço de uma sessão SSH.

```
1  QoS category  SSH {
2     port: enum {ssh , telnet};
3     cipherType: enum {idea, arcfour} ;
4     compressionLevel: increasing numeric ;
5     autheticationType: enum {RSA, password, both} ;
5     connectionAttempts: increasing numeric ;
6     X11forward: enum {yes, no} ;
7     keepAlive: enum {yes, no} ;
8     strictHostKeyChec: enum {yes, no} ;
9  }
```

Listagem 6.5 - Categoria de QoS para SSH

Em um cenário real, o perfil associado ao servidor deve indicar quais são os valores das propriedades com os quais ele pode trabalhar. Por sua vez, o perfil do cliente deve indicar os valores aceitáveis destas propriedades. Pode ocorrer, por exemplo, que um módulo cliente disponha apenas de um determinado tipo de criptografia (linha 3) ou que este só consiga se autenticar por senha (linha 5). Neste caso, um contrato somente será estabelecido se o servidor puder oferecer, pelo menos, as citadas propriedades.

6.2.3 *Configurador de QoS*

O *Configurador de QoS* é um módulo especial, programado para gerenciar um contrato envolvendo uma categoria específica de QoS. Uma das tarefas de um *Configurador de QoS* consiste em combinar conectores elementares e recursos de sistema de suporte, em um conector capaz de atender as características de QoS requeridas. Depois de configurar o conector de QoS, o *Configurador de QoS* solicita ao

Gerente de Configuração, através da API de configuração, a instanciação e interligação dos elementos deste conector.

No momento da carga de uma aplicação, quando um comando de configuração de ligação é iniciado, o *Gerente de Configuração* toma a referência ao conector que vai mediar as interações, e inspeciona sua configuração. Se existe um contrato de QoS associado a este conector, o *Gerente de Configuração* cria uma instância dos módulos *Configurador* e *Monitor de QoS* apropriados, e aguarda uma resposta do *Configurador de QoS*. Para o *Gerente de Configuração*, o comando de ligação é considerado completo apenas quando o *Configurador de QoS* conclui sua tarefa, ou seja, a composição e a instanciação dos conectores componentes, e a reserva dos recursos de sistema. Se o *Configurador de QoS* não puder compor um conector de QoS, o *Gerente de Configuração* considera que o comando de ligação falhou.

Um *Configurador de QoS* pode, adicionalmente, ser programado para usar o serviço de um *broker*, a fim de descobrir onde se encontram os recursos e conectores elementares necessários para compor o conector de QoS. Em princípio, o funcionamento do *broker*, ou os meios pelos quais ele é alimentado com as informações, está fora do escopo desta proposta. Mais informações podem ser obtidas em [190], por exemplo.

Um *Configurador de QoS* pode ser programado para solicitar a execução de outros comandos de configuração, além daqueles indicados para instanciar o conector de QoS. Por exemplo, no caso de replicação de módulos, o *Configurador de QoS* seria encarregado de instanciar tantas réplicas quantas tivessem sido configuradas no contrato de QoS.

Conforme o que foi focalizado na seção 6.2.2, um contrato de QoS também pode conter a informação de uma política a ser adotada, se este não puder ser mantido. Quando uma quebra de contrato ocorre, o módulo *Monitor de QoS* comunica este fato ao *Configurador de QoS*, que deve estar programado para tratar este tipo de evento. Dependendo da política selecionada no contrato, o *Configurador de QoS* pode: (i) gerar uma condição de exceção para o módulo cliente, mantendo a ligação sem os requisitos de QoS, (ii) renegociar o contrato, ou (iii) terminar a interação. No caso de uma renegociação, o *Configurador de QoS* tenta reconfigurar o conector de QoS. Se não for possível reconfigurar o conector para manter o nível de QoS requerido, a ligação é

terminada. Um *Configurador de QoS* pode encapsular mecanismos sofisticados, como o descrito em [191], para implementar as políticas de manutenção de QoS.

6.2.4 *Monitor de QoS*

Uma das funções de um suporte para QoS é o monitoramento de contratos. As propriedades de QoS contratadas, e inicialmente impostas, devem ser monitoradas para verificar se estão sendo satisfeitas. Se o valor de uma propriedade de QoS monitorada não mais atende aos valores contratados, alguma ação deve ocorrer. Por exemplo, os módulos envolvidos podem receber uma notificação, na forma de uma exceção.

A medição de um valor associado a uma propriedade de QoS pode ser feita em alguns níveis do sistema. Assim, uma medida pode ser obtida pelo *hardware* do computador, nos dispositivos da rede de comunicação ou no sistema operacional. As medidas também podem ser obtidas em um nível mais próximo da aplicação, tal como por exemplo, no *middleware* de suporte. Além de medir continuamente as condições efetivas de QoS, o sistema de monitoramento precisa conhecer o contrato de QoS para verificar se os limites estão sendo violados.

Em nossa proposta, um contrato de QoS é monitorado, durante a operação da aplicação, por um módulo *Monitor de QoS.* Cada categoria de um contrato de QoS possui seu módulo *Monitor de QoS*. Este módulo tem acesso às informações do contrato de QoS que devem ser monitoradas. Para realizar as medições apropriadas, um módulo *Monitor de QoS* pode ser programado para acessar vários níveis do sistema, como discutido anteriormente.

Cada *Monitor* obtém continuamente as medidas das propriedades da categoria de QoS, pela qual ele é responsável, e compara-as com as informações definidas no contrato. No caso em que o valor medido de alguma propriedade esteja fora da faixa contratada (ou seja, um contrato de QoS não está mais sendo "honrado"), o *Monitor de QoS* invoca o módulo *Configurador de QoS* correspondente, com uma lista das propriedades fora da especificação. O *Configurador de QoS*, então, inicia um procedimento, baseado no contrato de QoS e em sua programação interna.

6.2.5 Conectores de QoS

A infra-estrutura necessária para o suporte de arquiteturas de *software* com requisitos de QoS é diversificada. Por exemplo, os recursos requeridos para garantir uma comunicação confiável são diferentes dos requeridos para tolerância a falhas. O *framework* R-RIO não oferece diretamente a infra-estrutura ou mecanismos para impor requisitos de QoS específicos. Pressupõe-se que esta infra-estrutura já esteja disponível e que os mecanismos adequados à sua utilização possam ser encapsulados em conectores.

No modelo de componentes, apresentado na seção 4.2 - capítulo 4, e na discussão da seção 5.2 - capítulo 5, ressaltou-se que é possível ao conector acessar recursos dos subsistemas de suporte. Na abordagem proposta para QoS, esta capacidade é empregada para suprir os contratos de QoS. Um *Conector de QoS* pode, por exemplo, ser programado para utilizar serviços Xbind [192] com a finalidade de iniciar a sinalização adequada em redes ATM. Este conector também pode utilizar APIs do escalonador de CPUs para adequar políticas de escalonamento ou fazer reservas de memória, se isso for necessário para impor um contrato. Em outro nível, um *Conector de QoS* pode utilizar serviços de *middleware*, como por exemplo o RT-CORBA, com vistas à configuração do próprio ambiente de execução para determinadas condições de operação.

Para gerenciar um contrato de QoS que envolva mais de uma categoria de QoS ou propriedades muito diferentes, pode-se utilizar uma composição de conectores elementares, cada qual programado com a finalidade de dar suporte a uma parte do contrato. Por exemplo, para impor um contrato que utiliza a categoria de QoS *SSH*, descrita na seção 6.2.2, conectores diferentes poderiam ser responsáveis pelas propriedades de criptografia, compressão e autenticação. Isto ajuda a observação da separação de interesses e é especialmente conveniente se os conectores elementares já estiverem disponíveis.

Ao instanciar um *Conector de QoS*, o *Configurador de QoS* deve passar os argumentos adequados para que este inicie sua operação corretamente. Assim, um *Configurador de QoS* precisa ser programado, considerando-se a categoria de QoS que ele vai gerenciar e os parâmetros de inicialização dos conectores a serem utilizados. O valor dos argumentos é dependente do contrato de QoS.

6.2.6 Avaliação

A abordagem proposta para configuração de QoS em arquiteturas de *software* apresenta características desejáveis sob o ponto de vista da engenharia de *software*:

- A implementação da política de gerência de QoS é encapsulada em módulos *Configurador de QoS* e *Monitores*. Separam-se, assim, os aspectos de QoS da funcionalidade específica da aplicação.

- A arquitetura de QoS proposta permite, em princípio, que novos contratos e categorias de QoS sejam introduzidos dinamicamente nas aplicações. Isto é possível porque a implantação e a manutenção de contratos de QoS são realizadas através de comandos de configuração, seguindo o modelo proposto para o *framework* R-RIO.

- Uma nova categoria de QoS pode ser introduzida e descrita em CBabel, utilizando-se as abstrações apresentadas na seção 6.2.2. Como infra-estrutura para esta nova categoria, devem ser desenvolvidos os módulos *Configurador de QoS* e *Monitor*, que suportarão a mesma. Deve-se garantir também que estejam disponíveis conectores para impor as propriedades de QoS desejadas. O mapeamento entre a descrição em CBabel e os módulos de suporte será feito a partir do nome desta nova categoria.

- A construção dos módulos *Configurador* e *Monitor de QoS* é, em grande parte, reutilizável. As interações entre estes módulos e o *Gerente de Configuração* é realizada através das APIs de configuração e reflexão arquitetural do *framework* R-RIO. A criação e a inicialização destes módulos pelo *Gerente de Configuração* também é padronizada. Os módulos *Configurador* e *Monitor de QoS* seguem o modelo de componentes do *framework* R-RIO e, portanto, podem ser instanciados e terminados dinamicamente. Informações específicas para a operação destes módulos podem ser passadas como argumentos, extraídos da descrição dos contratos de QoS.

  A técnica de interface entre o *Gerente de Configuração* e os módulos *Configurador* e *Monitor de QoS* é comparável ao mecanismo de CGI (*Common Gateway Interface*), utilizado para viabilizar aplicações na Web. Um servidor HTTP pode receber uma requisição, contendo um pedido de execução de uma aplicação, e uma lista de pares nome/valor. Este servidor inicia a aplicação, passando a lista de pares nome/valor como argumento para esta aplicação. Cada aplicação interpreta os

valores passados, de acordo com sua programação interna. No caso dos módulos *Configurador* e *Monitor de QoS*, um contrato de QoS, com formato padrão, é utilizado como argumento básico sobre o qual procedimentos são executados. Os procedimentos específicos, executados por estes módulos, visando o gerenciamento de uma dada categoria de QoS, devem ser programados explicitamente.

Um ambiente de desenvolvimento, empregando nossa abordagem, poderia dispor de uma biblioteca de categorias de QoS, e respectivos módulos *Configurador* e *Monitor*. Esta biblioteca seria populada gradativamente, à medida que novos aspectos de QoS se fizessem necessários. Contratos de QoS seriam implantados através do reuso das categorias de QoS contidas na biblioteca.

6.3 Trabalhos correlatos

Durante a formulação de nossa proposta para QoS, foram avaliados alguns trabalhos correlatos. O enfoque destes trabalhos varia no nível de abstração em que QoS é tratado, nos mecanismos oferecidos para a especificação e programação de QoS, e nas características de QoS suportadas. Enquanto algumas abordagens oferecem APIs para acessar diretamente os recursos dos sistemas de suporte, outras permitem a descrição de categorias de QoS em níveis mais altos de abstração.

Na abordagem usada em [193], a programação de QoS está disponível no nível de programação tradicional. *QoS-sockets*, uma biblioteca que implementa um mecanismo de *sockets* com QoS, permite a programação de aspectos de QoS ligados à comunicação, através de parâmetros especiais, passados na criação dos *sockets*. A biblioteca QoS-sockets é restrita a aspectos de QoS de comunicação e é direcionada para programadores especializados. Além disso, o código relativo à programação de QoS é entrelaçado com o código da aplicação. Assim sendo, a separação de interesses não é observada, resultando em pouca possibilidade de reuso, o que torna difícil a evolução dinâmica.

BeeHive [194] oferece um conjunto de APIs de propósito específico, através do qual objetos podem requisitar a imposição de características de QoS para um gerente de recursos. Cada API permite que um objeto cliente solicite a imposição de uma propriedade de QoS em termos da aplicação como, por exemplo, o tempo médio entre

falhas. O gerente de recursos traduz cada requisição da API em requisições mais específicas, de reserva de recursos e de uso dos mecanismos dos sistemas de suporte. Estão sendo desenvolvidas, atualmente, APIs para características de tempo-real, tolerância a falhas e requisitos de segurança.

O suporte de *middlewares* reflexivos vem sendo empregado para permitir a adaptação de aplicações a novos requisitos de operação. A adaptação de uma aplicação é possível porque este tipo de *middleware* permite, por sua vez, a adaptação de seu funcionamento através de mecanismos de reflexão (veja discussão na seção 9.3.3).

O projeto Adapt [195], da Universidade de Lancaster, propõe a adaptação de *middleware*, por meio de ligações abertas (*open bindings*), como forma de oferecer QoS para aplicações multimídia. O gerenciamento de QoS é feito manipulando-se diretamente grafos de objetos, que representam os caminhos de comunicação fim-a-fim da aplicação e implementam características como a compressão, controle e sincronização de fluxos de mídia. Adapt não considera, no entanto, QoS para domínios diferentes dos ligados à multimídia. Além disso, o mecanismo de ligações abertas, isoladamente, não oferece políticas para gerenciamento de QoS. O conceito de ligações abertas é equivalente ao de conectores no *framework* R-RIO, sendo que estes não se restringem ao suporte de QoS. A composição e reconfiguração de conectores é análoga à manipulação de grafos de objetos em Adapt. O modelo de configuração de R-RIO, adicionalmente, facilita a implantação de políticas para a adaptação das aplicações em reação a mudanças de QoS.

O projeto TAO ("The ACE ORB") da Universidade de Washington em St. Louis [169], desenvolveu um ORB tempo-real, compatível com CORBA, que objetiva otimizar invocações de métodos tempo-real dentro do próprio ORB. Conforme visto na seção 3.3.1, TAO foi utilizado como uma das bases para a especificação de CORBA Tempo-Real (RT-CORBA) [125]. TAO permite a adaptação de algumas características do próprio ORB, como a política de associação de objetos a *threads*, a política de escalonamento de *threads*, o controle da prioridade de execução e alguns aspectos do IIOP (*Internet Inter-Orb Protocol*). Entretanto, a maior parte destas características não pode ser especificada na descrição das interfaces da aplicação em OMG-IDL. A programação de tais características deve ser feita diretamente no código dos objetos, utilizando-se uma API especial (ver discussão na seção 3.3.1).

Uma outra proposta, que utiliza a adaptação de *middleware* como meio de impor QoS, é discutida em [196]. Nesta proposta é apresentada uma arquitetura de QoS, Quartz. Um protótipo de Quartz foi implementado sobre uma infra-estrutura CORBA. Em Quartz, requisitos de QoS devem ser descritos na aplicação, codificando-se, explicitamente, chamadas a uma API. Nestas chamadas são passados parâmetros contendo informações sobre os requisitos de QoS a serem impostos. A partir de uma tradução destes parâmetros, componentes da arquitetura de QoS são selecionados e compostos para impor a qualidade desejada.

Alguns projetos contemporâneos procuram abordar QoS com uma solução integral, oferecendo *frameworks* que geralmente integram (i) uma forma de descrever QoS no nível da aplicação (através de uma linguagem ou anotações em uma linguagem), e (ii) um *middleware* com suporte a QoS.

Em [185], Svend Frolund apresenta uma linguagem para descrição de QoS, *QoS Modeling Language* (QML). QML foi concebida com o objetivo de ser independente de outras linguagens de programação. QML, possibilita a descrição de contratos e perfis de QoS. Em [197] apresenta-se uma solução para integrar QML e OMG-IDL no contexto da programação, e uma extensão para UML, a fim de se descrever QoS no contexto de projeto. QML possui uma especificação semântica formalizada [198], o que facilita a verificação das propriedades de QoS descritas. Existe também um mecanismo direcionado à verificação da coerência estática de contratos de QoS. Alguns testes de compilação e de geração de código para descrições de características de tolerância a falhas foram feitos pelo autor, para validar a proposta. Entretanto, está fora do escopo de QML oferecer uma infra-estrutura padronizada para a imposição de QoS. Em nossa proposta, QML foi adaptada para o contexto de arquiteturas de *software* e ADLs. Adicionalmente, propusemos uma estrutura que pode ser usada de forma recorrente, para impor e gerenciar as características de QoS descritas.

O projeto QuO (*Quality Objects*) [187] é um *framework* destinado à implantação de aplicações distribuídas com requisitos de QoS. Seu objetivo é oferecer a habilidade de especificar, monitorar e controlar aspectos de QoS em uma aplicação. QuO utiliza um ambiente de execução baseado em CORBA. Interações entre clientes e objetos utilizam o conceito de *conexão*, um encapsulamento que inclui os padrões de uso desejados e requisitos de QoS, especificados na forma de um contrato. Um dos

componentes principais do *framework* QuO é QDL (*Quality Description Language*), uma linguagem para especificar estados possíveis de QoS, recursos de sistema, mecanismos para medir e prover QoS, e mecanismos de adaptação de QoS. QDL é composta de três sub-linguagens, que se concentram em propriedades distintas: (i) CDL, uma linguagem de descrição de contratos entre clientes e objetos em termos de uso de QoS; (ii) SDL, uma linguagem para a descrição da estrutura interna da implementação de objetos e a quantidade de recursos que eles requerem; e (iii) RDL, uma linguagem para descrever os recursos disponíveis e seus *status*.

Atualmente está disponível um protótipo de um compilador para CDL, através do qual podem ser descritos contratos de replicação de objetos. Observamos que QDL permite a descrição de características de QoS através de uma linguagem de aspectos, separada da descrição funcional da aplicação e, assim, também inspirou nossa proposta. Entretanto, em QDL é necessário que se descrevam recursos de suporte, e o uso destes recursos por objetos da aplicação. Estas informações apresentam detalhes relevantes para o suporte a QoS, próximos a uma implementação, em um nível diferente de uma arquitetura de *software*. Além disso, o mapeamento das descrições de QDL em artefatos de *software*, proposto pelos autores, está intimamente ligado a CORBA, não sendo trivial a sua generalização.

O grupo Monet, da Universidade de Illinois em Urbana-Champaign, propõe um *framework* para QoS, chamado $2K^{Q}$ [199]. Este *framework* utiliza a infra-estrutura de um *middleware* baseado em componentes, o 2K [97]. $2K^{Q}$ trata QoS particionando o processo de iniciação fim-a-fim de aspectos de QoS em duas fases: (i) compilação de QoS distribuído e (ii) instanciação de QoS em tempo de execução. A primeira fase é executada off-line, a partir de uma especificação de QoS feita pelo programador, para determinada aplicação. Na segunda fase, os serviços fim-a-fim e serviços de *middleware* com suporte a QoS são dinamicamente configurados e iniciados, de acordo com a especificação de QoS. Nesta segunda fase, também são utilizados protocolos dinâmicos de descobrimento e configuração para sincronizar serviços e recursos. Uma aplicação de vídeo-sob-demanda (VOD) e um compilador específico para descrições de parâmetros de transmissão de quadros foram desenvolvidos para validar o *framework*. A integração com o 2K e a habilidade de reconfiguração ainda devem ser desenvolvidas. A compilação de QoS em $2K^{Q}$ é utilizada a fim de gerar código executável para a gerência de QoS. Tal compilador deve ser programado para reconhecer as

especificidades das características de QoS que estão sendo tratadas, assim como os recursos existentes em cada sistema. A técnica de compilação de QoS poderia ser utilizada para a geração dos módulos *Configurador* e *Monitor de QoS* de nossa proposta. Entretanto, existe a necessidade de um compilador específico para cada categoria de QoS sendo descrita na aplicação, o que talvez não seja muito prático em certos casos. Em nossa abordagem, o código para atender a novas características de QoS pode ser inserido nos módulos *Configurador* e *Monitor de QoS* sem a dependência de uma ferramenta.

De uma maneira geral, os trabalhos estudados oferecem alguma forma para se descrever e gerenciar características de QoS. A abordagem destes trabalhos difere no nível do sistema em que QoS é considerado, como já foi destacado, e nos domínios de características que podem ser utilizados. Em nossa proposta, aspectos de QoS em arquiteturas de *software* podem ser descritos e gerenciados dinamicamente, sem restrições ao domínio das características. A estrutura proposta facilita a programação do código específico para o gerenciamento de QoS. Da mesma forma, esta estrutura pode ser utilizada como arcabouço para integrar abordagens orientadas a características de QoS específicas, encapsulando-se, nos módulos de gerência de QoS, o acesso aos seus mecanismos.

6.4 Conclusão

Neste capítulo propôs-se a inclusão de aspectos de QoS no nível de arquitetura de *software*. A abordagem utilizada inclui uma forma para a descrição de contratos de QoS em ADLs, como CBabel, e a configuração de uma estrutura de suporte para implantar os contratos descritos, em tempo de execução. Esta abordagem adere ao modelo de componentes de nossa proposta, e permite o atendimento de requisitos operacionais das aplicações, mantendo a separação de interesses.

Para a utilização da referida abordagem em R-RIO, poucas mudanças são necessárias. Por exemplo, em CBabel, é necessária a introdução da sintaxe para a descrição de categorias e contratos de QoS. Este procedimento altera minimamente a linguagem, visto que a especificação do relacionamento de categorias de QoS e módulos, através de perfis de QoS, é feita separadamente (veja comentário na seção 6.2.2). No Gerente de Configuração (seção 7.2, capítulo 7), alguns acréscimos são

necessários para que este instancie e acione os módulos *Configurador* e *Monitor de QoS*, quando um contrato estiver associado a um conector que esteja sendo instanciado. Após estas mudanças, R-RIO pode suportar, potencialmente, contratos de QoS de qualquer domínio. Pode-se considerar a introdução de QoS em R-RIO como um exemplo de utilização dos conceitos do próprio *framework*.

A aplicabilidade da abordagem de QoS proposta, em R-RIO, será validada através de uma aplicação concreta, com requisitos de QoS, apresentada no capítulo 8. No próximo capítulo são discutidas questões de ordem prática de nossa tese e alguns detalhes do protótipo desenvolvido para o *framework* R-RIO, para o qual foram concebidos os exemplos do capítulo 8.

Capítulo VII

Implementação

7.1 Introdução

À medida que os fundamentos do *framework* R-RIO foram sendo consolidados, também se avaliaram aspectos práticos e alternativas para a implementação de seus elementos constituintes. Esta fase de nossa pesquisa teve o objetivo de verificar se os conceitos do *framework* poderiam ser colocados em prática. Neste contexto, um protótipo para o serviço de gerência de configuração (capítulo 4) foi implementado, permitindo que os exemplos discutidos nesta tese fossem efetivamente desenvolvidos.

Na primeira parte deste capítulo, apresenta-se a estrutura do protótipo desenvolvido para serviço de gerência de configuração de R-RIO [142]. Em seguida, são discutidos outros aspectos práticos, como a construção de conectores genéricos, e o mapeamento de módulos e conectores, descritos no nível de arquitetura, para artefatos de *software*, no nível de implementação. Em seguida, apresenta-se uma proposta para as APIs de configuração e reflexão arquitetural, que oferecem acesso aos serviços de reflexão do *framework*, parcialmente implementadas no protótipo.

7.2 O *Middleware* para o serviço de Gerência de Configuração

O protótipo para o serviço de gerência de configuração de R-RIO é um *middleware* composto por um conjunto de elementos de *software*, que são distribuídos por nós de processamento onde módulos e conectores são instanciados e gerenciados. A partir de arquiteturas de *software* descritas em CBabel, este *middleware* pode receber comandos ou *scripts* de configuração para criar imagens executáveis destas arquiteturas ou reconfigurá-las. Java foi utilizada como linguagem de programação e base para a implementação dos serviços de gerência e comunicação. A escolha de Java levou em

consideração as características da linguagem (orientada a objetos, métodos sincronizados, primitivas *wait*/*notify*, etc.) e as características do ambiente de execução (JVM, interpretação de *ByteCodes*, reflexão estrutural, serialização de objetos, RMI, *sockets*, etc.), que facilitariam a construção do protótipo.

A figura 7.1 apresenta um diagrama dos elementos do protótipo. Em cada nó de processamento é criada uma instância do suporte R-RIO, onde é executado o serviço de **gerente de configuração local**, que recebe solicitações para a execução de comandos de configuração. A cada pedido recebido e executado, o repositório de informações de meta-nível sobre a arquitetura é atualizado. A execução efetiva dos comandos é delegada, em cada nó, para um **configurador executivo**. No caso do protótipo, para realizar sua tarefa, o configurador executivo herda as características do *ClassLoader* da JVM e implementa um carregador de classes adaptado [200] para o suporte à configuração.

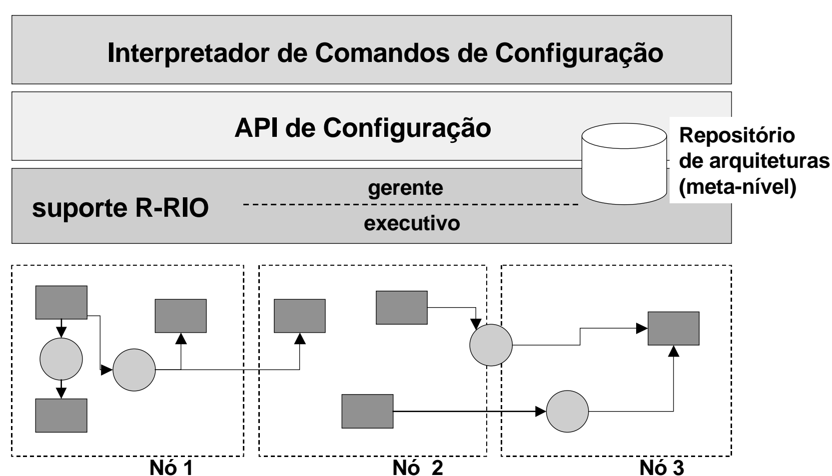


Figura 7.1 - Arquitetura do *middleware* de suporte R-RIO

O suporte R-RIO ainda conta com um elemento chamado **gerente de configuração global**. Este elemento tem funções semelhantes ao gerente de configuração local e, adicionalmente, é responsável por:

- entrar em contato com os gerentes de configuração local dos nós onde vão ser processados os comandos;
- manter atualizadas, de forma global, as informações de meta-nível sobre as arquiteturas, pré-processadas a partir de uma descrição em CBabel;

- executar os comandos de ligação envolvendo conectores com implementação distribuída (uma parte cliente e uma parte servidora - em nós diferentes).

Observa-se que, no protótipo desenvolvido, o Gerente de Configuração tem uma implementação centralizada (existe apenas uma instância do gerente de configuração global), e o executivo é replicado em cada nó incluído no ambiente (através do gerente de configuração local e do configurador executivo). Contudo, o serviço de gerência de configuração poderia ter uma implementação distribuída, utilizando-se gerentes de configuração global separados, para controlar partes diferentes de uma arquitetura em execução.

Um **interpretador de comandos** também é provido. Este interpretador traduz comandos de configuração e invoca a API de configuração. A API de configuração pode ser utilizada diretamente, mas o interpretador de comandos oferece um serviço a ser empregado remotamente, por qualquer aplicação ou interface. Desta forma versões diferentes de interfaces para gerenciar as arquiteturas de *software* podem ser executadas, desde interfaces textuais, executando no mesmo nó que o interpretador de comandos, até interfaces gráficas, executando como *applets* Java em um navegador.

7.3 Mapeamento

O mapeamento de módulos, portas e conectores para uma implementação depende do ambiente particular em uso (capítulo 5). Em nosso protótipo, módulos primitivos são definidos por classes Java, e módulos compostos podem ser construídos a partir de configurações arbitrárias de módulos primitivos. Também é possível a composição de módulos, utilizando-se a herança de Java, mas isto implica na perda da capacidade de reconfiguração dos módulos componentes individuais.

Portas são associadas a declarações de métodos Java (às assinaturas dos métodos) no nível de configuração, e a invocações de métodos no nível de código. Observa-se que apenas os métodos explicitamente associados a portas são diretamente visíveis no nível de configuração. Os outros métodos são usados apenas através dos mecanismos de ligação e referência de Java.

Classes de módulos e conectores (mapeados em classes Java) são associadas a instâncias de módulos e conectores através da CBabel. Em tempo de configuração e

carga, estas instâncias são criadas como objetos Java. Classes de conectores são tratadas de forma especial pelo serviço de gerenciamento de configuração. Por exemplo, se um conector possui uma implementação distribuída, o serviço de gerenciamento de configuração precisa carregar as partes deste conector nos nós apropriados. Tal tratamento não é necessário para os módulos.

Alguns detalhes sobre mapeamento de componentes para uma implementação, no contexto do protótipo, são apresentados nas próximas subseções.

7.3.1 Conectores Reflexivos por Contexto

A implementação de um conector segue, em linhas gerais, uma estrutura padrão, que pode ser programada seguindo-se uma receita. Basicamente, no lado do módulo que inicia uma interação (uma invocação de método), cada invocação de método é serializada (no protótipo foi usado o mecanismo de serialização nativo de Java) e, na seqüência, no lado do módulo destinatário, é realizada uma invocação dinâmica para o método em questão. Se o conector encapsula um mecanismo de comunicação, os dados serializados são transmitidos pela rede. A figura 7.2 apresenta o caso de uma arquitetura simples, que utiliza RMI para comunicação, no momento da instanciação do conector.

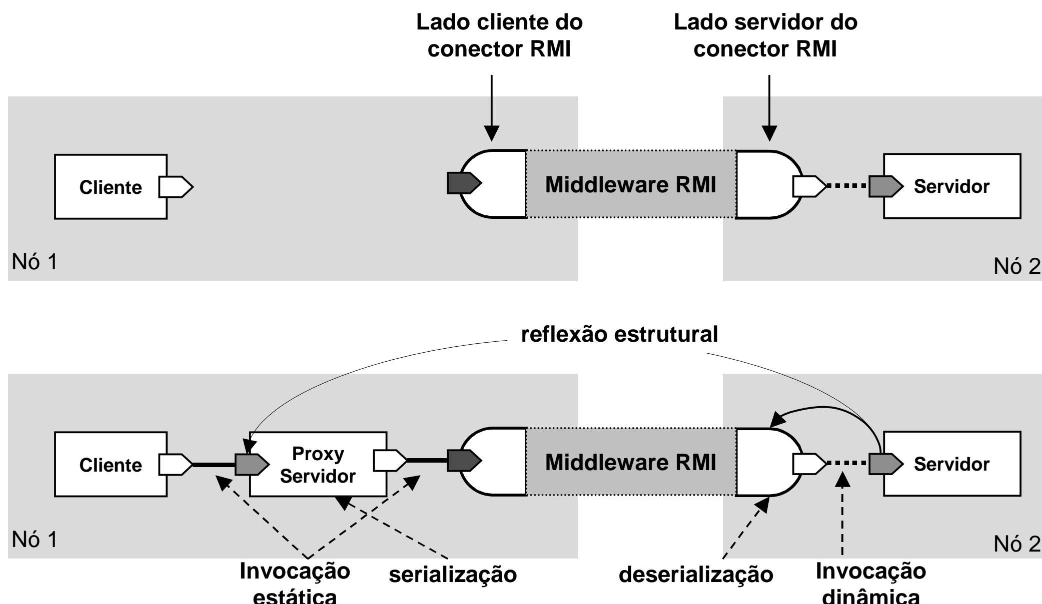


Figura 7.2 - Funcionamento do conector reflexivo por contexto

Durante a carga de uma arquitetura, após a instanciação dos módulos, os conectores especificados são instanciados para interligá-los. Inicialmente, são criados os

objetos das classes que fazem parte da implementação do conector, os quais encapsulam os detalhes de RMI (primeira parte da figura 7.2). Como se trata de um conector distribuído, existem duas partes: uma, instanciada no mesmo nó do objeto cliente, encapsula o *stub* do cliente; a outra, criada no mesmo nó do objeto servidor, encapsula o *stub* do servidor. Em seguida, a assinatura da interface do objeto servidor é obtida por reflexão arquitetural, consultando-se as informações de meta-nível armazenadas no gerente de configuração global. Estas informações são, então, utilizadas para a adaptação de um objeto *proxy*, que representa o objeto servidor perante o objeto cliente, e para a adaptação da parte servidora do conector, que deverá montar invocações dinâmicas para o objeto servidor.

O módulo cliente realiza, normalmente, uma invocação ao servidor (no protótipo, o que ocorre é uma invocação a um método). Esta invocação é, na realidade, direcionada ao *proxy* do servidor, que tem a mesma interface do objeto servidor original. O *proxy* cuida de serializar as informações necessárias, para que a invocação ao objeto servidor seja realizada no nó remoto. Em seguida, o *proxy* encaminha estas informações à parte cliente do conector através de uma interface padronizada. As informações da invocação são transmitidas à parte servidora do conector.

No nó onde se encontra o objeto servidor, a parte servidora do conector recupera as informações transmitidas e monta dinamicamente uma invocação para o objeto servidor, fazendo o papel do objeto cliente. A invocação não pode ser feita diretamente, porque não se conhece a interface do objeto servidor até que as informações da invocação cheguem à parte servidora do conector. Além disso, este mesmo conector pode estar ligando outros pares de módulos com interfaces diferentes e, assim, a invocação não pode ser estática. Esta é uma das adaptações, ao contexto em operação, realizadas no conector. O retorno de resultados e as exceções são transferidas para o cliente de forma semelhante.

Os passos bem definidos para a concepção, facilitam a implementação de novos conectores. Além disso, por terem uma interface padronizada, os conectores são intercambiáveis. Fica ao encargo do *proxy,* que também utiliza a reflexão arquitetural, adaptar um novo conector ao contexto em operação. Por exemplo, na situação da figura 7.2 seria possível substituir o conector RMI por um conector CORBA, por questões de interoperabilidade, ou por *sockets*, para melhorar o desempenho.

Uma outra característica que surge, em conseqüência da utilização do mecanismo apresentado, é a possibilidade de se substituírem dinamicamente os objetos que estão interagindo. No exemplo anterior, o cliente poderia desfazer a ligação com o servidor atual e refazê-la com um segundo servidor, durante a operação. As referências ao novo par de módulos, a ser interligado pelo conector, seriam obtidas ao se executar o comando de ligação, e a adaptação do conector ao novo contexto seria realizada conforme discutido nos parágrafos anteriores.

Para o protótipo de R-RIO, foram desenvolvidos conectores genéricos, encapsulando aspectos de: (i) comunicação (*socket* UDP / TCP, multicast, RMI e JDK-CORBA), (ii) *log* de interações, (iii) canal de eventos (*Observer*) e (iv) coordenação (sincronização e concorrência). Estes conectores podem ser compostos, combinando-se suas características. Algumas composições de conectores foram testadas. Por exemplo, na aplicação produtor-consumidor utilizou-se uma combinação de conectores encapsulando *sockets* e aspectos de coordenação. Uma outra combinação, com conectores encapsulando um canal de eventos e RMI, também foi testada (seção 8.4.2 - capítulo 8). Estes testes foram realizados codificando-se explicitamente o relacionamento entre os conectores, pois o gerente de configuração implementado não oferecia suporte para composição de conectores. Mesmo nestes termos, a adaptação das aplicações a novas condições de operação, mostrou-se simples.

7.3.2 Mapeamento de módulos

Em nosso protótipo, o gerente de configuração controla a carga dos módulos, redirecionando as funções do carregador original da JVM. Para o controle da operação de um módulo, o Gerente de Configuração necessita que o mesmo tenha características de objeto ativo. Para isso, neste protótipo, os módulos precisam herdar da classe *Thread* de Java. O método *run*, utilizado (tornando o objeto ativo) ou não, também deve ser declarado. Em outros ambientes de execução é possível utilizar a API de serviços de carga de módulos executáveis. Por exemplo, em um ambiente Unix, poderiam ser empregadas as funções *dlopen()* e *dlsym()* com este objetivo [82].

A listagem 7.1 apresenta um exemplo resumido do código de um módulo que pode interagir com dois outros módulos. No caso do protótipo em Java, o mapeamento das declarações das portas de saída de um módulo em CBabel é feito indicando-se uma

referência para os objetos com os quais esta classe irá interagir (linhas 3 e 4). Isto permite que um objeto interaja com outro sem criar uma instância do mesmo (através do método *new()*, por exemplo), uma vez que esta estrutura só será conhecida durante a configuração. Variáveis declaradas dentro dos módulos, em CBabel, são mapeadas para Java diretamente (linha 6), junto com um método de acesso a estas variáveis (linha 8 - versão simplificada).

```
1  public class nomeDaClasseDoModulo extends Thread {
2
3    public classeModuloA m1;
4    public classeModuloB m2;
5
6    private tipo1 Var1;
7
8    public synchronized tipo1 getState () {
9       return Var1;
10   }
11              . . .
12   public void run() {
13
14   }
15 }
```

Listagem 7.1 - Estilo a ser seguido para a programação do módulo

Seguindo este roteiro para o mapeamento, de forma automática ou manual, uma classe em Java está preparada para ser manipulada pelo suporte à configuração do protótipo. Outros tipos de construção são permitidos, tais como o uso de herança de Java. O programador deve estar ciente de que, com estas construções, a granularidade dos elementos configuráveis diminui.

Suporte

No protótipo de R-RIO, procurou-se manter o maior nível de flexibilidade e transparência possível. Evitou-se introduzir novos conceitos à linguagem Java, e buscou-se implementar os conceitos de R-RIO de maneira fiel. Por exemplo, um objeto pode ser manipulado pelo suporte R-RIO apenas herdando da classe *Thread*, padronizada em Java. A própria implementação do suporte R-RIO utiliza apenas recursos de Java, como a carga de classes, de forma controlada (R-RIO tem a sua versão do *ClassLoader*, com funções adicionais). Isto torna o protótipo portável para qualquer sistema que inclua uma JVM.

A estratégia adotada para a implementação, entretanto, limita o controle que o suporte R-RIO possui sobre as arquiteturas em execução. Apenas os métodos

disponíveis na classe *Thread* de Java podem ser usados para implementar os comandos de configuração. Além disso, estes métodos não são totalmente seguros, e alguns deles correm o risco de não fazer mais parte da classe *Thread* em versões futuras de Java, exatamente por estes problemas de segurança (segundo a própria documentação de Java [201]). Uma alternativa seria a de desenvolver-se uma classe *rrioModule*, com o comportamento básico de um módulo, segundo o modelo de componentes de R-RIO, a qual toda implementação de módulo herdaria.

Outros trabalhos correlatos adotam este recurso e impõem que os seus componentes utilizem uma estrutura básica, normalmente chamada de *container* ou *object-factory*, a partir da qual os mesmos são programados. Por exemplo, JavaBeans, precisa que os componentes herdem da classe *Beans*. Em 2K, é necessário que os componentes implementem a interface *ComponentConfigurator* para que se tornem configuráveis.

Na próxima seção, o mapeamento de módulos e conectores discutido é ilustrado para o caso da aplicação produtores-consumidores com *buffer*.

7.3.3 Implementação dos módulos do exemplo produtor-consumidor com *buffer*

O código de uma classe *produtor* (listagem 7.2), mapeado a partir da descrição em CBabel, é programado para produzir uma seqüência de números inteiros, que são depositados no *buffer*.

```
1  public class produtor extends Thread {
2
3    public buffer b; // declaração da porta de saída
4    public void run () {
5    int i;
6    try {
7         for (i=1;i<=1000;i++) {
8           if (b.put(i))
9              System.out.println ("Produtor: >>>>> Put buffer item " + i);
10          else
11             System.out.println ("Erro!");
12        }
13        System.out.println ("Fim.");
14     }
15     catch (java.lang.InterruptedException e) {} ;
16   }
17 }
```

Listagem 7.2 - Código do produtor

Os códigos do consumidor (listagem 7.3) e do *buffer* (listagem 7.4) têm estrutura semelhante.

```
1  public class consumidor extends Thread {
2
3   public buffer b;
4   public void run () {
5
6     int i, item;
7
8     try {
9       for (i=1;i<=1000;i++) {
10        item = b.get();
11        if (item != -1)
12         System.out.println ("Consumidor: >>>>> Get  buffer item " + item);
13        else
14           System.out.println ("Erro!");
15      }
16      System.out.println ("Fim.");
17    }
18    catch (java.lang.InterruptedException e) {} ;
19  }
20 }
```

Listagem 7.3 - Código do consumidor

No código do *buffer* pode ser identificada a declaração da variável *n_itens*, declarada na descrição do módulo em CBabel, como privativa da classe em Java, e o seu método de acesso *getState ()*, que executará em exclusão mútua quando invocado (pelo modificador *synchronized*).

```
1 public class buffer extends Thread {
2   private int n_itens = 0;                 // total de elementos no buffer
3   public synchonized int getState ()       // retorna total de elementos no buffer
4                { return n_itens; }
5   public void run () {}
6   public int Get () { ... }                // retira elementos
7   public int Put(int i) { ... }            // põe elementos
8 }
```

Listagem 7.4 - Código do buffer

Observa-se que os códigos dos módulos não incluem outros aspectos, como a distribuição ou coordenação, além dos básicos da aplicação: produzir, consumir e armazenar. Esta característica permite que se tenha dimensão das vantagens da separação de interesses. Comparado com o exemplo apresentado no início do capítulo 1, o código está limpo. O programa dos módulos é de simples entendimento e de fácil manutenção. Em tempo de configuração, outros aspectos podem ser adicionados, tais como a localização física dos módulos ou a seleção da classe de conector utilizada para implementar a interação entre eles.

