Uma Avaliação das Arquiteturas para Interoperabilidade entre Objetos

RENATO CERQUEIRA[1]
ROBERTO IERUSALIMSCHY[1]

[1]Departamento de Informática PUC-Rio
Rua Marquês de São Vicente, 225
CEP: 22453-900
Rio de Janeiro - RJ, Brasil
{rcerq,roberto}@inf.puc-rio.br

Abstract

This paper studies some issues related to interoperability of object-oriented languages. First, it discusses the relationship between the development of software components for reuse and the object interoperability issue. After this discussion, it presents some aspects that make difficult this interoperability among objects defined in different languages and also among distributed objects, and the main proposals to solve the interoperability problem. Finaly, it does an analysis of the practical application of the main interoperability frameworks. This analysis has been based on a case study, and has tried to identify the main failures of studied frameworks and their possible evolution.

KEY WORDS: Object-oriented Languages, Object Models, Interoperability, Componentware

1 Introdução

Cada vez mais, é inquestionável a importância de técnicas mais formais e ferramentas para o desenvolvimento de software. Dentro deste quadro, técnicas que dêem suporte à reutilização tornam-se fundamentais para que seja possível otimizar e melhorar a qualidade do processo de desenvolvimento.

Um ideal a ser alcançado pela Engenharia de Software, na linha de reutilização, é a construção de novas aplicações a partir de *componentes de software* já existentes e devidamente testados, como ocorre em outras engenharias [5]. Componentes de software são unidades computacionais com um fim específico que podem ser usadas para compor várias aplicações. De um modo geral, essas unidades encapsulam suas implementações e oferecem um determinado serviço através de um protocolo (interface) bem definido. A técnica de projeto através de componentes de software desloca a ênfase no desenvolvimento de software da decomposição funcional para o reconhecimento de tipos abstratos de dados que encapsulem código e dados.

Deseja-se que, através da técnica de componentes de software, seja possível estender sistemas já existentes através da simples troca ou acréscimo de um novo componente. Para

simplificar esta extensão de aplicações, é importante que o acréscimo ou substituição de um componente não implique na recompilação dos demais componentes da aplicação. É interessante também que haja um suporte adequado por parte do sistema operacional que permita a substituição de um componente de uma aplicação sem que seja necessário que esta passe por um novo processo de *linking* (isto pode ser feito através do uso de bibliotecas dinâmicas) e, até mesmo, que estas alterações possam ser feitas com a aplicação já em execução [23]. Essas características podem simplificar bastante as tarefas de atualização e extensão de uma aplicação.

É interessante observar que as linguagens de programação, e as formas de desenvolver programas, têm evoluído constantemente em direção a um maior suporte para abstração, que se reflete em um maior desacoplamento entre o uso de um tipo e sua implementação. Uma análise de algumas linguagens representativas de suas gerações ilustra esta evolução:

FORTRAN Nenhum suporte a tipos de dados. Qualquer tipo abstrato a nível de projeto é resolvido durante a codificação. A ligação (*binding*) entre uma abstração e sua implementação é feita em tempo de programação.

PASCAL Suporte a tipos, mas com visibilidade total. Como o compilador conhece todos os tipos, a ligação de tipos abstratos é feita durante a compilação.

MODULA 2 Suporte a tipos abstratos. A ligação é feita em tempo de link-edição.

C++ Suporte a orientação a objetos. A concretização de um tipo é feita em tempo de execução, através de *late-binding*. Entretanto, persiste a restrição que a implementação e o uso de um tipo asbtrato devem estar no mesmo processo e linguagem.

Dentro deste quadro, podemos colocar o conceito de componentes de software como uma evolução natural nesta sequência, aumentando o desacoplamento entre uso e implementação de tipos abstratos.

Além de sua aplicabilidade ao desenvolvimento de componentes de software, o paradigma de orientação a objetos também já se mostrava uma ferramenta muito útil para o desenvolvimento de sistemas distribuídos. A idéia de modelar sistemas distribuídos como uma coleção de objetos interagindo é considerada uma arquitetura apropriada para integrar recursos em ambientes heterogêneos e distribuídos [16, 11]. Isto pode ser ilustrado por iniciativas como a de consórcios industriais empenhados na padronização de tecnologias OO aplicadas a sistemas distribuídos, como é o caso da *Open Software Foundation* (OSF) e da OMG [19].

Objetos oferecem um modelo natural para sistemas distribuídos porque os componentes destes sistemas só podem se comunicar usando mensagens através de uma interface bem definida. Assume-se que estes componentes possuem localmente os procedimentos necessários para responder às mensagens por eles recebidas. O uso de objetos se adequa às características de heterogeneidade e desacoplamento de sistemas abertos e distribuídos. Ele atende ao requisito de suporte à heterogeneidade porque as mensagens enviadas a componentes distribuídos dependem somente da interface destes componentes, e não de suas características internas. Já o requisito de desacoplamento é satisfeito porque componentes podem ser alterados de uma forma isolada e transparente, desde que suas interfaces sejam mantidas.

