3
Apresentação do Problema e da Solução Proposta

3.1.
Contextualização do Problema

Com base na discussão realizada sobre *matchmaking* no capítulo anterior, pode-se perceber que sua aplicação é bastante recorrente em diversos domínios. Além disso, é notável sua importância na resolução de diversos problemas em várias áreas importantes de pesquisa acadêmica e da indústria em geral, tais como sistemas multi-agentes, *Web* semântica [Fensel, 2003] e descoberta de serviços.

Porém, o que se percebe é que os diversos grupos de pesquisa, que realizam trabalhos relacionados a *matchmaking*, sempre criam sua própria arquitetura de *software*. Assim, muitas soluções apresentadas definem pontos que correspondem a inovações para a área, mas muitos componentes dessas arquiteturas apresentam muitas funcionalidades semelhantes, que poderiam ser especificadas e desenvolvidas como componentes de *software* reusáveis.

O grande problema desse cenário é que muito antes dos pesquisadores se aterem a resolverem um problema de *matching* em um dado domínio, que é o objetivo de seus trabalhos, eles precisam construir toda uma infra-estrutura de suporte para o processo de *matchmaking*. Pode-se citar como exemplo a necessidade de componentes que permitam a comunicação entre o *matchmaker* e outros sistemas (sistemas especialistas, agentes de *software*, dentre outros), o acesso às informações sobre as instâncias do domínio, bem como de algoritmos eficientes e que produzam resultados de *matching* com alto grau de precisão.

Tomando essa situação, o problema que se deseja atacar nesse trabalho de mestrado é o de definir uma infra-estrutura flexível e extensível para *matchmaking*, que disponibilize para seus usuários componentes reusáveis que visam a atender os problemas enumerados anteriormente, dentre outros. A conseqüência imediata do uso de tal solução é que o pesquisador ou o

desenvolvedor que a estiver utilizando poderá focar seus esforços apenas no desenvolvimento do algoritmo de *matching* específico do domínio.

Pode-se perceber que, devido à variedade dos domínios de aplicação de soluções de *matchmaking*, definir uma infra-estrutura genérica e flexível não é uma tarefa tão simples. Os itens abaixo enumeram alguns requisitos que devem ser atendidos por uma infra-estrutura com tais características:

- *Independência de domínio de aplicação*: Esse, aparentemente, é o requisito mais básico a ser atendido. Devido ao fato de *matchmaking* ser utilizado em vários domínios de aplicação, uma infra-estrutura genérica para *matchmaking* também deve ser independente com relação ao domínio sendo utilizado;
- *Independência do modelo de dados utilizado*: Não se sabe num primeiro momento qual modelo de dados foi utilizado para descrever as instâncias de um dado domínio. Dessa forma, a infra-estrutura deve ser flexível para acomodar diferentes modelos de descrição, tais como banco de dados, arquivos XML ou ontologias;
- *Flexibilidade e Extensibilidade na definição de estratégias de matching*: A infra-estrutura deve permitir a definição e a introdução de vários algoritmos para *matching* entre instâncias de um dado domínio. Também deve ser facilitada a definição de estratégias de *matching* que visam a dar eficiência ao processo, possibilitando, por exemplo, que não seja necessária a consulta a todas as instâncias do domínio para que um resultado de *matching* possa ser obtido;
- *Possibilitar que o processo de matching possa ser configurável*: A infra-estrutura deve permitir que a entidade que está requerendo o *matching* possa definir parâmetros de configuração do processo. Um exemplo de parâmetro seria a escolha do algoritmo de *matching* a ser utilizado ou o valor de corte a ser utilizado durante o processo de comparação entre as instâncias do domínio;
- *A infra-estrutura deve garantir interoperabilidade*: A infra-estrutura deve garantir a qualquer entidade, independente de plataforma ou de implementação, acesso aos serviços disponibilizados para a execução do processo de *matching*;

- *Simplicidade*: Com relação à simplicidade, podem-se definir dois objetivos. O primeiro corresponde à simplicidade na definição dos componentes da infra-estrutura, responsáveis pela execução do *matching*. O segundo diz respeito à simplicidade de uso desses componentes por uma aplicação que requisita a execução de um processo de *matching*. É fato que, normalmente, as soluções relacionadas a *matchmaking* tendem a serem complexas e custosas computacionalmente. Dessa forma, criar componentes para resolver tais problemas pode não ser uma tarefa simples. Porém, deve-se garantir que uma vez definidos, seja simples a utilização dos mesmos por uma aplicação que acesse os serviços de *matchmaking* disponibilizados pela infra-estrutura.

Como dito anteriormente, esse trabalho de mestrado também corresponde ao resultado de um projeto entre o Laboratório de Engenharia de Software (LES) e o Instituto Fraunhofer FIRST da Alemanha. Além das caracteríticas definidas anteriormente, esse projeto possui uma demanda particular relacionada ao uso de linguagens de anotação semântica como base para a descrição das instâncias do domínio sobre o qual as soluções de *matching* serão aplicadas. Assim, a infra-estrutura deve garantir acesso a modelos de dados definidos através de ontologias, mais especificamente aquelas descritas por meio das liguagens RDF, RDFS e OWL.

3.2. Apresentação da Solução Proposta

Levando em consideração o problema descrito na seção anterior, bem como os requisitos que foram definidos, o objetivo desse trabalho de mestrado consiste em desenvolver uma infra-estrutura baseada em *software* para *matchmaking*. A Figura 3 apresenta uma visão geral da arquitetura da infra-estrutura proposta. Segue abaixo uma rápida discussão acerca dos seus componentes:

- *Application*: Aplicação construída para atender às necessidades de um dado domínio de aplicação que utiliza a infra-estrutura proposta para resolver problemas de *matching* nesse domínio;

- *Matching Module Framework*: Um *framework* que visa a resolver problemas de *matching* entre indivíduos (ofertas e demandas) de um dado domínio de aplicação;
- *OntoAPI*: Uma API para acesso a dados anotados semanticamente, cuja linguagem de descrição utilizada seja RDF, RDFS ou OWL.

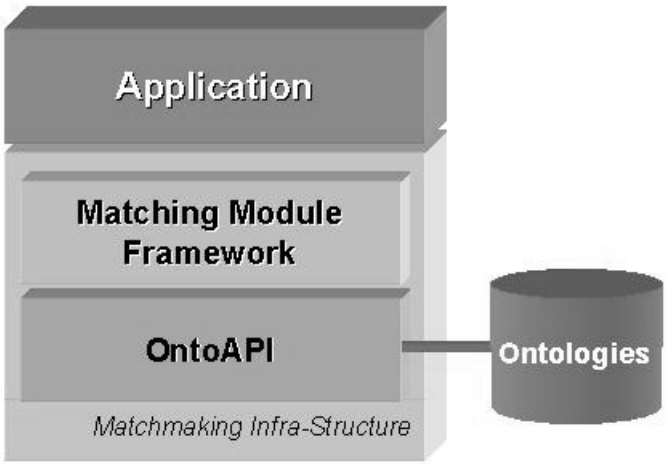

Figura 3 – Arquitetura da infra-estrutura proposta

Como pode ser visto, a arquitetura apresenta dois componentes que possuem propósitos muito bem definidos. O *framework* Matching Module visa a garantir à infra-estrutura vários requisitos relacionados ao processo de *matchmaking*. Ele disponibiliza vários pontos de flexibilização relacionados ao domínio de conhecimento, ao modelo de dados e à utilização de estratégias de *matching*.

