Desenvolvimento de Aplicações Móveis Corretas[1]

Lucio Mauro Duarte[2] e Fernando Luís Dotti
[lduarte, fldotti] @inf.pucrs.br
Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática - Programa de Pós-Graduação em Ciência da Computação
Av. Ipiranga, 6681 - CEP 90619-900 - Porto Alegre - RS

***Resumo* – O desenvolvimento de aplicações móveis para sistemas abertos é uma tarefa árdua. A depuração de tais aplicações é especialmente difícil, pois, além de tratarem-se de aplicações distribuídas, são móveis, dificultando ainda mais o processo de teste. Considerando-se aplicações móveis em ambientes abertos (e.g. Internet), não se pode afirmar se uma eventual falha faz parte da aplicação móvel ou do ambiente onde ela roda. O projeto ForMOS[1] (Métodos Formais para Código Móvel em Sistemas Abertos) tem, entre seus objetivos, a geração de aplicações móveis corretas. Neste sentido, uma linguagem de especificação formal para aplicações móveis foi proposta. A partir da especificação de uma aplicação móvel com este formalismo, é possível simular seu funcionamento e gerar código para execução sobre uma plataforma de suporte a mobilidade. Este artigo descreve brevemente o formalismo utilizado e o ambiente de simulação e, com maior atenção, a geração de código executável de aplicações móveis. Como exemplo, uma aplicação móvel já especificada, simulada, e cujo código já foi gerado e executado, é descrita.**

Abstract – **The development of mobile applications for open systems is a hard task. Debugging such applications is specially difficult because they are not only distributed, but also mobile applications, increasing the complexity of the testing phase. Considering mobile applications in open environments (e.g. Internet), one can not be sure whether an error is a result from the mobile application or from the environment where it runs. The ForMOS[1] (Formal Methods for Mobile Code in Open Systems) project has, among its objectives, the generation of correct mobile applications. In this context, a formal specification language for mobile applications was proposed. From the specification of a mobile application with this formalism, it is possible to simulate its behavior and to generate code to be executed in a mobility support platform. This paper briefly describes the used formalism and the simulation environment and, in detail, the code generation for mobile applications. As an example, an already specified and simulated mobile application, and whose code was already generated and executed, is presented.**

I. INTRODUÇÃO

O contínuo e rápido crescimento das capacidades de comunicação e processamento levou ao surgimento de ambientes computacionais massivamente distribuídos. Estes ambientes são freqüentemente chamados *ambientes abertos*, sendo caracterizados por: massiva distribuição geográfica, alta heterogeneidade e dinamismo, inexistência de controle global, falhas parciais e falta de segurança. A atual Internet é o exemplo mais presente de tal tipo de ambiente. Desenvolver aplicações para estes ambientes é uma tarefa complexa e esforços de pesquisa têm sido direcionados para melhorar o suporte ao desenvolvimento e execução de aplicações distribuídas. O desenvolvimento de tecnologias de *código móvel* [1] é um resultado dos esforços de pesquisa para melhorar o suporte ao desenvolvimento e execução de aplicações distribuídas em ambientes abertos. Estas tecnologias permitem que trechos de código possam ser transmitidos através da rede para serem executados em locais remotos. A migração não é transparente ao desenvolvedor das aplicações, mas explicitamente manipulada por ele. Muitas aplicações são consideradas fortes candidatas ao uso desta tecnologia, tais como comércio eletrônico [14], gerência de *workflows* [16], gerência de redes [1][17], implementação de serviços de telecomunicação [20], recuperação de informações distribuídas [1] e redes ativas [2]. Mobilidade de código é também apropriada para a crescente área de mobilidade de nodos físicos, pois é possível lançar componentes para execução remota, desligar ou desconectar o nodo (e.g. *laptop*), ligá-lo mais tarde e receber os resultados das computações remotas.

Entretanto, quando consideramos a geração de aplicações corretas baseadas em código móvel, notamos a carência de uma abordagem formal adequada ao problema em questão, que permita um melhor entendimento, análise e comparação com outros modelos, e mesmo prever sua evolução. De maneira geral, se uma aplicação é descrita utilizando uma linguagem de especificação apropriada, é possível aplicar técnicas de verificação para garantir propriedades desejadas do sistema, bem como garantir a correção da implementação. Quando consideramos sistemas complexos, como sistemas móveis executando em ambientes abertos, a necessidade de utilização de métodos formais para o seu desenvolvimento se torna ainda maior, devido à maior dificuldade em se inferir o comportamento do sistema através de testes (a mesma aplicação executando com os mesmos dados de entrada pode gerar resultados diferentes dependendo do estado da rede).

