Conector Genérico: Um Mecanismo para Reconfiguração de Aplicações baseadas em Componentes

Thaís Batista *
Departamento de Informática
Universidade Federal do Rio Grande do Norte – UFRN

Christina von Flach Chavez *
Departmento de Ciência da Computação
Laboratório de Sistemas Distribuídos – LaSiD
Universidade Federal da Bahia – UFBA

Noemi Rodriguez
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro – PUC-Rio

thais,flach,noemi@inf.puc-rio.br

Resumo

Na programação baseada em componentes é interessante que as aplicações sejam modeladas como um conjunto de serviços executados por componentes que são selecionados em tempo de execução. Para dar suporte a este aspecto, neste trabalho é apresentado o conceito de Conector Genérico, um mecanismo que em tempo de execução seleciona componentes que oferecem os serviços estabelecidos na configuração de uma aplicação, ativa cada serviço e devolve os resultados. Para desempenhar estas funções, o Conector explora a potencialidade de Lua, uma linguagem interpretada com tipagem dinâmica, e de LuaOrb, um *binding* entre CORBA e Lua que permite acesso dinâmico a objetos CORBA. Através do Conector Genérico é possível configurar uma aplicação como um conjunto de serviços sem conhecer previamente os componentes que os oferecem. Estes são selecionados *on-the-fly*, o que consiste em um poderoso mecanismo de reconfiguração dinâmica.

1 Introdução

A programação de aplicações baseadas em componentes tem sido assunto de várias pesquisas nos últimos anos [NGT92, SN99, MTK97] principalmente por promover a reusabilidade de

*Aluna de Doutorado do DI - PUC-Rio

componentes de software, reduzindo o tempo e o custo de desenvolvimento. Consideramos que componentes são entidades de software compiladas separadamente que, interagindo entre si através de suas interfaces, colaboram para realizar uma tarefa. Dois tipos de modelos têm um papel fundamental no suporte ao desenvolvimento de software baseado em componentes: os modelos de integração de objetos, que abstraem as complexidades relativas à programação e utilização de componentes, e os modelos que facilitam a estruturação das aplicações como um conjunto de componentes em interação.

Os modelos de integração de objetos, também chamados de infra-estruturas de componentes de software [1] [OHE96] ou simplesmente modelos de componentes [Ber96, Sti94], incluem abstrações para definição da interface, comunicação, heterogeneidade e distribuição de componentes. Estes modelos utilizam o padrão cliente-servidor no qual a interação entre os componentes é modelada como solicitações cliente-servidor. CORBA, COM e JavaBeans são exemplos de modelos de componentes. O CORBA tem se destacado em relação aos demais modelos por ser uma especificação aberta, independente de fabricante e de linguagem. Por este motivo, no contexto deste trabalho, consideraremos o CORBA como representante desta categoria.

A estruturação de aplicações como um conjunto de componentes em interação, comumente tratada como configuração da aplicação[2], envolve as tarefas de (i) determinar o conjunto de componentes que provêm a funcionalidade desejada e (ii) estabelecer as interações entre eles. Tradicionalmente, configuração da aplicação restringe-se a interligar interfaces de componentes cujas requisições de um lado correspondem a provisões do outro lado [KM85, MTK97, All97]. Este modelo de configuração, estabelecido pelo Paradigma da Configuração [KM85], apresenta alguns problemas relacionados ao padrão restrito de interligar diretamente requisições a provisões de serviços.

