3
Separação dos Interesses de Mobilidade

Este capítulo apresenta a primeira contribuição desta dissertação: uma análise das soluções existentes com relação à eficácia destas para a separação dos interesses de mobilidade. As questões referentes à mobilidade têm sido estudadas sob diferentes enfoques, os quais incluem usualmente padrões de projeto (Aridor & Lange, 1998), plataformas e *frameworks* de agentes móveis (Lange, 1998; Bellifemine et al., 1999; Tripathi et al., 1999), e extensões de linguagens de modelagem (Klein et al., 2001; Muscutariu & Gervais, 2001; Mouratidis et al., 2002; Belloni & Marcos, 2004). Entretanto, tais abordagens não possuem como objetivo central a separação explícita do código de mobilidade em relação aos outros interesses de uma aplicação baseada em agentes. Em geral, as plataformas e *frameworks* existentes aliviam parte do problema, uma vez que tornam transparentes à aplicação várias questões associadas à mobilidade de código. Infelizmente, tais soluções não são suficientes para prover a modularização dos interesses de mobilidade, o que dificulta a legibilidade, manutenibilidade e reusabilidade da aplicação. De fato, as soluções existentes impõem uma série de restrições à aplicação, como será apresentado no decorrer deste capítulo.

De acordo com nosso conhecimento, somente um trabalho na literatura de agentes móveis procurou demonstrar uma abordagem alternativa através da qual se torna possível dar suporte à modularização dos interesses de mobilidade, bem como permitir a introdução transparente de mobilidade nas aplicações. Trata-se da abordagem *RoleEP* (*Role-Based Evolutionary Programming* ou Programação Evolucionária Baseada em Papéis) de Ubayashi e Tamai implementada pelo *framework Epsilon/J* dos mesmos autores (Ubayashi & Tamai, 2001). Mesmo objetivando dar solução a estes dois problemas de SMAs móveis, esta abordagem ainda não permite a flexibilização no uso de diferentes plataformas de mobilidade.

Neste capítulo, são analisadas três abordagens diferentes para a implementação de interesses de mobilidade em uma aplicação: uso direto das APIs de plataformas de mobilidade (Seção 2.6), uso de padrões de projeto para

permitir melhor estruturação dos elementos de mobilidade específicos da aplicação e uso da abordagem *RoleEP*. São apontadas as vantagens e desvantagens de cada abordagem através de um estudo de caso que envolve o uso de mobilidade. Tal estudo também será utilizado no Capítulo 7 para demonstrar a aplicabilidade de nossa solução para o problema de separação de mobilidade.

3.1. Descrição do Estudo de Caso

Expert Committee (EC) é um sistema multi-agentes aberto para suporte ao gerenciamento de submissões e revisões de artigos submetidos a uma conferência (Deloach et al., 2001; Zambonelli et al., 2001). Agentes de software foram introduzidos no EC para assistir os pesquisadores e a comissão de organização de um evento em tarefas que fazem parte do processo de revisão e submissão e que podem ser automatizadas. O sistema faz uso de dois tipos de agentes: agentes de informação e agentes de usuário. Vários papéis (Seção 2.1) podem ser atribuídos a cada agente de usuário, mas para os propósitos deste trabalho são importantes os papéis de revisor e *chair*. Agentes de usuário desempenham estes papéis a fim de colaborar uns com os outros. A Figura 8 apresenta classes do EC representando tipos de agentes (`ResearcherAgent`, `InformationAgent`), papéis (`Reviewer`, `Chair`) e planos (`CVUpdatePlan`, `DistributionPlan`). O papel *chair* está associado ao plano para distribuição de propostas de revisão (Figura 8).

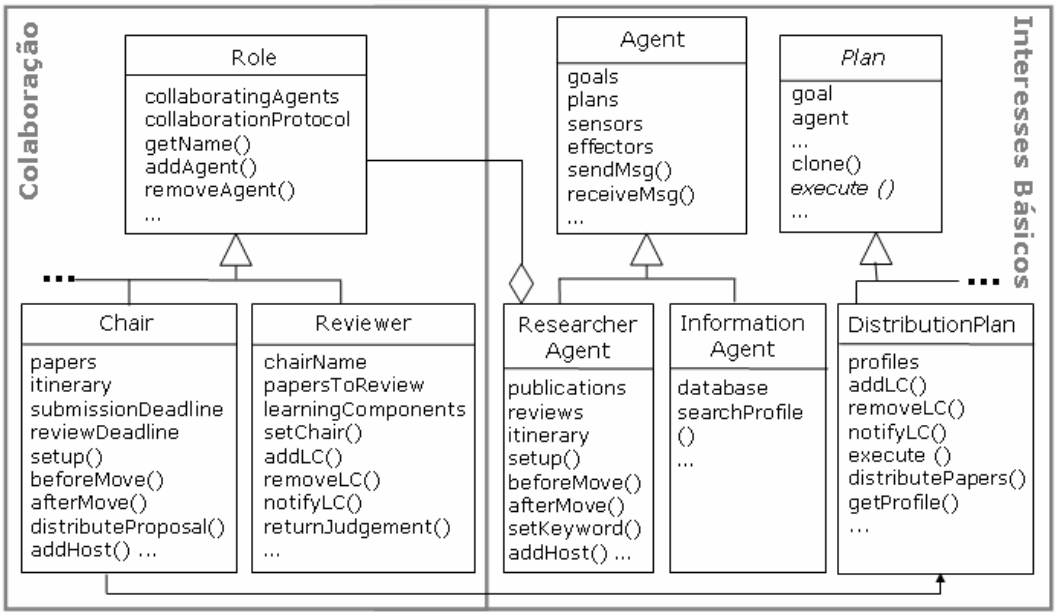

Figura 8. Tipos de Agentes, Papéis e Planos no Expert Committee

Os agentes de usuário do EC precisam se movimentar em algumas circunstâncias, inclusive quando estão desempenhando papéis específicos. Por exemplo, quando um agente de usuário está desempenhando o papel de *chair*, é necessário consultar dados sobre revisores a fim de otimizar a distribuição de propostas de revisão de acordo os interesses de pesquisa de cada revisor. Se informações sobre um revisor não estão disponíveis, o *chair* colabora com um agente de informação e solicita a este agente a busca de informação sobre os interesses do revisor especificado.

