Configurando Protocolos de Interação na Abordagem R-RIO[1]

Alexandre Sztajnberg
IME/UERJ e GTA/COPPE/UFRJ
alexszt@ime.uerj.br

Marcelo Lobosco Orlando Loques
CAA/IC/UFF
{lobosco, loques}@caa.uff.br

Resumo

Aplicações modernas possuem natureza dinâmica e necessitam de suporte para se adaptarem a novas demandas funcionais e não-funcionais. Novas demandas funcionais são criadas por usuários das aplicações, que precisam funções não previstas originalmente ou necessitam alterar funções existentes. Durante o ciclo de vida das aplicações, podem também surgir requisitos não-funcionais que, em geral, não estão associados diretamente com a aplicação. Modelos mais usuais para o desenvolvimento de software não oferecem as facilidades necessárias para a produção de aplicações que se adaptem dinamicamente a essas demandas. Portanto, uma abordagem que facilite a concepção dessas aplicações se faz necessária. Apresentamos neste artigo a abordagem R-RIO, que estende o conceito tradicional de configuração, permitindo que se descreva a arquitetura das aplicações a partir de componentes e suas interações e, adicionalmente, seus aspectos não-funcionais como a distribuição, coordenação e protocolos de comunicação. A arquitetura de uma aplicação é descrita com o uso de CBabel, a linguagem de configuração de R-RIO. A partir da descrição da arquitetura um middleware, chamado configurador, pode criar instâncias de seus componentes e iniciar a execução da aplicação. O configurador também oferece suporte para adaptação das aplicações que necessitam reagir à mudanças de requisitos ou de condições de operação, inclusive durante a fase de funcionamento. O exemplo do Jogo da Velha (TicTacToe) é utilizado para ilustrar a concepção de uma aplicação com um protocolo de interação complexo utilizando-se R-RIO e para permitir uma comparação com outras abordagens.

Abstract

Modern applications have dynamic nature and need support to be adapted in face of new functional and non-functional demands. New functional demands are created by applications' users that need functions that were not originally anticipated or need to change existing ones. Non-functional requirements that, in general, are not directly associated with the application functions can also appear during the applications' life-cycle. Usual software development models do not provide the desired facilities to produce applications that can be dynamically adapted. Therefore, an approach to facilitate the development of these applications is necessary. In this paper we present the R-RIO approach, that extends the traditional configuration concept, allowing application architectures to be described by components and their interactions and, additionally, by their functional and non-functional aspects, such as distribution, coordination and communication protocols. The architecture of an application is described using CBabel, R-RIO's configuration language. A middleware, called configurator, makes it possible to create and execute program instances, described through the configuration language, which can be adapted to new requirements even during their operation. The TicTacToe game is used to illustrate the design of an application with a complex interaction protocol using R-RIO and to allow comparisons with other approaches.

1. Introdução

Aplicações distribuídas modernas possuem natureza dinâmica e necessitam de suporte para se adaptarem a novas demandas funcionais e não-funcionais [21]. Novas demandas funcionais são criadas por usuários das aplicações, que precisam de funções não previstas originalmente ou necessitam alterar funções existentes. Além disso, podem surgir novos requisitos não-funcionais que, de uma forma geral, não estão associados diretamente com a

[1] R-RIO. Reflective-Reconfigurable Interconnectable Objects.

aplicação, como distribuição física de componentes, protocolos de comunicação, qualidade de serviço, disponibilidade ou tolerância a falhas, por exemplo. Modelos tradicionais para a concepção de *software* não oferecem as facilidades necessárias para a produção de aplicações que se adaptem dinamicamente a novos requisitos. Na maioria dos casos as aplicações são fruto de um grande esforço de programação, apresentando um código complicado, com vários aspectos entrelaçados (*code tangling* [23]) e composição estática. É comum, por exemplo, um mesmo bloco de código conter instruções para a comunicação via *sockets*, sincronização via semáforos, além das instruções que executam a computação intrínseca da aplicação, criando assim uma dificuldade para a reutilização de código. Para facilitar a concepção destas aplicações, uma nova abordagem se faz necessária. Esta abordagem deve considerar que todas aplicações precisam evoluir dinamicamente em seus aspectos funcionais e não-funcionais ou mesmo em sua estrutura.

Apresentamos neste artigo a abordagem R-RIO, que estende o conceito de configuração, permitindo que se descreva a estrutura de aplicações a partir de componentes e suas interligações e, adicionalmente, seus aspectos não-funcionais como distribuição, coordenação ou protocolos de comunicação, citando apenas alguns. Em R-RIO a configuração de aspectos diferentes pode ser feita de forma separada, utilizando-se um elemento de conexão de módulos, com características reflexivas. A arquitetura de uma aplicação é descrita com o uso de uma linguagem de configuração. A partir desta descrição, um *middleware* chamado configurador possibilita a criação de instâncias desta aplicação. O configurador também oferece suporte para adaptação das aplicações que necessitam reagir à mudanças de requisitos ou nas condições de operação, inclusive durante sua execução.

Ao longo deste artigo será utilizado o exemplo do Jogo da Velha (*TicTacToe)* para ilustrar vários pontos relevantes da proposta. Como ponto de partida será utilizada a implementação do jogo oferecida no kit de desenvolvimento da linguagem Java disponibilizada pela Sun Microsystems [15]. Esta implementação apresenta vários dos problemas de entrelaçamento de código já apontados. 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 atualização na apresentação do jogo ou a introdução de um outro jogador é tarefa complicada neste caso.

A primeira parte deste trabalho discute os conceitos básicos de R-RIO e sua abordagem para a concepção de aplicações. Na segunda parte, o exemplo do jogo da velha é revisitado sob o ponto de vista de uma proposta correlata e uma solução é implementada a partir dos conceitos de R-RIO. Observa-se que esta implementação, além de apresentar qualidades como a separação de interesses, pode evoluir dinamicamente. Nas ultimas seções, os conceitos são consolidados e algumas conclusões apresentadas.