Um dos problemas ocorre quando os componentes utilizados seguem os modelos integradores, por exemplo, componentes CORBA, cuja interface declara apenas os serviços oferecidos e apenas na implementação são decididos quais serviços utilizar. Neste caso, como a priori não se tem a informação sobre os serviços que serão requisitados pelo componente, na configuração da aplicação não se tem como fazer a interligação direta entre componentes. Para resolver este problema, alguns trabalhos estendem a interface do componente incluindo os serviços requisitados [BR96], outros empacotam o componente em outro, que apresenta a informação a respeito dos serviços requisitados [SO98]. Recentemente, alguns trabalhos [DR97, MTK97, SN99] têm ressaltado que enriquecer a interface do componente com informação de conexão ou empacotá-lo em um novo componente, compromete a reusabilidade e, o frequente empacotamento pode ocasionar problemas de performance. Em [BR99] discute-se como uma linguagem de *scripting* interpretada pode oferecer suporte à configuração e reconfiguração dinâmica de aplicações baseadas em componentes. Diferentes formas de interconexão são apresentadas porém, pressupõem o conhecimento dos componentes que oferecem os serviços que compõem uma aplicação. Uma situação desejável é modelar a aplicação sem conhecer previamente componentes que oferecem determinados serviços. Estes serão

[1] *software component infrastructure*

[2] Atualmente, os trabalhos nesta área estão sendo integrados com *Arquiteturas de Software*.

inseridos na aplicação no decorrer da execução. Assim, no desenvolvimento da aplicação são estabelecidos quais os serviços utilizados, e os componentes que oferecem estes serviços devem ser descobertos durante a execução. Isto introduz uma grande flexibilidade para a modelagem da aplicação e ao mesmo tempo endereça um importante aspecto das aplicações atuais: a necessidade de reconfiguração dinâmica.

Outro problema é a falta de suporte para adaptação de componentes que não foram concebidos para trabalhar juntos [SN99]. A resolução deste problema é tratada em [BR99] e não faz parte do escopo deste trabalho.

Neste trabalho, pretendemos discutir como o conceito de Conector [Sha94, BF96, AG95, DR97] pode ser aplicado para oferecer suporte a interconexão dinâmica de componentes CORBA, oferecendo uma maneira transparente de configurar e reconfigurar uma aplicação. Apresentaremos o Conector Genérico, um mecanismo que procura um componente que pode oferecer o serviço solicitado, ativa o serviço e devolve os resultados ao cliente. O nome "Conector" deriva da idéia de entidade de tempo de execução que controla a interação entre componentes [DR97], oferecendo uma maneira de separar a implementação do componente de aspectos de interconexão para formar uma aplicação. A denominação "genérico" é empregada para especificar tal conector cuja função não é interconectar componentes com perfis pré-determinados , mas procurar qualquer componente que atenda ao serviço requisitado. Tal componente pode até mesmo ser de origens diferentes (CORBA, COM, JAVA, etc), desde que exista um *binding* definido entre a linguagem de configuração utilizada e o modelo de componente.

A finalidade do Conector utilizado neste trabalho difere daquela de conectores usados em trabalhos de Arquitetura de Software [SDZ96, BF96, All97] para descrever abstratamente os tipos de componentes envolvidos em uma interação e como eles se relacionam, estabelecendo restrições de como as mensagens fluem entre os componentes. Por exemplo, em um sistema cliente-servidor, o conector descreve o comportamento do cliente como uma sequência de solicitações alternadas com a recepção dos resultados e o comportamento do servidor como espera de solicitações alternadas por retorno de resultados. Além disso, estabelece que as atividades devem ser sequenciadas na ordem: cliente solicita serviço, servidor espera solicitações, servidor envia o resultado, cliente recebe o resultado. Desta forma, estes conectores modelam o tipo de interação entre os componentes de uma aplicação. Em contraste, o Conector Genérico tem objetivo diferente e como tipo de interação considera sempre o modo cliente-servidor.

A linguagem de extensão Lua [IFC96], desenvolvida na PUC-Rio, será usada neste trabalho como linguagem de configuração da aplicação. Esta linguagem foi escolhida por ser interpretada e possuir um sistema de tipos flexível que permite acesso dinâmico a componentes. Além disso, este trabalho está inserido em um projeto que investiga a flexibilidade que uma linguagem interpretada pode conferir a um modelo de componentes [CCI99]. Várias ferramentas desenvolvidas no contexto do projeto serão usadas na nossa abordagem. LuaOrb [ICR98], um *binding* entre Lua e CORBA, baseado na Interface de Invocação Dinâmica (DII), disponibiliza o acesso dinâmico a objetos CORBA, possuindo um papel relevante na configuração da aplicação. O Conector Genérico explora a facilidade oferecida por LuaOrb para acesso dinâmico aos componentes.

