Uma Abordagem para Incorporar Mecanismos de Inteligência Artificial a Agentes Móveis

Paulo Sérgio da Silva[1], Manuel de Jesus Mendes[2]

[1]Departamento de Engenharia Elétrica
Faculdade de Engenharia de Bauru
Universidade Estadual Paulista (UNESP)
Caixa Postal 473 – 17001-970 – Bauru – SP – Brazil

[2]Departamento de Engenharia de Computação e Automação Industrial
Faculdade de Engenharia Elétrica e de Computação
Universidade Estadual de Campinas (UNICAMP)
Caixa Postal 6101 – 13081-970 – Campinas – SP – Brazil

pss@feb.unesp.br, mendes@dca.fee.unicamp.br

Abstract. *The development of mobile agents employing Artificial Intelligence techniques has not been much considered in the literature. Toolkits specially developed for constructing Intelligent Mobile Agents are not available yet. This lack keeps mobile agent developers restricted to the procedural approach of writing software. This paper presents the development of a flexible and extensible Intelligence Framework that aims to overcoming this limitation. The implementation of a rule-based forward-chaining inference engine in agreement with the suggested framework is also discussed.*

Resumo. *O desenvolvimento de agentes móveis empregando técnicas de Inteligência Artificial não tem sido muito considerado na literatura. Bibliotecas de classes especialmente desenvolvidas para a construção de Agentes Móveis Inteligentes ainda não estão disponíveis. Esta carência obriga os desenvolvedores a utilizarem somente a abordagem procedimental na construção de tais agentes. Este trabalho apresenta o desenvolvimento de um Framework de Inteligência flexível e extensível capaz de superar esta limitação. A implementação, de acordo com o framework proposto, de um mecanismo de inferência para frente baseado em regras também é discutida.*

1. Introdução

Agentes Móveis (AM) são elementos autônomos de software que podem migrar entre os computadores de uma rede heterogênea, procurando e interagindo com serviços em nome de seus usuários. *Sistemas de Agentes Móveis* (SAM) utilizam-se de servidores especializados, denominados *agências*, para interpretar o comportamento do agente e permitir sua comunicação com os serviços oferecidos pelo computador hospedeiro ou por outros computadores da rede. Um computador conectado em rede pode hospedar uma ou mais agências, que, por sua vez, podem servir a um ou mais agentes.O ambiente composto por todas as agências às quais os agentes têm acesso é denominado de *Ambiente Distribuído de Agentes* (ADA).

Um agente móvel possui, inerentemente, autonomia de navegação, podendo, a qualquer momento, solicitar seu envio para uma outra agência. Qualquer computador conectado em rede, que possua uma agência, deve ser capaz de executar um agente móvel, sem que para isso o código executável do agente precise ser instalado com antecedência nessa máquina. Tais sistemas utilizam-se das facilidades de mobilidade de código oferecidas por certas linguagens, como por exemplo, Java, Telescript, Tcl/Tk, Python, Obliq e Facile, que prevêem mecanismos para transporte das classes através da rede, em tempo de execução (Cetus, 2002).

Atualmente, existem diversas ferramentas para a construção de aplicações baseadas em agentes móveis, como o *Aglets* da IBM, o *Grasshopper* da IKV, o *Voyager* da *ObjectSpace*, o *Odyssey* da General Magic e o *Concordia* da Mitsubishi (Cetus, 2002). Contudo, até o momento, aspectos relativos à integração entre as técnicas de IA e a mobilidade ainda não foram adequadamente considerados na literatura. Uma vez que um agente móvel passa a maior parte de sua existência operacional distante de seu usuário, é importante a utilização, em seu desenvolvimento, de técnicas que possam ampliar sua autonomia, flexibilidade e capacidade de adaptação. Portanto, técnicas provenientes da IA como, por exemplo, o raciocínio lógico, o planejamento e o aprendizado de máquina, merecem avaliação. No entanto, existe uma carência de arquiteturas e de bibliotecas de classes exclusivamente desenvolvidas para a construção de *Agentes Móveis Inteligentes* (AMI), isto é, agentes móveis que realizem suas tarefas com o auxílio das técnicas de IA. O que se encontra são ferramentas de uso geral para a construção de agentes inteligentes que também podem ser utilizadas para a criação de um AMI. Em todas essas ferramentas, a técnica de IA que mais está presente é a inferência baseada em regras. O aprendizado de máquina aparece em algumas delas e o planejamento em nenhuma. Porém, na arquitetura e no desenvolvimento de todas, não são considerados dois requisitos extremamente importantes para o uso e aceitabilidade das técnicas provenientes da IA na construção de AMIs: primeiro, a redução do tamanho (em bytes) do código executável de suas classes e, segundo, a redução da quantidade de dados (em bytes) a serem transportados pelo agente. Justifica-se a preocupação com o tamanho, uma vez que estudos de desempenho dos principais SAMs (Silva et al., 1999; Ismail e Hagimont, 1999) demonstram que eles não são capazes de transportar agentes cuja quantidade de código e de dados exceda 1Mbytes. Essa situação restringe, até o momento, o desenvolvimento dos agentes móveis à abordagem procedimental de programação de software, o que, para muitos pesquisadores, é um retrocesso (Nwana e Ndumu, 1999).

Uma estratégia realista para iniciar a integração entre mobilidade e inteligência é a construção de agentes móveis com poucas, mas bem estabelecidas técnicas de IA e, à medida que essas técnicas se mostrem úteis, e seus impactos sobre a mobilidade forem avaliados, acrescentar outras de forma incremental. A idéia não é abdicar da programação procedimental, mas sim acrescentar módulos “leves“ de IA que possam ser facilmente acoplados aos agentes móveis. Com o objetivo de viabilizar esta avaliação, este trabalho apresenta um *framework* que permite, inicialmente, incorporar a agentes móveis, construídos com os principais sistemas de mobilidade contemporâneos, um *mecanismo de inferência baseada em regras com encadeamento para frente*. Vários *padrões de projeto* (Gamma et al., 1995) são utilizados na concepção de ambos, visando um acoplamento “fraco” entre seus diversos componentes e subsistemas, de modo a reduzir a quantidade de código e de dados transportados pelos agentes e ampliar a possibilidade de extensão e reuso dos elementos identificados. A idéia é que o *framework* proposto,

além de "leve", seja suficientemente flexível para que, no futuro, outros mecanismos como, por exemplo, de inferência em lógica nebulosa e de aprendizado, possam ser incorporados, ampliando ainda mais a autonomia e a capacidade dos agentes móveis na realização de tarefas em ambientes dinâmicos e distribuídos.

Este artigo está estruturado da seguinte forma: a seção 2 discute alguns trabalhos relacionados; na seção 3 são especificados os requisitos que o *Framework de Inteligência* deve apresentar e discute-se como esse requisitos são satisfeitos; a seção 4 apresenta uma breve descrição do mecanismo de inferência com encadeamento para frente baseado em regras e de sua modelagem; a seção 5 descreve os resultados da implementação do *framework* e do mecanismo de inferência e a seção 6 apresenta as conclusões.