Alguns esforços para alcançar modelos computacionais para aplicações móveis foram propostos, como por exemplo o Cálculo Pi [18][19], *abstract state machines* [20], *mobile ambients* [21], e *actors* [22]. Entretanto, para seu uso na prática, linguagens de especificação e de programação de alto nível com semântica descrita conforme estes modelos, se fazem necessárias. Existem algumas linguagens de programa-

[1] Financiamento CNPq (520269/98-5), ProTeM-Laboratórios, e FAPERGS.
[2] Este autor é parcialmente financiado pela CAPES.

ção propostas, muitas delas descritas em [3], como por exemplo KLAIM [5], Objective Caml [6], Pict [23], Nomadic Pict [24] e Java [4]. Entretanto, no nível de especificação ainda não existe um método formal largamente utilizado para aplicações móveis.

Em [8], o uso de uma técnica de descrição formal, chamada Gramáticas de Grafos [9][10], foi introduzida para especificar aplicações móveis. Lá, uma classe específica de Gramáticas de Grafos chamada Gramática de Grafos Baseada em Objetos (GGBO) foi introduzida e então estendida com as noções de localização e mobilidade, permitindo a especificação de aplicações móveis baseadas em objetos.

Além de um método formal de descrição, é importante contar com métodos e ferramentas de análise e geração de código. Neste artigo, além de descrevermos sucintamente o método de especificação, apresentaremos o ambiente de simulação de GGBO, juntamente com suas extensões para mobilidade, e a geração de código executável para aplicações móveis, dando maior ênfase ao último tópico. Este artigo está organizado da seguinte maneira: na seção II a linguagem de especificação adotada e na seção III o ambiente de simulação utilizado são brevemente apresentados. A seção IV descreve a geração de código executável a partir da especificação e a seção V apresenta um exemplo de aplicação móvel. Por fim, conclusões e trabalhos futuros e bibliografia são apresentados.

II. Linguagem de Especificação

O emprego de uma *linguagem de especificação formal (LEF)* permite gerar 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 [7]. Dessa forma, pode-se testar e provar matematicamente que um sistema possui as características esperadas. Conforme já discutido, a LEF utilizada neste trabalho baseia-se em uma forma restrita de gramáticas de grafos, chamada de Gramáticas de Grafos Baseadas em Objetos. Além de ser uma linguagem visual, esta apresenta a vantagem de as especificações adquirirem um estilo baseado em objetos, que é bastante familiar à maioria dos desenvolvedores. Por conseqüência, isso as torna fáceis de construir e entender, mesmo por não especialistas, e possibilita que elas sirvam como base para uma implementação.

As entidades envolvidas em uma especificação de gramáticas de grafos são definidas em um *grafo de tipos*. Este grafo apresenta as entidades, os nomes e tipos de seus atributos e os tipos de mensagens que cada entidade pode receber.

O comportamento do modelo é determinado pela aplicação de regras aos grafos que representam o estado real do sistema a partir de um *grafo inicial*, no qual é representado o estado inicial do sistema. As aplicações de múltiplas regras podem ocorrer em paralelo se não houver conflito entre elas (duas ou mais regras não podem modificar um mesmo item). Quando duas aplicações de regras estão em conflito, a escolha de qual delas será aplicada é não-determinística.

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 atual *G*. Ou seja, uma regra *r: L ® R* é aplicada somente se *L* é um subgrafo atual de *G*. O lado direito da regra define o grafo resultante da aplicação desta regra. Portanto, *R* é o grafo que resulta ao aplicar-se a regra *r* sobre o grafo *L*. A interpretação operacional de uma regra *r: L ® 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*.

No caso de GGBO, a ocorrência do lado esquerdo de uma regra corresponde a encontrar a mensagem que dispara esta regra no grafo de estados e verificar se os atributos lidos/alterados por esta regra têm os valores esperados, isto é, os valores especificados no lado esquerdo da regra acontecem. Sendo uma linguagem baseada em objetos, só podem aparecer nas regras referentes a esta entidade atributos internos da própria entidade.

A ocorrência de um passo de derivação também pode estar ligada à satisfação de condições associadas à regra que trata uma mensagem. Estas condições podem definir certos valores que alguns atributos da entidade devem possuir para que se possa aplicar uma dada regra.

A semântica de gramáticas de grafos é definida como o conjunto de todas as computações que podem ser realizadas usando as regras da gramática, a partir do estado inicial.

A. Regras Básicas de Aplicações Móveis

Um cenário típico de mobilidade envolve dois tipos de entidades: lugares e componentes móveis. *Lugares* funcionam como possíveis localizações de componentes móveis, onde estes podem executar. Lugares oferecem funcionalidades básicas como armazenamento, comunicação e poder de processamento. Além disso, lugares, possuem a capacidade de receber componentes móveis e de fornecer serviços de movimentação a estes. *Componentes móveis* são componentes de software que podem se movimentar entre lugares durante sua execução, tirando proveito dos recursos e serviços de cada lugar por onde passam. Componentes móveis possuem dados internos (estado do componente), código e dados básicos, como localização, identificação, etc.

