LUCIO MAURO DUARTE

DESENVOLVIMENTO DE SISTEMAS DISTRIBUÍDOS COM CÓDIGO MÓVEL A PARTIR DE ESPECIFICAÇÃO FORMAL

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação.

Programa de Pós-Graduação em Ciência da Computação, Faculdade de Informática.
Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Fernando Luís Dotti

PORTO ALEGRE
2001

Aos meus pais, pela confiança, apoio, carinho e amor.

Aos meus entes queridos que me acompanharam durante parte desta caminhada e que não puderam trilhá-la até o fim comigo, mas sempre acreditaram em mim e deram-me forças para prosseguir mesmo em meio às grandes dificuldades.

"So when you feel like hope is gone,

look inside and be strong.

And you'll finally see the truth:

That a hero lies in you"

Da música "Hero", interpretada por Mariah Carey

Agradecimentos

À CAPES, que viabilizou este trabalho através de seu suporte financeiro.

Ao PPGCC-FACIN, pelas instalações e recursos disponibilizados.

Ao corpo docente do PPGCC, pela solicitude e atenção, e, em especial, à Profa. Dra. Lucia Maria Martins Giraffa, que além de professora foi uma grande amiga ao longo da jornada.

Aos professores da linha de PPD Prof. Dr. Celso Maciel da Costa, Prof. Dr. César Fonticielha De Rose e Prof. Dr. Avelino Francisco Zorzo, pela amizade desenvolvida e pelo auxílio sempre presente.

Aos funcionários do PPGCC, Dario, Leandro e Viviane, pela amizade e disponibilidade.

À Profa. Dra. Leila Ribeiro, pela ajuda e acompanhamento do trabalho desenvolvido.

Ao meu orientador, Prof. Dr. Fernando Luís Dotti, pelo encaminhamento criterioso do trabalho desenvolvido, pela disponibilidade para dúvidas e questionamentos e, principalmente, pela amizade e companheirismo, que certamente contribuíram para um melhor desenvolvimento deste trabalho.

Aos meus inesquecíveis colegas e amigos de curso: Alexandre Zamberlam, Caroline Gasperin, Cristiano Sangoi, Daniel Mesquita, Daniela Ribas, Eder Mathias, Elisa Boff, José Carlos Palma, Josyane Barros, Leandro Cassol, Murilo Juchem, Rodrigo Goulart, Sediane Lunardi e Victor Sant'Anna. Quem dera todos pudessem encontrar e conquistar amigos como vocês. Estes dois anos não seriam tão felizes e realizadores se não fosse o nosso convívio, a nossa amizade sincera e a nossa união.

Aos novos colegas da turma de 2001 do PPGCC, pelas novas amizades conquistadas e pela integração alcançada.

A minha família e amigos, pela confiança no meu sucesso em mais este desafio.

E, finalmente, a Deus, por ter me dado a graça de viver em uma família maravilhosa, que me dá as condições de alcançar meus objetivos, e por poder ter encontrado um ambiente tão acolhedor e repleto de amigos verdadeiros para realizar este trabalho, tendo a certeza de que minhas realizações e as amizades formadas não terminam aqui.

Sumário

Lista de Figuras

Lista de Abreviaturas

AN	*Active Node* – Nodo Ativo
CB	*Code Base* – Base de Código
DSR	*Dynamic Source Routing*
FACIN	Faculdade de Informática
GGBO	Gramáticas de Grafos Baseadas em Objetos
JAMES	*Java-Based Agent Modeling Environment for Simulation*
JNA	*JavaNet Agents*
LEF	Linguagem de Especificação Formal
NS	*Naming Server* – Servidor de Nomes
PPGCC	Programa de Pós-Graduação em Ciência da Computação
PSM	Plataforma de Suporte à Mobilidade
PUCRS	Pontifícia Universidade Católica do Rio Grande do Sul
SBO	Sistema Baseado em Objetos
SCD	*Soft-Component Description Language*
SDCM	Sistema Distribuído com Código Móvel
PReq	Pacote de Requisição de Rota
PResp	Pacote de Resposta à Requisição de Rota
PE	Pacote de Erro
CR	*Cache* de Rotas
LRR	Lista de Requisições Recentes

Resumo

Este trabalho disserta sobre o desenvolvimento de aplicações distribuídas envolvendo mobilidade de código, abordando aspectos relativos à especificação formal, simulação e geração de código. Maior ênfase é dada ao mapeamento da linguagem de especificação formal adotada para código executável sobre uma plataforma de suporte à mobilidade de código. O formalismo utilizado é uma forma restrita de Gramáticas de Grafos, chamada de Gramáticas de Grafos Baseada em Objetos (GGBO). Em GGBO, sistemas são descritos por entidades e suas relações, sendo que cada entidade possui uma lista de regras que descrevem o seu comportamento. A partir de um mapeamento prévio de entidades e regras de GGBO para um ambiente de simulação, foram realizadas modificações que permitiram executar os sistemas especificados em um ambiente real, utilizando recursos de uma plataforma de suporte para tornar possível a movimentação de entidades. O mapeamento usado para a geração de código a ser executado foi avaliado através da realização de testes e de uma aplicação exemplo. Estudos de caso envolvendo redes ativas e um algoritmo de roteamento para redes *ad hoc* foram especificados em GGBO, tiveram seus comportamentos simulados e seus códigos gerados e executados em um ambiente distribuído.

Abstract

This work presents the development of distributed applications involving code mobility, discussing aspects related to formal specification, simulation and code generation for such applications. It focuses on the mapping from the adopted formal specification language to code executable on a mobility support platform. The formalism used in this work is a restricted version of Graph Grammars, called Object-Based Graph Grammars (OBGG). In OBGG, systems are described as a set of entities and their relationships. The behaviour of each entity is described through a set of rules. Starting from a previous mapping of OBGG entities and their rules to a simulation environment, modifications were made to allow the execution of the specified systems in a real environment, using the features provided by a support platform to make it possible to move entities. The mapping used to generate code to be executed was evaluated through some tests and the creation of a sample application. Case studies involving active networks and a routing algorithm for *ad hoc* networks were specified in OBGG, their behavior were simulated and their code were generated and executed in a distributed environment.

1 Introdução

1.1 Contexto

Com o contínuo crescimento das capacidades de comunicação e processamento, impulsionado por avanços tecnológicos e pelo aparecimento da Internet, surgiram ambientes computacionais que se caracterizam pela grande distribuição geográfica, pela sua alta heterogeneidade e dinamismo, pela inexistência de controle global, pela ocorrência de falhas parciais e pela falta de segurança [ASS99b]. Dados o ambiente caracterizado e as necessidades atuais de usuários, os sistemas passaram a ser compostos por um conjunto de componentes que se comunicam e executam em máquinas diferentes. Outra característica observada é a abertura de sistemas. Por *sistema aberto* entende-se, no contexto deste trabalho, um sistema autônomo quanto à sua execução, com ciclo de vida independente de outros sistemas (o que significa que ele pode não estar disponível em alguns momentos, caracterizando o seu dinamismo) e que possui capacidade de comunicação com os demais sistemas. Essa capacidade de comunicação com sistemas sobre os quais ele não tem, *a priori*, informação alguma é que o define como aberto. Neste trabalho, denomina-se um ambiente com as características citadas anteriormente, onde sistemas abertos coexistem, podendo interagir, como um *ambiente aberto*.

1.2 Caracterização de Sistemas Distribuídos com Código Móvel

Em sistemas distribuídos tradicionais, normalmente, os componentes permanecem no mesmo nodo da rede por todo o seu ciclo de vida, sendo, portanto, componentes estáticos. As interações entre estes componentes ocorrem através da troca de mensagens via rede, geralmente utilizando mecanismos tais como RPC (*Remote Procedure Call*) ou RMI (*Remote Method Invocation*). Com o aumento da conectividade entre sistemas e seus componentes, conforme o contexto apresentado na Seção 1.1, alguns efeitos colaterais podem ser notados. Um desses efeitos colaterais diz respeito ao aumento de tráfego na rede, devido à intensa comunicação remota. Além disso, as redes tornam-se cada vez maiores, trazendo o problema da escalabilidade de sistemas. Ou seja, alguns sistemas que produzem bons resultados em pequenas redes, tornam-se não aplicáveis em um ambiente de larga escala como a Internet [FUG98]. O uso cada vez maior de computadores móveis

também traz a necessidade do aumento de flexibilidade, a fim de que estes nodos dinâmicos possam ser suportados.

Os problemas expostos trazem algumas exigências quanto à construção de sistemas distribuídos para suprir as necessidades atuais de usuários. Algumas características que devem estar presentes em um sistema distribuído atual, destacadas em [KNA96], são: *descentralização de processamento*, através da distribuição da capacidade de processamento entre vários componentes, transcendendo as limitações de sistemas centralizados; *melhor uso de recursos de comunicação*, a fim de reduzir a ocorrência de congestionamentos na rede, melhorando o tempo de resposta do sistema em caso de interações remotas; *melhor uso de recursos computacionais*, de forma que os recursos disponíveis na rede possam ser compartilhados entre vários componentes, sem que haja a necessidade de múltiplas instâncias de um mesmo recurso; *suporte a operações desconectadas*, criando a possibilidade de se iniciar um processamento, desconectar-se da rede e, posteriormente, reconectar-se e recuperar o resultado do processamento realizado, o que significa economia de energia (principalmente para computadores móveis alimentados por bateria); e *flexibilidade de configuração*, possibilitando a fácil alteração do sistema, visando atender diferentes requisitos de usuários, e sua adaptação a mudanças do ambiente dinâmico em que ele executa.

A partir dos requisitos apresentados e da impossibilidade de sistemas que utilizam a abordagem Cliente-Servidor de suprirem tais requisitos, surgiram os *sistemas distribuídos com código móvel (SDCM)*, que são sistemas abertos que envolvem o conceito de mobilidade de código, também chamados de *aplicações móveis*. A tecnologia de mobilidade de código [FUG98] permite que trechos de código possam se mover entre nodos de uma rede. Com isso, além dos sistemas terem componentes distribuídos, eles também podem conter componentes móveis, os quais podem executar em locais diferentes em cada momento. Um outro conceito envolvendo mobilidade de código é o de agente móvel. Um *agente móvel* [ASS99a] é um componente móvel que possui, como principal característica, a sua autonomia. Um agente móvel é autônomo quanto a sua execução e quanto à tomada de decisão de quando e para onde se mover para cumprir as tarefas que lhe competem.

A mobilidade de código trouxe a possibilidade de diminuir-se o tráfego na rede causado por execuções remotas, como em uma arquitetura Cliente-Servidor, por exemplo, onde um cliente faz requisições a um servidor, possivelmente remoto, que as processa e

retorna o resultado a este cliente. Em vez disso, agora o cliente pode enviar um componente, que carrega todos os dados que devem ser processados para o local onde está o servidor. Depois disso, toda a comunicação passa a ser realizada localmente, economizando o uso dos recursos de comunicação.

A possibilidade de mover-se o componente para o lugar onde está o recurso necessário a sua execução permite o aumento no compartilhamento de recursos disponíveis na rede, já que múltiplos componentes podem ser movidos para o mesmo local a fim de utilizar o mesmo recurso. Componentes móveis fornecem a capacidade de processamento distribuído sobre a rede, além da flexibilidade de configuração do sistema, visto que novos componentes podem ser movidos e incorporados a um nodo e, através de sua interface, passar a prover novos serviços. Operações desconectadas também são viabilizadas pelo uso de mobilidade de código. Dessa forma, pode-se enviar um componente para executar uma determinada tarefa em um local remoto e desconectar da rede, recuperando-se o componente com o resultado da tarefa realizada tempos depois.

A partir dos benefícios do uso de mobilidade citados, as principais áreas de aplicação de SDCM são: controle e configuração dinâmica de dispositivos [PLO99]; gerência de redes [YEM93] [GOL95]; recuperação de informações distribuídas [JAI00]; serviços avançados de telecomunicações [LIM96] [MAG96a] [MAG96b]; gerência de *workflow* [CAI96]; comércio eletrônico [WHI94] [MER96]; e redes ativas [PSO99]. Novos nichos de utilização para mobilidade de código vêm sendo pesquisados e outras possíveis aplicações deverão ser conhecidas mais adiante.

1.3 Motivações e Objetivos do Trabalho

Todas as características advindas da abertura e distribuição dos sistemas, somadas às possibilidades fornecidas pela mobilidade de código, se, por um lado, tornaram os sistemas bastante flexíveis e poderosos, por outro lado, também tornaram bastante complexa a tarefa de desenvolver sistemas para tais ambientes. Isto se deve, principalmente, à grande dificuldade de encontrar e corrigir erros em sistemas que possuem componentes que executam de forma distribuída e que podem modificar sua localização. Devido à carência de suporte ao desenvolvimento de SDCM, esforços e pesquisas têm sido feitos para fornecer tal suporte. Um exemplo é o monitor descrito em [DOT99] e [DOT00a] que busca, através da monitoração de componentes móveis que executam na rede, fornecer informações relevantes ao desenvolvedor para auxiliá-lo no processo de teste e depuração

de seu sistema. Existe ainda um ambiente de modelagem e simulação apresentado em [UHR00], que emprega o uso de simulação para a depuração de sistemas.

Outros trabalhos seguem a abordagem de utilização de uma *linguagem de especificação formal (LEF)*. A utilização de uma LEF permite gerar-se uma descrição precisa de um sistema em uma notação com sintaxe e semântica bem definidas. Esta semântica associa um modelo matemático ao sistema que pode, então, ser analisado usando-se técnicas de verificação formal [DEH00]. Dessa forma, pode-se testar e provar, matematicamente, que um sistema possui certas características. Entre os trabalhos que utilizam esta abordagem podem ser citados: Mobile UNITY [ROM98], que apresenta uma notação para descrição de componentes e uma linguagem para descrever a interação entre eles; Ambit [CAR98], no qual é empregada a idéia de *ambientes*, onde um ambiente é um componente delimitado onde podem ocorrer computações; MobiS [MAS99] , que é uma linguagem de especificação baseada em múltiplos espaços de tuplas; e SCD [YOO98], que utiliza uma variante de Redes de Petri [PET73] para modelar sistemas. Existem ainda linguagens de programação que se baseiam no formalismo de Cálculo-π [MIL91], o qual é um modelo de computação concorrente em que os componentes são processos e canais, sendo os últimos usados para comunicação entre processos. Neste modelo, a localização de um processo é dada pela configuração de seus canais em um dado instante de tempo. A mudança nesta configuração determina a sua movimentação.

Uma outra abordagem utilizando uma LEF foi proposta em [DOT00b], como parte do projeto *ForMOS (Métodos Formais para Código Móvel em Sistemas Abertos)*[1], o qual envolve o desenvolvimento e adaptação de métodos e ferramentas de especificação, simulação, verificação e geração de código para SDCM. Na abordagem apresentada neste projeto, o formalismo de Gramáticas de Grafos [EHR79] [ROZ97] é utilizado para a especificação de SDCM. Este formalismo utiliza uma linguagem visual de especificação baseada em conceitos bastante simples e que, por ser formal, possibilita realizar-se a verificação de propriedades do sistema. Este formalismo teve seus elementos básicos mapeados para entidades de um ambiente de simulação de Gramática de Grafos existente [COP00], o qual gera código para simulação de sistemas descritos nesse formalismo. Ferramentas de verificação formal estão em estudo e deverão permitir a obtenção de

[1] Parcialmente financiado pelo CNPq, ProTeM-Laboratórios e FAPERGS.

provas formais da correção de sistemas especificados na LEF utilizada. O projeto prevê, ainda, a geração de código executável[2] para os sistemas especificados.

Este trabalho[3] insere-se no ForMOS, onde a sua principal contribuição inclui a geração de código executável a partir de especificações feita na LEF. Este código é gerado em uma linguagem de programação para uma plataforma de suporte à mobilidade de código. Portanto, o objetivo principal desta dissertação de mestrado, é a criação de uma forma de gerar código para SDCM a partir de uma especificação na LEF adotada no projeto ForMOS.

Como forma de atingir o objetivo principal, este trabalho também realizou outras contribuições para o projeto ForMOS. Entre estas contribuições está o uso da LEF adotada no projeto, o que permitiu criticar esta linguagem e identificar algumas possibilidades de melhorias. Também o uso do simulador de Gramáticas de Grafos utilizado no projeto gerou algumas contribuições. Foi realizada, neste trabalho, uma extensão no simulador, a qual permite que especificações de SDCM possam ser mapeadas para código de simulação e, assim, torna-se possível realizar simulações para este tipo de sistema no ambiente existente. Assim como ocorreu com a LEF, o uso do simulador possibilitou a identificação de deficiências do simulador e a ausência de algumas funcionalidades necessárias pôde ser percebida. Com isso, foram realizadas melhorias no funcionamento do simulador e novas características foram adicionadas, tornando-o um ambiente mais propício à realização de simulações de especificações feitas na LEF do projeto, principalmente, para simulações de especificações de SDCM.

A realização de um estudo de caso, no qual foram utilizadas as ferramentas desenvolvidas no projeto, permitiu não só gerar as contribuições acima relatadas, mas também avaliar o mapeamento da LEF para código executável. Este estudo de caso visou abranger todas as ferramentas trabalhadas no projeto ForMOS, como forma de avaliar a sua utilização e de que forma elas contribuem para o desenvolvimento de sistemas que envolvem mobilidade de código.

---

[2] Neste trabalho, código executável significa um código a ser executado em um ambiente real.

[3] Financiado pela CAPES.

1.4 Estrutura do Texto

Este volume apresenta, no Capítulo 2, o referencial teórico, contemplando os assuntos mais importantes do trabalho: mobilidade de código e especificação formal. No Capítulo 3 é apresentada a LEF trabalhada e no Capítulo 4 descreve-se o simulador utilizado. No Capítulo 5, trata-se sobre a questão da geração de código e todos os passos seguidos para gerar código para SDCM a partir de especificações formais na LEF adotada. O Capítulo 6 apresenta a descrição de um estudo de caso desenvolvido durante este trabalho, envolvendo a definição de um ambiente de rede ativa. O Capítulo 7 descreve um segundo estudo de caso desenvolvido a partir do ambiente de rede ativa definido no primeiro estudo de caso. Para os dois estudos de caso são apresentadas as especificações feitas na LEF para cada uma de suas entidades. O Capítulo 8 apresenta alguns trabalhos relacionados com o trabalho aqui descrito. O Capítulo 9 contém as conclusões sobre o trabalho desenvolvido. O Capítulo 10 apresenta as referências bibliográficas utilizadas.

2 Referencial Teórico

Este referencial teórico apresenta os dois principais assuntos envolvidos neste trabalho, que são mobilidade de código e especificação formal. Cada um dos assuntos é apresentado abordando-se os principais conceitos e o suporte existente.

2.1 Mobilidade de Código

Nesta seção serão apresentados os conceitos básicos envolvendo mobilidade de código. Além disso, discute-se sobre as linguagens de programação e as plataformas de suporte a código móvel.

2.1.1 Conceitos Básicos

Mobilidade de código pode ser definida como a capacidade de um componente de software de poder mover-se entre nodos de uma rede. A mobilidade de código deriva da idéia de migração de processos [MIL99b], que diz respeito à transferência de processos de sistema operacional das máquinas em que estão executando para outras, com a intenção, em geral, de prover tolerância a falhas e balanceamento de carga. A diferença entre migração de processos e mobilidade de código é que, na primeira, a movimentação de um componente (processo) é definida pelo sistema operacional distribuído, ocorrendo transparentemente ao desenvolvedor da aplicação, enquanto que, na segunda, a movimentação é definida pelo usuário, o qual determina o momento da movimentação e o destino da mesma. Ou seja, com a mobilidade de código, não existe a transparência de movimentação presente na migração de processos.

Em [FUG98], são propostas definições para discutir sobre mobilidade. Algumas dessas definições são aqui utilizadas para apresentar conceitos pertinentes. Assim, segundo a classificação proposta por [FUG98], existem duas formas de mobilidade de código:

- *Mobilidade forte*: quando um componente[4] se move, é movido o seu código e seu estado de execução (valores de atributos internos, ponteiro de instruções, pilha de execução, etc.), possibilitando que, ao chegar ao destino, a execução possa ser

[4] O termo *componente*, aqui usado, refere-se a uma estrutura de *software* que encapsula dados e um comportamento e que pode conter atividade interna (*thread* interna em execução).

retomada da instrução imediatamente seguinte à movimentação, partindo-se do mesmo estado de execução atingido antes da movimentação;

- *Mobilidade fraca*: quando o componente se move, é movido apenas o seu código e os valores de seus atributos internos. Dados de inicialização podem acompanhar a movimentação (tal como um ponto de entrada no código, a partir do qual o componente deve executar), mas não existe a movimentação do estado de execução.

Mecanismos que suportam mobilidade fraca fornecem a capacidade de mover-se um código entre nodos da rede, podendo-se realizar uma ligação dinâmica deste com um componente já existente no destino ou executá-lo como um novo componente. Para o suporte à mobilidade forte, segundo [FUG98], existem dois mecanismos: migração e clonagem remota. Através do mecanismo de *migração*, quando se inicia o processo de movimentação, é realizada a suspensão da execução do componente que irá se mover. Este, depois de suspensa a sua execução, é transmitido ao destino e sua execução é então retomada. Já através do mecanismo de *clonagem remota*, não é realizada a movimentação do próprio componente. Em vez disso, quando se inicia o processo de movimentação, é criada uma cópia do componente no destino. O estado de execução na origem é transmitido para a cópia criada e esta passa a executar como o componente original. Dessa forma, a cópia no destino assume o lugar do componente original.

Os mecanismos descritos acima podem ocorrer de forma proativa ou reativa. Quando a movimentação é *proativa*, o momento e o destino da movimentação são determinados de forma autônoma pelo próprio componente. Na movimentação *reativa*, o processo de movimentação do componente é disparado por outro componente, o qual possui o controle de movimentação do componente que se move.

Com o surgimento da tecnologia de código móvel, surgiram também três novos paradigmas de programação. Tais paradigmas são apresentados em [CAR97], sendo diferenciados de acordo com a localização dos componentes envolvidos antes e depois da execução de um serviço, o componente responsável pela execução do código e o local onde a computação realmente ocorre. Os paradigmas identificados são os seguintes:

- *Avaliação Remota*: No paradigma de Avaliação Remota, um componente *C1* possui o código para realizar um serviço, mas não possui os recursos necessários,

onde um recurso representa um elemento de dados passivo, como um arquivo ou uma impressora, um dispositivo de rede ou qualquer outro dispositivo físico. Tais recursos estão em um local remoto *L2*, onde um componente *C2* está executando. *C1* então envia o código para o componente *C2* para que este o execute usando os recursos locais. Ao final da execução, *C2* retorna os resultados para *C1*;

- *Código por Demanda*: No paradigma de Código por Demanda, um componente C1 possui os recursos de que precisa para executar um serviço no local L1 onde se encontra. No entanto, C1 não possui o código para utilizar estes recursos. Por isso, C1 interage com um componente C2, localizado em um local L2, o qual possui o código necessário. Dessa forma, C1 requisita o código de C2, que o envia a C1. Ao receber o código requisitado, C1 pode então executar o serviço com os recursos locais;

- *Agentes Móveis*: No paradigma de Agentes Móveis, um componente C1, executando em um local L1, possui o código para executar um serviço, mas os recursos necessários estão em um local L2. C1 então se move para L2 carregando consigo o código para executar o serviço. Após a movimentação, o componente C1, agora executando em L2, realiza o serviço com os recursos locais.

Como pode ser visto, o foco dos paradigmas de Avaliação Remota e Código por Demanda está na transferência de código entre componentes, enquanto que, no paradigma de Agentes Móveis, é o componente, juntamente com seu estado e o código necessário, que é transferido para o local onde estão os recursos. A partir do paradigma de Agentes Móveis, criou-se o conceito de *agente móvel*. A denominação de agente móvel está associada ao fato de que este tipo de componente incorpora características de agentes. Um *agente* é definido como uma entidade computacional que trabalha em favor de outras entidades de forma autônoma e que realiza ações proativas e/ou reativas [GRE97]. Portanto, um agente móvel possui tais características de agentes, com o acréscimo da sua capacidade de mobilidade. Segundo a definição apresentada em [ASS99a], "um agente móvel é um componente autocontido, responsável pela execução de uma atividade, que é capaz de, autonomamente, migrar através de uma rede". Com isto, tem-se que um agente móvel é uma forma de código móvel com autonomia de execução e movimentação. Um agente móvel possui, portanto, a capacidade de decidir quando e para onde se mover para cumprir a tarefa que lhe foi confiada, assumindo movimentação proativa.

2.1.2 Linguagens de Programação

O desenvolvimento do conceito de mobilidade de código e o surgimento de aplicações que utilizam esta tecnologia foram, em grande parte, proporcionados pela existência de linguagens de programação que possuem mecanismos de suporte à movimentação de código. Tais linguagens são diferenciadas de outras linguagens de programação por um conjunto de propriedades essenciais. Estas propriedades definem uma base mínima para a construção de aplicações móveis. Segundo [KNA95], tais propriedades são as apresentadas a seguir.

2.1.2.1 Suporte à manipulação, transmissão, recebimento e execução de códigos de componentes

Para suportar o trabalho com componentes que contêm código, a linguagem deve fornecer uma forma de identificação desses componentes, a fim de diferenciar um componente de outro. Primitivas ou uma biblioteca de funções com as quais seja possível ao desenvolvedor expressar que um dado componente deve ser transmitido, também é um requisito importante. Além disso, é necessário que a linguagem forneça suporte à execução de componentes recebidos dentro do espaço de endereços do receptor. Isto é, o componente recebido deve ser executado utilizando os recursos locais do receptor. É também desejável que a linguagem forneça construções e abstrações que permitam a total manipulação de código dentro da própria linguagem.

2.1.2.2 Suporte para execução em sistemas computacionais heterogêneos

A heterogeneidade é uma característica de sistemas distribuídos e, portanto, deve ser considerada em uma linguagem que deve suportar a transmissão de código entre diferentes nodos da rede, os quais podem estar executando sobre diferentes sistemas computacionais. As linguagens devem, portanto, suportar a transmissão de componentes entre máquinas de diversas arquiteturas, de modo a permitir a construção de aplicações mais adaptadas às características dos ambientes atuais, tais como a Internet, onde uma grande variedade de *hosts*, executando sobre sistemas heterogêneos, coexistem e comunicam-se entre si.

2.1.2.3 Desempenho compatível com as necessidades das aplicações.

Quanto ao desempenho, é desejável que a implementação da linguagem minimize, tanto quanto possível, os custos de espaço e tempo gastos no uso de componentes móveis.

Componentes móveis consomem tempo e espaço devido ao seu tamanho, tempo de transmissão, tempo necessário para convertê-lo em sua forma executável, etc. Estes custos devem ser baixos o suficiente para que o uso de mobilidade de código não deixe de ser vantajoso para a aplicação. Ou seja, a implementação dos mecanismos de movimentação de componentes não deve tornar-se um fator de grande influência no desempenho geral da aplicação. Caso contrário, torna-se prejudicial à aplicação a sua construção considerando componentes que se movem, sendo assim preferível construí-la usando outras técnicas mais tradicionais.

Outra característica que pode ser considerada é a segurança, já que um componente móvel transita entre diversos domínios. Por isso, um cuidado especial deve ser tomado para fornecer proteção contra ameaças à segurança provenientes de aplicações móveis. Algumas das medidas de segurança, segundo discutido em [THO97], são a garantia da integridade de dados privados, impedindo a sua modificação por componentes não autorizados, e garantia de autenticidade das identidades de entidades que se comunicam, assegurando que é possível confiar que os dados transmitidos são encaminhados ao seu destino correto.

Algumas das linguagens de programação que possuem suporte à criação e uso de componentes móveis são apresentadas em [DUA00a]. Dentre as linguagens existentes, a mais utilizada é a linguagem Java [GOS96], da Sun MicroSystems. Por sua portabilidade e heterogeneidade, Java tem sido utilizada na construção de sistemas distribuídos e tem servido de base para a criação de plataformas de suporte à mobilidade de código. Isto porque, seguindo a característica de portabilidade de Java, o código é interpretado em cada local em que é recebido, provendo a possibilidade de movimentação também entre máquinas heterogêneas. Além disso, Java também possui mecanismos de serialização e desserialização de objetos, tornando possível a transmissão de código entre diferentes *hosts*. Existem também mecanismos de suporte à ligação dinâmica de código, muito utilizados na execução das *applets*, que são aplicações Java que podem ser transmitidas juntamente com uma página HTML para execução em um local remoto [KNA96].

2.1.3 Plataformas de Suporte

Baseando-se nas linguagens de programação existentes com suporte à mobilidade de código, surgiram as *plataformas de suporte à mobilidade (PSM)*. Essas plataformas

fornecem bibliotecas para as linguagens de programação em que são implementadas, provendo operações para movimentação de código. Assim, as PSM implementam os mecanismos de movimentação na linguagem de programação que estendem e fornecem uma biblioteca de funções para a utilização desta implementação. Dessa forma, o desenvolvedor não se ocupa em implementar os mecanismos de movimentação, já providos pela PSM, e pode dedicar-se exclusivamente à aplicação que deve ser desenvolvida.

Segundo apresentado em [KAR98], são requisitos básicos de uma plataforma de suporte à mobilidade prover as funções apresentadas a seguir.

2.1.3.1 Mobilidade de Componente

Cada *host* que pretende receber componentes móveis deve prover um ambiente de execução para o mesmo, fornecido por uma plataforma. A função mais básica de uma plataforma é fornecer serviços de movimentação de componentes. A movimentação de um componente envolve suspender a sua execução, salvar seu estado e transmiti-lo a um local remoto. A plataforma executando no local remoto deve, então, ser capaz de restaurar o estado do componente e reativar sua execução. O estado de um componente inclui todos os seus dados internos, bem como o estado de execução de sua *thread*. O estado de execução da *thread* é representado, em um nível mais inferior, por seu contexto de execução e por sua pilha de execução. Caso seja possível transmitirem-se estes dados juntamente com o componente, torna-se viável restaurar a sua execução no destino exatamente no ponto onde ela foi paralisada antes da movimentação. Na impossibilidade de capturar os dados neste nível mais baixo, uma alternativa é obter-se o estado de execução em nível de função, sendo que o código do componente deve prover alguma maneira de retomar o fluxo da execução no destino. Isto pode ser feito através da definição de um ponto de entrada no código, que pode ser representado pela invocação de uma função do componente. Com isso, ao chegar ao destino, esta função do componente é ativada. Portanto, o desenvolvedor deve descrever nesta função qual comportamento deve ser seguido para a retomada da execução do componente.

Além do estado de execução do componente, também deve ser provida uma forma de transferir o seu código. Quanto a isso, existem três possibilidades: o componente leva seu código consigo ao se mover; o código do componente encontra-se previamente instalado no local de destino; ou o componente ou o próprio local de destino possui uma referência a

uma base de código de onde o local de destino pode obter o código necessário. A primeira possibilidade é a mais desejável, já que possibilita que o componente execute em qualquer lugar onde haja uma plataforma que permita a sua execução. Entretanto, exige o suporte à transferência real do código do componente juntamente com seu estado de execução. A segunda possibilidade exige que o código seja pré-instalado em cada local por onde o componente possa passar, restringindo o domínio de atuação do componente. A terceira e última possibilidade utiliza a idéia de código por demanda. Ou seja, quando termina a movimentação dos dados do componente, o local de destino requisita de uma base de código o código necessário para a execução do componente recebido. Esta opção supera um dos problemas da opção anterior, já que não é necessário ter o código instalado em todos os possíveis locais a serem visitados por um componente. Em compensação, exige a existência de uma base de código que possa receber requisições de lugares (que são locais onde componentes podem executar) e exige que estes ou os componentes recebidos por estes possuam uma referência a uma base de código. Além disso, problemas de falhas na base de código e escalabilidade de cada base de código devem ser considerados. Também se exige a manutenção das bases de código para que possuam sempre os códigos corretos e atualizados. Para isso, é necessário o estabelecimento de mecanismos de registro e identificação dos diversos códigos.

2.1.3.2 Identificação de Componentes e Lugares

Cada entidade (componente ou lugar) em um sistema deve ser identificada de uma forma a ser diferenciada de qualquer outra entidade. Isto permite que componentes e lugares comuniquem-se entre si e que seja possível ao desenvolvedor acompanhar a execução de seus componentes. Identificações também são úteis para determinar o destino de uma movimentação, tornando possível especificar a identificação do destino da movimentação ou, como ocorre em algumas plataformas, fornecer a identificação de um componente com qual seja necessário comunicar-se. Neste último caso, o componente que se move é transferido para o local onde se encontra o componente com o qual ele deve interagir.

Outro mecanismo que deve ser fornecido pela plataforma e que utiliza a identificação única de componentes é a localização de componentes e lugares. Assim, deve ser possível obter a localização de qualquer componente ou lugar dada a sua identificação, ao que se dá o nome de resolução de identificadores ou resolução de referências, no caso de as

entidades serem identificadas por referências. A resolução de referências fornece transparência de localização, já que não é necessário saber onde determinado componente está para que se possa interagir com ele.

Em algumas plataformas, as identificações ou referências são fornecidas com base no *hostname* e em números de portas, sendo identificações dependentes de localização. Nestas plataformas, o nome do componente se modifica a cada movimentação, exigindo que sejam providos nomes com transparência de localização no nível de aplicação. Tais plataformas usam um esquema de *proxies* locais. Uma *proxy* é uma estrutura que possui a mesma interface do componente que ela referencia e que encapsula a localização atual do componente. Neste caso, todas as comunicações direcionadas ao componente são encaminhadas a sua *proxy*, a qual repassa as mensagens ao componente no seu local de execução atual. A outra possibilidade de resolução de referências envolve o uso de nomes globais, independentes de localização. Em plataformas que usam este tipo de resolução de referências, os nomes dos componentes não se modificam quando estes se movem. Neste caso, devem ser providos servidores de nomes, os quais têm a função de mapear nomes simbólicos para a localização atual da entidade. Além disso, cada componente, ao se mover, deve atualizar a informação de sua localização nos servidores onde esteja registrado.

2.1.3.3 Segurança

Deve ser garantida a integridade dos dados e código de componentes trafegando na rede e a integridade dos lugares por onde os componentes passam e com os quais eles se comunicam. Partes de um componente devem poder ser mantidas em sigilo, exigindo que haja mecanismos para selecionar quais porções do componente podem ser visualizadas e/ou modificadas por outros componentes ou lugares. Caso haja falhas na segurança, também deve haver uma forma de verificar que ocorreu alguma alteração nos dados ou no código do componente. Mecanismos de criptografia podem ser utilizados para fornecer comunicação segura na transmissão de componentes através da rede. Quando um componente requisita a sua transferência para um outro lugar, deve ser possível a este lugar verificar, com algum tipo de autenticação, a veracidade da identificação do componente, a fim de decidir recebê-lo ou não e, caso o receba, quais direitos lhe serão dados quanto a sua execução no ambiente local.

Adicionalmente, podem ser fornecidos mecanismos para suportar alguma forma de contabilização de recursos utilizados por componentes móveis, tais como tempo de CPU, espaço de armazenamento, etc. Isto possibilitaria que os componentes trafegassem com uma certa quantia de créditos que pudessem ser utilizados para "pagar" pelos recursos a serem utilizados.

2.1.3.4 Plataforma Voyager

Muitas das PSM existentes foram criadas sobre a linguagem Java. Em geral, as bibliotecas construídas sobre Java suportam a resolução de referências utilizando o conceito de *proxy* para prover independência de localização. A *proxy* possui, em sua estrutura, uma interface do componente (que, no caso de Java, é um objeto) que ela referencia, e a informação da localização atual deste componente, permitindo que ela possa receber ativações de métodos do componente e, caso sejam válidas (sejam relativas a métodos constantes na interface pública do objeto Java), encaminhe-as ao componente em questão.

Voyager [OBJ00] é uma das plataformas implementadas em Java, desenvolvida pela ObjectSpace, Inc. Esta plataforma tem sido bastante utilizada na construção de aplicações móveis. Voyager utiliza *proxies* para realizar a resolução de referências. Assim, todo componente remoto em Voyager é referenciado por uma *proxy*. Utilizando a resolução de referências da PSM, quando uma mensagem é enviada a um componente, esta é recebida pela *proxy* do componente. Caso o componente de destino da mensagem esteja no mesmo local do componente que enviou a mensagem, a comunicação ocorre como uma mensagem Java comum (mensagem é enviada através de uma chamada local de método). Se o componente que enviou a mensagem e o componente de destino estiverem em locais distintos, a mensagem é recebida pela *proxy* do destinatário e uma cópia dos argumentos da mensagem é serializada e enviada através da rede. Ao chegar a seu local de destino, os argumentos são desserializados e recebidos pelo componente de destino, que realiza as ações necessárias.

A Figura 1 apresenta uma situação onde um componente localizado em um local 1 gera uma mensagem para outro componente localizado em um local 2.

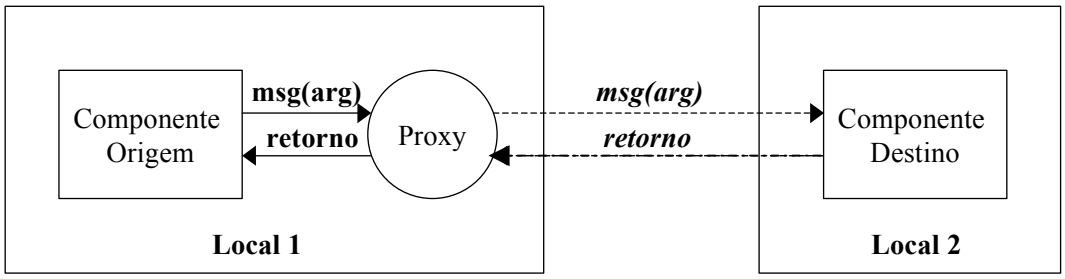

Figura 1. Resolução de referências na plataforma Voyager.

Esta mensagem é recebida localmente pela *proxy* do componente de destino. O argumento *arg* é serializado e a mensagem é repassada ao componente remoto. Este desserializa o argumento e processa a mensagem. O processamento da mensagem gera um valor de retorno que é serializado e passado pelo componente para a *proxy*. A *proxy* desserializa o valor de retorno e o repassa para o componente que gerou a mensagem. Caso alguma exceção tivesse ocorrido no processamento da mensagem no destino, o aviso de ocorrência de exceção seria feito da mesma forma que ocorre com o valor de retorno. Por esse esquema de resolução de referências, todo componente precisa possuir uma interface definida e todo componente que queira se comunicar com outro componente deve obter uma *proxy* local deste.

A plataforma Voyager suporta mobilidade fraca com movimentação através do mecanismo de clonagem remota, conforme classificação citada na Seção 2.1.1. Isto significa que, ao mover-se, um componente tem uma cópia sua gerada no local de destino da movimentação, a qual possui o estado interno igual ao estado interno atual do componente original. Dessa forma, quando um componente se move, é criado um clone deste no destino e este clone é inicializado com o mesmo estado do componente original e, posteriormente à movimentação, o clone assume lugar deste. Durante a movimentação de um componente, todas as mensagens encaminhadas a ele são suspensas. O componente é serializado e copiado para o destino e, somente após a cópia tornar-se verdadeiramente o componente ao final da movimentação, as mensagens suspensas são processadas no novo local.

A movimentação de um componente causa a necessidade de alteração de suas referências. Isto é feito através de um mecanismo chamado *forwarder*. Ao se mover, um componente deixa, no seu local de origem, uma informação de qual é o seu novo endereço. Quando uma mensagem é enviada via *proxy* para um componente que se moveu, é gerada uma exceção que retorna esta informação deixada pelo componente. Esta informação é

armazenada pelo *forwarder*, que serve como um "apontador" para o componente. Ao receber o retorno da exceção, a *proxy* atualiza internamente o endereço do componente e envia a mensagem ao novo local. A partir daí, todas as mensagens são enviadas diretamente ao local atual do componente. O mecanismo de *forwarder* é apresentado na Figura 2.

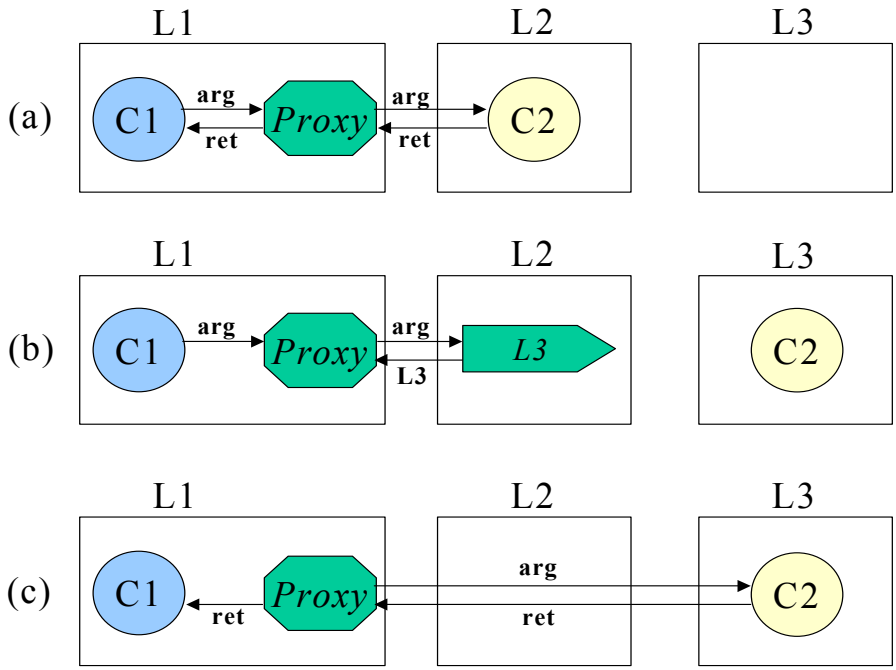

Figura 2. Mecanismo de *forwarder* da plataforma Voyager.

Na Figura 2, é apresentada uma situação onde um componente C1, localizado em um lugar L1, comunica-se com um componente remoto C2, localizado em L2, através da *proxy* deste último. Na Figura 2.a, a comunicação ocorre normalmente, com o envio da mensagem de C1 e seus argumentos para a *proxy* local, que repassa a mensagem para o componente C2 em L2. Quando C2 se move para L3, é criado um *forwarder* em L2, contendo o novo local de C2. Com isso, quando uma nova mensagem é enviada a C2, ela é encaminhada pela *proxy* a L2. Ao chegar em L2, a mensagem é recebida pelo *forwarder*, gerando o retorno de uma exceção contendo o novo endereço de C2 (Figura 2.b). A *proxy* então atualiza a sua referência a C2 para o lugar L3 e encaminha a mensagem ao seu destino (Figura 2.c). A partir daí, a comunicação ocorre como normalmente.

Voyager oferece duas possibilidades de componentes móveis: objetos móveis, que possuem movimentação reativa (têm sua movimentação disparada por outro componente, o qual controla sua movimentação), e agentes móveis, que possuem autonomia de movimentação, assumindo uma movimentação proativa. Para restaurar o estado de um

componente móvel após a sua movimentação, Voyager utiliza a definição de uma função que serve como ponto de entrada para a retomada da execução. Assim, o desenvolvedor deve definir uma função a ser ativada quando a movimentação for finalizada. Caso nenhuma função deste tipo seja definida, o componente passa a estar passivo após uma movimentação, esperando por uma ativação futura.

Voyager também fornece a opção de criar um gerente de segurança (*Voyager Security Manager*) com o qual é possível definir restrições quanto à execução de certas operações de componentes.

2.2 Especificação Formal

Nesta seção, são apresentados os conceitos básicos relativos à especificação formal. Ao final, é discutido, sucintamente, o uso de especificação formal para descrever sistemas distribuídos e SDCM.

2.2.1 Conceitos Básicos

Uma *especificação* é uma descrição de alto nível (abstrata) do sistema a ser construído, a qual deve ser compacta, precisa e não ambígua. Uma *especificação formal* [CLA96] [AST97] é uma descrição de um sistema feita em uma linguagem com sintaxe e semântica precisamente definidas; ou seja, definidas utilizando-se conceitos matemáticos. Esta descrição é feita utilizando-se uma linguagem de especificação formal (LEF).

Dada uma especificação formal do sistema, pode-se provar a existência ou não de propriedades ou características neste sistema. Esta prova é feita através de uma abordagem de *verificação formal* [DEH00]. A verificação formal consiste em utilizarem-se técnicas matemáticas para assegurar-se que um sistema computacional apresenta algumas propriedades.

Mesmo que técnicas de verificação não sejam empregadas, a especificação formal do sistema ajuda no seu desenvolvimento por oferecer uma maneira não ambígua de descrevê-lo. Por descrição não ambígua, entenda-se uma descrição que é clara e precisa o suficiente, utilizando-se para isso de uma linguagem formal, para permitir apenas uma única interpretação do sistema descrito. Esta interpretação única nem sempre é fornecida por descrições feitas em linguagem natural, onde diferentes interpretações podem gerar diferentes resultados.

2.2.2 Aplicações

Pode-se dizer que a especificação formal de sistemas não possui uma aplicação específica. Ela pode ser empregada na descrição de aplicações de controle de sistemas de produção ou para permitir a verificação formal de sistemas que envolvam algum risco caso não estejam corretos, tais como, sistemas para hospitais, usinas nucleares, experimentos químicos, entre outros. Existem trabalhos que utilizam formalismos para o desenvolvimento de microcódigos, como o do processador Motorola 68020 [BOY96], por exemplo. Além disso, em [STA96] é apresentada a utilização de verificação formal, a partir da descrição do sistema em uma LEF, para o desenvolvimento de *software* industrial. Existe ainda a possibilidade de aplicação de especificação formal em sistemas de controle, como discutido em [BRI95], entre outras.