O agente de informação controla o banco de dados local e é capaz de realizar as buscas no banco. Contudo, se a informação não está disponível no banco local, o papel *chair* deve se movimentar para ambientes remotos a fim de encontrar a informação sobre os interesses de pesquisa do revisor. O agente desempenhando o papel *chair* então se movimenta pelos *hosts* da rede, delegando a tarefa de busca aos agentes de informação espalhados pela Internet.

No cenário de movimentação do agente desempenhando o papel de *chair*, é fundamental que os interesses de mobilidade e os outros interesses do sistema, como as funcionalidades básicas e atividades de colaboração, sejam especificados tão separadamente quanto possível. Nas próximas seções, analisamos três abordagens que podem ser utilizadas para a separação dos interesses de mobilidade em sistemas como o EC.

3.2. APIs de Plataformas

Para o desenvolvimento de SMAs, os desenvolvedores têm se apoiado vastamente na utilização de APIs OO de plataformas de mobilidade conhecidas. Apesar da existência de várias destas plataformas, estas somente permitem a introdução de mobilidade em SMAs a partir de práticas inerentemente intrusivas. As APIs impõem restrições arquiteturais ao projeto de agentes, as quais ocasionam os problemas de espalhamento e entrelaçamento do código de mobilidade. Por exemplo, a fim de adicionar a característica de mobilidade aos agentes, os desenvolvedores geralmente devem modificar o projeto de sistemas:

(1) declarando que os agentes dos sistemas estendem classes de agentes pertencentes às APIs das plataformas de mobilidade;
(2) implementando métodos abstratos declarados em classes das plataformas de mobilidade;
(3) declarando e, possivelmente, definindo a implementação de interfaces pertencentes às APIs das plataformas de mobilidade; e
(4) invocando explicitamente métodos das APIs em classes dos sistemas que implementam outros interesses de agentes que não a mobilidade.

A Figura 9 ilustra parcialmente o projeto do sistema EC com o objetivo de demonstrar as restrições arquiteturais impostas pelas APIs OO das plataformas de mobilidade ao projeto de SMAs. Para efeito de simplificação, a Figura 9 mostra apenas as classes mais importantes, uma vez que as outras seguem essencialmente o mesmo padrão apresentado. Também são omitidos métodos e classes referentes a outras características geralmente presentes em SMAs.

Na Figura 9, cada conjunto de classes circundado por um retângulo com linha cinza possui como finalidade principal a modularização de um interesse específico do agente de software, a saber, interação, colaboração, mobilidade e interesses básicos. Entretanto, é fácil notar que os interesses de mobilidade ainda entrecortam classes implementando outros interesses do agente. De fato, esta intrusão dos interesses de mobilidade em outros interesses do sistema tem forte impacto na estrutura básica do agente, bem como no projeto dos interesses de colaboração e interação do sistema. Em outras palavras, embora parte dos interesses de mobilidade esteja localizada em classes internas ao retângulo de mobilidade, tais como `JADEAgent` e `Serializable`, o código de mobilidade está inevitavelmente replicado e espalhado através de várias hierarquias de classes do agente de software ilustrado. Cada problema de espalhamento ou entrelaçamento referente à mobilidade é indicado através de um número no interior de um pequeno círculo (Figura 9).