2. A abordagem R-RIO

R-RIO procura atender a um conjunto de requisitos que pretende oferecer uma base para a engenharia de sistemas de uma forma geral:

Reusabilidade. Técnicas utilizadas para a concepção de aplicações devem permitir que partes de aplicações previamente concebidas possam ser reutilizadas de forma a minimizar esforços de desenvolvimento. Em um caso extremo, uma aplicação pode ser criada completamente a partir de partes já existentes [31]. Dois conceitos estão associados a reusabilidade: modularidade e interoperabilidade. O projeto de componentes modulares está

vinculado a dois aspectos: encapsulamento, em que o comportamento do módulo é auto-contido e padronização de interface, em que a as regras de acesso à funcionalidade do módulo são claramente definidas.

Separação de Interesses. A separação de interesses (*separation of concerns*) é um paradigma que vem sendo considerado para a concepção de grandes sistemas [1, 3, 6, 19]. Neste paradigma o projeto de uma aplicação é dividido em especificações de requisitos funcionais e não-funcionais. Idealmente, aspectos relacionados com as computações básicas de uma aplicação podem ser tratados independentemente de aspectos não-funcionais, como coordenação, comunicação e distribuição (da mesma forma, aspectos não-funcionais como protocolos de comunicação, QoS e gerência de recursos do ambiente de execução, também chamados operacionais, podem ser tratados separadamente). A obtenção da separação de interesses exige, entretanto, uma disciplina do projetista. A arquitetura da aplicação deve ser planejada com esta 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, evolução dinâmica é um requisito de suporte para a modificação e inclusão de novos componentes em aplicações que não podem ser completamente paradas para atividades de reconfiguração [34]. Por exemplo, a atualização de *software* (*up-grade*) de componentes, inclusão de tolerância a falhas, introdução de novos parâmetros de QoS, alteração de estilos de comunicação ou restrições temporais são atividades que podem implicar em mudanças em tempo de execução. Estes ajustes podem ser tratados, em muitos de seus aspectos, de forma transparente para a aplicação.

Abrangência. O requisito de abrangência de uma tecnologia apresenta várias facetas. Primeiramente deseja-se que com uma dada tecnologia um grande número de problemas possa ser solucionado e que para isso ela tenha expressividade adequada. Adicionalmente são observadas a disponibilidade, disseminação e aceitação das tecnologias. Ferramentas facilmente disponíveis e disseminadas facilitam a criação de massa crítica de *experts* e em conseqüência são mais aceitas e tendem à evoluir.

A proposta de R-RIO é oferecer uma nova alternativa para a construção de aplicações baseada em configuração. Esta proposta difere de outras existentes, como [26, 32, 4], por integrar explicitamente o paradigma de separação de interesses. Em R-RIO o conceito de configuração está baseado em: **(1)** uma metodologia para a concepção de aplicações, por composição, a partir de módulos componentes, atendendo a aspectos funcionais básicos das mesmas; **(2)** uma linguagem de configuração, que permite descrever a interconexão de módulos para formar a estrutura das aplicações ou módulos de maior complexidade e descrever características ou aspectos não-funcionais das aplicações, como por exemplo, a sincronização, distribuição, tolerância a falhas ou parâmetros de qualidade de serviço; e **(3)** um sistema de suporte, chamado de configurador, que permite a criação, conexão e remoção de componentes das aplicações.

A metodologia de R-RIO incentiva a modularidade na concepção dos elementos básicos e a separação explicita entre a criação de componentes e o estabelecimento das interações entre estes componentes. Em princípio, isto permite retirar dos módulos a responsabilidade de implementar e gerenciar as interações, e concentrar esta responsabilidade em elementos de conexão, chamados de conectores em R-RIO. Esta separação básica de aspectos aumenta a possibilidade de reuso dos módulos funcionais e facilita a adequação dos conectores para diferentes estilos de interação.

2.1. Módulos, Portas e Conectores

O modelo de R-RIO está baseado em três elementos básicos:

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

O primeiro passo para a construção de uma aplicação é a descrição dos módulos que irão compor a mesma e a seleção dos conectores que interligarão estes módulos. Em seguida, a configuração do relacionamento destes elementos é determinado através das ligações entre módulos e conectores. Adicionalmente, podem ser descritos requisitos não-funcionais da aplicação, permitindo que esta composição de módulos e conectores seja ajustada em diferentes aspectos. O exemplo de uma configuração possível para o jogo da Velha, composta por três módulos é apresentado na Figura 1.

Um módulo é um elemento que potencialmente encapsula parte da funcionalidade de uma aplicação. Elementos funcionais claramente distintos da aplicação dão origem a classes de módulos distintas [32], como por exemplo clientes, servidores, geradores e tratadores de eventos. Na Figura 1 podemos identificar 3 instâncias de módulos, criadas a partir de uma descrição das classes destes módulos. Classes de módulos possuem um mapeamento para uma implementação. Por exemplo, os módulos da Figura 1 podem ser implementados como objetos em Java , C++ ou CORBA [30].

