Modelagem de Sistemas com Restrições Temporais em Redes de Petri Orientadas a Objetos

Fabrício Vale de A. Guerra

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Jorge César Abrantes de Figueiredo

(Orientador)

Dalton Dario Serey Guerrero

(Orientador)

Campina Grande, Paraíba, Brasil

GUERRA, Fabrício Vale de Azevedo

G934M

Modelagem de Sistemas com Restrições Temporais em Redes de Petri Orientadas a Objetos

Dissertação (Mestrado) - Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande, PB, Agosto de 2005.

108p. Il.

Orientadores: Jorge César Abrantes de Figueiredo
Dalton Dario Serey Guerrero

1. Redes de Petri
2. Simulação de Modelos
3. Análise de Desempenho

CDU – 519.711

"MODELAGEM DE SISTEMAS COM RESTRIÇÕES TEMPORAIS EM REDES DE PETRI ORIENTADAS A OBJETOS"

FABRÍCIO VALE DE AZEVEDO GUERRA

DISSERTAÇÃO APROVADA EM 30.08.2005

PROF. JORGE CÉSAR ABRANTES DE FIGUEIREDO, D.Sc
Orientador

PROF. DALTON DARIO SEREY GUERRERO, D.Sc
Orientador

PROFª JOSEANA MACÊDO FECHINE , D.Sc
Examinadora

PROF. ALEXANDRE CABRAL MOTA, Dr.
Examinador

CAMPINA GRANDE – PB

Resumo

Com o desenvolvimento cada vez maior de sistemas paralelos e distribuídos, a temporização de modelos tornou-se objeto de várias pesquisas. Em diversas áreas, modelos temporizados foram propostos com objetivo de modelar funcionalidades com restrições temporais e medir o desempenho dos sistemas em diferentes situações. Os modelos propostos normalmente são extensões de modelos existentes, nos quais identificou-se a necessidade do tratamento quantitativo de parâmetros de tempo. Este trabalho trata essencialmente de uma extensão temporizada para Redes de Petri Orientadas a Objetos (RPOO), formalismo que, a exemplo de tantos outros, apresentou deficiências quando aplicado à modelagem de sistemas de tempo real, por não prover meios intrínsecos de tratamento de informações temporais.

RPOO é um formalismo que se propõe a fazer com que várias das características das Redes de Petri e da Orientação a Objetos complementem-se e contribuam positivamente para a construção, simulação e validação de modelos. RPOO trata-se de uma linguagem de modelagem que pode ser vista tanto como uma extensão OO para Redes de Petri de alto nível quanto como um meio de prover semântica formal de concorrência a modelos OO. A proposta original do formalismo não oferece o tratamento de tempo em nenhuma destas duas perspectivas. A extensão temporizada que propomos promove a integração do tratamento temporal tanto em nível OO quanto em nível de redes de Petri. Como resultado, a extensão dá maior expressividade ao formalismo, no sentido de poder modelar e analisar com maior facilidade algumas classes de problemas (mais especificamente, problemas que apresentam restrições temporais).

Como complemento para o trabalho de definição formal, um protótipo de simulador com suporte à extensão temporizada é implementado. É apresentado, também, um estudo de caso, no qual o protótipo é utilizado na análise de desempenho do IP Móvel, protocolo cuja modelagem em RPOO puro identificou deficiências do formalismo em relação a aspectos temporais. Várias medidas de desempenho do protocolo — que não puderam ser derivadas em trabalhos com os modelos RPOO puros — são apresentadas como resultado do processo de simulação.

Abstract

Over the last years, with the crescent and rapid advance of distributed and parallel systems, timed models have become subject of several researches. Over different areas, different timed models have been proposed to model time constrained functionalities and promote system performance analysis in different situations. The proposed models are, in general, extensions of existing models, in which it was identified the need for coping with time parameters in a quantitative way.

Essentially, this work concerns the definition of a timed extension for *Redes de Petri Orientadas a Objetos* (RPOO), a formalism that, like many others, has presented problems when applied to the modelling of real time systems, due to the lack of inherent means to explicitly cope with time information.

RPOO is a formalism that integrates several features of both the OO and the Petri net languages in order to improve the production, simulation and validation of formal models. RPOO is a modelling language that can be seen either as an object-oriented extension for high-level Petri nets or as a way to give formal concurrent semantics for object-oriented models. Its original proposition does not provide inherent, quantitative time modeling in none of these two perspectives. Our timed extension integrates the time treatment in both the OO and the Petri net levels. As a result, the extension is more *expressive* than the original formalism proposition, in the sense that it makes easier the task of modeling certain classes of problems (the ones which present time constrained features).

As a mean to validate the formal definitions, a simulator prototype supporting some features of the proposed extension was implemented. Also, we present a case study, in which the prototype was applied in the analysis of Mobile IP, a protocol whose modelling in original RPOO identified deficiencies of the formalism when it comes to time treatment. Several protocol performance measures — which could not be obtained in previous works with original RPOO — are presented as results of automatic simulation process with the developed simulator prototype.

Agradecimentos

Agradeço a meus pais, irmãos e amigos, que me deram apoio pessoal inestimável durante o desenvolvimento deste trabalho, e aos professores e estudandes do Grupo de Métodos Formais da UFCG, pela excepcional convivência ao longo dos últimos dois anos.

Agradeço, especificamente, aos meus orientadores, bem como a Cássio Leonardo Rodrigues, José Amâncio dos Santos, Taciano Morais e Silva, Erica de Lima Gallindo e Pasqueline Lacerda, que gentilmente contribuíram de diversas formas para o bom andamento deste trabalho.

Gostaria de fazer uma menção especial aos familiares que se foram durante o período em que trabalhei nesta dissertação. Alberione José Guerra (1950-2004) e Maria da Conceição Dantas (1961-2003) certamente estariam felizes em me ver superar mais esta etapa de meus estudos.

Dedico este trabalho a Antônia Dantas de Almeida (1955-2004), minha primeira professora, e a seu filho, Lucas.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Contexto

Modelos temporizados Com o desenvolvimento cada vez maior de sistemas paralelos e distribuídos, a temporização de modelos, de uma forma geral, tornou-se objeto de várias pesquisas. Diversos modelos temporizados foram propostos com o intuito de especificar-se, analisar-se e medir-se o desempenho de tais sistemas, levando-se em consideração funcionalidades com restrições de tempo. Geralmente, os modelos propostos são extensões de modelos existentes (*timed automata [41], real-time statecharts [16], stochastic process algebra [19], etc.*), com suporte explícito (quantitativo) à modelagem de atividades onde o tempo é um fator relevante. Normalmente, os modelos ou formalismos originais são incrementados com a definição de *relógios*, variáveis temporizadas, novas regras de transição e mecanismos para a coleta de dados e análise de desempenho.

A obtenção de medidas de desempenho normalmente advém da simulação de cenários específicos, através de *modelos de simulação*, ou da análise de quadros gerais de execução, através de *modelos analíticos*. Encaixam-se nesta última abordagem os modelos *markovianos*, como as *Stochastic Petri Nets [32; 48]*, onde admite-se que todas as atividades representadas no modelo tomam uma quantidade de tempo cujo valor é exponencialmente distribuído. Modelos analíticos, nestes moldes, apresentam resultados mais precisos com respeito ao desempenho do sistema em análise. Sua aplicação, entretanto, acarreta uma série de problemas. Para que ela seja efetivada, os modelos precisam abstrair muitas características dos sistemas modelados, seja para possibilitar a geração de grafos que contemplem todos

os quadros de execução dos modelos, seja para que eles (os modelos) adeqüem-se a algum modelo matemático conhecido [47].

Já com respeito aos modelos de simulação, os resultados dizem respeito a situações específicas, não sendo, por isso, tão precisos quanto os resultados dos modelos analíticos. Entretanto, eles oferecem um maior grau de liberdade para que o modelador especifique funcionalidades cujo tempo de execução não se enquadre entre as distribuições estatísticas exigidas pelos modelos analíticos (tais restrições variam de acordo com o modelo analítico adotado). O fato das simulações não contemplarem *todas* as possibilidades de execução do sistema modelado é compensado, em parte, pela simulação exaustiva de uma grande variedade de cenários.

RPOO Redes de Petri Orientadas a Objetos [18] é um formalismo - surgido há alguns anos - voltado à modelagem de sistemas. RPOO propõe-se a fazer com que várias das características das redes de Petri e da Orientação a Objetos complementem-se e contribuam positivamente para a construção, simulação e validação de modelos. Desta forma, devido à sua perspectiva OO, o formalismo (RPOO) nos oferece vantagens como modularidade e sua aplicação pode facilmente inserir-se na fase de *design* da maioria das metodologias de desenvolvimento de software OO. Já numa perspectiva *redes de Petri* (RP), os modelos RPOO têm semântica formal de concorrência e paralelismo, adequando-se bem à modelagem de sistemas distribuídos sem a limitação de execução seqüencial inerente a modelos OO convencionais.

Conforme observado em outros trabalhos [9; 18], RPOO é uma linguagem de modelagem que pode ser vista tanto como uma extensão OO para redes de Petri de alto nível quanto como um meio de prover semântica formal de concorrência a modelos OO. Na prática, a linguagem permite a decomposição OO de modelos de redes de Petri de alto nível. Desta forma, a integração de diversas redes de Petri menores dá-se de acordo com uma semântica de análise e projeto orientados a objetos ao invés das composições hierárquicas convencionais.

Um modelo RPOO consiste de um conjunto de classes e suas redes de Petri correspondentes. As classes são descritas e relacionam-se umas com as outras nos moldes convencionais da maioria dos modelos OO - elas podem ser vistas como classes UML. As redes de Petri descrevem o comportamento dos objetos: para cada classe, há exatamente uma rede de

Petri correspondente no modelo RPOO. Desta maneira, uma classe e sua rede correspondente constituem um modelo formal para um conjunto de objetos.

Como em todo modelo OO, objetos podem interagir através de mensagens (tipicamente expressas como chamadas de métodos). A troca de mensagens entre objetos se dá através de *inscrições de interação* anotadas junto às transições das redes que descrevem as classes. As inscrições descrevem *ações* que são efetuadas pelo objeto quando uma transição dispara. Os vários tipos de ações que podem ser associadas às transições são: entrada de dados (denota o recebimento de uma mensagem por parte de um objeto), saída síncrona e assíncrona de dados (indica o envio de uma mensagem –chamada de método– de um objeto para outro), instanciação (usada para criar objetos), desligamento (desfaz ligações entre objetos), finalização (termina a execução de um objeto). Uma descricão mais apurada dos modelos RPOO e sua semântica pode ser encontrada no Capítulo 2.

No que diz respeito a aplicações, o formalismo RPOO foi utilizado na modelagem de diversos sistemas, desde o seu advento. Algumas destas aplicações tinham fins didáticos [18; 39], outras tinham por objetivo por o formalismo à prova, verificando sua aplicabilidade à modelagem de sistemas reais [7; 9]. Atualmente, há várias atividades em andamento relacionadas ao formalismo [10; 12].

Algumas ferramentas foram e estão sendo desenvolvidas para dar suporte à aplicação do formalismo, principalmente no que diz respeito à simulação e validação de sistemas de objetos. Em especial, tem-se um *Simulador de Sistemas de Objetos* (SSO) [11], ferramenta que provê algumas funcionalidades de simulação de configuração de objetos e foi usada nas primeiras análises de modelos RPOO. Ainda no que diz respeito a ferramentas, o verificador de modelos *Veritas* [37] propõe-se a verificar a corretude de modelos RPOO através da análise de espaços de estados em face de propriedades definidas na linguagem CTL (uma extensão *temporizada* da lógica proposicional). Recentemente, foi desenvolvido o JMobile [12], ferramenta de suporte a simulação e geração automática de espaços de estados RPOO. O JMobile tem sido utilizado em conjunto com o Veritas na verificação de modelos RPOO [10].

1.2 Caracterização do Problema

Uma das aplicações de RPOO efetivou-se na modelagem do IP Móvel (IPM ou *Mobile IP* [34]), protocolo que especifica mecanismos para que se mantenha a conectividade de dispositivos móveis quando estes migram por redes diferentes. Tempo é um fator importante no IPM: vários de seus mecanismos incluem *timeouts* e, além disso, há um grande interesse em medir-se a perda de datagramas[1] decorrente da mudança de rede por parte dos dispositivos móveis (vários modelos prestaram-se a tratar este último tópico [42; 1; 5]). RPOO não contempla adequadamente estes fatores [9]. Para isso, o formalismo deveria dar suporte à temporização de maneira explícita, oferecendo mecanismos próprios –*intrínsecos*– de tratamento de parâmetros temporais, para que o desenvolvedor não seja obrigado a *modelar* o tempo por conta própria.

Podemos observar que medir desempenho e modelar restrições de tempo são tarefas que podem ser contempladas por modelos não-temporizados, i.e., por modelos sem quantificação explícita de tempo. Entretanto, modelos desta natureza não oferecem os mecanismos adequados para estimativa de desempenho, forçando o desenvolvedor a *modelar* o tempo e os mecanismos pelos quais ele muda. Este processo naturalmente acarreta algumas dificuldades de modelagem, obrigando o desenvolvedor a validar sua estratégia particular de temporização, além de verificar as características inerentes ao sistema em si.

A modelagem do IPM evidenciou, num primeiro momento, a necessidade de temporização de mensagens assíncronas trocadas entre objetos (mensagens cujo envio e recebimento se dão em *interações* diferentes, em *instantes* diferentes). Durante o trabalho com o protocolo [9], notou-se que a transmissão de pacotes entre as várias entidades do protocolo deveria *tomar* algum tempo para que seu desempenho pudesse ser devidamente medido. Além disso, o protocolo descreve procedimentos onde dados devem expirar após um determinado período de tempo ou onde a retransmissão de mensagens é necessária após algum tempo de *espera*. O protocolo será analisado em mais detalhes no Capítulo 5.

O tempo poderia ser modelado com os recursos originais do formalismo, como ponderamos acima, mas isto acarretaria em uma série de dificuldades que tornam esta estratégia inviável. Considerando as características mais comuns dos modelos temporizados, para re-

[1] Vale salientar que a necessidade de parâmetros de tempo pode advir também da própria natureza de alguns sistemas e não do que queremos medir efetivamente: é o caso dos sistemas de tempo real.

alizarmos a análise de desempenho do IPM através de modelos RPOO *puros*, estes deveriam apresentar as seguintes características:

- Ter uma classe de instância única (*singleton*) que represente o tempo e que seja acessível a todas as outras entidades do sistema de maneira sincronizada;
- Todas as interações temporizadas entre objetos RPOO devem sincronizar com um objeto da classe que modela o tempo a fim de obter o tempo atual do modelo e enviar a mensagem com o devido *atraso*;
- A *pré-condição* das interações envolvendo o recebimento de mensagens de todos os objetos deve ser alterada. Tais interações precisam acessar o tempo atual do modelo para verificar se podem ou não ser efetuadas;
- Para que o tempo possa *avançar*, a classe que *modela o tempo* deve analisar *todas* as informações pertinentes aos objetos, certificar-se de que nenhuma delas pode ser utilizada no tempo atual, calcular o menor valor de tempo no qual alguma delas pode ser utilizada em alguma interação e, por fim, avançar o tempo do sistema para este valor. Para tal operação, seria necessária a análise de *todas* as pré-condições de *todas* as interações entre objetos;
- No caso de atividades internas temporizadas, as dificuldades para avaliar o progresso do tempo aumentam. Não só as pré-condições das atividades de interação entre objetos devem ser analisadas, mas também as pré-condições das atividades internas das redes de Petri que modelam os objetos. Na prática, isto requer acesso sincronizado a *todos* os lugares de *todas* as redes de Petri de *todos* os objetos do sistema.

Considerando que, em RPOO, cada objeto é uma *thread* (pode ser executado independentemente dos outros objetos) e que, por ser descrito por uma rede de Petri, pode apresentar internamente vários níveis de paralelismo, a sincronização de atividades necessária numa *modelagem de tempo* em modelos RPOO *puros* torna-se razoavelmente complexa e bastante danosa à compreensão/extensão dos modelos finais. Desta forma, a definição de uma extensão temporizada para RPOO é importante para a evolução do formalismo, expandindo sua aplicação a classes de sistemas cujas funcionalidades apresentem restrições temporais e permitindo avaliação de desempenho.

1.3 Trabalhos Relacionados

Apesar de RPOO não apresentar este tipo de extensão, temporização em si, trata-se de um aspecto bastante estudado no âmbito das redes de Petri.

Desde o seu advento, nos anos da década de sessenta, as redes de Petri têm mostrado-se adequadas à modelagem de sistemas que apresentam características como concorrência e paralelismo (aplicações bem sucedidas deste formalismo têm-se dado na modelagem e validação de protocolos de rede e sistemas distribuídos [23]).

Não raro, nestes tipos de modelos, surge a necessidade de medir-se o desempenho do sistema em questão. Esta é uma tarefa passível de ser levada a cabo por modelos não-temporizados de redes de Petri. Entretanto, modelos sem temporização, onde a noção de tempo apresenta-se de forma qualitativa (arcos dirigidos entre nós de um grafo de ocorrência, por exemplo), não oferecem os mecanismos adequados para estimativas de desempenho, obrigando o desenvolvedor a modelar explicitamente o tempo em suas redes de Petri. Este processo naturalmente acarreta algumas dificuldades de modelagem, uma vez que todas as partes constituintes do sistema provavelmente haveriam de ter suas transições conectadas a uma mesma entidade que regesse o tempo. Sem parâmetros adequados de temporização, os modelos construídos para medir desempenho tornam-se então mais complexos e difíceis de ser analisados/compreendidos.

Para contornar estes tipos de problemas, as principais formalizações de redes de Petri foram estendidas de maneira a incluir mecanismos intrínsecos para o tratamento de parâmetros temporais. Neste contexto, temporização tem sido então objeto de pesquisas ao longo de décadas. Durante todo este tempo, diversas propostas foram publicadas, algumas delas tendo o suporte de ferramentas.

De maneira geral, podemos dizer que os esquemas de temporização estão concentrados na atribuição de parâmetros de tempo a lugares [14], transições [25], arcos [6] ou a fichas [22]. Um número razoável de classes e subclasses de redes temporizadas foram propostas dentro desta perspectiva.

A ferramenta Design/CPN [13], em cujos modelos fichas e transições carregam informações temporais, foi usada com sucesso para medir desempenho de diversos sistemas [23]. A ferramenta implementa os modelos de simulação descritos por Jensen [22], que possuem

um relógio global e utilizam fichas temporizadas.

Nas *Timed Petri Nets* [35], a cada transição é associado o tempo a partir do qual ela pode disparar e o tempo que ela leva para disparar. Além disso, cada transição tem que disparar assim que tornar-se habilitada por seus lugares de entrada e por sua restrição temporal. *Timed* Petri Nets também são discutidas em [49; 50; 51].

No tocante às Stochastic Petri Nets [32; 48], temos associada às transições uma duração de disparo aleatória, exponencialmente distribuída. Trata-se de um modelo analítico, markoviano e, em decorrência da complexidade de análise, só deve ser aplicado a sistemas com tamanho limitado [29]. Além disso, sua aplicação não é apropriada a sistemas que não seguem as restrições matemáticas impostas pelo modelo.

Time Petri Nets e Time Stream Petri Nets são comparadas em [6]. Nas *Time* Petri Nets, um intervalo de tempo é associado a cada transição. O limite inferior de um intervalo de uma transição indica o tempo mínimo que deve transcorrer antes que ela possa disparar. O limite superior indica o tempo máximo (depois dele, a transição não poderá mais disparar). As Time Stream Petri Nets valem-se de um princípio análogo, mas os intervalos são associados a arcos de entrada de transições e não a transições.

Há ainda várias outras extensões temporizadas para as redes de Petri. Em [33], temos comparações entre *Hierarchical Duration M-nets* e *Causal Time M-nets* enquanto [44] lida com *Interval Timed Coloured Petri Nets* e [14] nos mostra uma aplicação de *Probabilistic Timed Petri Nets*.

Observou-se que as extensões temporizadas propostas para as redes de Petri podem ser simuladas por redes não-temporizadas [23], ainda que com enorme prejuízo à legibilidade dos modelos. Além disso, observou-se que as classes temporizadas de redes de Petri são equivalentes entre si, apesar de alguns autores defenderem a existência de classes não-equivalentes [6].

Temporização também tem sido objeto de várias pesquisas no tocante à Orientação a Objetos. Desde a última década, vários modelos OO temporizados foram propostos, muitos deles *estendendo* UML. UML-RT, conforme utilizada em [4; 26], integra alguns conceitos da linguagem ROOM (Real-time Object-oriented Modeling) [40] a UML, de maneira a tornar esta última mais adequada à modelagem de sistemas cujas funcionalidades apresentem restrições de tempo. Shu, em [41] provê uma semântica formal aos diagramas de estados UML

e faz verificação de modelos através de autômatos temporizados (*timed automata*). *Real-time UML* é mais uma extensão de UML para sistemas de tempo real e conta com o suporte da ferramenta Rhapsody [4]. Constam também na literatura experiências com Real-time Corba [28] bem como com várias outras extensões não baseadas em UML [2].

A maioria dos modelos OO temporizados foi concebida para a modelagem de Sistemas de Tempo Real, que apresentam um considerável grau de concorrência e paralelismo. Concentrou-se então bastante esforço em dar aos modelos OO uma semântica para lidar com tais características, o que justifica a criação/utilização de elementos como cápsulas, *bindings* e portas em linguagens como ROOM e UML-RT. Em conseqüência desta concentração de esforços, o tratamento temporal na maioria dos modelos é reduzido à associação de *atrasos* a arcos entre estados de um diagrama de estado. Já no que diz respeito às redes de Petri, os pesquisadores da área puderam concentrar-se nos aspectos de temporização uma vez que o formalismo original já se mostrara adequado para expressar concorrência e paralelismo. Conseqüentemente, estratégias mais bem elaboradas de tratamento temporal foram propostas no mundo das redes de Petri ao longo de três décadas de pesquisas sobre este assunto.

1.4 Objetivos e Relevância

O objetivo principal deste trabalho é o de estender o formalismo RPOO, tornando-o mais adequado à modelagem de sistemas de tempo real e à medição de desempenho de protocolos de rede.

A maioria dos mecanismos de tratamento de tempo nos modelos OO incide na descrição do comportamento interno dos objetos [2; 4; 40; 41; 43]. Normalmente, nestes modelos, o tempo é tratado em máquinas de estados. Um dos resultados que temos por objetivo é o de apresentar uma abordagem de temporização num nível mais alto de abstração, sendo efetivado na troca de mensagens entre objetos. Em outras palavras, podemos dizer que queremos que a temporização se dê em *nível OO*.

Em todo caso, a extensão não deve inibir a possibilidade de modelarem-se restrições temporais para atividades internas aos objetos de um sistema. Mais especificamente, o modelo temporizado proposto deve ser facilmente integrado com o modelo de temporização proposto para as Timed Coloured Petri Nets (TCPN [22]), uma vez que, em seu estado atual, RPOO

usa redes de Petri coloridas (Coloured Petri Nets) para descrever classes.

A extensão deve ser validada através de sua aplicação à modelagem e análise de desempenho do IP Móvel, protocolo cuja modelagem em RPOO *puro* evidenciou a necessidade de uma extensão temporizada. Neste sentido, objetivamos, também, a implementação de um protótipo de ferramenta que dê suporte à simulação dos modelos da extensão *temporizada* de RPOO.

Em relação aos modelos de temporização em redes de Petri convencionais, este trabalho apresenta a vantagem de incidir sobre modelos de um tipo de metodologia (OO) amplamente utilizada no mercado. Em relação aos modelos OO temporizados, o principal ponto positivo é o de tratar o tempo em nível de interação entre objetos, o que não acontece na maioria dos modelos propostos. Outrossim, a escolha, para este trabalho, de *temporizar* RPOO em detrimento de outros modelos também passíveis de temporização deve-se ao fato deste formalismo destacar-se como uma linguagem cuja extensão temporizada poderá reunir as características de ser executável, passível de verificação formal, temporizada e com semântica de paralelismo e concorrência tanto em nível de interação entre objetos quanto em nível de descrição de objetos.

1.5 Estrutura deste Documento

Este documento está organizado da seguinte maneira: no Capítulo 2 é apresentada uma visão informal de RPOO, seguida das definições formais acerca do que foi exposto informalmente. São abordados, essencialmente, os aspectos de RPOO que são relevantes para uma visão geral do formalismo e para a definição de uma extensão temporizada.

No Capítulo 3 são introduzidas, num primeiro momento, as propostas de temporização em nível informal. Depois disso, os conceitos discutidos são definidos formalmente, constituindo uma extensão de RPOO, denominada RPOOt. As definições são extensões das definições apresentadas no Capítulo 2.

No Capítulo 4, descreve-se o protótipo de um simulador para a linguagem RPOOt, abordando suas funcionalidades, suas limitações e os principais aspectos de sua implementação. Alguns exemplos didáticos são apresentados no intuito de introduzir o uso da ferramenta e detalhar algumas variações sintáticas com respeito às definições formais.