Dadas estas duas entidades, definidas conforme apresentado no parágrafo anterior, e um ambiente onde se pressupõe a inexistência de falhas de comunicação, foram definidas regras básicas que descrevem o comportamento padrão de lugares e componentes móveis dentro do processo de movimentação. Tais regras foram definidas seguindo as definições de GGBO e foram divididas em três grupos: regras de comunicação, regras de lugares e regra de componentes móveis.

As *regras de comunicação* definem as formas de troca de

mensagens entre entidades. As regras de comunicação são, na verdade, descritas como *esquemas de regras para passagem de mensagens*. 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 especificada. Os esquemas de regras de comunicação definem que: componentes móveis, em lugares diferentes ou não, podem trocar mensagens; componentes móveis podem mandar mensagens somente para o lugar onde estão; e lugares podem trocar mensagens entre si.

Como se trabalha com GGBO (ou seja, realiza-se a especificação de sistemas baseados em objetos), 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.

As *regras de lugares* determinam o comportamento padrão de um lugar no processo de movimentação de componentes. Segundo as regras definidas, um componente *Comp1* pode requisitar ao lugar *Orig*, onde se encontra, a sua movimentação para um lugar *Dest*. Isto ocorre através do envio, por parte do componente, ao lugar *Orig* de uma mensagem *Move*, a qual leva como parâmetros o componente que pede a movimentação e o local de destino desta movimentação. Além disso, podem acompanhar a mensagem alguns requisitos que devem ser atendidos para que a movimentação ocorra, tais como disponibilidade de recursos, possibilidade de fornecimento de determinados serviços, etc.

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.

Quando chega a resposta positiva do lugar de destino, o lugar de origem atualiza seu estado interno, para refletir a saída do componente e 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 condição, e outra mensagem ao componente, para que este atualize seu atributo interno de localização, passando a referenciar o seu novo local.

Caso o pedido de movimentação seja negado pelo lugar de destino, o lugar de origem envia uma mensagem *NoGo* ao componente requisitante informando-o do fato.

A *regra de componentes móveis* ocorre sempre que um pedido de movimentação é atendido. Ao receber a confirmação de sua movimentação física, o componente atualiza seu atributo interno de localização para o seu novo lugar de execução. Cabe citar que um componente móvel deve sempre estar em um lugar. Portanto, o seu atributo de localização deve sempre conter uma referência a algum lugar do cenário de execução. Após atualizar seu atributo de localização, o componente envia a si mesmo uma mensagem de *Continue* para retomar sua execução.

As regras aqui definidas servem para serem utilizadas dentro das especificações para aplicações móveis. Desta forma, a aplicação deve possuir uma regra que inicia 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 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*. Nestas regras, a aplicação define como tratar uma movimentação bem sucedida (recebimento de *Continue*) e uma negativa de um pedido de movimentação (recebimento de *NoGo*). Assim, 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.

III. Ambiente de Simulação

No contexto deste trabalho, a principal vantagem do uso de simulação é a possibilidade de validar uma estratégia de projeto antes da implementação. O simulador desenvolvido no ambiente do projeto PLATUS [12], utilizando a linguagem Java, permite a realização de simulações sobre modelos descritos utilizando GGBO. Este simulador trabalha com entidades, que são os componentes do sistema, e mensagens, que são o meio de comunicação entre entidades. O *kernel* de simulação é responsável pela entrega das mensagens, bem como pelo controle temporal da simulação. Cada mensagem recebe um *timestamp* do *kernel*. O controle do tempo de simulação é feito por um relógio global controlado pelo *kernel*. A cada avanço do relógio simulado, o *kernel* identifica as mensagens que devem ser enviadas naquele instante e as encaminha a seus destinos.

Cada entidade possui uma lista de regras que descrevem o seu comportamento. Assim, ao receber uma mensagem, a entidade verifica, dentre as regras de sua lista, quais podem tratar aquela mensagem. Esta seleção baseia-se em condições associadas às regras. Do conjunto de regras que podem tratar a mensagem recebida, uma é escolhida aleatoriamente, seguindo a mesma semântica de gramáticas de grafos. A regra escolhida é então executada. Caso uma mensagem recebida não possa ser tratada devido, por exemplo, a uma condição não satisfeita associada à regra que deveria tratá-la, ela é recolocada na fila de mensagens e haverá novas tentativas de tratá-la até que a condição seja satisfeita. Se a condição não for nunca satisfeita, configurando um problema na especificação da aplicação, a mensagem permanecerá na fila de mensagens até que o sistema seja finalizado.

Ferramentas gráficas para auxiliar no uso do simulador estão em desenvolvimento. Tais ferramentas deverão permitir a criação de modelos de simulação através da conversão da representação gráfica para o código correspondente.

Com base no simulador para GGBO, foram criadas entida-

des especiais para representar o comportamento de lugares e componentes móveis simulados. Estas entidades possuem o comportamento descrito na seção II. Com isto, além das simulações já possíveis através do simulador, com a adição destas entidades especiais, tornou-se possível utilizá-lo como uma ferramenta de suporte ao desenvolvimento também de aplicações móveis.

