Extensão da Linguagem Promela para Especificação de Sistemas baseados em Agentes Móveis*

Aline M.S. Andrade   Flávio M. Assis Silva
Frederico J.R. Barboza
{*aline, fassis*}*@ufba.br*   *fjbarboza@tre-ba.gov.br*

LaSiD - Laboratório de Sistemas Distribuídos
DCC - Departamento de Ciência da Computação
UFBA - Universidade Federal da Bahia
Av. Adhemar de Barros, S/N
Campus de Ondina - Salvador, BA
CEP 40170-110

Resumo

O estudo de aspectos relacionados ao desenvolvimento de sistemas e aplicações baseadas em agentes móveis tem atraído a atenção da comunidade acadêmica e da indústria nos últimos anos. O desenvolvimento de conceitos e ferramentas para garantir a correção de sistemas baseados em agentes móveis, no entanto, ainda é um tópico de estudo. Neste artigo apresentamos uma extensão da linguagem Promela que introduz primitivas para a especificação de componentes presentes em sistemas baseados em agentes móveis. Promela é a linguagem de especificação utilizada no SPIN, um ambiente de verificação de modelos e simulação, bastante utilizado na análise de protocolos distribuídos. Esta extensão permite a especificação de sistemas baseados em agentes móveis utilizando-se primitivas que modelam diretamente componentes presentes em tais sistemas e a utilização do SPIN como ferramenta de verificação de modelos e simulação.

1 Introdução

Nos últimos anos o estudo de aspectos relacionados ao desenvolvimento de sistemas de agentes móveis e de suas aplicações tem atraído a atenção da comunidade acadêmica e da indústria. Um agente móvel é um componente de software que é capaz de migrar autonomamente entre nós de um sistema distribuído. Devido a benefícios que podem trazer na implementação de sistemas distribuídos, tais como melhoria do uso dos recursos de comunicação, melhor uso dos recursos computacionais, suporte à extensibilidade de sistemas e suporte à operação desconectada, agentes móveis têm sido propostos para o desenvolvimento de sistemas/aplicações em diversas áreas, como gerenciamento de redes, comércio eletrônico, recuperação de informações distribuídas, serviços avançados de telecomunicações, redes ativas e sistemas de gerenciamento de workflows ([13, 8]).

No entanto garantir a correção de sistemas baseados em agentes móveis não é uma tarefa trivial. Tais sistemas são inerentemente distribuídos e concorrentes. Adicionalmente, eles podem ser estruturados em componentes que podem migrar autonomamente no ambiente distribuído, completamente independentes uns dos outros. Em ambientes abertos, como a Internet, fatores adicionais aumentam a complexidade de se garantir a correção de sistemas baseados em agentes móveis, como a possibilidade de agentes estarem se movendo em um ambiente formado por diversas organizações autônomas e sujeito a falhas.

O uso de ferramentas baseadas em métodos formais é tradicionalmente uma das formas de se abordar a questão de geração de sistemas corretos e, portanto, tem sido proposto o uso de tais ferramentas para tratar esta questão também em relação a sistemas baseados em agentes móveis. Métodos formais permitem a representação de sistemas através de um modelo matemático com sintaxe e semântica

*Trabalho desenvolvido no contexto dos projetos ForMOS (Métodos Formais para Código Móvel em Sistemas Abertos) (CNPq), IQ-Mobile (Improving the Quality of Open Systems with Code Mobility through Rigorous Development) (CNPq, CNR/Itália), ARGO (Tolerância a Falhas em Sistemas Distribuídos Assíncronos (CNPq, INRIA/IRISA, França) e projeto individual (segundo autor) Pesquisa sobre Correção e Confiabilidade de Aplicações baseadas em Agentes Móveis (Bolsa DCR, CNPq)

bem definidas. Por isto podem ser utilizados não somente como uma ferramenta para descrever de forma precisa as interações entre partes de um sistema, como também para permitir a verificação formal de propriedades. Outros enfoques para suportar a geração de sistemas corretos e que, portanto, aplicam-se também ao caso de aplicações baseadas em agentes móveis, são o uso de ferramentas de simulação, testes e de geração automática do código (em todo ou em parte).

Um dos paradigmas para verificação formal que vem sendo utilizado com sucesso na prática é o método de verificação de modelos. A verificação de modelos se baseia normalmente em uma representação do sistema em um modelo de estados (sistema de transição) e de propriedades a serem verificadas em relação a este sistema utilizando-se lógica temporal. Com o uso de uma ferramenta automática que analisa os estados do sistema, pode-se verificar se as propriedades especificadas são satisfeitas ou não.