Com o mecanismo descrito neste trabalho o ambiente de configuração Lua passa a oferecer suporte à configuração de aplicações em dois níveis. Em um nível, o programador estabelece os componentes que oferecem os serviços utilizados. No outro nível, mais automático, o programador especifica apenas os serviços que formam a aplicação e explora a flexibilidade oferecida pelo Conector Genérico que torna transparente as complexidades relacionadas a busca de componentes.

Este trabalho está estruturado da seguinte forma. A seção 2 apresenta brevemente os conceitos da linguagem Lua. A seção 3 discute LuaOrb. A seção 4 apresenta o Conector Genérico e alguns exemplos que ilustram a sua utilização na estruturação e reconfiguração de aplicações. Na seção 5 são delineadas as conclusões.

2 A Linguagem Lua

Lua é uma linguagem de configuração interpretada que surgiu da necessidade de se usar uma linguagem de extensão para adaptar aplicações industriais [IFC96]. Lua possui um sistema de tipos dinâmico. Variáveis não têm tipos, apenas valores estão associados a um tipo.

Como em linguagens procedimentais convencionais, Lua oferece estruturas de controle (`while`, `if`, etc.), definições de funções com parâmetros e variáveis locais. Além disso, inclui aspectos não usuais como:

- Funções são consideradas valores de *primeira classe*, o que significa que podem ser armazenadas em variáveis, passadas como argumentos para outras funções e retornadas como resultado. A definição de uma função cria um valor do tipo *function* e atribui este valor a uma variável global. Funções podem retornar vários valores, o que dispensa a passagem de parâmetros por referência para se obter múltiplos retornos.

- O tipo *table* (tabela) implementa *arrays associativos*, e é o único mecanismo para estruturar dados em Lua. Tabelas são objetos criados dinamicamente e podem ser indexados por qualquer valor da linguagem (inteiros, strings, reais, tabelas e valores de funções), exceto `nil`. Esta flexibilidade é a base do mecanismo para descrição de objetos em Lua. Vários tipos de dados como arrays, conjuntos, registros, podem ser implementados através de tabelas. Tabelas são criadas com expressões especiais chamados construtores. O construtor mais simples é { } que cria uma nova tabela vazia. Como tabelas são objetos, não são valores, variáveis nunca contém tabelas, apenas referências a elas.
  O suporte parcial à programação orientada a objetos em Lua (não permite herança) é oferecido através de funções e tabelas. Como funções são valores de primeira classe e campos de tabelas podem conter funções, a tabela representa o objeto e os métodos são funções armazenadas nos campos da tabela. A diferença entre métodos e outras funções é que o próprio objeto é passado implicitamente como o primeiro parâmetro de seus métodos. No corpo da função o objeto é acessado usando o parâmetro *self*.
  `Definições de métodos`: *function objeto:método(parâmetros) ... end*

`Chamada do método`: *objeto:método(parâmetros)*
que é açúcar sintático para *objeto.método(objeto, parâmetros)*.

- As *Facilidades Reflexivas* oferecidas por Lua permitem, por exemplo, verificação de tipo através da função *type* que retorna um string descrevendo o tipo do seu argumento. Como registros e objetos são representados por tabelas, pode-se percorrer facilmente todos os campos do objeto ou verificar a presença de determinados campos. Parte da natureza reflexiva é oferecida por *fallbacks* que permitem ao programador mudar o comportamento de Lua em condições especiais, como na presença de erros durante a execução ou no acesso a um campo inexistente em uma tabela. Por exemplo, se na expressão `receiver:foo(params)`, o *receiver* não for um objeto Lua ou *foo* não for um método de *receiver*, um erro deve ser sinalizado. Com fallbacks, o programa pode tratar este erro de maneira significante para o contexto onde ele está inserido. Esta facilidade é a base da implementação de LuaOrb.