7.3.4 Mapeamento de contratos de coordenação

O mapeamento de um contrato de coordenação para um programa também segue

uma receita. Para o protótipo em Java, o mapeamento utiliza as características da própria linguagem, que permitem adicionar a uma classe o comportamento de um monitor. A descrição do contrato de coordenação para o produtor-consumidor é reproduzida na listagem 7.5(a), e na listagem 7.5(b) o código em Java é apresentado.

```
connector {

  condition vazio = true,
            cheio = false;

  staterequired int n_itens;
  int MAX_ITENS = 10;

  exclusive {

    out port PutT {
      guard  (vazio) {
        after {
          cheio = true;
          if (n_itens == MAX_ITENS) {
            vazio = false;
          }
        }
      }
    } GPut;

    out port GetT {
      guard  (cheio) {
        after {
          vazio = true;
          if (n_itens == 0) {
            cheio = false;
          }
        }
      }
    } GGet;
  }

  in port PutT;
  in port GetT;

} SincMutexPCcon;
```

```
class SincMutexPCcon {

 static buffer buf; // módulo buffer
 static final MAX_ITENS = 10;
 boolean vazio = true, cheio = false;

 public synchronized int PutT(int i) {
    try {
      // guard
      while (vazio == false) wait();
      // outport PutT
      int result = buf.put(i);
      // after
      cheio = true;
      if (buf.getState() == MAX_ITENS)
          vazio = false;
      notifyAll();
      // return in port PutT
      return (result);
    }
    catch (InterruptedException e) {
      return -1;
    }
 }

 public synchronized int GetT() {
    try {
      // guard
      while (cheio == false) wait();
      // out port GetT
      int result = buf.get();
      // after
      vazio = true;
      if (buf.getState() == 0)
          cheio = false;
      notifyAll();
      // return in port GetT
      return (result);
    }
    catch (InterruptedException e) {
      return -1;
    }
  }
}
```

Listagem 7.5 - (a) conector em CBabel e (b) seu mapeamento para Java

A partir da descrição do conector *SincMutexPCcon* associado a um contrato de coordenação, é gerada uma classe *SincMutexPCcon* em Java. As variáveis de condição são mapeadas para *booleans* em Java e serão usadas em conjunto com as primitivas *wait* e *notifyAll* (sublinhadas no código), para reproduzir semântica semelhante à dos guardas de CBabel. A exclusão mútua das portas de saída é garantida pelo modificador *synchronized*. As portas de entrada são mapeadas nos métodos *PutT* e *GetT*, e as portas

de saída nas invocações aos métodos de *buffer*, que ocorrem dentro destes métodos. Portanto, todo código entre a porta de entrada e a porta de saída do conector é executado em exclusão mútua. O acesso ao estado do *buffer* é feito invocando-se o método *getState()*, anteriormente programado no módulo *buffer*. No nível da arquitetura, este método é representado por uma porta de entrada "especial", especificada pela declaração de uma variável *n_itens*, na descrição do módulo, em CBabel.

O mapeamento dos contratos de coordenação de CBabel para Java utiliza os recursos de sincronização e concorrência disponíveis na linguagem e inclui, por outro lado, as adaptações necessárias para que a semântica dos guardas no *framework* R-RIO seja respeitada. Por exemplo, Java não possui variáveis de condição nas quais processos podem se bloquear. Por isso um *loop*, com um teste (*while*), é utilizado em conjunto com a primitiva *wait*. Da mesma forma, usa-se a primitiva *notifyAll*, que libera todos os processos bloqueados no objeto, indistintamente; daí a necessidade do *loop* com o teste. No mapeamento para outras linguagens, outras soluções podem ser adotadas, tais como o uso de bibliotecas como *pthreads* (padrão POSIX), ou mesmo o mecanismo de semáforos, disponíveis em vários sistemas.

7.4 API de Configuração

A interface de programação de configuração permite que as aplicações acessem diretamente os serviços do suporte de R-RIO, os quais implementam os comandos de configuração, segundo o modelo descrito na seção 4.3. Através desta API as arquiteturas descritas em CBabel são efetivamente carregadas para executar. As rotinas de reconfiguração de uma aplicação também podem ser programadas, utilizando esta API, em conjunto com a API de reflexão arquitetural. Apresentamos o conjunto das primitivas disponíveis no protótipo. Detalhes de implementação são discutidos em [142].

Os métodos da API são executados em exclusão mútua e retornam uma cadeia de caracteres, contendo o resultado da operação. Os argumentos que contêm informações dos nomes de módulos, classes e *hosts* também são do tipo *String* (cadeia de caracteres).

```
1  public synchronized String
2    instantiate
3         (String className, String moduleName, Object[] args, String host);

4    link
5         (String moduleName1, Vector ports1, String host1,
6          String moduleName2, Vector ports2, String host2,
7          String className, String connectorName);

8    start, block, resume, remove
9         (String moduleName, String host);
```

Listagem 7.6 - Métodos da API de configuração

Para instanciar um módulo, o método *instantiate* deve ser invocado. O argumento *moduleName* indica o nome do módulo, no nível da arquitetura, e o argumento *className*, a classe correspondente em Java. O argumento *args* contém uma lista com pares tipo-valor, com os parâmetros de inicialização do módulo, se existirem. O argumento *hosts* indica o nó onde o módulo deve ser instanciado. De uma forma geral, os outros métodos seguem esta padronização.

O método para ligação de componentes, *link*, pode ser utilizado em qualquer estilo de ligação, conforme discutido no capítulo 4. Para isso, sua assinatura possui alguns argumentos que podem ser usados, ou não, dependendo da configuração que se deseja realizar. No caso mais completo, é possível ligar-se um subconjunto de portas de um módulo a um subconjunto diferente de portas de outro módulo, através de um conector. Neste caso, os primeiros três argumentos (listagem 7.6, linha 5) são, respectivamente, o nome do primeiro módulo, as portas de saída que serão ligadas e o nó onde se encontra este módulo. Os próximos três argumentos são relacionadas com o módulo destinatário (o vetor *ports2* contém a lista das portas de entrada que serão ligadas). Em seguida, os argumentos representam o nome da classe Java do conector a ser usado, e o seu nome na arquitetura. Em casos onde não há ambigüidade, os argumentos do conector ou as informações sobre as portas que estão sendo ligadas podem ser omitidos.

Os métodos *start*, *block*, *resume* e *remove* necessitam apenas dos argumentos do módulo e a localização deste para serem executados.

O uso da API de configuração (e de reflexão arquitetural, seção 7.8) não requer referências aos componentes e recursos reais sendo manipulados, bastando os nomes utilizados no nível da arquitetura. A conversão para referências concretas é feita pelo suporte R-RIO. No caso do protótipo em Java, emprega-se a reflexão estrutural da

própria linguagem para obter as referências a objetos dinamicamente, a partir das informações da arquitetura.

O acesso da API é feito invocando-se os métodos no gerente de configuração global. Para isso, quando um módulo deseja usar a API de configuração, uma instância da classe *proxyGC* deve ser criada. O *proxyCG* implementa o interpretador de comandos, focalizado na seção 7.2, e redireciona as invocações dos métodos da API para o configurador global. Assim sendo, é possível iniciar comandos de configuração remotamente.

Um exemplo do uso da API de configuração dentro de uma aplicação é apresentado na seção 8.5 - capitulo 8. Esta API também foi usada nas implementações de uma console de comando, a partir da qual comandos de configuração podem ser iniciados manualmente, e de uma interface gráfica experimental [202], para interagir com o suporte R-RIO.

7.5 API de Reflexão Arquitetural

A consulta ao repositório de meta-nível, que armazena as informações sobre a arquitetura das aplicações em execução, é feita através da API de reflexão arquitetural. A tabela 7.1 apresenta a proposta de um conjunto de métodos para esta API. Estes métodos podem ser utilizados a fim de se obterem informações sobre uma parte da arquitetura, como, por exemplo, os módulos de um nó ou a interface de determinado módulo. A API proposta tem o objetivo de tornar simples a consulta de informações de uma arquitetura e, ao mesmo tempo, oferecer as informações necessárias para se inferir sobre a mesma.

Observa-se, na tabela 7.1, que as consultas, de forma geral, retornam nomes (*String*) ou vetores (*Vector*) contendo uma lista de nomes. Em alguns métodos, como *getPortSignatures* e *getLinks*, o resultado apresenta uma lista de objetos contendo campos internos com as informações relevantes. Por exemplo, no caso do método *getLinks*, para cada ligação existente na arquitetura, corresponde um objeto, no vetor retornado, que contém o módulo origem, o módulo destino e o conector envolvido na ligação.

Tabela 7.1 - Métodos da API de reflexão arquitetural

public synchronized	
Vector **getArchitecture** (String architectureName, URL loc)	permite a movimentação das informações de meta-nível de uma arquitetura, obtidas a partir de uma dada localização.
Vector **getModules** (String architectureName)	retorna uma lista com o nome dos módulos de uma arquitetura
Vector **getConnectors** (String architectureName)	retorna uma lista com o nome dos conectores usados em uma arquitetura
Vector **getLinks** (String architecture)	retorna uma lista com objetos contendo o nome dos componentes (módulos e conectores) de cada ligação descrita em uma arquitetura
Vector **getHost** (String moduleName)	a partir do nome e localização do módulo, obtém-se uma lista de nomes do nós onde este está instanciado
String **getState** (String moduleName, String Host)	a partir do nome e localização do módulo, obtém-se o estado atual do mesmo
Vector **getPorts** (String moduleName, String Host)	a partir do nome e localização do módulo, obtém-se uma lista com o nome das portas
Vector **getPortSignature** (String moduleName, String Host, String portName)	a partir do nome e localização do módulo, e o nome de uma porta deste módulo, obtém-se a assinatura desta porta
Vector **getUserModules** (String moduleName, String Host)	a partir do nome e localização do módulo, obtém-se uma lista contendo o nome dos módulos dependentes deste (seus clientes)
Vector **getUsedModules** (String moduleName, String Host)	a partir do nome e localização do módulo, obtém-se uma lista contendo o nome dos módulos dos quais este é dependente (seus servidores)
Vector **getUserConnectors** (String connectorName)	a partir do nome do conector, obtém-se uma lista contendo o nome dos conectores dependentes deste
Vector **getUsedConnectors** (String connectorName)	a partir do nome do conector, obtém-se uma lista contendo o nome dos conectores dos quais este é dependente
String **getClassName** (String componentName)	a partir do nome do módulo ou conector, obtém-se o nome da classe em Java
Vector **getContract** (String contract Type, String connectorName)	a partir do nome de um conector e o tipo de um contrato, um vetor com objetos contendo as propriedades do contrato é retornado, se um contrato existir. O conteúdo do vetor é dependente do tipo de contrato

Os métodos *getUserX* e *getUsedX* apresentam informações sobre as dependências de cada módulo ou conector, ou seja, a relação dos componentes usados e a relação daqueles servidos por estes. Estas informações são importantes para a programação de reconfigurações. Por exemplo, antes de se substituir um conector, pode-se bloquear todos os componentes dependentes do mesmo para evitar que novas interações sejam iniciadas durante esta operação. Entretanto, vale observar que uma reconfiguração, em um nível geral, é dependente da aplicação. Assim sendo, não se pode garantir a segurança de uma operação de reconfiguração apenas utilizando os métodos *getUserX* e *getUsedX* (ver discussão na seção 9.4.1, capítulo 9). Estes métodos também podem ser utilizados pelo próprio suporte R-RIO. Assim, em um conector composto por vários elementos encadeados, cada conector pode obter a informação do seu sucessor utilizando o método *getUsedConnectors()*.

Em 2K [97] informações de meta-nível são encontradas, localmente, em cada componente. Cada componente deve, por sua vez, implementar a interface *ComponentConfigurator* com esta finalidade. Desse modo, o programador precisa implementar os métodos desta interface e iniciar seus valores, quando o componente é criado. A varredura de uma arquitetura complexa, com o fim de se obter informações sobre a estrutura da aplicação, impõe que se visite cada componente para coletar-se os dados necessários. Em nossa proposta, tais informações estão disponíveis de forma concentrada, minimizando o esforço da varredura. Aqui vale a observação que, para arquiteturas com um número grande de componentes, pode haver problema de escalabilidade. Por exemplo, em uma arquitetura contendo 10.000 módulos, os recursos necessários para armazenar um vetor contendo referências a estes módulos e o esforço para varrer o mesmo, também devem ser considerados. Em algumas aplicações deste tipo, módulos podem ser agrupados em módulos compostos, ou a referência a grupo de módulos pode ser utilizada, minimizando o número de referências a ser manipulada. Além disso, neste tipo de aplicação, deve ser verificado se as operações de reconfiguração podem ser realizadas sobre uma parte específica da arquitetura, sem prejuízo da consistência.

O método *getContract* retorna as informações a respeito de um contrato, de um tipo determinado, associado a um conector. Se um contrato deste tipo não estiver associado ao conector, o vetor de retorno é vazio. É necessário passar, como argumentos, o nome do conector e o tipo do contrato. O vetor devolvido como retorno

vai conter objetos com as propriedades do contrato (tanto os providos, como os requeridos). O número de objetos e o seu conteúdo é dependente do tipo de contrato (de interação, coordenação, distribuição ou QoS). Para um mesmo tipo de contrato, o conteúdo também pode ser variável, pois as propriedades podem ser usadas diferentemente nos vários contratos descritos em uma aplicação.

Um método especial, *getArchitecture*, permite a movimentação da informação de meta-nível de uma arquitetura. A transferência pode ser feita a partir de um repositório de meta-nível ou de um arquivo. Este método pode ser empregado para dividir a carga da manutenção do repositório de meta-nível ou para tolerância a falhas, por exemplo. É possível também utilizá-lo para carregar uma arquitetura recém compilada e iniciar sua execução. Na interface gráfica desenvolvida [202], por exemplo, o mesmo é utilizado para sincronizar inicialmente o estado da arquitetura e desenhar a topologia da aplicação. Posteriormente, os demais métodos são usados para atualizações periódicas.

7.6 Conclusão

A funcionalidade do protótipo de R-RIO mostrou-se abrangente e flexível, o bastante para permitir o desenvolvimento de exemplos usados na validação do *framework* proposto na tese (apresentados no capítulo 8).

Através das APIs de configuração e reflexão arquitetural, é possível a um módulo especializado (uma implementação do *Configurador de QoS*, por exemplo) alterar dinamicamente a arquitetura de uma aplicação. Este módulo pode obter as informações necessárias, através da API de reflexão arquitetural, para inferir a composição de uma arquitetura, visando a tomar decisões de reconfiguração. Além disso, para efetivamente iniciar as mudanças da arquitetura, o módulo especializado pode utilizar a API de configuração e solicitar a execução dos comandos de configuração necessários.

As lições aprendidas na implementação do protótipo permitiram o refinamento de vários conceitos de nossa proposta. O protótipo ainda possui funcionamento limitado em relação ao modelo de componentes e gerência de configuração do capítulo 4. Entre os pontos que devem ser implementados e otimizados, futuramente, estão a API de

reflexão arquitetural e a ligação direta entre portas de conectores e portas de módulos, permitindo a execução de arquiteturas mais complexas.

Protótipos para outras duas ferramentas, não incluídas neste capítulo, foram desenvolvidos: (i) uma interface gráfica (GUI), que permite ao projetista configurar arquiteturas de *software* de forma visual e interagir com o gerente de configuração para executar e reconfigurar a aplicação e, (ii) um compilador para CBabel que, integrado ao gerente de configuração e a GUI, compila e verifica as arquiteturas descritas para criar imagens da arquitetura para execução, "alimentar" o repositório de meta-nível e a GUI. Estas ferramentas são apresentadas em [202].

Como parte das investigações, durante o desenvolvimento do protótipo, também foram levantadas algumas informações de desempenho. O objetivo foi averiguar a sobrecarga que o mecanismo dos conectores genéricos representa na interação entre módulos. Os resultados podem ser vistos em [203]. Algumas questões sobre desempenho em R-RIO e conclusões sobre os resultados obtidos experimentalmente são discutidas na seção 9.3.4, capítulo 9.

Esta página foi intencionalmente deixada em branco

Capítulo VIII

Validação

8.1 Introdução

Ao longo dos capítulos anteriores, a aplicação dos produtores-consumidores, em versões diferentes, foi utilizada para exemplificar os conceitos apresentados. Esta aplicação contém os elementos necessários para testar nossa proposta, além de ser também utilizada em trabalhos correlatos (capítulo 3), o que facilita uma comparação direta entre os mesmos. Para complementar a validação de nossa proposta, apresentamos neste capítulo alguns exemplos de diferentes domínios de aplicação, utilizando os modelos e conceitos propostos no *framework* R-RIO. A arquitetura das aplicações é descrita em CBabel. Alguns cenários, onde as aplicações têm que ser reconfiguradas, são discutidos. Para cada exemplo, quando aplicável, é verificado se os requisitos de reusabilidade, separação de aspectos, modularidade e abrangência são contemplados.

8.2 Ceia dos filósofos

O problema da **ceia dos filósofos** possui requisitos de coordenação que podem ser descritos em CBabel com naturalidade. O problema assume que *N* filósofos estão sentados à uma mesa[9]. Nesta mesa estão postos *N* pratos e *N* garfos. Para comer, um filósofo deve pegar dois garfos (o da sua direita e o da sua esquerda). Sua rotina consiste em pensar durante um tempo finito, depois comer durante um tempo finito e, em seguida, dormir durante um tempo, também finito. O objetivo do problema é encontrar soluções para manter os *N* filósofos em sua rotina, garantindo a sincronização

---

[9] Na literatura, este problema é, geralmente, apresentado para 5 filósofos.

(um filósofo precisa de dois garfos para comer) e a exclusão mútua (dois filósofos não podem usar o mesmo garfo ao mesmo tempo) necessárias, sem que haja esfomeação.

Propomos como solução que os filósofos, os garfos e a mesa sejam descritos como módulos. A mesa e os garfos são recursos compartilhados pelos filósofos. O acesso aos métodos destes módulos deve ser feito sob exclusão mútua e serialmente para não haver inconsistências Assim, utilizamos duas classes de conectores para encapsular os aspectos de coordenação: uma, para acesso à mesa, e outra, para acesso aos garfos. A configuração dos módulos é apresentada na listagem 8.1.

```
1  port int ResrvT (int MakeReservation); // tipo de porta
2  port int LevT (int ReleaseReservation);

3  module filosofo { // classe de módulo filósofo
4    out port ResrvT  GetLeftFork;
5    out port LevT    ReleaseLeftFork;
6    out port ResrvT  GetRightFork;
7    out port LevT    ReleaseRightFork ;
8    out port ResrvT  GetTable;
9    out port LevT    LeaveTable;
10  }
11  module garfo { // classe de módulo garfo
12    in port ResrvT  Reserve;
13    in port LevT    Release;
14  }
15  module mesa { // classe de módulo mesa
16    int usedPlaces;
17    in port ResrvT  Reserve;
18    in port LevT    Release;
19  }
```

Listagem 8.1 - Módulos filósofo, garfo e mesa

Um módulo filósofo (linha 3) possui 4 portas de saída. Duas portas, *GetTable e LeaveTable*, são utilizadas para solicitar um lugar à mesa e liberar a mesma. Dois pares de portas são utilizados para pegar e liberar os garfos da direita e da esquerda, *Get/ReleaseRightFork* e *Get/ReleaseLeftFork*, respectivamente.

Um módulo garfo (linha 11) possui duas portas de entrada, que devem ser mutuamente exclusivas. Uma das portas, *Reserve*, recebe pedidos dos filósofos adjacentes para reservar o garfo. A outra porta, *Release*, recebe os pedidos de liberação do garfo.

A mesa (linha 15), por sua vez, possui duas portas, que do mesmo modo devem ser mutuamente exclusivas. A mesa tem capacidade para receber *N* filósofos por vez.

Para facilitar a descrição da arquitetura, os tipos de porta utilizados em quase todos os componentes foram declarados separadamente (linhas 1 e 2). Optou-se por

criar referências particulares a estas portas dentro dos módulos, para não haver ambigüidade na descrição das ligações entre os mesmos.

```
1  connector RequestTable (int n) {
2    int MAX_PLACES = n - 1;
3    condition PlaceAvailable = true;
4    staterequired (int usedPlaces);
5    exclusive, selfexclusive {
6      out port ResrvT {
7        guard  (PlaceAvailable) {
8          after {if (usedPlaces > MAX_PLACES) PlaceAvailable = false;}
9        }
10      }
11     out port LevT {
12       guard  (true) {
13         after { PlaceAvailable = true;}
14       }
15     }
16   }
17   in port ResrvT;
18   in port LevT;
19 }
```

Listagem 8.2 - Conector para acesso ao módulo mesa

O conector que interliga os filósofos à mesa é configurado com um guarda, que garante o acesso a esta de, no máximo, *N-1* filósofos (listagem 8.2). Trata-se de uma das soluções possíveis para se evitar-se o bloqueio (*deadlock*) do sistema [160]. O número de filósofos é passado como parâmetro, quando o conector é instanciado. Assim, este conector precisa inspecionar o número de filósofos que já estão à mesa (linha 4 e 8) e compará-los com o número máximo permitido de filósofos. O guarda é fechado quando a mesa está ocupada com *N-1* filósofos. Os pedidos para reservar um lugar à mesa ou libera-la são repassados em exclusão mútua e de forma serial (linha 5).

```
1  connector RequestFork { // classe de conector que interliga filósofos aos garfos
2    condition Free = true;
3    exclusive, selfexclusive {
4      out port ResrvT {
5        guard  (Free) {
6          before {Free = false;}
7        }
8      }
9      out port LevT {
10       guard {
11         after {Free = true;}
12       }
13     }
14   }
15   in port ResrvT;
16   in port LevT;
17 } ReqFork [i];
```

Listagem 8.3 - Conector para acesso aos módulos garfo

A classe de conector que interliga os filósofos aos garfos (listagem 8.3) também garante que o acesso aos mesmos será feito em exclusão mútua e serialmente (linha 3).

Um guarda mantém o estado do garfo (*Free*) e bloqueia o acesso ao garfo se ele estiver ocupado (linha 5), ou libera o acesso quando o garfo é desocupado (linha 10).

A arquitetura da aplicação é apresentada na listagem 8.4. Neste exemplo, foi utilizada a parametrização para criação de instâncias de filósofos e de garfos (linhas 4 e 5). Observa-se que na configuração das ligações, existe uma instância do conector de acesso, *RequestFork*, para cada módulo garfo (linhas 8 a 13). Para o acesso à mesa, só uma instância de conector deve existir, por isso utilizou-se o nome da classe deste conector. O uso de índices, em tais casos, facilita a descrição da aplicação.

```
1   module ceiaFilosofos (int N) {
2     instantiate mesa Mesa;
3     for (i=0 to N) {
4       instantiate filosofo Filo[i];
5       instantiate garfo Garfo[i];
6     }
7     for (i=0 to N) {
8       link Filo[i].GetLeftFork to Garfo[i] by RequestFork [i];
9       link Filo[i].ReleaseLeftFork to Garfo[i] by RequestFork [i];
10      link Filo[i].GetRightFork to Garfo[(i+1) mod 5]
11                                      by RequestFork [(i+1) mod N];
12      link Filo[i].ReleaseRightFork to Garfo[(i+1) mod 5]
13                                      by RequestFork [(i+1) mod N];
14      link Filo[i] to Mesa by RequestTable (N);
15    }
16  }
17  instantiate ceiaFilosofos as 5F;
18  start 5F;
```

Listagem 8.4 - Arquitetura da aplicação

A aplicação pode ser reconfigurada facilmente para um ambiente distribuído, utilizando-se uma composição dos conectores de acesso ao garfo e mesa com um conector RMI, por exemplo. Uma reconfiguração similar foi proposta no exemplo dos produtores-consumidores, para o aspecto de distribuição (seção 5.3.3, capítulo 5). Também existe flexibilidade para se impor outras características de coordenação. Por exemplo, se fosse necessário dar prioridade a determinados filósofos, isto seria feito reconfigurando-se o conector *RequestTable*.

8.3 Chão de fábrica

Uma aplicação de chão de fábrica, constituída de uma estrutura de células de manufatura, representa uma classe de aplicações em que seus elementos, tipicamente, devem ser reconfigurados ou devem atender a novos requisitos dinamicamente.

Neste exemplo, uma célula de manufatura pode ser configurada para

desempenhar determinada função: (i) produzir uma parte de um produto, (ii) montar o produto final a partir de outras partes, ou (iii) embalar o produto acabado. Uma célula pode ser reconfigurada para produzir novos componentes ou para alterar a forma de produção. A comunicação entre as células se dá através de esteiras rolantes. A fábrica em questão é uma cervejaria onde algumas células produzem cerveja e outras, *chopp*. Um tipo diferente de célula está preparado para envasar o produto em garrafas, barris ou latas, de acordo com sua especialidade.

Na descrição da arquitetura da fábrica, as células de produção e envasamento são representadas por módulos. Por usa vez, as esteiras são representadas por conectores. Uma vez em operação, é possível alterar a estrutura da fábrica, através de comandos de configuração. A listagem 8.5 apresenta a configuração dos módulos.

```
1   module CelEnvas (int Config, int Capacidade){ // tipo  células de envasamento
2                              // o tipo de envasadora pode ser configurado na
3                              // inicialização da instância do módulo ou através da porta ConfigTipo
4      int Espaco;             // mantém o estado da célula
4      in port int EntrProd (int tipo, int ctl) EntradaProduto;
5      out port ProdTin SaidaProdutoEnvasado ;
6      in port int (int tipo) ConfigTipo;
7   }
8
9   module CelProd (int Config){ // tipo célula de produção genérica
10     out port ProdTin SaidaProduto;
11     in port int (int tipo) ConfigTipo;
12  }
13
14  port int ProdTin (int EntradaQualquerProduto, int Serie);
15
16  module Container (int Capacidade){ // tipo container
17   in port ProdTin EntradaProd;
18  }
```

Listagem 8.5 - Módulos unidade de fabricação

A classe de módulo representando uma célula produtora (linha 9) possui as seguintes portas:

- uma porta de saída (linha 10), por onde o produto produzido é encaminhado a uma célula envasadora;
- uma porta de entrada (linha 11), por onde um operador pode configurar o tipo de produto a ser produzido (cerveja ou *chopp*).

As células de envasamento são representadas por uma classe de módulo (linha 1), que possui as seguintes portas:

- uma porta de entrada (linha 4), por onde o produto a ser envasado é recebido;

- uma porta de saída (linha 5), por onde o produto envasado é enviado para o estágio seguinte;
- uma porta de entrada (linha 6), por onde um operador configura o tipo de recipiente a ser utilizado (garrafa, barril ou lata).

Além das células de produção, existe um *container* que recebe a produção para ser estocada. O *container* também é representado por um módulo. O módulo *container* é configurado para aceitar produtos acabados (barril, lata ou garrafa) até sua capacidade máxima (inicializado quando o módulo é instanciado).

As unidades de fabricação são interligadas por esteiras rolantes[10]. Um tipo de esteira comum é responsável pelo transporte dos produtos entre as unidades de fabricação. Além deste, dois tipos de esteiras "inteligentes" são necessárias. Um deles é utilizado para interligar as unidades de envasamento ao *container*. Se a capacidade do *container* se esgotar, esta esteira deve parar o transporte automaticamente. Um outro tipo, uma esteira de manobra, interliga unidades de produção a unidades de envasamento. Se uma unidade de envasamento estiver sobrecarregada, esta esteira deve estar programada para escoar a produção, automaticamente, em direção a uma unidade de envasamento alternativa. Cada esteira é modelada adequadamente por um conector. Tanto a função de transporte, função primária da esteira, quanto a sincronização com os módulos envasadores e *container,* podem ser encapsuladas em conectores.

Um conector que representa uma esteira comum possui apenas uma porta de entrada e uma de saída para fazer o transporte de produtos (listagem 8.6).

```
1  connector EsteiraComum{
2     in port ProdTin Entrada;
3     out port ProdTin Saida;
4  } Esteira1, Esteira2;
```

Listagem 8.6 - Conector esteira comum

No conector que interliga uma célula ao módulo *container* (listagem 8.7), um guarda bloqueia a remessa de novos produtos quando o limite do *container* é atingido

---

[10] Observa-se que para o transporte de líquido uma esteira rolante não é apropriada. O ideal , neste caso, é utilizar tubulações e válvulas apropriadas para esta finalidade. Entretanto, o esquema apresentado é geral, e pode ser usado em aplicações reais. Utilizamos nomes de produtos e componentes conhecidos para simplificar o entendimento do exemplo.

(linha 6). Observa-se que a capacidade da esteira poderá ser configurada com um limite inferior ao do *container*, como garantia, através do parâmetro *Capacidade*, passado na sua criação (linha 1).

```
1   connector EsteiraContainer (int Capacidade){
2       condition Vazio = true;
3       int Num = 0;
4       in port ProdTin Entrada;
5       out port ProdTin {
6          guard (Vazio) {
7             after {
8                Num = Num + 1;
9                if (Num == Capacidade) Vazio = false;
10            }
11         }
12      } Saida;
13  } EsteiraC;
```

Listagem 8.7 - Conector esteira *container*

O conector representando a esteira de manobra possui duas tarefas no controle do fluxo da produção (listagem 8.8). Primeiro, através de um guarda, a remessa de novos produtos para o módulo do próximo estágio da produção pode ser bloqueada, se este estiver sobrecarregado (linha 8). Esta condição será verificada inspecionando-se a variável *Espaco* no módulo célula envasadora (linha 3). O escoamento da produção poderá ficar bloqueado durante um prazo, determinado por um parâmetro (linha 1), na instanciação da aplicação. Esgotado este tempo (linha 6), o fluxo é redirecionado para uma porta de serviço, conectada à célula envasadora alternativa (linha 7). Com isso, permite-se que um aumento na produção, nos primeiros estágios, tenha condição de ser absorvido por células envasadoras com tecnologia mais antiga ou de menor capacidade, sem paralisar a produção, enquanto a célula envasadora principal estiver sendo redimensionada para a nova carga.

```
1   connector EsteiraMnb (int to) {
2      condition Vazio = true;
3      staterequired Espaco;
4      in port ProdTin EntradaProduto;
5      out port ProdTin {
6         timeout (to); // aguarda bloqueada por duas unidades de tempo
7         alternative (Reenvia); // expirado o prazo a requisição é enviada pela porta Reenvia
8         guard (Vazio)
9            after {
10
11              if (Espaco == 0) Vazio = false;
12              else Vazio = true;
13           }
14        }
15     } SaidaProdutoEnvasado ;
16     out port ProdTin Reenvia; // porta alternativa
17  } EsteiraManobra;
```

Listagem 8.8 - Conector esteira de manobra

Descritas as unidades de fabricação e as esteiras, configura-se, então, a estrutura da fábrica (listagem 8.9). Inicialmente, as células são instanciadas (linhas 5 a 11). Em seguida, as mesmas são interconectadas pelas esteiras (linhas 13 a 18). Observa-se que ao se instanciar o módulo *container* foi passada a sua capacidade como parâmetro (linha 11). Isto feito, a operação da fábrica pode ter início (linha 23). A figura 8.1 representa a configuração recém instanciada.

```
1    module BeerFactory {
2     int CHOPP = 0, CERV = 1;
3     int LATA = 0, GARR = 1, BARR = 2;
4
5     instantiate CelProd as ProdChopp (CHOPP);
6     instantiate CelProd as ProdCerv (CERV);
7
8     instantiate CelEnvas as EnvasLata (LATA)
9     instantiate CelEnvas as EnvasBarr (BARR);
10
11    instantiate Container as Container1 (100);
12
13    link ProdChopp.SaidaProduto to EnvasBarr.EntradaProduto by Esteira1;
14    link ProdCerv.SaidaProduto to EnvasLata.EntradaProduto by Esteira2;
15    link EnvasBarr.SaidaProdutoEnvasado to Container1.Entrada
16                                                        by EsteiraC (90);
17    link EnvasLata.SaidaProdutoEnvasado to Container1.Entrada
18                                                        by EsteiraC (90);
19  } BF;
20
21
22  instantiate BF;
23  start BF;
```

Listagem 8.9 - Fábrica de cerveja configurada

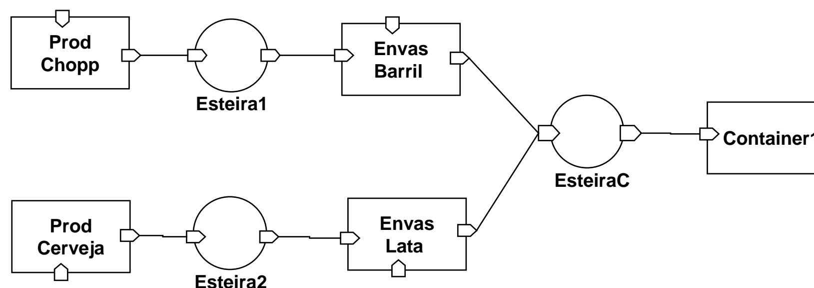


Figura 8.1 - Configuração inicial da fábrica

A arquitetura da fábrica pode ser reconfigurada para adaptá-la a um novo esquema de produção. Isto poderia ocorrer, por exemplo, no caso de a fábrica ter que aumentar a produção de cerveja. Uma das possíveis alterações na descrição da configuração é indicada na listagem 8.10. Neste caso, inseriu-se mais uma unidade envasadora, que utiliza garrafas, na linha de produção (linha 4). A referida unidade é

interligada à célula produtora de cerveja por uma esteira de manobra, que substitui a esteira original (linha 10). A produção excedente, que não tenha condição de ser envasada em latas, é escoada para esta envasadora alternativa. A célula produtora de cerveja também deve ser ligada à célula de envasamento de latas pela esteira de manobra (linha 8). A figura 8.2 apresenta a nova situação.

```
1   // A linha de produção é reconfigurada segundo as novas necessidades
2   module BeerFactory {
3
4     instantiate CelEnvas as EnvasGarr (GARR);
5   ...
6     link EnvasGarr.SaidaProdutoEnvasado to Container1.Entrada
7                                                         by EsteiraC (90);
8     link ProdCerv.SaidaProduto to EnvasLata.EntradaProduto
9                                                         by EsteiraManobra;
10    link EsteiraManobra.Reenvia to EnvasGarr.EntradaProduto;
11
12  ...
13  } BF;
```

Listagem 8.10 - Nova configuração da fábrica

Alternativamente, seria possível a um operador reprogramar a fábrica, sem paralisar totalmente a produção, através de comandos de configuração. Por exemplo, um console de operação ficaria disponível na divisão de engenharia de produção, a qual, por seu turno, estaria encarregada de realizar reconfigurações na planta de fabrica. A listagem 8.11 apresenta os passos para a reconfiguração, levando a configuração inicial da fábrica, figura 8.1, para a nova configuração, representada na figura 8.2.

```
1   block ProdCerv;
3
4   instantiate CelEnvas as EnvasGarr (GARR);
5
6   link EnvasGarr.SaidaProdutoEnvasado to Container1.Entrada
7                                                       by EsteiraC (90);
8   link ProdCerv.SaidaProduto to EnvasLata.EntradaProduto
9                                                       by EsteiraManobra;
10  link EsteiraManobra.Reenvia to EnvasGarr.EntradaProduto;
11
12  start EnvasGarr;
13  resume ProdCerv;
```

Listagem 8.11 - *Script* de reconfiguração da fábrica

Observa-se que, durante a reconfiguração, a paralisação das células foi a mínima necessária. Apenas a célula produtora de cerveja foi bloqueada (linha 1), para que as alterações pudessem ser feitas (linhas 4 a 10). Em seguida, iniciou-se a operação da nova célula envasadora (linha 12) e retomou-se a operação da célula produtora de cerveja (linha 13). Uma outra anotação que vale ser feita é que as atividades de

reconfiguração, neste exemplo, foram programadas por um operador externo, com o devido conhecimento do esquema de produção (a arquitetura da aplicação). Desta forma, as operações de parada e ativação de máquinas são realizadas com segurança.

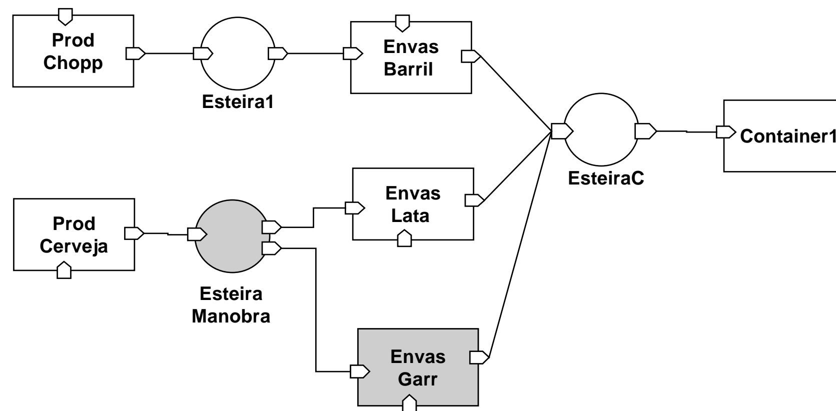


Figura 8.2 - Reconfiguração da fábrica

8.4 TicTacToe

O exemplo do Jogo da Velha (*TicTacToe)* será utilizado para ilustrar dois pontos de R-RIO: o uso de conectores reflexivos por contexto e o encapsulamento de *design patterns* em conectores. Como ponto de partida, será usada a implementação do jogo oferecida no *kit* de desenvolvimento da linguagem Java, disponibilizado pela Sun Microsystems [201]. Esta implementação apresenta vários dos problemas de entrelaçamento de código, apontados no início do capítulo 2. Por exemplo, em uma única classe, *TicTacToe*, estão contidas as estruturas de dados para representar o jogo, a interface com o jogador, métodos implementando a *inteligência* do computador (o adversário do jogador) e a interface gráfica que exibe o jogo. Uma nova versão da apresentação gráfica do jogo ou a introdução de um outro jogador são tarefas complicadas neste caso.

Apresentamos nesta seção uma solução para o Jogo da Velha em R-RIO, utilizando uma arquitetura mais modular do que a original. Realizamos também a comparação desta solução com uma versão elaborada sob a ótica de AOP (apresentada no capítulo 3), o que permite ressaltar algumas vantagens de nossa proposta.

8.4.1 TicTacToe em AOP

AspectJ™ é uma implementação de AOP, baseada em Java, sendo desenvolvida na Xerox [117]. No ambiente AspectJ, uma solução alternativa do Jogo da Velha foi utilizada para exemplificar como um *design pattern* do tipo ***Observer*** pode ser concentrado em um aspecto, separando explicitamente a programação dos objetos, da maneira pela qual estes interagem. O *design pattern* ***Observer*** trata da geração de eventos em um objeto observável e da reação esperada em objetos *observadores* destes eventos [35].

A estrutura da aplicação apresenta 4 objetos (figura 8.3, retirada de [204]): o jogo propriamente dito, implementado por uma classe *TicTacToe*, um objeto *Player*, representando o jogador que interage com o jogo, e dois objetos que exibem a evolução de uma partida, *BoardDisplay* e *StatusDisplay*. Na figura 8.3, apenas os métodos relacionados com o protocolo de interação são destacados, mas deve subentender-se que as classes implementam os métodos relacionados com os aspectos funcionais, por exemplo *startGame*, para iniciar o jogo, ou *putMark*, para colocar a marca de um jogador.

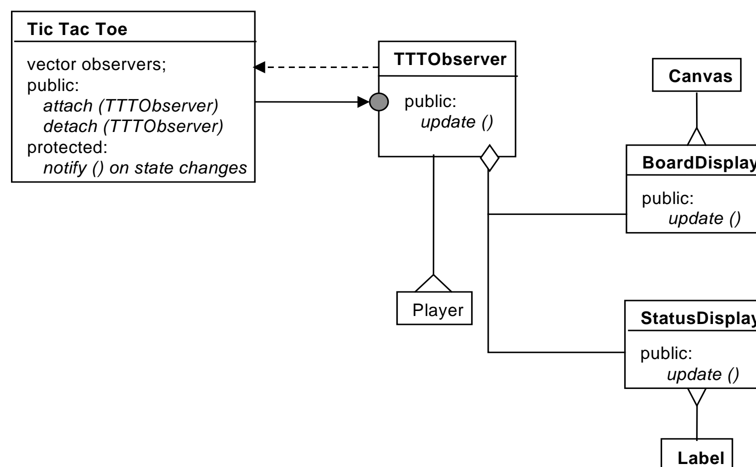


Figura 8.3 - Classes da aplicação TicTacToe

AspectJ acrescenta uma construção sintática à linguagem Java original, chamada ***aspect***, para a descrição de aspectos de uma aplicação. No corpo de um *aspect*, podem ser definidos trechos de código, chamados ***weaves***, que serão entrelaçados com um programa Java, ampliando ou modificando as funções básicas da aplicação. Um pré-processador, denominado *weaver,* pode entrelaçar o código descrito nos *weaves*, em determinados pontos de junção das classes programadas em Java. É possível usar um

weave para introduzir novas variáveis ou métodos (***introduce***), ou ainda para estender o código de um método (***advise***) de uma classe.