Além disso, o *framework* disponibiliza uma camada de serviços, que visa a facilitar a comunicação entre as suas instâncias e as aplicações que as utilizam para resolver problemas de *matching*. Outra característica relacionada aos serviços, mais intrinsecamente àqueles relacionados ao *matching*, é que suas requisições são feitas através de documentos XML, que descrevem vários parâmetros que podem ser configurados pela aplicação que requer a execução do processo de *matchmaking*.

A OntoAPI, adicionalmente, resolve o problema relacionado ao acesso a ontologias. Seu objetivo é disponibilizar um conjunto de serviços que encapsulam grande parte do esforço que as aplicações precisam dispender quando tratam com dados semânticos. Ela também disponibiliza uma série de

mecanismos que permitem, dentre outras coisas, a localização de ontologias, de forma transparente à aplicação que a estiver utilizando, e o *cache* de informações descritas nas mesmas, garantindo maior eficiência nas consultas realizadas através da API.

Como o *framework* Matching Module disponibiliza um ponto de flexibilização relacionado ao modelo de dados sendo utilizado para descrever as instâncias de um dado domínio, a OntoAPI foi o componente de *software* utilizado como base para a implementação desse *hot spot* do *framework*. Dessa forma, suas instâncias poderão processar dados anotados semanticamente através dos serviços disponibilizados pela API, resolvendo, assim, o requisito particular do projeto com o Instituto FIRST, relacionado ao uso de ontologias pela infra-estrutura.

As próximas seções apresentam uma discussão mais detalhada sobre o *framework* Matching Module e a OntoAPI. Nelas serão apresentadas as principais características desses dois módulos de *software*, buscando elucidar como eles atendem os requisitos levantados para a infra-estrutura que está sendo proposta.

3.3. O Framework Matching Module

O *framework* Matching Module visa a resolver problemas de *matching* entre indivíduos, ofertas e demandas, de um dado domínio de aplicação. A Figura 4 apresenta uma visão simplificada da arquitetura do *framework*, em que são definidos os componentes que fazem parte de uma única instância.

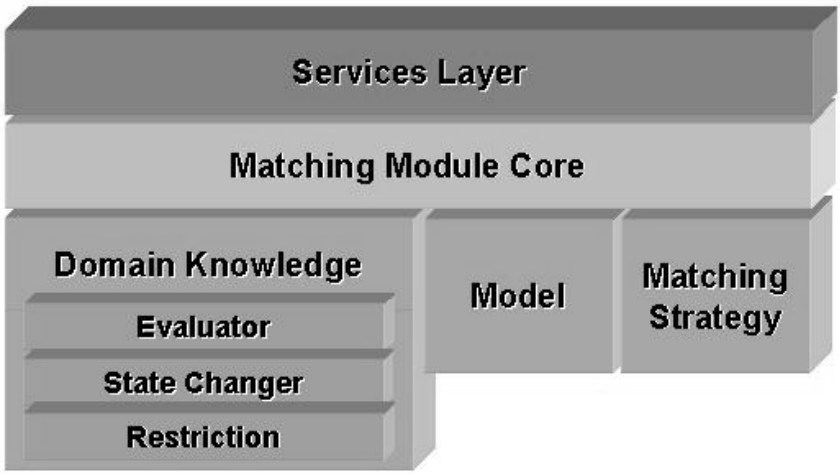

Figura 4 – Arquitetura simplificada do Framework Matching Module

O componente *Matching Module Core* corresponde à parte não flexível do *framework*. Os componentes *Domain Knowledge*, *Model* e *Matching Strategy* perfazem os seus principais pontos de flexibilização. O componente *Services Layer* representa uma camada com a qual uma aplicação pode se comunicar com o *framework*, requerendo seus serviços.

A arquitetura anterior pode ser dita simplificada, pois a camada de serviços do *framework* não mantém apenas uma instância de cada um dos pontos de flexibilização definidos acima, como apresentado na figura correspondente. Na realidade, nela estão disponibilizadas todas as instâncias das várias implementações desses componentes. Dessa forma, uma aplicação que solicita a execução de um processo de *matching* pode escolher, dentre os vários componentes disponíveis, aqueles que melhor se adaptam às características do problema de *matchmaking* que ela deseja resolver.

Assim, quando uma requisição de *matching* é solicitada, uma instância dinâmica do *framework* será definida, com base nos parâmetros da requisição do problema. Essa instância é dita dinâmica pois corresponde à composição de uma representação de cada um dos componentes apresentados anteriormente. A Figura 5 apresenta uma visão geral dessa arquitetura.

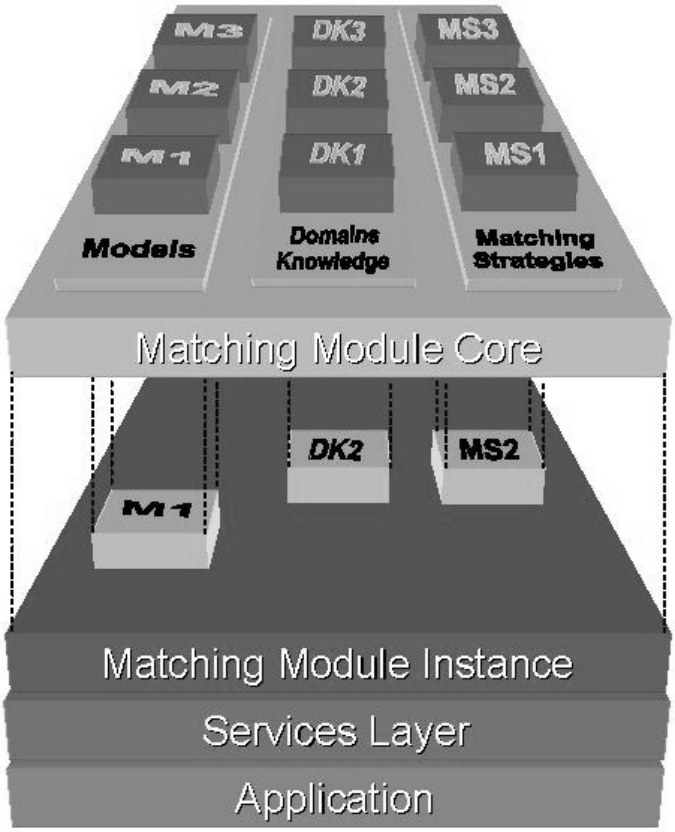

Figura 5 – Arquitetura do Framework Matching Module

As próximas seções discutem as principais características desses componentes, além de levantar os problemas relacionados ao processo de *matchmaking* que eles visam a resolver.