Maiores informações sobre a linguagem em http://www.tecgraf.puc-rio.br/lua/.

3 LuaOrb

LuaOrb [ICR98, CCI99] é um *binding* entre a linguagem Lua e o CORBA que, da mesma maneira que todos os bindings entre linguagens de programação e o CORBA, define mapeamentos entre tipos Lua e IDL. Este mapeamento é baseado no sistema de tipos dinâmicos da linguagem Lua e na Interface de Invocação Dinâmica (DII) do CORBA. Como a DII permite que o usuário selecione o tipo do objeto e a operação durante a execução, LuaOrb optou por usar este mecanismo na sua interface com o CORBA ao invés de usar stubs IDL. Estes aspectos permitem que objetos CORBA sejam acessados via Lua como qualquer outro objeto Lua, sem declaração prévia.

Do lado do cliente, LuaOrb aproveita as características dinâmicas de Lua permitindo o acesso a objetos remotos CORBA, da mesma maneira que se faz acesso a objetos Lua. Além disso, o acesso é feito de forma dinâmica e transparente. Para usar um objeto CORBA, inicialmente tem-se que criar um *proxy* para este objeto através da função `createproxy(nome_da_interface, referência_objeto)`. Esta função retorna um objeto Lua que representa o objeto CORBA. Através do mecanismo de *fallbacks*, LuaOrb intercepta as operações aplicadas sobre o *proxy* e transforma-as em operações remotas.

Depois da criação do *proxy*, os métodos da sua interface podem ser chamados. A implementação da função `createproxy` é basicamente um mapeamento direto para a equivalente no ORB API. A seguir mostraremos um exemplo do uso de objetos CORBA via Lua. A Figura 1 mostra a interface do componente *paciente* definida em IDL.

A figura 2 ilustra o uso do objeto definido na interface da Figura 1.

Como mostrado no exemplo acima, a chamada a um método de um objeto CORBA via LuaOrb é feita usando a sintaxe de chamada de métodos em Lua.

LuaOrb oferece suporte para o modo de invocação síncrona adiada (com espera adiada). Para fazer esta chamada deve-se incluir o prefixo `deferred_` antes do nome do método. Por

```
struct dados_paciente{
    string nome;
    short pressao_max, pressao_min;
    short temperatura, pulso;
}
interface paciente{
   long adiciona_paciente(in dados_paciente dados);
   paciente monitora();
}
```

Figura 1: Interface do componente *paciente* em IDL

```
paciente = createproxy(``paciente'')
dados = {nome=``Joao''; pressao_max=12; pressao_min=8; temperatura=37; pulso=90}
ident = paciente:adiciona_paciente(dados);
```

Figura 2: Uso de Objetos CORBA via Lua

exemplo, `paciente:deferred_monitora()`.

A chamada adiada retorna um handler a ser usado para recuperar o resultado da execução do método. Como este tipo de chamada é bastante útil para programação orientada a eventos, LuaOrb oferece a função `completion_event` que observa a finalização da execução do método especificado na chamada adiada e estabelece a nova função que será executada após o término. Esta função é declarada da seguinte forma: `completion_event(objeto:deferred_método(parâmetros), função)`. A função determinada no segundo parâmetro da `completion_event` é invocada com o resultado do método da chamada adiada como parâmetro quando esta chamada retorna. Por exemplo, na chamada `completion_event(paciente:deferred_monitora(), print)`, quando o método `monitora` é finalizado, a função `print` é invocada tendo como parâmetro o componente retornado por `monitora`.

Do lado do servidor, LuaOrb usa as facilidades oferecidas pela DSI para permitir a extensão de objetos CORBA via Lua [CRI99]. Objetos Lua podem fazer o papel de servidores CORBA e clientes podem acessá-los via stubs ou DII. Analogamente ao que acontece do lado do cliente, é criado um *proxy* para o objeto Lua. Este objeto pode ser dinamicamente instalado no servidor. A interface `ServerLua` foi definida para permitir a um cliente instanciar um novo objeto e definir sua implementação com código Lua.