Vários formalismos que expressam mobilidade foram desenvolvidos (por exemplo [12, 14, 3, 4, 7]). Baseadas nestes formalismos, algumas linguagens e ferramentas para verificação foram desenvolvidas (por exemplo: [16, 5, 15]). No entanto, não existe ainda um ambiente integrado com diferentes ferramentas de suporte à análise de correção (verificador de modelo, simulador e gerador de código, por exemplo) para suportar o desenvolvimento de sistemas baseados em agentes móveis. No contexto dos projetos ForMOS (*Métodos Formais para Sistemas Móveis em Ambientes Abertos*) e IQ-Mobile (*Improving the Quality of Mobile Applications with the use of Formal Methods*), estamos investigando o desenvolvimento de um tal ambiente integrado, baseando-se na utilização de métodos formais. Um destes ambientes que vêm sendo desenvolvidos nestes projetos é baseado em Gramáticas de Grafos (baseado no formalismo apresentado em [7]). Outro é baseado na ferramenta SPIN e é sobre este ambiente que trata este artigo. Este último ambiente está sendo desenvolvido também para utilização no contexto do projeto ARGO (*Tolerância a Falhas em Sistemas Distribuídos Assíncronos*), para a validação de protocolos distribuídos com mobilidade.

SPIN é um ambiente que fornece ferramentas automáticas para simulação e verificação de modelos, que tem sido bastante usado no desenvolvimento de aplicações distribuídas e projetos de protocolos [9]. Promela, a linguagem de especificação utilizada no SPIN, permite a criação dinâmica de processos concorrentes, que se comunicam via canais de mensagens síncronos ou assíncronos. Promela, no entanto, não possui primitivas para a especificação de mobilidade, ou seja, primitivas que expressem explicitamente componentes móveis (agentes) e seus movimentos.

O objetivo deste artigo é apresentar uma extensão da linguagem Promela que foi feita para possibilitar a modelagem explícita de sistemas baseados nestes componentes. Em particular, foram introduzidas na linguagem primitivas para a especificação de agentes móveis, do ambiente onde executam, de suas ações de movimento, da comunicação entre agentes e do comportamento dos mecanismos de comunicação em relação a falhas. Estas primitivas podem ser usadas diretamente na especificação de sistemas baseados em agentes móveis. Um tradutor foi então desenvolvido que mapeia estas primitivas em primitivas de Promela. A partir desta tradução, o SPIN pode então ser usado para verificar e simular aplicações baseadas em agentes móveis.

O benefício deste sistema é que, com esta extensão de Promela, podemos passar a utilizar um ambiente pronto com ferramentas para simulação e verificação, bastante usado na prática, para a análise de aplicações móveis, tendo o especificador primitivas que modelam diretamente conceitos da sua área de aplicação (conceitos de um ambiente de agentes móveis). Além disso, a linguagem Promela é uma linguagem bastante simples, com uma estrutura semelhante a uma linguagem de programação. Entendemos que esta característica facilita a produção de especificações por pessoas que não possuem experiência prévia com técnicas de especificação formal, mas que têm conhecimento de linguagens de programação de uso geral.

Este artigo está estruturado da seguinte maneira. Na seção 2 apresentamos uma descrição genérica do modelo adotado de agentes móveis e de seu ambiente. Na seção 3 descrevemos a extensão feita à linguagem Promela com as primitivas para modelar aplicações baseadas em agentes móveis. Na seção 4 apresentamos um exemplo de uso desta extensão para a especificação de um mecanismo de comunicação entre agentes com transparência de migração. Na seção 5 discutimos trabalhos relacionados. Finalmente, na seção 6 apresentamos as conclusões do trabalho.

2 Modelo do Ambiente de Agentes Móveis

Um *agente móvel* é um elemento de software capaz de autonomamente migrar entre nós de um sistema distribuído. Mais especificamente, um agente migra de uma *agência* a outra [13]. De um ponto de vista conceitual, uma agência representa um lugar lógico no sistema distribuído. De um ponto de vista funcional, uma agência representa a funcionalidade necessária em um nó para suportar o ciclo de vida de agentes (criação, execução, movimento, etc.). Quando um agente migra, sua execução é suspendida na agência original e o agente é transportado (código, dados e estado de execução) para uma outra agência no ambiente distribuído, onde a execução é retomada. Durante sua execução, um agente móvel pode criar um ou mais agentes móveis, chamados de seus *agentes filhos*. A figura 1 ilustra três agentes, $ag1$, $ag2$ e $ag3$, e cinco agências, *agência1, agência2, agência3, agência4* e *agência5*. Na figura, o agente $ag1$ moveu-se de *agência1* para a *agência2* e, nesta agência, criou dois agentes filhos, $ag2$ e $ag3$. Cada um deles migrou para uma agência distinta.

Em aplicações baseados em agentes móveis, as agências, além das funções básicas de manter o ciclo de vida dos agentes (como receber agentes, instanciar agentes e enviá-los para outras agências), podem também executar funções específicas daquela aplicação. Por exemplo, manter algum repositório de informação que os agentes queiram acessar naquela agência. Em nosso modelo, portanto, cada agência executa dois tipos de atividades: *atividades básicas* e *atividades específicas*. As atividades básicas representam atividades que toda agência realiza, independentemente de aplicação. O comportamento específico de uma agência corresponde a ações dependentes de aplicação, dentre elas, ações específicas a serem executadas na ocorrência de determinados eventos. Por exemplo, uma agência em uma determinada aplicação pode necessitar ter um comportamento específico sempre que um agente chega na agência.

