Especificação Formal de Sistemas ODP usando a Linguagem Mondel[1]

Marcos Rogério Salvador
E-mail: salvador@cs.utwente.nl

Centre for Telematics and Information Technology
Faculty of Informatica - University of Twente
P.O. Box 217 - 7500 AE
Enschede, The Netherlands

Wanderley Lopes de Souza
E-mail: desouza@dc.ufscar.br

Grupo de Sistemas Distribuídos e Redes
DC - Universidade Federal de São Carlos
Via Washington Luis, km 235 - 13565-905
São Carlos (SP), Brasil

Abstract: the inability of formal specification languages in covering all ODP framework is recognised and has motivated a number of investigations. Besides its object-oriented nature, the language Mondel also incorporates resources to formally describe distribution and database aspects that are very important for the specification of ODP systems. The suitability of Mondel for ODP is demonstrated through the development of an architectural semantics which is then used as a basis for the specification of the Trading function, in order to illustrate the use of the language in a real situation.

Resumo: a incapacidade das linguagens de especificação formal em cobrir todo o escopo ODP é reconhecida e tem levado a um grande número de investigações. A linguagem Mondel, investigada e apresentada neste artigo, por sua concepção orientada a objeto, seus recursos de distribuição e de banco de dados, representa uma alternativa bastante interessante para a especificação de sistemas ODP. Sua adequabilidade para a descrição da arquitetura ODP é demonstrada através do desenvolvimento de uma semântica arquitetônica na linguagem. Essa semântica é então usada como base no desenvolvimento de um estudo de caso, no qual a função Trading é especificada, para ilustrar a aplicação da linguagem em uma situação real.

1. Introdução

Especificações informais, baseadas em linguagens naturais, são ineficientes para prover definições precisas. A especificação formal de um sistema, usando uma Técnica de Descrição Formal (TDF) [Turner83], visa resolver ou pelo menos minimizar esse problema. Uma TDF é uma linguagem baseada em modelos matemáticos que permite, a partir da especificação (informal) dos requisitos de um sistema, especificá-lo de forma mais clara, precisa e concisa. O emprego de uma TDF permite ainda a validação dessa especificação, a geração (semi) automática da implementação correspondente e a geração (também semi-automática) de seqüências de teste para avaliar a conformidade da implementação em relação à especificação.

Atualmente, três TDFs são padrões internacionais: *Language of Temporal Ordering Specification* (LOTOS) [ISO8807] e *Extended State Transition Language* (Estelle) [ISO9074], padronizadas pela *International Organization for Standardization* (ISO), e *Specification and Description Language* (SDL) [ITU-Z100], padronizada pela *International Telecommunication Union-Telecommunication* (ITU-T; antigo CCITT). Estelle e LOTOS foram concebidas visando principalmente a descrição de serviços e protocolos de comunicação, sobretudo os relativos ao modelo de referência *Open Systems Interconnection* (RM-OSI) [ISO7489], enquanto que SDL foi concebida inicialmente para a descrição dos sistemas de telecomunicação.

Atualmente, essas mesmas TDFs e a linguagem Z[2] [Nich95], desenvolvida visando a engenharia de *sofware* de um modo geral, estão sendo empregadas no modelo de referência *Open Distributed Processing* (RM-ODP) [ISO10746-1, ISO10746-2, ISO10746-3, ISO10746-4], uma evolução natural do RM-OSI, que estende o foco de padronização, até então voltado para questões de

[1] Trabalho realizado com recursos da CAPES e do CNPQ.

[2] Uma TDF é uma linguagem de especificação formal recomendada como um padrão internacional. A linguagem Z ainda está em fase de padronização junto a ISO e, portanto, ainda não é considerada uma TDF.

interconexão, para o comportamento fim-a-fim dos sistemas distribuídos.

Reconhecidamente, nenhuma dessas linguagens é capaz de cobrir todo o escopo ODP, bem mais amplo e complexo que o do OSI. Isto se deve basicamente aos seguintes motivos: o modelo ODP requer o suporte à reconfiguração dinâmica[3], à expressão de propriedades não-funcionais e, principalmente, aos conceitos da orientação a objeto; diferentemente de LOTOS e Estelle, que foram concebidas especialmente para serem empregadas no modelo OSI, nenhuma dessas linguagens foi concebida visando ODP.

Diante deste quadro, o uso e a adequabilidade das linguagens de especificação têm sido bastante investigados nos últimos anos [Bow95, Lini96, Sinn94a, Sinn94b, Sinn95, Vogel94]. Várias abordagens [Fischer93b, Sinn97b] e extensões [ISO97, Carri89] têm sido propostas no sentido de melhorar a expressividade dessas linguagens e ao mesmo tempo aproveitar a experiência adquirida com elas. Da mesma forma, algumas novas linguagens têm sido estudadas.

O objetivo deste trabalho é destacar a adequabilidade da linguagem de especificação Mondel [Boch90b], que foi concebida sob o paradigma da orientação a objeto, na descrição formal de sistemas baseados no RM-ODP. O restante deste artigo está organizado da seguinte forma: a próxima seção define o que é processamento distribuído aberto, introduz o modelo ODP, descrevendo seus principais aspectos, e identifica algumas das limitações das linguagens de especificação no contexto ODP. A seção 3 introduz a linguagem Mondel, apresentando sucintamente algumas das suas principais características, enquanto que a seção 4 descreve como alguns dos principais conceitos ODP podem ser formalizados em Mondel. A seção 5 apresenta um estudo de caso, baseado na formalização desenvolvida na seção 4, no qual a função Trading [ISO13235-1] é especificada, para demonstrar como os recursos da linguagem, aliados a uma boa arquitetura inicial, são importantes para ODP. Finalmente, a seção 6 apresenta algumas conclusões e considerações finais.

2. Processamento Distribuído Aberto

A crescente demanda por informação aliada aos avanços tecnológicos das redes de comunicação e dos computadores pessoais tem levado à interconexão de sistemas de informação espalhados ao redor do mundo. Entretanto, diferenças tecnológicas, administrativas e organizacionais existentes entre esses sistemas ainda impedem um processamento distribuído efetivo.

A fim de prover, de forma transparente, a utilização de serviços distribuídos sobre arquiteturas heterogêneas de *software*, plataformas heterogêneas de *hardware* e ambientes heterogêneos de rede, um grupo formado por especialistas da ISO e do ITU-T vem empregando esforços na elaboração do **Reference Model for Open Distributed Processing (RM-ODP)**, cujo objetivo é definir uma estrutura para coordenar a padronização de ODP.

O RM-ODP está estruturado em quatro partes, sendo que a primeira não é normativa e a última ainda não foi recomendada como um padrão internacional:

- **Part 1: Overview [ISO10746-1]:** apresenta a motivação de ODP, uma visão geral da sua arquitetura e um material explanatório sobre o RM-ODP e sobre como ele deve ser entendido e usado por seus usuários;
- **Part 2: Descriptive Model [ISO10746-2]:** contém a definição dos conceitos requeridos para a especificação de um sistema ODP em um nível de detalhe que serve de suporte à parte 3. Também introduz os princípios de conformidade para sistemas ODP e descreve como eles podem ser usados;

[3] Capacidade de suportar, dinamicamente, alterações em componentes do sistema.

- **Part 3: Prescriptive Model [ISO10746-3]:** identifica as características que qualificam um sistema como distribuído aberto, i.e., como um sistema ODP. Também define como sistemas ODP devem ser especificados, fazendo uso dos conceitos definidos na parte 2;
- **Part 4: Architectural Semantics [ISO10746-4]:** contém a formalização de um subconjunto dos conceitos definidos nas partes 2 e 3. Essa formalização é obtida através do mapeamento desses conceitos nas construções de cada uma das TDFs adotadas, especificamente LOTOS, SDL, Z e Estelle.

O objetivo principal do modelo ODP é definir uma infra-estrutura a partir da qual *distribuição*, *interoperabilidade*, *internetworking*[4] e *portabilidade* possam ser alcançados.

2.1 Fundamentos

Para que os objetivos de ODP pudessem ser alcançados havia a necessidade de uma abordagem que incorporasse abstração, encapsulamento e modularidade, recursos estes oferecidos pelo paradigma da orientação a objetos e que nortearam a sua adoção. Abstração é crucial para lidar com heterogeneidade, pois permite que um dado serviço seja implementado de várias maneiras, usando diferentes tecnologias, e contribui para que a portabilidade e a interoperabilidade sejam alcançadas. Encapsulamento garante que as informações contidas em uma dada entidade sejam acessíveis somente através de interações em interfaces bem definidas. Por fim, a modularidade e a capacidade de compor novos módulos a partir de outros existentes é importante para a construção de sistemas flexíveis e escaláveis e por proporcionar maior produtividade por meio do reuso.