IV. Geração de Código para GGBO

Dada uma especificação formal do sistema na LEF utilizada, é essencial que haja uma forma de mapear esta especificação para uma linguagem de programação, a fim de transformá-la em código executável. Esse mapeamento, no entanto, não é simples, pois exige que as características do sistema, descritas na especificação, sejam preservadas quando o sistema é mapeado para a implementação. Além disso, a semântica do formalismo utilizado também deve ser respeitada para garantir uma implementação fiel ao sistema especificado.

Para realizar esta tarefa de transformação de especificação formal em código executável, partiu-se do mapeamento proposto no simulador PLATUS, apresentado na seção anterior. Como ele apresenta um mapeamento de GGBO para a linguagem Java para criar seus modelos de simulação e a LEF criada para descrever aplicações móveis (apresentada em III) é também baseada em GGBO, optou-se por utilizar o mapeamento realizado pelo simulador como ponto de partida.

Segundo o mapeamento realizado para criar os modelos de simulação, as entidades são descritas como classes Java que executam como uma *thread*. A entidade possui e gerencia uma fila de recebimento de mensagens. Para cada mensagem recebida pela entidade e que é aceita por ela (mensagem que gera a execução de uma regra da entidade), é criada uma *thread* para tratar da execução da regra associada à mensagem. Terminada a execução da regra, a *thread* criada para tratá-la é finalizada. Como descrito na seção anterior, a comunicação entre entidades e o controle do relógio global do sistema são realizados pelo *kernel*.

Partindo-se da estrutura definida no simulador, a qual foi prevista para a simulação de sistemas de produção, foram realizadas modificações que permitiram utilizá-lo para, não só a simulação, mas também para a execução propriamente dita de aplicações móveis implementadas a partir de especificações formais feitas em GGBO. Para isso, foram realizadas 3 etapas, descritas a seguir.

A. Etapa 1: Geração de Código para Execução Local

Como primeira etapa da geração de código para execução, realizou-se a retirada do *kernel* de simulação. A saída desta entidade centralizadora determinou que a troca de mensagens passasse a ser direta entre as entidades, sem a intervenção do *kernel*. Outra característica que ficou alterada é que o relógio simulado deixou de existir e passou-se a ter a execução regida pelo tempo real. Dentro da etapa de execução local, o efeito da troca de mensagens feita diretamente entre as entidades não apresentou problemas, pois passou a ser realizada como uma simples invocação de método entre as entidades envolvidas.

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, uma simples alteração possibilitava utilizar o mesmo código da simulação para realizar-se uma execução local.

B. Etapa 2: Geração de Código para Execução Distribuída

A segunda 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 plataforma de suporte a mobilidade (PSM), sendo que a plataforma escolhida foi a Voyager [13]. Voyager é uma plataforma de suporte à distribuição e à mobilidade de código desenvolvida pela ObjectSpace, Inc. Esta plataforma é totalmente desenvolvida em Java e fornece o suporte necessário à comunicação distribuída, resolução de referências, ativação remota, transparência de localização e mobilidade de componentes, etc.

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, e não mais através de invocações locais de métodos. Da mesma forma que na etapa anterior, o código gerado permitia, com algumas poucas alterações, a utilização tanto para simulação quanto para execução distribuída.

C. Etapa 3: Execução envolvendo mobilidade

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

Para suportar a idéia de movimentação de componentes, foram criadas duas novas entidades especiais: *Place* e *MAgent*. A primeira define um padrão para um componente que executa distribuidamente como um lugar. À cada entidade que estende *Place*, são associadas regras predefinidas que determinam o seu comportamento, de acordo com o definido na seção II. A entidade *MAgent* define um padrão para um componente móvel, incluindo operações para movimentação da entidade através da plataforma de suporte. Para esta entidade (e para todas as entidades que a estenderem) também foi criada uma regra que determina seu comportamento segundo a regra de componentes móveis criada na LEF utilizada.

Como as regras básicas de movimentação foram incorporadas às entidades *MAgent* e *Place*, o usuário se preocupa apenas com as regras específicas das entidades de sua aplicação, as quais estenderão uma destas duas entidades. Dessa forma, uma hierarquia de herança de entidades foi definida, onde *Entity* representa a entidade padrão do simulador, a qual é estendida por *Place* e *MAgent.* Ao construir sua aplicação, o usuário cria suas entidades que estendem ou *Place* ou *MAgent*.

A fim de possibilitar a movimentação das entidades que estendem *MAgent*, foram feitas outras alterações na classe *Entity*. A mudança mais importante foi adicionar a possibilidade de parada das atividades internas da entidade e salvamento dos atributos necessários à movimentação, bem como a possibilidade de reiniciar o funcionamento no local de destino. Alguns exemplos de aplicações móveis foram e estão sendo desenvolvidos para testar a geração de código proposta. Os resultados obtidos na execução do código gerado são consistentes com os resultados de simulação.