Pode-se observar que os mesmos conceitos do paradigma OO que dão suporte ao desenvolvimento de componentes de software reutilizáveis são também importantes do ponto de

vista de sistemas distribuídos. Na realidade, a capacidade de integrar objetos distribuídos é uma extensão da idéia de reutilização de componentes de software. Ao se aplicar a idéia de componentes de software a sistemas distribuídos, passa-se a ter a possibilidade de se compor aplicações através de componentes distribuídos em um ambiente heterogêneo.

O uso do paradigma OO para integrar componentes independentes e heterogêneos é característico também em desenvolvimentos mais recentes na área de arquiteturas para integração de aplicações em sistemas de computadores pessoais. Um exemplo disto é o *Object Linking and Embedding* (OLE) do sistema Microsoft Windows [24].

Apesar da adequação do paradigma OO para o desenvolvimento de componentes de software reutilizáveis, na prática os resultados obtidos são inferiores ao esperado. Alguns autores até argumentam que o paradigma OO falhou no objetivo de dar suporte ao desenvolvimento de componentes [25].

Este aparente fracasso pode ser atribuído a vários fatores, como pode ser visto em [2, 25, 28]. Entre esses fatores se encontra o baixo grau de interoperabilidade entre componentes de software desenvolvidos com diferentes linguagens OO disponíveis no mercado. Desta forma, componentes desenvolvidos com uma determinada linguagem têm seu escopo limitado a esta linguagem.

Este problema de interoperabilidade ocorre principalmente porque diferentes linguagens OO adotam modelos incompatíveis de objetos. Classes implementadas em uma linguagem não podem ser prontamente usadas por uma outra linguagem. Por exemplo, um programador C++ não pode usar uma classe desenvolvida em Smalltalk, da mesma forma que o contrário também não é possível. Assim, as fronteiras das linguagens orientadas a objetos acabam se transformando em barreiras para a interoperabilidade de componentes de software, o que compromete em muito o desenvolvimento de componentes para reuso.

No caso de algumas linguagens, como por exemplo C++, este problema de escopo é muito mais grave porque não existe um padrão binário para a representação de objetos. Assim, bibliotecas de classes C++ produzidas por um compilador não podem, de um modo geral, ser usadas por programas gerados por um outro compilador C++. Isto limita o reuso destas classes já em sua forma binária, dificultando a distribuição destas na forma de bibliotecas prontas para serem usadas. Apesar deste problema ser circunstancial, já que não há impedimentos reais para a definição de um padrão binário, no caso de C++ ele toma dimensões maiores devido à grande disseminação do uso de C++ comercialmente.

Ironicamente, tais barreiras não existem para as bibliotecas de procedimentos tradicionais. Desenvolvedores constróem com freqüência bibliotecas de procedimentos que podem ser compartilhadas através de uma variedade de linguagens, desde que as linguagens envolvidas sigam convenções de *link*. Estas bibliotecas também têm a vantagem de que suas implementações podem ser modificadas livremente, sem que módulos clientes precisem ser recompilados, o que difere da situação das bibliotecas de classes de C++.

Para desenvolvedores que precisam fornecer bibliotecas de classes em sua forma binária, estes são sérios obstáculos. Soluções amarradas a uma única linguagem não atendem às exigências de sistemas abertos e de redes heterogêneas. Quando se fica amarrado a um compilador específico a situação é mais grave. Certamente, estar amarrado a um compilador ou a uma linguagem para se poder usar uma determinada biblioteca de classes pode ser motivo suficiente para não incluir esta biblioteca em um sistema operacional ou em outro produto de propósito geral.

Outros problemas de interoperabilidade surgem quando se deseja integrar objetos distribuídos. Neste caso, é necessário se integrar objetos em espaços de endereçamento distintos. Mais especificamente, torna-se necessário permitir a comunicação entre objetos em processos diferentes e até mesmo em máquinas diferentes, de uma forma transparente ao desenvolvedor.

Como já foi visto, para prover a integração de objetos distribuídos usa-se um mecanismo de troca de mensagens entre objetos, através de interfaces bem definidas. Existe um crescente consenso, que merece ser destacado, de que as mesmas soluções para o problema de interoperabilidade entre objetos distribuídos podem ser adaptadas para se resolver o problema de integração entre objetos desenvolvidos em linguagens diferentes. A definição da interface de um objeto e o reconhecimento desta por outros componentes é o ponto chave para se tentar solucionar o problema de interoperabilidade.