Baseado nesse paradigma, um sistema ODP pode ser caracterizado, de forma bastante breve, como um conjunto de objetos, cada qual encapsulando seu estado, que interagem, através da troca de mensagens, em interfaces bem definidas, restringidos por proibições, permissões e obrigações, impostas por políticas determinadas em contratos (descritos informalmente) que regem a cooperação entre objetos.

A parte 2 do RM-ODP identifica um conjunto de conceitos essenciais para a construção da arquitetura ODP. Esses conceitos são organizados em uma estrutura hierárquica através da qual conceitos mais complexos são derivados de conceitos mais básicos. A seção 4 apresenta alguns desses conceitos.

2.2 Pontos de Vista

Um sistema distribuído não trivial pode envolver uma grande quantidade de informação, tornando a sua especificação, a partir de uma única descrição, uma tarefa impraticável. Para contornar este problema, um conjunto de pontos de vista, equivalente às visões da engenharia de *software*, foi definido no RM-ODP. Um ponto de vista é uma abstração que focaliza partes de interesse particular de um sistema ODP. Cada ponto de vista tem uma linguagem associada, que expressa os conceitos e as regras a serem usados na descrição de um dado sistema. Os seguintes pontos de vista foram definidos:

- **Empresa:** um ponto de vista sobre um sistema e seu ambiente que focaliza o propósito, o escopo e as políticas do sistema.
- **Informação:** um ponto de vista sobre um sistema e seu ambiente que focaliza a semântica da informação e o processamento a ser executado sobre a informação.
- **Computacional:** um ponto de vista sobre um sistema e seu ambiente que focaliza a distribuição através de decomposição funcional do sistema em objetos que interagem nas suas interfaces.
- **Engenharia:** um ponto de vista sobre um sistema e seu ambiente que focaliza os mecanismos e as funções requeridas para suportar interações distribuídas entre os objetos do sistema.

[4] Interações significativas entre sistemas, possivelmente residindo em diferentes domínios (ex. organizacional).

- **Tecnologia:** um ponto de vista sobre um sistema e seu ambiente que focaliza a escolha da tecnologia a ser utilizada para a implementação do sistema.

O uso de cada uma das linguagens dos pontos de vista permite que uma grande e complexa especificação seja separada em partes gerenciáveis, cada uma focalizando questões relevantes a diferentes membros da equipe de desenvolvimento.

2.3 Transparências de Distribuição e Funções ODP

Transparência é o requisito central de ODP, é o que distingue um sistema distribuído aberto de uma rede de computadores. Em um sistema ODP, os detalhes e as diferenças existentes nos mecanismos usados para contornar os problemas causados pela distribuição devem ser abstraídos das aplicações. Para esse fim, o RM-ODP define um número de transparências (ex. de acesso, de migração) e de funções (ex. Trading [ISO13235-1], Type Repository [ISO10389]), assim como também descreve como usá-las. Tanto as transparências quanto as funções são (aparentemente) suficientemente genéricas e podem ser combinadas para atender a uma vasta gama de aplicações (ex. teleconferência, telemedicina, controle de tráfego aéreo).

2.4 Semântica Arquitetônica de ODP

O termo semântica arquitetônica descreve a formalização dos conceitos de uma dada arquitetura, especificados informalmente, em uma linguagem particular. O seu principal objetivo é definir precisamente os conceitos de uma arquitetura e, conseqüentemente, guiar a especificação dos padrões dessa arquitetura.

Idealmente, os trabalhos de desenvolvimento da semântica arquitetônica de ODP, que adotam oficialmente as linguagens LOTOS, SDL, Estelle e Z, deveriam prescrever como cada conceito ODP deveria ser representado em cada uma das linguagens. Contudo, esta abordagem exigiria um consenso a respeito da melhor abordagem para a interpretação desses conceitos, o que na maioria das vezes é impraticável. Assim, esses trabalhos têm adotado uma forma mais descritiva do que prescritiva [Turner97].

Ao passo que esses trabalhos têm propiciado um melhor entendimento da arquitetura ODP bem como a detecção e a eliminação de alguns erros e ambiguidades presentes em alguns dos seus conceitos (ex. objeto), eles também têm revelado uma certa incapacidade não só das linguagens adotadas, mas também do estado da arte da descrição formal no contexto ODP. Especificamente, existe a necessidade para suportar uma interpretação totalmente genérica e não prescritiva da arquitetura ODP, a reconfiguração dinâmica, a expressão de propriedades não funcionais e, principalmente, os conceitos da orientação a objeto. Conceitos básicos da orientação a objeto, como objeto, tipo, subtipo e herança, são difíceis de serem representados pelas TDFs LOTOS, Z, SDL e Estelle [Sinn94a, Sinn95, Sinn97b].

Além disso, nenhuma dessas linguagens é capaz de especificar um único sistema sob todos os pontos de vista[5], dado os diferentes graus de abstração exigidos. Assim, cada linguagem tem sido proposta para um determinado ponto de vista. LOTOS, SDL e Estelle, pela capacidade de encapsulamento e pelas semânticas de interação bem definidas, têm sido indicadas principalmente para os pontos de vista computacional, enquanto que Z, pela capacidade de descrever um sistema em termos de entradas e saídas e suas transformações, tem sido indicada principalmente para o ponto de vista da informação. Embora flexível, essa abordagem exige a verificação de consistência entre as especificações desenvolvidas sob os diversos pontos de vista, possivelmente em diversas linguagens. Estudos neste sentido podem ser encontrados em [Bow96a, Bow96b].

[5] De fato nenhuma linguagem, seja de especificação ou de implementação, disponível atualmente é capaz de suportar tamanha abrangência.

Diante desse quadro, várias pesquisas considerando a utilização das linguagens de especificação em ODP vêm sendo realizadas. O grupo de Sistemas Distribuídos e Redes da UFSCar, em particular, vem investigando a linguagem Mondel, sobretudo sob o ponto de vista computacional, em virtude das suas características. A linguagem Mondel é apresentada a seguir.

3. Linguagem Mondel

Mondel [Boch90] foi desenvolvida dentro de um projeto conjunto de pesquisa, envolvendo o *Centre de Recherche Informatique de Montréal* (CRIM)/*Université de Montréal* (UdeM) e a *Bell Northern Recherche* (BNR), com o propósito de modelar os aspectos operacionais e de gerenciamento das redes de comunicação. Entretanto, suas características tornam-na apropriada para outros tipos de sistemas distribuídos, conforme constatado em [Barb91a, Barb91b, Boch91b, Mond90a, Boch92].

3.1 Características

Mondel é uma linguagem executável que possui uma sintaxe [Boch91a] e uma semântica [Barb90] formalmente definidas. Ela alia o poder de abstração das TDFs baseadas em sistemas de transição (ex. LOTOS) à flexibilidade e expressividade das linguagens de implementação orientadas a objeto. A Tabela 1 [Boch90] apresenta uma comparação entre as características de Mondel e de algumas das linguagens que influenciaram o seu desenvolvimento.

Tabela 1 - Comparação entre Mondel e outras Linguagens

	Ada95	C++	LOTOS	Modula 3	Mondel	Smalltalk
Comunicação	PC	PC	Rv	Co-Rotinas	Rv, PC	PC
Exceção	X	X		X	X	
Semântica Formal			X		X	Ltd
Generalidade	X		Ltd	X	X	
Herança	X	X		X	X	X
Suporte a Objeto	Ltd	X	X	X	X	X
Paralelismo	X		X	Co-Rotinas	Entre/Intra-Objeto	Ltd
Persistência					X	
Legibilidade	Alta	Média	Baixa	Alta	Alta	Média
Tipificação	X	X	X	X	X	

PC = Chamada de Procedimento, Rv = Rendez-vous, Ltd = Limitado

Em Mondel tudo é modelado por objetos, que podem ser classificados em *actor*, *passive* ou *persistent*, de acordo com suas finalidades. Objetos podem ser relacionados de várias maneiras, tais como agregação e herança múltipla. Esta última, também chamada relacionamento "é do tipo" ou "é um", automaticamente estabelece relações de subtipo/supertipo que podem ser verificadas dinamicamente.

Uma especificação Mondel consiste de um conjunto de objetos que interagem invocando operações através de um mecanismo *Rendez-vous* (comunicação síncrona), possivelmente restringidos por guardas. A ordem na qual essas interações podem ocorrer, essencial em sistemas distribuídos, pode ser definida de forma paralela, exclusiva, seqüencial e ainda baseada em autômatos.