Agentes e agências comunicam-se entre si trocando mensagens através de canais. Alguns sistemas de agentes móveis existentes atualmente provêm formas de comunicação entre agentes que suportam entrega de mensagens com *transparência de migração*, ou seja, uma mensagem enviada a um agente é entregue a ele independentemente de sua migração [11, 10]. Outros sistemas só mantêm o canal de comunicação entre agentes enquanto eles não migram (por exemplo, [1]). No nosso modelo, de maneira análoga a Nomadic Pict [14], o emissor de uma mensagem deve especificar a agência onde o recebedor estará. Fornecendo esta forma mais básica de comunicação, no entanto, o modelo pode ser usado para se especificar diferentes formas de se entregar mensagens com transparência de migração.

A comunicação entre agentes e entre agências é assíncrona, ou seja, não são conhecidos limites de tempo máximos para que uma mensagem seja transmitida de um agente a outro ou de uma agência a outra. Além de se comunicar com outros agentes, um agente pode se comunicar também com a agência onde se encontra. A comunicação entre um agente e uma agência é, no entanto, síncrona. Não há limites conhecidos também para o tempo de processamento relativo de agentes e agências.

Em relação a confiabilidade, dois tipos de canais são considerados: canais confiáveis, onde mensagens são eventualmente entregues em seus destinos sem duplicação, perda nem corrompimento; e canais não confiáveis, onde mensagens podem ser perdidas, mas não duplicadas nem corrompidas.

3 Representação do Modelo de Ambiente de Agentes Móveis sobre Promela

Nesta seção descreveremos a extensão realizada sobre Promela para implementar o modelo descrito na seção anterior. Na seção 3.1 apresentamos uma breve introdução a Promela. Em seguida, na seção 3.2, descrevemos a extensão em si da linguagem com as primitivas para especificação de sistemas baseados em agentes móveis.

3.1 Uma breve introdução a Promela

Promela é uma linguagem para verificação de modelos que permite a criação dinâmica de processos concorrentes. Ela se aplica à especificação de protocolos e sistemas distribuídos em geral e possui enfoque principal nas interações entre processos, permitindo que se abstraiam detalhes do comportamento dos sistemas.

Um programa em Promela consiste de processos, canais de mensagens e variáveis. Em Promela, processos são instanciados dinamicamente. A partir de uma mesma especificação de comportamento de um processo (ou seja, a especificação de um *tipo de processo*), várias instâncias de processos podem ser criadas. Os processos executam em paralelo, com intercalação de suas operações.

Para se descrever o comportamento de um processo, Promela oferece comandos de seleção e repetição baseados nos comandos guardados de Dijkstra [6], que envolvem não determinismo. Quando mais de uma condição de guarda associada a uma seqüência de comandos for executável, a escolha entre qual das seqüências executar é feita de forma não determinista. Sequências de comandos podem ser encapsuladas e executadas como uma unidade, não intercaladas com comandos de outros processos. Isto reduz a complexidade de um modelo de validação, pois diminui a quantidade de intercalação entre estados.

Os processos se comunicam através dos canais de mensagens, que podem ser síncronos ou assíncronos.

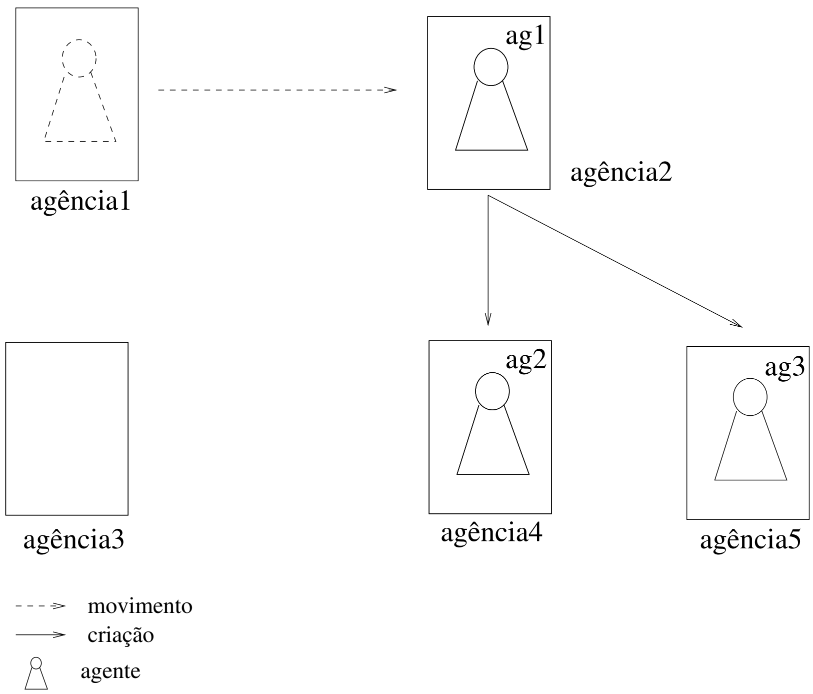

Figura 1: Modelo do ambiente de agentes móveis

As mensagens a serem enviadas em um canal podem ser de vários tipos, inclusive canais.