2 Os Sistemas Propostos para Interoperabilidade de Objetos

Na tentativa de se resolver os problemas de interoperabilidade de objetos, alguns sistemas têm sido propostos; alguns se limitam a propor uma arquitetura [18, 19, 16], outros têm protótipos desenvolvidos em ambientes acadêmicos [4, 14], e outros já têm implementações comerciais [12, 24, 30, 10]. Essas propostas procuram estender as fronteiras de interoperabilidade entre objetos para além dos limites tradicionais impostos pelas linguagens de programação, pelo espaço de endereçamento de processos e, destacadamente, pelas estações de trabalho. A meta é permitir a comunicação entre objetos em ambientes distribuídos e heterogêneos.

Várias situações podem exemplificar as questões que devem ser tratadas ao se tentar resolver o problema de interoperabilidade. Um cenário típico, que levanta várias questões, é referente ao uso de herança de implementação. Por exemplo, suponha uma situação em que uma aplicação sendo desenvolvida em C++ queira usar uma classe X implementada em outra linguagem. A partir desta situação, pode-se questionar o sentido ou a eficiência de definir uma subclasse em C++ herdando da classe X. Uma subclasse de X em C++ teria seus métodos implementados em linguagens diferentes e haveria a necessidade de mapear e representar as variáveis de instância da classe X em C++.

Um outro cenário que serve de exemplo para os problemas de interoperabilidade é o seguinte:

- o objeto A está em um ambiente cujo modelo de objetos[1] suporta coleta de lixo e integridade referencial de referências para objetos, e possui uma referência para o objeto B;
- o objeto B está em um ambiente cujo modelo de objetos suporta a destruição explícita de objetos;
- o objeto B é destruído por um programa em seu ambiente local.

Um erro vai ocorrer na próxima vez que A tentar fazer referência a B, já que este não existe mais. Além disso, esta situação não é suposta como possível no modelo de objetos de

[1] O termo *modelo de objetos* é usado neste texto referindo-se ao conjunto de conceitos usados para descrever objetos em uma determinada linguagem, especificação ou metodologia orientada a objetos.

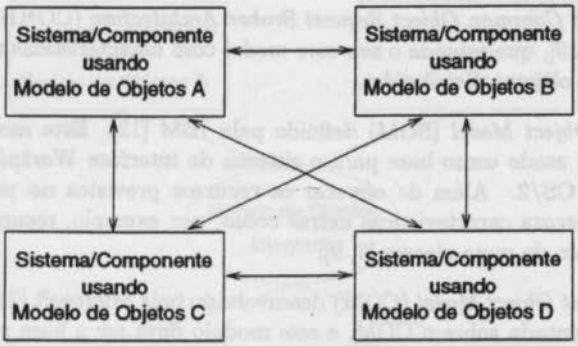

Figura 1: Mapeamento de modelos de objetos sem um modelo integrador.

A. Sendo assim, pode não existir um conceito no modelo de A para tratar este tipo de erro. Estes são apenas alguns dos problemas a serem tratados pelas arquiteturas que se propõem a resolver a questão de interoperabilidade.

De uma maneira geral, todas as arquiteturas propostas para interoperabilidade de objetos procuram definir um *modelo de objetos integrador* que ofereça um conjunto de abstrações suportado por todos os componentes. Mas, para que o esforço empenhado na definição e adoção de um modelo de objetos integrador não seja em vão, é necessário que haja a padronização do modelo e de sua API (*application programming interface*). Esta padronização permitiria, ou ao menos facilitaria, o desenvolvimento independente de componentes de software para ambientes heterogêneos e distribuídos. É com este objetivo que a OMG definiu o padrão CORBA, que já está em sua segunda versão [19].

A figura 1 mostra a arquitetura que existe quando não há um modelo de objetos integrador disponível. Nesta situação, um sistema ou componente adotando um modelo de objetos que precisa usar objetos baseados em outros modelos, requer um mapeamento específico para cada um dos modelos. Em geral, o mapeamento entre N modelos de objetos diferentes requer N(N-1) mapeamentos. Por isto, a tarefa de adicionar um novo modelo é complexa, já que devem ser feitos os mapeamentos do novo modelo para todos os outros.

Já a figura 2 mostra como um modelo de objetos integrador simplifica a arquitetura. Neste caso, tudo que deve ser provido é um mapeamento entre o modelo do componente com o modelo integrador. Sendo assim, o mapeamento entre N modelos diferentes requer somente N mapeamentos. Além do mais, adicionar um modelo novo de objetos requer simplesmente definir um único mapeamento, entre o modelo novo e o integrador.

Vários modelos de objetos têm sido desenvolvidos para servir a várias áreas de aplicação. Alguns destes foram ou estão sendo desenvolvidos explicitamente para servir como modelo integrador. Entre os modelos definidos com este propósito, os seguintes merecem destaque:

- O *Core Object Model*, definido pela OMG, que especifica as características às quais um modelo de objetos deve dar suporte para ser compatível com a arquitetura proposta pela OMG (*Object Management Architecture*) [18]. Este modelo inclui um modelo formal de tipos, operações e subtipagem.