2. Trabalhos Relacionados

Embora nenhum outro *framework* tenha sido identificado na literatura, exclusivamente desenvolvido para incorporar mecanismos de IA a agentes móveis, existem alguns trabalhos relacionados à proposta deste trabalho, especialmente no que se refere ao mecanismo de inferência implementado.

Desses, o mais conhecido e utilizado é o JESS, cujos principais componentes são um *shell* para sistemas especialistas e uma linguagem *script*, ambos desenvolvidos em Java por Friedman-Hill (2001). JESS suporta o desenvolvimento de sistemas especialistas baseados em regras, possibilitando sua inserção em qualquer outro código escrito em Java. A sintaxe da linguagem JESS é muito similar à sintaxe da linguagem CLIPS e seu mecanismo de inferência com encadeamento para frente é baseado no algoritmo de casamento *Rete* (Forgy, 1982). No entanto, JESS não é capaz de interagir com procedimentos externos com o objetivo de sensoriamento, sendo possível a utilização desses somente para a atuação. Além disso, as atribuições das principais funções do mecanismo de inferência estão distribuídas por muitas classes, o que as torna fortemente interdependentes e com baixa coesão de responsabilidades. Isso faz com que a quantidade de código e de dados transportada seja maior que o necessário. A utilização do JESS para incorporar inferência baseada em regras em um agente acrescenta uma quantidade extra de código de aproximadamente 442 kbytes e mais cerca de 19 kbytes relativos a dados, valores que não incluem a base de conhecimento (regras e fatos) do agente.

Bigus e Bigus (1998) desenvolveram uma biblioteca de classes em Java, denominada CIAgent (*Constructing Intelligent Agents*), na qual se encontram, tanto um mecanismo de inferência com encadeamento para frente, quanto um mecanismo com encadeamento para trás, ambos *situados*, isto é, com a capacidade de interagir com procedimentos externos para sensoriamento e atuação. Contudo, a representação de conhecimento utilizada, na forma de regras em lógica booleana (proposicional), é muito limitada. Além disso, o mecanismo não mantém nenhum tipo de memória de trabalho para armazenar a descrição do estado atual do ambiente. Dessa forma, a inferência é realizada somente com base na percepção corrente. Por outro lado, nas classes que representam as regras e o seu conjunto, os autores incluem comportamentos claramente dependentes dos algoritmos de controle da inferência. Neste trabalho, adotou-se a postura de manter tais comportamentos exclusivamente nas classes que compõem o mecanismo de inferência, fazendo com que os objetos de representação de conhecimento sejam independentes do uso pretendido para eles. Essa estratégia aumenta a flexibilidade e o reuso de tais componentes. A utilização do CIAgent acrescenta uma quantidade extra de códi-

go de aproximadamente 47 kbytes.

O *Agent Building and Learning Environment* (ABLE) é uma coleção de ferramentas desenvolvidas em Java no T. J. Watson Research Center da IBM para facilitar a criação de agentes inteligentes e aplicações baseadas em agentes (IBM, 2000). Essas ferramentas oferecem um conjunto de componentes reutilizáveis, chamados de *AbleBeans*, que implementam funcionalidades de acesso, filtragem e transformações de dados, além de mecanismos de inferência baseada em regras, usando lógicas booleana e nebulosa, e de aprendizado, baseado em redes neurais. Agentes específicos para uma determinada aplicação podem ser construídos, usando-se um ou mais desses componentes. Embora os mecanismos oferecidos sejam situados, eles apresentam as mesmas desvantagens encontradas no sistema de Bigus e Bigus (1998). A utilização do ABLE acrescenta uma quantidade extra de código de aproximadamente 496 kbytes.

O *Agent Building Environment* (ABE) da IBM (1997) fornece uma arquitetura e uma biblioteca de classes para o desenvolvimento de aplicações baseadas em agentes inteligentes escritos em C++ e Java. O ABE é constituído por seis tipos de componentes: *agentes*, *mecanismos*, *conhecimentos*, *bibliotecas*, *adaptadores* e *visualizadores*. Os *agentes* são criados através da especialização de uma classe básica que fornece as operações necessárias para que as aplicações os configurem. Os *mecanismos* são responsáveis por interpretar as instruções que controlam o comportamento dos agentes. Estas informações são codificadas na forma de *conhecimento*. As *bibliotecas* permitem o armazenamento de conjuntos nomeados de conhecimentos e de seus respectivos metadados. Através dos *adaptadores*, os mecanismos podem acessar procedimentos externos para avaliar se determinadas condições são satisfeitas e, então, executar certas ações com base nos resultados obtidos. *Visualizadores* determinam como ferramentas de autoria de conhecimento e interfaces gráficas de usuários podem ser incorporados às aplicações. A parte central do ABE é um mecanismo de inferência para frente que controla o comportamento do agente e permite a configuração e o uso dos adaptadores, através de um conjunto de regras e fatos expressos em KIF (*Knowledge Interchange Format*). Esse é o único mecanismo fornecido com o ABE. Embora novos mecanismos possam ser integrados, isso só será possível de forma proprietária.

Apesar de sua arquitetura bastante flexível, o ABE apresenta algumas características que dificultam e até mesmo impedem sua utilização com agentes móveis. Embora seja possível acrescentar novos adaptadores escritos em Java, o núcleo de suas funcionalidades é escrito em C++, o que impede sua mobilidade na maioria dos SAMs atuais. Mesmo que a mobilidade fosse possível, o tamanho do código de suas bibliotecas de ligação dinâmica é proibitivo para agentes móveis, cerca de 4,7 Mbytes. Outra dificuldade é o fato de que, para utilizarem as funcionalidades oferecidas pelos componentes ABE, os agentes devem herdar comportamentos de uma classe abstrata específica. No entanto, os agentes, para serem móveis, também precisam herdar comportamentos de uma classe base disponibilizada pelo SAM. Como Java não permite herança múltipla, não é possível combinar os comportamentos inteligente e móvel dessa forma. Além disso, fatos deduzidos durante um episódio de inferência não são mantidos na base de conhecimento após o seu encerramento. Essa característica é crítica para agentes móveis, que deverão poder interromper sua execução num computador e reiniciá-la noutro.

O ABE também padece de mais dois problemas, comuns às demais ferramentas: (1) as APIs dos mecanismos estão intimamente vinculadas a uma linguagem de repre-

sentação de conhecimento específica e (2) todos possuem seus analisadores sintáticos (*parsers*) acoplados de forma inseparável de suas funcionalidades centrais. A primeira questão é crítica, pois não é possível garantir, pelo menos atualmente, que as representações de conhecimento sejam as mesmas em todos os computadores nativos e/ou visitados. Já a segunda pode representar um desperdício de banda, pois geralmente as tarefas de análise e conversão para um formato de representação envolvem quantidades razoáveis de código e dados, nem sempre necessários após a inicialização do agente.

3. Um *Framework de Inteligência* para Agentes Móveis

Nesta seção especificam-se os requisitos que o *framework* deve cumprir e propõe-se uma arquitetura capaz de satisfazê-los.

