

Extensão de Promela para Especificação de Falhas em Sistemas Baseados em Agentes Móveis

Daniel Aguiar da Silva, Aline Maria Santos Andrade, Flávio M. Assis Silva

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
40.170-110 Salvador – BA

daguiar@im.ufba.br, {aline, fassis}@ufba.br

Resumo. *Este trabalho propõe a adição de falhas ao modelo de extensão da linguagem Promela desenvolvido pelo LaSiD/UFBA para a especificação de sistemas baseados em agentes móveis. Tal modelo de extensão incorpora primitivas para a especificação de agente, do ambiente onde executa (as agências), das operações de envio e recepção de mensagens e de movimento. A adição de falhas consiste na introdução de primitivas para a especificação de falhas de agentes e agências (falhas do tipo crash e de omissão de mensagens), possibilitando a utilização do Spin na especificação, verificação e simulação de sistemas de agentes móveis na presença de tais falhas.*

Abstract. *An extension of Promela was proposed by LaSiD /UFBA for the specification of mobile agent systems. In this extension primitives for specifying mobile agents, agencies (the logical environment where mobile agents execute), the exchange of messages between agents, movement actions and omission failures on communication channels were defined. In this work, we describe primitives that we added to this extension for the specification of agent and agency (crash) failures and for modeling faults when an agent or agency tries to send or receive a message.*

1. Introdução

Os sistemas distribuídos são utilizados de forma ampla nas soluções computacionais atualmente como, por exemplo, sistemas de transações bancárias, sistemas de empresas aéreas de vendas de passagens, entre outras. Falhas em sistemas como estes pode causar perdas que vão desde um simples trabalho a enormes prejuízos financeiros.

Técnicas para implementação de tais sistemas vêm evoluindo com o surgimento de novas soluções, que proporcionam um melhor aproveitamento dos recursos da rede, como é o caso dos sistemas baseados em agentes móveis [12]. Um agente móvel é um componente de *software* responsável por atividades específicas do sistema, que é capaz de migrar de forma autônoma entre os nós da rede durante sua execução.

Além da complexidade da regra de negócio do *software* que compõe um sistema distribuído, há também a complexidade de interação entre os processos, que devem se comunicar via canais de comunicação e cooperar entre si sob limitações da rede. A complexidade destes sistemas aumenta quando executam em ambientes abertos, como a *Internet*, onde há grande heterogeneidade tanto de *software* como de *hardware*. A

natureza das falhas que podem ocorrer nesses sistemas pode ser a mais variada, podendo ocorrer no *hardware* ou no *software*. Por isso, estudos têm sido feitos no intuito de se desenvolver sistemas tolerantes a falhas, ou seja, que provejam serviços apesar da ocorrência de falhas.

O desenvolvimento de sistemas atualmente utiliza métodos, em geral não formais, e mecanismos que não garantem a sua correção. Apesar dos esforços feitos para alcançar bons níveis de qualidade através de processos e normas de certificação, os *software* produzidos, em geral, contêm erros, tanto de especificação, quanto de implementação. Por isso o processo de manutenção tem sido um processo extremamente caro. A manutenção tem sido o processo mais longo no ciclo de desenvolvimento, levando mais tempo que o restante das etapas.

Garantir que o sistema atende aos seus requisitos, ou seja, que se comporte da maneira especificada, é uma tarefa complexa, principalmente se o sistema atua em ambientes onde situações adversas como falhas são comuns, como, por exemplo, na *Internet*.

Métodos formais têm sido aplicados no intuito de dar uma base matemática ao desenvolvimento de sistemas. Esses métodos utilizam modelos matemáticos que fazem uso de sintaxe e semântica bem definidas [8, 9]. Assim, técnicas de verificação formal podem ser aplicadas para garantir correção no processo de desenvolvimento.

Os métodos formais podem ser aplicados em qualquer etapa do ciclo de desenvolvimento do *software*, tanto no processo de especificação, para verificar se o *software* atende aos seus requisitos, como no processo de implementação, para concluir acerca da correção da implementação em relação à especificação e também na geração de casos de testes. O encontro de soluções mais elegantes, simples e eficientes [9] também é um possível resultado obtido através da aplicação de metodologia formal.

Um paradigma dos métodos formais muito utilizado é a verificação de modelos (*model checking*). Através da verificação de modelos, a verificação automática de propriedades do sistema é possível. Para isto é gerado um modelo da aplicação em um sistema de transição finito e especificadas as propriedades, normalmente em lógica temporal, que são verificadas sobre o modelo.