O protocolo de interação baseado no *design pattern Observer*, descrito em AspectJ, é apresentado na listagem 8.12. Observa-se que os métodos *attach*, *detach* e *notify* não fazem parte das classes originais. Estes elementos são encapsulados no aspecto *TTTDisplayProtocol* (linha 1) e devem ser introduzidos no código da classe *TicTacToe* original, através do *weaver*. Seguindo a listagem 8.12, pode-se fazer outras observações sobre o aspecto *TTTDisplayProtocol*:

- A classe *TicTacToe* precisa conter uma instância de *TTTObserver* para "tornar-se observável". Isto se consegue através dos *weaves* ***introduce*****,** que introduzem, no código original de *TicTacToe*, o código necessário à criação da instância de um vetor de observadores (linha 2), além de outros métodos para suportar o cadastro (*attach* e *detach*) de observadores (linhas 5 e 8);
- Na classe *TTTObserver* é introduzido um método *update*, que, por sua vez, invoca os métodos *update* nos observadores "concretos", *status* e *display* (linha 11);
- Nos observadores concretos também são introduzidos métodos para tratar as atualizações de estado. Assim, tanto na classe *Board* (linha 14) como na classe *Status* (linha 15) são chamados os métodos *updateStatus,* para receber o novo estado do jogo e *repaint*, a fim de atualizar a janela de exibição.

```
1    aspect TTTDisplayProtocol {
2      introduce Vector TicTacToe.observers = new Vector();
3      introduce TicTacToe TTTObserver.theGame;
4
5      introduce void TicTacToe.attach(TTTObserver obs) {
6        observers.addElement(obs); }
7
8      introduce void TicTacToe.detach(TTTObserver obs) {
9        observers.removeElement(obs); }
10
11     introduce void TTTObserver.update() {
12       board.update(theGame); status.update(theGame); }
13
14     introduce void BoardDisplay.update(TicTacToe game),
15                void StatusDisplay.update(TicTacToe game) {
16       updateStatus(game.getStatus()); repaint(); }
17
18     // para todos os métodos que alterem o estado
19     advise * TicTacToe.startGame(*),* TicTacToe.putMark(*)
20            * TicTacToe.newPlayer(*),* TicTacToe.endGame(*){
21       static after {
22         for (int i = 0; i < observers.size(); I++)
23           ((TTTObserver)observers.elementAt(I)).update(); }
24   } }
```

Listagem 8.12 - Protocolo de interação em AspectJ™

Além de introduzir métodos e variáveis, o aspecto *TTTDisplayProtocol* ainda acrescenta código no final dos métodos originais do objeto *TicTacToe*, para que os eventos de mudança de estado sejam emitidos. Isto é feito com o modo ***advise*** (linha 20). No final dos métodos *startGame*, *putMark*, *newPlayer* e *endGam*e do objeto *TicTacToe*, é adicionado um código para varrer o vetor de observadores e invocar explicitamente o método *update* dos mesmos (esta foi a implementação utilizada para o método *notify*, incluído na listagem 8.12, linhas 22 e 23).

8.4.2 TicTacToe em R-RIO

A estrutura do Jogo da Velha, apresentada na seção anterior, pode ser implementada segundo a metodologia para configuração de aplicações do *framework* R-RIO. A proposta é estruturar o jogo com três classes de módulos: *TTTGame*, *TTTPlayer* e *TTTDisplay*, e realizar a interação entre estes módulos através de um conector, *CObserver*, que encapsula o *design pattern Observer* (figura 8.4). Cada um dos módulos da arquitetura poderia ainda ser descrito pela composição de dois módulos: um deles, implementando a funcionalidade básica e, o outro, especializado em exibir as informações graficamente. Esta separação de interesses facilitaria, também, a modificação do jogo para a participação de dois jogadores, ao invés de se ter o computador como adversário.

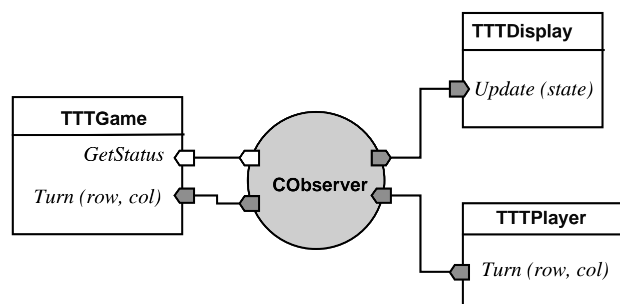


Figura 8.4 - Aplicação TicTacToe em R-RIO

Em nossa solução, o módulo *TTTGame* pode ser concebido sem a preocupação com o número de observadores, nem com a geração de notificações. Como toda interação entre *TTTPlayer* e *TTTGame* passa pelo conector, o próprio conector toma a iniciativa de enviar para os observadores as requisições de *update* contendo as informações de *status* do jogo. Uma requisição de *update* é enviada sempre que

necessário e apropriado (antes de uma requisição ou após a requisição ser completada). Uma descrição dos módulos desta arquitetura, em CBabel, é apresentada na listagem 8.13.

```
1    port int Turn (int Row, int Col);
2    port int GetStatus (void);
3    port int Update (int Status);
4
5     module  TTTGame {
6        in port Turn;
7        in port GetStatus;
10
11        map CLASS Java "game"
12    } Game;
13    module  TTTPlayer {
14        out port Turn;
15
16        map  CLASS Java "player";
17    } Player;
18    module  TTTDisplay {
19        in port Update;
20
21        map  CLASS Java "game_grid";
22    } DisplayGrid;
```

Listagem 8.13 - Declaração de portas e módulos

Inicia-se a descrição da aplicação definindo-se os tipos de portas (linhas 1 a 3) e as classes de módulos a serem utilizadas. Estes elementos são declarados fora de qualquer escopo, de modo a permitir a reutilização dos mesmos. Os módulos são descritos indicando-se as instâncias de portas utilizadas (linhas 6 e 7, por exemplo) e o mapeamento para uma implementação. Também assim, o módulo *TTTGame* é implementado pela classe *game.class*, programada em Java (linha 11).

Em seguida o conector que interliga os módulos é definido (listagem 8.14).

```
1    connector CObserver{
2      in  port Turn TurnIn;
3      out port Update;
4      out port GetStatus;
5      out port Turn TurnOut;
6
7      exclusive {Update, GetStatus, TurnOut};
8      exclusive {TurnIn, TurnOut};
9      selfexclusive {TurnOut};
10
11     interaction contract {
12         // descrição dos contratos de interação entre as portas do
13         // conector, implementando o design pattern Observer
14         TurnIn > TurnOut > GetStatus > Update;
15     }
16
17      map CLASS Java "CObserver";
18    } CObs;
```

Listagem 8.14 - Declaração do conector

Na classe de conector *CObserver*, as portas também devem ser declaradas (linhas 2 a 4). Note que isto foi feito com duas instâncias diferentes de portas do tipo *Turn*. *TurnIn* (linha 2) será utilizada pelo módulo *Player*, e a porta *TurnOut* (linha 4) pelo módulo *Game*. As portas *Update* e *GetStatus* não receberam nomes de instância. Aspectos de sincronização entre as portas do conector são declarados em seguida (linhas 7 a 9). O *design pattern Observer* é associado ao conector através do contrato de interação (linhas 11 a 15). Dentro deste contrato, descreve-se como o conector irá gerenciar as interações entre os módulos, segundo o *design pattern* (detalhes da sintaxe podem ser consultados na seção A.4.1 - apêndice A). Com isso, encapsula-se, no conector, o aspecto de interação entre os módulos, sem a necessidade de interferir nos mesmos.

```
1    module  TicTacToe  {
2      instantiate   Game, Player, DisplayGrid;
3      link Player, DisplayGrid to Game by CObs;
4    }
5    module  TicTacToe  TTT;
6    instantiate  TTT;
7    start TTT;
```

Listagem 8.15 - Declaração da estrutura da aplicação

Após a descrição dos módulos e conectores, a própria aplicação é descrita como um módulo, *TicTacToe* (listagem 8.15). Neste módulo descreve-se a configuração da topologia de interconexão dos módulos e conectores (linhas 2 e 3). Em seguida, cria-se também uma instância de *TicTacToe* (linhas 5 e 6). Neste momento são criadas todas as instâncias dos módulos internos e realizadas as conexões especificadas. A partir daí, a instância *TTT* da aplicação está criada, e com o comando *start* (linha 7) sua execução é iniciada.

Uma outra possibilidade para a aplicação é a configuração dos jogadores e observadores em um sistema distribuído A listagem 8.16 exemplifica esta nova configuração. Um conector implementando comunicação por RMI é acrescentado (linha 1) e composto com o conector *CObs*, para ligar os módulos (linha 7). Estes são instanciados em nós diferentes (linhas 4 a 6).

```
1    connector Comm {
2       map CLASS Java "RmiGCon";
3    } PCRmi;
4    instantiate   DisplayGrid at nó1;
5    instantiate   Player at nó2;
6    instantiate   Game at nó3;
7    link Player, DisplayGrid to Game by CObs.PCRmi;
```

Listagem 8.16 - Reconfigurando a aplicação

8.4.3 Comparação

Com esforço equivalente a AspectJ, e com menos interferência nos módulos originais, foi possível conceber a mesma aplicação em R-RIO. Em relação a *AspectJ*, esta solução apresenta a vantagem de poder evoluir dinamicamente. Por exemplo, um número arbitrário de observadores pode ser introduzido, mesmo durante a execução, sem a necessidade de novo pré-processamento. A criação de múltiplas instâncias pode ser feita, atribuindo-se um nome diferente a cada uma delas, ou através de construções com índices na linguagem de configuração.

Na solução em R-RIO, observadores não precisam executar métodos de *attach* ou *detach* no módulo do jogo. Basta que o novo observador seja ligado ao jogo através de sua porta de entrada. É necessário, entretanto, que um método *GetStatus*, por exemplo, que retorne o estado do jogo, esteja disponível no módulo *TTTGame*. Na versão de AOP, o objeto *TTTObserver* tem de tomar a iniciativa de *varrer* um vetor de observadores e invocar o método *update* em cada um. É possível, em R-RIO a otimizar a comunicação com um grupo de observadores, encapsulando-se no conector um mecanismo para disseminar as informações de atualização de estado a todos os observadores (figura 8.5).

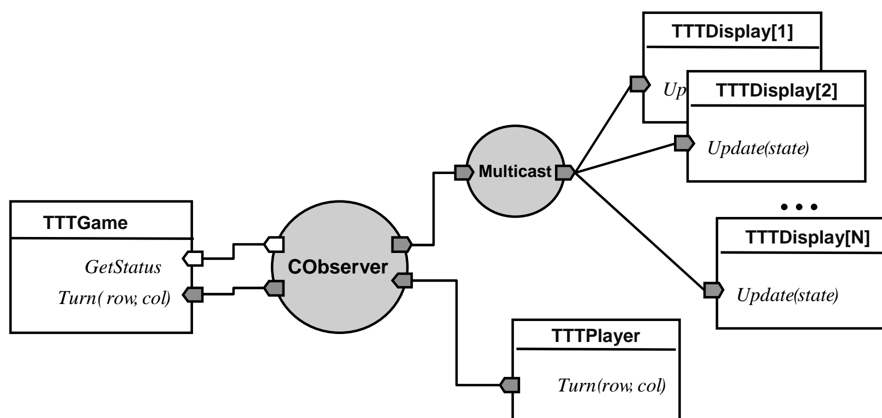


Figura 8.5 - Versão distribuída do jogo

No caso de AspectJ, a separação de interesses é mantida entre aspectos funcionais e não funcionais, até que estes sejam pré-processados pelo *weaver*. Após este passo, o código dos aspectos não-funcionais (no exemplo, o código dos aspectos de interação) é entrelaçado nas classes do código original, resultando em um único programa em Java. Observa-se que, para este esquema funcionar, existe uma quebra de encapsulamento do código dos objetos originais.

Em nossa proposta, a observação da separação de interesses e o uso de conectores reflexivos por contexto (seção 4.4.1, capítulo 4) permitem que se adapte, automaticamente, a arquitetura básica do exemplo para suportar outros jogos de tabuleiro com estruturas similares. Assim, as classes do Jogo da Velha, `TTTplayer`, `TTTgame` e `TTTdisplay`, da figura 8.6(a), são substituídas por classes de um jogo de Xadrez (figura 8.6(b)). Observa-se que as interfaces dos módulos deste novo jogo não são equivalentes às do primeiro. A reflexão baseada no contexto permite que o conector *CObserver* seja adaptado automaticamente ao contexto das novas interfaces, no momento em que os módulos são interligados.

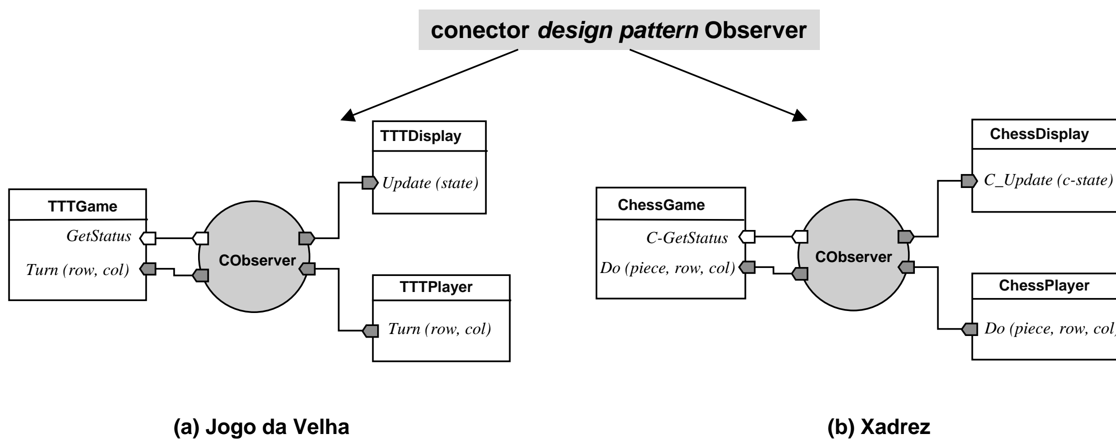


Figura 8.6 - Conector CObserver em duas aplicações diferentes

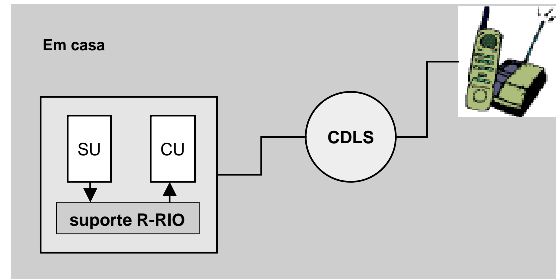
8.5 Telefone 3 em 1

As áreas de computação pervasiva e ubíqua têm sido alvo de intensa pesquisa [1, 205]. Uma das visões destes novos paradigmas é o uso de dispositivos portáteis como PDAs, *notebooks*, ou mesmo telefones "inteligentes", para manter conexão à Internet, para acessar bases de dados ou receber mensagens eletrônicas, por exemplo, enquanto se está em trânsito, entre a casa e o trabalho.

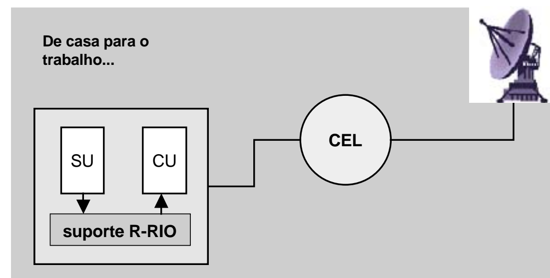
O *telefone 3-em-1* é apresentado como uma das aplicações possíveis da tecnologia *Bluetooth* [206], que visa prover comunicação móvel para dispositivos que se encontram continuamente conectados a outros dispositivos, através de rádio. A conexão entre dispositivos através de *Bluetooth* seria transparente e dinâmica, mesmo no caso de mobilidade, servindo como opção de infra-estrutura (entre outras como o IrDA [207] e HomeRF [208]) para a computação ubíqua.

Em nossa versão para esta aplicação, o *telefone 3-em-1* é um dispositivo que pode operar em três modos:

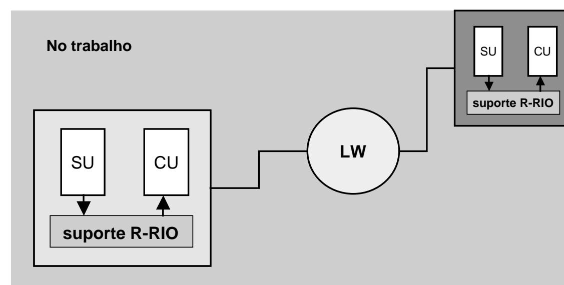
- em casa, no raio de alcance de uma estação-base, ele funciona como um telefone sem fio, ligado a uma linha telefônica fixa (com custo de telefonia fixa);



- no trajeto de casa para o trabalho, ou do trabalho para casa, quando se está em movimento e no alcance de uma antena, o telefone pode ser usado como aparelho celular (com custo de telefonia celular);



- quando o aparelho detecta que está na área de alcance de outro telefone 3-em-1, ou em uma rede sem-fio (*wireless network*), ele pode ser usado como comunicador sem fio, *walky talky* ou estação da rede (com custo reduzido).



Em nosso exemplo, consideramos que o *telefone 3-em-1* possui interfaces de comunicação para as três possibilidades de interação.

Cada sistema possui uma instância do suporte R-RIO embutida. Pressupõe-se, também, uma infra-estrutura de IP móvel [209], em que o número IP dos sistemas pode mudar dinamicamente. Problemas de latência de *handoff* [210] são considerados desprezíveis, e o roteamento, neste contexto, é considerado resolvido.

Admite-se que no *telefone 3-em-1* existem sensores que conseguem detectar a entrada do aparelho no raio de comunicação de um outro sistema com o qual ele pode interagir. Deste modo, ao se aproximar suficientemente de uma estação base de telefone sem-fio, isto é detectado pelo sensor correspondente, e o *telefone 3-em-1* pode operar como telefone sem fio. Similarmente, ao entrar no raio de comunicação de um outro

sistema, passa a existir, entre o *telefone 3-em-1* e este outro sistema, um enlace de comunicação. Este enlace e a infra-estrutura de IP móvel são suficientes para permitir a comunicação por pacotes IP.

A simulação da aplicação do *telefone 3-em-1* ilustra o uso de conectores para encapsular aspectos de QoS e o emprego de reconfiguração dinâmica para alterar a arquitetura a aplicação em face a mudanças de condições de operação. Também é ilustrada a reutilização da mesma arquitetura para a inclusão de suporte de voz ou vídeo, além do suporte a texto.

Na solução em R-RIO, cada sistema é modelado como um módulo: (i) o aparelho do usuário, que pode iniciar a interação com os outros pares, (ii) a estação base para o telefone sem-fio, (iii) a antena de uma ERB (Estação-rádio-base) de celular e (iv) um outro aparelho 3-em-1 (no ambiente de trabalho).

A interação entre o telefone 3-em-1 e o seu par interlocutor será intermediada por um conector. Este pode ser concebido para encapsular os seguintes aspectos não-funcionais:

- a comunicação, que deverá ser adaptada para cada um dos modos de operação descritos anteriormente;
- correção de erro adicional, quando necessário;
- criptografia / decriptografia, quando solicitado;
- QoS, de acordo com contratos estabelecidos.

Tais aspectos não-funcionais podem ser viabilizados, em cooperação, pelo telefone 3-em-1, localmente e por seu par interlocutor. Por exemplo, a correção de erro adicional e a criptografia devem ser suportadas pelos dois interlocutores. Para isso, os recursos locais, de cada interlocutor, precisam estar disponíveis e corretamente configurados.

A figura 8.7 apresenta a arquitetura do telefone 3-em-1, e a listagem 8.17, a descrição dos componentes e configuração em CBabel. O telefone 3-em-1 é composto pelos seguintes módulos:

<u>ControlPanel, Presentation e Input</u>. Através destes módulos o usuário interage com outro usuário. A interface pode ser um simples *teletexto,* como no protótipo

desenvolvido, ou uma interface com suporte à mídia de voz, como em um telefone padrão, voz e vídeo, ou ainda uma tela gráfica como nos PDAs.

Monitor. Encapsula as funções dos sensores de *hardware* e detecta mudanças nas condições de operação. Ao detectar uma mudança, o Monitor avisa ao Configurador de QoS desta mudança, e este pode iniciar uma reconfiguração.

Configurador de QoS. Conhece a arquitetura da aplicação e encapsula a gerência desta arquitetura. O Configurador de QoS interpreta as mudanças indicadas pelo Monitor e reconfigura a aplicação para operar nas novas condições. A reconfiguração consiste na seleção de um conector que satisfaça a aplicação nas atuais condições de operação.

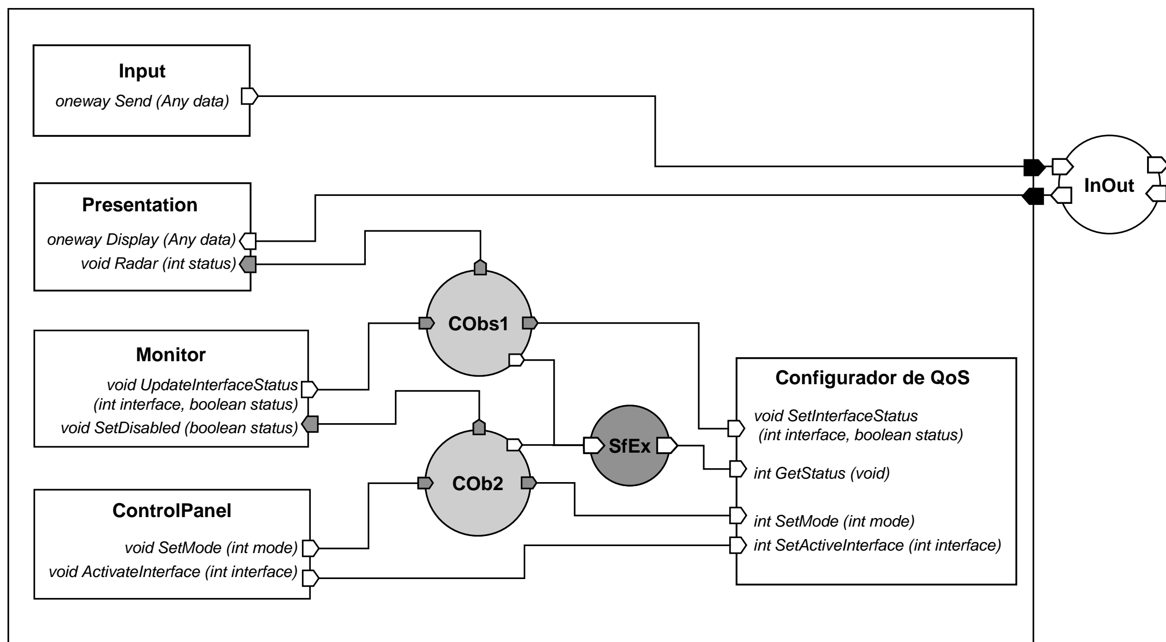


Figura 8.7 - Arquitetura do telefone 3-em-1

O módulo *Input* (linha 19) é responsável pela entrada das informações (voz, dados ou vídeo). A porta assíncrona (*oneway* - linha 20) e o uso do tipo *Any* para transportar as informações oferecem flexibilidade a este módulo. O módulo *Presentation* (linha 23) é responsável por exibir as informações, recebendo um fluxo, também assíncrono, proveniente do interlocutor, pela invocação do método *Display*. O método *Radar* (linha 25), quando invocado, atualiza o posicionamento do ícone do telefone na tela do simulador.

```
1   module Monitor { // portas de saída indicam o status de cada interface
2      out port void UpdateInterfaceStatus(int interface, boolean status);
3      in port void SetDisabled (boolean status);
4   }
5
6   module Configurador-de-QoS {
7      in port int GetStatus (void);    // para a apresentação
8      in port void SetInterfaceStatus(int interface, boolean status);
9      in port int SetMode (int mode); //  automático ou manual
10     in port int SetActiveInterface (int interface);
11                                     // em modo manual, indica a interface a ser usada
12  }
13
14  module ControlPanel {
15     out port void ActivateInterface (int interface);
16     out port void SetMode (int mode); //  automático ou manual
17  }
18
19  module Input {
20     out port oneway Send (Any data); // parâmetro de tipo Any permite qualquer tipo
21  }
22
23  module Presentation {
24     in port oneway Display (Any data); // texto / voz / imagem
25     in port void Radar (int position); // posição do telefone 3-em-1
26  }
```

Listagem 8.17 - Descrição de componentes e estrutura em CBabel

A listagem 8.18 descreve os conectores internos da aplicação. O conector *CObserver* (da seção 8.4.2) é aqui reutilizado para intermediar as interações entre os módulos *Monitor* / *Configurador de QoS* / *Presentation* (instância *Obs1* - linha 10) e *ControlPanel* / *Monitor* / *Configurador de QoS* (instância *Obs2* - linha 21). As mudanças de posicionamento do *telefone 3-em-1* detectadas pelo *Monitor* são enviadas para o *Configurador de QoS*. Este processa tais informações e, baseado em sua programação, seleciona a instância adequada para substituir o conector *InOut* atual. O resultado deste processamento é encaminhado ao módulo *Presentation* para exibição (invocando-se o método *Radar*).

Se o modo de operação manual é selecionado no *ControlPanel*, esta informação é encaminhada para o *Configurador de QoS*, que tomará as medidas cabíveis, como, por exemplo, preparar-se para atender a requisições diretas de ativação de uma interface determinada.

Um terceiro conector, *SfEx*, é utilizado para contornar a condição de corrida que pode acontecer, quando o *Monitor* inicia uma atualização de status, e o *ControlPanel* começa a avisar que o modo manual foi ativado. Um simples contrato de coordenação indica que a porta *GetStatus* deve ser usada em exclusão mútua.

```
1     connector SfEx {
2        in port int GetStatusIn (void);
3        out port int GetStatusOut (void);
4        interaction contract {
5           GetStatusIn.GetStatusOut;
6        }
7        selfexclusive (GetStatusOut);
8     }
9
10    connector CObserver {
11       in port void UpdateInterfaceStatus(int interface, boolean status);
12       out port void SetInterfaceStatus(int interface, boolean status);
13       out port int GetStatus (void);
14       out port void Radar (int position);
15
16       interaction contract {
17         UpdateInterfaceStatus > SetInterfaceStatus > GetStatus > Radar;
18       }
19    } Obs2;
20
21    connector CObserver {
22       in port int SetModeIn (int mode);
23       out port void SetMode (int mode);
24       out port int GetStatus (void);
25       out port void SetDisabled (boolean status);
26
27       interaction contract {
28         SetModeIn > SetModeOut > GetStatus > SetDisabled;
29       }
30    } Obs1;
```

Listagem 8.18 - Descrição dos conectores internos ao *telefone 3-em-1*

Os conectores que ligam o *telefone 3-em-1* ao seu par possuem a mesma interface, diferindo no mapeamento para a implementação. Cada conector encapsula os mecanismos de comunicação relativos a uma das interfaces de comunicação disponíveis. O módulo *Configurador de QoS* seleciona o conector adequado a cada um dos tipos possíveis de ligação. A listagem 8.19 apresenta a descrição destes conectores. Uma classe de conector, *Comm*, é declarada (linha 1), e três instâncias, com mapeamentos diferentes, são criadas a partir desta classe (linhas 13 a 15). Observa-se na configuração que, tanto o fluxo de entrada quanto o de saída estão contidos em um único conector (linhas 3 a 5). Esta solução foi utilizada para facilitar o encapsulamento dos mecanismos de comunicação das interfaces cujos fluxos de entrada e saída possuem dependência entre si. Por exemplo, os slots de tempo de um canal *BlueTooth* podem ser reservados assimetricamente para transmitir ou receber, dentre um número finito de slots disponível. Esta dependência não está descrita em um contrato de QoS, por ser particular a cada tecnologia, e poderá ser encapsulada no código do conector.

As portas dos conectores de comunicação são assíncronas (linha 2 a 4 - *oneway*), permitindo o transporte de qualquer tipo de tráfego. Para a transmissão de

dados em estilo síncrono, instâncias adequadas dos módulos *Presentation* e *Input* podem ser selecionadas, em substituição às atuais, considerando-se que as mensagens de confirmação estariam ao encargo destes módulos. Alternativamente, a comunicação pode ser intermediada por um conector com a finalidade de converter o estilo de interação.

```
1    connector Comm {
2       in port oneway SendIn (Any data);
3       out port oneway SendOut (Any data);
4       in port oneway ReceiveIn (Any data);
5       out port oneway ReceiveOut (Any data);
6
7       interaction contract {
8          SendIn > SendOut;
9          ReceiveIn > ReceiveOut;
10      }
11   }
12
13   connector Comm { map CLASS Java "cellularC"; } CEL;
14   connector Comm { map CLASS Java "baseStationC"; } CDLS;
15   connector Comm { map CLASS Java "bluetooh"; } LW;
```

Listagem 8.19 - Descrição dos conectores básicos para ligação externa

A configuração da estrutura do telefone 3-em-1 segue a descrição da listagem 8.20. Inicialmente, instâncias dos módulos são criadas (linhas 2 a 6). Em seguida, estes módulos são ligados através dos conectores descritos na listagem 8.18. A composição destes conectores é descrita, de forma compacta, na cláusula de ligação dos módulos (linhas 9 e 10). Completa-se a configuração, indicando-se a visibilidade externa das portas de entrada e saída de informações (linhas 13 e 14). Utilizou-se, para isso, a notação com a referência à instância e o nome da porta específica, separada por um ponto. A arquitetura pode ser visualizada na figura 8.7.

```
1    module Phone3in1 {
2      instantiate Monitor;
3      instantiate Configurador-de-QoS;
4      instantiate Presentation;
5      instantiate Input;
6      instantiate ControlPanel;
7
8       // ligação dos módulos pela composição dos conectores CObserver e SfEx
9      link Monitor, Presentation to Configurador-de-QoS by CObs1.SfEx;
10     link ControlPanel, Monitor to Configurador-de-QoS by CObs2.SfEx;
11
12     // portas visíveis do módulo Phone3in1
13     export Presentation.Display;
14     export Input.Send;
15   }
```

Listagem 8.20 - Estrutura do telefone 3-em-1

A aplicação é configurada segundo a descrição da listagem 8.21. Primeiramente,

uma instância do sistema é criada. O *telefone 3-em-1*, batizado de *thePhone*, é instanciado no nó *mobile1* (linha 1). Neste exemplo, isto significa que os módulos são carregados sobre o suporte R-RIO executando no aparelho, e que este foi nomeado *mobile1*. Em seguida, os outros sistemas, interlocutores potenciais do *thePhone*, são instanciados (linhas 2 a 4). Nas linhas 6 a 9 os sistemas são iniciados. Escolheu-se um cenário inicial, em que o módulo *thePhone* opera como celular. Assim, uma ligação com a antena é feita (linha 11), e o módulo *thePhone* pode operar. Observa-se que, numa situação real, a gerência sobre a criação e ativação das instâncias de cada sistema seria distribuída, e de responsabilidade administrativa de entidades diferentes.

```
1   instantiate Phone3in1 as thePhone at mobile1;
2   instantiate Phone3in1 as officePhone at work;
3   instantiate RIOCellAntenna as CA at antenna;
4   instantiate RIOBaseStation as BS at home;
5
6   start CA at antenna;
7   start BS at home;
8   start thePhone at mobile1;
9   start officePhone at work;
10
11  link thePhone to CA by CEL;
```

Listagem 8.21 - Estrutura da aplicação

As informações de *status* atualizadas pelo *Monitor* podem fazer o *Configurador de QoS* iniciar uma reconfiguração. Por exemplo, ao sair do alcance da antena de celular e entrar na área de um outro telefone 3-em-1, o *Configurador de QoS* pode reconfigurar a aplicação, executando o *script* da listagem 8.22. *Scripts* semelhantes são usados nas demais situações de mudança.

```
1   block thePhone at mobile1;
2   link thePhone at mobile1 to officePhone at work by LW;
3   resume thePhone at mobile1;
```

Listagem 8.22 - *Script* para reconfiguração da aplicação quando o telefone 3-em-1 sai da área de alcance da antena de telefonia celular e entra na área de um outro telefone 3-em-1

Os conectores descritos na listagem 8.19 são básicos da aplicação. Eles encapsulam apenas os mecanismos de comunicação relativos à interface que está sendo usada. O que diferencia a abordagem de R-RIO de outras alternativas, é que tais conectores podem operar em composição com outros conectores que oferecem as características não-funcionais, como a criptografia e controle adicional de erro (veja discussão na seção 5.2, capítulo 5). Na listagem 8.23 a descrição destes conectores é apresentada, juntamente com as categorias de QoS utilizadas.

```
1    connector Comm ErrCtl {
2       map CLASS java "crc32";
3    } CRC;
4
5    connector Comm ErrCtl {
6       map CLASS java "parity";
7    } Parity;
8
9    connector CryptDecrypt (Boolean mode){ // mode indica se é criptografia ou decriptografia
10      in port oneway In (Any data);        // para cada sentido uma função
11      out port oneway Out (Any data);
12      map CLASS java "des";
13   }
14
15   QoScategory  ErrorControl {
16     errorDetection: enum {no, parity, crc32, crcISO}
17     errorCorrection: enum {go-back-N, selective-restrans}
18   }
19
20   QoScategory  Cryptography {
21      mechanism: enum {no, des, md5}
22   }
23
24   QoScategory  Communication {
25      flow: enum {voice, data }
26      bandwidth: increasing numeric Kbps;
27   }
```

Listagem 8.23 - Conectores e categorias de QoS

Os perfis de QoS devem ser configurados e, em seguida, verificados e armazenados no repositório de meta-nível. O *Configurador de QoS* é programado para reconhecer e impor os termos do contrato de QoS em cada situação de operação. Um usuário poderia definir o perfil de QoS do seu sistema através de menus de opção, como ocorre para outras características dos aparelhos celulares mais modernos.

Para as categorias de controle de erro e criptografia, o *Configurador de QoS* precisa negociar o contrato com o interlocutor, mas o atendimento a estes contratos não é dependente da tecnologia utilizada em dado momento. Os contratos da categoria de comunicação, por outro lado, são dependentes da tecnologia e requerem ações diferentes do *Configurador de QoS*, em cada situação. Por exemplo, no caso de *Bluetooth*, o canal síncrono (que suporta uma taxa de 64 kbps, *full-duplex*) disponível será reservado para fluxos de voz. Em se tratando de fluxo de dados, uma certa banda pode ser reservada, até o limite disponível na tecnologia (em torno de 720 kbps, em uma direção). Para realizar estas reservas, não será necessário negociar com o interlocutor, mas verificar a disponibilidade de recursos. No caso de comunicação celular, os valores reservados devem ser condizentes com a tecnologia.

Além das informações de contrato de cada categoria, o *Configurador de QoS*

precisa ter acesso a uma lista de conectores com os quais poderá viabilizar a imposição do contrato. É necessário que estas informações de meta-nível também estejam disponíveis. Assim, suponhamos que o usuário selecione para seu perfil de QoS: (i) a criptografia com padrão *des* (*Data Encryption Standard*) e (ii) controle de erro com *CRC* (*Cyclical Redundance Check*), observando-se que a comunicação está sendo realizada, neste momento, com outro *telefone 3-em-1*. O *Configurador de QoS* pode reconfigurar a aplicação com uma composição adequada de conectores, para atender ao contrato de QoS solicitado segundo o *script* da listagem 8.24. Assim, no comando *link* (linha 2), que liga o fluxo de saída do *thePhone* ao *Display* do *officePhone*, é utilizada uma composição do conector *DES* (para criptografia) e o conector *CRC*, no lado do *thePhone*, e uma composição dos conectores *CRC* e *DES* (para decriptografia), no lado do *officePhone*. O fluxo no outro sentido tem configuração semelhante. O conector *LW* continua sendo empregado para viabilizar a comunicação nos dois sentidos.

```
1  block thePhone
2  link thePhone.Send to officePhone.Display
3                     by DES(crypt) > CRC > LW > CRC > DES(decrypt);
4  link officePhone.Send to thePhone.Display
5                     by DES(crypt) > CRC > LW > CRC > DES(decrypt);
6  resume thePhone
```

Listagem 8.24 - Reconfiguração para introduzir QoS

O protótipo desenvolvido para o *telefone 3-em-1* possui funcionalidade básica semelhante ao utilitário *talk* do Unix, ou seja, cada interlocutor pode enviar e receber textos de forma assíncrona. A antena celular, o outro telefone 3-em-1 e a estação base de telefonia convencional funcionam de forma semelhante, recebendo e transmitindo caracteres. O protótipo incorpora um painel de visualização, onde são apresentadas as regiões de visibilidade de cada um dos três interlocutores e um ícone identificando o telefone 3-em-1.

Estando em modo automático de operação, à medida que o telefone 3-em-1 se movimenta (isto é simulado, arrastando-se o ícone com o mouse), o *Monitor* detecta a entrada e saída de regiões diferentes, e os conectores de comunicação são reconfigurados pelo módulo *Configurador de QoS* (em nosso exemplo, o *Configurador de QoS* está programado para selecionar o conector adequado a cada situação de operação, mas não realiza propriamente uma composição de conectores). Quando tal evento ocorre, a comunicação se estabelece com o interlocutor correspondente. Para

cada conector de comunicação, foram simuladas características diferentes. Assim, a comunicação com a antena celular é feita por *sockets*, e a comunicação com outro *telefone 3-em-1*, via RMI. Quando o usuário muda o modo de operação para manual, ele pode indicar através de botões qual das interfaces deve ser utilizada. Cada conector também simula um esquema de tarifação, segundo o enunciado do exemplo. Isto é feito registrando-se o número de caracteres transmitidos em um arquivo de *log*.

Observações

Os módulos de *Apresentação* e *Input* podem ser intercambiados ou sofrer acréscimos, sem a necessidade de alterar a arquitetura da aplicação. Todas as decisões de projeto estão encapsuladas em módulos e características não-funcionais são suportadas por conectores.

A aplicação pode ser enriquecida, acrescentando-se a possibilidade de selecionar, automaticamente, a interface de comunicação a ser usada, por um critério de menor tarifação. Esta opção pode ser habilitada quando mais de uma interface de comunicação tiver condições de operar, ou se mais de uma operadora de telecomunicações puder oferecer o mesmo serviço. Para isso, faz-se necessário que se descreva um contrato para satisfazer o perfil do usuário do *telefone 3-em-1* e considerar as tarifas praticadas pelas operadoras e concessionárias (cada usuário terá suas preferências pessoais). Questões de tarifação tendem a se tornar cada vez mais importantes no contexto de computação ubíqua [211], pois influenciarão diretamente no modo como serviços distribuídos serão utilizados pelo usuário final.

8.6 Conclusão

O desenvolvimento dos exemplos apresentados neste capítulo permitiu comprovar a aplicação prática dos conceitos do *framework* R-RIO, tornando evidentes os benefícios de nossa abordagem. Foi possível desenvolver as aplicações por composição, de forma incremental, e demonstrar que a realização de mudanças de forma dinâmica, para adaptá-las a novas situações, é factível.

Aderindo à metodologia de configuração, sugerida em R-RIO, foi possível manter a separação de interesses na arquitetura das aplicações. Tal prática mostrou-se

importante para permitir a evolução dinâmica das mesmas. Por exemplo, no desenvolvimento dos módulos *Input* e *Presentation* do protótipo do *telefone 3-em-1*, reutilizou-se o código de uma interface gráfica de entrada e saída *teletexto*, elaborado fora do contexto de nossa tese. Isto exigiu um trabalho de limpeza no programa original, que continha rotinas de comunicação por *socket*. Assim, obteve-se um código que concentra apenas os aspectos de entrada e exibição de dados. As outras funções do *telefone 3-em-1* foram encapsuladas em outros componentes, como descrito na seção 8.5. Em conseqüência, a programação dos *scripts* de reconfiguração da arquitetura da aplicação tornou-se simples. Da mesma forma, a evolução da própria aplicação, por exemplo, para incluir a transmissão e exibição de voz ou vídeo, foi também simplificada.

A validação dos aspectos práticos do *framework* R-RIO, através dos exemplos apresentados, é parte importante de nosso trabalho. Consideramos os resultados alcançados encorajadores. Adicionalmente, por comparação, a flexibilidade obtida na configuração e na manutenção das aplicações desenvolvidas, com a utilização do *framework* R-RIO, parece-nos mais difícil de ser alcançada através de outras propostas (examinadas no capítulo 3).