O uso de especificação formal pretende prover sistemas de maior qualidade e maior confiabilidade, principalmente com o auxílio de ferramentas de verificação formal, sem, no entanto, aumentar os custos de desenvolvimento e, talvez, até diminuindo-os. Apesar disso, existem argumentos contra a utilização de formalismos, alguns deles discutidos em [FRA94]. Um desses argumentos diz respeito às notações das LEF, as quais, na maioria das vezes, exigem familiaridade com conceitos matemáticos e lógica simbólica. Isto cria a necessidade de treinamento de pessoal para trabalhar com estes formalismos, já que a maior parte dos desenvolvedores atualmente não possui conhecimentos sobre as LEF disponíveis.

2.2.3 LEF para Sistemas Distribuídos e SDCM

A especificação de sistemas distribuídos requer técnicas que possam auxiliar no tratamento de seus problemas mais comuns, como acesso a recursos compartilhados, comunicação entre processos remotos e alocação, replicação e migração de componentes [TAE00]. Existem hoje inúmeras LEF que podem ser utilizadas para a descrição de sistemas distribuídos, algumas delas apresentadas em [DUA00b]. Entre as LEF existentes, algumas das mais conhecidas e utilizadas são TLA (*Temporal Logic of Actions*) [LAM94], SDL (*Specification and Description Language*) [ITU00] e CSP (*Communicating Sequential Processes*) [HOA85]. Além dessas, pode-se ainda utilizar Gramáticas de Grafos [EHR79] [ROZ97] como um formalismo para descrever sistemas distribuídos.

Muitas LEF têm sido criadas com o intuito de serem utilizadas para descrever sistemas com mobilidade enquanto outras são extensões de linguagens anteriores,

fornecendo uma forma de representação de mobilidade. Algumas dessas linguagens serão apresentadas no Capítulo 8 como trabalhos relacionados.

A seguir é descrito o formalismo de *Gramáticas de Grafos* e suas principais características são apresentadas. Gramáticas de Grafos é a base da LEF utilizada neste trabalho, a qual será discutida no Capítulo 3.

2.2.3.1 Gramáticas de Grafos

Para formalizar um problema, é desejável ter-se uma maneira natural e intuitiva de descrição. Propriedades essenciais de sistemas complexos, como distribuição, paralelismo e comunicação devem ser consideradas. O fato de serem formais e, ao mesmo tempo, intuitivas e de também poderem tratar com simplicidade aspectos de concorrência e distribuição de sistemas, faz de Gramáticas de Grafos um método promissor para o desenvolvimento de *software* confiável [DEH00]. Gramáticas de Grafos são uma generalização de gramáticas de Chomsky [CHO59], substituindo-se as *strings* por grafos. Diferentemente do que ocorre nas regras em gramáticas de Chomsky, uma regra de grafos $r: L \rightarrow R$ não consiste somente dos grafos L (lado esquerdo) e R (lado direito), mas também de uma parte adicional: um mapeamento de vértices e arcos de L em vértices e arcos de R de maneira compatível. Assim, se um arco e_L for mapeado em um arco e_R, então o vértice origem de e_L deve ser mapeado para o vértice origem de e_R, ocorrendo o mesmo para o vértice destino. Gramáticas de Grafos, segundo a abordagem algébrica [EHR79], especificam um sistema em termos de estados (modelados por grafos) e mudanças de estados (modeladas por *derivações*).

A interpretação operacional de uma regra $r: L \rightarrow R$, seguindo esta abordagem de especificação, é a seguinte:

- Itens de L que não têm imagem em R são *removidos*;
- Itens de L que são mapeados para R são *preservados*;
- Itens de R que não têm uma pré-imagem em L são *criados*.

As entidades envolvidas em um sistema são descritas através de um *grafo de tipos*, onde cada entidade é apresentada com seus atributos, as mensagens que pode processar e gerar e os relacionamentos que possui com outras entidades.

O comportamento da especificação é determinado pela aplicação de regras aos grafos representando o estado real do sistema, partindo-se de um *grafo inicial*, no qual é representado o estado inicial do sistema. Múltiplas regras podem ser aplicadas em paralelo se não houver conflito entre elas (duas ou mais regras não podem modificar um mesmo item). A aplicação de uma regra a um grafo *G* é chamada de *passo de derivação*. Um passo de derivação só é possível se existe uma ocorrência do lado esquerdo da regra no grafo *G*. Ou seja, uma regra é aplicada somente se o grafo presente no lado esquerdo desta regra ocorre atualmente no grafo *G*.

A semântica de Gramáticas de Grafos pode ser definida como o conjunto de todas as computações que podem ser realizadas usando as regras da gramática, partindo-se do estado inicial do sistema.

3 Linguagem de Especificação Formal para SDCM

A LEF proposta em [DOT00b] baseia-se em Gramáticas de Grafos (apresentada na Seção 2.2.3.1) e é uma linguagem concebida para descrever SDCM. Mais especificamente, esta LEF trabalha com *Gramáticas de Grafos Baseadas em Objetos (GGBO)*, a qual é uma forma restrita de Gramáticas de Grafos. Esta escolha tem a vantagem de as especificações adquirirem um estilo baseado em objetos, que é bastante familiar à maioria dos desenvolvedores, tornando-as, por conseqüência, fáceis de construir e entender. Isto também possibilita que elas sirvam como base para uma implementação, além de facilitar a análise da gramática.

Um *sistema baseado em objetos* (*SBO*) pode ser descrito como um sistema no qual entidades autônomas chamadas *objetos* podem se comunicar e cooperar entre si através de *mensagens*. Cada objeto possui seu estado interno e o comportamento de um objeto é descrito pelas *reações* que ele assume ao receber mensagens. Uma reação pode gerar a modificação do estado interno do objeto e/ou o envio de mensagens a outros objetos.

Representando-se SBO com GGBO, as reações ao recebimento de uma mensagem são especificadas através de regras. Toda regra em GGBO tem como condição, em seu lado esquerdo, a existência de uma mensagem a ser tratada. A ocorrência do lado esquerdo de uma regra corresponde a encontrar a mensagem que dispara esta regra e verificar se os valores dos atributos lidos/alterados por esta regra são os especificados no lado esquerdo da regra. Assim, múltiplas regras podem ser executadas em paralelo, desde que seus lados esquerdos aconteçam.

Como forma de descrever um SBO através de Gramáticas de Grafos, as entidades deste devem ser apresentadas em um grafo, chamado *grafo modelo*. Neste grafo modelo são apresentadas as entidades e os relacionamentos que serão utilizados nas Gramáticas de Grafos. Assim, objetos, mensagens e atributos são modelados como vértices, como mostra a Figura 3.

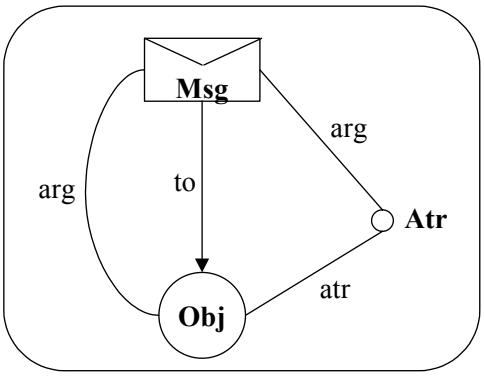

Figura 3. Grafo modelo de GGBO.

Uma mensagem deve ter como destino um objeto (arco direcionado, rotulado com *to*), o qual possui atributos (arco *atr*). Cada mensagem pode ter como argumentos objetos e/ou atributos (arcos *arg*).

Na Figura 4, é apresentado o grafo modelo de GGBO para SDCM. Como forma de representar um SDCM através de GGBO, o grafo modelo apresentado na Figura 3 foi estendido para que fosse possível representar *componentes móveis* (*component*) e *lugares* (*place*), sendo estas as entidades básicas de um SDCM.

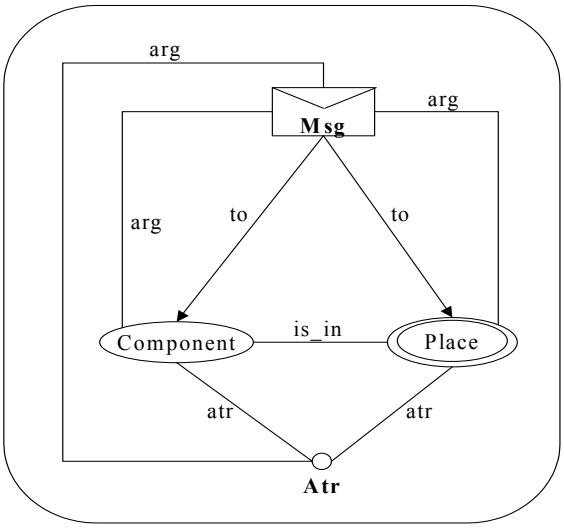

Figura 4. Grafo modelo de GGBO para SDCM

Lugares representam locais onde um componente pode executar. Eles oferecem serviços básicos (comunicação, armazenamento e poder de processamento) e a possibilidade de acesso a outros componentes que possuam uma interface bem definida (ex. serviço de nomes, serviço de eventos, etc.). *Componentes móveis* são componentes de

software que podem se mover durante sua execução de um lugar para outro e utilizar os serviços oferecidos pelo lugar aonde se encontram. Um componente móvel possui estado interno, código e um conjunto de atributos (identificador, lugar de origem, etc.). Mensagens podem ser enviadas de um componente móvel para outro, de um componente para o lugar onde está, de um lugar para os componentes que nele estão ou de um lugar para outro.

Conforme apresentado no grafo modelo de GGBO para SDCM na Figura 4, o arco rotulado com *is_in* informa o lugar em que um componente está, definindo que um componente está sempre em algum lugar. Os arcos direcionados rotulados com *to* indicam o receptor de uma mensagem. Uma mensagem deve possuir exatamente um receptor, apontado pela seta no fim do arco rotulado com *to* que parte da mensagem. Os arcos não direcionados, conectados à mensagem, indicam os parâmetros da mesma (podendo ser componentes, lugares e/ou atributos). Os arcos não direcionados que partem de componentes e lugares representam seu estado interno, apontando seus atributos. Os arcos e vértices são rotulados e possuem diferentes formas para que seja possível identificar-se o tipo de cada entidade. Um lugar é representado pelo símbolo com duas elipses concêntricas e um componente móvel, por uma elipse única.

Lugares e componentes móveis são especificados segundo o comportamento esperado para estas entidades. Este comportamento define as regras que regulam a atividade de cada entidade. Como citado na Seção 2.2.3.1, uma regra é especificada apresentando, do lado esquerdo, o subgrafo que deve estar presente no grafo do estado atual do sistema para que a mesma seja aplicada e, do lado direito, o grafo resultante da aplicação da regra. A seta que indica a transição de estados é rotulada com o nome da regra a ser aplicada, a qual pode possuir uma condição de aplicação, como mostra a Figura 5.

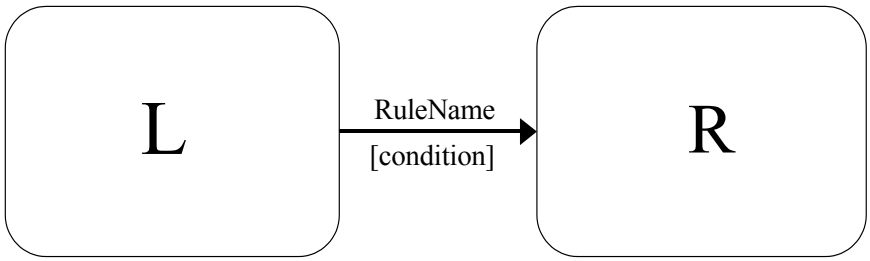

Figura 5. Exemplo de regra em gramáticas de grafos.

Dessa forma, uma regra tem um nome que a identifica (*RuleName*), um lado esquerdo, contendo um grafo *L* que determina o subgrafo que deve ser encontrado para que a regra seja aplicada, e um lado direito, contendo um grafo *R* que determina as alterações feitas no grafo *L* pela aplicação da regra. Uma regra pode ainda conter uma condição (*condition*) de aplicação. Esta condição apresenta-se como uma expressão lógica que envolve atributos da entidade à qual a regra pertence. O resultado de uma aplicação de regra (passo de derivação) segue a semântica descrita na Seção 2.2.3.1.

3.1 Regras Básicas das Entidades de um SDCM

Neste trabalho, considera-se que lugares e componentes móveis executam em um ambiente onde se pressupõe: A) a inexistência de falhas de comunicação; B) a garantia da integridade referencial (referências a componentes que se movem continuam válidas após a sua movimentação); C) a possibilidade de comunicação local e remota. Com relação ao item A, apesar de o projeto ForMOS considerar ambientes abertos, onde falhas podem acontecer, este trabalho restringe-se ao caso de um ambiente sem falhas por questões de simplicidade. A consideração de falhas merece um estudo posterior. Já os itens B e C são assumidos por estas serem características encontradas em plataformas de suporte à mobilidade disponíveis.

Tendo-se lugares e componentes móveis, definidos conforme apresentado no início desse capítulo, e o ambiente descrito, foram definidas regras básicas que descrevem o comportamento padrão de lugares e componentes móveis dentro do processo de movimentação. Tais regras foram criadas seguindo as definições de GGBO e são divididas em quatro grupos: regras para criação e deleção, regras de comunicação, regras de lugares e regra de componentes móveis. Cada grupo de regras é apresentado a seguir, sendo descrito, ao fim desta seção, como tais regras podem ser utilizadas.

3.1.1 Regras para Criação e Deleção de Entidades

A criação e a deleção de entidades são estabelecidas através de esquemas de regras. *Esquemas de regras* indicam configurações básicas que devem ocorrer nos lados esquerdo e direito de uma regra. Elementos adicionais da regra são referentes à regra da aplicação específica.

Conforme define Gramáticas de Grafos, a criação de uma entidade ocorre sempre que esta entidade aparece no lado direito de uma regra, mas não está presente no lado

esquerdo. Assim, o esquema de regra para criação de uma entidade é o apresentado na Figura 6.

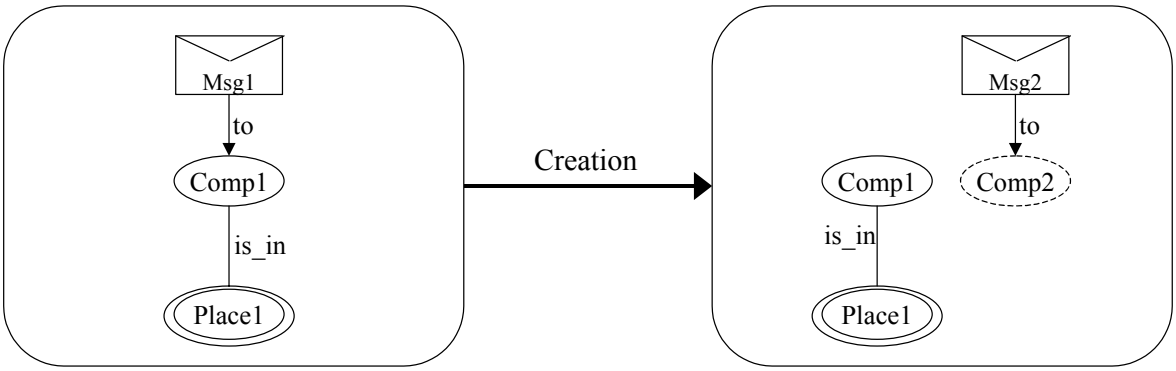

Figura 6. Esquema de regra de criação de entidade.

A ocorrência da mensagem *Msg1* do lado esquerdo do esquema de regra *Creation* se deve ao formalismo ser baseado em objetos e, por isso, toda computação ocorre a partir do recebimento de uma mensagem. Pelo que é definido no esquema de regra, ao receber uma certa mensagem *Msg1*, um componente *Comp1* pode criar outro componente *Comp2*. *Comp2* aparece em forma pontilhada porque se define que, ao ser criado, um componente não foi ainda inicializado. Isto é, o componente ainda não possui os valores iniciais para seus atributos internos. Antes de ser inicializado, o componente não pode tratar qualquer outra mensagem que não seja a mensagem de inicialização. A inicialização do componente ocorre segundo o esquema de regra *Init* apresentado na Figura 7, através do recebimento de uma mensagem que contém os valores necessários para estabelecer o estado inicial do componente (mensagem representada por *Msg2* na Figura 6).

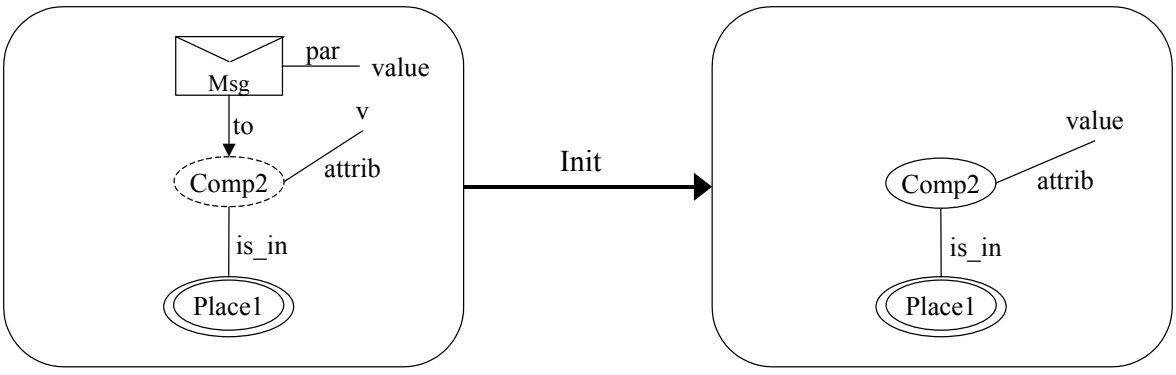

Figura 7. Esquema de regra de inicialização de uma entidade.

Após receber a mensagem de inicialização, o componente pode então passar a tratar as demais mensagens encaminhadas a ele. Isto é denotado com o componente passando a

ser representado com linha contínua, significando que o componente foi inicializado e está ativo. Deve-se dizer que se considera apenas a criação dinâmica de componentes móveis. Neste trabalho, lugares são criados no início do sistema e permanecem constantes durante toda a execução. Não se considera aqui a possibilidade de falha de um lugar.

Quanto à deleção de uma entidade, ela ocorre sempre que uma entidade aparece no lado esquerdo da regra mas não está presente no lado direito, significando que a aplicação da regra causou a exclusão da entidade do sistema.

3.1.2 Regras de Comunicação

As *regras de comunicação* definem as formas de troca de mensagens entre entidades. As regras de comunicação são descritas por esquemas de regras para passagem de mensagens.

Como dito anteriormente, em GGBO toda computação é resultado da passagem de mensagens. Por isso, todas as regras construídas representam a transformação do grafo devido ao recebimento de uma mensagem. A forma como uma entidade obtém uma referência para a entidade com a qual queira se comunicar não é indicada. Este mecanismo deve ser definido pela aplicação. As possibilidades são o armazenamento de uma referência em um atributo interno da entidade ou a passagem de uma referência à entidade de origem como parâmetro de uma mensagem, permitindo que a entidade que recebe a mensagem possa retornar uma resposta.

A passagem de mensagens pode ocorrer entre componentes móveis, entre lugares e entre lugares e componentes móveis. O esquema de regra *Send* (Figura 8) define a troca de mensagens entre componentes móveis, onde um componente *Comp1*, ao receber uma mensagem *Msg1*, envia uma mensagem *Msg2* ao componente *Comp2*. Note-se que esta comunicação pode ocorrer local ou remotamente.

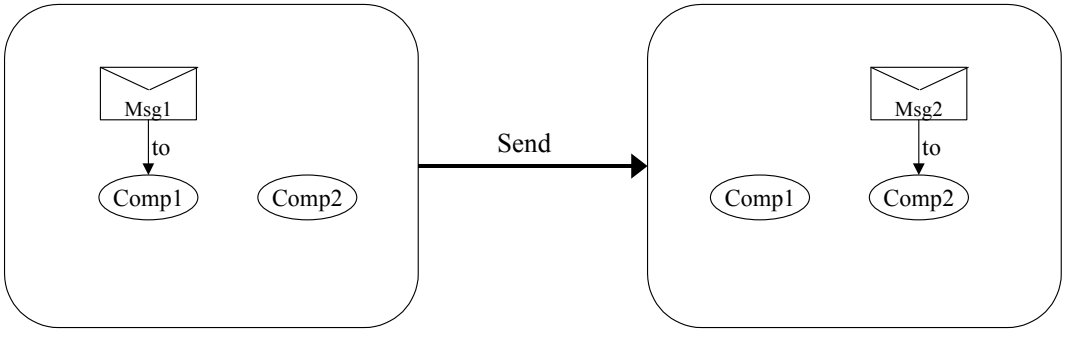

Figura 8. Esquema de regra de passagem de mensagens entre componentes móveis.

A Figura 9 apresenta o esquema de regra para comunicação entre lugares (*CooperationRequest*), o qual é idêntico ao esquema de regra de comunicação entre componentes móveis, apenas substituindo-se os componentes por lugares.

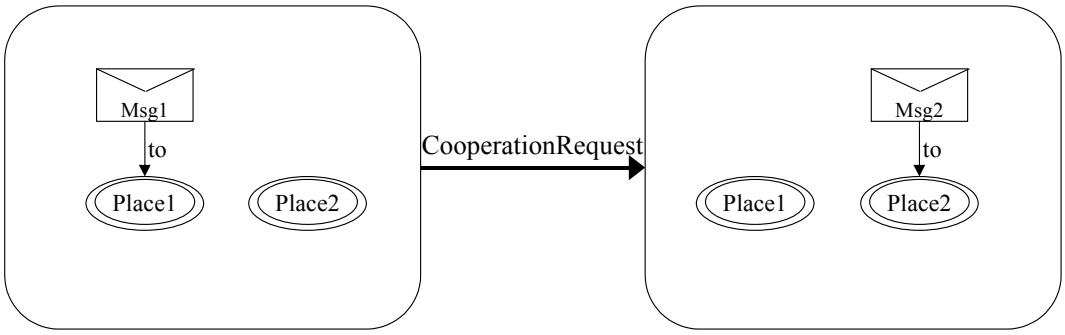

Figura 9. Esquema de regra de passagem de mensagens entre lugares.

Na Figura 10 é apresentado o esquema de regra *ServiceRequest*, representando a passagem de mensagens de um componente para um lugar. Como pode ser visto, este esquema de regra impõe uma restrição para comunicação entre componentes e lugares: o componente deve estar no lugar com o qual se comunica. Ou seja, um componente só pode requisitar serviços ao lugar onde se encontra (através do envio de mensagens). Dessa forma, não existem mensagens remotas de componentes para lugares.

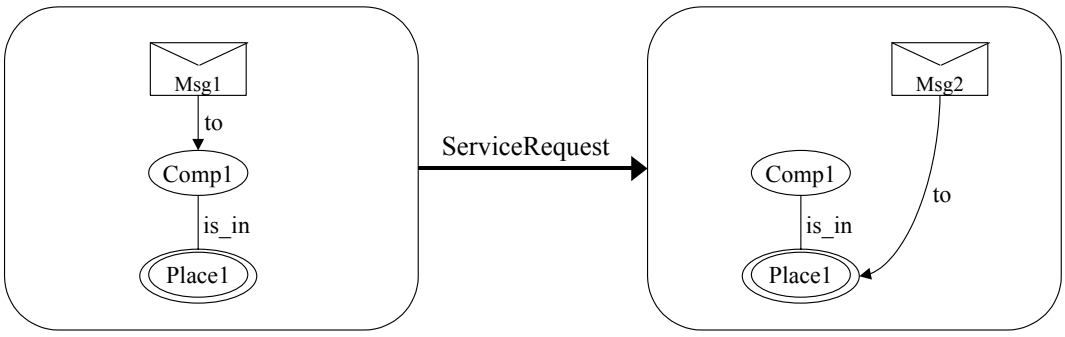

Figura 10. Esquema de regra de passagem de mensagens de um componente para um lugar.

O esquema de regra de passagem de mensagens de um lugar para um componente pode ser visto na Figura 11.

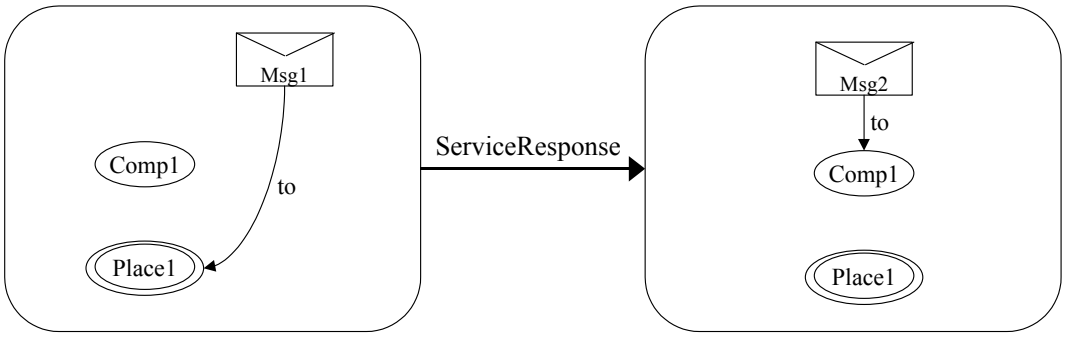

Figura 11. Esquema de regra de passagem de mensagens de um lugar para um componente.

Diferentemente do esquema de regra apresentado na Figura 10, o esquema de regra *ServiceResponse* não define nenhuma restrição para que um lugar envie mensagens para componentes localizados remotamente. Portanto, define-se que lugares podem enviar mensagens para qualquer componente, local ou remoto.

3.1.3 Regras de Lugares

As *regras de lugares* determinam o comportamento padrão de um lugar no processo de movimentação de componentes. A passagem de mensagens entre entidades segue os esquemas de passagem de mensagens descritos na subseção anterior, com a adição dos elementos necessários à composição do cenário desejado.

Na regra *RequestMove*, apresentada na Figura 12, um componente *Comp1* requisita ao lugar *Orig* a sua movimentação para o lugar *Dest*. Isto ocorre através do envio, por parte do componente, de uma mensagem *Move* ao lugar *Orig*. Tal mensagem leva, como parâmetros, o componente que pede a movimentação e o local de destino desta movimentação. Além disso, pode acompanhar a mensagem a informação de alguns requisitos que devem ser atendidos para que a movimentação ocorra, tais como disponibilidade de recursos e possibilidade de fornecimento de determinados serviços no destino. O uso destes requisitos é para consideração futura.

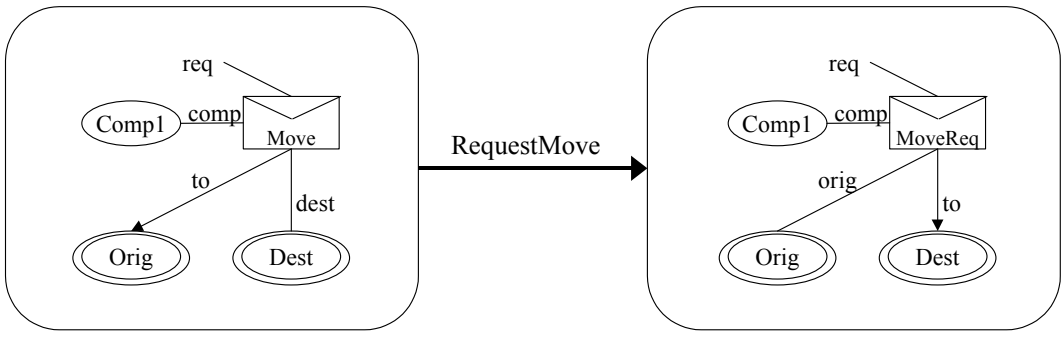

Figura 12. Regra de lugar *RequestMove*.

A regra *Move* (Figura 13) define que, ao receber uma mensagem de requisição de movimentação de um componente, o lugar de destino da movimentação verifica seu estado interno e confere se pode atender os requisitos necessários e receber o componente requisitante. A partir desta verificação, ele gera uma mensagem de resposta ao pedido de movimentação, encaminhada ao lugar de origem, informando se pode ou não receber o componente.

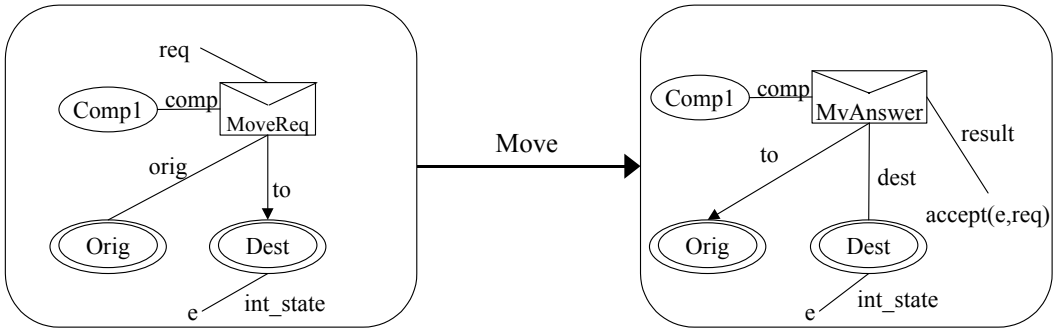

Figura 13. Regra de lugar *Move*.

Quando chega uma resposta positiva do lugar de destino, o lugar de origem segue o comportamento apresentado na regra *AcceptMove* (Figura 14). Assim, o lugar de origem realiza três operações: atualiza seu estado interno, para refletir a saída do componente; gera uma mensagem ao lugar de destino informando que o componente agora está localizado no novo lugar e, portanto, ele pode atualizar seu estado interno para refletir esta situação; e gera uma mensagem ao componente para que este atualize seu atributo interno de localização, passando a referenciar o seu novo local.

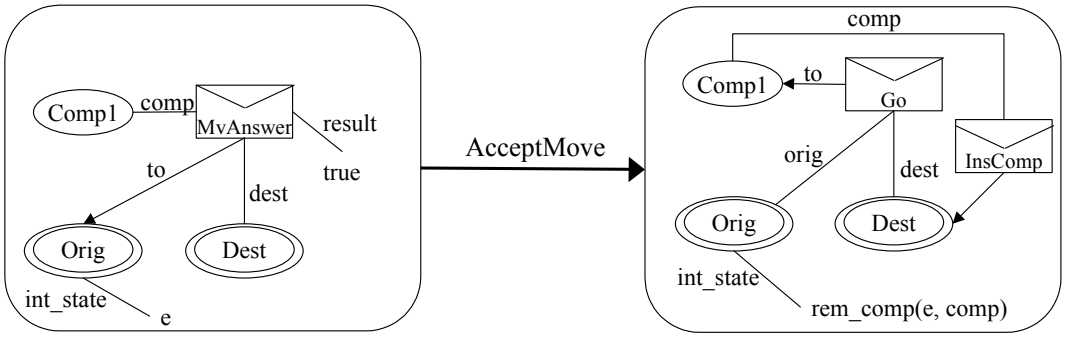

Figura 14. Regra de lugar *AcceptMove*.

O lugar de destino, ao receber a mensagem de aviso de inserção de componente do lugar de origem, tem a regra *InsertComponent* (Figura 15) ativada, causando a atualização

do seu estado interno com a inserção do componente movido.

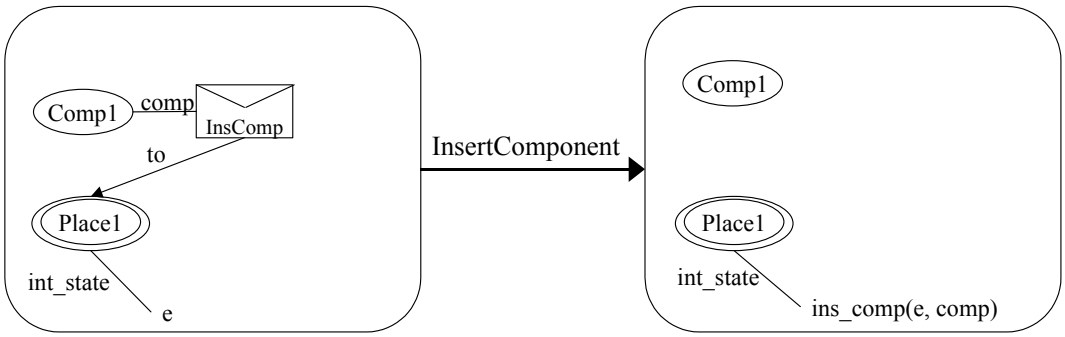

Figura 15. Regra de lugar *InsertComponent*.

A regra *DenyMove* (Figura 16) apresenta o comportamento do lugar de origem quando a resposta a um pedido de movimentação de componente é negado pelo lugar de destino.

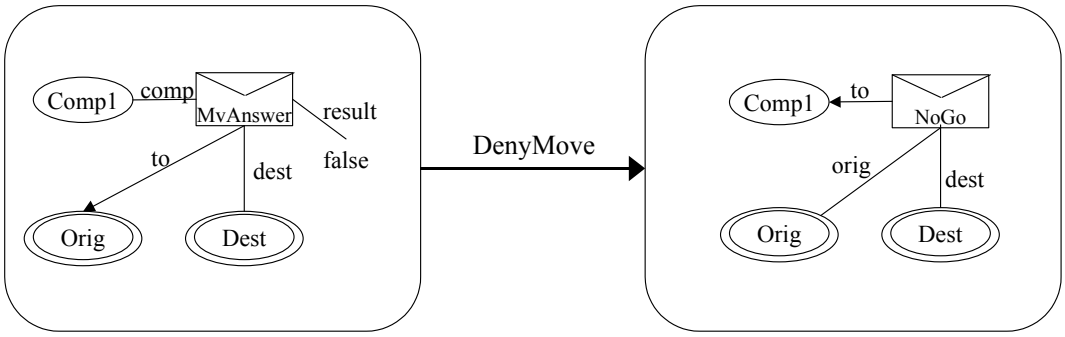

Figura 16. Regra de lugar *DenyMove*.

Tendo sido negado o pedido, o lugar de origem envia uma mensagem ao componente que requisitou a movimentação informando-o de que não foi possível movê-lo.

3.1.4 Regra de Componentes Móveis

A *regra de componentes móveis* é apresentada na Figura 17. Esta é a única regra necessária para componentes móveis dentro do processo de movimentação.

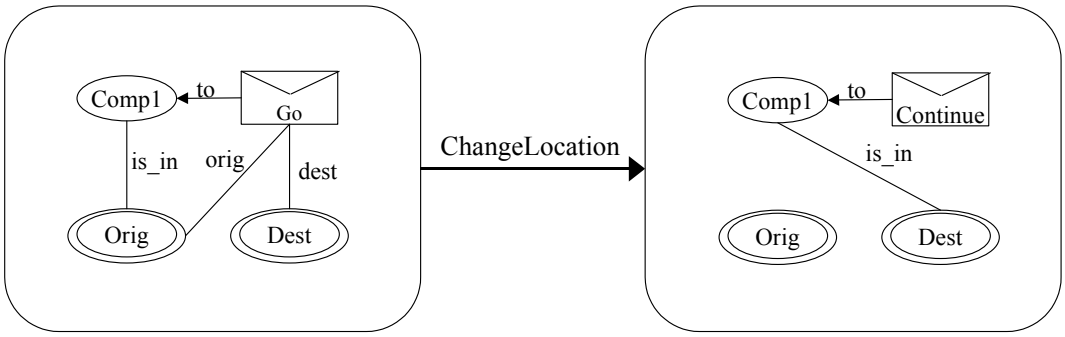

Figura 17. Regra de componentes móveis.

A regra *ChangeLocation* ocorre sempre que um pedido de movimentação é atendido e o local de origem realizou a regra *AcceptMove* (vide Figura 14). Ao receber a confirmação de sua movimentação, o componente atualiza seu atributo interno de localização para o seu novo lugar de execução.

3.1.5 Utilização das Regras Definidas

As regras aqui definidas servem para serem utilizadas dentro das especificações para SDCM. Como pode ser notado, não existe, entre estas regras, nenhuma regra que gere a mensagem de *Move* que dispara a execução da regra *RequestMove* (Figura 12) e nenhuma regra que seja disparada pelas mensagens *NoGo*, gerada pela regra *DenyMove* (Figura 16), e *Continue*, gerada pela regra *ChangeLocation* (Figura 17). Isto porque estas regras ausentes devem ser definidas pela aplicação móvel. Desta forma, conforme pode ser visto no esquema apresentado na Figura 18, a aplicação deve possuir uma ou mais regras que iniciam o processo de movimentação, com o envio de uma mensagem de *Move* de um componente móvel para o seu lugar de origem. Esta mensagem dispara o processo de movimentação, segundo as regras anteriormente definidas. A resposta ao pedido de movimentação é obtida pela aplicação com a existência de uma regra que é disparada pelo recebimento, pelo componente, de uma mensagem de *Continue* e outra que é disparada por uma mensagem *NoGo*.

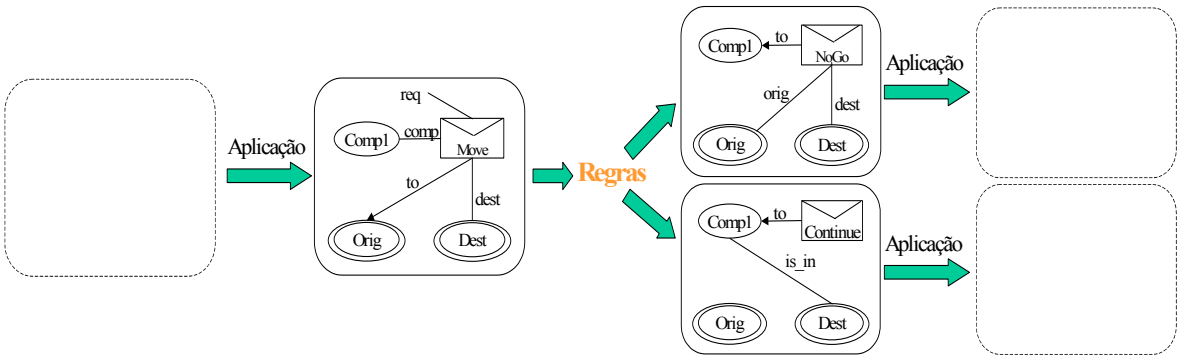

Figura 18. Esquema de uso das regras definidas.

Isto é, do ponto de vista do especificador da aplicação, um pedido de movimentação pode ter sucesso, indicando que o componente foi movido para o destino desejado e recebeu uma mensagem de *Continue* para retomar sua execução, ou ter a movimentação negada pelo destino, indicando que o componente permanece no local de origem. Neste último caso, uma mensagem de *NoGo* é enviada ao componente, sendo que a aplicação deve prever o comportamento a ser assumido para tratar esta situação.

Com uso das regras descritas, a especificação da aplicação só envolve questões pertinentes à própria aplicação, participando da movimentação apenas na geração do pedido de movimentação e na obtenção e tratamento da resposta.

3.1.6 Exemplo de Especificação

Para exemplificar o uso das regras básicas, será apresentado um exemplo de aplicação móvel especificada em GGBO. Esta aplicação envolve quatro tipos de entidades: *customer*, *mobile component*, *place* e *information server*. O cenário que envolve estas entidades define que um *customer* deseja saber qual o melhor preço para um dado produto *product1* e qual o lugar onde este produto é oferecido por este preço. Para isso, o *customer* envia um *mobile component (MC)* para se mover entre os *places*. Ao chegar a cada *place*, o *MC* consulta o *information server (IS)* local, perguntando qual o preço para *product1*. Comparando os preços de cada local por onde passa, o *MC* armazena o melhor preço e o *place* onde ele foi encontrado. Terminadas as visitas previstas, o *MC* retorna ao local onde está o *customer* e retorna para este os resultados de sua pesquisa.

Definido o cenário, o próximo passo é a especificação formal do sistema. A especificação possui três partes: o *grafo de tipos*, o *grafo inicial* e as *regras das entidades*.

O grafo de tipos, como descrito anteriormente, apresenta as entidades envolvidas no

sistema e seus atributos. O grafo de tipos desta aplicação é apresentado na Figura 19.

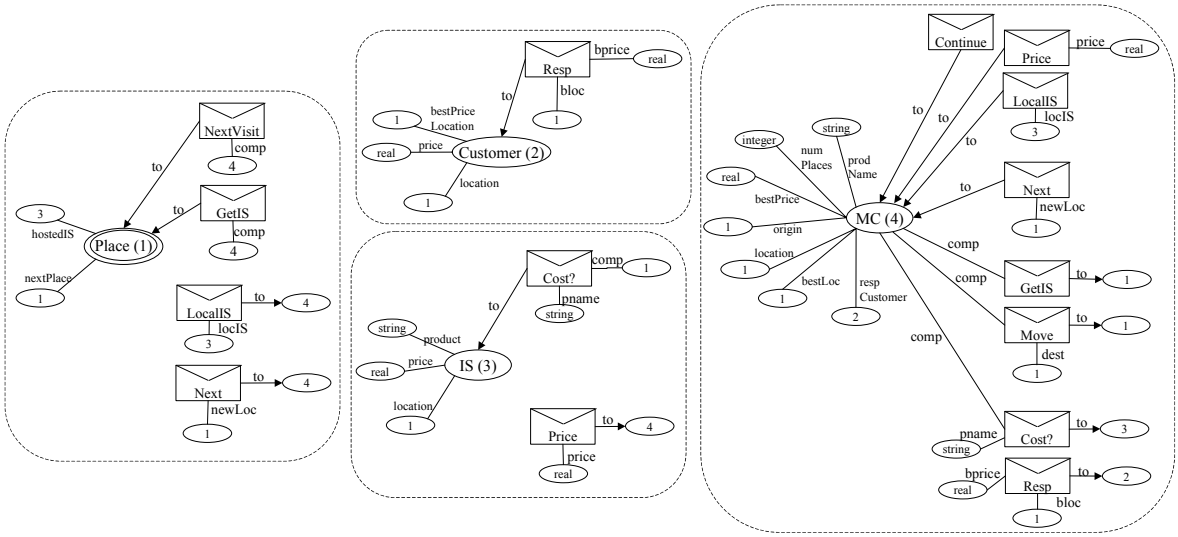

Figura 19. Grafo de tipos das entidades da aplicação de exemplo.

Os números utilizados nos grafos de tipos apresentados representam as entidades segundo o número colocado entre parênteses após o nome de cada entidade. Assim, o número 1 representa uma referência a um *Place*, o número 2 representa uma referência a um *Customer*, o número 3 representa uma referência a um *IS* e o número 4 representa uma referência a um *MC*. Dessa forma, por exemplo, o atributo *location* de um *MC* armazena uma referência a um *Place*. Estes número foram utilizados para simplificar os grafos de tipos.

No grafo de tipos da aplicação, cada entidade é apresentada com o seu próprio grafo de tipos, onde a entidade é apresentada no lado esquerdo com seus atributos. No lado direito acima, são apresentadas as mensagens que podem ser recebidas pela entidade. Cada mensagem aparece com seus parâmetros. No lado direito abaixo, são mostradas as mensagens enviadas pela entidade, também apresentadas com os seus respectivos parâmetros, sendo indicadas as entidades de destino de cada mensagem.

De acordo com o grafo de tipos, um *MC* possui um atributo *numPlaces* que indica quantos lugares devem ser visitados e um atributo *respCustomer*, que define uma referência ao *customer* ao qual deve ser retornado o resultado do seu trabalho. Além disso, ele possui atributos de controle de sua localização e de armazenamento dos valores de melhor preço e lugar onde este foi encontrado. Um *place* possui um *IS* e a informação do próximo *place*. Um *IS* possui as informações de nome de um produto e o preço para o mesmo. Um *costumer* possui atributos para receber os resultados da pesquisa do *MC* e a

informação de localização. O grafo inicial para esta aplicação é apresentado na Figura 20.

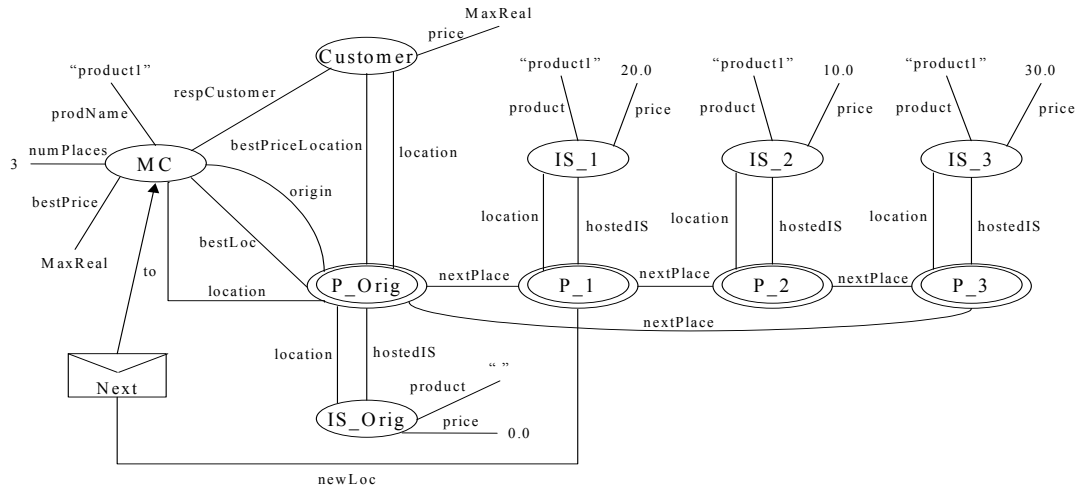

Figura 20. Grafo inicial do sistema.

Como descrito, no grafo inicial tem-se um *customer*, localizado em um lugar *P_Orig* que possui um *MC* que deve procurar o melhor preço para *product1* em 3 *places*. Cada *place* possui o seu *IS*, contendo o preço para *product1*, e a informação de qual o próximo *place*[5]. A mensagem *Next*, enviada a *MC* define o início da execução do sistema. A Figura 21 e a Figura 22 apresentam as regras para o *MC*.