3.3.1. O Componente Domain Knowledge

A definição e a representação da informação necessária ao processo de *matching* para um determinado domínio de aplicação é uma tarefa bastante complexa. Primeiro, é necessário definir algum tipo de função que possa ser aplicada sobre suas instâncias, obtendo-se como resultado o grau de similaridade entre as mesmas. Esse valor de similaridade é necessário para que seja possível propor as melhores alternativas de *matching* para uma dada demanda do domínio.

Outra questão importante a ser estabelecida é definir quando um resultado de *matching* entre instâncias de um dado domínio é válido. Muitas vezes o grau de similaridade entre uma oferta e uma demanda é alto, porém algum tipo de restrição definida para o *matching* pode invalidar essa oferta em particular.

Suponha, por exemplo, que se tenha uma pessoa e uma lista de cursos, tendo-se como objetivo definir qual o melhor curso para essa pessoa cursar. Uma restrição para esse *matching* pode ser, por exemplo, o valor a ser pago pelo curso. Dessa forma, o melhor curso obtido pela função que calcula a similaridade pode ter um custo mais alto que aquele definido como restrição para o processo de *matching*, o que invalida o resultado obtido.

O *framework* Matching Module define, então, o componente *Domain Knowledge* como um agregador de informações sobre um determinado domínio de conhecimento. Dessa forma, uma vez que se queira executar o *matching* entre instâncias de um domínio, um perito deve implementar esse componente do *framework*, definindo, assim, a informação necessária para execução do *matching*.

Como dito anteriormente, para todo domínio de conhecimento é necessária a definição de algum tipo de função que avalie o valor de similaridade entre suas instâncias. Dessa forma, cada *Domain Knowledge* deve especificar um *Evaluator*, que define um conjunto de funções de avaliação entre instâncias de um dado domínio. Segue abaixo uma descrição de cada uma dessas funções:

- *Função eval1to1*: Define o grau de similaridade entre duas instâncias simples do domínio. Assim, por exemplo, dada uma pessoa e um curso, a aplicação dessa função sobre essas duas entidades resultaria num valor de similaridade relacionado a algum critério definido pelo perito no domínio em questão (para o caso de pessoas e de cursos, o domínio poderia ser o de gerenciamento de competências);
- *Função eval1toN*: Define o grau de similaridade entre uma instância (demanda) e um conjunto de instâncias (ofertas) do domínio. Assim, caso seja necessário propor um conjunto de cursos para uma dada pessoa, essa função pode ser utilizada para definir graus de similaridade entre uma pessoa e um conjunto de cursos;
- *Função evalNto1*: Define o grau de similaridade entre um conjunto de instâncias (demandas) e uma instância (oferta) do domínio. Assim, caso seja necessário propor um curso para um conjunto de pessoas, essa função pode ser utilizada para esse objetivo;

- *Função evalNtoN*: Define o grau de similaridade entre conjuntos de instâncias do domínio em questão. Assim, para casos em que, por exemplo, deseja-se propor um conjunto de cursos para um conjunto de pessoas, essa função deve ser utilizada.

Nem todas as funções de um *Evaluator* necessitam serem especificadas, mas apenas aquelas que efetivamente serão utilizadas no processo de *matchmaking*. Dessa forma, se para o domínio em questão a única necessidade é realizar *matching* entre instâncias simples (por exemplo, pessoa e curso), basta que seja definida a função de similaridade eval1to1. Outra questão é que para certos domínios algumas funções de similaridade podem ser construídas com base em outras funções já definidas. Por exemplo, o grau de similaridade relacionado com a função eval1toN pode ser definido por N aplicações da função eval1to1 sobre a demanda e o conjunto de ofertas sendo analisado.

Uma característica importante relacionada às funções definidas por um *Evaluator* é que o resultado das mesmas corresponde a um valor de similaridade composto, ou seja, o valor de similaridade é n-dimensional. Dessa forma, ao propor o melhor curso para uma pessoa, se dois cursos empataram com relação a i-ésima dimensão do valor de similaridade, a (i+1)-ésima dimensão será utilizada como fator de desempate. Caso os dois cursos tenham exatamente o mesmo grau de similaridade, o primeiro obtido será definido como a melhor opção.

Durante o processo de *matching* muitas soluções podem ser testadas para que se obtenha o melhor resultado possível. Para muitos casos, essas soluções são obtidas de forma incremental. Suponha, por exemplo, o caso em que se deseja obter o melhor conjunto de cursos para uma dada pessoa. Num primeiro passo do processo, deve-se obter o curso mais interessante para essa pessoa cursar. Para a escolha do segundo curso, deve-se levar em consideração que a pessoa já fez o primeiro curso. Assim, se o primeiro curso tinha como tópico “programação orientada a objetos”, não é interessante propor um segundo curso que ensine a mesma coisa novamente.

Dessa forma, para os casos em que a solução de um problema de *matchmaking* deve ser construída de forma incremental, deve-se definir alguma forma de alterar o estado das instâncias do domínio de forma que uma escolha

anterior possa afetar as novas escolhas que irão compor o resultado final do *matching*.

Para tanto, em todos os domínios em que as soluções devam ser construídas de forma incremental, as implementações de *Domain Knowledge* devem especificar um componente *State Changer*. O objetivo desse componente é definir um procedimento para alterar o estado das instâncias do domínio de conhecimento em questão quando, para cada passo do processo de *matchmaking*, uma escolha é realizada.

A Figura 6 apresenta um exemplo de uso do componente *State Chager* em que um curso é proposto para uma pessoa. O procedimento, nesse caso, é inserir o curso proposto como cursado, além de incluir todos os tópicos vistos nesse curso como tópicos que a pessoa conhece. Dessa forma, após a execução do procedimento de mudança de estado, o próximo passo do processo de *matching* poderá levar em consideração uma nova pessoa, que fez o curso proposto no passo anterior e que sabe os tópicos ensinados no mesmo.