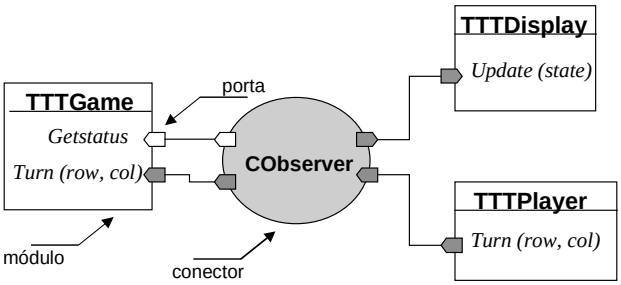

Figura 1 - Aplicação TicTacToe

Portas são elementos auxiliares do modelo de configuração que representam pontos de interação de um módulo. As portas podem ser de entrada, através das quais os módulos oferecem serviços, ou de saída, por onde os módulos requisitam serviços de outros módulos. Na linguagem de configuração a porta é utilizada como referência na ligação de dois componentes. Por exemplo, ao ligarmos uma porta de saída de um módulo cliente a porta de entrada de um módulo servidor, estamos acoplando estas referências, permitindo a interação entre estes componentes. As portas também são elos importantes entre o nível de configuração e a implementação. No nível da implementação uma porta pode ser identificada como a assinatura de um método ou procedimento, no caso de portas de entrada, ou uma invocação de método ou chamada de procedimento, no caso de portas de saída. Desta forma o configurador pode traduzir as referências dos pontos de interação entre os dois níveis. Na Figura 1, o módulo *Player* está preparado para fazer requisições através da porta *Turn*. No

nível da implementação as requisições podem ser mapeadas transparentemente em invocações de métodos. Da mesma forma, o módulo *Game* oferece serviços através de suas portas de entrada *Turn* e *Getstatus*, que podem ser mapeados como métodos oferecidos.

O conector é um elemento que oferece a abstração necessária para se determinar, no nível da configuração, a interação de módulos. Conectores são responsáveis, primariamente, por encaminhar as requisições que devem ir de uma porta de saída de um módulo para uma porta de entrada de outro módulo e o retorno dos resultados no sentido contrário, quando necessário. Este encaminhamento é executado segundo um padrão ou protocolo para o qual o conector foi concebido. Conectores possuem um conjunto de características que os tornam especialmente atraentes para tratar os aspectos relacionados com a interação e a comunicação entre componentes [7]. Dentre estas características destacam-se:

- o conhecimento da interface e as referências dos módulos por eles interligados;
- a capacidade de examinar e manipular o conteúdo de requisições e respostas antes de encaminhá-las aos seus destinos finais;
- a possibilidade de controlar o repasse das requisições e respostas para os seus destinos.

O conector pode operar, em algumas configurações, simplesmente repassando requisições e respostas entre os módulos (em casos simples, o conector pode usar diretamente um mecanismo do ambiente, e.g. RMI em Java). Em outras situações, além do encaminhamento de requisições, o conector pode atuar manipulando os valores dos argumentos das requisições que passam através dele, estabelecer os protocolos de interação e a forma com a qual o encaminhamento das requisições é feito, como se fossem contratos [12], ou ainda adicionar novas características à interação entre módulos. O fato de um conector poder controlar as interações entre módulos através da interceptação e manipulação de requisições e respostas torna evidente a reflexividade deste elemento, embora o esquema reflexivo usado seja diferente ao que se tem feito através de linguagens especializadas ou ambientes de suporte a reflexão dinâmica [11 e 29]. Assim sendo, outros aspectos, além da comunicação, também podem ser configurados a partir dos conectores. Por exemplo, requisitos como a sincronização podem ser encapsulados em conectores, tornando a programação interna dos módulos independente deste aspecto [25].

As interconexões de módulos são indicadas explicitamente durante a configuração através de comandos de *ligação*. Através destes comandos as referências a portas de entrada e saída passam a ser conhecidas e podem ser acopladas pelo conector (referências associadas a interações não configuráveis não precisam ser declaradas; por exemplo, em um ambiente Java elas são resolvidas e realizadas automaticamente). Em um comando de ligação são indicados os módulos a serem interconectados e o tipo de conector a ser utilizado. Na aplicação da Figura 1, por exemplo, após a criação das instâncias dos módulos o conector *CObserver* foi utilizado para interligar as portas de entrada e saída compatíveis (porta *Turn* do módulo *Player* à porta *Turn* do módulo *Game*).

O mapeamento de conectores para o nível da implementação não é rígido. Um conector pode assumir a forma de uma consulta a tabela para ligar referências entre a chamada e o corpo de um procedimento contidos em módulos diferentes. Um conector também pode ser implementado como um objeto que oferece e invoca métodos, que será interposto entre os módulos funcionais da aplicação. Adicionalmente, ao mapear um conector em uma implementação, o projetista ainda pode utilizar mecanismos auxiliares como primitivas de sincronização, bibliotecas, protocolos de comunicação, entre outros, oferecidos pelo sistema de suporte nativo, para programar o controle e o funcionamento deste conector.

2.2. Separação de Interesses e Programação de Aspectos

No contexto de R-RIO, o requisito separação de interesses é atendido das seguintes formas:

- selecionando-se, inicialmente, módulos para compor a estrutura funcional da aplicação e conectores para intermediar a interação destes módulos;
- concebendo-se, posteriormente, a arquitetura da aplicação pela interconexão dos módulos e conectores para atender os requisitos de interação demandados pela aplicação;
- configurando-se, de maneira desacoplada, aspectos não-funcionais através de ajustes nos conectores ou por composição de módulos e conectores adaptados, a partir de módulos e conectores mais simples.

Com estas possibilidades, o projetista poderá usar uma disciplina na concepção das aplicações para que os módulos sejam responsáveis pelos aspectos funcionais e os conectores pelos aspectos não-funcionais. Esta disciplina, embora não obrigatória, potencializa a reusabilidade dos módulos e conectores.