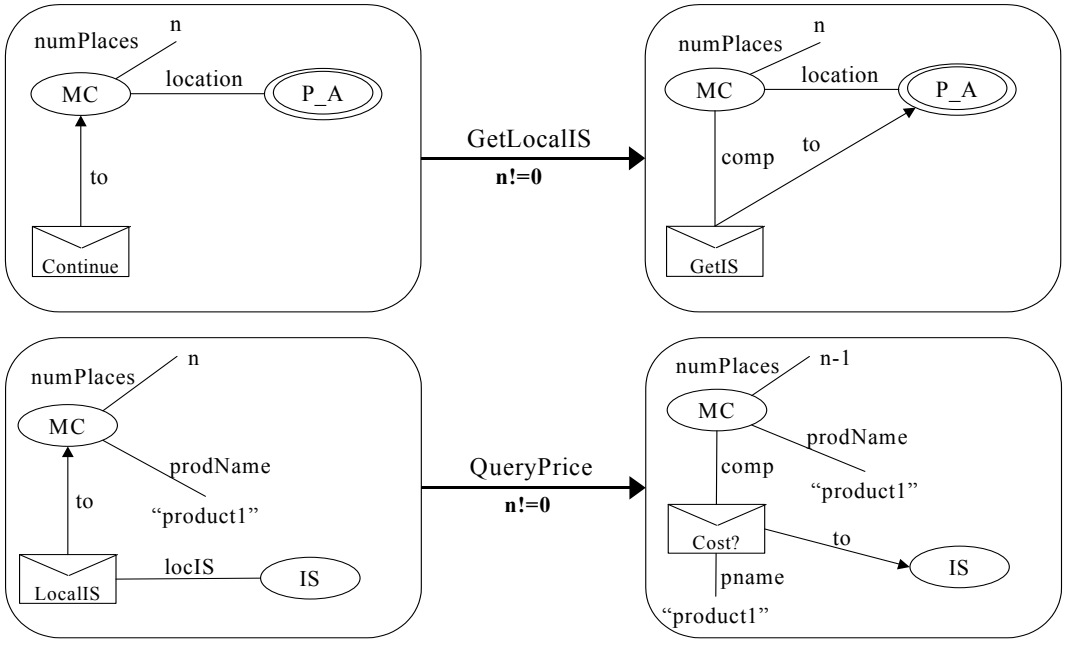

Figura 21. Regras do *MC* – Parte 1.

[5] Assume-se aqui uma topologia onde cada lugar conhece o seu próximo, seguindo-se uma ligação unidirecional entre os lugares.

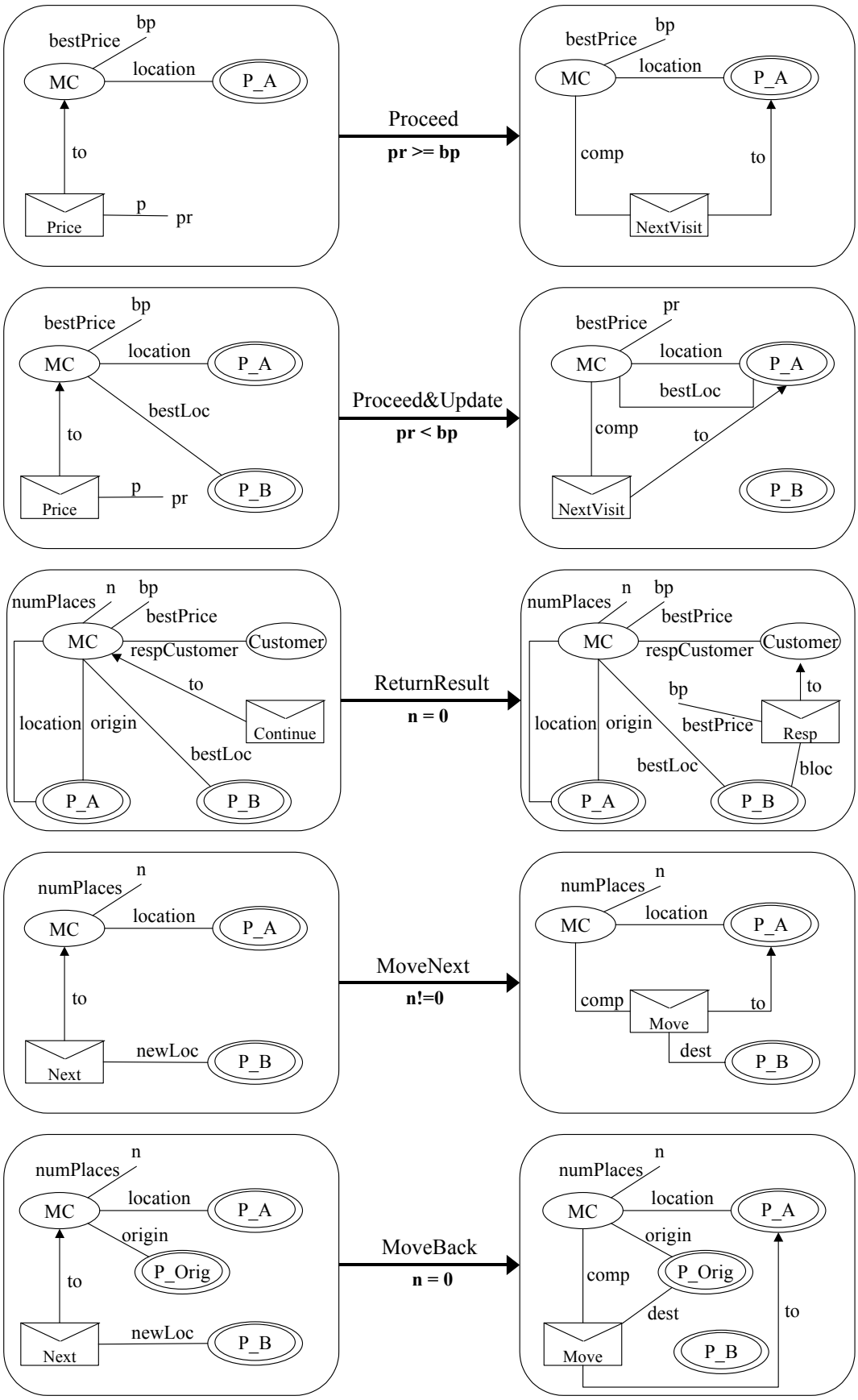

Figura 22. Regras do *MC* – Parte 2.

O comportamento descrito pelas regras apresentadas define que, partindo-se da mensagem *Next* inicial (conforme apresentado no grafo inicial), o *MC* realiza a regra *MoveNext* (Figura 22). Apesar de haver outra regra que também trata a mesma mensagem (*MoveBack*, na Figura 22) a condição de execução (constante sob a seta de transição da regra) define que a primeira seja executada. Executando a referida regra, o *MC* realiza um pedido de movimentação para o lugar recebido como parâmetro da mensagem. Após o envio da mensagem *Move*, o comportamento seguido é o descrito na regras básicas de movimentação, apresentadas neste capítulo. Assim, a próxima regra a ser executada é a regra que trata uma mensagem de *Continue*. Novamente há duas regras que tratam a mesma mensagem. Analogamente ao caso anterior, a condição de execução define qual tratará a mensagem. No caso, a regra *GetLocalIS* (Figura 21). Esta regra serve para que o *MC* obtenha uma referência ao *IS* local para poder consultar o preço oferecido. As regras do *place* são apresentadas na Figura 23. O comportamento do *place* ao receber a mensagem *GetIS* é apresentado na regra *InformIS*, retornando a referência pedida pelo *MC*.

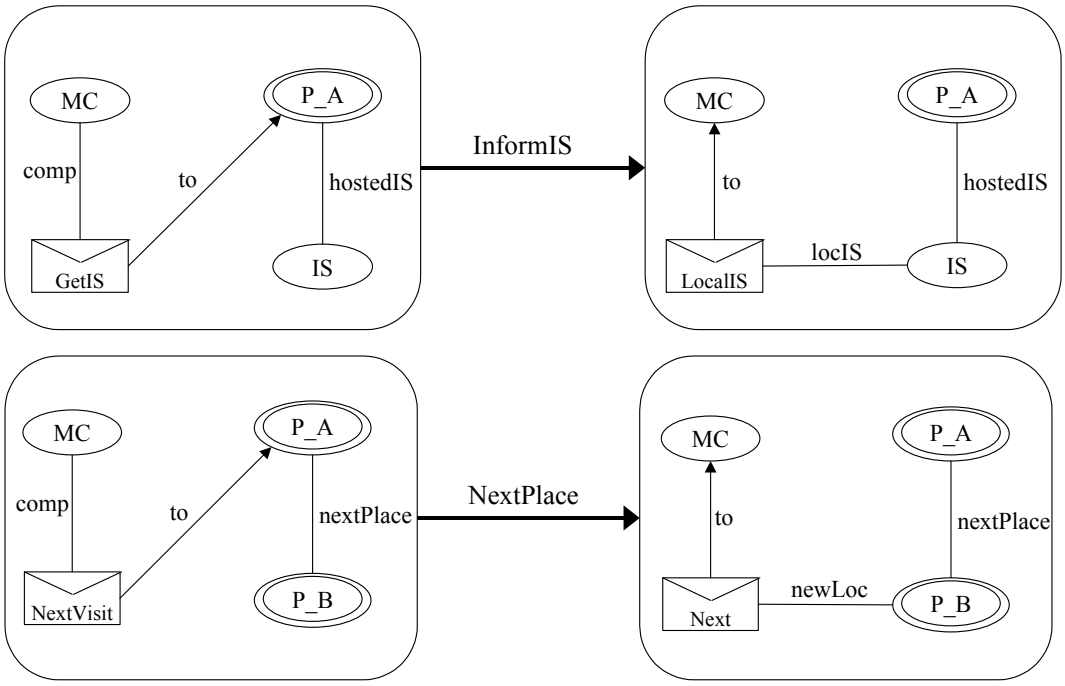

Figura 23. Regras do *place*.

Recebida a referência pedida, o *MC* pode então realizar a consulta ao preço oferecido para o produto *product1*, conforme descrito na regra *QueryPrice* (Figura 21). Note-se que, neste instante, o *MC* decrementa o número de *places* a visitar. O comportamento do *IS* quando consultado é descrito na Figura 24.

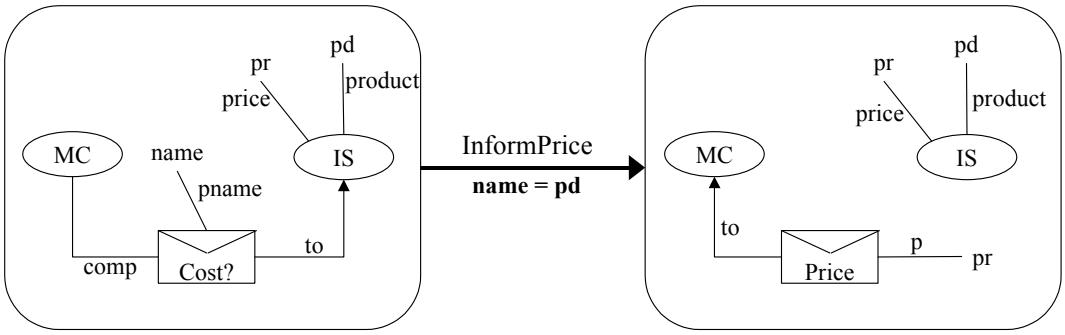

Figura 24. Regra do *IS*.

O IS retorna ao MC, conforme a regra *InformPrice*, o preço para *product1* como parâmetro da mensagem *Price*. Quando recebe a mensagem contendo o preço para *product1*, *MC* pode executar duas regras (apresentadas na Figura 22): *Proceed*, se o preço recebido é maior ou igual ao melhor preço já encontrado; ou *Proceed&Update*, caso o preço seja menor que o menor preço encontrado até então. Se ocorrer o segundo caso, o *MC* atualiza seus atributos, armazenando o preço recebido como o melhor encontrado e o seu lugar atual como sendo o lugar que possui o melhor preço. Tanto neste caso, como no caso de o preço não ser melhor que o encontrado anteriormente, o *MC* solicita ao *place* onde está a informação de qual é o próximo *place* (mensagem *NextVisit*), de modo que ele possa consultar o próximo lugar.

Ao receber a mensagem de *NextVisit*, o *place* executa a regra *NextPlace* (Figura 23), enviando a referência ao próximo *place* na mensagem *Next*, a qual dispara novamente todo o processo descrito até agora. A mudança ocorre quando todos os três *places* já foram visitados. Neste momento, ao receber a mensagem de *Next*, o *MC* verifica que *numPlaces* tem o valor 0 e executa a regra *MoveBack* (Figura 22), retornando ao lugar de origem. Terminada a movimentação e recebida a mensagem de *Continue*, o *MC* executa a regra *ReturnResult* (Figura 22), enviando os resultados de sua pesquisa para o *customer*. Recebidos os resultados do *MC*, o *customer* executa a regra *GetResp*, descrita na Figura 25, alterando seus atributos internos.

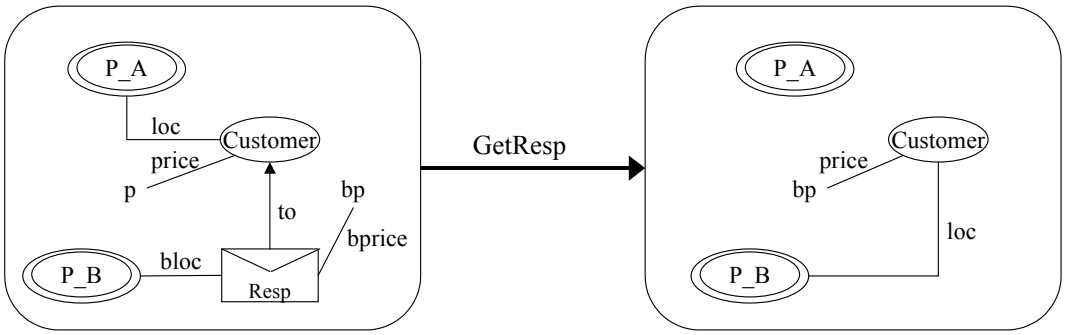

Figura 25. Regra do *customer*.

Com isto, a execução da aplicação está completa e o *customer* possui a identificação do *place* que possui o menor preço para o produto *product1*.

4 Simulador PLATUS

Modelos de simulação podem ser bastante úteis como um meio de especificação e teste. Um modelo de simulação é uma representação simplificada de um processo ou sistema que permite analisá-lo. A principal vantagem é a possibilidade de validar uma estratégia de projeto, bem como os algoritmos a serem utilizados, antes mesmo de sua implementação. Além disso, a existência de um modelo formal descrevendo o sistema torna as interdependências entre todos os seus componentes explícitas e claras [COP00].

O simulador desenvolvido no ambiente do projeto PLATUS [COP01] permite a realização de simulações sobre modelos descritos utilizando GGBO, apresentando um mapeamento de GGBO para criar seus modelos de simulação. Tal simulador representa uma ferramenta para testar a especificação de um sistema e, assim, identificarem-se erros durante o projeto do mesmo.

O grafo de tipos utilizado pelo simulador é apresentado na Figura 26, retirada de [COP01]. Os vértices representam *entidades*, *mensagens* ou *tipos abstratos de dados*. Entidades representam os objetos modelados. Os elementos *tmin* e *tmax* são *timestamps* que representam atributos especiais de uma mensagem. Estes *timestamps* são associados a cada mensagem e descrevem o intervalo de tempo no qual uma mensagem deve ser tratada em termos de tempo mínimo e máximo[6]. Tipos abstratos de dados (ADT) podem ser atributos de entidades ou parâmetros de mensagens.

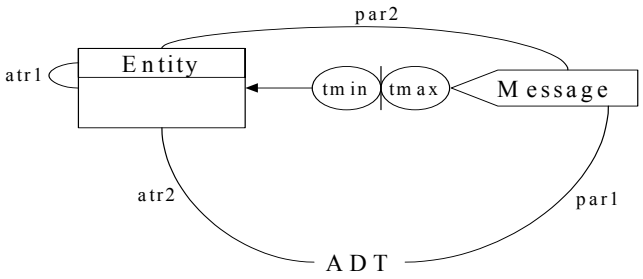

Figura 26. Grafo de tipos do simulador PLATUS.

O simulador conta com um *kernel*. O *kernel* é uma entidade especial que gerencia um relógio simulado que serve para sincronizar as demais entidades. O *kernel* é responsável

[6] No momento, somente *tmin* está implementado no simulador, determinando a demora para a entrega da mensagem a seu destino.

pela passagem de mensagens entre entidades. Toda entidade que queira enviar uma mensagem à outra entidade deve enviar a mensagem ao *kernel*, o qual fica encarregado de encaminhá-la ao destinatário. As mensagens recebidas por uma entidade são colocadas em um *buffer* de mensagens.

Cada entidade executa como uma *thread* e possui uma lista de regras que descrevem o seu comportamento. Ao retirar uma mensagem do *buffer*, a entidade verifica, dentre as regras de sua lista, quais podem tratar aquela mensagem. Esta seleção baseia-se, como definido em GGBO, na ocorrência do lado esquerdo da regra e em uma possível condição associada à regra. Do conjunto de regras habilitadas a tratar a mensagem recebida, uma é escolhida aleatoriamente, seguindo a mesma semântica de Gramáticas de Grafos. A regra escolhida é executada por uma *thread* criada para este fim. Ao final da execução da regra, a *thread* é finalizada. Caso uma mensagem recebida não possa ser tratada (condição de execução da regra que trata a mensagem não é satisfeita ou lado esquerdo da regra não ocorre no estado atual do sistema), ela é recolocada no *buffer* de mensagens e novas tentativas de tratá-la são feitas.

A implementação de uma entidade no simulador fornece a possibilidade de múltiplas regras de leitura de atributos executarem paralelamente. Isto é, múltiplas *threads* são criadas para executar simultaneamente diferentes regras que lêem atributos. Quando uma regra de escrita deve ser executada (regra que altera o valor de algum atributo da entidade), esta fica bloqueada até que todas as regras de leitura em execução terminem, a fim de evitar que ela altere os dados que estão sendo lidos pelas demais regras. Logo que não haja mais nenhuma regra de leitura executando, a regra de escrita é executada, bloqueando-se a execução de qualquer outra regra. Assim, tem-se que regras de leitura executam concorrentemente e regras de escrita executam isoladamente. Com isto, garante-se a geração de estados consistentes das entidades. Para isto, as regras têm de ser marcadas como regras de leitura ou escrita. A Figura 27 ilustra a execução de regras no simulador, apresentando um cenário onde executam 4 regras ao longo do tempo.

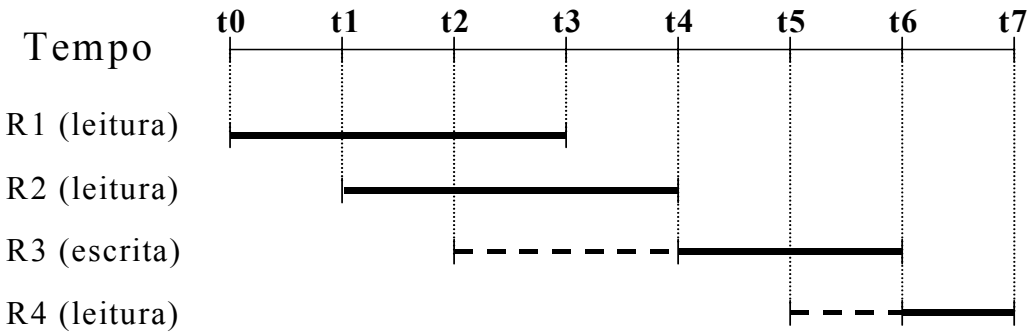

Figura 27. Exemplo da semântica de execução de regras do simulador.

O cenário da Figura 27 apresenta uma regra R1 de leitura que começa a executar em um tempo t0. No tempo t1, uma regra R2, também de leitura, começa a executar. Como R2 é uma regra de leitura, ela pode executar ao mesmo tempo em que R1. No tempo t2, uma regra R3 de escrita tenta executar mas, como existem regras de leitura executando, ela fica bloqueada até o tempo t4, quando ambas as regras de leitura que estavam em execução (R1 e R2) foram finalizadas. No tempo t5, uma regra R4 de leitura tenta executar, mas fica bloqueada por haver uma regra de escrita (R3) em execução. Somente no tempo t6 é que R4 passa a executar, quando R3 foi finalizada. Deve se citar que, caso outra regra de leitura tentasse executar, por exemplo, no tempo t3, ela seria postergada até que a regra de escrita R3 fosse finalizada, visto que a execução de qualquer outra regra é bloqueada após uma regra de escrita ter sido selecionada para executar. Isto impede a postergação indefinida da regra de escrita.

Deve-se dizer que, segundo GGBO, múltiplas regras poderiam executar simultaneamente, fossem elas de leitura ou de escrita,. Isto porque todas regras consideram os valores do lado esquerdo da regra no instante de início de sua execução. No momento, a semântica utilizada no simulador é a descrita anteriormente. Dessa forma, por enquanto, trabalha-se com um subconjunto das possibilidades de execuções simultâneas de regras permitidas por GGBO. Esforços têm sido realizados para permitir que regras de leitura executem ao mesmo tempo em que regras de escrita, tal como define a semântica de GGBO.

Segundo estudos preliminares, uma possibilidade de permitir o conjunto total de paralelismo entre regras seria, para cada entidade, realizar a duplicação de seus atributos. Assim, regras de leitura executariam lendo os atributos replicados e regras de escrita trabalhariam sobre os atributos originais da entidade. Ao final de um conjunto de regras disparadas, os resultados das atualizações das regras de escrita sobre os atributos da

entidade seriam copiados para os atributos replicados. Quando um novo conjunto de regras fosse ser executado, os atributos replicados já conteriam os valores atualizados. Nesta abordagem, a fim de minimizar a quantidade de atributos replicados, poder-se-ia incluir em cada regra a informação de quais atributos a regra altera. Desse modo, apenas os atributos necessários em cada execução em paralelo seriam replicados. Uma análise do impacto deste esquema no desempenho da entidade ainda não foi feita.

Ferramentas gráficas para auxiliar no uso do simulador estão em desenvolvimento. Tais ferramentas deverão permitir a criação de entidades e modelos e a geração automática de código, através da conversão da representação gráfica para o conjunto de classes correspondentes. Com isso, pretende-se que se possa criar graficamente os modelos em GGBO e, a partir destes, gerar código de simulação para os mesmos.

5 Geração de Código para Especificações Formais

A geração de código a partir de uma LEF exige conhecimento sobre o formalismo e a linguagem de programação utilizados. É preciso estudar como mapear as entidades presentes na LEF para uma representação coerente na linguagem de programação. Ou seja, para que a geração de código seja válida, este mapeamento deve ser cuidadoso para que o código gerado reflita a especificação criada, garantindo-se, assim, as vantagens obtidas por se ter uma descrição formal do sistema.

Em relação à geração de código para SDCM, também se considera a escolha de uma PSM. Como visto na Seção 2.1.3, uma PSM fornece uma biblioteca de mecanismos de suporte à mobilidade de código que estende uma linguagem de programação. A utilização de uma PSM torna mais fácil trabalhar-se com código móvel, mas, no caso da geração de código, exige que também se tenha um bom conhecimento das abstrações e dos serviços por ela oferecidos. A geração de código deve contemplar a criação de um código na linguagem de programação considerando o uso dos recursos fornecidos pela biblioteca da PSM, a fim de utilizar seus mecanismos de mobilidade de código.

O objetivo principal deste trabalho, como anteriormente citado, é permitir que, a partir de uma especificação formal de um SDCM, feita através da LEF descrita na Seção 3, gere-se código em uma linguagem de programação para uma PSM. Para atingir este objetivo, partiu-se do mapeamento que o simulador PLATUS usa para criar seus modelos de simulação. Como citado no Capítulo 4, tais modelos são criados a partir do mapeamento de uma especificação em GGBO para código em linguagem Java. Dessa forma, precisou-se adaptar o mapeamento existente para tornar possível a geração de código executável sobre uma PSM. Esta adaptação foi realizada em 4 etapas distintas:

1. *Geração de código para simulação*: nesta etapa, uma especificação em GGBO é mapeada para entidades e regras que executam no simulador PLATUS;
2. *Geração de código para execução local*: nesta etapa, operou-se a retira do *kernel* de simulação, tornando direta a comunicação entre as entidades e passando-se a ter a execução regida pelo tempo real;
3. *Geração de código para execução distribuída*: nesta etapa, ocorreu a introdução da PSM, permitindo a comunicação remota de entidades, as quais passaram a poder executar distribuidamente;

4. *Geração de código para execução com mobilidade*: na última etapa, introduziram-se modificações na estrutura das entidades para que fosse possível movimentá-las usando-se recursos da plataforma utilizada.

As etapas sucintamente apresentadas serão detalhadas nas seções seguintes. Para ilustrar o código gerado em cada etapa e demonstrar os reflexos no código das modificações realizadas, serão utilizados os códigos gerados para a entidade *MC* da aplicação de exemplo apresentada na Seção 3.1.6. O grafo de tipos da entidade *MC* é apresentado novamente na Figura 28.

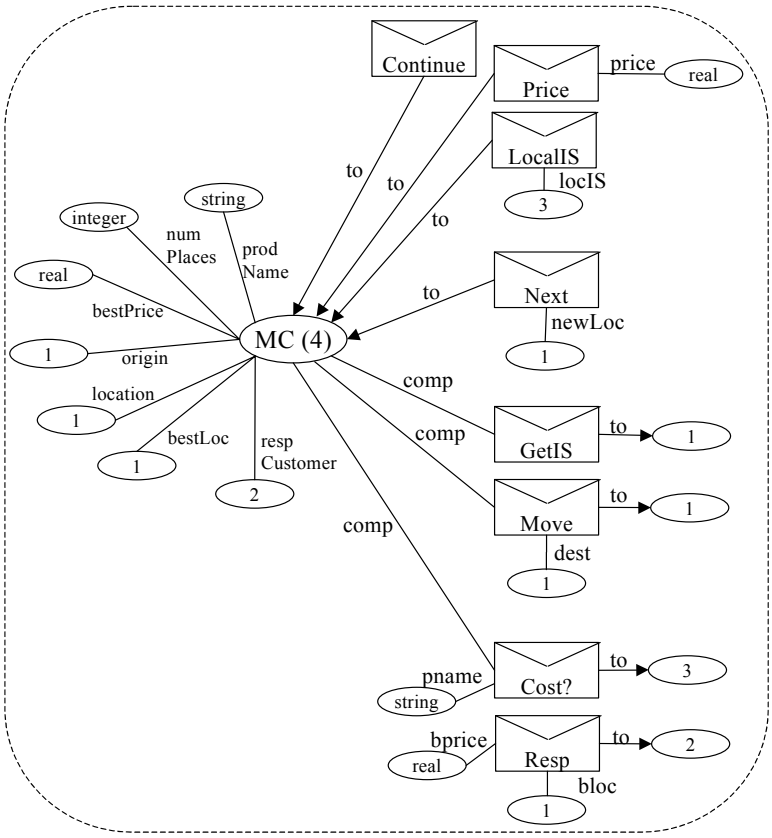

Figura 28. Grafo de tipos para entidade usada como exemplo.

Para exemplificar uma regra, será usada a regra *MC_R1* da referida entidade, cuja especificação é novamente apresentada na Figura 29.

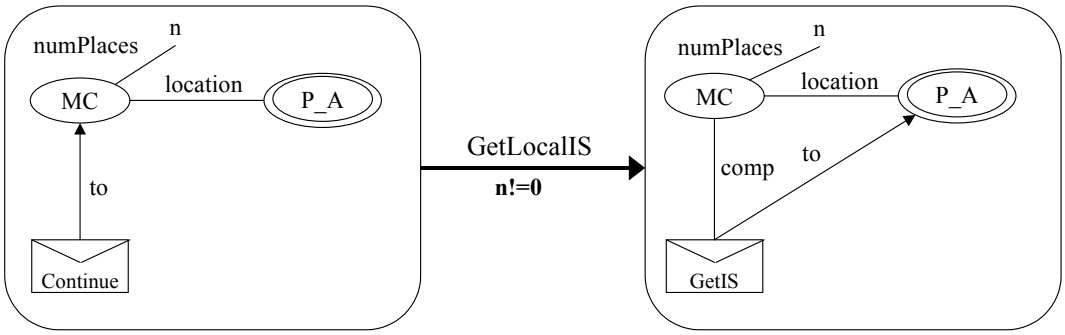

Figura 29. Especificação da regra usada como exemplo.

Nesta regra, há o recebimento de uma mensagem de *Continue* pela entidade *MC*, a qual está localizada em um lugar *P_A*. A regra possui, como condição de execução, que o valor de *numPlaces* deve ser diferente de 0. A aplicação da regra causa o consumo da mensagem *Continue* e a geração da mensagem *GetIS* e o seu envio para um lugar *P_A*, tendo como parâmetro uma referência à entidade *MC*.

5.1 Geração de Código para Simulação

Para a apresentação da geração de código para simulação, será discutido, inicialmente, o mapeamento de GGBO proposto no simulador PLATUS e, posteriormente, as modificações realizadas neste mapeamento no âmbito deste trabalho.

5.1.1 Mapeamento de GGBO para o simulador PLATUS

O simulador PLATUS foi concebido para a simulação de sistemas de produção e sistemas de tempo real. O mapeamento criado para representar uma especificação em GGBO é composto por três partes, descritas a seguir.

5.1.1.1 Mapeamento de uma Entidade de GGBO

O simulador PLATUS mapeia o comportamento básico de uma entidade de GGBO em uma classe Java denominada *Entity*. Nesta classe, encontram-se implementados os mecanismos de recebimento e tratamento de mensagens de uma entidade. Toda entidade de aplicação deve estender *Entity*, incorporando este comportamento básico. Dessa forma, existe uma hierarquia entre as entidades de aplicação e a classe *Entity*, onde as primeiras estendem a segunda, assim como mostrado na Figura 30, com *Entidade* representando uma entidade de aplicação.

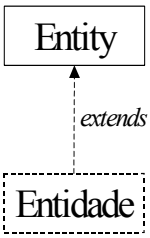

Figura 30. Hierarquia de entidades no simulador PLATUS.

O código básico de uma entidade de aplicação é apresentado na Figura 31.

```
1. public class <nome_entidade> extends Entity
2. {
3.  protected <tipo_atributo1> <nome_atributo1>;
4.  ...
5.  protected <tipo_atributoN> <nome_atributoN>;

6.  public <nome_entidade> (Kernel k) {super(k);}

7.  public void init (<tipo_param1> <nome_param1>, ..., <tipo_paramN>
8. <nome_paramN>) {
9.   <nome_atributo1> = <nome_param1>;
10.   ...
11.   <nome_atributoN> = <nome_paramN>;
12.  }

13.  public void set<nome_atributo1> (<tipo_atributo1> <nome_param1>) {
14.   <nome_atributo1> = <nome_param>;
15.  }
16.  ...
17.  public void set<nome_atributoN> (<tipo_atributoN> <nome_paramN>) {
18.   <nome_atributoN> = <nome_paramN>;
19.  }

20.  public <tipo_atributo1> get<nome_atributo1> () {
21.   return <nome_atributo1>;
22.  }
23.  ...
24.  public <tipo_atributoN> get<nome_atributoN> () {
25.   return <nome_atributoN>;
26.  }

27.  public void registerRules () {
28.   super.registerRules();
29.   addRule (new <nome_entidade>_R1 (<nome_msg1>, getKernel(), this));
30.   ...
31.   addRule (new <nome_entidade>_RN (<nome_msgN>, getKernel(), this));
32.  }
33. }
```

Figura 31. Constituição básica do código de uma entidade de aplicação.

As partes marcadas em negrito na Figura 31 representam o que varia na definição de uma entidade. Assim, a linha 1 apresenta o cabeçalho da classe que descreve a entidade, sendo que, como comentado, esta classe estende a classe *Entity*. A seguir (linhas 3 a 5) são

definidos os atributos da entidade, sendo fornecidos o tipo e o nome de cada atributo da entidade. Definidos os atributos, vem o construtor da classe (linha 6). O construtor de uma classe que descreve uma entidade de simulação recebe sempre uma instância do *kernel* como parâmetro. Dessa forma, a entidade sabe qual instância do *kernel* deve usar para comunicar-se com as demais entidades da aplicação. Esta informação é armazenada em um atributo da classe *Entity*. O método *init* (linhas 7 a 12) é usado para inicializar os atributos de uma entidade. Para isso, recebe, como parâmetros, todos os valores iniciais dos atributos. Estes valores são, então, atribuídos aos respectivos atributos. Para permitir que os valores dos atributos sejam modificados ou lidos ao longo da simulação da aplicação, são definidos, para cada atributo, um método *set* e um método *get* para escrita e leitura do atributo, respectivamente, assim como mostra o trecho entre as linhas 13 e 26 da Figura 31.

A última parte do código define o registro do conjunto de regras da entidade (linhas 27 a 32). Uma regra é adicionada ao conjunto de regras da entidade recebendo um nome (formado pelo nome da entidade mais um número de ordem da regra, tal como *Entidade_R1*). As regras são inicializadas com o nome da mensagem que dispara a regra, a instância do *kernel* usada na simulação (mesma recebida no construtor da entidade) e uma referência à entidade à qual a regra pertence.

5.1.1.2 Mapeamento de uma Regra em GGBO

Cada regra da aplicação é descrita por uma classe Java, a qual estende a classe *Rule*. A classe *Rule* define os métodos básicos de uma regra. Para uma regra, o código básico, mostrado na Figura 32, apresenta o método construtor, que recebe a identificação da regra, a referência à instância do *kernel* utilizada e a referência à entidade a qual a regra está associada (linhas 3 a 5). O método *answerMsg* (linhas 6 a 9) define qual o tipo da mensagem que a regra trata. O retorno deste método é utilizado pela entidade para definir quais de suas regras são candidatas a tratar uma mensagem recebida. Os métodos *conditionOk* (linhas 10 a 15) e *leftSideOccurs* (linhas 16 a 21) servem para definir condições e/ou restrições para a execução da regra. Com isso, no método *conditionOk* definem-se as condições associadas à regra, que são as condições colocadas sob a seta de transição na especificação da regra, que pode ser vista na Figura 29, no início desse capítulo. No método *leftSideOccurs* podem ser definidas restrições de valores de atributos da entidade em questão para que a regra seja executada, de acordo com o que é apresentado no lado esquerdo da especificação da regra. As condições testadas nos

métodos *conditionOk* e *leftSideOccurs* podem considerar apenas os atributos da entidade à qual a regra está associada e os parâmetros da mensagem tratada pela regra.

```
1. public class <nome_entidade>_R1 extends Rule
2. {
3.  public <nome_entidade>_R1 (String strId, Kernel k, Entity e) {
4.   super(strId, k, e);
5.  }

6.  public boolean answerMsg (Message m) {
7.   if (m.getId().equals(<nome_msg>)) return true;
8.   return(false);
9.   }

10.  public boolean conditionOk (Message m) {
11.   <nome_entidade> e = (<nome_entidade>) owner();
12.   Parameters par = m.getPar();
13.   if (<condicao>) return true;
14.   return false;
15.  }

16.  public boolean leftSideOccurs (Message m) {
17.   <nome_entidade> e = (<nome_entidade>) owner();
18.   Parameters par = m.getPar();
19.   if (<condicao>) return true;
20.   return false;
21.  }

22.  public void apply (Message m) {
23.   <nome_entidade> e = <nome_entidade> owner();
24.   Parameters par = m.getPar();
25.   <tipo_par1> <nome_par1> = (<tipo_par1>) par.get(<descr_par1>);
26.   ...
27.   <tipo_parN> <nome_parN> = (<tipo_parN>) par.get(<descr_parN>);

28.   <alteracoes no estado interno da entidade e/ou criação de novas
29.    entidades>

30.   par = new Parameters ();
31.   par.add(<descr_new_param1>, <valor_new_param1>);
32.   ...
33.   par.add(<descr_new_paramN>, <valor_new_paramN>);
34.   e.sendMessage(<timestamp>, new Message(e, <destino>, <nome_msg>,
35.                 par));
36.  }

37.  public boolean writeOnAttrib () {return <valor_retorno>);}
38. }
```

Figura 32. Constituição básica do código de uma regra de aplicação.

Caso uma regra trate uma mensagem recebida pela entidade (retorno verdadeiro do método *answerMsg*) e todas as condições e restrições de execução da regra sejam atendidas (retornos verdadeiros dos métodos *conditionOk* e *leftSideOccurs*), então ela pode

ser aplicada. A aplicação de uma regra é definida no método *apply* (linhas 22 a 36). Neste método, a mensagem recebida é tratada, podendo causar a alteração do estado interno da entidade (alteração dos valores de atributos) e o envio de novas mensagens a outras entidades. A primeira providência na aplicação da regra é obter a entidade a que pertence a regra (linha 23). Os valores dos parâmetros da mensagem a ser tratada são então obtidos (linhas 24 a 27). A seguir, são realizadas eventuais alterações no estado interno da entidade, com a utilização dos métodos *set* da entidade para modificar os valores de seus atributos. Também pode ser realizada a criação de novas entidades.

A aplicação de uma regra pode gerar mensagens, conforme apresentado nas linhas 30 a 35 da Figura 32. Cada parâmetro de uma mensagem a ser enviada é adicionado à lista de parâmetros passando-se a sua descrição (*string* que nomeia o parâmetro) e o valor associado ao parâmetro. As linhas 34 e 35 apresentam o comando de envio de mensagem. O método utilizado para tanto é o *sendMessage*, que implementa um mecanismo de envio da mensagem ao *kernel*, o qual se encarregará de enviá-la ao destino especificado. O método *sendMessage* recebe um *timestamp* associado à mensagem (o qual define o tempo de simulação do atraso na entrega da mensagem) e a própria mensagem, a qual contém a entidade que está enviando a mensagem, a entidade a que se destina a mensagem, o nome da mensagem e os seus parâmetros.

O último método de uma regra é o método *writeOnAttrib* (linha 37), usado para definir se a regra altera ou não atributos da entidade. Ou seja, este método define se a regra é de escrita ou de leitura. O valor do retorno deste método (*true* ou *false*) vai definir, segundo a semântica de concorrência implementada no simulador (vide Capítulo 4), quais regras podem ser executadas em paralelo.

5.1.1.3 Mapeamento do Grafo Inicial

Uma aplicação especificada em GGBO, para ser simulada no simulador PLATUS, deve possuir um programa principal, o qual descreve o estado inicial do sistema. Neste programa principal são instanciadas as entidades da aplicação, é feita a sua inicialização e as mensagens iniciais da aplicação são geradas. Isto é, o programa principal mapeia o grafo inicial do sistema.

O código básico de um programa principal de uma aplicação especificada em GGBO a ser simulada é apresentado na Figura 33.

```
1. class <nome_programa>
2. {
3.  public static void main(String args[]) {
4.   Kernel k = new Kernel ();

5.   k.setDeltaT(0.01);

6.   <nome_entidade1> e1 = new <nome_entidade1> (k);
7.   ...
8.   <nome_entidadeN> eN = new <nome_entidadeN> (k);

9.   e1.init(<valor_param1>, ..., <valor_paramN>);
10.  ...
11.  eN.init(<valor_param1>, ..., <valor_paramN>);

12.  k.addEntity(e1);
13.  ...
14.  k.addEntity(eN);

15.  Parameters par = new Parameters ();
16.  par.add(<descr_param1>, <valor_param1>);
17.  ...
18.  par.add(<descr_paramN>, <valor_paramN>);

19.  k.postMessage(0.0, new Message(null, <destino>, <nome_msg>, par);

20.  k.start();
21.  }
22. }
```

Figura 33. Código básico de um programa principal de uma aplicação.

O programa principal é definido em uma classe Java que recebe o nome da aplicação (linha 1). No método principal desta classe (*main*) é descrito o estado inicial do sistema. A linha 4 apresenta a instanciação do *kernel* a ser utilizado na simulação da aplicação. Na linha 5 está o comando para a definição do intervalo de avanço do relógio simulado. Assim, de acordo com o valor colocado, o relógio avança 0,01 unidades do tempo simulado por vez. A seguir vêm as instanciações das entidades da aplicação (linhas 6 a 8), com a passagem de uma referência à instância do *kernel* utilizada. As entidades são, depois, inicializadas com os valores iniciais de seus atributos (linhas 9 a 11). As entidades instanciadas devem ser adicionadas à lista de entidades servidas pelo *kernel*, a fim de que possam trocar mensagens entre si. A adição das entidades ocorre conforme apresentado nas linhas 12 a 14.

Uma mensagem inicial é definida através da criação da sua lista de parâmetros (linhas 15 a 18) e do seu envio ao *kernel* conforme o comando da linha 19. Múltiplas mensagens iniciais podem ser definidas. O *kernel* é inicializado através do comando da linha 20, o que causa o início da simulação da aplicação.

Apesar de não existir ainda um tradutor implementado para mapear especificações em GGBO para código do simulador, segundo uma análise prévia, este mapeamento parece passível de ser realizado de forma automática. Para realizar este mapeamento, devem ser cumpridas duas etapas: 1) mapeamento da especificação para estruturas de dados; 2) utilização das estruturas de dados para gerar o código de simulação. Em [DIA01] foi proposta uma forma de realizar este mapeamento, sendo definidas estruturas de dados para comportar as informações pertinentes de uma especificação em GGBO. Posteriormente, essas informações são utilizadas para gerar código de simulação. Este mapeamento, no entanto, considera apenas um conjunto reduzido de especificações em GGBO e, portanto, a geração de código de simulação através desse mapeamento é dependente do tipo de aplicação.

5.1.2 Modificações realizadas

Neste trabalho, o simulador foi estendido para a realização de simulações de SDCM. Para isso, foram criadas duas entidades especiais, representando as duas entidades envolvidas em um cenário de um SDCM: a entidade *Place*, que representa um lugar, e a entidade *MAgent*, que representa um componente móvel, conforme as definições apresentadas no Capítulo 3. Dessa forma, essas duas entidades apresentam os atributos definidos para lugares e componentes móveis e as regras que definem seus comportamentos básicos, apresentadas na Seção 3.1.

A partir das entidades criadas, criou-se um novo nível na hierarquia de entidades do simulador, conforme apresenta a Figura 34.

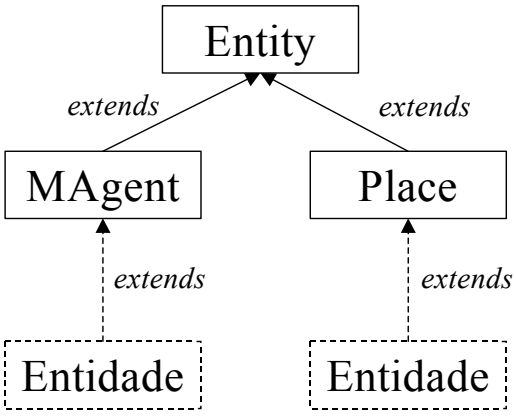

Figura 34. Hierarquia de entidades do simulador com a inclusão das novas entidades.

Segundo a hierarquia apresentada, *Entity* representa a entidade padrão do simulador, a qual é estendida por *Place* e *MAgent*. Ao construir sua aplicação, agora o usuário cria suas entidades (representadas por Entidade) estendendo *Place* ou *MAgent*. Quando uma entidade da aplicação estende *Place* ou *MAgent*, ela incorpora o comportamento básico de uma entidade, como a capacidade de envio, recebimento e tratamento de mensagens, e o comportamento básico de um lugar ou um componente móvel. Dessa forma, a entidade da aplicação é descrita apenas com as características pertinentes ao âmbito da aplicação a qual pertence.

Na etapa de simulação, entidades da aplicação poderiam ser criadas diretamente baseadas em *Entity*, as quais possuíssem seu comportamento descrito pelas regras definidas para lugares e componentes móveis (vide Seção 3.1). Assim, não seria necessário ter-se mais um nível na hierarquia para que fosse possível simularem-se aplicações envolvendo mobilidade. O uso do nível em que estão *Place* e *MAgent* é necessário quando os mecanismos de movimentação da plataforma são utilizados. Com isso, o uso de tais mecanismos é codificado dentro destas entidades especiais, sem que seja preciso fazer-se o mesmo para cada entidade que representasse um lugar ou um componente móvel em uma aplicação. Esta hierarquia com três níveis é apresentada e utilizada desde a etapa de simulação para tornar mais simples o entendimento nas etapas seguintes.

As próximas etapas dizem respeito à geração de código executável, baseando-se no código gerado para simulação. Deve-se dizer que é assumida, neste trabalho, a correção do mapeamento de uma especificação em GGBO para código do simulador, o que ainda não está provado formalmente, mas é indicado pela realização de testes. Partindo-se desse pressuposto, argumenta-se aqui que o mapeamento criado para transformar o código de simulação em código executável foi realizado de forma a manter a semântica do simulador.