No Capítulo 5, é apresentado o processo de modelagem do protocolo IP Móvel com o protótipo de ferramenta (denominado Renew/RPOOt). Além dos modelos RPOOt para o protocolo, algumas medidas preliminares de desempenho são apresentadas, bem como o processo de simulação (em grades computacionais) que as derivou.

No Capítulo 6, são apresentadas considerações finais a respeito dos resultados deste trabalho em seus diversos níveis de incidência: formalização, protótipo de ferramenta e suporte a análise de desempenho. Alguns trabalhos relacionados são analisados e várias possibilidades de trabalhos futuros são identificadas.

Finalmente, no Anexo A, é descrito o processo de validação do protótipo de ferramenta RenewRPOOt. A validação se deu através da comparação entre simulações com o Renew/RPOOt e espaços de estado obtidos a partir de uma ferramenta específica para geração de espaços de estados RPOO.

Capítulo 2

Fundamentação Teórica

2.1 RPOO - Visão Informal

Conforme explicado anteriormente, um modelo RPOO é – numa perspectiva informal – um conjunto de classes e suas redes de Petri correspondentes. As classes são descritas e se relacionam umas com as outras como classes UML. As redes de Petri descrevem o comportamento dos objetos — há exatamente uma rede de Petri de alto nível para cada classe do modelo. Por esta razão, um modelo RPOO pode ser obtido a partir de um diagrama de classes e a aplicação do formalismo pode se dar de forma mais adequada na fase de *design* da maioria das metodologias de desenvolvimento OO.

O formalismo define várias operações que podem ser efetuadas pelos objetos. Tais operações fazem parte da especificação das classes e são chamadas de *ações*. Neste contexto, várias ações RPOO, também chamadas de *ações elementares*, podem ser efetuadas pelos objetos (na Tabela 2.1, podem-se ver alguns exemplos de ações elementares). Ações são representadas por *inscrições* em transições nas redes de Petri que descrevem as classes. Uma inscrição pode conter várias ações e todas as ações de uma inscrição são executadas atomicamente quando a transição que contém a inscrição dispara. A um conjunto de ações atomicamente executadas dá-se o nome de *evento*. Apenas eventos são executados por um simulador RPOO (alguns eventos, entretanto, podem conter apenas uma ação). Como a execução de um evento consiste da execução de suas ações subjacentes, será usado o termo *executar* tanto com respeito a eventos quanto a ações.

Em RPOO, cada objeto é uma *thread* e a interação entre objetos pode se dar de maneira

Ação	Descrição
x = new X	Instanciação da classe X
$x?mensagem$	Recebimento de mensagem
$x.mensagem$	Envio assíncrono de mensagem
$x!mensagem$	Envio síncrono de mensagem
unlink x	Ação de desligamento com objeto x

Tabela 2.1: Algumas ações elementares

síncrona ou *assíncrona*. Quando um objeto *a* chama um método (envia uma mensagem) do objeto *b* em modo assíncrono, o sistema onde são executados os objetos passa a um estado no qual os dados passados como parâmetros estão *pendentes* e aptos a ser consumidos numa ação posterior por parte do objeto *b*. Em chamadas síncronas, as mensagens são enviadas *e* consumidas atomicamente.

Um conjunto de objetos interconectados e suas *mensagens pendentes* formam uma *estrutura* de um *sistema de objetos*. Além da estrutura, um sistema de objetos apresenta uma dinâmica que define como ações são executadas. Em linhas gerais, um simulador RPOO deve manipular um sistema de objetos, executando ações e decidindo que ações devem ser executadas em caso de concorrência. Ilustraremos, informalmente, os efeitos, em nível de sistema de objetos, da execução de algumas ações RPOO. Não obstante, alguma noção acerca dos conceitos básicos e da terminologia usada no mundo das redes de Petri se faz importante para uma boa compreensão dos exemplos.

Criação ou instanciação É apresentada, na Figura 2.1, a configuração de um sistema de objetos com apenas um objeto, no qual uma ação RPOO de instanciação pode ser executada. A rede de Petri que descreve o comportamento dos objetos é representada de maneira abstrata, em forma de nuvem. A transição **trans0** está *habilitada* e, ao *disparar*, executa sua ação RPOO, criando um novo objeto (do tipo ilustrativo **RpooObject**).

A ação **obj2=new RpooObject** é, em si, um *evento* — pois pode ser executada atomicamente — e o efeito de sua execução é ilustrado na Figura 2.2. Note que, numa ação de criação, o objeto que a executou terá, nas estruturas internas do sistema de objetos, uma referência (ligação) para o objeto criado, o que possibilita o envio de mensagens de um para

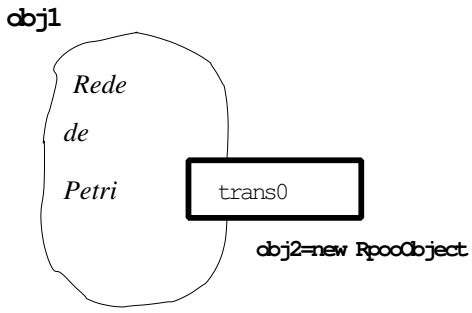

Figura 2.1: 'obj1' pode criar um novo objeto RPOO, do tipo RpooObject

o outro. O objeto executor da ação pode, também, armazenar uma referência para o objeto criado nos *lugares* da rede de Petri que descreve seu comportamento.

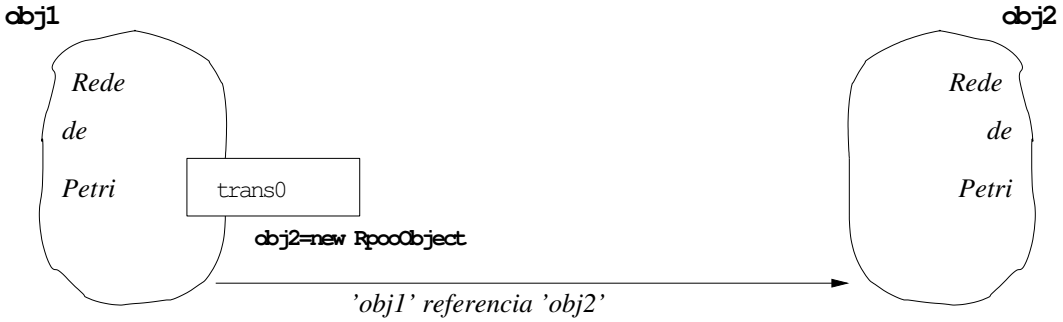

Figura 2.2: 'obj1' *possui* uma referência para 'obj2'

Saída assíncrona de dados A Figura 2.3 ilustra uma configuração para um sistema de objetos com dois objetos inter-relacionados. Mais uma vez, a rede de Petri que descreve o comportamento dos objetos é representada de maneira abstrata. O retângulo em destaque (mais escuro) indica que o estado atual da rede de Petri (implícita) *habilita* o *disparo* da transição em questão e, por conseguinte, a execução de sua ação RPOO. A inscrição **obj2.set(param)** denota uma ação de chamada assíncrona de método (ou envio assíncrono de mensagem).

Depois da execução da chamada assíncrona, temos a configuração ilustrada na Figura 2.4. Trata-se de uma mensagem pendente (**param**) de **obj1** para **obj2**. A figura também mostra que **trans2**, em **obj2**, está habilitada e, portanto, pode executar sua respectiva *ação de entrada* (tal ação consumirá a mensagem pendente **param**). Denota-se este tipo de ação por um ponto de interrogação (**?**). A inscrição **obj1?set(param)** indica que, para que a transição (**trans2**) dispare, **obj1** deve ter executado a ação **obj2.set(param)** numa ação *anterior*

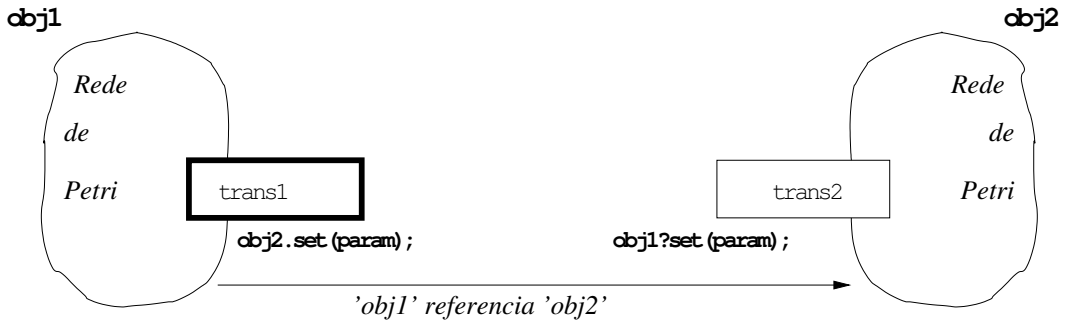

Figura 2.3: 'obj1' pode enviar mensagem para 'obj2'

(note que a referida transição precisa, também, estar habilitada pela rede de Petri abstraída na figura).

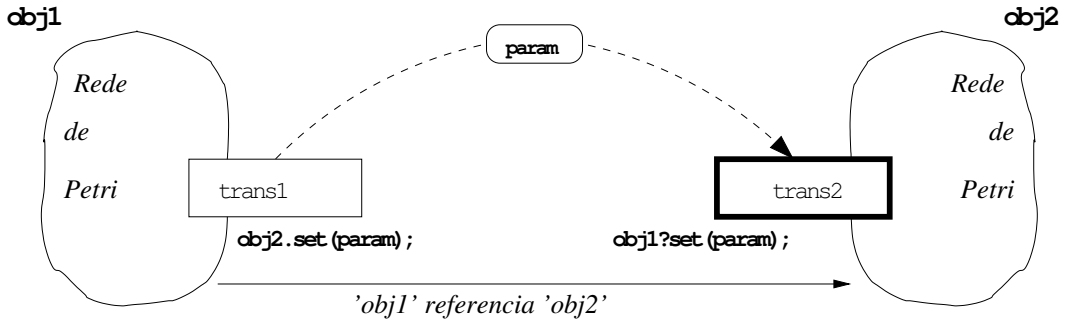

Figura 2.4: Uma mensagem *pendente* de 'obj1' para 'obj2'

Entrada de dados A Figura 2.5 ilustra o estado do sistema de objetos após a execução da ação de entrada ilustrada na Figura 2.4. Note que a mensagem não está mais pendente e que não temos mais transições habilitadas. Note também que a ação de recebimento foi efetuada num instante de tempo *posterior* à ação de envio mas, no entanto, não podemos *quantificar* o tempo decorrido entre envio e recebimento.

Saída síncrona de dados A Figura 2.6 ilustra mensagens RPOO de saída de dados em modo síncrono num sistema de objetos com três instâncias inter-relacionadas. Denotam-se chamadas síncronas de métodos pelo ponto de exclamação (**!**), conforme podemos ver na inscrição de **trans1**. Nesta inscrição RPOO, temos o envio de duas mensagens síncronas, **obj2!set(param)** e **obj3!set(param)**. Por se tratarem de mensagens síncronas, as ações devem ser executadas em conjunto com ações de entrada correspondentes – **obj1?set(param)**

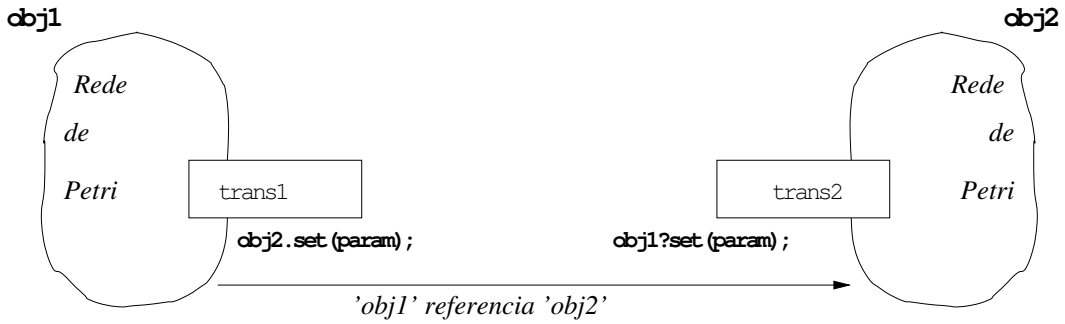

Figura 2.5: Mensagem *consumida* por 'obj2'

– inscritas em transições de **obj2** e **obj3**. Todas essas ações são parte do mesmo *evento* e, para que tal evento *ocorra* (seja executado), as três transições ilustradas devem estar habilitadas por suas respectivas redes de Petri. A execução do evento funciona como se todas as transições envolvidas disparassem ao mesmo tempo. Após a execução de chamadas síncronas, não há, na estrutura do sistema de objetos, mensagens pendentes decorrentes de tais chamadas: as mensagens são enviadas e consumidas atomicamente.

Note que o *evento* RPOO executado neste cenário é formado pelas ações **obj2!set(param)** (em **obj1**), **obj3!set(param)** (em **obj1**), **obj1?set(param)** (em **obj2**) e **obj1?set(param)** (em **obj3**). No contexto deste sistema de objetos, nenhuma dessas quatro ações – tampouco qualquer combinação de duas ou três delas – constitui, isoladamente, um *evento*, visto que nenhuma delas pode ser executada atomicamente, independente das outras.

Ação de desligamento Na Figura 2.7 ilustramos, por fim, a possibilidade de execução de uma ação de desligamento. Na figura, mais uma vez, temos dois objetos (**obj1** e **obj2**) inter-relacionados. Dois eventos podem ser executados em **obj1**: um deles contém a ação de desligamento **unlink obj2**; o outro contém uma ação de envio assíncrono de mensagem. Caso este último evento seja executado, teríamos uma configuração análoga à da Figura 2.4, com uma mensagem pendente de **obj1** para **obj2**. Caso o evento que contém a ação de desligamento seja executado, teremos a configuração ilustrada na Figura 2.8. Note que, devido ao fato de **obj1** não estar mais ligado a **obj2**, não será possível a troca de mensagens entre esses dois objetos. Assim, **trans2**, em **obj1**, não está mais habilitada pois inscreve uma ação cuja execução requer uma referência a **obj2**.

Convém observarmos que as situações ilustradas nos exemplos anteriores são meramente

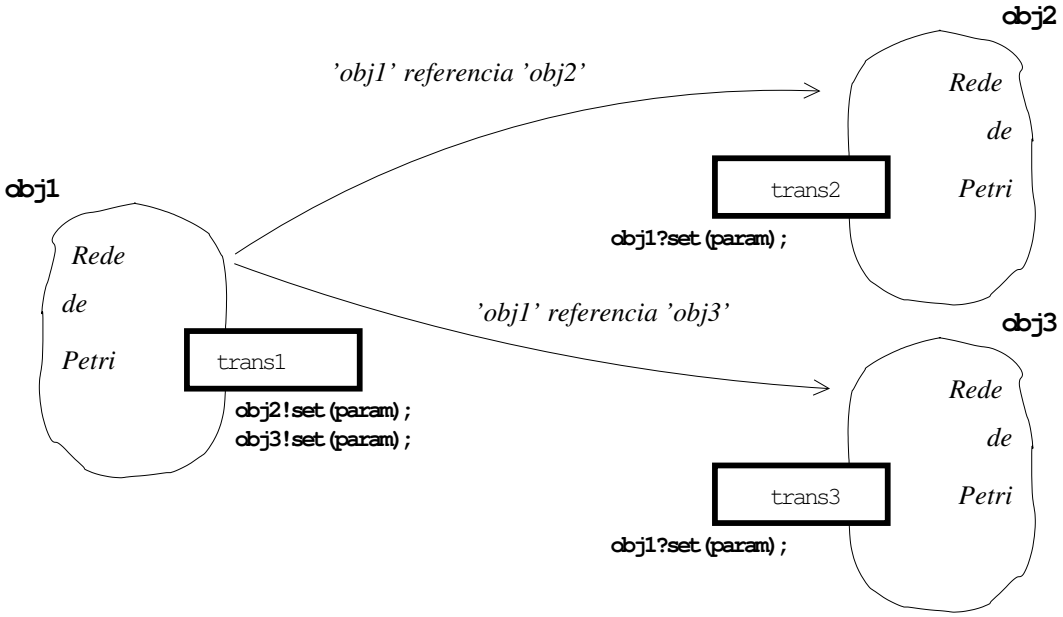

Figura 2.6: Mensagens sincronizadas

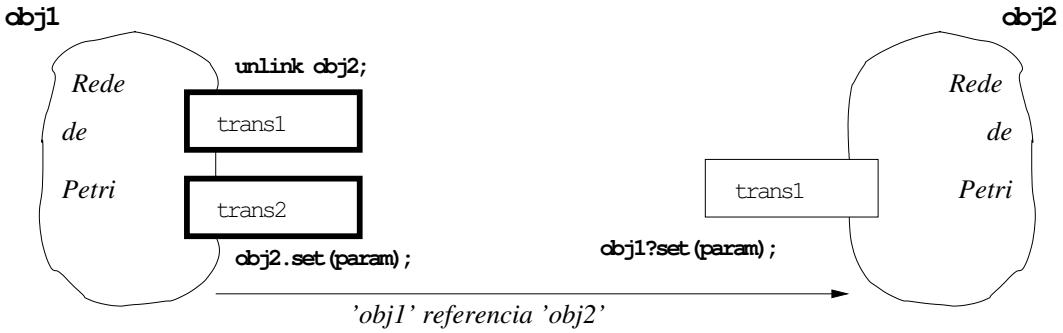

Figura 2.7: Ação de desligamento

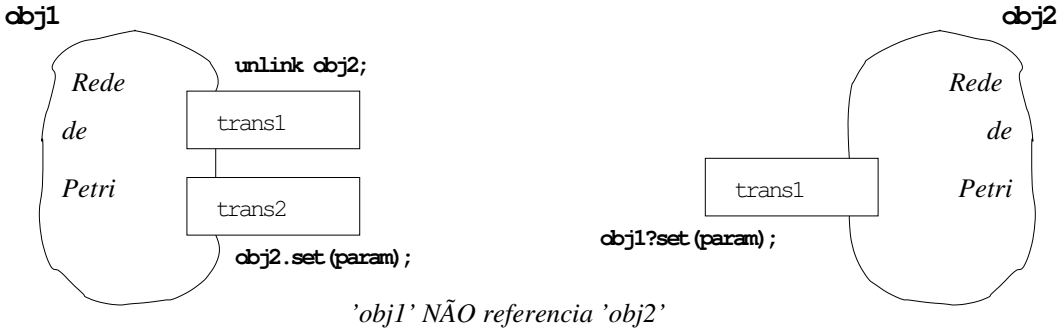

Figura 2.8: Efeito da execução da ação de desligamento

didáticas, o que justifica a utilização de referências *absolutas*, estáticas. Na prática, como veremos no estudo de caso, as inscrições RPOO contêm variáveis/expressões, cujos valores

são *avaliados* em tempo de execução. Desta forma, numa ação do tipo *obj2.set(param)*, *obj2* seria uma variável cuja ligação (*binding*) seria calculada com base nas referências que o objeto executor da ação possui na estrutura do sistema de objetos. A prática mais comum, entretanto, é guardar referências nos *lugares* das redes de Petri que descrevem as classes. O valor dos dados (*param*) passados como parâmetro também são avaliados, no caso mais comum, a partir de valores oriundos dos lugares das redes de Petri.

2.2 RPOO - Aspectos Formais

Nesta seção descreveremos os aspectos formais de RPOO importantes para a compreensão das ações explicadas informalmente na Seção 2.1 e da extensão proposta no Capítulo 3. Uma vez que estaremos essencialmente tratando de definições formais, a maior parte do restante deste capítulo será transcrita do formalismo original [18]. A sintaxe utilizada sofreu algumas alterações de acordo com a conveniência das diversas ferramentas utilizadas para dar suporte ao formalismo. Apresentaremos, entretanto, a notação utilizada originalmente, postergando os devidos ajustes sintáticos para a seção de ferramentas.

2.3 Definições Básicas

Classes e Objetos Em RPOO, a modelagem das entidades que compõem o sistema se baseia na noção de *classes*. O princípio da modelagem consiste em *classificar* os objetos de um sistema de acordo com suas propriedades em comum. Portanto, as classes são definidas como *conjuntos de objetos que compartilham as mesmas propriedades*. Para representá-las, utilizaremos a notação

$$\mathsf{C}_1, \mathsf{C}_2, \ldots, \mathsf{C}_n.$$

Para representar objetos, utilizaremos a notação

$$o_1, o_2, \ldots, o_i, \ldots$$

Para descrever que um objeto o_i pertence a uma classe C_j, utilizamos o operador de pertinência $\in$. Assim, $o_i \in \mathsf{C}_j$ significa que o objeto o_i pertence à classe C_j. Cada descrição

de classe do modelo é internamente identificada por um *nome de classe*. Representamos os elementos deste conjunto por

$$\mathbb{C}_1, \mathbb{C}_2, \ldots, \mathbb{C}_n.$$

Para efeitos de simplificação, convencionamos que o $\mathbb{C}_i$ corresponde à classe C_i de objetos. Conceitualmente, objetos e identificadores são biunivocamente relacionados. Para evidenciar esta relação, denota-se o identificador do objeto o_i por $\vec{o_i}$. Também assume-se que os identificadores dos objetos da classe C_i pertencem ao conjunto N_i (de nomes). Assim, podemos identificar o tipo de um objeto pelo tipo de seu identificador, ou seja,

$$o_i \in \mathsf{C}_j \quad \Longleftrightarrow \quad \vec{o_i} \in \mathsf{N}_j.$$

Também definimos o conjunto universo de identificadores, denotado por N, como a união dos conjuntos N_i. Desta forma, se há n classes no modelo, então $\mathsf{N} = \mathsf{N}_1 \cup \mathsf{N}_2 \cup \cdots \cup \mathsf{N}_n$.

Tipos de dados e Sortes Boa parte da modelagem consiste da descrição dos tipos de dados que devem ser manipulados pelos objetos. Para a modelagem de dados que adotamos, fazemos uso de uma Σ-álgebra $\langle \mathfrak{D}, \mathfrak{O} \rangle$, onde $\mathfrak{D}$ é um conjunto de domínios (ou de tipos de dados) e $\mathfrak{O}$ o respectivo conjunto de operações. Com isso, deixamos em aberto a escolha de alguma linguagem particular para descrever tipos de dados e operações numa eventual implementação de ferramentas.

Os domínios em $\mathfrak{D}$ pertencem a dois grupos: os tipos de dados específicos do problema e os tipos formados a partir dos identificadores de objetos. Denotamos por $\mathsf{D}_1, \mathsf{D}_2, \ldots, \mathsf{D}_m$, os tipos de dados específicos do problema, onde *m* é um natural finito e determinado. Os tipos de identificadores são os conjuntos N_i e o conjunto N, definidos anteriormente. A definição destes elementos como domínios do modelo permite que objetos armazenem e processem identificadores como outros tipos de dados. Logo, se em um sistema há m tipos específicos de dados e n classes, os domínios em $\mathfrak{D}$ são:

$$\mathfrak{D} = \{\mathsf{D}_1, \mathsf{D}_2, \ldots, \mathsf{D}_m, \mathsf{N}_1, \mathsf{N}_2, \ldots, \mathsf{N}_n, \mathsf{N}\}.$$

Os conjuntos $\mathfrak{D}$ e $\mathfrak{O}$ representam conceitos semânticos sobre domínios e operações que podem ser tratados em um sistema. Para a aplicação destes conceitos, isto é, para descrever os dados dos modelos, são utilizados os nomes dos domínios e a descrição das operações.

Damos a estes elementos o nome de *sortes*. Formalmente, as *sortes* podem ser definidas a partir da assinatura Σ da seguinte forma:

$$\Sigma = \langle \mathcal{S}, \mathcal{A} \rangle$$
$$\mathcal{S} = \{\mathbb{S}_1, \mathbb{S}_2, \ldots, \mathbb{S}_m, \mathbb{C}_1, \mathbb{C}_2, \ldots, \mathbb{C}_n, \mathbb{O}\}$$

O conjunto $\mathcal{S}$ consiste em uma sorte para cada um dos domínios em $\mathfrak{D}$. As operações sobre os dados concretos são definidas pelo conjunto de símbolos denotado por $\mathcal{A}$. Os $\mathbb{S}_m$ elementos correspondem às sortes dos tipos de dados específicos do sistema. A sorte $\mathbb{S}_i$ corresponde ao domínio D_i. Uma vez que o conjunto $\mathbb{C}$ está contido no conjunto $\mathcal{S}$, podemos anotar os modelos com os nomes de classes. Os valores possíveis para estes tipos de dados são identificadores de objetos das classes. Isto torna possível utilizar identificadores como dados nos modelos e permite identificar a relação entre as classes do sistema através da análise do tipo do identificador no modelo.