Uma ferramenta bastante utilizada para verificação de modelos é o SPIN (*Simple Promela Interpreter*) [7, 14]. Além de especificar modelos através da linguagem Promela (*Process Meta Language*), o SPIN faz simulação e verificação de propriedades através de um verificador de modelos que utiliza Lógica Temporal Linear (LTL) clássica [10, 11, 13].

A linguagem Promela provê o formalismo necessário para descrição do comportamento do sistema através de uma maior abstração em relação às linguagens de programação. Pelo fato de ter uma sintaxe semelhante à de uma linguagem de programação convencional, como C, Promela facilita a especificação formal por pessoas que não possuem familiaridade com o formalismo matemático, mas que têm conhecimento de linguagens de programação.

A fim de possibilitar verificação de propriedades de sistemas baseados em agentes móveis, em [3] é proposta uma extensão à linguagem Promela, em que foi desenvolvido um modelo que provê as primitivas e as funcionalidades básicas de agentes e agências com suporte a mobilidade, não existentes em Promela.

O foco deste trabalho é especificar sistemas baseados em agentes móveis na presença de falhas. Para tanto, propõe a adição de falhas ao modelo criado em [3], possibilitando a especificação formal de sistemas baseados em agentes móveis na presença de falhas do tipo *crash* e falhas de omissão de envio, de recepção e de omissão geral. Através do Spin visa prover um ambiente para especificação, verificação e simulação para estes sistemas.

Este artigo está estruturado da seguinte maneira: Na seção 2 apresentamos os conceitos de falha, erro e defeito e também a classificação de falhas. Na seção 3 apresentamos uma breve introdução ao verificador de modelos Spin e à Promela. Na seção 4 apresentamos a extensão de Promela para especificação de sistemas de agentes móveis desenvolvida em [3]. Na seção 5 apresentamos a adição das primitivas de falhas à Promela Estendida e sua implementação. Na seção 6 apresentamos um exemplo simples de uma aplicação para ilustrar o funcionamento do ambiente proposto. A seção 7 conclui o artigo apresentando trabalhos relacionados e futuros.

2. Conceitos de Falha, Erro e Defeito

Um sistema seguro deve continuar operacional e manter a integridade das informações, mesmo após a falha de um ou mais processos. Para isso deve possuir mecanismos que detectem a falha de tais processos e tomar as medidas cabíveis para contornar a falha e conseqüente erro que o processo falho causará à operacionalidade do sistema. Essas medidas podem incluir redundância dos dados e substituição dos componentes faltosos por outros operacionais, por exemplo.

Uma falha pode ser de diversas naturezas, como uma operação ilegal, uma finalização inesperada de um componente, omissões de envio ou recepção de mensagens ou até comportamentos arbitrários. Essas falhas podem ocorrer, por exemplo, devido à fadiga e interferência (componente físico), ou até por falha algorítmica (componente de *software*) [6]. A ocorrência de uma falha no sistema ocasiona o erro. O erro é caracterizado pelo estado incorreto do sistema e acarreta a apresentação de defeitos.

O defeito é definido como desvio de especificação. Um sistema com defeito apresenta comportamentos que não refletem comportamentos especificados, não estando apto a prover os serviços desejados. O defeito não pode ser tolerado, por isso deve-se evitar que o sistema apresente o defeito. O defeito ocorre a partir do processamento do sistema com estado interno incorreto ou inconsistente [1, 6].

2.1. Classificação de Falhas

A classificação das falhas pode ser feita baseada em como um componente se comporta em caso de falha. Assim, ao falhar, algumas suposições podem ser feitas e então conclusões sobre o componente podem ser tiradas.

Os tipos, ou classes de falhas são definidos como se segue:

Falhas *crash* – A falha *crash* causa a perda do estado interno do componente do sistema ou o término prematuro de sua execução. O componente não continua a execução com estado incorreto, deixando assim de funcionar.

Falhas de omissão – O componente deixa de responder a algumas entradas. Pode haver omissão de envio na qual o componente falha ao tentar enviar um conjunto de

mensagens, ou omissão de recepção, na qual o componente falha ao tentar receber um conjunto de mensagens. As falhas que envolvem os dois tipos de omissão supracitados são as falhas de omissão geral.