3.1. Especificação dos Requisitos

Os seguintes requisitos foram identificados como importantes:

R1. Todos os componentes do *framework* devem ser inteiramente codificados em Java. Esse requisito é necessário porque os principais SAMs contemporâneos são desenvolvidos em Java, não existindo evidências de que essa tendência irá se alterar. Embora *Prolog* e *Lisp* sejam as linguagens usualmente associadas à programação em IA, recentemente, muitos trabalhos comerciais nessa área têm sido implementados em C e C++ (Bigus & Bigus, 1998). Como linguagem de propósito geral, Java pode substituí-las com, no mínimo, as vantagens da portabilidade, da robustez, da segurança e do oferecimento de mecanismos de suporte à execução concorrente, à serialização de objetos, à conexão de rede e ao carregamento dinâmico de classes.

R2. Durante sua execução, um mecanismo deve ser capaz de receber, de forma assíncrona, percepções do agente móvel sobre si mesmo e sobre seu ambiente. Este requisito permite que o agente móvel solicite o serviço de um mecanismo, toda vez que perceber a ocorrência de algum evento importante em seu ambiente ou em sua própria execução. Uma *percepção* é a união de um evento com um conjunto de informações que o qualificam. Uma mesma percepção pode ser enviada a mais de um mecanismo. O agente é a entidade que deverá decidir sobre quais eventos são importantes e para quais mecanismos as percepções devem ser enviadas, facilitando-se assim a incorporação de comportamentos reativos.

R3. Durante sua execução, um mecanismo deve ser capaz de solicitar informações adicionais ao agente. A solicitação de informações adicionais corresponde a chamadas de procedimentos externos ao mecanismo, denominados *sensores*. Esse requisito é importante para permitir o comportamento situado dos agentes. A ausência do procedimento externo não deve provocar a falha do mecanismo, nem bloquear sua execução, que deverá simplesmente continuar, assumindo-se que a informação não está disponível. Também deve ser transparente, para o mecanismo, se o procedimento externo faz parte do código do agente ou da agência, o que facilita a transferência desses procedimentos do agente para as agências, sempre que o uso de um sensor se tornar ubíquo.

R4. Como resultado de sua execução, um mecanismo deve ser capaz de sugerir uma ação ao agente. A sugestão de ações corresponde à sugestão de chamadas de procedimentos externos ao mecanismo, denominados *atuadores*. Todas as observações, tecidas em R3 para os sensores, valem também para os atuadores. Ao contrário dos tra-

balhos encontrados na literatura, considera-se conveniente que, tanto as funções de sensoriamento, quando as de atuação, sejam mediadas pelo agente. Assim, assegura-se a autonomia do agente, que pode decidir por não avaliar o ambiente em um dado momento ou por executar uma ação proposta por outro de seus mecanismos.

R5. O *framework* deve permitir que os mecanismos sejam facilmente incorporados em um agente móvel já existente. Não se deve impor ou assumir que a implementação do agente especialize qualquer classe específica para que ele seja capaz de utilizar um mecanismo. Esse requisito é importante porque a maioria dos SAMs já impõe que a classe principal do agente especialize uma classe abstrata fornecida que define as operações básicas utilizadas para controlar a mobilidade e o ciclo de vida do agente, além de permitir sua interação com a agência hospedeira. Deve ser possível conectar e desconectar os mecanismos dinamicamente, em tempo de execução, de modo que, através da substituição e/ou agregação de novas partes, o agente possa adaptar-se às alterações de seu ambiente, bem como transportá-las somente quando necessário.

R6. Os mecanismos não devem estar vinculados a nenhum formato não executável de representação de conhecimento. Um mecanismo não deve supor ou impor que o agente seja instruído em um determinado formato de representação de conhecimento, pois isso limitaria sua utilização e obrigaria o usuário a se adaptar ao formato escolhido. O melhor é que exista, para cada tipo de mecanismo, um formato comum de representação *executável*, na forma de objetos de software, para o qual os outros possam ser traduzidos. Isso reduz os esforços de análise sintática e de conversão para o formato interno de representação de cada mecanismo. Para ampliar a possibilidade de seu reuso, esse formato executável de representação deve ser puramente declarativo, não incluindo informações que o vincule a algum tipo específico de utilização.

R7. As classes que compõem o *framework* e os mecanismos agregados devem ser leves, em termos de tamanho de código e de dados. Uma das principias vantagens atribuídas aos agentes móveis é justamente a redução do tráfego na rede. Portanto, o *framework* em si e os mecanismos agregados aos agentes através dele não devem ser pesados a ponto de dificultarem o processo de migração.

R8. As classes que compõem o *framework* e os mecanismos agregados devem ser leves em termos de tempo de execução. Uma vez que na maior parte de seu ciclo de vida um agente móvel será executado em uma máquina que não a de seu proprietário, é importante que seu código seja eficiente, não consumindo recursos computacionais em demasia, recursos esses que normalmente são pagos.

R9. É desejável que os formatos internos de representação de conhecimento utilizados pelos mecanismos sejam opacos. Embora os aspectos de segurança estejam fora do escopo deste trabalho, é interessante dificultar ao máximo o exame, por estranhos, de informações e princípios que regem o comportamento do agente.

R10. Os mecanismos devem ser flexíveis o bastante para permitirem que seus principais algoritmos sejam independentes uns dos outros. Esse requisito tem por objetivo facilitar a utilização de algoritmos mais recentes e/ou alternativos para as principais funcionalidades dos mecanismos, ampliando sua reutilização, sem a necessidade de reprojeto. Dessa forma, algoritmos mais adequados em termos de custo de transporte poderão ser utilizados, à medida que se tornarem disponíveis. Os usuários também poderão criar seus próprios componentes para as principais funcionalidades.

R11. As classes que compõem o framework e os mecanismos agregados não devem estar diretamente vinculadas a qualquer tipo de interface de usuário ou mecanismo de armazenamento persistente de dados. O objetivo desse requisito é maximizar a reutilização do *framework* e dos mecanismos. A idéia é que exista uma separação lógica nítida entre os elementos principais do *framework* e os componentes da interface de usuário e/ou de aplicação, de forma que a substituição de uma interface por outra mais recente, mais amigável ou mais adequada, possa ocorrer sem que seja necessário o reprojeto dos mecanismos e até mesmo a interrupção de sua execução. Essa separação também é importante, pois os componentes das interfaces poderão ser deixados na agência nativa, já que, geralmente, não são necessários nas demais.

R12. As classes que compõem o framework e os mecanismos não devem ser diretamente vinculadas a informações e/ou comportamento de uma aplicação cliente específica. Esse requisito visa maximizar o domínio de aplicações que poderão utilizar-se do *framework*, não restringindo seu uso somente a agentes móveis.

3.2. Estratégia de Desenvolvimento

Em linhas gerais, a estratégia utilizada no desenvolvimento do *Framework de Inteligência* foi a seguinte: (1) não eliminar totalmente a programação procedimental, e (2) projetar o *framework* na forma de módulos "leves" que possam ser transportados pelo agente ou "carregados" sob demanda, após sua chegada em uma agência. Para cada mecanismo, é preciso uma análise cuidadosa para identificar e isolar suas principais funcionalidades e determinar quais delas podem ser transferidas para as agências.