V. Exemplo de Aplicação

Para desenvolver estes exemplos, as etapas seguidas foram a definição do cenário da aplicação, especificação da aplicação em GGBO para aplicações móveis, mapeamento da especificação para simulação, realização de simulações para verificar se o comportamento da aplicação confere com o esperado e mapeamento do código simulado para código executável.

Para apresentar de forma mais concreta as etapas seguidas, serão apresentados, a seguir, os resultados de cada etapa para um exemplo de aplicação simples.

A. Descrição do Cenário da Aplicação

Esta aplicação envolve 4 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 passar, 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 devolve a este o resultado.

B. Especificação do Sistema

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 na seção II, apresenta as entidades envolvidas no sistema e seus atributos. O grafo de tipos desta aplicação é apresentado na Fig. 1.

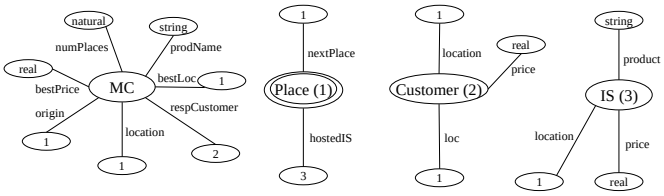

Fig. 1. Grafo de tipos da aplicação.

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.

Deve-se citar que o grafo de tipos da Fig. 1 não é um grafo de tipos completo. Dever-se-ia apresentar também as mensagens que cada entidade pode receber. Isto não consta no grafo de tipos apresentado por questões de facilidade de visualização. O grafo de tipos apenas com as mensagens que cada entidade pode receber é apresentado na Fig. 2.

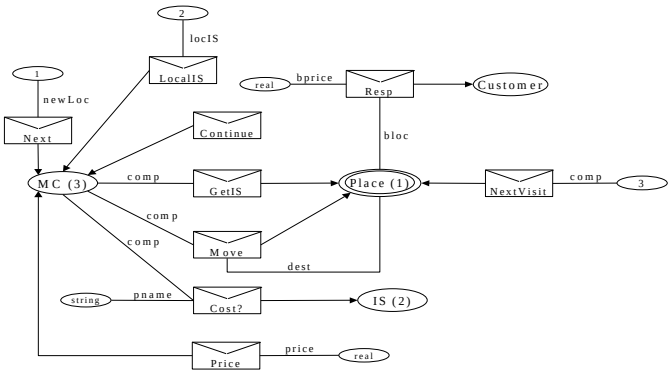

Fig. 2. Grafo de tipos de mensagens da aplicação.

O grafo inicial, também discutido na seção II, descreve a situação inicial do sistema. O grafo inicial para esta aplicação é apresentado na Fig. 3.

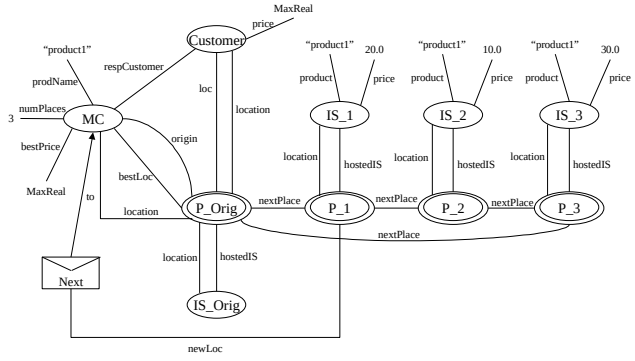

Fig. 3. Grafo inicial da aplicação.

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*[3]. A mensagem *Next*, enviada a *MC* define o início da execução do sistema. Para a entidade *MC*, são agora definidas suas regras na Fig. 4 e na Fig. 5. As regras das demais entidades envolvidas na aplicação são suprimidas.