Capítulo IX

Conclusão

9.1 Introdução

Este capítulo conclui a tese. Inicialmente é apresentada uma avaliação do *framework* R-RIO, segundo os requisitos discutidos no capítulo 2. Segue-se uma discussão sobre tópicos que merecem destaque no contexto de nossa proposta. Logo após, são propostas algumas idéias para a continuação deste trabalho. Finalmente, enumeramos as contribuições desta tese e apresentamos as conclusões finais.

9.2 Avaliação do *framework* R-RIO

Nesta seção, avaliamos nossa proposta, no que tange ao atendimento dos requisitos enumerados no capítulo 2: reusabilidade, separação de interesses, evolução dinâmica e abrangência.

9.2.1 Reusabilidade

De uma forma geral, em nossa proposta, aplicações podem ser rapidamente construídas e prototipadas com base na reutilização de componentes. Módulos e conectores podem ser reutilizados para a composição de elementos mais complexos. Através da composição é possível a obtenção de conectores com novas características. Módulos compostos podem encapsular soluções de *software* para determinada área de aplicação. A reusabilidade é, entretanto, dependente da aplicação de uma disciplina, por parte do projetista, que imponha a modularidade e a separação de interesses na descrição dos componentes e na concepção da arquitetura (seção 4.5 - capítulo 4). Ainda assim, pode acontecer que o reuso não seja possível em todas as situações, como será discutido nos próximos parágrafos.

Em alguns casos, a reutilização imediata de módulos e conectores pode não ser possível. Por exemplo, ao selecionar um conector para a interligação de um par de módulos, o projetista precisa conhecer como os mesmos interagem e até alguns de seu funcionamento. Dois exemplos ilustram este ponto: (i) na aplicação apresentada na seção 8.4, os módulos do jogo precisam seguir um estilo de arquitetura específico, de modo que se possa utilizar a capacidade de mediação do conector *CObserver*; (ii) por outro lado, conectores usados para comunicação em sistemas distribuídos são geralmente considerados ortogonais à aplicação. Entretanto, se observarmos os aspectos de tratamento de exceção, os interesses ficam misturados. O lado mecânico de propagação das exceções e de seu tratamento pode ser padronizado e automatizado, como feito no protótipo de R-RIO. Contudo, o tratamento completo de uma exceção eventualmente exige a intervenção do programador da aplicação, através de codificação especial, algumas vezes afetando diversos módulos da arquitetura. Isto precisa ser planejado no projeto da aplicação. Em resumo, existem limites do que se pode resolver independentemente, no meta-nível, através de conectores, limitando também a reusabilidade. Talvez, como colocado por Saltzer, o tratamento de aspectos fim-a-fim seja inerentemente um interesse dependente da aplicação [212]. Mais pesquisas são necessárias para aumentar o reuso nestes casos.

9.2.2 Separação de Interesses

Ao final dos capítulos 2 e 3, salientou-se a importância de as abordagens utilizadas para a concepção de aplicações também oferecerem mecanismos para descrever a configuração das mesmas. Considera-se, como aspectos distintos de uma configuração, a seleção de componentes e sua topologia de interligação. PM-N e BC não contribuem diretamente para facilitar a descrição da configuração de uma aplicação. Por exemplo, a utilização de MOPs em linguagens reflexivas, a programação de filtros em FC ou a descrição de aspectos em AOP, devem ser cuidadosamente planejadas no nível da programação, para espelhá-la. Embora tais propostas valorizem a separação de interesses, elas não facilitam diretamente a concepção de grandes aplicações, pois, de forma geral, a topologia das mesmas fica escondida no próprio código.

Sistemas mais complexos, que utilizam reflexão como técnica de suporte, devem ser planejados visando-se uma meta-arquitetura, 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 adaptada a sistemas de domínios diferentes, e, se for simples a integração deste conjunto com a aplicação, a utilização de reflexão será vantajosa. A proposta de R-RIO segue, de certa forma, esta última idéia: o sistema de suporte ao modelo de configuração, em conjunto com o conector, pode ser considerado uma arquitetura de meta-nível. Em adição, os comandos de configuração, por exemplo, para instanciar módulos, ou para selecionar um dado conector para interligar módulos, atuam sobre a topologia e o comportamento da aplicação em um nível diferente da funcionalidade da aplicação.

Embora o modelo de componentes de R-RIO não tenha relação direta com o modelo de objetos adotado em FC, as duas abordagens têm em comum o uso de composição para descrever a topologia de uma aplicação. Na linguagem Sina (seção 3.2.3), por exemplo, é possível indicar explicitamente, para cada classe, objetos com os quais existe relacionamento. Em conjunto com o esquema de filtros, é possível implementar, ainda, outros tipos de relacionamento. Assim, as aplicações são compostas por instâncias de objetos que interagem segundo as indicações de relacionamento programadas no código das classes, intermediadas por filtros. Podemos comparar a combinação de filtros envolvidos na comunicação entre dois objetos, em FC, com o conector de R-RIO. 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 FC e a nossa proposta é o fato de que, em R-RIO, o relacionamento e o estilo de interação entre módulos não são indicados na programação dos módulos. De acordo com o conceito de portas, apenas a assinatura e o sentido básico das requisições (de entrada ou saída) têm que ser conhecidos. Aspectos específicos da interação entre módulos são definidos nos conectores, adicionados externamente aos módulos, independentemente de considerações no nível da programação.

A separação de interesses permite que aspectos funcionais e não-funcionais sejam isolados no nível da programação interna dos componentes. Aspectos funcionais são encapsulados, preferencialmente, nos módulos, e aspectos não-funcionais, nos conectores. Em princípio, aspectos não-funcionais não interferem na concepção dos módulos. Estilos diferentes de interação dos componentes são planejados e implantados,

separadamente, nos conectores. Entretanto, aspectos funcionais e não-funcionais fazem parte de uma estrutura maior que é a aplicação. Na concepção de CBabel, 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 3.4, no capítulo 3, em que na programação de guardas é necessária uma referência ao método sob sua supervisão). Assim, em CBabel é possível a descrição de aspectos funcionais e não-funcionais, mantendo-se a visão da configuração da aplicação como um todo.

9.2.3 Evolução Dinâmica

Em nossa proposta, uma aplicação pode ser adaptada para atender a novos requisitos, tanto no nível de configuração, quanto no nível de operação. No nível da configuração, novas versões de uma aplicação podem ser configuradas adaptando-se, seletivamente, partes da versão original descritas em CBabel. No nível da operação, o modelo de configuração do *framework* facilita a evolução dinâmica das aplicações, pela possibilidade de seleção e substituição de componentes funcionais e não-funcionais e, também, pela adaptação da topologia da arquitetura durante sua execução.

Algumas características de nossa proposta tornam a evolução dinâmica factível: (i) o mapeamento entre os elementos do modelo de componentes e uma implementação permite que o encapsulamento de cada elemento seja preservado durante a execução; (ii) a API de configuração faculta a realização de manobras de reconfiguração em uma aplicação, como a substituição de componentes ou alterações nas ligações entre os módulos; (iii) aliado a isso, a possibilidade de verificações da configuração da arquitetura sendo manipulada, com informações obtidas através da API de reflexão arquitetural, torna possível que estas manobras sejam feitas com segurança. Como ressaltado na seção 4.6 - capítulo 4, tais caraterísticas não estão presentes, em conjunto, em outras propostas.

9.2.4 Abrangência

Durante o desenvolvimento de nosso trabalho, houve preocupação constante com a flexibilidade e abrangência. O *framework* R-RIO não poderia limitar a concepção das aplicações com relação ao domínio, linguagens de programação ou ambiente de execução.

Um dos problemas fundamentais na utilização das propostas avaliadas no capítulo 3 está relacionado com a abrangência, em relação ao ambiente de execução. Por exemplo, em MetaJava (seção 3.2.1.2), o acesso ao meta-nível é possível através de uma extensão especial do interpretador Java e uma biblioteca de classes. Esta infra-estrutura permite a introdução de código para interceptar vários eventos durante a execução das classes funcionais da aplicação, reificando-as para classes de meta-nível. Esta solução, entretanto, restringe o desenvolvimento da aplicação à utilização da linguagem Java e à execução no 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.

A abordagem utilizada em AOP também impõe restrições ao seu uso. As linguagens e o *weaver* examinados na avaliação (seção 3.2.3.2) 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, potencialmente, implantada em qualquer ambiente de suporte, não se considerou sua utilização para integrar partes de uma aplicação, concebidas através linguagens de programação diferentes. O *weaver* entrelaça código-fonte e gera código-fonte.

As tecnologias BC e AS/PC permitem integrar componentes (módulos) desenvolvidos virtualmente em qualquer linguagem de programação, desde que a interface deste módulo seja bem definida. Isto se torna possível, se um suporte (um *middleware*, por exemplo) adequado estiver disponível. Em R-RIO, define-se uma receita para o mapeamento das descrições da arquitetura em CBabel, em elementos de uma linguagem de programação. Esta integração é completada pelos conectores, que também podem ser construídos a partir de diversas tecnologias, inclusive *middlewares* como RMI e CORBA. 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. Por exemplo, aspectos como distribuição e comunicação estão disponíveis ou podem ser implantados em todas

elas. Entretanto, aspectos de coordenação não estão disponíveis nas outras propostas avaliadas, baseadas em AS/PC ou BC. Em CBabel, através de uma sintaxe conveniente, disponibilizamos a descrição de um conjunto abrangente de aspectos não-funcionais, como a interação, distribuição, coordenação e QoS, o que permite a configuração mais completa da arquitetura de *software* de uma aplicação.

O *framework* R-RIO é flexível quanto ao domínio de aplicação e ao tipo de componentes funcionais de *software* que podem ser integrados a uma arquitetura. A facilidade de composição de elementos permite a combinação de funcionalidades para atender aos diversos requisitos das aplicações. Em princípio, também não existem restrições quanto aos recursos demandados ao sistema operacional hospedeiro. Por exemplo, para aplicações em execução sobre a Internet, utilizando WWW e tecnologias como *applets* e *servlets*, será necessário dispor de um suporte à configuração, seguindo o modelo apresentado na seção 4.3 - capítulo 4. A adoção do conceito de conectores neste contexto também é factível. Por exemplo, uma técnica para interposição de código, antes da invocação de um método de um *servlet*, é possível através de filtros [213]. Isto seria suficiente para implementar uma grande variedade de conectores.

O uso de conectores, da mesma forma, não impõe limitações com relação aos estilos de interação entre os módulos. Ao contrário, o conector é o lugar adequado para encapsular protocolos de interação diversificados, tais como o canal de eventos (permitindo a concepção de aplicações baseadas em eventos) e *push-pull* (no estilo usado em aplicações modernas sobre o WWW), além de estilos de interação mais triviais, como troca direta de mensagens, caixa-postal, ou simples chamadas a procedimentos e métodos. Isto foi demonstrado através dos exemplos do capítulo 8.

9.3 Tópicos de discussão

Discutem-se, nesta seção, alguns tópicos, decorrentes das investigações realizadas durante a elaboração do *framework* R-RIO, e dos resultados da validação da tese.

9.3.1 Composição de Interesses

A composição de interesses / aspectos (*composition of concerns*) de uma aplicação é um assunto ainda aberto, identificado por vários pesquisadores, ([31, 69,

100, 178], por exemplo). No contexto de nosso trabalho, identificamos dois problemas principais: (i) a transferência de controle entre os elementos que gerenciam os diversos aspectos não-funcionais e (ii) efeitos colaterais de um aspecto em outro.

Para PM-N, ainda precisam ser identificados mecanismos sintáticos, para estruturar arquiteturas de meta-nível, e regras semânticas bem definidas, para transferir o controle entre componentes de meta-nível. Além disso, técnicas de apoio à composição e verificação de arquiteturas de meta-nível ainda devem ser desenvolvidas. AS/PC e ADLs, por sua vez, já oferecem conceitos e mecanismos claros para composição de módulos e conectores. Adicionalmente, é possível fazer uso das técnicas e formalizmos propostas para a verificação de propriedades de arquiteturas de *software*, como [72, 179, 130]. A integração das tecnologias AS/PC e PM-N, em nossa proposta, oferece uma direção para solucionar o problema da composição de interesses. Resultados iniciais neste sentido estão disponíveis em [145].

Na composição de conectores, é importante identificar como os conectores elementares, da arquitetura de meta-nível, obtêm o controle sobre um módulo de nível-base para, por exemplo, acessar e modificar o seu estado. Quando não existe concorrência interna, tanto no nível-base, quanto no meta, existem soluções que são ortogonais e podem ser implementadas facilmente em uma ambiente de suporte. Por exemplo, em aplicações com *checkpoint* de estado, métodos especiais (supridos pelo nível-base) podem ser usados para obter e atualizar o estado da aplicação que precisa ser preservado. Em arquiteturas de *software* descritas em CBabel, pressupõe-se que uma porta especial de acesso estará disponível nos módulos e conectores, quando variáveis representando estado são declaradas nestes componentes. Entretanto, na presença de concorrência e sincronização, em qualquer nível, necessita-se de mecanismos mais refinados.

Em alguns casos é possível identificar, *a priori*, um estilo particular de interação entre os níveis base e meta, como acontece em *frameworks* de suporte a sistemas de transações distribuídas. Soluções de propósito geral, baseadas em Java, são apresentadas em [56 e 69]. No entanto, nestes exemplos, as relações entre estes níveis, ou são restritas, ou são apenas possíveis através de modificações no ambiente básico de operação (Máquina Virtual Java, neste caso), prejudicando a portabilidade. Em R-RIO, a interação entre módulos poderia ser controlada por um sistema de transações. O

suporte para este aspecto seria provido por conectores, definindo uma arquitetura de meta-nível específica. O chaveamento entre nível-base e meta-nível ocorreria através de portas, seguindo o modelo de componentes de nossa proposta. Adicionalmente, o modelo de transações poderia ser incluído como parte do aspecto de interação, definido no *framework* R-RIO, e a sintaxe de CBabel seria adaptada, para descrever interações entre módulos segundo este modelo.

9.3.2 Arquiteturas de *Software* e *Design Patterns*

Nos últimos anos, tem se tentado encarar o *software*, sob o ponto de vista de engenharia, como o *hardware*, onde devem existir padrões, peças de reposição e peças compatíveis de fabricantes diferentes. O objetivo é produzir *software* com soluções já testadas e partes reutilizáveis, de preferencia "retiradas da prateleira", adaptando-se o que seja necessário e "inventando" somente o imprescindível.

O conceito de componentes foi adaptado ao desenvolvimento de *software* para facilitar o reuso de partes e a composição de aplicações, a partir destas partes. *Middlewares* foram aperfeiçoados para permitirem que os componentes se "encaixassem" mais facilmente. Em um passo seguinte, o conceito de *framework*, *software* contendo uma solução inteira para um domínio de aplicação, começou a ser empregado na produção de aplicações.

Recentemente, *design patterns* têm sido propostos, a fim de facilitar a produção de *software*, reutilizando a solução intelectual de problemas recorrentes de programação em um dado contexto.

A aplicação do conceito de *desgin patterns* a arquiteturas de *software* e sistemas distribuídos, vem sendo experimentada e discutida recentemente [73, 214, 215]. Neste contexto, um *design pattern* para *arquiteturas de software* "descreve um problema de projeto particular, recorrente, que surge em contextos específicos, e apresenta um esquema genérico e bem-aceito para sua solução. O esquema da solução é especificado, descrevendo-se seus componentes, suas responsabilidades e relacionamentos, além de seus estilos de colaboração".

Em nosso entender, no contexto de AS/PC, *design patterns* podem ser utilizados tanto na programação dos componentes, como na infra-estrutura que os interliga. Em R-RIO, empregaram-se *design patterns* em várias partes do *framework*, a partir de um

processo natural de desenvolvimento, apenas procurando-se adotar soluções comprovadamente úteis. Esta visão é compartilhada por outros pesquisadores como [127, 216, 217].

No suporte de R-RIO podemos encontrar os *patterns proxy*, *serializer,* e uma versão adaptada dos *patterns factory* e *acceptor-connector*. A implementação do modelo de configuração, por sua vez, utiliza os *patterns strategy* e *activator* [214, 215] nas rotinas da API de configuração e reconfiguração.

Em outro contexto, considerando-se uma arquitetura de *software*, os conectores de R-RIO podem encapsular *design patterns* de interação, separando este interesse dos componentes funcionais (ver exemplo do Jogo da Velha, seção 8.4.2, utilizando o *pattern Observer*).

Outra aplicação de *design pattern* no contexto de R-RIO, proposta em um desdobramento de nossa tese, destaca que os elementos do *framework* são recorrentes, e que, em conjunto, formam um *design pattern*. Com isso, passamos a ter uma receita que facilita a implementação do suporte para configuração em plataformas operacionais diferentes, e o mapeamento das descrições de arquiteturas em CBabel para várias linguagens. Em [147] este *design pattern* foi desenvolvido e batizado de *Architecture Configurator*.

9.3.3 R-RIO como *middleware* reflexivo

Em [218] argumenta-se que o conceito de *middleware* reflexivo (*reflective middleware*), e o próprio conceito de reflexão, ainda não têm um consenso estabelecido entre todos os pesquisadores da área. Existem definições diferentes e algumas complementares, tentando expressar coisas semelhantes. Ainda não existem padrões para a especificação de políticas para o uso da reflexão. Questões como "quem, o que, quando e por que" usar reflexão e adaptar as aplicações ainda estão abertas. Geof Coulson, editor da revista eletrônica *Distributed Systems On-Line* do IEEE, propôs uma definição de *middleware* reflexivo, a partir da compilação dessas discussões: "um *middleware* que disponibiliza a inspeção e adaptação de seu comportamento através de uma auto-representação conectada por uma relação de causa (*causally connected self representation*) apropriada". Partindo desta definição, são colocadas as seguintes perguntas: "Por que o *middleware* deveria ser reflexivo e quais são os benefícios

potenciais disso?" [219] A resposta mais direta (discutida em nossa tese) seria: "a reflexão pode fazer pelo *middleware* o que ela faz por qualquer sistema: facilita a adaptação ao ambiente de execução e torna-o capaz de reagir a mudanças".

Nossa proposta responde a tais questões ao integrar mecanismos de reflexão com a tecnologia AS/PC, na qual questões de adaptação dinâmica são resolvidas. No *framework* R-RIO define-se o modo pelo qual políticas para adaptação de uma arquitetura de *software* são configuradas, o tipo de elementos que têm o papel de impor estas políticas e a maneira como a reflexão é empregada neste contexto (seção 5.5 - capítulo 5). O suporte de R-RIO é, portanto, um *middleware* reflexivo, que integra tanto os recursos de consultas sobre a arquitetura das aplicações, tipicamente encontrado em ambientes com reflexão, como a capacidade de reconfiguração das mesmas, normalmente disponível em sistemas de suporte à configuração.

9.3.4 Questões de Desempenho

Há considerações a se fazer com relação a determinadas aplicações que são bem modeladas em R-RIO, mas podem ter restrições de desempenho. Algumas classes de aplicação requerem garantias de um desempenho mínimo, principalmente no que diz respeito a tempo de resposta. Embora cientes da importância do desempenho, não foi objetivo desta tese abordar o problema.

Dois pontos diretamente ligados a R-RIO (não associados somente a uma máquina virtual, sistema operacional ou estrutura de rede, por exemplo) mereceram atenção:

- o custo computacional introduzido pelo uso de conectores e
- o custo computacional de um procedimento típico de reconfiguração.

O custo computacional dos conectores foi avaliado no protótipo. Nesta avaliação, mediu-se o desempenho da interação entre dois módulos, intermediada por um conector. No caminho crítico de um conector simples (que apenas encaminha as requisições e resultados), encontram-se dois níveis de indireção, onde são necessárias (i) duas consultas ao repositório de informações de meta-nível, (ii) rotinas de serialização e desserialização de informações e (iii) invocações dinâmicas de métodos. Observou-se que estas rotinas são responsáveis por mais de 90% do custo

computacional da uma interação. Os resultados desta avaliação podem ser consultados em [203].

O custo direto dos procedimentos de reconfiguração, no protótipo de R-RIO, não foi avaliado especificamente. Em medidas realizadas em sistemas similares como, por exemplo, o sistema 2K, o custo de uma reconfiguração típica foi avaliado [94]. Constatou-se que os procedimentos de reconfiguração não comprometeram o desempenho esperado da aplicação. 2K utiliza um esquema de *wrapping*, para permitir a reconfiguração de componentes, e o serviço de nomes de CORBA, para manter as referências dos componentes. Embora R-RIO difira conceitualmente de 2K em vários aspectos, o esforço computacional envolvido em uma reconfiguração típica pode ser considerado equivalente nos dois sistemas. Por exemplo, as consultas ao serviço de nomes de CORBA (um repositório de meta-nível) e a execução de métodos de reconfiguração pelo *wrapper* do componente têm o seu equivalente em R-RIO: as APIs de reflexão arquitetural e configuração.

Existem alguns pontos no protótipo que admitiriam otimização, para melhorar seu desempenho. Por exemplo, em [203] verificou-se que, na implementação dos conectores, (i) alguns objetos de apoio poderiam ser reutilizados ao invés de se criarem novas instancias e (ii) que seria possível otimizar o uso da classe *String* de Java. Entretanto, existe um impacto computacional inerente à flexibilidade sendo oferecida, semelhante em qualquer sistema deste tipo, que não compromete a aplicabilidade dos conceitos de nossa proposta e nem sua utilização prática. É possível, por outro lado, obter alguma melhora no desempenho das interações entre módulos, através de otimizações na arquitetura da aplicação. Por exemplo, a configuração de alguns conectores compostos pode ser otimizada, diminuindo o número de indireções e consultas ao repositório de meta-nível (ver seção 9.4.2).

9.4 Pontos em aberto e perspectivas para o futuro

Alguns pontos, ligados à implementação de nossa proposta, poderiam ser aprofundados em trabalhos futuros. Por exemplo, no protótipo de R-RIO, a API de reflexão arquitetural foi apenas parcialmente implementada, e os mecanismos para gerência de contratos de QoS não foram incluídos no serviço do Gerente de Configuração. Tais pontos, já identificados no texto, poderiam ser incluídos em versões

futuras do protótipo. Nas próximas subseções, são discutidas outras questões, não ligadas diretamente ao protótipo.

9.4.1 Reconfigurações seguras

A adaptação de determinada arquitetura a uma nova configuração geralmente implica na manipulação de seus componentes para bloquear novas interações, terminar módulos, incluir novos módulos ou substituir conectores. No *framework* R-RIO, uma reconfiguração é executada através de comandos da API de configuração, sendo, em alguns casos, necessário executar uma seqüência deles. Assim, é importante que se execute completamente uma reconfiguração, de forma consistente. Para garantir estas características em uma implementação, os comandos de configuração poderiam ser executados com propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade) [220, 156]. Isto significa que uma operação de reconfiguração seria executada integralmente, e os resultados seriam os esperados. Como trabalho futuro, as características mencionadas poderiam ser integradas ao protótipo de R-RIO.

9.4.2 Otimização

Um recurso que pode ser agregado futuramente à CBabel é a capacidade de otimização das arquiteturas descritas. Um compilador, ou ferramenta separada, poderia otimizar a arquitetura de uma aplicação, da mesma forma que um compilador de uma linguagem de programação o faz com o código de máquina a ser gerado. Por exemplo, é possível que seja detectada a possibilidade de se fundirem conectores ou módulos básicos. Como ilustração, a cadeia de conectores da figura 9.1(a) poderia ser fundida em um único conector (figura 9.1(b)). O conector resultante seria mais eficiente pois, no mínimo, algumas cópias de memória e chamadas a métodos estariam sendo evitadas. As regras para realizar otimizações devem ser, entretanto, inseridas na programação da ferramenta de otimização.

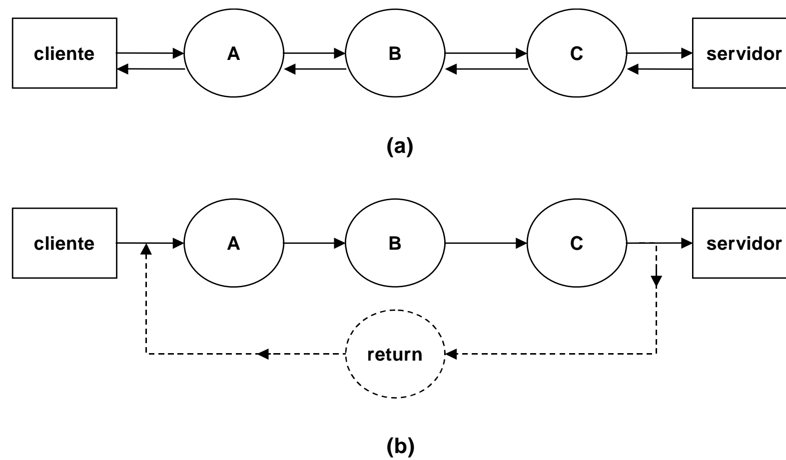


Figura 9.1 - (a) fluxo de requisições e respostas em uma cadeia de conectores; (b) otimização do fluxo de resposta

Em CBabel, as informações necessárias a algumas otimizações já são disponíveis. Por exemplo, os contratos de interação implícitos, ou declarados explicitamente, contêm informações que permitem a compactação estrutural de conectores. Adicionalmente, a partir dos contratos de coordenação, se declarados, as restrições para estas compactações podem ser inferidas.

9.4.3 *Software Architecture Description Loader*

Em um sistema operacional convencional, arquivos executáveis, ou módulos de carga, contêm código e são preparados para serem carregados na memória e executados dentro de um contexto. Esta "preparação" é usualmente feita por um compilador ou *link-editor*.

Um módulo executável geralmente é constituído de várias partes que aderem a um formato convencionado, sendo que a maioria delas contém informações de meta-nível. Apenas o segmento de código e, possivelmente, um segmento de dados são efetivamente carregados e executados. Todas as outras informações encapsuladas em um módulo executável têm o objetivo de guiar o carregador, para que a carga seja feita adequadamente. Assim, informações sobre a quantidade de memória necessária, tamanho da pilha ou a política de uso do *heap*, podem fazer parte de um módulo executável.

Argumentamos que seria útil dispor de mecanismo de carga, em um nível mais alto de abstração. Neste sentido, propomos o conceito de um carregador de descrições de arquiteturas de *software* (*Software Architecture Description Loader* - SADL).

Basicamente o SADL teria a mesma função de um carregador convencional, mas este operaria no nível da arquitetura de *software*. Ao invés de carregar arquivos executáveis, o SADL carregaria componentes da arquitetura, módulos e conectores. As informações típicas de meta-nível, relacionadas a uma arquitetura, seriam, por exemplo, o mapeamento de componentes, a localização configurada para as instâncias de módulos, contratos de interação, coordenação e QoS.

O SADL interpretaria as informações de meta-nível de uma arquitetura para tomar as providências necessárias à carga dos componentes. Uma outra característica de SADL seria a de suportar linguagens de *script*, permitindo que reconfigurações fossem tratadas de forma semelhante à carga inicial da arquitetura. Em um *script*, uma seqüência de comandos de configuração poderia ser descrita. As meta-informações necessárias à execução destes comandos seriam consultadas ao ambiente (reflexivo) de execução.

Entendemos que o conceito de SADL é uma aplicação direta do *framework* R-RIO. CBabel, por exemplo, contém os elementos necessários à descrição de arquiteturas de *software*. Adicionalmente, é possível converter uma arquitetura descrita em CBabel em uma forma neutra, para ser armazenada no repositório de meta-nível e distribuída para os nós onde os componentes devem ser instanciados. Em cada nó, uma instância do suporte SADL seria integrada ao suporte de R-RIO. O suporte SADL deve ser capaz de gerenciar a configuração local de uma arquitetura, utilizando vários padrões de módulos de carga [32] (módulos objeto, *shared objects*, DLLs, componentes, módulos executáveis, etc.).

Pontos a serem considerados para implantar o conceito de SADL:

- padronização da forma de armazenar e transportar informações arquiteturais de meta-nível. Em [221] propõe-se uma representação padronizada, em XML, para os conceitos empregados por diversas ADLs. Esta proposta poderia ser tomada como base para representar as informações de meta-nível necessárias a SADL.
- implantação da infra-estrutura de configuração do *framework* R-RIO (CBabel e Gerente de Configuração) sobre outros ambientes de execução, além de Java (ver discussão na seção 9.4.4). Observa-se que o *design pattern Architecture Configurator*, apresentado em [147], pode ser usado como base para as implementações desta infra-estrutura.

9.4.4 Integração com outros ambientes

Durante a elaboração do protótipo de R-RIO, foram investigadas algumas possibilidades para permitir a interação de módulos executando em ambientes heterogêneos. Foram desenvolvidos conectores usando-se a infra-estrutura de comunicação de três implementações diferentes de CORBA: o pacote *org.omg*, incluído no kit de desenvolvimento Java (JDK), JacORB, uma implementação experimental desenvolvida em Java [222], e TAO, apresentado na seção 6.3, desenvolvido em C++. Utilizando-se estes conectores, tornou-se viável fazer um módulo R-RIO interagir com objetos desenvolvidos em Java e C++, que executavam sem o suporte da infra-estrutura R-RIO. Também foi possível acessar diretamente recursos do sistema operacional, através do conector desenvolvido em TAO. Nesta última experiência, empregou-se o suporte de JNI - *Java Native Interface*, para realizar a invocação de métodos C++ a partir de uma classe Java, e vice-versa. Comprovou-se, assim, que o conector pode funcionar como uma ponte de *software* para permitir a comunicação de módulos que executam com o suporte de ambientes diferentes.

Como pesquisa futura, caberia o desenvolvimento de um conector genérico, o qual faria a tradução de requisições partindo de um módulo para uma representação neutra, (como a utilizada em XML-RPC [223]). O suporte poderia empregar a técnica de reflexão por contexto (ver seções 4.4.1 e 7.3.1), com o objetivo de identificar o ambiente de execução do módulo e realizar a conversão dinamicamente. As informações necessárias para implementar este mecanismo estão disponíveis no repositório de meta-nível, obtidas da cláusula *map* (ver apêndice A) em CBabel.

A execução de módulos sobre ambientes heterogêneos requer que a infra-estrutura que implementa o modelo de configuração de R-RIO seja integrada a diversos sistemas de suporte. Por exemplo, a implantação do *framework* R-RIO sobre CORBA mostrou-se factível [98]. Neste contexto, é possível utilizar os serviços de nome e repositório de interfaces, disponíveis em CORBA, para facilitar a implementação do repositório de informações de meta-nível, e das APIs de configuração e reflexão arquitetural. A utilização de outros *middlewares*, como Jini, também merece ser pesquisada.

Uma outra linha a ser seguida é a investigação da implantação dos conceitos de R-RIO sobre um ambiente Web. Isto pode ser feito utilizando-se, por exemplo, uma

combinação de mecanismos como *applets*/*servlets* para a implementação dos módulos. A implementação de conectores poderia ser feita encapsulando-se o mecanismo de comunicação entre *applet* e *servlet*, via HTTP, ou outros mecanismos como RMI e *sockets*.

A implantação de R-RIO, diretamente sobre o próprio sistema operacional, também pode ser investigada. Em [82] utilizou-se uma biblioteca de *threads* e a API do serviço de carga do Unix, para implementar-se um serviço similar ao Gerente de Configuração de R-RIO. Este tipo de implementação pode facilitar o acesso aos recursos da máquina, embora existam os meios de fazê-lo nas outras alternativas.

Vale reforçar a observação do final da seção 9.4.3 que, para tornar possível a integração entre sistemas de suporte, módulos e conectores (construídos em ambientes heterogêneos) e as arquiteturas descritas em CBabel, é necessário prover-se uma forma neutra para representar as informações de meta-nível.

9.4.5 Integração de ferramentas para verificação

Em nossa opinião, além de facilitar o entendimento sobre os conceitos do *framework* R-RIO, aspectos formais podem ser aplicados de forma prática.

No apêndice B, apresenta-se a proposta de um formalismo para o modelo de componentes do *framework* R-RIO. Fez-se uso de Redes de Petri para modelar os componentes de uma arquitetura. Além dos módulos, foi possível capturar, nesta proposta, a abstração de uma porta e a abordagem de conectores como elementos de meta-nível. A partir da descrição de uma arquitetura de *software* em CBabel, é possível obter um modelo em Redes de Petri, ou derivar outras representações baseadas em um *sistema de estados e transições rotuladas*. A partir destes modelos, poderiam ser utilizadas ferramentas, como a desenvolvida em [224], para verificação de características quais a ausência de *deadlock* e *livelock*, ou a verificação da consistência geral de uma arquitetura, antes de se avançar para a etapa de implementação.

No modelo de componentes de R-RIO, optou-se por separar o conceito de módulos e conectores porque (i) o suporte à configuração pode oferecer um tratamento diferenciado para os conectores e, (ii) algumas otimizações específicas podem ser feitas sobre os mesmos (seção 9.4.2). Os contratos de aspectos não-funcionais descritos em CBabel, associados aos conectores, também podem ser modelados formalmente. Isto

permite verificar se os aspectos não-funcionais de uma arquitetura estão descritos de forma consistente.

Como proposta para trabalhos futuros, algumas ferramentas para verificação dos aspectos funcionais e não-funcionais, descritos em arquiteturas de *software*, poderiam ser implementadas e integradas a um compilador para CBabel. Algumas das referidas verificações poderiam ser realizadas, também, durante a operação de uma aplicação, a partir das informações armazenadas no repositório de meta-nível. Isto seria importante para a manutenção da consistência da arquitetura de uma aplicação, quando esta sofresse uma reconfiguração.

9.5 Principais contribuições desta tese

Nossa tese apresenta as seguintes contribuições:

- uma discussão clara sobre os benefícios e problemas das tecnologias BC, PM-N e AS/PC. Com base nesta discussão, observamos que estas tecnologias poderiam ser integradas, de forma a se obterem vantagens somadas na concepção de aplicações, tais como: separação de interesses, modularidade, evolução dinâmica, flexibilidade e abrangência;
- a identificação de uma correspondência entre os conceitos de reflexão e o modelo de componentes de R-RIO. Esta correspondência levou-nos a propor, com segurança, as APIs de reflexão arquitetural e configuração. A partir desta investigação, também foi proposta a utilização de reflexão computacional para *customizar* as interações entre módulos e, em outro nível, a reflexão por contexto para a adaptação automática de conectores às interfaces dos módulos;
- a proposta do *framework* R-RIO, que permite a configuração da arquitetura de *software* de aplicações de forma flexível. Este *framework* ajuda a manutenção da separação de interesses durante todo o ciclo de vida destas aplicações, inclusive na gerência de sua operação. Como conseqüência, os componentes de uma arquitetura são mais facilmente reusáveis;
- a descrição de aplicações, segundo o modelo de componentes e a metodologia sugerida para a configuração destas aplicações no *framework* R-RIO, produz arquiteturas de *software* que podem evoluir dinamicamente, em face de novas

demandas dos usuários ou de novas condições de operação;

- em adição a verificações normalmente realizadas sobre descrições de aspectos funcionais em ADLs, a descrição de aspectos não-funcionais de uma arquitetura de *software*, como proposto em R-RIO, também permite a verificação de propriedades de tais aspectos;
- as experiências para validação da tese comprovaram os benefícios da abordagem do *framework* R-RIO.

As contribuições e resultados parciais obtidos ao longo das pesquisas, foram apresentados em [93, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234 e 235].

9.6 Conclusão

R-RIO, o principal resultado desta tese, integra as tecnologias AS/PC, PM-N e BC em um único *framework*, conceitual e prático, para concepção e gerência de aplicações. Inicialmente, mostrou-se que a tecnologia de Programação por Configuração, baseada em ADL, oferece benefícios diretos no desenvolvimento e manutenção de sistemas. A definição de uma arquitetura de *software*, neste contexto, encaixa-se naturalmente no ciclo-de-vida de tais sistemas, facilitando as atividades relacionadas ao desenvolvimento do *software*. Isto nos levou a experimentar a integração da parafernália de linguagens, ferramentas e mecanismos necessários à prática de engenharia de *software*, em um ambiente centralizado em uma ADL, para obter-se um *framework* produtivo.

A descrição de um sistema através de uma ADL é, por si só, uma especificação, revestida de características formais, podendo-se valer de técnicas disponíveis para refinar e provar propriedades das arquiteturas de *software*. Além disso, de acordo com o proposto na tese, uma ADL também pode ser usada como base para expressar interesses específicos, durante o desenvolvimento de aplicações de domínios diferentes. Por exemplo, CBabel foi projetada objetivando-se a inclusão de contratos para a descrição de aspectos não-funcionais, tais como a coordenação e QoS. Em conseqüência, surgem duas novas características complementares e interessantes em nossa proposta: (i) verificar, antecipadamente, se módulos funcionalmente compatíveis poderão obedecer a um contrato (de QoS, por exemplo); (ii) montar conectores compostos, que podem

incluir a configuração de suporte específico para atender aos requisitos de um contrato. Esta última característica ajuda a resolver dependências de recursos de uma aplicação. A verificação de contratos e a reserva de recursos poderiam ser incluídas na etapa de configuração e carga da arquitetura, de forma automática ou com o mínimo de intervenção humana. Adicionalmente, o uso explícito de conectores para interações entre módulos expõe as informações necessárias para guiar o reuso de *software*.

Identificou-se uma correspondência imediata entre os conceitos fundamentais das tecnologias AS/PC e PM-N. Alguns dos problemas principais na implantação destas tecnologias em sistemas reais também são bastante similares. Componentes de meta-nível podem ser mapeados em conectores, e a adaptação da aplicação pode ser obtida pela seleção destes conectores, para compor as arquiteturas através de PC. A capacidade de configuração de módulos e conectores simplifica a composição e evolução da arquitetura de *software*, através da execução de mudanças planejadas e não-planejadas. Em particular, a disponibilidade dos conectores reflexivos por contexto, também propostos durante a concepção de R-RIO, permite a configuração de arquiteturas altamente dinâmicas comparáveis às que se encontram no ambiente da Internet.

Em um nível abstrato, uma descrição de arquitetura de *software* é ortogonal à implementação. Isto permite que a implementação de componentes seja associada à arquitetura em um estágio separado. Em princípio, isto facilita o mapeamento de componentes arquiteturais para linguagens de programação com paradigmas diferentes e para qualquer ambiente de execução.

De acordo com esta visão, a adoção dos conceitos de AS/PC pode facilitar o uso de diversas abordagens de PM-N, como a reflexão computacional e estrutural, disponibilizando mecanismos simples para especificar componentes e arquiteturas de meta-nível, através de ADLs, utilizando as técnicas de PC. Esta é uma das principais motivações para integrar estas tecnologias.

Pela comparação do mesmo conjunto de exemplos, desenvolvidos a partir do *framework* proposto, e com protótipos de outras propostas, verificou-se que R-RIO possui expressividade equivalente e pode ser considerado mais flexível. Adicionalmente, a abordagem utilizada em nosso *framework* permite um tratamento integrado do gerenciamento das aplicações desenvolvidas, desde a análise até a execução e manutenção, diferentemente das outras propostas avaliadas, onde isso é feito

de maneira empírica. Por exemplo, nosso modelo de configuração admite que o mesmo processo utilizado para conceber a arquitetura de uma aplicação seja empregado consistentemente na sua reestruturação, em resposta a novas demandas, mesmo depois desta ser implantada e estar em produção. Tais características formam uma base para o desenvolvimento de aplicações que precisam evoluir dinamicamente.

Referências Bibliográficas

[1] DEMERS, A., *et al.*, "Research Issues in Ubiquitous Computing", *In: Proceedings of the 13th annual ACM symposium on Principles of distributed computing (PODC)*, pp. 2-8, Los Angeles, CA, EUA, Agosto, 1994.

[2] ABOWD, G. D., MYNATT, E. D., "Charting past, present and future research in ubiquitous computing", *ACM Transactions on Computer-Human Interaction*, Vol. 7, No. 1, pp. 29-58, Março, 2000.

[3] WEGNER, P., "Dimensions of Object-Oriented Modeling", *IEEE Computer*, pp. 12-20, Outubro, 1992.

[4] POUNTAIN, D., SZYPERSKI, C., "Extensible Software Systems", *Byte Magazine*, pp. 57-62, Maio, 1994.

[5] LADDAGA, R., VEITCH, J., "Dynamic Object Technology", *Communications of the ACM*, Vol. 40, No. 5, pp. 37-38, Maio, 1997.

[6] WEGNER, P., "Interoperability", *ACM Computing Surveys*, Vol. 28, No. 1, pp. 285-287, Março, 1996.

[7] KICZALES, G., LAMPING, J., LOPES, C. V., *et al.*, "Open Implementations Design Guidelines", *In: Proceedings of the ACM 19th International Conference on Software Engineering*, 1997.

[8] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, Julho, 1996.