Falhas de tempo – As falhas de tempo são também chamadas falhas de desempenho, onde um componente produz suas respostas prematuramente ou mais tarde do que o esperado.

Falhas bizantinas – As falhas bizantinas abrangem todos os grupos citados anteriormente, adicionando qualquer outro tipo de falha existente.

3. O verificador de modelos Spin e a linguagem Promela

O Spin (*Simple Promela Interpreter*) [7, 14] é um verificador de modelos muito utilizado na simulação e verificação de sistemas concorrentes, tanto síncronos como assíncronos. Os modelos de verificação do Spin visam a prova da correção da interação de processos, abstraindo detalhes de implementação de cada processo.

A linguagem de especificação utilizada pelo Spin é a Promela (*Process Meta Language*) [14]. Promela é uma linguagem voltada para especificação de sistemas concorrentes e é utilizada para especificação de protocolos e sistemas distribuídos. Provê o formalismo necessário para produzir especificações rigorosas através de uma maior abstração em relação às linguagens de programação. O fato de se assemelhar a uma linguagem de programação convencional, como C, facilita a especificação formal por pessoas que não possuem familiaridade com o formalismo matemático, mas que têm conhecimento de linguagens de programação.

Uma especificação Promela consiste na definição de um ou mais protótipos de processos. A partir desses protótipos vários processos podem ser instanciados. A comunicação entre os processos é feita através de canais de comunicação que podem ser síncronos ou assíncronos.

Um processo é modelado em Promela a partir da palavra reservada “*proctype*”. Assim, tal processo pode ser instanciado, através da palavra reservada “*run*”, para que possa executar. Inicialmente um único processo chamado “*init*”, de especificação obrigatória, é executado. A partir de tal processo os demais processos devem ser instanciados, criando o ambiente desejado na especificação para verificação e simulação. A seguir um exemplo ilustrativo:

```
1 proctype processo1() {
2   bool x;
3   ...
4   }
5
6   init {
7     run processo1()
8   }
```

Figura 1. Exemplo de especificação em Promela.

Além da notação para a definição de processos e variáveis, Promela conta com estruturas de repetição e seleção baseadas nos comandos de guarda de Dijkstra [2].

Um comando de guarda é dado pela seguinte estrutura:

<guarda> -> <comando>

<guarda> é uma expressão booleana e <comando> é dado por uma ou mais operações que refletem na alteração do estado do sistema. As operações do comando serão efetuadas caso a guarda seja verdadeira. Quando há mais de uma condição de guarda, e mais de uma é verdadeira, a escolha do comando a ser executado será não determinista.

4. Extensão de Promela para Especificação de Sistemas de Agentes Móveis

Devido ao fato de sistemas baseados em agentes móveis serem distribuídos e estes agentes terem a autonomia de migração entre pontos da rede, tais sistemas dispõem de toda a complexidade discutida anteriormente. Apesar de pesquisas diversas com propostas de formalismos que expressam mobilidade terem sido desenvolvidas, e, baseadas nestas, linguagens e ferramentas para verificação terem sido definidas, nenhum trabalho integra ambientes com ferramentas de suporte à análise de correção, como verificador de modelos, simulador e gerador de código [3].

O trabalho proposto em [3] enfoca o ponto apresentado acima, procurando prover, através do Spin, tal ambiente. Para tanto, propõe a extensão da linguagem Promela para suporte à especificação de sistemas baseados em agentes móveis, especificando primitivas de agentes, do ambiente onde executam, das ações de movimento, da comunicação entre agentes e do comportamento dos canais de comunicação em relação a falhas.

As primitivas utilizadas para modelar agentes e agências são “*agent*” e “*agency*”, respectivamente. Ao utilizar tais primitivas em uma especificação, um tradutor as mapeia em processos Promela, utilizando a notação oferecida pela linguagem. Assim, o Spin permanece apto a interpretar o código gerado.

```
1 agency agencia1 {
2   /* código da agência */
3 }
4 agent agente1 {
5   /* código do agente */
6 }
```

Figura 2. Exemplo da utilização das primitivas de agentes e agências.

```
1 proctype agencia1() {
2   /* código da agência */
3 }
4 proctype agente1() {
5   /* código do agente */
6 }
```

Figura 3. Código Promela gerado a partir das primitivas *agency* e *agent*.

A comunicação entre agente e agência se dá através da troca de mensagens via canais de comunicação síncrona. Não são definidos limites de tempo de processamento relativos entre agentes e agências.