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

2.2.1. Distribuição. A abordagem proposta permite que os módulos de uma aplicação possam ser instanciados em qualquer nó de um sistema distribuído. Na implementação dos módulos, não é necessária nenhuma referência quanto a sua localização ou qualquer outro suporte para a interação de módulos distribuídos. A localização de uma instância de um módulo é transparente em relação à sua funcionalidade. Na linguagem de configuração, a instanciação de um módulo pode vir acompanhada de um parâmetro indicando onde esta instância deve ser criada. Este é o único momento em que se associa um módulo à sua localização.

2.2.2. Comunicação. Em uma aplicação distribuída a interação entre os módulos é feita através de um sistema de comunicação, provavelmente usando diferentes estilos de interação. Tanto na concepção quanto durante a operação de uma aplicação distribuída, ajustes na comunicação entre os módulos podem ser necessários, como por exemplo a seleção do mecanismo a ser utilizado, o protocolo de comunicação mais adequado ou a determinação de parâmetros de qualidade de serviço (QoS). Ajustes durante a operação podem ser feitos sob controle da própria aplicação pela reconfiguração de conectores. Uma opção é concentrar-se as tomadas de decisão sobre as reconfigurações em módulos especiais, utilizando-se inclusive linguagens especializadas, que interagem com o configurador para adaptar a aplicação [2].

Como a comunicação entre os módulos de uma aplicação é intermediada por conectores, detalhes operacionais de comunicação podem ser ajustados através destes. A linguagem de configuração permite que anotações sobre a comunicação sejam declaradas nos conectores de maneira simples. O projetista pode configurar os aspectos de comunicação sem a preocupação imediata de como serão implementados. Se uma implementação de conector com as características desejadas estiver disponível, é suficiente indicá-la na configuração e esta será automaticamente adaptada para a aplicação (ver seção 4). Caso contrário, o novo conector

requerido deverá ser implementado ou obtido por composição (ver seção 2.2.4). Em qualquer caso, durante a implementação de um conector, o programador pode fazer uso de facilidades disponíveis no sistema nativo. Por exemplo, o programador poderá se valer de mecanismos de comunicação como *sockets* ou RPC. Muitas das interfaces de programação destes mecanismos também permitem que se determinem parâmetros de QoS, complementando as possibilidades de ajuste nos aspectos de comunicação que se pode fazer através de conectores.

2.2.3. Coordenação. Em nossa proposta os aspectos de coordenação de uma aplicação são especificados em alto nível de abstração, de forma separada dos módulos sobre os quais irão atuar. Instâncias diferentes de um mesmo tipo de módulo podem estar submetidas a esquemas de coordenação distintos. O modelo para o tratamento do aspecto de coordenação prevê três tipos de situação:

- concorrência entre módulos de uma aplicação,
- concorrência e exclusão mútua entre métodos de um mesmo módulo, e
- a sincronização entre métodos de um mesmo módulo.

Em R-RIO todos os módulos de uma aplicação são executados potencialmente de forma concorrente. Cada módulo pode ser considerado uma unidade autônoma que pode concorrer pelo processamento com outros módulos ou pode executar paralelamente se estiver em um ambiente paralelo ou distribuído. Este modelo é adotado em propostas semelhantes [1 e 23].

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 guardas lógicos. Um guarda pode controlar a admissão de requisições à porta de entrada de um método, retardando o tratamento destas requisições até que um conjunto de condições, associadas a este guarda, seja satisfeito. Em R-RIO, guardas são conectores que estarão ligados diretamente às portas de entrada dos módulos. O código de conectores de coordenação não precisa ser entrelaçado com os módulos. Isso é factível, por exemplo, com a introdução de métodos para a inspeção do estado no módulo sob coordenação. Mantendo-se esta separação de interesses, é possível que conectores de coordenação sejam reconfigurados e que a implementação de conectores com guardas seja automatizada.

2.2.4. Composição. Ao descrever-se a arquitetura de uma aplicação, essencialmente descreve-se uma composição de módulos e conectores. Sob esta perspectiva toda aplicação pode ser considerada uma composição de módulos. Um módulo composto é formado por módulos primitivos ou outros módulos compostos, interligados por conectores. Os outros módulos da aplicação têm acesso ao módulo composto apenas através das portas com visibilidade externa. Para o resto da aplicação, um módulo composto tem a mesma apresentação de um módulo primitivo. O módulo composto da Figura 2 (*MC1*) é formado por três módulos primitivos (*Mp1*, *2* e *3*).

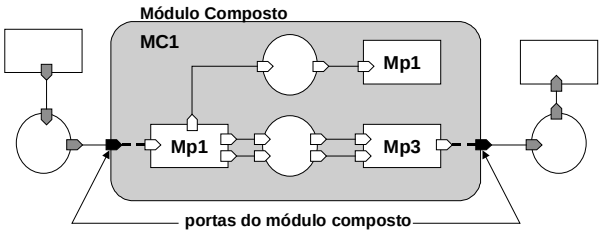

Figura 2 - Uma aplicação com módulo composto

A composição pode ser usada para a construção de módulos com função agregada ou mais especializados. Além disso, módulos compostos podem ser usados na estruturação de outras aplicações, ampliando as possibilidades de reutilização.