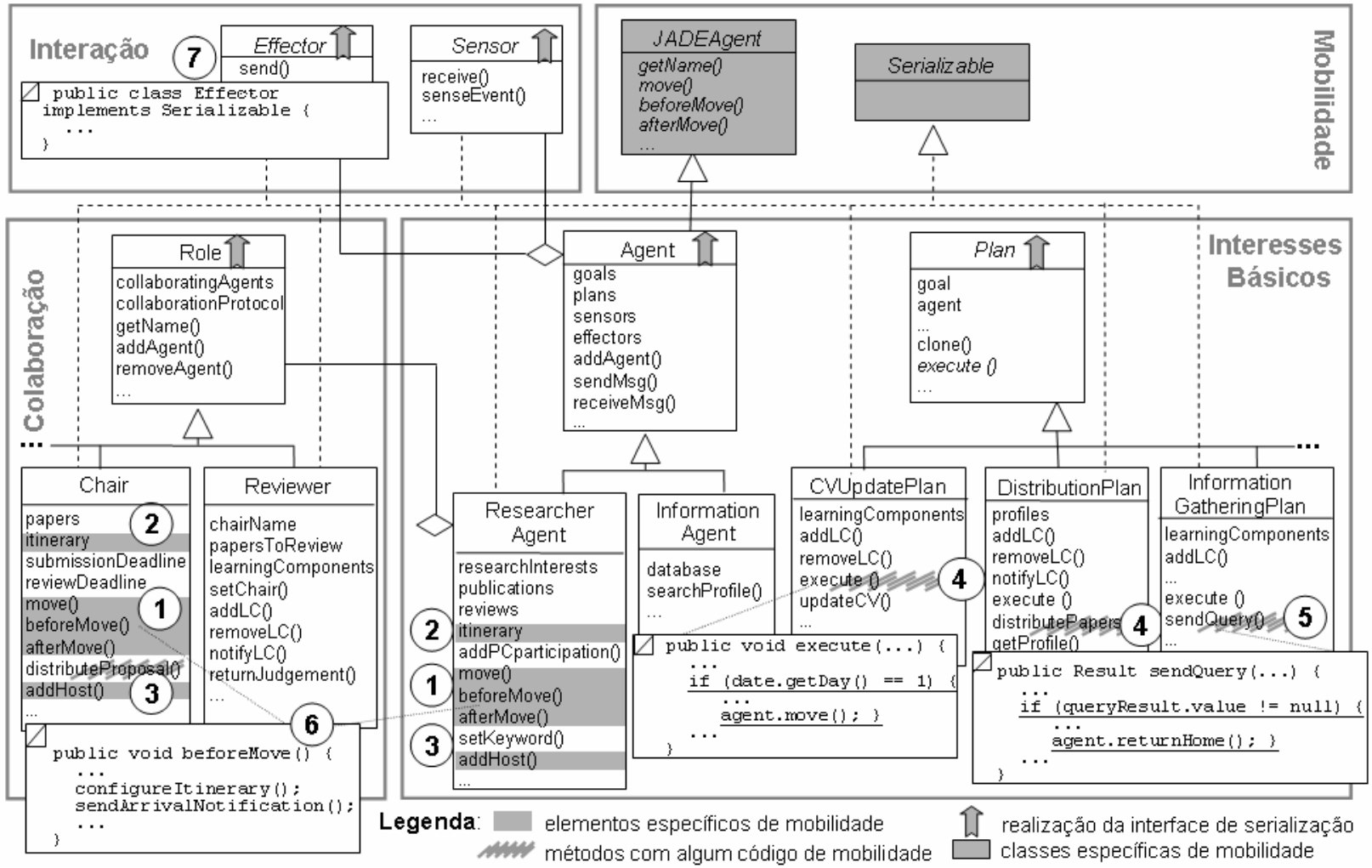

Figura 9. Solução para EC baseada somente em APIs

Na Figura 9, existem várias classes que representam tipos e papéis de agentes. Para que estes tipos e papéis tenham a capacidade de mobilidade, suas classes correspondentes devem estender a classe abstrata `JADEAgent`. Entretanto, o uso do mecanismo de herança nesta situação resulta em replicação e entrelaçamento de código: a mobilidade do agente está implementada em meio às funcionalidades básicas e às atividades de colaboração do agente (problema ①).

As classes referentes aos papéis e tipos de agentes também precisam manter atributos específicos para o tratamento das funções de mobilidade, como, por exemplo, o atributo `itinerary` (problema ②). Conseqüentemente, as classes do sistema também devem dispor de métodos adicionais específicos para a manutenção destes atributos; por exemplo, métodos para efetuar a manutenção do itinerário do agente, como `addHost()` (problema ③).

Adicionalmente, vários métodos do sistema possuem código referente à mobilidade a fim de *decidir* se o agente deve ou não se movimentar para um *host* remoto (problema ④), ou ainda, para *decidir* se o agente deve ou não retornar ao *host* original (problema ⑤). Pode haver ainda o espalhamento de pré e pós-condições que usualmente devem ser satisfeitas quando um determinado agente se move para outro ambiente (problema ⑥). Tais restrições podem ser genéricas, isto é, independentes de agente/aplicação, ou dependentes de circunstâncias. Finalmente, existe ainda a necessidade de que várias classes declarem a implementação da interface `Serializable`, a fim de que os objetos instanciados a partir destas possam ser movidos junto com os agentes (problema ⑦).

Vale ressaltar que, no caso do problema ⑦, a interface `Serializable` é apenas um exemplo: as APIs OO das plataformas de mobilidade geralmente disponibilizam várias interfaces com métodos que são implementados pelos sistemas a fim de que planos e ações de um agente possam ser executados automaticamente em momentos específicos de seu ciclo de vida; por exemplo, planos e ações de um agente podem ser executados automaticamente antes, durante ou após a criação e movimentação de agentes.

Para todos os problemas descritos, o código de mobilidade encontra-se replicado em vários pontos do código, espalhado no corpo de vários métodos de planos, papéis e tipos de agente. Além disso, as restrições impostas pelas APIs

OO de plataformas de mobilidade também introduzem problemas conceituais na especificação de SMAs.

Um primeiro exemplo envolve a classe `Agent`, superclasse de todos os agentes do sistema ilustrado na Figura 9. A classe `Agent` deve estender a classe `JADEAgent`, mesmo quando determinados tipos de agentes na hierarquia de classes são estacionários, como é o caso de `InformationAgent`. Não é possível definir que a classe `ResearcherAgent` (Figura 9) é subclasse direta de `JADEAgent`, pois a classe `ResearcherAgent` já estende a classe `Agent`. Isto ocorre porque a maioria das linguagens de programação OO, tais como Java, não provê suporte à definição de herança múltipla. O mesmo problema se repete quando os desenvolvedores necessitam especificar que somente alguns papéis de agentes são móveis. Por exemplo, a classe `Chair` na Figura 9 especifica um papel de agente móvel, enquanto a classe `Reviewer` define um papel de agente estacionário.

Além deste problema conceitual, o uso de plataformas de mobilidade favorece ainda a existência de conflitos potenciais. Por exemplo, a classe `JADEAgent`, que se refere à classe de agentes da plataforma de mobilidade sendo utilizada, define um método abstrato `getName()`. Ocorre que a classe `Agent`, pertencente ao SMA, também define um método `getName()`, mas com uma finalidade diferente do método correspondente na classe `JADEAgent`. Assim, a introdução de mobilidade pode requerer modificações no projeto e na implementação de SMAs, como renomeação de métodos e conseqüentes mudanças nas chamadas a estes.

3.3. Padrões de Projeto

As limitações de trabalhos relacionados apresentadas na Introdução e na seção anterior deste capítulo se referem àquelas que decorrem do uso direto de APIs OO de plataformas para solucionar o problema de separação dos interesses de mobilidade em SMAs. Entretanto, além do uso direto de APIs, uma outra abordagem apresentada como solução para os problemas de entrelaçamento e espalhamento da mobilidade em SMAs é a utilização de padrões de projeto em conjunto com as plataformas de mobilidade.