- O modelo do *Common Object Request Broker Architecture* (CORBA), também definido pela OMG [19], que estende o seu *core model* com características mais específicas para o cenário de objetos distribuídos.

- O *System Object Model* (SOM) definido pela IBM [12]. Este modelo segue o padrão CORBA e é usado como base para o sistema de interface *Workplace Shell* do sistema operacional OS/2. Além de oferecer os recursos previstos no padrão CORBA, este modelo apresenta características extras como, por exemplo, recursos de reflexividade[2] através do uso de meta-classes [6, 9].

- O *Component Object Model* (COM) desenvolvido pela Microsoft [24]. A tecnologia OLE está implementada sobre o COM, e este modelo deve ser a base para todos os futuros sistemas da Microsoft. O COM apresenta uma característica muito polêmica que é a ausência do mecanismo de herança [28, 25].

- O modelo *Inter-Language Unification* (ILU) desenvolvido pela Xerox [30]. ILU segue a mesma linha de CORBA, procurando acrescentar algumas melhorias a este.

- O *Reduced Instruction Set Model* (RISC) proposto em [16]. Este modelo é baseado no trabalho do comitê X3H7 da ANSI [17] e propõe a existência de poucos elementos básicos que combinados possam definir características mais sofisticadas do modelo. Para isto, é proposto o uso de mecanismos de reflexividade para compor as características estendidas.

- O *Interoperable Common Object Model* (ICOM) proposto em [4]. Este modelo se restringe a integrar linguagens orientadas a objetos estaticamente tipadas. Baseado nas idéias do modelo RISC, o ICOM apresenta um extenso conjunto de características OO em um *framework* que usa técnicas de reflexividade para dar suporte ao modelo.

As arquiteturas dos modelos integradores podem diferir na forma de representar objetos, na semântica atribuída aos objetos, e em recursos que vão além da noção básica de um objeto como uma unidade computacional independente. Entre os exemplos destes recursos que podem ou não fazer parte de um modelo de objetos incluem-se a possibilidade de definição de atributos na interface de um objeto, grau de separação entre tipo e implementação da classe, herança múltipla, tipos genéricos, modelo de execução, construção de objetos, e mecanismos de reflexividade. Por exemplo, enquanto o SOM oferece reflexividade através de um mecanismo de meta-classe, o COM não oferece nenhum mecanismo de reflexividade. Um outro exemplo é o mecanismo de herança, que está presente tanto em CORBA (somente através de herança de interface) quanto no SOM (que oferece tanto herança de interface quanto de implementação), enquanto o COM não inclui este conceito em seu modelo.

Apesar de todas essas possibilidades de diferenças, basicamente todos os modelos que foram concebidos com a preocupação em relação à questão de interoperabilidade propõem que o que deve ser conhecido de um objeto é somente sua interface, isto é, quais operações são oferecidas por um determinado tipo de objeto. Um objeto que exporte somente a sua

[2] O termo reflexividade é usado neste texto para denotar a habilidade de um modelo representar seus próprios conceitos e efetuar operações que modifiquem o seu próprio comportamento.

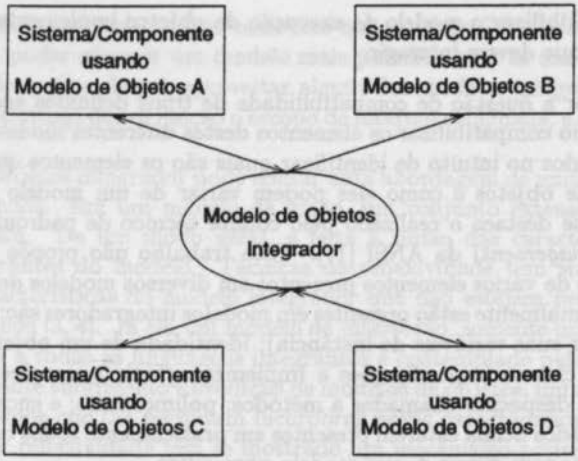

Figura 2: Mapeamento entre modelos de objetos usando um modelo integrador.

interface (o seu tipo) para os seus clientes oferece o grau de desacoplamento necessário para pôr em prática o seu reuso como componente de software, seja este reuso entre componentes implementados com linguagens diferentes ou entre objetos distribuídos. Se não houvesse as diferenças entre modelos de objetos, para dar suporte à interoperabilidade bastaria, simplificadamente, oferecer aos clientes de um componente o tipo deste em uma forma compreensível (ou seja, na linguagem do cliente) e um mecanismo que fizesse a ligação entre cliente e objeto servidor.