As classes, que devem migrar, foram minuciosamente projetadas para reduzir o tamanho de seu código e de seus dados. Assumiu-se que as funcionalidades transferidas para as agências são oferecidas aos agentes por um *Serviço de Inteligência* (SI). Essa abordagem possui três vantagens: (1) contribui para a redução da quantidade de código e de dados transportados; (2) permite que as agências imponham suas próprias políticas de segurança e (3) garante a "autonomia" das agências, que podem, ou não, oferecer o SI. Além disso, o estabelecimento de uma interface padrão para o SI desacopla a troca de informações entre o AMI e as agências, no que concerne às necessidades específicas dos mecanismos de IA. Desse modo, os procedimentos para "encaminhamento" das solicitações de sensoriamento e atuação podem ser desenvolvidos de forma independente do sistema de mobilidade.

A Figura 1 mostra os principais componentes do *Framework de Inteligência*. O *Código do Agente* representa a programação procedimental realizada pelo desenvolvedor, com base na aplicação específica desejada. No entanto, esse código também deve: (1) configurar o(s) mecanismo(s) durante o processo de criação do agente; (2) iniciar e interromper a execução do(s) mecanismo(s); e (3) servir de "ponte" entre o(s) mecanismo(s) e o SI. O *Comportamento Móvel* corresponde ao comportamento herdado do SAM. A *Interface de Usuário* é o canal de comunicação entre o agente e o seu usuário.

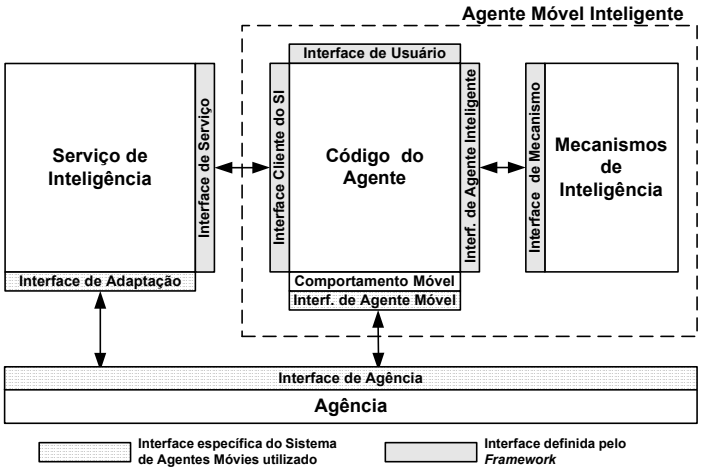

Figura 1. Principais componentes do *Framework de Inteligência*

3.3. Análise dos Requisitos

R1 não necessita de maiores esclarecimentos. R2 exige que o mecanismo seja visível ao agente, isto é, que o agente possua uma referência para ele. R3 e R4 exigem que o agente seja visível ao mecanismo. No entanto, de acordo com R5, o mecanismo deve ser facilmente utilizado com os diversos SAMs baseados em Java, o que significa que, para utilizá-lo com cada um desses sistemas, não deve ser necessário alterar o seu código e recompilá-lo. Porém, os agentes dos diversos SAMs não possuem uma interface comum, o que impede que o mecanismo possa referenciá-los de modo único. Para adaptar o comportamento móvel, esperado pelas agências do SAM escolhido, ao comportamento de cliente esperado pelo mecanismo, utilizou-se o padrão de projeto *Adapter* em sua versão bidirecional (Gamma et al., 1995, p.139). A Figura 2 ilustra a aplicação desse padrão. A classe `Agente Móvel Inteligente` adapta o comportamento de mobilidade herdado da classe `Agente Móvel` ao comportamento esperado pelos mecanismos de inteligência, definido pela interface `Agente Inteligente`. O relacionamento indireto estabelecido pela interface `Agente Inteligente` também satisfaz R12.

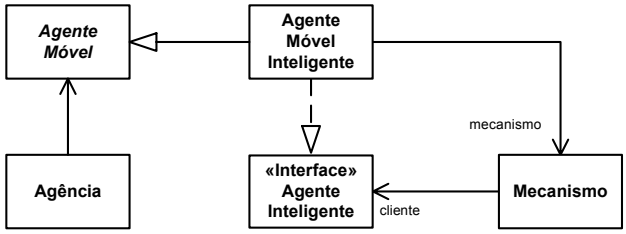

Figura 2. Aplicação do padrão de projeto *Adapter*

Embora cada mecanismo específico possua seu próprio comportamento, algumas partes desse comportamento serão comuns. Para evitar a duplicação de código e de dados utilizou-se o padrão de projeto *Method Template* (Gamma et al., 1995, p.325). Como mostrado na Figura 3, esse padrão define o esqueleto de um algoritmo em um método (`métodoGabarito`) e posterga a definição de alguns de seus passos para as subclasses (`operaçãoPrimitiva`). Dessa forma, a parcela invariante do comportamento é concentrada na classe `Mecanismo Abstrato`, que também define a assinatura das operações primitivas que um `Mecanismo Concreto` deve implementar. O agente utilizará as operações do `Mecanismo Abstrato` para configurá-lo, registrar-se como seu

cliente, iniciar e interromper sua execução e enviar-lhe suas percepções (R2). Percepções relativas ao estado interno do agente serão geradas pelo próprio código do agente. Percepções sobre o estado do ambiente serão enviadas pelo SI local ao código do agente, que as transferirá para o mecanismo. Essa "ponte" permite que o agente selecione para quais de seus mecanismos uma percepção será enviada.

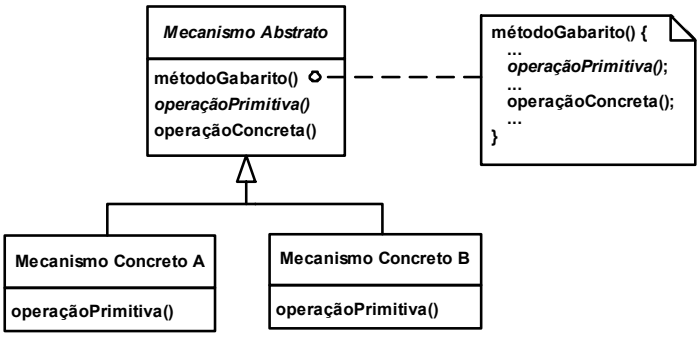

Figura 3. Aplicação do padrão de projeto *Method Template*

Durante sua execução, o mecanismo utilizará as operações oferecidas pela interface `Agente Inteligente` para solicitar informações adicionais (R3) e/ou sugerir a execução de uma ação (R4). O agente poderá honrar uma solicitação de informação adicional de três formas distintas: (1) ele mesmo poderá fornecer a informação desejada; (2) ele poderá solicitá-la a outro de seus mecanismos de IA ou, então, (3) poderá solicitá-la ao SI local. O mesmo se aplica à execução de uma ação. Para desacoplar o código do agente da implementação concreta do SI e vice-versa, além de adaptar o comportamento pré-existente do sistema de mobilidade ao comportamento do SI, utilizou-se, novamente, o padrão de projeto *Adapter*, conforme mostrado na Figura 4.