Variáveis em Promela são tipadas, podendo ser globais (ou seja, acessíveis a todos os processos de uma especificação) ou locais (acessíveis a apenas um processo). O conjunto de todas as variáveis com seus respectivos valores, num certo instante, representa o estado do sistema naquele instante.

Promela permite que o usuário especifique assertivas com condições booleanas que são verificadas automaticamente. Além disso, contém elementos que servem para guiar o verificador de modelos (para indicar, por exemplo, se um determinado ponto da execução de um processo corresponde a um ponto de terminação válido para aquele processo). Um usuário poderá usar estas funcionalidades para especificar assertivas particulares a serem verificadas sobre o comportamento do sistema que está sendo especificado.

3.2 Extensão de Promela para Representação de Componentes de um Ambiente de Agentes Móveis

Para a especificação de sistemas baseados em agentes móveis foram criadas primitivas para a especificação de agentes móveis, de agências, de ações de movimento de agentes, da comunicação entre agentes e do comportamento dos mecanismos de comunicação em relação a falhas, ou seja, componentes e funcionalidades básicas para a especificação de tais sistemas.

Uma especificação de sistema baseado em agentes móveis terá a forma de um texto onde poderão ser usados as primitivas introduzidas por nós e comandos de Promela. Uma especificação é então traduzida para uma outra, onde estas primitivas são mapeadas em componentes originalmente existentes em Promela. Esta especificação resultante pode ser utilizada no SPIN para se fazer verificação e simulação.

A seguir apresentamos as primitivas introduzidas.

Agências

Uma agência é especificada utilizando-se a primitiva *agency*. A sintaxe para a especificação de uma agência é ilustrada abaixo:

```
agency <nome da agencia>{

  onAgentArrive(AgencyMsg e){
   <comandos>
  }
  onAgentLeave(AgencyMsg e){
   <comandos>
  }
  onMessageDiscard(AgencyMsg e){
   <comandos>
  }
  onAgencyStart{
   <comandos>
  }

}
```

Ao se especificar uma agência, indicamos seu nome (`<nome da agencia>`) e seu corpo de comandos. Em seu corpo de comandos podemos especificar ações a serem executadas pela agência na ocorrência de quatro eventos:

- chegada de um agente: as ações a serem tomadas quando este evento ocorre são especificadas no bloco de instruções *onAgentArrive*;
- movimento de um agente para outra agência: as ações a serem tomadas quando este evento ocorre são especifidas no bloco de instruções *onAgentLeave*;
- mensagem descartada: este evento ocorre quando a agência possui uma mensagem para ser entregue a um agente que não se encontra nesta agência. As ações a serem tomadas na ocorrência deste evento são especificadas no bloco de instruções *onMessageDiscard*;
- início de execução de uma agência: as ações a serem tomadas no momento de inicialização de uma agência são especificadas no bloco de instruções *onAgencyStart*.

Estes eventos representam situações típicas onde sistemas sendo especificados podem ter comportamentos próprios a serem determinados.

Na ocorrência dos três primeiros eventos, podemos operar sobre uma variável do tipo *AgencyMsg*. *AgencyMsg* é um tipo embutido da extensão por nós proposta e carrega informações sobre o evento específico que ocorreu, como, por exemplo, no caso do evento *onAgentArrive*, informações sobre o agente que chegou e de qual agência ele vem.

No bloco *onAgencyStart*, o comando *start* pode ser utilizado para disparar a execução de agentes na agência. Este comando tem a sintaxe mostrada abaixo:

```
start <nome do agente>;
```

No comando acima, o nome do agente a ser criado é passado como parâmetro (a sintaxe para a especificação de agentes é mostrada abaixo).

Como indicado na seção 2, uma agência possui um comportamento genérico e um específico. As ações de tratamento dos eventos correspondem à especificação do comportamento específico da agência. O comportamento genérico da agência é incorporado a cada agência de maneira implícita pelo nosso sistema. O comportamento genérico corresponde a funcionalidades para implementar o transporte de agentes e um mecanismo básico de comunicação entre agentes.

Cada agência especificada é mapeada em um conjunto de processos Promela pelo tradutor.

Agentes

A primitiva *agent* é usada para se especificar o comportamento de processos que modelam agentes. Uma especificação de um agente possui a seguinte estrutura geral:

```
agent <nome do agente>{
  <comandos>
};
```

Os comandos presentes na parte indicada com `<comandos>` definirão o comportamento do agente. Além das instruções de Promela, três primitivas especiais podem ser usadas, *move*, *send* e *receive*. A instrução *move*, quando executada, resulta na migração do agente de uma agência para outra. Um comando *move* possui a seguinte estrutura:

```
move (<nome da agencia destino>,
      <resultado da operacao>)
```

Durante uma operação de movimento, dependendo da propriedade da rede de comunicação em relação a falhas, a migração pode ocorrer com sucesso ou não. Se a migração ocorrer com sucesso, a execução do agente continuará a ser realizada na agência destino. Se a requisição de movimento for mal sucedida, o agente permanecerá na sua agência de origem (o resultado da operação é informado no parâmetro `<resultado da operacao>`). As primitivas *send* e *receive* serão descritas mais abaixo.