No intuito de suportar as chamadas transações, provenientes da área de bancos de dados, objetos persistentes podem ter operações qualificadas como atômicas. Este recurso permite que o sistema volte ao estado imediatamente anterior ao início da execução de uma transação[6], caso uma exceção ocorra. Visando também a integridade dos objetos persistentes e, conseqüentemente, da especificação, a ordem na qual transações atômicas podem ocorrer é controlada automaticamente em Mondel, tal que elas sejam executadas uma após a outra.

Mondel propõe uma metodologia de desenvolvimento [Mond90b] para guiar o projeto de um

[6] Recuperação automática apenas no nível mais alto, não para subações.

sistema, levando da sua descrição informal a uma especificação executável, que pode ser validada pelas ferramentas da linguagem. Para auxiliar essas atividades, um Interpretador e um Verificador são fornecidos pela linguagem.

Atualmente a linguagem Mondel tem sido investigada buscando prover: um mapeamento total da sintaxe ASN.1 [ISO8824] e o suporte à reconfiguração dinâmica de especificações [Erra92], pela UdeM, e a geração (semi) automática de implementações nas linguagens C++ [Boch90], também pela UdeM, e Java [Branco98], pelo GSDR/DC/UFSCar.

As características da linguagem Mondel, descritas sucintamente nesta seção, demonstram a sua potencialidade para a descrição de sistemas distribuídos de um modo geral. Entretanto, para que se possa afirmar que Mondel é capaz de descrever adequadamente um sistema ODP é necessário demonstrar como a arquitetura ODP pode ser representada na linguagem. A próxima seção aborda esta questão.

4. Semântica Arquitetônica de ODP em Mondel

O desenvolvimento de uma semântica arquitetônica requer uma análise meticulosa da arquitetura a ser formalizada e das estruturas da linguagem a ser utilizada e envolve, basicamente, dois tipos de atividades: análise conceitual, baseada na descrição informal da arquitetura, e a especificação formal na linguagem.

A definição dos principais conceitos ODP bem como a interpretação destes em Mondel são apresentadas a seguir:

- **Objeto:** um modelo de uma entidade que contém dados e oferece serviços. Todo objeto é distinto e é caracterizado pelo seu comportamento e pelo seu estado.

Um objeto é representado em Mondel pela instanciação de um *template* (erroneamente chamado de classe em algumas literaturas), denominado tipo na terminologia Mondel. Objetos podem ser qualificados, de acordo com suas finalidades, em *actor*, *passive* ou *persistent*. Objetos do tipo *actor* são usados para representar objetos do mundo real, que normalmente contém algum comportamento (ex. pessoa, computador). Objetos do tipo *passive*, em contrapartida, são usados para representar objetos que não contém comportamento definido explicitamente[7] (ex. estruturas de dados). Por fim, objetos do tipo *persistent* são usados para representar objetos que devem persistir mesmo após o término de uma transação (ex. registro de banco de dados) e devem ser destruídos explicitamente (ex. object!dispose). A Figura 1 ilustra um *template* (genérico) de objeto em Mondel.

```
type ObjectTemplate = <ObjectType> with
    {definição dos atributos, que podem ser privados ou visíveis}
operation
    {definição das operações disponíveis a outros objetos, i.e., da sua interface}
behavior
    {definição do comportamento, que inclui a aceitação das operações definidas na sua interface}
where
    {definição dos procedimentos internos usados no comportamento do objeto}
endtype ObjectTemplate
```

Figura 1 - Template de Objeto Genérico

Todo objeto, ao ser instanciado (ex. define ref = new ObjectTemplate), recebe automaticamente uma identidade (ref), única e exclusiva, através da qual ele pode ser acessado por outros objetos.

[7] Na realidade todo objeto contém um comportamento, como para sua destruição (ex. self!dispose) ou para ter seus atributos acessados por outros objetos (ex. x:=object.attribute). De fato, uma simples leitura do valor de um atributo é realizado por uma operação *read*.

Essa identidade permite que o objeto associado seja passado como parâmetro em chamadas de operações remotas.

Todo objeto é caracterizado pelo seu próprio estado, que é encapsulado, independente de ter sido instanciado do mesmo *template* ou não, e pelo seu comportamento, que é executado independentemente das outras instâncias. O comportamento de um objeto pode exibir características seqüências, indeterminísticas, especificadas através de declarações *choice* (ex. choice x or y .. end;), paralelas, descritas através de declarações *parallel* (ex. parallel x and y ... end;), ou ainda de forma determinística baseada em autômatos, definidos por meio de procedimentos. A Figura 2 ilustra esta última forma.

procedure ativo = *accept* falha *do* *return*; *end*; inativo; *endproc* ativo	*procedure* inativo = *accept* ok *do* *return*; *end*; ativo; *endproc* inativo

Figura 2 - Máquina de Estados Finita em Mondel

Cada estado, e as interações (e o comportamento) possíveis nesse estado, é mapeado para um procedimento, sendo que as transições de estado ocorrem através de chamadas entre esses procedimentos.

- **Ação:** representa qualquer coisa que aconteça e é sempre associada a pelo menos um objeto. Uma ação pode ser uma interação ou uma ação interna.

- **Ação interna**: qualquer ação que ocorra sem a participação do ambiente do objeto, i.e., sem a participação dos outros objetos do sistema.

Qualquer instrução definida no comportamento de um objeto que, quando executada, não resulte na invocação de uma operação em outro objeto representa uma ação interna em Mondel (ex. x:=y^). O encapsulamento do objeto assegura que a ocorrência dessa ação interna não seja notada pelo seu ambiente.

- **Interação**: qualquer ação que ocorra com a participação do ambiente do objeto.

Qualquer instrução definida no comportamento de um objeto que, quando executada, resulte na invocação de uma operação em outro objeto e na sua conseqüente aceitação. Isto é, para que uma interação ocorra de fato é necessário que o objeto invocado esteja pronto para aceitar a invocação[8], i.e., que ambos os objetos estejam sincronizados. Além disso, interações estão sujeitas a permissões e proibições, determinadas por guardas[9]. Um cenário ilustrando uma invocação (Figura 3a) e a sua conseqüente aceitação (Figura 3b) é apresentado a seguir.

type Client = actor with private s:Server; behavior	type Server = actor with operation register(reference:Client);

[8] Um objeto está pronto para aceitar uma invocação quando o ponto de execução de seu comportamento está posicionado na declaração *accept* da operação invocada.

[9] Guardas são definidos em interações através de cláusulas *provided* associadas a declarações *accept* e expressam condições para a aceitação de invocações.

```
⇒ s.register(self);
endtype Client
```

Figura 3a - Template do Objeto Cliente

```
behavior
    ...
⇒ accept register provided <condição> do ... end;
    ...
endtype Server
```

Figura 3b - Template do Objeto Servidor

- **Interface:** uma abstração do comportamento de um objeto, que consiste de um subconjunto das suas (possíveis) interações junto com um conjunto de restrições sobre quando elas podem ocorrer. Um objeto pode conter múltiplas interfaces em ODP, cada uma representando uma parte da sua funcionalidade, incluindo várias instâncias da mesma interface (*template* de interface).

Mondel não é capaz de representar múltiplas interfaces por objeto de forma direta e elegante, como Estelle o faz, por exemplo. Em Mondel, um objeto tem apenas uma interface, identificada pela referência desse objeto. O suporte a múltiplas interfaces por objeto, em linguagens incapazes de faze-lo diretamente, tem despertado bastante interesse da comunidade da computação distribuída e tem resultado em várias propostas, sobretudo no âmbito da *Common Object Request Broker Architecture* (CORBA) [OMG96-03-04], definida pelo *Object Management Group* (OMG). De um modo geral, essas propostas (ex. [OMG97-05-17]) giram em torno da modelagem de interfaces através de objetos (denominados objetos de interface, que são instâncias de *templates* de interface, seguindo a terminologia ODP) distintos que, de alguma forma, se relacionam com o objeto que de fato implementa o comportamento das operações identificadas nessas interfaces, i.e., com o objeto servidor. A Figura 4 apresenta uma arquitetura genérica que ilustra essa idéia.