4 Conector Genérico

O Conector Genérico é uma abstração que permite se escrever uma aplicação apenas com os serviços que se deseja usar, sem estabelecer o(s) componente(s) que oferece(m) tais serviços. Os componentes são selecionados *on-the-fly* pelo Conector Genérico. No programa de configuração, o serviço é chamado da mesma forma que se chama um método de um componente conhecido, com a variável representante do Conector Genérico colocada ao lado do nome do método, dando a idéia de que o serviço é oferecido pelo Conector Genérico. Uma tabela, chamada *tabela de configuração*, gerenciada pelo Conector Genérico, reflete os serviços solicitados pela aplicação e os respectivos componentes que foram selecionados para oferecer os serviços. Consultando esta tabela, tem-se uma visão dos componentes que foram introduzidos *on-the-fly*.

O Conector Genérico é um objeto Lua representado por um *proxy* criado através da função `nome_proxy = generic_createproxy()`. Para dispor dos serviços do Conector Genérico, deve-se primeiro criar o *proxy* no programa de configuração e depois chamar o serviço desejado da seguinte forma: `nome_proxy:servico(parâmetros)`. Esta chamada é interceptada pela implementação do Conector Genérico que, implicitamente, aciona a função `connect(servico, parâmetros, nome_solicitante)`, cuja finalidade é procurar em um repositório padrão (um serviço de nomes ou um serviço de Trader) pelo componente que oferece `servico(parâmetros)`. Uma vez encontrando vários componentes que oferecem `servico(parâmetros)`, um deles é selecionado (para isto pode ser solicitada a intervenção do usuário). Em seguida, o Conector Genérico constrói a solicitação, que consiste em "montar" uma sequência de comandos Lua para criação do *proxy* do componente e ativação do serviço. Finalmente o serviço é ativado.

A *tabela de configuração* mantida pelo Conector Genérico contém registros com os seguintes campos: `(Interface do Servico, Servidor que o oferece, Cliente que solicitou)`, onde `Interface do Servico` é o par (nome, assinatura). Antes de procurar `servico (parâmetros)` no repositório padrão, o Conector Genérico consulta esta tabela para verificar se o mesmo serviço já foi encontrado anteriormente, evitando acesso ao repositório. Por outro lado, existe a flexibilidade de se estabelecer que toda busca deve ser feita sobre o repositório. Neste caso, a tabela de configuração não é consultada. Desta forma, como a seleção do componente é resolvida em tempo de execução, este mecanismo consiste em uma maneira de reconfigurar a aplicação visto que, a cada chamada de um mesmo serviço, um componente diferente pode ser selecionado para executá-lo.

Para ilustrar o uso do mecanismo, utilizaremos como base um exemplo simples e clássico na literatura de sistemas reconfiguráveis: o Sistema de Monitoração de Pacientes (SMP) [KM85]. Neste sistema, pacientes são monitorados por sensores que disparam alarmes quando algum dado vital está fora dos limites pré-determinados. O alarme é enviado para uma central de enfermagem ou, opcionalmente, para uma central médica, que irá tomar as devidas providências. A figura 1 ilustra a IDL do componente *paciente*. A figura 3 mostra uma possível configuração do sistema.

Neste exemplo o Conector Genérico é usado para representar um componente que tem em sua interface o método `alarme`. Este método é passado como segundo parâmetro da função completion_event. Consideramos que o método `monitora` permanece em execução enquanto os dados vitais do paciente permanecerem dentro dos limites. Quando algum dado vital sair do limite estabelecido, o método retorna um evento que implicará na chamada automática do método `alarme`. Neste caso, o Conector Genérico está atuando como um mecanismo de interconexão de componentes, uma vez que, a finalização do método `monitora`

```
pac = 1
while pac<5 do
    p = createproxy(‘‘paciente’’, pac)
    c = generic_createproxy()
    completion_event(p:deferred_monitora(), c:alarme)
    pac = pac + 1
end
```

Figura 3: Configuração do Sistema de Monitoração de Pacientes usando o Conector Genérico

do componente *paciente* determina a execução do método `alarme` do componente a ser descoberto pelo Conector Genérico.