A comunicação entre agentes deve ser feita através das primitivas de envio e

recepção de mensagens “*send*” e “*receive*”, respectivamente, com especificação da localização do agente destinatário da mensagem ao seu envio. Porém, pode-se utilizar o modelo para especificar formas de entrega de mensagens com transparência de migração. Essa comunicação é assíncrona, assumindo-se que não são conhecidos limites de tempo para transmissão de mensagens. Assíncrona também é a comunicação entre agências.

Os canais definidos neste modelo podem ser confiáveis ou não confiáveis. Os canais confiáveis fazem a eventual entrega das mensagens sem duplicação, corrompimento ou perda, enquanto que os canais não confiáveis podem perdê-las, porém, não podendo duplicá-las ou corrompê-las.

A ação de movimento de agentes é especificada através da primitiva “*move*”, onde é especificado o nome da agência de destino. O resultado da operação é fornecido em um segundo parâmetro, onde o agente é informado do sucesso da operação, ou de sua falha.

5. Adição de falhas ao modelo estendido de Promela

A forma aqui adotada para descrever um comportamento faltoso baseia-se no fato de que tal comportamento de um sistema é também um estado de transição possível deste, portanto, programável [2].

A representação de falhas em um modelo pode ser obtida através da transformação deste modelo em um segundo, onde variáveis virtuais (que não fazem parte do modelo original) são inseridas no intuito de adicionar o estado que represente o comportamento faltoso. Assim, através deste segundo modelo transformado, pode-se verificar de forma automatizada a correção de propriedades do modelo original na presença de falhas.

A adição de falhas ao modelo estendido de Promela consiste em adicionar primitivas que representem as falhas *crash* e as falhas de omissão de envio e de recepção. As falhas de omissão gerais deverão ser simuladas a partir da combinação das primitivas das falhas de omissão de envio e de omissão de recepção. A utilização destas primitivas implica na transformação do modelo original especificado pelo usuário em um segundo modelo, onde o comportamento do modelo original estará submetido à ocorrência de falhas, que farão parte dos estados de transição deste. Nestas condições, o sistema poderá ser verificado levando-se em consideração a ocorrência de falhas.

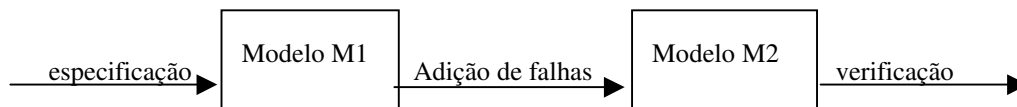


Figura 4. Especificação de sistemas na presença de falhas.

5.1. Especificação das falhas *crash* e de omissão

A adição das primitivas de falhas a uma especificação implica na adição de novos estados, relativos às falhas, aos componentes (agentes e agências) indicados para falhar. Estes estados farão parte do conjunto de estados acessíveis, podendo ser atingidos ou não em uma execução. Caso esses novos estados sejam atingidos durante a execução, o

componente passa a exibir tal comportamento, agindo de acordo com a especificação da falha que o afetou. A seguir apresentamos as primitivas de falhas adicionadas ao modelo proposto em [3], demonstrando a forma de utilização em uma especificação.

Para especificar a falha *crash* de uma ou mais agências deve-se usar a primitiva “*agencyCrash*” da seguinte forma:

agencyCrash <lista de agências>

O parâmetro <lista de agências> deve conter os nomes das agências que deverão falhar separados por vírgula.

Para especificar a falha *crash* de um ou mais agentes deve-se usar a primitiva “*agentCrash*” da seguinte forma:

agentCrash <lista de agentes>

O parâmetro <lista de agentes> deve conter os nomes dos agentes que deverão falhar separados por vírgula.

```
1 network reliable;
2 agencyCrash agencia1;
3 agentCrash agente2;
4 ...
5 agency agencia1 {código da agência} ...
6 agent agente2 {código do agente} ...
```

Figura 5. Exemplo de especificação de falha crash de agências e agentes.

A figura 5 faz uma pequena ilustração de uma especificação de falha *crash* de agente e agência. As primitivas relacionadas às falhas devem ser adicionadas no corpo da especificação, logo após a definição da característica da rede (linha 1). As linhas 2 e 3 definem que a agência “agência1” e o agente “agente2” falharão por *crash*, e que, durante sua execução, poderão exibir o comportamento faltoso, finalizando sua execução de forma prematura.