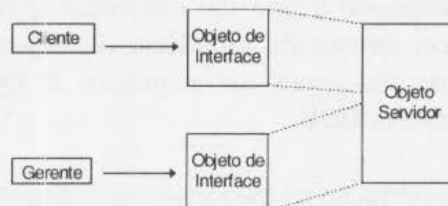

Figura 4 - Múltiplas Interfaces por Objeto

A modelagem adotada para Mondel baseia-se nesse mesmo princípio. Várias configurações são possíveis para esse fim. A adotada neste trabalho, e que (aparentemente) parece ser a mais apropriada, é apresentada a seguir, através de um exemplo.

Um objeto servidor oferece duas interfaces, denominadas A e B respectivamente. A interface A oferece a operação A1 e a interface B oferece a operação B1. As Figuras 5a e 5b apresentam as assinaturas dessas interfaces.

```
type InterfaceA_Signature = actor with
operation
    operationA1(...);
endtype InterfaceA_Signature
```

Figura 5a - Assinatura da InterfaceA

```
type InterfaceB_Signature = actor with
operation
    operationB1(...);
endtype InterfaceA_Signature
```

Figura 5b - Assinatura da InterfaceB

A definição dessas assinaturas separadamente não é necessária, mas contribui para a modularidade das especificações além de estar em conformidade com o ponto de vista computacional do modelo ODP. As operações definidas para as interfaces A e B são herdadas pelos objetos (de interface) que desempenharão de fato essas interfaces. As Figuras 6a e 6b apresentam os *templates* dessas interfaces.

```
type InterfaceTemplateA = InterfaceA_Signature with
private server:Server;
behavior
```

```
type InterfaceTemplateB = InterfaceB_Signature with
private server:Server;
behavior
```

```
    accept operationA1 do
        return server!operationA1;
    end;
endtype InterfaceTemplateA
```

Figura 6a - Template da InterfaceA

```
    accept operationB1 do
        return server!operationB1;
    end
endtype InterfaceTemplateB
```

Figura 6b - Template da InterfaceB

Cada interface define um atributo apontando para o objeto que implementa de fato o comportamento das operações nela identificadas. Objetos clientes invocam as operações nos objetos de interface que, então, repassam a chamada invocando as operações (possivelmente com um nome diferente para evitar invocações diretas) correspondentes nos objetos que implementam o comportamento dessas operações. O *template* do objeto servidor é apresentado na Figura 7.

```
type ServerTemplate = InterfaceA_Signature and InterfaceB_Signature with
visible private interfaceA:InterfaceA;
          interfaceB:InterfaceB;
behavior
    define interfaceA = new InterfaceA(self);    {self representa a referência do servidor}
    define interfaceB = new InterfaceB(self);
    parallel
        accept operationA1 do ... end;
    and accept operationA2 do ... end;
    end;
endtype ServerTemplate
```

Figura 7 - Template do Objeto Servidor

As principais vantagens desta configuração são: (1) o objeto servidor é capaz de oferecer várias instâncias de um mesmo tipo ou de diferentes tipos de interfaces sob demanda[10], cada uma definindo um contexto particular para cada cliente; (2) a refência do objeto de interface pode ser usada para se verificar dinamicamente o tipo da interface; (3) interfaces podem ser passadas como parâmetros em interações, através da referência do objeto de interface; (4) permite a descrição de alguns aspectos não funcionais (ex. requisitos de QoS), por meio de atributos, e comportamentais, como requerido em ODP.

- **Ponto de Interação:** Uma localização na qual existe um conjunto de interfaces.

Mondel não é capaz de representar este conceito precisamente, visto que cada objeto suporta apenas uma interface diretamente. No caso de Mondel, um ponto de interação descreve uma localização na qual existe apenas uma interface, conhecida pela sua referência (ex. define server = new Server;). Toda e qualquer interação só é possível se o objeto cliente souber a localização do ponto de interação do objeto servidor, ou seja, se o objeto cliente tiver a referência da interface do objeto servidor (ex. server!register;).

- **Papel:** um identificador de um comportamento, que pode aparecer como um parâmetro em um *template* para um objeto composto, i.e., pode estar associado com um dos objetos componentes do objeto composto.

Mondel não é capaz de modelar um papel de forma tão elegante quanto Estelle, por exemplo. Entretanto, um papel pode ser representado através de duas abordagens: através de um parâmetro em uma interação (ex. trader!export(exporter, ...);), de tal forma que o objeto invocado possa verificar se o papel informado dá permissão para a execução da operação desejada (ex. accept export provided role = exporter do ... end;); através de atributos identificando os papéis dos objetos envolvidos em um relacionamento (ex. agregação). A Figura 8 ilustra esta última abordagem[11].

[10] Para esse fim, algumas alterações devem ser feitas na definição de tipo Server (ex. definir operações que permitam aos clientes descobrirem instâncias de interfaces, tal que a cada invocação de uma dessas operações, uma nova interface seja criada).

[11] Esta abordagem também pode ser usada, conforme o grau de abstração desejado, para descrever objetos binding, i.e., intermediadores

```
type Relationship = persistent with
    consumer:User;
    producer:VideoServer;
endtype Relationship
```

Figura 8 - Possíveis papéis em um Relacionamento

- **Tipo:** um predicado caracterizando uma coleção de <X>s, onde um <X> pode ser um objeto, uma interface ou uma ação. Diferentes <X>s podem satisfazer o mesmo tipo e um único <X> pode satisfazer a vários tipos.

A noção de tipo em ODP é bastante genérica e, por isso, não é representada fielmente por nenhuma linguagem existente, incluindo Mondel. O que é possível em Mondel é a tipificação de objetos e interfaces (não ações), baseado em seus *templates*, o que na realidade constitui um *tipo de template* na terminologia ODP.

Tipos de objetos e interfaces são normalmente verificados estaticamente pelo compilador da linguagem, mas também podem ser verificados dinamicamente. Como na maioria das linguagens, seja de especificação ou de implementação, essa verificação é feita com base nos nomes dos *templates* e não na sua estrutura. Embora seja extremamente simples e economize tempo e recursos de processamento, esta abordagem traz desvantagens no contexto ODP em termos de compatibilidade comportamental[12], alcançada por meio de relações de subtipo/supertipo.

- **Subtipo/Supertipo:** um tipo A é um subtipo de um tipo B, e B é um supertipo de A, se todo <X> que satisfaz A também satisfaz B. Em Mondel, um tipo B é compatível com um tipo A quando A é um ancestral imediato de B ou quando existe uma série de tipos descritos numa relação de herança que leve de B até A. Ou seja, subtipificação e compatibilidade são conceitos idênticos na atual versão da linguagem. A Figura 9 ilustra como a verificação de tipos e de relações de subtipo/supertipo pode ser feita em Mondel.

```
case objectC of
        type ObjectTemplateA => writeln "ObjectTemplateC is subtype of ObjectTemplateA";
        type ObjectTemplateB => writeln "ObjectTemplateC is subtype of ObjectTemplateB";
        type ObjectTemplateC => writeln "ObjectTemplateC is of the type ObjectTemplateA";
end;
```

Figura 9 - Verificação Dinâmica de Tipos e Subtipos

O grande problema com essa abordagem é que relações hierárquicas podem ser detectadas apenas quando existe uma herança, e relações de herança só podem ser estabelecidas estaticamente. Dado o potencial de evolução e modificação de ODP, seria mais apropriado uma abordagem baseada na verificação de predicados, que não se utilizasse de hierarquias definidas estaticamente, em detrimento do tempo e consumo de recursos necessários para esse fim.

- **Composição:** uma combinação de dois ou mais objetos produzindo um novo objeto, em um novo nível de abstração. As características do novo objeto são determinadas pelos objetos que foram combinados e pelo modo como eles foram combinados. O comportamento de um objeto composto é a correspondente composição dos comportamentos dos objetos combinados.

Composição de objetos pode ser realizada em Mondel através de herança e agregação. No nível mais alto, uma especificação (que também é um objeto) pode ser considerada como uma composição (por agregação) formada por um conjunto de instâncias de definições de tipos, i.e.,

---

entre clientes e servidores em uma interação.

[12] Propriedade que permite, baseado em subtipificação de interfaces, determinar quando um dado serviço pode ser substituído por outro, sem que o ambiente perceba essa troca.

por um conjunto de objetos.