Conectores podem ser considerados como módulos especializados em promover a interação entre outros módulos. Entretanto, a separação dos conceitos de módulos e conectores se justifica com a possibilidade de se associar formalismos para a verificação automática de propriedades das aplicações sendo configuradas [4, 2]. Assim sendo, a concepção de um conector composto através da composição de conectores mais simples também é possível. Embora a idéia da composição de conectores seja simples, existem algumas implicações que devem ser consideradas, tais como:

- verificar a ortogonalidade, ausência de conflitos ou interações indesejadas dos aspectos sendo compostos para garantir a coerência do resultado [16, 28, 14];
- dispor de ferramentas para facilitar a composição automática de conectores, a partir de características descritas na configuração;

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

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

2.2.5. Conectores Genéricos. O mapeamento de certos tipos conectores para uma implementação pode ser otimizado com a introdução do conceito de *conector genérico*. Em princípio a implementação de conectores é dependente das interfaces de outros componentes, e o conector deve conhecê-las para poder intermediar suas interações. Um conector Java-RMI, por exemplo, só poderá ser utilizado para o conjunto de portas para o qual ele foi preparado. Observou-se, entretanto, que, para certos casos, a implementação de um conector pode se adaptar à interface dos módulos em tempo de instanciação. Desta forma, um conector genérico é implementado independentemente do conjunto de portas dos módulos que ele irá conectar. Apenas no momento da ligação dos módulos o conector é adaptado para permitir o fluxo das requisições e respostas específicas. Por exemplo, um conector genérico que encapsule o aspectos de comunicação utilizando RMI poderá ser utilizado, de forma flexível, para ligar quaisquer pares de módulos. Como vantagens, não existe a necessidade do uso de bibliotecas por parte dos módulos envolvidos na comunicação, e nem a necessidade de declaração de interfaces e geração de *proxys* e *stubs* [22].

2.2.6. Outros Aspectos. Além dos aspectos apresentados, outros podem ser necessários na concepção de aplicações distribuídas e são, em princípio, realizáveis nos conectores. Dentre eles destacam-se:

- **Replicação e tolerância a falhas** - Permite a configuração de módulos replicados e a seleção de esquemas de tolerância a falhas. Experiências com configuração de replicação e

tolerância a falhas podem ser vistas em [24];

- **Persistência -** Permite configurar aspectos de persistência para os módulos. Este aspecto pode ser utilizado em conjunto com o aspecto de tolerância a falhas para facilitar a implementação de esquemas de recuperação de módulos que falharam e em sistemas de suporte à transações [13];
- **Tempo Real** - Características relacionadas com restrições temporais ou o escalonamento de módulos [9] podem ser especificados através do uso de conectores.

De uma maneira geral, aspectos relacionados com a interação entre módulos, como a validação do conteúdo dos parâmetros de requisições, controle de versões ou a criptografia das informações manipuladas durante as interações, podem ser encapsulados em conectores.

2.3. A linguagem de Configuração

A linguagem de configuração que estamos propondo permite, como outras linguagens de configuração, a descrição, o armazenamento e a recuperação da descrição de módulos, conectores e a estrutura de aplicações. Em adição aos recursos tradicionais das linguagens de configuração estamos oferecendo recursos para a descrição de aspectos não-funcionais, como os apresentados na seção 2.2. A linguagem proposta é batizada de CBabel (*Building Applications by Evolution with Connectors*). Os fundamentos de uma versão anterior da linguagem, Babel, são discutidos em [27].

A linguagem de configuração foi concebida com dois grupos de declarações. O primeiro grupo é responsável pela declaração dos tipos e classes dos elementos a serem usados na aplicação. Neste grupo estão as declarações *module*, *port* e *connector*. O segundo grupo permite que se declare a estrutura da aplicação através de comandos. *instantiate*, *link* e *start* estão neste grupo. Com o suporte do configurador estes comandos também podem ser utilizados para instanciar aplicação ou para reconfigurar-se esta mesma aplicação durante sua operação. Durante a instanciação da aplicação, o configurador pode fazer adaptações na configuração para que ela opere em um ambiente específico, como por exemplo selecionar versões compatíveis de módulos para determinado sistema operacional ou selecionar um conector de comunicação adequado para as condições de tráfego.

3. Trabalhos Correlatos

Propostas correlatas como *Aspect Oriented Programming* (AOP) [18] e linguagens que oferecem reflexão computacional [11], além de outros ambientes de configuração [26, 4 e 32], foram analisadas em outro trabalho [33]. Essas propostas têm em comum a utilização de alguma forma de interceptação do fluxo de execução e interposição de código [17] para obter a separação de interesses. AOP, por exemplo, possui uma ferramenta chamada *weaver*, que entrelaça o código de aspectos programados separadamente em um único bloco [23]. Linguagens de programação com características reflexívas normalmente utilizam uma combinação de identificação e interceptação do fluxo de execução do nível base (*reificação*), para disparar a execução de um código interposto no meta-nível (*reflexão*). Nos dois casos, existe uma ferramenta que administra a interceptação e interposição de código. Identificamos também que a tecnologia de configuração possui características semelhantes a cada uma das tecnologias estudadas, mas oferece explicitamente um esquema para descrever a arquitetura das aplicações através de composição e a possibilidade de verificações de propriedades dessas arquiteturas em alto nível de abstração.

4. TicTacToe com AOP

Para ilustrar as características em torno de R-RIO, uma solução para o jogo da Velha, utilizando-se uma arquitetura mais modular do que a original e um *design pattern*, é apresentada nesta sessão. Realiza-se também uma comparação desta solução com uma versão elaborada sob a ótica de AOP, o que permite ressaltar algumas vantagens da abordagem R-RIO.

Em AspectJ™ [5, 20], uma implementação de AOP, uma solução alternativa do jogo da Velha foi utilizada para exemplificar como um *design pattern* do tipo ***Observer*** [10], que trata da geração de eventos em um objeto observável e da reação esperada de *observadores*, pode ser concentrado em um aspecto, separando explicitamente a programação dos objetos de como estes interagem. A arquitetura (Figura 3) apresenta 4 objetos: o jogo, propriamente dito, implementado por uma classe *TicTacToe*, um objeto *Player*, que interage com o jogo e dois objetos que apresentam os movimentos do jogo, *BoardDisplay* e *StatusDisplay*. Apenas os métodos relacionados com o protocolo de interação são destacados, mas deve ser subentendido 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.