Supondo que estejam disponíveis os componentes que implementam as interfaces descritas em IDL na Figura 4, o Conector Genérico iria encontrar o componente *enfermagem* e o componente *centralmedica* que satisfazem a chamada ao método `alarme` presente no programa de configuração, e selecionaria um deles de acordo com a política escolhida pelo usuário: selecionar o primeiro componente encontrado, selecionar o último componente encontrado, solicitar intervenção do usuário. A política *default* é selecionar o primeiro componente encontrado. Supondo que o componente *enfermagem* seja o primeiro encontrado, o Conector, usando LuaOrb, determina a execução do método.

O potencial de reconfiguração é um aspecto intrínseco à filosofia do Conector Genérico. Considerando-se o mesmo exemplo e supondo-se que o componente *enfermagem* não estivesse disponível numa subsequente execução da aplicação, o Conector Genérico selecionaria outro componente, que no caso seria o componente *centralmedica*. Este cenário ilustra como a reconfiguração acontece de maneira automática e sem a necessidade do programador da aplicação precisar estabelecer a situação de reconfiguração.

O componente *centralurgencia*, apesar de oferecer um método `alarme` com um parâmetro *paciente*, possui outro parâmetro que não é compatível com a chamada realizada no programa de configuração pois o método `monitora` retorna apenas um dado do tipo paciente que será o argumento do método `alarme`. Este é um outro aspecto endereçado por nossa abordagem de suporte a configuração de aplicações: compatibilizar componentes que originalmente não foram projetados para trabalhar juntos e que possuem incompatibilidade na interface. O trabalho [BR99] discute e exemplifica como uma linguagem interpretada pode ser um instrumento para adaptação de componentes incompatíveis.

Uma outra maneira de configurar a aplicação descrita na Figura 3 está ilustrada na Figura 5. No exemplo, ao invés de passar um método CORBA como segundo argumento da função `completion_event`, é passada uma função Lua.

A função `trataevento`, chamada na finalização do método `monitora`, determina a execução do método `alarme` cujo componente será encontrado pelo Conector Genérico e incrementa o número de alarmes solicitados pelo paciente. Caso o alarme seja repetido cinco vezes, supõe-se que é uma condição de extrema emergência e para isto o método `urgencia` será invocado.

```
interface enfermagem {
   void addpaciente(in  paciente p)
   void alarme(in paciente p)
}

interface centralmedica {
   void alarme(in paciente p)
}

interface centralurgencia {
   void alarme(in paciente p, in dado_critico d)
   void urgencia(in paciente p)
}
```

Figura 4: Interface IDL de componentes acessíveis pelo Conector Genérico

Neste exemplo a capacidade de reconfiguração da aplicação não está apenas no Conector Genérico, mas também no comando condicional oferecido pela linguagem Lua que pode dinamicamente determinar a seleção de novos componentes de acordo com condições satisfeitas em tempo de execução. O uso destes recursos é bastante conveniente para uma aplicação como o SMP onde é importante que o paciente possa ser atendido o mais rápido possível por qualquer uma das entidades com competência para tal.

A flexibilidade oferecida pelo mecanismo do Conector Genérico é bastante adequada à configuração de aplicações baseadas em CORBA, formadas basicamente por um conjunto de serviços disponibilizados por vários componentes. Além disso é bastante compatível com a idéia de um banco de componentes provendo vários serviços e o Conector Genérico permitindo o programador abstrair-se da tarefa de buscar os componentes fornecedores dos serviços desejados.

Esta abordagem favorece o potencial de reusabilidade dos componentes pois torna possível o uso de componentes da maneira como eles foram projetados, sem modificações, e os componentes não têm conhecimento de que estão participando de uma ou outra interação. Isto contrasta com abordagens que, para inserir componentes em uma aplicação, modificam a interface do componente para cada interação onde ele pode ser engajado [AWB+94] e também com abordagens onde a conexão entre componentes é determinada dentro da definição de um componente composto.