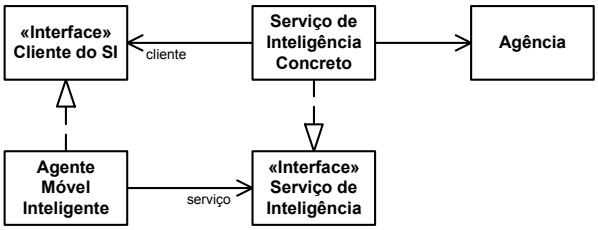

Figura 4. Aplicação do padrão de projeto *Adapter*

A interface `Serviço de Inteligência` adapta o comportamento da agência ao comportamento esperado pelos AMIs. Através dela, o agente poderá registrar-se como cliente do `Serviço de Inteligência Concreto` e solicitar informações adicionais ou a execução de uma determinada ação. Por meio da interface `Cliente do SI`, o SI poderá notificar ao agente a ocorrência de eventos no domínio da agência hospedeira. A aderência aos requisitos R6, R7, R8, R9 e R10 está estreitamente relacionada com os mecanismos concretos. No que se segue serão apresentadas algumas diretrizes a serem adotadas para atendê-los. Para satisfazer R6, para cada tipo de mecanismo, deve ser definido um conjunto de *Interfaces de Representação de Conhecimento* que especifiquem um grupo bem definido e coeso de operações que *objetos "puros" de conhecimento* deverão implementar. As operações dessas interfaces devem permitir o acesso aos componentes constitutivos de cada uma das entidades de conhecimento presentes na representação adotada. *Informações operacionais*, que especificam como o conhecimento representado por esses objetos será utilizado pelos mecanismos, devem ser acrescentadas como *adornos* às entidades básicas. Embora pareça pouco produtivo transferir

para o desenvolvedor a responsabilidade pela implementação dos objetos de conhecimento, a flexibilidade de tal enfoque supera o trabalho adicional, pois esses objetos, uma vez criados, são facilmente reutilizados. Além disso, a quantidade de código e de dados transportados também pode ser reduzida, visto que as tarefas de análise sintática e de tradução podem ser delegadas a um componente específico que não precisará ser transportado.

Para satisfazer R7, a representação de conhecimento utilizada internamente deve ser compacta. Informações compartilhadas por várias sentenças devem ser representadas uma única vez, evitando-se a migração desnecessária de código e dados. No que se refere à redução de dados, o padrão de projeto *Flyweight* (Gamma et al., 1995, p.195) deve ser utilizado. Esse padrão, mostrado na Figura 5, usa o compartilhamento para suportar eficientemente grandes quantidades de objetos de granularidade fina (*flyweights*).

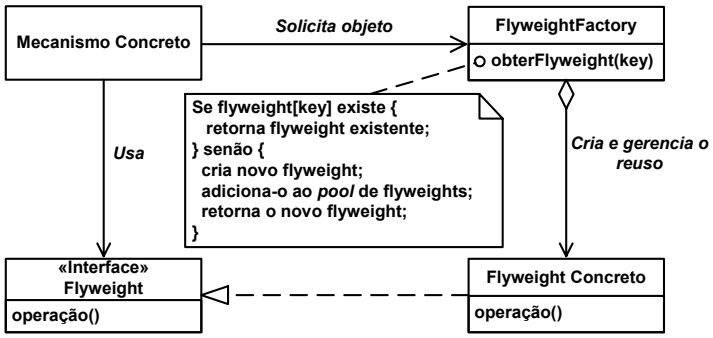

Figura 5. Aplicação do padrão de projeto *Flyweight*

Para atender R8, os algoritmos utilizados pelos mecanismos devem ser eficientes. Para satisfazer R9 devem ser evitadas, nas representações internas de conhecimento, descrições e/ou comentários codificados no formato de texto puro. Para atender R10, as principais funcionalidades dos mecanismos devem ser isoladas e atribuídas a componentes distintos. O padrão de projeto *Strategy* (Gamma et al., 1995, p.305), mostrado na Figura 6, é adequado a este propósito, pois permite definir uma família de algoritmos, encapsular cada um deles e torná-los intercambiáveis.

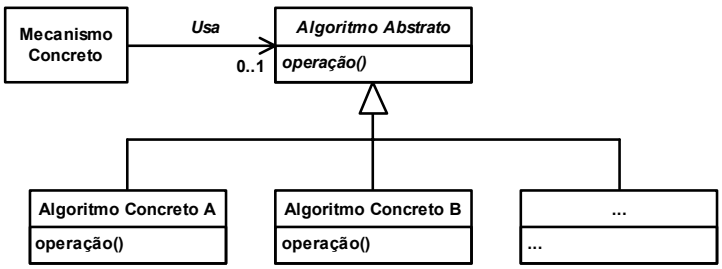

Figura 6. Aplicação do padrão de projeto *Strategy*

Para satisfazer R11, os elementos do *framework* foram agrupados em subsistemas, de acordo com a arquitetura clássica de 3 camadas (Figura 7). Existem quatro tipos de subsistemas: (1) subsistemas cujos elementos são fornecidos pelo SAM adotado: *Plataforma de Mobilidade* e *Comportamento Móvel*; (2) subsistemas que representam o núcleo do *Framework de Inteligência*: *Mecanismo Abstrato* e *Interfaces do SI*; (3) subsistemas específicos de cada mecanismo concreto adicionado ao *framework*: *Interface de Representação de Conhecimento* e *Mecanismo Concreto*; e (4) subsistemas cujos

elementos deverão ser desenvolvidos pelo usuário do *framework*: *Lógica do Agente*, *Interface de Usuário* e *Serviço de Inteligência*.

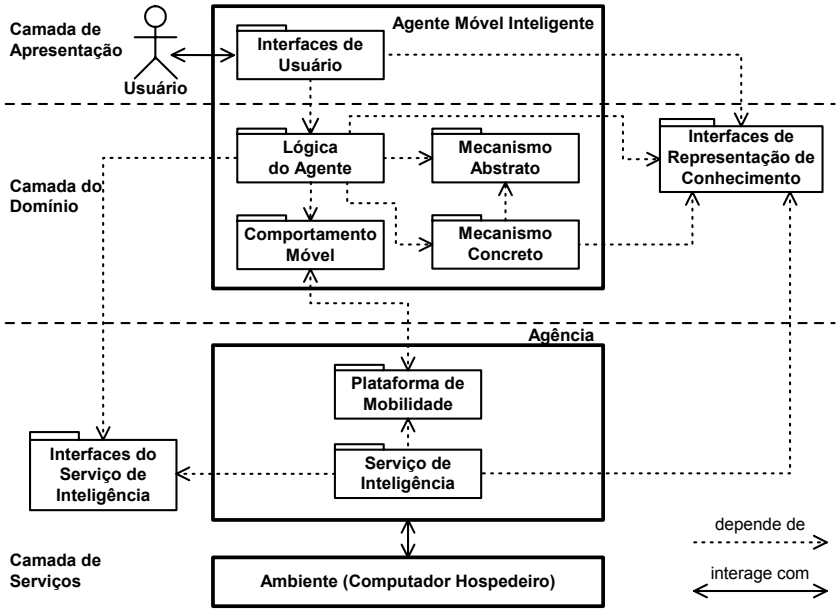

Figura 7. Arquitetura proposta para o *Framework de Inteligência*

A Figura 8 mostra o diagrama de classes para o núcleo do *framework*.