Se uma composição for realizada através de agregação, então o comportamento do objeto como um todo consistirá do comportamento dele mesmo e dos comportamentos descritos em cada um dos objetos componentes. Como os objetos envolvidos na agregação são processados independentemente uns dos outros, pode-se dizer que o comportamento de um objeto agregado é composto por um conjunto de comportamentos autônomos que são processados em paralelo visando o mesmo objetivo. Por exemplo, um carro é composto de um motor e de um câmbio. Tanto o motor quanto o câmbio têm seus próprios estados e comportamentos, entretanto, os dois são processados separada e simultaneamente para dar suporte ao carro. A Figura 10 ilustra uma composição (por agregação) de objetos.

```
type Carro = actor with
  motor:Motor;
  cambio:Cambio;
behavior
  define motor = new Motor(self);
  define cambio = new Cambio(self);
  ...
endtype Carro
```

Figura 10 - Composição de Objetos e Comportamentos

Se a herança for usada, então o objeto que herdou os outros objetos conterá suas próprias características, obviamente, em conjunto com as características dos objetos herdados. O seu comportamento, por sua vez, consistirá de uma composição paralela, através de uma declaração *parallel*, contendo o seu próprio comportamento e o comportamento dos objetos herdados.

Uma vez composto, um objeto pode ser decomposto apenas quando a composição foi alcançada através de agregação, expressa por meio de atributos. Por exemplo, para decompor o objeto carro, basta fazer acesso aos atributos motor e câmbio (ex. define motor = carro.motor;).

- **Refinamento:** O processo de transformar uma dada especificação em uma outra mais detalhada, sendo que uma especificação e seus refinamentos normalmente não coexistem numa mesma descrição.

Mondel permite o refinamento de uma especificação tanto numa mesma descrição quanto em diferentes descrições. O refinamento sucessivo dos componentes de uma especificação, numa mesma descrição, é possível através da herança. A Figura 11 ilustra este tipo de refinamento.

```
type DUABindHandler1 = actor
endtype DUABindHandler1

type DUABindHandler2 = DUABindHandler1 with          {primeiro refinamento}
operation
    Bind(...):DirectoryBindResult;
    Unbind(...);
endtype DUABindHandler2

type DUABindHandler3 = DUABindHandler2 with          {segundo refinamento}
behavior
    choice
        accept Bind do ... end;
        or accept Unbind do ... end;
    end;
endtype DUABindHandler3
```

Figura 11 - Refinamento

O refinamento de comportamentos não é possível. No projeto de Mondel foi incluída uma

cláusula *refined* para permitir o refinamento de atributos e comportamentos herdados. Entretanto, este recurso não está implementado na atual versão da linguagem. O mesmo acontece com o refinamento de especificações como um todo (ex. unit RefinedUnit predefined BaseUnit =), em diferentes descrições. Embora prevista no projeto da linguagem, esta característica ainda não foi implementada.

O refinamento de uma especificação, em diferentes descrições, é possível através da decomposição de objetos. Por exemplo, numa dada descrição um carro pode ser especificado através de um único *template*, denominado Carro. Numa outra descrição, esse mesmo *template* pode ser refinado em (outros) três diferentes *templates*: Carro, Motor e Câmbio.

- **Pré-condição:** um predicado que deve ser satisfeito para que uma ação ocorra.

Em Mondel, uma pré-condição pode ser expressa através de uma declaração *if then* associada a um comportamento (ex. if ok then ...end;), e das declarações *ifexist* e *forall* com cláusulas *suchthat* associadas (ex. ifexist x:OfferInfo suchthat offerDB.contains(x) then ... end;). Uma pré-condição para a aceitação de uma invocação pode ser realizada através da especificação de uma cláusula *provided* associada a uma declaração *accept* (ex. accept export provided authentication(caller) do ... end;).

Uma pré-condição também pode ser descrita quando um estilo orientado a estado[13] é usado. Neste caso, a pré-condição define que o objeto tem que estar em um determinado estado para que uma dada interação possa ocorrer.

- **Política:** um conjunto de regras relacionadas a um propósito particular, onde cada regra pode ser expressa em termos de uma obrigação, permissão ou proibição.

Políticas são expressas implicitamente em uma especificação Mondel, i.e., diretamente no comportamento dos objetos, por meio de permissões e proibições.

- **Obrigação:** uma prescrição exigindo que um comportamento particular ocorra.

Não existem meios para a especificação de obrigações em Mondel. Um comportamento ocorre ou não dependendo das interações do objeto com o ambiente.

- **Permissão:** uma prescrição permitindo que um comportamento particular ocorra.

Declarações *accept* prontas para receber uma invocação caracterizam uma permissão. Uma declaração *accept* está pronta para receber uma invocação quando ela não está executando nenhuma outra invocação. Pré-condições avaliadas como verdadeiras também caraterizam permissões.

- **Proibição:** uma prescrição proibindo que um comportamento particular ocorra.

Uma proibição é o caso inverso de uma permissão. Um comportamento correspondente a uma declaração *accept* em execução caracteriza uma proibição para novas invocações, i.e., novas invocações sobre a mesma declaração estão proibidas de ocorrer até que o comportamento em execução seja terminado. Pré-condições avaliadas como falsas também caracterizam proibições.

[13] Um estilo de especificação é, simplificadamente, uma abordagem específica, baseada em certos recursos, usada para se descrever aspectos particulares de um sistema. LOTOS, por exemplo, provê um estilo orientado a restrição que pode ser usado para restringir o acesso a recursos de um sistema.

Uma vez que as estruturas básicas, a partir da qual outras podem ser construídas, foram desenvolvidas (e verificadas!), é esperado que o processo de desenvolvimento de especificações seja aliviado, pois muitas das decisões normalmente tomadas durante o projeto das aplicações foram tomadas antecipadamente.

5. Especificação Computacional da Função Trading em Mondel

Um importante componente da infra-estrutura ODP, a função Trading [ISO13235-1], desempenhada por um objeto Trader, oferece meios para que, dinamicamente, objetos possam oferecer, ou exportar, serviços e possam descobrir, ou importar, serviços que tenham sido oferecidos.

Especificações formais dessa função foram desenvolvidas em LOTOS [Sinn97a], SDL [ISO13235-H], Object-Z [Dong93] e mais recentemente em *Enhancements to LOTOS* (E-LOTOS) [ISO97]. Um estudo comparativo também foi realizado em [Fischer93a] entre LOTOS, SDL e Z, baseado na especificação do Trader sob alguns pontos de vista. A especificação a ser apresentada nesta seção é uma síntese da que foi desenvolvida em [Salv97], onde um Trader com suporte a federação e a atividades de administração foi especificado e validado, e destaca principalmente os aspectos arquitetônicos.

5.1 Definição do Domínio

Seguindo a metodologia proposta para Mondel, o primeiro passo a ser realizado em uma especificação é a definição do domínio, visando capturar os elementos relevantes do domínio da aplicação junto com as suas características essenciais.

O elemento fundamental em ODP é o serviço. Um serviço expressa um capacidade ou função existente em um objeto e é divulgado através de uma oferta de serviço (OfferInfo), que, além do próprio serviço, descreve a localização da interface na qual ele pode ser acessado. Todo serviço deve ser obrigatoriamente de um tipo conhecido pelo Trader[14], conhecimento este (que pode ser) realizado por um repositório de tipos (TypeRepository). Um tipo de serviço (TypeStruct) descreve a assinatura da interface em que o serviço se encontra e algumas propriedades não-funcionais (PropStruct) que podem ser atribuídas ao serviço (ex. custo do uso do serviço). A Figura 12 apresenta o modelo da informação desenvolvido em [Salv97], usando a notação OMT [Rumb91].

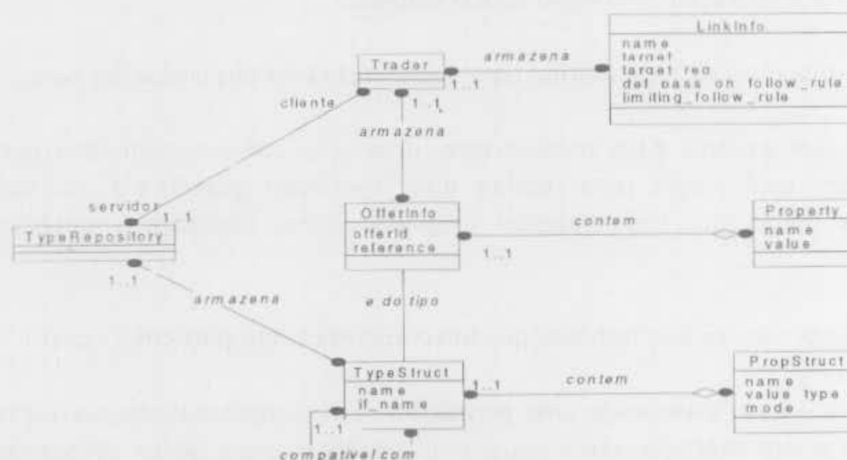

Figura 12 - Modelo de Objetos da Função Trading

Durante a modelagem dessas informações buscou-se um modelo com granularidade fina que

[14] Serviços podem ser considerados instâncias de tipos de serviço.

abrangesse todos os tipos de Trader e aliasse a modularidade e o reuso, propiciando escalabilidade e fácil manutenção.