Padrões de projeto com foco em SMAs móveis foram propostos por Aridor e Lange (1998). Tais padrões são classificados em: (i) padrões de movimentação – *Itinerary*, *Forwarding*, *Ticket*, etc.; (ii) padrões de plano de tarefas – *Master-Slave*, *Plan*, etc.; (iii) padrões de colaboração – *Meeting*, *Locker*, *Messenger*, *Facilitator*, etc. Para a análise do uso de padrões de projetos em SMAs móveis, tomamos como exemplo os padrões *Master-Slave* e *Itinerary*.

Master-Slave é um dos padrões mais simples e fundamentais do projeto de SMAs móveis. Através deste padrão, um agente mestre[12] pode delegar uma tarefa específica a um agente escravo[13], que é então enviado a um *host* destino onde pode executar a tarefa que lhe foi atribuída. O agente escravo posteriormente retorna ao *host* original com os resultados encontrados (Aridor & Lange, 1998).

Existem pelo menos duas situações onde este padrão pode ser aplicado: (1) quando, por questões de performance, é necessário que as tarefas dos agentes mestre e escravo sejam executadas simultaneamente, possivelmente em *hosts* separados, ou (2) quando um agente estacionário precisa executar em um *host* remoto uma tarefa que lhe foi solicitada. Neste último caso, o agente mestre estacionário delega a execução da tarefa ao agente escravo, que possui a capacidade de se mover para outros *hosts* de uma rede.

A idéia principal é utilizar as classes abstratas `Master` e `Slave` para especificar os papéis dos agentes no cumprimento de uma tarefa: retornar um escravo, ou enviá-lo a um outro *host,* iniciar a execução da tarefa, tratar exceções que ocorrem durante a execução, etc. O diagrama da Figura 10 ilustra a colaboração entre as classes do padrão *Master-Slave*.

Outro padrão de projeto fundamental em SMAs móveis é o padrão *Itinerary*, que trata das questões relacionadas ao itinerário de um agente e seu roteamento através de múltiplos *hosts* em uma rede. O padrão *Itinerary* possui como funções principais: (1) manter uma lista de destinos a visitar, (2) definir um esquema de roteamento através dos *hosts* da rede, (3) tratar exceções como o que fazer se um determinado destino não é encontrado ou fazer o retorno do agente a destinos anteriormente visitados, e (4) indicar o próximo *host* da rede para onde o agente deve ser transferido (Aridor & Lange, 1998).

12 Do inglês: *master*.
13 Do inglês: *slave*.

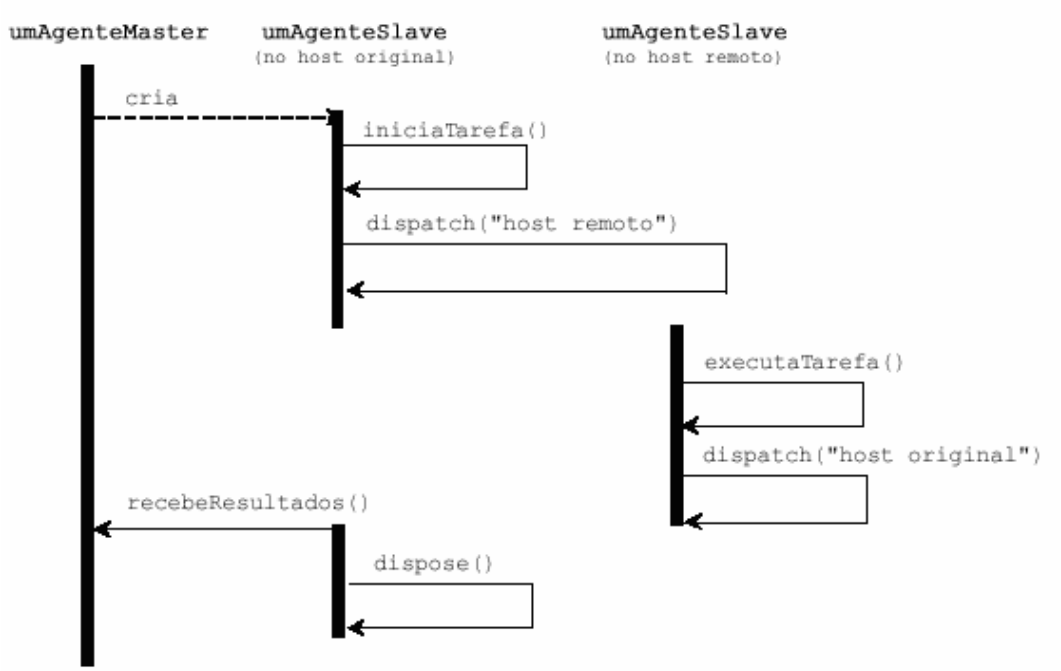

Figura 10. Colaboração de Classes no Padrão Master-Slave

O padrão *Itinerary* desloca a responsabilidade de navegação do objeto agente para um objeto à parte, instanciado a partir de uma classe denominada `Itinerário`. O objeto agente cria o objeto itinerário e o inicia com dois argumentos: uma lista de destinos a serem visitados e uma referência que mantém a ligação entre os dois objetos. Através do método `go()` do objeto itinerário, o agente se move para o próximo destino especificado. O objeto itinerário é transferido sempre em conjunto com o objeto agente e a referência entre os dois deve ser mantida em cada destino da rede. A Figura 11 ilustra a colaboração de classes no padrão *Itinerary*. A Figura 12 demonstra a utilização do padrão *Itinerary* no sistema EC.