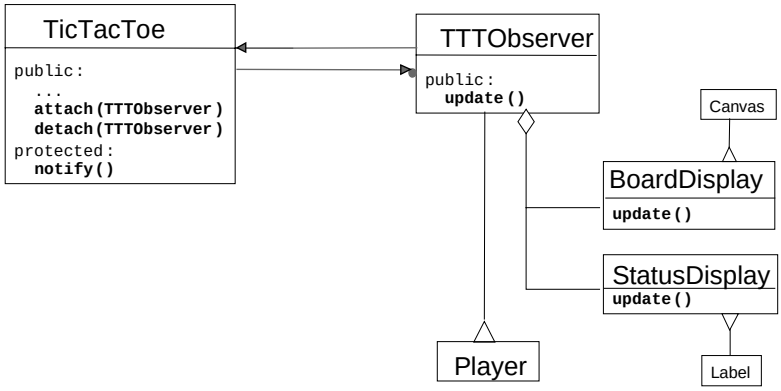

Figura 3 - Arquitetura da aplicação TicTacToe retirado de [5]

O protocolo de interação baseado no *design pattern Observer*, descrito em AspectJ™ (um *superset* de Java), é apresentado na Listagem 1. Observa-se que os métodos *attach*, *dettach* e *notify* e a forma de utilização destes métodos não fazem parte dos objetos originais. Estes elementos são encapsulados no aspecto *TTTDisplayProtocol* e devem ser introduzidos no código do módulo original através do *weaver*.

A ferramenta de entrelaçamento (o *weaver*) pode introduzir novas variáveis ou métodos (***introduce***) e pode também acrescentar código (***advise***) em determinados pontos de junção em classes programadas (em Java) para desempenhar as funções básicas da aplicação. Seguindo a listagem 1 as seguintes observações podem ser feitas sobre o aspecto *TTTDisplayProtocol*:

- a classe *TicTAcToe* precisa declarar uma instância de *TTTObserver* para "tornar-se observável". Isto é feito através dos *weaves* ***introduce*** que introduzem no código original de *TicTacToe* a criação da instância de um vetor de observadores (linha 2) e outros métodos para suportar o cadastro (*attach* e *dettach*) 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. No caso, tanto na classe *Board* (linha 14) como na classe *Status* (linha 15) são chamados os métodos *updateStatus* e *repaint*.

```
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 1 - Protocolo de interação em AspectJ™

Além de introduzir métodos e variáveis, o aspecto 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 acrescentado 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 Figura 3).

A separação de interesses é mantida entre aspectos funcionais e não funcionais, até que os aspectos sejam pré-processados pelo *weaver*. Após este passo, código dos aspectos não-funcionais é entrelaçado nas classes do código original, resultando em um único programa em Java. Observa-se que para este o esquema funcionar existe alguma quebra de encapsulamento do código dos objetos originais. Vale notar que o *weaver* trabalha de forma reflexiva para gerar o código final, utilizando-se de informações de meta-nível como o nome de métodos e pontos de junção como o início ou o fim destes métodos. Uma ferramenta como AspectJ possui mecanismos comparáveis aos de linguagens reflexivas como OpenC++ [8].

5. TicTacToe em R-RIO

A arquitetura do jogo da Velha apresentada na seção anterior pode ser é implementada segundo a abordagem de 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 1). Cada um destes módulos, embora tenham-se omitido detalhes, podem ser concebidos por composição, por exemplo agregando módulos especializados em exibir as informações ou entrada de dados.

O módulo *TTTGame* pode ser concebido sem a preocupação com o número de

observadores ou com a geração de notificações. Como toda interação (requisições e respostas) entre *TTTPlayer* e *TTTGame* passa pelo conector, o próprio conector toma a iniciativa de enviar requisições de *update* para os observadores, juntamente com as informações de *status*, quando necessário e apropriado (antes de uma requisição ou após a requisição ser completada). Para isso ser possível basta que um método G*etstatus*, por exemplo, que retorne o estado do jogo, esteja disponível no módulo *TTTGame*. A descrição dos módulos desta arquitetura, com CBabel, é apresentada na Listagem 2.

```
1    port Turn {
2       parms (int Row, int Col);
3       return (int Ack);
4    }
5
6    port GetStatus {
7       parms (void);
8       return (int Status);
9    }
10
11   port Update {
12       parms (int Status);
13       return (int Ack);
14   }
15
16   // declaração de outros tipos  portas (omitidas)
```

```
17   module   TTTGame {
18       in port Turn;
19       in port Getstatus;
20       // … outras portas do módulo
21       map CLASS Java "game"
22   } Game;
23   module   TTTPlayer {
24       out port Turn SaidaItem;
25       // … outras portas do módulo
26       map  CLASS  Java "player";
27   } Player;
28   module   TTTDisplay {
29       in port Update;
30       // … outras portas do módulo
31       map  CLASS Java "game_grid";
32   } DisplayGrid;
```

Listagem 2 - Declaração de portas e módulos

Inicia-se a descrição da aplicação definindo-se tipos de portas (linhas 1, 6 e 11) e classes de módulos que serão utilizados. Os tipos de portas e módulos são declarados aqui para serem reutilizados posteriormente. Os módulos são declarados indicando-se instâncias de portas (linhas 18 e 19, por exemplo) e o mapeamento para uma implementação (por exemplo, o módulo *TTTGame* é implementado pela classe *game.class*, programada em Java - linha 21). De forma semelhante os outros módulos são configurados. Em seguida o conector que interliga os módulos é definido (listagem 3).