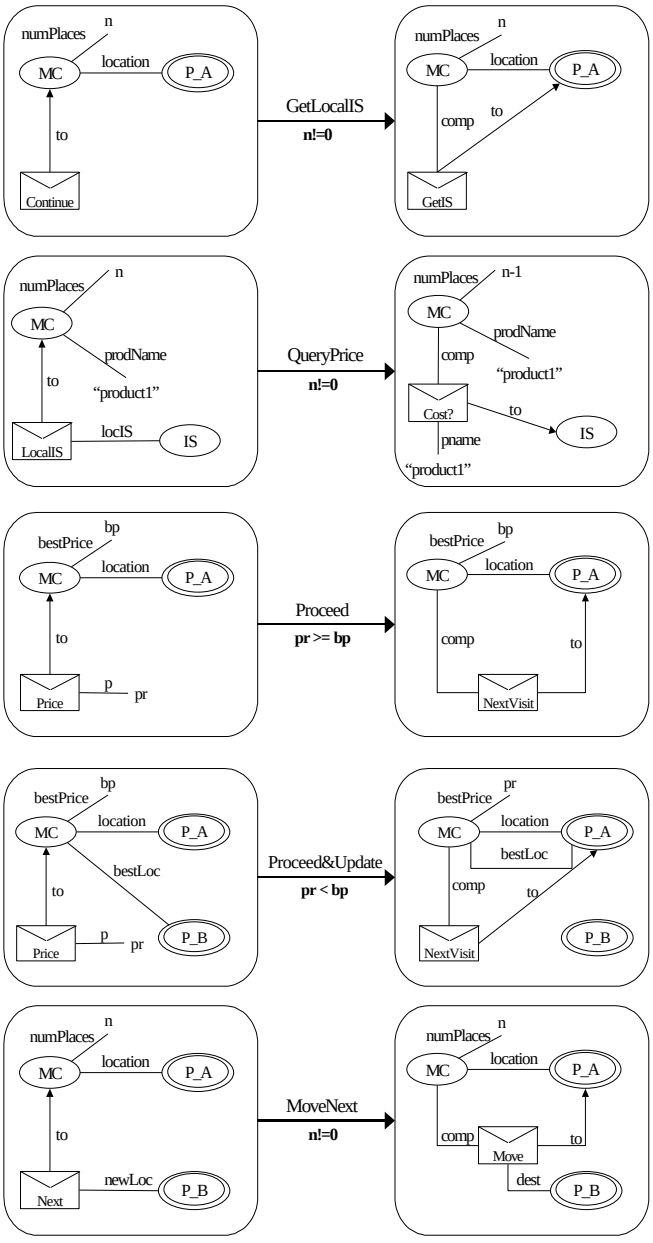

Fig. 4. Regras do *MC* – Parte 1.

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

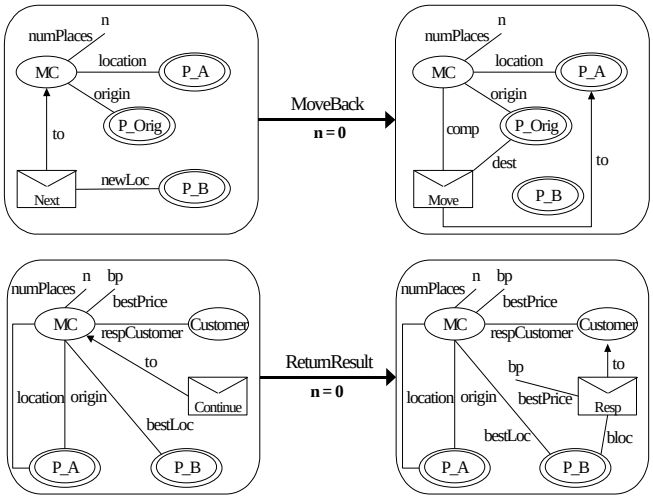

Fig. 5. Regras do MC – Parte 2.

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* (Fig. 4). Apesar de haver outra regra que também trata a mesma mensagem (*MoveBack*, na Fig. 5) 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 é o descrito na regras básicas de movimentação, apresentadas na seção II. Assim, a próxima regra a ser executada é a regra que trata uma mensagem de *Continue*. Novamente têm-se 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*. Esta regra serve para que o *MC* obtenha uma referência ao *IS* local para poder consultar o preço oferecido. O comportamento do *place* ao receber a mensagem *GetIS* é retornar a mensagem *LocalIS* ao *MC* requisitante.

Recebida a referência pedida, o *MC* pode então realizar a consulta ao preço oferecido para o produto *product1*, conforme descrito na regra *QueryPrice* (Fig. 4). Note-se que, neste instante, o *MC* decrementa o número de *places* a visitar.

Um *IS*, quando consultado sobre o custo de um dado produto de nome *pname* (mensagem *Cost?*), retorna a mensagem *Price* ao *MC*, contendo o preço para o produto consultado.

Quando recebe a mensagem contendo o preço para o produto desejado, *MC* pode executar duas regras (vide Fig. 4): *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* responde 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 3 *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* (Fig. 5), retornando ao lugar de origem. Terminada a movimentação e recebida a mensagem de *Continue*, o *MC* executa a regra *ReturnResult* (Fig. 5), enviando os resultados de sua pesquisa para o *customer*, o qual consome a mensagem.