Variáveis, termos e equações Para tratar de variáveis termos e equações, em RPOO, usamos os seguintes conceitos derivados de Σ: $\mathcal{V}$ e $\mathcal{T}$ denotam, respectivamente, um conjunto de variáveis para Σ, e o conjunto de termos derivados. Ambos são formados pela união disjunta de famílias de variáveis e de termos, indexadas por sortes de Σ. Ou seja, definimos:

$$\mathcal{V} = \mathcal{V}_{\mathbb{S}_1} \cup \ldots \cup \mathcal{V}_{\mathbb{S}_n} \cup \underbrace{\mathcal{V}_{\mathbb{C}_1} \cup \ldots \cup \mathcal{V}_{\mathbb{C}_n} \cup \mathcal{V}_{\mathbb{O}}}_{\text{variáveis-objeto}}$$
$$\mathcal{T} = \mathcal{T}_{\mathbb{S}_1} \cup \ldots \cup \mathcal{T}_{\mathbb{S}_n} \cup \underbrace{\mathcal{T}_{\mathbb{C}_1} \cup \ldots \cup \mathcal{T}_{\mathbb{C}_n} \cup \mathcal{T}_{\mathbb{O}}}_{\text{termos-objeto}}$$

Dizemos que a variável "v é do tipo $\mathbb{S}_i$" se $v \in \mathcal{V}_{\mathbb{S}_i}$. Formalmente, isto significa que só valores do domínio adequado podem ser atribuídos a v. Ou seja, se $v \in \mathcal{V}_{\mathbb{S}_i}$ e denotamos uma atribuição por $v \mapsto d$, então $d \in \mathsf{D}_i$. Analogamente, dizemos que o termo "t é do tipo $\mathbb{S}_i$" se $t \in \mathcal{T}_{\mathbb{S}_i}$. Formalmente, isto significa que $[\![t]\!] \in \mathsf{D}_i$, onde $[\![t]\!]$ é a avaliação de t. Dizemos que v é uma variável-objeto se $v \in \mathcal{V}_{\mathbb{C}_i}$, para alguma classe $\mathbb{C}_i$. Analogamente, termos $t \in \mathcal{T}_{\mathbb{C}_i}$ são chamados de *termos-objeto*.

Também usamos o conceito de equações como pares de termos escritos na forma "$t_1 = t_2$" e sua interpretação convencional. Ao conjunto das equações definidas sobre Σ denotamos

$\mathcal{E}$. Estendemos o conceito de "tipo de um termo" para multi-conjuntos de termos. Se m é um multi-conjunto de termos (um multi-conjunto sobre $\mathcal{T}$), dizemos que "o tipo de m é $\mathbb{S}_i$" se $m \in \mathcal{T}^{\mathsf{ms}}_{\mathbb{S}_i}$.

2.4 Modelo RPOO

Definição 2.1 (Modelo RPOO) *Um modelo RPOO $\mathcal{M}$ é definido pela seguinte tupla:*

$$\mathcal{M} = \langle \mathbb{C}, \mathcal{O} \rangle$$

onde $\mathbb{C}$ é um conjunto de classes identificadas para o problema modelado e $\mathcal{O}$ é a configuração inicial*, definida no* sistema de objetos*, respeitando-se as regras de associação descritas pelas classes. Definiremos* configuração *de sistemas de objetos mais adiante.*

Classes Cada classe definida em $\mathbb{C}$ é formada por três partes: *nome*, *corpo* e *inscrições de interação*. O *nome* é o identificador único da classe. Dessa forma, os nomes são representados através do conjunto formado pelos $\mathbb{C}_i$ elementos definidos sintaticamente. O *corpo* é a descrição formal do comportamento dos objetos pertencentes às classes. As inscrições de interação existem para descrever o efeito externo das ações ocorridas nos objetos, de acordo com as regras do *sistema de objetos*. Naturalmente, as inscrições de interação estão associadas aos elementos que representam a ocorrência de ações no formalismo que descreve o corpo da classe.

Corpo das Classes Para descrever o comportamente dos objetos que compõem uma classe, utilizamos redes de Petri coloridas (ou CPN), definidas por Jensen [21]. Desta forma, a semântica das redes de Petri descritas a seguir será a mesma apresentada por Jensen, adaptada para a notação que estamos utilizando, baseada na assinatura Σ.

Conceitualmente, é possível utilizar CPN porque os elementos que definem as redes de Petri são preservados. A única consideração que devemos fazer é em relação à sintaxe das redes de Petri. Os elementos que originalmente armazenam ou manipulam informações não são mais mapeados a domínios de dados, mas sim às sortes. Esta alteração nos possibilita definir o conjunto de dados e operações que as redes podem manipular. Isto significa que

agora temos o domínio de dados e operações definidos originalmente para CPN, acrescido dos tipos das classes definidas no modelos, que passam a ser tratadas como tipos de dados.

Multi-conjuntos A noção de multi-conjuntos é utilizada, em RPOO, tanto em nível de descrição de objetos, através das redes de Petri, quanto em nível de *interação entre objetos*, através das definições RPOO propriamente ditas. Antes de nos concentrarmos nas definições de redes de Petri, apresentamos a definição de multi-conjuntos, conforme consta no trabalho de Jensen sobre redes de Petri coloridas [21]:

Definição 2.2 *Um multi-conjunto m, sobre um conjunto não-vazio S, é a função $m : S \rightarrow \mathbb{N}$, onde a soma:*

$$\sum_{s \in S} m(s)$$

é finita. O inteiro não-negativo $m(s)$ é o número de ocorrências *do elemento s no multi-conjunto m.*

É comum representarmos multi-conjuntos por uma soma formal:

$$\sum_{s \in S} m(s)`s$$

Utilizamos a notação S^{ms} para indicar o conjunto de todos os multi-conjuntos possíveis sobre S.

Definição 2.3 (Estrutura de redes de Petri) *Uma estrutura de rede de Petri é uma tupla $\langle P, T, F \rangle$, em que P é um conjunto finito de* lugares*, T é um conjunto finito de* transições *e $F \subseteq (P \times T) \cup (T \times P)$ é um conjunto de* arcos*, com P e T disjuntos.*

Além de uma estrutura, uma rede de Petri colorida possui declarações de tipos de lugares, expressões de arcos, guardas nas transições, e expressões de inicialização de marcação. Todas estas definições são construídas a partir de elementos da assinatura de Σ.

Definição 2.4 (Rede de Petri colorida) *Sejam uma assinatura Σ, uma estrutura de rede de Petri $N = \langle P, T, F \rangle$ e funções $C : P \rightarrow \mathcal{S}$, $G : T \rightarrow 2^{\mathcal{E}}$, $E : F \rightarrow \mathcal{T}^{ms}$, e $I : P \rightarrow \mathcal{T}^{ms}$ denominadas, respectivamente, de* função de cores*,* de guardas*,* de expressões de arcos*, e de*

inicialização. *Dizemos que* $\langle N, C, G, E, I\rangle$ *é uma* rede de Petri colorida *definida sobre* Σ *desde que:*

$$\forall f \in F:\ E(f) \text{ seja do tipo } C(p),\ \text{onde } f = \langle p, t\rangle \text{ ou } f = \langle t, p\rangle \text{ e}$$

$$\forall p \in P:\ I(p) \text{ é uma expressão fechada (não tem variáveis) e é do tipo } C(p).$$

A função de cores, que mapeia os lugares da rede a sortes, tem por finalidade determinar o tipo de dado correspondente a cada lugar. Os elementos de um domínio existentes em cada lugar da rede são denominados *fichas*. A função de guarda mapeia equações a transições. A função de expressão de arcos indica o "peso" do arco, ou seja, a quantidade de fichas a serem retiradas ou inseridadas nos lugares associados ao arco. Para que uma transição seja habilitada, as expressões da função de guarda e da função de expressões devem ser satisfeitas. Por último, a função de inicialização associa uma expressão fechada a cada lugar para determinar sua marcação inicial.

Comportamento das Redes de Petri Com os elementos apresentados até agora, nós podemos definir o que é *marcação* e o que *estado* de uma rede de Petri. Chamamos *marcação* o conjunto formado pela relação *lugares da rede* x *fichas*. A marcação determina o *estado* de uma rede de Petri.

Definição 2.5 (Fichas e Marcações) *Uma* ficha *é um par* $\langle p, d\rangle \in P \times \mathsf{D}_i$*, tal que* $C(p) = \mathbb{S}_i$*. Denotamos por* F *o conjunto das fichas de uma rede. E uma* marcação *é um multi-conjunto sobre* F*. O conjunto das marcações de uma rede é denotado por* M*. A marcação inicial* m_0 *é obtida pela avaliação das expressões de inicialização:*

$$\forall p \in P, m_0(p) = I(p).$$

A marcação de uma rede de Petri somente se modifica mediante o *disparo* (ou *ocorrência*) de uma transição. O disparo das transições, por sua vez, determina o comportamento da rede. As expressões dos arcos e as fichas contidas nos lugares de entrada de uma transição desempenham o papel de *pré-condições* para que o disparo de uma transição ocorra. A atribuição de valores às variáveis contidas nas expressões de arcos e guardas associadas às transições determina o *modo* de disparo para uma transição. Naturalmente, vários modos de disparo podem existir para uma transição.

Definição 2.6 (Modo de transição) *Um* modo *de uma transição t é uma atribuição de valores para as variáveis de $\mathcal{V}$. Se denotamos a atribuição por a, então escrevemos t^a para denotar a transição t no modo a.*

Quando conveniente, subentendemos a e escrevemos apenas t. Este recurso deve ser utilizado para simplificar a notação e enfatizar a transição em detrimento do modo. A partir da definição de modo, podemos determinar a regra de disparo de uma transição.

Definição 2.7 (Transição habilitada) *Sejam t uma transição, m uma marcação e a um modo de t. Dizemos que t^a está* habilitada *na marcação m, e denotamos isso por $m[t^a\rangle$, se $[\![e]\!]_a$ é verdade para toda equação $e \in G(t)$ e se*

$$[\![E(p,t)]\!]_a \leq m(p) \quad \textit{para todo } p \in P.$$

Desta forma, duas condições definem a habilitação e disparo de uma transição. Uma é que a expressão da guarda seja avaliada para verdadeira no modo a. A outra condição é que existam mais fichas nos lugares de entrada da transição do que o resultado da avaliação do arco de entrada da transição. Disparar uma transição pode mudar o estado de uma rede. A seguir, definimos a marcação alcançada após o disparo.

Definição 2.8 (Marcação alcançável) *Sejam t^a e m tais que $m[t^a\rangle$. Temos que m' é a marcação alcançada a partir da ocorrência de t^a na marcação m. Denotamos $m[t^a\rangle m'$ e definimos m' por*

$$m'(p) = m(p) - [\![E(p,t)]\!]_a + [\![E(t,p)]\!]_a \qquad \textit{para todo } p \in P. \tag{2.1}$$

Se $m[t^a\rangle m'$, dizemos que m' é diretamente alcançável *a partir de m. Chamamos cada $m[t^a\rangle m'$ de* disparo *ou de* ocorrência*.*

A definição acima indica que fichas são retiradas dos lugares de entrada e inseridas nos lugares de saída de t de acordo com a avaliação das expressões dos arcos que ligam t aos seus lugares de entrada e saída.

Inscrições de Interação O disparo de uma transição pode alterar a marcação de uma rede de Petri e também ter efeito sobre outros objetos do sistema. Esta influência externa é determinada pelas inscrições de interação anotadas nas transições. Seis tipos de inscrições de

interação podem ser associadas às transições das redes: entrada de dados, saída de dados (que pode ser síncrona ou assíncrona), instanciação, eliminação de ligações e ação de auto-destruição. Embora as ações de auto-destruição não estejam associadas à comunicação entre os objetos, consideramo-as também como *inscrições de interação* porque elas têm efeito sobre o contexto em que o objeto está inserido. As ações que não têm efeito externo, isto é, que alteram apenas o estado interno dos objetos, são denominadas ações internas e não necessitam de inscrições de interação.

Definição 2.9 (Inscrições de interação) *Seja t um termo, s um termo-objeto, $\mathbb{C}$ um nome de classe e v uma variável-objeto. A seguinte sintaxe abstrata caracteriza as inscrições de interação*

$$I ::= s.t \mid s!t \mid s?t \mid v = \textbf{new}\ \mathbb{C} \mid \textbf{unlink}\ s \mid \mathrm{end}$$
$$R ::= \epsilon \mid I \circ R$$

O conjunto de todas as inscrições de interação é denotado por R.

Em I encontramos os seis tipos de inscrições de interação. Inscrições na forma $s.t$, $s!t$ e $s?t$ são chamadas, respectivamente, de *saída assíncrona de dados*, *saída síncrona de dados* e *entrada de dados*. Os elementos s e t são termos que, ao serem avaliados, determinam o identificador do objeto e o dado envolvido na interação, respectivamente. As inscrições na forma $v = new\ \mathbb{C}$ são chamadas de *instanciações*. A variável objeto v é ligada ao identificador do objeto instanciado; logo, os valores de $v \in$ N. Inscrições do tipo $unlink\ s$ são denominadas *desligamentos*. Neste caso, o termo s determina o identificador do objeto cuja ligação deve ser eliminada. Por último, end é chamada de inscrição de *auto-destruição*. A regra R permite a composição de inscrições. O símbolo ϵ identifica ações internas.

Finalmente, podemos formalizar o conceito sintático de classe através dos elementos definidos até aqui.

Definição 2.10 (Classe) *Seja $\mathbb{C}_i$ um nome de classe, K uma rede de Petri e $I : T \rightarrow$ R uma função que mapeia transições a inscrições de interação. Dizemos que $\langle \mathbb{C}_i, K, I \rangle$ é uma* classe *de nome $\mathbb{C}_i$ e corpo K. Em geral, usamos $\mathbb{C}_i$ como representativo da classe, subentendendo corpo e inscrições de interação.*

Os termos que compõem as inscrições são avaliados para produzir um elemento de dado ou um identificador de objeto correspondente à ação. Na definição seguinte, usamos a interpretação de inscrições (denotada por $[\![I(t)]\!]_a$) para mapear disparos em ações potenciais e, em seguida, definimos o significado de uma classe.

Definição 2.11 *Seja uma classe $\mathbb{C}_i$ com corpo K. O* comportamento *de seus objetos, denotado pela relação $\rightarrow_i$, consiste no conjunto de ações potenciais definidas por*

$$m \xrightarrow{[\![I(t)]\!]_a} m' \qquad \textit{para cada disparo } m[t_a\rangle m' \textit{ em } K.$$

Definição 2.12 *O* significado *de uma classe $\mathbb{C}_i$ com corpo K é o conjunto dos objetos com estados definidos por $[K\rangle$ e comportamento determinado por $\rightarrow_i$, ou seja,*

$$[\![\mathbb{C}_i]\!] = \{\ o_j \mid o_j \textit{ tem estados } [K\rangle \textit{ e comportamento } \rightarrow_i \ \}.$$

A semântica das ações (que resultam da interpretação das inscrições) será formalmente definida na seção subseqüente.

2.5 Sistema de Objetos

Nesta seção, trataremos, finalmente, do comportamento de um modelo RPOO. A parte do formalismo que trata deste assunto é denominada *sistema de objetos*. Num sistema de objetos, temos a representação de todos os objetos que fazem parte de um sistema, as ligações entre estes objetos e eventuais mensagens pendentes.

A observação de um sistema de objetos em um instante particular de tempo do seu funcionamento é denominada de *configuração*. Configuração é, portanto, o termo usado para o conceito de estado em sistemas de objetos. A configuração de um sistema de objetos será dividida em duas partes: *estrutura* e *dinâmica*.

A estrutura é o conjunto dos elementos que formam a configuração: são os objetos, suas ligações e mensagens pendentes. A *dinâmica* é, como se pode inferir do termo, a representação das potencialidades dinâmicas da configuração. Logo, é a abstração do estado interno dos objetos da configuração — a *dinâmica* consiste apenas da enumeração das ações que os objetos estão "aptos" a executar (uma vez que já definimos uma representação explícita para

o estado dos objetos). É a partir da *dinâmica* de uma configuração que determinamos os eventos que podem ocorrer.

Inicialmente, nosso foco é o conceito de estrutura de uma configuração. Formalizamos esse conceito e apresentamos uma notação e linguagem que nos permitem expressar estruturas de configurações. Em seguida, enfocamos o conceito de *dinâmica*, completando a formalização de configurações.

2.5.1 Estrutura de um Sistema de Objetos

Nas definições a seguir, $\mathcal{O}$ denota um conjunto de objetos e D um domínio de dados, ambos potencialmente infinitos.

Definição 2.13 *A estrutura de um sistema de objetos é a tupla $\langle O, L, M\rangle$, onde:*

$O \subseteq \mathcal{O}$ é um conjunto finito de objetos (ditos objetos vivos*);*

L é o conjunto de ligações entre objetos $\langle a_1, a_2\rangle \in \mathcal{O} \times \mathcal{O}$;

M é um multi-conjunto sobre $\mathcal{O} \times D \times \mathcal{O}$, chamado de mensagens pendentes.

As ligações descritas na forma $\langle a_1, a_2\rangle$ definem que o objeto a_1 possui uma referência para o objeto a_2. Em RPOO, quando um objeto não está mais em atividade, isto é, quando ele não está mais no sistema de objetos, dizemos que o objeto está "morto". No caso da ligação descrita anteriormente, a_2 pode ser um objeto morto. Para descrever que existe uma mensagem m pendente, enviada do objeto a_1 para o objeto a_2, utilizamos o formato $\langle a_1, m, a_2\rangle$. Sendo que o valor de m pode ser um dado de qualquer um dos domínios do problema ou um identificador de objeto.

Para que seja possível "calcular" o comportamento de cada instância e conseqüentemente as estruturas subseqüentes que representam o comportamento do sistema modelado, precisamos associar cada objeto da *configuração inicial* a uma classe. A partir da estrutura de uma configuração inicial, novos objetos podem surgir somente através da execução de uma ação de instanciação por algum outro objeto. Neste caso, a associação do novo objeto instanciado à classe a que ele pertence é declarada na inscrição que representa a ação. Assim, o mapeamento acontece de forma direta nestes casos.

Definição 2.14 (Instanciação)

Seja $E = \langle O, L, M \rangle$ a estrutura de um sistema de objetos e

$\mathbb{C}$ *a representação de uma classe, definimos $I : O \rightarrow C$ como sendo a* função instância*, que mapeia cada objeto da estrutura para uma das classes definidas no modelo.*

2.5.2 Dinâmica de um Sistema de Objetos

O comportamento de um sistema de objetos é dado pelas modificações que podem ser observadas sobre sua estrutura. Embora estas modificações se dêem em função da ocorrência de ações nos objetos que a compõem, as alterações na estrutura de um sistema de objetos devem obedecer um conjunto de regras que descrevem cada ação. Esta é a parte dinâmica da configuração de um sistema de objetos.

Representações algébricas de estruturas Para simplificar o entendimento da notação a ser apresentada no restante desta seção, vamos redefinir as estruturas na forma de somas algébricas.

A gramática a seguir define a sintaxe abstrata utilizada para a representação algébrica de estruturas:

$$E \ ::= \ 0 \mid x\vec{y} \mid E + m_y^x \mid E + E$$

Cada objeto de uma estrutura é representado pelo seu rótulo. A expressão algébrica $E = a + b$, por exemplo, indica que na estrutura E, nós temos dois objetos, a e b. As ligações são descritas no formato $a\vec{b}$. Desta forma, a expressão $E = a + b + a\vec{b}$, denota uma estrutura E, que possui dois objetos, a e b, o objeto a *conhecendo* o objeto b. Por último, as mensagens são representadas no formato: m_b^a. Assim, na expressão $E = a + b + a\vec{b} + m_b^a$, temos uma estrutura E, com dois objetos, a e b, o objeto a conhecendo o objeto b e havendo uma mensagem m pendente para o objeto b, enviada pelo objeto a.

Efeito de Ações sobre Estruturas A seguir, apresentamos um conjunto de regras para definir cada tipo de ação identificada na formalização do sistema de objetos. Utilizamos a notação $\langle E, a \rangle \rightarrow E'$ para indicar que E' é a estrutura resultante da aplicação da ação a sobre a estrutura E. Para representar as ações, vamos utilizar os símbolos e a sintaxe

definidos em 2.9. Nas definições seguintes, o elemento que precede o símbolo ":" representa o objeto agente da ação.

Definição 2.15 (Ação interna) *Seja E uma estrutura e $x : \tau$ uma ação interna, temos que*

$$\langle E, x{:}\tau \rangle \rightarrow E.$$

Conceitualmente, o efeito de uma ação interna sobre uma estrutura é nulo.

Definição 2.16 (Ação de criação) *Seja E uma estrutura e* x:y=new Y *uma ação de criação, em que o objeto x cria o objeto y, definimos*

$$\langle E, x{:}y = new\ Y \rangle \rightarrow E + y + x\vec{y},\ \textit{se}\ y \notin! E.$$

A ação de criação é uma ação na qual um objeto cria uma instância de outro. Neste caso, um novo objeto passa a fazer parte da estrutura resultante e uma ligação entre o objeto agente da ação e o objeto criado é inserida. O símbolo $\notin!$ denota a não-pertinência do objeto y ao conjunto de objetos vivos de E. Desta forma, a ação só pode ser executada se o objeto criado não fizer parte da estrutura anterior à execução da ação.

Definição 2.17 (Ação de entrada de dados) *Seja E uma estrutura, $x : y?m$ uma ação de entrada de dados, em que o objeto x consome a mensagem m enviada pelo objeto y e $\bar{z}$ uma referência a $z \in O$, temos que*

$$\langle E + m_y^x, x{:}y?m \rangle \rightarrow E + x\vec{z}, \textit{se } \mathbf{m} = \bar{z}$$

$$\langle E + m_y^x, x{:}y?m \rangle \rightarrow E, \textit{se } \mathbf{m} \neq \bar{z}$$

Este tipo de ação representa o consumo de uma mensagem pendente para um determinado objeto. A mensagem consumida é excluída da estrutura resultante. Se a mensagem consumida contém uma referência para outro objeto, então uma nova ligação entre o objeto que consumiu e a referência contida na mensagem é criada.

Definição 2.18 (Ação de saída de dados) *Seja E uma estrutura e $x{:}y.m$, $x{:}y!m$ ações de saída assíncrona e síncrona de dados, respectivamente, em que o objeto x envia a mensagem m para objeto y, definimos que*

$$\langle E, x{:}y.m \rangle \rightarrow E + m_y^x, \textit{se } x\vec{y} \in E.$$

$$\langle E, x{:}y!m\rangle \rightarrow E + m_y^x, \textit{se } \vec{xy} \in E.$$

Uma ação de saída de dados significa o envio de uma mensagem para um objeto destinatário. Contudo, este envio só pode acontecer se o objeto agente da ação possui uma ligação com o objeto destino. No caso de uma ação de saída síncrona, o objeto que envia a mensagem para o outro permanece em estado de espera até que objeto destinatário consuma a mensagem. Na ação de saída assíncrona, o objeto que envia a mensagem pode continuar seu processamento antes do consumo da mensagem pelo objeto destinatário.

Definição 2.19 (Ação de desligamento) *Seja E uma estrutura e $x{:}unlink\ y$ uma ação de desligamento, em que o objeto x se desliga do objeto y, definimos que*

$$\langle E + \vec{xy}, x{:}unlink\ y\rangle \rightarrow E.$$

Em uma ação de desligamento um objeto se desliga de outro objeto – apenas a ligação deixa de existir na estrutura do sistema de objetos.

Definição 2.20 (Ação de auto-destruição) *Seja E uma estrutura, X o conjunto de todas as ligações com origem no objeto x e $x{:}end$ uma ação final, em que x se destrói, temos*

$$\langle E + X + x, x{:}end\rangle \rightarrow E.$$

Neste caso um objeto se destrói e deixa de existir como objeto ativo do sistema de objetos. As regras definem que somente as ligações em que o objeto origem é o agente da ação sejam excluídas da estrutura (ligações do tipo $\langle x, y\rangle$).

Em RPOO, é possível que se associe mais de uma inscrição de interação a uma ação do objeto. Neste caso, temos uma ação composta em que o efeito final resultante é definido pela concatenação do efeito das ações elementares da composição.

Definição 2.21 (Ação composta) *O efeito de uma ação composta é definido concatenando-se o efeito das ações elementares da composição*

$$\frac{\langle E, x{:}a_1\rangle \rightarrow E' \quad \langle E', x{:}a_2\rangle \rightarrow E''}{\langle E, x{:}a_1 \circ a_2\rangle \rightarrow E''}$$

Eventos Diferentes objetos de um modelo RPOO podem executar ações concorrentemente. A execução do conjunto completo destas ações pelos diferentes objetos do modelo é denominada *evento*. Antes de definir eventos formalmente, vamos estender a relação $\rightarrow$ para caracterizar o efeito combinado de um conjunto de ações sobre uma estrutura. Inicialmente do efeito de um conjunto vazio de ações, que não modifica a estrutura:

$$\langle E, \emptyset \rangle \rightarrow E$$

A partir dessa base, definimos o efeito de um conjunto de ações, combinando o efeito isolado de cada ação. Seja c um conjunto de ações de objetos diferentes de x:

Definição 2.22

$$\frac{\langle E, c \rangle \rightarrow E' \quad \langle E', x{:}a \rangle \rightarrow E''}{\langle E, c \cup x{:}a \rangle \rightarrow E''}, \qquad \nexists\, x{:}a' \in c$$

Para completar a definição de eventos, precisamos considerar também as restrições de sincronização para ações de saída síncrona. Nestes casos, as ações complementares devem ocorrer simultaneamente, isto é, devem fazer parte de um mesmo evento.

