4
Avaliação de Flexibilidade

Como visto na seção 2.3, a flexibilidade de um software é uma característica relativa, pois depende do tipo de modificação sendo considerada. Além disso, a avaliação de flexibilidade pode ser feita de diferentes formas, seja levando em consideração apenas o número de modificações a serem introduzidas, seja levando em consideração os custos cognitivos necessários para projetar e aplicar a modificação.

Neste capítulo, é apresentada uma avaliação de flexibilidade do OiL que segue uma abordagem comum em avaliações de usabilidade, onde se destacam três elementos básicos: (a) um conjunto de casos de uso relevantes a serem analisados, que nesta avaliação são implementações de algumas funcionalidades do middleware; (b) um conjunto de aspectos a serem observados durante as análises, que nesta avaliação são as dimensões cognitivas do CDN apresentadas na seção 2.4; e (c) uma base de comparação com o sistema sendo avaliado, que pode ser uma base hipotética com características ideais, mas nesta avaliação é utilizada uma outra implementação real de middleware. O uso do CDN na análise de cada um dos casos de uso permite considerar o custo das modificações tanto em relação às alterações necessárias no software como também em relação aos aspectos cognitivos relacionados à idealização e compreensão dessas modificações.

A base de comparação escolhida é o Mico (Puder et al., 2006), um ORB desenvolvido em C++ apresentado na seção 4.1. O Mico apresenta alguns objetivos que são compartilhados com o OiL, como um foco maior em flexibilidade do que em desempenho. Acreditamos que essa similaridade de objetivos facilite identificar diferenças na implementação que estejam mais ligadas ao uso da linguagem de programação do que a decisões específicas de projeto devido a objetivos diferentes dos dois projetos. Nas demais seções deste capítulo são apresentadas avaliações de flexibilidade do OiL e do Mico com base nos casos de modificação considerados, que buscam representar a flexibilidade de três aspectos do middleware: (a) aspectos da implementação, representado pela escolha do protocolo de RMI suportado (seção 4.2); (b) modelo de programação oferecido, representado pelo suporte a chamadas síncronas e assíncronas (seção 4.3);

(c) conjunto de recursos oferecidos, representado tanto por funcionalidades modulares, como invocação de chamadas remotas, criação de *servants* e processamento de requisições, como também funcionalidades transversais como concorrência, interceptação de chamadas, entre outras (seção 4.4). Ao final deste capítulo, são feitas considerações gerais sobre a experiência de utilizar ambos os sistemas considerados na implementação desses casos de avaliação, em particular, identificando as características da implementação do OiL e de linguagens dinâmicas que contribuem para a flexibilidade da implementação.

Ao longo de todo este capítulo é feita menção às dimensões cognitivas do CDN apresentadas na seção 2.4. Nesses casos, o nome das dimensões cognitivas são escritas *em destaque* no texto para enfatizar seu significado especial. Todas as menções à implementação do Mico neste trabalho se referem a versão 2.3.12. A implementação do OiL analisada é a versão 0.5 alpha.

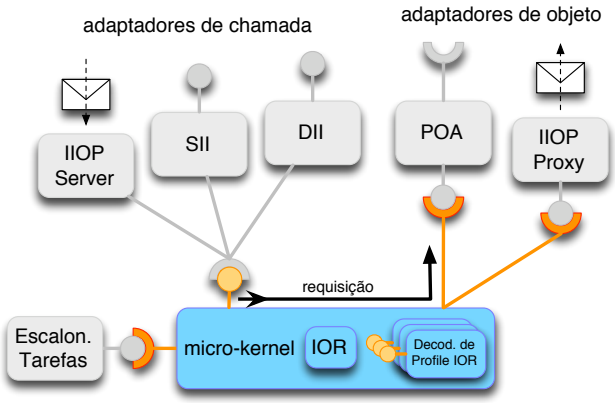
4.1 Referência de Comparação: Mico

O Mico (Puder et al., 2006) é uma implementação do padrão do CORBA inteiramente escrita na linguagem C++. Inicialmente, ele foi projetado como uma implementação mínima de CORBA com o propósito de se tornar uma ferramenta de ensino sobre implementações de middleware. Posteriormente, o projeto recebeu inúmeras contribuições de desenvolvedores da comunidade de software livre, tornando-se uma das implementações mais completas de CORBA. Por um lado, essas contribuições desviaram um pouco o projeto do seu propósito inicial, mas permitiram que o Mico fosse adotado como ferramenta de desenvolvimento efetiva em diferentes projetos. Como forma de conciliar essa evolução do projeto com o seu propósito inicial, os projetistas optaram por uma arquitetura baseada no conceito de *micro-kernel*, onde se mantém um núcleo mínimo que pode ser estendido com recursos avançados através de componentes mais complexos que são anexados ao sistema.

O *micro-kernel* do Mico é projetado com base em três tipos principais de componentes, que podem ser acoplados a ele: *adaptadores de chamada*, *adaptadores de objeto* e um *escalonador de tarefas*, conforme ilustrado na figura 4.1. Adaptadores de chamada são componentes que iniciam chamadas. Essas chamadas são recebidas pelo *micro-kernel*, que utiliza um dos adaptadores de objeto para fazer o despacho da chamada para o *servant* adequado. Adaptadores de chamada tipicamente monitoram canais de rede por requisições remotas (*e.g.* `IIOPServer`) ou são utilizados pela aplicação para fazer chamadas através da própria interface oferecida pelo ORB (*i.e.* DII, do inglês *Dynamic Invocation Interface*) ou de *proxies* (*i.e.* SII, do inglês *Static Invocation Interface*).

As implementações dos *proxies* são obtidas através de ferramentas de geração automática de código, que no caso é o compilador de IDL do Mico. De forma similar, adaptadores de objeto tipicamente enviam requisições para objetos remotos utilizando canais de rede (*e.g.* `IIOPProxy`) ou despacham as chamadas diretamente a *servants* fornecidos pela aplicação (*i.e.* POA, do inglês *Portable Object Adapter*). O escalonador de tarefas é um componente utilizado pelo *micro-kernel* do Mico na criação e gerenciamento de execução de tarefas como recebimento de chamadas através dos múltiplos adaptadores de chamada e despacho de chamadas através dos adaptadores de objeto. O escalonador cria *threads* para execução das tarefas e fica responsável por despachar eventos como disponibilidade de dados em conexões de rede e temporizadores.

Figura 4.1: Arquitetura do *micro-kernel* do Mico.



Basicamente, as tarefas do *micro-kernel* do Mico são oferecer interfaces de programação para a aplicação e intermediar a interação entre adaptadores de chamada e adaptadores de objeto, para realização das chamadas de operações. Para tanto, o *micro-kernel* do Mico implementa as interfaces de programação de CORBA necessárias para, por exemplo, registrar *servants*, criar referências de objeto, controlar o processamento de chamadas e interceptar chamadas.

O *micro-kernel* do Mico adota uma representação para referência de objetos usada tanto internamente quanto para a interação entre os adaptadores de objeto e de chamada. Essa representação das referências de objetos é baseada no IOR (*Interoperable Object Reference*) de CORBA, que é um modelo bastante genérico que consiste em um conjunto de informações arbitrárias codificadas em seqüências de *bytes*, denominadas *profiles*. A estrutura de cada *profile* é identificada por um valor numérico associado a ele. O uso de uma seqüência de *bytes* para armazenar as informações do *profile* exige que sejam registrados no *micro-kernel* decodificadores de *profiles*, que permitam decodificar essas informações e apresentá-las de forma mais facilmente manipulável aos adaptadores de chamada e objeto que as utilizarão. Esses decodificadores

de *profile* funcionam de forma similar aos `ReferenceProfilers` da camada do protocolo CORBA do OiL.

O padrão CORBA define suporte opcional para outros protocolos além do GIOP, que são denominados ESIOP (*Environment Specific Inter-ORB Protocol*). O modelo de múltiplos adaptadores de objeto no Mico é utilizado para permitir a implementação do suporte para ESIOP. Por isso, assim como o OiL, o Mico é projetado para permitir a introdução do suporte a outros protocolos de RMI, sob um mesmo modelo de programação.

4.2 Mudança do Protocolo de RMI

Uma parte importante de um middleware é o suporte a comunicação distribuída. Isso tipicamente envolve a utilização de um protocolo de comunicação, que no caso de ORBs, consiste do protocolo de RMI. Esse protocolo influencia algumas propriedades da comunicação, tais como os tipos de valores que podem ser transmitidos, o desempenho das chamadas, alguns aspectos semânticos das chamadas, etc. Portanto, em algumas situações é interessante alterar ou ajustar o protocolo com o intuito de adaptar o middleware para um uso diferente.

Nesta seção, consideraremos a implementação de suporte para dois tipos bem diferentes de protocolos de RMI. O primeiro é o protocolo GIOP do padrão CORBA. O GIOP é um protocolo genérico que oferece suporte para mapeamento para diferentes tecnologias de rede. É um protocolo binário de alto desempenho projetado para uso em aplicações com interfaces estáticas, ou seja, que não mudam após o início de sua execução. Além disso, o GIOP é um protocolo avançado, que inclui mensagens para realização de chamadas e também mensagens adicionais de controle de canais. Os valores codificados usando o GIOP não incluem todas as informações para sua decodificação. É necessário conhecer os tipos dos dados codificados para poder decodificá-los adequadamente e essas informações não estão disponíveis nas mensagens do GIOP. Isso resulta na necessidade de manter no middleware informações de tipo dos valores providos pela aplicação como forma de permitir a decodificação correta das mensagens GIOP. O GIOP também permitir a coexistência de múltiplos mapeamentos, por exemplo, permitir a utilização do IIOP, GIOP mapeado para TCP/IP, e o SSLIOP, GIOP mapeado sobre SSL (*Secure Socket Layer*).

O segundo protocolo é o LuDO, um protocolo textual para redes TCP/IP desenvolvido neste trabalho especificamente para ser simples e facilmente implementado, pois requer poucos recursos do middleware. O LuDO é similar a

outros protocolos textuais como XML-RPC, porém a codificação é feita usando código Lua ao invés de XML. O conjunto de tipos válidos do protocolo é um subconjunto dos tipos de Lua, em particular, tipos simples como números, valores lógicos e *strings*, além de tabelas, que são utilizadas como forma de representação de valores estruturados. Os valores codificados no LuDO incluem todas as informações para sua decodificação, inclusive os tipos dos valores.

Tanto o OiL como o Mico foram projetados inicialmente para darem suporte a diferentes protocolos, em particular o GIOP de CORBA. Esse protocolo atualmente ele ainda é o principal protocolo oferecido por ambos. Portanto, podemos considerar que o suporte ao GIOP seja implementado com cuidado nos dois casos. A análise da forma como essa implementação desse suporte se acomoda no modelo do Mico e do OiL nos permite identificar alguns aspectos da flexibilidade desses ORBs. Por outro lado, a implementação do LuDO foi originalmente motivada como uma forma de validar e investigar a facilidade da implementação de múltiplos protocolos no OiL. Por isso, a sua implementação evita sofisticações ou a busca por extrema eficiência. Ao invés disso, a sua implementação prima por simplicidade e facilidade de entendimento. A introdução do suporte ao LuDO no Mico também pode ser vista da mesma forma, uma vez que ela é utilizada neste trabalho especificamente para avaliar a facilidade de implementação de protocolos alternativos no Mico na forma de ESIOP. Por essas razões, podemos considerar a implementação do LuDO nos dois sistemas como uma modificação tipicamente exploratória.

As seções seguintes apresentam uma discussão sobre a facilidade de implementação de cada um desses protocolos no OiL e no Mico.

4.2.1 GIOP no OiL

O protocolo GIOP utiliza um modelo de chamadas muito diferente do modelo de chamadas de operações em Lua. Como resultado, a implementação desse protocolo do OiL impõe alguns desafios importantes que nos permite identificar algumas questões sobre sua flexibilidade, em particular aquelas relacionadas ao uso da linguagem Lua.

Interfaces e Tipos

Uma primeira necessidade da implementação do GIOP no OiL é manter informações sobre as interfaces dos objetos distribuídos. Essas informações são necessárias por duas razões: (a) para mapear os tipos de dados Lua para os tipos de dados do GIOP — por exemplo, valores numéricos em Lua são representados pelo tipo `number`, enquanto que no GIOP há sete tipos numéricos

diferentes; e (b) para recuperar as informações sobre os tipos dos valores codificados, pois os valores codificados no GIOP não contém informações sobre os seus tipos e essa informação é necessária para decodificá-los corretamente. Contudo, Lua não define o conceito de interfaces de objeto através do qual essas informações possam ser armazenadas. De certa forma, isso mostra uma pouca *proximidade de mapeamento* de Lua para implementação de um protocolo como o GIOP, que se baseia em tipos de dados diferentes dos de Lua e também no conceito de interfaces que não existe em Lua. Para solucionar esse problema, utilizamos as tabelas de Lua para criar a abstração de interfaces — o que indica uma baixa *resistência a abstrações* da linguagem. Tabelas são ideais para descrever dados arbitrários como informações de interface, pois são dinamicamente tipadas e podem ser inspecionadas usando mecanismos da própria linguagem. Outra abstração introduzida é o repositório de interfaces, que é usado para armazenar as descrições das interfaces e inspecioná-las.