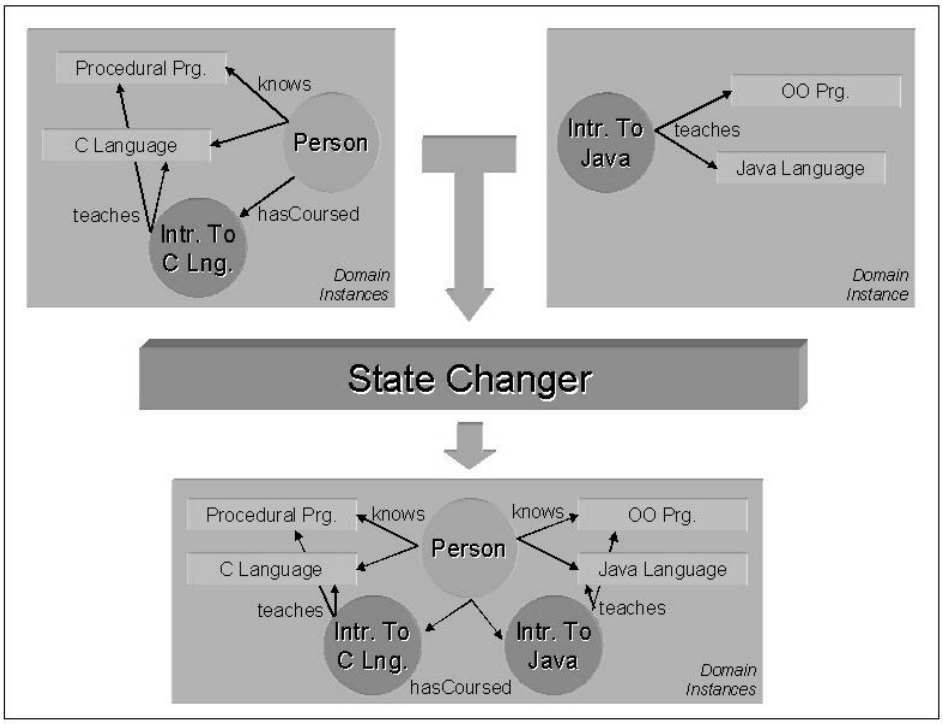

Figura 6 – Exemplo de uso do State Changer

Um *Domain Knowledge* também define um mecanismo para testar a validade das soluções de *matching* entre instâncias de um dado domínio. Para tanto, pode ser definida uma série de restrições que devem ser atendidas de

maneira que o *matching* entre as instâncias em questão possa ser considerado válido. Cada restrição é definida como uma extensão do componente *Restriction*, que pode ser adicionado ao conjunto de restrições definidas para um *Domain Knowledge*.

Tais restrições podem versar, por exemplo, sobre valores de propriedades relacionadas às instâncias do domínio, bem como sobre outros aspectos que dizem respeito a elas. Para que avaliações desnecessárias não sejam executadas, uma vez que podem levar a processamentos de alto custo computacional, o *framework* sempre executa avaliações de restrições como primeiro passo no processo de *matching*.

3.3.2. O Componente Model

Para que as instâncias de um dado domínio de conhecimento possam ser utilizadas pelo *matchmaker*, faz-se necessário que as informações acerca das mesmas estejam descritas utilizando-se algum modelo de dados. Como exemplo de tais modelos, pode-se citar: bancos de dados relacionais, arquivos XML e modelos de dados semanticamente mais ricos, tais como ontologias.

O *framework* Matching Module tem como uma de suas características ser independente do modelo de dados utilizado para descrever as instâncias de um dado domínio. Para tanto, o *framework* disponibiliza o componente *Model*, que pode ser visto como uma interface genérica de acesso às informações acerca das várias entidades do domínio em questão, descritas com base em um modelo de dados qualquer.

Implementações do componente *Model* podem acessar, por exemplo, um banco de dados, um conjunto de arquivos XML ou um conjunto de ontologias. Para que o *framework* possa ter acesso às informações de cada entidade do domínio descritas nesses modelos de dados, o *Model* disponibiliza um conjunto de objetos *Individual*. Cada instância de *Individual* representa uma visão de todas as informações acerca de uma entidade do domínio descrita num modelo de dados, bem como suas relações, sejam elas explicitas ou implícitas.

Um *Individual* disponibiliza ao *framework* um conjunto de métodos que permitem consultar os valores das propriedades da entidade do domínio que ele encapsula. Esses métodos podem ser separados em dois grandes grupos: aqueles que disponibilizam propriedades de tipos primitivos (*string*, inteiro, ponto flutuante, etc) e aqueles que disponibilizam propriedades que são outras instâncias de *Individual* ou que são listas de instâncias de *Individual*. Assim, por exemplo, se para uma instância de *Individual*, que representa uma pessoa no modelo de dados, fosse consultado os cursos que já cursou, a resultado obtido seria uma lista de instâncias de *Individual*, em que cada instância representa um curso no modelo de dados.

A Figura 7 mostra que as relações presentes entre as entidades do domínio são mantidas entre os *Individuals* que as representam no *Model* (ex: relação *hasCoursed*), bem como os valores de propriedades de tipos primitivos (ex: propriedade *name*). Os relacionamentos implícitos entre as entidades do domínio são materializados explicitamente no *Model*.

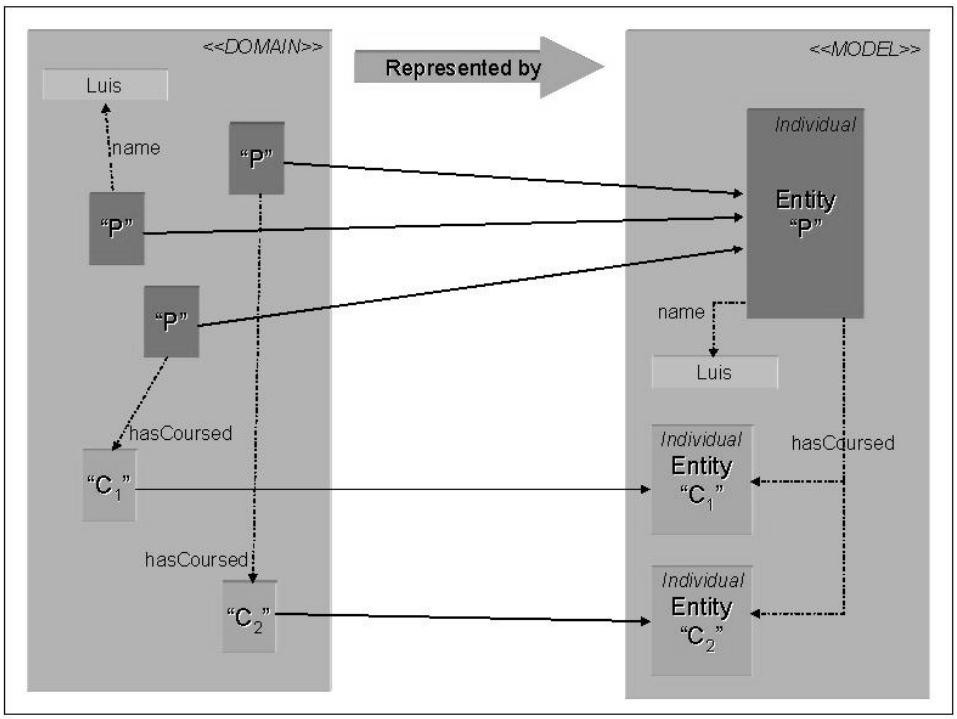

Figura 7 – Exemplo de um cenário de representação de um domínio por um Model

As informações disponibilizadas por um *Model* podem ser utilizadas em dois momentos durante a execução do *matching*. O primeiro deles é durante a execução da função de similaridade, que utilizará as informações das entidades

do domínio para definir o grau de similaridade entre as mesmas. O outro momento é durante a rotina de análise das restrições definidas para os resultados de *matching*, visando a verificar se as soluções propostas são realmente válidas.

3.3.3. O Componente Matching Strategy

Até esse momento, tem-se falado muito sobre como definir as informações que são específicas de um dado domínio e como acessar aquelas acerca das instâncias do mesmo, definidas em algum modelo de dados. Isso foi importante, pois o processo de *machmaking* depende desse ferramental para que possa ser executado.