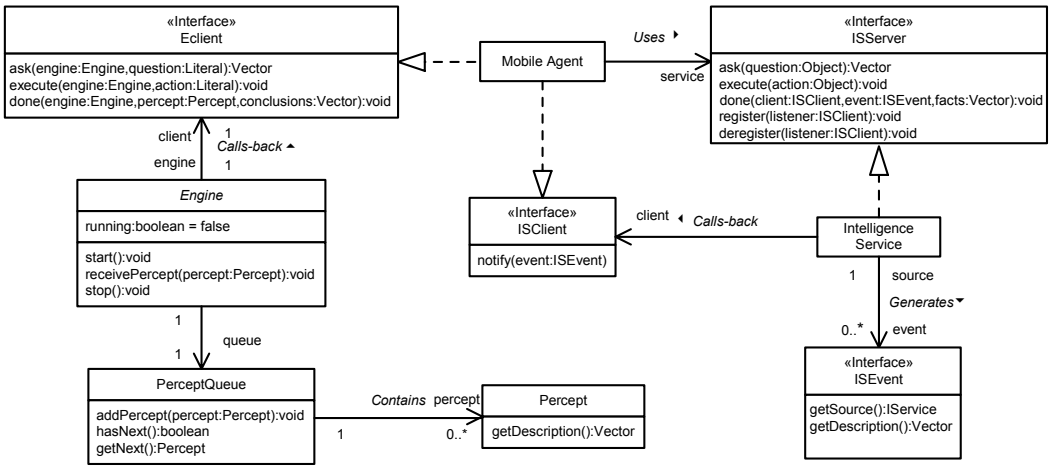

Figura 8. Diagrama de classes para o núcleo do *Framework*

4. Mecanismo de Inferência para Frente

A escolha do mecanismo de inferência baseada em regras com encadeamento para frente, como ponto de partida, deve-se a diversos fatores: (1) ele representa uma das técnicas mais conhecidas e estabelecidas de IA; (2) de acordo com diversos pesquisadores (IBM, 1997), esse tipo de mecanismo é requisito primário para a maioria dos sistemas inteligentes, podendo ser usado na configuração e no controle de outros mecanismos – portanto, seu desenvolvimento potencializa futuras incorporações; (3) os agentes móveis enfrentarão, com maior freqüência, a necessidade de selecionar suas ações com base tanto no estado atual de sua base de conhecimento quanto em suas percepções sobre seu ambiente – justamente o tipo de tarefa para a qual um procedimento de inferên-

cia com encadeamento para frente é o mais adequado; (4) vários tipos de conhecimento, como por exemplo, políticas corporativas de negócios e de segurança são, por natureza, expressas em regras; (5) o paradigma simbólico ainda é predominante em IA e, provavelmente, esse cenário ainda perdurará por algum tempo. Existem várias razões para isso. Talvez a mais importante seja que muitas técnicas de IA simbólica, como os sistemas baseados em regras, levam consigo uma metodologia e uma tecnologia associadas que estão se tornando familiares aos pesquisadores e engenheiros dos demais ramos da Ciência da Computação. Por exemplo, atualmente, as regras tornaram-se conceitos de primeira classe no desenvolvimento da *Web Semântica* (Berners-Lee et al., 2001).

4.1. Descrição

Um mecanismo de inferência para frente baseado em regras manipula uma base de conhecimento (BC) contendo *regras* e *fatos*. Em uma *etapa de inicialização*, as regras são transferidas para o mecanismo, que as traduz para sua representação interna e as armazena em sua *memória de regras*. Os fatos presentes na BC original também são traduzidos e armazenados em uma *memória de trabalho*. A seguir, durante a *etapa de execução*, toda vez que a memória de trabalho é modificada pelo acréscimo de um novo fato ou pela remoção de um fato já existente, é executado um *ciclo de inferência*, composto por três fases. Na *fase de casamento* (*matching*) o mecanismo encontra um subconjunto de regras cujas premissas são todas satisfeitas pelos fatos presentes na memória de trabalho. Cada par composto por uma regra e pelos fatos que casaram com suas premissas forma uma *ativação*, que é armazenada em um *conjunto de conflitos* (*conflict set*). Na próxima fase, *resolução de conflitos*, o mecanismo seleciona uma ativação dentre as presentes no conjunto de conflitos. Esta seleção é baseada em uma *estratégia de resolução de conflitos*. Finalmente, na fase de ação, o mecanismo executa a ação associada à regra da ativação escolhida. Para *sistemas puramente lógicos*, a única ação possível é a atualização da memória de trabalho, pela adição ou remoção de um fato. *Mecanismos situados* permitem que procedimentos externos (*effectors*) sejam chamados nessa fase, não se limitando à atualização da memória de trabalho. Além disso, os mecanismos situados também permitem que procedimentos externos (*sensors*) sejam chamados durante a fase de casamento, quando as premissas estão sendo avaliadas.

4.2. Representação de Conhecimento

Assumiu-se que o mecanismo de inferência é dirigido por regras na forma:

$$E \wedge P_1 \wedge \ldots \wedge P_m \wedge S_i \wedge \ldots \wedge S_n \rightarrow Q_1 \wedge \ldots \wedge Q_k \wedge A_1 \wedge \ldots \wedge A_r$$

onde, ~ representa a *negação por falha*, $\wedge$ o conectivo lógico "E" e:

```
m, n, k, r ≥ 0
E     →   NomeEvento(Termo,...);
Pi    →   Predicado(Termo,...) | ~Predicado(Termo,...);
Qi    →   Predicado(Termo,...) | ~Predicado(Termo,...);
Si    →   NomeSensor(Termo,...) | ~NomeSensor(Termo,...);
Ai    →   NomeAtuador(Termo,...)
```

A Figura 9 mostra as interfaces criadas para os conceitos da *Lógica de Predicado de Primeira Ordem* aplicáveis. As operações dessas interfaces permitem o acesso aos componentes constitutivos de cada uma. A informação de que um predicado representa um *evento*, um *sensor* ou um *atuador*, é associada a ele como um adorno.