Além de representar interfaces, também é necessário fornecer mecanismos para que a aplicação associe essas interfaces aos objetos, ou seja, possa indicar a interface de cada *servant* e *proxy*. Uma opção seria utilizar parâmetros adicionais nas operações `newservant` e `newproxy` dos componentes `ServantManager` e `ProxyManager` para informar a interface desses objetos na sua criação e implementar esses componentes de forma que eles guardem essa informação para que seja repassada para os componentes da camada do protocolo GIOP. Contudo, a introdução desse recurso pode se mostrar desnecessária ou indesejável para outros protocolos. Por exemplo, um protocolo como o LuDO não exige definição de interfaces dos objetos distribuídos. Um problema similar pode acontecer com outros protocolos que precisem associar outras informações aos objetos, como por exemplo, códigos de autenticação para autenticar acesso a um objeto específico.

A necessidade de prever as informações que um protocolo pode precisar associar aos objetos representa um *compromisso prematuro*. Felizmente, a tipagem dinâmica de Lua, e mais especificamente a estrutura dinâmica oferecida pelas tabelas, fornece um *mecanismos de provisão* para esse problema. Tanto as referências de objeto como os próprios objetos que implementam os *servants* são tabelas e portanto podem ser estendidos com informações adicionais, como a indicação da interface IDL do objeto. Por exemplo, a criação de um *servant* pode ser feita como ilustrado na figura 4.2, onde a informação da sua interface é associada ao objeto que o implementa através de um campo adicional denominado `__type`. Inclusive, esse campo pode ser introduzido através de meta-tabelas.

A utilização do campo `__type` não interfere no componente

Figura 4.2: Criação de *servant* associando informação da sua interface.

```
1 local impl = {}
2 function impl:say()
3   print("Hello, World!")
4 end
5
6 impl.__type = "Hello"
7 ServantManager.servants:register(impl)
```

`ServantManager` da camada de apresentação, que ignora essa informação. No componente `RequestListener`, a implementação do *servant* é recuperada através do componente `ServantManager` e sua interface pode ser consultada através do campo `__type`. A facilidade de associar novas informações aos objetos funciona como um *mecanismo de provisão* que evita a definição de *compromissos prematuros*, como a determinação dos parâmetros necessários para criação de um novo *servant* no componente `ServantManager` sem conhecer o protocolo de RMI a ser utilizado na montagem do ORB. Por outro lado, a necessidade de definir o campo `__type` ou a sua utilização na implementação do ORB não fica clara, caracterizando-se como uma *dependência oculta* e também resulta em maior *propensão a erros*, pois o campo pode ser usado para outra função ou removido do objeto erroneamente. Lua oferece outros recursos que também podem ser utilizados como *mecanismos de provisão* equivalentes nesse caso, como uso de listas de parâmetros variáveis (*varargs*) ou uma tabela adicional passada como parâmetro contendo campos indicando as informações requeridas pelo protocolo.

Como o suporte ao GIOP se baseia em definições de interfaces, podemos tirar proveito disso na implementação dos *proxies*. Por exemplo, quando um *proxy* é indexado, é interessante só permitir a indexação de operações definidas na sua interface. A desvantagem desse mecanismo é que a implementação do componente `ProxyManager` estaria condicionada à definição de interfaces, o que impediria sua utilização com protocolos que não se baseiam nesse conceito, como é o caso do LuDO. Novamente, essa decisão de implementar o suporte a interfaces nos componentes da camada de apresentação é um *compromisso prematuro* que pode se mostrar inadequado ao uso com alguns protocolos de RMI. Um *mecanismo de provisão* para isso é a possibilidade de oferecer implementações alternativas desses componentes, como é feito pela extensão da camada de apresentação para suporte de interfaces descrita na seção 3.3.3. Dessa forma, a escolha da implementação a ser utilizada pode ser feita na montagem no ORB, quando a necessidade do recurso pode ser corretamente determinada de acordo com o protocolo de RMI escolhido.

Múltiplos Mapeamentos

A implementação do GIOP no OiL também envolve introduzir o suporte a múltiplos mapeamentos para mecanismos de transporte diferentes, além de permitir a utilização simultânea deles. Isso requer que o ORB associe informação sobre os múltiplos protocolos a algumas estruturas de dados internas. Por exemplo, uma referência de objeto deve poder acomodar as informações de localização de todos os diferentes mapeamentos do GIOP, de forma similar ao IOR de CORBA, que contém um conjunto de *profiles* codificados de maneiras diferentes, de acordo com cada mapeamento do GIOP.

No OiL, a representação de uma referência de objeto é definida pelo componente da camada do protocolo `ObjectReferrer`. Isso permite que o protocolo implemente a sua representação de referência de objeto da forma mais adequada. Por isso, na implementação do GIOP, o componente `ObjectReferrer` cria referências de objeto que possam acomodar as informações do IOR de CORBA. O IOR é composto por um conjunto de *profiles*, sendo que cada um é codificado de uma maneira específica definida por um mapeamento do GIOP. A decodificação de um IOR é feita pelo componente `ObjectReferrer`, que utiliza componentes adicionais, denominados `ReferenceProfiler`, para decodificar os diferentes tipos de *profile*. As informações extraídas do *profile* são armazenadas em uma tabela. O uso de uma tabela neste caso é particularmente útil, pois não é possível prever a estrutura das informações desses *profiles*. Tabelas permitem armazenar informações com estrutura arbitrária, de forma que possam ser inspecionadas por outros componentes. Essa funcionalidade se caracteriza como um *mecanismo de provisão* neste caso.

Além das informações de localização nas referências de objeto, também é necessário implementar suporte para criação de conexões usando essas informações. Isso é feito pelos componentes `ClientChannels` e `ServerChannels`. Como há diferentes tipos de canais que podem ser criados usando mapeamentos distintos, é necessário manipular múltiplas instâncias desses componentes simultaneamente. Por isso, os componentes `OperationRequester` e `RequestListener`, da camada do protocolo, apresentam receptáculos que permitem múltiplas conexões com os diferentes `ClientChannels` e `ServerChannels`. Cada uma dessas conexões é identificada pelo *tag* do mapeamento GIOP a que o componente conectado oferece suporte. Os receptáculos múltiplos permitem inspecionar as conexões estabelecidas. Isso facilita que os componentes `OperationRequester` e `RequestListener` possam identificar quais os mapeamentos GIOP disponíveis para comunicação com outros ORBs.

Esse modelo baseado em componentes permite uma baixa *viscosidade* na introdução do suporte a um novo mapeamento do GIOP, pois exige apenas

a introdução de componentes que implementem os detalhes específicos do mapeamento, como a forma de codificação do *profile* (`ReferenceProfiler`) e a forma de criação de canais a partir das informações específicas do *profile* (`ClientChannels` e `ServerChannels`). Também é interessante elaborar um *script* de montagem que facilite a aplicação a adicionar esses componentes a uma montagem de ORB quando apropriado. Inclusive, como essa adição de componentes pode ser feita tanto na criação do ORB quanto durante seu uso, ela evita *compromissos prematuros* no desenvolvimento do ORB, já que não é possível determinar previamente as necessidades reais da aplicação. Por outro lado, tanto o uso de componentes como dos *scripts* de montagem são novas abstrações que devem ser absorvidas pelo programador para poder introduzir o suporte a um novo mapeamento de GIOP no OiL. Em particular, essa arquitetura de componentes que permite acoplar novos componentes que implementam um mapeamento específico do GIOP também é um conceito obrigatório. Portanto, apesar de diminuir a *viscosidade*, a arquitetura de componentes do ORB, nesse caso, aumenta a *barreira de abstração*.

Ao contrário das referências de objeto, o OiL impõe que as requisições sejam representadas como tabelas com uma estrutura pré-determinada, como apresentado na seção 3.3.5. Essa estrutura permite que a camada de apresentação possa extrair os resultados de uma chamada. Por outro lado, essas requisições são tabelas de Lua, portanto podem armazenar informações arbitrárias. Isso permite que os componentes da implementação do GIOP possam associar informações adicionais às requisições através da introdução de novos campos na tabela da requisição, como por exemplo, o canal de comunicação a ser utilizado para obter a resposta ou as informações de tipo necessárias para decodificar a resposta.

4.2.2 GIOP no Mico

Como visto na seção 4.1, o Mico é projetado com base no padrão CORBA. Por essa razão, a implementação do Mico apresenta diversos compromissos assumidos especificamente para permitir a implementação de protocolos similares ao GIOP, mas que podem se mostrar inadequados para outros protocolos. Nesta seção, nos concentraremos em identificar alguns desses *compromissos prematuros* e como eles se adequam à implementação do GIOP e seus vários mapeamentos.

Interfaces e Tipos

O padrão CORBA foi fortemente influenciado por sistemas de RPC em C (Schmidt e Vinoski, 2000). Em particular, muitos dos tipos de dados manipulados no GIOP têm tipos equivalentes em C, e consequentemente C++. Além disso, objetos em C++ obrigatoriamente definem uma interface através de uma classe de forma similar às interfaces IDL implementadas por objetos CORBA. Isso torna C++ uma linguagem com uma *proximidade de mapeamento* com GIOP maior que Lua.

Por outro lado, C++ não oferece suporte para inspecionar as interfaces definidas no programa, portanto não há como inferir uma interface IDL correspondente. Ao invés disso, o Mico define que a aplicação deve especificar as interfaces dos objetos CORBA em IDL, que são utilizadas na geração automática de código especializado responsável por fornecer ao Mico as informações sobre tipos de dados e interfaces de objeto, assim como implementar a interface de programação oferecida para a aplicação. Essa abordagem baseada em geração automática de código faz com que o gerador de código, denominado *compilador de IDL*, seja parte integrante da implementação do ORB.

Estruturas de Dados

Um desafio no projeto do *micro-kernel* do Mico é definir as estruturas de dados que serão utilizadas pelos componentes acoplados a ele, de forma a atender as necessidades desses componentes. Duas dessas estruturas são particularmente importantes: as referências de objeto e as requisições de operação. As referências de objeto são utilizadas pelos adaptadores de objeto para identificar o destino das chamadas encaminhadas a eles, e são fornecidas pelos adaptadores de chamada de forma que o *micro-kernel* possa determinar um adaptador de objeto adequado para tratá-la. As requisições de operação são utilizadas para representar chamadas simultâneas sendo tratadas pelo *micro-kernel* e são criadas pelos adaptadores de chamada e depois processadas por um adaptador de objeto.

Cada protocolo de RMI pode necessitar de um conjunto de informações diferentes a ser associado a essas estruturas. Por exemplo, o GIOP faz multiplexação de diferentes requisições em uma mesma conexão, portanto é necessário manter a contagem dos identificadores das requisições pendentes naquela conexão, de forma a evitar duas requisições com o mesmo identificador. Outro exemplo são as informações de localização específicas de cada protocolo que são armazenadas na referência de objeto, como é o caso dos *profiles* IOR definidos por CORBA.

Na implementação do Mico há duas alternativas comumente utilizadas para associar informações específicas dos adaptadores a uma estrutura definida pelo *micro-kernel*: (a) prever quais as informações que deverão ser mantidas e estender a estrutura delas de forma que ela seja capaz de conter todas essas informações previstas (*compromisso prematuro*); (b) estender a estrutura de forma que ela apresente uma parte dinâmica que permita armazenar uma estrutura arbitrária de dados, assim como prover mecanismos de introspecção que permitam identificar a estrutura do dado (pouca *proximidade de mapeamento*). Um exemplo disso seria utilizar uma estrutura em C++ que contém um ponteiro genérico (`void*`) e um campo numérico indicando qual o tipo do objeto referenciado pelo ponteiro genérico. Uma forma alternativa seria utilizar uma classe abstrata cuja interface permite identificar a classe real do objeto e realizar uma conversão explícita (*cast*) para essa classe. Dessa forma, a interface da classe real do objeto permitiria o acesso a informações associadas ao objeto que são informadas dinamicamente na criação do objeto.

As referências de objeto do Mico seguem o modelo do IOR de CORBA, que, como discutido anteriormente, é uma estrutura dinâmica capaz de armazenar informações arbitrárias codificadas em uma seqüência de *bytes*. De forma similar, a referência de objeto no Mico é implementada pela classe `IOR`, que contém uma seqüência de objetos `IORProfile` representando os *profiles* do IOR. Cada protocolo de RMI deve implementar uma sub-classe de `IORProfileDecoder`, que deve ser instanciada e essa instância deve ser cadastrada através de métodos de classe (`static`) da própria classe `IORProfileDecoder`, de forma que possa ser utilizada pelo *micro-kernel* para criar os objetos `IORProfile` daquele protocolo. Esse modelo permite que cada protocolo defina a forma que as informações são extraídas do seu *profile* e como elas ficam armazenadas na referência de objeto, de forma similar ao que é feito pelo componente `ObjectReferrer` do OiL.

As interfaces das classes `IOR` e `IORProfile` são definidas de acordo com as prováveis necessidades dos protocolos RMI, conforme especificado pelo padrão CORBA. Isso representa um *compromisso prematuro*, pois essas interfaces podem se mostrar desnecessárias ou insuficientes para alguns protocolos. Para ilustrar isso, considere a definição parcial da classe `IORProfile` mostrada na figura 4.3. A operação `components` (linha 16) é utilizada para obter componentes de um tipo de *profile* estruturado definido em CORBA, denominado *multiple-component profile*. Esse tipo de *profile* facilita o compartilhamento de informações do *profile* entre diferentes protocolos. Por um lado, esse suporte é importante para facilitar a implementação de protocolos que precisem armazenar informações dinamicamente estruturadas para interagir com outros

protocolos. Contudo, em muitos protocolos, inclusive alguns mapeamentos do próprio GIOP de CORBA, esse é um recurso desnecessário e que representa uma abstração que o implementador de protocolos no Mico deve aprender (*barreira de abstração*). Apesar da classe `IOPProfile` definir uma implementação *default* dessa operação, de forma geral, o programador deve conhecer o seu propósito e avaliar a necessidade de redefini-la.