Porém, a execução de um problema de *matchmaking* leva, na maioria das vezes, a processos com altos custos computacionais. Dessa forma, também é necessário definir estratégias de *matching* mais robustas, que, dependendo do tipo de *matching* a ser executado, possam garantir processamentos mais rápidos, mas que procurem manter um alto grau de precisão nos resultados obtidos.

Nesse ponto, vale descrever um pouco os tipos de *matching* que podem ser executados pelas instâncias do *framework*. Mantendo um paralelismo com as funções de avaliação de similaridade, definidas por um *Evaluator*, o *framework* disponibiliza quatro tipos de *matching*: 1to1, 1toN, Nto1 e NtoN. Cada um desses tipos de *matching* utiliza a respectiva função de similaridade para sua execução. Dessa forma, a partir de dois conjuntos de instâncias do domínio (por exemplo, pessoas e cursos) uma instância do *framework* pode definir para cada pessoa qual é o curso mais relevante a ser feito (1to1), para cada pessoa qual é a seqüência de cursos mais apropriada (1toN), dentre outras.

Assim, existe um forte relacionamento entre uma estratégia de *matching* e uma função de similaridade. Na realidade, pode-se ver uma estratégia de *matching* como uma função de busca e de ordenação dos valores de similaridade obtidos a partir das funções de avaliação definidas num *Evaluator*. Como exemplo, suponha o caso do *matching* 1to1. Dada uma pessoa e um

conjunto de cursos, uma estratégia de *matching* deve obter para cada curso o respectivo valor de similaridade com relação à pessoa. O melhor curso é, então, aquele cujo valor obtido foi maior ou menor, dependendo da natureza do problema a ser resolvido (maximização ou minimização).

Esse tipo de comportamento corresponde a um algoritmo de busca clássico, conhecido na literatura como algoritmo de força bruta [Cormen, 2002]. Perceba que, para problemas dessa natureza, em que se deseja obter a oferta mais adequada para uma demanda, não há muito que fazer além de executar uma busca sobre todas as possibilidades, pois supondo que fosse realizada uma pesquisa somente sobre as (n-1) instâncias do conjunto, não poderia ser garantido que a única instância não pesquisada não representaria a melhor escolha, uma vez que ela não foi testada.

Porém, como levantado anteriormente, em alguns casos existe a necessidade de utilizar algoritmos mais eficientes. Suponha o caso em que seja necessário resolver o problema de propor a melhor seqüência de cursos para uma dada pessoa. Esse problema é NP-Completo [Cormen, 2002]. Dessa forma, é necessário aplicar algum tipo de heurística, que possa disponibilizar resultados de *matching* de alta precisão, utilizando processamentos com custos computacionais polinomiais.

Visando a necessidade de flexibilizar esse processo, o *framework* Matching Module disponibiliza uma *Matching Strategy*, que define uma interface sobre a qual diferentes estratégias de *matching* possam ser definidas e utilizadas nos processamentos realizados pelas instâncias do *framework*. Para tanto, são disponibilizadas três estratégias de *matching* que são adequadas para resolver os quatro tipos de *matching* possíveis de serem executados:

- *Brute Force Strategy*: Essa estratégia de *matching* testa todas as possibilidades e obtém resultados garantidamente ótimos. Porém, ela deve ser utilizada apenas nos casos de *matching* 1to1 e Nto1, pois para problemas do tipo 1toN e NtoN ela pode levar a tempos computacionais exponenciais;
- *Greedy Strategy*: Essa estratégia é baseada nos algoritmos de busca gulosos, ou seja, para cada passo de construção da solução do problema ela utiliza o melhor resultado local. Ela provê soluções

rápidas para problemas do tipo 1toN e NtoN, mas seus resultados não são garantidamente ótimos;

- *Tabu Search Strategy*: Essa estratégia é baseada nos algoritmos de busca tabu, ou seja, a partir de uma solução inicial para o problema, o algoritmo procura por soluções vizinhas a essa primeira, visando a melhorar o resultado corrente. Ela provê soluções bastante rápidas e eficazes para problemas do tipo 1toN e NtoN.

Existe um forte relacionamento entre as três estratégias de *matching* oferecidas pelo *framework*. A *Greedy Strategy* utiliza a *Brute Force Strategy* para obter, a cada passo da construção da solução, a melhor opção local. A *Tabu Search Strategy* utiliza a *Greedy Strategy* para obter a solução inicial e para calcular as possíveis soluções vizinhas, que são obtidas por sucessivas trocas de elementos da solução atual por novos elementos do problema. Assim, a Figura 8 apresenta um esquema que define o relacionamento entre essas estratégias.

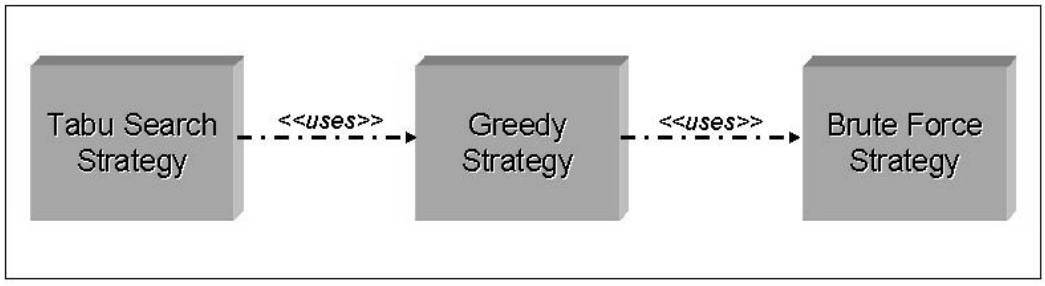

Figura 8 – Relacionamento entre as estratégias de matching

3.3.4. A Camada de Serviços

Uma vez definida toda essa infra-estrutura para que o processo de *matchmaking* possa acontecer, o próximo passo é disponibilizar o seu uso para aplicações que necessitam resolver problemas de *matching* nos mais variados domínios de conhecimento. O *framework* Matching Module disponibiliza para tanto uma camada de serviços, que pode ser utilizada por qualquer aplicação para fazer uso de suas funcionalidades.

Uma vez que existe um forte apelo por soluções de *software* que garantam interoperabilidade e que estejam conforme com as novas tendências sendo utilizadas na *Web*, o *framework* disponibiliza sua camada de serviços baseada no paradigma de *Web Services*.

Todas as implementações relacionadas aos mais deferentes domínios de conhecimento, modelos de dados e estratégias de *matching* podem ser cadastrados e disponibilizados para uso via a camada de serviços do *framework*. Isso implica que na solicitação do processo de *matching* por uma aplicação qualquer, a mesma pode via uma interface de serviço configurar o *matching*, podendo especificar sobre qual domínio deseja executá-lo, bem como sobre qual modelo de dados as suas instâncias estão descritas, além de poder especificar o tipo de estratégia que deseja utilizar, levando em consideração as características do problema a ser resolvido.