5.2. Implementação do modelo de falhas em Promela

O modelo adotado para representar a ocorrência das falhas consiste em adicionar variáveis booleanas, que indiquem a situação dos agentes e agências, sinalizando se estes exibirão ou não o comportamento faltoso. Comandos de guarda são inseridos no corpo da especificação Promela de agentes e agências escolhidos para falhar, no intuito de prover o processamento que simule a falha. A variável é testada pela guarda, prosseguindo a execução normal do componente ou não a depender do resultado da guarda.

Ao falhar por *crash*, o componente tem a sua execução interrompida de forma prematura, deixando de interagir com os outros componentes do sistema. Caso este componente seja uma agência, esta não mais gerenciará as mensagens enviadas de e para os agentes nela situados, deixando de prover o ambiente ideal para a execução destes agentes. Assim, os agentes que se localizam na agência no momento de sua falha deixam também de interagir com outros componentes do sistema.

Ao falhar por omissão, o componente afetado não finalizará sua execução de forma prematura, como no caso da falha *crash*, apenas deixará de enviar ou receber um conjunto de mensagens durante a sua execução. A falha da agência por omissão também influencia no andamento da execução do agente, uma vez que esta gerencia as trocas de mensagens dos mesmos. Assim, quando uma agência falha por omissão, as mensagens enviadas pelos seus agentes podem não ser entregues, assim como as mensagens enviadas aos seus agentes.

```
1 ...
2 bool agencialCrashes = false; /* Variável virtual de crash */
3 ...
4 proctype agencial { /* Definição de agência */
5 ...
6     (!agencialCrashes) -> /* Prossiga a execução */
7 ...
8     (agencialCrashes) -> /* Processo finaliza execução */
9 }
```

Figura 6. Ilustração da utilização de variável virtual e comandos de guarda.

Para ativar o comportamento faltoso processos relativos às falhas especificadas são criados e, após algum tempo não determinado de execução esses processos modificam o estado das variáveis virtuais adicionadas à especificação.

Como os canais utilizados na especificação são síncronos, para que as mensagens possam ser enviadas é necessário que haja um processo receptor para extração da mensagem do canal. Caso esta condição não seja atendida, o processo emissor da mensagem pode ficar em modo de espera aguardando indefinidamente a extração desta por um processo receptor. Esta situação influenciaria diretamente na verificação do sistema, uma vez que o processo que estiver em modo de espera fica impossibilitado de continuar sua execução normal, podendo causar o mesmo estado de espera em outros processos e até a parada de todo o sistema.

Para que isso não ocorra, quando os agentes e agências deixam de receber as mensagens a serem enviadas, os processos relativos às falhas passam a interceptar a recepção destas mensagens, tendo que executar um conjunto de atividades.

O processo de *crash* relativo à agência deve, além de receber e descartar as mensagens destinadas à respectiva agência, agenciar as tentativas de movimento dos agentes. Este gerenciamento é necessário, uma vez que ao tentar se mover, o agente envia mensagem de requisição à agência, e esta deve responder informando sobre o sucesso da operação. Neste caso, o processo de *crash* envia uma mensagem ao agente informando que a operação não pôde ser concluída. Já em relação ao agente, tal processo apenas recebe as mensagens destinadas ao seu agente, descartando-as.

O processo de omissão de recepção deve receber as mensagens destinadas à sua agência ou agente respectivo, descartando-as. Já o processo de omissão de envio não executa nenhuma atividade específica relacionada à troca de mensagens. Este processo limita-se a modificar o estado da variável virtual que indica a situação do componente com relação à falha de omissão de envio.

A figura 7 apresenta o código Promela de partes de um processo de falha *crash* de agência. O corpo do processo compreende o código que vai das linhas 1 a 12. A linha 5 contém parte da especificação de um comando de guarda, onde a guarda consiste em testar o valor da variável “agencia1Crashes” que vai definir o comportamento da agência “agencia1” em relação à falha *crash*. Na linha 8 o processo recebe as mensagens destinadas à agência e na linha 9 processa as requisições de movimento de agentes em relação à agência, emitindo mensagem sobre a falha da tentativa de movimento.

```

1 proctype agencia1Crash() {
2 ...
3 do
4 ...
5   ::(!agencia1Crashes) ->
6     agencia1Crashes = true;
7     do
8       ::receiveMsg(...)
9       ::processMove(...)
10    od
11 od
12 }

```

Figura 7. Código de processo crash de agência.