Figura 4.3: Interface da classe `IORProfile` do Mico.

```
1  class IORProfile {
2    /* ... */
3    static IORProfile *decode (DataDecoder &);
4    static IORProfile *decode_body (DataDecoder &, ProfileId, ULong len);
5    static void register_decoder (IORProfileDecoder *);
6    static void unregister_decoder (IORProfileDecoder *);
7
8    virtual void encode (DataEncoder &) const              = 0;
9    virtual const Address *addr () const                   = 0;
10   virtual ProfileId id () const                          = 0;
11   virtual ProfileId encode_id () const                   = 0;
12   virtual void objectkey (Octet *, Long length)          = 0;
13   virtual const Octet *objectkey (Long &length) const    = 0;
14   virtual Boolean reachable ()                           = 0;
15   virtual void print (std::ostream &) const              = 0;
16   virtual CORBA::MultiComponent *components () { return 0; };
17   virtual void prepare_mobile () {}
18   /* ... */
19 };
```

De certa forma, esse modelo de referências do Mico é uma estrutura dinamicamente tipada, pois permite acomodar objetos de tipos diferentes e ainda identificar esses tipos através da operação `id` (linha 10). Com base nesse mecanismo, é possível fazer conversões explícitas de tipo (*casting*) com alguma segurança e chamar operações da interface específica do objeto quando necessário. Isso mostra que, apesar de C++ não ser uma linguagem dinâmica, ela oferece mecanismos de abstração que permitem implementar verificações dinâmicas de tipos. Contudo, a principal desvantagem desse uso é a *propensão a erros* introduzida. Para ilustrar isso, considere o trecho de código extraído da implementação do Mico mostrado na figura 4.4. A operação `id` (linha 3) é utilizada para identificar se o `profile` é um `IIOPProfile` (sub-classe específica de `IORProfile`). Contudo, esse mecanismo de inspeção de tipo é implementado pelo próprio programador e não é verificado pela tipagem estática de C++, o que exige maior atenção do programador. No trecho da figura 4.4, essa preocupação do programador é representada pelo uso do operador `dynamic_cast` (linha 4), que permite verificar dinamicamente o tipo real de um objeto usando o suporte especial de C++ para isso, chamado de RTTI (*Run-Time Typing Information*). A chamada a `assert` é utilizada para identificar mais facilmente um possível erro de tipos introduzido pelo programador. Contudo, o uso de RTTI é custoso e geralmente evitado nas

conversões de tipo no código do Mico. Essa prática torna eventuais erros nos mecanismos de inspeção de tipos implementados pelo programador, como a operação `id` do `IORProfile`, difíceis de identificar.

Figura 4.4: Trecho que mostra uso de tipagem dinâmica no Mico.

```
1 for (CORBA::ULong i = 0; (tprof = ior_template_->get_profile(i)) != NULL; i++) {
2   bool match = false;
3   if (tprof->id() == CORBA::IORProfile::TAG_INTERNET_IOP) {
4     MICO::IIOPProfile* iiop_prof = dynamic_cast<MICO::IIOPProfile*>(tprof);
5     assert(iiop_prof != NULL);
6     for (CORBA::ULong j = 0; j < creds->length(); j++) {
```

Outra estrutura de dados importante na implementação do Mico são as requisições de operação. Sempre que a primeira requisição é despachada ao adaptador de objeto do GIOP, uma nova conexão é estabelecida utilizando o *profile* apropriado da referência de objeto. Entretanto, a obtenção do resultado da requisição é feita assincronamente. Por isso, o adaptador de objeto precisa associar à requisição a conexão a ser utilizada para receber a resposta. Outras informações podem ser necessárias para obtenção da resposta dependendo do protocolo específico sendo utilizado. No caso do GIOP, é necessário um número de identificação da requisição naquele canal.

Para permitir que os protocolos de RMI possam associar informações arbitrárias às requisições de operação no Mico, elas apresentam um *mecanismo de provisão* através dos campos `request_hint` e `invoke_hint` que são ponteiros genéricos (`void*`). Esses campos são destinados a uso exclusivo dos adaptadores de objeto (`request_hint`) e adaptadores de chamada (`invoke_hint`) que manipulam essa requisição. Como os campos são ponteiros genéricos, eles podem armazenar referências para qualquer objeto. Idealmente, os objetos armazenados nesses campos são definidos pelo próprio adaptador e armazenam as informações específicas do protocolo.

O *micro-kernel* do Mico é construído de forma a garantir que apenas as requisições criadas por um adaptador de chamada sejam devolvidas a ele, pois não faz sentido dar a resposta de uma chamada por meio de outro adaptador senão aquele que originou a chamada. Da mesma forma, quando uma requisição é despachada através de um adaptador de objeto, essa requisição não é mais tratada por outro adaptador de objeto. Por isso, geralmente não é necessário prover um mecanismo de inspeção que permita identificar o tipo do valor armazenado nos campos `request_hint` e `invoke_hint`, pois o tipo é indiretamente identificado pelo componente que manipula a requisição. Por exemplo, todas as requisições criadas pelo adaptador de chamada `IIOPProxy` terão no campo `invoke_hint` valores da classe `IIOPProxyInvokeRec`, que é definida por esse adaptador.

Podemos considerar as referências de objeto no projeto do Mico também como uma forma de estrutura dinamicamente tipada. Isso se justifica porque esses objetos permitem valores de diferentes tipos e um mecanismo para identificar esses tipos de forma indireta, através do adaptador que manipula a estrutura.

Múltiplos Mapeamentos

A implementação dos múltiplos mapeamentos do GIOP no Mico pode ser feita através de diferentes adaptadores de objeto. Contudo, isso implicaria em um certo grau de redundância na implementação desses adaptadores, pois eles seriam similares em muitos aspectos. Para evitar isso, os adaptadores do Mico que implementam o GIOP foram projetados para suportar diferentes mecanismos de transporte, de forma similar ao suporte a múltiplos mapeamentos GIOP oferecido pela camada do protocolo CORBA do OiL.

O modelo de IOR adotado no *micro-kernel* do Mico permite utilizar diferentes *profiles* para armazenar as informações de localização específicas de cada mapeamento do GIOP. Por isso, podemos dizer que o modelo de referências imposto pelo *micro-kernel* é adequado à implementação do GIOP (*proximidade de mapeamento*).

A implementação da decodificação de cada *profile* é feita através da especialização de quatro classes definidas pelo Mico: `IORProfileDecoder`, `IORProfile`, `AddressParser` e `Address`. A sub-classe de `IORProfileDecoder` permite decodificar a seqüência de *bytes* do *profile* no IOR e criar um objeto da sub-classe `IORProfile` com essas informações. Como discutido anteriormente, a classe `IORProfile` oferece uma interface pré-definida pelo *micro-kernel* para extrair informações. Um exemplo disso é o método `address` (linha 9 da figura 4.3), que permite obter uma identificação do endereço especificado pelo *profile*. Essa identificação é um objeto da sub-classe `Address`, que deve ser especializado para permitir comparação de igualdade desses objetos com base no seu conteúdo. Isso é necessário para permitir o *micro-kernel* identificar se dois *profiles* se referem ao mesmo destino e gerenciar automaticamente a reutilização das conexões. A sub-classe de `AddressDecoder` é utilizada para criar identificadores de endereço a partir de referências textuais como `corbaloc` de CORBA. A funcionalidade oferecida por essas quatro classes é equivalente à funcionalidade do `ReferenceProfiler` do OiL. Os objetos da sub-classe de `Address` oferecem as operações `make_transport` e `make_transport_server` para criação de canais de comunicação utilizando o mecanismo de transporte específico do mapeamento GIOP. Esses canais são implementadas por sub-classes de `Transport` e `TransportServer`, que são comparáveis aos canais

criados pelos componentes `ClientChannels` e `ServerChannels` do OiL.

Uma das desvantagens dessa abordagem baseada em classes é a dificuldade de reutilizá-las ou adaptar sua implementação para interagir com implementações alternativas das classes relacionadas. Por exemplo, cada sub-classe de `Address` cria canais de comunicação utilizando um mecanismo de transporte implementado por sub-classes de `Transport` e `TransportServer`. Essa associação fica explícita no código da sub-classe de `Address`. Portanto, para oferecer suporte a outro mecanismo de transporte, é necessário criar outra sub-classe de `Address`, ou modificar sua implementação para usar outras sub-classes de `Transport` e `TrasnportServer`. No Mico, para evitar a criação de duas sub-classes de `Address` similares, a classe `InetAddress` implementa tanto mecanismo de transporte TCP como UDP, que pode ser selecionado através do atributo do objeto `_family`, como é ilustrado na figura 4.5.

Figura 4.5: Definição explícita de dependências entre classes no Mico.

```
1  CORBA::Transport * MICO::InetAddress::make_transport () const {
2    CORBA::Transport *ret;
3    switch (_family) {
4      case STREAM: ret = new TCPTransport;
5                   break;
6      case DGRAM:  ret = new UDPTransport;
7                   break;
8      default:     assert (0);
9                   return 0;
10   }
11   ret->open();
12   return ret;
13 }
14 CORBA::TransportServer * MICO::InetAddress::make_transport_server () const {
15   switch (_family) {
16     case STREAM: return new TCPTransportServer;
17     case DGRAM:  return new UDPTransportServer;
18     default:     assert (0);
19                  return 0;
20   }
21 }
```

A principal desvantagem dessa forma de representação de dependências é a dificuldade de adicionar ou remover o suporte a novos transportes, pois é necessário modificar a classe `InetAddress` para incluir ou remover esse suporte, como é feito para TCP e UDP, ou prover uma nova implementação da classe. Isso caracteriza uma *viscosidade* maior do que em um modelo em que essas dependências ficam externas e é possível modificá-las sem alterar a implementação da classe, como é o caso do uso de receptáculos em componentes. Além disso, isso representa um *compromisso prematuro* assumido na implementação do Mico, pois é necessário escolher os mecanismos de transporte oferecidos na classe `InetAddress`, quando não é possível determinar quais serão efetivamente necessários pela aplicação. Entretanto, essa abordagem não é uma imposição da linguagem, inclusive alguns padrões de projeto

são utilizados em C++ para evitar esse tipo de *compromisso prematuro* de forma similar ao que é feito com componentes, como por exemplo o padrão *Abstract Factory* (Gamma et al., 1994) e *Component Configurator* (Schmidt et al., 2000).

Por outro lado, uma vantagem importante do uso de classes com dependências explícitas em relação ao uso de componentes é que essa dependência é utilizada pelo compilador para verificar se a interface dessas classes casam com a interface utilizada pela sub-classe de `Address`. No modelo de componentes, essa dependência só fica explícita quando se analisa a montagem efetiva dos componentes, que geralmente só é definida pela aplicação que usa o middleware. Portanto, além de diminuir as *dependências ocultas*, a abordagem baseada em classes utilizada no Mico também é menos *propensa a erros* do que o modelo de componentes do OiL, que permite criar montagens inválidas como a criação de um `ProxyManager` sem um `OperationInvoker`.

4.2.3 LuDO no OiL

Como visto anteriormente, o LuDO foi projetado principalmente para avaliar a possibilidade de implementação de um protocolo alternativo no OiL. Comparado ao GIOP, o LuDO é um protocolo de RMI mais simples, que utiliza os recursos de descrição de dados de Lua como formato de codificação de valores. O fato do LuDO ser baseado em Lua facilita a sua implementação no OiL (*proximidade de mapeamento*), pois a linguagem Lua já oferece suporte para decodificação (*parsing*) e manipulação dos dados transmitidos. Além disso, outras características do LuDO facilitam sua implementação em Lua: (a) os tipos de dados suportados no LuDO são um subconjunto dos tipos de Lua. (b) os dados codificados no LuDO são tipados, o que permite decodificar esses valores sem necessidade de informação auxiliar sobre as interfaces dos *servants*. O primeiro item permite um mapeamento direto dos tipos de Lua para os tipos aceitos pelo LuDO. O segundo item, aliado ao suporte à reflexão computacional de Lua, permite implementar o suporte LuDO sem necessidade de informações de interface dos objetos. Isso é possível porque a implementação do LuDO pode identificar os tipos de cada dado a ser codificado utilizando reflexão computacional. No momento da decodificação, as informações de tipo contidas no próprio dado codificado são utilizadas para identificar qual o tipo de Lua correspondente.

Essa *proximidade* do LuDO com Lua permite uma implementação simples no OiL. Em parte, isso é propiciado também pelo fato do OiL impor o uso de poucas abstrações na implementação do protocolo — como descrito na

seção 3.3.5 — e apresentar alguns *mecanismos de provisão* que facilitam ajustar o OiL às necessidades específicas do protocolo, como discutidas na seção 4.2.1. Em particular, o OiL não impõe uma representação para referências do protocolo, pois elas são vistas como objetos opacos, cuja estrutura é definida inteiramente pelo protocolo. Por essa razão, as referências de objeto criadas pelo `ObjectReferrer` da implementação do LuDO consistem simplesmente de uma trinca de valores indicando o endereço da máquina e porta do processo onde o objeto remoto reside, além da sua chave de objeto. No LuDO, essas informações são suficientes para que o `OperationRequester` possa realizar chamadas a objetos remotos. O *mecanismo de provisão* oferecido pelo uso de tabelas para representar requisições também foi utilizado na implementação do LuDO para associar a conexão a ser utilizada na obtenção da resposta da requisição.