A Figura 35 apresenta o código gerado para a entidade *MC* da aplicação de teste, que pode ser vista por completo na Seção 3.1.6 e cujo grafo de tipos foi reapresentado na Figura 28. O código apresenta os atributos internos da entidade (linhas 3 a 7), o método construtor da classe que representa a entidade (linha 8), passando uma referência à instância do *kernel* de simulação a ser utilizada, o método de inicialização da entidade *MC* (linhas 9 a 17) e o seu método de registro de regras (linhas 18 a 26).

```
1.  public class MC extends MAgent
2.  {
3.    public int numPlaces;
4.    public String prodName;
5.    public float bestPrice;
6.    public ShoppingPlace bestLoc, origin;
7.    public Customer respCustomer;

8.    public MC (Kernel k) {super(k);}

9.    public init (int numPlaces, String prodName, ShoppingPlace origin,
10.             Customer respCustomer) {
11.     this.numPlaces = numPlaces;
12.     this.prodName = prodName;
13.     this.origin = origin;
14.     this.bestLoc = origin;
15.     this.location = origin;
16.     this.respCustomer = respCustomer;
17.     this.bestPrice = 10000000000; }

18.   public void registerRules() {
19.     super.registerRules();
20.     addRule(new MC_R1 ("GetLocalIS", getKernel(), this));
21.     addRule(new MC_R2 ("QueryPrice", getKernel(), this));
22.     addRule(new MC_R3 ("Proceed", getKernel(), this));
23.     addRule(new MC_R4 ("Update_Proceed", getKernel(), this));
24.     addRule(new MC_R5 ("MoveNext", getKernel(), this));
25.     addRule(new MC_R6 ("MoveBack", getKernel(), this));
26.      addRule(new MC_R7 ("ReturnResult", getKernel(), this));}
27. }
```

Figura 35. Exemplo de código de simulação de entidade.

O código da regra cuja especificação foi apresentada na Figura 29 é mostrado na Figura 36.

```
1.  public class MC_R1 extends Rule
2.  {
3.    public MC_R1 (String strId, Kernel k, Entity e) {super(strId, k, e);}

4.    public boolean answerMsg (Message m) {
5.      if (m.getId().equals("Continue")) return true;
6.      return(false); }

7.    public boolean conditionOk (Message m) {
8.      MC mc = (MC) owner();
9.      if (mc.numPlaces != 0) return true;
10.     return false; }

11.   public boolean leftSideOccurs (Message m) {return true;}

12.   public void apply (Message m) {
13.     MC mc = (MC) owner();
14.     Parameters par = new Parameters();
15.     par.add("comp", mc);
16.     mc.sendMessage(10.0, new Message(mc, mc.getLocation(), "GetIS", par));}

17.   public boolean writeOnAttrib () {return false;}
18. }
```

Figura 36. Exemplo de código de simulação de regra.

O método *answerMsg* (linhas 4 a 6) define que esta regra trata a mensagem *Continue*. O método *conditionOk* (linhas 7 a 10) testa se a condição de execução da regra (o número de lugares a visitar deve ser igual a 0) é satisfeita. O método *apply* (linhas 12 a 16) apresenta o resultado da aplicação da regra , que é a geração da mensagem *GetIS*, destinada ao lugar onde o *MC* se encontra, passando uma referência ao componente por parâmetro. Já o método *writeOnAttrib* (linha 17) determina que esta é uma regra de leitura. O método *leftSideOccurs*, nesta regra, não possui nenhuma restrição a ser testada e, portanto, apenas retorna o valor verdadeiro para o teste de condição.

A movimentação de uma entidade, na simulação, é representada pela alteração do valor de seu atributo de localização, significando que cada novo valor representa uma nova movimentação. Isto é modelado com as regras apresentadas no Seção 3.1 para lugares e componentes móveis.

Nas seções seguintes, serão apresentadas as alterações realizadas para passar de código de simulação para código executável e de que forma argumenta-se a manutenção da semântica do código para simulação em cada etapa.

5.2 Geração de Código para Execução Local

Como primeira etapa da geração de código para execução, realizou-se a retirada do *kernel* do simulador. Como visto no Capítulo 4, o *kernel* é a entidade que controla o tempo de simulação e a comunicação entre as entidades. A saída desta entidade centralizadora determinou que a troca de mensagens passasse a ser direta entre as entidades, sem a sua intervenção. Outra característica que fica alterada é que o relógio simulado deixou de existir e passou-se a ter a execução regida pelo tempo real. Com isso, passou-se a desconsiderar o *timestamp* colocado nas mensagens, utilizado na simulação para determinar quais mensagens devem ser encaminhadas a seus destinos em cada instante do tempo simulado. O *timestamp* da mensagem serve para representar a demora da mensagem para chegar na entidade de destino. Com a introdução do tempo real, a semântica do código de simulação é preservada, visto que o tempo utilizado nas mensagens passou a ser definido pelo tempo real gasto na transmissão da mensagem até o seu destino.

Para tornar possível a geração de código tanto para a simulação quanto para a execução local, realizaram-se alterações na estrutura das entidades e das regras. Assim, o código mapeado permitia realizar-se uma simulação, intermediada pelo *kernel*, ou a

execução local, com a comunicação direta entre as entidades, sem interferência do *kernel*. Ou seja, nesta etapa, uma simples alteração possibilitava utilizar o mesmo código da simulação para realizar-se uma execução local. A escolha era definida pela existência ou não de um *kernel*. A Figura 37 apresenta como ficou o código da entidade *MC* para uma execução local.

A principal mudança que pode ser notada é no método construtor (linha 8), que não possui mais uma referência a uma instância de *kernel*, já que a comunicação entre entidades agora é direta. Também devido à saída do *kernel*, a outra mudança ocorre na inicialização das regras da entidade (linhas 19 a 25), onde também não é mais passada uma referência a um *kernel*.

```
1.  public class MC extends MAgent
2.  {
3.    public int numPlaces;
4.    public String prodName;
5.    public float bestPrice;
6.    public ShoppingPlace bestLoc, origin;
7.    public Costumer respCostumer;

8.    public MC () {super(null);}

9.    public init (int numPlaces, String prodName, ShoppingPlace origin,
10.                Customer respCustomer) {
11.     this.numPlaces = numPlaces;
12.     this.prodName = prodName;
13.     this.origin = origin;
14.     this.bestLoc = origin;
15.     this.respCustomer = respCustomer;
16.     this.bestPrice = 10000000000; }

17.   public void registerRules() {
18.     super.registerRules();
19.     addRule(new MC_R1 ("GetLocalIS", this));
20.     addRule(new MC_R2 ("QueryPrice", this));
21.     addRule(new MC_R3 ("Proceed", this));
22.     addRule(new MC_R4 ("Update_Proceed", this));
23.     addRule(new MC_R5 ("MoveNext", this));
24.     addRule(new MC_R6 ("MoveBack", this));
25.     addRule(new MC_R7 ("ReturnResult", this)); }
26. }
```

Figura 37. Exemplo de código de execução local de entidade.

Quanto às regras, as mudanças podem ser visualizadas no código da regra *MC_R1* na Figura 38.

```
1.  public class MC_R1 extends Rule
2.  {
3.    public MC_R1 (String strId, Entity e) {super(strId, e);}

4.    public boolean answerMsg (Message m) {
5.      if (m.getId().equals("Continue")) return true;
6.      return(false);}

7.    public boolean conditionOk (Message m) {
8.      MC mc = (MC) owner();
9.      if (mc.numPlaces != 0) return true;
10.     return false;}

11.   public boolean leftSideOccurs (Message m) {return true;}

12.   public void apply (Message m) {
13.     MC mc = (MC) owner();
14.     Parameters par = new Parameters();
15.     par.add("comp", mc);
16.     mc.getLocation().sendMessage(10.0, new Message(mc, "GetIS", par)); }
17. }
```

Figura 38. Exemplo de código de execução local de regra.

Como acontece com o código da entidade, tudo o que envolvia o uso do *kernel* no código da regra é modificado. Assim, o método de inicialização (linha 3) não recebe mais a referência ao *kernel* de simulação. A outra modificação importante é no envio das mensagens geradas pela aplicação da regra. As mensagens, antes enviadas para o *kernel* (como na linha 24 da Figura 36), agora são enviadas diretamente à entidade de destino através da invocação de seu método *sendMessage* (linha 16 da Figura 38). Os parâmetros de uma mensagem também foram modificados. Não é mais passada como parâmetro a identificação da entidade de destino da mensagem. Isto porque, visto que a mensagem é enviada diretamente à entidade, essa informação, utilizada pelo *kernel* para encaminhar a mensagem, não é mais necessária.

Como visto, as modificações do código de simulação para o código de execução local foram mínimas e o código de execução local obedece a um padrão que permite a automatização de sua geração, assim como para o código de simulação.

5.3 Geração de Código para Execução Distribuída

A próxima etapa compreendeu permitir que as entidades, que executavam todas no mesmo lugar, passassem a poder executar de forma distribuída. Esta transformação exigiu a utilização de suporte à distribuição, o qual foi fornecido por uma PSM. A plataforma escolhida foi a Voyager, apresentada na Seção 2.1.3. Com a introdução do uso da

plataforma, a comunicação entre entidades passou a ser feita através de recursos fornecidos pelo suporte utilizado, tal como chamadas remotas de métodos.

A escolha desta plataforma se deu em razão de que ela suporta um ambiente de acordo com as características relacionadas na Seção 3.1, fornecendo integridade referencial (se uma entidade se move, outras entidades que a referenciavam continuam com referências válidas) e comunicação remota e local confiáveis, dentro do cenário assumido de inexistência de falhas. Com isso, viabilizou-se a preservação da semântica do código de simulação, fornecendo-se o mesmo ambiente suportado no simulador, no qual oferece-se comunicação confiável e integridade de referências. A Figura 39 mostra o código gerado para a entidade *MC* para execução distribuída.

```
1.  public class MC extends MAgent implements IMC
2.  {
3.    protected int numPlaces;
4.    protected String prodName;
5.    protected float bestPrice;
6.    protected IShoppingPlace bestLoc, origin;
7.    protected ICustomer respCustomer;

8.    public MC (){super(null);}

9.    public void init (int numPlaces, String prodName, ICustomer respCustomer,
10.                     IShoppingPlace origin) {
11.     this.numPlaces = numPlaces;
12.     this.prodName = prodName;
13.     this.origin = origin;
14.     this.bestLoc = origin;
15.     this.respCustomer = respCustomer;
16.     this.bestPrice = 10000000000; }

17.   public void registerRules() {
18.     super.registerRules();
19.     addRule(new MC_R1 ("GetLocalIS", this));
20.     addRule(new MC_R2 ("QueryPrice", this));
21.     addRule(new MC_R3 ("Proceed", this));
22.     addRule(new MC_R4 ("Update_Proceed", this));
23.     addRule(new MC_R5 ("MoveNext", this));
24.     addRule(new MC_R6 ("MoveBack", this));
25.     addRule(new MC_R7 ("ReturnResult", this)); }
26. }
```

Figura 39. Exemplo de código para execução distribuída de entidade.

Para utilizar os recursos da plataforma, cada entidade deve possuir uma interface Java, onde são definidos os métodos públicos da entidade. Ou seja, na interface da entidade definem-se quais métodos podem ser invocados remotamente. Por isso, *MC* passou, nesta etapa, a implementar a interface *IMC* (linha 1). *IMC* estende a interface *IMAgent*, que é implementada por *MAgent*, que, por sua vez, estende a interface *IEntity*, implementada por

Entity. Dessa forma, da mesma maneira que se tem uma hierarquia de entidades, tem-se também uma hierarquia das interfaces implementadas por estas entidades.

Como a comunicação entre entidades passa a ser suportada pela plataforma, uma referência a uma entidade é uma referência a uma *proxy* da entidade, conforme definida na Seção 2.1.3.2. Assim atributos que referenciam outras entidades passam a ser do tipo da interface que estas entidades implementam. Dessa forma, o atributo *respCustomer*, que era do tipo *Customer*, passou a ser do tipo *ICustomer* (linha 7), que representa a interface da entidade *Customer*. Logo, uma invocação de um método de *Customer* é feita através da sua *proxy* local, a qual possui a mesma interface que *Customer*.

A mesma mudança nos tipos dos atributos é visualizada no código das regras, como pode ser visto no código de *MC_R1* na Figura 40.

```
1.  public class MC_R1 extends Rule
2.  {
3.    public MC_R1 (String strId, Entity e) {super(strId, e);}

4.    public boolean answerMsg (Message m){
5.      if (m.getId().equals("Continue")) return true;
6.      return(false); }

7.    public boolean conditionOk (Message m) {
8.      MC mc = (MC) owner();
9.      if (mc.getNumPlaces() != 0) return true;
10.     return false; }

11.   public boolean leftSideOccurs (Message m) {return true;}

12.   public void apply (Message m) {
13.     MC mc = (MC) owner();
14.     Parameters par = new Parameters();
15.     IMC im = (IMC) mc.getReference(mc);
16.     par.add("comp", im);
17.     mc.getLocation.sendMessage(10.0, new Message(im, "GetIS", par)); }

18.   public boolean writeOnAttrib () {return false;}
19. }
```

Figura 40. Exemplo de código para execução distribuída de regra.

Além da já citada modificação dos tipos dos atributos, outra modificação é que, visto que a comunicação ocorre através da interface das entidades, uma entidade precisa obter uma referência remota sua (*proxy*) para passar como parâmetro de uma mensagem. Isto é obtido através do método *getReference* (linha 15), implementado nas entidades *Place* e *MAgent*, o qual utiliza mecanismos da plataforma para obter uma *proxy* da entidade.

Para exemplificar o código gerado para mapear um grafo inicial de um sistema para execução distribuída (isto é, considerando os comandos da plataforma), a Figura 41 apresenta o código gerado para o grafo inicial apresentado anteriormente na Figura 20.

```
1.  class Shopping {
2.   public static void main(String args[]) {
3.    try {
4.     Voyager.startup("//localhost:8000");

5.     IMC mc = (IMC) Factory.create("distributed.MC", //localhost:8000");
6.     Icostumer c = (ICostumer) Factory.create("distributed.Costumer",
       "//localhost:8000");
7.     IIS is_orig = (IIS) Factory.create("distributed.IS", "//localhost:8000");
8.     IIS is_1 = (IIS) Factory.create("distributed.IS", "//localhost:6000");
9.     IIS is_2 = (IIS) Factory.create("distributed.IS", "//localhost:8000");
10.    IIS is_3 = (IIS) Factory.create("distributed.IS", "//localhost:6000");
11.    IshoppingPlace p_orig = (IShoppingPlace)
       Factory.create("distributed.ShoppingPlace", "//localhost:8000");
12.    IshoppingPlace p_1 = (IShoppingPlace)
       Factory.create("distributed.ShoppingPlace", "//localhost:6000");
13.    IshoppingPlace p_2 = (IShoppingPlace)
       Factory.create("distributed.ShoppingPlace", "//localhost:8000");
14.    IshoppingPlace p_3 = (IShoppingPlace)
       Factory.create("distributed.ShoppingPlace", "//localhost:6000");

15.    mc.init (3, "product1", 1000000, p_orig, c, p_orig, p_orig);
16.    c.init(0, p_orig, p_orig);
17.    is_orig.init(0, " ");
18.    is_1.init(20, "product1");
19.    is_2.init(10, "product1");
20.    is_3.init(30, "product1");
21.    p_orig.init(is_orig, p_1);
22.    p_1.init(is_1, p_2);
23.    p_2.init(is_2, p_3);
24.    p_3.init(is_3,p_orig);

25.    Parameters par = new Parameters ();
26.    par.add("newLoc", p_1);
27.    mc.send(mc, 0.0, new Message (null, "Next", par));
28.   }
29.   catch (Exception e) {System.err.println(e.getMessage());}
30.  }
31. }
```

Figura 41. Exemplo de código gerado para inicialização sobre a plataforma.

A linha 4 apresenta a inicialização da instância local da plataforma na porta 8000. Entre as linhas 5 e 14 são feitas as inicializações das entidades do sistema. Cada entidade é criada através do método *Factory.create* da plataforma, informado-se o local (endereço IP e porta) onde esta deve ser instanciada. O retorno deste método é uma referência remota à entidade criada. As linhas 15 a 24 apresentam os métodos de inicialização dos atributos de cada entidade. Entre as linhas 25 e 27 encontra-se o código que descreve o envio da mensagem inicial para o grafo inicial representado (mensagem *Next* para a entidade *MC*).

Assim como na etapa anterior, as poucas diferenças entre o código de simulação para o código de execução distribuída permitem que se considere possível gerar-se automaticamente o código para executar sobre a plataforma.

5.4 Geração de Código para Execução com Mobilidade

Tendo-se entidades distribuídas, pôde-se passar a implementar a idéia de que tais entidades pudessem se movimentar. A capacidade de movimentação é suportada pela plataforma utilizada, provendo mecanismos tais como serialização de componentes e resolução de referências relativas a componentes móveis, como visto na Seção 2.1.3.4.

A fim de possibilitar a movimentação das entidades que estendem *MAgent*, foram feitas alterações na classe *Entity*. *Entity* não executa mais como uma *thread* (isto é, não estende mais a classe *Thread*), como descrito no Capítulo 4, mas passa a possuir uma *thread* interna, ligada à entidade como um atributo transiente, a qual recebe as mensagens enviadas à entidade. Isso foi necessário porque, sendo uma *thread*, não seria possível serializá-la e movê-la. Como um atributo transiente, a *thread* é finalizada na origem e novamente instanciada quando a entidade chega ao destino. Feita esta alteração, torna-se possível retomar o estado de execução da entidade após a sua movimentação. A movimentação de uma entidade compreende os seguintes passos:

1. Parar a entidade no lugar de origem;
2. A) Salvar o estado interno da entidade;

   B) Salvar o estado de execução da entidade;
3. Transferir a entidade para o lugar de destino;
4. Resolver referências à entidade movida;
5. Reinicializar a entidade.

A transferência da entidade (passo 3) é feita através da invocação do mecanismo de mobilidade da plataforma, a qual fica responsável por garantir a transferência da entidade de um lugar a outro. Este mecanismo é invocado a partir da aplicação da regra *Move* (apresentada na Figura 13 da Seção 3.1.3), o que ocorre quando um *Place* recebe uma mensagem de *MoveReq*. A Figura 42 destaca o trecho de código da regra *Move* onde é iniciado o processo de movimentação.

```
1.  IMAgent ma = (IMAgent) par.get("comp");
2.  IPlace orig = (IPlace) par.get("orig");
3.  IPlace dest = (IPlace) p.getReference(p);
4.  boolean r;
5.  if (p.attendsRequirements(orig, ma))
6.  {
7.   if (ma.go(dest))
8.    r = true;
9.   else r = false;
10. }
11. else r = false;
```

Figura 42. Trecho de código da regra que inicia o processo de movimentação da entidade.

O comando da linha 5 representa o teste que pode ser utilizado para determinar se o lugar atende alguns requisitos para a movimentação, baseando-se no lugar de origem e no componente a ser movido. Como tais requisitos serão definidos é um tema para estudos futuros. Na linha 7 é dado o comando que informa ao *MAgent* que ele deve se mover. O método *go* recebe, como parâmetro, uma referência ao lugar de destino da movimentação. A Figura 43 apresenta o código do método *go*.

```
1.  public boolean go (IPlace place)
2.  {
3.   try
4.   {
5.    stopEntity();
6.    Agent.of(this).moveTo(place, "atPlace");
7.   }
8.   catch(Exception exception) {return false;}
9.   return true;
10. }
```

Figura 43. Código do método que implementa a movimentação da entidade.

O método *go* causa a parada da entidade através do método *stopEntity* (linha 5) e invoca o serviço de movimentação da plataforma de suporte (linha 6). Portanto, o comando da linha 6 fornece a implementação real do passo 3 na movimentação da entidade. O primeiro parâmetro indica o lugar de destino da movimentação, enquanto o segundo determina o método a ser executado logo que a entidade esteja relocada. A indicação deste método serve como um ponto de reinicialização da execução da entidade. Ao final da movimentação, caso a movimentação tenha sido bem sucedida, o método indicado como parâmetro do comando de movimentação (linha 6 da Figura 43) é executado pelo *MAgent*. Este método é apresentado na Figura 44. O método *initialize* (linha 3) define a implementação da reinicialização da entidade.

```
1.  public void atPlace (IPlace place)
2.  {
3.   this.initialize(place);
4.  }
```

Figura 44. Código do método de reinicialização da entidade.

Com a resposta positiva do lugar de destino (recebida através da mensagem *MvAnswer*), o lugar de origem assume que o componente móvel foi relocado e envia as mensagens definidas na regra *AcceptMove* (apresentada na Figura 14), informando ao lugar de destino para incluir o novo componente e, ao *MAgent*, para atualizar o seu atributo de localização.

O passo 2.A, que refere-se ao salvamento dos valores dos atributos internos da entidade, também é suportado pela plataforma, a qual serializa os atributos da entidade na origem e os desserializa no destino. Dados da entidade que não possuem a capacidade de serem transmitidos através da rede (não são serializáveis) são definidos como transientes e são novamente inicializados ao término da movimentação. O passo 4 é implementado pela plataforma através do mecanismo de resolução de referências apresentado na Seção 2.1.3.4. Os passos 1, 2.B e 5 não são suportados pela plataforma, sendo implementados neste trabalho. O passo 1 consiste em fazer cessar qualquer atividade interna da entidade. Isto implica parar a *thread* interna da entidade, responsável pelo recebimento de mensagens, e garantir que não há regras em execução. Assim, quando se inicia um processo de movimentação, a *thread* interna é finalizada, impedindo que novas mensagens sejam processadas. Finalizada a *thread*, antes da transferência da entidade, deve-se esperar que todas as regras em execução sejam finalizadas. Tão logo isso ocorra, a entidade pode então ser movida. A codificação desse passo foi incluída no código da entidade no método *stopEntity*, apresentado na Figura 45. Este método é invocado dentro do método que implementa a movimentação real da entidade, apresentado na Figura 43.

```
1.  protected void stopEntity () {
2.    setLive(false);
3.    mbuf.finishWait();
4.    try {thread.join();} catch (Exception e)
5.    {System.err.println(e.getMessage());}

6.    if (this.activeThreads.size() > 0) {
7.      Enumeration e = activeThreads.elements();
8.      while (e.hasMoreElements()) {
9.        Thread t = (Thread) e.nextElement();
10.       try {t.join();} catch (Exception exception)
11.       {System.err.println(exception.getMessage());} } }
12.  }
```

Figura 45. Código do método de paralisação da entidade.

A primeira providência (linha 2) é modificar o *status* da entidade, definindo-a como inativa, o que impede que ela receba novas mensagens enquanto estiver se movimentando. Tais mensagens entrarão no *buffer* da entidade quando esta estiver no local de destino da movimentação. Após isso, é forçado o término do trabalho da *thread* interna. O que ocorre é que a *thread* interna pode estar bloqueada, à espera de um aviso do *buffer* de que existe uma nova mensagem a ser tratada. O comando da linha 3 força o desbloqueio da *thread* interna, caso esteja bloqueada esperando novas mensagens chegarem ao *buffer*. Quando a *thread* recebe o sinal de desbloqueio, ela testa se existe uma mensagem nova a ser tratada e se a entidade ainda está recebendo mensagens (entidade está ativa). Como a entidade não está mais ativa, a *thread* é finalizada. A informação da finalização da *thread* interna é obtida com o código presente nas linhas 4 a 5. O último passo é esperar que as regras em execução sejam finalizadas, o que é feito através do código descrito entre as linhas 6 e 11.

A reinicialização da entidade (passo 5), realizada após uma movimentação, compreende a reinicialização da *thread* interna da entidade, restabelecendo a sua capacidade de receber mensagens, e permitir que regras possam passar a executar, a fim de tratar mensagens pendentes e futuras mensagens. O código de reinicialização da entidade é apresentado na Figura 46. O método *initialize* também está incluído no código da entidade e é invocado sempre que uma movimentação é bem sucedida, dentro do método *atPlace*, apresentado na Figura 44.

```
1. public void initialize ()
2. {
3.    setLive(true);
4.    thread = new InternalThread (this);
5.    thread.start();
6. }
```

Figura 46. Código de reinicialização da entidade.

Os passos seguidos para reinicializar a entidade são modificar seu atributo de *status* para definir a entidade como ativa (linha 3), permitindo que ela volte a receber mensagens, e reinicializar a *thread* interna (linhas 4 e 5).

O passo 2.B, neste trabalho, diz respeito a garantir que todas as mensagens enviadas a uma entidade são recebidas e processadas por ela, mesmo que ela se mova. Isto é garantido pelos mecanismos de parada e restauração das atividades da entidade que foram

implementados (apresentados na Figura 45 e na Figura 46, respectivamente), em conjunto com um mecanismo de comunicação confiável, implementado para garantir que mensagens enviadas durante a movimentação da entidade sejam recebidas por ela. Tal mecanismo deveria ser fornecido pela plataforma, mas, durante os testes realizados, ficou demonstrado que havia perdas de mensagens, o que exigiu a criação deste mecanismo dentro do presente trabalho.

Conforme a implementação realizada, a regra de movimentação acontece isoladamente (ou seja, sem a concorrência com o processamento de outras regras) e as referências à entidade são mantidas válidas pela plataforma utilizada. Ao final da movimentação, a entidade passa a executar no novo local, a partir do mesmo estado em que se encontrava no local anterior, processando a próxima mensagem do *buffer*. Dessa forma, mantém-se a semântica esperada na especificação.

Os códigos gerados para entidades e regras permanecem os mesmos que na etapa anterior (execução distribuída). As mudanças realizadas restringiram-se às alterações na classe *Entity*, já citadas, e na implementação das regras de movimentação, onde a movimentação física da entidade é implementada através do uso dos recursos da plataforma em meio à execução das regras. Com isso, a movimentação de uma entidade é representada por sua movimentação física, realizada através dos passos apresentados anteriormente, e pela alteração de seu atributo de localização.

Como resultado das etapas seguidas, chegou-se à geração de código para execução com mobilidade que, segundo discutido, mantém a semântica do código mapeado para o simulador PLATUS e que permite, segundo uma análise inicial, que tal código seja gerado automaticamente. A geração automática de código é uma etapa considerada como um trabalho futuro. O que se quer é que seja possível criar-se uma especificação em GGBO em um editor e, a partir dessa especificação, gerar código para simulação no PLATUS e/ou para execução sobre a plataforma Voyager. Uma ressalva que deve ser feita é que também se cogita a possibilidade de realizar-se a geração de código para outras plataformas de suporte à mobilidade. Considera-se que a adaptação do mapeamento aqui proposto para uma outra plataforma exigiria apenas a modificação na regra de lugar que ocasiona a movimentação do componente e na implementação do método que movimenta o componente, codificado na entidade *MAgent*.

6 Estudo de Caso 1 – Redes Ativas

Como forma de avaliar o trabalho desenvolvido, foi realizado um estudo de caso que envolvia um cenário mais complexo do que os testes desenvolvidos anteriormente. O estudo de caso, a seguir descrito, envolveu conceitos de *redes ativas* [PSO99] e tinha como objetivo representar um cenário de redes ativas através de GGBO. Os conceitos de redes ativas serão apresentados na seção seguinte, sendo descrito como eles foram representados em GGBO.

Cabe salientar que, visto que este estudo de caso visava avaliar o trabalho desenvolvido, todos os elementos envolvidos são entidades que se baseiam em *Place* ou em *MAgent*. Quando é dito que uma entidade se baseia em outra se quer dizer que uma entidade incorpora o comportamento e os atributos da entidade na qual ela se baseia. O termo *herança* não pode se usado aqui uma vez que GGBO não apresenta uma forma de representação de hierarquia de herança entre entidades.

6.1 Redes Ativas

Tradicionalmente, a função de uma rede é encaminhar pacotes de um nodo a outro. O processamento dentro da rede ocorre com a realização do roteamento de pacotes, a realização de algum tipo de controle de congestionamento e a implementação de esquemas de qualidade de serviço (QoS). Sendo assim, as redes tradicionais não possuem a capacidade de processar dados submetidos pelo usuário. Isto é, todo o processamento ocorre de forma estática, através de instruções predefinidas na configuração dos dispositivos da rede. Devido a esta característica de funcionalidade restrita de processamento, redes deste tipo são denominadas de *redes passivas*. Neste tipo de rede, torna-se difícil a integração de novas tecnologias, visto que isto normalmente exige a troca de equipamentos ou a reconfiguração destes pela administração da rede. Também fica dificultada a introdução de novos serviços sem modificações no modelo arquitetural já existente para a rede. A fim de superar tais dificuldades e outras encontradas em redes convencionais, surgiu a idéia das chamadas *redes ativas* [PSO99].

Redes ativas representam uma nova abordagem para a arquitetura de redes, sendo que elas são denominadas como *ativas* porque os roteadores podem, assim como os outros nodos da rede, realizar computações sobre os dados de usuário que passam por eles a partir

de programas fornecidos pelo próprio usuário. Assim, os usuários podem, de certa forma, "programar" a rede. Em redes ativas é permitido, portanto, que programas sejam dinamicamente inseridos nos nodos da rede, a fim de configurá-los de acordo com as necessidades das aplicações que estão executando. Para isso, existe a possibilidade de pacotes carregarem não apenas dados, mas também programas a serem executados nos nodos por que passarem. Em uma rede ativa, torna-se muito tênue a diferença entre nodos internos da rede (roteadores, *switches*, etc.) e nodos de usuário, uma vez que ambos possuem a capacidade de realizar as mesmas computações. Assim, o usuário pode, de certa maneira, considerar a rede como parte do seu próprio sistema, podendo adaptá-la para obter maior eficiência de suas aplicações.

Existem três abordagens para arquiteturas de redes ativas:

- *Pacotes ativos*: Nodos não armazenam código. O código é transportado pelos pacotes ativos (também chamados de *cápsulas* em algumas arquiteturas) que trafegam na rede. Os nodos podem realizar computações com os códigos trazidos pelos pacotes. Alguns exemplos desta abordagem são o *Smart Packets* [SCH98] e o *Active IP Option* [WET96];
- *Nodos ativos*: Pacotes não carregam o código real, mas apenas alguns identificadores ou referências a funções predefinidas existentes nos nodos ativos. Assim, os pacotes contêm a definição do tipo da função que será executada sobre os seus dados e fornecerem os parâmetros para a execução desta função. O código real encontra-se nos nodos ativos, os quais executam a função necessária para tratar o pacote recebido. Exemplos desta abordagem são as arquiteturas *DAN* [DEC98] e *ANTS* [WET98];
- *Pacotes e nodos ativos*: Esta abordagem é uma combinação das duas anteriores, onde os pacotes ativos carregam o código real e os nodos ativos possuem códigos mais complexos. Assim, o usuário tem a possibilidade de utilizar tanto a abordagem de pacotes ativos como a de nodos ativos, além de poder utilizá-las em conjunto. Como exemplos, podem ser citadas as arquiteturas *SwitchWare* [GUN98] e *NetScript* [YEM96].

Neste trabalho, foi considerada, para a realização do estudo de caso, uma arquitetura baseada em redes ativas onde os seus elementos são nodos ativos, cápsulas, serviços, bases de código e servidores de nomes. Esta arquitetura baseia-se na abordagem de nodos ativos.

Os *nodos ativos* possuem a capacidade de receber e enviar cápsulas de e para outros nodos. Cada *cápsula* carrega dados e possui a informação de qual serviço é necessário para tratá-la. Um *serviço* é um componente móvel que possui o código que define como tratar ou interagir com um tipo de cápsula, determinando o que fazer com os dados que ela carrega. Assim, cada nodo possui uma lista de serviços disponíveis, determinando os tipos de cápsulas que ele pode tratar. Nodos ativos podem requisitar novos serviços de uma base de código. Uma *base de código* representa um componente que possui a lista de serviços disponíveis para a rede em que se encontra. A base de código provê o envio de uma instância do serviço para os nodos ativos requisitantes.

A comunicação entre todos os elementos se faz através de referências remotas. Para obter uma referência remota a um elemento, o elemento requisitante precisa possuir o nome que identifica o elemento procurado e enviar uma requisição de resolução de nome para um servidor de nomes. O *servidor de nomes* é um elemento que possui uma lista de índices, onde cada índice possui uma referência associada a um nome de identificação de uma entidade. Além de pedidos de resolução de nomes, o servidor de nomes também fornece os serviços de registro, remoção e alteração de referências. O servidor de nomes é necessário uma vez que, em GGBO, não é possível ter-se uma lista de referências a entidades. Por isso, cada índice do servidor de nomes é uma entidade que possui atributos para armazenar a referência a uma entidade e o nome associado àquela entidade. Cada índice possui uma referência ao próximo índice da lista, o que permite o caminhamento dentro da lista para buscar uma referência. Mais detalhes sobre o servidor de nomes podem ser conhecidos em [ROD01].

6.2 Especificação em GGBO de Redes Ativas

O funcionamento padrão da arquitetura definida neste trabalho foi especificado em GGBO, determinando os atributos das entidades envolvidas e o comportamento básico de cada uma delas. Tais entidades são agora descritas através de seus grafos de tipos e suas regras.

6.2.1 Especificação de um Nodo Ativo

Para representar um nodo ativo, foi criada a entidade *ActiveNode* (AN). Esta entidade baseia-se na entidade *Place*, podendo, portanto, entre outras coisas, receber componentes

móveis e oferecer serviços de movimentação de componentes. O grafo de tipos para a entidade *ActiveNode* é apresentado na Figura 47 e na Figura 48.

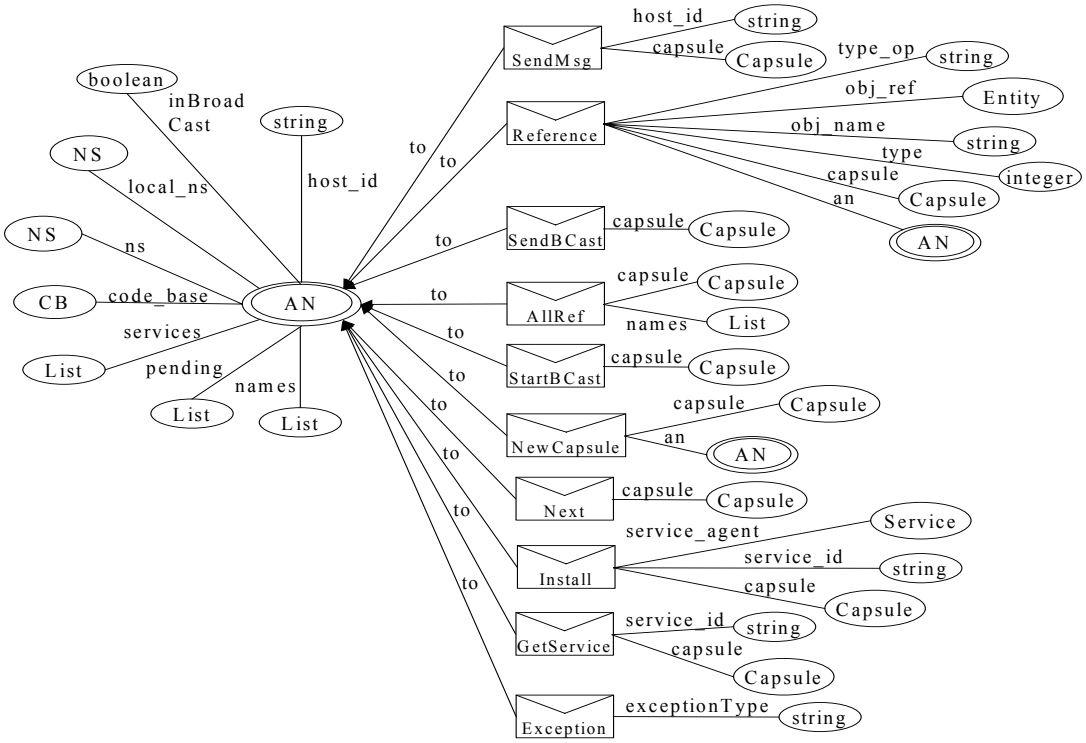

Figura 47. Grafo de tipos da entidade *ActiveNode* – parte 1.

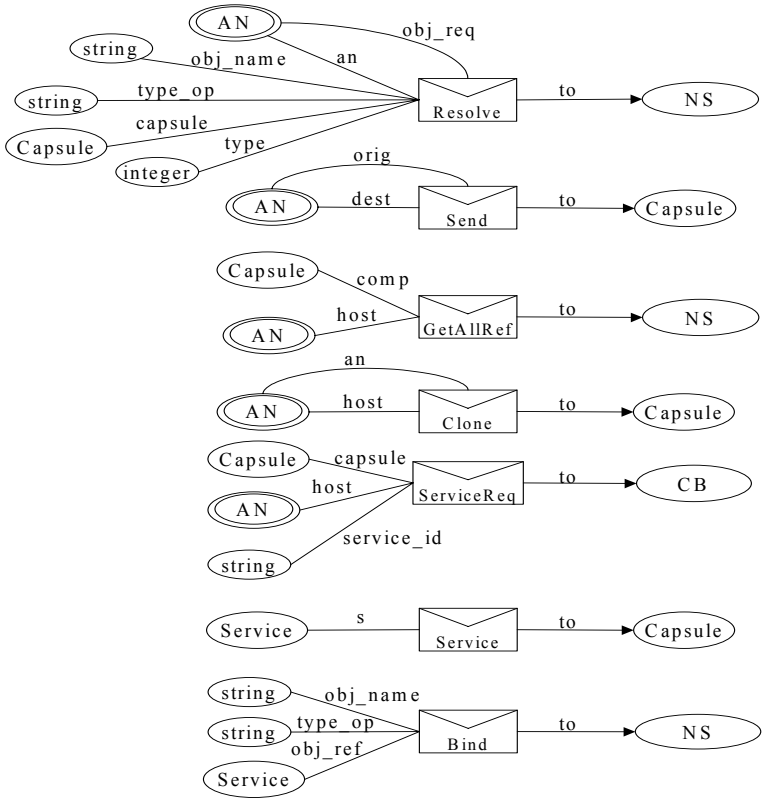

Figura 48. Grafo de tipos da entidade *ActiveNode* – parte 2.

A Figura 47 mostra a primeira parte do grafo de tipos, onde constam os atributos da entidade e as mensagens recebidas por ela. A Figura 48 mostra as mensagens enviadas pela entidade *ActiveNode*.

Um AN é identificado por um nome (atributo *host_id*), usado para a comunicação com outros nodos. Cada AN possui uma referência a uma base de código (atributo *code_base*) à qual pode requisitar instâncias de serviços de que necessite. As identificações dos serviços já disponíveis em um AN constam em sua lista de serviços (atributo *services*). Um AN possui também uma referência a um servidor de nomes local (atributo *local_ns*) usado para registrar referências aos serviços locais. Dessa forma, sempre que um serviço é requisitado por uma cápsula através de sua identificação, o AN verifica se o serviço consta em sua lista de serviços. Caso ele conste, o AN faz um pedido de resolução de nome para o servido de nomes local, a fim de obter uma referência ao serviço procurado. O servidor de nomes local também serve para armazenar as identificações dos nodos conhecidos pelo AN, com os quais ele pode se comunicar. Um AN contém ainda uma referência a um servidor de nomes global (atributo *ns*), o qual serve para registrar informações gerais sobre a rede, tais como as identificações de todos os nodos da rede e todos os serviços disponíveis. O atributo *inBroadCast* serve para controlar a execução de *broadcasts*, como será visto mais adiante.

O tipo *List* é um tipo definido neste trabalho e representa uma lista que possui as seguintes operações:

- *List add (item)*: Adiciona *item* ao fim da lista e retorna a lista resultante;
- *List addFirst (item)*: Adiciona *item* no início da lista e retorna a lista resultante;
- *List addListFirst (List)*: Adiciona todos os elementos da lista recebida por parâmetro ao início da lista original e retorna a lista resultante;
- *List addList (List)*: Adiciona todos os elementos da lista recebida por parâmetro ao fim da lista original e retorna a lista resultante;
- *List remove ()*: Remove o primeiro elemento da lista e retorna a lista resultante;
- *List remove (item)*: Remove *item* da lista e retorna a lista resultante;
- *item get ()*: Retorna o primeiro elemento da lista, retirando-o desta;
- *List invertList ()*: Inverte a ordem dos elementos da lista e retorna a lista resultante;

- *boolean isInList (item)*: Retorna verdadeiro se *item* está na lista e falso caso contrário;
- *boolean isEmpty ()*:Retorna verdadeiro a lista está vazia e falso caso contrário.

Um *item* é um valor cujo tipo pertence a um dos tipos básicos de Gramáticas de Grafos (*boolean*, *integer*, *real*, *string* ou *character*) ou um outro tipo definido pelo especificador, tal como *List*. Deve-se ressaltar que tipos abstratos de dados em GGBO devem ser especificados usando especificação algébrica. No âmbito deste trabalho, tipos abstratos serão, por uma questão de simplicidade, definidos de maneira informal. É assumida, dessa forma, a correção das operações realizadas sobre tipos abstratos de dados aqui definidos.

Um AN possui três funções: enviar cápsulas em modo *unicast* (de um AN para outro), enviar cápsulas em modo *broadcast* (de um AN para todos os AN da rede) e prover serviços para tratar as cápsulas recebidas. A função mais básica de um AN é o envio de cápsulas a outros AN, tal qual um nodo envia pacotes para outros nodos em uma rede convencional. Esta função é realizada através das regras *SendMessage* e *GetReference*, apresentadas na Figura 49 e na Figura 50, respectivamente.

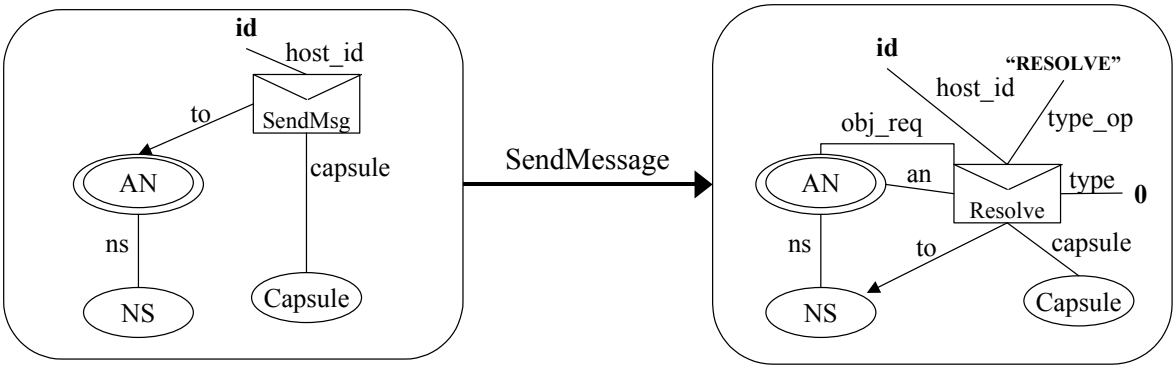

Figura 49. Regra *SendMessage* de um *ActiveNode*.

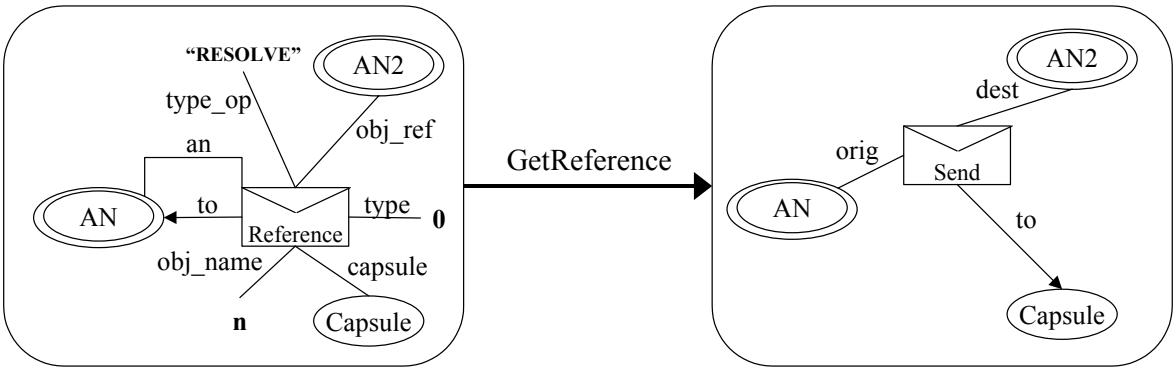

Figura 50. Regra *GetReference* de um *ActiveNode.*

A regra *SendMessage* (Figura 49) representa o recebimento, por parte do AN, de um pedido de encaminhamento de uma cápsula a um outro AN. O pedido é recebido através da mensagem *SendMsg*. Esta mensagem possui, como parâmetros, a identificação do AN de destino da cápsula e uma referência à cápsula que deve ser enviada. De posse da identificação do destino da cápsula, o AN faz uma requisição ao servidor de nomes local (representado pela entidade NS) para obter uma referência ao nodo de destino. Esta requisição é feita através de uma mensagem *Resolve*, que leva, como parâmetros básicos, o tipo da operação (parâmetro *type_op*), usado internamente pelo NS, uma referência à entidade requisitante (parâmetro *obj_req*) e a identificação do nodo cuja referência é requisitada (parâmetro *name*).

Como GGBO é inerentemente paralela, representar computações seqüenciais não é natural. Observe-se, por exemplo, a regra *SendMessage* na Figura 49. O parâmetro *type* é usado para que, ao obter a resposta a uma requisição, o AN possa ter a informação de como utilizar a referência recebida, podendo realizar uma comunicação *unicast* (valor 0), uma comunicação *broadcast* (valor 1) ou pode ainda encaminhar um serviço a outro nodo (valor 2). No caso da regra citada, o valor 0 do parâmetro *type* indica que a referência a ser obtida servirá para uma comunicação *unicast*. Com isso, o parâmetro *type* não possui nenhuma utilidade para o destinatário da mensagem, mas serve apenas para armazenar uma informação que será útil posteriormente.

O parâmetro *capsule* serve para que a referência à cápsula que deve ser enviada não seja perdida ao término da regra. De acordo com o formalismo utilizado, só há duas possibilidades de não se perder valores após uma aplicação de regra: repassando o valor como parâmetro de uma nova mensagem, de forma que ele retorne por outra mensagem para ser usado em uma regra posterior da entidade, ou armazenando o valor em um atributo da entidade. Na verdade, nenhuma das duas alternativas é muito agradável, haja vista que a primeira exige que se utilizem parâmetros de mensagens sem função alguma, enquanto que a segunda exige a criação de atributos que servem só como forma de armazenamento temporário de valores. Neste caso, está sendo usado um parâmetro de uma mensagem para manter a referência. O parâmetro *an* serve para a mesma finalidade, mas não é utilizado nesta regra, estando presente apenas por constar entre os parâmetros da mensagem *Resolve*.