Definição 2.23 (Evento) *Um conjunto de ações e é dito um evento se, para toda a ação elementar x:y!m em e, existe a ação elementar y:x?m, também pertencente a e. Ou:*

$$\textit{e é um evento} \qquad \Longleftrightarrow \qquad \forall\, x, y, m : x{:}y!m \in e \Longrightarrow y{:}x?m \in e$$

Configurações e ocorrência/alcançabilidade Para definir o conceito de configuração precisamos associar a cada objeto da estrutura de um sistema de objetos o seu estado interno. Como vimos anteriormente, o estado interno de um objeto é representado pela rede de Petri e sua marcação. Vamos representar o estado interno de um objeto no formato x_i, onde x é o objeto a que o estado se refere e $i = 0, 1, 2, \ldots$ indica uma seqüência de estados. Usamos o índice 0 para indicar o estado inicial.

Para representar o comportamento dos objetos definimos uma relação de transição $\longrightarrow$, com elementos na forma $\langle x_i, a \rangle \longrightarrow x_j$. Cada elemento da relação é dito uma *ação potencial*, cuja interpretação é: "se x_i é o estado interno de x, então ao executar a ação a, x passa para o estado x_j". Observe que a relação não garante que a ação a ocorra pelo fato de x estar

no estado x_i. Define apenas que x satisfaz as condições para que a aconteça e que caso a ocorra, o novo estado de x é x_j. A ação é dita *iminente* ou *habilitada* para x.

Podemos definir formalmente o conceito de configuração:

Definição 2.24 (Configuração) *Uma* configuração *de um sistema de objetos consiste de uma estrutura E; um conjunto de estados internos para cada objeto vivo na estrutura (denotamos cada conjunto de estados internos por σ); e uma relação $\longrightarrow$ que caracteriza o comportamento desses objetos. Logo, denotamos a configuração pela tupla $\langle E, \sigma, \longrightarrow \rangle$.*

Para caracterizar o comportamento dos sistemas definimos *ocorrência* e *alcançabilidade* sobre a definição de configuração. A relação de ocorrência, escrita $C \vdash C'$, indica que a ocorrência de um evento altera a configuração de um sistema de C para C'. Definimos alcançabilidade através da notação $C \vdash^* C'$. Dizemos que C' é alcançável a partir da configuração C. A seguinte regra caracteriza a relação: seja e um evento composto de ações dos objetos pertencentes à estrutura E

$$\frac{\langle E, e\rangle \rightarrow E' \quad \langle \sigma, e\rangle \rightarrow \sigma'}{\langle E, \sigma\rangle \vdash \langle E', \sigma'\rangle}, \qquad \text{onde } x{:}a \in e \Rightarrow x \in E$$

A regra determina que um sistema com a configuração $\langle E, \sigma\rangle$ pode chegar à configuração $\langle E', \sigma'\rangle$ desde que: E' seja o resultado da aplicação de um evento e; e que σ' seja gerado a partir de σ, pela modificação dos estados referentes às ações dos objetos. Estamos considerando que toda a ação em e é possível de ocorrer na estrutura E.

Grafos de alcançabilidade Podemos representar o comportamento de um modelo no sistema de objetos através de seu grafo de alcançabilidade. Neste caso, os vértices correspondem às configurações alcançáveis do sistema e os arcos aos eventos.

Definição 2.25 *Seja C_0 a configuração inicial de um sistema de objetos e E seu conjunto de eventos. Seu* grafo de alcançabilidade*, denotado por $\mathrm{G}(C_0)$, é um grafo E-rotulado $\langle V, A\rangle$, onde*

$$V = \{\ C_i \mid C_0 \vdash^* C_i\ \}$$
$$A = \{\ \langle C_1, e, C_2\rangle \in V \times E \times V \mid C_1 \vdash_e C_2\ \}.$$

Um sistema de objetos é representado por sua configuração inicial C_0. O conjunto de vértices de $\mathrm{G}(C_0)$ é definido como o conjunto das configurações C_i alcançáveis a partir de C_0, ou seja, tais que $C_0 \vdash^* C_i$. Os arcos são definidos entre cada par de configurações C_1 e C_2 tais que C_2 seja imediatamente alcançável a partir de C_1. Neste caso, o evento e é usado para rotular o arco.

Os caminhos no grafo determinam seqüências de ocorrências no sistema – e vice-versa. Os modelos podem ser analisados a partir de simulações ou da exploração dos caminhos do grafo de alcançabilidade.

Capítulo 3

RPOOt: Uma Extensão Temporizada de RPOO

No capítulo anterior apresentamos uma visão informal de RPOO, seguida da formalização dos seus principais aspectos. Neste capítulo, discutiremos conceitualmente as características de nossa estratégia de temporização. Apresentaremos igualmente uma visão informal de RPOOt e depois seguiremos com sua formalização propriamente dita.

3.1 RPOOt - Idéia Geral

Todas as interações entre objetos RPOO se dão através dos seis tipos de *ações* definidos para o formalismo. O primeiro ponto considerado foi justamente a escolha de quais desses tipos de ações deveriam ser temporizados. Normalmente, a necessidade de se temporizar um ou outro tipo de ação surgiria como conseqüência de aplicações práticas do formalismo RPOO. A falta de ferramentas apropriadas muitas vezes obrigou os desenvolvedores a produzir modelos RPOO e simulá-los (automaticamente) através de modelos CPN equivalentes[1]. A utilização de determinados tipos de ações RPOO, entretanto, produz modelos equivalentes demasiadamente complexos, dificultando bastante o processo de análise dos modelos através de simulação automática. Desta forma, a maioria das aplicações práticas (não-didáticas) do formalismo (que se valeram de simulação automática) faziam uso quase que exclusivamente do envio e recepção de mensagens síncronas e assíncronas.

[1] Há um algoritmo de conversão de RPOO para CPN no trabalho de Guerrero [18]

Uma dessas aplicações (IP Móvel [9; 36]) evidenciou que o tratamento explícito de parâmetros temporais em mensagens assíncronas trocadas entre objetos nos permitiria endereçar adequadamente diversas funcionalidades do protocolo modelado, bem como medir seu desempenho em diferentes circunstâncias. Saída assíncrona de dados foi, portanto, o primeiro tipo de ação que consideramos em nosso modelo temporizado. Outro tipo de ação que optamos por temporizar é a instanciação. Apesar de não termos nenhum experimento prático que comprove a necessidade de temporizar este tipo de ação, entendemos que este mecanismo pode ser útil na modelagem de diversos protocolos onde a alocação de recursos não pode ser tratada de maneira instantânea.

Outra perspectiva importante, à parte a escolha das ações a serem temporizadas, diz respeito ao suporte a modelos analíticos - que oferecem uma análise mais precisa em termos de desempenho - ou modelos de simulação - que apresentam menos restrições matemáticas, conforme explicamos no Capítulo 1. RPOOt oferece suporte para modelos de simulação, pois estes dão maior liberdade ao desenvolvedor na modelagem de sistemas cujas funcionalidades apresentam atrasos arbitrários, não necessariamente seguindo alguma distribuição estatística exigida pelo formalismo.

A escolha do suporte a modelos de simulação é importante, também, para integrar o formalismo RPOOt com o formalismo utilizado na descrição das classes. RPOO usa redes de Petri coloridas (CPN's [21]) para a descrição das classes. Por conseguinte, a escolha mais adequada para tal, em RPOOt, é a extensão temporizada das CPN's, as TCPN's [22], que ofereceriam ao desenvolvedor a possibilidade da modelagem temporal de atividades internas aos objetos, já que RPOOt, em si, concentra-se na interação entre objetos.

Nas TCPN's os modelos possuem um relógio global e os dados que transitam nas redes de Petri possuem uma informação temporal (*timestamp*) que indica em que instante eles podem ser utilizados. As regras de disparo das transições são devidamente definidas levando-se em consideração a disponibilidade das informações (fichas), de acordo com tempo indicado no relógio global. É esta a idéia central da qual nos valemos para a formalização de RPOOt, o que facilitará a integração dos dois formalismos.

Desta forma, nossa extensão temporizada deve, em linhas gerais tratar, contemplar dois pontos importantes:

- O acréscimo de um relógio global aos modelos e a temporização das mensagens tro-

cadas entre os objetos;

- A definição de regras de transição que levem em consideração tanto o relógio global como a informação temporal inerente às mensagens.

As definições RPOOt incidem na *configuração* de sistemas de objetos. A exemplo do capítulo anterior, dividimos a configuração em *estrutura* e *dinâmica*. O primeiro ítem supracitado diz respeito à estrutura dos sistemas de objetos RPOOt, ao passo que o segundo será tratado em sua dinâmica.

Apresentamos, num primeiro momento, uma visão informal de RPOOt, para depois abordarmos os aspectos formais de estrutura e dinâmica da configuração dos sistemas de objetos.

3.2 RPOOt — Visão Informal

Criação não-instantânea De acordo com o que foi discutido na seção anterior, os sistemas de objetos RPOOt conterão, em sua estrutura, um relógio global (ou tempo de modelo — *model time*), e as mensagens carregarão uma informação temporal. Além disso, para proporcionarmos a criação não-instantânea de objetos, os sistemas de objetos relacionarão — através de uma *função de validade* — cada instância com o tempo de modelo a partir do qual elas poderão ser utilizadas em qualquer ação.

A Figura 3.1 nos mostra a configuração de um sistema de objetos com apenas um objeto, no qual uma ação RPOOt de *instanciação temporizada* pode ser executada. O **tempo de modelo** tem valor **0** e a função de **validade** indica o tempo (**0**) a partir do qual o objeto **obj1** poderá ser utilizado. A transição **transition0** está *habilitada* e, ao *disparar*, executa sua ação RPOOt, criando um novo objeto, **obj2**, do tipo ilustrativo **RpootObject**.

A ação **obj2=new RpootObject@10** indica que o processo de instanciação deve tomar **10** unidades de tempo. O efeito de sua execução é ilustrado na Figura 3.2. Note que o objeto que a executou terá, nas estruturas internas do sistema de objetos, uma referência (ligação) para o objeto criado, mas esta ligação não viabiliza, ainda, o envio de mensagens de **obj1** para **obj2** (nem qualquer outra ação envolvendo **obj2**). Isto ocorre porque o objeto recém-criado, ilustrado em linha tracejada, ainda não é válido, pois a função de **validade** do sistema de objetos indica que ele só poderá ser utilizado de qualquer forma uma vez que o tempo de

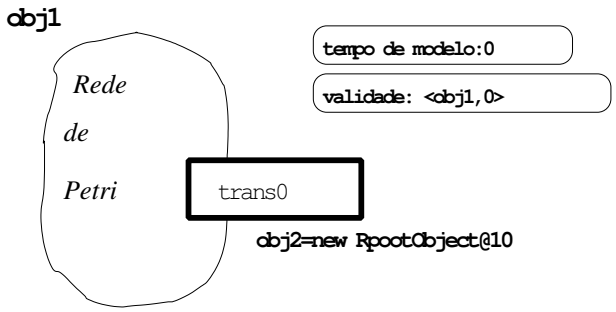

Figura 3.1: 'obj1' pode criar um novo objeto, do tipo RpootObject

modelo seja maior ou igual a **10** (na figura, temos que o **tempo de modelo** é igual a **0**). Na prática, podemos interpretar este efeito como um indício de que, durante as 10 unidades de tempo posteriores ao instante da execução de **obj2=new RpootObject@10** (Figura 3.1), **obj2** ainda *estará sendo criado*.

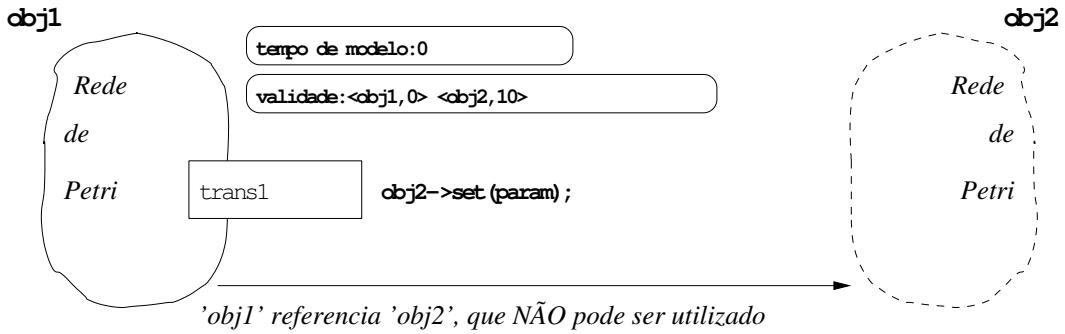

Figura 3.2: 'obj1' *possui* uma referência para 'obj2', que ainda não pode ser utilizado

Quando o tempo de modelo atingir o valor indicado na função de **validade** para **obj2**, este poderá finalmente ser utilizado, e a ação RPOO **obj2.set(param)** poderá ser executada por **obj1**, admitindo que **transition1** está habilitada por sua respectiva rede de Petri. A Figura 3.3 ilustra este cenário.

Mensagens assíncronas temporizadas Em RPOOt, a informação temporal das mensagens presentes na estrutura dos sistemas de objetos indica o tempo de modelo a partir do qual elas podem ser efetivamente consumidas por ações de entrada. Apenas mensagens cujo *timestamp* seja menor ou igual ao tempo de modelo podem ser consumidas: dizemos que tais mensagens estão *prontas*. Quando nenhuma ação puder ser executada em nenhum objeto do sistema, o relógio global deve avançar para o menor valor de tempo onde uma ação

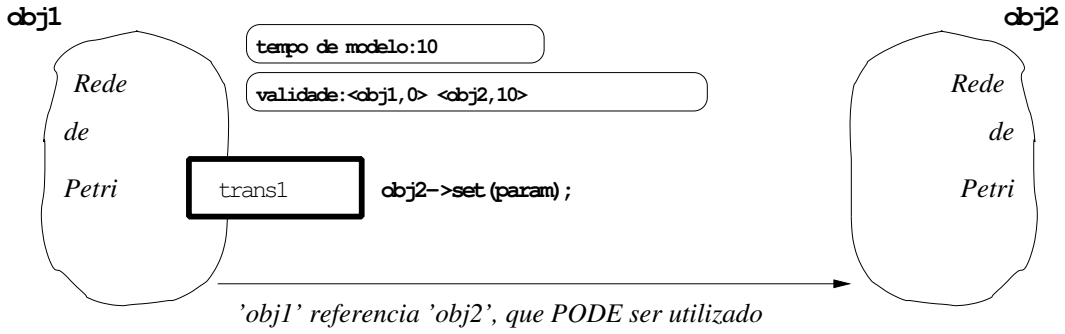

Figura 3.3: 'obj1' *possui* uma referência para 'obj2'

(em qualquer objeto) é passível de execução.

Desta forma, as ações de saída (assíncrona) temporizada podem especificar o *timestamp* dos dados trocados entre os objetos como sendo t unidades de tempo maior que o tempo corrente de modelo, o que indicaria que a troca de mensagem *levaria*, no mínimo, t unidades de tempo.

A Figura 3.4 nos dá uma visão informal de uma configuração (temporizada em sua estrutura) de um sistema de objetos com dois objetos. Omitiremos a função de validação, admitindo que ambos os objetos são *válidos*. O tempo de modelo é **10** e a ação **obj2.set(param)@+5** pode ser executada através do disparo de **transition1**, que está habilitada. O **@+5**, na ação, indica que, a informação **param** enviada a **obj2** terá um *timestamp* 5 unidades de tempo maior que o tempo corrente de modelo.

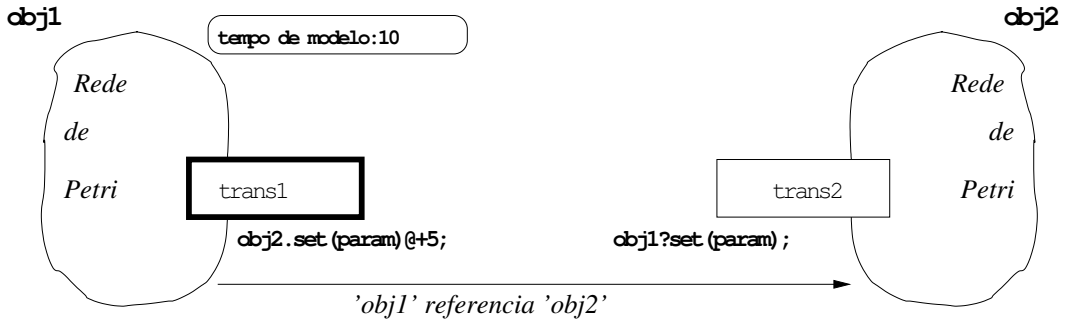

Figura 3.4: 'obj1' pode enviar uma mensagem temporizada para 'obj2'

A configuração resultante da execução da ação **obj2.set(param)@+5** é ilustrada na Figura 3.5. Podemos notar que há uma mensagem pendente (**param@15**) de **obj1** para **obj2**. Podemos notar, também, que **transition2**, em **obj2** não está habilitada, uma vez que a informação temporal (**15**) da mensagem necessária à execução de sua ação (**obj1?set(param)**)

consiste de um valor maior que o tempo corrente de modelo (**10**).

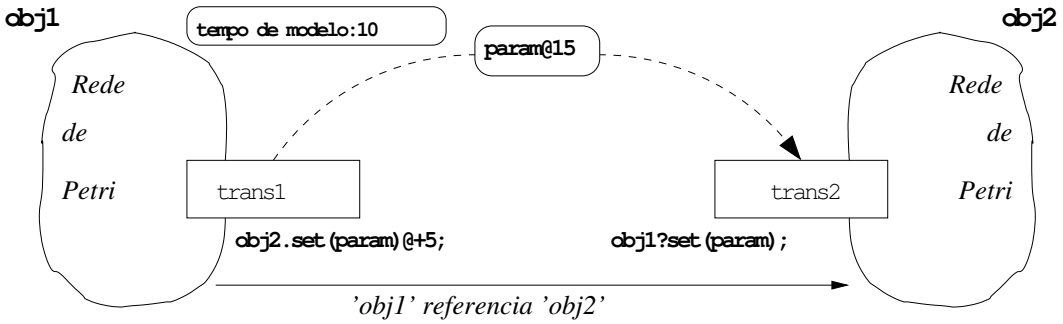

Figura 3.5: Uma mensagem temporizada *pendente* de 'obj1' para 'obj2'

Considerando que, no cenário ilustrado, não há mais nenhuma ação passível de execução no tempo corrente de modelo (Figura 3.5), o tempo de modelo será reconfigurado para o menor valor de tempo onde alguma ação pode ser executada. Esta operação resultará na configuração que mostramos na Figura 3.6. Com o tempo de modelo igual a **15**, a mensagem **param** está *pronta* para ser consumida, e a ação **obj1?set(param)** em **obj2** poderá ser executada. O efeito da execução desta ação, no tocante ao sistema de objetos, é o *consumo* da mensagem **param**, que deixará de fazer parte de sua estrutura. Esta configuração é ilustrada na Figura 3.7.

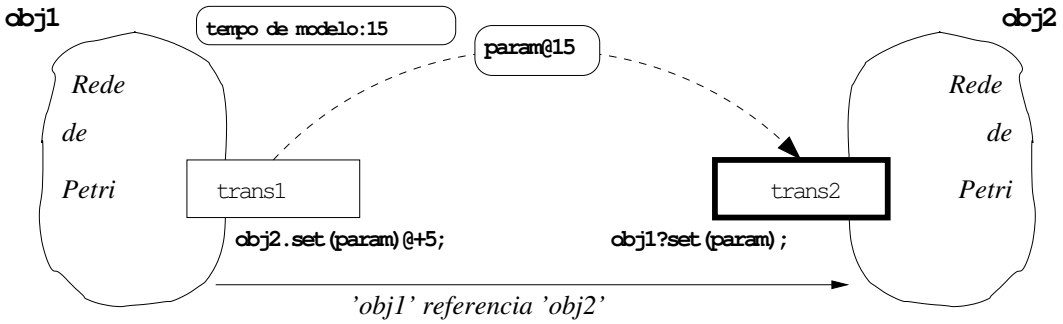

Figura 3.6: Mensagem temporizada pode ser consumida por 'obj2'

3.3 Redes de Petri Temporizadas - TCPN

Em RPOOt, substituiremos as CPN's pelas TCPN's, no que diz respeito ao formalismo utilizado para descrever o corpo das classes. Iniciaremos a formalização de RPOOt, portanto,

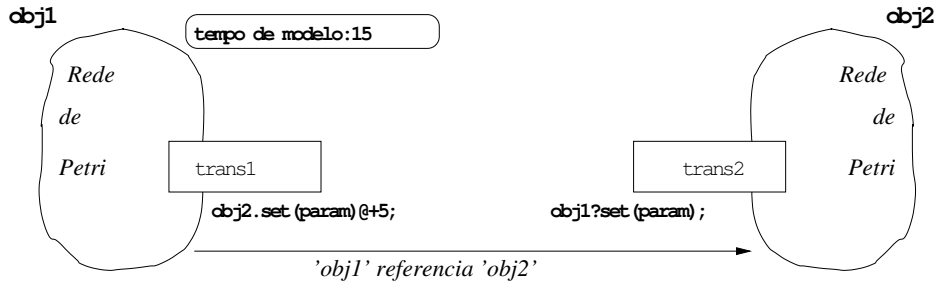

Figura 3.7: Mensagem temporizada foi consumida por 'obj2'

apresentando a definição de TCPN. Conforme já antecipamos, a idéia geral das Timed Colored Petri Nets é a de estender as CPN's, acrescentando aos modelos um relógio global (o *model time* ou tempo de modelo) e permitindo que as fichas manipuladas pelas redes possam carregar uma informação temporal (que indica o tempo de modelo a partir do qual elas podem efetivamente ser utilizadas pelas transições). Nesta seção, apresentaremos a formalização das TCPN's, tal qual descrita no trabalho de Jensen [22], utilizando, no entanto, a mesma notação que adotamos no Capítulo 2, baseada na assinatura Σ.

Começamos por apresentar algumas definições e notações acerca de multi-conjuntos temporizados:

Definição 3.1 *Um multi-conjunto temporizado tm, sobre um conjunto não vazio S, é uma função $tm : S \times \mathbb{R} \rightarrow \mathbb{N}$, onde a soma*

$$tm(s) = \sum_{r \in \mathbb{R}} tm(s, r)$$

é finita para todo $s \in S$. O inteiro não negativo tm(s) é o número de ocorrências do elemento s no multi-conjunto temporizado tm. A lista:

$tm[s] = [r_1, r_2, r_{tm(s)}]$ *contém os valores de tempo $r \in \mathbb{R}$ para os quais $tm(s, r) \neq 0$.*

É comum representarmos um multi-conjunto temporizado por uma soma formal:

$\sum_{s \in S} tm(s)`s@tm[s]$

Utilizamos a notação S^{tms} para indicar o conjunto de todos os multi-conjuntos temporizados possíveis sobre S.

Para $tm \in S^{tms}$ e $r \in \mathbb{R}$, definimos $tm_r \in S^{tms}$ como segue:

$$tm_r = \sum_{r \in \mathbb{R}} tm(s)@tm[s]_r$$

onde $tm[s]_r$ é a lista obtida a partir de $tm[s]$ pela adição de r a cada um de seus elementos.

Da mesma forma, para um multi-conjunto convencional $m \in S^{ms}$ e $r \in \mathbb{R}$ definimos $m_r \in S^{tms}$ como sendo:

$$m_r = \sum_{r \in \mathbb{R}} m(s)@[r, r, ..., r]$$

onde a lista $[r, r, ..., r]$ tem tamanho $m(s)$ para todo $s \in S$.

Estendemos, agora, a definição de redes de Petri coloridas (2.4):

Definição 3.2 (TCPN) *Uma Rede de Petri Colorida Temporizada é uma tupla $TCPN = \langle CPN, r_0 \rangle$, onde:*

CPN é uma rede de Petri colorida (2.4) onde $E : F \rightarrow \mathcal{T}^{ms} \cup \mathcal{T}^{tms}$ e $I : P \rightarrow \mathcal{T}^{ms} \cup \mathcal{T}^{tms}$

$r_0 \in \mathbb{R}$ é um numero real, chamado tempo inicial

Definição 3.3 *Uma marcação é um multi-conjunto temporizado sobre o conjunto de fichas da rede (2.5). A marcação inicial m_0 é obtida pela avaliação das expressões de inicialização no tempo r_0:*

$$\forall p \in P, m_0(p) = I(p)_{r_0}.$$

Um estado é um par $\langle m, r \rangle$ onde m é uma marcação e r é um valor de tempo.

Denotamos por M e S os conjuntos de todas as marcações e estados de uma rede, respectivamente.