4.2.4 LuDO no Mico

Como o LuDO foi originalmente projetado para ser facilmente implementado em Lua, ele se baseia intensamente em recursos específicos de Lua, que não estão necessariamente disponíveis em C++, em particular, o suporte à interpretação de código Lua necessária para decodificação de mensagens LuDO. Como solução para esse problema, utilizamos o próprio interpretador de Lua na decodificação de mensagens LuDO. Isto é possível porque o interpretador de Lua é uma biblioteca C com uma API de fácil utilização que permite acesso aos valores Lua gerados pelo interpretador. Adicionalmente, é necessário converter os valores de C++ manipulados nas chamadas para os tipos do LuDO e vice-versa. Grande parte dos tipos de dados de C++ podem ser representados no LuDO, salvo algumas restrições, que não são impeditivas[1]. As informações de interface oferecidas pelo código gerado pelo compilador de IDL são utilizadas para realizar as conversões dos valores de C++ para o LuDO e vice-versa.

Um outro problema na implementação do LuDO foi ajustá-lo ao modelo de representação de referências de objeto baseado no IOR de CORBA, que é imposto pelo *micro-kernel* do Mico. Idealmente, a implementação do LuDO deve ser simples, sem a necessidade de tratar complexidades relacionadas à manutenção de múltiplos protocolos. Portanto, se o nosso objetivo é criar uma versão do Mico com suporte apenas ao protocolo LuDO, uma estrutura sofisticada como a de referências de objeto baseadas no IOR de CORBA é desnecessária e implica em uma *barreira de abstração* maior. Muitos outros

[1] O tipo numérico do LuDO é o mesmo de Lua, que é compatível com o tipo `double` de C++, portanto todos os valores numéricos em C++ que podem ser convertidos para `double`, sem perda de precisão, não apresentam problemas no LuDO.

compromissos assumidos na implementação do Mico se mostram inadequados na implementação de uma versão apenas com suporte ao protocolo LuDO.

Figura 4.6: Uso de IOR de CORBA contendo referências LuDO no Mico

```
1 CORBA::Boolean MICO::LuDOProxy::has_object(CORBA::Object_ptr obj) {
2   CORBA::IOR *tmpl = _orb->ior_template();
3   /*
4    * we have every object whose IOR has the LUDO_IOR_TAG
5    * and the profile doesn't point to this process (to prevent
6    * infinite loops when trying to talk to nonexisting objects
7    * in existing servers).
8    */
9   CORBA::IORProfile *p1, *p2;
10  p1 = obj->_ior_fwd()->profile(LUDO_IOR_TAG);
11  if(p1) {
12    p2 = tmpl->profile(LUDO_IOR_TAG);
13    return !p2 || !(*p1 == *p2);
14  }
15  return false;
16 }
```

Para ilustrar isso, considere a implementação da operação `has_object` do adaptador de objeto que permite realizar chamadas através do LuDO, que é implementado pela classe `LuDOProxy`, como mostrado na figura 4.6. Essa operação é utilizada pelo *micro-kernel* para identificar se as chamadas para a referência indicada pelo parâmetro `obj` devem ser despachadas através daquele adaptador de objeto. Na implementação do LuDO no Mico, só há dois adaptadores de objeto, um para despachar chamadas aos *servants* registrados localmente (POA) e outra para despachar chamadas a objetos remotos através do protocolo LuDO. Portanto, essa operação basicamente deve identificar se a referência é local ou não. Para auxiliar nessa tarefa, o *micro-kernel* coleciona as informações de localização do próprio ORB, que são definidas pelos adaptadores de chamada que recebem requisições remotas, de forma similar aos pontos de acesso do OiL. Essas informações de localização são disponibilizadas pelo *micro-kernel* na forma de um IOR (linha 2), onde cada *profile* contém a informação de localização de um adaptador de chamada que recebe requisições remotas. Como a implementação do LuDO só apresenta um adaptador desse tipo, o suporte a múltiplos *profiles* é desnecessário. Além disso, essa estrutura de múltiplos *profiles* também dificulta a comparação das referências, pois é necessário encontrar o *profile* correspondente ao protocolo LuDO em ambas as referências, através da operação `profile` da classe `IOR` (linhas 12 e 10), para então poder compará-las (linha 13).

A operação `_ior_fwd` (linha 10) é outro compromisso inadequado para o LuDO. Essa operação é definida primordialmente para permitir obter uma referência de objeto alternativa que esteja associada à referência de objeto original, que pode ser obtida através da operação `_ior`. Isso é necessário para implementar o mecanismo de desvio de chamadas do protocolo GIOP, chamado

de *location forward*, onde é possível que o objeto remoto mude sua localização. Esse recurso não é implementado pelo LuDO, portanto a possibilidade de definir referências alternativas se torna um recurso desnecessário nesse caso e consequentemente uma *barreira de abstração* desnecessária.

Um agravante desses compromissos inadequados ao LuDO é a dificuldade de corrigí-los devido à alta *viscosidade* do *micro-kernel* do Mico. Isso se justifica porque a alteração desses compromissos implica em substituir todo o *micro-kernel* por uma implementação alternativa que utilizasse um modelo diferente. Contudo, o *micro-kernel* é implementado como uma estrutura monolítica que implementa uma quantidade significativa de funcionalidades que são comparáveis às fucionalidades da camada de apresentação do OiL. Portanto, sua substituição ou modificação representa um custo significativo. Por essa razão, a implementação do suporte LuDO no Mico foi feita adotando o modelo definido pelo *micro-kernel* original, adequando o protocolo ao modelo imposto. Por exemplo, o formato de referências textuais do LuDO não pôde ser implementado no Mico, pois as referências textuais são geradas pelo *micro-kernel* seguindo o modelo de *stringfied* IOR de CORBA.

4.3 Mudança de Modelo de Programação

O modelo de programação oferecido pelo middleware tem influência direta na estrutura da aplicação que o utiliza. Em alguns casos, um modelo de programação inadequado pode dificultar a utilização do middleware em uma aplicação. Por essa razão, é interessante permitir variações no modelo de programação como forma de ampliar a aplicabilidade do middleware. O padrão CORBA, por exemplo, define basicamente três diferentes formas de realização de chamadas de método de objetos remotos:

chamadas síncronas Após o envio da requisição, a execução do cliente é suspensa até que o resultado seja recebido.

chamadas assíncronas com *polling* Após o envio da requisição, um objeto de *polling* é devolvido e a execução prossegue. O objeto de *polling* pode ser utilizado para verificar se os resultados estão disponíveis, obter efetivamente os resultados ou suspender a execução até que os resultados estejam disponíveis.

chamadas assíncronas com *callback* Após o envio da requisição, a execução prossegue, mas a aplicação fornece um objeto de *callback* que será executado quando os resultados estiverem disponíveis.

Há uma quarta forma denominada chamada síncrona retardada, porém restrita à interface de invocação dinâmica (DII), e sua semântica pode ser simulada com chamadas assíncronas com *polling*. Neste trabalho optamos por avaliar apenas a implementação do suporte a chamadas síncronas e assíncronas com *polling* por serem modelos bastante utilizados na prática e fáceis de ser implementados. Chamadas assíncronas com *callback* são particularmente difíceis de implementar sem suporte para concorrência, pois pode ser necessário interromper a execução da aplicação para executar o objeto de *callback*.

Tanto no OiL como no Mico, as chamadas são feitas através de objetos *proxies* que representam objetos remotos. Portanto, as variações na forma com que as chamadas são realizadas implicam na mudança da implementação dos *proxies*. Contudo, a interface dos *proxies* é definida de acordo com as interfaces dos *servants* que eles representam, que por sua vez são definidas pela aplicação. Consequentemente, a implementação dos *proxies* é diretamente influenciada pela aplicação. De forma geral, isso faz com que o middleware deva utilizar algum mecanismo de meta-programação que permita que ele possa se adaptar automaticamente às interfaces definidas pela aplicação.

Meta-programação se refere ao desenvolvimento de meta-programas, que são programas que geram ou manipulam programas. Neste trabalho, vamos considerar três formas de meta-programação:

Geração de código-fonte é uma forma de meta-programação que consiste em desenvolver programas que geram o código-fonte de um programa. Esse código-fonte é então compilado e incorporado ao próprio programa, que pode ser feito tanto durante o desenvolvimento do programa através de compiladores estáticos, ou mesmo durante a execução do programa através do suporte de interpretação oferecido pelas linguagens dinâmicas. Exemplos de geradores de código são compiladores de compilador, que com base em uma definição de gramática formal geram um compilador daquela linguagem.

Pré-processamento consiste em utilizar uma meta-linguagem que pode manipular o código-fonte de um programa que é compilado ou interpretado, podendo alterar e gerar trechos de código. Um exemplo disso é o pré-processador de C.

Reflexão computacional é quando o próprio programa age como seu meta-programa, manipulando sua estrutura reificada, podendo alterar seu próprio comportamento ou mesmo inserir novos recursos.

Nas próximas seções vamos avaliar a utilização de mecanismos de meta-programação no OiL e no Mico para implementação de *proxies*.

4.3.1 Chamadas Síncronas e Assíncronas no OiL

O componente `ProxyManager` é responsável pela criação dos *proxies*. Portanto, qualquer mudança nas funcionalidades dos *proxies* requer a modificação desse componente. A implementação típica do `ProxyManager` consiste em implementar um *proxy* como um objeto que tem suas operações definidas através de meta-tabelas. As operações de cada *proxy* são obtidas através de *caches*. O *cache* permite criar novas operações para os *proxies* à medida que elas são chamadas e também permite reutilizar operações já criadas. No caso de *proxies* sem interface, apenas um *cache* é utilizado. Caso contrário, todos os *proxies* de uma única interface compartilham o mesmo *cache* de operações, que só permite criação das operações definidas naquela interface.

As operações contidas em cada *cache* são fechos de funções Lua criados dinamicamente pela função `newoperation` do `ProxyManager`. Isso permite utilizar essa mesma implementação do componente `ProxyManager` para criar fábricas de *proxies* que façam tanto chamadas síncronas quanto assíncronas, bastando implementar a função `newoperation` adequadamente. A figura 4.7 mostra a implementação de uma função para criação de operações síncronas e a figura 4.8 mostra a implementação de uma função para criação de operações assíncronas através de um objeto de futuro, implementado pela classe `Future` (linha 1).

Figura 4.7: Implementação de operações síncronas no OiL.

```
1  local SyncProxies = ProxyManager()
2
3  local function SyncProxies.newoperation(operation)
4    return function(self, ...)
5      local reference = self.__reference
6      local invoker = self.__invoker
7      local request = assert(invoker:newrequest(reference, operation, ...))
8      assert(invoker:getreply(request))
9      if request.success then
10       return unpack(request, 1, request.n) -- devolve resultados.
11     else
12       error(request[1]) -- lança erro obtido como resultado.
13     end
14   end
15 end
```

A tipagem dinâmica de Lua desempenha um papel essencial para permitir a reutilização da implementação do `ProxyManager`, pois as operações contidas no *cache* são vistas como funções quaisquer cuja assinatura é definida pelo atributo `newoperation` de acordo com o uso dos *proxies* na aplicação. Essa facilidade é um *mecanismo de provisão* que permite deixar em aberto a assinatura e semântica exata das operações de um *proxy*. Tal mecanismo é

Figura 4.8: Implementação de operações síncronas no OiL.

```
1  local Future = oo.class()
2  -- operacao utilizada para consulta da disponibilidade dos resultados
3  function Future:ready()
4    local invoker = self.__invoker
5    return invoker:getreply(self, true) -- 'true' indica consulta sem espera.
6  end
7  -- operação utilizada para obtenção dos resultados
8  function Future:evaluate()
9    local invoker = self.__invoker
10   assert(invoker:getreply(self))
11   if self.success then
12     return unpack(self, 1, self.n) -- devolve resultados.
13   else
14     error(self[1]) -- lança erro obtido como resultado.
15   end
16 end
17
18 local AsyncProxies = ProxyManager()
19
20 local function AsyncProxies.newoperation(operation)
21   return function(self, ...)
22     local reference = self.__reference
23     local invoker = self.__invoker
24     local request = assert(invoker:newrequest(reference, operation, ...))
25     return Future(request) -- 'request' vira um objeto 'Future'.
26   end
27 end
```

muito útil em experimentações com o modelo de programação oferecido pelo OiL.

Com base em definições de interface, também é possível utilizar outros mecanismos de meta-programação no OiL. Por exemplo, é possível gerar código-fonte que implemente os *proxies*. Inclusive, essa geração de código pode ser feita durante a execução da aplicação e ser interpretada em seguida para criação de *proxies*. A principal vantagem da geração dinâmica de código para implementar *proxies* é poder fazer isso sem precisar conhecer os recursos de reflexão da linguagem. Por exemplo, no caso de Lua, isso implica que não é necessário conhecer o conceito de meta-tabelas ou meta-métodos e a implementação dos *proxies* é feita concatenando trechos de código que criam uma função comum de Lua. Em outras palavras, a interpretação de código nesse caso implica em uma *barreira de abstração* menor para o programador. Essa abordagem não foi adotada na implementação de *proxies* no OiL, pois a compilação de código Lua implicaria em uma sobrecarga considerável na criação desses *proxies*.