Embora mencionado que ofertas de serviço são armazenadas no Trader, isso não acontece de fato na nossa abordagem. Na realidade, as ofertas são automaticamente armazenadas no banco de dados (orientado a objetos) da linguagem (não do Trader) ao serem criadas (e identificadas exclusivamente) pelo Trader, tornando-se visíveis a ele por meio das referências das suas interfaces, i.e., dos seus pontos de interação.

5.2 Definição das Operações

Esta etapa visa a identificação das operações necessárias para prover a funcionalidade esperada da aplicação e a alocação dessas operações nos objetos identificados na etapa anterior.

Em ODP, a funcionalidade de um objeto é definida em suas interfaces. No caso do Trader, sua funcionalidade é descrita por quatro interfaces funcionais[15]: Lookup (importação), Register (Exportação), Admin (Administração) e Link (Federação). Essas interfaces são, por sua vez, suportadas por quatro interfaces abstratas (contendo apenas atributos). Conforme descrito na seção 4, interfaces são representadas por objetos em Mondel. A Figura 13 apresenta o modelo de objeto do Trader (e suas interfaces).

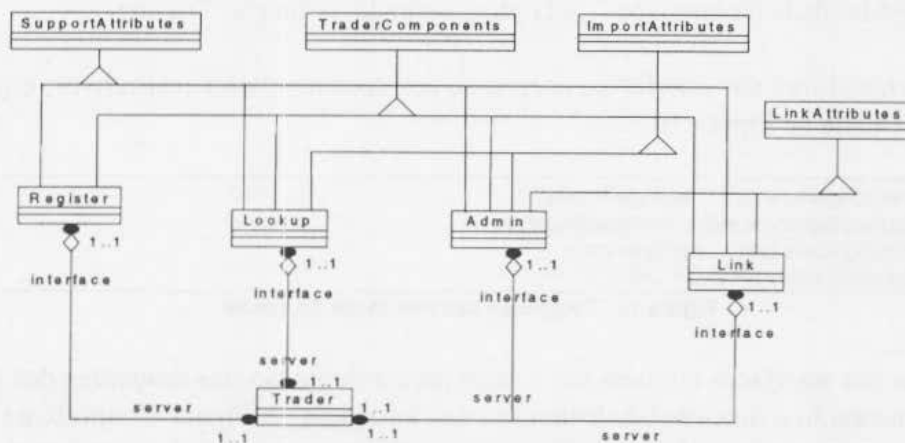

Figura 13 - Modelo de Objetos do Trader

Embora a função Trading permita que várias instâncias das interfaces Lookup e Register sejam criadas, adotou-se, por questões de simplicidade, um mapeamento um para um, ou seja, cada Trader pode ter apenas uma instância de cada uma dessas interfaces (bem como das outras).

A Figura 14 apresenta a assinatura da interface de importação, que contém apenas uma operação, *query*, que é usada para que importadores possam descobrir os serviços que eles precisam.

```
type LookupSignature = actor with
  query(requester:Importer:    service_type:string;    constr:string;    pref:string;    policies:set[Policy];
desired_props:set[string]):set[QueryResult] pure;
endtype LookupSignature
```

Figura 14 - Assinatura da Interface de Importação

Outro aspecto fundamental na nossa abordagem é o fato do processo de busca propriamente dito não ser realizado pelo Trader mas sim por cada oferta simultaneamente, propiciando melhor desempenho. Para esse fim, toda oferta oferece uma operação, denominada *evaluate*, que é herdada do *template* ServiceOfferEvaluator e é invocada pelo Trader (ou por qualquer objeto que

[15] Um Trader pode ser configurado em um número de interfaces de acordo com a funcionalidade desejada.

conheça o ponto de interação da oferta[16]).

A interface Register fornece meios para que exportadores possam oferecer serviços a outros objetos, através das seguintes operações: *export*, para oferecer serviços; *withdraw*, para retirar serviços; *describe*, para descrever serviços; *modify*, para alterar algumas propriedades de serviço. A Figura 15 apresenta a assinatura dessa interface.

```
type RegisterSignature = actor with
operation
  export(reference:actor; service_type:string; properties:set[Property]):integer;
  withdraw(id:integer);
  describe(id:integer):Offer pure;
  modify(id:integer; del_list:set[string]; modify_list:set[Property]) atomic;
endtype RegisterService
```

Figura 15 - Assinatura da Interface de Exportação

Dessas operações mereceu atenção especial a operação *modify*, que efetua alterações sobre uma ou mais propriedades das ofertas e, por questões de integridade, foi qualificada como atômica. Como de conhecimento, os atributos de um dado objeto podem ser alterados apenas por ele mesmo. Portanto, esse objeto deve oferecer operações apropriadas para esse fim. Neste sentido, toda instância de propriedade de serviço (que é associada a uma dada oferta) oferece a operação *set_value*, que é herdada da interface CosTrading definida na função Trading.

A partir das assinaturas das interfaces, chegou-se aos seus *templates* respectivos, cuja estrutura básica é apresentada na Figura 16.

```
type Lookup = LookupSignature with ... endtype Lookup
type Register = RegisterSignature with ... endtype Register
type Admin = AdminSignature with ... endtype Admin
type Link = LinkSignature with ... endtype Link
```

Figura 16 - Templates das Interfaces do Trader

As assinaturas das interfaces também são a base para a definição dos *templates* dos objetos que de fato implementarão a funcionalidade descrita nas interfaces. A Figura 17 apresenta a estrutura básica dos diversos *templates* de objeto Trader que podem ser instanciados, conforme o padrão.

```
type SimpleTrader = LookupSignature and RegisterSignature and ImportAttributes and SupportAttributes and
TraderComponents with ... endtype SimpleTrader
type StandAloneTrader = SimpleTrader and AdminSignature and LinkAttributes with ... endtype StandAloneTrader
type LinkedTrader = StandAloneTrader and LinkSignature with ... endtype LinkedTrader
```

Figura 17 - Tipos de Templates para o Objeto Trader

Note que foi adotada uma estruturação hierárquica em árvore para modelar os *templates* do Trader. Isso foi necessário para contornar os conflitos de nomes de atributos e de operações que apareciam caso uma herança múltipla fosse utilizada indiscriminadamente.

5.3 Definição dos Comportamentos

Esta última etapa consiste da definição do comportamento dos vários objetos identificados nas etapas anteriores. O primeiro passo desta etapa é a identificação da ordem na qual as operações identificadas na etapa anterior podem ser atendidas pelo *Trader*. Seguindo os tradicionais métodos para a descrição de seqüências de interações e adotando-se os recursos apropriados da linguagem, foi possível definir todas as seqüências possíveis de interações. Por exemplo, as operações *query* e *describe* podem ser executadas em paralelo, pois elas não causam nenhuma alteração no estado do Trader[17]. Por outro lado, as outras operações da interface Register alteram

[16] A função Trading explicita que o uso do Trader não é obrigatório, apesar das reconhecidas vantagens do seu uso.