Definição 3.4 *Seja t^a uma transição em um modo a. t^a está habilitada no estado $s = \langle m, r \rangle$, no tempo r' se, e somente se, as seguintes propriedades são satisfeitas:*
(i)$\forall p \in P,\ (\llbracket E(p,t) \rrbracket_a) r' \leq m(p)$
(ii)$r \leq r'$
(iii)r' é o menor elemento de R para o qual existe uma t^a satisfazendo (i) e (ii).
Denotamos por $s[t^a, r'\rangle$ uma transição t, em um modo a, habilitada no estado s, no tempo r'.

A primeira condição da definição anterior nos garante que apenas estarão habilitadas transições em modos para os quais tenhamos as fichas necessárias **com** *timestamps* adequados. Dizemos que tais transições, em seus respectivos modos, estão *prontas*. A segunda

condição garante que o tempo nunca *volta*, ao passo que a terceira garante que as transições estarão habilitadas, em seus respectivos modos, na seqüência em que tornam-se *prontas*.

Definição 3.5 *Quando uma transição t, no modo a, está habilitada no estado $\langle m, r\rangle$, no tempo r', ela pode* disparar*, mudando o estado $\langle m, r\rangle$ para o estado $\langle m', r'\rangle$ onde:*

$$\forall p \in P, m'(p) = m(p) - (\llbracket E(p,t) \rrbracket_a)_{r'} + (\llbracket E(t,p) \rrbracket_a)_{r'}$$

Se $s[t^a, r\rangle s'$, dizemos que s' é diretamente alcançável *a partir de s. Chamamos cada $s[t^a, r\rangle s'$ de* disparo *ou de* ocorrência*.*

3.4 Estrutura de um Sistema de Objetos RPOOt

Inicialmente, estendemos RPOO de forma que as *mensagens pendentes* na *estrutura* de um *sistema de objetos* sejam um multi-conjunto temporizado *ou* um multi-conjunto *convencional*. A estrutura conterá, também, um relógio global e uma função de validade, que relacionará os objetos da estrutura com o tempo a partir do qual eles podem ser utilizados.

Estendemos, agora a definição de *estrutura* de um sistema de objetos (2.13), apresentando o que chamamos de *estrutura temporizada* ou *estrutura RPOOt*. Mais uma vez, a exemplo da notação utilizada no capítulo anterior, $\mathcal{O}$ denota um conjunto de objetos e D um domínio de dados, ambos potencialmente infinitos.

Definição 3.6 *Uma estrutura temporizada é a tupla $\langle E, v, r\rangle$, onde:*

E é uma estrutura RPOO (Definição 2.13), com $M \subseteq \mathcal{O} \times D \times \mathcal{O}^{ms} \cup \mathcal{O} \times D \times \mathcal{O}^{tms}$

$v : O \to \mathbb{R}$ é a função de validade, onde O é o conjunto de objetos vivos de E;

$r \in R$ é o tempo do modelo *ou* relógio global*.*

A função de validade v nos indica o tempo a partir do qual os objetos vivos são válidos *(podem ser* utilizados*) no âmbito do sistema de objetos, ou seja, o tempo a partir do qual referências aos objetos vivos podem ser utilizadas na execução de qualquer tipo de ação. Note que a função de validade* não *deve ser interpretada como um indicativo do tempo* até o qual *um objeto será válido. Um objeto $'o'$ será válido se, e somente se, $v(o) \leq r$.*

A definição de estruturas temporizadas proporciona a criação de regras de transição que permitam o envio assíncrono de mensagens temporizadas e a criação não-instantânea de

Ação temporizada	Descrição
c=new $C@delay$	Criação não-instantânea
$x.mensagem@delay$	Saída de dados (assíncrona) temporizada

Tabela 3.1: Ações *temporizadas*

objetos RPOOt.

3.5 Dinâmica de um Sistema de Objetos RPOOt

Nesta seção, apresentaremos, primeiramente, quais ações RPOO devem ser temporizadas. Em seguida, estendemos o comportamento de um sistema de objetos de modo que tais ações apresentem a semântica adequada, levando em consideração os fatores de tempo introduzidos na Definição 3.6.

A Tabela 3.1 nos apresenta as duas ações RPOO que serão temporizadas, descrevendo-as, informalmente.

Além das inscrições que denotam ações temporizadas, RPOOt preservará todas as inscrições RPOO originais, como vemos a seguir.

Definição 3.7 (Inscrições de interação) *Seja t um termo, s um termo-objeto, $\mathbb{C}$ um nome de classe, v uma variável-objeto e d um termo (com $[\![d]\!] \in \mathbb{R}$). A seguinte sintaxe abstrata caracteriza as inscrições de interação RPOOt:*

$$I ::= s.t \mid s.t@d \mid s!t \mid s?t \mid v{:}\mathbb{C} \mid v{:}\mathbb{C}@d \mid \widetilde{s} \mid \text{end}$$
$$R ::= \epsilon \mid I \circ R$$

Efeito de Ações sobre Estruturas Redefinimos, agora, as regras de transição que descrevem os efeitos das ações RPOO definidas no Capítulo 2, uma vez que promovemos mudanças com respeito à *estrutura* (da configuração) dos sistemas de objetos. Denotaremos por $E_{v,r}$ a estrutura *temporizada* $\langle E, v, r\rangle$. Utilizamos a notação $\langle E_{v,r}, a\rangle \rightarrow E'_{v',r'}$ para indicar que $E'_{v',r'}$ é a estrutura temporizada resultante da aplicação da ação a sobre a estrutura E.

Para representar as ações, vamos utilizar a sintaxe e os símbolos apresentados na Definição 3.7. Ainda no que diz respeito à notação utilizada, o elemento que precede o

símbolo ":" representará o objeto agente da ação, nas definições que se seguem.

Definição 3.8 (Ação interna) *Seja $E_{v,r}$ uma estrutura temporizada e $x : \tau$ uma ação interna ao objeto x, então*

$$\langle E_{v,r}, x{:}\tau \rangle \rightarrow E_{v,r},\ se\ v(x) \leq r$$

Conceitualmente, o efeito de uma ação interna sobre uma estrutura é nulo. Note que apenas objetos cuja função de validade é inferior ao tempo de modelo podem executar ações internas. Esta é a única restrição adicionada à definição de ações internas RPOO (2.15).

Definição 3.9 (Ação de criação temporizada) *Seja $E_{v,r}$ uma estrutura temporizada,* x:y=new Y@d, *$d \in \mathbb{R}$, uma ação de criação em que o objeto x cria o objeto y e $v' : O \rightarrow \mathbb{R}$ uma função com $v'(y) = r + d$ e $v'(z) = v(z)$ se $z \neq x$, então*

$$\langle E_{v,r}, x{:}y = new\ Y \rangle \rightarrow [E + y + x\vec{y}]_{v',r},\ \textit{se } y \notin! E \textit{ e } v(x) \leq r.$$

A ação de criação temporizada é capaz de expressar as criações instantâneas e não-instantâneas de objetos. Quando o termo d for maior que zero, o objeto y não poderá ser utilizado (não poderá executar ou ser utilizado em ações de nenhuma natureza) enquanto o tempo de modelo for menor que $r + d$. Na prática, isto indica que o objeto y *leva* d unidades de tempo para ser criado. Caso o termo d seja avaliado em um número menor ou igual a zero, a criação do objeto y será instantânea e o objeto recém-criado poderá executar ações logo após sua criação, o que equivale ao efeito descrito na Definição 2.16. O objeto x, executor da ação, deve ser um objeto *válido*, ou seja, sua função de validade deve indicar um número menor ou igual ao tempo de modelo r. Note que a única diferença entre a função v' e a função v é que aquela é definida também para o objeto y. O símbolo $\notin!$ denota a não-pertinência do objeto y ao conjunto de objetos vivos de E. Desta forma, a ação só pode ser executada se o objeto criado não fizer parte da estrutura anterior à execução da ação.

Definição 3.10 (Ação de entrada de dados) *Seja E uma estrutura RPOO; $x : y?m$ uma ação de entrada de dados, em que o objeto x consome a mensagem m enviada pelo objeto y; $[m_y^x]@[t]$ o multi-conjunto temporizado $1'\langle x, m, y\rangle@[t]$, $t \in \mathbb{R}$, $\langle x, m, y\rangle \in \mathcal{O} \times D \times \mathcal{O}$; $\bar{z}$ uma referência a $z \in \mathcal{O}$ temos*

$$\langle (E + m_y^x@[t])_{v,r}, x{:}y?m \rangle \rightarrow (E + x\vec{z})_{v,r}, \textit{se } m = \bar{z},\ v(x) \leq r,\ v(z) \leq r \textit{ e } t \leq r$$

$$\langle (E + m_y^x)_{v,r}, x{:}y?m \rangle \rightarrow E_{v,r}, \textit{se } m \neq \bar{z}\ (z \in O),\ v(x) \leq r \textit{ e } t \leq r.$$

Estendemos a Definição 2.17 de modo que o relógio global e a informação temporal das mensagens sejam levados em consideração nas ações de entrada de dados. Apenas mensagens temporizadas cujo *timestamp* t seja menor ou igual ao tempo de modelo r podem ser consumidas. Tais mensagens são chamadas mensagens *prontas*. Com a execução da ação de entrada, a mensagem consumida deixa de fazer parte da estrutura do sistema de objetos. Se a mensagem consumida contém uma referência para outro objeto, então uma nova ligação entre o objeto receptor da mensagem e a referência contida nela é criada. Note que a definição anterior não restringe o consumo de mensagens para situações onde haja apenas um elemento m_y^x no multi-conjunto de mensagens: a estrutura E nos abstrai de todas as outras mensagens pendentes.

Definição 3.11 (Ação de saída de dados temporizada) *Seja $E_{v,r}$ uma estrutura temporizada, $x{:}y.m@d$ uma ação de saída temporizada (e assíncrona) de dados, em que o objeto x envia a mensagem m para objeto y, definimos que*

$$\langle E_{v,r}, x{:}y.m \rangle \rightarrow E + m_y^x[r + d], \textit{se } \vec{xy} \in E,\ v(x) \leq r \textit{ e } v(y) \leq r$$

Mais uma vez, estendemos uma definição RPOO (2.18) para levar em conta o relógio global das estruturas temporizadas e a informação temporal das mensagens. Para saída de dados (envio de mensagens) temporizada, as mensagens que integrarão a estrutura do sistema de objetos (com a execução das ações de saída) não poderão ser consumidas antes que o tempo de modelo seja maior ou igual a $r + d$. Na prática, em face da semântica definida para entrada de dados em 3.10, isto indica que a operação *leva*, no mínimo, d unidades de tempo para tomar efeito. Veja que a mensagem não será *forçosamente* consumida quando o tempo de modelo chegar a $r + d$: este é o tempo de modelo *a partir do qual* ela estará disponível (*pronta*).

O envio de mensagem temporizada engloba o envio assíncrono de mensagem em RPOO. De fato, se d for avaliado num valor menor ou igual a zero, o efeito do envio de mensagem temporizada, em RPOOt, será o mesmo do envio de mensagem assíncrona, em RPOO, pois a mensagem estará disponível para consumo no mesmo tempo de modelo em que foi enviada.

Definição 3.12 (Ação de saída síncrona de dados) *Seja $E_{v,r}$ uma estrutura temporizada, $x{:}y!m$ uma ação de saída síncrona de dados, em que o objeto x envia a mensagem m para objeto y, definimos que*

$$\langle E_{v,r}, x{:}y!m\rangle \rightarrow E + m_y^x[r], \textit{se } \vec{xy} \in E,\ v(x) \leq r \textit{ e } v(y) \leq r$$

O efeito das ações de saída de dados síncrona, em RPOOt é o mesmo de suas equivalentes em RPOO (Definição 2.18). Elas estarão disponíveis pra consumo (e deverão ser consumidas) *sempre* no mesmo tempo de modelo (r) em que foram enviadas. A única ressalva é que ambos os objetos envolvidos na troca de mensagens devem ser *válidos*.

Definição 3.13 (Ação de desligamento) *Seja $(E + \vec{xy})_{v,r}$ uma estrutura temporizada, $x{:}unlink\ y$ uma ação de desligamento, em que o objeto x se desliga do objeto y, definimos que*

$$\langle (E + \vec{xy})_{v,r}, x{:}unlink\ y\rangle \rightarrow E, \textit{se } \vec{xy} \notin E,\ v(x) \leq r \textit{ e } v(y) \leq r.$$

O efeito desta ação é o mesmo de sua ação equivalente em RPOO (Definição 2.19), novamente com a única ressalva de que ambos os objetos envolvidos na ação devem ser *válidos*.

Definição 3.14 (Ação de auto-destruição) *Seja $E_{v,r}$ uma estrutura temporizada, X o conjunto de todas as ligações com origem no objeto x, $x{:}end$ uma ação de finalização em que o objeto x se destrói, v' uma função com $v'(z) = v(z)$ se $z \neq x$ e $v'(z)$ indefinida para $z = x$, temos*

$$\langle (E + X + x)_{v,r}, x{:}end\rangle \rightarrow E,\ \textit{se } v(x) \leq r.$$

Ações de auto-destruição, assim como todas as outras, só podem ser executadas por objetos válidos.

Passagem do tempo Até agora abordamos regras de transição cujas definições levam em conta informações temporais em face de um relógio global. Agora definimos o mecanismo pelo qual o relógio global pode *avançar*.

Definição 3.15 *Seja A o conjunto de todas as ações, E^t o conjunto de todas as estruturas temporizadas, $E_{v,r}$ e $E_{v,r'}$ estruturas temporizadas e $r' > r$ o menor valor em $\mathbb{R}$ para o qual alguma ação pode ser executada em $E_{v,r'}$, então*

$$\nexists a \in A, E_1 \in E^t | \langle E_{v,r}, a \rangle \rightarrow E_1 \implies \langle E_{v,r} \rangle = \langle E_{v,r'} \rangle \; e \; \langle E_{v,r}, \emptyset \rangle \rightarrow \langle E_{v,r'} \rangle$$

Quando nenhuma ação puder ser executada numa determinada estrutura, o sistema de objetos deve passar para uma configuração onde o tempo de modelo de sua estrutura é o menor valor (maior que o tempo corrente) onde alguma ação pode ser executada. Vale salientar que a Definição 3.15 considera todas as ações possíveis, o que inclui ações internas. Isto facilita a integração de RPOOt com as TCPN's. Usando-se este tipo rede de Petri temporizada para descrevermos o comportamento dos objetos, o relógio global deverá mover-se para o menor tempo, entre todos os objetos do sistema, onde uma ação é possível. Em outras palavras, o relógio da estrutura do sistema de objetos RPOOt estará *sincronizado* com o relógio das redes que representam seus objetos.

Apesar de primarmos por uma utilização de RPOOt com as TCPN's, as definições das ações RPOOt incidem no âmbito da configuração do sistemas de objetos, mantendo-se razoavelmente independente do formalismo utilizado para a descrição das classes. Desta forma, pode-se utilizar RPOOt em conjunto com redes não-temporizadas (como as CPN's) ou mesmo em conjunto com um formalismo fora do domínio das redes de Petri[2].

Ações compostas e eventos Em RPOOt, assim como em RPOO "puro", as ações compostas resultam na concatenação do efeito das ações elementares da composição. Portanto, ações compostas são definidas para RPOOt da mesma forma que para RPOO (Definição 2.21). O mesmo acontece com os eventos, que preservam a semântica operacional descrita na Definição 2.22.

Configurações e ocorrência/alcançabilidade Uma vez que estruturas fazem parte do que definimos como configuração e estamos tratando de estruturas temporizadas, estendemos a definição de configuração (2.24).

[2]O mesmo acontece com o formalismo original, cujas definições relativas ao sistema de objetos permitem a utilização de qualquer formalismo para a descrição das classes

Definição 3.16 (Configuração temporizada) *Uma* configuração temporizada *ou* configuração RPOOt *de um sistema de objetos consiste de uma estrutura temporizada $E_{v,r}$; um conjunto de estados internos para cada objeto vivo na estrutura (mais uma vez, denotamos cada conjunto de estados internos por σ); e uma relação $\longrightarrow$ que caracteriza o comportamento desses objetos. Denotamos a configuração pela tupla $\langle E_{v,r}, \sigma, \longrightarrow \rangle$.*

Formalmente, a única diferença entre uma configuração RPOO (2.24) e uma configuração RPOOt (3.16) é o uso, nesta última, de uma estrutura temporizada (3.6) em vez de uma estrutura *convencional* (2.13).

Nesta mesma linha de raciocínio, definimos ocorrência e alcançabilidade sobre a definição de configurações temporizadas. A relação de ocorrência, escrita $Ct \vdash Ct'$, indica que a ocorrência de um evento altera a configuração temporizada de um sistema de Ct para Ct'. Definimos alcançabilidade através da notação $Ct \vdash^* Ct'$. Dizemos que Ct' é *alcançável* a partir a da configuração Ct. A seguinte regra caracteriza a relação:

Definição 3.17 *Seja $E_{v,r}$ uma estrutura temporizada e e um evento composto de ações RPOOt dos objetos pertencentes a $E_{v,r}$, então*

$$\frac{\langle E_{v,r}, e\rangle \rightarrow E'_{v,r} \quad \langle \sigma, e\rangle \rightarrow \sigma'}{\langle E_{v,r}, \sigma\rangle \vdash \langle E'_{v,r}, \sigma'\rangle}, \qquad \textit{onde } x{:}a \in e \Rightarrow x \in E$$

Grafos de alcançabilidade temporizados Podemos representar o comportamento de um modelo no sistema de objetos através de seu grafo de alcançabilidade. Neste caso, os vértices correspondem às configurações alcançáveis do sistema e os arcos aos eventos.

Definição 3.18 *Seja Ct_0 a configuração inicial de um sistema de objetos e E seu conjunto de eventos. Seu* grafo de alcançabilidade temporizado*, denotado por $\mathrm{G}(Ct_0)$, é um grafo E-rotulado $\langle V, A\rangle$, onde*

$$V = \{\ Ct_i \mid Ct_0 \models^* Ct_i\ \}$$
$$A = \{\ \langle Ct_1, e, Ct_2\rangle \in V \times E \times V \mid Ct_1 \vdash_e Ct_2\ \}.$$

Um sistema de objetos é representado por sua configuração temporizada inicial Ct_0. O conjunto de vértices de $\mathrm{G}(Ct_0)$ é definido como o conjunto das configurações Ct_i alcançáveis a partir de Ct_0, ou seja, tais que $Ct_0 \models^* Ct_i$. Os arcos são definidos entre cada

par de configurações Ct_1 e Ct_2 tais que Ct_2 seja imediatamente alcançável a partir de Ct_1. Neste caso, o evento e é usado para rotular o arco.

Os caminhos no grafo determinam seqüências de ocorrências no sistema—e vice-versa. Os modelos podem ser analisados a partir de simulações ou da exploração dos caminhos do grafo de alcançabilidade.

Modelo RPOOt Por fim, estendemos a Definição 2.1, definindo um modelo RPOOt:

Definição 3.19 (Modelo RPOOt) *Um modelo RPOOt $\mathcal{M}$ é definido como:*

$$\mathcal{M} = \langle \mathbb{C}, \mathcal{O}_{v_0,r_0} \rangle$$

onde $\mathbb{C}$ é um conjunto de classes identificadas para o problema modelado e $\mathcal{O}$ é a configuração inicial *do sistema de objetos, tendo sua estrutura temporizada $\langle E, v_0, r_0 \rangle$, com $r_0 = 0$ e $v_0(x) = r_0,\ \forall x \in E$.*

Capítulo 4

Protótipo de um Simulador RPOOt

Abordamos, neste capítulo, a implementação do protótipo de um simulador RPOOt, de acordo com o formalismo definido no Capítulo 3. O simulador RPOOt será uma extensão da ferramenta Renew (*Rerefence Nets Workshop* [27]), um simulador desenvolvido na Universidade de Hamburgo (Alemanha), com suporte a Redes de Referência — um formalismo que, assim como RPOO, integra redes de Petri e orientação a objetos.

4.1 Visão Geral sobre Redes de Referência

As tentativas de integração entre redes de Petri e orientação a objetos recaem, geralmente, sobre duas abordagens: *objetos dentro de redes* [2] — onde dados que transitam por redes de Petri são objetos — ou *redes dentro de redes* [18] — onde o comportamento dos objetos é dado por uma rede de Petri e as fichas da rede podem ser, também, redes.

O formalismo *Redes de Referência*, implementado em Renew, faz uso dessas duas abordagens ao mesmo tempo: as redes de Petri definem classes de objetos e as informações utilizadas nas redes podem ser tanto objetos Java como objetos descritos a partir de redes de Petri. Como em RPOO (e RPOOt) a linguagem de descrição dos dados é deixada em aberto (utilizamos uma assinatura Σ para descrever os tipos e operações de forma abstrata), não há maiores obstáculos, neste sentido, em estender a ferramenta de modo que ela dê suporte à construção e simulação de modelos RPOOt.

Outro fator importante é a existência, nas redes de referência, de inscrições cuja semântica seria *equivalente* à de inscrições que denotam as ações RPOO de instanciação, saída

Método	Descrição
obj:new Obj	cria um novo objeto, *obj*, cujo comportamento será descrito pela rede/classe Obj
obj:method(params)	passa as informações representadas por *params*, de modo sincronizado, para o objeto *obj*, através do canal *method*. Isto equivale a chamar o método *method* do objeto *obj* passando *params* como parâmetro
:method(params)	recebe as informações representadas por *params*, de modo sincronizado

Tabela 4.1: Inscrições das redes de referência

síncrona e entrada de dados, o que facilita o processo de implementação. Ainda em relação à escolha de Renew como base do nosso protótipo de simulador RPOOt, outro ponto positivo é o fato de que as classes das redes de referência podem ser descritas por TCPN's (com Java sendo usada como linguagem de descrição de dados), da mesma forma que em RPOOt, com sincronia entre os relógios de todos os objetos.

Em redes de referência, classes são descritas como redes de Petri, nas quais Java é utilizada como *linguagem de inscrição*. Assim como em RPOOt, a interação entre objetos é representada através de inscrições junto a transições. A Tabela 4.1 descreve informalmente a sintaxe das inscrições das redes de referência.

As inscrições do tipo **obj:new Obj** têm a mesma função das inscrições de instanciação RPOO. A execução deste tipo de inscrição cria novas instâncias de classes baseadas em redes de Petri. A instanciação, assim como em RPOO, viabiliza a troca de mensagens entre o objeto executor da inscrição e o objeto recém-criado.

Inscrições do tipo **obj:method(params)** são executadas em sincronia com inscrições correspondentes do tipo **:method(params)**. Tais inscrições são equivalentes às inscrições de saída síncrona de dados e entrada de dados RPOO, respectivamente. Note que, ao contrário do que acontece em RPOO, não precisamos explicitar a identificação do objeto que *chama* o método na inscrição de entrada. O nome **method** nas inscrições citadas anteriormente pode ser compreendido como um nome de método de uma classe, mas é também chamado

de *canal de sincronização*, *downlink* (no caso de inscrições de saída), ou ainda *uplink* (no caso de inscrições de entrada). As redes de referência, tal que implementadas em Renew, apresentam a limitação de não permitir mais de um *uplink* por transição (o que não ocorre em RPOO com a ação de entrada de dados). Não há limitações quanto ao número de *downlinks* numa única transição.

4.1.1 Exemplo de Modelagem com Redes de Referência

É ilustrado a seguir um exemplo simples de modelagem com redes de referência, abordando o uso das inscrições descritas na Tabela 4.1. Consideremos as classes **A** e **B**, apresentadas na Figura 4.1. Instâncias da classe **A** podem criar várias instâncias da classe **B** através da transição **t1**, cuja ocorrência executa a inscrição **b:new B**. Todas as instâncias criadas serão armazenadas no *lugar de saída* da transição **t1**. Sempre que uma instância de **B** é criada, a transição **t2** — que contém a inscrição de saída síncrona **b:set(p)** — em **A** pode sincronizar com a transição **t3** — que contém a inscrição de entrada síncrona — em **B**. A ocorrência sincronizada das duas transições funciona como se uma instância de **A** chamasse o método **set** de uma instância de **B**.

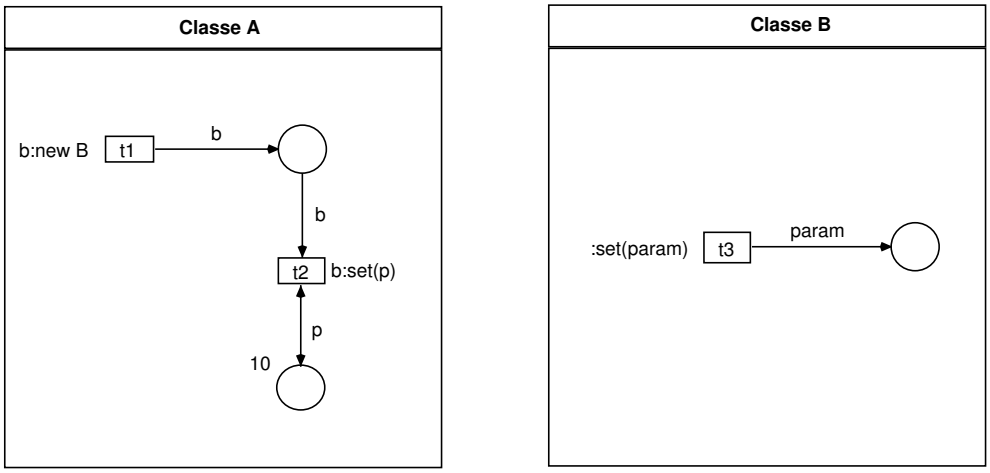

Figura 4.1: Um modelo de redes de referência

Na Figura 4.2 mostramos o efeito da simulação de um sistema de objetos cuja configuração inicial (denotada por um retângulo) consiste de uma única instância da classe **A** (representado por um círculo). As transições em destaque indicam sua habilitação. As infor-

mações ao lado dos lugares indicam sua marcação. Os rótulo dos arcos entre as configurações indicam a transição cujo disparo levou à mudança de configuração.