4.3.2 Chamadas Síncronas e Assíncronas no Mico

O Mico é implementado em C++, que não oferece suporte para reflexão computacional para gerar implementações de *proxy*, como é feito no OiL. Adicionalmente, o mecanismo de pré-processamento utilizado em C++ restringe-se

a expansões simples de código e algumas formas condicionais. Em particular, a meta-linguagem utilizada para pré-processamento em C++ não permite inspeção do código. Portanto, para utilizar esse mecanismo na implementação de *proxies* no Mico seria necessário definir uma meta-linguagem mais adequada para processamento de código C++. Ao invés disso, o Mico utiliza geração de código como mecanismo de meta-programação.

De forma similar ao que é feito no OiL, o compilador de IDL interpreta definições de interface e tipos descritos na linguagem IDL, criando objetos que representam essas informações e permitem manipulá-las. A estrutura escolhida para esses objetos criados pelo compilador de IDL do Mico segue o modelo do repositório de interfaces de CORBA, que implementa um mecanismo de introspecção de interfaces de objetos como parte do suporte a reflexão computacional de CORBA. O compilador então extrai as informações desses objetos para criar a implementação dos *proxies* através da geração de código-fonte C++. Esse código gerado deve então ser compilado juntamente com a aplicação, servindo como uma cola entre a aplicação e o middleware.

O compilador IDL é o componente do Mico responsável pela forma como as chamadas são realizadas. Comparativamente, o compilador IDL do Mico desempenha um papel similar ao `ProxyManager` no OiL. Portanto, para implementar o suporte a chamadas síncronas e assíncronas é necessário modificar o compilador IDL ou oferecer implementações alternativas dele.

A implementação do compilador IDL do Mico é organizada na forma de três componentes. O primeiro é o *front-end*, que é responsável pela interpretação da linguagem de descrição de interfaces e alimentação do repositório de informações de interface, que constitui o segundo componente do compilador de IDL do Mico. O terceiro componente é denominado *back-end* e é responsável por gerar o código com base nas informações contidas no repositório de interfaces. Essa organização facilita a implementação de novos *back-ends*, o que permite gerar diferentes implementações de *proxies*. Na geração de código C++, é necessário gerar declarações explícitas de tipo para estruturas de dados manipuladas e interfaces de *proxies* e *servants*. Essas declarações servem de base para verificações estáticas de conformidade de tipo quando o código gerado é compilado juntamente com o código da aplicação. Isso diminui a *propensão a erros* da utilização do middleware.

Por outro lado, a tipagem estática faz com que a geração de *proxies* síncronos e assíncronos seja consideravelmente diferente. Isso implica em maior trabalho na implementação do suporte a diferentes formas de chamada, pois há pouco reuso. Por exemplo, apesar do padrão CORBA definir um modelo de chamadas assíncronas, denominado AMI (*Asynchronous Method Invocation*),

poucas implementações de CORBA oferecem esse recuso. No Mico, para implementar esse recurso seria necessário modificar o *back-end* do compilador IDL, de forma que ele gerasse a implementação adicional dos *proxies* assíncronos. Em geral, a modificação de um compilador de IDL para adicionar suporte a novos tipos de *proxy* é complexa, como ilustrado em (Arulanthu et al., 2000). Neste trabalho, não implementamos o suporte a chamadas assíncronas no Mico, pois acreditamos que isso traria uma contribuição pouco significativa para a avaliação apresentada neste capítulo e demandaria um grande esforço de programação.

As características das linguagens dinâmicas permitem a utilização de diferentes mecanismos de meta-programação. Em particular, esses mecanismos podem ser utilizados em tempo de execução. Por outro lado, em linguagens menos dinâmicas os mecanismos de meta-programação utilizados são geralmente mais limitados e restritos à etapa de desenvolvimento da aplicação. Essa restrição se reflete na necessidade de definir *compromissos prematuros*. Por exemplo, a aplicação deve prever durante o seu desenvolvimento a necessidade de utilizar diferentes tipos de *proxy*, de forma que o middleware possa gerar o suporte adequado. No caso particular de *proxies* síncronos e assíncronos que apresentam interfaces diferentes, a própria tipagem estática já impõe essa restrição. Entretanto, em algumas aplicações, pode não ser possível ou conveniente assumir esse compromisso durante seu desenvolvimento, sendo necessário que o middleware inclua o suporte para ambas as formas de chamada. O uso de meta-programação em tempo de execução funciona como um *mecanismo de provisão* para esse problema, pois o middleware pode gerar o suporte para chamadas síncronas ou assíncronas sob demanda, de acordo com a necessidade da aplicação durante sua execução.

Uma facilidade particular de Lua é utilizar seus recursos de introspecção para manipulação de descrições de dados feitas com tabelas. Por essa razão, geralmente não é necessário implementar suporte específico para introspecção de dados, como é o caso da implementação do IR de CORBA utilizado no compilador de IDL do Mico, para permitir manipular as informações sobre interface. O interpretador de definições IDL utilizado no OiL não implementa nenhum suporte específico para permitir manipular as definições IDL, ao invés disso são utilizadas tabelas que contêm essas informações e podem ser inspecionadas pelo OiL através dos mecanismos de introspecção de Lua. Outro exemplo disso é a implementação do DOM (*Document Object Model*) para manipulação de definições HTML e XML. A facilidade de Lua nesses casos se caracteriza como uma maior *proximidade de mapeamento* com sistemas

de manipulação de descrição de dados, como é o caso dos geradores de implementação *proxy*.

4.4 Mudança de Recursos Oferecidos

Idealmente, o middleware oferece um conjunto abrangente de funcionalidades como forma de ampliar sua utilização e facilitar seu uso em diferentes aplicações. Entretanto, a implementação de uma funcionalidade geralmente implica em maior consumo de recursos da plataforma, como maior memória para o código do programa, mesmo que a aplicação não necessite de todas as funcionalidades oferecidas pelo middleware. Por exemplo, alguns processos oferecem serviços através de um conjunto de *servants* sem a necessidade de realizar chamadas a objetos remotos. Similarmente, outros processos não oferecem serviços através de *servants* e apenas chamam operações de objetos remotos disponibilizados por outros processos. Nesses casos, é interessante poder modificar o middleware de forma que as funcionalidades desnecessárias não sejam incluídas.

Nesta seção, é apresentada uma análise da facilidade de modificar o middleware para incluir ou excluir diferentes funcionalidades. São consideradas funcionalidades cuja implementação pode ser isolada a alguns poucos componentes — denominadas *modulares* — como o suporte a chamadas remotas (lado cliente) e criação de *servants* (lado servidor), assim como modificações que abrangem diversos componentes — denominadas *transversais* — tais como o suporte a concorrência e interceptação de chamadas.

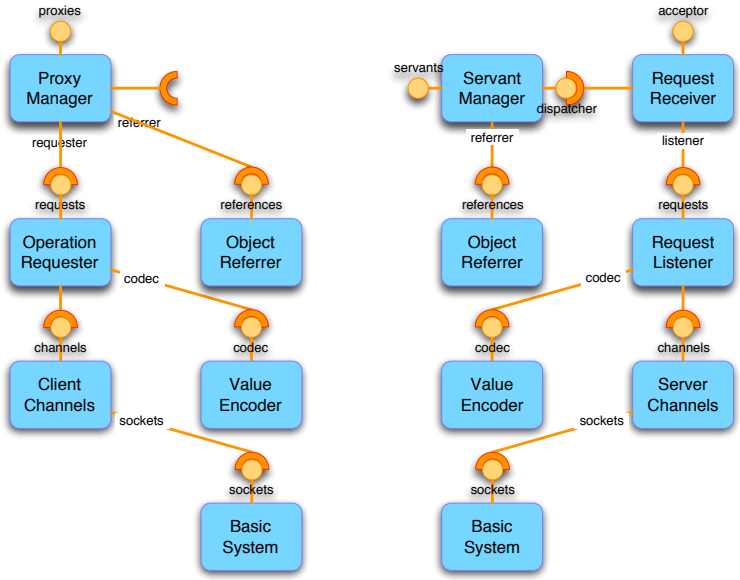
4.4.1 Funcionalidades Modulares no OiL

A arquitetura de componentes do OiL é modelada de forma que certas funcionalidades oferecidas pelo ORB sejam implementadas por um grupo de componentes independentes. Isso permite a configuração das funcionalidades do ORB através da seleção dos componentes que o formam. Por exemplo, é possível desabilitar o suporte a *proxies*, bastando desconsiderar o componente `ProxyManager` na montagem do ORB. Nesse caso, a aplicação pode realizar chamadas diretamente através da faceta `request` do componente `OperationRequester`.

A arquitetura faz separação entre componentes que implementam funcionalidades de servidor e cliente. As funcionalidades de servidor incluem o suporte para aplicações que oferecem serviços através de *servants*, como o registro de *servants*, recebimento e processamento de chamadas, etc. Já as

funcionalidades de cliente incluem o suporte para aplicações que realizam chamadas remotas, como criação de *proxies*, envio de requisições e recebimento de respostas, etc. Essa separação nos permite montar ORBs apenas com funcionalidades de servidor ou cliente, em uma arquitetura similar à ilustrada na figura 4.9.

Figura 4.9: Montagens do OiL com suporte para cliente ou servidor.
4.9(a): Suporte cliente. 4.9(b): Suporte servidor.



Outras funcionalidades podem ser inseridas no ORB através da seleção de componentes adequados ou mesmo conexões entre componentes. Por exemplo, o suporte oferecido pelo componente `TypeRepository` da camada de CORBA permite que o ORB reconheça descrições em IDLs CORBA, ou importe definições de interface através de um repositório de interfaces remoto, e ainda permite o acesso às definições cadastradas usando a interface de inspeção de interfaces de CORBA (*Interface Repository*). A implementação desses recursos demanda um maior uso de memória. A remoção desse componente pode diminuir até um terço da memória utilizada pelo ORB. Outro exemplo é o suporte para resolução de referências locais, que é habilitado através da conexão do componente `ServantManager` ao componente `ProxyManager`, que na criação de novos *proxies* o utiliza para identificar se a referência é local e recuperar sua implementação se for o caso. A criação de *servants* implicitamente também é habilitada através de uma conexão entre o componente `ValueEncoder` e o `ServantManager`, como visto nas seções 3.4.1 e 3.5

O uso de componentes que definem suas dependências através de receptáculos que podem ser conectados a diferentes implementações evita a definição de alguns *compromissos prematuros* na implementação do middleware,

pois esses relacionamentos podem ser resolvidos posteriormente, na montagem do ORB. Por outro lado, receptáculos cujas conexões só são definidas na montagem do ORB introduzem *dependências ocultas*, uma vez que não é possível determinar a implementação exata do serviço acessado através do receptáculo. Isso torna a implementação difícil de entender e a sua utilização mais *propensa a erros* devido a problemas na montagem do ORB. No OiL, isso é amenizado através da definição dos *scripts* de construção e montagem de componentes, que podem ser compostos para formar diferentes roteiros de montagem de ORBs, de forma similar ao código na figura 3.3 na página 49. Entretanto, não é viável oferecer *scripts* para todas as diferentes montagens de ORBs OiL, portanto, ainda pode ser necessário que a aplicação defina a forma de montagem do ORB ou pequenas alterações na arquitetura de um ORB montado, como conexão e desconexão de componentes.

4.4.2 Funcionalidades Modulares no Mico

O modelo de *micro-kernel* e adaptadores acopláveis do Mico oferece uma facilidade para composição de funcionalidades no ORB. De certo modo, esse modelo se assemelha à arquitetura baseada em componentes do OiL. Por exemplo, é possível ativar apenas o suporte a chamadas remotas usando GIOP acoplando o adaptador de chamadas usado localmente (SII) e o adaptador de objeto que envia chamadas através de protocolos GIOP. Similarmente, acoplar ao *micro-kernel* apenas o adaptador de chamadas que recebe requisições GIOP e o adaptador de objetos locais (POA - *Portable Object Adaptor*), ativa apenas o suporte a *servants* e processamento de chamadas.

Por outro lado, o modelo do Mico apresenta diferenças importantes. Primeiramente, o modelo de adaptadores é bastante simples, onde cada adaptador é um conjunto de classes que oferecem os serviços do adaptador através de suas interfaces. Apesar de algumas dependências dos adaptadores serem resolvidas através da iniciação de seus atributos outras dependências importantes são resolvidas através de referências globais acessíveis por métodos de classe (`static`). Exemplos disso incluem a obtenção do *micro-kernel* (*e.g.* `CORBA::ORB_instance`), adaptadores obrigatórios (*e.g.* `MICO::IIOPServer::_instance`), decodificadores de *profile* IOR (*e.g.* `CORBA::IORProfile::register_decoder`), etc. Essa abordagem dificulta a criação de múltiplos ORB (*compromisso prematuro*), pois cada uma dessas referências globais só permite armazenar uma única instância, que, em alguns casos, não pode ser compartilhada entre diferentes ORBs. No caso particular dos decodificadores de *profile* IOR, essa abordagem também implica em

uma *dependência oculta*, pois cada adaptador de objeto que envia requisições usando um protocolo diferente só funciona corretamente se o decodificador de *profile* correspondente estiver registrado. Essa dependência do adaptador de objeto e seu decodificador de *profile* não é clara no código.