6. Funcionamento do ambiente

Para demonstrar o funcionamento do ambiente definimos um exemplo simples, no qual verificaremos uma propriedade do sistema na presença de falha *crash*. O modelo consiste de duas agências, chamadas “agencia1” e “agencia2”. Nestas agências estão localizados dois agentes, o “agente1” e o “agente2” respectivamente. Durante o ciclo de execução, o agente “agente1” envia mensagens ao agente “agente2”.

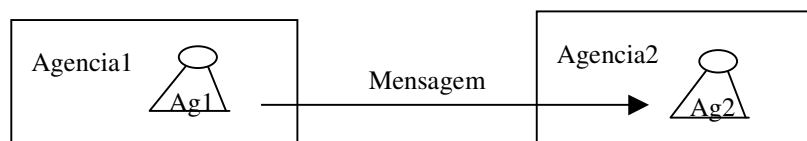


Figura 8. “agente1” envia mensagem a “agente2”.

Durante a execução a agência “agencia2” falha por *crash* e tem sua execução finalizada prematuramente. A partir de então, a comunicação entre os agentes do sistema estará comprometida, uma vez que a “agencia2” não mais proverá serviços de trocas de mensagens ao “agente2”. A figura abaixo ilustra o funcionamento do modelo com relação à falha *crash*.

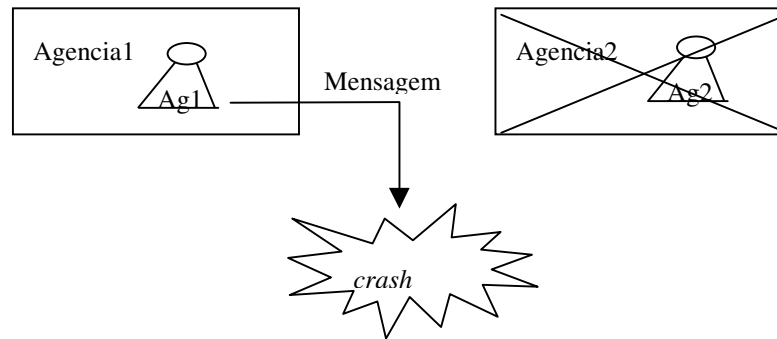


Figura 9. Processo de *crash* intercepta mensagens para “agencia2”.

A Figura 10 mostra parte do código da especificação do exemplo. Na linha 2 a utilização da primitiva “agentCrash” define que o “agente2” falhará por *crash*. O “agente1” é definido nas linhas de 7 a 14.

```

1 network reliable;
2 agentCrash agente2;
3 ...
4 bool sent = false;
5 bool received = false;
6 ...
7 agent agente1 {
8 ...
9   do
10    ...
11    ::send($agente2$, $$agencia1$$ , msgPacket) ->
12    sent = true;
13  od;
14 };
15 agent agente2 {
16 ...
17   do
18    ::receive(type, sender, locate, msgPacket) ->
19    received = true;
20  od
21 od;
22 };

```

Figura 10. Código do exemplo.

Para verificar a entrega de mensagens enviadas do “agente1” ao “agente2”, foram definidas variáveis booleanas para registrar o envio e a recepção da mensagem, linhas 4 e 5. Ao enviar uma mensagem ao “agente2” (linha 11), o “agente1” sinaliza o envio através da variável “sent” (linha 12). Ao receber a mensagem enviada pelo “agente1” (linha 18), o “agente2” sinaliza a operação através da variável “received”.

Definidas as proposições, através das variáveis “sent” e “received”, especificamos as propriedades que desejamos verificar através de fórmulas LTL, possibilitando a verificação automática pelo Spin. Neste exemplo, desejamos verificar se todas as mensagens enviadas pelo “agente1” serão entregues ao “agente2”. Assim, a propriedade que desejamos verificar é a entrega eventual de mensagens, que diz que toda mensagem enviada por um agente chegará, no futuro, ao agente de destino. Através

dos operadores \square e \diamond , que significam “sempre” e “no futuro” compomos a fórmula LTL a seguir:

$$\square (\text{sent} \rightarrow \diamond \text{received})$$

O Spin utilizará a negação da fórmula acima para verificar se essa propriedade é satisfeita no modelo, verificando que não é o caso. Isto ocorre porque em um determinado momento da execução, a variável “received” não se tornará verdadeira após o envio da mensagem, pois o processo de falha *crash* interceptará a mesma.