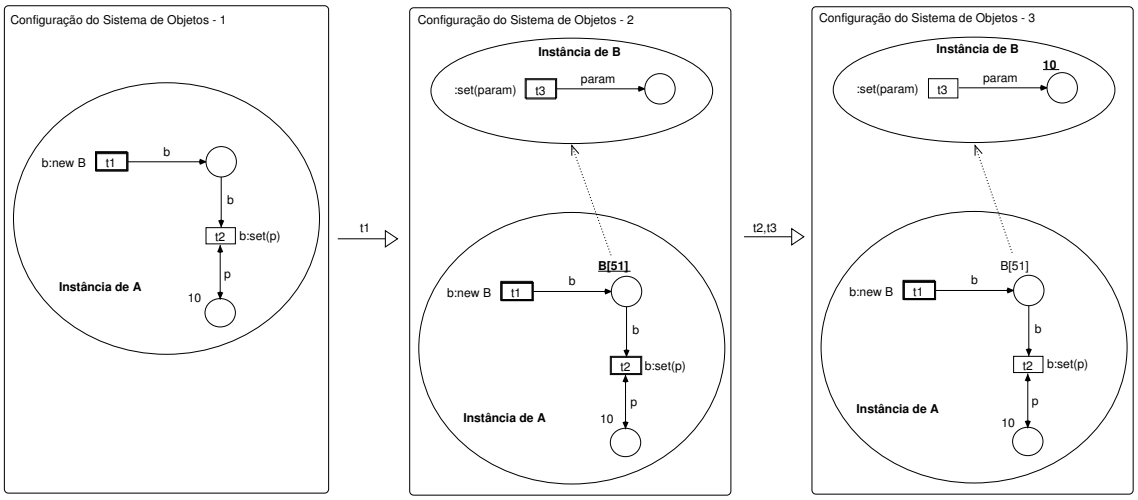

Figura 4.2: Simulação de um sistema de objetos

Inicialmente, apenas a transição **t1** está habilitada. Seu disparo executa a inscrição **b:new B** e nos leva à **Configuração do Sistema de Objetos - 2**, onde há uma referência (identificada por **B[51]**) a uma instância da classe **B** armazenada no lugar de saída de **t1**. Nesta configuração ambas as transições **t2** e **t3** estão habilitadas, e podem executar em sincronia suas respectivas inscrições, **b:set(p)** e **:set(param)**, respectivamente. As duas inscrições só poderão ser executadas de modo síncrono. Em RPOO, o conjunto destas duas inscrições é denominado *evento*. O disparo síncrono das transições **t2** e **t3** leva o sistema para o estado descrito na **Configuração do Sistema de Objetos - 3**. Note que o valor **10** foi passado da instância de **A** para a instância de **B**. Este comportamento está de acordo com a semântica das ações de saída síncrona RPOO.

Além dessas inscrições, que lidam especificamente com objetos cujo comportamento é descrito por uma rede de Petri, podemos escrever junto às transições expressões sobre objetos Java provenientes dos *lugares de entrada* das transições. Qualquer expressão Java é permitida para este tipo de inscrição. Expressões Java também são utilizadas como inscrições nos arcos.

Quando são usadas, numa determinada transição, expressões Java envolvendo chamada de métodos, tais métodos são invocados independentemente da ocorrência da transição, pois o simulador precisa de seus resultados para verificar a habilitação da transição. Isto pode

levar a alguns efeitos indesejados, caso o resultado da chamada de um determinado método seja avaliada em valores diferentes na verificação de habilitação e na ocorrência da transição em si. Para contornar este tipo de problema, há nas redes de referência um tipo especial de inscrição, chamado de *action*. Inscrições *action* indicam expressões Java que são chamadas *apenas* quando a transição na qual estão inscritas dispara. Faremos menção a este tipo de inscrição através do termo em inglês *action*, para evitar confusão com o termo *ação*, que utilizamos para referenciar ações RPOOt. As inscrições das redes de referência com respeito à instanciação de objetos, *downlinks* e *uplinks* serão consideradas (e referenciadas como) ações RPOO, uma vez que são equivalentes às ações RPOO de instanciação, saída síncrona de dados e entrada de dados.

4.2 Extensão Proposta

Uma vez identificados pontos comuns entre as redes de referência e RPOO, nossa proposta é a de estendermos Renew para produzir e simular modelos RPOOt, implementando apenas aspectos de nosso formalismo que não são contemplados de alguma forma pela ferramenta e adequando sintaxe e operadores de acordo com suas restrições (apesar de apresentarem uma semântica equivalente, os operadores usados em Renew não são os mesmos usados em RPOO).

Nos modelos de nossa extensão, diferentemente do que acontece nos modelos formais de RPOO(t), temos dois tipos diferentes de ações de entrada de dados: um para entrada de dados em modo síncrono, outro para entrada de dados em modo assíncrono. Com isso, nossa extensão não se torna aplicável à modelagem de sistemas nos quais seja imprescindível a modelagem de uma funcionalidade onde dados podem ser recebidos, pela mesma entrada, em modos síncrono *e* assíncrono. Como isto não é um caso muito comum — não foi identificado em nenhuma experiência anterior com o formalismo RPOO —, consideramos que tal restrição não acarretará maiores problemas para os desenvolvedores RPOO.

A implementação completa das ações RPOO e RPOOt levaria mais tempo do que o disponível para o desenvolvimento deste trabalho. Nosso protótipo de ferramenta — que chamaremos Renew/RPOOt — apresentará apenas as ações de troca assíncrona de dados (temporizada e não-temporizada), além das ações envolvendo instanciação e troca de men-

Método	**Descrição**
obj**->**method(params)	ação RPOO de saída assíncrona de dados
obj**?**method(params)	ação RPOO de entrada de dados (utilizada apenas para mensagens enviadas em modo assíncrono)
obj**->**method(params)@delay	ação RPOOt de saída temporizada de dados

Tabela 4.2: Inscrições das redes de referência

sagens em modo síncrono, que já estavam implementadas na ferramenta que optamos por estender. Consideramos que tais ações são suficientes para a produção de um estudo de caso convincente e deixamos a implementação das outras ações para eventuais trabalhos futuros.

A Tabela 4.2 descreve informalmente a sintaxe e os símbolos utilizados para as ações que serão implementadas em Renew/RPOOt.

Por questões de implementação, como veremos pouco mais adiante, escolhemos o operador '->' para denotar ações de saída de dados em modo assíncrono, substituindo o operador '**.**' que utilizamos para as definições formais no Capítulo 2. Nas ações de entrada, permitiremos a omissão da referência ao objeto que enviou a mensagem. Assim, poderemos escrever ações de entrada na forma '**?**method(params)'.

4.3 Implementação

A idéia central é implementar a troca assíncrona de mensagens RPOO e, em seguida, fazer com que estas mensagens sejam *temporizadas*, conforme especificado em RPOOt. Teríamos, desta forma, um mecanismo de temporização *entre* objetos, além da temporização *interna*, que já é oferecida pela ferramenta. Apesar da idéia parecer bastante simples, sua implementação requer mudanças em diversos módulos, desde alterações em nível de interface com o usuário até a definição de um compilador que interprete apropriadamente as inscrições anotadas para os novos tipos de mensagem.

Para entendermos as alterações necessárias em seus respectivos níveis, convém explicarmos em linhas gerais o processo de produção, compilação e simulação de modelos da ferramenta Renew.

4.3.1 Funcionamento Geral

As redes de Petri que descrevem as classes são desenhadas através de um módulo gráfico que estende o JHotDraw [24] — biblioteca Java desenvolvida para o suporte à produção de interfaces gráficas. As redes podem ser desenhadas independente de formalismos particulares. Aos seus lugares e transições podemos atribuir um nome e diversas inscrições, que serão interpretadas de acordo com a escolha do formalismo que deve ser usado para compilar e simular as redes. O formalismo pode ser escolhido através do menu principal da aplicação, que é ilustrada na Figura 4.3.

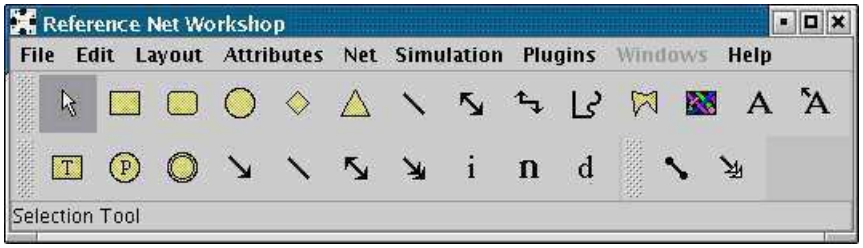

Figura 4.3: Interface gráfica da ferramenta

As redes desenhadas através da interface gráfica são convertidas para as chamadas *shadow nets* — representação que abstrai todas as informações gráficas das redes, preservando apenas as informações relevantes para o processo de compilação (lugares, transições, arcos, inscrições, etc). Cada classe/rede do modelo terá uma *shadow net* correspondente, e o conjunto de todas as *shadow nets* é inserido numa coleção, instância da classe **ShadowNetSystem**. Tal coleção será passada para um compilador, que deve estender a classe **AbstractNetCompiler**.

A função do compilador é receber uma **ShadowNetSystem** e converter todas as *shadow nets* subjacentes para o que chamamos de *redes semânticas* — redes que provêem objetos necessários ao processo de simulação. O compilador trata todos os elementos das redes — lugares, transições, arcos, inscrições e declarações — lendo as informações textuais das *shadow nets*, interpretando-as e gerando, para cada uma delas, objetos que serão usados no processo de simulação — tais objetos devem implementar interfaces bem definidas. As inscrições de interação entre objetos, por exemplo, são representadas nas redes semânticas por implementações da interface **TransitionInscription**.

O simulador executa instâncias das redes semânticas geradas pelo compilador. Cada

instância das redes semânticas terá, por sua vez, instâncias de classes ou interfaces (como **TransitionInscription**) que representem seus vários elementos no contexto da simulação (um lugar, no contexto de uma simulação, deve conter internamente um multi-conjunto, o que não é necessário em outros contextos). A partir das instâncias de **TransitionInscription** (que são parte das instâncias das redes semânticas), o simulador pode verificar a habilitação de uma transição e eventualmente dispará-la. Para isso, uma **TransitionInscription** deve criar uma instância de **TransitionOcurrence**, que, por sua vez, instancia implementações das interfaces **Binder** — responsável por calcular a ligação (*binding*) das variáveis envolvidas numa determinada inscrição — e **Executable** — responsável por *executar* uma inscrição, avaliando suas expressões, de acordo com os valores atribuídos (ligados) às variáveis envolvidas. São os objetos **Executable** os responsáveis por tarefas como remoção/adição de fichas (no caso de **Executables** gerados a partir de inscrições em arcos) e remoção/adição de mensagens pendentes (no caso de **Executables** gerados a partir de inscrições RPOOt).

Desta forma, para cada nova inscrição definida, devemos prover implementações de quatro interfaces (**TransitionInscription**, **TransitionOcurrence**, **Binder** e **Executable**) que, devidamente inter-relacionadas, viabilizam o processo de compilação/simulação. Além disso, precisamos definir um compilador em face da sintaxe e semântica das novas inscrições, reaproveitando o processo de compilação das inscrições que desejamos preservar.

4.3.2 Extensões Implementadas

Praticamente toda a interface gráfica da ferramenta foi reaproveitada. Uma pequena extensão foi necessária, no entanto, para tornar possível a visualização de mensagens pendentes entre objetos, já que o formalismo originalmente suportado pela ferramenta não apresentava troca de mensagens assíncronas. Outra modificação de ordem menos expressiva foi a inclusão de um menu que permite ao usuário escolher (através do menu da aplicação) o compilador do formalismo RPOOt para ser utilizado na compilação das redes (diversos compiladores estão disponíveis na ferramenta, além do compilador RPOOt).

Para que pudéssemos aproveitar ao máximo os mecanismos de cálculo de ligações (para variáveis envolvidas em inscrições) oferecidos pela ferramenta, distribuímos as mensagens pendentes entre os objetos destinatários das mensagens. Em RPOOt, todas as mensagens residem num multi-conjunto presente na *estrutura* do sistema de objetos, que também ar-

mazena informações acerca dos objetos vivos e de suas inter-ligações. Renew não apresenta nenhum mecanismo central que se assemelhe às *estruturas* dos sistemas de objetos RPOOt: todas as informações para o cálculo de ligações estão distribuídas entre os objetos. Assim, adotamos a estratégia de distribuir tais mensagens em multi-conjuntos graficamente ocultos, que são acrescentados a cada objeto. A soma de todos esses multi-conjuntos é rigorosamente igual ao multi-conjunto de mensagens da estrutura de sistema de objetos RPOOt. Com esta estratégia, foi possível calcular a ligação das variáveis das inscrições de entrada de dados RPOO de uma maneira muito parecida com a qual se calcula ligações de arcos de entrada de transições. Para que isso tome efeito, entretanto, as redes precisam ser *pré-compiladas* para o acréscimo dos multi-conjuntos ocultos nas classes RPOOt que apresentam inscrições de entrada de dados. Desta forma, reutilizamos, para esta sorte de inscrição, as implementações de **TransitionInscription**, **TransitionOcurrence**, **Binder** e **Executable**, utilizadas para os arcos de entrada — em termos de implementação, nesta ferramenta em particular, arcos são tratados como inscrições de transições.

No caso das inscrições de saída assíncrona temporizada, foi preciso prover nossas próprias implementações para as interfaces supracitadas. Contudo, utilizamos as mesmas implementações pra representar ambas as ações de saída assíncrona temporizada e não-temporizada. Quando a informação temporal das mensagens assíncronas forem omitidas, a execução das ações levará em conta que elas levam *zero* unidades de tempo, o que torna seus efeitos equivalentes aos efeitos de uma ação de saída assíncrona RPOO.

Para a implementação do compilador do formalismo (implementado em *JavaCC*), responsável por gerar as redes semânticas, passíveis de simulação, foi definida uma gramática para o formalismo. Por questões técnicas, em termos de implementação e reaproveitamento de código, não utilizamos, para denotar algumas ações, os mesmos símbolos utilizados quando da definição formal de RPOOt. Particularmente, o símbolo "**.**", usado para denotar ações de saída assíncrona de dados nas definições formais, já é empregado com outra semântica no formalismo suportado pela ferramenta. Por esta razão, escolhemos o operador "**->**" para denotar saída de dados em modo assíncrono, preservando a aplicação do operador "**.**" à chamada de métodos de classes e objetos Java utilizados para descrever os dados do sistema modelado. O operador de saída de dados em modo síncrono (**!**) não foi utilizado devido ao fato de existir, nas redes de referência, uma inscrição equivalente que faz uso do símbolo

"**:**". Em teoria, nosso compilador seria uma extensão do *Timed Java Compiler*, usado para redes de referência com redes de Petri temporizadas (TCPN's). Como a tecnologia *JavaCC*, utilizada para escrever os compiladores da ferramenta, não oferece mecanismos de extensão, o compilador RPOOt foi escrito com base no *Timed Java Compiler*, adicionando-se o código necessário ao tratamento das novas inscrições.

As extensões comentadas anteriormente são suficientes para que os simuladores disponíveis na versão 2.0.1 de Renew executem corretamente os modelos RPOOt, mas eles (os simuladores) apresentam um pequeno problema quando estamos tratando de simulações em *background*.

Estamos lidando com *modelos de simulação* e, para contornar algumas de suas limitações (conforme discutido nos Capítulos 1 e 3), precisamos realizar *muitas* simulações (contemplando cenários variados), cada uma delas com uma grande quantidade de dados de entrada, o que geralmente faz com que cada simulação em particular se estenda por várias horas. É por esta razão que temos por meta efetuar diversas simulações em paralelo, sem interface gráfica, em *background*, através de grades computacionais [15].

Alguns simuladores providos por Renew não detectam o fim das simulações (quando nenhuma transição pode disparar), e o único que detecta (**SequentialSimulator**) não finaliza sua execução (fica à espera de interações com o usuário). Para a execução em *grades*, isto equivale a uma tarefa que nunca finaliza sua execução e portanto, seu resultados nunca retornam para a máquina que a distribuiu. Desta forma, tivemos que estender o simulador **SequentialSimulator** de forma que o término das simulações fosse detectado *e* a execução do simulador, em si, fosse finalizada.

4.3.3 Validação da Extensão

Apesar do fato da simulação dos modelos de redes de referência indicarem uma relação entre a instanciação de objetos e canais síncronos com as ações RPOO de instanciação e saída síncrona, não há nenhum trabalho que comprove formalmente a relação aparente entre os formalismos, considerando estes aspectos. Por esta razão, realizamos um trabalho de *validação* da extensão da ferramenta.

Para este processo, em face das dificuldades de se construir uma prova formal de equivalência, adotamos a idéia de analisar resultados da simulação de modelos no Renew/RPOOt

em face de grafos de ocorrência (para os mesmos modelos) gerados a partir do JMobile [12], ferramenta desenvolvida especificamente para modelos RPOO. Foram produzidos diagramas de seqüência de mensagens RPOO a partir de diversas simulações. A presença de *todas* essas seqüências de mensagens no grafo de ocorrência (não temporizado) dos modelos usados no processo de simulação nos indica que a semântica implementada na extensão do Renew para RPOOt está de acordo com a semântica RPOO implementada pelo JMobile.

A modelagem levou em consideração apenas dois tipos de inscrições Renew — *downlink* e *uplink* que seriam equivalentes às inscrições de saída síncrona de dados e entrada de dados — e as inscrições adicionadas na extensão que implementamos, para saída de dados em modo assíncrono e entrada de dados. Ações de instanciação não foram consideradas pois não fizeram parte da extensão a implementação de ações RPOOt de criação não instantânea.

O processo de validação das implementações realizadas neste trabalho é descrito em maiores detalhes no Anexo A.

4.4 Classes de Suporte

As extensões descritas anteriormente proporcionam a *simulação* de modelos temporizados, entretanto, para que possamos efetivamente tomar proveito das simulações, foi necessária uma estratégia de coleta de dados em tempo de simulação. Outra deficiência seria a dificuldade em utilizar-se de distribuições estatísticas para modelar alguns atrasos específicos (tarefa razoavelmente comum na modelagem de protocolos de rede). Estes fatores foram endereçados através de classes Java incluídas ao pacote do formalismo, podendo ser utilizadas em todos os modelos RPOOt.

Módulo texto Foi desenvolvido um pequeno módulo Java, chamado *Logger*, capaz de gerar *logs*, observar variáveis e produzir relatórios de simulação. A estratégia do Logger é a de coletar os dados durante o disparo de transições/ocorrência de eventos, gerar arquivos e prover algumas medidas acerca das variáveis observadas (média, valor máximo/mínimo, desvio padrão, etc) durante as simulações. O módulo não oferece nenhum recurso em nível de interface com o usuário, pois foi desenvolvido especificamente para ser utilizado em simulações em *background*, nas quais não há interação com o usuário durante o processo de

simulação. O propósito deste módulo não é o de gerar gráficos ilustrando medidas de desempenho mas sim o de gerar informações textuais — na forma de arquivos — que possam ser utilizadas em relatórios de desempenho ou na geração de gráficos — após o processo de simulação — por ferramentas apropriadas para tal propósito.

A utilização de Loggers deve se dar forçosamente através de inscrições *action* junto às transições da rede (assim, uma determinada informação só será coletada quando uma determinada transição disparar). Para se valer das funcionalidades deste módulo de acompanhamento de simulações, o desenvolvedor deve ler a documentação da classe Logger e chamar os métodos apropriados em inscrições *action*, no seu modelo. Na Tabela 4.3, destacamos alguns métodos da classe Logger e suas respectivas funcionalidades. Exemplos da utilização de Loggers serão explorados no estudo de caso apresentado no Capítulo 5.

Módulo gráfico Foi desenvolvido, também, um módulo de coleta de dados com a possibilidade de produção de gráficos durante o processo de simulação (ver Figura 4.4). Trata-se de uma extensão do JFreeChart [17], que permite a visualização gráfica dos dados coletados. A visualização pode acontecer durante ou após a simulação dos modelos. Os dados coletados também podem ser armazenados em arquivos de texto, possibilitando a construção de gráficos para a análise de desempenho através de ferramentas externas (como *gnuplot*). Em seu estado atual, o módulo gráfico suporta apenas a coleta de dados e produção de gráficos em duas dimensões.

Figura 4.4: Módulo gráfico de coleta de dados

Os usuários do simulador têm a liberdade de desenvolver seus próprios módulos de acompanhamento de simulações, coleta de dados e geração de relatórios. Quaisquer classes e objetos Java podem ser utilizados em inscrições *action* com este propósito sem prejudicar o funcionamento do processo de simulação, contando que tais classes e objetos sejam utilizados tão somente para isso, não causando *efeitos colaterais* aos objetos usados no problema

Método	**Descrição**
void countUpdates(String dc)	registra a associação entre o tempo corrente do modelo e o número de atualizações da coleção de dados representada por *dc*
void log(String dc, double x, double value)	registra a associação entre *x* e *y* na coleção de dados representada por *dc*
void log(String dc, double[] params)	registra todos os valores de *params[]*, convertidos para forma textual, na coleção de dados representada por *dc*
void log(String dc, double value)	a associação entre o tempo corrente do modelo e o valor *value* na coleção de dados representada por *dc*
void logAdd(String dc, double value)	registra, na coleção de dados identificada por *dc*, a associação entre o tempo corrente do modelo e a soma entre *value* e o último valor registrado em *dc*
void logSub(String dc, double value)	registra, na coleção de dados identificada por *dc*, a associação entre o tempo corrente do modelo e a subtração entre *value* e o último valor registrado em *dc*
void logText(String text)	registra o text contido em *text* num arquivo independente de qualquer coleção de dados
long updates(String dc)	obtém o número de atualizações de uma determinada coleção de dados
double average(String dc)	obtém média, por unidade de tempo, dos valores registrados numa determinada coleção de dados
double lastValue(String dc)	obtém o último valor registrado para uma determinada coleção de dados

Tabela 4.3: Alguns métodos da classe Logger

modelado. Em outras palavras, o desenvolvedor deve ter em mente que a *observação* da simulação não deve efetuar nenhuma alteração nos dados do sistema modelado.

Distribuições estatísticas Ainda no que concerne bibliotecas de suporte, outro ponto importante é a possibilidade de simulação de distribuições estatísticas conhecidas tais que Poisson, Normal, Exponencial, etc. Não raro, no âmbito dos protocolos de redes, as distribuições estatísticas são bastante úteis na modelagem de atrasos e tráfegos de rede específicos. Para a simulação de modelos RPOOt, disponibilizamos as distribuições mais conhecidas, implementadas como extensões do pacote estatístico *DSOL* [20].

Capítulo 5

Estudo de Caso: Modelagem do Protocolo IP Móvel com Renew/RPOOt

5.1 Contexto

Na última década, as tecnologias de comunicação sem fio apresentaram um avanço rápido. A variedade crescente de dispositivos móveis que dispõem de conexões à Internet, como PDA's e celulares, mudaram de certa forma a percepção que temos acerca da Internet: surgiu um novo paradigma de computação — conhecido como computação móvel (*mobile computing*). Segundo MacCann e Roman [31], a computação móvel representa um ponto maior de partida do paradigma tradicional de computação distribuída, já que os componentes de hardware e software podem dinamicamente migrar por entre localizações geograficamente distribuídas. Mateus e Loureiro[30] descrevem a computação móvel como "o pior cenário de um sistema distribuído, onde são freqüentes os problemas de comunicação e mudanças topológicas".

Pesquisa e desenvolvimento no âmbito da computação móvel tiveram foco em diferentes aspectos, como projeto de hardware, gerência de dados e protocolos para redes sem fio. Bergner et al [3] afirmam que as pesquisas correntes consideram apenas áreas isoladas, apresentando deficiências no tocante uma visão mais integrada da computação móvel. A inexistência de conceitos definidos com precisão e a falta de formalismos como fundamento para a evolução das linguagens, estratégias de desenvolvimento e ferramentas são algumas das razões para a falta de uma metodologia abrangente e integrada para o desenvolvimento de sistemas móveis.