Outra diferença importante do modelo do Mico é a *viscosidade* do *micro-kernel*. A classe `CORBA::ORB`, que define a interface e a implementação do *micro-kernel*, é diretamente utilizada pelos adaptadores e a própria aplicação. Portanto, sua implementação não pode ser modificada substituindo a classe a ser utilizada na instanciação do *micro-kernel*. Isso também dificulta oferecer implementações alternativas do *micro-kernel*. Essa alta *viscosidade* aliada à definição prévia das funcionalidades e características dos serviços que ele oferece (*compromissos prematuros*) compromete significativamente a sua flexibilidade. Um exemplo que ilustra isso é que, apesar do modelo do *micro-kernel* favorecer a montagem de um ORB com um número variado de adaptadores, a implementação padrão do *micro-kernel* força a criação de alguns adaptadores obrigatórios, como o adaptador de chamadas que recebe requisições GIOP da rede e o adaptador de objeto que envia essas requisições pela rede. Isso impede criar um ORB no Mico apenas com suporte a chamadas de operação (cliente) ou apenas com suporte ao recebimento de chamadas (servidor), sem modificar a implementação da classe `CORBA::ORB`. Uma limitação similar ocorre quando tenta-se criar um ORB exclusivamente com suporte LuDO, sem suporte GIOP.

Em resumo, por um lado o modelo de *micro-kernel* do Mico reduz a *viscosidade* da implementação do ORB através da possibilidade de acoplar diferentes adaptadores, que podem estender e modificar algumas de suas funcionalidades. Por outro lado, a falta de um modelo para os adaptadores que permita que suas dependências e relacionamentos com outros componentes possam ser definidas na criação do ORB limita consideravelmente as vantagens do modelo. Outro fator que contribui para aumentar a *viscosidade* da implementação do Mico é a utilização de uma implementação de *micro-kernel* fixa, que impõe diversas características do ORB e particularmente a interação com os adaptadores e as estruturas de dados que eles manipulam (*compromissos prematuros*).

4.4.3 Funcionalidades Transversais no OiL

Algumas funcionalidades não são facilmente mapeadas em componentes individuais de forma que possam ser descartadas simplesmente removendo esses componentes da montagem do ORB. Ao invés disso, essas funcionalidades são mais facilmente implementadas como modificações em componentes res-

ponsáveis por outras funcionalidades. Um exemplo desse tipo de funcionalidade é o suporte à verificação de interfaces de objetos nas chamadas de operação através do ORB, como visto na seção 4.2.1. Nesse caso, a implementação consistiu na modificação dos componentes `ProxyManager` e `ServantManager`, cuja funcionalidade principal é oferecer suporte a *proxies* e *servants*, respectivamente. Outros exemplos de funcionalidades transversais incluem concorrência, interceptação de chamadas e *logging*.

Como as funcionalidades transversais implicam em modificações de diferentes componentes do ORB, elas tipicamente levam a um *compromisso prematuro* durante a implementação desses componentes, pois não é possível determinar a necessidade dessas funcionalidades pela aplicação. Contudo, a arquitetura baseada em componentes oferece um *mecanismo de provisão* através de diferentes implementações de um mesmo componente, que pode apresentar suporte para a funcionalidade transversal ou não. Essa foi a solução adotada para o suporte a interfaces descrito na seção 4.2.1, onde são disponibilizadas implementações alternativas dos componentes `ProxyManager` e `ServantManager`.

Por outro lado, quando existe mais de uma funcionalidade transversal que afeta um mesmo componente ou componentes que apresentam implementações alternativas, pode ser inviável oferecer implementações de todas as combinações possíveis. Como exemplo disso, considere a ativação do suporte a concorrência no ORB, que consiste em permitir que múltiplas linhas de execução independentes possam utilizar um mesmo ORB. Nesse caso, é necessário modificar todos os componentes que apresentam regiões críticas, ou seja regiões de seu código suscetíveis a erro devido ao acesso simultâneo de diferentes *threads*. A implementação de tais componentes deve utilizar mecanismos de sincronização de *threads*, de forma que elas cooperem para executar as regiões críticas e evitem erros decorrentes de condições de corrida. No OiL, isso acontece em componentes da camada do protocolo de RMI que permitem a reutilização de um mesmo canal de comunicação por diferentes *threads*, pois os *sockets* criados na montagem `cooperative` são integrados ao escalonador de *threads* de forma que as operações bloqueantes desses *sockets* resultam em pontos de escalonamento implícitos. Dessa forma, durante o envio de uma seqüência de *bytes* por uma dada *thread*, outras *threads* podem interferir nesse envio caso também tentem enviar dados através dessem mesmo *socket*. Uma forma de evitar essa situação é utilizar mecanismos de sincronização entre *threads* de forma que elas coordenem suas execuções de forma a evitar acessos concorrentes a um mesmo *socket*. Contudo, para permitir o uso de diferentes protocolos de RMI, tanto em ORBs com ou sem concorrência, seria necessário fornecer duas implementações potencialmente similares de cada protocolo di-

ferente.

Ao invés disso, o OiL oferece uma única implementação de cada protocolo de RMI, que utiliza funções de sincronização oferecidas pelos objetos *socket* criados pelo componente `BasicSystem`, como ilustrado na figura 4.10 (linhas 7, 9, 24, 29 e 31). Quando o ORB é montado sem suporte a concorrência, o componente `BasicSystem` utilizado cria *sockets* com implementações alternativas dessas funções de sincronização que não produzem nenhum efeito, desabilitando o suporte a concorrência nos componentes da camada de RMI. De certa forma, essas chamadas de funções de sincronização funcionam como pontos de intercessão previamente definidos na implementação do componente que são utilizados para introduzir o suporte à sincronização.

Figura 4.10: Sincronização de *threads* em componentes da camada de RMI.

```
1  function newrequest(self, reference, operation, ...)
2    --[[LOG]] log:ludo("attempt to call ",operation," of ",reference)
3    local result, except = self.context.channels:retrieve(reference)
4    if result then
5      local channel = result
6      local request = self:mkrequest(channel, reference.object, operation, ...)
7      repeat until channel:trylock("write")
8      local result, except = channel:send(self:marshall(request))
9      channel:freelock("write")
10     if result then
11       --[[LOG]] log:ludo("sent new request ",request)
12       result = request
13     end
14   --[[LOG]] else log:ludo("unable to get channel for ",reference)
15   end
16   return result, except
17 end
18
19 function getreply(self, request)
20   --[[LOG]] log:ludo("attempt to get reply for request ",request)
21   local result, except = true, nil
22   if request.success == nil then
23     local channel = request.channel
24     if channel:trylock("read", request) then
25       local codec = self.context.codec
26       while result and (result ~= request) do
27         result, except = self:readrequest(channel)
28         --[[LOG]] else log:ludo("got reply for request ",result)
29         channel:denylock("read", result)
30       end
31       channel:freelock("read")
32     end
33   --[[LOG]] else log:ludo("reply for request ",request," was ready")
34   end
35   return result, except
36 end
```

Uma abordagem similar é utilizada para oferecer suporte opcional para geração de relatórios de execução ou *logging*, onde cada componente gera mensagens que descrevem as decisões e ações mais relevantes. Os relatórios de execução são utilizados tanto como suporte à depuração como também como recurso de auditoria das aplicações. Diferentemente do suporte a concorrência, o recurso de *logging* é uma funcionalidade transversal que afeta todos os

componentes do ORB e de forma significativa, exigindo a definição de diversos pontos de intercessão na implementação dos componentes. Para ilustrar isso, considere as linhas da figura 4.10 que começam com a marcação `--[[LOG]]`. Essas linhas apresentam chamadas de funções de *logging* nos componentes. Devido ao grande número dessas chamadas, elas podem causar um impacto considerável no desempenho do ORB mesmo quando não produzem nenhum efeito. Por exemplo, no OiL, quando as funções de *logging* não realizam ação alguma, o custo de suas chamadas representa aproximadamente 20% do tempo de processamento do ORB.

A introdução de chamadas de função adicionais em pontos pré-definidos do código, como forma de introduzir pontos de intercessão, representa um *compromisso prematuro* na implementação dos componentes devido ao impacto dessas chamadas no desempenho do sistema, mesmo que elas se mostrem desnecessárias. Uma alternativa a isso é a utilização de meta-programação para alterar a implementação dos componentes no momento da sua montagem ou durante sua execução. Como visto na seção 4.3, as características dinâmicas de Lua permitem a utilização de diferentes formas de meta-programação. Por exemplo, o suporte de *logging* nos componentes do OiL pode ser desabilitado através de pré-processamento. Para tanto, são utilizadas marcações especiais no código (*e.g.* `--[[LOG]]`) que indicam as linhas que devem ser descartadas antes da sua compilação. Em Lua, o pré-processamento pode ser aplicado diretamente no código-fonte durante o desenvolvimento, ou durante a carga da implementação pela aplicação. Para tanto, pode-se definir um novo carregador de módulos Lua (`package.loaders`) que aplique o pré-processamento antes de carregar o módulo. Esse novo carregador de módulos pode inclusive carregar implementações alternativas de um módulo já carregado previamente. Por exemplo, o carregador pode assumir que todos os módulos com o sufixo `.nolog` devem ser carregados como o módulo de mesmo nome, sem o sufixo, porém aplicando pré-processamento para excluir as linhas com a marcação que indica o suporte de *logging*. Isso permitiria ter diferentes implementações de um mesmo módulo em memória e a criação de ORBs com diferentes funcionalidades transversais em um mesmo processo ou estado Lua.

Apesar do dinamismo da linguagem Lua permitir o pré-processamento da implementação dos componentes na sua carga e mesmo a carga de um mesmo módulo com diferentes opções de pré-processamento, essa abordagem pode ser ineficiente por carregar implementações similares que resultam em grandes porções de código duplicado. Além disso, essa abordagem não permite facilmente habilitar as funcionalidades transversais após a montagem do ORB. Uma alternativa para isso é utilizar uma forma de meta-programação mais

dinâmica como a reflexão computacional. Por exemplo, através da biblioteca de depuração de Lua que permite a definição de ganchos de execução que podem ser utilizados para introduzir novos trechos de código em um programa em execução.

Na seção 2.2.1, são apresentadas duas técnicas para aumentar o suporte a reflexão de Lua: (a) através de *proxies* de valores; e (b) através de classes de objeto. As classes LOOP seguem a abordagem (b), o que permite interceder nas operações definidas pelas classes, através de modificações na classe. Como os componentes do OiL são implementados como classes LOOP, é possível modificar as classes para introduzir funcionalidades transversais. Entretanto, a modificação de uma classe afeta todas as suas instâncias igualmente. Em particular, a mudança de uma classe que implementa os componentes de diferentes ORBs, resulta na modificação simultânea de todos esses ORBs. Consequentemente, essa abordagem não propicia a criação de diferentes ORB que apresentem funcionalidades transversais distintas.

O LOOP também oferece *templates* que criam componentes utilizando *proxies* para o acesso de suas portas, como na abordagem (a) citada anteriormente. Isso permite interceptar operações feitas através das portas do componente e notificar esses eventos a tratadores especiais, denominados *interceptadores de porta*. A definição de intercetadores no LOOP pode ser feita por *template* de componente, por fábrica de componente ou em componentes individuais. Isso permite interceder de formas diferentes em componentes de ORBs distintos. O suporte a interceptação de chamadas do OiL é implementado utilizando o suporte a interceptadores de porta do LOOP, como apresentado na seção 3.3.6.

A intercessão em portas de componentes é um mecanismo restrito. Por exemplo, para implementar um suporte similar aos interceptadores de chamada do padrão CORBA (*Portable Interceptors*), é necessário interceder na codificação e decodificação das mensagens GIOP. Entretanto, no OiL isso é feito internamente pelos componentes `OperationRequester` e `RequestListener` da camada de RMI, sem operações através de portas. Portanto, nesse caso é necessário utilizar outro mecanismo para interceder no funcionamento interno desses componentes, como apresentado na seção 3.5.1. Em implementações anteriores do OiL, isso foi feito dividindo esses componentes, entretanto essa divisão de mostrou artificial pois era motivada principalmente como forma de permitir a utilização de interceptadores de porta como mecanismo de intercessão.

De forma geral, a implementação de funcionalidades transversais no OiL de maneira flexível é difícil. Entretanto, o dinamismo da linguagem Lua e

a flexibilidade da arquitetura baseada em componentes permitem diferentes abordagens para minimizar esse problema. No OiL, são adotadas diferentes abordagens de acordo com a natureza da implementação da funcionalidade:

O suporte a verificação de interfaces é oferecido através de implementações alternativas de alguns componentes que podem ser selecionadas na montagem do ORB. A principal desvantagem dessa abordagem é a ineficiência devido a duplicação de código similar entre as implementações alternativas. Essa solução foi adotada pois o suporte à verificação de interfaces pode ser introduzido nos componentes relevantes através da extensão sua implementação, onde parte da implementação original é reutilizada através de herança de classes.

O suporte a concorrência é implementado através da introdução de chamadas de função em pontos estratégicos da implementação dos componentes. Essas funções funcionam como pontos de intercessão pré-definidos que são utilizados para implementação de mecanismos de sincronização entre *threads*. A principal desvantagem dessa abordagem é o *compromisso prematuro* de fornecer pontos de intercessão que podem ser desnecessários na utilização do middleware, resultando em degradação do desempenho. Essa solução foi adotada pois o modelo de concorrência cooperativa utilizado pelo OiL simplifica muitos aspectos da sua implementação, resultando na necessidade de um número reduzido de chamadas de funções de sincronização e controle de *threads*.

O suporte a logging é implementado de forma similar ao suporte a concorrência, entretanto as chamadas de função são identificadas através de marcações especiais no código que podem ser removidas através de pré-processamento. As principais desvantagens dessa abordagem são a necessidade de definir uma meta-linguagem para manipulação do código fonte e implementá-la, e também a dificuldade em utilizar simultaneamente implementações com diferentes opções de pré-processamento, o que impede a criação de ORBs com diferentes funcionalidades transversais. Essa solução foi adotada porque o uso de simples marcações no código para indicar as várias chamadas de função que implementam o suporte de *logging* permite removê-las usando um pré-processamento simples, resultando em melhor desempenho da implementação.