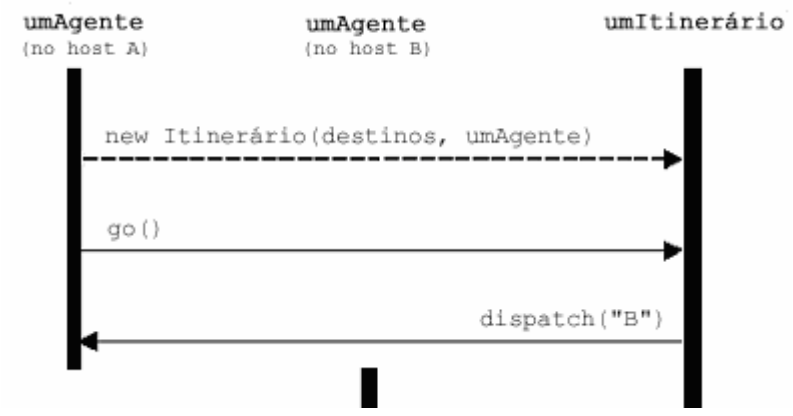

Figura 11. Colaboração de Classes no Padrão Itinerary

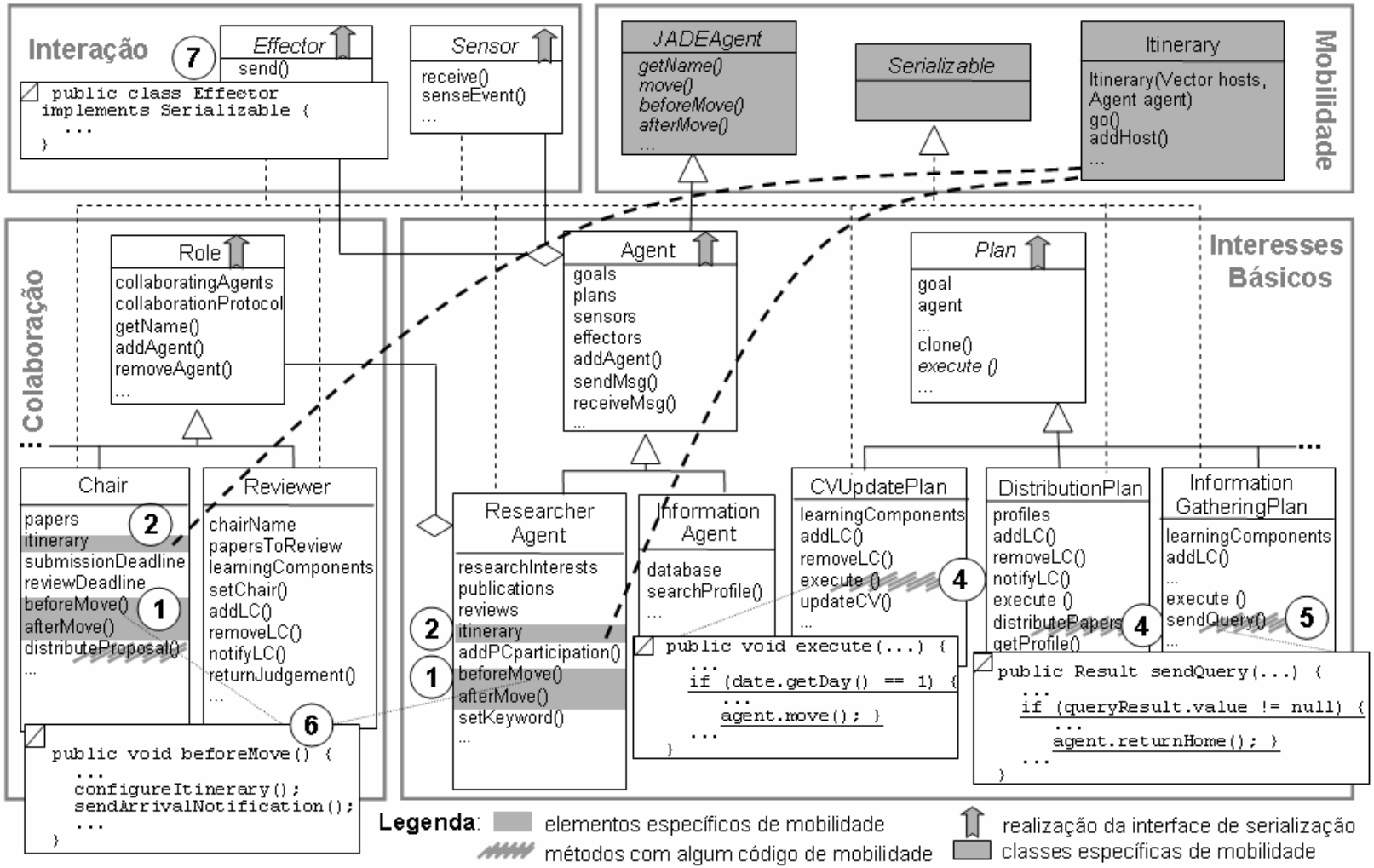

Figura 12. Solução para EC baseada em APIs e Padrões de Projeto

Na Figura 12, todas as informações sobre itinerário de tipos e papéis de agentes estão encapsuladas na classe *Itinerary*. Note que os métodos específicos para tratamento de itinerário, como por exemplo, `addHost()` (problema ③), não estão mais espalhados e entrelaçados nas classes `Chair` e `ResearcherAgent`. Note também que nestas classes não há mais uma invocação direta ao método `move()`, que dispara o movimento do agente na plataforma de mobilidade; a invocação de `move()` é delegada ao método `go()` da classe `Itinerary`.

Ubayashi e Tamai (2001) descrevem a aplicação de padrões no projeto de SMAs móveis como uma das soluções para o problema de mobilidade. Entretanto, uma vez que as classes de agentes móveis, mesmo usando padrões de projeto como *Itinerary*, ainda estendem a classe de agentes de uma plataforma de mobilidade específica, algumas restrições arquiteturais ainda são naturalmente impostas ao projeto de agentes, como: (i) declarar que os agentes dos sistemas estendem classes de agentes pertencentes às APIs; (ii) implementar métodos abstratos declarados em classes das plataformas de mobilidade; (iii) definir a implementação de interfaces pertencentes às APIs; (iv) invocar explicitamente métodos das APIs em classes dos sistemas que implementam outros interesses de agentes que não a mobilidade.