[17] Operações que não alteram o estado do seu objeto são qualificadas como *pure;* transações são qualificadas como *atomic*.

o estado do Trader e, portanto, não pode ser executadas simultaneamente. A Figura 18 apresenta essa estruturação em um *template* de um Trader básico, com suporte a atividades de importação e de exportação apenas.

```
type SimpleTrader = RegisterSignature and LookupSignature and SupportAttributes and ImportAttributes with
behavior
  choice
    parallel
        loop accept query do ... end; end;
    and loop accept describe do ... end; end;
    end;
  or loop accept export do ... end; end;
  or loop accept withdraw do ... end; end;
  or loop accept modify do ... end; end;
  end;
endtype SimpleTrader
```

Figura 18 - Especificação de um Trader com suporte a Importação e Exportação

Para evitar que os objetos tentassem interagir diretamente com a interface do Trader propriamente dita, um controle de acesso, baseado no tipo do objeto que está interagindo, foi definido. Um Trader só aceitará uma invocação *query*, por exemplo, se ela for originária da interface Lookup. A Figura 19 ilustra a aceitação da operação *query*, que tem associada uma pré-condição desempenhando essa restrição.

```
loop
  accept query provided security_obj.authentication(requester) = "Lookup" do ... end;
end;
```

Figura 19 - Aceitação da Operação Query

A Figura 20 descreve de maneira breve o comportamento da operação *query*. Basicamente, uma importação apresenta o seguinte comportamento: primeiro, o Trader verifica se o tipo de serviço desejado é conhecido {1a}, gerando uma exceção apropriada em caso negativo {1b}; em seguida, as ofertas que são do tipo desejado[18] são pesquisadas {2} e avaliadas (por elas próprias) {3}; por último, as ofertas selecionadas são retornadas ao importador {4}.

```
define service_types = type_repos.list_types;
if service_types.contains(service_type) then                                              {1a}
  forall x:OfferInfo suchthat offerDB.contains(x) and x.service_type = service_type do     {2}
    if x.evaluate(constr) then offers!add(new Offer(x.reference, properties_aux)); end;    {3}
  end;
else raise new UnknownServiceType(service_type); end;                                     {1b}
return offers;                                                                            {4}
```

Figura 20 - Processo de Seleção de Ofertas em Importações

Conforme mencionado anteriormente, ofertas de serviço são de fato armazenadas no banco da dados da linguagem, que oferece duas declarações, *ifexit* e *forall*, para a recuperação de uma ou mais instâncias de um dado tipo, respectivamente. Contudo, no caso de uma federação, onde existem dois ou mais Traders, somente essas declarações não seriam suficientes, pois não seria possível determinar a que Trader cada oferta pertenceria. Assim, cada Trader mantém uma lista privada (ex. offerDB) contendo as interfaces das suas ofertas. Baseado em pré-condições apoiadas sobre essa lista é possível armazenar, recuperar e destruir as ofertas apropriadas.

Propriedades de serviço podem conter valores de vários tipos (de acordo com a sua definição em PropStruct), o que traz problemas durante o processo de avaliação de ofertas em importações (e durante o processo de aceitação e de alteração de ofertas, conseqüentemente), pois Mondel é uma linguagem fortemente tipificada e não suporta polimorfismo diretamente. Para contornar essa

[18] Ofertas nas quais seu tipo sejam um subtipo do tipo desejado também podem ser selecionadas, visto que elas também apresentam as mesmas características (compatibilidade comportamental).

limitação, o tipo genérico *object* foi usado para receber os valores das propriedades. Assim, verificando-se o seu tipo específico (e o tipo permitido descrito na definição do tipo de serviço associado), é possível realizar o processamento apropriado para cada tipo de valor.

O padrão da função Trading define uma grande quantidade de exceções para descrever possíveis erros no sistema. O suporte ao tratamento de exceções da linguagem permitiu não só a expressão dessas situações de erro de forma direta mas também de forma clara, podendo ser facilmente identificada tanto pelo interpretador quanto por inspeção. Uma vez gerada uma exceção, o ponto de execução é interrompido e retornado ao ponto imediatamente seguinte à instrução que a gerou (no nível mais alto), podendo então ser tratada eficientemente baseada no seu tipo. Isso é possível porque exceções são modeladas por objetos (derivados do tipo *exception*) na realidade e, desta forma, podem ter seu tipo verificado dinamicamente.

5.4 Validação

Conforme descrito inicialmente, a especificação formal de um sistema é a base para uma implementação eficaz. A especificação da função Trading desenvolvida em [Salv97], cuja síntese foi apresentada nas seções anteriores, foi validada através de várias atividades de simulação, contra a sua especificação de requisitos e de conformidade, mostrando-se correta e em conformidade com o seu padrão.

Além do ponto de vista da correção, a validação permitiu uma melhor avaliação das ferramentas afins da linguagem que, de um modo geral, mostraram-se capazes de atingir seus objetivos. Entretanto, falta um ambiente de janela para facilitar as atividades de simulação, bastante intensas no caso da especificação Mondel da função Trading. Também falta o suporte para toda a sintaxe da linguagem nas atividades de verificação. Por essa razão, não possível verificar a presença de *deadlocks*, de ponteiros inválidos e outros na especificação desenvolvida.

6. Conclusões

A incapacidade das linguagens de especificação, incluindo LOTOS, SDL, Estelle e Z, no contexto ODP é reconhecida e tem sido bastante investigada. Várias abordagens e extensões têm sido propostas visando ODP, destacando-se a extensão E-LOTOS, que representa de fato uma evolução no estado da arte da descrição formal. Contudo, E-LOTOS ainda apresenta uma sintaxe e uma semântica instáveis e, por essa razão, também não oferece nenhuma ferramenta para auxiliar o processo de desenvolvimento de especificações. Além disso, certas características da orientação a objeto ainda não foram atacadas (ex. herança). Ou seja, ainda levará algum tempo para que E-LOTOS possa ser usada na prática, o que recai no problema da falta de expressividade das linguagens adotadas oficialmente em ODP.

Em função desses problemas, a linguagem Mondel foi investigada e apresentada neste artigo. Primeiramente, uma semântica arquitetônica foi desenvolvida, trabalho esse que permitiu não só que a expressividade da linguagem perante a arquitetura ODP fosse analisada mas também que algumas comparações com outras linguagens fossem realizadas. Comparada a LOTOS, por exemplo, a mais indicada dentre as linguagens adotadas oficialmente em ODP para o ponto de vista computacional, Mondel apresentou as seguintes vantagens: a legibilidade do código Mondel é bem superior à de LOTOS; os conceitos ODP são descritos de forma mais direta e clara em Mondel e em maior número; Mondel favorece o reuso por meio da herança[19], que por sua vez, não é suportada em LOTOS; objetos podem ser passados livremente como parâmetros em invocações, o que não ocorre em LOTOS, pois instâncias de processos, usadas para representar objetos, não são tratadas como entidades de primeira classe; Mondel suporta a verificação

[19] A validade da herança em sistemas distribuídos, ou simplesmente herança distribuída, tem gerado bastante controvérsia em vista dos seus prós e contras, sendo que nenhuma implementação foi desenvolvida até o momento. No nível de especificação este recurso é bastante útil, uma vez que um sistema distribuído é descrito de forma monolítica.

dinâmica de relações subtipo/supertipo entre objetos, embora baseada em nomeação ao invés de em predicados.

A partir dessa semântica, um estudo de caso baseado na função Trading foi desenvolvido, ilustrando a importância de uma arquitetura inicial formalizada e ao mesmo tempo realçando algumas características da linguagem, tais como persistência e tratamento de exceções. Graças à persistência de objetos e seus recursos de armazenamento e recuperação, a função Trading, que é nada mais do que um repositório de dados com funções específicas, pôde ser especificada de forma bastante rápida, sem a necessidade de qualquer mecanismo de banco de dados específico. O tratamento de exceções, por sua vez, permitiu a descrição de várias situações de erro especificadas no padrão da função Trading diretamente.

Comparada novamente a LOTOS, desta vez sob um escopo mais genérico, Mondel apresentou as seguintes desvantagens: não é muito conhecida, embora seja de aprendizado bastante fácil; não conta com muita experiência adquirida e possui poucas ferramentas. LOTOS, por sua vez, conta com uma vasta literatura e uma grande quantidade de ferramentas que atendem a várias situações. Mais comparações, não somente voltadas para ODP, entre Mondel e outras linguagens formais, podem ser encontradas [Barb91a, Boch90a, Vogel94].

Mesmo com as desvantagens citadas acima, acreditamos que Mondel ainda represente uma alternativa bastante interessante para ODP, em vista das poucas alterações que (aparentemente) precisariam ser feitas em sua semântica para permitir uma maior e melhor cobertura a ODP, o que, certamente, constitui um grande problema (ex. LOTOS).

Acreditamos que este trabalho contribua de três maneiras: a primeira pela disseminação da linguagem Mondel, com sua concepção orientada a objeto; a segunda pela semântica arquitetônica desenvolvida, que propicia uma visão mais precisa de como cada conceito pode ser representado na linguagem, atuando como uma biblioteca de componentes que podem ser combinados para formar especificações; a terceira e última pela especificação da função Trading, talvez o componente mais importante da arquitetura ODP, que tem sido usada como referência nas comparações entre especificações realizadas em diferentes linguagens e, por esse motivo, permitiu e ainda permitirá outras investigações.

Como etapas futuras, este trabalho pode ser aperfeiçoado no sentido de melhorar e expandir a semântica desenvolvida (ex. incluir mais e melhores exemplos, modelar interações de fluxo e interações *multi-party*, para descrever sistemas multimídia). Além disso, tão logo a capacidade de reconfiguração dinâmica seja incorporada, o mapeamento Mondel/ASN.1 seja finalizado e o tradutor Mondel/Java seja construído, várias outras direções poderão ser vislumbradas.