Teoricamente, qualquer um desses modelos de objetos poderia servir como base para uma arquitetura integradora de componentes. Entretanto, enquanto o problema de interoperabilidade pode ser resolvido com a padronização e adoção universal de um desses modelos, é possível que nenhum deles venha a alcançar tal sucesso, ainda mais quando se leva em conta a disputa de mercado entre empresas com diferentes propostas [26]. Além disto, a tecnologia de software orientado a objetos é relativamente nova e de rápido desenvolvimento. Isto indica que os modelos de objetos devem continuar a evoluir. Novos modelos podem ser desenvolvidos e outros já existentes podem incorporar novas características. Diante deste quadro com grandes possibilidades de mudanças, é importante que um modelo que se candidate a padrão seja extensível, para poder incorporar novas características sem interferir com software já existente.

3 Alguns Critérios para Avaliação de Modelos Integradores

Como já foi visto, na tentativa de se solucionar os problemas de interoperabilidade entre modelos de objetos diferentes, alguns modelos integradores têm sido propostos com o objetivo específico de permitir a integração entre os demais modelos. Esses modelos integradores devem, basicamente, tratar de dois tipos de problemas:

- como identificar a compatibilidade entre tipos definidos em modelos diferentes e fazer o mapeamento entre eles;

- como compatibilizar o modelo de execução de objetos implementados em arquiteturas diferentes e que devem interagir.

Para se tratar a questão de compatibilidade de tipos definidos em modelos diferentes, torna-se necessário compatibilizar os elementos destes diferentes modelos. Alguns trabalhos têm sido conduzidos no intuito de identificar quais são os elementos que podem fazer parte de um modelo de objetos e como eles podem variar de um modelo para outro. Dentre esses trabalhos, se destaca o realizado pelo comitê técnico de padronização X3H7 (*Object Information Management*) da ANSI [17]. Este trabalho não propõe um modelo mas faz um levantamento de vários elementos presentes em diversos modelos de objetos. Alguns dos conceitos que normalmente estão presentes em modelos integradores são: estado de um objeto (representado por suas variáveis de instância); identidade de um objeto; identificadores de objetos; tipos e classes (especificações e implementações); operações; subtipos; herança; mecanismos para despachar chamadas a métodos; polimorfismo; e encapsulamento. Apesar dos elementos citados acima estarem presentes em praticamente todos os modelos de objetos, suas semânticas podem diferir de um modelo para outro.

A maioria dos modelos integradores tem adotado a solução de usar uma linguagem de definição de interface (*Interface Definition Language* – IDL) para descrever tipos de objetos [19, 30, 24, 4]. As descrições em IDL passam por um compilador que as traduz para adaptadores, chamados *stubs*, em uma linguagem específica. Os *stubs* gerados podem ser ligados com os seus módulos clientes para compor a aplicação final. Uma solução alternativa ao uso de uma IDL é obter as informações referentes ao tipo de um objeto diretamente da sua implementação [1]. Esta última abordagem tem a vantagem de não exigir do desenvolvedor o conhecimento do modelo integrador e de mais uma linguagem, mas exige mais ainda do sistema de interoperabilidade.

Ao se definir um modelo de objetos integrador, vários aspectos devem ser ponderados. Como é bem observado em [3, 29], alguns desses aspectos merecem um destaque especial:

- o conjunto de modelos a serem integrados.
- a maneira como os modelos são integrados.
- o grau de extensibilidade do modelo integrador.
- o grau de flexibilidade e transparência do ponto de vista dos modelos integrados.

Quanto à questão referente ao conjunto de modelos a serem integrados, algumas propostas procuram integrar linguagens de diferentes paradigmas de programação em um modelo de objetos independente de linguagem [19, 27]. Este conjunto inclui tanto linguagens OO quanto linguagens não OO, que podem ser procedurais, funcionais ou lógicas. Esta abordagem as vezes é chamada de *Broad Domain*.

Já outras propostas procuram atender a um conjunto mais reduzido de linguagens. De um modo geral são escolhidas linguagens OO com características semelhantes, por exemplo linguagens OO estaticamente tipadas [3]. Esta abordagem as vezes é chamada de *Narrow Domain*.

A primeira abordagem tem a seu favor o argumento de poder prover a interoperabilidade entre um número maior de modelos, com o custo de impossibilitar ou pelo menos dificultar

em muito o oferecimento de um modelo mais rico em recursos OO. Já a outra abordagem tem a vantagem de poder oferecer um modelo mais parecido com os das linguagens de seu domínio, diminuindo o risco de sub-aproveitar algum dos modelos integrados. No entanto, esta abordagem já restringe por definição o escopo de interoperabilidade, e consequentemente o escopo de reuso.

Um modelo de objetos integrador pode adotar uma abordagem de união ou de interseção dos modelos integrados. Em um modelo de união, um conjunto extenso de recursos OO é incluido no modelo. De um modo geral, é feita a união das características de todas as linguagens integrantes do modelo. Técnicas de reflexividade têm sido sugeridas para implementar as características do modelo integrador que não estejam presentes em alguns dos modelos integrados [3, 4]. Já em um modelo de interseção, somente um subconjunto dos recursos OO comum a todas as linguagens integrantes é contemplado pelo modelo.