A resposta à requisição feita ao NS é tratada pela regra *GetReference* (Figura 50). A mensagem *Reference* traz a referência requisitada, entre outros parâmetros, nos quais se

inclui a referência à cápsula que deve ser enviada. O AN, utilizando a referência ao nodo de destino obtida do NS, envia a mensagem *Send* à cápsula, passando a mesma referência ao nodo de destino como parâmetro. Isto informa a cápsula de que ela deve mover-se para o AN indicado.

A função de envio de cápsulas em modo *broadcast* é realizada através do envio de múltiplas cápsulas em modo *unicast*. A regra *SendBroadCast* (Figura 51) trata o recebimento de um pedido de envio de *broacast*. O pedido é feito através da mensagem *SendBCast*, que tem, como parâmetro, uma referência à cápsula que deve ser enviada. O envio em modo *broadcast* compreende obter a lista de AN para os quais a cápsula deve ser enviada e, para cada AN de destino, obter uma referência ao destino, criar uma cópia da cápsula e enviar a cópia. Por isso, a primeira ação do AN ao receber a mensagem *SendBCast* é requisitar ao NS a lista de todos os AN conhecidos. No caso de uma rede fixa, os AN conhecidos são aqueles com quem o AN tem ligação direta. Já para uma rede móvel, onde os nodos não possuem conexões físicas uns com os outros, os AN conhecidos representam todos os nodos alcançáveis pelos sinais enviados pelo AN de origem. A requisição dos AN conhecidos é feita através da mensagem *GetAllRef*, tendo uma referência à cápsula a ser enviada e uma referência ao AN requisitante como parâmetros.

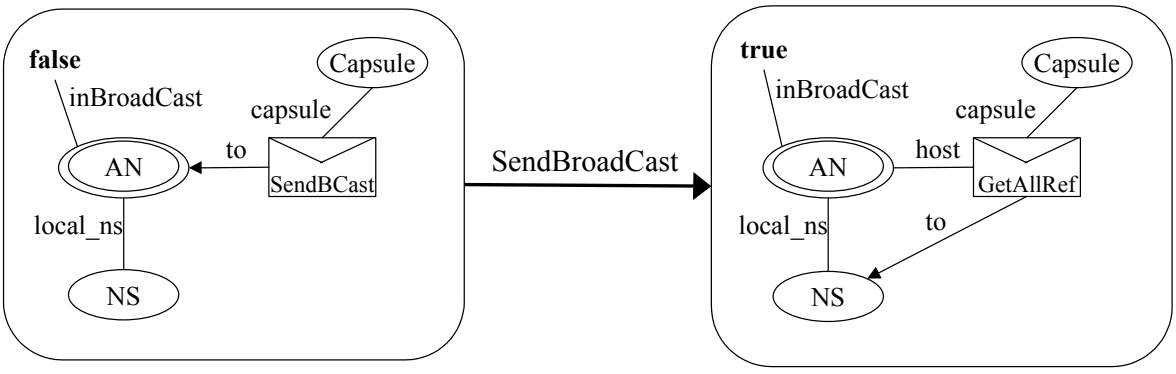

Figura 51. Regra *SendBroadCast* de um *ActiveNode*.

A regra *StartBroadCast* (Figura 52) trata o recebimento da lista de AN conhecidos. Caso a lista não esteja vazia (existe pelo menos um AN conhecido), o AN requisitante armazena em seu atributo *names* a lista de identificações recebidas e envia a si mesmo uma mensagem *StartBCast*, contendo a referência à cápsula como parâmetro. O teste sobre o atributo *inBroadCast* serve para impedir que outro *broadcast* inicie e as informações do *broadcast* atual sejam sobrescritas. Isto porque as referências aos destinos de um *broadcast* são armazenadas no atributo *names* e assim, caso outro *broadcast* seja iniciado, os dados

contidos em *names* seriam perdidos. Como especificado, um novo *broadcast* só acontecerá quando um *broadcast* inicial tenha terminado (valor de *inBroadCast* é *false*).

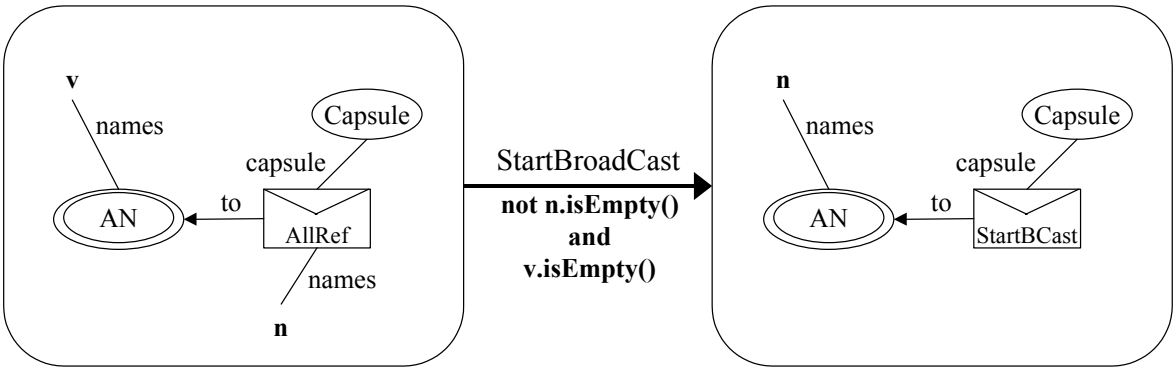

Figura 52. Regra *StartBroadCast* de um *ActiveNode*.

A mensagem *StartBCast*, tratada pela regra *ResolveNextName* (Figura 53), serve para iniciar o processo de envio em *broadcast*. Ao receber tal mensagem, o AN retira o primeiro elemento da lista e envia um pedido de obtenção de referência ao NS. Note-se que agora o parâmetro *type* da mensagem *Resolve* tem o valor 1, o que identifica que a resposta servirá para um *broadcast*.

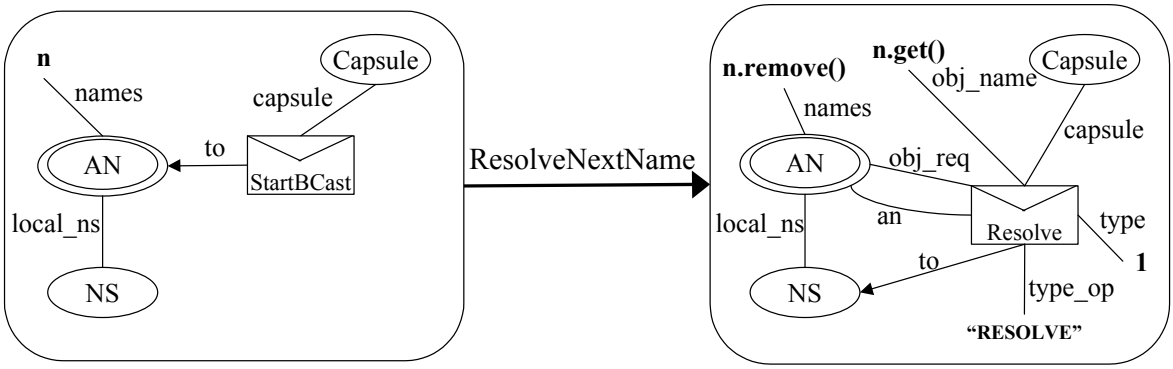

Figura 53. Regra *ResolveNextName* de um *ActiveNode*.

Quando a resposta é retornada pelo NS, é executada a regra *CloneCapsule* (Figura 54). Como já foi dito, para enviar uma cápsula em *broadcast*, é preciso criar uma cópia da cápsula para cada AN de destino. Assim, o AN envia a mensagem *Clone* para a cápsula, requisitando que esta se autoduplique, e uma mensagem *Next* para si mesmo, contendo a referência à cápsula.

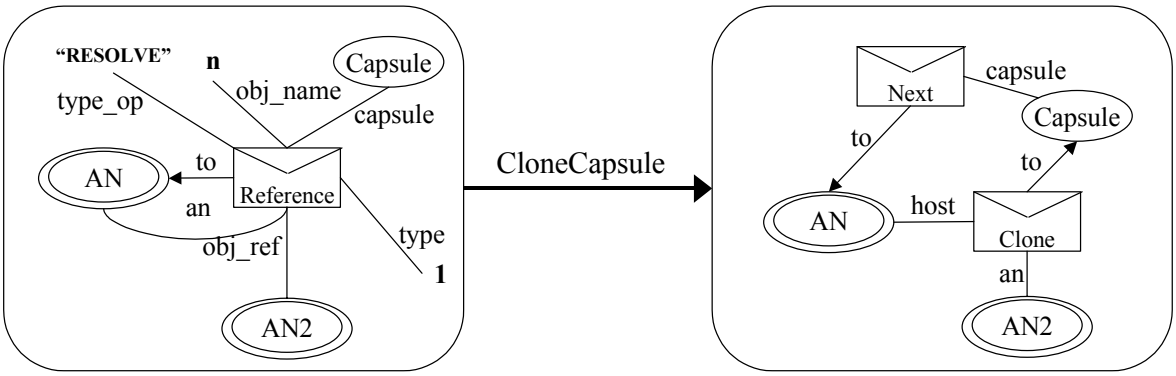

Figura 54. Regra *CloneCapsule* de um *ActiveNode*.

A mensagem *Next* é usada para representar a repetição do processo de envio da cápsula para um destino, como será visto logo adiante. Quando a cápsula retorna a referência a sua duplicata, através da mensagem *NewCapsule*, a regra *SendToNext* (Figura 55) é executada. Com isso, é encaminhada uma mensagem *Send* para a duplicata, enviando-a para o AN que havia sido retirado da lista.

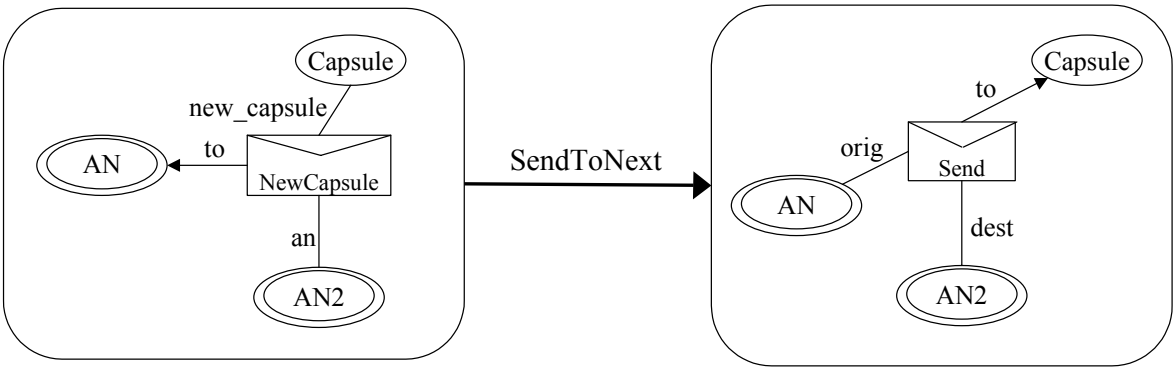

Figura 55. Regra *SendToNext* de um *ActiveNode*.

Ao receber a mensagem *Next*, o AN repete a processo visto até aqui. Dessa forma, ele retira o próximo AN da lista e requisita uma referência ao AN com o envio da mensagem *Resolve* ao NS, conforme apresentado na Figura 56.

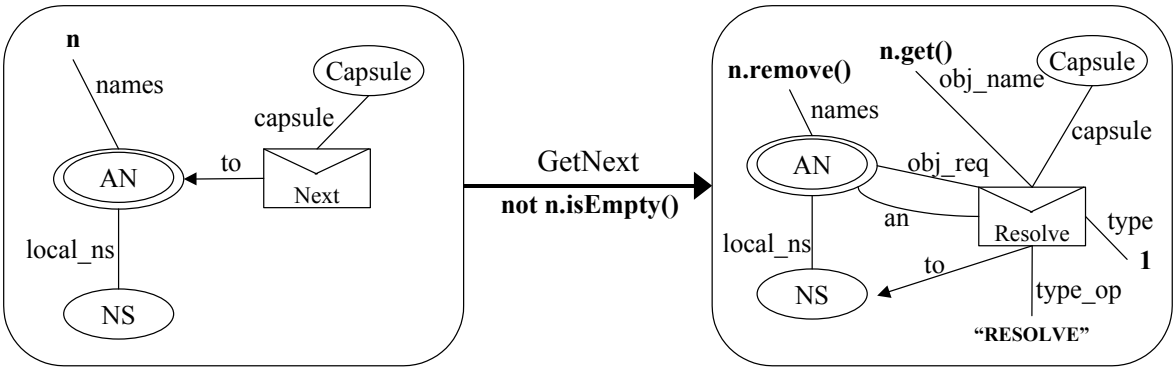

Figura 56. Regra *GetNext* de um *ActiveNode*.

Quando a reposta do NS é retornada, é novamente executa a regra *CloneCapsule* (Figura 54) e as regras subseqüentes. Este processo se repete até que a lista de AN conhecidos esteja vazia. Isto significa que a cápsula foi enviada a todos os AN conhecidos. A regra que determina o fim do envio de um *broadcast* é a regra *EndBCast*, apresentada na Figura 57.

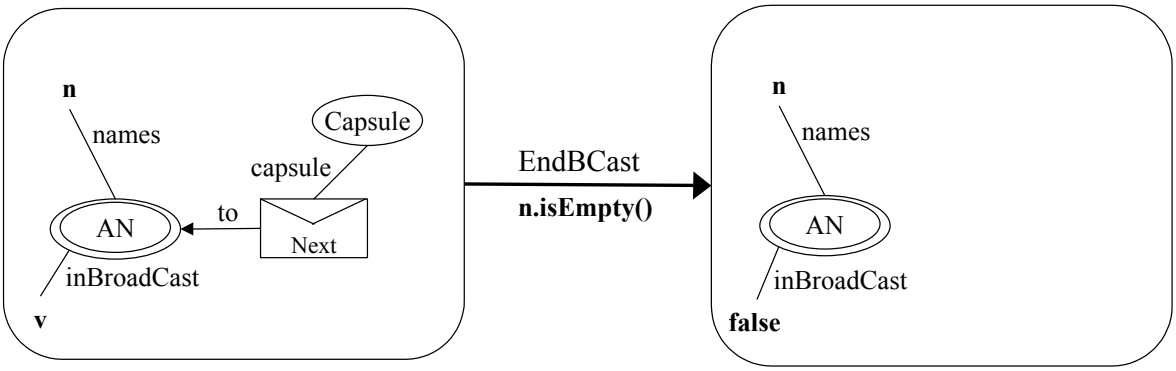

Figura 57. Regra *EndBCast* de um *ActiveNode*.

A regra *EndBCast* tem a função de mudar o valor do atributo *inBroadCast* para *false* (independentemente do valor atual desse atributo), indicando que um novo *broadcast* pode iniciar.

A função mais importante de um AN, que o diferencia realmente de um nodo comum, é a instalação dinâmica de serviços. Isto lhe dá a capacidade de agregar serviços sob demanda. Assim, um AN pode iniciar possuindo apenas os serviços básicos e, de acordo com o necessário, pode requisitar e instalar os serviços de que precisar. Como apresentado no grafo de tipos de um AN (vide Figura 47), cada nodo ativo possui a lista de serviços de que dispõe (atributo *services*). Assim, quando um AN recebe uma cápsula e esta lhe informa de qual serviço necessita, sua primeira providência é verificar se ele possui o serviço pedido (serviço consta na lista). Se o serviço não constar na lista de serviços disponíveis, o AN então verifica se já não gerou uma requisição pedindo o mesmo serviço. O AN realiza esta verificação buscando a identificação do serviço procurado na lista de pedidos pendentes (atributo *pending*). Caso o serviço também não tenha sido requisitado ainda, o AN gera então uma requisição para a base de código conhecida (CB) pedindo o serviço, conforme especificado na regra *RequestService* descrita na Figura 58.

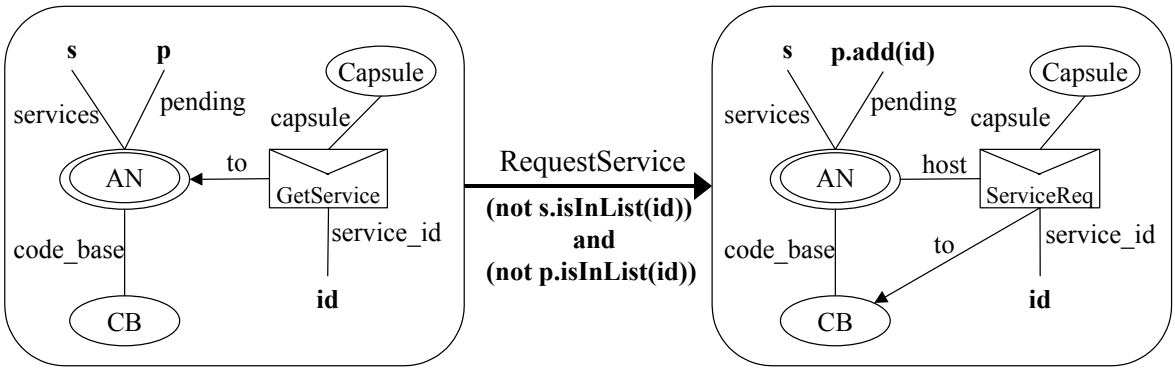

Figura 58. Regra *RequestService* de um *ActiveNode*.

Um pedido de requisição de serviço é feito através da mensagem *ServiceReq*. Nesta mensagem constam a identificação do serviço requisitado, uma referência ao AN que está requisitando serviço e uma referência à cápsula que necessita do serviço. Este último parâmetro serve para a manutenção da referência à cápsula, a fim de utilizá-la mais tarde, tal como já discutido anteriormente.

Ao chegar ao AN, o serviço requisitado faz um pedido de instalação. A instalação de um serviço compreende a inclusão de sua identificação na lista de serviços disponíveis do AN e o seu registro no servidor de nomes local. Ao mesmo tempo em que realiza a instalação do serviço, o AN passa para a cápsula requisitante a referência ao serviço para que ela possa utilizá-lo através da mensagem *Service*. O comportamento de um AN para a instalação de um serviço é descrito na regra *ServiceInstallation* na Figura 59.

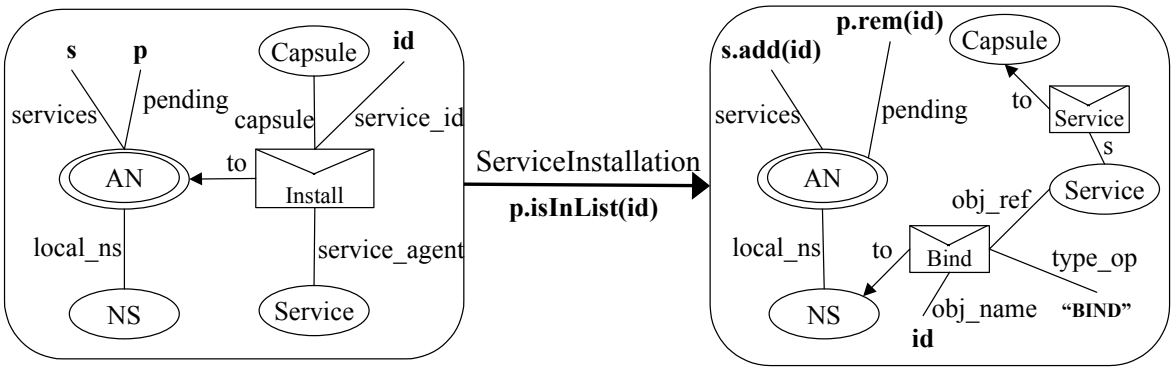

Figura 59. Regra *ServiceInstallation* de um *ActiveNode*.

A condição estabelecida na regra *ServiceInstallation* de que só são aceitas instalações de serviços previamente requisitados (ou seja, que constam na lista de requisições pendentes) é usada para impedir a instalação de serviços desnecessários ao nodo ou já

instalados. Assim que um serviço requisitado é recebido, ele é retirado da lista de requisições pendentes.

Se uma cápsula chegar ao AN e requisitar um serviço que já foi requisitado à base de código e que ainda não foi fornecido, o pedido do serviço será postergado, conforme a semântica de GGBO. Assim, logo que o serviço seja instalado, ele passará a constar e na lista de serviços disponíveis e os pedidos serão tratados pela aplicação da regra *FindService* (Figura 60).

Caso, ao receber um pedido de um serviço de uma cápsula, o AN possua o serviço, ele requisita uma referência ao serviço ao servidor de nomes local (regra *FindService*, na Figura 60) e, repassa a referência obtida à cápsula (regra *ProvideService*, na Figura 61).

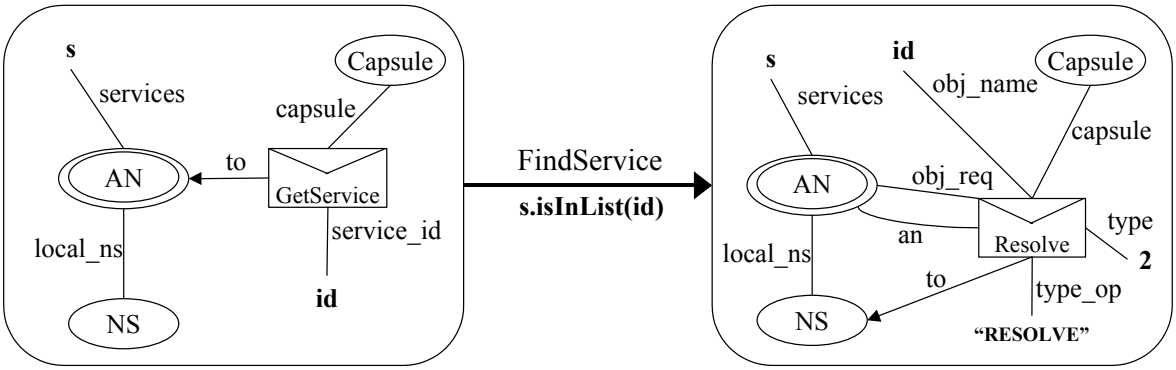

Figura 60. Regra *FindService* de um *ActiveNode*.

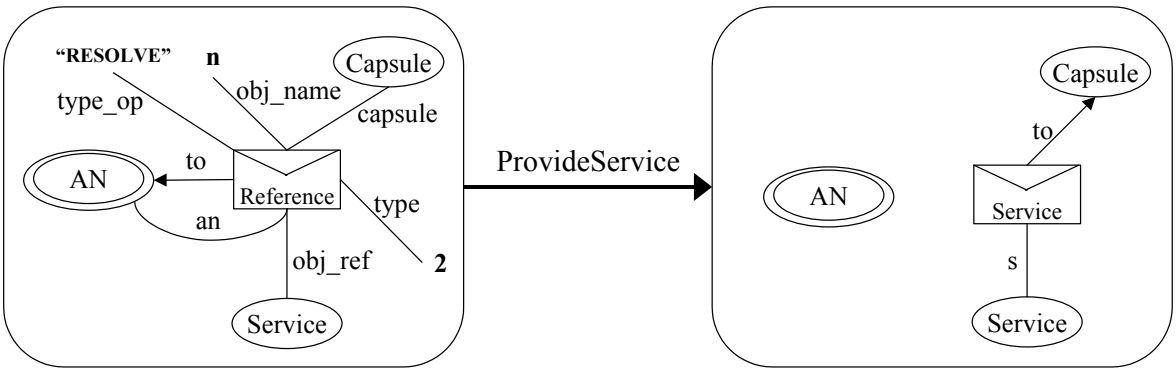

Figura 61. Regra *ProvideService* de um *ActiveNode*.

Como visto, o AN realiza uma função de intermediador entre cápsulas e serviços, provendo o necessário para que estes possam interagir.

6.2.2 Especificação de uma Cápsula

Para representar uma cápsula foi criada a entidade *Capsule*. Esta entidade foi definida segundo o grafo de tipos apresentado na Figura 62.

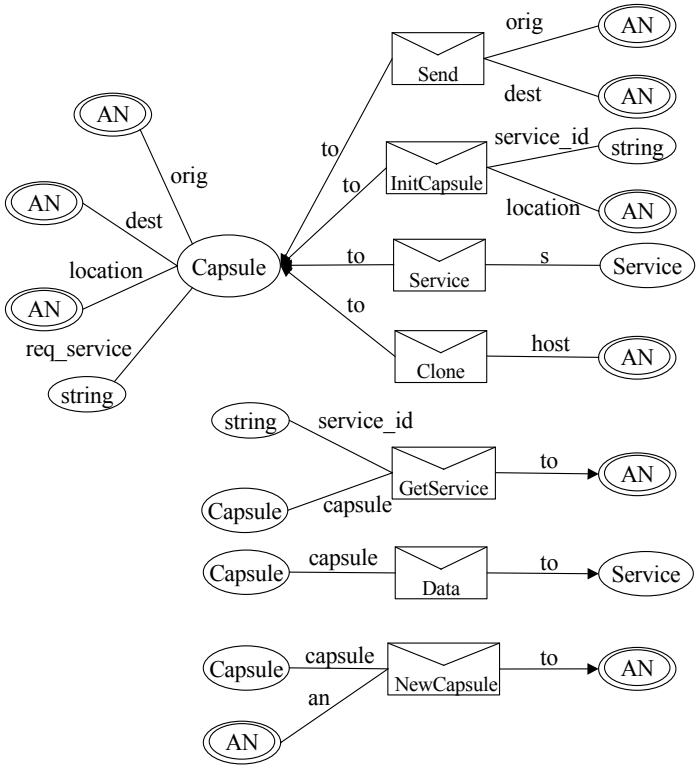

Figura 62. Grafo de tipos da entidade *Capsule*.

Uma cápsula realiza o papel de um pacote que trafega entre os nodos da rede. Dessa forma, ela possui sempre a informação de qual o nodo que originou a cápsula (atributo *orig*) e qual o nodo de destino (atributo *dest*). Sendo que a entidade *Capsule* estende o comportamento da entidade *MAgent*, ela também possui a informação de sua localização corrente (atributo *location*). O atributo restante (atributo *req_service*) informa qual o tipo de serviço exigido pela cápsula e que deve ser provido pelo AN que receber esta cápsula.

Cápsulas podem ser criadas dinamicamente, o que determina que sua especificação inclua uma regra que defina como ocorre a inicialização de uma cápsula recém criada. A inicialização de uma cápsula criada dinamicamente é necessária para que ela possa receber os valores iniciais de seus atributos. Como visto em 3.1.1, as regras de inicialização de entidades devem se basear no esquema de regra definido e apresentado na Figura 7. Assim, o esquema de regra de inicialização de uma entidade é adaptado para representar a inicialização de uma cápsula, conforme apresentado na Figura 63.

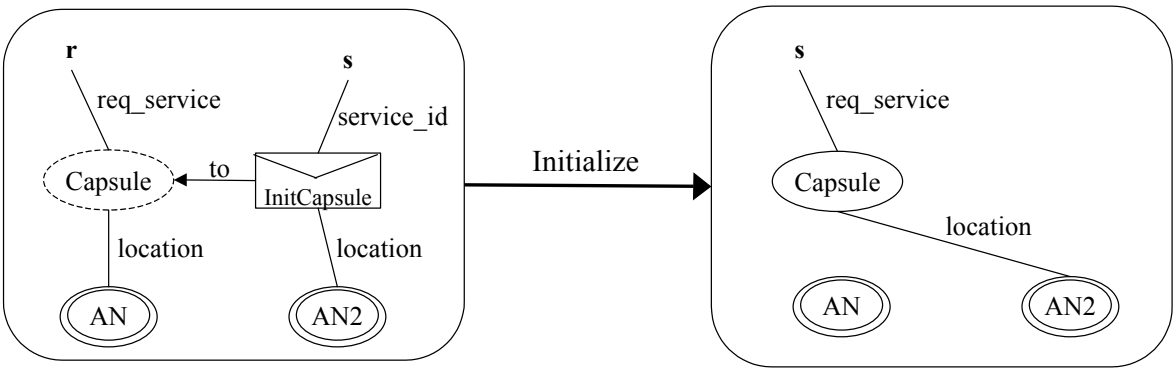

Figura 63. Esquema de regra de inicialização de uma *Capsule*.

A inicialização de uma cápsula compreende apenas a atribuição de valor inicial ao seu atributo que indica o serviço necessário para interagir com ela e a seu atributo de localização.

A função básica de uma cápsula é mover-se entre nodos da rede. A movimentação de uma cápsula ocorre a partir de uma requisição do AN onde se encontra. Um AN requisita a movimentação de uma cápsula através da mensagem *Send*, conforme ilustrado na regra *GoToHost*, apresentada na Figura 64.

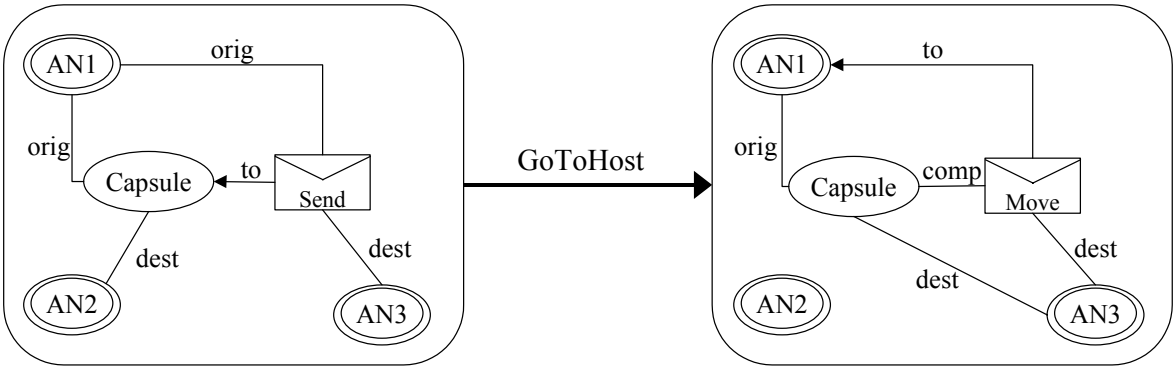

Figura 64. Regra *GoToHost* de uma *Capsule*.

A movimentação de uma cápsula é realizada conforme o estabelecido nas regras básicas de lugares e componentes móveis, descritas na Seção 3.1. Isto significa que o processo de movimentação é iniciado pelo envio da mensagem *Move* pela cápsula ao AN local. Como também definido nas regras básicas, o final de uma movimentação determina o recebimento, pelo componente móvel, de uma mensagem *Continue* (movimentação foi bem sucedida) ou *NoGo* (não foi possível mover o componente). O tratamento de uma movimentação sem sucesso fica a cargo do especificador da aplicação específica, não sendo, portanto, considerado aqui. No caso de uma movimentação sem problemas, o

comportamento a ser seguido pela cápsula é apresentado na Figura 65 na regra *AskForService*.

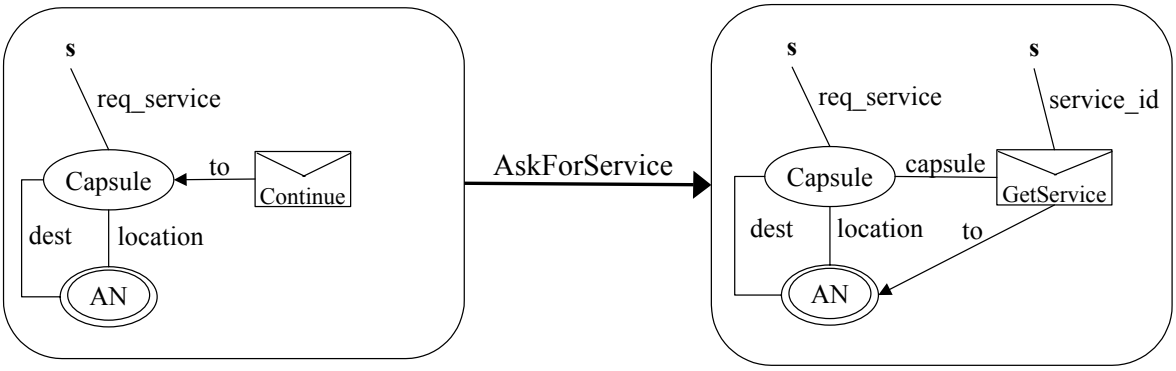

Figura 65. Regra *AskForService* de uma *Capsule*.

Logo que chega a um AN, a cápsula deve requisitar o serviço de que necessita. Por isso, ela envia a mensagem *GetService* ao AN em que se encontra informando qual serviço deseja. A referência ao serviço é retornada pelo AN através da mensagem *Service*. Esta mensagem é tratada de acordo com o estabelecido pela aplicação da qual a cápsula e o serviço fazem parte. Assim, definiu-se o esquema de regra *SendToService* (Figura 66) para representar a interação inicial entre uma cápsula e o serviço que a trata.

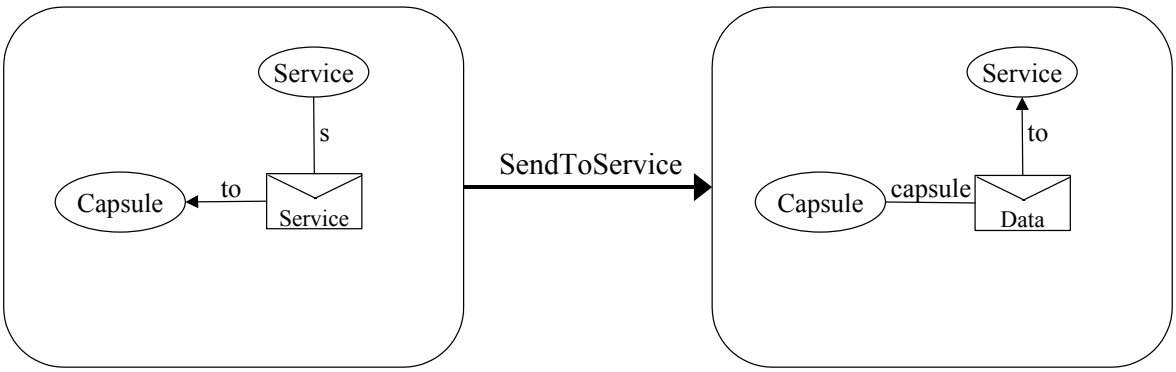

Figura 66. Esquema de regra *SendToService* de uma *Capsule*.

Este esquema de regra define que a cápsula deve enviar uma mensagem ao serviço passando sua referência, permitindo que cápsula e serviço possam interagir. Além da referência, a cápsula pode ainda passar dados que sejam necessários, conforme a aplicação necessite. A partir daí, a interação entre estas duas entidades ocorre conforme a aplicação específica.

Como visto na descrição do comportamento de um AN (vide Seção 6.2.1), para que seja feito o envio de cápsulas em modo *broadcast*, é necessário que elas sejam duplicadas.

Para isso, o esquema de regra *CreateClone* define como ocorre a duplicação de uma cápsula. Este esquema de regra é apresentada na Figura 67.

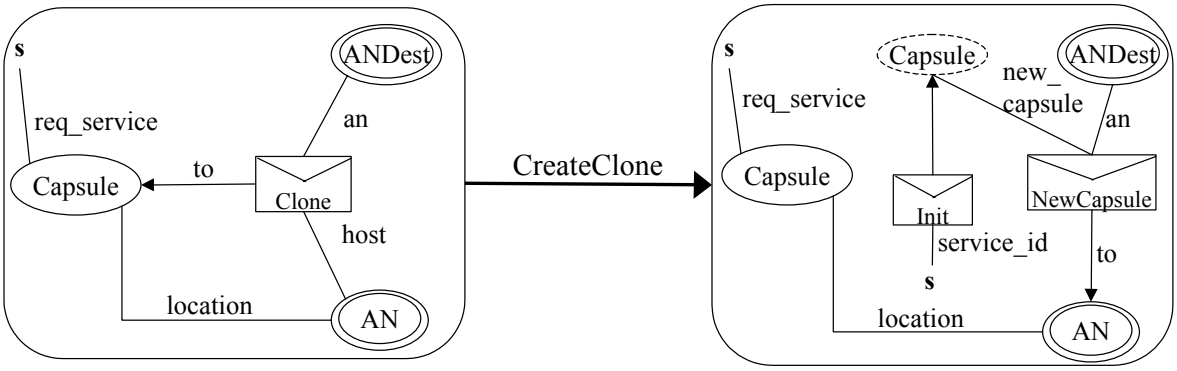

Figura 67. Esquema de regra *CreateClone* de uma *Capsule*.

A duplicação da cápsula determina a criação de uma nova cápsula e a sua inicialização com o mesmo estado interno da cápsula original. Atributos de entidades que estendam *Capsule* também devem ser considerados quando for definida a regra de duplicação destas entidades particulares.

6.2.3 Especificação de um Serviço

Um serviço é representado pela entidade *Service*. *Service* assume o comportamento da entidade *MAgent*, uma vez que um serviço deve poder ser movido de uma base de código para um AN. O grafo de tipos da entidade *Service* é apresentado na Figura 68.

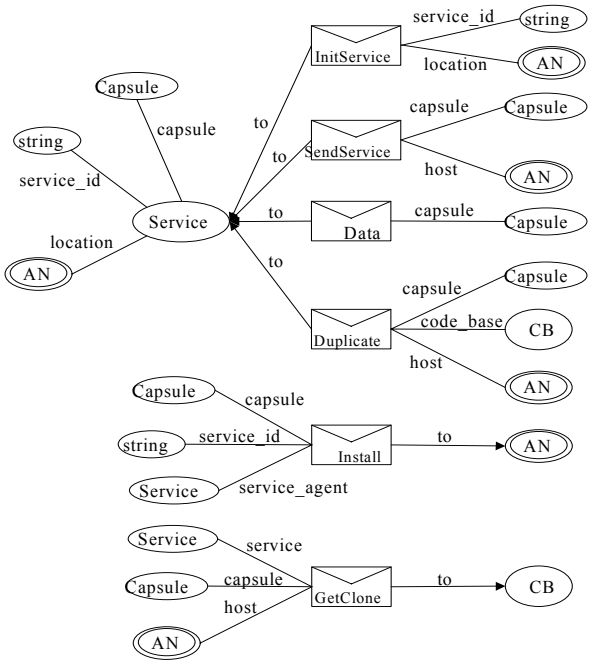

Figura 68. Grafo de tipos da entidade *Service*.

Um serviço possui uma identificação (atributo *service_id*) e, sendo um componente móvel, armazena também a informação de sua localização atual (atributo *location*). O atributo *capsule* serve apenas para armazenamento temporário. O uso desse atributo será visto mais adiante.

Como já apresentado quando da descrição das especificações de nodos ativos e cápsulas, um serviço executa em um AN e interage com as cápsulas que necessitarem deste serviço. Assim como as cápsulas, serviços podem ser criados dinamicamente, já que novos serviços podem ser implantados na rede a qualquer momento. A Figura 69 apresenta o esquema de regra de inicialização de um serviço.

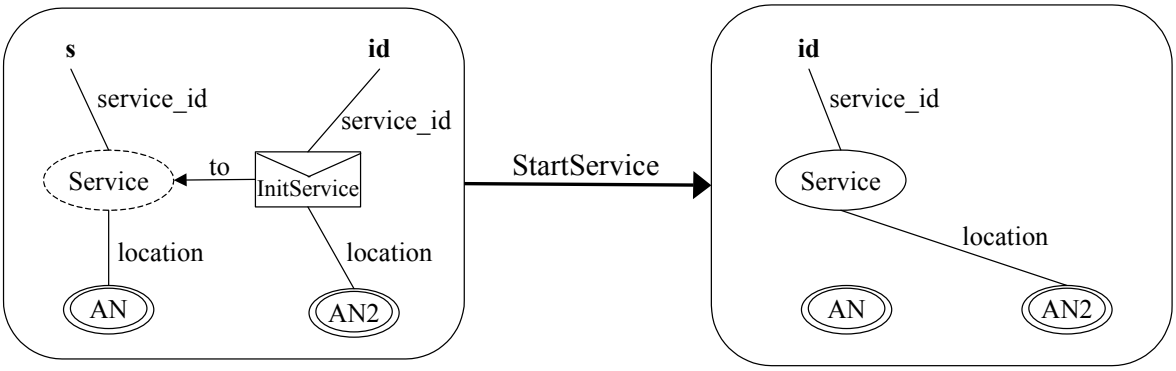

Figura 69. Esquema de regra de inicialização da entidade *Service*.

Sendo um componente móvel, um serviço move-se através do comportamento descrito nas regras básicas da Seção 3.1. O comando para a movimentação de um serviço é dado por uma base de código (CB). Portanto, um serviço move-se sempre de uma base de código para um nodo ativo, onde deverá executar. A movimentação do serviço é iniciada pelo recebimento da mensagem *SendService*, segundo apresentado na regra *GoService*, a qual pode ser vista na Figura 70.

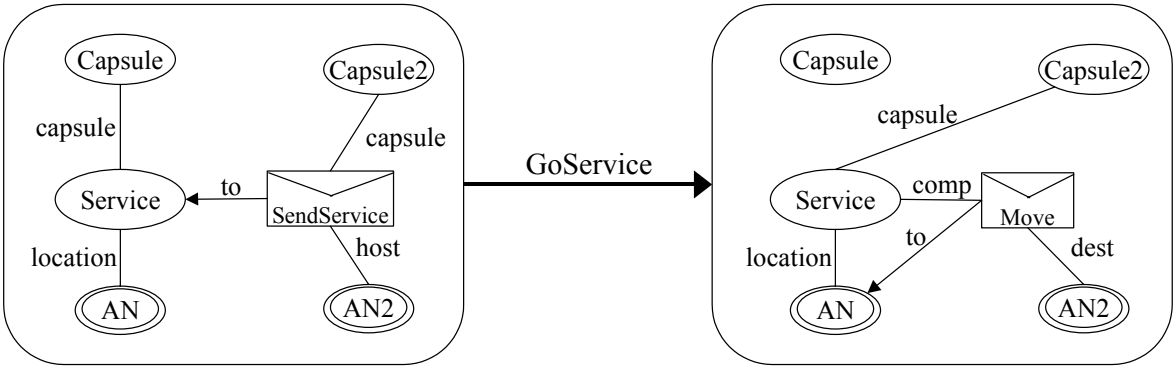

Figura 70. Regra *GoService* de um *Service*.

Nesta regra, o atributo *capsule* de um serviço é utilizado para armazenar a referência à cápsula que requisitou o serviço, a fim de não perdê-la durante o processo de movimentação. Assim, logo que a movimentação termina, o serviço pode recuperar a referência à cápsula de forma a encaminhá-la ao AN em que se encontra através da mensagem *Install*. Esta mensagem informa o AN de que o serviço deseja ser instalado e aguarda a interação com a(s) cápsula(s) que o requisitaram. A regra *RequestInstallation* (Figura 71) apresenta o envio do pedido de instalação.

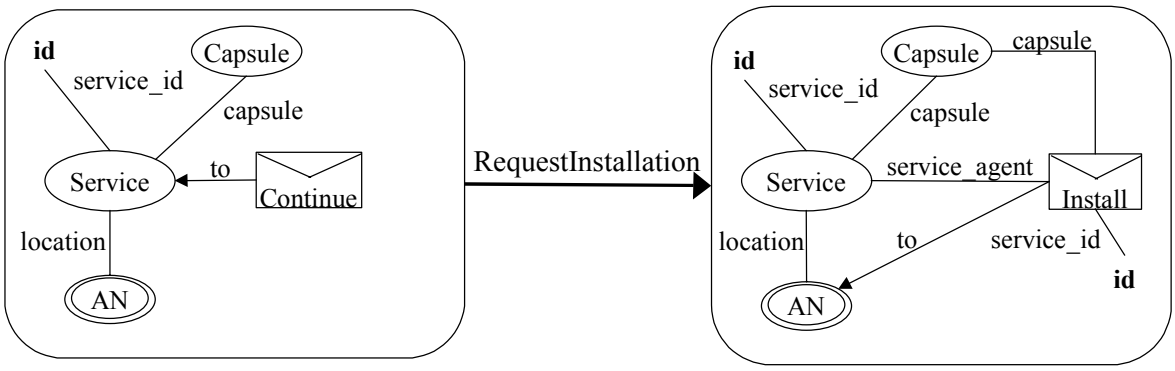

Figura 71. Regra *RequestInstallation* de um *Service*.

Como ocorre com as cápsulas, serviços também precisam ser duplicados. Isto porque a base de código mantém os serviços originais e envia aos AN as réplicas. A duplicação de um serviço ocorre conforme o esquema de regra apresentado na Figura 72.

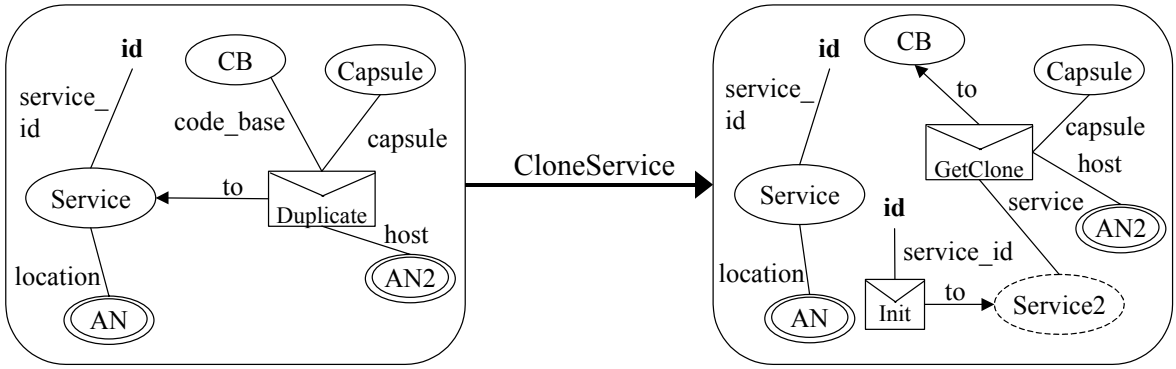

Figura 72. Esquema de regra de duplicação de um *Service*.