O suporte a interceptação de chamadas é implementado usando reflexão computacional através da intercessão nas operações feitas em portas dos componentes que formam o ORB. A principal desvantagem dessa abordagem é a complexidade devido ao uso de novas abstrações relacionadas à reflexão com-

putacional e as limitações particulares do modelo de reflexão computacional de Lua. Essa solução foi adotada por ser possível de ser implementada através da intercessão em portas dos componentes que formam o ORB.

Outras abordagens podem ser utilizadas para implementar funcionalidades transversais usando a possibilidade de reconfiguração de componentes e as características dinâmicas de Lua. Por exemplo, a reflexão computacional de Lua pode ser utilizada para implementar recursos de programação orientada a aspectos (Batista e Vieira, 2007), que permitem modularizar a implementação de funcionalidades transversais em elementos denominados *aspectos*, que podem ser automaticamente entrelaçados aos componentes afetados na montagem do ORB ou mesmo durante sua execução. Assim como a reflexão computacional e o uso de interceptadores, a utilização de programação orientada a aspectos introduz um conjunto de novas abstrações que aumentam a *barreira de abstração* da implementação, tornando-a mais difícil de compreender e manipular.

4.4.4 Funcionalidades Transversais no Mico

De forma similar ao OiL, algumas funcionalidades oferecidas pelo Mico afetam diferentes porções da implementação, sendo difíceis de serem isoladas em módulos independentes. Em particular, o Mico oferece suporte para as mesmas funcionalidades transversais discutidas na seção anterior: verificação de interfaces, concorrência, interceptação de chamadas e *logging*. Contudo, o suporte à verificação de interfaces não é opcional pois é imposto pela tipagem estática de C++. No restante desta seção, são apresentadas as diferentes abordagens para implementação das demais funcionalidades transversais no Mico.

O mecanismo mais utilizado na implementação do Mico para selecionar funcionalidades transversais é o pré-processamento através de diretivas `#ifdef` do pré-processador de C. Um exemplo disso é o suporte à concorrência preemptiva oferecido pelo Mico. Ao contrário do modelo de concorrência utilizado no OiL, a concorrência preemptiva geralmente exige o uso mais intenso de mecanismos de sincronização de *threads* pois o programador não tem controle da execução das *threads* e seu acesso a recursos compartilhados, em particular, variáveis e estruturas de dados. Por isso, o suporte a concorrência no Mico é mais intrusivo e mais abrangente que no OiL, afetando todos os elementos da arquitetura, desde o *micro-kernel* até os adaptadores e o escalonador de tarefas. A solução adotada nesse caso é similar à implementação do suporte a *logging* no OiL. A implementação do suporte a concorrência do

Mico é feita utilizando marcações (`#ifdef HAVE_THREADS`) que são utilizadas pelo pré-processador para inserir ou remover código da implementação.

A vantagem da utilização de pré-processamento em C++ em relação a Lua é que C++ já dispõe de um pré-processador padrão, apesar de utilizar uma meta-linguagem limitada. Em Lua é necessário implementar o pré-processamento. Entretanto, é possível utilizar implementações para suporte a pré-processamento de código Lua como a apresentada em (Fleutot e Tratt, 2007). O pré-processador de C pode ser utilizado para pré-processar código em outras linguagens, inclusive Lua, porém esse uso não é recomendado pois o pré-processador de C pressupõe algumas características do código que são particulares de C e suas linguagens derivadas, como o formato de comentários cujo conteúdo é ignorado pelo pré-processador.

Diferentemente de linguagens dinâmicas, a linguagem C++ permite otimizações de compilação de código que podem excluir por completo chamadas de funções que não produzem efeito algum. Portanto, o uso de chamadas de função adicionais na implementação dos componentes para introduzir suporte de *logging*, como é feito no OiL, não implica necessariamente em custo adicional, pois tais chamadas podem ser otimizadas e removidas do código compilado por completo quando suas implementações são vazias. Entretanto, na implementação do Mico, essa abordagem não é adotada, pois é necessário impor limitações sérias nas funções de *logging* de forma que o compilador identifique com segurança que elas podem ser descartadas no programa compilado. Tanto o suporte a *logging* como interceptação de chamadas é codificado diretamente na implementação do *micro-kernel* e dos adaptadores, sem facilidades para desabilitá-los.

De forma geral, o Mico adota duas abordagens para implementação de funcionalidades transversais. A primeira é introduzir a implementação da funcionalidade sem prover mecanismos adequados para desabilitá-la quando necessário. Essa abordagem se reflete como um *compromisso prematuro* de que as aplicações farão uso dessas funcionalidades ou o custo dessa implementação não afeta significamente as aplicações. Essa é a abordagem adotada na implementação do suporte a *logging* e interceptação de chamadas, por exemplo. A segunda abordagem é utilizar diretivas de pré-processamento como um *mecanismo de provisão* que permita desabilitar partes da implementação antes da compilação. Portanto, é necessário que cada aplicação compile o middleware com opções de pré-processamento adequadas. Essa abordagem é utilizada na implementação do suporte à concorrência preemptiva.

4.5 Avaliação Geral

Nas seções anteriores, foram discutidos alguns aspectos da flexibilidade do OiL e do Mico através da análise de um conjunto de implementações introduzidas na infra-estrutura básica oferecida por cada middleware. Apesar dos casos avaliados permitirem identificar vários aspectos da flexibilidade dessas implementações, é importante ressaltar que eles não cobrem todos os aspectos. Nesta seção, vamos rever os aspectos levantados nas seções anteriores, analisando cada uma das 14 dimensões do CDN (seção 2.4), porém com um enfoque mais geral. Também apresentamos alguns aspectos que se revelaram no uso do OiL em outros experimentos, em particular, em projetos de alunos de graduação e pós-graduação em Ciência da Computação que trabalharam com o OiL que são apresentados na seção 7.1.

Viscosidade Tanto o modelo de componentes do OiL como o modelo de *micro-kernel* do Mico efetivamente facilitam algumas adaptações, tais como a introdução do suporte a novos protocolos, pois isso pode ser feito através da introdução de novos componentes, seja uma nova camada de protocolo, no caso do OiL, ou novos adaptadores e decodificadores de *profile*, no caso do Mico. Contudo, o Mico é mais viscoso em relação à alteração de suas funcionalidades básicas, não apenas devido ao *micro-kernel*, mas também alguns adaptadores obrigatórios, como é o caso dos que implementam o suporte ao GIOP, ou mais especificamente o IIOP. Isso é razoável levando em consideração que o Mico segue o padrão CORBA, o qual define que o suporte ao IIOP é obrigatório. Entretanto, uma das principais causas dessa *viscosidade* são as dependências explícitas entre as classes que implementam esses componentes que são definidas no próprio código dessas classes. Dessa forma, para remover ou substituir um componente, é necessário modificar todas as referências explícitas às classes que o implementam. O modelo de componentes utilizado pelo OiL permite que essas dependências sejam definidas pela aplicação na criação do ORB.

A tipagem dinâmica de Lua também contribui para reduzir a *viscosidade* da implementação do OiL, pois as modificações não implicam em ajustes nas declarações de tipos, como geralmente ocorre nas modificações no Mico. Isso é particularmente evidente em modificações que afetam estruturas de dados ou interfaces que são manipuladas pelos componentes.

Dependências Ocultas Se por um lado, as dependências explícitas do Mico aumentam a *viscosidade*, o uso de componentes no OiL introduz *dependências*

ocultas entre as classes que implementam os componentes, pois elas só são determinadas na montagem do ORB, que é feita pela aplicação. Como visto na seção 2.4 isso implica em uma maior dificuldade de compreensão da implementação do OiL e consequentemente torna as modificações mais difíceis de serem idealizadas. Inclusive, essa é uma dificuldade recorrente de alunos que trabalharam com o OiL. O Mico também apresenta algumas *dependências ocultas* de forma similar ao OiL, como por exemplo a dependência entre os adaptadores de chamada e objeto que implementam um dado protocolo e o decodificador de *profile* IOR daquele protocolo.

A tipagem dinâmica de Lua permite mais facilmente introduzir *dependências ocultas*, como é o caso de estruturas de dados que são manipuladas por diferentes componentes. Um exemplo disso é a referência de objeto. No Mico é possível rastrear os trechos de código que utilizam essa estrutura através do seu tipo. Em Lua, esse rastreamento é mais difícil. A reflexão computacional também contribui para introduzir *dependências ocultas*, pois muitas vezes a reflexão é utilizada para modificar partes da implementação de forma implícita. Um exemplo disso são os *proxies* implementados através de meta-tabelas que criam cada operação implicitamente no momento da primeira chamada. Nesse caso, a origem da implementação das operações não é explícita no *proxy*.

Compromissos Prematuros A decisão de restringir a implementação do Mico ao padrão CORBA pode ser vista como um compromisso que dificulta a sua utilização em cenários não previstos ou contemplados pelo modelo de CORBA. Contudo, os compromissos da implementação do Mico influenciados por CORBA, como apresentados neste trabalho, podem ser evitados através de alterações na sua implementação que não invalidam sua implementação ou o modelo de *micro-kernel*.

Acreditamos que a tipagem estática leva a adoção de *compromissos prematuros*, uma vez que, quanto mais se conhece da implementação do sistema, melhor ele pode ser representado através do sistema de tipos, permitindo que a verificação estática de tipos possa identificar mais erros na implementação do sistema. Por exemplo, a padronização de interfaces manipuladas pelo *micro-kernel* através do modelo IOR de CORBA permite definir uma interface para manipulação dessas referências. Dessa forma, a manipulação inadequada dessas estruturas pode ser identificada pelo compilador de C++. Por outro lado, se não houver um modelo de referências de objeto definido pelo *micro-kernel*, então não é possível definir uma interface padrão para sua manipulação. Consequentemente, cada adaptador que manipulasse a referência faria uso de *casting* para uma interface particular, de acordo com uma indicação de tipo da

referência. Nesse caso, o compilador não poderia verificar as operações sobre essas referências, pois o sistema de tipos não descreve essa parte da implementação.

Mecanismos de Provisão O suporte a reflexão computacional através de meta-tabelas e o uso extensivo de tabelas no OiL fornecem um *mecanismo de provisão* efetivo para muitos aspectos da sua implementação. Tabelas podem ser vistas como elementos genéricos que podem ser ajustados para se comportar de diferentes maneiras, podendo se ajustar de acordo com necessidades futuras que não puderam ser previstas. Tais *mecanismos de provisão* foram efetivos para facilitar implementações no OiL de novas funcionalidades. Exemplos apresentados anteriormente incluem as tabelas que representam requisições de operação e são estendidas para guardar a conexão utilizada, e a implementação do mecanismo de interceptação de portas de componentes funciona como um *mecanismo de provisão* oferecido pelo OiL e é utilizado na implementação do suporte a interceptação de chamadas. No Mico, os *mecanismos de provisão* são implementados pelo programador e tipicamente não são expressos adequadamente pelo sistema de tipos de C++ e portanto não são verificados durante a compilação, como é o caso do uso de ponteiros genéricos (`void*`).

Avaliação Progressiva A tipagem dinâmica efetivamente facilita avaliações progressivas de uma implementação parcial. No OiL, é possível utilizar a implementação, mesmo que alguns componentes estejam com a implementação incompleta. Isso facilita a implementação incremental, o que é particularmente útil em modificações experimentais, pois permite avaliar algumas decisões de projeto durante o desenvolvimento. No Mico, é necessário desenvolver a implementação completa dos seus componentes para poder executar alguma coisa. Outro aspecto que facilita *avaliações progressivas* é a interpretação de código, pois permite que a implementação incremental possa ser feita de forma interativa, como por exemplo através de um console que executa trechos de código fornecido pelo usuário.

Por outro lado, a tipagem dinâmica não permite identificar o quanto da implementação já está feita ou correta. Inclusive, não é incomum um aluno desenvolver parte de uma implementação no OiL e esquecer de completá-la, pois os testes desenvolvidos pelo aluno não utiliza os recursos faltando. Nesse sentido, a tipagem estática funciona como um conjunto de testes gerados automaticamente. Dessa forma, o programador pode identificar a conclusão de uma etapa do desenvolvimento mais facilmente, que é quando a parte da implementação modelada no sistema de tipos está completa. Contudo, isso não

indica se a implementação está correta.

Propensão a Erros A tipagem estática de C++ permite identificar alguns erros que em Lua só são identificados durante a execução. Contudo, não é possível identificar todos os erros da implementação (*e.g.* semânticos) através dessa tipagem estática. Por exemplo, erros na codificação dos dados para envio através da rede não são identificados pela tipagem estática de C++. Consequentemente, outros mecanismos são necessários para prevenção de erros, como a execução de testes. O uso de testes de regressão tem se mostrado eficiente na garantia de qualidade da implementação do OiL.

No caso específico do Mico, as verificações de tipo na compilação permitem identificar alguns erros da montagem do *micro-kernel* e seus adaptadores obrigatórios, pois as dependências explícitas entre esses elementos são modeladas através do sistema de tipos. Contudo, alguns aspectos da montagem do *micro-kernel* e adaptadores adicionais não são verificadas pelo compilador de C++. Em particular, é possível adicionar adaptadores que implementem o protocolo LuDO e não registrar um decodificador de *profile* correspondente, o que impossibilitaria o uso dos adaptadores. O OiL, por outro lado, não oferece suporte para detecção de erros na montagem de componentes. Em geral, erros na montagem resultam em erros de tipo que são detectados pela tipagem dinâmica de Lua. Em algumas situações, a indicação do erro de Lua pode ser o suficiente para identificar o erro na implementação, contudo esse nem sempre é o caso.