A declaração de um agente será mapeada em um processo Promela. No momento de sua criação, um canal local à agência onde foi criado é passado ao agente. Este canal será usado pelo agente para se comunicar com outros agentes. Quando o agente se move, ele passa a ter acesso a um canal local da agência destino e deixa de ter acesso ao canal passado a ele pela agência origem. O agente inicia a operação de *move* enviando uma mensagem de requisição de movimento pelo seu canal de comunicação, indicando para qual agência quer se mover.

Comunicação e a Rede de Troca de Mensagens

Cada especificação possui, implicitamente, um componente que representa a rede de comunicação entre agentes. Uma mensagem enviada de um agente a outro é inicialmente enviada à rede que a encaminha à agência onde o agente recebedor deve se encontrar.

Esta rede poderá se comportar de forma confiável ou não. No caso de uma rede confiável, uma mensagem enviada será eventualmente entregue e uma migração de um agente sempre termina com sucesso. No caso da rede não confiável, poderá haver omissão de mensagens e a operação de movimento de um agente poderá terminar sem sucesso. Ao se fazer uma especificação, informa-se o tipo de comportamento a ser adotado pela rede, utilizando-se as instruções:

`network reliable`
ou
`network unreliable`

Estas instruções devem aparecer como a primeira instrução do arquivo com a especificação do sistema. Elas afetam os canais de comunicação entre agentes. A comunicação entre um agente e uma agência onde o agente se encontra é sempre confiável. Todos os canais se comportam de forma FIFO (*first-in, first-out*), ou seja, mensagens enviadas antes são consumidas antes.

A troca de mensagens entre agentes é especificada com a primitiva *send*. A sintaxe desta primitiva é:

```
send (<agente>, <agencia>, <mensagem>)
```

Com a primitiva acima, a mensagem `<mensagem>` é enviada ao agente `<agente>` na agência `<agencia>`. O envio da mensagem é feito sem transparência de localização, por isso para se enviar uma mensagem a um agente, o agente emissor deve indicar a agência onde o agente recebedor deve estar. Quando a mensagem enviada chegar na agência destino, a agência recebe a mensagem e a entrega ao agente. Se o agente não estiver na agência especificada quando esta for entregar a mensagem, um evento *onMessageDiscard* é gerado. Como indicado acima, a agência pode especificar ações específicas para serem executadas quando um evento deste tipo for gerado e, com isto, executar ações específicas nesta situação (como, por exemplo, tentar entregar a mensagem ao agente em sua localização atual).

Para ler uma mensagem da rede, o agente deve fazer uso da primitiva *receive*, que possui a seguinte sintaxe:

```
receive (<tipo>, <emissor>, <localizacao>,
         <mensagem>)
```

onde o parâmetro `<tipo>` irá indicar o tipo da mensagem lida, o parâmetro `<emissor>` conterá o nome do agente que enviou a mensagem, o parâmetro `<localizacao>` indica a agência onde o agente emissor estava localizado no momento que enviou a mensagem e o parâmetro `<mensagem>` conterá a mensagem em si.

A rede de comunicação é mapeada pelo tradutor em nosso sistema em um processo Promela. Modelando-a desta maneira, ao invés de termos um canal entre cada par de agências, procuramos diminuir a quantidade de canais das especificações, tornando menor a quantidade de recursos necessários para se fazer a verificação ou simulação sobre o SPIN.

4 Exemplo

Como previamente ressaltamos, no modelo por nós adotado para representar mobilidade em Promela o emissor de uma mensagem deve indicar, além do agente a que a mensagem se destina, a localidade onde este deve se encontrar. Nesta seção será apresentada parte da especificação de uma infra-estrutura de comunicação que suporta transparência de migração, ou seja, uma mensagem pode ser enviada a um agente, sem que o emissor tenha que especificar o local onde o agente recebedor se encontra e, caso este último migre, ainda assim o sistema tentará entregar a mensagem a ele. A infra-estrutura especificada nesta seção é uma variação da infra-estrutura apresentada em [16], especificada em Nomadic Pict.

Esta infra-estrutura baseia-se na existência de um único servidor de consultas *qs* e de um *daemon* em cada uma das agências. O *qs* contém uma tabela que mantém, para cada agente existente no sistema, a sua localização. Cada um dos *daemons* mantém um *cache* local desta tabela, com um subconjunto dos agentes e a localização presumida de cada um deles. Quando um agente é criado em uma agência, a informação sobre sua localização é registrada no *qs* e no *daemon* desta agência. Quando um agente se move, sua informação de localização é atualizada no *qs* e no *cache* do *daemon* da agência destino. O *cache* do *daemon* da agência de onde partiu não é alterado. O *qs*, portanto, mantém informações atualizadas sobre a localização dos agentes. Os *daemons*, no entanto, podem ter informações desatualizadas.

Quando um agente deseja enviar uma mensagem a outro agente, ele solicita ao *daemon* da agência onde se encontra que o faça. O *daemon* irá verificar se possui o registro da localização do agente destino em seu *cache* local. Caso não tenha, ele repassa a mensagem ao *qs* para que ele faça a entrega. Caso tenha, ele envia a mensagem para o *daemon* da agência onde ele acredita que o agente destinatário está. Ao receber a mensagem, este *daemon* tenta entregar a mensagem ao agente. Caso o agente esteja na agência, a mensagem será entregue. Caso contrário, a mensagem será também repassada ao *qs* para encaminhá-la ao agente.