A utilização de serviços na arquitetura proposta deverá ser melhor entendida quando for apresentada a especificação do algoritmo de *Dynamic Source Routing*, na Seção 7.2.

6.2.4 Especificação de uma Base de Código

A base de código é uma entidade que se baseia na entidade *MAgent* e é a responsável por manter a lista de serviços disponíveis na rede. O grafo de tipos da entidade *CodeBase* (CB), que representa uma base de código, é apresentado na Figura 73 e na Figura 74.

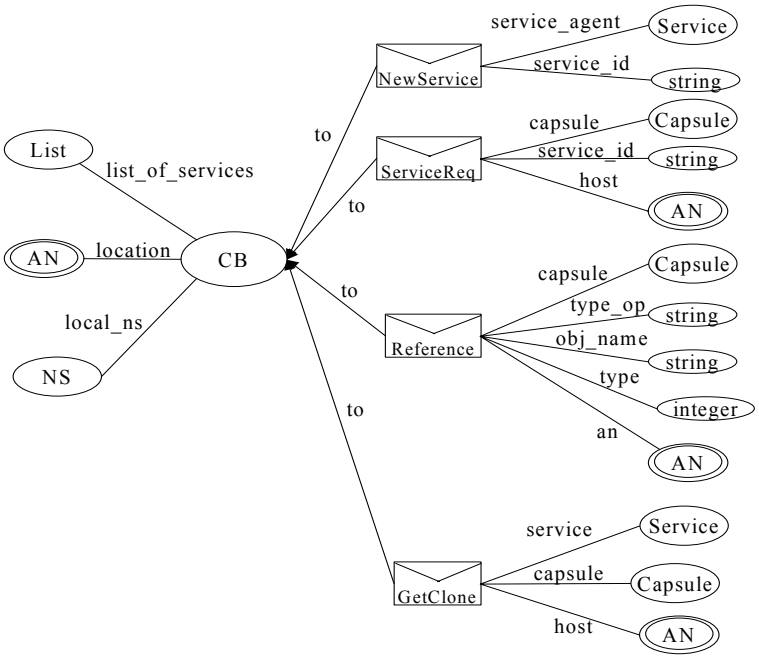

Figura 73. Grafo de tipos da entidade *CodeBase* – Parte 1.

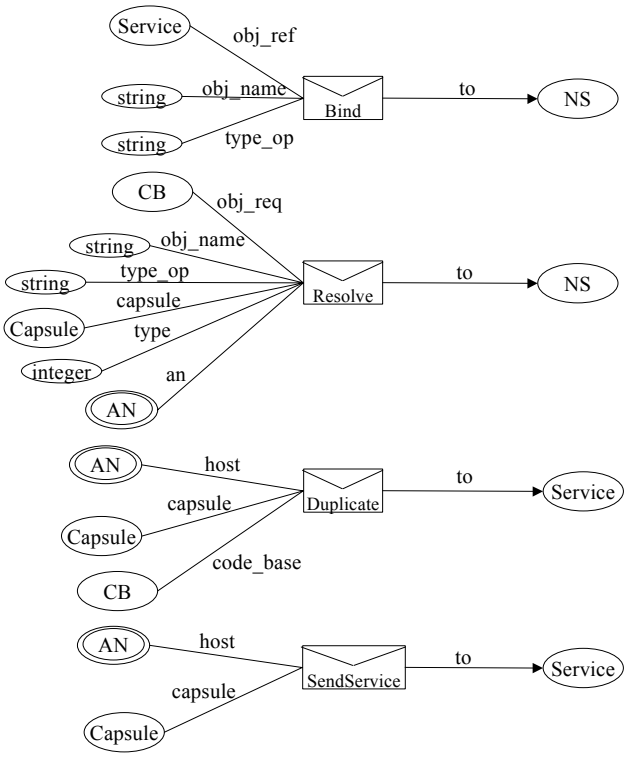

Figura 74. Grafo de tipos da entidade *CodeBase* – Parte 2.

Uma base de código, como já dito, controla a lista de serviços que podem ser providos pelos nodos da rede. Esta lista mantém as identificações dos serviços disponíveis (atributo *list_of_services*). Para obter a referência a um serviço cuja identificação consta nesta lista, a CB possui uma referência a um servidor de nomes local à CB (NS) no qual são registrados os serviços de toda a rede.

Como entidade controladora da lista de serviços, a CB pode receber requisições de adição de novos serviços, conforme apresenta a regra *AddService* ilustrada na Figura 75.

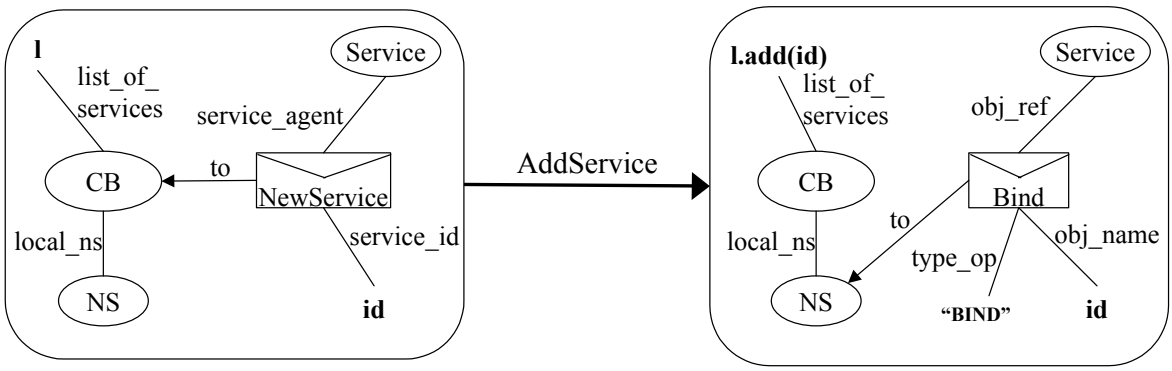

Figura 75. Regra *AddService* de uma *CodeBase*.

O registro de um serviço é feito através de sua identificação (*service-id*) e de sua referência (*service_agent*), as quais são passadas ao NS. A identificação do serviço é também adicionada à lista de serviços da CB, significando que o novo serviço está agora disponível.

Além da manutenção da lista de serviços, uma CB também provê o envio de serviços a serem executados em nodos da rede. Isto ocorre quando um AN requisita, através da mensagem *ServiceReq*, uma instância de um serviço. A CB utiliza a identificação do serviço recebida por parâmetro e verifica se tal identificação consta na sua lista. Depois dessa verificação, é feita uma requisição ao NS local para a obtenção de uma referência ao serviço pedido pelo AN, segundo descrito na regra *GetServiceReference*, apresentada na Figura 76.

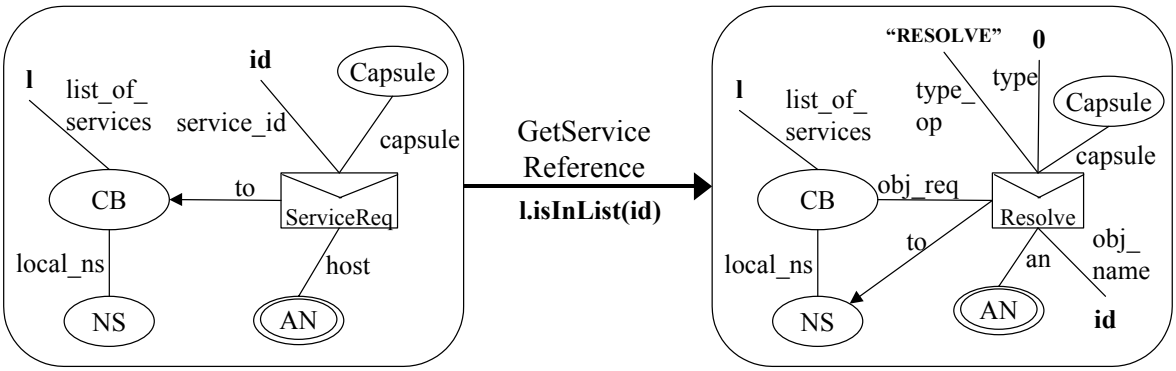

Figura 76. Regra *GetServiceReference* de uma *CodeBase*.

De posse da referência pedida, a CB, a fim de enviar o serviço ao AN que o requisitou, solicita ao serviço a sua duplicação, assim como pode ser visto na regra *ServiceReference* na Figura 77.

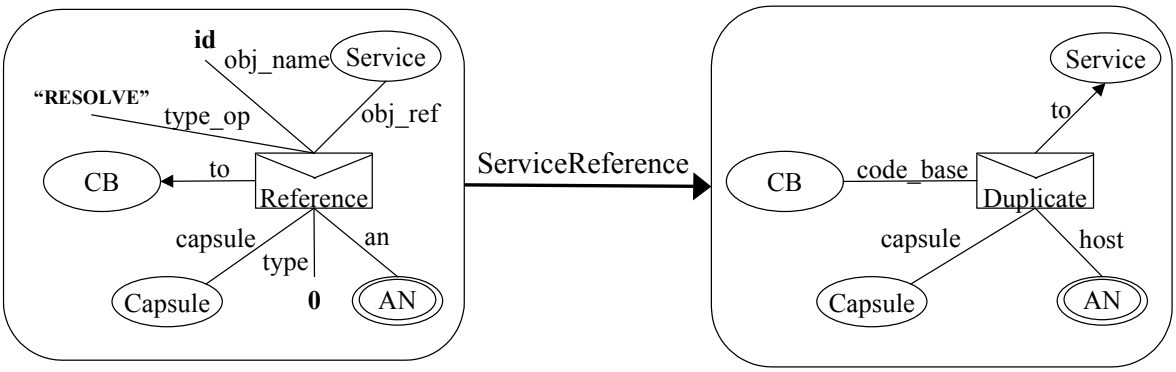

Figura 77. Regra *ServiceReference* de uma *CodeBase*.

A regra *MoveService* apresenta o recebimento da referência à nova instância do serviço (Figura 78), a qual é, então, encaminhada ao AN requisitante.

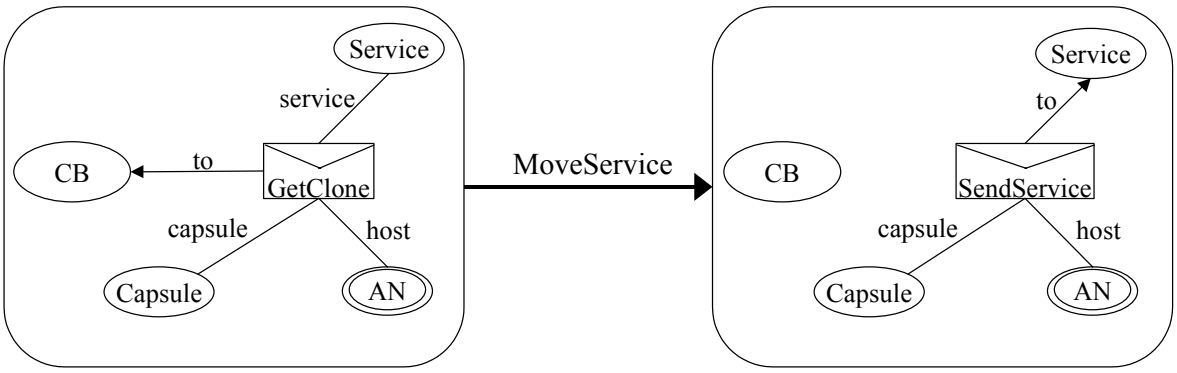

Figura 78. Regra *MoveService* de uma *CodeBase*.

Não foi especificado um comportamento a ser adotado quando a CB recebe um pedido de um serviço que não conste em sua lista. Isto porque se assume que todas as

cápsulas que trafegam na rede possuem um serviço previamente registrado pela CB. Assim, os pedidos de serviços pelos AN só ocorrerão em relação a serviços constantes na lista de serviços da CB. Obviamente que em um ambiente aberto não se pode ter essa garantia, uma vez que cápsulas podem transitar entre redes diversas. Mas, por simplificação da especificação, tal questão não é tratada aqui, podendo fazer parte de um estudo futuro.

7 Estudo de Caso 2 – Roteamento em Redes Ativas

Um segundo estudo de caso desenvolvido apresentava um cenário de uma rede onde os nodos são nodos ativos (AN). Ou seja, um cenário onde a rede era uma rede ativa tal como descrito no primeiro estudo de caso. Assim, todo o trabalho nessa rede ocorre segundo a especificação de redes ativas apresentada na Seção 6.2. Nesta rede, definida como uma rede fixa (nodos são interligados por conexões físicas) no cenário deste estudo de caso, cada AN está conectado a alguns AN da rede, com os quais pode se comunicar diretamente. O encaminhamento de mensagens entre nodos da rede é feito segundo o algoritmo de roteamento *Dynamic Source Routing (DSR)*, o qual é apresentado a seguir.

7.1 Algoritmo de Roteamento DSR

O algoritmo de roteamento *Dynamic Source Routing (DSR)* [JOH01] foi idealizado para prover roteamento de mensagens entre nodos em redes *ad hoc*. Uma rede *ad hoc* é um tipo de rede móvel (nodos não possuem conexão física entre si), onde os dispositivos computacionais são capazes de trocar informações diretamente entre si [CAM99]. Dessa forma, não existe a conexão dos nodos com uma estação base, a qual intermedia a comunicação entre nodos, tal como ocorre na comunicação entre telefones celulares. Numa rede *ad hoc* qualquer nodo pode ser trabalhar como um roteador, encaminhando pacotes recebidos de outros nodos para os seus nodos de destino.

O DSR é um algoritmo do tipo *source routing*, isto é, com roteamento na fonte. Assim, o nodo origem deve, ao enviar o pacote, determinar todo o caminho a ser percorrido por este pacote. O DSR não gera mensagens periódicas para atualização de informações de roteamento, como ocorre com outros algoritmos, o que significa economia de banda de rede e de energia da bateria dos dispositivos móveis. Apesar disso, seu uso é aconselhável apenas em pequenas redes, com cinco a dez nodos.

O protocolo DSR é composto por dois mecanismos: o mecanismo de Descoberta de Rotas e o mecanismo de Manutenção de Rotas. Estes mecanismos são apresentados a seguir.

7.1.1 Descoberta de Rotas

Descoberta de Rotas é o mecanismo pelo qual um nodo de origem obtém, dinamicamente, uma rota a um nodo destino, para quem deseja enviar um pacote. Este mecanismo é usado toda vez que o nodo de origem não conhece uma rota para o nodo de destino.

Todos os nodos possuem uma *cache de rotas (CR)*, na qual ficam armazenadas as rotas conhecidas para outros nodos, podendo ser armazenadas várias rotas para um mesmo destino. Assim, quando um nodo fonte F deseja enviar um pacote a um nodo destino D, ele verifica em sua *cache* se possui uma rota para D. Caso haja uma rota previamente conhecida para D, o pacote é encaminhado por esta rota. Se não possuir uma rota, o nodo F inicia o processo de descoberta de rota, enviando um *broadcast* de um *pacote de requisição de rota (PReq)*. Neste pacote consta a identificação do nodo que originou a requisição, a identificação do nodo para o qual está sendo requisitada uma rota e uma lista de nodos pelos quais a requisição passou, representando a rota da origem ao destino, a qual é inicializada com a inclusão da identificação do nodo originador. Também é adicionado ao PReq um número único de identificação da requisição, gerado pelo originador. A lista de nodos visitados é inicializada com a identificação do nodo que originou o PReq.

Quando um nodo recebe um PReq, ele segue o seguinte algoritmo:

⇒ Se ele for o destino procurado, então:

- Gera um *pacote de reposta (PResp)*, contendo a lista de nodos do PReq recebido, mais a sua identificação, formando a rota completa da origem ao destino. Este pacote é encaminhado ao originador, sendo enviado ao primeiro nodo rota. A resposta ao originador pode ser encaminhada de três formas: através de uma rota para o originador presente na CR; através da obtenção de uma rota pela realização de um novo processo de descoberta de rota; ou enviando a resposta pela rota contrária aquela constante na lista de nodos do PReq;

⇒ Caso não seja o destino procurado, então:

- Se a identificação do nodo já estiver na lista de nodos do PReq recebido, o nodo ignora a requisição, já que já a havia recebido anteriormente;
- Se não estiver na lista de nodos, o nodo verifica se possui, em sua CR, uma rota para o destino procurado:

- Se houver uma rota, gera um PResp para o originador, adicionando a rota de sua CR à lista de nodos do PReq recebido;
- Se não houver uma rota, adiciona-se à lista de nodos e gera um *broadcast* encaminhando o PReq aos seus vizinhos.

Cada nodo também possui uma *lista de requisições recentes (LRR)*, contendo a identificação de todas as requisições recebidas recentemente. Assim, ao receber uma requisição, o nodo verifica se já não havia recebido aquela mesma requisição. Se isto tiver ocorrido, o nodo ignora a requisição, não a repassando adiante. Isto impede que haja uma inundação de requisições na rede.

Quando um nodo recebe um PResp, ele age da seguinte forma:

⇒ Se for o destino da resposta, então:

- Adiciona a rota formada pela lista de nodos constante na resposta a sua CR e a utiliza para encaminhar o pacote de dados ao seu destino;

⇒ Caso não seja o destino, então:

- Retira sua identificação da rota de retorno do PResp, adiciona a rota resultante (rota do nodo até o origem) a sua CR e envia a resposta ao próximo nodo da lista.

Quando um nodo recebe um pacote de dados e verifica que é o destino, ele encaminha o pacote à aplicação devida. Caso não seja o destino, o nodo encaminha o pacote ao próximo nodo da rota. Para armazenar pacotes de dados que estejam esperando a descoberta de uma rota para serem enviadas, cada nodo também possui um *buffer* de pacotes. Assim, logo que uma resposta de requisição de rota é recebida, o nodo retira do *buffer* todas as mensagens a serem enviadas aquele destino e as encaminha.

Tanto a CR quanto a LRR possuem *timers* para controlar o tempo de validade de cada entrada. Dessa forma, cada rota aprendida deve ser descoberta novamente após um dado período de tempo. Os pacotes no *buffer* de pacotes a serem enviados também possuem um *timer*, de forma a impedir que pacotes fiquem indefinidamente na espera.

7.1.2 Manutenção de Rotas

Manutenção de Rotas é o mecanismo pelo qual um nodo pode detectar se a topologia da rede mudou. Isto significa que algumas rotas deixaram de ser válidas e/ou novas rotas foram criadas.

Para realizar a manutenção de rotas, os pacotes de dados enviados entre dois nodos exigem confirmação por parte do receptor. Esta confirmação é feita com o envio de uma mensagem de *acknowledgment*. Cada nodo, ao receber um pacote de dados para encaminhar adiante, coloca uma cópia do pacote em um *buffer* de retransmissão. Assim, caso não receba a confirmação do recebimento do pacote pelo nodo receptor dentro de um intervalo de tempo, o nodo que enviou o pacote realiza a retransmissão do mesmo. Se após um certo número de tentativas não houver resposta do receptor, o nodo gera um *pacote de erro (PE)*, informando que um dado *link* não está ativo. Quando um nodo recebe um PE, ele verifica em sua CR todas as rotas que incluam aquele *link* e as remove. Caso o nodo que recebeu o PE seja o nodo que originou o pacote, ele pode tentar enviá-lo por uma rota alternativa constante em sua CR ou realizar novamente o processo de descoberta de rotas.

Como visto, os mecanismos de Descoberta de Rotas e de Manutenção de Rotas são utilizados por demanda. Ou seja, só são iniciados quando necessário. Dessa maneira, reduz-se o tráfego na rede causado por mensagens de roteamento.

7.1.3 Exemplo de Realização do Algoritmo DSR

Para ilustrar o funcionamento básico do algoritmo DSR, tome-se como exemplo a rede apresentada na Figura 79, onde as ligações entre os nodos representam conexões físicas, no caso de uma rede fixa, ou os nodos alcançáveis a partir de cada nodo no caso de uma rede móvel. Na situação de uma rede móvel, os nodos alcançáveis por um nodo são aqueles que estão dentro da área de ação do sinal do equipamento utilizado pelo nodo.

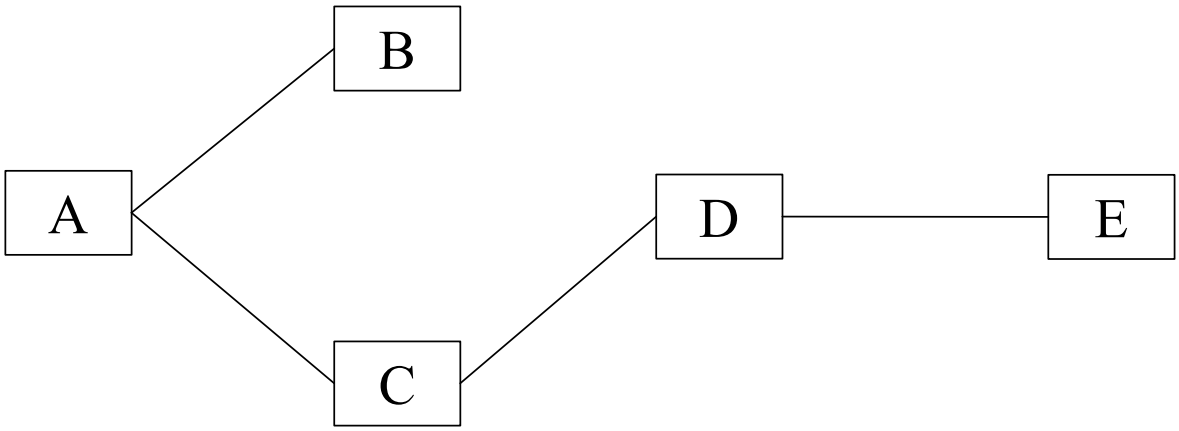

Figura 79. Exemplo de rede em que executa o algoritmo DSR.

Suponha-se que o nodo A deseja enviar dados para o nodo E. Assim, o nodo A gera um *broadcast* para os seus nodos vizinhos (neste caso, os nodos B e C) da requisição de rota contendo uma lista de nodos visitados, inicializada com o próprio nodo A. Ao chegar ao nodo B, este identifica que não é o nodo procurado e não possui uma rota para o nodo E. Logo, B gera um novo *broadcast*, o qual, dentro da topologia desta rede, alcançará apenas o nodo A. Note-se que o nodo A ignorará a requisição recebida por identificar que tal requisição já passou por ele, visto que ele consta na lista de nodos da requisição gerada por B. Já a requisição que chega a C, é encaminhada adiante, com a adição de C à lista de nodos. O nodo D age da mesma forma que C, encaminhando a requisição aos seus nodos vizinhos, até que a mensagem chegue no nodo E. Este processo é ilustrado na Figura 80.

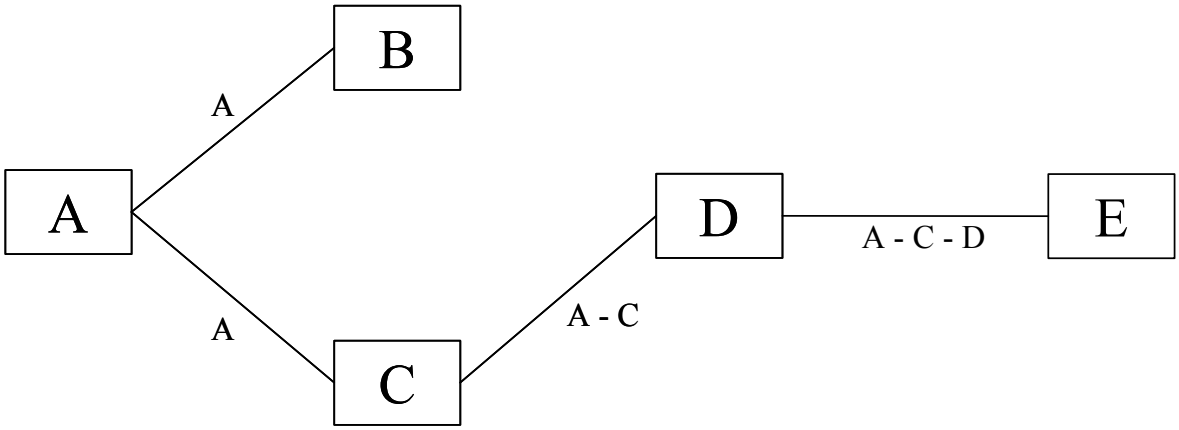

Figura 80. Ilustração da geração de uma busca de rota.

Quando a requisição chega ao nodo E, este identifica que é o nodo procurado e gera uma resposta ao nodo A contendo a lista de nodos por que a requisição passou (rota de A até E). Assume-se aqui que a resposta é encaminhada percorrendo a rota inversa (rota de E até A). Dessa forma, a resposta chega ao nodo A conforme apresentado na Figura 81. Cabe

salientar que, mesmo que a forma de encaminhamento da resposta fosse outra (através de um novo processo de descoberta de rotas, por exemplo), o comportamento dos nodos por onde a resposta passa é sempre mesmo.

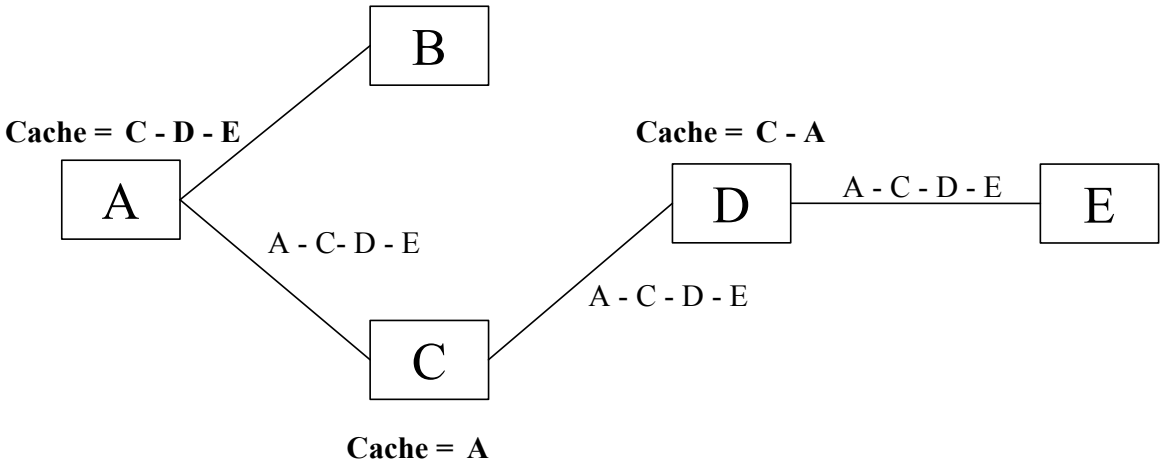

Figura 81. Ilustração de uma resposta a uma requisição de rota.

Cada nodo por onde a resposta passa (D e C) aproveita a informação da rota inversa para adicionar uma rota para o nodo A em sua CR. Assim, o nodo D sabe que possui uma rota até A através de C e o nodo C sabe que possui um rota direta para A.

De posse de uma rota para o nodo E, o nodo A encaminha os dados necessários através da rota obtida, assim como mostra a Figura 82.

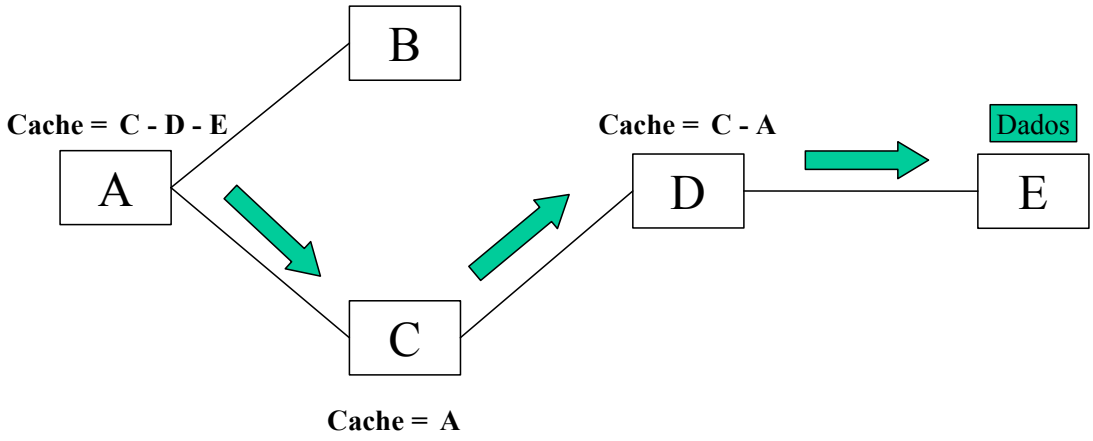

Figura 82. Ilustração da entrega de dados através da rota obtida.

Como dito na descrição do algoritmo DSR, o nodo A poderia manter mais de uma rota para E em sua CR. Isto possibilita que ocorram menos processos de descoberta de rotas na rede, uma vez que se uma rota falhar, os dados podem ser encaminhados por uma rota alternativa.

7.2 Especificação em GGBO do Algoritmo DSR

São agora apresentadas as especificações das entidades envolvidas no algoritmo DSR já apresentado. Cabem, antes disso, algumas considerações quanto à especificação feita. A primeira delas é que os *timers* que controlam a validade das entradas das tabelas envolvidas no algoritmo (*cache* de rotas e lista de requisições recentes) não foram modelados. Isto porque a questão do trabalho com tempo em GGBO foge ao escopo deste trabalho, dado que o importante é o trabalho com o formalismo de GGBO.

A segunda observação que deve ser feita é que também não foram modelados os *buffers* de mensagens, usados para armazenar pacotes a serem retransmitidos em caso de erro (*Retransmission Buffer*) e para armazenar pacotes que estão à espera da obtenção de uma rota ao destino para serem encaminhados (*Send Buffer*). A modelagem destes *buffers* exigiria a criação de listas cujos elementos seriam referências a entidades que representam os pacotes, o que não é permitido em GGBO. Portanto, para modelar tais *buffers* teriam de ser criados servidores de nomes para armazenar referências. Como o objetivo deste estudo de caso é, principalmente, avaliar o mapeamento para código de especificações em GGBO, preferiu-se, por simplicidade, não modelar os *buffers* citados.

A última observação a ser feita é que, segundo o ambiente que foi considerado para a especificação das regras básicas (discutido no início da Seção 3.1), não se considera a existência de falhas. Portanto, assume-se que os pacotes chegam ao seu destino. Com isso, a existência do pacote de erro (PE) e o comportamento necessário para tratá-lo não foram incluídos na especificação a ser apresentada.

Dessa forma, a especificação do algoritmo DSR concentrou-se no que mais importava avaliar que era a movimentação de entidades e a comunicação entre elas, segundo as regras definidas.

A seguir são apresentadas as especificações da entidades envolvidas no algoritmo. Conforme modelado, as entidades são o serviço DSR e as cápsulas *RouteRequest*, *RouteReply* e *Packet*.

7.2.1 Especificação do Serviço DSR

O serviço DSR fornece o comportamento necessário para tratar as cápsulas do algoritmo DSR. Dessa forma, a especificação deste serviço determina a forma de execução

do algoritmo no ambiente de redes ativas proposto. A Figura 83 e a Figura 84 apresentam o grafo de tipos do serviço DSR.

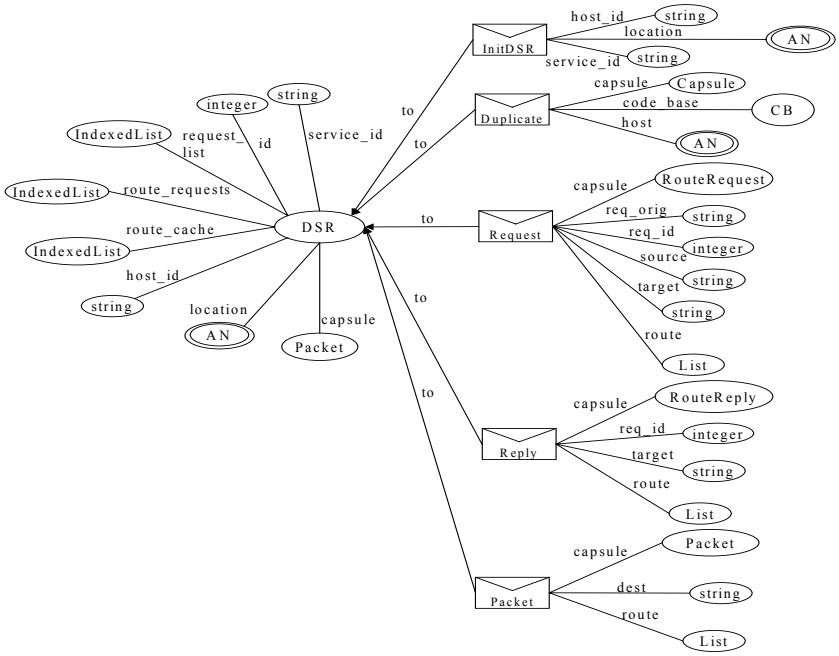

Figura 83. Grafo de tipos do serviço DSR – Parte 1.

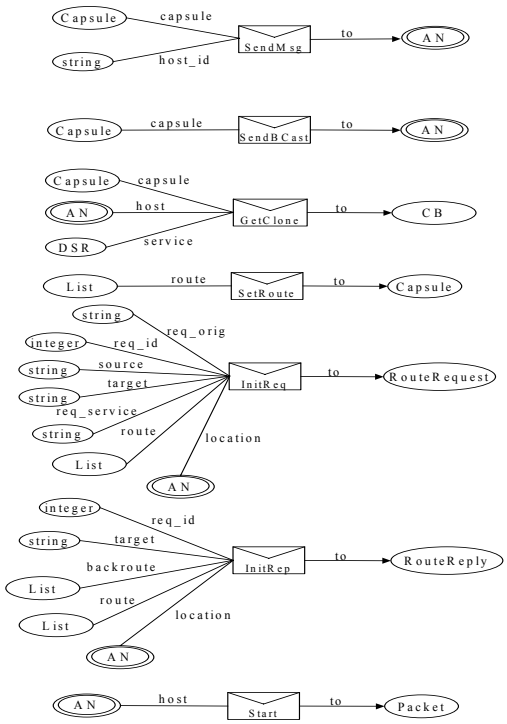

Figura 84. Grafo de tipos do serviço DSR – Parte 2.

O serviço de DSR possui a sua identificação (atributo *service_id*), que compreende um nome que o identifica como sendo o serviço usado para executar o algoritmo DSR, e a

identificação do nodo em que está executando (atributo *host_id*). O atributo *id* serve para gerar os identificadores únicos usados quando o serviço envia uma requisição para obter uma rota a um dado destino. O seu uso será melhor esclarecido a seguir, na apresentação das regras do serviço DSR. O atributo *capsule* serve para armazenamento temporário de uma cápsula de dados. Onde aparecem parâmetros de mensagens do tipo *Capsule*, define-se que tais parâmetros podem ser de qualquer um dos tipos que estendem o comportamento de *Capsule*.

O serviço DSR controla a *cache* de rotas do nodo (atributo *route_cache*) e a lista de requisições recentes (atributo *route_requests*). Além disso, o serviço ainda mantém uma lista contendo as requisições enviadas e que estão a espera de resposta (atributo *request_list*). O tipo *IndexedList*, usado no grafo de tipos do serviço DSR, é um tipo definido neste trabalho e representa um lista indexada, onde cada item da lista é um par <índice, valor>. Os valores da lista podem ser de um dos tipos básicos de Gramáticas de Grafos (*boolean*, *integer*, *real*, *string* ou *character*) ou de um tipo definido pelo especificador, tal como o tipo *List*, já apresentado na Seção 6.2.1, ou mesmo do próprio tipo *IndexedList*. Índices podem ser apenas dos tipos básicos. As operações definidas para o tipo *IndexedList* são:

- *IndexedList add (index, item)*: Adiciona o par <*index*, *item*> à lista e retorna a nova lista;
- *IndexedList remove (index)*: Remove o elemento da lista cujo índice é *index* e retorna a nova lista;
- *item get (index)*: Retorna o valor *item* associado ao índice *index*;
- *boolean isInList (index)*: Retorna verdadeiro se existe algum elemento da lista cujo índice seja *index* e falso, caso contrário;
- *boolean isEmpty ()*: Retorna verdadeiro se a lista está vazia e falso, caso contrário.

O serviço DSR é inicializado através da mensagem *InitDSR*, contendo a identificação do serviço e a identificação do nodo em que ele está executando, segundo apresentado na regra de mesmo nome, a qual pode ser vista na Figura 85.

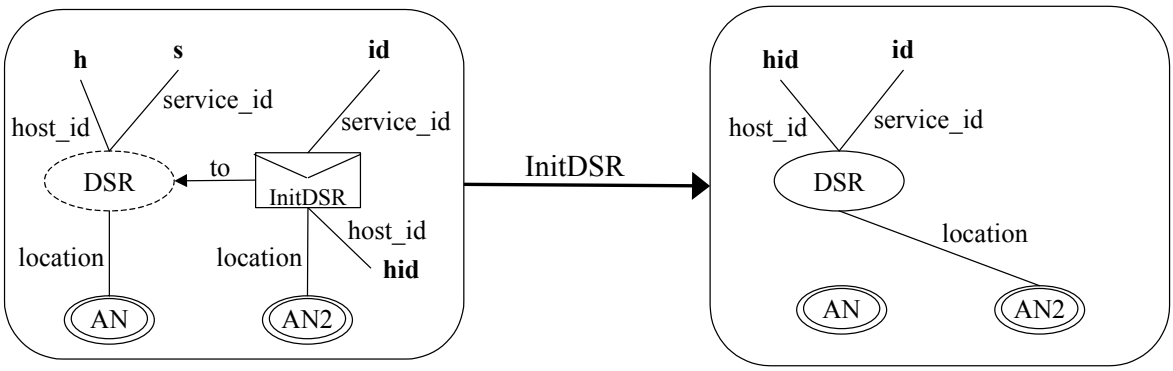

Figura 85. Regra de inicialização do serviço DSR.

O serviço DSR, assim como qualquer serviço, possui a capacidade de duplicar-se. No caso do serviço DSR, esta duplicação ocorre de acordo com a regra *CloneDSR*, apresentada na Figura 86. A duplicação de um serviço é usada para poder prover instâncias deste serviço para todos os nodos da rede que precisarem tratar cápsulas que necessitam deste serviço.

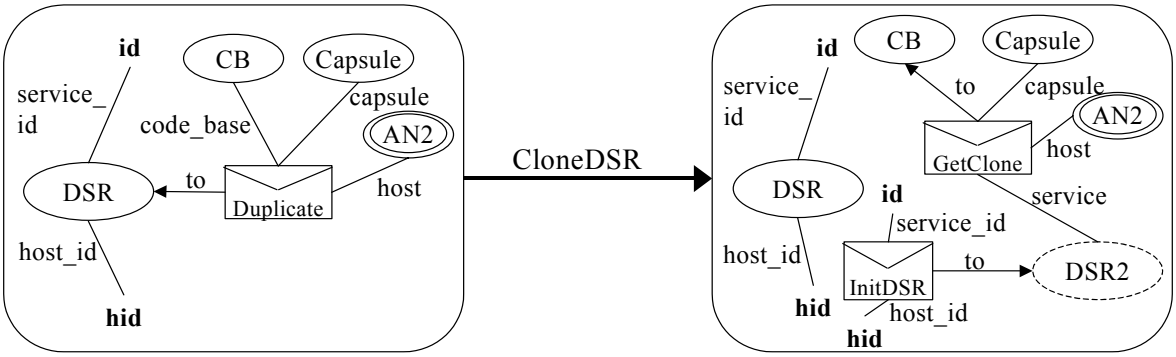

Figura 86. Regra *CloneDSR* do serviço DSR.

A função básica do serviço DSR é encaminhar cápsulas de dados por rotas constantes na *cache* de rotas (CR). As cápsulas a serem enviados são recebidos pelo serviço DSR através da mensagem *Packet*, a qual traz a identificação do nodo de destino da cápsula, a referência à cápsula a ser enviada e uma lista de nodos. A lista de nodos define a rota a ser seguida pela cápsula. Quando a cápsula está sendo originada, esta lista está vazia, o que determina que ainda não existe uma rota definida para a cápsula. Cápsulas de dados são originadas por aplicações usem o algoritmo DSR para roteamento destas cápsulas. Uma cápsula de dados a ser roteada segundo o algoritmo DSR deve estender o comportamento da cápsula *Packet*, a ser apresentada na Seção 7.2.4.

Quando o serviço DSR recebe uma cápsula de dados a ser enviada a outro nodo, ele verifica qual é o nodo de destino. Se o nodo de destino for um nodo remoto (não o próprio nodo onde o serviço está) e a lista de nodos estiver vazia, o serviço identifica que é uma cápsula a ser enviada. Para enviar a cápsula, o serviço verifica se já possui uma rota para o destino na sua CR. Possuindo uma rota, ele encaminha a cápsula para o primeiro nodo da rota e envia uma mensagem à cápsula para que ela atualize a informação de sua rota a ser seguida. Este procedimento é descrito na regra *SendPacket*, apresentada na Figura 87.

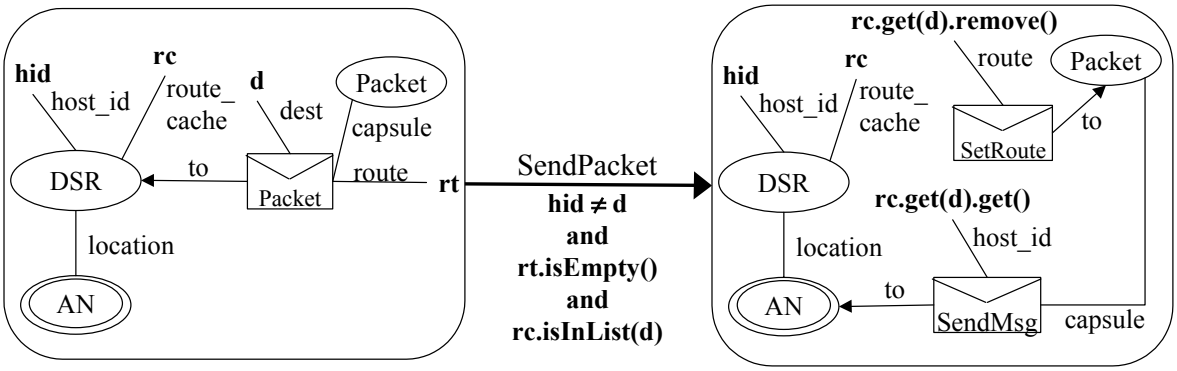

Figura 87. Regra *SendPacket* do serviço DSR.

Cada nodo que recebe a cápsula roteada, encaminha-a ao próximo nodo da rota, conforme a regra *ForwardPacket*, mostrada na Figura 88.

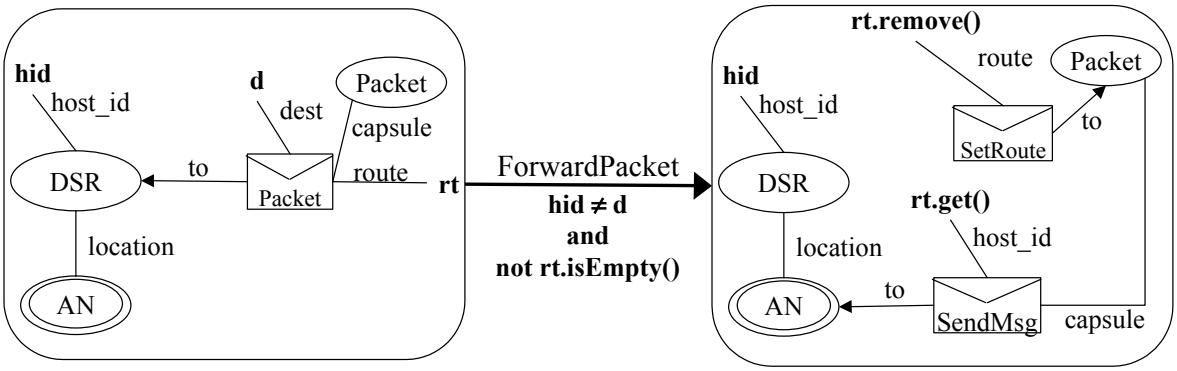

Figura 88. Regra *ForwardPacket* do serviço DSR.

Quando a cápsula de dados chega ao destino, ela é tratada conforme a aplicação à qual pertence. Para isso, o serviço DSR passa à cápsula de dados a referência ao nodo local (através da mensagem *Start*) e, a partir daí, o comportamento é definido pela aplicação. O recebimento de uma cápsula de dados pelo nodo de destino é apresentada na regra *GetPacket*, na Figura 89.