```
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    contracts {
12       // descrição dos contratos 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 3 - Declaração do conector

Na classe de conector *CObserver* as portas também devem ser declaradas (linhas 2 a 4). Note-se que duas instâncias de portas do tipo *Turn* foram declaradas. *TurnIn* (linha 2) será

utilizada pelo módulo *Player* e a porta *TurnOut* (linha 4) será utilizada pelo módulo *Game*. As portas *Update* e *Getstatus* não receberam nomes de instância. O configurador poderá renomear qualquer referência, quando isso se fizer necessário, para resolver conflitos de nomes. Aspectos de sincronização entre as portas do conector são declaradas em seguida (linhas 7 a 9). O *design pattern Observer* é associado ao conector através da cláusula *contracts.* Dentro desta cláusula pode ser programado como o conector vai gerenciar as interações entre os módulos. Os aspectos declarados em conectores podem dar origem a um esqueleto de código que deve ser preenchido ou complementado no nível da linguagem de programação. Para o conector *CObserver*, o código será gerado em Java. Por outro lado, se uma implementação deste conector já estivesse disponível, a referência na cláusula *map* (linha 17) seria suficiente para a declaração do conector, e as linhas de 2 a 15 poderiam ser omitidas.

```
1    module  TicTacToe  {
2      instantiate   Game, Player, DisplayGrid;
3      instantiate   CObs;
4      link Player, DisplayGrid to Game by CObs;
5    }
6    module  TicTacToe  TTT;
7    instantiate  TTT;
8    start TTT;
```

Listagem 4 - Declaração da estrutura da aplicação

Após a descrição dos módulos e conectores, a própria aplicação é descrita como um módulo (listagem 4). Neste módulo é descrita configuração da estrutura de interconexão dos módulos e conectores declarados (linhas 2 a 4). Em seguida, cria-se também uma instância de *TicTacToe* (linhas 6 e 7). Neste momento são criadas todas as instâncias dos módulos internos e realizadas as conexões especificadas. A partir daí a instância *TTT* da aplicação foi criada e com o comando *start* (linha 8) sua execução iniciada.

Na abordagem R-RIO, observadores não precisam executar métodos de *attach* ou *dettach* no módulo do jogo. Basta que o novo observador seja ligado ao jogo através de sua porta de entrada. Além disso, a comunicação com um grupo de observadores pode ser otimizada. Na versão de AOP, o objeto *TTTObserver* tem que tomar a iniciativa de *varrer* um vetor de observadores e invocar o método *update* em cada um. No conector, é possível utilizarmos um esquema de *multicast* para disseminar as informações de atualização de estado para todos os observadores. Uma outra possibilidade é a criação dos jogadores e observadores em um sistema distribuído. Estes e outros aspectos como a comunicação entre os módulos podem ser contemplados pela composição de conectores diferentes, cada qual responsável por um aspecto da interação, para formar um conector que encapsule todos os aspectos desejados, além da implementação da interação entre os módulos segundo o padrão *Observer*. A listagem 5 exemplifica esta nova configuração com um conector implementando comunicação por RMI (linha 1) e instanciando os módulos em estações diferentes (linhas 5 a 7).

```
1    connector Comm {
2       map CLASS Java "RmiGCon";
3    } PCRmi;
4
5    instantiate   DisplayGrid at estação1;
6    instantiate   Player at estação2;
7    instantiate   Game at estação3;
8    instantiate   PCRmi, CObs;
10   link Player, DisplayGrid to Game by CObs | PCRmi;
```

Listagem 5 - Reconfigurando a aplicação

Comentários. Com esforço equivalente a AOP e com menos interferência nos módulos originais foi possível conceber a mesma aplicação em R-RIO. Em relação à *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 para cada uma ou através de construções com índices na linguagem de configuração. Vale ainda lembrar que como é a arquitetura da aplicação que está sendo descrita, também é possível a utilização de uma ferramenta para geração do código dos módulos e conectores, caso estes não estejam disponíveis em um repositório.

6. Conclusão

R-RIO, uma nova abordagem para a construção de aplicações distribuídas foi apresentada. Nesta abordagem, baseada em configuração, utiliza-se conectores para intermediar as interações entre módulos e também encapsular aspectos não-funcionais das aplicações. A arquitetura de uma aplicação é descrita através de uma linguagem de configuração e os vários aspectos envolvidos podem ser tratados separadamente. A implementação de separação de aspectos não é trivial se o objetivo é permitir aplicar este conceito durante todo o processo de gestão de uma aplicação, inclusive durante a execução, e permitindo-se reconfigurações. Assim sendo, também foi proposto um mapeamento do modelo conceitual para o nível de implementação. Este mapeamento poderá ser feito através de ferramentas automáticas e pode ainda contar, eventualmente, com ambientes especializados.

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

A metodologia de R-RIO para a concepção de aplicações foi comparada com protótipos de outras propostas, além de AOP, como *Composition Filters* [6], OpenC++ [8], MetaXa [11], UniCon [32] e Conic/Regis/Darwin [26], através do desenvolvimento de um mesmo conjunto de exemplos, seguindo a mesma linha do que foi feito no presente trabalho [33]. Verificou-se que R-RIO possui expressividade equivalente, e pode ser mais flexível do que as outras abordagens. Adicionalmente, a abordagem utilizada em nosso modelo permite um tratamento integrado do gerenciamento das aplicações desenvolvidas, desde a concepção até a execução e manutenção, diferentemente de outras propostas avaliadas, onde isso é feito de maneira empírica. Nossa proposta de configuração, por exemplo, permite que o mesmo modelo utilizado para conceber a arquitetura de uma aplicação seja utilizado na sua reestruturação, em resposta a novas demandas, mesmo depois desta ser implantada e estar em produção. Esta característica torna R-RIO uma base para o desenvolvimento de aplicações que possam evoluir dinamicamente.