7. Conclusão

Trabalhos que objetivaram a especificação de falhas em um sistema são poucos. O modelo de falhas aqui desenvolvido é baseado no formalismo proposto em [2], onde o autor propõe uma solução para especificação formal de falhas, propiciando a utilização de técnicas de verificação automática de propriedades de sistemas na presença das mesmas.

Em [5] o objetivo é o mesmo proposto aqui, porém a linguagem de especificação formal é baseada em gramática de grafos, a linguagem OBG. São então disponibilizados um método e uma ferramenta de suporte à simulação, onde o modelo especificado em OBG é testado, tendo seu comportamento e desempenho observados. Porém, em [5] não são propostas ferramentas para verificar formalmente as propriedades de sistemas especificados em OBG.

Este trabalho adiciona falhas ao modelo proposto em [3], possibilitando a especificação, verificação formal e simulação do funcionamento de sistemas baseados em agentes móveis e protocolos de comunicação na presença de falhas através de uma ferramenta eficiente. O método utilizado consiste em modelar as falhas como uma transformação de um modelo inicialmente proposto em um segundo, onde as falhas fazem parte dos estados possíveis do modelo. A partir deste segundo modelo obtido por meio de tais transformações pode-se verificar a correção do modelo (inicial) na presença de falhas (modelo transformado).

As falhas suportadas pelo modelo aqui proposto são as *crash* e as falhas de omissão, que foram subdivididas em falhas de omissão de envio e recebimento de mensagens e as falhas de omissão geral. Devido à complexidade das falhas de tempo e das falhas bizantinas, um maior estudo será necessário para a elaboração do modelo de tais falhas, ficando para os próximos trabalhos.

Referências

- [1] Jalote, Pankaj. Fault tolerance in distributed systems. Prentice Hall, Englewood Cliffs, New Jersey, 1994.
- [2] Gärtner, Felix C. Specifications for Fault Tolerance: A Comedy of Failures. Technical Report TUD-BS-1998-03, Darmstadt University of Technology, Department of Computer Science, Germany, 1998.
- [3] Andrade, A. M.; Silva, F. A. e Barboza, F. J. R. Extensão da Linguagem Promela para Especificação de Sistemas baseados em Agentes Móveis. LaSiD, Departamento de Ciência da Computação, Universidade Federal da Bahia, 2002.
- [4] Andrade, A. M.; Silva, F. A.; Barboza, F. J. R. e Oliveira, R. A. Um ambiente para

especificação e verificação automática de aplicações baseadas em agentes móveis. LaSiD, Departamento de Ciência da Computação, Universidade Federal da Bahia, 2003.

- [5] Dotti, F. L.; Santos O. M.; Rödel E. T. On the Use of Formal Specifications to Analyse Fault Behaviors of Distributed Systems. Proceedings of the first Latin-American Symposium, LADC 2003. São Paulo, Brazil, 2003, páginas 341-360.
- [6] Weber, Taisy Silva. Tolerância a Falhas: conceitos e exemplos. PPGC – UFRGS.
- [7] Holzmann, G. J. The Model Checker Spin. IEEE Transactions on Software Engineering, vol. 23, No 5, Maio 1997.
- [8] Wing, Jeannette M. A Specifier's Introduction to Formal Methods. IEEE Computer 23(9):8-24, September 1990.
- [9] Déharbe David B.P.; Moreira Anamaria Martins; Ribeiro Leila; Rodrigues Vanderlei Moraes. Introdução a Métodos Formais: Especificação, Semântica e Verificação de Sistemas Concorrentes. Revista de Informática Teórica e Aplicada. Volume VII, número 1, p. 7-48. Setembro 2000.
- [10] Clarke, Grumberg e Peled. Model Checking. the MIT Press, Cambridge, Massachusetts, London, England.
- [11] Ben-Ari M. Mathematical Logic for Computer Science. Prentice Hall, 1993. ISBN 0-13-564139-X.
- [12] Silva, F. A. Agentes Móveis. LaSiD, Departamento de Ciência da Computação, Universidade Federal da Bahia, 1999.
- [13] D. Calvanese; G. Giacomo; M. Vardi. Reasoning about Actions and Planning in LTL Action Theories. Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza".
- [14] Spin Basic Manual, www.spinroots.com; último acesso em 15/11/2003.