Para tanto, a camada de serviços do *framework* Matching Module disponibiliza o seguinte conjunto de serviços:

- *Obter os domínios de conhecimento*: Com esse serviço uma aplicação tem acesso aos diferentes domínios de conhecimento, que são implementações de um *Domain Knowledge*, sobre os quais o processo de *matching* pode ser realizado;
- *Obter os modelos de dados*: Esse serviço disponibiliza os diferentes modelos de dados, implementações de *Model*, suportados pelo *matchmaker*;
- *Obter as estratégias de matching*: De forma semelhante, esse serviço disponibiliza todas as estratégias de *matching* que foram cadastradas para uso;
- *Executar a operação de matching*: Principal serviço disponibilizado pela camada de serviços. Com ele a aplicação especifica os parâmetros e solicita a execução do *matching*, recebendo como retorno o resultado obtido pelo seu processamento.

Para que a aplicação utilize o serviço de execução do *matching*, ela deve especificar todos os parâmetros do problema como um documento XML. Da mesma forma, a aplicação receberá como resultado um documento XML, que especificará todas as soluções obtidas ordenadamente, bem como os valores de

similaridade relacionados a cada uma dessas soluções. Maiores detalhes sobre a implementação da camada de serviços, e em particular sobre a estrutura dos documentos XML, serão apresentados posteriormente no Capítulo 4.

3.4. A OntoAPI – Uma API Para Acesso a Ontologias

A OntoAPI é uma API construída sobre o *framework* Jena, cujo objetivo principal é dar suporte ao acesso a dados anotados semanticamente, que utilizem RDF, RDFS ou OWL como linguagens de descrição. A grande vantagem em usar a OntoAPI, quando comparada ao *framework* Jena, é que a mesma disponibiliza um pequeno conjunto de classes e métodos que encapsulam grande parte do esforço necessário para obtenção de informações sobre as instâncias (dos conceitos) e os metadados de uma ontologia.

A OntoAPI utiliza um modelo de representação baseado em abstrações dos conceitos relacionados a ontologias, ao invés de adotar construções baseadas em alguma linguagem particular. Por exemplo, uma instância de um conceito é representada como um objeto da classe *Individual*. Essa classe disponibiliza uma série de métodos que, por exemplo, permitem retornar de maneira simples os valores das propriedades dessa instância. Dessa forma, o que se obtém com o uso da OntoAPI é uma nova representação das informações descritas na ontologia, não mais como triplas de RDF, mas como objetos das classes definidas pela API.

A OntoAPI disponibiliza dois mecanismos que oferecem grandes vantagens para as aplicações que a utilizam. O primeiro deles corresponde à busca automática de novas ontologias. Durante o processamento da instância de uma ontologia, se uma de suas propriedades referenciar uma outra instância de uma ontologia ainda desconhecida, a OntoAPI, de forma transparente, tentará encontrar essa ontologia, de maneira a obter a instância referenciada pela propriedade.

O segundo mecanismo corresponde ao *caching* de ontologias e de suas instâncias. De forma geral, esse *caching* visa a proporcionar maior eficiência no acesso aos dados da ontologia. Como exemplo, suponha que o valor de uma

propriedade de uma dada instância tenha sido requisitado. Esse valor é procurado na ontologia e armazenado no *cache*. Novas requisições ao mesmo valor poderão, então, ser resolvidas no nível de *cache*, não necessitando ser novamente consultado na ontologia.

Além desses mecanismos, que garantem ganhos significativos ao usuário ou à aplicação que utilizam a OntoAPI, faz-se importante ressaltar que a API foi especificada e construída pensando em flexibilidade e em extensibilidade. Dessa forma, ela possui vários pontos de flexibilização, que podem ser estendidos de acordo com a necessidade de um domínio ou de uma aplicação específica, proporcionando comportamentos diferenciados para cada composição de implementações dessas extensões. Também é oferecido ao usuário a opção de agregar novas funcionalidades à OntoAPI, aumentando assim o seu conjunto de serviços disponibilizados.

A Figura 9, abaixo, apresenta a arquitetura de integração entre uma aplicação e a OntoAPI. Essa arquitetura permite observar que os serviços prestados pela API estão intrinsecamente baseados naqueles disponibilizados pelo *framework* Jena. Nesse sentido, o principal propósito da OntoAPI é oferecer às aplicações, que necessitem fazer acesso a dados anotados semanticamente, uma camada de serviços que permitam diminuir a complexidade e o tempo de programação dessas aplicações, oferecendo uma interface fácil de ser aprendida e utilizada, bem como disponibilizando as informações descritas nas ontologias num modelo de dados simplificado e baseado numa perspectiva de orientação a objetos.

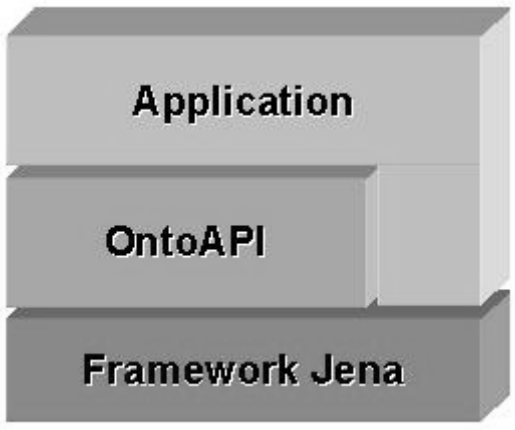

Figura 9 – Arquitetura de integração entre uma aplicação e a OntoAPI

Segue abaixo uma rápida discussão sobre algumas das principais funcionalidades prestadas pela OntoAPI, cujo diagrama de Casos de Uso correspondente encontra-se na Figura 10:

- *Carregar as informações de uma ontologia*: Permite ler o conteúdo de uma ontologia armazenada em uma fonte externa qualquer. Esse processo tem como suporte o mecanismo de localização de ontologias, que verifica num conjunto de fontes a existência da ontologia que se deseja carregar;
- *Obter uma instância (dos conceitos) de uma ontologia*: Permite obter um objeto, do tipo *Individual*, que referencia uma instância de um dado conceito numa ontologia. Essa referência é obtida através da especificação da URI (*Uniform Resource Identifier*) a ser consultada. Caso a ontologia não tenha sido carregada, o mecanismo de localização da OntoAPI será acionado de forma transparente para o usuário;
- *Obter o valor de uma propriedade de uma instância*: Uma vez que a aplicação tenha obtido uma instância, por meio do serviço anterior, a mesma pode requisitar a obtenção de valores para cada uma das propriedades relacionadas a essa instância. O resultado desse serviço pode ser um valor primitivo (*string*, inteiro, etc), um objeto que referencie outra instância descrita na ontologia ou ainda um conjunto de objetos que referenciem diferentes instâncias. Para cada um dos possíveis valores de retorno, a API disponibiliza métodos para realizar a conversão de tipo apropriada;
- *Obter os metadados de uma ontologia*: Permite obter o conjunto de classes e de propriedades definidos numa ontologia;
- *Obter informações sobre os metadados de uma ontologia*: Permite obter informações sobre as classes e as propriedades descritas numa ontologia. Essas informações podem corresponder, por exemplo, ao conjunto de subclasses de uma classe, ao conjunto de super-propriedades de uma propriedade, ao domínio de uma propriedade, ao conjunto de propriedades definidas para uma classe, dentre outros.