O protótipo de um ambiente Java para construção de sistemas, baseado na proposta de R-RIO é descrito em [22], estando disponível para atividades de pesquisa em http://www.caa.uff.br/~rrio. A escolha de Java deveu-se a algumas de suas características, tais como, o suporte nativo para concorrência e sincronização e a possibilidade de se intervir no carregamento das classes em tempo de execução. No entanto, outra combinação de linguagem de programação e suporte, como CORBA, poderia ter sido escolhida para implementar o protótipo [Loq97].

O modelo de R-RIO está sendo aperfeiçoado em vários pontos. A possibilidade da inclusão de novos aspectos será experimentada. A capacidade de composição e reflexão do modelo serão exploradas visando ampliar sua utilidade e abrangência em domínios diferentes de aplicações distribuídas e não distribuídas.

Referências Bibliográficas

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

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

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

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

[5] ____, "AspectJ™ Specification", Xerox, Paulo Alto Center, Março, 1998.

[6] Bergmans, Lodewijk M.J.; “Composing Concurrent Objects - Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programmming”, PhD Thesis, Universiteit Twente, Finlandia, 1996.

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

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

[9] Fraga, J., Farines, J., Furtado, O., Siqueira, F., “Programação de Aplicações Distribuídas Tempo-Real em Sistemas Abertos, XXIII Seminário Integrado de Software e Hardware, Recife, PE, Agosto, 1996.

[10] Gamma, E. et. *Alli*, “Design Patterns - Elements of Reusable Object-Oriented Software”, Addison-Wesley Publishing Co., EUA, 1995.

[11] Golm, M.; “Design and Implementation of a Meta Architecture for Java”, M.Sc. Thesis, Instituto de Matemática, Erlanngen-Nüremberg University, Alemanha, 1997.

[12] Helm, R., Holland, I.M. e Gangopadhyay, D., "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", OOPSLA'90, Ottawa, Outubro, 1990.

[13] Issarny, V. and Bidan, C., “Aster: A CORBA-based Software Interconnection System Supporting Distributed System Customization”, International Conference on Configurable Distributed Systems, Annapolis, Maio, 1996.

[14] Jackson, M, e Zave, P., "Distributed Feature Composition: A Virtual Architecture for Telecommunications Services", IEEE Trasaction on Software Enginnering, Vol. 24, No. 10, pp. 831-847, Outubro, 1998.

[15] Arnold, K. e Gosling, J., "The Java Programming Language", Second Edition, Sun Microsystems, Addison Wesley, EUA, 1998.

[16] Johnson, R. E., "Frameworks = (Components + Patterns)", Communications of the ACM, pp. 39-42, Outubro, Vol. 40, No. 10, 1997.

[17] Jones, M. B., “Interposition Agents: Transparently Interposing User Code at the System Interface”, 14th ACM Symp.on Operating Systems Principles, Asheville, 1993.

[18] Kiczales, G. and Irwing, J. et alli, “Aspect-Oriented Programming”, Position Paper for the ACM Workshop on Strategic Directions in Computing Research, MIT, June, 1996.
[19] Kiczales, G., Lamping, J., Lopes, C. V., Maeda, C., Mendhekar, A., "Open Implementations Design Guidelines", 19th International Conference on Software Engineering, ACM Press, 1997.
[20] Kiczales, G., Lopes, C. V., "Aspect Oriented Programming with AspectJ - Tutorial / Part 4", Xerox PARC , www.parc.xerox.com/aop, 1998.
[21] Laddaga, R. and Veitch, J., “Dynamic Object Technology”, Communications of the ACM, Vol. 40, No. 5, May, 1997.
[22] Lobosco, M. "R-RIO: Um Ambiente de Suporte à Construção e Evolução de Sistemas Distribuídos", dissertação de mestrado, CAA/UFF, Março,1999.
[23] Lopes, Cristina I. V.; “D: A Language Framework for Distributed Programming”, Doctor of Philosophy Thesis, College of Computer Science, Northeastern University, Novembro, 1997.
[24] Loques, O., Botafogo, R., Leite, J., “A Configuration Approach for Distributed Object-Oriented System Customization”, IEEE Third International Workshop on Object-oriented Real-time Dependable Systems, Newport Beach, Fevereiro, 1997.
[25] Loques, O., Leite, J., Sztajnberg, A e Lobosco, M., “Towards Integrating Meta-Level Programming and Configuration Programming”, SBES 99, Florianópolis, Outubro, 1999.
[26] Magee, J., Dulay, N., Kramer, J., "Structuring parallel and distributed programs", Int. Workshop on Configurable Distributed Systems, pp. 102-117, IEE, Março, 1992.
[27] Malucelli, V. V, "Babel – Construindo Aplicações por Evolução", dissertação de mestrado, DEE / PUC-RJ, Fevereiro, 1996.
[28] Monroe, R. T., Kompanek, A., Melton, R. e Garlan, D., "Architectural Styles, Design Patterns and Objects", IEEE Software, Vol. 14, No. 1, pp. 43-52, Janeiro de 1997.
[29] Oliva, A., e Buzato, L. E., "Composition of Meta-Objects in Guaraná", ECOOP 98, Workshop Reflection, Canada, 1998.
[30] OMG, “The Common Object Request Broker: Architecture and Specification”, Revision 2.0, Julho, 1996.
[31] Rine, D. C., "Supporting Reuse with Object Technology", IEEE Computer, Vol. 30, No. 10, pp. 43-45, Outubro, 1997.
[32] Shaw, M., DeLine, R., et alli, “Abstractions for software architecture and tools to support them”, IEEE Transactions on Software Engineering, Vol. 21, No. 4, Abril, 1995.
[33] Sztajnberg, A, “Cap 2 - Análise de Propostas Correlatas”, proposta de tema de tese de doutorado, COPPE/UFRJ, Julho, 1999.
[34] Wegner, P., “Interoperability”, ACM Computing Surveys, Vol. 28, No. 1, Março,1996