Referências Bibliográficas

[Barb90] Barbeau M., Bochmann G. v. *Formal Semantics of Mondel.* Technical Report,. CRIM/BNR Project, CRIM, Canada, 1990.

[Barb91a] Barbeau M., Saqui-Sannes P. de, Bochmann G. v. *Design, Formal Specification and Validation of Centralized and Distributed Control in a Transmission System.* Technical Report, CRIM/BNR Project, CRIM, Canada, 1991.

[Barb91b] Barbeau M. *Object-Oriented Specifications in OSI and Distributed Processing.* Technical Report, D'IRO, Université de Montréal, Canada, 1991.

[Boch90a] Bochmann G. v. *System Specification with Mondel and relation with other formalism.* Technical Report, D'IRO, Université de Montréal, Canada, March 1990.

[Boch90b] Bochmann G. v., Barbeau M., Erradi M., Lecomte L., Mondain-Monval P., Willians N. *Mondel: An Object-Oriented Specification Language.* Technical Report, D'IRO, Université de Montréal, Canada, 1990.

[Boch91a] Bochmann G. v., Barbeau M., Bean A., Erradi M., Lecomte L., Liu A. *The Description of the Specification Language Mondel V1*. Technical Report, CRIM/BNR Project, CRIM, Canada, April 1991.

[Boch91b] Bochmann G. v., Lecomte L., Mondain-Monval P. *Formal Description of Network Management Issues*. Proc. Int. Symposium on Integrated Network Management, North Holland, Arlington, US, April 1991.

[Boch92] Bochmann G. v., Poirier S., Mondain-Monval P. *Object-Oriented Design for Distributed Systems: The OSI Directory Example*. Anais do IFIP Int. Conference on Upper Layer Protocols, Architectures and Applications, Vancouver, 1992.

[Bow95] Bowman H., Derrick J., Linington P, Steen M. *FDTs for ODP*. Computer Standards and Interfaces, September 1995.

[Bow96a] Bowman H., Boiten E., Derrick J., Steen M. *Viewpoint consistency in ODP, a general interpretation*. First IFIP Int. Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS), Chapman & Hall, March 1996.

[Bow96b] Bowman H., Derrick J., Linington P., Steen M. *Cross viewpoint consistency in Open Distributed Processing*. Software Engineering Journal, January 1996.

[Branco98] Branco L. H. C., Prado A. F. do, Souza W. L. de. *Utilização do Paradigma Draco para implementar especificações Mondel na linguagem Java*. (em preparação).

[Carri89] Carrington D. A., Duke D., Duke R., King P., Rose G. A., Smith G. *Object-Z: an Object-Oriented Extension to Z*. Proc. of the Formal Techniques (FORTE)´89.

[Dong93] Dong J. S., Duke R. *An Object-Oriented Approach to the Formal Specification of ODP Trader*. Proc. of the Int. Conference on Open Distributed Processing (ICODP), Berlin, Germany, September 1993.

[Erra92] Erradi M., Bochmann G. v., Hamid I. A. *Type Evolution in a Reflective Object-Oriented Language*. TR, D'IRO, Université de Montréal, Canada, 1992.

[Fischer93a] Fischer J., Prinz A., Vogel A. *Different FDT's confronted with different ODP-viewpoints of the Trader*. Proc. of the Formal Methods Europe (FME), 1993.

[Fischer93b] Fischer J., Vogel A. *Towards a formal semantics of the ODP-Reference Model*. Proc. of the Int. Conference on Open Distributed Processing (ICODP), Berlin, Germany, September 1993.

[ISO7489] ITU-T Rec. X.200 | ISO/IEC IS 7498-1. *Information Technology - Open Systems Interconnection - Basic Reference Model*. 1994.

[ISO8807] ISO/IEC 8807. *Information Processing Systems - Open System Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. 1987.

[ISO8824] ISO/IEC 8824. *Information Technology - Open System Interconnection - Specification of Abstract Syntax Notation One (ASN.1)*. 1990.

[ISO9074] ISO/IEC 9074. *Information Processing Systems - Open System Interconnection - Estelle - A Formal Description Technique Based on an Extended State Transition Model*. 1989.

[ISO10389] ISO/IEC JTC1/SC21 N 10389. *Open Distributed Processing - Type Repository Function*. 1996.

[ISO10746-1] ITU-T Draft Rec. X.901 | ISO/IEC DIS 10746-1. *Reference Model for Open Distributed Processing - Part 1: Overview*. 1995.

[ISO10746-2] ITU-T Rec. X.902 | ISO/IEC IS 10746-2. *Reference Model for Open Distributed Processing - Part 2: Foundations*. 1995.

[ISO10746-3] ITU-T Rec. X.903 | ISO/IEC IS 10746-3. *Reference Model for Open Distributed Processing - Part 3: Architecture*. 1995.

[ISO10746-4] ITU-T Draft Rec. X.904 | ISO/IEC DIS 10746-4. *Reference Model for Open Distributed Processing - Part 4: Architectural Semantics*. 1995.

[ISO13235-1] ITU-T Draft Rec. X.9tr | ISO/IEC DIS2 13235. *ODP Trading Function - Part 1: Specification*. Document approved for standardization. Jan 1997.

[ISO13235-H] ITU-T Draft Rec. 9tr | ISO/IEC CD 13235 (ANNEX H). *A Sample Specification of a*

Concrete Trader Template and its Application using SDL/ASN.1. 1994.

[ISO97] ISO/IEC JTC1/SC21/WG7. *Enhancements to LOTOS*. Revised Working Drafts on Enhancements to LOTOS (V4), January 1997.

[ITU-Z100] ITU-T Rec. Z.100. *Specification and Description Language SDL*. International Telecommunication Union, Geneva, 1992.

[Lini96] Linington P.F., Derrick J., Bowman H. *The specification and conformance of ODP systems*. Int. Workshop on Testing of Communicating Systems, Darmstadt, Germany, IFIP TC6/WG6.1, September 1996.

[Mond90a] Mondain-Monval P., Bochmann G. v. *An Object-Oriented Software Architecture for the OSI Basic Reference Model*. TR, D'IRO, Univ. Montréal, Canada, 1990.

[Mond90b] Mondain-Monval P., Bochmann G. v. *An Object-Oriented Software Design Methodology*. TR, D'IRO, Université of Montréal, Canada, 1990.

[Nich95] Nicholls J. *Z Notation*. Version 1.2, The University of Oxford, September 1995.

[OMG96-03-04] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Revision 2.0, Document 96-03-04, Framingham, MA, 1995.

[OMG97-05-17] Object Management Group. *Response for Multiples Interfaces and Composition RFP*. Document ORBOS 97-05-17, June 2, 1997.

[Rumb91] Rumbaugh J. I. *Object-Oriented Modeling and Design*. Prentice-Hall, Inc, 1991.

[Salv97] Salvador M. R., Souza W. L. de. *Especificação Formal de Objetos do Modelo ODP em Mondel*. Dissertação de Mestrado, Programa de Pós-Graduação em Ciências da Computação, Universidade Federal de São Carlos, Agosto, 1997.

[Sinn94a] Sinnott R. O. *The Development of an Architectural Semantics for ODP*. Technical Report, Department of Computing Science and Mathematics, University of Stirling, Scotland, March 1994.

[Sinn94b] Sinnott R. O., Turner K. J. *Modelling ODP viewpoints*. Proc. of the Int. Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), Portland, Oregon, USA, October 1994.

[Sinn95] Sinnott R. O., Turner K. J. *Applying Formal methods to Standard Development: The Open Distributed Processing Experience*. Computer Standards and Interfaces, October 1995.

[Sinn97a] Sinnott R. O., Turner K. J. *Applying the Architectural Semantics of ODP to Develop a Trader Specification*. Computer Networks and ISDN Systems, March 1997.

[Sinn97b] Sinnott R. O. *An Architecture Based Approach to Specifying Distributed Systems in LOTOS and Z*. PhD Thesis. Department of Computing Science and Mathematics, University of Stirling, Scotland, UK, April 1997.

[Turner83] Turner K. J. *Using Formal Description Techniques - An Introduction to ESTELLE, LOTOS and SDL*. John Wiley, New York, January 1993.

[Turner97] Turner K. J. *Relating Architecture and Specification*. Computer Networks and ISDN Systems, March 1997.

[Vogel94] Vogel A. *A Formal Approach to an Architecture for Open Distributed Processing*. TR, D'IRO, Univ. de Montréal, Canada, 1994.