Quando o *qs* recebe uma mensagem para reencaminhar a um agente, ele consulta sua tabela e irá enviar a mensagem para o *daemon* da agência correspondente a este agente. Além disto, ele envia uma mensagem ao *daemon* da agência onde o agente emissor da mensagem se encontra, atualizando a informação de localização do agente recebedor. Este procedimento pode se repetir, caso o agente recebedor se migre durante o processo.

A primitiva para o envio de uma mensagem com transparência de migração é:

```
inline highSend(pcurrentlocal,pmensagem){
  send($daemon$,pcurrentlocal,pmensagem);
};
```

onde a agência local *pcurrentlocal* e a mensagem *pmensagem* são passadas como parâmetro. Observe que a implementação desta primitiva consiste apenas em enviar a mensagem para o *daemon* local (`$daemon$`). A notação `$<nome do agente>$` é usada, na versão atual do sistema, para se referir a um agente cujo comportamento foi especificado utilizando-se a primitiva `agent` com o nome `<nome do agente>`.

A especificação (parcial) do *daemon* é (as reticências indicam trechos da especificação não mostrados aqui):

```
01 agent daemon{
02  ...
03  do
04   ::receive(tipo,emissor,local,
05            msgPacket);
06     if
07      ::(msgPacket.tipo==message)->
08         getMap(map,msgPacket.agenteID,
09              local,resultado);
10        if
11         ::(resultado==FOUND);
12           msgPacket.agenciaID=local;
13           msgPacket.tipo=try_delivery;
14           send($daemon$,
15                msgPacket.agenciaID,
16                msgPacket);
17         ::(resultado==NOT_FOUND);
18           send($q$,$$qs$$,msgPacket);
19        fi;
20      ::(msgPacket.tipo==update)->
21        setMap(map,msgPacket.agenteID,
22               msgPacket.agenciaID)
23      ::(msgPacket.tipo==try_delivery)->
24        send(msgPacket.agenteID,
25             currentLocal, msgPacket);
26    fi;
27  od;
28 }
```

No trecho acima, o *daemon* recebe uma mensagem nas linhas 4 e 5. Esta mensagem pode ser uma mensagem a ser entregue a outro agente, sem o prévio conhecimento de onde ele se encontra (tipo *message*), uma mensagem a ser entregue a um agente local (tipo *try_delivery*) ou uma mensagem de atualização do *cache* do *daemon* (tipo *update*). As linhas 7 a 19 correspondem ao tratamento de uma mensagem do tipo *message*. O *daemon* verifica a existência de informação de localização sobre o agente recebedor em seu *cache* (linhas 8 e 9) e reenvia a mensagem ao *daemon* da agência correspondente ao agente recebedor (linhas 11 a 16) ou ao *qs* (linhas 17 e 18), dependendo se possui ou não uma entrada em seu *cache* para o agente (de acordo com o valor da variável `resultado`). Nas linhas 20 a 22, o *daemon* verifica se a mensagem é de atualização de seu *cache* e o atualiza. Nas linhas 23 a 25 o *daemon* verifica se a mensagem é para ser entregue a um agente local e tenta entregá-la. Observe que, caso este procedimento não obtenha sucesso, será gerado um evento *onMessageDiscard* nesta agência. O tratamento deste evento está mostrado abaixo, linhas 13 a 18, e consiste em reenviar a mensagem para o *qs* (`qs`) na linha 17 (na versão atual, a notação `$$<nome de agencia>$$` é utilizada para se referenciar a uma agência cujo comportamento foi especificado com a primitiva `agency` com o nome `<nome da agencia>`):

```
01 agency site1{
02   onAgencyStart{
03     start daemon;
04     start ag1;
05     ...
06     packet atualiza;
07     ...
08     atualiza.tipo = update;
09     atualiza.agenteID = $ag1$;
10     atualiza.agenciaID = $$site1$$;
11     send($daemon$,$$site1$$,atualiza);
12   };
13   onMessageDiscard(AgencyMsg e){
14     packet    pkt;
15     copyPacket(e.pinfo, pkt);
16     pkt.tipo=message;
17     send($q$,$$qs$$,pkt);
18   };
19 }
```

Nas linhas 2 a 12 acima, está especificado o comportamento da agência ao ser instanciada. Neste instante, é criada uma instância do *daemon* local (linha 3) e de um agente *ag1* (linha 4). Em seguida é atualizada a informação de localização de *ag1* no *daemon* local (linha 11).