[9] CHIBA, S., "A Metaobject Protocol for C++", *In: ACM OOPSLA'95 - 10th Conference on Object-Oriented Programming Systems, Languages and Applications*, SIGPLAN Notices, Vol. 30, No. 10, pp. 285-299, Austin, Texas, EUA, Outubro, 1995.

[10] KICZALES, G., RIVIÈRES, J., BOBROW, D. G., *The Art of the Metaobject Protocol*, MIT Press, 1991.

[11] ZIMMERMANN, C., CAHILL, V., "It's Your Choice - On the Design and Implementation of a Flexible Metalevel Architecture", *In: Proceedings of the IEEE 3rd International Conference on Configurable Distributed Systems*, Maryland, EUA, Maio, 1996.

[12] LOPES, C. I. V., *D: A Language Framework for Distributed Programming*, Tese de D.Sc., College of Computer Science, Northeastern University, EUA, Novembro, 1997.

[13] BERGMANS, M. J. L., *Composing Concurrent Objects - Applying Compositional-Filters for the Development and Reuse of Concurrent Object-Oriented Programs*, Tese de D.Sc., Universiteit Twente, Holanda, Junho, 1996.

[14] KRAMER, J., "Configuration Programming - A Framework for Development of Distributed Systems", *In: Proceedings of the IEEE International Conference on Computer Systems and Software Engineering*, Tel Aviv, Israel, Maio, 1990.

[15] BISHOP, J., FARIA, R., "Connectors in Configuration Programming Languages: are They Necessary?", *IEEE 3rd International Conference on Configurable Distributed Systems*, Annapolis, Maryland, Maio, 1996.

[16] SHAW, M., DELINE, R., KLEIN, D., 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.

[17] SHAW, M., GARLAN, D. ALLEN, R., KELIN, D., *et al.*, *Candidate Model Problems in Software Architecture*, The *Software* Architecture Group, CMU, Version 1.3, Janeiro, 1995.

[18] Sun Microsystems, *JavaTM Remote Method Invocation Specification*, Revision 1.50, JDK 1.2, Outubro, 1998.

[19] BASILI, V. R., BOEHM, B., "COTS-Based Systems Top 10 List", *IEEE Computer*, Vol. 34, No. 5, pp. 91-96, Maio, 2001.

[20] STEVENS, W. R., *Unix Network Programming*, Prentice-Hall, Englewood Cliffs, NJ, EUA, 1993.

[21] JONES, M. B., "Interposition Agents: Transparently Interposing User Code at the System Interface", *In: Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Asheville, EUA, 1993.

[22] AGHA, G., WEGNER, P. ,YONEZAWA, A., *Research Directions in Concurrent Object-Oriented Programming*, The MIT Press, 1993.

[23] GUERRAQUI, R. *et al*, "Strategic Directions in Object-Oriented Programming", *ACM Computing Surveys*, Vol. 28, No. 4, pp. 691-700, Dezembro, 1996.

[24] HÖLZLE, U., "Integrating Independently-Developed Components in Object-Oriented Languages", *In: Proceedings of the ECOOP'93 - 7th European Conference on Object-Oriented Programming*, Springer-Verlag, LNCS, pp. 36-56, Kaiserslautern, Alemanha, Julho, 1993.

[25] WIRFS-BROCK, A., *Panel Designing Reusable Designs: Experiences Designing Object-Oriented Frameworks*, Sigplan Notices Special Issue OOPSLA-ECOOP'90 Addendum to the Proceedings (Jerry L. Archibald and K.C. Burgess Yakemovic eds.), pp.19-24, Canada, Outubro, 1990.

[26] VOAS, J. M., "The Challenges of Using COTS Software in Component-Based Development", *IEEE Computer*, Vol. 31, No. 8, pp. 44-45, Junho, 1998.

[27] BROWN, A. W., WALLNAU, K. C., "Engineering of Component-Based Systems", *In: Component-Based Software Engineering*, *Selected Papers from the Software Engineering Institute*, IEEE Press, 1996.

[28] RINE, D. C., "Supporting Reuse with Object Technology", *IEEE Computer*, Vol. 30, No. 10, pp. 43-45, Outubro, 1997.

[29] ATKINSON, C., *Object-Oriented Reuse, Concurrency and Distribution: na Ada-based approach*, Addison Wesley Publishing, EUA, 1991.

[30] AGHA, G., FROLUND, S. *et al.*, "Abstraction and Modularity Mechanisms for Concurrent Computing", *IEEE Parallel and Distributed Technology, Systems and Applications*, Maio, 1993.

[31] AKSIT, M., "Composition and Separation of Concerns in the Object-Oriented Model", *ACM Computing Surveys*, Vol. 28[A], No. 4, Dezembro, 1996.

[32] FRANZ, M., "Dynamic Linking of Software Components", *IEEE Computer*, pp. 74-81, Março, 1997.

[33] RUMBAUGH, J., *et al.*, *Object-Oriented Modeling and Design*, Prentice-Hall Inc., 1991.

[34] TAIVALSAARI, A., "On the Notion of Inheritance", *ACM Computing Surveys*, Vol. 28, No. 3, pp. 438-479, Setembro, 1996.

[35] GAMMA E., HELM R., JOHNSON R., VILISSIDES, J., *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison Wesley Publishing, EUA, 1995.

[36] JOHNSON, R. E., "Frameworks = (Components + Patterns)", *Communications of the ACM*, Vol. 40, No. 10, pp. 39-42, Outubro, 1997.

[37] FAYAD, M. E., SCHMIDT, D. C., (Guest Editors), "Object-Oriented Application Frameworks", *Communications of the ACM*, Vol. 40, No. 10, pp. 32-38, Outubro, 1997.

[38] PANCAKE, C. M., "The Promise and Cost of Object Technology: A Five-Year Forecast", *Communications of the ACM*, Vol. 38, No. 10, Outubro, 1995.

[39] HOPKINS, J., "Component Primer", *Communications of the ACM*, Vol. 43, No. 10, pp. 27-30, Outubro, 2000.

[40] UML Revision Task Force, *OMG Unified Modeling Language Specification v1.3*, document ad/99-06-08, Object Management Group, Junho, 1999.

[41] MEYER, B., "The significance of components", Beyond Objects, *Software Development Online*, Novembro, 1999.
http://www.sdmagazine.com/documents/s=752/sdm9911k/9911k.htm

[42] KRIEGER, D., ADLER, R. M., "The Emergence of Distributed Component Platforms", *IEEE Computer*, pp. 43-53, Março, 1998.

[43] LEWANDOWSKI, S. M., "Frameworks for Component-Based Client/Server Computing", *ACM Computing Surveys*, Vol. 30, No. 1, pp. 5-27, Março, 1998.

[44] PURTILO, J., "The Polylith Software Bus", *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 1, pp. 151-174, Janeiro, 1994.

[45] BROWN, N., KINDEL, C., *Distributed Component Object Model Protocol - DCOM/1.0*, Microsoft Corp., 1998.
http://www.microsoft.com/com

[46] WANG, Y-M., CHUNG, P-Y. E., "Customization of distributed systems using COM", *IEEE Concurrency*, pp. 8-12, Julho-Setembro, 1998.

[47] Sun Microsystems, *Enterprise JavaBeans™ - v.1.0*, 1999.
http://www.javasoft.com/products/ejb/docs.html

[48] BIRRELL, A., NELSON, B. J., "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 39-59, Fevereiro, 1984.

[49] Open Software Foundation, *DCE 1.2.2 Introduction to OSF DCE*, Open Group Product Documentation, F201 ISBN 1-85912-182-9, Novembro, 1997.

[50] W3 Consortium, *Extensible Markup Language (XML) 1.0 Specification*, W3C Recommendation REC-xml-19980210, Fevereiro, 1998.

http://www.w3.org/TR/REC-xml

[51] STROUDS, R., "Transparency and Reflection in Distributed Systems", *ACM Operating Systems Review*, Vol. 22, No. 2, Abril, 1993.

[52] LISBÔA, M. L., *Reflexão Computacional no Modelo de Orientação a Objetos*, 25º Jornadas Argentinas de Informatica y Investigacion Operativa, Buenos Aires, Argentina, Setembro, 1995.

[53] RUBIRA, C. M. F., *Structuring Fault-Tolerant OO Systems Using Inheritance and Delegation*, Tese de D.Sc., Dep. of Computer Science, University of Newcastle, Outubro, 1994.

[54] FABRE, J. C., PERENNOU, T., BLAIN, L., "Implementing Fault Tolerant Applications Using Reflective Object-Oriented Programming", *In: Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing*, 1995.

[55] FRAGA, J., MAZIERO, C., LUNG, L.C., LOQUES, O. G., "Implementing replicated services in open systems using a reflective approach", *In: Proceedings of the 3rd International Symposium on Autonomous Decentralized Systems*, pp. 273-280, Berlin, Alemanha, Abril, 1997.

[56] GOLM, M., *Design and Implementation of a Meta Architecture for Java*, Dissertação de M.Sc., Instituto de Matemática, Erlanngen-Nüremberg University, Alemanha, Janeiro, 1997.

[57] MITCHELL, S. E., *et al.*, "Adaptive Scheduling using Reflection", ECOOP'97 *Workshop on Reflective Real-time Object-Oriented Programming and Systems (W3)*, Jyvaskyla, Finlândia, Junho, 1997.

[58] CHIBA, S. *et al*, "Weak Protection for Reflective Operating Systems", ECOOP'97, *Workshop on Reflective Real-Time Object-Oriented Programming and Systems (W3)*, Jyvaskyla, Finlândia, Junho, 1997.

[59] GOWING, B., CAHILL, V., "Making Meta-Object Protocols Practical for Operating Systems", *In: Proceedings of the 4th International Workshop on Object Orientation in Operating Systems*, pp. 52-55, 1995.

[60] WU, Z., "Reflective Java and a Reflective Component-Based Transaction Architecture", OOPSLA'98, *Workshop on Reflective Programming in C++ and Java (W13)*, Vancouver, Canada, Outubro, 1998.

[61] SINGHAI, A., SANE, A., CAMPBELL, R., "Reflective ORBs: Supporting Robust, Time-critical Distribution", ECOOP'97, *Workshop on Reflective Real-time Object-Oriented Programming and Systems (W3)*, Jyvaskyla, Finlândia, Junho, 1997.

[62] GOWING, B., CAHILL, V., "Meta-Object Protocols for C++: The Iguana Approach", *Reflection'96*, pp. 137-152, San Francisco, EUA, Abril, 1996.

[63] KLEINÖDER, J., GOLM, M., "MetaJava: an Efficient Run-Time Meta Architecture for Java", *IEEE International Workshop on Object Orientation in Operating Systems*, Seattle, Washington, Outubro, 1996.

[64] LOQUES, O. G., LEITE, J., BOTAFOGO, R., LOBOSCO, M., "Configuração, Reflexão e CORBA: Algumas Considerações", *III Workshop do Projeto ASAP*, UFCE, Fortaleza, Ceará, 1996.

[65] CAMPO, M., PRICE, N. H., "Meta-Object Manager: A framework for Customizable Meta-Objetc Support for Smaltalk 80", *In: anais do I Simpósio Brasileiro de Linguagens de Programação*, Belo Horizonte, MG, Setembro, 1996.

[66] GOLM, M., KLEINÖDER, J., "metaXa and the Future of Reflection", OOPSLA'98, *Workshop on Reflective Programming in C++ and Java (W13)*, Vancouver, Canada, Outubro, 1998.

[67] LEE, A. H., ZACHARY, J. L., "Reflections on Metaprogramming", *IEEE Transactions on Software Engineering*, Vol. 21, No. 11, pp. 883-893, Novembro, 1995.

[68] FRAGA, J., FARINES, J., FURTADO, O., SIQUEIRA, F., "Programação de Aplicações Distribuídas Tempo-Real em Sistemas Abertos", *In: XXIII Seminário Integrado de Software e Hardware*, Recife, PE, Agosto, 1996.

[69] OLIVA A., BUZATO L., "Composition of Meta-Objects in Guaraná", OOPSLA'98, *Workshop on Reflective Programming in C++ and Java (W13)*, pp. 86-90, Vancouver, Canada, 1998.

[70] PERRY, D. E., WOLF, A. L., "Foundations for the Study of Software Architecture", *ACM SIG Software Engineering Notes*, Vol. 17, No. 4, Outubro, 1992.

[71] ALLEN, R. J., GARLAN, D., "Beyond Definition/Use: Architectural Interconnection", *IDL Workshop*, ACM SIGPLAN Notices, Vol. 29, No. 8, pp. 35-44, Agosto, 1994.

[72] ALLEN, R. J., *A Formal Approach to Software Architecture*, Tese de D.Sc., School of Computer Science, TR#CMU-CS-97-144, Carnegie Mellon University, EUA, Maio, 1997.

[73] SHAW, M., GARLAN, D., *Software architecture: perspectives on an emerging discipline*, Prentice-Hall Inc., EUA, 1996.

[74] MAGEE, J., KRAMER, J., "Dynamic Structure in Software Architectures", *In: Proceedings of the ACM SIGSOFT'96 - 4th Symposium on the Foundations of Software Engineering*, pp. 3-14, San-Francisco, California, EUA, Outubro, 1996.

[75] MEDVIDOVIC, N., TAYLOR, R. N., "A Framework for Classifying and Comparing Architecture Description Languages", *In: Proceedings of the 6th European Software Engineering Conference*, Zurich, Suiça, Setembro, 1997.

[76] CLEMENTS, P., "A Survey of Architecture Description Languages", *8th International Workshop on Software Specification and Design*, Alemanha, Março, 1996.

[77] PRYCE, N., "Component Interaction in Distributed Systems", *In: Proceedings of the IEEE 4th International Conference on Configurable Distributed Systems*, Annapolis, Maryland, EUA, Maio, 1998.

[78] ISSARNY, V., BIDAN, C., "Aster: A Framework for Sound Customization of Distributed Runtime Systems", *In: Proceedings of the 16th IEEE International Conference on Distributed Computing Systems*, pp. 586-593, Hong-Kong, Maio 1996.

[79] WERNER, J. A. V., *Metodologia e Suporte para Sistemas Distribuídos Configuráveis*, Dissertação de M.Sc., DEE/PUC-RJ, Março, 1995.

[80] DEREMER, F., KRON, H. H., "Programming-in-the-large versus programming-in-the-small", *IEEE Transactions on Software Engineering*, Vol. 2, No.2, pp. 80-86, Junho, 1976.

[81] GARLAN, D., ALLEN, R., OCKERBLOOM, J., "Exploiting Style in Architectural Design Environments", *In: Proceedings of the ACM SIGSOFT'94 - 2nd Symposium on the Foundations of Software Engineering*, pp. 175-188, New Orleans, LA, EUA, Dezembro, 1994.

[82] SZTAJNBERG, A, *Flexibilidade em Sistemas Distribuídos Configuráveis*, Dissertação de M.Sc., DEE/PUC-RJ, Março, 1995.

[83] DEAN, T. R., CORDY, J. R., "A Syntactic Theory of Software Architecture", *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, pp. 302-313, Abril, 1995.

[84] LUCENA, C. J. P., ALENCAR, P. S. C., "A Formal Description of Evolving Software Systems Architectures", *In: Anais do I Prêmio COMPAC/ Uniemp*, pp. 3-16, São Paulo, 1996.

[85] MORICONI, M., QIAN, X., RIEMENSCHNEIDER, R. A, "Correct Architecture Refinement", *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, pp. 356-372, Abril, 1995.

[86] INVERARDI, P., 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.

[87] ABOWD, G., ALLEN, R., GARLAN, D., "Using style to give meaning to software architecture", *In: Proceedings of the ACM SIGSOFT'93 - 1st International Symposium on Foundations of Software Engineering*, Software Engineering Notes, Vol. 3, No. 118, pp. 9-20, Dezembro, 1993.

[88] NITTO, E. D., ROSEMBLUM, D., "Exploiting ADLs to Specify Architectural Styles Induced by *Middleware* Infrastructures", *In: Proceedings of the 21st Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, Maio, 1999.

[89] HÜRSCH, W. L e LOPES, C. V., *Separation of Concerns*, NU-CCS-95-03, Northeastern University, Boston, EUA, 1995.

[90] ENDLER, M., WEI. J., "Programming generic dynamic reconfigurations for distributed applications", *IEE International Workshop on Configurable Distributed Systems*, pp. 68-79, Março, 1992.

[91] ISSARNY, V., BIDAN, C., "Aster: A CORBA-based Software Interconnection System Supporting Distributed System Customization", *In: Proceedings of the IEEE 3rd. International Conference on Configurable Distributed Systems*, pp. 586-593, Annapolis, EUA, Maio, 1996.

[92] CRANE, S., DULAY, N., "A Configurable Protocol Architecture for CORBA Environments", *In: ISADS'97 - International Symposium for Autonomous Decentralized Systems*, Berlin, Alemanha, Abril, 1997.

[93] SZTAJNBERG, A., LOQUES, O. G., "Reflexão Computacional e Multimídia Distribuída", *In: anais do XV Simpósio Brasileiro de Redes de Computadores*, pp. 417-431, São Carlos, SP, Maio, 1997.

[94] KON, F., "Automatic Configuration of Component-Based Distributed Systems", Tese de D.Sc., Department of Computer Science, UIUC, EUA, Maio, 2000.

[95] Jagannathan, S., Agha, G., "A Reflective Model of Inheritance", *In: Proceedings of the ECOOP'92 - 6th European Conference on Object-oriented Programming*, LNCS 615, Springer-Verlag, pp. 350-371, Utrecht, Holanda, Junho, 1992.

[96] MAGEE, J., DULAY, N., KRAMER, J., "Structuring parallel and distributed programs", *In: IEE International Workshop on Configurable Distributed Systems*, pp. 102-117, Março, 1992.

[97] KON, F., CAMPBELL, R. H., "Dependence Management in Component-Based Distribued System", *IEEE Concurrency*, pp. 26-36, Janeiro, 2000.

[98] LOQUES O. G., BOTAFOGO R., LEITE J., "A Configuration Approach for Distributed Object-Oriented System Customization", *In: Proceedings of the 3rd International IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, pp. 185-189, Newport Beach, EUA, 1997.

[99] RAVERDY, P-G., VAN GONG H., LEA, R., "DARTS: A Reflective *Middleware* for Adaptive Applications", OOPSLA 98, *Workshop on Reflective Programming in C++ and Java (W13)*, pp. 37-45, Vancouver, Canada, 1998.

[100] WELCH, I., STROUD, R., "Dalang - A Reflective Java Extension", OOPSLA'98, *Workshop on Reflective Programming in C++ and Java (W13)*, pp. 11-15, Vancouver, Canada, Outubro, 1998.

[101] WELCH, I., R. STROUD, "From Dalang to Kava - The Evolution of a Reflective Java Extension", *Reflection '99*, LNCS 1616, pp. 2-21, Springer- Verlag, 1999.

[102] MALUCELLI, V. V., *Babel - Building Applications by Evolution*, Dissertação de M.Sc., DEE/PUC-RJ, Rio de Janeiro, Brazil, 1996.

[103] MAGEE, J., DULAY, N., KRAMER, J., "Regis: A Constructive Development Environment for Parallel and Distributed Programs", *IEE/IOP/BCS Distributed Systems Journal*, Vol. 1, No. 5, pp. 304-312, Setembro, 1994.

[104] Object Management Group, *Multiple Interfaces and Composition*, RPF Revised Joint Submission, OMG TC Document orbos/97-05-17, Version 4.2.7, Junho, 1997.

[105] MAGEE, J., KRAMER, J., "Composing Distributed Objects in CORBA", ISADS'97, *3rd International Symposium on Autonomous Decentralized Systems, Berlin*, Germany, Abril, 1997.

[106] PRYCE, N., CRANE, S., "A Uniform Approach to Configuration and Communication on Distributed Systems", *In: Proceedings of the IEEE 3rd International Conference on Configurable Distributed Systems*, Annapolis, EUA, Maio, 1996.

[107] AGHA, G., "Linguistic Paradigms for Programming Complex Distributed Systems", *ACM Computing Surveys*, Vol. 28, No. 2, pp. 295-296, Junho, 1996.

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

[109] GREEN, D., *The Java Tutorial - The Reflection API*, Sun Microsystems, 1999. http://www.javasoft.com.docs/books/tutorial/reflect

[110] SZTAJNBERG, A., *Flexibilidade e Separação de Interesses para a Concepção e Evolução de Aplicações Distribuídas*, Proposta de tema de tese de D.Sc., PEE/ COPPE/UFRJ, Rio de Janeiro, Julho, 1999.

[111] CHIBA, S., "Load-Time Structural Reflection in Java", *In: ECOOP'2000 - 14th European Conference on Object-oriented Programming*, LNCS 1850, pp. 313-336, Springer-Verlag, Berlin, 2000.

[112] OSSHER, H., TARR P., *Hyper/J™: Multi-Dimensional Separation of Concerns for Java*, Technical Report RC21452(96717), IBM T. J. Watson Research Center, Abril, 1999.

[113] KELLER, R., HÖLZLE, U., "Binary Component Adaptation", *In: Proceedings of the ECOOP'98 - 12th European Conference on Object-oriented Programming*, LNCS 1445, pp. 307-329, Springer-Verlag, 1998.

[114] SCHNEIDER, J-G., *Sina and the Composition Filters Object Model*, Setembro, 1996. http://www.iam.unibe.ch/~scg/Research/ComponentModels/sina.html

[115] AKSIT, M., BERGMANS, L., VURAL, S., "An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach", *In: Proceedings of the ECOOP'92 - 6th European Conference on Object-oriented Programming*, LNCS 615, Springer-Verlag, pp. 372-395, Junho, 1992.

[116] KICZALES, G., IRWING, J. *et al.*, "Aspect-Oriented Programming", *ACM Computing Surveys*, Vol. 28, No. 4 special 164, Junho, 1996.

[117] AOP Group, *AspectJ Specification*, Xerox, Paulo Alto Center, Março, 1998.

[118] KICZALES, G., LOPES, C., *Aspect-Oriented Programming with AspectJ*, Xerox PARC, 1998.
http://www.parc.xerox.com/aop

[119] IRWING, J., LOINGTIER, J-M., GILBERT, J. R. *et al.*, "Aspect-Oriented Programming of Sparse Matrix Code", In *Proceedings International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE),* Springer-Verlag LNCS 1343, Marina del Rey, CA, Dezembro, 1997.

[120] LAMPING, J., *The Interaction of Components and Aspects*, ECOOP'97, AOP Workshop document, 1997.

http://www.cs.utwente.nl/aop-ecoop97

[121] LOQUES, O. G., LEITE, J., CARRERA, E. V, "P-RIO: A Modular Parallel-Programming Environment", *IEEE Concurrency*, Vol. 6, No. 1, pp. 47-57, Janeiro-Março, 1998.

[122] MENS, K., LOPES, C., TEKINERDOGAN, B., KICZALES, G., *Aspect-Oriented Programming Workshop Report*, ECOOP'97, Workshop on Aspect-Oriented Programming (W15), Jyvaskyla, Finlândia, Junho, 1997.

[123] MAFFEIS, S., "Adding group communication and fault-tolerance to CORBA", *In: Proceedings of the 1995 USENIX Conference on Object Oriented Technologies*, Monterey, CA, EUA, Junho, 1995.

[124] Object Management Group, *CORBA Messaging Specification*, OMG document, ORBOS/98-05-05 ed., Maio, 1998.

[125] SCHMIDT, D. C., KUHNS, F., "An Overview of the Real-Time CORBA Specification", *IEEE Computer*, Vol. 33, No. 6, pp. 56-63, Junho, 2000.

[126] VINOSKI, S., "New Features for CORBA 3.0", *Communications of the ACM*, Vol. 41, pp. 44-52, Outubro, 1998.

[127] KOBRYN, C., "Modeling Components and Frameworks with UML", *Communications of the ACM*, Vol. 43, No. 10, pp. 31-38, Outubro, 2000.

[128] ARNOLD, K., O'SULLIVAN, B., SCHEIFLER, R. W. *et al*, *The Jini™ Specification*, 1st ed., Addison Wesley Publishing, EUA, 1999.

[129] WALDO, J., "Alive and Well: Jini Technology Today", *IEEE Computer*, Vol. 33, No. 6, pp. 107-109, Junho, 2000.

[130] LUCKHAM, D. C., *et al*, "Specification and Analysis of System Architecture Using Rapide", *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, pp. 336-355, Abril, 1995.

[131] CRANE, S., MAGEE, J., DULAY, N., TWIDLE, K., *Regis Programmer's Manual*, The Regis Implementation Group, Release 0.6.1, Imperial College of London, Agosto, 1995.

[132] Darwin Group, *The Darwin Language - Version 3c*, Department of Computing, Imperial College of Science, Technology and Medicine, Março, 1997.

[133] ZELESNIK, G., *Adding Support for Connector Abstractions in the UniCon Compiler*, Department of Computing, Carnegie Mellon University, 1997.

http://www.cs.cmu.edu/afs/cs/project/vit/www/unicon/adding-connectors/expert-creation.html

[134] BOOCH, G., JACOBSON, I., RUMBAUGH, J., *Unified Modeling Language*, Versão 1.0, Rational Software Corp., 1997.

[135] LARSEN, G., "UML 2001: A Standardization Odyssey", *Communications of the ACM*, Vol. 42, No. 10, pp. 29-37, Outubro, 2000.

[136] KLEPPE, A., WARMER, J., *The Object Constraint Language: Precise Modeling with UML*, Object Technology Series, Addison Wesley Publishing, EUA, 1999.

[137] DOBLE, J., MESZAROS, G., CROCKER, R., *Software Architecture: Is it What's Missing From OO Methodologies*, OOPSLA'2001, Tutorial 27, Tampa Bay, Florida, EUA, Outubro, 2001.

[138] BROOKS, F. P., *The mythical man month: experiences on software engineering*, 1st. ed., Addison Wesley Publishing, EUA, Junho, 1978.

[139] PLEEGER, S. L., "Ascending Mount Software", *IEEE Spectrum*, Books section, Vol. 37, No. 5, Maio, 2000.

[140] BACH, M. J., *The Design of the UNIX Operation System*, Englewood Cliffs, NJ, Prentice-Hall, 1987.

[141] ASTLEY, M., AGHA, G., "Customizations and Composition of Distributed Objects: *Middleware* Abstractions for Policy Management", *In: Proceedings of the ACM SIGSOFT'98 - 6th International Symposium on Foundations of Software Engineering*, Lake BuenaVista, Florida, pp. 1-9, Novembro, 1998.

[142] LOBOSCO, M., *R-RIO: Um Ambiente para Suporte à Construção e Evolução de Sistemas*, Dissertação de M.Sc., Instituto de Computação/UFF, Março, 1999.

[143] MONROE, R. T., KOMPANEK, A., MELTON, R., GARLAN, D., "Architectural Styles, Design Patterns and Objects", *IEEE Software*, Vol. 14, No. 1, pp. 43-52, Janeiro, 1997.

[144] JACKSON, M, e ZAVE, P., "Distributed Feature Composition: A Virtual Architecture for Telecommunications Services", *IEEE Transactions on Software Engineering*, Vol. 24, No. 10, pp. 831-847, Outubro, 1998.

[145] SPITZNAGEL, B., GARLAN, D., "A Compositional Approach for Constructing Connectors", *The Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, Royal Netherlands Academy of Arts and Sciences, Amsterdam, Holanda, Agosto, 2001.

[146] SZTAJNBERG, A., LOBOSCO, M., LOQUES, O. G., "Configurando protocolos de interação na abordagem R-RIO", *In: anais do XIII Simpósio Brasileiro de Engenharia de Software*, pp. 29-45, Florianópolis, Outubro, 1999.

[147] de CARVALHO, S. T., *Um design pattern para a configuração de arquiteturas de software*, Dissertação de M.Sc., Instituto de Computação/UFF, Março, 2001.

[148] MEDVIDOVIC, N., *Architecture-Based Specification-Time Software Evolution*, Tese de D.Sc., University of California Irvine, 1999.

[149] CRANE, S., MAGEE, J., PRYCE, N., "Design Patterns for Binding in Distributed Systems", OOPSLA'95, *Workshop on Design Patterns for Concurrent and Distributed Object-Oriented Systems*, Outubro, 1995.

[150] PERRY, D. E., "Software Interconnection Models", *In: Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, EUA, May, 1987.

[151] MEI, H., "A Complementary Approach to Requirements Engineering - Software Architecture Orientation", *ACM SIGSOFT Software Engineering Notes*, Vol. 25, No. 2, pp. 40-45, Março, 2000.

[152] ZAVE, P., "Classification of Research Efforts in Requirements Engineering", *ACM Computing Surveys*, Vol. 29, No. 4, pp. 315-321, Dezembro, 1997.

[153] RICJARDSON, D., INVERARDI, P., *ROSATEA: International Workshop on the Role of Software Architecture in Analysis and Testing*, ACM SIGSOFT Software Engineering Notes, Vol. 24, No. 4, July, 1999.

[154] MATTHIJS, F., *et al.*, "Aspects should not die", ECOOP'97, *Workshop on Aspect-Oriented Programming (W15)*, Jyvaskyla, Finlândia, Junho, 1997. http://www.cs.utwente.nl/aop-ecoop97

[155] CYSNEIROS, L. M., LEITE, J. C. S. P., NETO, J. M. S., "A Framework for Integrating Non-Functional Requirements into Conceptual Models", *Requirements Engineering Journal*, Springer-Verlag London, Vol. 6, No. 2, pp. 97-115, 2001.

[156] BIRMAN, K. P., *Building Secure and Reliable Network Applications*, 1st ed., Manning Publications, EUA, 1996.

[157] ANDRESON, K. *Thoughts on AOP*, 1997. http://openmap.bbn.com/~kanderson/aop/AOP_thoughts.htm

[158] SHAW, M., DELINE, R., ZELESNIK, G., "Abstractions and Implementations for Architectural Connections", *In: Proceedings of the IEEE 3rd International Conference on Configurable Distributed Systems (ICCDS'96)*, 1996.

[159] PURAO, S., JAIN, H., NAZARETH, D., "Effective Distribution of Object-Oriented Applications", *Communications of the ACM*, Vol. 41, No. 8, pp. 100-108, Agosto, 1998.

[160] BEM-ARI, M., *Principles of Concurrent and Distributed Programming*, Englewood Cliffs, NJ, Prentice-Hall, 1990.

[161] POWELL, M. L., e*t al.*, "SunOS 5.0 Multithread Architecture", *In: Proceedings of Usenix Winter Technical Conference*, pp. 65-79, Dalas, TX, EUA, 1991.

[162] FRØLUND, S., "Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages", *In: Proceedings of the ECOOP'92 - 6th European Conference on Object-oriented Programming*, LNCS 615, pp. 185-196, Springer-Verlag, 1992.

[163] LEA, R., JACQUEMOT, C., PILLEVESSE, E., "COOL: System Support for Distributed Programming", *Communications of the ACM*, Vol. 36, No. 9, pp. 37-47, Setembro, 1993.

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

[165] MEYER, B., "Systematic Concurrent Object-Oriented Programming", *Communications of the ACM*, Vol. 36, No. 9, pp. 56-80, Setembro, 1993.

[166] LÖHR, K-P., "Concurrency Annotations for Reusable Software", *Communications of the ACM*, Vol. 36, No. 9, pp. 81-89, Setembro, 1993.

[167] MATSUOKA, S., "Analysis of inheritance anomaly in object-oriented concurrent programming languages", *In: Research Directions in Concurrent Object-Oriented Programming,* The MIT Press, pp. 107-150, 1993.

[168] GOSLING, J., JOY, B., STEELE, G., *The Java Language Specification*, Sun Microsystems, Addison Wesley Publishing, EUA, 1996.

[169] SCHMIDT D. C., LEVINE, D., MUNGEE, S., "The Design of the TAO Real-Time Object Request Broker", *Computer Communications*, Elsevier Science, Vol. 21, No. 4, Abril, 1998.

[170] BUHR, P. A., FORTIER, M., COFFIN, M. H, "Monitor Classification", *ACM Computing Surveys*, Vol. 27, No. 1, pp. 63-108, Março de 1995.

[171] LEDGARD, H., *ADA: Na Introduction*, Springer-Verlag, EUA, 1983.

[172] SILVA, A. R., PEREIRA, J., MARQUES, J. A., "Object Synchronizer", Chapter 8, pp. 111-131, *In: Pattern Languages of Program Design 4*, Neil Harrison, Brian Foote e Hans Rohnert (Eds.), *Software Patterns Series*, Addison Wesley Publishing, EUA, 1999.

[173] MEYER, B., "Applying Design by Contract", *IEEE Computer*, Vol. 25, No. 10, pp. 40-51, Outubro, 1992.

[174] HELM, R., HOLLAND, I. M., GANGOPADHYAY, D., "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", *In: Proceedings of the EC24 - 4th European Conference on Object-oriented Programming*, pp. 169-180, Ottawa, Outubro, 1990.

[175] BEUGNARD, A., JÉZÉQUEL, J-M., PLOUZEAU, N. *et al*, "Making Components Contract Aware", *IEEE Computer*, pp. 38-45, Julho, 1999.

[176] KINIRY, J., *Leading to a Kind Description Language: Thoughts on Component Specification*, Caltech Technical Report, CS-TR-99-04, California Institute of Technology, Maio, 1999.

[177] KRAMER, J., MAGEE, J., "Exposing the Skeleton in the Coordination Closet", *Coordination'97*, pp. 336, Berlin, Alemanha, 1997.

[178] MÉTAYER, D. L., PÉRIN, M., "Multiple Views in Software Architecture: Consistency and Conformance", *In: Proceedings of the 1st IFIP Conference on Software Architecture (WICSA1)*, San Antonio, EUA, 1999.

[179] ASTLEY M., AGHA G. A., "Customization and Composition of Distributed Objects: *Middleware* Abstractions for Policy Management", *In: Proceedings of the ACM SIFSOFT'98 - 6th International Symposium on the Foundations of Software Engineering*, Software Engineering Notes, Vol. 23, No. 6, pp. 1-9, 1998.

[180] CAZZOLA, W., SAVIGNI, A., SOSIO, A., *et al*, *Architectural Reflection: Concepts, Design, and Evaluation*, Technical Report RI-DSI 234-99, DSI, University degli Studi di Milano, Maio, 1999.

[181] MEYER, B., "Tell Less, Say More: The Power of Implicitness", *IEEE Computer*, pp. 97-98, Julho, 1998.

[182] KRAMER, J., MAGEE, J., "The Evolving Philosophers Problem: Dynamic Change Management", *IEEE Transactions on Software Engineering*, Vol. 16, No. 11, pp. 1293-1306, 1991.

[183] GOUDARZI, K. M., KRAMER, J., "Maintaining Node Consistency in the Face of Dynamic Change", *In: Proceedings of the IEEE 3rd International Conference on Configurable Distributed Systems*, pp 62-69, Annapolis, Maryland, EUA, Maio, 1996.

[184] STAEHLI, R., WALPOLE, J., MAIER, D., "Quality of Service Specification for Multimedia Presentations", *IEEE Multimedia Systems*, Vol. 3, No. 5/6, pp. 251-265, Novembro, 1995.

[185] FRØLUND, S., KOISTINEN, J., "Quality-of-service specifications in distributed object systems", *IEE Distributed Systems Engineering Journal*, V. 5, The British Computer Society, pp-179-202, UK, 1998.

[186] SELIC, B., "A Generic Framework for Modeling Resources with UML", *IEEE Computer*, Vol. 33, No. 6, pp. 64-69, Junho, 2000.

[187] ZINKY, J. A., BAKKEN, D. E., SCHANTZ, R. E., "Architectural Support for Quality of Service for CORBA Objects", *Theory and Practice of Object Systems*, John Wiley & Sons, Inc., Vol. 3, No. 1, 1997.

[188] MELLIAR-SMITH, P. M., MOSER, E., NARASIMHAN, P., "Separation of Concerns: Functionality vs. Quality-of-Service", *In: Proceedings of the 3rd Workshop on Object-Oriented Real-Time Dependable Systems*, Newport, CA, EUA, pp. 272-274, Fevereiro, 1997.

[189] BRITO, O. F. G., *Tolerância a Falhas no Ambiente RIO*, Dissertação de M.Sc., DEE/PUC-RJ, Março, 1995.

[190] XU, D., WICHADAKUL, D., NAHRSTEDT, K., "Resource-Aware Configuration of Ubiquitous Multimedia Service", *In: Proceedings of the ICME2000 - IEEE International Conference on Multimedia and Expo 2000*, NewYork, NY, July, 2000.

[191] KOLIVER, C., FARINES, J-M., FRAGA, J. S., REIS, H. L., "Um Modelo para Adaptação de QoS Orientado ao Usuário Final", *In: anais do 18º Simpósio Brasileiro de Redes de Computadores*, pp. 135-150, Belo Horizonte, MG, Maio, 2000.

[192] LAZAR, A. A., LIM, K. S., MARCONCINI, F., "Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture", *IEEE Journal on Selected Areas in Communications*, No. 7, pp. 1214-1227, Setembro, 1996.

[193] FLORISSI, P. G. S., *QoSME: QoS Management Environment*, Tese de D.Sc., Technical Report CUCS-036-95, Columbia University, NY, EUA, 1996.

[194] STANKOVIC, J. A., SON, S. H., LIEBERHERR, J., *BeeHive: Global Multimedia Database Support for Dependable, Real-Time Applications*, Computer Science Report No. CS-97-08, University of Virginia, April, 1997.

[195] COULSON, G., BLAIR, G. S., DAVIES, N., *et al.*, "Supporting Mobile Multimedia Applications Through Adaptive *Middleware*", *IEEE Journal on Selected Areas in Communications*, No. 17, Setembro, 1999.

[196] SIQUEIRA, F., CAHILL, V., "Quartz: A QoS Architecture for Open Systems", *In: anais do 18º Simpósio Brasileiro de Redes de Computadores*, pp. 553-568, Belo Horizonte, MG, Maio, 2000.

[197] FRØLUND, S., KOISTINEN, J., *Quality of Service Aware Distributed Object Systems*, HPL-98-142, Hewlett-Packard, Palo Alto, EUA, Agosto, 1998.

[198] FRØLUND, S., KOISTINEN, J., *QML: A Language for Quality of Service*, HPL-98-10, Hewlett-Packard, Palo Alto, EUA, Fevereiro, 1998.

[199] NAHRSTEDT, K., WICHADAKUL, D., XU, D., "Distributed QoS Compilation and Runtime Instantiation", *In: Proceedings of the IEEE/IFIP International Workshop on QoS 2000 (IWQoS2000)*, Pittsburgh, PA, June, 2000.

[200] HARICH, J., "The trick to using a basic Java 1.1 network and file class loader", *JavaWorld*, Outubro, 1997.

http://www.javaworld.com/javaworld/javatips/jw-javatip39.html.

[201] Sun Microsystems, *Java® 2 SDK, Standard Documentation - Version 1.2.1*, 1998.

[202] SZTAJNBERG, A., *Ferramentas para R-RIO*, Relatório Técnico R-RIO-001, GTA/COPPE/UFRJ, 2000.

[203] SZTAJNBERG, A., *Medidas de desempenho de conectores em R-RIO*, Relatório Técnico R-RIO-002, GTA/COPPE/UFRJ, 2000.

[204] AOP Group, *AspectJ: 0.2 Beta*, Xerox, Paulo Alto Center, Março, 1998.

http://www.parc.xeox.com/aop

[205] ANDREW C. H., BENJAMIN C. L., PONNEKANTI, S., "Pervasive computing: what is it good for?", *In: Proceedings of the ACM International Workshop on Data Engineering for Wireless and Mobile Computing*, pp. 84-91, Seattle, WA, EUA, Agosto, 1999.

[206] HAARTSEN, J. C., "The Bluetooth Radio System", *IEEE Personal Communications*, pp. 28-36, Fevereiro, 2000.

[207] WILLIAMS, S., "IrDA: Past, Present and Future", *IEEE Personal Communications*, pp. 11-19, Fevereiro, 2000.

[208] NEGUS, K. J., STEPHENS, A. P., LANSFORD, J., "HomeRF: Wireless Networking for the Connected Home", *IEEE Personal Communications*, pp. 20-27, Fevereiro, 2000.

[209] PERKINS, C. E., "Mobile IP", *IEEE Communications*, Maio, 1997.

[210] VARSHNEY, U., VETTER, R., "Emerging Mobile and Wireless Networks", *Communications of the ACM*, Vol. 43, No. 6, pp. 73-81, Junho, 2000.

[211] GINZBOORG, P., "Seven Comments on Charging and Billing", *Communications of the ACM*, Vol. 43, No. 11, pp. 89-92, Novembro, 2000.

[212] SALTZER J.H., *et al.*, "End-to-End Arguments in System Design", *In: Proceedings of the 2nd International Conference on Distributed Computing Systems*, pp. 509-512, Paris, França, 1981.

[213] HALLOWAY, S., *Improving Code Reuse With Servlet Filters*, JDC Tech Tips June 26, Sun Microsystems, 2001.

http://java.sun.com/jdc/JDCTechTips/2001/tt0626.html

[214] BUSCHMAN, F., MEUNIER, R., ROHNERT, H., *et al.*, *Patter-Oriented Software Architecture: A System of Patterns*, John Wiley and Sons, Chichester, UK, 1996.