C. Geração de Código

A geração de código compreende mapear a especificação para código Java executável. Para exemplificar como fica o código de uma entidade, a Fig. 6 mostra o código da entidade *MC*.

```
public class MC extends MAgent implements IMC, Serializable {
  protected int numPlaces;
  protected String prodName;
  protected float bestPrice;
  protected IShoppingPlace bestLoc, origin;
  protected ICostumer respCostumer;

  public MC () { super(); }

  public void init (int numPlaces, String prodName, float bestPrice,
IShoppingPlace bestLoc, ICostumer respCostumer, IShoppingPlace
origin, IShoppingPlace location)  {
    setNumPlaces(numPlaces);
    setProdName(prodName);
    setBestPrice(bestPrice);
    setBestLoc(bestLoc);
    setRespCostumer(respCostumer);
    setOrigin(origin);
    super.initialize();  }

  public void initialize () {super.initialize(); }

  public void registerRules() {
    super.registerRules();
    addRule(new MC_R1 ("GetLocalIS", this));
    addRule(new MC_R2 ("QueryPrice", this));
    addRule(new MC_R3 ("Proceed", this));
    addRule(new MC_R4 ("Update_Proceed", this));
    addRule(new MC_R5 ("MoveNext", this));
    addRule(new MC_R6 ("MoveBack", this));
    addRule(new MC_R7 ("ReturnResult", this));  }
}
```

Fig. 6. Código do *MC*.

A entidade estende a entidade especial *MAgent*, adotando o comportamento de um componente móvel. A primeira parte define os atributos da entidade. Para poder comunicar-se remotamente, cada entidade implementa uma interface, como *MC* implementa *IMC*. *MC* ainda implementa *Serializable* para poder ser transmitido na rede, permitindo sua movimentação. Os atributos da entidade que referenciam outras entidades são do tipo da interface que estas entidades implementam. Ou seja, têm-se referências remotas (*proxies*) às entidades. O método *init* serve para fornecer os valores iniciais para os atributos da entidade e para inicializar a entidade remotamente. Para acessar os atributos remotamente, devem ser fornecidos métodos de leitura e escrita de valores de atributos. Estes métodos são suprimidos aqui. O último método é comum a todas as entidades. Este método (*registerRules*) serve para definir as regras pertencentes à entidade. Cada regra é definida por um nome e a entidade a qual ela pertence. Um exemplo de código de regra para execução é apresentado na Fig. 7.

```
class MC_R1 extends Rule implements java.io.Serializable {
  public MC_R1 (String strId, Entity e)  { super(strId, e); }

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

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

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

  public void apply (Message m) {
    MC mc = (MC) owner();
    Parameters par = new Parameters();
    IMC im = (IMC) mc.getLocation().getReference(mc);
    par.add("comp", im);
    IShoppingPlace location = mc.getLocation();
    mc.send(location, 10.0, new Message(im, "GetIS", par));
    System.out.println("MC :: GetLocalIS");
  }

  public boolean writeOnAttrib () {return false;}
}
```

Fig. 7. Código da regra MC_R1.

A regra MC_R1 corresponde à regra *GetLocalIS* (Fig. 4). Como toda regra, ela estende a classe *Rule*, a qual define o comportamento básico de regras. Toda regra possui 5 métodos que devem ser definidos: o método *answerMsg* define o tipo de mensagem que a regra trata (no caso, *Continue*); o método *conditionOk* é usado para testar se a condição de execução da regra é atendida (no caso desta regra, se está verificando se o número de *places* a visitar é diferente de zero); o método *leftSideOccurs* serve para se testar alguma restrição que tenha de ser feita em relação ao grafo do lado esquerdo da regra, tal como que um componente só pode pedir um serviço a um lugar se ele estiver nesse lugar (quando ter-se-ia o atributo de localização do componente referenciando o lugar ao qual requisitou o serviço); o método *apply* define o processamento necessário para transformar o grafo do lado esquerdo da regra no grafo do lado direito desta (envio da mensagem *GetIS* para o *place* onde o componente está, passando uma referência a este como parâmetro); e o método *writeOnAttrib* define se a regra altera algum atributo da entidade à qual a regra está associada. Este último método é usado para controlar acessos concorrentes à entidade.

Com isso, é gerado o código para todas as entidades e suas respectivas regras. Para executar é criada uma classe principal

que instancia e inicializa cada entidade.

VI. Conclusões e Trabalhos Futuros

O presente trabalho está em um estágio onde a LEF, o simulador e o mapeamento para código executável já constituem boas ferramentas de auxílio ao desenvolvimento de aplicações móveis. Tais ferramentas vêm sendo melhoradas através da realização de testes específicos sobre características inerentes a este tipo de sistema, como comunicação entre entidades, sincronização, movimentação de entidades e acesso concorrente a uma entidade. Além disso também foi identificada uma falha na plataforma usada em relação ao suporte à comunicação quando o componente se movimenta. Testes mostraram que havia perda de mensagens recebidas por uma entidade quando estas chegavam após o início da movimentação da entidade.

Algumas extensões ainda terão de ser feitas, dado que, por enquanto, não estão sendo consideradas alterações dinâmicas no sistema, como a criação e deleção de componentes. Dessa forma, atualmente trabalha-se com sistemas em que todos os seus componentes são criados na sua inicialização. A introdução da criação e deleção de entidades terá impacto na linguagem de especificação, no simulador e mesmo na geração de código. Este impacto já está sendo considerado e a implementação da criação de entidades já se encontra em andamento.