A especificação do *qs* é apresentada abaixo:

```
01 agency qs {
02   onAgencyStart{
03     start q;
04   };
05 }
```

No trecho acima, é declarado o comportamento de uma agência *qs*, que, na sua inicialização, instancia o agente *q* (linha 3) que conterá a lógica relativa ao servidor de consultas. A especificação parcial do agente *q* é apresentada abaixo:

```
01 agent q{
02  ...
03  do
04   :: receive(tipo, emissor, local,
05             msgPacket);
06      if
07       ::(msgPacket.tipo==message)->
08         copyPacket(msgPacket,tryPacket);
09         tryPacket.tipo=try_delivery;
10         getMap(map, tryPacket.agenteID,
11              local, resultado);
12       if
13        ::(resultado==FOUND)->
14             tryPacket.agenciaID=local;
15          copyPacket(tryPacket,
16                     updPacket);
17          updPacket.seq=0;
18          updPacket.agenciaID=
19                   tryPacket.agenciaID;
20          updPacket.agenteID=
21                   tryPacket.agenteID;
22          updPacket.tipo=update;
23          send($daemon$,
24               tryPacket.agenciaID,
25               tryPacket);
26          send($daemon$,
27               updPacket.localID,
28               updPacket);
29       fi;
30     ::(msgPacket.tipo==migrate)->
31       setMap(map, msgPacket.agenteID,
32             msgPacket.agenciaID);
33    fi;
34  od;
35 };
```

Nas linhas 4 e 5, o agente *q* recebe uma mensagem, que pode ser de dois tipos: *message*, ou seja, uma mensagem a ser enviada a um agente quando o emissor não conhece ainda a localização do agente recebedor, ou *migrate*, que corresponde a uma solicitação de atualização da informação de localização de um agente que está se movendo. Nas linhas 7 a 29, o agente *q* trata uma mensagem do tipo *message*. Ele consulta a informação de localização deste agente (linhas 10 e 11), reenvia a mensagem para o *daemon* da agência onde o agente recebedor se encontra (linhas 14 e 23) e atualiza a informação de localização do agente recebedor no *cache* do *daemon* da agência do agente emissor (linhas 15 a 22 e 26 a 28). Nas linhas 30 a 32, o agente *q* trata uma mensagem do tipo *migrate*, ou seja, um agente está se movendo e sua informação de localização deve, portanto, ser atualizada em seu *cache*. Para que esta atualização possa ser feita, um agente deve utilizar a operação *highMove* para se mover. A especificação desta operação é apresentada abaixo:

```
01 inline highMove(pagenteID,pcurrentLocal,
02                 psite, presultado){
03   atomic{
04     move(psite,presultado);
05     packet updPacket, mvPacket;
06     mvPacket.tipo=migrate;
07     mvPacket.seq=0;
08     mvPacket.agenteID=pagenteID;
09     mvPacket.agenciaID=psite;
10     copyPacket(mvPacket,updPacket);
11     updPacket.tipo=update;
12     send(daemon,pcurrentLocal,updPacket);
13     send($q$,$$qs$$,mvPacket);
14   }
15 };
```

Na especificação acima, após o agente se mover (linha 4), a informação sobre a localização do agente é atualizada no *daemon* local da nova agência (linha 12) e no *qs* (linha 13). Observe que as instruções seguintes à instrução *move* (linha 4) já são executadas no contexto da nova agência (ou seja, a partir deste ponto, o agente já terá migrado).

5 Trabalhos Relacionados

Muitos formalismos foram propostos para a especificação de sistemas móveis, como π-calculus [12], Mobile Ambients [3], Distributed Join Calculus [4], Nomadic π-calculus [14], entre outros. Os formalismos diferem na maneira como modelam mobilidade ou na habilidade de explicitamente modelar aspectos específicos do desenvolvimento de aplicações/protocolos móveis. Linguagens de programação como Pict [2] e Nomadic Pict [16] são baseadas em π-calculus e Nomadic π-calculus, respectivamente. A ferramenta Mobility Workbench [15] faz manipulação e análise de sistemas móveis especificados em (polyadic) π-calculus.

No π-calculus a mobilidade é representada implicitamente através da mudança na configuração dos canais de comunicação entre os processos. A localização de um processo em um dado instante pode ser especificada pela configuração de conexões (canais de comunicação) que ele tem com outros processos naquele instante. Nos Mobile Ambients, Nomadic π-calculus e Distributed Join Calculus, noções explícitas de agentes e localizações existem. Mobile Ambients explicitamente levam em consideração domínios administrativos e possuem primitivas explícitas para modelar permissões. O Nomadic π-calculus é uma variação do π-calculus, em que construtores simples de baixo nível para representar mobilidade são definidos. Este formalismo está sendo usado para analisar e desenvolver formas de comunicação transparente entre agentes móveis. O Distributed Join Calculus é um formalismo que incorpora um modelo de falha simples, que define uma semântica para os efeitos de falha nas localizações. A linguagem Nomadic Pict tem uma arquitetura em dois níveis. O primeiro nível consiste de primitivas dependentes da localização, incluindo comunicação e migração de agentes. O segundo nível estende o nível anterior com comunicação independente de localização.

A extensão que estamos propondo de Promela utiliza noções explícitas de agentes e localizações (como em Mobile Ambients, Nomadic π-Calculus e Distributed Join Calculus), e a localização de um agente é especificada através de um canal entre o agente e sua agência (de modo análogo a π-Calculus). De forma similar a Join Calculus, a nossa extensão possui também uma semântica simples de falhas. Aspectos de segurança, no entanto, não estão presentes na nossa extensão. De maneira semelhante a Nomadic Pict, nós introduzimos primitivas de comunicação dependentes de localização com as quais infra-estruturas para comunicação independente de localização podem ser expressas.