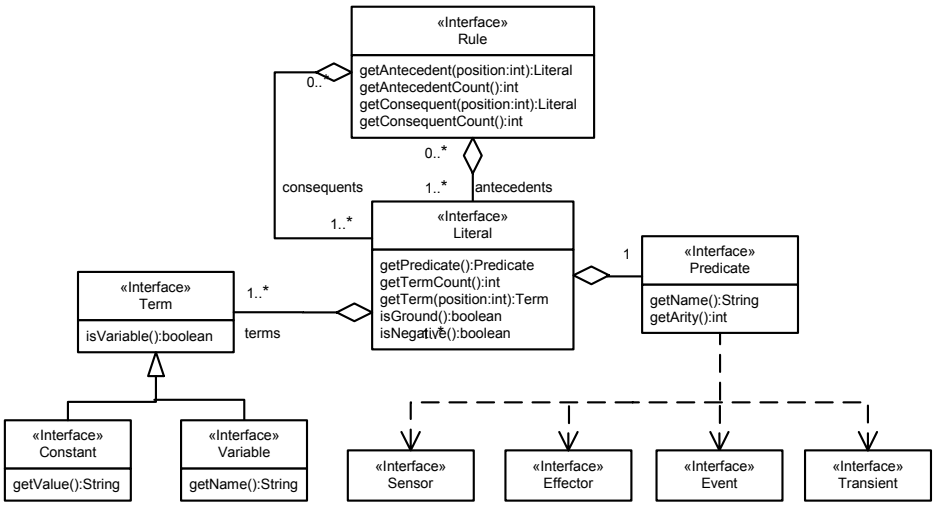

Figura 9. Diagrama de classes para a Representação de Conhecimento

4.3. Mecanismo de Inferência

Para satisfazer R10, as principais funcionalidades do mecanismo foram atribuídas a componentes distintos, que encapsulam seus principais algoritmos (Figura 10).

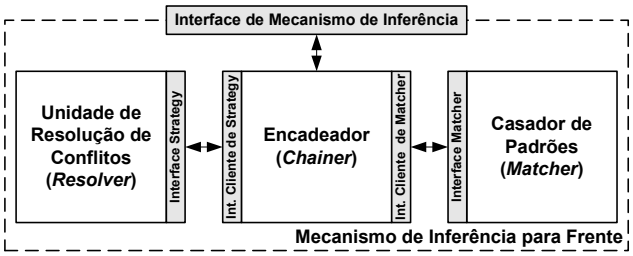

Figura 10. Principais componentes do mecanismo de inferência para frente

O agente utiliza a *Interface de Mecanismo Inferência* para configurar e controlar a execução do processo de inferência. A separação da lógica de controle da inferência (*Encadeador*) dos algoritmos de casamento e de resolução de conflitos satisfaz R10 e contribui para a redução do código e dos dados transportados pelo agente (R7), permitindo diversas combinações. Por exemplo, por ser genérico, o *Encadeador* poderá ser oferecido pela agência hospedeira, enquanto que o agente leva consigo somente os seus próprios *Casador de Padrões* e a *Unidade de Resolução de Conflitos*. A Figura 11 mostra o diagrama de classes para o mecanismo de inferência.

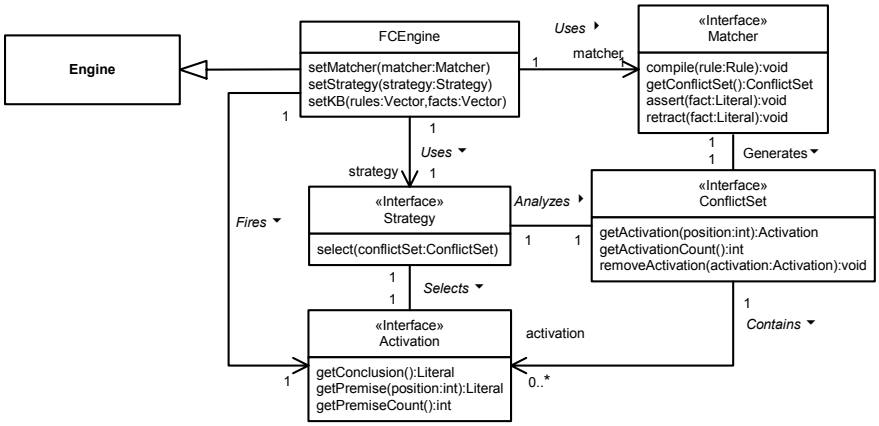

Figura 11. Diagrama de classes para o Mecanismo de Inferência

4.4. Casador de Padrões

Um *Casador de Padrões* de acordo com o algoritmo Rete (Forgy, 1982) foi incluído como parte do *framework*. Esse algoritmo possui duas fases distintas: *compilação* e *casamento*. Na primeira, um conjunto de regras é transformado em uma rede de *nós* interligados, cada um dos quais aplica um ou mais *testes* à expressão antecedente de uma regra. Na segunda, quando um fato é adicionado ou removido da memória de trabalho do mecanismo, ele é processado pelos nós e, caso atinja o nó terminal de uma regra, a expressão conseqüente dessa regra é verdadeira e a regra é *ativada*. A estratégia de resolução de conflito escolhe, dentre as diversas regras ativadas, qual deve ser executada. Para permitir a solicitação de informações adicionais durante o processo de inferência, foi criado um tipo de nó dedicado chamado `SensorNode`.

A utilização do *Rete* contribui para aliviar o custo computacional do processo (R7) e elimina a duplicidade de estruturas de dados entre regras (Friedman-Hill, 2001), atendendo a R8. Como as regras são compiladas em uma rede de nós, ele também satisfaz o requisito R9. Para reduzir ainda mais a quantidade de código e de dados, o *Casador de Padrões* foi decomposto nos componentes mostrados na Figura 12.

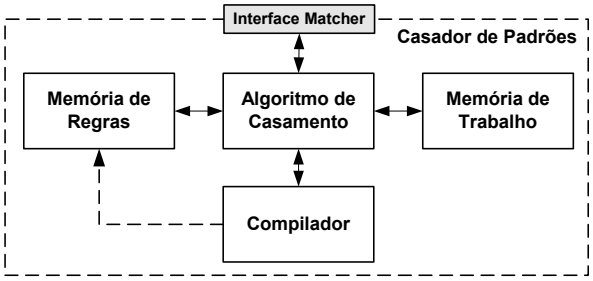

Figura 12. Principais componentes de um *Casador de Padrões*

O código e os dados do *Compilador* não precisam migrar, pois sua funcionalidade só é necessária durante a configuração do agente, em sua agência nativa. A decomposição também permite identificar operações e atributos estritamente relacionados com a fase de casamento, facilitando o projeto de estruturas mais "leves", que possam remover, automaticamente, fatos que sejam exclusivamente dependentes da agência hospedeira. Para tanto, durante o processo de compilação, todos os antecedentes de uma regra, que representem tais fatos, são marcados pelo *Compilador* como *transientes*.