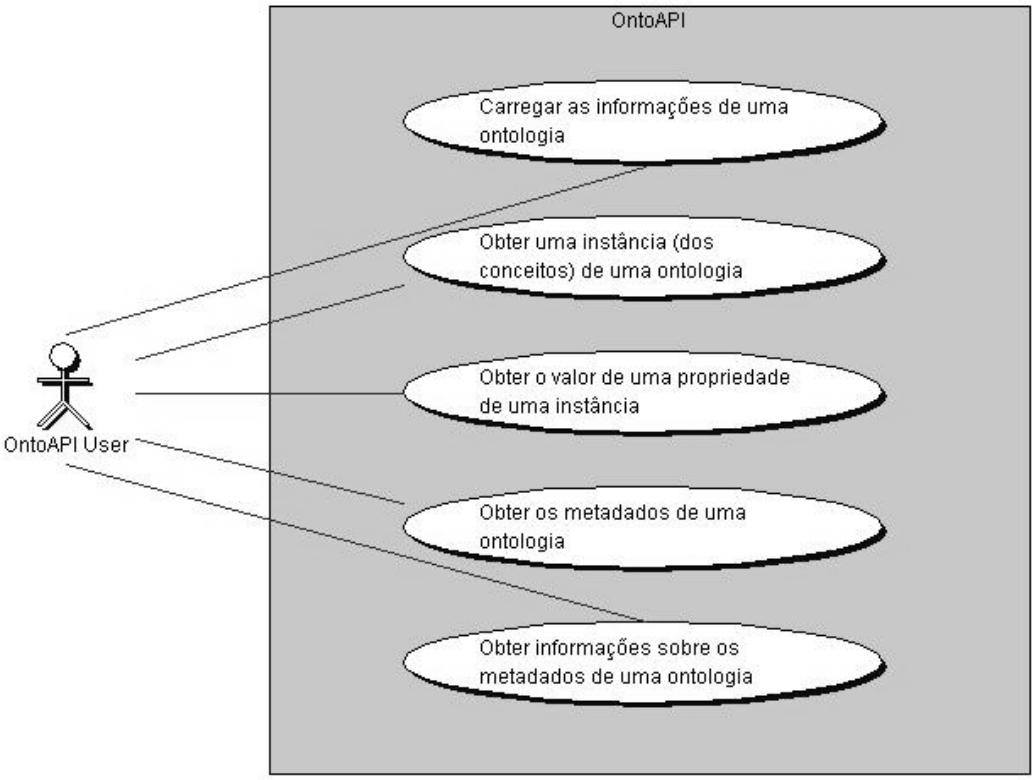

Figura 10 – Casos de uso da OntoAPI

Contudo, nem todos os serviços prestados pelo *framework* Jena mantêm correspondência com os oferecidos pela OntoAPI. O Jena disponibiliza um amplo conjunto de funcionalidades para lidar com ontologias, enquanto a API foca em serviços de acesso às informações das mesmas. De modo a resolver esse problema, a OntoAPI disponibiliza à aplicação as informações das ontologias no modelo de dados definido pelo *framework* Jena. Isso garante à aplicação a capacidade de utilizar o Jena para executar as funções que a API não oferece.

Uma outra questão a ser levantada acerca da OntoAPI é que ela dá suporte à manipulação de várias ontologias simultaneamente. Dessa forma, todas as ontologias que uma aplicação necessita manipular são agregadas a uma única instância da OntoAPI. Contudo, quando um serviço é requisitado, tal como obter o valor de uma propriedade de uma instância, apenas um subconjunto de ontologias relevantes para obtenção dessa informação será utilizado. A escolha desse subconjunto de ontologias é transparente para o usuário da API, além de ser definida como um de seus pontos de flexibilização, o que permite ajustes específicos dessa operação para casos particulares de seu uso.

A Figura 11 apresenta a arquitetura da OntoAPI. Segue também uma rápida discussão sobre seus principais componentes.

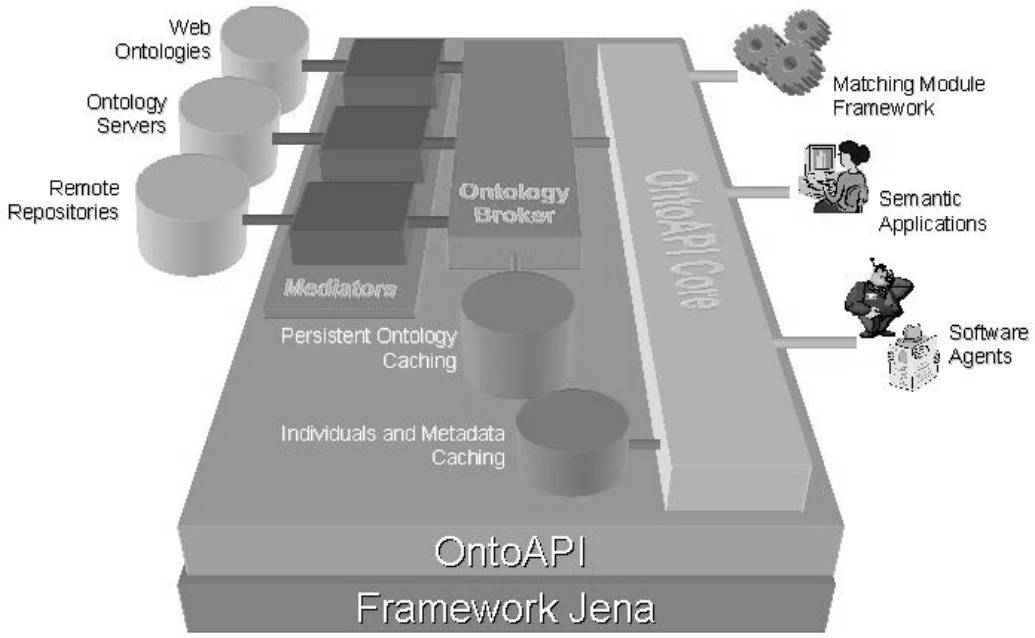

Figura 11 – Arquitetura da OntoAPI

O OntoAPI Core representa a interface de comunicação entre os serviços prestados e as aplicações que os utilizam. Como pode ser visto na arquitetura, vários tipos de componentes de *software* podem fazer uso da API, tais como agentes de *software*, aplicações para uso na *Web* Semântica e, em particular, o *framework* Matching Module.

Todo processamento relacionado ao tratamento e à obtenção de informação das ontologias é realizado por esse componente. Ele também define um primeiro nível de *caching*, relacionado às informações sobre as instâncias dos conceitos e sobre os metadados das ontologias manipuladas pela API.

O componente *Ontology Broker* é responsável pela localização das ontologias solicitadas pelo componente OntoAPI Core. Esse processo de localização tem como principal característica ser transparente para a aplicação ou para o usuário que utiliza a OntoAPI. Dessa forma, se uma ontologia está disponível na *Web* e uma aplicação necessitar utilizá-la, a única preocupação que o usuário deverá ter é fornecer uma conexão de rede ativa na sua máquina. O restante do processo fica como responsabilidade da API.