Devido ao constante surgimento e evolução de modelos de objetos, um modelo integrador deve prover mecanismos que lhe permitam incorporar novas características ao modelo. Novamente, técnicas de reflexividade têm se mostrado um mecanismo poderoso para permitir a extensão de um modelo integrador [16].

Uma característica importante de um modelo integrador é a sua transparência, tanto para o desenvolvedor de um componente quanto para o desenvolvedor usuário de um componente. Quanto mais transparente for o modelo, menos detalhes deste o desenvolvedor precisará saber e mais fácil será o reuso de componentes já existentes, isto é, componentes que foram desenvolvidos sem se levar em conta o modelo integrador. Sem dúvida, é uma qualidade muito interessante um modelo integrador facilitar o reuso de componentes de software desenvolvidos antes da adoção do modelo.

Entretanto, enquanto simplicidade e transparência são princípios válidos do ponto de vista de alguns desenvolvedores, outros podem precisar de mais flexibilidade ou informações que violem a transparência do modelo. Tal flexibilidade e não-transparência são necessárias para prover o desempenho esperado para determinados serviços. Por exemplo, um desenvolvedor pode querer a flexibilidade para substituir um componente de um serviço já existente ou pode precisar descobrir a localização de serviços equivalentes disponíveis em outras estações em um ambiente distribuído.

As maneiras de oferecer a flexibilidade ou a não-transparência desejada são variadas. Em um modelo que define serviços através de suas interfaces, um serviço pode ser definido de tal forma que permita que informações que violem a transparência do modelo sejam obtidas. Outros modelos permitem que a interface de um objeto seja consultada em tempo de execução. Assim, um cliente pode descobrir os serviços de um objeto dinamicamente e fazer uso desses serviços.

4 Uma Análise dos Modelos Integradores na Prática

Algumas implementações de modelos integradores vêm adquirindo uma posição de destaque cada vez maior, devido à sua crescente disseminação na indústria de desenvolvimento de software. Essas implementações são:

- o modelo do *Common Object Request Broker Architecture* (CORBA), definido pela OMG [19]. Apesar da OMG só definir o padrão, existem várias implementações de CORBA [21]. As principais características deste padrão são:

de programação, não necessariamente orientadas a objetos, se torna interessante em duas situações:

- Quando deseja-se utilizar um componente de software já existente, que foi desenvolvido em uma linguagem diferente da que está sendo utilizada e que, devido a sua complexidade, a re-implementação deste componente na nova linguagem não é uma solução prática. Este é o caso típico de reutilização dos chamados "legados de software".
- Quando deseja-se utilizar a linguagem mais indicada (isto é, a que oferece mais recursos) para resolver um determinado problema, ou a que a equipe de desenvolvimento está mais familiarizada. Neste caso, muitas vezes, deseja-se integrar linguagens de diferentes paradigmas de programação.

No primeiro caso, uma situação que deve ser comum é a de o componente desejado não ter sido desenvolvido para ser usado diretamente pelo modelo integrador a disposição. Já no segundo caso, é interessante poder se usar a linguagem desejada da maneira mais natural possível. Em ambos os casos, quanto menos o modelo integrador interferir no desenvolvimento de um componente e de seus clientes (isto é, quanto mais o modelo for transparente), maior será sua aceitação e maior será o grau de reuso proporcionado por ele. Alguns trabalhos vêm explorando mais a fundo a questão de *adaptadores*, que permitam o uso, através de um modelo integrador, de componetes de software que foram desenvolvidos sem levar em conta este modelo [14].

Outro aspecto relevante é que, de um modo geral, os modelos aqui apresentados defendem, conceitualmente, a importância da separação entre tipo (especificação) e classe (implementação) para facilitar a interoperabilidade entre objetos heterogêneos (característica não apresentada por C++, a linguagem que mais influencia estas arquiteturas). Entretanto, nem sempre esta separação é observada na implementação destes modelos. Por exemplo, SOM amarra uma classe específica à implementação de um determinado tipo, inclusive inserindo informações de implementação nas descrições de interface de IDL. Sobre este aspecto de separação de tipos e classes, o modelo COM é bem rigoroso e, apesar de ser usado geralmente em C++, impõe regras de programação para permitir o uso de sua arquitetura. Estas regras, de um modo geral, restringem o uso de alguns mecanismos de C++ como, por exemplo, herança de implementação. Desta forma, COM consegue prover a separação entre tipos e implementações, mesmo usando C++. Este modelo seria usado mais naturalmente em linguagens como School [22], Sather [20] e Theta [7], que já oferecem a separação entre os conceitos de tipo e classe.