[215] SCHMIDT, D. C., STAL, M., ROHNERT, H., BUSCHMANN, F., *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*, John Wiley & Sons, New York, 2000.

[216] SRINIVASAN, S., "Design Patterns in Object-Oriented Frameworks", *IEEE Computer*, Vol. 32, No. 2, pp. 24-31, Fevereiro, 1999.

[217] SCHMIDT, D. C., CLEELAND, C., "Applying Patterns to Develop Extensible ORB *Middleware*", *IEEE Communications Magazine Special Issue on Design Patterns*, 1999.

[218] BLAIR, G., CAMPBELL, R., "Summing Up", *Middleware*'2000, *Workshop on Reflective Middleware*, Palisades, NY, Abril, 2000.

http://www.comp.lancs.ac.uk/computing/users/johnstlr/rm2000/summingup.html

[219] COULSON, G., "What is Reflective *Middleware*?", *IEEE Distributed Systems On-Line*, IEEE, 2000.

http://computer.org/dsonline/*middleware*/RM.htm

[220] GRAY, J., REUTER, A., *Transaction Processing: Concepts and Techniques*, Morgan Kaufman Publishers, San Mateo, California, 1993.

[221] DASHOFY, E. M., HOEK, A., TAYLOR, R. N., "A Highly-Extensible, XML-Based Architecture Description Language", *In Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA)*, Amsterdam, Holanda, 2001.

[222] BROSE, G., *JacORB Programming Guide, v0.9*, Institut fur Informatik, Freie Universitat, Berlin, Alemanha, Junho, 1998.

[223] UDELL, J., "Exploring XML-RPC", *Byte Magazine*, Agosto, 1999.

http://www.byte.com/documents/s=109/byt19990830s0001/

[224] MAGEE, J., KRAMER, J., *Concurrency: state models & Java programs*, 1st ed., John Wiley & Sons, UK, 1999.

[225] LOBOSCO, M., LOQUES, O. G., SZTAJNBERG, A., "Configuração de Sistemas Distribuídos Baseada em Java", *In: anais do I Simpósio de Sistemas de Simulação e Controle*, 41-48, IPqM, Rio de Janeiro, Setembro, 1998.

[226] LOBOSCO, M., LOQUES, O. G., SZTAJNBERG, A., "R-RIO: Um Ambiente para Suporte à Construção e à Evolução de Sistemas", *XIII Simpósio Brasileiro de Engenharia de Software 99 (Caderno de Ferramentas)*, pp. 53-56, Florianópolis, Outubro, 1999.

[227] SZTAJNBERG, A., LOBOSCO, M., LOQUES, O. G., "Uma Plataforma para Suporte à Evolução de Sistemas Distribuídos", *In: anais do II Workshop em Sistemas Distribuídos*, pp.25-32, Curitiba, Maio, 1998.

[228] SZTAJNBERG, A., LOBOSCO, M., LOQUES, O. G., "Separação de Interesses em Sistemas Distribuídos", *In: anais do I Simpósio de Sistemas de Simulação e Controle*, pp.33-40, IPqM, Rio de Janeiro, Setembro, 1998.

[229] SZTAJNBERG, A., LOBOSCO, M., LOQUES, O. G., "A Abordagem R-RIO para Concepção de Aplicações Distribuídas", *In: anais do XVII Simpósio Brasileiro de Redes de Computadores (seção de posters)*, pp. 339-353, Salvador, BA, Maio, 1999.

[230] LOQUES, O. G, LEITE, J., SZTAJNBERG, A, LOBOSCO, M., "Towards Integrating Meta-Level Programming and Configuration Programming", *In: anais do XIII Simpósio Brasileiro de Engenharia de Software*, pp. 341-354, Florianópolis, Outubro, 1999.

[231] LOQUES, O. G., LEITE, J., SZTAJNBERG, A., LOBOSCO, M., "Integrating Meta-Level Programming and Configuration Programming", OOPSLA'99, *Workshop on Reflection and Software Engineering*, pp. 137-152, Denver, EUA, Novembro, 1999.

[232] SZTAJNBERG, A., LOQUES, O. G., "Reflection in the R-RIO Configuration Programming Environment", IEEE *Middleware*'2000, *Workshop on Reflective Middleware*, Palisades, NY, EUA, Abril, 2000.

[233] LOQUES, O. G., SZTAJNBERG, A., LEITE, J., e LOBOSCO, M., "On the Integration of Configuration and Meta-Level Programming Approaches", *In: Reflection and Software Engineering* (special edition), *LNCS 1826*, Springer-Verlag, pp.191-210, Berlin, Alemanha, Junho, 2000.

[234] SZTAJNBERG, A., LOQUES, O. G., "Bringing QoS Specifications to the Architectural Level", ECOOP'2000, *Workshop on QoS for Distributed Object Systems*, Junho, França, 2000.

[235] SZTAJNBERG, A., LOQUES, O. G., "R-RIO: Reflective-Reconfigurable Interconnectable Objects", *ACM OOPSLA'2000 Companion*, pp. 85-86, Minneapolis, Outubro, 2000.

[236] SZTAJNBERG, A., *Políticas para ligação de componentes em R-RIO*, Relatório Técnico R-RIO-003, GTA/COPPE/UFRJ, 2000.

[237] BERNERS-LEE, T., MASINTER, L., MCCAHILL, M., *Uniform Resource Locators (URL)*, Request for Comments: 1738, Network Working Group, Category: Standards Track, Dezembro, 1994.

[238] SZTAJNBERG, A., *Estudo sobre as variações de ligação de conectores em R-RIO*, Relatório Técnico R-RIO-004, GTA/COPPE/UFRJ, 2000.

[239] HOARE, C. A. R., *Communicating Sequential Processes*, Prentice-Hall International, 1 ed., 1985.

[240] SZTAJNBERG, A., *Restrições para a instanciação de conectores compostos em R-RIO*, Relatório Técnico R-RIO-005, GTA/COPPE/UFRJ, 2001.

[241] MIKHAJLOV, L., SEKERINSKI, E., *The Fragile Base Class Problem and Its Solution*, TUCS Technical Report No. 117, Turku Center of Computer Science, University of Turku, Finlândia, Junho, 1997.

[242] PETERSON, J. L., *Petri Net theory and modeling of systems*, Englewood Cliffs, NJ, Prentice Hall, 1981.

[243] ISSO IS 8807, *LOTOS - a formal description technique based on temporal ordering of observational behavior*, Technical Report, ISSO, Genebra, Suíça, Setembro, 1989.

[244] International Organization for Standardization, *ESTELLE: A Formal Description Technique Based on the Extended State Transition Model*, Technical Report IS 9074, ISSO, 1989.

[245] CCITT Com X-R 17-E, *Rec Z100 Functional Specification and Description Language (SDL)*, Technical Report, CCITT, 1992.

[246] CIANCARINI, P., MASCOLO, C., "Model Checking a Software Architecture", *International Workshop on the Role of Software Architecture in Testing and Analysis*, Marsala, Sicília, Itália, Julho, 1998.

[247] DWYER, M. B., PASAREANU, C. S., "Model Checking Generic Container Implementations", *In: Proceedings of a Dagstuhl Seminar*, LNCS 1766, Springer-Verlag, 2000.

[248] DWYER, M. B., PASAREANU, C. S., Corbett, J., *Translating Ada Programs for Model Checking: A Tutorial*, Kansas State University, Outubro, 1998.

[249] FERNANDES, E. J. G. *Especificação e Verificação Modular-Hierárquica de Sistemas Concorrentes*, Dissertação de M.Sc. PEE/COPPE/UFRJ, Julho, 2001.

[250] SOBRAL, J. B. M., *Uma Linguagem para Especificação de Sistemas Distribuídos em Alto Nível de Abstração*, Tese de D.Sc., COPPE/PEE/UFRJ, 1996.

[251] ALMEIDA, N. N., *Modularidade e Composicionalidade de Sistemas Concorrentes*, Tese de D.Sc., PEE/COPPE/UFRJ, Março, 1997.

[252] DUARTE, C. H. C., *Proof-Theoretic Foundations for the Design of Extensible Software Systems*, Tese de D.Sc., Department of Computing, Imperial College, University of London, UK, 1998.

[253] BARBOSA, V. C. *An Introduction to Distributed Algorithms*, The MIT Press, EUA, 1996.

[254] MANNA, Z., PNUELI, A., *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, Nova York, EUA, 1992.

[255] ABADI, M., CARDELLI, L., *A theory of objects*, Monographs in computer science, Springer-Verlag, Nova York, EUA, 1996.

Esta página foi intencionalmente deixada em branco

Apêndice A

CBabel: uma ADL para R-RIO

A.1 Introdução

Neste apêndice apresenta-se a linguagem CBabel, para a descrição e configuração de arquiteturas de *software*, parte integrante do *framework* R-RIO.

Na seção A.2, são apresentadas a sintaxe e a semântica para a declaração de módulos, portas e conectores, materializando os conceitos do modelo de componentes de R-RIO. Em seguida, na seção A.3, são examinadas as declarações de configuração, que permitem formação da arquitetura da aplicação. A sintaxe para a descrição de contratos de aspectos não-funcionais é apresentada na seção A.4. Nas seções seguintes são discutidos alguns detalhes sobre composição de módulos e conectores. Na última seção apresenta-se o BNF de CBabel.

A.2 CBabel - *Building Applications by Evolution with Connectors*

A ADL desenvolvida para R-RIO, chamada CBabel - *Building Applications by Evolution with Connectors*[11], permite, como outras ADLs, a descrição de componentes e a arquitetura de aplicações, bem como o armazenamento e a recuperação desta descrição. Em adição a estas características, CBabel oferece recursos para a descrição e configuração de aspectos não-funcionais das aplicações.

---

[11] Em [102] é apresentada Babel, uma versão de CBabel, sem conectores e contratos de aspectos não-funcionais, proposta para um ambiente de suporte à configuração, em um contexto diferente de R-RIO [79].

Uma declaração de componente em CBabel possui sintaxe semelhante à da declaração de módulos e interfaces na linguagem OMG-IDL [8], com algumas diferenças semânticas, porém. A arquitetura e o estilo de ligação entre clientes e servidores estão implícitos nas descrições de interfaces em IDLs, mas não podem ser referenciados ou configurados como nas ADLs.

Como a maioria das ADLs, CBabel é essencialmente declarativa, ou seja, não existem propriamente comandos a serem executados na descrição da arquitetura. Por outro lado, não se pode dissociar completamente as declarações que descrevem a arquitetura da aplicação em CBabel, do efeito que estas devem produzir quando a arquitetura é colocada em execução, ou durante uma reconfiguração. Assim, por exemplo, na apresentação das declarações correlacionadas a um método da API de configuração, a linguagem será relacionada com o efeito esperado nestas fases.

A.2.1 Referências

Na linguagem de configuração, as referências a classes de módulos e conectores, tipos de portas, ou referências a instâncias destes elementos, são representadas por nomes. Um nome é constituído por uma cadeia de caracteres. As referências têm a sua visibilidade regida pelo escopo onde são declaradas (módulos ou conectores compostos, veja seção A.3) e onde estão sendo referenciadas. Uma referência pode usar implicitamente o escopo como informação de visibilidade, ou pode utilizar explicitamente a informação de toda a localização, uma referência completa. Referências completas são representadas por nomes separados por pontos em um esquema hierárquico. Na listagem A.1 estas variações são apresentadas.

```
STRING                  // uma referência a um elemento qualquer (módulo, porta ou conector)
módulo.porta            // uma referência a porta de um módulo
módulo at nó            // uma referência a um módulo em um nó
módulo.porta at nó      // uma referência a uma porta de um módulo em um nó
```

Listagem A.1 - Referências em CBabel

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

A.2.2 Portas

A declaração de um tipo ou instância de porta é apresentada na listagem A.2. A indicação do sentido da porta (*in* ou *out*) tem o objetivo de explicitar se a mesma responderá a requisições (um servidor) ou se esta iniciará uma requisição (um cliente) e, neste caso, é declarada apenas em uma instância de porta. No caso de portas síncrona, um parâmetro é usado indicar o tipo de retorno. O parâmetro *oneway* é utilizado para indicar que as requisições serão assíncronas. Uma lista de tipos e nomes de parâmetros também podem ser declarados. Esta será usada como argumento em uma requisição. O sentido destes parâmetros (entrada ou saída) está associado ao sentido declarado na instância da porta.

```
1    [in | out] port <oneway | tipo-do-retorno> <nome-tipo-porta>
2       (lista-de-parâmetros)
3    [instâncias;]
```

Listagem A.2 - Porta em CBabel

Observações:

- um tipo de porta pode ser declarado separadamente de um módulo e este tipo poderá ser instanciado e especializado em módulos diferentes.
- um tipo de porta é declarado sem o sentido. Na declaração de instâncias desta porta, para formar a 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 OMG-IDL. Por exemplo,

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

Se um nome for declarado para o parâmetro, este poderá ser referenciado explicitamente na arquitetura. Caso contrário, o parâmetro será utilizado apenas na verificação da compatibilidade de assinaturas.

A.2.2.1 Portas complementares

Uma instância de porta é complementar a outra se o tipo de porta é o mesmo, mas os sentidos são inversos, ou seja uma porta é ***in*** e outra é ***out***. Adicionalmente, uma política para verificação da compatibilidade de contexto pode relaxar esta definição

[236]. Por exemplo, em alguns casos, os parâmetros de uma instância de porta podem ser um subconjunto da outra instância de porta, e estas portas serem consideradas complementares para efeitos de ligação. Isto poderá ocorrer porque, na configuração da arquitetura em questão, os módulos podem interagir desta forma, ou porque o projetista utilizou um conector que "torna" as portas compatíveis.

A.2.3 Módulos

A declaração de uma classe de módulo ou referências a instância de módulo é feita através de uma declaração **module**. Uma classe de módulo pode ser usada na declaração de instâncias de módulo. É possível declarar também uma referência a instância de módulo sem associação com um nome de classe. A descrição do corpo do módulo pode conter a descrição das portas de entrada e saída, ou referências a portas já descritas anteriormente (listagem A.3).

```
module <nome-classe-módulo> [(lista-de-parâmetros)] {
   <variáveis internas ao módulo>
   port <nome-tipo-porta> [{atributos}] <nome-porta>;
   map <mapeamento>;
} [instâncias;]
```

Listagem A.3 - Módulo em CBabel

No corpo do módulo também é permitida a declaração de variáveis. O tipo de tais variáveis pode ser qualquer dentre os previstos na OMG-IDL. A declaração de uma variável no corpo do módulo implica na sua visibilidade no nível da arquitetura de *software*, e implica, também que, em um mapeamento para uma implementação, esta seria uma variável do objeto (figura A.1). Em conjunto com a declaração de uma ou mais variáveis, é necessário prever a existência de uma porta de entrada (e um método associado a esta porta) que permite a consulta do valor destas variáveis, de forma atômica, e em exclusão mútua com todos os outros métodos do módulo. Este método será utilizado, na maioria das vezes, por conectores que precisam conhecer o estado interno do módulo.

No corpo do módulo, a declaração **map** indica o mapeamento para uma implementação. Detalhes sobre o mapeamento serão apresentados adiante.

O corpo de um módulo também pode conter a descrição de outros módulos, para a formação de módulos compostos. A composição de módulos em CBabel é discutida

na seção A.5.

A.2.3.1 Definição e criação de referências

Referências a instâncias de módulos podem ser declaradas juntamente com a declaração da classe de módulos ou separadamente. Estas referências não criam efetivamente uma unidade de execução, mas selecionam um nome para ser utilizado na descrição da arquitetura da aplicação. A listagem A.4 apresenta alguns exemplos de declaração de classes e instâncias de módulos.

module *modA* {<corpo>}	*// define a classe modA.*
module *modB* (**int** *a*, **string** *nome*, **int** *b*) {<corpo>}	*// define a classe modB com uma lista de parâmetros.*
module *modA iModA*;	*// cria uma referência de instância, ou seja, a associação entre a classe modA e a instância iModA. Não define a classe modA (esta já deve ter sido declarada).*
module {<corpo>} *iModA*;	*// cria uma referência de instância de um módulo particular, que não estará disponível para ser reutilizado.*
module *modC* <(parâmetros)> {<corpo>} *iMod1*, *iMod2*;	*// define a classe modC, com parâmetros e cria as referências para as instancias iMod1 e iMod2 (os argumentos das instancias só serão passados no momento de sua criação.)*

Listagem A.4 - Definição e referências a módulos

A.2.3.2 Criação de instâncias

A criação de uma instância de módulo em CBabel indica a ordem e o contexto desejado para a criação desta instância em tempo de execução. Na ADL, entretanto, o efeito é apenas declarativo. A criação de uma instância é declarada com o comando `instantiate`. É necessário que as classes de módulos já tenham sido declaradas. A listagem A.5 apresenta alguns exemplos.

instantiate *modA* **as** *iMod4*;	*// cria a instância iMod4 a partir da classe modA.*
instantiate *modB* **as** *iMod5* (*"abcd"* , 4);	*// cria a instância iMod5 passando os argumentos necessários.*
instantiate *iMod2* (*"abc"*);	*// cria a instância iMod2 a partir da classe modC. A associação já havia sido feita anteriormente.*
instantiate *modA* **as** *iMod6*, *iMod7*, *iMod8*;	*// cria as instâncias iMod6, iMod7 e iMod8 a partir da classe modA.*

Listagem A.5 - Instâncias de módulos

A.2.3.3 Com vetores

Para facilitar a descrição de arquiteturas que empregam conjuntos de componentes de uma mesma classe, onde seria necessária a seleção de nomes diferentes de referências para as instâncias, a declaração de vetores de módulos pode ser utilizada (primeira declaração da listagem A.6). A vantagem do uso de vetores está na possibilidade de comandos parametrizados e uma descrição mais compacta da arquitetura. A utilização da segunda forma da listagem A.6, facilita a descrição de vetores de várias dimensões ou um esquema mais complexo de parametrização.

`instantiate modA as modArr [0..4];`	*// cria um vetor, modArr, de instâncias da classe modA com 5 posições.*
`for i=1 to 10 instantiate modA as modArr [i];`	*// cria um vetor , modArr de instâncias de modA com 10 posições.*

Listagem A.6 - Vetores de módulos

A.2.3.4 Com indicação de localização

A criação de instâncias de módulos pode indicar o nó em que a instância será criada. Embora tal característica seja uma aspecto não-funcional, optou-se por introduzir este suporte diretamente na declaração dos módulos (listagem A.7). A indicação ou omissão da localização das instâncias a serem criadas tem influência direta na seleção de um conector adequado para interligar módulos. Se os módulos a serem ligados estiverem em nós diferentes, o conector precisa oferecer suporte à comunicação, por exemplo.

`instantiate imodA at nó1;`	*// cria a instância imodA, já declarada anteriormente no nó nó1.*
`instantiate modA as imodM ("teste", 23) at nó1`	*// cria a instância imodM da classe modA no nó nó1. São passados os argumentos na criação da instância.*

Listagem A.7 - Localização de instâncias de módulos

O nó efetivo pode ser definido por um mapeamento, se uma URL não for indicada diretamente.

A.2.4 Conectores

A declaração de classes de conectores e referências a instâncias de classes de

conectores é semelhante à dos módulos. Classes ou instâncias de conectores são declaradas através da declaração **connector** (listagem A.8). No corpo de um conector podem-se declarar referências a tipos ou instâncias das portas de entrada e saída. Se a função do conector é apenas encaminhar as requisições de um módulo para outro, as portas são declaradas aos pares. Uma porta de entrada de um conector encaminha o fluxo recebido de um módulo para uma porta de saída, que interliga o módulo destinatário. A utilização de referências a instâncias de portas não é obrigatória nos conectores, estas existirão implicitamente. Entretanto, ao se fazer referência a um conector dentro de uma configuração, é necessário que as portas do mesmo possam ser identificadas sem ambigüidade. Isto pode ser verificado automaticamente por um compilador.

```
1   connector <nome-classe-conector> [(lista-de-parâmetros)] {
2      in | out port <nome-tipo-porta> [nome-porta];
3      map <mapeamento>;
4   }[instâncias;]
```

Listagem A.8 - Conector em CBabel

Dentro de um conector também podem ser declarados conectores elementares para formar a configuração de um conector composto. Exemplos deste tipo de declaração são apresentados na seção A.6.

A.2.4.1 Discussão

Os conectores são elementos que interligam e intermedeiam a interação de módulos e, portanto, não fazem parte dos aspectos funcionais da aplicação. Discute-se aqui a necessidade da criação explícita de instâncias de conectores em CBabel. Em princípio, isto poderia ser resolvido automaticamente pelo ambiente de execução, sem que tal informação estivesse explicitamente declarada.

Identificam-se três pontos que levam ao raciocínio contrário. Primeiro, um conector pode manter de forma persistente algumas informações sobre as interações que ele intermedeia para efeitos de contabilidade ou sincronização (por exemplo, um conector responsável por manter uma sessão, com possibilidade de *checking-points*, *roll-back* e *logging*). Um segundo ponto seria o caso de uma arquitetura que, por requisito de desempenho, precisa manter duas instâncias distintas de uma mesma classe de conector. O terceiro ponto é a reconfiguração das aplicações. Se a reconfiguração de

um conector se faz necessária, na ausência de uma instância separada, todos os módulos, usando uma mesma classe de conector, sofreriam as conseqüências desta reconfiguração. É, portanto, necessário que, em determinados casos, existam instâncias particulares de conectores e que estas possam ser referenciadas.

Observa-se ainda que a localização do conector, mesmo quando este tiver uma implementação distribuída, deve acompanhar a localização dos módulos que ele interliga, ou será dependente de suas características, como veremos adiante. Não faz sentido, a princípio, indicar-se a localização do conector.

A.2.4.2 Instanciação de conectores

Referências para instâncias de conectores podem ser declaradas de forma semelhante ao que se faz com os módulos. Por exemplo, uma referência, **iCon,** à instância de um conector da classe **CConA** é declarada como:

```
connector CConA {<corpo>} iCon;
```

Instâncias de conectores, entretanto, não são criadas com o comando `instantiate`. Estas são criadas apenas no momento em que forem utilizadas para conectar módulos. O ambiente de execução cria automaticamente uma instância, a partir de uma referência de conector, se esta última ainda não existir (por exemplo, no caso da referência já ter sido usada em outra ligação).

Se uma referência a uma instância de conector não for feita, ou seja, apenas a classe do conector estiver indicada, o ambiente de execução toma a decisão de criar uma instância, com um nome mantido pelo suporte à configuração (que não poderá ser acessado no nível da arquitetura), segundo as seguintes regras:

a) conectores com implementação não distribuída

- se um conector tiver de ser instanciado em um nó onde ainda não exista uma instância, com nome dado internamente pelo configurador, cria-se uma nova instância;
- se o conector for utilizado por módulos no mesmo nó em que um outro conector do mesmo tipo já tenha sido instanciado com um nome interno, esta instância será usada.

b) conectores com implementação distribuída

- se todos os módulos envolvidos estiverem em nós onde já exista uma instância da metade associada a um conector (ver seção A.6.1) com o nome dado internamente, utiliza-se a instância já existente;
- nos nós onde ainda não existir uma instância com um nome dado internamente, uma nova instância é criada.

Se a ligação de dois módulos for feita sem a referência de uma classe de conectores, independente de haver uma referência à instância ou não, o ambiente de execução empregará conectores *default*. Por exemplo, em um ambiente de execução com Java, no caso de módulos em um mesmo nó, o conector *default* pode ser implementando pelo próprio mecanismo de invocação de métodos de Java (MI - *Method Invocation*). Na situação de módulos em nós diferentes, pode utilizar-se o mecanismo de invocações a métodos em objetos remotos (RMI - *Remote Method Invocation*).

A.2.5 Mapeamento de componentes para implementação

Classes ou instâncias de módulos e conectores podem conter uma indicação de seu mapeamento em uma implementação. Esta característica é um diferencial de R-RIO / CBabel em relação a outras propostas. Através do mapeamento, é possível indicar-se uma unidade de execução, objeto, biblioteca ou recurso necessário, que deve estar disponível no ambiente de execução, para implementar o componente. Assim, podemos declarar que determinado módulo é implementado por um programa cujo código está em um arquivo, da seguinte forma:

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

No exemplo anterior, foi declarado que o **código** do módulo (*CODE*), programado em linguagem ***C***++ está armazenado no arquivo *"arquivo"*.

Se uma declaração ***map*** não estiver presente na declaração de um componente, no caso de módulos, o nome da classe do módulo será usado como chave para tentar-se "encontrar" um mapeamento. No caso de conectores serão usadas as alternativas *default*.

A cláusula *map*, essencialmente, indica que existe um mapeamento explícito do componente que está sendo descrito no nível da arquitetura para uma implementação concreta. O mapeamento é fundamental para uma transição da etapa de descrição da arquitetura para as etapas de implementação e execução. O mapeamento também será importante para a manutenção da aplicação. Este esquema oferece a flexibilidade, por exemplo, para se redefinir a forma pela qual um módulo ou conector executam suas tarefas, sem alterações na arquitetura do *software*.

A interpretação de uma cláusula de mapeamento é dependente do sistema de suporte. A sintaxe dos argumentos da cláusula *map* é padronizada, mas o seu conteúdo pode ter significado local específico. Por exemplo, se o ambiente de execução estiver baseado em Java, um módulo pode ser mapeado no *ByteCode* de uma classe. Em um ambiente de execução baseado em Microsoft Windows, esta mesma classe pode estar mapeada em uma DLL. Esta característica reflexiva permite a independência de linguagem de programação para a implementação de módulos e conectores.

A forma geral da cláusula ***map*** é:

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

onde:

tipo: dependendo do tipo de componente, 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 capaz de 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*, RMI, um estilo de conector, ou alguma alternativa similar ao caso do módulo.

linguagem: indica a linguagem de programação utilizada.

referência: nome do arquivo ou a URL [237] com a referência do elemento que está sendo utilizado.

Um compilador para CBabel também pode utilizar as informações das

declarações *map* para gerar código automaticamente, se isto for solicitado explicitamente, ou se o "alvo" do mapeamento não existe em um repositório. Por outro lado, cabe ao serviço de gerência de configuração identificar a forma adequada de combinar estas informações para efetivamente instanciar o módulo ou conector.

A.2.5.1 Funcionamento da cláusula *map*

A cláusula *map* produzirá efeito na aplicação em tempo de carga ou quando houver reconfiguração. As informações da cláusula *map* serão usadas para comandar execução dos métodos da API de configuração, que tratam da criação de instâncias. Observe-se, por exemplo, o trecho de código da listagem A.9, que descreve uma classe de módulo *AC* e indica uma referência para uma instância *m1* desta classe.

```
1   module AC () {
2      in port A;
3      in port B;
4      map Class Java "rrio.examples.teste.tserv";
5   }
6   instantiate AC as m1;
```

Listagem A.9 - Módulo com cláusula *map*

Durante a carga da aplicação, as informações da declaração de instanciação em CBabel serão utilizadas para a montagem do comando de instanciação. Por exemplo, no protótipo do ambiente de configuração desenvolvido em Java (capítulo 7) a invocação da API, para a instanciação do módulo *m1* seria:

```
configManag.instantiate (rrio.examples.teste.teserv, m1)
```

Este comando solicita a criação de uma instância, chamada *m1*, de uma classe denominada *testserv*, programada em Java, que se encontra em *rrio.examples.teste.teserv.class*. A informação de que *m1* é uma instância de módulo da classe *AC* é também armazenada no repositório de informações de meta-nível do Gerente de Configuração. No caso da implementação em Java, as informações relevantes são o nome do objeto e a classe que deve ser carregada.

Para os conectores, a cláusula *map* no nível da configuração produz efeitos no momento em que uma classe ou instância de conector é referenciada em uma ligação. Quando o gerente de configuração precisa carregar um conector, a cláusula *map* é consultada, e então, o conector é carregado e iniciado. O caso do conector difere do dos

módulos porque o Gerente de Configuração pode decidir o que fazer se a cláusula *map* não estiver presente, acionando um conector *default*. Isto não acontece no caso dos módulos, porque o Gerente de Configuração não tem como inferir sobre a implementação de aspectos funcionais.

A.3 Configuração de aplicação

Após a declaração de módulos e conectores, a interligação destes elementos será descrita para formar a arquitetura. CBabel oferece um conjunto de declarações de configuração para ser utilizado na descrição da configuração dos elementos em uma aplicação. Este conjunto satisfaz o modelo de gerência de configuração de R-RIO (seção 4.3 - capítulo 4), mas está associado ao nível de arquitetura de *software*. Seu efeito é declarativo, indicando a composição dos elementos da arquitetura de *software* sendo descrita, e não ações ou comandos de configuração que devem ser executadas.

O serviço de gerência de configuração, por outro lado, implementa um conjunto de primitivas de configuração, semelhante ao conjunto de declarações de configuração de CBabel, disponíveis através de uma API. Esta API será usada para materializar a arquitetura, quando uma aplicação é carregada para execução ou durante reconfigurações (capítulo 7). Em verdade, as declarações de configuração, deverão ser convertidas em comandos de configuração para que sejam efetivamente realizadas.

A.3.1 Instâncias de módulo (instantiate)

Esta declaração indica que uma instância de módulo deve ser criada (seção A.2.3.2). Seu correspondente na API de configuração aloca os espaços de memória necessários, carrega o código módulo (indicado na cláusula *map*) e registra a existência do módulo no repositório de informações de meta-nível, no ambiente de execução.

Sintaxe:

```
instantiate <module-type> as <module-name> [at <node>]
instantiate <node> at <ip address>
```

A.3.2 Interconexão de módulos e conectores (link)

A declaração ***link*** permite indicar as interconexões dos componentes de uma

aplicação. 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 a serem ligadas precisam ser complementares. Se uma porta ***out*** passa um parâmetro com tipo ***int*** o módulo parceiro deve ter uma porta ***in*** esperando um argumento do tipo ***int*** para que possa haver a ligação, como na listagem A.10:

```
...
// declarado no módulo cliente
out string port (int request_output) pedido;
...
// declarado no módulo servidor
in string port (int request_input) resposta;
```

Listagem A.10 - Portas complementares

Se a ligação das portas de dois módulos for declarada sem referência a um conector intermediário, o Gerente de Configuração seleciona um configurador *default* (veja a discussão na seção 4.2 - capítulo 4) para realizar a interação das mesmas. Uma outra possibilidade é a ligação entre dois módulos sem a referência das portas. Neste caso, todas as portas de tipos complementares ou compatíveis são conectadas automaticamente por contexto.

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>
```

A.3.2.1 Ligação por contexto

Esta forma interliga todas as portas dos módulos por nome e assinaturas das portas (exemplos na listagem A.11). Uma política para a ligação automática por contexto é descrita em [236]. Nesta política procura-se ligar o maior número possível de portas destes módulos. As ligações são feitas, com base em regras simples que verificam se as portas podem ser consideradas complementares.

```
1  link modA to modB;
2  link modA, modZ to modB
3  link modA to modB, modC
4  link modV1 [0..5] to modV2 [6..10]
```

Listagem A.11 - Ligação por contexto

A.3.2.2 Ligação porta-a-porta

Indica-se explicitamente a porta de saída / entrada dos módulos a serem ligados. Isto permite que o conector que interliga os pares de portas seja selecionado especificamente para esta ligação. Exemplos na listagem A.12.

```
1   link modA.portA to modB.portL
2   link modA.pN to modB.pJ, modC.pK
```

Listagem A.12 - Ligação porta-a-porta

Se a ligação for feita explicitando-se a porta de apenas um dos módulos, a outra porta, cuja referência foi omitida, é selecionada por contexto e apenas este par de portas é ligado com este comando.

```
1   link modA.portX to modB;     // liga a porta portX do módulo modA ao módulo modB. As outras
                                 portas não são ligadas
```

Listagem A.13 - Ligação de porta por contexto

A.3.2.2 Indicando-se o conector

Em CBabel, uma ligação pode ser descrita, indicando-se a classe de um conector ou uma referência para uma instância. No primeiro caso, o gerente de configuração instancia os recursos necessários para que os módulos sejam ligados por um conector de uma classe determinada, mas não se pode exigir exclusividade ou se fazer previsão de escalabilidade, pois não se indicou uma instância de conector (listagem A.14).

```
1   connector CtypeA {}
2   connector CtypeB {}

3   link modA.portA to modB.portX by CtipoA
4   link modA, modB to modX by CtipoB
```

Listagem A.14 - Ligação de porta por contexto

No segundo caso, indicando-se uma referência para uma instância de conector, cada instância pode conter recursos que serão usadas com exclusividade pela mesma. Desta forma é possível indicar-se explicitamente que duas ligações distintas serão intermediadas por conectores diferentes, da mesma classe (listagem A.15).

```
1   connector CtypeA {} C1, C2;     // duas instâncias

2   link modA to modB by C1         // C1 e C2 são da mesma classe, mas instancias diferentes
3   link modC to modD by C2
```

Listagem A.15 - Ligação com indicação do conector

Seria possível incluir em CBabel outras variações sintáticas para a ligação de módulos, com o objetivo de facilitar a descrição de arquiteturas mais complexas. Um estudo sobre algumas destas possibilidades é apresentado em [238].

A.3.3 Visibilidade das portas (export)

Esta declaração é utilizada para indicar-se a visibilidade das portas de um módulo ou conector composto (seções 4.2.4 e 4.2.5, capítulo 4) por outros módulos da aplicação. Em outras palavras, em um componente composto, apenas as portas exportadas terão visibilidade externa, as demais são escondidas (como se dá com a diretiva *hide* de CSP [239]). A "ligação" é efetuada sempre do módulo ou conector componente para o componente composto e deve ser declarada no escopo do componente composto. Exemplos de sua utilização serão apresentados na seção A.5.

Sintaxe:

```
export <port>
```

A.3.4 Iniciando a execução de um módulo (start)

Assinala o início da execução de uma instância de módulo localizado no nó indicado por sua URL. Em CBabel esta declaração é usada para determinar a ordem do início da execução dos módulos. Para o serviço de gerência de configuração, este será um comando para que se inicie o módulo efetivamente.

Sintaxe:

```
start <instância-de-módulo> [at <nó>]
```

A.3.5 Bloqueando um módulo (block)

Utiliza-se esta declaração para bloquear a execução de uma instância de módulo em um nó. O comando *block* pode ser usado em um *script* CBabel ou, diretamente, em uma reconfiguração. O programador deve estar ciente de que, ao fazer uso deste comando de configuração, a aplicação pode entrar em um estado inconsistente. Por exemplo, um módulo pode ser bloqueado dentro de uma região crítica. Isto poderia levar a aplicação ao *deadlock* até que este seja desbloqueado.

Sintaxe:

```
block <instância-de-módulo> [at <nó>]
```

A.3.6 Retomando a execução de um módulo (resume)

A declaração *resume* indica que a execução de uma instância de módulo, que estava bloqueada, deve continuar.

Sintaxe:

```
resume <instância-de-módulo> [at <nó>]
```

A.3.7 Removendo um módulo da arquitetura (remove)

A declaração *remove* retira uma instância de módulo localizada em determinado nó do ambiente de execução. A referência a este módulo é apagada do repositório de informações de meta-nível.

Sintaxe:

```
remove <instância-de-módulo> [at <nó>]
```

A.3.8 Grupos de módulos

CBabel suporta a configuração de arquiteturas que envolvem grupos de módulos. Grupos de módulos são indicados através de listas de módulos entre parênteses e podem ser identificadas por um identificador CBabel. A admissão de novos membros e a saída de membros de um grupo também pode ser descrita. A interação de módulos, membros de um grupo, deve ser realizada por conectores que suportam difusão. A sintaxe para a utilização de grupos em CBabel é apresentada na seção A.8.

A.4 Contratos para aspectos não-funcionais

Em CBabel adotou-se uma sintaxe específica para descrever cada visão de contrato de aspectos não-funcionais. As visões de contrato para interação, distribuição e coordenação possuem características particulares, que nos levaram a esta opção.

A.4.1 Contratos de Interação

Em CBabel, as interações entre os módulos são definidas pela declaração de ligação (*link*). Isto é suficiente para a criação da arquitetura de interligação dos módulos. O estilo de interação entre duas portas também pode ser síncrono ou assíncrono. Entretanto, contratos de interação podem indicar, por outro lado, como uma requisição é encaminhada dentro do conector. Isto permite especificar como as portas do conector são "coladas", explicitando como ocorre o fluxo das requisições e respostas durante as interações mediadas pelo mesmo. No caso trivial, portas de entrada e saída complementares do conector formam o caminho "natural" do fluxo das requisições, sendo estas apenas encaminhadas através desta rota. Em um caso mais complexo, o conector pode ser concebido para redirecionar o fluxo para outros elementos, antes de encaminhar uma requisição/resposta para a porta do módulo destinatário.

A informação dos contratos de interação são importantes para um compilador ou outras ferramentas de verificação. Este tipo de informação é também chamado de *roles & glue* (papéis e cola) em trabalhos fundamentados em notações de sistemas de transições rotuladas [72]. Em R-RIO, os contratos de interação facilitam a obtenção de uma rede de Petri, a partir de uma descrição em CBabel (apêndice B). Adicionalmente, atividades tais como a composição de conectores e a geração de código também podem usar este tipo de informação.

- **default**

O contrato trivial de interação contido em um conector tem uma semântica equivalente à política para ligações por contexto entre módulos. Portas com mesmo nome ou assinaturas e sentidos opostos (*in* e *out*) são consideradas ligadas, ou seja, os argumentos passados para a porta de entrada do conector devem ser encaminhados para a porta de saída, o mesmo acontecendo para o retorno, mas em sentido contrário (veja exemplo da figura A.1).

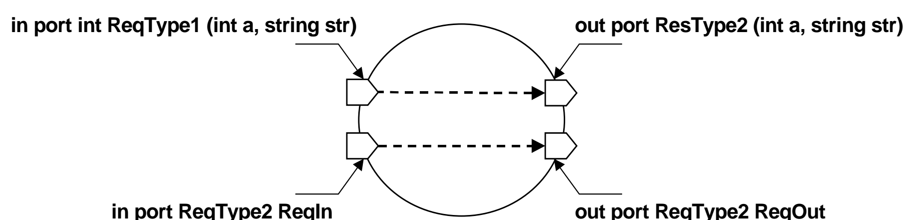


Figura A.1 - Contrato de interação *default*

Se nenhuma informação adicional for fornecida, um contrato *default* é admitido para efeito de configuração.

- **seqüencial**

A fim de se definir um fluxo específico das informações de uma interação, descreve-se a seqüência em que as portas serão utilizadas. ***Interaction contract*** é utilizado para descrever o contrato. Para o exemplo da figura A.1, se fosse necessário explicitar o contrato entre as portas *ReqIn* e *ReqOut*, este seria expresso da seguinte forma:

```
Interaction contract {
   ReqIn > ReqOut;
}
```

A notação com o sinal de maior, '>', indica que o fluxo de informações se daria primeiro recebendo-se (porta de entrada) uma requisição por *ReqIn*, encaminhando-se a mesma para *ReqOut*, que por sua vez a envia (porta de saída) ao módulo ligado a esta. Se as portas possuem um parâmetro de retorno, isto indica que a interação é síncrona e que uma resposta deve ser aguardada, e seu fluxo vai seguir o caminho inverso. Caso contrário, a interação termina.

- **paralelo**

Em algumas configurações, pode ser necessário especificar que o fluxo de uma requisição será encaminhado concomitantemente para duas ou mais portas. Neste caso, utiliza-se a notação ' | '.

```
Interaction contract {
   ReqIn > (ReqOutA | ReqOutB);
}
```

No trecho de código anterior, indica-se que uma requisição será recebida pelo porta *ReqIn*, que a encaminha para *ReqOutA* e *ReqOutB*, de forma paralela ou concorrente. Se a interação é síncrona, a implementação do conector deve prever uma forma para que a porta *ReqIn* receba uma única resposta, para encaminhá-la para o módulo.

A.4.2 Contratos de Distribuição

Durante a execução de uma aplicação, módulos componentes podem ser

instanciados em nós distribuídos. A sintaxe para a configuração do aspecto de distribuição é bastante simplificada. Na realidade, já foi apresentada juntamente com o comando de instanciação de módulos.

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

- os módulos são instanciados no mesmo nó, possivelmente onde a configuração está sendo descrita, ou
- os módulos serão distribuídos por nós disponíveis, segundo alguma política (nós com menor carga de processamento, por exemplo),

Se a arquitetura necessitar, de uma forma explícita, indicar onde os módulos devem ser instanciados, basta então que se façam as declarações de instanciação com a extensão de localização, `at` *<nó>*, indicando-se o nó onde a referida instância será criada pelo Gerente de Configuração .

Observação

A localização de um componente de uma arquitetura não faz parte, em princípio, de sua funcionalidade. Assim sendo, este aspecto é considerado não-funcional. Entendeu-se, no entanto, que este aspecto é utilizado com freqüência e a sintaxe para descrevê-lo, junto da instanciação do módulo, torna-se apropriada.

A.4.3 Contratos de Coordenação

Para descrever contratos de coordenação em CBabel, optou-se por uma *linguagem de aspectos* integrada. Através desta linguagem descrevem-se a concorrência e sincronização, conforme o modelo de coordenação descrito na seção 5.3.3.

A linguagem do aspecto de concorrência permite declarar guardas de sincronização e o grau de concorrência de um conjunto de portas.

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

```
1   exclusive {
2      <lista-de-portas>
3   }
```

Listagem A.16 - Declaração *exclusive*

selfexclusive - indica se um método não pode ser executado concorrentemente. Se uma porta não for marcada como *selfexclusive*, várias requisições para este método podem executar concorrentemente.