Operações Complicadas A gerência de memória em C++ é feita inteiramente pelo programador. De certa forma, essa tarefa não envolve *operações complicadas*, pois C++ oferece abstrações adequadas para alocação e liberação de memória. Entretanto, quando a gerência de memória envolve estruturas complexas formadas por cadeias de objetos interconectados, a gerência de memória passa a ser uma operação complexa, pois envolve múltiplos níveis de indireção e ordens específicas em que as ações devem ser executadas. Em Lua, essa complexidade é inteiramente ocultada pelo gerenciamento automático de memória realizado pelo coletor de lixo. Um aspecto particular de Lua que evita algumas *operações complicadas* é a flexibilidade das tabelas para implementar de forma trivial estruturas de dados cuja implementação em C++ geralmente envolve *operações complicadas* de manipulação de ponteiros e alocação dinâmica de memória. Por outro lado, o uso de reflexão computacional pode ser visto como uma *operação complicada*, pois muitas vezes não é trivial compreender as conseqüências das modificações feitas no programa através da reflexão.

Barreira de Abstração O OiL impõe uma *barreira de abstração* composta pelo modelo de componentes utilizado e pelos componentes da sua arquitetura, que podem ser numerosos (ver capítulo 3) e as interações entre eles. Para implementação de novos protocolos, é necessário conhecer a separação entre a camada de apresentação e a camada de protocolo, assim como as interfaces e estruturas de dados utilizadas nas interações entre essas camadas. Por outro lado, essas abstrações são relativamente simples. Em particular, a maior parte das estruturas de dados não tem estrutura pré-definida e as interfaces que regem as interações têm no máximo seis operações.

O Mico, por outro lado, não define um modelo de componentes especial, pois os componentes da arquitetura baseada em *micro-kernel* são implementados como objetos C++ comuns. Além disso, a arquitetura do Mico define poucos componentes, onde podemos destacar sete: *micro-kernel*, adaptadores de chamada locais e remotos, adaptadores de objeto locais e remotos, escalonador de tarefas e decodificadores de *profile*. Entretanto, a *barreira de abstração* do Mico inclui muitas abstrações impostas pelo padrão CORBA, tais como o modelo de referências do IOR e alguns aspectos do protocolo GIOP, como o redirecionamento de chamadas (*location forward*).

O uso de reflexão computacional em Lua não constitui uma *barreira de abstração* pois é visto como uma funcionalidade adicional da linguagem e não como um mecanismo fundamental para a maior parte das aplicações. Entretanto, o OiL faz uso de reflexão na sua implementação, em particular na implementação de *proxies*. Para compreender a implementação atual de *proxies* do OiL, é necessário conhecer os mecanismos de reflexão de Lua. Portanto, na implementação específica do OiL, a reflexão computacional constitui uma *barreira de abstração*.

Demanda de Abstrações Nem o OiL nem o Mico demandam a criação de novas abstrações para implementação de funcionalidades. Em particular, no OiL, não é necessário definir novos tipos de componentes, pois modificações podem ser feitas reimplementando os componentes definidos pela arquitetura. Lua também não exige a definição de novas abstrações na criação de novos programas, apesar dos programas tipicamente definirem novas abstrações através de tabelas. O mesmo é válido para o Mico, pois não é necessário definir novos tipos de adaptadores ou outros componentes que se conectam ao *micro-kernel*. C++ também permite a criação de programas sem necessidade de definir novas abstrações através de classes.

Resistência a Abstrações Tanto Lua como C++ permitem que o programador utilize novas abstrações, seja através de tabelas e meta-tabelas em Lua, ou através de classes em C++. O OiL também oferece mecanismos para criação de novas abstrações através da criação de novos componentes e camadas arquiteturais. De forma similar, a arquitetura de *micro-kernel* do Mico, permite a introdução de novos elementos na arquitetura, que podem ser definidos através de classes de objetos como os demais elementos da arquitetura.

Visibilidade De forma geral, a implementação do OiL tende a ser mais facilmente visualizada que a do Mico. No OiL, a implementação de um componente é geralmente feita por uma única classe, que é inteiramente definida em um único arquivo. Diferentemente disso, a implementação dos componentes no Mico é separada em dois arquivos, onde um contém declarações de variáveis, funções e classes, enquanto que o outro contém suas definições. Essa separação pode dificultar a visualização da implementação do componente ou mesmo de partes dele, como uma única função, pois sem o auxílio de um editor de texto adequado pode ser difícil a visualização da implementação como um todo. A dificuldade de visualização resultante dessa separação dificulta a compreensão do Mico, assim como o planejamento de modificações. Outro aspecto que dificulta a *visibilidade* da implementação do Mico é a maior *prolixidade* da linguagem C++ devido às declarações explícitas de tipo, como será discutido a seguir.

Prolixidade Apesar da sintaxe de C++ fazer uso de *tokens* léxicos menores do que em Lua, de forma geral, a implemementação do Mico é mais prolixa do que a implementação do OiL. Uma das principais razões para isso é a necessidade de definir os tipos de valores do programa através de declarações explícitas. Essa *prolixidade* efetivamente compromete a *visibilidade* das implementações e, consequentemente, a facilidade de aprendizado do sistema. A *prolixidade* causada pelas declarações explícitas também torna algumas modificações mais viscosas, pois exigem modificações nessas declarações. As tabelas de Lua também contribuem para uma menor *prolixidade*, pois tipicamente permitem descrever estruturas de dados complexas de forma sucinta.

Expressividade de Papéis Uma dificuldade da leitura da implementação do OiL, é a utilização de tabelas para implementar diversas abstrações. Em muitos trechos de código Lua, o papel de uma abstração representada por uma tabela não é claro a menos que se analise todos os pontos do código em que ela é utilizada. Inclusive, uma tabela pode desempenhar múltiplos papéis, por exemplo uma classe LOOP pode ser tanto uma meta-tabela, uma fábrica de

objetos, um molde de objetos ou um objeto com operações. Por outro lado, essa polivalência também aparece, em menor grau, nas classes de C++, que são utilizadas para definir estruturas de dados, implementação de objetos, interfaces, etc. Portanto, também não é fácil identificar o papel de uma classe da implementação do Mico. A utilização das funções auxiliares oferecidas pelos módulos da biblioteca LOOP permitem minimizar um pouco esse problema, pois tornam explícito o papel de algumas tabelas, em particular aquelas que implementam classes (operação `oo.class` oferecida pelo LOOP), assim como as que representam *templates* ou fábricas de componentes.

Proximidade de Mapeamento Pelo fato do OiL ser implementado em uma linguagem dinâmica, ele se mostrou mais adequado à implementação de um protocolo de RMI para chamadas dinamicamente tipadas como LuDO, do que para a implementação de um protocolo estático como o GIOP, ou seja, que se baseia na consistência dos tipos entre todos os objetos distribuídos no sistema[2]. Similarmente, o Mico se mostra mais adequado à implementação de um protocolo estático como o GIOP. Adicionalmente, essa *proximidade* do Mico com o GIOP é amplamente facilitada devido à grande influência do padrão CORBA na sua implementação. Por outro lado, os compromissos assumidos na implementação do Mico devido à sua adoção do padrão CORBA, o tornaram inviável para implementação do LuDO, sendo necessário adaptar o protocolo para permitir sua implementação. A adoção do modelo de CORBA também influencia para que o Mico se mostre pouco adequado para implementação de uma versão reduzida com suporte a um único protocolo, pois sua implementação foi projetada incluindo suporte específico para múltiplos protocolos.

Tanto o OiL como o Mico são implementações que podem ser estendidas através da introdução de novos componentes, que podem definir novos tipos que não são conhecidos pela implementação original. Inclusive, esses tipos podem ser utilizados por diferentes componentes que trocam valores através da implementação original. Um exemplo disso no Mico consiste da inclusão de um novo decodificador de *profile* que cria objetos de uma sub-classe de `IORProfile`, que não é conhecida pelo *micro-kernel*. Esse objeto, é então repassado, através do *micro-kernel*, para o adaptador de chamada que ofereça suporte para aquele *profile*, que precisa fazer um *casting* para a sub-classe de `IORProfile` implementada pelo objeto. A tipagem estática de C++ não pode garantir a corretude desse tipo de *casting*, que só pode ser feito durante a execução da aplicação com o auxílio do suporte a RTTI. Portanto, a tipagem

[2]A verificação estática de tipos em um sistema distribuído só é possível se o verificador de tipos puder garantir a implantação de toda a aplicação.

dinâmica de Lua pode ser considerada mais adequada para implementar esse tipo de interação que envolve tipos não conhecidos durante pela implementação original.

Consistência Nas avaliações apresentadas nesse trabalho, não foi possível identificar diferenças significativas na consistência de representações na implementação do Mico ou do OiL.

Notação Secundária O suporte a *notação secundária* de C++ e Lua é representado principalmente pelo suporte a comentários no código. Esse recurso particular é muito similar em ambas as linguagens. Nem o OiL nem o Mico definem suporte adicional para *notação secundária*.

A tabela da figura 4.11 mostra um resumo da comparação do OiL com o Mico apresentada neste capítulo. A segunda coluna indica o resultado da comparação com o Mico em cada dimensão cognitiva. As demais colunas indicam as principais características do OiL que se destacam na avaliação de cada dimensão cognitiva e se o seu impacto na flexibilidade geral do OiL é positivo ou negativo. Cada característica é representada por uma sigla de duas letras, listadas na legenda abaixo da figura. A sigla das características relacionadas à linguagem dinâmica são apresentadas em negrito. De forma geral, a tipagem dinâmica e o suporte à reflexão computacional de Lua, aliados à arquitetura baseada em componentes, tendem a melhorar a flexibilidade do OiL. Não apenas em relação ao custo da realização da modificação (*viscosidade*), mas também em relação aos outros aspectos cognitivos importantes, como menos *operações complicadas*. Em particular, o OiL apresenta uma boa avaliação em dimensões relevantes em modificações exploratórias, tais como *mecanismos de provisão* e *avaliação progressiva*. Por outro lado, o OiL tende a apresentar um maior número de *dependências ocultas*, o que dificulta a compreensão do sistema e, consequentemente, o planejamento das modificações.

À parte da comparação com o Mico, o OiL também apresenta problemas importantes que comprometem sua flexibilidade, como uma maior *propensão a erros* devido à falta de verificações estáticas, que é uma característica particularmente indesejável em modificações definitivas. A tipagem estática oferece grande vantagem nesse aspecto. Em particular, acreditamos que a tipagem estática baseada em inferência de tipos pode diminuir a *propensão a erros* sem implicar nos problemas apresentados anteriormente de linguagens com declarações explícitas de tipo, como C++. De forma geral, declarações explícitas tornam a linguagem mais *prolixa*, o que dificulta a sua *visibilidade* devido ao

Figura 4.11: Resumo das avaliação de flexibilidade do OiL com CDN.

Dimensão Cognitiva	OiL vs. Mico	Positivo	Negativo
Viscosidade	$<$ Mico	CS,**TD**	
Dependências Ocultas	$>$ Mico		CS,**TD**,**RC**
Compromissos Prematuros	$\leq$ Mico	CS,**TD**	
Mecanismos de Provisão	$>$ Mico	**RC**,TL,CS	
Avaliação Progressiva	$>$ Mico	**TD**,**IC**	
Propensão a Erros	$\simeq$ Mico		**TD**
Operações Complicadas	$<$ Mico	**GM**,TL	**RC**
Barreira de Abstração	$\geq$ Mico		CS,AI,**RC**
Demanda de Abstrações	$\simeq$ Mico		
Resistência a Abstrações	$\simeq$ Mico	**RC**,TL	
Visibilidade	$\geq$ Mico	SL,**TD**	
Prolixidade	$\leq$ Mico	SL,**TD**,TL	
Expressividade de Papéis	$\simeq$ Mico		TL
Proximidade de Mapeamento	$\geq$ Mico	**TD**,TL	

Legenda:

$<$	É menor
$\leq$	Tende a ser menor
$=$	É equivalente
$\simeq$	Parece equivalente
$\geq$	Tende a ser maior
$>$	É maior

IC	Interpretação de código
TD	Tipagem dinâmica
RC	Reflexão computacional
GM	Gerência automát. de mem.
CS	Componentes de software
AI	Arquitet. da implement.
SL	Sintaxe de Lua
TL	Tabelas de Lua
CL	Comentários de Lua

maior número de informações a ser visualizado. Além disso, as declarações explícitas também tornam a linguagem mais *viscosa* pois as modificações no código geralmente implicam em modificações das declarações. Por outro lado, a tipagem estática tende a diminuir vantagens importantes em modificações exploratórias, como *mecanismos de provisão* e *avaliação progressiva*. Outras alternativas à tipagem dinâmica incluem mecanismos de verificação estática de tipo em linguagens com suporte à tipagem dinâmica (Abadi et al., 1991; Siek e Taha, 2007).

Outro problema particular da implementação do OiL é a *barreira de abstração* imposta pelo uso da biblioteca LOOP na implementação de componentes. Algumas das abstrações definidas pelo LOOP são úteis para permitir implementações de componentes compostos por diferentes segmentos, assim como implementações de funcionalidades como interceptação de portas e introdução dinâmica de portas em múltiplos componentes. Uma possível linha de investigação futura deste trabalho é propor modelos de implementação de componentes mais simples que o definido pelo LOOP.