Portanto, os padrões de projeto possuem limitações em modularizar os interesses de mobilidade de SMAs. As funções de mobilidade continuam entrelaçadas com as funcionalidades básicas dos agentes. Se os agentes de software possuem múltiplos papéis e ações de mobilidade, o código destes agentes tende a se tornar muito complexo, pois as chamadas às funções de mobilidade se encontram espalhadas em várias classes do sistema. Estes problemas não são devidos à estrutura ou à semântica dos padrões de projeto, mas à dependência mantida entre estes padrões e as APIs OO de plataformas de mobilidade.

3.4. Abordagem *RoleEP*

Além do uso de padrões de projeto em conjunto com plataformas de mobilidade, a abordagem *RoleEP* (Ubayashi & Tamai, 2001), implementada pelo *framework Epsilon/J*, procura dar suporte à introdução transparente de mobilidade em aplicações. Esta abordagem pode ser descrita a partir da definição dos dois

problemas que se propõe a resolver: (1) dificuldade de entendimento das funções de colaboração e mobilidade de agentes como um todo, porque tais interesses se apresentam entrelaçados no código de SMAs, e (2) dificuldade de especificar comportamentos de agentes explicitamente, uma vez que estes são dinamicamente influenciados por contextos externos. Para resolver estes problemas, a abordagem *RoleEP* fornece: (1) um mecanismo para separação dos interesses de mobilidade/colaboração nas aplicações, e (2) um estilo de programação que permite a evolução sistemática e dinâmica de SMAs móveis. Nesta seção, são apresentados os principais conceitos de *RoleEP* e do *framework Epsilon/J*.

3.4.1. Conceitos Básicos

O modelo proposto pela abordagem *RoleEP* é composto de conceitos próprios, incluindo *agentes*, *papéis*, *objetos* e *ambientes*, como ilustrado na Figura 13. Para um melhor entendimento destes conceitos, as funcionalidades dos agentes são classificadas em (Ubayashi & Tamai, 2001): (i) *funções de mobilidade*, os agentes podem movimentar-se entre *hosts*; (ii) *funções de colaboração*, os agentes podem colaborar com outros agentes no mesmo ambiente através de troca de mensagens, e (iii) *funções básicas*, comuns a todos os tipos de ambientes e não contendo referências a funções de mobilidade/colaboração. *Ambientes* e *papéis* são os conceitos do modelo associados às funções de mobilidade e colaboração; o conceito de *objetos* está associado às funções básicas.

Na Figura 13, a separação de colaboração e mobilidade é ilustrada através de contornos separados e nomeados pelas expressões "Ambiente para colaboração" e "Ambiente para mobilidade". Em *RoleEP*, um *ambiente* é composto de atributos, métodos e papéis. Um *papel* é composto de atributos, métodos e interfaces de ligação[14]. Atributos e métodos de um papel estão disponíveis apenas no ambiente a qual o papel pertence. Uma *interface de ligação*, similar à interface de um método abstrato, é usada quando um objeto se associa a um papel. Dados e funções comuns a papéis são especificados por atributos e métodos de ambiente. Um *objeto* é composto de atributos e métodos. Embora não possa se movimentar, um objeto adquire esta capacidade por

14 Do inglês: *binding-interface*.

associação a um papel que possui funções de mobilidade. Desta forma, um objeto torna-se um agente pela sua associação a um papel e assim pode colaborar com outros agentes dentro do mesmo ambiente. Um objeto pode ser usado em vários ambientes simultaneamente.

Adicionalmente, uma interface de ligação define a interface com capacidade de receber mensagens de outros papéis no mesmo ambiente. Usando interfaces de ligação, as colaborações entre um conjunto de papéis podem ser especificadas separadamente da especificação dos objetos. A operação de ligação é permitida somente quando um objeto possui métodos concretos correspondentes à interface de ligação. As operações de ligação são implementadas pela criação de relações de delegação entre papéis e objetos dinamicamente. Isto é, se um papel recebe uma mensagem correspondente à sua interface de ligação, o papel delega a mensagem a um objeto associado ao papel. Muitos tipos de colaboração podem ser especificados por diferentes combinações de papéis e objetos.