5. Resultados

O *framework* proposto foi implementado utilizando-se o JDK 1.1.8 da *Sun Microsystems*. Todas as classes e interfaces foram feitas totalmente serializáveis. AMIs criados com os sistemas *Aglets* (IBM, 2002) e *Grasshopper* (IKV, 2002) foram colocados para migrar em um ambiente composto por 3 computadores (padrão IBM PC), executando o sistema operacional Windows NT 4.0 da *Microsoft*. Para verificar a correta mobilidade das classes e interfaces do *framework*, bem como o custo de migração de seu código e de seus dados, desenvolveu-se uma aplicação composta por três tarefas, cada uma a ser realizada em um computador distinto do ambiente, mas dependente de informações coletadas nas máquinas já visitadas e armazenadas na memória de trabalho do agente. Ao chegar em uma máquina, o agente inicia o mecanismo de inferência e realiza a tarefa associada àquela máquina. A execução correta do mecanismo e da tarefa demonstrou que todas as classes e interfaces do *framework* são transportadas corretamente. Antes de cada migração, a aplicação também salva em arquivo os dados serializados a serem

transportados, permitindo uma avaliação desse custo. A quantidade de código transportada foi obtida somando-se os tamanhos de todos os arquivos `.class` que migram. A Tabela 1 mostra o tamanho do código (não comprimido) dos subsistemas do *framework*.

Tabela 1. Tamanho do código dos subsistemas do *Framework de Inteligência.*

Subsistema	Código (bytes)
Mecanismo de Inferência	9.926
Rete: Algoritmo de Casamento+Memória de Regras + Memória de Trabalho	24.148
Rete: Compilador	15.947
Representação do Conhecimento	5.325
Interfaces do SI	874

Os três últimos subsistemas dessa tabela não precisam migrar: o *Compilador* só é necessário na agência nativa do agente e os dois últimos devem estar instalados como classes de sistema nas agências a serem visitadas. Logo, a quantidade de código transportada é de apenas 34.074 bytes ou, se as classes do mecanismo de inferência também já estiverem instaladas nas agências, 24.148 bytes.

A quantidade de dados transportados depende dos dados encapsulados nos objetos dos elementos *Mecanismo de Inferência, Algoritmo de Casamento*, *Memória de Regras* e *Memória de Trabalho*. A quantidade de dados presentes nos dois primeiros é fixa, para uma dada implementação. Porém, os dados presentes nos dois últimos são dependentes do número de regras e de fatos na BC. Portanto, a quantidade total de dados a ser transportada é dependente da aplicação pretendida. No entanto, é possível levantar-se os custos "individuais" de serialização das classes que os compõem. Os custos de serialização dos dados das classes dos diversos tipos de nós e de testes utilizados na estrutura do `Rete` representam os custos individuais elementos da *Memória de Regras*. O custo de serialização da *Memória de Trabalho* é constituído pelos custos individuais dos objetos das classes `Fact` e `Conjunction` contidos nas memórias dos *nós-de-duas-entradas*. A Tabela 2 mostra esses custos.

Tabela 2. Custo de serialização de regras e fatos no *Rete.*

	Classe	Custo de Serialização (bytes)
Memória de Regras	`Node1`	40
	`Node2`	110
	`SensorNode`	$71 + L_P + \sum(9 + L_{Ti}) + 18\ N_V$
	`Terminal`	$85 + L_P + \sum(9 + L_{Ti}) + 18\ N_V$
	`TestPredicate`	$4 + L_P$
	`TestConstant`	$8 + L_P$
	`TestVariable`	9
	`TestInterLiterals`	23
Memória de Trabalho	`Fact`	$16 + \sum(3 + L_{Ti})$
	`Conjunction`	40

Em cada expressão, L_P é o número de caracteres do predicado, L_{Ti} é o número de caracteres de seu i-ésimo termo, N_V é o número de seus termos do tipo variável, e o somatório é realizados sobre o seu número total de seus termos. A redução no tamanho do código e dos dados obtida é evidente quando se compara, para um dado conjunto de regras e fatos, os custos do *framework* proposto com os do JESS (Tabela 3).

Tabela 3. *Framework de Inteligência versus* JESS (bytes).

		Framework	JESS
Código	Mecanismo+Estratégia+Casador (fixo)	34.074	452.758
Dados	Mecanismo+Estratégia+Casador (fixo)	3.047	19.009
	Memória de Regras (variável)	987	3.991
	Memória de Trabalho (variável)	294	1.524
Total		**38.392**	**477.282**

6. Conclusões

A principal contribuição deste trabalho é demonstrar que é possível construir um *framework* para adicionar mecanismos de IA a agentes móveis. A segunda é o *framework* em si, que, devido à sua flexibilidade, permite ampliar os atuais SAMs baseados em Java. Além disso, a definição cuidadosa das interfaces dos componentes do *Mecanismo de Inferência* e do *Casador* possibilita o desenvolvimento de elementos personalizados. A partir dos AMIs desenvolvidos para testar o *framework*, concluiu-se que a programação declarativa simplifica a simulação da migração forte, usando-se a migração fraca, uma vez que não é necessário dividir o código do agente em blocos e manter a informação de qual bloco deve ser executado em cada agência. Atualmente, um cenário de coleta de informações na *Web* está sendo desenvolvido para comprovar a aplicabilidade da abordagem proposta e avaliar em que situações agentes móveis, munidos do mecanismo de inferência, suplantam a abordagem tradicional baseada na programação procedimental.

Referências

Berners-Lee, T.; Hendler, J. and Lassila, O. (2001) "The Web Semantic", Scientific American, p. 29-37, May.

Bigus, J. P. and Bigus, J. (1998) "Constructing intelligent agents with Java: a programmer's guide to smarter applications", John Wiley & Sons, New York.

Cetus (2002) "Distributed Objects & Components: Mobile Agents", http://cetus-links.org/oo_mobile_agents.html, May.

Forgy, C. L. (1982) "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Artificial Intelligence, v. 19, p. 17-37.

Friedman-Hill, E. J. (2001) "Jess, The Java System Shell Version 6.03a", http://herzberg.ca.sandia.gov/jess/readme.html, December.

Gamma, E.; Helm, R.; Johnson, R. and Vlissides, J. (1995) "Design Patterns Elements of Reusable Object-Oriented Software". Addison Wesley, New York.

IBM (1997) "Overview of IBM Agent Building Environment Developer's Toolkit Level 6", http://www. networking.ibm.com/iag/iaghome.html, June.

IBM (2000) "Overview of the Agent Building and Learning Environment", http://www. alphaworks.ibm.com/, September.

IBM (2002) "Aglets Home Page", http://www.trl.ibm.com/aglets/, September.

IKV (2002) "Grasshopper Home Page", http://www.ikv.de/grasshopper/, January.

Ismail, L. and Hagimont, D. (1999) "A Performance Evaluation of the Mobile Agent Paradigm", In: Proc. of Conference on Object-Oriented Programming, Systems, Languages, and Applications, Denver, p. 306-313.

Nwana, H., S. and Ndumu, D. T. (1999) "A Perspective on Software Agents Research", Knowledge Engineering Review, v. 12, n. 2, p. 1-18

Silva, L. M. et al. (1999) "The Performance of Mobile Agent Platforms", In: Proc. of 1st International Symposium on Agent Systems and Applications and 3rd International Symposium on Mobile Agents, Palm Springs.