O trabalho apresentado neste artigo é desenvolvido no contexto dos projetos ForMOS, IQ-Mobile e ARGO. No contexto dos dois primeiros projetos está se desenvolvendo também um ambiente para especificação, simulação e geração de código de sistemas móveis baseado em Gramáticas de Grafos Baseadas em Objetos (OBGG), proposta em [7]. Espera-se, com o desenvolvimento das duas abordagens, chegar-se a uma metodologia para o desenvolvimento de aplicações móveis corretas.

6 Conclusão

A utilização de métodos formais na especificação de sistemas tem sido uma abordagem adotada para se especificar os requisitos de uma aplicação e garantir que a especificação do sistema atenda a certas propriedades. A partir destes resultados, procura-se, através da aplicação de uma metodologia apropriada, obter implementações mais confiáveis.

Neste trabalho, partimos do ambiente SPIN, que contém ferramentas para especificação, simulação e verificação de sistemas concorrentes síncronos e assíncronos, e estendemos a sua linguagem de especificação (Promela) com primitivas para modelar sistemas móveis. Criamos um modelo de mobilidade, onde representamos explicitamente o conceito de agente, agência, comunicação entre agentes e movimento de agentes. Implementamos este modelo sobre Promela, e, com esta abstração, permitimos que o desenvolvedor de um sistema baseado em agentes móveis utilize o SPIN para especificar, verificar e simular o seu sistema utilizando primitivas que modelam diretamente componentes presentes em sistemas baseados em agentes móveis.

Para demonstrar a expressividade da linguagem Promela estendida para a especificação de sistemas móveis, apresentamos na seção 4 uma variação da especificação da infra-estrutura proposta em [16]. Em [16] os autores apresentam uma especificação desta infra-estrutura utilizando a linguagem Nomadict Pict. Comparando-se as duas especificações, consideramos que a especificação feita com a extensão de Promela proposta é mais próxima de uma linguagem de programação de uso geral. Com isto, acreditamos que esta extensão pode facilitar a produção de especificações por profissionais sem experiência prévia em técnicas de especificação formal, mas que possuem conhecimentos de linguagens de programação.

O foco deste artigo foi na apresentação da extensão criada sobre Promela para a especificação de sistemas baseados em agentes móveis. Os aspectos de verificação e simulação, através da análise e uso das ferramentas do SPIN para estes fins (como a forma como expressões de propriedades a serem verificadas devem ser especificadas, por exemplo), estão sendo atualmente investigados. Em particular, no estado de implementação em que o sistema se encontra, embora o especificador possua primitivas que expressam diretamente componentes de aplicações baseadas em agentes móveis, o resultado da simulação fornecido pelo SPIN é baseado no mapeamento gerado pelo tradutor. A escolha da forma como mapear componentes tem sido feita de modo a que os resultados da simulação não sejam prejudicados por este fato. Alternativas estão sendo avaliadas, no entanto, para tornar a apresentação destes resultados mais próxima do nível de abstração utilizado pelo especificador.

Referências

[1] J. Baumann, F. Hohl, K. Rothermel, and M. Strasser. Mole - concepts of a mobile agent system. Technical Report 1997/15, Universität Stuttgart, Fakultät Informatik, 1997.

[2] B.C.Pierce and D.N.Turner. Pict: A programming language based on the pi-calculus. Technical Report 476, CSCI, Indiana University, March 1997.

[3] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Lecture Notes in Computer Science*, 1378, 1998.

[4] C.Fournet, G.Gonthier, J.-J.Levy, L.Maranget, and D.Rémy. A calculus of mobile agents. *Lecture Notes in Computer Science*, 1119:406–421, August 1996.

[5] Rocco de Nicola, Gian Luigi Ferrari, and Rosario Pugliese. Klaim: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, May 1998.

[6] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[7] Fernando Luís Dotti and Leila Ribeiro. Code mobility in open systems: A formal approach. In *Parallel and Distributed Processing Techniques and Applications*, 2000.

[8] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[9] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[10] IKV++. Grasshopper programmer's guide, 1999.

[11] ObjectSpace Inc. Voyager technical overview, 1997.

[12] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[13] Flávio M. Assis Silva. *A Transaction Model based on Mobile Agents*. PhD thesis, Technische Universität Berlin (Technical University Berlin), Berlin, Germany, 1999.

[14] Asis Unyapoth. *Nomadic π-Calculi: Expressing and Verifying Communication Infrastructure for Mobile Computation*. PhD thesis, pembroke college, University of Cambridge, March 2001.

[15] Björn Victor. The mobility workbench user's guide: Polyadic version 3.122. Technical report, October 1995.

[16] Pawel T. Wojciechowski and Peter Sewell. Nomadic pict: Language and infrastructure design for mobile agents. In *Proceedings of ASA/MA'99 - First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents*, October 1999.