Os sistemas que permitem herança de implementação, como é o caso de SOM e outras implementações do padrão CORBA, são sensíveis ao problema conhecido por *fragile base class problem*. Isto ocorre porque o mecanismo de herança de implementação representa uma quebra do encapsulamento proposto pelo conceito de tipos abstratos de dados. Dos modelos integradores analisados, o único que não é sensível ao problema da classe base frágil é o COM [8].

Um outro aspecto no qual os modelos integradores hoje existentes ainda são deficientes é o referente a mecanismos de extensão do modelo. Como foi colocado na seção 3, a capacidade de extensão é uma característica importante para modelos integradores, para que estes possam atender a novas necessidades. Mas, se há um consenso em relação à importância

desta característica, não ocorre o mesmo quando se trata da questão de como oferecer esta extensibilidade. O paradigma adotado pela OMG para extensão de seu modelo ("*Core + Components*") parece adequado para definir novos modelos de objetos voltados para domínios específicos; entretanto, ele não parece ser tão adequado para a evolução de um determinado modelo, para que este possa atender a novos requisitos. A OMG define que os componentes adicionados ao seu modelo *Core* devem ser compatíveis, mas não especifica como deve ser esta compatibilidade.

Já o modelo SOM oferece um mecanismo de extensão de seu modelo, através do uso de metaclasses. Esta característica representa um avanço dos mecanismos de extensão, além de poder ser considerado o primeiro uso de técnicas de reflexividade para permitir a extensão de modelos integradores. Em [16] é proposto um modelo de objetos que explora o uso de técnicas de reflexividade para a construção e extensão de seu modelo.

Técnicas de reflexividade vêm ganhando uma importância cada vez maior entre os mecanismos de extensão de modelos de objetos. Entretanto, ainda há muitas dúvidas de como deve ser oferecida esta reflexividade, para não tornar o modelo excessivamente complexo para o desenvolvedor e não violar o princípio de transparência do modelo integrador. Um exemplo de experiência bem sucedida do uso de técnicas de reflexividade para a construção de um modelo de objetos, é a linguagem Lua [13]. Nesta linguagem, todo o modelo de objetos foi construído a partir do uso de arrays associativos e de um mecanismo de *fallbacks*.

5 Considerações Finais

Este trabalho apresentou uma breve análise de vários aspectos ligados às arquiteturas para interoperabilidade entre objetos, além de uma avaliação do uso de tais arquiteturas.

Como as arquiteturas analisadas ainda apresentam alguns problemas operacionais e estão trabalhando, basicamente, só com o modelo de objetos de C++, os resultados práticos da aplicação destas tecnologias ainda são muito modestos. Por exemplo, CORBA e SOM ainda estão limitados às linguagens C++ e C para poderem prover interoperabilidade entre objetos. Um outro exemplo é o COM que ainda não oferece recursos para a integração de objetos distribuídos, apesar do seu sucesso dando suporte à tecnologia OLE do sistema MS-Windows. Da mesma forma que ocorre em CORBA e SOM, os objetos que são integrados pela arquitetura COM só têm sido desenvolvidos em C++ ou C.

Todos esses modelos foram pouco avaliados sob o ponto de vista de interoperabilidade entre diferentes modelos de objetos, além de serem muito influenciados pelo modelo de C++. Desta forma, é difícil afirmar que estas arquiteturas apresentam características adequadas para a integração de modelos de objetos diferentes. Na realidade, ainda não há um consenso a respeito de que características são importantes para um modelo integrador. Além disto, pelos problemas apresentados durante o uso das arquiteturas analisadas, pode-se concluir que tais tecnologias ainda precisam de um maior amadurecimento para que possam ser realmente usadas em larga escala.

Entretanto, de toda a análise feita, este trabalho identifica três características básicas que um modelo integrador deve ter, além de um conjunto de conceitos básicos de orientação a objetos bem fundamentado:

- Separação entre tipos e implementações;

- Um alto grau de transparência para o usuário;
- Um poderoso mecanismo de extensão que permita a evolução do modelo.

Algumas linhas de pesquisa podem ser identificadas com o objetivo de aprofundar os estudos sobre a questão de interoperabilidade entre objetos. Uma destas linhas de pesquisa seria o aprofundamento do estudo sobre especificações formais de modelos de objetos. Para isto, os métodos de especificação formal de linguagens orientadas a objetos e sistemas distribuídos poderiam ser revistos, com o objetivo de estudar o potencial desses métodos para modelagem de sistemas multi-linguagens. De posse deste ferramental teórico, é possível o estudo de técnicas mais formais para a integração entre modelos de objetos. Especificações formais de modelos de objetos podem ser de grande utilidade, tanto para identificar mais precisamente os elementos de tais modelos, quanto para facilitar a comparação e mapeamento entre eles. Alguns outros trabalhos [16] também já identificaram a importância da especificação formal de modelos de objetos para o estudo de técnicas de integração desses modelos.