Muitos pesquisadores investigaram o uso de métodos formais no desenvolvimento de sistemas móveis. Wang et al [46] usam CPN's para modelar um protocolo de tunelamento (L2TP) em IP VPN — análise de grafo de ocorrência foi a técnica aplicada para verificar o comportamento do protocolo. Vinh [45] faz uso de RSL (RAISE Specification Language) para especificar formalmente migrações em um processo. A especificação é verificada através do verificador de modelos XSpin.

Também no que diz respeito à modelagem e simulação do IP Móvel, muitos trabalhos de valor foram desenvolvidos desde o advento do protocolo, nos anos noventas. Temos análise de desempenho através de modelos analíticos (especificados sob rigorosas restrições matemáticas) e de modelos de simulação (com resultados menos precisos), além de trabalhos com o objetivo de verificar o comportamento do protocolo independente de informações temporais.

Dang e Kemmerer [8] usam ASTRAL (linguagem de especificação formal) para especificar o IP Móvel. O verificador de modelo da linguagem é utilizado na análise do protocolo. Alguns autores [5] propõem-se a analisar extensões específicas do protocolo, que tratam de reduzir a perda de datagramas durante as migrações. Em outros trabalhos [9; 36], analisamos o comportamento do IP Móvel através de técnicas de simulação e análise de espaço de estados, sem no entanto derivar considerações acerca de desempenho.

5.2 IP Móvel — Visão Geral

O roteamento de datagramas IP através da Internet não prevê mobilidade entre seus diversos nodos, o que levou, com o advento das tecnologias móveis, a situações não contempladas pelos algoritmos de roteamento vigentes. Surge então o problema que o protocolo IP Móvel [34] propõe-se a resolver:

Um dispositivo móvel (ou *mobile node)*, por exemplo, um telefone celular, pode estar conectado à Internet por meio de um enlace de rádio-freqüência, através de um roteador, utilizando um endereço IP do domínio deste. Em virtude de suas características inerentes de mobilidade, a potência do sinal emitido pelo dispositivo móvel pode, num determinado momento, não ser mais adequada/suficiente para alcançar seu ponto de acesso à rede (o roteador em questão). O celular precisaria, então, re-conectar-se à Internet através de algum

outro roteador (que possa alcançá-lo devidamente), recebendo um novo endereço IP (no domínio do novo roteador).

Uma vez que os *hosts* com conexões abertas com o dispositivo móvel (*correspondent nodes*) o endereçam por seu endereço IP original, quando este muda de rede, as conexões TCP são rompidas e as comunicações UDP ficam sem resposta, pois os datagramas serão encaminhados até seu roteador original (também chamado de roteador nativo — *home agent*) que não pode mais transmitir dados ao dispositivo (nem tem idéia de onde ele se encontra).

A proposta do protocolo é a de manter a conectividade dos dispositivos móveis de modo que as migrações sejam transparentes às aplicações em rede, mantendo as conexões TCP e as comunicações UDP funcionando como se o dispositivo móvel estivesse em sua rede nativa.

A solução especificada no IP Móvel é a de fazer com que o celular passe a ter dois endereços IP. Um deles, denominado *home address*, permanecerá o mesmo ainda que o dispositivo móvel seja atendido por uma célula conectada a outra rede física. O outro, conhecido por *care-of-address* (COA), mudará toda vez que o dispositivo mudar de rede física.

Neste contexto, denomina-se *home network* a rede nativa do dispositivo móvel, onde ele se autentica e recebe seu *home address*; e *foreign network* qualquer outra rede para onde o dispositivo migra, e é atendido por um roteador diferente do roteador da *home network*.

A comunicação entre dispositivo móvel e a Internet seria, então, mediada por dois roteadores, o *home agent* (roteador da rede nativa) e o *foreign agent* (roteador da rede estrangeira). Admitindo um dispositivo móvel numa rede estrangeira e algum *host* na Internet enviando-lhe datagramas através do seu endereço IP fixo, o home agent, conhecendo a rede estrangeira onde se encontra o celular no momento, faria o redirecionamento de todos os datagramas recebidos da Internet para o foreign agent, que enfim os transmitiria ao seu destino.

Para que se dê todo este processo, o celular deve registrar seu COA junto ao Home Agent sempre que mudar de rede. Como a razão pela qual o celular muda de rede pode ser justamente o fato dele não mais poder alcançar seu Home Agent, este registro normalmente será intermediado pelo Foreign Agent.

Desta maneira, podemos dizer que o protocolo IP móvel compreende basicamente três mecanismos, quais sejam: a descoberta do novo endereço IP na rede estrangeira, registro do novo endereço (junto ao Home Agent) e tunelamento dos datagramas para o novo endereço.

Descoberta de um novo endereço O processo de descoberta do novo endereço é baseado no protocolo *Router Advertisement* (especificado na RFC 1256). *Home agent* e *foreign agent* enviam, em *broadcast*, em intervalos regulares (normalmente a cada segundo), mensagens (*agent advertisements*) que contêm um ou mais *care-of-address* (além de outras informações peculiares ao protocolo). Por se tratar de um protocolo específico, independente do IP Móvel, o processo de descoberta de um novo endereço não será modelado em detalhes.

Registro de um novo endereço Uma vez de posse de um novo endereço numa rede estrangeira, o celular deve registrá-lo junto a seu *home agent* para que os datagramas oriundos da Internet possam ser devidamente reencaminhados (tunelados) ao seu novo endereço. Para isso, o celular deve enviar, através do *foreign agent*, uma mensagem para o *home agent* indicando seu novo endereço. O *home agent* recebe a mensagem, valida o pedido, atualiza suas informações de roteamento e envia uma confirmação para o celular. Como não poderia deixar de ser, o registro de um novo endereço envolve questões de segurança. Tais questões não são objeto da modelagem em nosso estudo de caso.

Tunelamento O tunelamento funciona basicamente da seguinte maneira: o *home agent* encapsula os datagramas destinados ao celular (com seu IP original) dentro de um outro datagrama IP, cujo endereço de destino será o COA do celular. Tais datagramas são, então, encaminhados para a Internet: todo este processo cria um efeito de tunelamento.

O túnel pode ser construído do *home agent* até o *foreign agent* ou do *home agent* até o celular. Uma vez que o *foreign agent* pode identificar o celular pelo seu *home address*, todos os celulares "visitantes" poderiam ter por COA o próprio endereço IP do *foreign agent*. Assim, o túnel seria entre o *home agent* e o *foreign agent*.

Os túneis estabelecidos nos *home agents* serão válidos por um intervalo de tempo específico. Desta forma, o dispositivo móvel não precisa desfazer seu registro de COA (ele irá expirar, invariavelmente), mas precisa reenviar um pedido de registro para se manter conectado.

Temporização e desempenho do protocolo Há uma série de fatores que podem ser analisados no que diz respeito ao desempenho do IP Móvel. O protocolo apresenta vários parâmetros de temporização (como expiração de túneis, reenvio de registros), e o ajuste adequado

desses parâmetros deve levar a um desempenho otimizado. Um modo de achar valores ideais para esses parâmetros é através da análise dos resultados da simulação de diversos cenários.

O maior problema de desempenho para o protocolo acontece durante as migrações dos dispositivos móveis. No caso de um dispositivo deixar sua rede nativa, todos os datagramas endereçados a ele seriam perdidos *durante* o tempo em que o dispositivo móvel está obtendo um COA na rede estrangeira e solicitando seu registro junto à rede nativa. Para contornar este problema o *home agent* pode *reter* os datagramas por algum tempo, caso o dispositivo móvel não possa ser encontrado. Se o dispositivo solicitar registro de COA antes que este tempo expire, os datagramas retidos serão tunelados normalmente.

Outro problema de desempenho pode ser identificado quando um dispositivo móvel migra entre redes estrangeiras. Até que o registro do dispositivo seja atualizado junto a seu *home agent*, os datagramas serão tunelados para um *foreign agent* que não pode (mais) alcançá-lo. Neste caso, o dispositivo pode, antes de atualizar seu registro, enviar seu novo COA ao seu último *foreign agent* (através de mensagens conhecidas como *binding updates*). Este, por sua vez, deve redirecionar os datagramas tunelados (endereçados ao dispositivo) para seu *foreign agent* atual. Esta mecanismo é conhecido por *smooth handoff*.

5.3 Modelagem

O modelo de classes que produzimos para o IP Móvel é composto por cinco entidades principais: **MobileNode**, **Agent**, **Medium**, **CorrespondentNode** e **Network**. **Agent** é uma classe abstrata (interface), com duas sub-classes concretas: **HomeAgent** e **ForeigAgent**. **Agent**'s e **MobileNode**'s implementam uma interface comum (**Node**), através da qual instâncias de **Network** enviarão mensagens. O diagrama de classes do modelo é ilustrado na Figura 5.1[1].

Conforme podemos observar no diagrama de classes, um **MobileNode** relaciona-se com dois Agent's, que podem ser instâncias de **HomeAgent** ou **ForeignAgent**. Apenas um **Agent** é capaz de trocar mensagens com um **MobileNode**, mas estes guardarão, também, a identificação do último **Agent** que os serviu depois de completarem um processo de migração. Isto é necessário à modelagem do mecanismo conhecido como *smooth handoff*. Apesar da

[1]Note que, para manter aderência à terminologia utilizada na RFC do protocolo, todos os nomes de entidades e métodos são descritos em inglês.

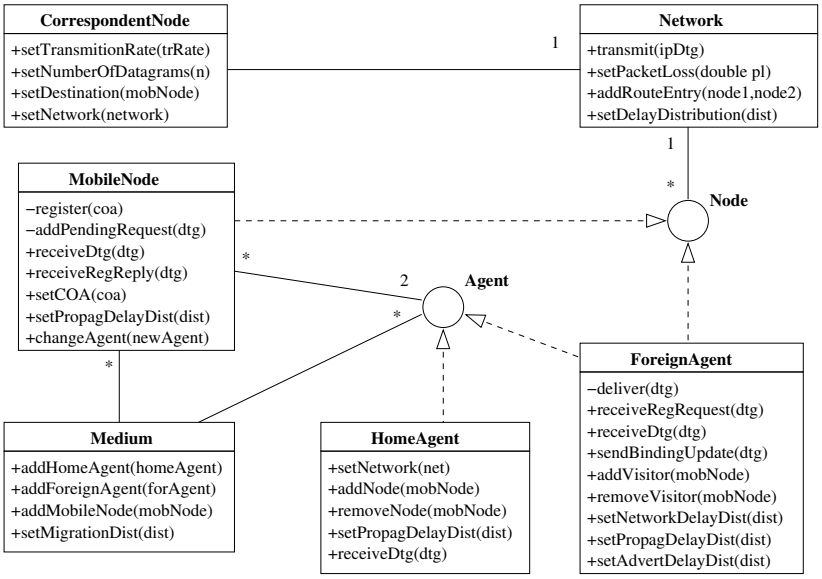

Figura 5.1: Diagrama de classes para o protocolo IPM

relação direta entre **MobileNode**'s e **Agent**'s no modelo, na prática, tais conexões se dão através de um *meio* de comunicação. A classe **Medium** (que é um *singleton*) desempenha este papel, referenciando todos os **Agent**'s e **MobileNode**'s do sistema e oferecendo meios de modelar as migrações. As outras classes são modelos simplificados de um conjunto de roteadores (**Network**) e de nodos *estáticos* (**CorrespondentNode**'s) que enviam datagramas aos **MobileNodes** do sistema.

A seguir, descrevemos brevemente algumas classes do modelo. Dividimos algumas redes em *pastas* organizadas de acordo com as funcionalidades da classe. As pastas não possuem nenhuma semântica com influência no comportamento dos modelos: são elementos gráficos usados tão somente para facilitar a compreensão das redes. Outro recurso do qual nos valemos com este mesmo intento é a utilização de *lugares virtuais*. Um lugar denotado por uma elipse delimitada por duas linhas corresponde a um lugar virtual: lugar que, para efeitos de simulação, é rigorosamente igual ao lugar de mesmo nome circundado por apenas uma linha[2]. As transições cuja funcionalidade explicaremos têm sua inscrições RPOOt escritas em negrito. Finalmente, as inscrições *action* usadas para coleta de dados estarão graficamente separadas das transições das redes apenas para denotar que tais inscrições não influem no comportamento dos modelos. O nome da transição cujo disparo executa as inscrições em destaque estará escrito acima delas, em negrito, conforme veremos mais adiante.

[2]Nas CPN's temos um conceito parecido, denominado lugar de fusão

CorrespondentNode A rede **CorrespondentNode**, ilustrada na Figura 5.2, é a mais simples das redes do modelo. A classe possui quatro propriedades, distribuídas entre os lugares da rede e configuráveis através de métodos **set**. Nas redes do modelo, transições com métodos de nomes padronizados na orientação a objetos (set, add, remove...) não terão nome. Elas farão o que os nomes dos métodos sugerem. No *nó de declaração* (no caso desta rede, disposto no lado superior-esquerdo da figura) constam apenas os datagramas utilizados pela rede, escritos exatamente com a mesma sintaxe de Java. O datagrama **br.gmf.mip** contém classes Java específicas do modelo do protocolo (como IpDatagram) enquanto o datagrama **de.renew.formalism.rpoot** dispõe de classes para auxiliar simulações em geral (como DistributionFactory).

O lugar **Network** armazenará uma referência a um objeto (do tipo Network), que representa o conjunto de roteadores que separa o CorrespondentNode do HomeAgent que representa a rede nativa do destinatário dos datagramas (o destinatário é armazenado no lugar **Destination**). Utilizaremos as referências dos objetos como sendo seus respectivos endereços de rede. Independente de onde se encontram os destinatários das mensagens, os datagramas enviados por um CorrespondentNode serão sempre roteados para o HomeAgent dos respectivos destinatários.

No lugar **Dist**, teremos um objeto do tipo Distribution, criado pela inscrição **DistributionFactory.createTrafficDist(kbps)**. Tal objeto simulará os atrasos de envios de datagramas considerando um tráfego baseado em rajadas com média de **kbps** quilobits por segundo.

A transmissão de datagramas é efetivada pelo disparo da transição **transmit**. A transição cria e configura um datagrama através das *ações internas* **ipDtg=new IpDatagram()** e **action ipDtg.configureData(this,mn,n)**, enviando o datagrama à rede através da ação de saída assíncrona temporizada **net->transmit(ipDtg)@delay**.

HomeAgent A rede/classe RPOOt HomeAgent é ilustrada na Figura 5.3. Na pasta **Setters and adders** temos alguns métodos auxiliares, que configuram o estado de um **HomeAgent**. Algumas transições não estão rotuladas pois os nomes dos métodos nelas inscritos são autoexplicativos. Em algumas inscrições de entrada (denotadas pelo símbolo **?**), omitiremos o objeto que enviou a mensagem para simplificar a notação.

No lugar **Network** teremos uma instância da classe Network. É para este objeto que

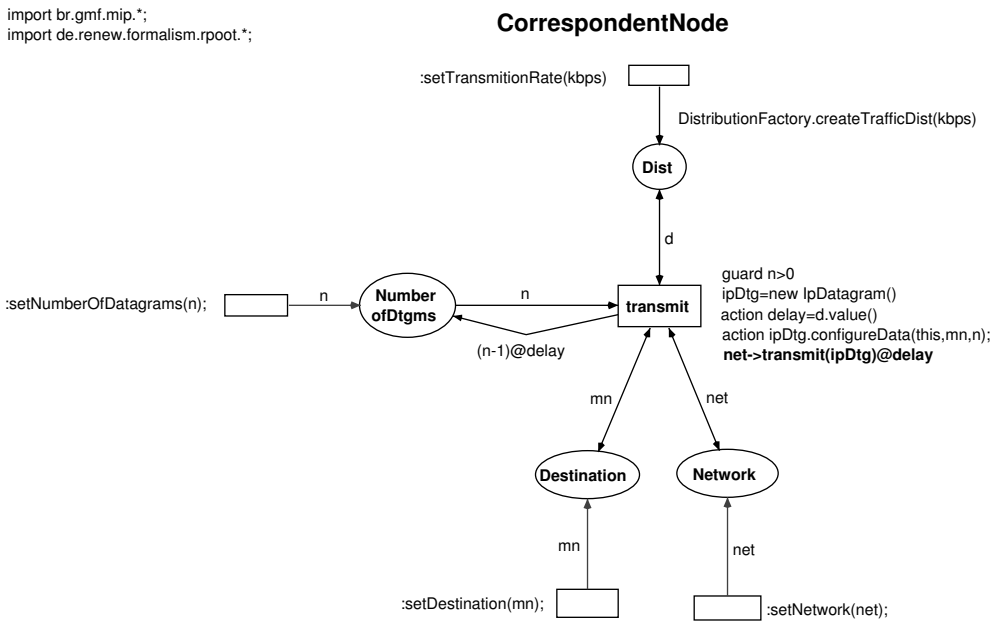

Figura 5.2: A rede CorrespondentNode

o **HomeAgent** redirecionará os datagramas endereçados a MobileNode's que migraram da rede nativa (ou seja, os que não constam no lugar **MobNodes**). A ação **:removeNode(mn)** modela a migração sob a perspectiva do HomeAgent. Sempre que um nodo deixa um HomeAgent, um túnel inválido é inicializado através da tupla **[mn,null,-1.0]** apenas para indicar que o nodo não está mais na rede nativa. Os dois últimos parâmetros do túnel representam o COA para o qual os datagramas devem ser redirecionados e o tempo no qual foi criado o túnel — que só será devidamente configurado mediante o recebimento de *registration requests*.

O lugar **PropagDelayDist** armazenará um objeto que capaz de gerar atrasos que representem o tempo de propagação de datagramas — classes que modelam distribuições estão disponíveis nos datagramas de suporte. Estes atrasos serão utilizados nas interações temporizadas com objetos do MobileNode. O lugar **NetDelayDist** tem esta mesma função, sendo utilizado nas interações temporizadas com objetos do tipo Network.

O processo de registro de COA é ilustrado na pasta **Registration**. Podemos observar que a transição **register** recebe da rede uma requisição de registro (*registration request*) através da ação de entrada **net?receiveDtg(request)**. Um datagrama de resposta à requisição é configurado a partir de ações internas (**reply=new IpDatagram**(), **modelTime=SearchQueue.getTime**() e **action reply.configureRegReply(request)**) e enviado à

rede através da inscrição de saída temporizada **net->transmit(reply)@d.value()**. Observemos, também, a inscrição de coleta de dados **action Logger.logAdd(this+"-RegTraffic",2)**. Apesar de separada graficamente, esta inscrição faz parte do conjunto de inscrições da transição **register**. A inscrição em questão registra o tráfego de dois datagramas de registro (uma requisição e uma resposta).

Temos neste, ponto, uma temporização interna à classe. Para cada túnel recém-criado em **Tunnels** adicionamos uma ficha temporizada em **Expired Tunnels** que será utilizada quando o túnel expirar, para remover as informações adicionadas a **Tunnels**.

Na pasta **Tunneling and delivering** estão descritas as funcionalidades com respeito a tunelamento e entrega de datagramas. A transição **tunnel expired** remove túneis cujo tempo de validade tenha expirado. Já a transição **deliver** modela a atividade de entrega de datagramas para MobileNode's que estão na rede nativa. A transição simplesmente recebe um datagrama (**?receiveDtg(ipDtg)**) e o envia ao seu destinatário com um *delay* de propagação (**mn->receiveDtg(ipDtg)@d.value()**), desde que o datagrama esteja endereçado a um MobileNobe presente no lugar **MobNodes**. A transição **tunnel** recebe um datagrama (**?receiveDtg(ipDtg)**) e o encapsula num novo datagrama, de acordo com as informações do túnel (em **Tunnels**), repassando o datagrama tunelado para a rede, como podemos ver na ação **net->transmit(new IpDatagram(this,coa,ipDtg))@d.value()**.

Na pasta **Buffering**, enfim, temos a implementação de um mecanismo de armazenamento de datagramas cujo destinatário não se encontre na rede nativa e não tenha um túnel com um COA devidamente configurado. Datagramas com essas características são armazenados em **Buffer** durante **HA.bufferingDelay()** unidades de tempo através da transição **buffer datagram**. O tempo de armazenamento é configurável, e o tamanho do *buffer* é observado durante as simulações (ver inscrições *action* em **Collecting Data**). Os datagramas armazenados podem ser tunelados mediante o cadastramento de um novo túnel, ou descartados, caso tenham passado em **Buffer** mais tempo do que o HomeAgent está configurado para mantê-los. Nossa intenção com a modelagem deste mecanismo é a de observar o quanto ganhamos de desempenho com o armazenamento de datagramas e qual a capacidade do *buffer*, de acordo com o tempo durante o qual os datagramas podem ficar armazenados. Para isso, observamos o tráfego de datagramas desperdiçados (**action Logger.logTraffic(this+"-Discarded",1)**).

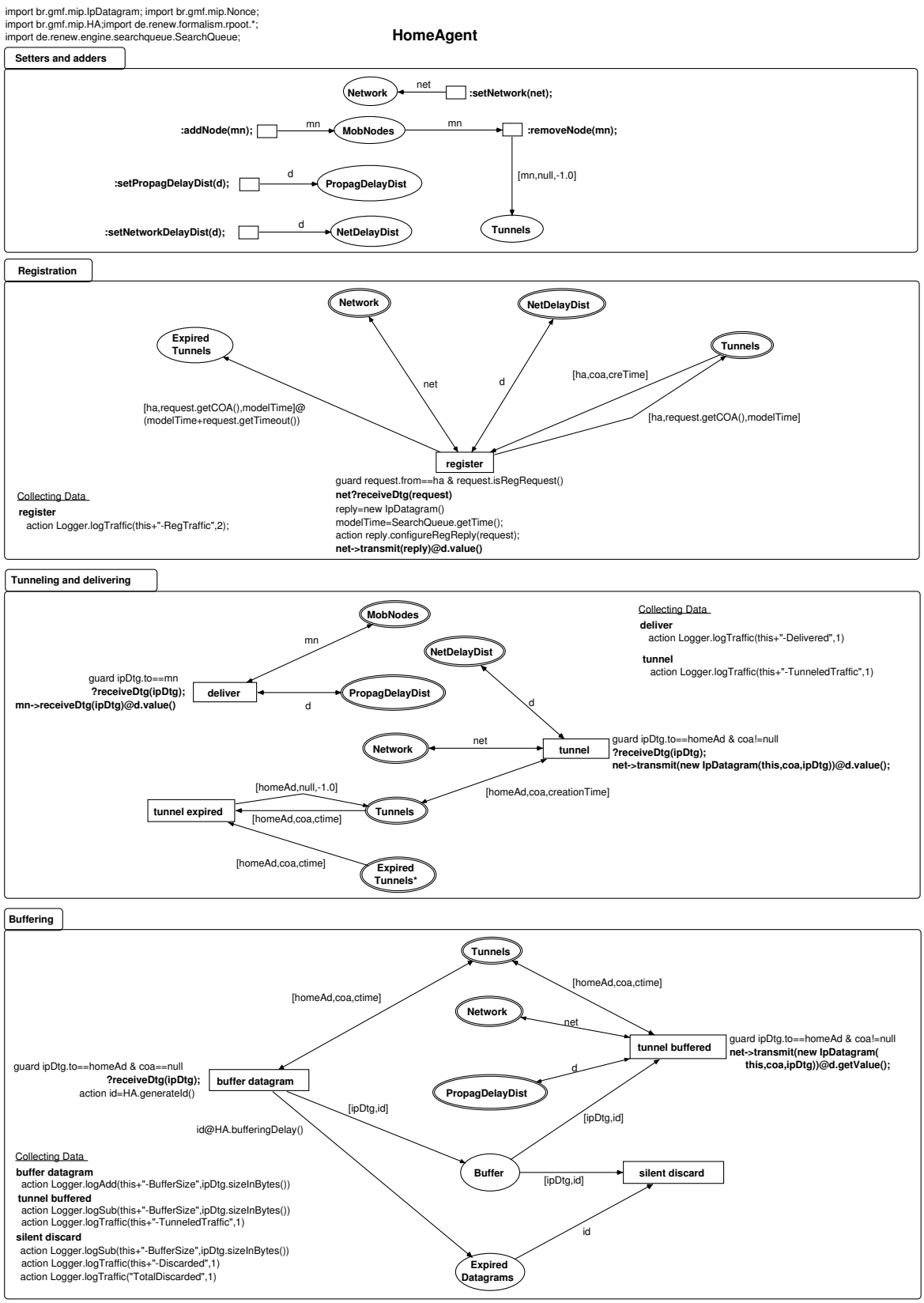

Figura 5.3: A rede HomeAgent

Foreign Agent Um ForeignAgent, ilustrado na Figura 5.4, tem como uma de suas principais funções o desencapsulamento (e subseqüente entrega) de datagramas endereçados a MobileNode's que visitem sua rede. Esta funcionalidade é modelada através das transições **detunnel** e **deliver**, presentes na pasta **Routing**. Observe que há uma ação de sincronização na transição **detunnel**. Com a exigência de sincronização, o *evento* RPOOt executado atomicamente consistirá das ações **?receiveDtg(ipDtg)**, **this:deliver(ipDtg.detunnel())**, **:deliver(dtg)** e **(dtg.to)->receiveDtg(dtg)@d.value()**. Em termos de simulação, as duas transições em foco *dispararão ao mesmo tempo*.