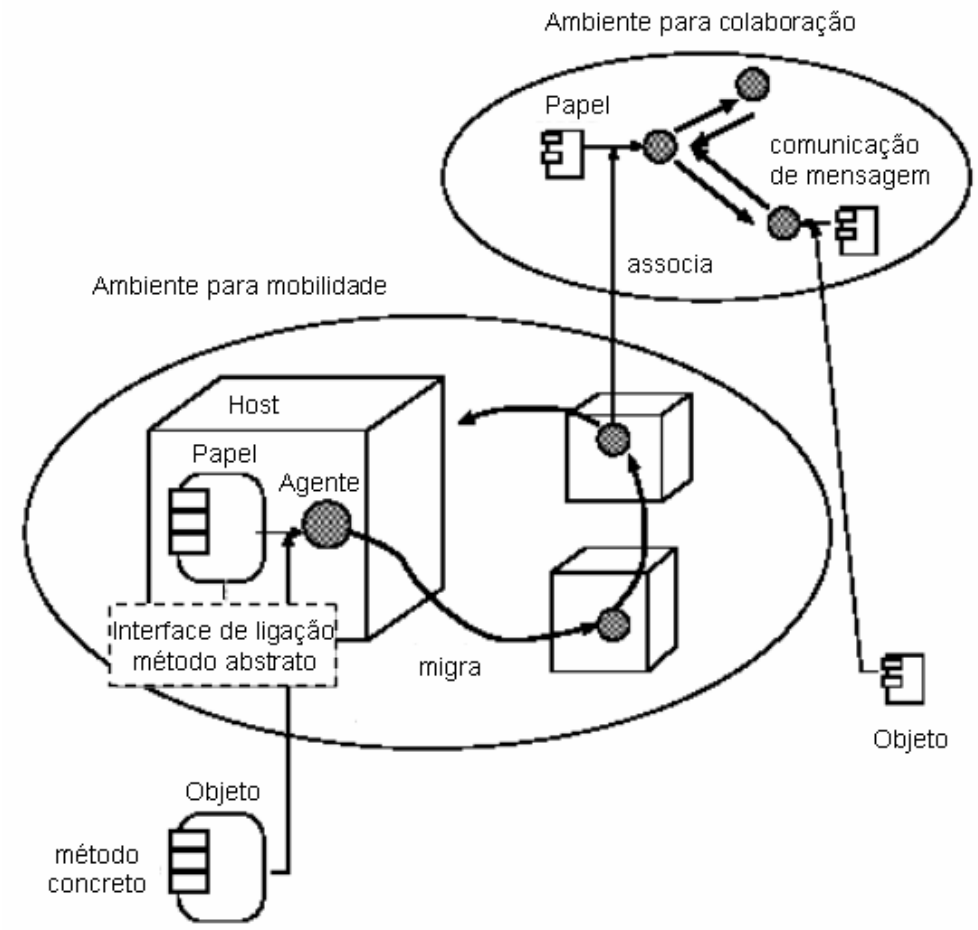

Figura 13. Conceitos do Modelo de RoleEP

Neste ponto, é importante observar que *RoleEP* se fundamenta sobre conceitos próprios de ambientes e papéis e isto traz distinções importantes face ao uso de abordagens tradicionais (Seções 3.2 e 3.3). Em geral, nos SMAs móveis o ambiente de um agente, isto é, o seu contexto de execução é um espaço limitado pelas configurações de acesso do servidor em cada *host* acessível ao agente. Um ambiente não se estende por mais de um *host*. Por outro lado, o conceito de papéis normalmente está associado à idéia do desempenho de funções em organizações. Como estes conceitos *não são fundamentais* em abordagens tradicionais, um sistema pode ser construído somente a partir da manipulação de objetos. Ao contrário, com *RoleEP*, uma aplicação não pode ser construída somente com objetos, sem que os conceitos próprios desta abordagem sejam utilizados.

Os conceitos de *RoleEP* são suportados em aplicações instanciadas a partir do *framework Epsilon/J* (Ubayashi & Tamai, 2001). Em *Epsilon/J*, uma classe de ambiente é definida como sendo uma subclasse da classe `Environment` e uma classe de papel é definida como uma subclasse da classe `Role`. A classe `Role` é uma subclasse da classe `Aglet` inicialmente sem funções de mobilidade. Uma classe de um objeto *Epsilon/J* é definida como uma subclasse da classe `EpsilonObj` com funções para operações de ligação. A Figura 14 ilustra o resultado parcial do processo de refatoração do sistema EC para utilização de serviços de mobilidade a partir de classes de *Epsilon/J*.

Um problema observado na Figura 14 é que os desenvolvedores devem estender várias classes abstratas do *framework Epsilon/J* a fim de instanciar SMAs móveis. Como demonstrado anteriormente, a necessidade de extensão de classes de um *framework*, usando mecanismos de orientação a objetos, diminui a modularização dos interesses de mobilidade. Além disso, para a reengenharia de SMAs móveis a partir de classes de *Epsilon/J*, são necessárias mudanças drásticas no projeto original das aplicações. Por exemplo, para que a classe de papel `Chair` tenha acesso aos serviços de mobilidade de *Aglets*, devem ser efetuadas modificações de alto impacto na hierarquia de herança do EC (Figura 14). No Capítulo 6, mostramos que para a utilização do AspectM, não é necessário estender classes do *framework*, ao mesmo tempo em que se mantém a flexibilidade no uso de plataformas de mobilidade.