A continuação deste trabalho prevê também o desenvolvimento de estudos de caso de maior porte, de maneira a testar melhor e validar a idéia da utilização do formalismo e das ferramentas apresentadas. Além disso, esforços estão sendo investidos para a verificação de especificações utilizando a LEF apresentada e também no sentido de provar que as implementações propostas mantêm a semântica de gramáticas de grafos.

Como se pode notar na seção II, as abstrações de lugares e componentes móveis definem as características de comunicação e migração esperadas. A definição destas abstrações modela o comportamento esperado do ambiente onde aplicações móveis executam. De maneira análoga, poderíamos modificar tais abstrações para representar plataformas específicas ou comportamentos do ambiente. Por exemplo, nós poderíamos representar plataformas de suporte onde somente componentes móveis no mesmo lugar pudessem interagir. Assim, quando interações entre componentes remotos fossem necessárias, um dos componentes deveria migrar para encontrar o outro. Outra possibilidade é a representação de falhas do ambiente. Poder-se-ia especificar, por exemplo, que mensagens podem não chegar, ou que lugares podem estar isolados temporariamente. Apesar deste tópico ser de investigação futura, acreditamos que da mesma maneira poderíamos usar esta facilidade para especificar ambientes de redes sem fio, onde características como largura de banda, alcançabilidade e latência poderiam mudar ao longo do tempo.

VII. Bibliograraria

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

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

[3] DUARTE, L. M.. *Estudo de Linguagens de Programação com Suporte à Mobilidade de Código*. Relatório Técnico, PPGCC – PUCRS, 2000. 68 f.

[4] GOSLING, J., MCGILTON, H.. *The Java Language Environment - A White Paper*. SunMicrosystems, 1996.

[5] DE NICOLA, R., FERRARI, G., PUGLIESE, R.. KLAIM: a Kernel Language for Agents Interaction and Mobility, *Transactions on Software Engineering*, vol. 24, nº 5, IEEE Computer Society, 1998. p. 315-330.

[6] LEROY, X.. *Objective Caml*. 1997.

[7] 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.

[8] 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.

[9] CORRADINI, A.. Concurrent Computing: From Petri Nets to Graph Grammars. Eletronic Notes in Theorical Computer Science 2, *Proc. of the SEGRAGRA'95 Workshop on Graph Rewriting and Computation*, 1995. p. 245-260.

[10] RIBEIRO, L.. *Parallel Composition and Unfolding Semantics of Graph Grammars*. Ph.D. thesis, Technical University of Berlin, Germany, 1996.

[11] CHOMSKY, N.. Context-Free Grammars and Pushdown Storage.65 MIT Research Laboratory Electronics Quarterly Progress Report, 187-94.

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

[13] OBJECTSPACE. *Voyager ORB 4.0 Dev. Guide*. Objectpace, Inc. 2000.

[14] STRAßER, M, ROTHERMEL, K., MAIHÖFER, C. Providing Reliable Agents for Electronic Commerce. in *Trends in Distributed Systems for Electronic Commerce - International IFIP/GI Working Conference TREC'98*. Springer. Berlin. 1998. pp.241-253.

[15] MAGEDANZ, T., POPESCU-ZELETIN, R. Towards "Intelligence on Demand" - On the Impacts of Intelligent Agents on IN. in *Proceedings of the 4th International Conference on Intelligence in Networks*. Bordeaux, France. November, 1996.

[16] BARBARA, D., MEHROTRA, S., RUSINKIEWICZ, M. INCAs: Managing Dynamic Workflows in Distributed Environments. *Journal of Database Management*. Winter. 1996.

[17] 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. April, 1996.

[18] MILNER, R. and PARROW J. A calculus for mobile processes I, Information and Computation, vol. 100, 1992, p. 1-40.

[19] MILNER, R, PARROW J., and WALKER, D. A calculus for mobile processes II, Information and Computation, vol. 100, 1992, pp. 41-77.

[20] MAIA, M. and BIGONHA, R. Interaction based semantics for mobile objects, In Proc. of the III Brazilian Symposium on Programming Languages, 1999.

[21] CARDELLI, L. and GORDON. A., Mobile Ambients - Foundations of Software Science and Computational Structures, Lecture Notes in Computer Science, vol.1378, Springer, 1998, pp. 140-155.

[22] AGHA, G. and KIM, W. Actors:A unifying model for parallel and distributed computing. Journal of Systems Architecture 45, 1999, p. 1263-1277.

[23] PIERCE, B. and TURNER, D. Pict: a programming language based on the pi-calculus, Tech. Report 476, Indiana University, 1997.

[24] WOJCIECHOWSKI, P., SEWELL, P. Nomadic pict:language and infrastructure design for mobile agents, In Proc. of the ASA/MA, 1999.