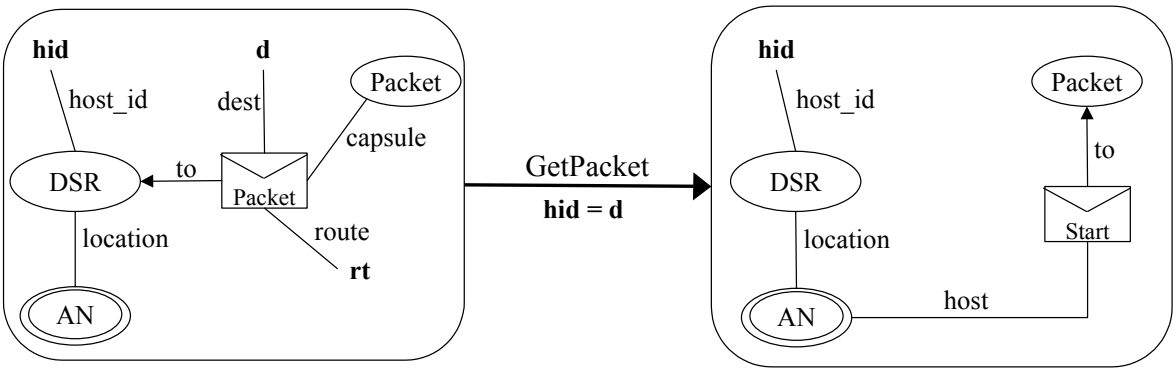

Figura 89. Regra *GetPacket* do serviço DSR.

No caso de não haver uma rota para o destino na CR do nodo que quer enviar uma cápsula de dados, o serviço inicia o processo de descoberta de rotas, seguindo o estabelecido no algoritmo DSR, descrito na Seção 7.1. Para isso, o serviço cria uma cápsula de *RouteRequest*. A regra *SendRequest* é apresentada na Figura 90 e ilustra esta situação.

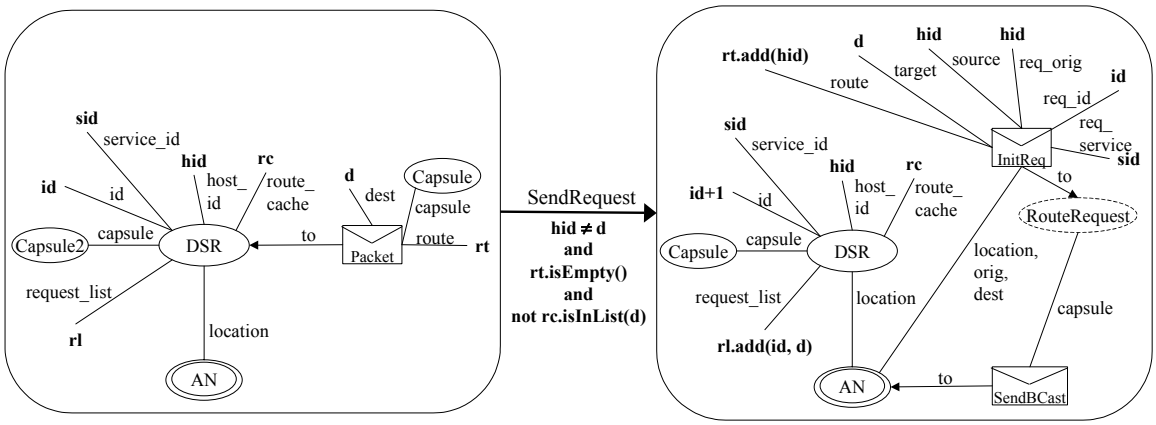

Figura 90. Regra *SendRequest* do serviço DSR.

A cápsula de *RouteRequest* é inicializada com os dados necessários e com uma lista de nodos visitados. Esta lista servirá para montar uma rota da origem ao destino e inicia contendo apenas a identificação do nodo de origem. A cápsula de *RouteRequest*, que representa um pacote de requisição de rota, é enviada em *broadcast*. O valor do atributo *id* do serviço DSR é adicionado, juntamente com a identificação do destino da cápsula, à lista *request_list* do serviço DSR. Isto serve para que seja possível, ao receber uma rota para o destino, identificar se a requisição foi mesmo feita por este nodo. O valor de *id* é incrementado para que a próxima requisição possua um valor diferenciado. A cápsula que deve ser enviada, e está à espera de uma rota, é armazenada no atributo *capsule* do serviço DSR. Como dito nas observações no início da descrição desta especificação, não se

considera o uso de *buffers* para armazenar cápsulas. Assim, a especificação do algoritmo DSR contempla apenas o roteamento de uma cápsula por vez, a qual é armazenada no atributo *capsule* do serviço DSR.

Ao receber uma cápsula *RouteRequest*, o serviço DSR do nodo, primeiramente, verifica se este nodo é o destino procurado. Neste caso, é criada uma cápsula *RouteReply* a ser enviada ao nodo que requisitou a rota. Esta cápsula carrega consigo a rota do nodo origem até o destino, formada pela lista de nodos visitados pela cápsula *RouteRequest* mais o nodo destino. Como discutido na apresentação do algoritmo DSR (vide Seção 7.1), o retorno a uma requisição de rota pode ocorrer de formas diferentes. Na especificação feita, determinou-se que o retorno da rota é feito seguindo-se a rota inversa. Ou seja, a cápsula *RouteReply* é enviada através dos nodos constantes na lista de nodos visitados na ordem contrária, do nodo destino até o origem. A criação da cápsula *RouteReply* e o envio desta para o primeiro nodo da rota inversa são descritos na regra *TargetFound* na Figura 91.

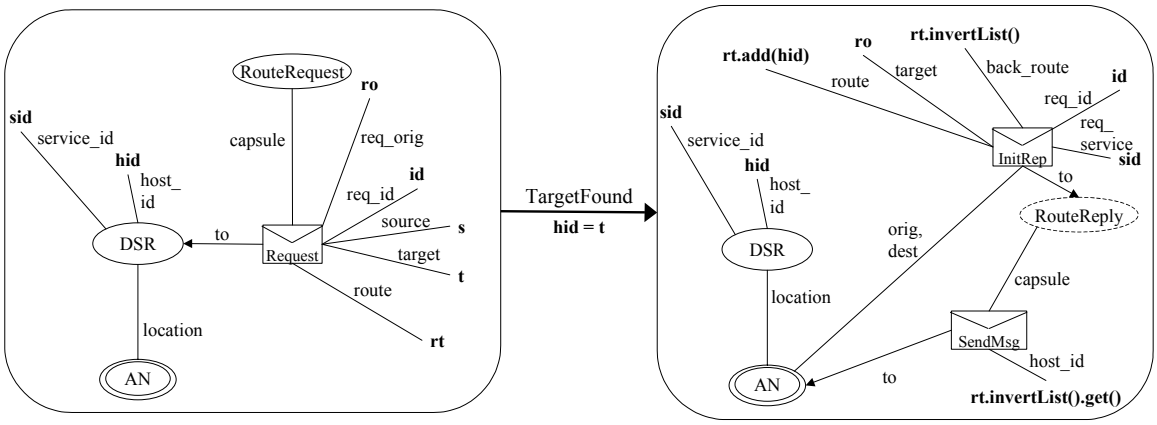

Figura 91. Regra *TargetFound* do serviço DSR.

Se o serviço DSR identifica que o nodo onde ele está não é o destino de um *RouteRequest* recebido, a sua próxima atitude é procurar uma rota para o destino em sua CR. Se houver uma rota para o destino na CR, o serviço DSR segue o mesmo procedimento da regra anterior, somente que a rota retornada é rota composta da lista de nodos visitados até então, mais a rota encontrada na CR. Isto porque a rota na CR possui apenas a informação do caminho do nodo local até o nodo procurado e deve, portanto, ser completada com o caminho do nodo de origem até o nodo local. Da mesma forma que antes, a resposta é enviada pela rota inversa. Esta situação é representada na regra *SendReply*, mostra na Figura 92.

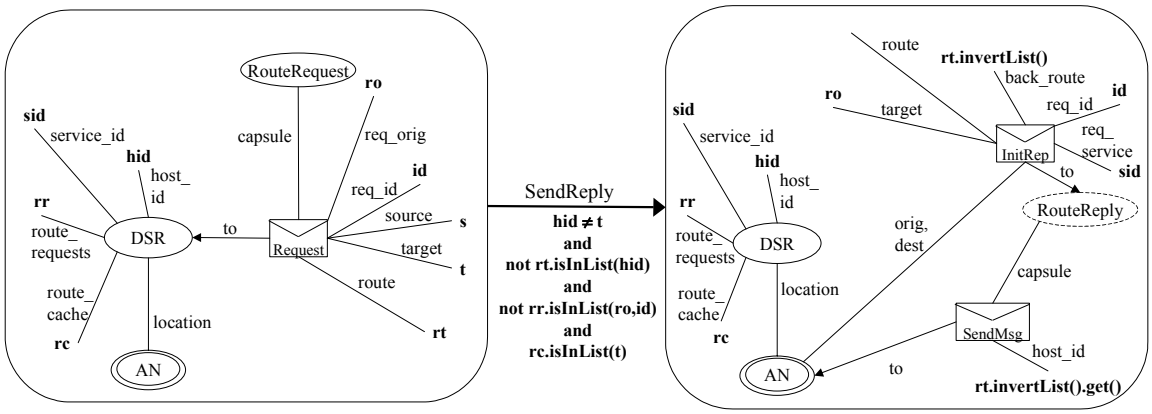

Figura 92. Regra *SendReply* do serviço DSR.

Antes de enviar uma rota que possua em sua CR, o serviço DSR também precisa verificar se a identificação do nodo já consta na lista de nodos visitados. Se ela já constar lá, significa que o nodo já havia recebido aquela requisição. Se isto ocorre, então é executada a regra *AlreadyInRoute*, apresentada na Figura 93, que determina que a requisição seja ignorada. Isto evita que haja uma proliferação indefinida de requisições na rede.

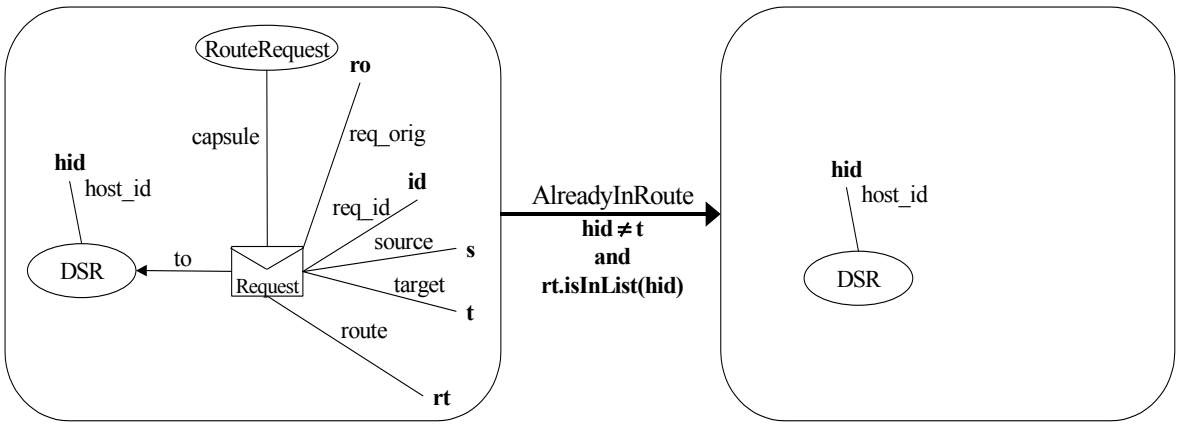

Figura 93. Regra *AlreadyInRoute* do serviço DSR.

Outra verificação feita pelo serviço DSR para saber se ele já havia recebido a mesma requisição antes é uma procura na sua lista de requisições recentes (LRR), representada pelo atributo *route_requests*. Lá constam todas requisições recebidas pelo serviço, com identificação de qual nodo originou a requisição e qual o nodo procurado. Se já houver o registro de um requisição igual à requisição recebida na LRR, a requisição é ignorada, tal como mostra a regra *RequestRepeated* na Figura 94.

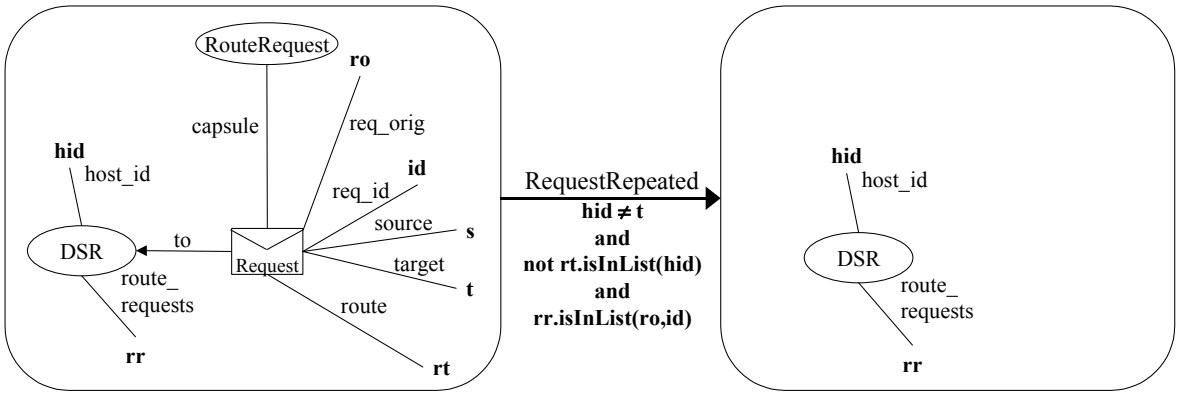

Figura 94. Regra *RequestRepeated* do serviço DSR.

Em último caso, se o nodo não é o destino, não existe um rota para o destino na CR, o nodo não consta na lista de nodos visitados e já não existe o registro da requisição recebida na LRR, o serviço DSR realiza o encaminhamento da requisição aos nodos conhecidos através do envio da cápsula *RouteRequest* em modo *broadcast*, conforme define a regra *ForwardRequest* na Figura 95.

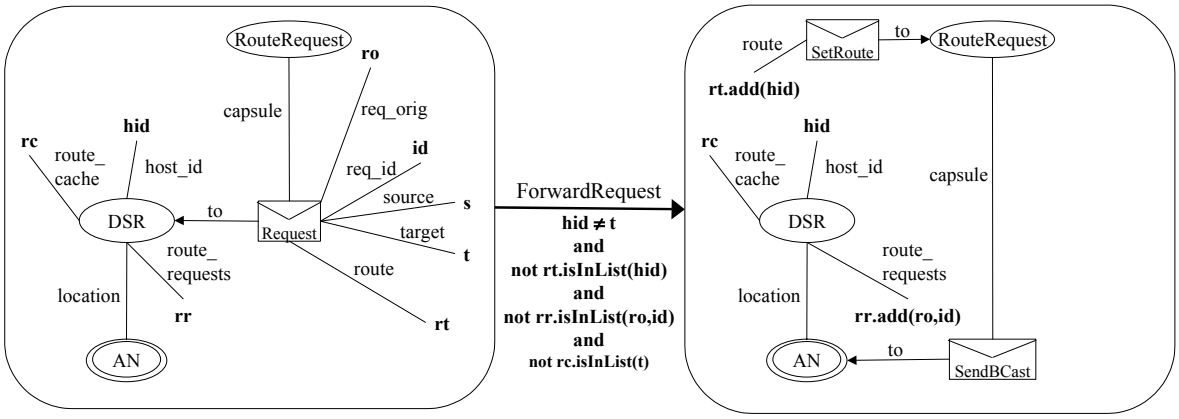

Figura 95. Regra *ForwardRequest* do serviço DSR.

O encaminhamento da requisição exige que a identificação do nodo local seja adicionada à lista de nodos visitados, causando a alteração da rota armazenada pela cápsula *RouteRequest*, e também a inclusão do registro do recebimento da requisição na LRR, a fim de evitar o encaminhamento de uma requisição repetida.

Após uma rota ter sido descoberta (requisição chegou ao destino procurado ou algum nodo possuía uma rota para o destino em sua CR), como visto, é retornada uma cápsula *RouteReply* pela rota inversa até o nodo origem da requisição. Ao receber uma cápsula *RouteReply*, o serviço DSR do nodo verifica se a resposta diz respeito a uma requisição gerada pelo nodo local. Isso é feito pelo serviço DSR através de uma procura pela

identificação da requisição na sua lista de requisições geradas (atributo *request_list* de um serviço DSR). No caso de a resposta ser relativa a uma requisição gerada pelo nodo local e o nodo ainda não havia recebido uma resposta (requisição consta na lista), o nodo então pode encaminhar a cápsula de dados. A cápsula a ser enviada é aquela que estava armazenada pelo serviço DSR. O encaminhamento da cápsula ocorre pela rota recebida na resposta, sendo, portanto, encaminhada pelo nodo local ao primeiro nodo na rota. A rota recebida é armazenada na CR para poder ser usada posteriormente, se necessário. As rotas da CR são indexadas pela identificação do nodo de destino. Como forma de identificar que a requisição feita para obter uma rota ao destino procurado não está mais pendente, o registro da requisição é retirado da lista de requisições geradas pelo nodo local. O procedimento descrito é realizado através da execução da regra *GetReply*, apresentada na Figura 96.

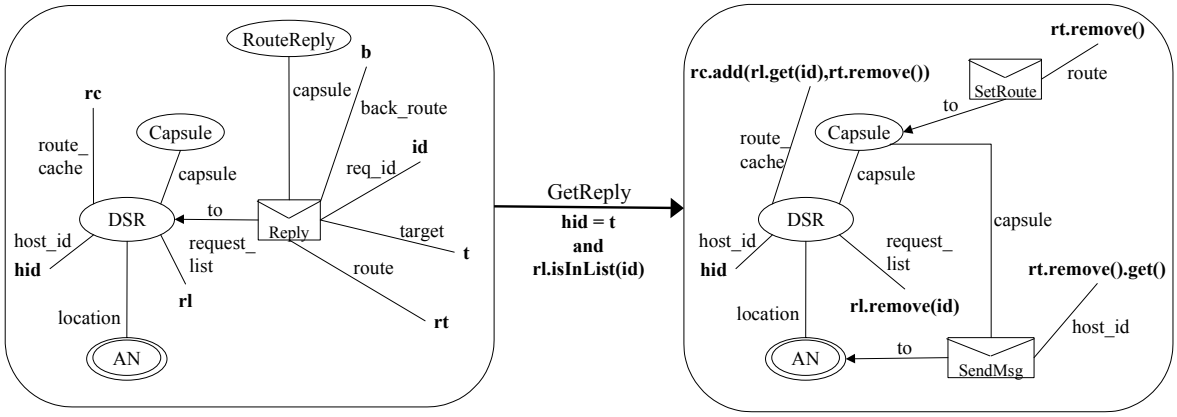

Figura 96. Regra *GetReply* do seviço DSR.

A retirada do registro da requisição da lista de requisições geradas pelo nodo local serve para que não haja sobreposição de rotas. Isto porque podem ser retornadas várias rotas para um mesmo destino. Segundo o algoritmo DSR, todas as rotas retornadas poderiam ser armazenadas e, em caso de falha em uma rota, poder-se-ia tentar o envio por outras. Como, nesta especificação, não se está considerando o caso de falha de rota, armazena-se apenas a primeira rota recebida, assumindo-se que esta rota apresenta o menor caminho do nodo origem ao destino. Assim, demais rotas que cheguem como resposta devem ser ignoradas. A regra *IgnoreReply*, na Figura 97, apresenta o tratamento de uma requisição já atendida.

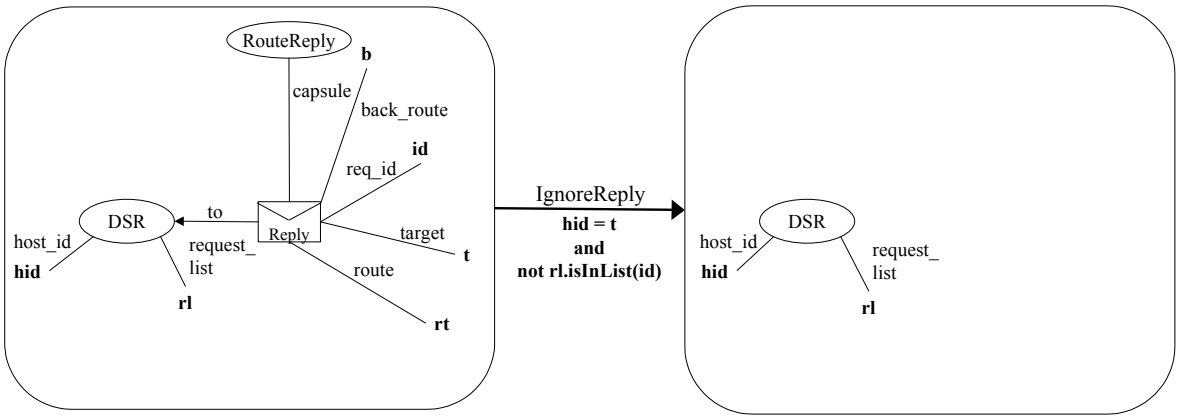

Figura 97. Regra *IgnoreReply* do serviço DSR.

Se o serviço DSR recebe uma cápsula *RouteReply* que não é destinada ao nodo local, ele realiza o encaminhamento desta para o próximo nodo de sua rota. O encaminhamento da cápsula *RouteReply* ocorre conforme a regra *ForwardReply*, mostrada na Figura 98.

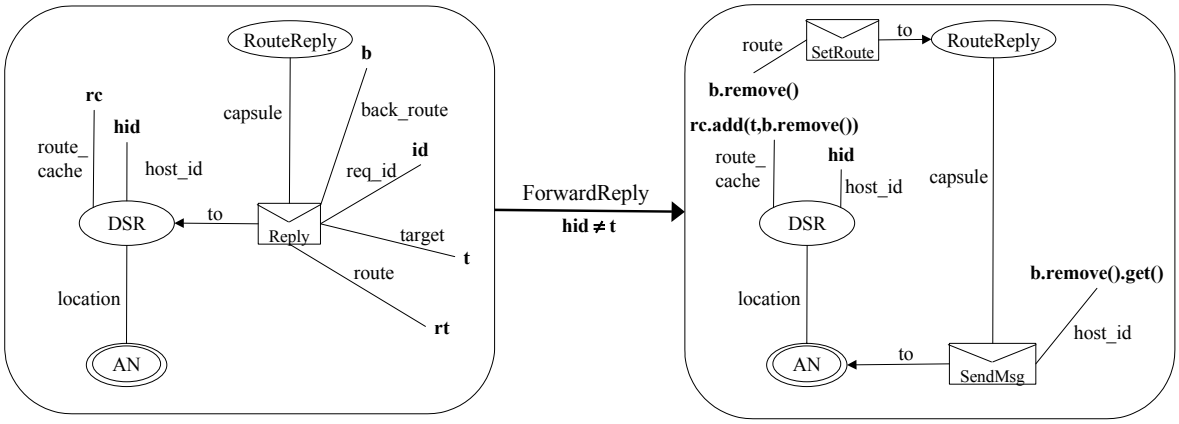

Figura 98. Regra *ForwardReply* do serviço DSR.

Aproveitando a informação de rota que a cápsula carrega, o serviço também armazena a rota do nodo local até o nodo de origem em sua CR.

7.2.2 Especificação da Cápsula de Requisição de Rota

A cápsula *RouteRequest* representa um pacote de requisição de rota, contendo, basicamente, as mesmas informações contidas em um pacote de requisição de rota usado no algoritmo DSR. O grafo de tipos da cápsula *RouteRequest* é apresentado na Figura 99.

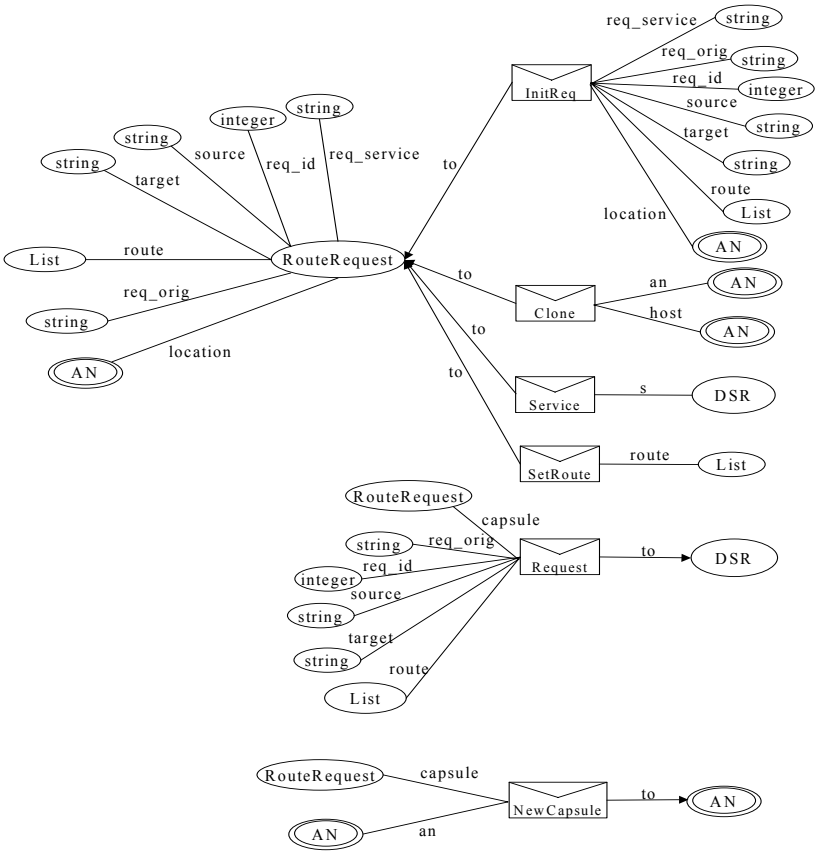

Figura 99. Grafo de tipos da cápsula *RouteRequest*.

Uma cápsula *RouteRequest* carrega consigo a identificação do nodo que originou a requisição (atributo *req_orig*), a identificação da requisição (atributo *req_id*) e a lista de nodos visitados (atributo *route*). Sendo uma cápsula, ela também possui a informação do serviço necessário para tratá-la (atributo *req_service*), uma referência ao nodo onde ela está (atributo *location*). O atributo *target* identifica o nodo procurado.

A cápsula *RouteRequest* é inicializada através da mensagem *InitReq*, como mostra a regra *InitRequest*, na Figura 100.

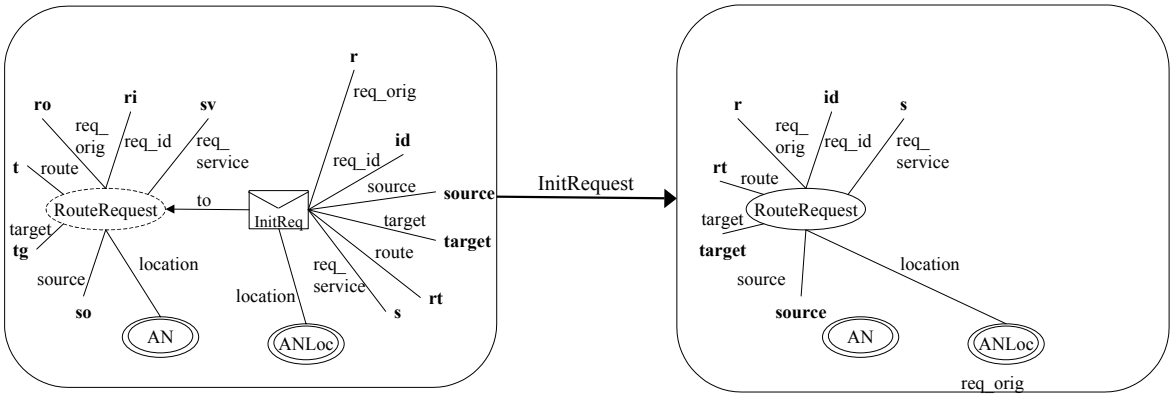

Figura 100. Regra de inicialização da cápsula *RouteRequest*.

A duplicação da cápsula *RouteRequest* é feita segundo a regra *CloneRequest*, apresentada na Figura 101.

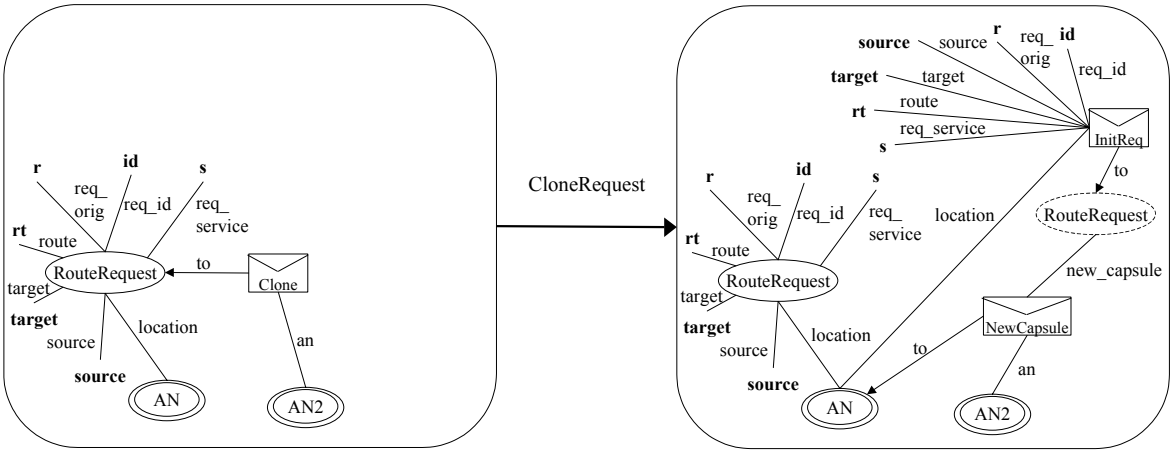

Figura 101. Regra de duplicação da cápsula *RouteRequest*.

A regra *InformRequest* descreve a interação entre a cápsula e o serviço DSR. A cápsula, seguindo esquema de regra apresentado na Figura 66 na Seção 6.2.2, ao receber a mensagem *Service*, utiliza a referência ao serviço recebida para enviar uma mensagem a este. A mensagem enviada é *Request*, contendo os dados necessários ao trabalho do serviço DSR. Esta regra é apresentada na Figura 102.

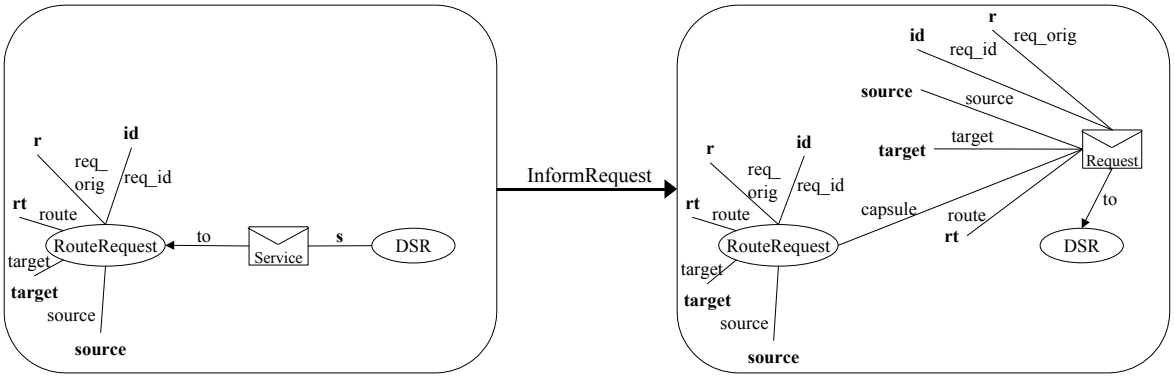

Figura 102. Regra *InformRequest* da cápsula *RouteRequest*.

A última regra da cápsula *RouteRequest* diz respeito à alteração do atributo que armazena a rota a ser seguida pela cápsula. Esta alteração ocorre segundo a regra *ChangeRoute*, ilustrada na Figura 103.

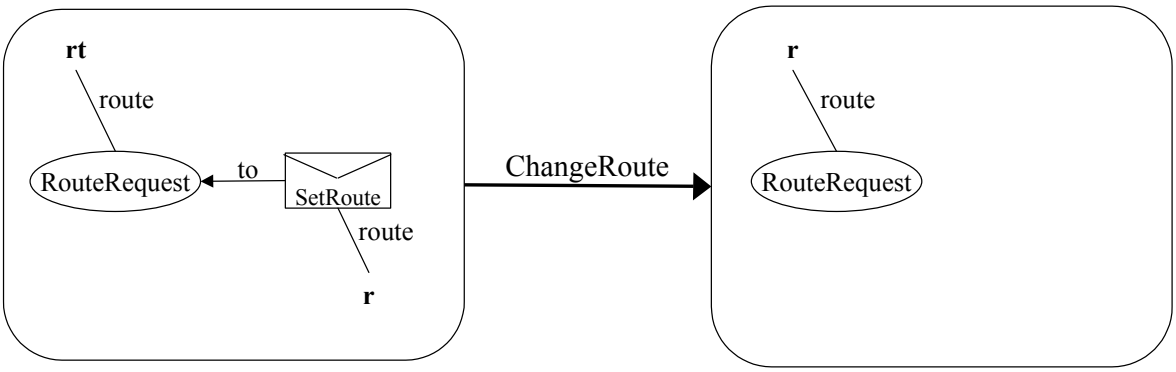

Figura 103. Regra *ChangeRoute* da cápsula *RouteRequest*.

A alteração da informação de rota da cápsula é usada para que o serviço DSR possa encaminhar a cápsula e retirar da lista de nodos os nodos já visitados.

7.2.3 Especificação da Cápsula de Resposta a uma Requisição de Rota

A cápsula *RouteReply* representa um pacote de resposta a uma requisição de rota. O grafo de tipos da cápsula *RouteReply* é apresentado na Figura 104.

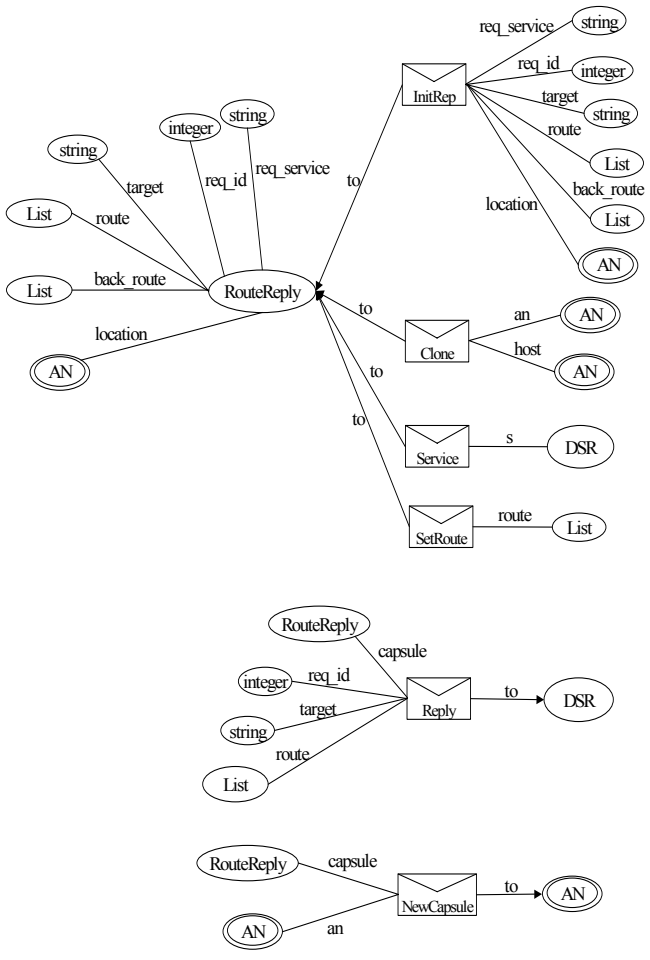

Figura 104. Grafo de tipos a cápsula *RouteReply*.

Uma cápsula *RouteReply* contém a identificação do serviço necessário para tratá-la (atributo *req_service*), a identificação da requisição à qual corresponde esta resposta (atributo *req_id*), a rota encontrada para o destino (atributo *route*) e o destino da resposta (atributo *target*). O atributo *back_route* contém a rota a ser seguida pela cápsula até o nodo que requisitou a rota.

A inicialização de uma cápsula *RouteReply* ocorre conforme a regra *InitReply* e a sua duplicação é descrita pela regra *CloneReply*, apresentadas, respectivamente, na Figura 105 e na Figura 106.

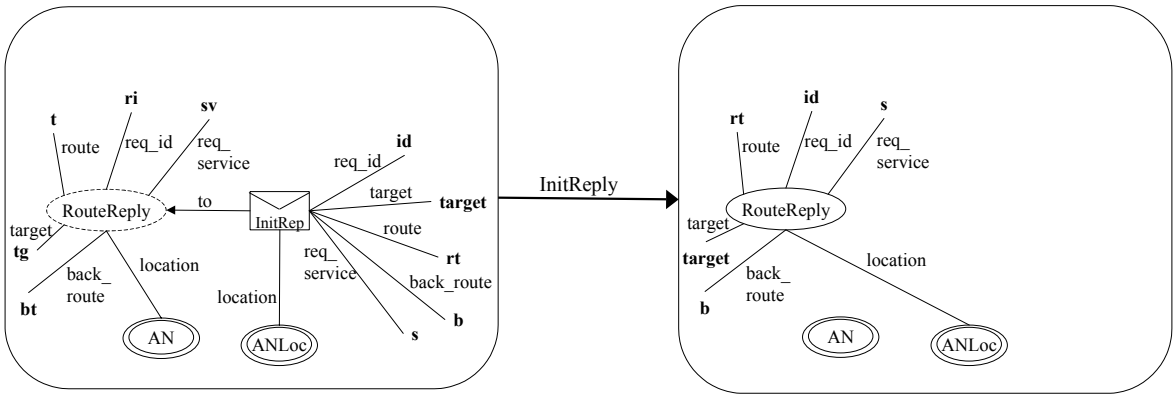

Figura 105. Regra de inicialização da cápsula *RouteReply*.

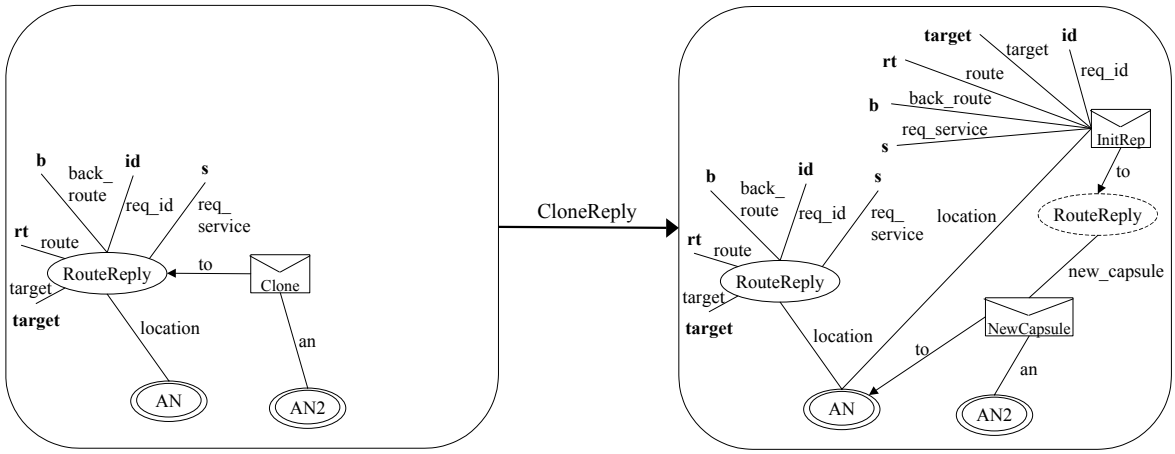

Figura 106. Regra de duplicação da cápsula *RouteReply*.

O envio das informações da cápsula para o serviço DSR dos nodos por que ela passa é feita através da mensagem *Reply*. Essa interação entre a cápsula e o serviço é feita segundo a regra *InformReply*, mostrada na Figura 107.

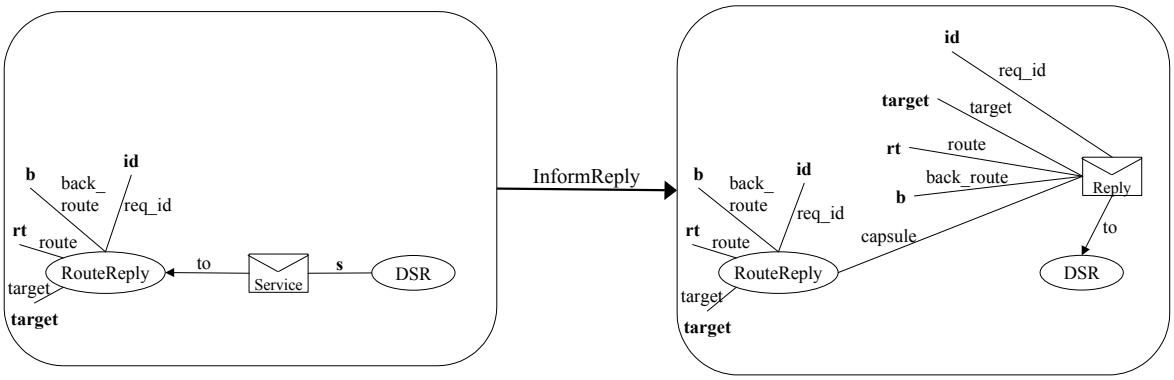

Figura 107. Regra *InformReply* da cápsula *RouteReply*.

A alteração da rota a ser seguida pela cápsula é feita de acordo com a regra *ChangeRoute*, apresentada na Figura 108.

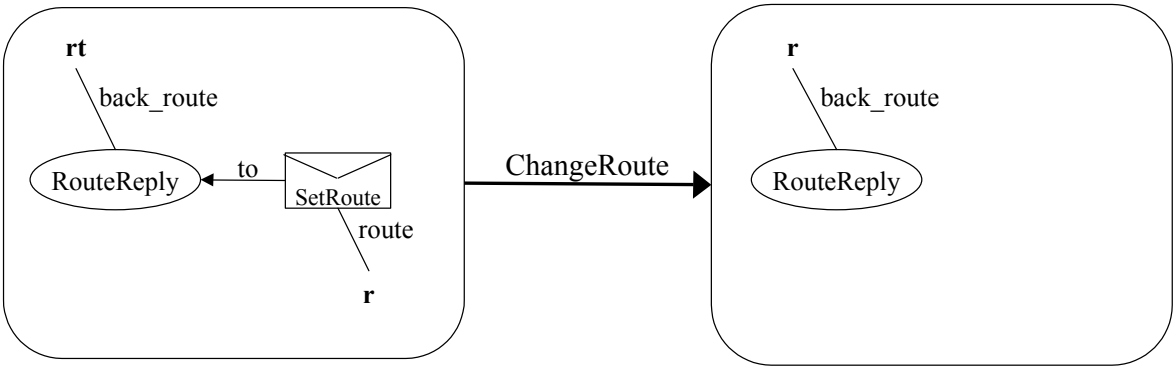

Figura 108. Regra *ChangeRoute* da cápsula *RouteReply*.

Note-se que a alteração, neste caso, é feita no atributo *back_route*. Isto porque ele é que contém a rota de retorno da resposta.

7.2.4 Especificação da Cápsula de Dados

O pacote de dados é representado pela cápsula *Packet*, que é a cápsula de dados. As cápsulas contendo dados da aplicação devem estender esta cápsula para poderem utilizar o algoritmo DSR. Assim, o comportamento necessário para interagir com o serviço DSR está definido na cápsula *Packet* e a cápsula que a estende acrescenta o comportamento relativo à aplicação da qual participa. A Figura 109 apresenta o grafo de tipos da cápsula *Packet*.

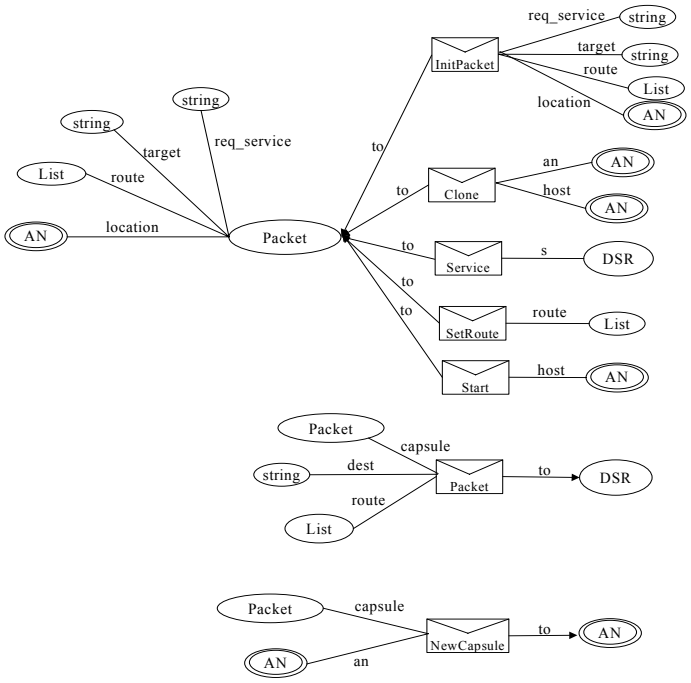

Figura 109. Grafo de tipos da cápsula *Packet*.

Como cápsulas de outros tipos deverão estender a cápsula *Packet*, a inicialização e a duplicação desta cápsula são apresentadas como esquemas de regras. O esquema de regra de inicialização de uma cápsula *Packet* é apresentado na Figura 110. A Figura 111 apresenta o esquema de regra de duplicação de *Packet*.