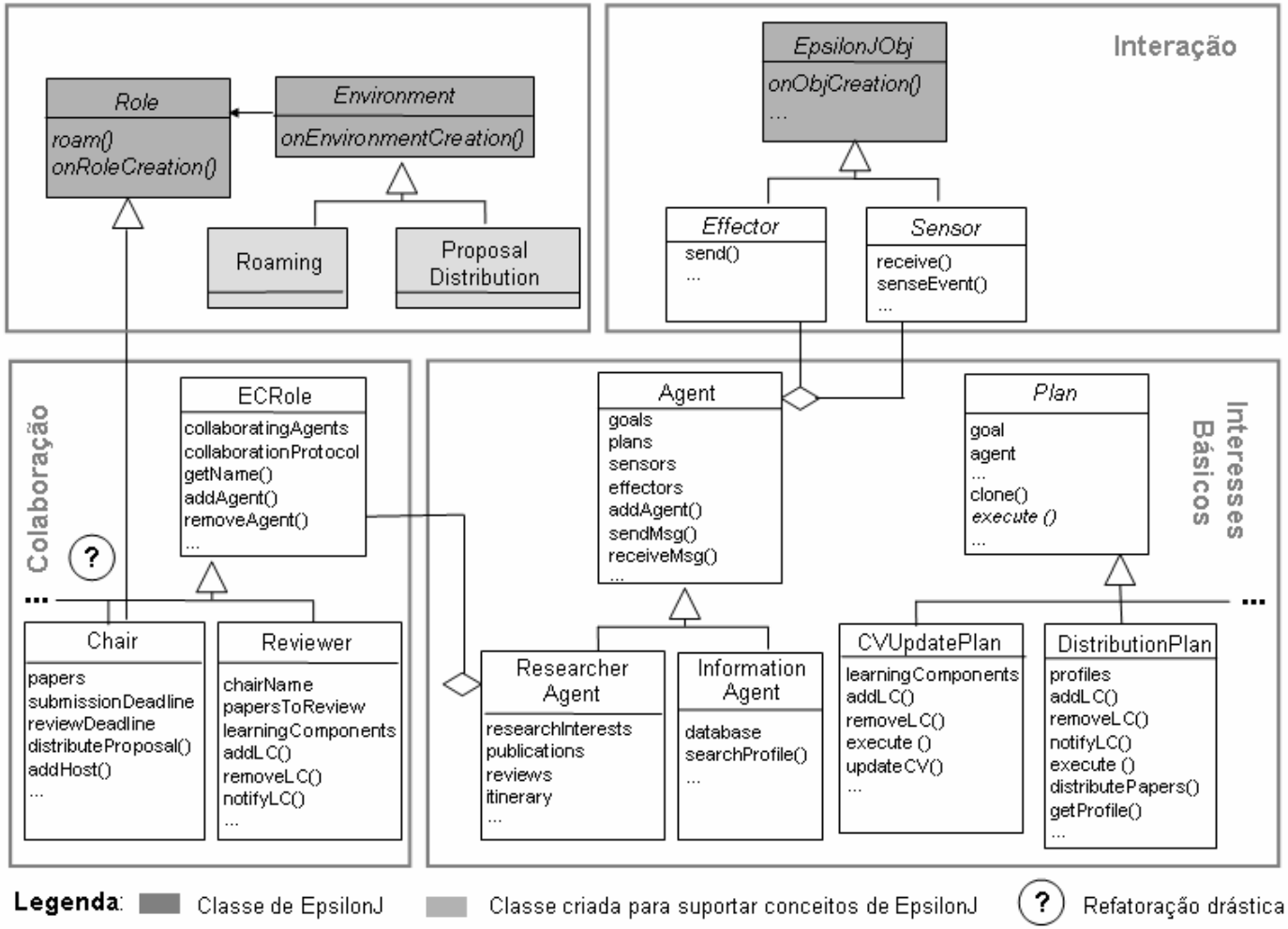

Figura 14. Solução para EC baseada em RoleEP e EpsilonJ

Ubayashi e Tamai (2001) apresentam os seguintes benefícios de *RoleEP*:

- Construção de componentes de mobilidade/colaboração. As funções de colaboração/mobilidade podem ser completamente separadas das funções básicas. Isto permite que uma mesma classe de ambiente pode ser reutilizada em vários SMAs móveis, isto é, as classes de ambiente podem ser pensadas como componentes de mobilidade/colaboração;
- Evolução modular de agentes. Um objeto se torna um agente pela associação a um papel que pertence a um ambiente. Com isso, um objeto pode dinamicamente se tornar um agente que desempenha múltiplos papéis. Em conseqüência, as aplicações que se adaptam a contextos externos podem ser facilmente especificadas.

De fato, *RoleEP* e *EpsilonJ* apresentam soluções para os problemas que se propõem a resolver, fornecendo: (1) um mecanismo para separação dos interesses de mobilidade e colaboração nas aplicações, e (2) um estilo de programação que permite evolução sistemática e dinâmica de SMAs móveis. Entretanto, deve ficar claro que o fato de *RoleEP* se fundamentar sobre conceitos próprios de ambientes, papéis, objetos e agentes traz distinções importantes face ao uso de abordagens tradicionais e da solução proposta neste trabalho para o problema de separação dos interesses de mobilidade. No Capítulo 8, *RoleEP*/*Epsilon/J* têm suas características comparadas à arquitetura ArchM e ao *framework* AspectM.

3.5. Requisitos de SMAs Móveis

A partir das discussões no Capítulo 2 e dos problemas identificados nas seções anteriores, esta seção apresenta a lista de requisitos de SMAs móveis tratados neste trabalho. Estes requisitos podem ser classificados em:

- **requisitos de projeto**: (P1) suporte à introdução transparente de mobilidade em agentes estacionários; (P2) SMAs não devem sofrer maiores impactos com as sucessivas manutenções no comportamento de mobilidade de um ou mais agentes, possíveis trocas de plataformas de mobilidade e mudança nas interfaces dos componentes de mobilidade; (P3) quando um papel de agente se move, o agente associado também

deve ser movido, uma vez que o papel depende das classes básicas que compõem o agente;

- **requisitos da aplicação**: (A1) a especificação de elementos móveis, como o tipo, os papéis, ou o conhecimento de agentes; (A2) a especificação dos pontos de instanciação dos agentes nas plataformas de mobilidade; (A3) a especificação do protocolo de instanciação, como inicialização/configuração de itinerário, tarefas e mestres dos agentes; (A4) a especificação das circunstâncias nas quais os agentes devem se mover, ou dos instantes em que a ação de movimentação deve ser disparada; (A5) a definição do protocolo de movimentação ou das questões inerentes à partida de agentes, como o envio de mensagens de partida, a finalização de processos e a especificação do próximo *host* a ser visitado; (A6) a definição do protocolo de inicialização ou das questões inerentes à chegada de agentes, como o envio de mensagem a outros agentes, a abertura de processos e a obtenção de informações como lista de objetos e/ ou agentes acessíveis; (A7) a definição de questões inerentes ao retorno de agentes para o *host* original, como especificação de circunstâncias excepcionais nas quais um agente deve retornar, ou checagem da necessidade de um agente continuar se movimentando pela rede; (A8) a definição do protocolo de destruição de agentes, como o envio de mensagens e finalização de processos;
- **requisitos específicos de tecnologias de implementação**, tais como, (T1) a especificação de quais objetos são serializáveis.

Evidentemente, a classificação acima não esgota a lista de requisitos de SMAs móveis. Observe que na lista acima são enumerados apenas requisitos relacionados ao projeto e à implementação de aplicações. Não são considerados requisitos derivados de e/ou diretamente relacionados a modelos específicos de plataforma. Por exemplo, questões relacionadas ao requisito de segurança em SMAs móveis não são tratadas neste trabalho.

Os identificadores dos requisitos acima (P1, P2, P3, A1, etc.) serão utilizados no Capítulo 8 para demonstrar que nossa solução para o problema de separação de mobilidade possui um escopo abrangente e tem impacto sobre vários requisitos importantes de SMAs móveis.