Para tanto, um *Ontology Broker* conta com um conjunto de mediadores. Cada mediador é responsável pela localização de ontologias num determinado ambiente. Assim, pode-se ter mediadores responsáveis pela localização de ontologias na *Web*, em servidores de ontologias (nesse caso, um mediador pode ser capaz de fazer *login* nesses servidores, consultar um catálogo e obter as ontologias necessárias), dentre outros.

O *Ontology Broker* também define um segundo nível de *caching* para a OntoAPI. Uma vez que obter determinadas ontologias pode representar um processo custoso computacionalmente, o usuário da OntoAPI tem a opção de ativar um *caching* persistente de ontologias. Dessa forma, todas as ontologias obtidas pelo conjunto de mediadores passam a serem armazenadas localmente, e futuras solicitações das mesmas poderão ser resolvidas localmente, sem a necessidade de um novo acesso às suas fontes externas de origem.

3.5. Considerações Finais

3.5.1. Sobre a Infra-Estrutura Proposta

Com relação à infra-estrutura proposta, pode-se dizer que ela atende de forma bastante satisfatória às características e aos requisitos levantados no início desse capítulo. Ela disponibiliza vários artefatos de *software* úteis para aplicações que precisam lidar com *matchmaking*, bem como define a possibilidade de extensão de suas funcionalidades.

A independência de domínio de aplicação é obtida pela definição de vários *hot spots* do *framework* Matching Module. Os componentes *Domain Knowledge*, *Evaluator*, *State Changer* e *Restriction* mapeiam as informações específicas de um domínio particular necessárias para a execução do processo de *matchmaking*.

A independência do modelo de dados também é definida como um ponto de flexibilização do *framework*. Os componentes *Model* e *Individual* tem como

objetivo representar, respectivamente, uma interface para acesso a informações descritas através de um modelo de dados particular e uma interface para representação de entidades (instâncias) do domínio, bem como para obtenção dos valores das propriedades relacionadas a elas.

Devido à complexidade das operações relacionadas ao processo de *matchmaking*, o Matching Module disponibiliza uma interface para definição de diferentes estratégias de *matching*, representada pelo componente *Matching Strategy*. Esse ponto de flexibilização visa a atender o requisito relacionado à flexibilidade e extensibilidade, levantados anteriormente.

Com relação a configurabilidade do processo de *matchmaking*, o *framework* disponibiliza a possibilidade de uma aplicação especificar vários parâmetros na requisição de um processo de *matching*, como será apresentado no Capítulo 4. Esses parâmetros tratam de vários aspectos do processo, sendo definidos segundo um documento XML, que é enviado ao *framework* durante a requisição de um *matching*.

A camada de serviços do *framework*, construída segundo o paradigma de *Web Services*, garante à infra-estrutura o atendimento ao requisito de interoperabilidade. Além disso, essa opção eleva o *framework* a uma categoria de aplicações que estão de acordo com as novas tendências tecnológicas que estão sendo amplamente utilizadas na *Web*.

Motivado pelo requisito de simplicidade de uso, a infra-estrutura disponibiliza vários componentes e serviços que escondem grande parte do esforço necessário para executar o *matching*. É notório que escrever funções de similaridades para alguns domínios não é uma tarefa fácil, mas uma vez disponíveis sua utilização torna-se bastante simples, pois basta que sejam referenciados, assim como outros parâmetros, na requisição do processo de *matching*.

A utilização da OntoAPI garante à infra-estrutura a possibilidade de acessar informações sobre instâncias descritas através de ontologias. Assim, a demanda relacionada ao projeto firmado com o Instituto Fraunhofer FIRST foi atendida pela construção de uma implementação do *hot spot* relacionado ao

modelo de dados, utilizando os serviços da OntoAPI como solução para acesso a informações descritas por ontologias.

3.5.2. Sobre o Framework Matching Module

O *framework* Matching Module agrega grande parte das funcionalidades da maioria das aplicações para *matchmaking* que foram apresentadas no capítulo de Fundamentos. Primeiramente, o fato do *framework* ser independente de domínio permite sua aplicação direta em todos os domínios em questão. A independência de modelos de dados, a possibilidade de definir restrições e diferentes algoritmos de *matching* também são exemplos da aplicabilidade do *framework* em todos os casos tratados por essas aplicações.

Porém, não é possível fazer uma comparação no nível de propósito final entre o *framework* e esses trabalhos. Isso se explica pelo fato de que eles versam sobre aplicação de *matching* para domínios específicos, às vezes utilizando linguagens proprietárias ou construtores que dificultam sua generalização para outros fins.

O Matching Module, no entanto, foi pensado e construído segundo um paradigma de flexibilidade e de extensibilidade, sendo independente de domínio de aplicação. O trabalho que mais se aproxima, conceitualmente, da proposta do Matching Module é o descrito em [Veit, 2001]. Nele é apresentado um *framework* para construção de funções de similaridade, que sejam independentes de domínio de aplicação. Esse trabalho pode ser visto como um complemento das funcionalidades do Matching Module, podendo ser utilizado como base para implementações das funções definidas por um *Evaluator*.

Uma característica bastante comum nos trabalhos relacionados é a definição de um processo de armazenamento das requisições de *matching* que não puderam ser resolvidas, visando a executá-las novamente quando alguma descrição definida posteriormente atender às suas demandas. O *framework* Matching Module não trata desse processo, sendo seu comportamento restrito apenas à definição de serviços de *matching*.

3.5.3. Sobre a OntoAPI

A OntoAPI disponibiliza parte dos serviços que as API's discutidas no capítulo de Fundamentos normalmente oferecem. Todo o conjunto de funções para persistência de ontologias não é oferecido pela OntoAPI, que foca sua atenção apenas nas operações de acesso às ontologias.

Quanto ao paradigma de representação, a OntoAPI apresenta uma abordagem dinâmica. Os construtores genéricos são representados por classes proprietárias da API, que não atendem a nenhuma representação de uma linguagem particular. Porém, os objetos dessas classes referenciam objetos das classes definidas pelo *framework* Jena, que é o componente de *software* utilizado como base para a construção da OntoAPI.

Como grandes diferenciais da OntoAPI com relação aos outros trabalhos apresentados, destacam-se os mecanismos de localização de ontologias e de *caching* das informações consultadas pela API. Uma outra questão importante é com relação ao projeto da OntoAPI, que disponibiliza às aplicações grande flexibilidade na definição de suas funcionalidades, bem como extensibilidade, garantindo a possibilidade de inclusão de novos serviços.

Uma questão muito importante, mas que não foi tratada no escopo atual desse trabalho, diz respeito ao *caching* persistente de ontologias. Uma vez que esse mecanismo visa a garantir que futuras requisições por ontologias possam ser resolvidas localmente, deve-se propor algum mecanismo auxiliar para controle da versão das ontologias armazenadas no repositório. Dessa forma, as aplicações poderiam ter como opção especificar se desejam acessar sempre a versão mais nova da ontologia, ou uma versão particular obtida anteriormente pelos mediadores associados ao *broker* de ontologias.