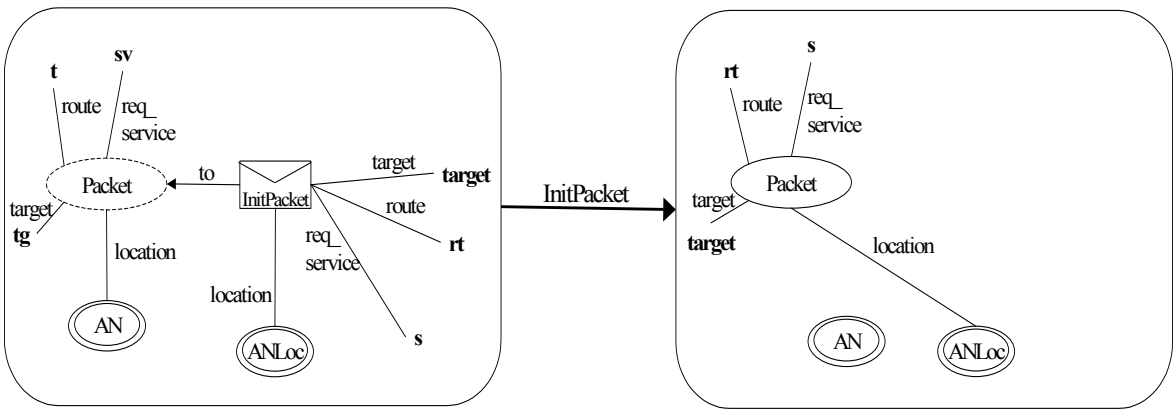

Figura 110. Esquema de regra de inicialização da cápsula *Packet*.

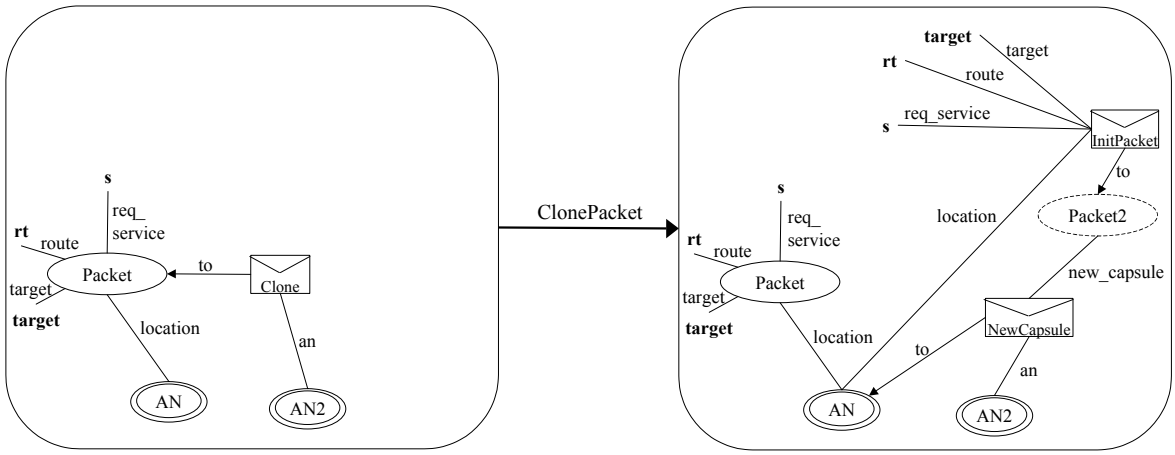

Figura 111. Esquema de regra de duplicação da cápsula *Packet*.

A alteração da informação de rota da cápsula *Packet* é feita segundo a regra *ChangeRoute*, apresentada na Figura 112.

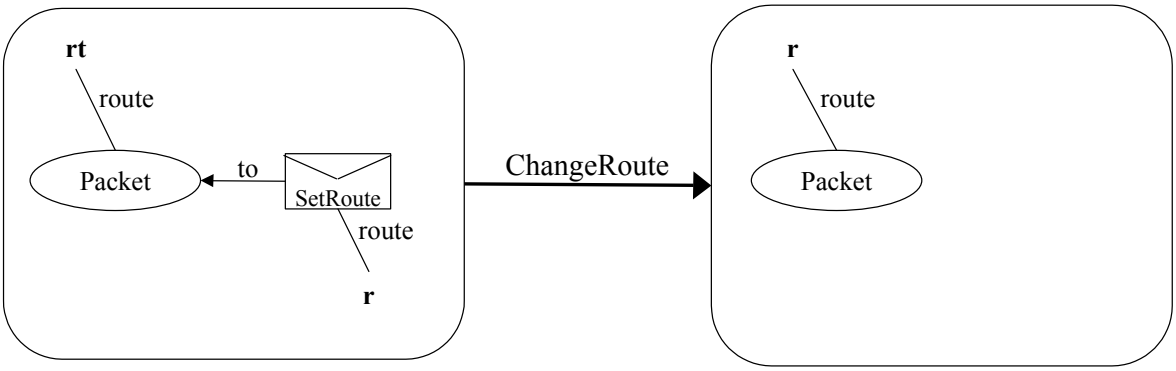

Figura 112. Regra *ChangeRoute* da cápsula *Packet*.

Ao chegar a um nodo a cápsula *Packet* envia seus dados para o serviço DSR local segundo a regra *InformPacket*, apresentada na Figura 113.

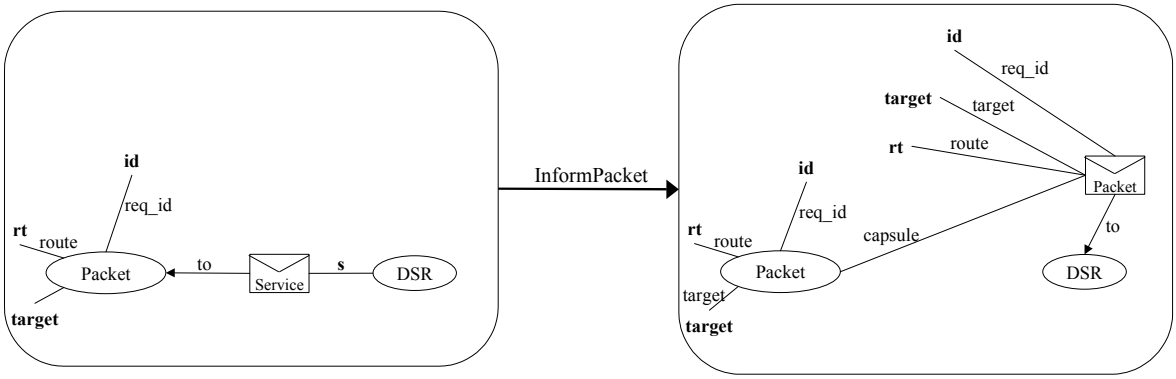

Figura 113. Regra *InformPacket* da cápsula *Packet*.

As cápsulas de tipos que estendem a cápsula *Packet* devem prover uma regra que trate a mensagem *Start*. Esta mensagem é enviada pelo serviço DSR quando a cápsula chegou ao destino. A regra a ser provida deve seguir o esquema de regra *BackTo Application*, apresentado na Figura 114.

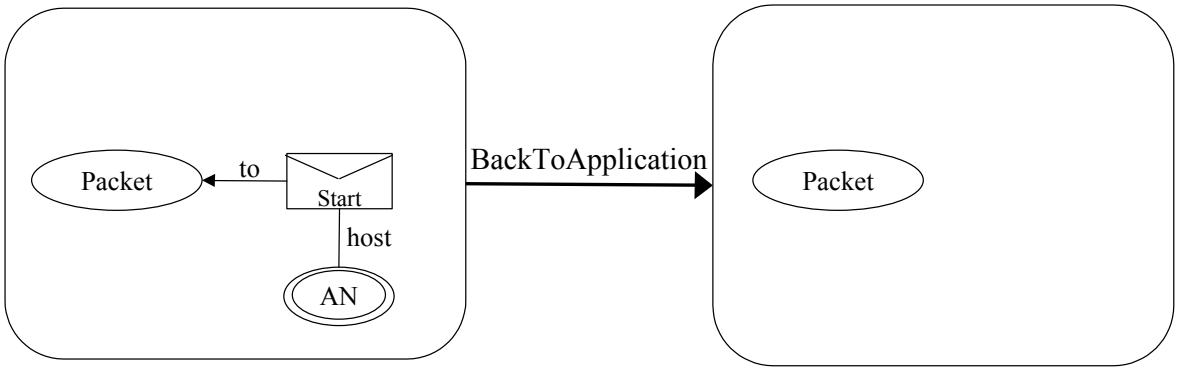

Figura 114. Esquema de regra *BackToApplication* da cápsula *Packet*.

A mensagem *Start* tem, como parâmetro, a referência ao nodo local e serve para identificar que a cápsula foi corretamente roteada e a aplicação pode continuar.

7.3 Código Gerado para o Cenário do Estudo de Caso

A partir das especificações de redes ativas (Seção 6.2) e do algoritmo DSR (Seção 7.2), foram gerados os códigos para simulação e para execução com mobilidade das entidades e regras envolvidas. Testes foram realizados sobre a arquitetura de redes ativas e, após considerar-se que as funções da rede ativa eram executadas corretamente, realizaram-se testes com o algoritmo DSR executando sobre essa arquitetura. Para realizar tais testes, foi criada uma topologia de rede de exemplo. Nesta topologia, todos os nodos eram nodos ativos, de acordo com a arquitetura de redes ativas proposta. A topologia envolvia 10 nodos, onde cada nodo possuía um servidor de nomes local. No servidor de nomes local encontravam-se as identificações dos nodos conhecidos pelo nodo local. Esta topologia é ilustrada na Figura 115.

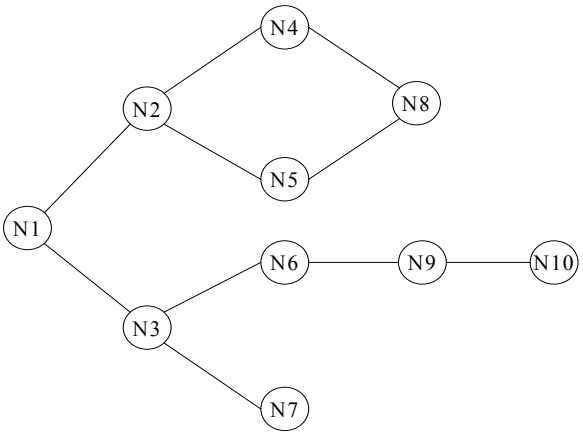

Figura 115. Topologia da rede usada para teste.

Além dos 10 nodos envolvidos na topologia apresentada, foi criado ainda um nodo *N0*, onde se localiza a base de código da rede. Este nodo foi criado apenas para alojar a base de código, de forma a não participar da troca de mensagens envolvidas no algoritmo DSR. Assim, este nodo serve para tratar exclusivamente de requisições para a base de código. Também está presente nesse nodo o servidor de nome global, onde está a lista de todos os serviços disponíveis na rede. No caso dessa rede, o único serviço disponível é o serviço DSR. Uma instância desse serviço é criada, no início do sistema, no nodo *N0*. Os demais nodos não possuem qualquer serviço executando quando iniciados.

Para os testes, criou-se uma cápsula de dados (cápsula *Packet*) a ser roteada a partir do nodo *N1*. Foram realizados os roteamentos para todos os demais nodos da rede. A topologia concebida permitiu testarem-se as principais situações do algoritmo DSR, tal como recebimento de requisições repetidas, recebimento de múltiplas rotas para um mesmo destino, tratamento de respostas a requisições de rotas, encaminhamento de pacotes, entre outras. Outro teste realizado envolveu a inicialização da *cache* de rotas de alguns nodos com algumas rotas iniciais. Isto permitia testar se o nodo realmente encaminhava uma rota conhecida quando recebia uma requisição que pedia esta rota.

Os testes foram realizados, inicialmente, no simulador. A realização de simulações permitiu a correção de erros de especificação. Após isso, passou-se para a execução real, com todos os nodos executando em uma mesma máquina. O comportamento assumido pela aplicação na execução real mostrou-se coerente em relação ao comportamento que fora obtido na simulação.

Testes posteriores foram realizados com os nodos distribuídos em 4 máquinas diferentes e com múltiplas cápsulas sendo roteadas. Os resultados corresponderam ao

esperado, isto é, as cápsulas de dados foram corretamente roteadas e os nodos comportaram-se conforme o especificado e conforme o que havia ocorrido na simulação.

7.4 Conclusões sobre os Estudos de Caso

A realização dos estudos de caso serviu para a avaliação do mapeamento criado e também para o teste do uso da LEF e do simulador. No que diz respeito à LEF, tipos abstratos de dados devem ser definidos, para utilização em GGBO, através de especificação algébrica. Por simplicidade e pela dificuldade que apresenta trabalhar-se com especificação algébrica para pessoas não acostumadas com tal linguagem, utilizou-se uma descrição informal de tipos abstratos de dados, assumindo-se a correção das operações realizadas sobre eles. Posteriormente, este tipos devem ser formalmente descritos para poderem ser utilizados corretamente dentro das especificações. Ainda não foi estabelecida uma forma de representar a inicialização de tipos abstratos de dados.

Uma dificuldade encontrada em relação à LEF foi a inexistência de uma forma de representação de hierarquia de herança de entidades. Com isso, não é possível criarem-se entidades a partir de uma entidade previamente especificada. Assim, deve-se repetir todas as definições necessárias dentro da especificação de uma nova entidade, mesmo que ela possua atributos e regras já definidos em outra entidade. Neste trabalho, como forma de tornar mais simples a especificação, estabeleceu-se uma forma de hierarquia, em que se definiu que entidades podiam "basear-se" em outras entidades quando se desejasse agregar o comportamento destas últimas a um comportamento específico das primeiras. Deve, portanto, ficar claro que isto foi um mecanismo adotado neste trabalho para simplificar a elaboração e a apresentação de entidades, mas que não existe hierarquia de herança de entidades em GGBO, sendo que uma especificação completa exigiria a repetição de definições comuns entre entidades. A inclusão e uma forma de representação de hierarquia de herança de entidades em GGBO está em estudo dentro do grupo do projeto ForMOS.

Além dos problemas citados, também se deve relembrar de uma questão discutida durante a explanação sobre a especificação dos estudos de caso que é a dificuldade em representar computações seqüenciais. Isto leva à utilização de atributos ou de parâmetros de mensagens que servem tão somente para armazenar referências ou valores. Além do espaço de armazenamento gasto para guardar estes valores ou referências e do custo de sua transmissão, às vezes, em múltiplas mensagens até que sejam realmente utilizados, existe também o acréscimo de complexidade no entendimento da especificação devido à poluição

visual e por ferir regras de encapsulamento. Considera-se a possibilidade da causa de tais problemas estar no uso incorreto da LEF, uma vez que se pode estar seqüencializando processamentos que poderiam ter sido especificados de forma a serem realizados paralelamente. Pode-se, ainda, discutir-se a validade de utilização da LEF para o propósito que foi colocado em relação ao que ela oferece.

Apesar disso tudo, a LEF mostrou ser, de certa forma, intuitiva, mesmo que, por vezes, fique difícil o seu entendimento. Ela fornece uma forma gráfica de representar o sistema, o que facilita o entendimento da especificação, além de ser baseada em objetos, o que adiciona à especificação conceitos bastante difundidos. A representação de mobilidade é simples e a compreensão da evolução do sistema não exige muito tempo de estudo da linguagem. Tem-se discutido formas de tornar especificações em GGBO mais claras, tais como a padronização do posicionamento de seus elementos.

Quanto ao simulador, ele provou ser bastante útil para o teste das especificações. Diversos erros não identificados na etapa de especificação foram notados quando se realizaram simulações sobre a especificação resultante. A correção destes erros tornou possível que, após mapear a especificação para código executável, o sistema executasse conforme o esperado e conforme o resultado da simulação. Assim mesmo, avalia-se que melhorias podem ser realizadas no simulador para que este se torne uma ferramenta ainda mais valiosa. Uma destas melhorias seria permitir a execução paralela de regras de escrita e leitura, o que ainda não é permitido pela implementação atual do simulador (vide Capítulo 4). Isto permitiria o aumento nas possibilidades de paralelismo dentro de uma entidade, segundo o que é definido em GGBO.

O comportamento do código executável demonstrou estar coerente com o comportamento do código simulado. A arquitetura de redes ativas mostrou funcionar corretamente, bem como o algoritmo DSR. Considera-se que este estudo de caso contribuiu com a proposta de uma arquitetura para redes ativas baseada no uso de componentes móveis, tirando proveito de suas potencialidades. A implementação do algoritmo DSR sobre esta arquitetura tornou possível testar a coerência do código gerado para a arquitetura de redes ativas em relação à especificação feita, bem como do próprio algoritmo DSR. Também cabe ressaltar que se desconhece outra implementação do algoritmo DSR com o uso de mobilidade. Dessa forma, apesar das restrições feitas em relação à definição original do algoritmo, considera-se esta implementação como uma contribuição que demonstra mais uma área de aproveitamento das características de

mobilidade de código. Não é possível determinar, por agora, quais as vantagens e desvantagens do uso de mobilidade para algoritmos de roteamento, principalmente para redes *ad hoc*, mas isso abre a possibilidade de consideração dessa idéia.

8 Trabalhos Relacionados

Muitos têm sido os trabalhos na área de modelagem e especificação de sistemas que envolvem o uso de mobilidade de código. Entre eles estão JAMES, Mobile UNITY, Cálculo-π, Ambit, MobiS e SCD, os quais são apresentados a seguir. Ao final do capítulo, são feitos alguns comentários sobre os trabalhos apresentados.

8.1 JAMES

JAMES (*Java-Based Agent Modeling Environment for Simulation*) [UHR00] é uma ferramenta voltada à realização de simulações de sistemas de Inteligência Artificial cujos componentes são agentes. Em JAMES a simulação ocorre através da troca de mensagens entre entidades concorrentes e distribuídas. É provido um ambiente de modelagem e simulação que foi criado tendo como principal foco o suporte a experimentos com programas de Inteligência Artificial, tais como agentes, os quais podem ser móveis.

8.2 Cálculo-π

O Cálculo-π [MIL91] [MIL99a] é um modelo de computação concorrente que tem como base o Cálculo de Processos ou *Calculus of Communicating Systems (CCS)* [MIL80]. Em CCS, têm-se processos que executam concorrentemente e comunicam-se através de canais estaticamente criados. O Cálculo-π acrescentou a isso a possibilidade de criação, deleção e alteração dinâmica de canais, permitindo que as configurações de canais entre processos possam mudar ao longo do tempo.

Cálculo-π baseia-se na noção de nomeação (*naming*). Nomes são as entidades mais primitivas do Cálculo-π e servem para a criação de processos, além de permitir sua localização e comunicação. Assim, quando dois processos possuem o nome de um mesmo canal, eles podem se comunicar através deste canal. A interação entre processos ocorre de forma síncrona, seguindo o modelo de comunicação *rendezvous*, no qual tanto quem envia quanto quem recebe fica bloqueado até a comunicação ser finalizada.

Canais podem transmitir outros canais e também processos. Dessa forma, a idéia de mobilidade baseia-se na estruturação de processos com capacidade de serem transmitidos de um local para outro através de canais. Quando um processo é movido a sua

configuração de canais se modifica. Com isso, a movimentação de um processo pode ser representada pela mudança na configuração de seus canais.

Existem hoje algumas linguagens que seguem a semântica de Cálculo-π, tais como KLAIM [NIC98], Pict [PIE97] e Nomadic Pict [WOJ99]. Estas linguagens incorporam a semântica do modelo de Cálculo-π, provendo uma programação baseada em processos. Mais sobre estas linguagens é apresentado em [DUA00b].

8.3 Mobile UNITY

Mobile UNITY [ROM98] [PIC97] é baseada no modelo UNITY, proposto em [CHA88], e provê uma notação para componentes e uma linguagem para expressar a interação entre esses componentes. Cada componente é definido por um programa UNITY, o qual é composto por um conjunto de variáveis que formam seu estado, predicados iniciais relacionados ao seu estado e uma lista de comandos que alteram seu estado e definem o comportamento do componente. Um sistema é descrito por um conjunto de componentes e as interações entre esses componentes. Como todas as variáveis de um componente Mobile UNITY são consideradas locais a este componente, nenhuma comunicação pode ocorrer sem que haja uma definição de interação na Seção *Interactions* de um sistema, a qual serve para prover comunicação entre componentes. Cada definição de interação apresenta como a interação ocorre (quais informações são trocadas) e apresenta condições de ocorrência da interação. Dessa forma, uma interação só ocorre quando a sua condição for satisfeita. A linguagem de especificação de Mobile UNITY é textual e lembra a linguagem CSP [HOA85]. A noção de localização é dada por um atributo do componente, sendo que a ocorrência de uma movimentação é descrita como uma alteração neste atributo.

8.4 Ambit

Ambit [CAR98] possui a idéia de *ambientes*. Um ambiente é um componente que possui uma delimitação física ou lógica e onde a computação ocorre. A delimitação de um ambiente define o escopo de atuação do ambiente. Exemplos de ambientes são um objeto (delimitado pelo seu conteúdo) e um *laptop* (delimitado por seu console e portas de dados). Ou seja, qualquer lugar que possa conter algum tipo de computação e possua uma delimitação definida, é definido como um ambiente. A delimitação de um ambiente

determina o que está dentro e o que está fora do ambiente. Um ambiente pode conter outros ambientes e cada ambiente possui um conjunto de agentes locais, que são as computações que executam dentro do ambiente e, de alguma forma, controlam o ambiente, fazendo com que o ambiente se mova, por exemplo. Ao se mover, o ambiente move-se como um todo. Além de ambientes, agentes também se movem. Uma movimentação pressupõe a ultrapassagem das delimitações do ambiente onde se está e a entrada nas delimitações de um outro ambiente. A interação entre componentes se dá quando os componentes comunicantes compartilham o mesmo ambiente, estando dentro das mesmas delimitações. A linguagem de especificação é textual e é chamada de Cálculo de Ambientes.

8.5 MobiS

MobiS [MAS99] é uma linguagem de especificação baseada em múltiplos espaços de tuplas que estende a linguagem PoliS [CIA98]. Especificações em MobiS são hierarquicamente estruturadas, denotadas por uma árvore de espaços aninhados que se modifica dinamicamente. Espaços podem se mover, modificando sua posição na árvore. Um espaço MobiS (representação de um sistema) pode conter tuplas ordinárias, que são seqüências ordenadas de valores, tuplas de programa, que representam códigos, e um espaço de tuplas. Tuplas de programas podem modificar um espaço removendo e adicionando tuplas e outros espaços. A área de atuação das modificações de uma tupla de programa se restringe ao espaço onde ela se encontra e ao espaço acima deste na árvore de espaços. Uma tupla de programa é definida por uma regra que determina quais ações são tomadas. Uma regra lê tuplas dentro do seu escopo de atuação, realiza uma computação seqüencial e produz novas tuplas. Segundo o formalismo, uma regra consiste de uma pré-ativação, que define quais tuplas devem estar no escopo da regra para que ela seja aplicada, uma computação local, que define uma computação seqüencial que não altera o estado do espaço de tuplas, e uma pós-ativação, que define as tuplas produzidas pela regra dentro do seu escopo de atuação. Espaços podem se mover, levando consigo toda a subárvore que está abaixo dele. Ou seja, todos os espaços que estão, dentro da árvore do sistema, abaixo do espaço que se move e ligados a ele (espaços ditos *filhos*), movem-se juntamente com o espaço *pai*. A mobilidade é representada pelo consumo e produção de espaços. Como o escopo de atuação de uma regra é local ao espaço em que ela se encontra e ao espaço *pai*, a movimentação ocorre de forma “passo-a-passo”, movendo-se um espaço de um lugar na

árvore para o nível mais acima, até que ele chegue ao topo da árvore e passe a ser movido para baixo, até chegar no local de destino de sua movimentação dentro da árvore.

8.6 SCD

Em relação à geração de código a partir de uma especificação formal, existe um trabalho descrito em [YOO98] que relata a idéia de realização de geração automática de código de uma especificação formal de agentes móveis. Este trabalho utiliza uma variante de Redes de Petri [PET73] para modelar o sistema, o qual é descrito por componentes especificados em uma linguagem de descrição chamada *Soft-Component Description (SCD) Language*. Esta linguagem permite a modelagem do comportamento de um componente pela definição de classes de componentes e possível composição de instâncias de componentes. Esta linguagem é textual e o modelo descrito em SCD é mapeado para código de uma plataforma de suporte à mobilidade denominada *JavaNet Agents (JNA)*. Para este mapeamento, foi construído um compilador que gera código Java a partir do modelo em SCD. Este código é então encapsulado em agentes JNA. É ainda realizada a simulação do sistema modelado através do uso de uma ferramenta de simulação e análise sobre Redes de Petri.

8.7 Comentários sobre os Trabalhos Relacionados

Como visto, muitos trabalhos envolvem a utilização de métodos formais para descrever sistemas com componentes móveis e para criação de modelos de simulação, como o JAMES. Apesar disso, poucos tratam da questão de geração de código a partir das especificações feitas. MobiS está em fase de integração com a modelagem UML [FOW00], o que pode ser um caminho para a geração de código, aproveitando a proximidade de UML com a implementação de sistemas. Além disso, as especificações são feitas de modo textual e com um conjunto de operadores lógicos e matemáticos, exigindo um certo conhecimento dos mesmos pelo especificador. Isto os torna pouco acessíveis e pouco utilizáveis por pessoal leigo no assunto.

Os trabalhos baseados em Cálculo-π fornecem uma forma de mapeamento da semântica deste formalismo para uma linguagem de programação. Uma possível dificuldade em trabalhar com estas linguagens seria realizar a programação baseada em processos, visto que, ultimamente, utiliza-se o paradigma de orientação a objetos.

SCD é o que possui maior proximidade com o trabalho descrito neste relatório, com as desvantagens de a linguagem formal utilizada ser textual e pouco intuitiva. Além disso, o código é portado para uma plataforma pouco difundida. Até por isso, segundo os autores, pretende-se gerar código para a plataforma Aglets da IBM [LAN97], que é bastante utilizada para o desenvolvimento de aplicações com código móvel.

9 Conclusões

O trabalho aqui apresentado visou a geração de código para uma linguagem de programação, a ser executado sobre uma plataforma de suporte à mobilidade, a partir de uma especificação formal. Especificações formais feitas em GGBO foram mapeadas para código, o qual foi executado sobre a plataforma Voyager. O mapeamento realizado baseou-se no mapeamento prévio criado no ambiente do simulador PLATUS para gerar código para simulação de especificações em GGBO. Com isso, tornou-se possível especificarem-se sistemas, em especial SDCM, gerar-se código de simulação para estes sistemas, simulá-los no ambiente PLATUS e, com poucas modificações, gerar-se código para executar sobre a plataforma Voyager.

A característica definida no projeto ForMOS, no qual este trabalho se insere, de se trabalhar com especificação, simulação e geração de código tendo como base um formalismo único, fornece ao projeto uma abordagem diferenciada de outros trabalhos correlatos. Tal abordagem facilita o trabalho de quem venha a utilizar as ferramentas resultantes do desenvolvimento deste projeto, visto que é preciso apenas conhecer-se o formalismo de GGBO uma vez que os mapeamentos para código de simulação e para execução real estão estabelecidos. O futuro uso de técnicas de verificação sobre GGBO deverá ainda contribuir para o desenvolvimento de sistemas mais confiáveis, minimizando a ocorrência de erros.

Considera-se que o objetivo principal do trabalho foi alcançado, visto que se chegou a um mapeamento de GGBO para código que executa sobre uma plataforma. A correção do mapeamento utilizado neste trabalho, o qual permite a geração de código a partir de especificações em GGBO, ainda não está formalmente provada. Como dito anteriormente, assumiu-se aqui a correção do mapeamento criado para o simulador PLATUS. Com isso, o que se fez foi buscar garantir que as características apresentadas pelo ambiente de simulação fossem preservadas no código para execução real.

Embora ainda não provado formalmente, os testes realizados e o estudo de caso desenvolvido indicaram a coerência do código gerado em relação ao código de simulação. Pôde-se constatar, através de testes de diversas situações e dos estudos de caso, que o código para execução real preserva, através dos mecanismos criados para a movimentação de entidades e para a comunicação entre entidades e das funcionalidades fornecidas pela plataforma utilizada, as características apresentadas na simulação. Por conseqüência, pode-

se considerar que, caso o código gerado para o simulador esteja correto em relação à especificação em GGBO (o que ainda também exigirá uma prova formal), o código gerado pelo mapeamento apresentado neste trabalho também estará correto. Também deve-se dizer que, segundo observado durante a realização dos testes e dos estudos de caso, o código gerado e executado segue o mesmo comportamento definido na especificação em GGBO.

Durante o desenvolvimento deste trabalho, além da contribuição com a geração de código para execução, também se contribuiu para o aperfeiçoamento da LEF e do simulador. Usando a LEF e o simulador para o desenvolvimento dos testes do mapeamento e para a realização do estudo de caso foi possível verificar-se a sua utilidade e as suas vantagens e desvantagens. Através dessa análise, identificaram-se problemas e deficiências na LEF utilizada e problemas em relação ao simulador puderam ser corrigidos ou possuem uma solução em estudo. Todos os problemas identificados foram levados para discussão com o grupo do projeto e servirão para melhorar tanto a linguagem quanto o simulador e, conseqüentemente, melhorar o código gerado.

Acerca da pesquisa desenvolvida no projeto ForMOS, visualizam-se alguns trabalhos futuros possíveis, além das considerações já feitas na Seção 7.4 sobre a LEF e o simulador. O primeiro deles seria o desenvolvimento de um tradutor que realizaria a geração automática de código a partir de uma especificação em GGBO. Acredita-se que seja possível gerar-se código de simulação automaticamente a partir de uma especificação em GGBO, visto que em [DIA01] já foi apresentado um trabalho nesse sentido. Conforme discutido no Capítulo 5, considera-se que seria também possível gerar-se código de execução de forma automatizada.

Para a criação do tradutor faz-se necessária a implementação de um editor de especificações. Neste editor seria possível criar, de maneira gráfica, especificações em GGBO. As especificações criadas seriam armazenadas em estruturas de dados a partir das quais o tradutor pudesse obter as informações necessárias para a geração de código. Um protótipo desse editor está já em desenvolvimento.

Outro trabalho futuro poderia considerar a utilização de outras plataformas de suporte à mobilidade. Isto possibilitaria a comprovação de que o mapeamento criado pode ser independente de plataforma e também criaria opções para o desenvolvimento de SDCM sobre outras plataformas que não a Voyager, talvez com melhor desempenho.

Também existe um possível trabalho quanto à consideração de falhas. Assim, poderiam ser representados modelos de falhas em lugares e em componentes móveis, incluindo o comportamento dinâmico de um ambiente aberto real. Isto permitiria a criação de aplicações móveis tolerantes a falhas ou a especificação e simulação de mecanismos de detecção e tolerância de falhas.

10 Referências Bibliográficas

[AST97] ASTESIANO, E., REGGIO, G. Formalism and Method. In Proc. Of TAPSOFT'97: Theory and Practice of Software Development, *Lecture Notes in Computer Science*, v. 1214, Springer, Lille, France, 1997. p. 93-114.

[BOY96] BOYER, R. S., YU, Y. Automated Proofs of Object Code for a Widely Used Microprocessor. *Communications of ACM*, v. 43, n. 1, 1996. p. 166-192.

[BRI95] BRINK, K., KATWIJK, J. V., TOETENEL, H. Applying Formal Software Requirements Specification in the Development of Control Applications. In *Proceedings of the First Annual Conference of the Advanced School for Computing and Imaging*, Heijen, Netherlands, 1995. p.11-17.

[CAI96] CAI, T., GLOOR, P., NOG, S. Dataflow: A Workflow Management System On The Web Using Transportable Agents. *Technical Report TR96-283, Dept. of Computer Science, Dartmouth College*, Hanover, NH, 1996.

[CAM99] CÂMARA, D., LOUREIRO, A. A. F. Roteamento em Redes Móveis Ad Hoc. In *I Workshop de Comunicação Sem Fio, Minicurso 4*, Belo Horizonte, Brasil, 1999, p. 14-15.

[CAR97] CARZANIGA, A., PICCO, G. P., VIGNA, G. Designing Distributed Applications with Mobile Code Paradigms. In *Proceedings of the 19th Int. Conference on Software Engineering (ICSE'97)*, ACM Press, 1997, p. 22-32.

[CAR98] CARDELLI, L., GORDON, A. D. Mobile Ambients. In *Proceedings of FoSSaCS'98, Lecture Notes in Computer Science*, n. 1378, Springer, 1998, p. 140-155.

[CHA88] CHANY, K., MISRA, J. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.

[CHO59] CHOMSKY, N. On Certain Formal Properties of Grammar. *Information and Control*, v. 2, n.2, 1959, p. 137-167.

[CIA98] CIANCARINI, P., FRANZÈ, F., MASCOLO, C. A Coordination Model to Specify Systems Including Mobile Agents. In *Proceedings of 9th IEEE International Workshop on Software Specification and Design (IWSSD9)*, Ise-shima, Japan. 1998. p. 96-105.

[CLA96] CLARKE, E., WING, J. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, v. 28, n. 4, 1996. p. 626-643.

[COP00] COPSTEIN, B., MÓRA, M. C., RIBEIRO, L. An Environment for Formal Modelling and Simulation of Control Systems. In *Proceedings of 33rd Annual Simulation Symposium*, SCS, 2000. p.74-82.

[COP01] COPSTEIN, B., MÓRA, M. C., RIBEIRO, L. Event Driven Simulation of Object-Based Graph Grammar Models. Submetido ao *34th Annual Simulation Symposium 2001*, Seattle, USA, 2001.

[DEC98] DECASPER, D., PLATTNER, B. DAN: Distributed Code Cashing For Active Networks. In *Proceedings of the IEEE INFOCOM'98*, San Francisco, USA, 1998.

[DEH00] DÈHARBE, D., MOREIRA, A. M., RIBEIRO, L., et al. Introdução a Métodos Formais: Especificação, Semântica e Verificação de Sistemas Concorrentes. *Revista de Informática Teórica e Aplicada*, v. 7, n. 1, Instituto de Informática - UFRGS, Porto Alegre, 2000. p. 7-48.

[DIA01] DIAS, L. L., OLIVEIRA, S. C. R. SIMSC-GG: Ferramenta de Simulação do Ambiente PLATUS. Trabalho de Conclusão, Curso de Bacharelado em

Informática, Faculdade de Informática, PUCRS, 2001, 133 f.

[DOT00a] DOTTI, F. L., DUARTE, L. M. Monitoring Mobile Code. In *Proceedings of Parallel and Distributed Processing Techniques and Applications 2000*, v. 4, USA, 2000. p. 2029-2034.

[DOT00b] DOTTI, F. L., RIBEIRO, L. Specification of Mobile Code Systems Using Graph Grammars. *Formal Methods for Open Object-Based Distributed Systems IV*, Kluwer Academic Publishers, Stanford, USA, 2000. p. 45-63.

[DOT99] DOTTI, F. L., NYGAARD, F., et al. Um Monitor de Objetos Móveis: Concepção, Arquitetura e Resultados Práticos. In *I Workshop de Comunicação Sem Fio 1999*, Anais do I Workshop, Belo Horizonte, Brasil, 1999. p. 113-122.

[DUA00a] DUARTE, L. M. Estudo de Linguagens de Programação com Suporte à Mobilidade de Código. *Relatório Técnico n. 016*, Programa de Pós-Graduação em Ciência da Computação – Mestrado, PUCRS, 2000. 68 f.

[DUA00b] DUARTE, L. M. Uma Análise de Linguagens de Especificação para Sistemas Distribuídos. *Relatório Técnico n. 015*, Programa de Pós-Graduação em Ciência da Computação – Mestrado, PUCRS, 2000. 51 f.

[EHR79] EHRIG, H. Introduction to the Algebraic Theory of Graph Grammars. In *1st Graph Grammar Workshop*, *Lecture Notes in Computer Science 73 (V.Claus, H. Ehrig and G. Rozenberg eds.)*, Springer Verlag, 1979, p. 1-69.

[FOW00] FOWLER, M., SCOTT, K. *UML Essencial: Um Breve Guia para a Linguagem Padrão de Modelagem de Objetos*. Segunda Edição, Porto Alegre: Bookman, 2000, 169 p.

[FRA94] FRASER, M. D., KUMAR, K., VAISHNAVI, V. K. Strategies for Incorporating Formal Specifications in Software Development. *Communications of ACM*, v. 37, n. 10, 1994. p. 74-86.

[FUG98] FUGGETA, A., PICCO, G. P., VIGNA, G. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, v. 24, 1998. p. 342-361.

[GOL95] GOLDSZMIDT, G., YEMINI, Y. Distributed Management by Delegation. In *Proceedings of 15th Int. Conf. On Distributed Computing*, 1995.

[GOS96] GOSLING, J., McGILTON, H. *The Java Language Environment - A White Paper*. SunMicrosystems, 1996. Disponível na Internet em http://java.sun.com/doc/language_environment/

[GRE97] GREEN, S., HURST, L, NANGLE, B., et al. *Software Agents: A Review*. Technical Report TCS-CS-1997-06, Trinity College Dublin, 1997, 48 p.

[GUN98] GUNTER, C. A., NETTLES, S. M., SMITH, J. M. The SwitchWare Active Network Architecture. *IEEE Network, Special Issue on Active and Programmable Networks*, v. 12, n. 3, 1998.

[HOA85] HOARE, C. A. R. *Communicating Sequential Processes*. Prentice Hall, 1985, 256 p.

[ITU00] ITU-T, ITU Recommendation Z.100. *The Specification and Description Language (SDL)*. ITU, Geneva, 2000.

[JAI00] JAIN, R., ANJUM, F., UMAR, A. A Comparison of Mobile Agent and Client-Server Paradigms for Information Retrieval Tasks in Virtual Enterprises. In *Proceedings of the Academia/Industry Working Conference on Research Challenges (AIWORC'00)*, 2000.

[JOH01] JOHSON, D. B., MALTZ, D. A., HU, Y., et al. The Dynamic Source

Routing Protocol for Mobile Ad Hoc Networks (Internet Draft), *MANET Working Group, IETF*, 2001, 60 p.

[KAR98] KARNIK, N. M., TRIPATHI, A. R. Design Issues in Mobile Agent Programming Systems. *IEEE Concurrency*, v. 6, n. 3, 1998, p. 52-61.

[KNA95] KNABE, F. C. Language Support for Mobile Agents. *Ph.D. thesis*, Carnegie Mellon University, 1995. Disponível na Internet em http://agents.umbc.edu/papers/knabe.shtml.

[KNA96] KNABE, F. C. An Overview of Mobile Agent Programming. *Proc. of the Fifth LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, number 1192 in Lecture Notes in Computer Science, Stockholm, Sweden, 1996.

[LAM94] LAMPORT, L. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, v. 16, n. 3, 1994. p. 872-923.

[LAN97] LANGE, D. B. *Java Aglet Application Programming Interface (J-AAPI) White Paper - Draft 2*. IBM Tokyo Research Laboratory, 1997.

[LER97] LEROY, X. *Objective Caml*. 1997. Disponível na Internet em http://pauillac.inria.fr/ocaml/

[LIM96] LIMONGIELLO, A., MELEN, R., ROCCUZZO, M., et al. ORCHESTRA: An Experimental Agent-Based Service Control Architecture For Broadband Multimedia Networks. *GLOBAL Internet'96*, 1996.

[MAG96a] MAGEDANZ, T., ECKARDT, T. Mobile Software Agents: A New Paradigm for Telecommunications Management. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Kyoto, Japan, 1996.

[MAG96b] MAGEDANZ, T., ROTHERMEL, K., KRAUSE, S. Intelligent Agents: An Emerging Technology for Next Generation Telecommunications? *INFOCOM'96*, San Francisco, CA, USA, 1996.

[MAS99] MASCOLO, C. MobiS: A Specification Language for Mobile Systems. *In Third Int. Conference on Coordination Models and Languages, P. Ciancarini and A. Wolf (editors), Lecture Notes in Computer Science 1594*, Springer-Verlag, Amsterdam, The Netherlands, 1999, p. 37-52.

[MER96] MERZ, M., LAMERSDORF, W. Agents, Services and Electronic Markets: How Do They Integrate?. In *Proceedings of The Int. Conf. On Distributed Platforms*, IFIP/IEEE, 1996.

[MIL80] MILNER, R. A Calculus of Communicating Systems, *Lecture Notes in Computer Science*, v. 92, Springer-Verlag, 1980.

[MIL91] MILNER, R. The Polyadic π-Calculus: a Tutorial. *Technical Report ECS-LFCS91-180*, LFCS, Dept. of Computer Science, University of Edinburgh, 1991.

[MIL99a] MILNER, R. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999.

[MIL99b] MILOJIČIĆ, D. S., DOUGLIS, F., PAINDAVEINE, et al. Process Migration. *Technical Report HPL-1999-21*, Computer Systems Laboratory, HP Laboratories, Palo Alto, 1999.

[NIC98] DE NICOLA, R., FERRARI, G., PUGLIESE, R. KLAIM: a Kernel Language for Agents Interaction and Mobility, *Transactions on Software Engineering*, v. 24, n. 5, IEEE Computer Society, 1998. p. 315-330.

[OBJ00] OBJECTSPACE. *Voyager ORB 4.0 Developer Guide*. ObjectSpace, Inc.

2000.

[PAR01] PARROW, J. An Introduction to the Pi-Calculus. In *Handbook of Process Algebra*, ed. Bergstra, Ponse, Smolka, Elsevier, 2001, p. 479-543.

[PET73] PETRI, C. A. Concepts of Net Theory. In *Mathematical Foundations of Computer Science: Proc. of Symposium and Summer School*, High Tatras, Math. Inst. of the Slovak Acad. of Sciences, 1973, p. 137-146.

[PIC97] PICCO, G.P., ROMAN, G. C., MCCANN, P.J. Expressing Code Mobility in Mobile UNITY, In *Proceedings of the Sixth European Software Engineering Conference (ESEC'97), Jazayeri, M., and Schauer, H., (editors), Lecture Notes in Computer Science 1301*, Springer-Verlag, 1997, p. 500-518.

[PIE97] PIERCE, B., TURNER, D. *Pict: A Programming Language Based on the Pi-Calculus*, Tech. Report 476, Indiana University, 1997.

[PLO99] PLÖSCH, R., WEINREICH, R. An Agent-Based Environment for Remote Diagnosis, Supervision and Control. In *Proceedings of the International Computer Science Conference (ICSC 99)*, Lecture Notes in Computer Science, Springer-Verlag, 1999, p. 385-392.

[PSO99] PSOUNIS, K. Active Networks: Applications, Security, Safety and Architectures. *IEEE Communications Surveys*, 1999.

[ROD01] RÖDEL, E. T. Especificação Formal de Aplicações Móveis: Estudo Comparativo e Métodos de Aplicação no Projeto ForMOS. *Trabalho Individual I*, Programa de Pós-Graduação em Ciência da Computação – Mestrado, PUCRS, 2001. 58 f.

[ROM98] ROMAN, G., MCCANN, P. J. An Introduction to Mobile UNITY. In *Proceedings of the Third International Workshop on Formal Methods*

for Parallel Programming: Theory and Applications (FMPPTA'98), in Parallel and Distributed Processing, Rolim, J., (editors), *Lecture Notes in Computer Science 1388*, Springer-Verlag, 1998, p. 871-880.

[ROZ97] ROZENBERG, G. *The Handbook of Graph Grammars, v. 1: Foundations*, World Scientific, 1997.

[SCH98] SCHWARTZ, B., ZHOU, W., JACKSON, A. W. *Smart Packets For Active Networks*. BBN Technologies, 1998.

[ASS99a] ASSIS SILVA, F. M. Agentes Móveis. *Anais da EINE'99*, Salvador, Bahia, Brasil, 1999.

[ASS99b] ASSIS SILVA, F. M. A Transaction Model Based on Mobile Agents. *PhD thesis*. Technical University of Berlin, 1999. 178 f.

[SMI99] SMITH, J., CALVERT, K., MURPHY, S., et al. Activating Networks: A Progress Report. *IEEE Computer*, n. 32, v.4, 1999, p. 32-41.

[STA96] STASKAUSKAS, M. G. An Experience In The Formal Verification of Industrial Software. *Communications of ACM*, v. 39, n. 12, Article 256, 1996.

[TAE00] TAENTZER, G., EHRIG, H. Semantics of Distributed System Specifications Based on Graph Transformation. In *Proc. of GI-Jahrestagung 2000*, Workshop Rigorose Entwicklung software-intensiver Systeme, Berlin, 2000.

[THO97] THORN, T. Programming Languages for Mobile Code. *ACM Computing Surveys*, v. 29, n. 3, 1997.

[UHR00] UHRMACHER, A. M., TYSCHLER, P., TYSCHLER, D. Modeling and Simulation of Mobile Agents. *Future Generation Computer Systems*, v.

17, n.2, 2000, p. 107-118.

[WET96] WETHERALL, D. J., TENNENHOUSE, D. L. The ACTIVE_IP Option. In *the 7th ACM SIGOPS European Workshop*, Connemara, Ireland, 1996.

[WET98] WETHERALL, D. J., GUTTAG, J. V., TENNENHOUSE, D. L. ANTS: A Toolkit For Building and Dynamically Deploying Network Protocol. In *Proc. IEEE OPENARCH'98*, 1998.

[WHI94] WHITE, J. E. Telescript Technology: The Foundation For The Electronic Marketplace. *Technical Report General Magic, Inc.*, White Paper, 1994.

[WOJ99] WOJCIECHOWSKI, P., SEWELL, P. Nomadic Pict: Language and Infrastructure Design for Mobile Agents, In *Proceedings of the ASA/MA'99*, 1999.

[YEM93] YEMINI, Y. The OSI Network Management Model. *IEEE Communications*, 1993, p. 20-29.

[YEM96] YEMINI, Y., SILVA, S. Towards Programmable Networks. In *Proc. IFIP/IEEE International Workshop on Distributed Systems, Operations and Management*, L'Aquila, Italy, 1996.

[YOO98] YOO, M. J., MERLAT, W., BRIOT, J. P. Modeling and Validation of Mobile Agents On The Web. In *Proc. of International Conference On Web-Based Modeling & Simulation*, San Diego, California, USA, 1998.