Uma outra linha de pesquisa relevante seria a realização de outros estudos de casos sobre a integração de modelos de objetos às arquiteturas para interoperabilidade, visando permitir uma melhor avaliação da real adequação dessas arquiteturas à integração de componentes de software desenvolvidos em linguagens diferentes. Para isto, seria interessante a escolha de linguagens OO que oferecessem diferenças conceituais relevantes ao modelo de objetos de C++.

6 Agradecimentos

Gostaríamos de agradecer à prof. Noemi Rodriguez por seus frutíferos comentários. Este trabalho foi realizado no âmbito do projeto TeCGraf (PUC-Rio/CENPES-PETROBRÁS), e parcialmente financiado pelo CNPq e pela CAPES.

Referências Bibliográficas

[1] Joshua Auerbach and James Russell. The Concert signature representation: IDL as intermediate language. Em *Proceedings of the Workshop on Interface Definition Languages*, 1994.

[2] Renato Cerqueira. Um estudo sobre interoperabilidade entre linguagens orientadas a objetos. Dissertação de mestrado, Dep. Informática, PUC-Rio, Rio de Janeiro, Brasil, 1996.

[3] Siva Challa. Towards an interoperable, reflective common object model for statically typed object-oriented languages. Em *OOPSLA '94 Workshop on Multi-Language Objects Models*, out 1994.

[4] Siva Challa and Dennis Kafura. Using reflection for implementing ICOM, an interoperable common object model. Technical Report TR-95-23, Dept. Computer Science, Virginia Tech, 1995.

[5] Brad Cox and Andrew Novobilski. *Object-oriented Programming: an evolutionary approach*. Addison Wesley, 1991.

[6] Scott Danforth and Ira Forman. Reflections on metaclass programming in SOM. Em *OOPSLA '94 Conference Proceedings*, out 1994.

[7] M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. Em *Proceedings of OOPSLA' 95*, 1995.

[8] Michael Foody. OLE and COM vs. CORBA. *Unix Review*, 14(4):43–45, 1996.

[9] Ira Forman, Scott Danforth, and Hari Madduri. Composition of before/after metaclasses in SOM. Em *OOPSLA '94 Conference Proceedings*, out 1994.

[10] Dennis Gentry. Distributed applications and NeXT's PDO. *Dr. Dobb's Journal*, 19(16):58–61, 1995.

[11] Lutz Heuser, John Dilley, Hari Madduri, Steven Rabin, and Shawn Woods. Development of distributed and client/server object-oriented applications: Industry solutions. Em *OOPSLA '94 Conference Proceedings*, out 1994.

[12] IBM. *SOMobjects Developer Toolkit: An Overview*, 1993.

[13] R. Ierusalimschy, L. H. Figueiredo, and W. Celes Filho. Lua—an extensible extension language. to appear in *Software: Practice & Experience*.

[14] Dimitri Konstantas. Object oriented interoperability. Em *Proceedings of the Seventh European Conference on Object Oriented Programming ECOOP 93*, jul 1993.

[15] J. Leite, M. Sant'Anna, and F. Freitas. Draco-PUC: A technology assembly for domain oriented software development. Em *Proceedings of the Third International Conference on Software Reuse*. IEEE Computer Society Press, 1994.

[16] Frank Manola. Metaobject protocol concepts for a RISC object model. Technical Report TR-0244-12-93-165, GTE Laboratories Incorporated, dez 1993.

[17] Frank Manola. X3H7 object model features matrix. Technical Report X3H7-93-007v10, Accredited Standards Committee X3, Technical Committee X3H7, fev 1995.

[18] Object Management Group. *Object Management Architecture Guide 2.0*, set 1992.

[19] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, dez 1993.

[20] Stephen Omohundro. The Sather Programming Language. *Dr. Dobb's Journal*, 18(11):42–48, 1993.

[21] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. Wiley, 1996.

[22] N. Rodriguez, R. Ierusalimschy, and J. L. Rangel. Types in School. *ACM SIGPLAN Notices*, 28(8), 1993.

[23] Manfred Stadel. Object oriented programming techniques to replace software components on the fly in a running program. *ACM SIGPLAN Notices*, 26(1):99–108, 1991.

[24] John Toohey and Edward Toupin. *Building OCXs*. QUE, 1995.

[25] J. Udell. Componentware. *Byte*, 19(5):46–56, 1994.

[26] P. Wayner. Objects on the march. *Byte*, 19(1):139–150, 1994.

[27] Sara Williams and Charlie Kindel. The Component Object Model. *Dr. Dobb's Journal*, 19(16):14–22, 1995.

[28] Sara Williams and Charlie Kindel. The problem with implementation inheritance. *Dr. Dobb's Journal*, 19(16):18, 1995.

[29] Shawn Woods. Multi-language object models. Em *OOPSLA '94 Workshop on Multi-Language Objects Models*, out 1994.

[30] Xerox Corporation. *ILU 1.8 Reference Manual*, 1995.