5 Conclusões

Este trabalho apresentou uma ferramenta importante para estruturação e reconfiguração de aplicações baseadas em componentes: o Conector Genérico. Através desta ferramenta é possível se configurar uma aplicação declarando o conjunto de serviços sem conhecer os

```
pac = 1
while pac<5 do
   p = createproxy(``paciente'', pac)
   c = generic_createproxy()
   completion_event(p:deferred_monitora(), trataevento)
   pac = pac + 1
end

numalarme = 0

function trataevento(p)
   c:alarme(p)
   numalarme = numalarme + 1
   if numalarme > 5 then
      c:urgencia(p)
   end
   completion_event(p:deferred_monitora(), trataevento)
end
```

Figura 5: Configuração com Conector Genérico e Função Lua

componentes que oferecem cada um deles. Na configuração da aplicação, o Conector Genérico aparece como o elemento que fornece o serviço, mas o seu papel é representar um componente que eventualmente o fornecerá. Tal componente só será identificado em tempo de execução, quando o Conector irá buscar um componente que realmente oferece o serviço, e em caso de sucesso, irá ativar o serviço e enviar a resposta para o solicitante. Desta forma, o Conector é uma entidade de suporte em tempo de execução que coordena a utilização de componentes em aplicações reconfiguráveis.

Os serviços de busca de objetos, como serviço de Nomes ou serviço de Trader, são os que possuem funcionalidade que mais se assemelha à do Conector Genérico. Porém, o Conector Genérico é uma abstração sobre estes serviços. Sua função ultrapassa à dos serviços de busca que são apenas repositórios de informações sobre componentes e oferecem formas de consulta sobre este repositório. O Conector Genérico confere à aplicação capacidade de reconfiguração e uso de serviços de componentes não determinados da mesma maneira que se usa serviços de componentes conhecidos. Além disso não apenas seleciona componentes que podem oferecer os serviços requisitados pela aplicação, mas também ativa o serviço e devolve o resultado.

O mecanismo apresentado revelou-se bastante adequado no suporte à configuração de aplicações CORBA, uma vez que a interface dos seus componentes contém um conjunto de assinaturas de métodos e, em geral, as aplicações consistem em um conjunto de serviços que implementam a funcionalidade desejada. Além disso, o Conector Genérico satisfaz a sugestão de [DR97] que critica o fato de se enriquecer interfaces de componentes com protocolos que capturam a informação de interação em contextos particulares.

Uma das principais vantagens desta abordagem é disponibilizar uma maneira implícita de selecionar componentes, unindo-os para formar uma aplicação, sem o programador ter de se preocupar em procurar o componente necessário nem fazer adaptações na interface do componente para adequá-lo ao contexto de utilização. Assim, evita-se especificar informação de conexão na interface do componente, como apresentado em alguns trabalhos [BR99, MTK97]. Com isto, o Conector Genérico possibilita maior independência dos componentes em relação ao seu contexto de uso e, portanto, promove o reuso de componentes.

Outro aspecto importante refere-se ao potencial de reconfiguração automática que o Conector Genérico introduz na aplicação uma vez que a cada chamada de um mesmo serviço, diferentes componentes podem ser selecionados para executá-lo. Com esta abordagem automática o ambiente de configuração Lua passou a oferecer um nível adicional de suporte à reconfiguração de aplicações. Em um outro nível, convencionalmente tratado em sistemas reconfiguráveis, a reconfiguração é estabelecida de maneira explícita.

Apesar deste mecanismo ter sido implementado utilizando a linguagem Lua e um *binding* para o CORBA, a mesma filosofia pode ser aplicada com outras linguagens e *bindings* para outros modelos de componentes, desde que a linguagem possua características dinâmicas semelhantes às de Lua e o *binding* seja baseado na interface dinâmica do modelo de componentes. Estes aspectos permitem que a aplicação seja configurada sem declaração prévia de tipos de objetos que serão descobertos em tempo de execução.