```
1   selfexclusive {
2      <lista-de-portas>
3   }
```

Listagem A.17 - Declaração *selfexclusive*

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      <lista-de-portas>
3   }
```

Listagem A.18 - Declaração *concurrent*

guard - guardas são definidos nas portas de saída de um conector, pois estas serão interligadas a portas de entrada de um módulo, onde o guarda precisa efetivamente atuar.

No conector onde será declarado um guarda, é necessário definir-se uma lista de **variáveis de condição** que serão usadas para avaliar se este guarda está aberto ou fechado (listagem A.19, linha 2). É preciso, também, indicarem-se as variáveis que definem o estado do módulo onde os guardas atuarão. Estas variáveis são inspecionadas neste módulo (listagem A.19, linha 3), em tempo de execução. Cabe ao programador saber quais são as variáveis a serem usadas.

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

Listagem A.19 - Declaração de variáveis de condição e consulta ao estado

Cada porta de saída de um conector pode definir um guarda (listagem A.20). A definição de um guarda possui 3 partes:

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

- uma expressão, que pode conter as variáveis de condição e variáveis declaradas na expressão *staterequired*, e outras variáveis auxiliares. Esta expressão é avaliada antes da execução do método (cláusula *before*);
- uma expressão, que pode conter as variáveis de condição e variáveis declaradas na expressão *staterequired*, e outras variáveis auxiliares. Esta expressão é avaliada após a execução do método (cláusula *after*);

A avaliação da expressão declarada nas cláusulas *before* e *after* pode alterar o valor das variáveis de condição, fazendo o guarda abrir ou fechar.

```
1   out port {
2   // declara um guarda
3      guard (<expressão-variáveis-de-condição>) {
4         before {<expressão-composta>};
5         after {<expressão-composta>};
6      }
```

Listagem A.20 - Declaração de guarda

Um guarda pode ser configurado, adicionalmente, com um prazo máximo de espera para que uma condição seja satisfeita (listagem A.21). Neste caso, uma porta alternativa é configurada para que a requisição que estava bloqueada no guarda seja enviada no caso (i) do tempo expirar ou no caso (ii) do guarda estar fechado e nenhuma cláusula de temporização ter sido declarada. Este esquema permite tratar situações em que não possa esperar indefinidamente por determinada condição, e facilitar um tratamento alternativo para as requisições ou exceções. 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.

```
1   guard (<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 A.21 - Declaração de guardas com *timeout*

O próximo exemplo ilustra o uso de guardas. Seja o módulo da listagem A.22 e um conector que interliga este módulo a outros módulos (listagem A.23).

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

Listagem A.22 - Módulo Buffer

```
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 A.23 - Conector Request

A figura A.2 esquematiza a configuração desejada.

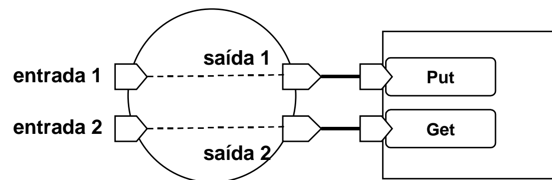


Figura A.2 - Conector com duas portas de entrada e saída

Supondo que o método *Put* do módulo *BB* necessite de um guarda, isto será configurado no conector 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          guard OkToPut {             // guarda de entrada
7             before {expressão};
8             after  {expressão};
9          }
10      } saida1;                      // porta com guarda
11
12      out port saida2;               // porta não guardada
13   } GRQ;
```

Listagem A.24 - Conector com um guarda

A figura A.3 esquematiza a nova configuração.

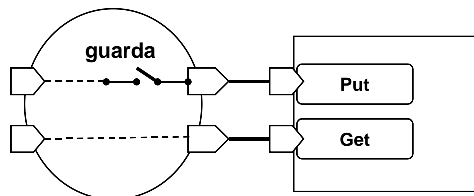


Figura A.3 - Conector com uma porta de saída com guarda

Vale lembrar que a declaração *guard* é seguida de uma expressão booleana, que contém 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. Estas expressões podem envolver variáveis de condição, variáveis declaradas no conector ou variáveis declaradas na cláusula *staterequired*.

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

Listagem A.25 - Conector com guarda

Observe o exemplo da listagem A.25 onde um conector com um guarda é descrito. 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 última utiliza a variável de condição *Vazio* e a variável inteira *UsedSlots* declarada na cláusula *staterequired*, que é inspecionada no módulo.

A.5 Composição de módulos

A descrição de aplicações e módulos compostos na linguagem de configuração é feita de forma simples. O exemplo da figura 4.4, apresentado no capítulo 4, pode ser descrito conforme aparece na listagem A.26.

A declaração das classes de módulos e conectores é feita fora do escopo de qualquer outro módulo (linhas 1 a 3), possibilitando que outras instâncias destas classes sejam criadas em qualquer ponto. Logicamente, a visibilidade da instância depende de onde a mesma foi declarada. Por exemplo, a visibilidade de Mp1, Mp2, Mp3 é interna a MC1, pois estas instâncias foram declaradas dentro do corpo deste (linhas 6 a 8).

```
1   module T1 {}                    // declara 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 A.26 - Descrevendo um módulo composto

A.6 Composição de conectores

A descrição de um conector composto é semelhante à de um módulo composto. O exemplo da seção 4.3.7, capítulo 4, será utilizado para ilustrar como a composição de conectores é descrita em CBabel.

```
1   connector Ccomposto {
2    connector { } α;                 // declara instâncias dos conectores componentes
3    connector { } β;
4    connector { } χ;
5    connector { } δ;
6    connector { } ε;
7    connector { } φ;
8    connector { } γ;
9
10   LINK α.portaS1 to β.portaE1;     // link permite ligação entre portas de conectores
11   LINK α.portaS2 to χ.portaE1;
12   LINK α.portaS3 to δ.portaE1;
13   LINK α.portaS4 to ε.portaE1;
14   LINK χ.portaS1 to φ.portaE1;
15   LINK ε.portaS1 to γ.portaE1;
16
17   export α.portaE1;                // exporta as portas com visibilidade externa
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  link X to Y by CC;               // ao se instanciar um conector composto, todas as instâncias
29                                   // de conectores componentes são automaticamente criadas .
```

Listagem A.27 - Descrição de um conector composto

Na listagem A.27 uma classe de conector composto, *Ccomposto*, e uma instância desta classe, *CC*, são descritas. Da linha 2 à linha 9 as instâncias dos conectores componentes são declaradas. Em seguida, são interligadas as portas dos conectores através de declarações *link*. Os nomes das portas utilizadas são ilustrativos, já que a declaração dos conectores não apresenta, nesta listagem, este nível de detalhe.

O passo seguinte 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 no momento em que ele é utilizado pela primeira vez em uma ligação. Neste ponto, o conector poderá ser usado pela aplicação. Dentro do conector composto, não foram criadas as instâncias dos conectores declarados. Quando a criação da instância do conector composto for executada, o serviço de gerência de configuração se encarregará de criar as instâncias dos conectores componentes, na ordem em que foram declaradas.

A.6.1 Notação compacta

A notação para a descrição de conectores compostos permite especificar todo tipo de composição. Como alternativa para descrever composições mais simples, seria conveniente oferecer uma notação mais compacta. Para esta notação, admite-se que a composição pode ser seqüencial ou paralela.

A.6.1.1 Composições seqüenciais

Composições seqüenciais de conectores permitem indicar o uso de conectores encadeados em uma única ligação:

```
link modA to modB by Crypt > Socket > Decrypt
```

A declaração de ligação anterior indica que os módulos *modA* e *modB* serão ligados pela seqüência de conectores *Crypt*, *Socket* e *Decrypt*. Isto quer dizer que uma requisição, partindo do módulo *modA* e chegando ao conector *Crypt* (porque partiu de uma porta de saída de *modA*, ligada a uma porta de entrada de *Crypt*), é repassada para o conector *Socket*, que, por sua vez, repassa-a ao conector *Decrypt* para, então, ser encaminhada ao módulo modB. O retorno enviado por *modB* faz o caminho contrário (figura A.4).

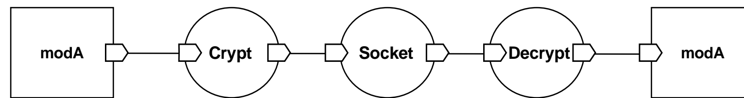


Figura A.4 - Cadeia seqüencial de conectores

Observa-se que, no exemplo, a ligação foi realizada por contexto, mas seria possível indicar portas específicas dos módulos *modA* e *modB* para serem ligadas por esta composição de conectores.

A.6.1.2 Composições paralelas

Composições paralelas simples poderiam ser resolvidas sintaticamente pela indicação de conectores separados por vírgulas e pelo contexto dos módulos sendo interligados. Vejamos o exemplo da figura A.5.

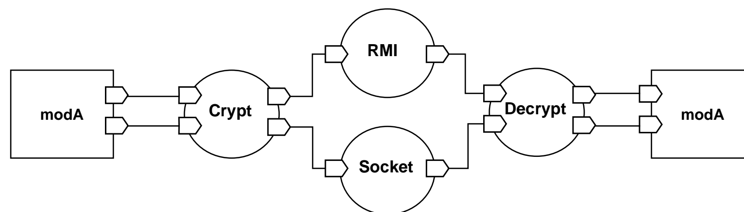


Figura A.5 - Combinando composição seqüencial e paralela

A ligação dos módulos nesta configuração pode ser descrita da seguinte forma:

```
link modA to modB by Cript > (RMI | Socket) > Decript
```

Observa-se que as portas não foram indicadas na declaração de ligação, pois as ligações são feitas por contexto. Não seria possível, com esta sintaxe compacta, indicar quais portas específicas iriam utilizar o conector *RMI* ou o conector *Socket*. Assim sendo, se a ligação não puder ser feita por contexto, o estilo padrão para declaração de conectores compostos, como apresentado no início desta seção, deve ser utilizado.

Observação

Existem alguns problemas na composição de conectores que devem ser mais amplamente discutidos, tais como a combinação de conectores com implementação distribuída e não-distribuída:

- Como ficariam encadeadas as partes distribuídas do conector?
- Onde ficariam localizados os conectores com implementação não-distribuídas e as partes dos conectores com implementação distribuída?
- É possível deixar para o Gerente de Configuração a decisão de como fazer a composição?

Na listagem A.28, por exemplo, apresenta-se uma parte da descrição de uma arquitetura, em que um módulo *buffer* precisa ter seus métodos sincronizados. Nesta arquitetura, existem produtores e consumidores que estão distribuídos por outros nós. Assim sendo, utiliza-se uma composição com conectores *socket* e um conector *guarda*.

```
1  instantiate produtor at est1
2  instantiate consumidor at est1
3  instantiate buffer at est3
4  link produtor, consumidor to buffer by guarda > socket // errado
5  link produtor, consumidor to buffer by socket > guarda // correto
```

Listagem A.28 - Descrevendo uma composição de conectores

A primeira ligação (linha 4) resultaria em uma arquitetura configurada erradamente. Um conector *guarda* distinto estaria sendo instanciado nos nós do *produtor* e *consumidor.* Em seguida, um conector *socket* também seria instanciado com uma metade cliente, em cada um dos nós clientes, e uma metade servidor, no nó do *buffer*. A segunda ligação está correta (linha 5). Uma metade cliente do conector *socket* é instanciada no nó do *produtor*, outra metade cliente, no nó do *consumidor* (isso é determinado automaticamente pela natureza distribuída do conector). A metade servidor do conector *socket* é encadeada com o conector guarda, no nó do módulo *buffer*.

Em [240], restrições para a instanciação de conectores compostos é discutida mais amplamente e possíveis soluções são apontadas.

A.7 Herança

Em CBabel, reusabilidade e extensibilidade da configuração de uma arquitetura são obtida por composição. Pode-se estender a funcionalidade de módulos e conectores elementares através de composição, conforme mencionado anteriormente, criando-se um módulo ou conector mais especializado, ou que agregue a funcionalidade desejada. A composição ou agregação são consideradas mecanismos legítimos para implementar

a herança [34]. Uma das vantagens de uso de composição, em relação aos esquemas tradicionais de herança em linguagens orientadas a objetos, é que se evitam com mais facilidade os problemas de anomalia de herança [167] e fragilidade de classe-base [241], que dificultam o reuso de *software*. Além disso, usando-se a composição como mecanismo para implementar herança nos conectores, preserva-se o encapsulamento e a ortogonalidade de aspectos funcionais e não-funcionais e maximiza-se a separação de interesses.

A.8 BNF

O BNF para CBabel é apresentado a seguir. A forma de apresentação foi simplificada, omitindo-se detalhes como, por exemplo, as produções para a declaração de tipos básicos e expressões compatíveis com a OMG IDL 2.0. Também para simplificar o BNF, convencionou-se que:

- os terminais e literais estão em **negrito** ou *itálico*;
- sempre que houver um lexema do tipo *nome*-**list** no lado direito de uma produção, fica implícito que existe uma outra produção na forma:

```
nome-list        :   nome-list  nome
                 |   nome
                 |   branco
                 ;
```

- sempre que houver um lexema do tipo *nome*-**Commalist** no lado direito de uma produção, fica implícito que existe uma outra produção na forma:

```
nome-list        :   nome-list ',' nome
                 |   nome
                 |   branco
                 ;
```

- quando houver um lexema terminado por *Name*, como, por exemplo, *moduleClassName*, fica implícito que existe uma produção da forma

```
Name             :   identifier
```

Arquitetura

```
elements            :   element-list

element             :   module_dcl
                    |   port_dcl
                    |   connector_dcl
                    |   link_dcl
                    |   node_dcl
                    |   instatiate_dcl
                    |   start_dcl
                    |   for_dcl
                    |   const_dcl
                    |   type_dcl
                    |   QoS_dcl
                    |   branco
                    ;
```

Portas, Módulos, Conectores

```
module_dcl          :   module moduleClassSig moduleBody instanceName-Commalist ’;’
moduleClassSig      :   moduleClassName parms
moduleBody          :   ’{’
                        element-list
                    |   export_dcl-list
                    |   map_dcl
                    |   QoSprofile_dcl
                        ’}’
                    |   branco


port_dcl            :   portAttr port portTypeSig portBody instanceName-Commalist ’;’
portAttr            :   in
                    |   out
                    |   branco

portTypeSig         :   returnAttr portTypeName parms
portBody            :   ’{’ guard_dcl ’}’
                    |   branco
returnAttr          :   type_spec
                    |   void
                    |   oneway

connector_dcl       :   connector connectorClassSig connectorBody instanceName-Commalist ’;’
connectorBody       :   ’{’
                        element-list
                    |   export_dcl-list
                    |   map_dcl
                    |   interactContract_dcl
                    |   exclusive_dcl-list
                    |   selfExclusive_dcl-list
                    |   concurrent_dcl-list
                    |   QoScontract_dcl
                        ’}’
                    |   branco

connectorClassSig   :   connectorClassName parms

map_dcl             :   map identifier identifier ’"’ externName ’"’ ’;’
```

Configuração

```
instantiate_dcl      :  instantiate moduleComplRef ’;’
                        instantiate moduleClassName as moduleComplRef ’;’
link_dcl             :  link moduleComplRef to moduleComplRef connection ’;’
                     |  link groupName to groupName connection ’;’
                     |  link ’(’ moduleComplRef-Commalist ’)’
                                    to ’(’ moduleComplRef-Commalist ’)’ connection ’;’
export_dcl           :  export portName ’;’
node_dcl             :  node nodeName dotted_identifier ’;’
start_dcl            :  start moduleName location ’;’
block_dcl            :  block moduleName location ’;’
resume_dcl           :  resume moduleName location ’;’
remove_dcl           :  remove moduleName location ’;’

for_dcl              :  for identifier ’=’ init to end forBody
forBody              :  conf-dcl
                     |  ’{’ conf-dcl-list ’}’

conf-dcl-list        :  conf-dcl-list conf-dcl
                     |  conf-dcl

conf-dcl             :  export_dcl | link_dcl | instantiate_dcl | start_dcl | block_dcl
                     |  resume_dcl | remove_dcl | for_dcl
init                 :  NUMBER
end                  :  NUMBER
```

Suporte a Grupo

```
group_dcl            :  groupName ’=’ ’(’ moduleComplRef-Commalist ’)’ ’;’
join_dcl             :  join moduleComplRef to groupName ’;’
leave_dcl            :  leave moduleComplRef from groupName ’;’
```

Contratos de Interação

```
interactContract_dcl :  Interaction contract ’{’ interaction-expr ’}’

interaction-expr     :  interaction-relation
                     |  ’(’ interaction-relation ’)’
                     |  interaction-relation interaction-opr interaction-relation
                     |  branco
interaction-relation :  portName interaction-opr portName
interaction-opr      :  ’>’
                     |  ’|’
```

Contratos de Coordenação

```
exclusive_dcl        :  exclusive ’{’ port_dcl-Commalist ’}’
selfExclusive_dcl    :  selfexclusive ’{’ port_dcl-Commalist ’}’
concurrent_dcl       :  concurrent ’{’ port_dcl-Commalist ’}’
guard_dcl            :  guard ’(’ condition-expr ’)’ ’{’ guardBody ’}’
guardBody            :  timeout_dcl
                     |  alternative_dcl
                     |  before_dcl
                     |  after_dcl
                     |  branco
```

timeout_dcl : **timeout** ’(’ time ’)’ ’;’
alternative_dcl : **alternative** ’(’ portName ’)’ ’;’
before_dcl : **before** ’{’ comp-expr ’}’
after_dcl : **after** ’{’ comp-expr ’}’
time : *NUMBER*

Contratos de QoS

QoS_dcl : QoScategory_dcl

QoScategory_dcl : **QoScategory** QoScategoryName ’{’ QoSpropertyType-list ’}’

QoSpropertyType: : propertyName ’:’ propertyType propertyUnit
;
propertyType : **enum** **’**{’ identifier-Commalist ’}’
| relSem **’**{’ identifier-Commalist ’}’ **with** order-dcl
| relSem **numeric**

propertyUnit : identifier
| <u>branco</u>
relSem : **decreasing**
| **increasing**
order-dcl : **order** ’{’ order-seq ’}’
order-seq : identifier **’<’** identifier
| identifier **’<’** order-seq

QoSprofile_dcl **QoSprofile** QoSprofileName **for** complRef ’{’ QoSservice-list ’}’

QoSservice : QoSrole QoScategoryName ’{’ QoSpropConstraint-list ’}’
QoSrole : **provide**
| **require**
QoSpropConstraint : propertyName constraint-opr propertyValue ’;’

propertyValue : **’**{’ identifier-Commalist ’}’
| identifier
| *NUMBER*
constraint-opr : ’==’ | ’>’ | ’<’ | ’>=’ | ’<=’

QoScontract_dcl : **QoScontract** ’{’ moduleInstancePair-Commalist ’}’ QoSExPolicy ’;’

moduleInstancePair : ’(’ moduleName ’,’ moduleName ’)’
QoSExPolicy : **abort** | **retry** | **renegotiate**

Expressões

condition-expr : boolean-expr
comp-expr : assig_stmt-list
| if-then-else_stmt-list
;
if-the-else_stmt : **if** ’(’ boolean-expr ’)’
then ’{’ comp-expr_list ’}’
else ’{’ comp-expr_list ’}’
assig_stmt : identifier ’=’ const-expr ’;’
boolean-expr : OMG IDL 2.0
const-expr : OMG IDL 2.0

Básico

```
dotted_identifier      :  dotted_identifier'.'identifier
                       |  identifier
moduleComplRef         :  moduleId  location
                       |  moduleId'.'portName  location
connectorCompRef       :  connectorId
                       |  connectorId'.'portName
moduleId               :  moduleName index args
connectorId            :  connectorName  index  args
                       |  connectorClassName  args
location               :  at  nodeName
                       |  branco

connection             :  by  connectorName
                       |  by  compositeConnector
                       |  branco
compositeConnector     :  connector-compos
                       |  '('  connector-compos  ')'
                       |  connector-compos  composition-opr  connector-compos
                       |  branco
connector-compos       :  connectorID  interaction-opr  connectorID
composition-opr        :  '>'
                       |  '|'
index                  :  '['
                       |  identifier
                       |  NUMBER
                          ']'
                       |  branco
args                   :  '('  arg-Commalist  ')'
                       |  branco
arg                    :  type_inst
parms                  :  '('  parm-Commalist  ')'
                       |  branco
parm                   :  type_spec  identifier
const_dcl              :  type_spec  identifier  '='  const_expr
type_dcl               :  type_spec  identifier  ';'
identifier             :  STRING
type_spec              :  primitive-type
                       |  complex-type
type_inst              :  primitive-type-inst
                       |  complex-type-inst
primitive-type         :  OMG IDL 2.0
complex-type           :  OMG IDL 2.0
primitive-type-inst    :  OMG IDL 2.0
complex-type-inst      :  OMG IDL 2.0
```

A.9 Conclusão

As declarações em CBabel podem ser divididas em dois grupos. O primeiro é responsável pela declaração das classes, tipos e instâncias dos elementos a serem usados na arquitetura de *software* de uma aplicação. Neste grupo estão incluídas as declarações *module*, *port* e *connector*. O segundo grupo permite que se descrevam as interligações destes elementos e o momento de instanciá-los. As declarações *link*, *instantiate* e *start*

estão neste grupo. Embora a cláusula para a criação de uma instância possa ser considerada um comando imperativo, no contexto de CBabel ela indica a ordem em que as instâncias devem ser criadas. Por outro lado, no contexto da operação, as declarações deste segundo grupo são efetivamente utilizadas como comandos imperativos para o serviço de gerência de configuração. Este esquema foi proposto intencionalmente, pois facilita a preparação de roteiros de configuração e reconfiguração.

CBabel possui algumas semelhanças com outras ADLs, como UniCon, por exemplo. Ela se diferencia, entretanto, em vários pontos. O mais evidente é a configuração de aspectos não-funcionais. Não existe, em UniCon, o conceito de aspecto, apenas algumas características específicas, incluídas na linguagem, podem ser parametrizadas. Em CBabel, além da configuração de aspectos disponível na linguagem, novos tipos de conectores podem ser introduzidos através da cláusula *map*. Uma outra diferença diz respeito ao próprio mapeamento para uma implementação. Em UniCon, após a geração de código executável, as informações sobre a arquitetura da aplicação deixam de existir. Neste particular, nossa proposta é semelhante à Regis. A arquitetura de uma aplicação descrita em Darwin ainda pode ser referenciada durante a operação da mesma, 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 nossa proposta, o mapeamento entre elementos do nível de configuração e suas respectivas implementações é flexível: nenhum recurso é de uso obrigatório.

Uma outra observação sobre CBabel é sua compatibilidade com OMG-IDL. Embora a sintaxe não seja exatamente a mesma, uma arquitetura descrita em CBabel pode ser facilmente convertida para IDL. Além disso, o conjunto de tipos básicos e o esquema de visibilidade de CBabel é o mesmo da OMG-IDL. Isto permite que *stubs* e *squeletons*, para componentes CORBA, sejam gerados automaticamente, facilitando a integração destes componentes em arquiteturas descritas em CBabel.

Esta página foi intencionalmente deixada em branco

Apêndice B

Aspectos formais

B.1 Introdução

Como ressaltado na seção 2.3.4 - capítulo 2, um modelo formal pode ajudar a tornar mais claro o conceito de uma arquitetura de *software* e representar de forma não-ambígua a descrição da mesma a partir de uma ADL. Além disso, através de um modelo formal, é possível indicar de forma precisa como realizar refinamentos e verificações na arquitetura descrita.

Neste apêndice, são apresentados tópicos relacionados com aspectos formais de ASs, tomando-se o *framework* R-RIO como exemplo. Apresentamos, inicialmente, um modelo de base para sistemas concorrentes. Outros modelos, ainda formais, e baseados no modelo base, tais como as redes de Petri, são aplicados a componentes de R-RIO para estudar aspectos específicos. Apontamos também como podem ser realizadas algumas verificações em descrições de arquiteturas de *software* em CBabel, a partir dos modelos apresentados.

B.2 Modelo de Base

Para a especificação formal de uma arquitetura de *software*, é indicado o uso de uma abstração capaz de modelar todos os estilos de interação e coordenação entre componentes concorrentes. Esta abordagem, freqüentemente utilizada na literatura [254], torna possível o uso de qualquer linguagem de descrição (CBabel, por exemplo), sem particularizar a metodologia utilizada.

Uma das abstrações mais genéricas que podem ser utilizadas na especificação de uma arquitetura de *software* é o Sistema de Transição. Por isto, o Sistema de Transição

pode ser escolhido como modelo base. Entretanto, por ser muito abstrato e genérico, outros modelos intermediários, entre o modelo base e a ADL de R-RIO, podem ser empregados para examinar aspectos específicos, como será visto mais à diante.

B.2.1 Sistema de Transição

Um sistema concorrente, tal como uma arquitetura de *software*, pode ser visto como possuindo um estado global, que é modificado ao longo do tempo através de ações elementares (transições).

Definição B1 (Sistema de transição) Um sistema de transição (*ST*) é definido como uma ênupla S = (Π, Σ, T, Θ) onde:

Π : É o conjunto de variáveis $\{\upsilon \mid \upsilon \in \Pi\}$, de sortes s (υ) bem definidas, que representa o estado global do sistema.

Σ : É o conjunto de todas as interpretações possíveis das variáveis em Π, restritas às sortes dos respectivos tipos, isto é, $\Sigma = \{\sigma \mid \sigma : \Pi \rightarrow s(\upsilon)\}$.

T : É um conjunto de transições, onde caracterizamos cada $\tau \in T$ através de uma relação R_T sobre $\Sigma \times \Sigma$. R_T define as transformações de estado válidas como resultado da transição τ, correspondendo a uma função parcial $\tau : \Sigma \rightarrow \Sigma$.

Θ : É uma fórmula envolvendo elementos de Σ, que definem o estado inicial do sistema.

B.2.2 Computações

O ST define o estado inicial do sistema concorrente e, através do seu conjunto de transições T, podemos obter seqüências de estados válidas para o sistema. O que chamamos de uma execução possível do sistema, consiste da seqüência de estados que o sistema passa ao longo de sua evolução. Esta seqüência de estados é chamada de computação do ST.

Definição B2 (Computação) Dado um ST S = (Π, Σ, T, Θ), definimos uma computação de S como sendo uma seqüência de estados $\sigma = (\sigma_0, \sigma_1, \sigma_2, ...)$ onde $\sigma_i \in \Sigma$, para i = 0, 1, 2, ... Para uma computação de S, as seguintes condições devem ser satisfeitas:

1. σ_0 satisfaz Θ, Θ ¦ σ_0.

2. Para todo $i \geq 0$, $\exists \tau \in T \mid (\sigma_i, \sigma_{i+1}) \in R_\tau(\tau)$

Como definido acima, o conceito de computação não torna explícitos os aspectos de concorrência do sistema, e supõe que o estado é global. Apesar de sua simplicidade, a definição de computação dada acima é coerente com os modelos síncrono e assíncrono, para semântica de sistemas distribuídos [253]

B.2.3 Hipótese do Intercalamento

É possível simplificar ainda mais a abstração obtida com o uso do ST, restringindo uma mudança de estado, como sendo resultado da ocorrência de apenas uma das transições. Assim, se um conjunto de transições ocorrer concorrentemente, esta situação terá que ser representada por uma seqüência de transições.

Este tipo de representação de concorrência é geralmente referenciado como, *o uso da hipótese do intercalamento*. A adoção desta representação não reduz a acurácia do modelo utilizado, como pode ser visto em [254]. Mesmo utilizando a hipótese do intercalamento, todas as seqüências possíveis de um sistema concorrente real podem ser obtidas. Além disso, é possível que se obtenham outras seqüências, que não ocorreriam no caso anterior.

B.2.3.1 Limitação de Referência Crítica

Admite-se que, em uma computação de um dado sistema, um conjunto de transições pode ocorrer concorrentemente. Para capturar estas computações, a hipótese do intercalamento é relaxada, observando-se a seguinte restrição:

Definição B3 (LCR: Limitação de referência crítica) Uma transição satisfaz a restrição LCR se ela realizar apenas uma referência crítica a uma variável. Onde uma referencia é crítica se:

1. Tentar modificar uma variável que pode ser usada ou modificada em uma transição concorrente.
2. Tentar usar uma variável que pode ser modificada em uma transição concorrente.

No estado atual da tecnologia dos sistemas de computação, a restrição LCR é

respeitada, por definição, se a computação ocorre em apenas um processador, pois o acesso a uma posição de memória é sempre feito em exclusão mútua.

B.2.4 Sistema de Transição Justo

O emprego da hipótese do intercalamento simplifica a representação de sistemas concorrentes sem perda de informação, mesmo na presença de interferências entre os mesmos. No entanto, existem comportamentos coerentes com a hipótese do intercalamento, que não correspondem ao comportamento de alguns sistemas concorrentes reais. Teremos que lidar com o requisito de progresso independente que estes sistemas apresentam.

Segundo a hipótese do intercalamento, em uma computação é necessário, apenas, que transições habilitadas sejam continuamente escolhidas, modificando o estado correspondente do sistema. Nada impede que apenas um subconjunto das transições seja sistematicamente escolhido, levando à evolução de determinadas partes do sistema, enquanto nada acontece nas demais.

Para resolver este problema criou-se o conceito de justiça. Algumas definições diferentes podem ser associadas ao conceito de justiça, sendo o objetivo comum entre elas a restrição do comportamento do sistema modelado, de modo a se aproximar do comportamento de um sistema real.

O primeiro tipo de justiça considerado, garante a evolução de todas as partes concorrentes do sistema. Forçando que uma transição continuamente habilitada não deixe de ser eventualmente escolhida.

Definição B4 (Transições justas) Um conjunto de transições $J \subseteq \mathrm{T}$, é dito justo se ele possuir transições habilitadas continuamente, durante um intervalo de tempo suficientemente longo, não necessariamente as mesmas, que são eventualmente disparadas um número arbitrariamente grande de vezes.

O conceito de transição justa, apesar de garantir uma condição de progressão independente importante, não é suficiente para restringir o comportamento do modelo de forma satisfatória. Quando existe comunicação ou sincronização entre processos, um processo pode interferir no outro de modo que nenhuma transição do outro esteja continuamente habilitada. De fato, a mesma transição pode ter seu estado alternado

entre habilitado e desabilitado, sem, no entanto, ser executada. Apesar deste comportamento ser justo, de acordo com a definição acima, ele não corresponde à realidade. Define-se então um conceito de justiça mais forte.

Definição B5 (Transições equânimes) Um conjunto de transições $F \subseteq T$, é dito equânime se ele possuir transições que, estando habilitadas um número arbitrariamente grande de vezes, então estas também serão disparadas um número arbitrariamente grande de vezes.

Onde, por um número arbitrariamente grande de vezes queremos dizer:

Definição B6 (Arbitrariamente grande de vezes) Seja uma seqüência de estados $\rho = (\rho(s) \mid s \geq 0)$. Dizemos que t_i acontece um número arbitrariamente grande de vezes em ρ se para algum inteiro $K > 0$, e para todo $s \geq 0$ poderemos encontrar pelo menos um $j \leq K$ tal que $t_i \in \rho(s + j)$

Ou seja, para qualquer s, $\rho(j)$ é um elemento da subseqüência de ρ especificada por $\{\rho(s), \rho(s + 1), ..., \rho(s + K)\}$.

Transições equânimes são utilizadas para garantir justiça no comportamento do sistema associado a interação entre módulos

De acordo com o que foi dito anteriormente, computações que não atendem às condições de justiça podem ocorrer em sistemas reais, ou na sua representação por um ST. Entretanto, a ocorrência de tais computações podem não ser desejadas, e, ainda, pode não haver interesse em incluir as mesmas na realização de verificações. Assim, definiremos um modelo mais adequado para sistemas concorrentes, acrescentando condições de justiça ao conceito de sistema de transição.

Definição B7 (Sistema de transição justo) Um sistema de transição justo (STJ) é definido como uma ênupla $S = (\Pi, \Sigma, T, \Theta, J, F)$ composta de um sistema de transição, mais um conjunto J de transições justas, e um conjunto F de transições equânimes.

Restringimos, então, a definição de uma de computação válida para seqüências que também satisfazem as condições de justiça definidos no STJ correspondente.

B.3 Modelo em redes de Petri

Nesta seção, apresentamos um segundo modelo formal mais apropriado aos aspectos de concorrência e sincronização: as redes de Petri [242]. Estas redes têm sua semântica dada por um STJ (definição B7) e, por isso, podem ser consideradas como um modelo intermediário. Assim:

Π : É o conjunto de lugares $\{\lambda \mid \lambda \in \Pi\}$, cuja sorte é o conjunto de inteiros não negativos, isto é: o número de fichas em cada lugar, que representa o estado global do sistema.

Σ : São todas as marcações possíveis dos lugares em Π, tomadas em conjunto.

T : É um conjunto de transições $\{\tau \mid \tau \in T\}$. O disparo de uma transição define as transformações de estado válidas.

Θ : É fórmula que define a marcação inicial da rede.

J e F : São subconjuntos de T, representando as transições justas e equânimes, respectivamente.

Computações : São os ramos da árvore de alcançabilidade apresentam todas as computações possíveis de uma rede.

Por sua natureza gráfica, Redes de Petri facilitam o entendimento dos sistemas modelados. Além disso, estas podem ser analisadas por ferramentas automáticas para a verificação de propriedades, tais como: a ausência de *deadlock*, *liveness*, *safety*, entre outras. O modelo em Redes de Petri também pode ser usado como ponto de partida para outras técnicas de investigação formal, tais quais LOTOS [243], Estelle [244] ou SDL [245], e para o uso de ferramentas de *verificação do modelo* (*model checking*) [246, 247, 224].

B.3.1 Módulos e portas

Na figura B.1, estão representados dois módulos distintos, cada qual com uma porta, indicada pela transição *P*, e dois estados distintos, modelando o estado do módulo antes e depois de enviar ou receber uma requisição. No modelo da figura B.1, ainda não é indicado se as portas são de entrada ou de saída.

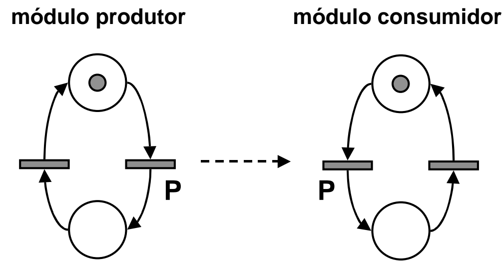


Figura B.1 - Dois módulos, com um método cada um, sem conexão

Uma porta é representada, neste modelo, por uma transição, como na figura B.2(a). Se a transição estiver representando uma porta de saída, seu disparo modela o envio de uma requisição através desta porta e, no caso da porta ser de entrada, este disparo representa o recebimento de uma requisição.

A ligação de uma porta de saída a uma porta de entrada, é representada pela fusão das duas transições (figura B.2(b)). Em CBabel esta fusão está implícita, ao se permitir que portas com assinaturas compatíveis sejam interconectadas pelo comando *link*. As duas transições são disparadas, agora, simultaneamente, modelando o envio, o recebimento, o processamento da requisição e a resposta da requisição, de forma síncrona e atômica.

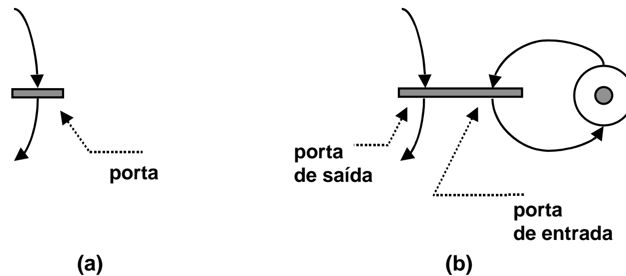


Figura B.2 - (a) representação de uma porta, (b) representação funcional de uma porta de saída interagindo com uma porta de entrada

B.3.2 Conectores

Antes de discutir-se como um conector é representado por uma rede de Petri, valem as seguintes observações. Ao se definir que uma porta de saída de um módulo cliente será ligada a uma porta de entrada de um módulo servidor, as referências destas portas passam a ser fundidas (como apresentado na figura B.2(b)). No exemplo da figura B.3(a), um módulo cliente está ligado a um módulo servidor. No nível funcional,

esta ligação não possui intermediários. Entretanto, no modelo de configuração de R-RIO, as interações entre as portas são intermediadas por um conector. Este conector está representado de forma pontilhada, indicando sua transparência neste nível. A figura B.3(b) apresenta a mesma situação observada agora pelo nível não-funcional ou meta-nível. Neste nível, o conector é materializado e pode ser referenciado. Neste momento passam a existir duas interações, síncronas, identificadas na figura B.3(c): uma, entre o módulo cliente e o conector, e outra, entre o conector e o módulo servidor.

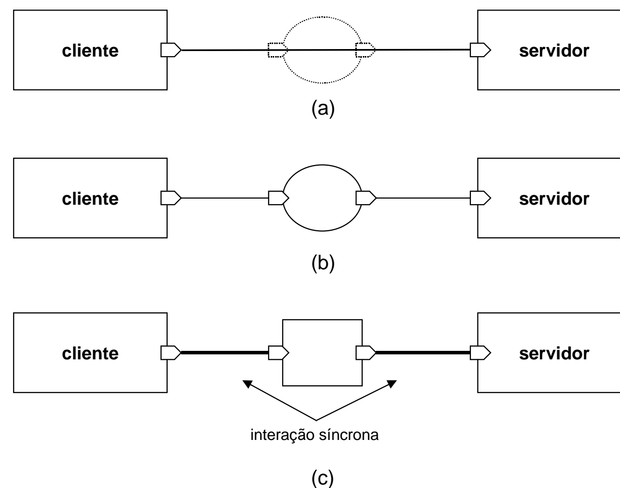


Figura B.3 - Conector como visão de meta-nível de uma interação entre módulos

A figura B.4 apresenta um nível maior de detalhamento sobre o papel do conector na interação dos dois módulos. As duas interações síncronas mencionadas ocorrem de forma aninhada no tempo e devem ser consideradas na representação em rede de Petri. O tempo do processamento da requisição, no servidor, está aninhado no tempo necessário para o conector encaminhar a requisição ao módulo servidor e a resposta a esta requisição para o módulo cliente. O fluxo da requisição, por sua vez está aninhado no tempo total, contado a partir da requisição feita pelo módulo cliente até o recebimento da resposta.

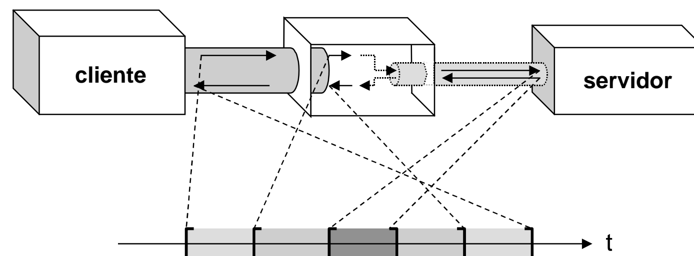


Figura B.4 - Uma interação de módulos no eixo dos tempos

Cada uma das interações (módulo cliente/conector e conector/módulo servidor) é síncrona e pode ser representada pela rede da figura B.5(a), que, em princípio, é uma interação entre uma porta de entrada e uma porta de saída. Adiante será mostrado que, no meta-nível, a transição (fundida) que representa uma interação entre duas portas pode ser dividida em 4 partes e ligadas por uma rede de Petri. Esta operação permite expor os aspectos não-funcionais da interação (figura B.5(b)), ocultos do nível funcional.

Figura B.5 - Visão funcional de uma interação síncrona entre uma porta de saída e uma porta de entrada, (b) a transição única de (a) é dividida em 4 transições na visão de meta-nível

Sob uma visão estritamente funcional, o modelo de um conector é mostrado na figura B.6. A primeira rede (a) apresenta o modelo de uma interação síncrona. O conector é representado apenas pela união das transições representando as portas dos dois módulos. Desta forma, o disparo da transição implica na requisição emitida pela porta de saída do módulo *A* e a requisição sendo recebida e executada pelo módulo *B*, com o respectivo retorno do resultado para o módulo *A*. Para que esta interação seja possível, além de o módulo *A* desejar realizá-la, o módulo B deve estar disposto e pronto (o que seria representado na rede por uma ficha nos lugares superiores).