Referências Bibliográficas

[AG95] R. Allen and D. Garlan. Formal Connectors. Technical Report CMU-CS-94-115, Carnegie-Mellon University, 1995.

[All97] R. Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie-Mellon University, 1997.

[AWB+94] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions using Composition Filters. *Object-Based Distributed Programming (ECOOP'93 Workshop), LNCs 791*, pages 152–184, 1994.

[Ber96] P. Bernstein. Middleware. *Communications of the ACM*, 39(2), February 1996.

[BF96] J. Bishop and R. Faria. Connectors in Configuration Programming Languages: are they necessary? In *Third International Conference on Configurable Distributed Systems*, pages 11–18, Annapolis, Maryland, May 6-8 1996.

[BR96] L. Bellissard and M. Riveill. Constructions des Applications Réparties. Ecole Placement Dynamique et Répartition de Charge, Juillet 1996.

[BR99] T. Batista and N. Rodriguez. Configuração Dinâmica de Aplicações CORBA. Technical Report 15/99, Pontifícia Universidade Católica do Rio de Janeiro – PUC-Rio, 1999.

[CCI99] R. Cerqueira, C. Cassino, and R. Ierusalimschy. Dynamic Component Gluing Across Different Componentware Systems. In *International Symposium on Distributed Objects and Applications (DOA'99)*, pages 362–371, Edinburgh, Scotland, September 1999. OMG, IEEE Press.

[CRI99] M. Catunda, N. Rodriguez, and R. Ierusalimschy. Dynamic Extension of CORBA Servers. In *Euro-Par'99 Parallel Processing*, pages 1369–1376, Toulouse, France, August 1999. Springer-Verlag. LNCS 1685.

[DR97] S. Ducasse and T. Richner. Executable Connectors: Towards Reusable Design Elements. In *Proceedings of ESEC/FSE'97, LNCS 1301*, pages 483–500, 1997.

[ICR98] R. Ierusalimschy, R. Cerqueira, and N. Rodriguez. Using Reflexivity to Interface with CORBA. In *Proceedings of the International Conference on Computer Languages*, pages 39–46, Chicago, IL, 1998.

[IFC96] R. Ierusalimschy, L. H. Figueiredo, and W. Celes. Lua - an extensible extension language. *Software: Practice and Experience*, 26(6), 1996.

[KM85] J. Kramer and J. Magee. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, 11(4):424–435, April 1985.

[MTK97] J. Magee, A. Tseng, and J. Kramer. Composing Distributed Objects in CORBA. In *Third International Symposium on Autonomous Decentralized Systems - ISADS 97*, pages 9–11, Berlin, Germany, April 9-11 1997.

[NGT92] O. Nierstrasz, S. Gibbs, and D. Tsichritzis. Component-Oriented Software Development. *Communications of the ACM*, 35(9):160–165, September 1992.

[OHE96] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide.* John Wiley & Sons., 1996.

[SDZ96] M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and Implementations for Architectural Connections. In *Third International Conference on Configurable Distributed Systems*, pages 2–10, Annapolis, Maryland, May 6-8 1996.

[Sha94] M. Shaw. Procedure Calls are the Assembly Language of Software Interconnection - Connectors Deserve First-Class Status. In *Proceedings of Workshop on Studies of Software Design.* LNCS, Springer-Verlag, 1994.

[SN99] J. Schneider and O. Nierstrasz. Components, scripts and glue. In John Hall Leonor Barroca and Patrick Hall, editors, *Software Architectures - Advances and Applications.* Springer 1999. available at http://www.iam.unibe.ch/cgi-bin/.

[SO98] C. Souza and M. Oliveira. ABACO: Um Ambiente de Desenvolvimento baseado em Objetos Distribuídos Configuráveis. In *Anais do XII Simpósio Brasileiro de Engenharia de Software*, pages 205–220, Maringá - PR, 1998.

[Sti94] J. Stikeleather. Why Distributed Computing is Inevitable. *Object Magazine*, pages 35–39, March 1994.