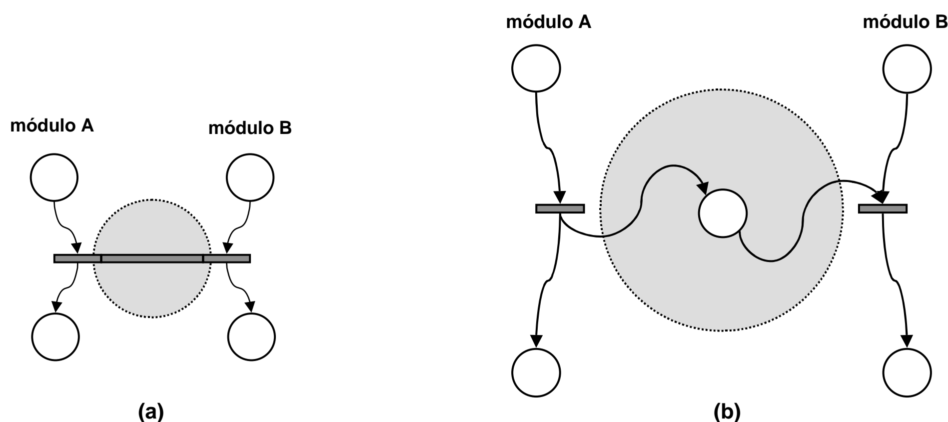


Figura B.6 - Visão funcional da interação de módulos (a) síncrona (b) assíncrona

A rede da figura B.6(b), apresenta o modelo de uma interação assíncrona. Para modelar o assincronismo, um estado intermediário foi introduzido, de forma a tornar o disparo da transição do módulo *A*, representando o envio de uma requisição pela porta de saída, independente do recebimento da requisição pelo módulo *B*. O modelo também indica que o módulo *A* prossegue em sua execução, sem aguardar o módulo *B*. Uma outra informação que pode ser obtida desta representação é a causalidade entre o envio da requisição pelo módulo *A* e o seu recebimento pelo módulo *B*. Além disso, observa-se que o módulo *B* não precisa estar preparado para processar a requisição imediatamente.

B.3.3 Conectores síncronos e assíncronos

A figura B.7 apresenta uma primeira abordagem para o modelo do conector que, além de intermediar a interação de módulos de forma síncrona (a) e assíncrona (b), pode encapsular aspectos não-funcionais, os quais não fazem parte da funcionalidade dos módulos. Observa-se que, nos dois casos, as transições representando as portas foram divididas e interligadas por lugares na rede. Estes representam os estados do módulo cliente aguardando a resposta (LA2) e o módulo servidor processando a requisição (LB2). Uma outra informação importante, que fica exposta, é o desmembramento da representação da porta em duas partes: uma, referente ao envio/recebimento de uma requisição, e outra, vinculada ao recebimento/envio da resposta. No contexto de R-RIO, esta visão ampliada, de meta-nível, é utilizada em favor da especificação de aspectos não-funcionais das aplicações, no nível da arquitetura.

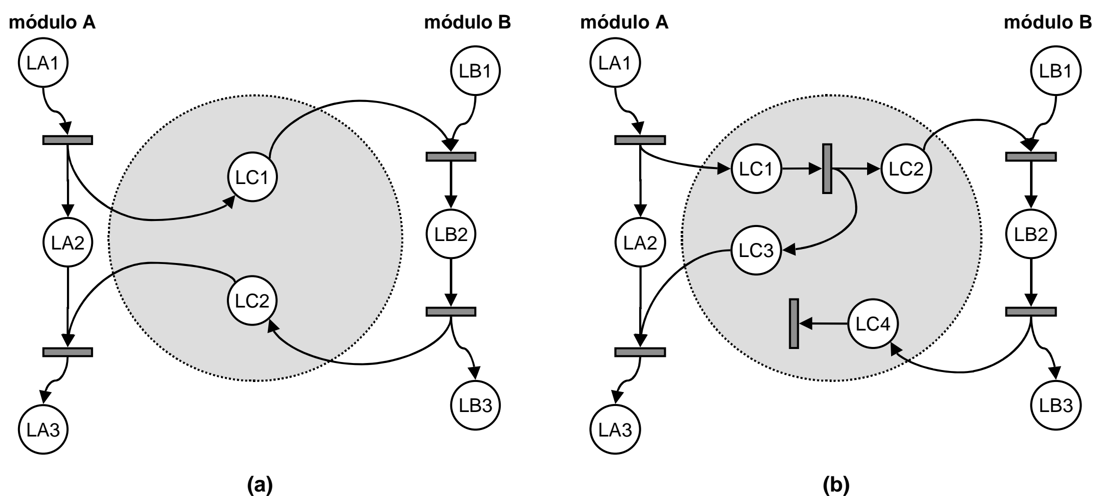


Figura B.7 - Visão de meta-nível da interação de módulos intermediados por conectores. (a) síncrona (b) assíncrona (1a versão)

A representação da figura B.7 é acrescida de informações importantes na figura B.8. Na primeira representação, o conector não tem controle sobre o ritmo com que estará apto a aceitar o trabalho de repassar as requisições. Ele está sempre pronto a receber novas mensagens, indicando uma interação assíncrona com a porta de saída do módulo cliente (na verdade com a parte de envio de requisição da porta) e com a porta de entrada do módulo servidor (na verdade com a parte de retorno da porta de entrada). Na figura B.8 isto é corrigido, adicionando-se dois lugares, LC2 e LC3, para representar o caráter síncrono de cada interação. O conector só aceita o pedido de um módulo cliente para intermediar uma requisição, se estiver preparado (LC2), o mesmo acontecendo com o retorno do módulo servidor (LC3). A concorrência no uso do conector, pode ser representada através de fichas coloridas colocadas nos lugares adequados.

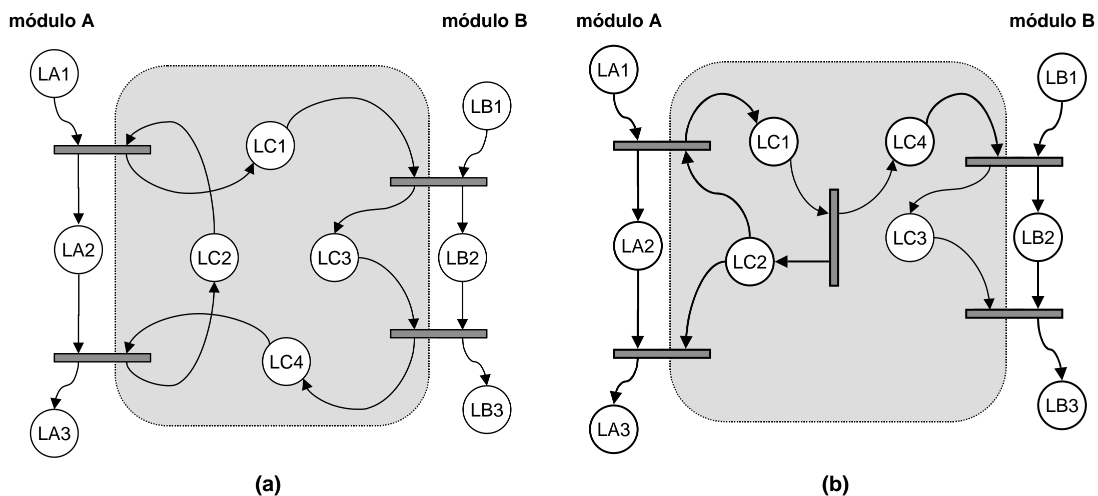


Figura B.8 - Visão de meta-nível da interação de módulos intermediados por conectores. (a) síncrona (b) assíncrona (2a versão)

O conector síncrono B.8(a) apresenta dois estados básicos. Um deles modela o encaminhamento da requisição do módulo *A* para o módulo *B* (LC1). O outro modela o encaminhamento da resposta do módulo *B* para o módulo *A*, após o processamento da requisição (LC4). Este modelo detalha o sincronismo da uma interação na visão funcional, que é atômico, introduzindo, no meta-nível, uma seqüência de estados e transições, dando características assíncronas ao conector. A causalidade continua sendo preservada, mas a evolução do estado do módulo *B* não é mais observada como se ocorresse concomitantemente à evolução do módulo *A*. O conector tem a capacidade de armazenar temporariamente uma requisição, para que seja aguardada a disposição do

módulo *B* em atender a mesma. A atomicidade e o sincronismo no nível funcional continuam sendo observados. O módulo *A* somente entrará no estado posterior ao retorno da requisição, após o processamento desta requisição e o encaminhamento da resposta pelo módulo *B*.

O modelo de um conector assíncrono, figura B.8(b), também apresenta detalhes de meta-nível que não são facilmente visíveis no nível funcional. Por exemplo, os dois estados previstos para o encaminhamento da requisição, LC1 e LC4, permitem que a liberação do módulo *A* (porque a requisição já foi recebida pelo conector para o encaminhamento) e o início do processamento da requisição, pelo módulo *B*, sejam modelados realmente de forma assíncrona. O processamento da requisição pode iniciar no módulo *B* (e terminar), antes que o módulo *A* continue seu processamento, depois de enviar a requisição.

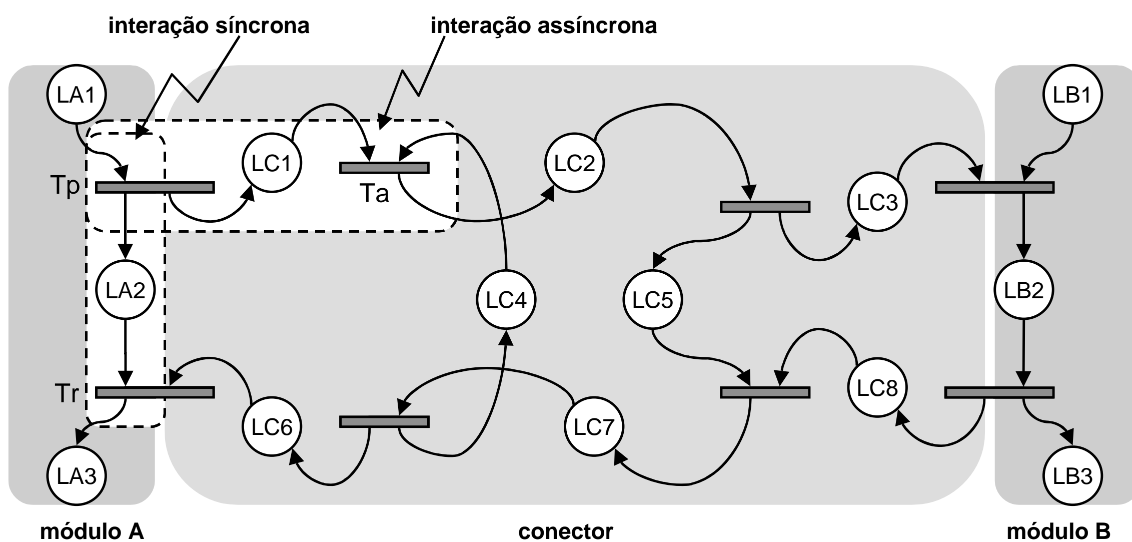


Figura B.9: Representação do meta-meta-nível das portas

É possível refinar o modelo para representar um meta-meta-nível. Detalhando-se um pouco mais as portas de entrada e saída de módulos e conectores, chegamos à solução da figura B.9. Sob o ponto de vista isolado dos módulos e do conector, as interações continuam síncronas. Por exemplo, após o envio de uma requisição (disparo de Tp), o módulo só poderá continuar seu processamento, quando receber uma resposta (disparo da transição Tr). O conector, por sua vez, só aceitará a tarefa de encaminhar uma requisição se uma requisição for feita, e se ele estiver preparado (disparo de Ta). Entretanto, a comunicação entre as partes das portas dos módulos e conector são

reificadas, mais uma vez, e agora são modeladas assincronamente, representadas por mais um lugar (LC1). Este novo lugar estaria representando mecanismos do ambiente de execução, como pilhas, *buffers* ou sistemas de comunicação. Este último modelo permite a representação de informações sobre a aplicação em vários níveis de abstração: funcional, não-funcional e de sistema de execução.

A partir do modelo da figura B.9, observamos, ainda, que a concorrência na utilização dos conectores possui um grau de liberdade a mais, pois o sistema de suporte poderá armazenar requisições enquanto o conector não puder aceitá-las.

Observação

O modelo proposto em Redes de Petri permite uma melhor compreensão sobre componentes de R-RIO, bem como a verificação de propriedades de interação e coordenação em arquiteturas de *software*. Entretanto, não estão representadas informações sobre a interface dos componentes, tais como o sentido, os argumentos e o retorno das portas. Isto impõe algumas limitações nas verificações que podem ser realizadas como, por exemplo, a verificação de consistência ou refinamentos em uma arquitetura. Tais limitações podem ser compensadas utilizando-se, complementarmente, o modelo proposto na seção B.4.

B.4 Modelo para os componentes

Nesta seção, apresentamos uma proposta inicial de adaptação do modelo apresentado em [249], onde a complexidade de um sistema é diminuída organizando-o em módulos mais simples. Isto possibilita que partes do sistema possam ser verificadas assim que são especificadas e, possivelmente, de forma isolada do restante do sistema.

Observa-se que, para este modelo, um conector é considerado como um módulo especializado (ver seção 4.2.3 - capítulo 4) e, assim, este tipo de componente não é representado de forma diferenciada.

B.4.1 Módulo

Como no modelo de componentes de R-RIO, um módulo é dividido em corpo e interface. O corpo define o comportamento do módulo e a interface define a interação possível com o resto do sistema. Neste modelo, a interface é composta de variáveis, que

também pertencem ao corpo, e para todos os efeitos semânticos são a mesma entidade. Só as variáveis do corpo que estão na interface são disponibilizadas para uso no resto do sistema, como meio de comunicação entre módulos.

B.4.1.1 Caso base

O conceito de módulo é definido indutivamente usando-se STJs. Detalhes sobre a semântica podem ser vistos em [249 - seção 3.2].

Como caso base, temos o módulo simples definido por um STJ:

Definição B8 (Módulo - Caso base) Definimos um módulo base, M_b, como um STJ $M_b = (\Pi_M, \Sigma_M, T_M, \Theta_M, \mathcal{J}_M, \mathcal{F}_M)$. Um subconjunto de Π_M, denominado I, é definido como a interface do módulo com o seu meio externo. Denominamos Π_M as variáveis do corpo do módulo.

Ou seja:

- Módulo. É um STJ que representa o comportamento desejado para uma parte do sistema concorrente.
- Corpo. Contém o conjunto das variáveis que definem o estado do módulo, Π_M. Chamamos de variáveis internas, aquelas definidas pelo conjunto $\Pi_M - I$, estas não são disponíveis para operações de composição de módulos.
- Interface. É um subconjunto das variáveis do módulo, $I \subseteq \Pi_M$, que estão disponíveis para operações de composição de módulos.

As condições de justiça são definidas de forma que se $\tau \in T$ realiza uma mudança em uma variável $\upsilon \in I$, então $\tau \in \mathcal{F}_M$, caso contrário $\tau \in \mathcal{J}_M$. Ou seja, o corpo do módulo pode evoluir sem a influência de estímulos externos. Ainda, quando existir alguma interação entre módulos, eventualmente, esta será tratada pelo módulo servidor.

O módulo base é o componente de menor granularidade presente em um sistema. Está completamente especificado no momento de sua criação, e só interage com outras partes do sistema através de sua interface externa.

A interface de um módulo é utilizada para se especificar as variáveis disponibilizadas para o exterior. Em algumas linguagens formais, esta mesma especificação é feita através de um operador chamado de *HIDE*. Em CBabel, utiliza-se a

declaração *export* com esta finalidade.

O uso de uma interface bem definida através de variáveis, apesar de ser uma opção teoricamente idêntica, facilita a análise matemática do relacionamento entre módulos.

B.4.1.2 Interface

É útil tornar o conceito de interface mais preciso, já que o comportamento externo do módulo é o comportamento percebido através da interface. Podemos obter o estado da interface de um módulo, utilizando para isso uma função que mapeia o estado do módulo no estado de sua interface:

Definição B9 (Mapeando estado em interface) Seja Σ_M e I o conjunto de ênuplas formado pelos possíveis valores das variáveis que definem o corpo e a interface do módulo M. Definimos f_I como uma função $f_I : \Sigma_M \rightarrow I$ que mapeia uma ênupla $\pi \in \Sigma_M$ em outra ênupla $\upsilon \in I$, onde $f_I(\pi) = \upsilon = \pi/I$.

Com a função f_I podemos definir a equivalência de estados do módulo quando vistos pela sua interface.

Definição B10 (Equivalência de estados pela interface) Dado um módulo M cuja interface é definida pelo conjunto I, definimos uma relação de equivalência $\equiv_{f_I}$ sobre os elementos de Σ_M relacionando os estados de M com a mesma interface externa. Seja σ_1, $\sigma_2 \in \Sigma_M$, então $\sigma_1 \equiv_{f_I} \sigma_2$ se e somente se $f_I(\sigma_1) = f_I(\sigma_2)$.

Definição B11 (Interface) Dado um módulo $M = (\Pi_M, \Sigma_M, T_M, \Theta_M ; J_M, F_M)$ e um conjunto de variáveis I que define sua interface externa, chega-se a um STJ correspondente ao comportamento da interface do módulo M, definido como $M^I = (\Pi^I_M, \Sigma^I_M, T^I_M, \Theta^I_M ; J_M, F^I_M)$, onde:

- $\Pi^I_M = I$
- $\Sigma^I_M = \Sigma_M / \equiv_{f_I} = \{ [\sigma]_{\equiv f_I} | \sigma \ \Sigma_M \}$
- Para cada $\tau \in T_M$ existe uma transição $\tau' \in T^I_M$ correspondente, de modo que $\tau'([\pi]_{\equiv f_I}) = [\tau(\pi)]_{\equiv f_I}$
- $\Theta^I_M = [\Theta_M]_{\equiv f_I}$

Apesar de podermos expressar o comportamento da interface usando um STJ, a interface não corresponde a um módulo. Como característica própria observamos que todas as suas transições são equânimes $\mathcal{F}^{I}_{M} = T^{I}_{M}$. Isto garante as condições de justiça envolvidas nas comunicações entre módulos, como visto na seção B.2.4.

B.4.1.3 Indução

O módulo base, como vimos, é definido por meio de um STJ que especifica as computações possíveis. Este módulo interage com outros componentes através de uma interface, que também pode ter seu comportamento especificado usando-se um STJ.

Para melhorar a capacidade de lidar com sistemas complexos, torna-se necessário utilizar módulos mais abstratos que o módulo base. Temos que poder definir módulos utilizando módulos já existentes, e especificar a ligação entre os módulos utilizados na especificação destes sistemas.

Primeiro, definiremos como especificar o corpo de um módulo através de uma composição de outros módulos.

Definição B12 (Módulo - Passo de indução) Definimos um módulo de indução, M_i, como um STJ $M_i = (\Pi_{Mi}, \Sigma_{Mi}, T_{Mi}, \Theta_{Mi}, \mathcal{J}_{Mi}, \mathcal{F}_{Mi})$ formado a partir de outros módulos $M_1, \ldots, M_n$ de forma que M_i é formado pela associação paralela das interfaces de seus módulos constituintes.

$$M_i = M^I_1 \parallel \ldots \parallel M^I_n$$

Define-se o conjunto $I_i \subseteq M_i$ como sua interface, e condições de justiça como no módulo base.

A princípio, podemos considerar que os módulos $M_1, \ldots, M_n$, ou melhor, suas interfaces, fazem parte do módulo M_i. Utiliza-se apenas as interfaces destes módulos na definição de M_i, porque os módulos interagem com o seu meio externo apenas através de sua interface, e o módulo M_i é definido externamente aos módulos $M_1, \ldots, M_n$.

Acima, mostramos como definir um módulo utilizando-se outros já existentes. Agora, definiremos como estabelecer a interconexão entre as variáveis dos módulos, o que torna possível a interação entre os mesmos.

Definimos a relação de *Bind*, que trata da composição de módulos para esta

funcionalidade.

Definição B13 (Bind) A operação de *Bind* tem o efeito de tornar duas variáveis em um módulo de indução, $\pi_1, \pi_2 \in \Pi_{M_i}$, iguais semanticamente. As duas variáveis devem ser do mesmo tipo e terão valores idênticos durante toda a evolução do sistema. Denotamos:

$$\pi_1 \equiv_{BIND} \pi_2 \qquad \text{ou} \qquad \text{BIND}(\pi_1, \pi_2)$$

Note que definimos a relação *Bind* apenas sobre módulos de indução, o que implica que apenas variáveis de interface podem ser utilizadas em um *Bind*.

Note que as variáveis disponíveis no corpo do módulo são as variáveis de interface dos módulos que o compõe. E que só definimos a relação de *Bind* sobre variáveis pertencentes a Π_{M_i}, em outras palavras, se os módulos fizerem parte de um mesmo módulo de indução.

Apesar de o módulo de indução M_i ser definido apenas usando-se a interface dos módulos elementares, poderíamos ter definido o módulo de indução usando o corpo dos mesmos, o que é até mais intuitivo, pois o módulo elementar inteiro influirá no comportamento do módulo indutivo. Entretanto, não se perde nada com a restrição de que o corpo do módulo de indução seja formado apenas pelas interfaces de outros módulos. A conseqüência é uma diminuição da complexidade de verificação, já que o espaço de estados para ser analisado pode se tornar significativamente menor.

B.4.2 Portas e Interação

As definições B12 e B13 capturam satisfatoriamente os conceitos de módulo e ligações entre módulos de R-RIO. Entretanto, na definição de interface, B11, um subconjunto, I, das variáveis de um dado módulo M, Π^I_M, representa os pontos de acesso a este módulo. Ao se interligar dois ou mais módulos, através de uma operação de *Bind*, variáveis da interface destes módulos são fundidas, passando a ter o mesmo tipo e o mesmo valor. Desta forma, a interação entre os módulos se dá por memória compartilhada.

O modelo de componentes de R-RIO não permite que variáveis de um módulo sejam acessadas diretamente. Toda interação entre módulos ocorre, exclusivamente,

através de portas, e o acesso a variáveis de um módulo é feito por métodos associados às portas. Assim sendo, a definição de interface precisa ser adaptada para contemplar estes conceitos. Em conseqüência, as definições de módulo, operação de *Bind*, e os procedimentos de verificação sobre sistemas, desenvolvidos a partir destes conceitos, devem ser também revistos.

No restante desta seção propomos um possível caminho para realizar as adaptações necessárias ao modelo, e discutimos alguns problemas antevistos. O desenvolvimento destas adaptações poderá ser feito futuramente.

B.4.2.1 Adaptação de portas e chamadas de métodos

Para contemplar o conceito de portas e chamadas de métodos, no que diz respeito à interação entre módulos, considerarmos uma chamada de método como sendo um conceito dual em relação à escrita em uma variável em memória compartilhada. Assim, propomos uma nova definição para a interface, substituindo o conjunto de variáveis por um conjunto de portas.

Proposição B1 (Porta) A porta é um ponto de acesso associado a um módulo. Através de uma porta de saída, um módulo (cliente) pode fazer requisições a outro módulo (servidor). De forma análoga, um módulo (servidor) recebe requisições feitas por outros módulos (clientes) através de porta de entrada. Uma porta é definida por uma ênupla (τ, V_{arg}, V_{ret}), onde:

τ : É uma transição associada à porta tal que $\tau \in T_{M,}$

V_{arg} : É o conjunto de variáveis de argumento / conjunto de variáveis de parâmetro,

V_{ret} : É a variável de retorno.

O comportamento de uma porta de entrada é assimétrico ao de uma porta de saída. Assim,

- Em uma porta de saída, a ocorrência da transição, τ_O, é conseqüência de uma computação interna do módulo cliente, para enviar uma requisição a uma porta de entrada compatível.

- Em uma porta de entrada, a transição, τ_I, ocorre em reação ao envio de uma requisição. A ocorrência da transição promove uma mudança de estado no módulo servidor e a computação de um resultado.

B.4.2.2 Operação de Link

A operação que interconecta uma porta de entrada a uma porta de saída, de módulos diferentes, denominada *Link*, pode ser definida como uma relação entre as transições, argumentos e retorno, associados a estas portas.

Um dos efeitos desta relação é considerar-se as transições τ_O e τ_I das duas portas envolvidas, como sendo fundidas uma única transição. Isto permite considerar uma operação de interação entre as portas relacionadas como sendo atômica, pois uma única transição estará ocorrendo, sob a hipótese do intercalamento. Tal técnica foi também empregada no modelo com rede de Petri (seção B.3).

Em uma operação de *Link* adicionalmente ocorrem duas operações de *Bind.* A primeira, liga as variáveis do conjunto de argumentos da porta de saída às variáveis do conjunto de parâmetros da porta de entrada:

$$V_{arg_O} \equiv_{BIND} V_{arg_I}$$

No outro sentido, um segundo *Bind* é usado para ligar as variáveis de retorno:

$$V_{ret_O} \equiv_{BIND} V_{ret_I}$$

Diferentemente do que ocorre com as variáveis em uma operação de *Bind*, uma operação de *Link* não funde a transição de todas as portas com mesmo nome. Assim, a relação entre os vários pares de portas que vão interagir deve ser mantida separadamente, salvo se existir uma configuração de grupo. Além disso, o sentido das portas também é contemplado.

De acordo com o que foi dito no final da seção B.4.1.1, o uso de variáveis na interface facilita a análise matemática do relacionamento entre módulos e a própria definição do operador *BIND*. Embora existam vantagens adicionais no uso de portas (seção 4.2.2 – capítulo 4) é possível que, ao se desenvolver a nova definição de interface, a forma final da operação de *Link* torne-se complexa.

B.4.2.3 Interação

Inicialmente, considera-se um método como sendo composto de uma transição τ. Esta transição é a mesma selecionada como ponto de entrada para o método e é, igualmente, a associada à porta de entrada. Um método também possui um conjunto de

variáveis locais, e compartilha o conjunto de variáveis V_{arg} com a porta, mas não pode escrever nos mesmos.

Uma interação entre módulos ocorre atomicamente. As sub-operações de envio da requisição e o envio do retorno, do modelo de componentes de R-RIO, estão contempladas na nas operações de *Bind* definidas na operação de *Link* (seção 4.2.2). A execução do método se dá pela ocorrência da transição τ. Embora todas as referidas sub-operações ocorram atomicamente, existe uma relação de causalidade entre o conjunto de variáveis V_{arg}, enviados na requisição, e a variável V_{ret} enviada como retorno.

Definiu-se, anteriormente, uma transição como uma função parcial, que muda o estado de um sistema e que ocorre de forma atômica. Assim, para tornar consistente a definição de uma interação, é necessário considerar que, ao se ligar uma porta de saída a uma porta de entrada, a transição associada a cada uma destas passa a ser considerada como uma única transição. Este requisito é ainda garantido pelo uso da hipótese do intercalamento.

B.4.3 Concorrência

A adaptação que estamos propondo restringe o modelo de concorrência de R-RIO ao considerar que uma interação ocorre atomicamente. Por um lado, tenta-se manter a simplicidade do modelo formal. Mas, por outro lado, isto limita tanto o grau de concorrência no acesso a uma mesma porta, quanto a própria computação associada a um método.

Em um modelo mais completo (e mais complexo), poderíamos ainda considerar a relação entre o conjunto de variáveis V_{arg}, e a variável V_{ret}, como sendo de causalidade, mas não mais a atomicidade da transição τ. Em conseqüência poderiam surgir as seguintes características:

- uma mesma porta poderia receber uma ou mais requisições antes de completar uma interação que está sendo processada;
- a execução do método associado a uma porta de entrada poderia conter várias transições de estado e, ainda, ser recursiva;
- operações primitivas do tipo *wait() e signal()/notify ()*, que bloqueassem e

desbloqueassem a evolução da computação de um método, poderiam ser permitidas.

O desenvolvimento de um modelo que incluísse tais características não é trivial. Primeiramente, o estado global do sistema seria variável, podendo crescer ilimitadamente. Seria também necessário incluir o conceito de *thread* para modelar o atendimento concorrente de chamadas.

Uma solução possível seria disponibilizar uma cópia das variáveis de parâmetro V_{arg}, da variável de retorno V_{ret} e das variáveis locais do método, associado à porta de entrada, para cada operação de *Link*. Assim, cada computação concorrente do mesmo método ocorreria sobre o mesmo conjunto de transições, mas sobre variáveis diferentes.

Embora estes problemas possam parecer administráveis, em geral, mesmo em alguns dos modelos mais completos, encontrados na literatura atual, estes problemas não são inteiramente tratados [255].

B.5 Verificações

Nesta seção examinamos a possibilidade de se realizarem algumas verificações em arquiteturas de *software*, com base nos modelos apresentados.

B.5.1 Verificação de Modelo (*Model Checking*)

De acordo como observado anteriormente, é possível converter uma arquitetura descrita em CBabel em uma representação do tipo LSTS, cujo modelo de base é um STJ. Este mapeamento é facilitado pela descrição explícita de aspectos e contratos de interação e coordenação nos conectores. Assim, além de permitir a verificação sintática, descrições em CBabel podem ser usadas como ponto de partida para examinar-se a coerência de arquiteturas de *software*, bem como para a realização de verificações de propriedades das mesmas.

Utilizamos a aplicação dos produtores-consumidores com *buffer* limitado, para exemplificar a possibilidade de verificação do modelo de uma arquitetura de *software*. Inicialmente, a descrição da arquitetura é convertida para uma rede condição-ação e, depois, os passos para a verificação do modelo são indicados.

B.5.1.1 Tradução das principais propriedades para lógica temporal

A arquitetura descrita em CBabel, no capítulo 4, será adaptada, de acordo com a formulação natural do problema, apresentada no capítulo 3, para introduzir algumas informações de estado no *buffer* (como a existência de um produtor produzindo ou um consumidor que esteja esperando para consumir). Esta adaptação visa facilitar o procedimento de verificação, sem alterar a arquitetura. O foco de interesse é o contrato de coordenação no conector. A computação dos módulos não é importante neste contexto. A listagem B.1 apresenta a descrição do conector, contendo um contrato de coordenação, e as informações adicionais que serão usadas na verificação do modelo.

```
1  connector {
2      condition vazio = true, cheio = false;
3      int n_itens; // o estado será mantido no conector
4      int MAX_ITENS = 10;
5      int produzindo = 0;
6      int consumindo = 0;
7      exclusive {
8        out port PutT {
9          guard  (vazio) { // SusProd
10           before {
11              produzindo = produzindo + 1;
12           }
13           after {
14             cheio = true;
15             produzindo = produzindo - 1;
16             n_itens = n_itens + 1;
17             if (n_itens == MAX_ITENS) vazio = false;
18         } }
19       } GPut;
20       out port GetT {
21         guard  (cheio) { // SusCons
22           before {
23              consumindo = consumindo + 1;
24           }
25           after {
26             vazio = true;
27             consumindo = consumindo - 1;
28             n_itens = n_itens - 1;
29             if (n_itens == 0) cheio = false;
30         } }
31       } GGet;
32     }
33     in port GetT Get;
34     in port PutT Put;
35   } SincMutexPCcon;
```

Listagem B.1 - Conector com informações adicionais

A descrição adaptada do contrato de coordenação, associado ao conector *SincMutexPCcon*, pode ser transformada em fórmulas de lógica temporal. Este é o primeiro passo para a verificação do modelo: identificar as propriedades que se deseja examinar e descrevê-las em lógica temporal.

A primeira propriedade está relacionada à exclusão mútua. Produtores e consumidores acessam o *buffer* em exclusão mútua.

F.1) (Produzindo > 0) ⊃ ((Produzindo = 1) ∧ (Consumindo = 0))

Se existe algo produzido no *buffer* (Cheio = TRUE) e espaço vazio (Vazio = TRUE), então um produtor, ou um consumidor, poderá eventualmente produzir, ou consumir.

F.2) ((Vazio = TRUE) ∧ (Cheio = TRUE) ⊃ (eventualmente ((Produzindo > 0) ∨ (Consumindo > 0)))

Se o *buffer* está completamente cheio (Vazio = FALSE) então um produtor ficará suspenso no guarda (SusProd > 0) e não poderá produzir (Produzindo = 0), até que um consumidor consuma um item (Consumindo > 0).

F.3) ((Vazio = FALSE) ∧ (SusProd > 0)) ⊃ (¬ ((Produzindo > 0) = TRUE) until (Consumindo > 0))

... ou melhor

F.3) ((Vazio = FALSE) ∧ (SusProd > 0)) ⊃ (¬ ((Produzindo > 0) = TRUE) until (Vazio = TRUE))

Se o *buffer* está completamente vazio (Cheio = FALSE), então um consumidor desejando consumir ficará suspenso no guarda (SusCons > 0) e não poderá consumir (Consumindo = 0) até que um produtor produza um item (Produzindo > 0).

F.4) ((Cheio = FALSE) ∧ (SusCons > 0)) ⊃ (¬ ((Consumindo > 0) = TRUE) until (Cheio = TRUE))

Deseja-se verificar se as fórmulas são válidas para todo espaço de computação da aplicação.

B.5.1.2 Verificação da validade das fórmulas

Uma análise informal das fórmulas não é exaustiva, pois não garante que o conjunto de computações seja completamente verificado. O aumento da complexidade das fórmulas diminui ainda mais esta garantia. A técnica da verificação do modelo permite, que uma análise seja feita de forma sistemática em todo o espaço de computação. Uma das maneiras de se empregar esta técnica segue os seguintes passos:

1. conversão da aplicação para uma LSTS do tipo condição-ação;
2. geração da árvore de alcançabilidade ou tabela de marcação;
3. seleção de uma fórmula-alvo para ser verificada e
4. varredura sistemática do espaço de computação, verificando-se a validade da fórmula.

B.5.1.3 Rede condição-ação

Na conversão da arquitetura da aplicação em uma rede condição-ação, cada transição pode ser rotulada por uma condição de disparo e uma ação resultante. A rede condição-ação do exemplo é apresentada na figura B.10. Por simplicidade, foram omitidos alguns estados, tais como a inicialização de recursos. Nos modelos do produtor, consumidor e do *buffer* não são consideradas as etapas de produção, consumo e depósito, por não serem relevante nas fórmulas a verificar. Seria simples incluir mais um lugar e uma transição para representar a produção de um item e o consumo do mesmo, mas resultaria apenas em aumento da árvore de marcações.

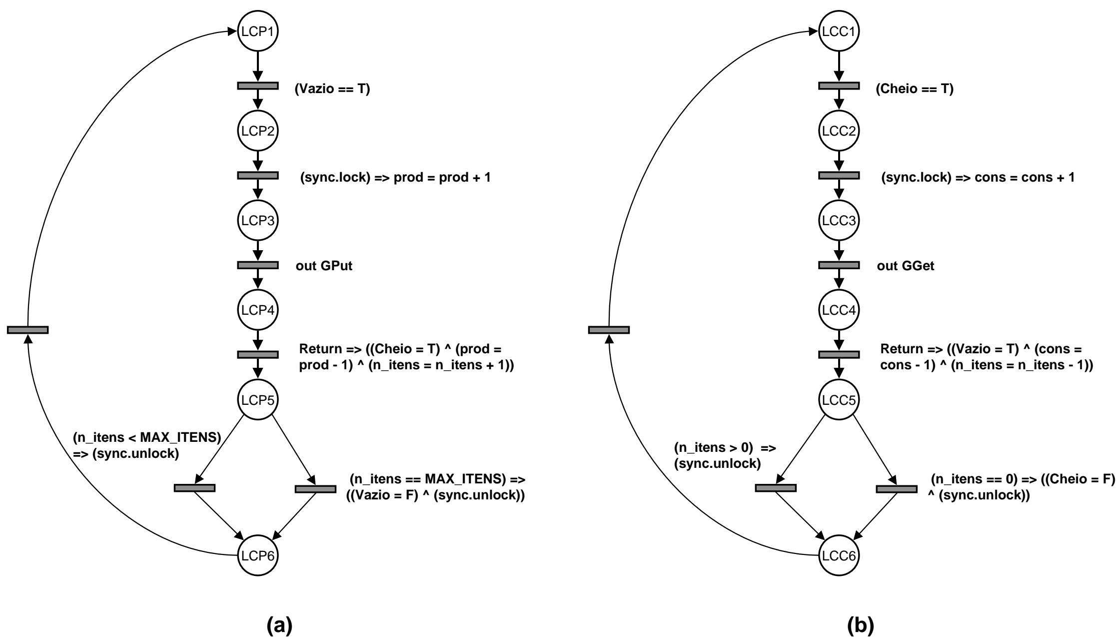


Figura B.10 - Rede condição-ação do conector (a) porta de saída GPut e (b) porta de saída GGet

Vale a observação de que os pares condição-ação do tipo *sync.lock => lock = TRUE* indicam que uma operação de *lock* no monitor foi bem sucedida, ou seja, o monitor está sendo acessado em exclusão mútua. Esta é a condição para o disparo da transição. Tal artifício é necessário para que a semântica de monitores, utilizada no contrato de coordenação, seja modelada corretamente.

B.5.1.4 Verificação da validade

A partir da rede condição-ação, a realização dos próximos passos é mecânica. Entretanto, a obtenção da árvore de alcançabilidade ou tabela de marcações, e a

varredura do espaço de computações da aplicação é bastante trabalhosa e suscetível de erros, se executadas manualmente. Ferramentas de *software* podem automatizar este trabalho. Mesmo assim, cenários simples, como um produtor e um consumidor no exemplo dado, seriam utilizados para evitar-se a explosão de estados. Com uma ferramenta, a tabela de marcações é percorrida e, em cada célula, verifica-se (simbolicamente) se o estado correspondente é válido para uma dada fórmula selecionada. Se para todos os estados percorridos a fórmula for válida, então ela é válida no modelo.

Em [248] descreve-se uma seqüência de passos para a conversão de programas descritos em Ada, em uma máquina de estados representada na linguagem Verus. Esta nova descrição é introduzida em uma ferramenta, juntamente com uma série de propriedades, descritas em lógica temporal, a serem verificadas no programa. Partindo de descrições de arquiteturas de *software* em CBabel, seria possível seguir uma seqüência de passos semelhantes, automatizando este procedimento para as arquiteturas de *software*.

B.5.2 Verificação por partes

O modelo algébrico original, utilizado na proposta de seção B.4, permite a verificação de um sistema de forma hierárquica. Cada módulo de um sistema é verificado a partir da verificação de seus módulos componentes. Este procedimento é repetido sistematicamente até que todos os módulos do sistema sejam verificados. Em [249] prova-se que, para o modelo de módulo proposto, esta metodologia é correta. Desta forma, simplificam-se os procedimentos de verificação, diminuindo a complexidade do modelo sob análise, e torna-se factível a automatização dos mesmos.

Em princípio, a verificação de uma arquitetura de *software*, por partes, também pode ser realizada. A dualidade entre memória compartilhada e chamada a método sugere que o problema é apenas de uma redefinição cuidadosa.

O desenvolvimento completo do modelo de módulo, adaptado para componentes e configurações de R-RIO, permitiria avaliar se a mesma técnica de verificação por partes seria aplicável a uma *arquitetura* de software. Sob o ponto de vista funcional, parece não haver impedimento. Entretanto, como o uso de variáveis, no modelo original, também tornam mais simples os procedimentos de verificação, é possível que a

adaptação dos procedimentos de verificação para o novo modelo resulte em procedimentos muito complexos, ou que algumas verificações, realizadas no modelo original, não sejam, aí, factíveis.

Adicionalmente, a verificação de aspectos de interação, coordenação e QoS, mesmo traduzidos para lógica temporal e associados às descrições dos módulos, não é simples. Como já levantado anteriormente, o efeito de composições onde estes aspectos estão presentes pode dificultar os procedimentos de verificação. Desta forma, mais pesquisas se fazem necessárias.

B.6 Conclusão

A definição de uma arquitetura de *software* através de uma representação formal, ajuda a tornar mais claros os modelos de componentes e configuração de R-RIO. Os modelos propostos neste apêndice, em conjunto, capturam os conceitos de módulos, conectores e portas, bem como aspectos de interação e coordenação.

Descrições em CBabel, contém as informações necessárias para a obtenção sistematizada da representação correspondente em redes de Petri. Isso permite aplicar os conceitos do modelo de base apresentados na seção B.2. Além disso, este modelo permite a conversão de descrições em CBabel para outros formalismos relacionados com LSTSs [72, 224, 250, 251, 252]. Assim, pode ser possível o uso dos procedimentos de verificação propostos para os mesmos, com o objetivo de verificar arquiteturas de *software*.

A proposta de modelo apresentada na seção B.4 complementa o modelo de base, capturando detalhes sobre a interface dos componentes e a topologia de interconexão dos mesmos. Uma continuação desta proposta poderá adaptar a capacidade de realizar verificações por partes, apresentada no modelo original, em arquiteturas de *software* com um número grande de componentes. É possível, também que em uma continuação deste trabalho, seja apontado o caminho para a realização de refinamentos e otimizações nas arquiteturas descritas a partir do modelo proposto.