Caso chegue um datagrama endereçado a um MobileNode que não está mais na rede delimitada pelo ForeignAgent (o lugar **Visitors** armazena uma lista dos MobileNode's *visitantes*), o datagrama pode ser descartado (transição **discard**), caso o MobileNode não tenha notificado o ForeignAgent de sua nova localidade (tal informação será armazenada no lugar **Handoff**). Para cada datagrama descartado, coletamos informação a respeito dos dados desperdiçados pelo ForeignAgent (**action Logger.logTraffic(this+"-Discarded",1)**) e globalmente (**action Logger.logTraffic("TotalDiscarded",1)**).

Na pasta **Smooth handoff**, modelamos o mecanismo homônimo, tendo em vista medirmos um possível ganho de desempenho caso o ForeignAgent ofereça a possibilidade de redirecionar datagramas mediante as devidas notificações.

No intervalo de tempo durante o qual um MobileNode migra, obtém um novo COA e o registra, muitos datagramas podem ter sido tunelados (pelo HomeAgent) para o antigo COA do MobileNode. Para que tais datagramas não sejam simplesmente descartados, o dispositivo móvel pode enviar um *binding update* para o seu último ForeignAgent, para que este redirecione os datagramas para a rede atual do MobileNode. A recepção deste tipo de *binding update* é modelada pela transição **create handoff entry**, que recebe da rede a requisição (**?receiveDtg(ipDtg)**) e cria uma entrada no lugar **Handoff** descrevendo para qual COA os datagramas devem ser redirecionados (**[homeAddress,coa,id]**). Esta entrada pode ser utilizada pela transição **forward** para efetivar o redirecionamento, mediante a chegada de algum datagrama.

Na pasta **Setters and adders** temos alguns métodos de configuração. Os mais importantes deles são os que estão inscritos nas transições **remove visitor** e **add visitor**, que modelam migrações sob a perspectiva de um ForeignAgent. Sempre que um

MobileNode chega à rede delimitada por um ForeignAgent, este executará a ação **mn->setCOA(this)@ad.value()+d.value()** indicando seu novo COA (no caso, o COA utilizado será o endereço/referência do próprio ForeignAgent).

MobileNode A rede MobileNode é ilustrada na Figura 5.5. Na pasta **Migration**, temos modelada a migração no que diz respeito a um MobileNode. Essencialmente, ocorrem apenas trocas de referências, o novo agente (**newAg**) sendo armazenado no lugar **Agent** e o antigo agente (**currentAg**) sendo armazenado em **PreviousAgent**.

Uma vez ocorrida a migração, o novo agente do MobileNode lhe concederá um COA, através do envio de uma mensagem assíncrona temporizada. A recepção desta mensagem é modelada na transição **new COA**, na pasta **Registration and binding update**. Caso o último agente do MobileNode tenha sido um ForeignAgent (e não seu HomeAgent), esta transição deve sincronizar com a transição **update binding and register COA** (através da ação de saída síncrona **this:register(newCoa)**). Esta última transição, por sua vez, envia um **binding update** endereçado ao seu último agente (para que este possibilite o mecanismo de *smooth handoff*) e um pedido de registro (*binding request*) endereçado ao seu HomeAgent. Além disso, a requisição é adicionada como pendente (**this:addPendingRequest(regRequest)**) para ser reenviada depois de algum tempo. O reenvio de requisições pendentes é modelado na pasta **Registration maintenance** através da transição **resend reg req**. Independentemente da chegada de uma resposta positiva à requisição pendente (pasta **Network data input**), esta deverá sempre ser reenviada pois os registros efetuados junto ao HomeAgent expiram.

Medium Por fim, temos a rede Medium (Figura 5.6), que é responsável unicamente por promover migrações dos MobileNode's por entre os ForeignAgent's de tempos em tempos. Como podemos ver nas inscrições da transição **migrate from HA**, o processo de migração consiste da sincronização de diversas ações. Atomicamente, o dispositivo móvel envolvido na migração é removido de seu HomeAgent (**homeAg:removeNode(mn)**), adicionado a um ForeignAgent (**forAg:addVisitor(mn)**) e muda seu agente para o ForeignAgent ao qual está sendo adicionado (**mn:changeAgent(forAg)**). Observe que um MobileNode ficará em comunicação com um dado agente por uma quantidade de tempo dada pelo objeto armazenado

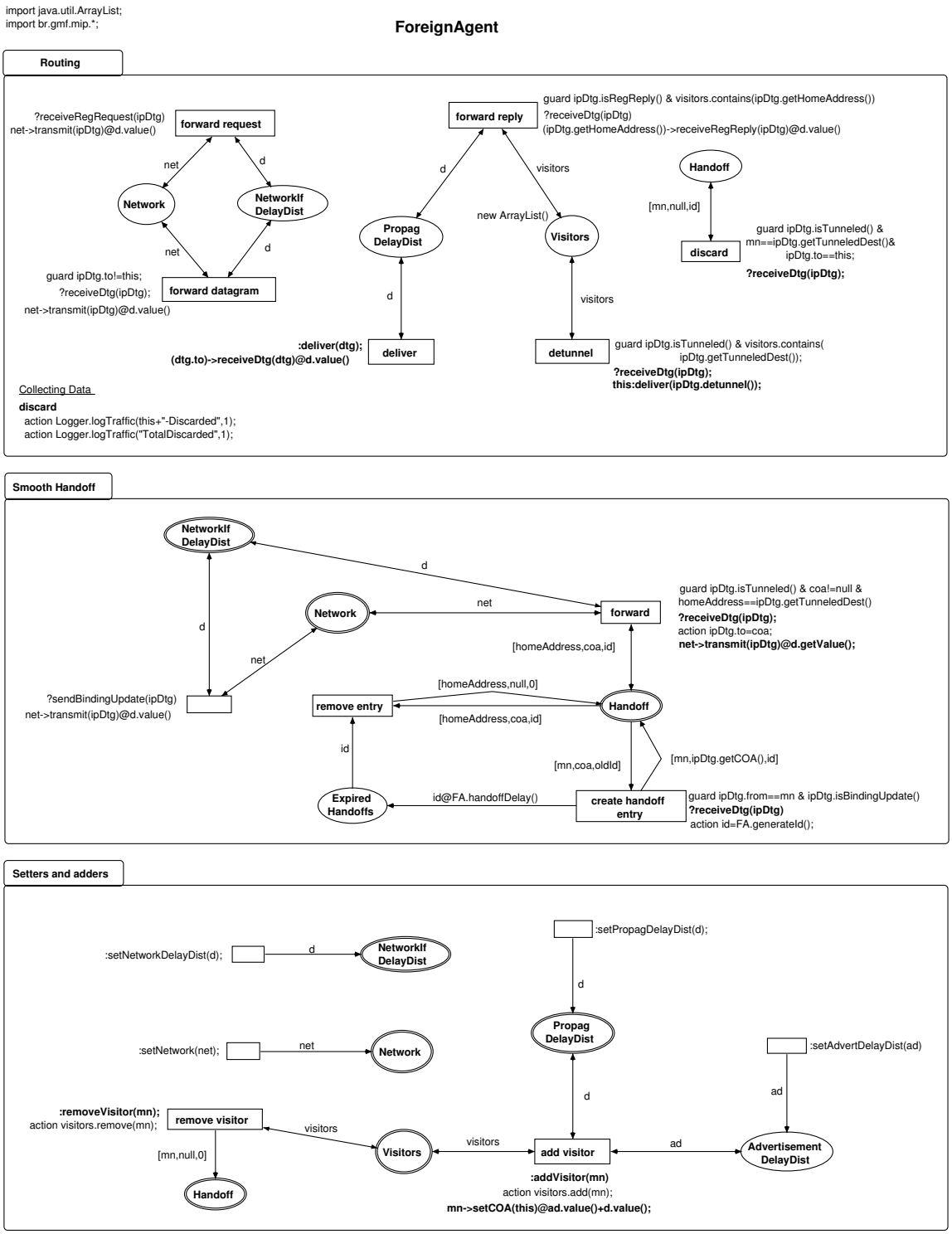

Figura 5.4: A rede ForeignAgent

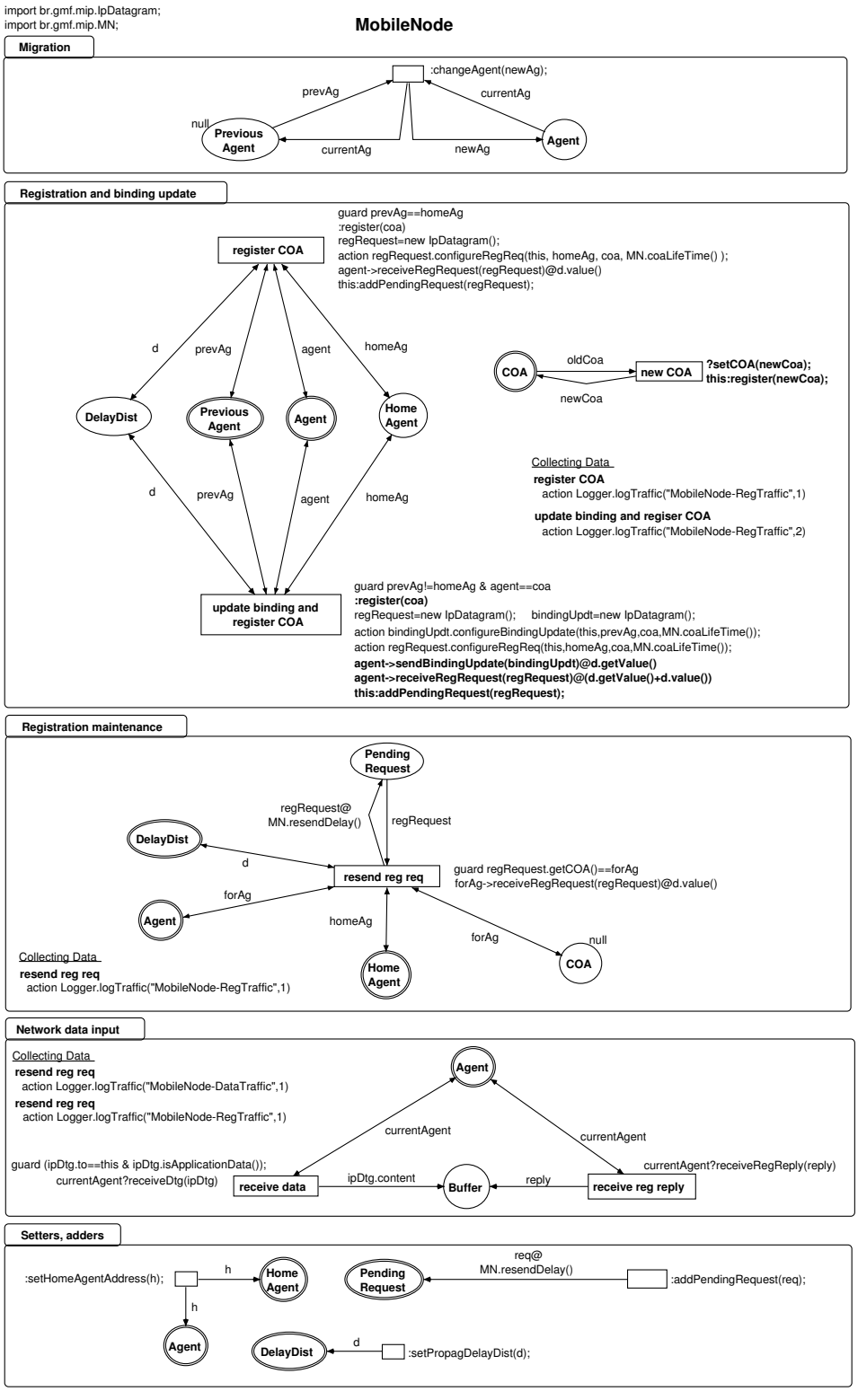

Figura 5.5: A rede MobileNode

no lugar **Dist**. Para este modelo de Medium, cada MobileNode migra para todos os ForeignAgents presentes no lugar **ForeignAgent**.

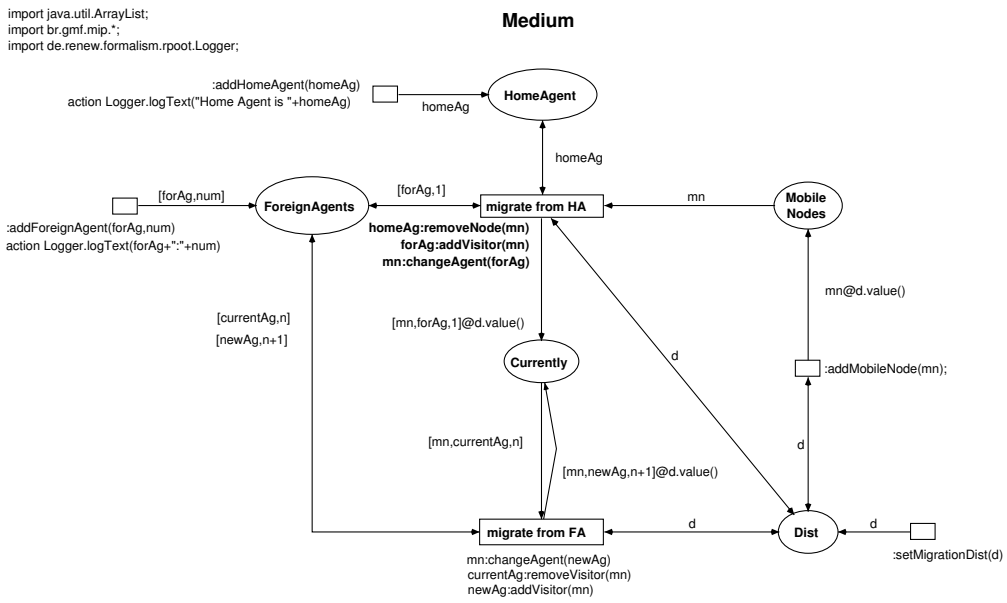

Figura 5.6: A rede Medium

5.4 Análise de Desempenho

Nosso propósito, nesta seção, não é o de prover medidas exatas e definitivas acerca do desempenho do IP Móvel, mas sim o de mostrar que a ferramenta Renew/RPOOt pode ser utilizada para derivar informações que não poderíamos obter facilmente com modelos RPOO puros[3].

De acordo com o que foi discutido ao fim da Seção 5.2, as simulações tiveram por objetivo apresentar informações sobre a queda de desempenho durante as migrações dos dispositivos móveis, identificando parâmetros adequados à configuração do sistema. Mais especificamente, observamos a quantidade de dados desperdiçados (em quilobits/s) em decorrência de migrações e ajustamos a configuração dos agentes de forma que essa perda fosse atenuada.

Uma vez que foi necessária uma quantidade razoável de simulações para alcançar este objetivo, executamos diversas instâncias do simulador Renew/RPOOt em grades computacionais. Tivemos um total de dezoito configurações diferentes, cada uma sendo simulada

[3]Em trabalhos publicados anteriormente [9; 36], identificamos as dificuldades de modelarmos ações temporizadas (ao estilo das ações ilustradas na seção anterior) em modelos RPOO convencionais.

duas vezes, o que nos dá um total de 36 simulações. O tempo de simulação de cada configuração variou especialmente de acordo com o número de dispositivos móveis e com a quantidade de dados a eles enviada. A simulação dos cenários mais simples levaram, em média, pouco mais de uma hora, ao passo que a simulação dos cenários mais elaborados chegou a ultrapassar as dezoito horas. Se todas as simulações tivessem sido realizadas uma por uma, seqüencialmente, todo este processo teria levado em torno de seis dias. Com o uso de grades computacionais, o tempo total foi o tempo da simulação mais demorada (cerca de dezoito horas).

5.4.1 Armazenamento de Datagramas no *Home Agent*

Quando um datagrama chega a um *home agent*, endereçado a um dispositivo móvel que não está conectado à rede nativa (e não completou o processo de registro), o agente pode simplesmente descartar o datagrama ou armazená-lo em algum *buffer* por algum tempo. Neste último caso, o datagrama seria tunelado caso o registro do dispositivo móvel fosse aceito antes que este tempo expirasse. É este mecanismo que analisamos neste momento através dos resultados de nossas simulações.

Para tal propósito, observamos a simulação de vários cenários (várias *configurações iniciais*), todos eles com um *home agent* e um *foreign agent* interligados por um enlace de 155Mbps. Em cada simulação, variamos, essencialmente, o tempo máximo durante o qual o *home agent* pode armazenar um datagrama, a freqüência de migrações e a velocidade pela qual os dados são enviados pelos *correspondent nodes* para os dispositivos móveis. Com isso verificando em que situações e em que proporções o armazenamento de datagramas leva a algum ganho em termos de desempenho. A seguir, sumarizamos alguns resultados de simulação de acordo com os cenários considerados.

Configurações sem Armazenamento de Datagramas no *Home Agent*

Observamos que a perda de datagramas não é muito expressiva para configurações onde as migrações (50, no total) ocorrem de maneira distribuída, sem grandes quantidades de migrações em curtos intervalos de tempo (consideramos 100ms um curto intervalo de tempo para dezenas de migrações). A Figura 5.7 ilustra a distribuição das migrações, apresentando

a quantidade de dispositivos móveis que deixaram sua rede nativa em intervalos de **1s**. Podemos ver que as migrações estão bem distribuídas em cerca de 25s (entre os tempos de modelo **5000** e **30000**), chegando, no máximo, a **11** migrações por segundo.

Com as migrações se dando desta forma e os dispositivos estáticos enviando dados em rajada, com média de 4Kbps, o *home agent* dos dispositivos móveis chegou a desperdiçar uma banda de pouco mais de **8Kpbs**, como podemos ver na Figura 5.8. Podemos observar, também, que o desperdício de dados ocorre em pequenos intervalos de tempo, o que nos indica uma pequena quantidade de datagramas descartados.

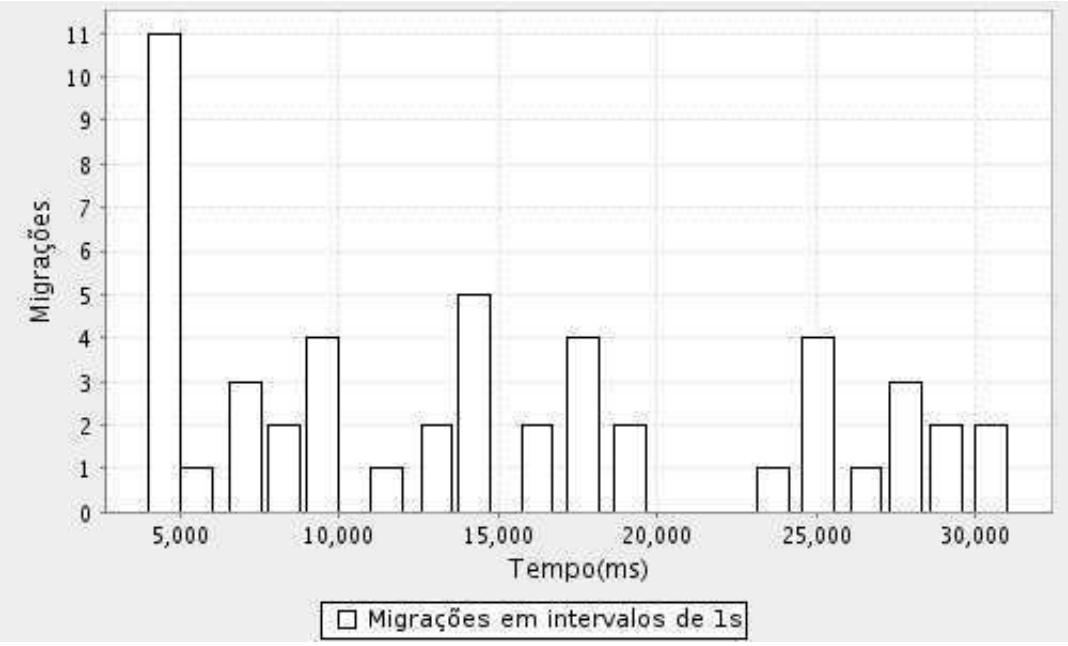

Figura 5.7: Migrações envolvendo 50 dispositivos móveis

Considerando migrações mais concentradas, distribuídas num intervalo de 4s, envolvendo 100 dispositivos móveis, já teremos uma queda de desempenho mais acentuada. A banda desperdiçada neste cenário é ilustrada (em função do tempo de modelo) na Figura 5.9. Podemos perceber um desperdício máximo de pouco menos de **30Kbps**, o que já é bem superior ao que tivemos no cenário anterior. A proporção entre a quantidade de dados desperdiçados e a quantidade total de dados que passaram pelo *home agent* já é considerável, como podemos perceber no gráfico ilustrado na Figura 5.10

Concentrando ainda mais o número de migrações e aumentando a taxa de transmissão dos *correspondent nodes*, podemos identificar com mais precisão uma queda de desempenho

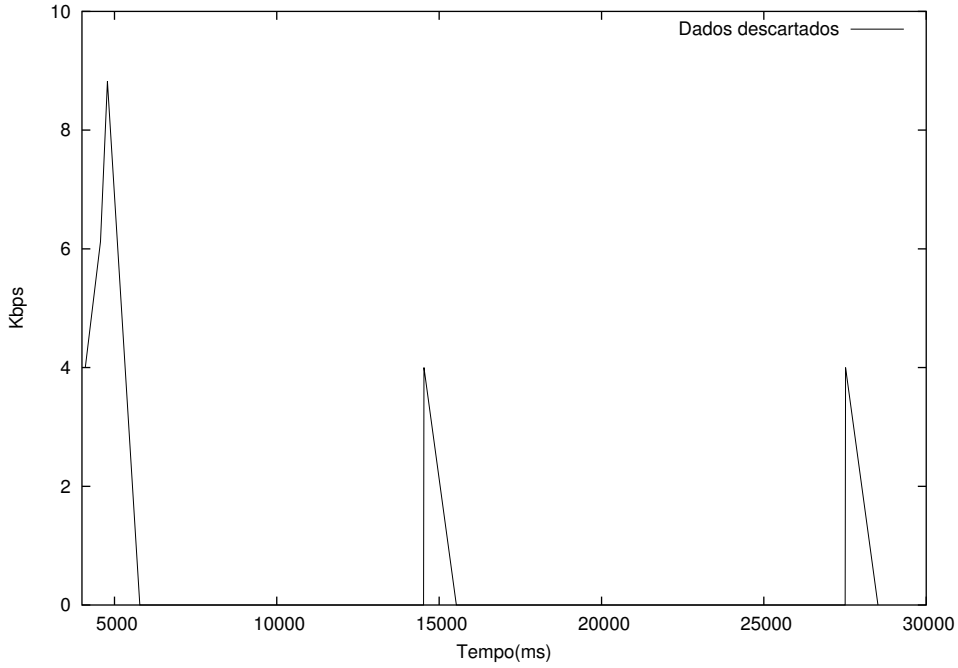

Figura 5.8: Banda desperdiçada

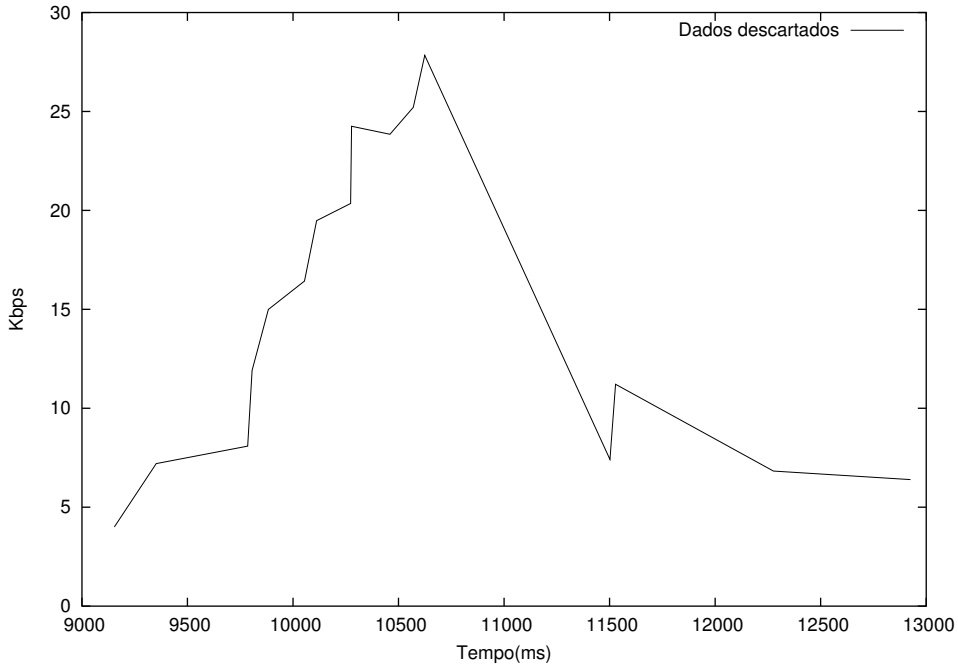

Figura 5.9: Banda desperdiçada

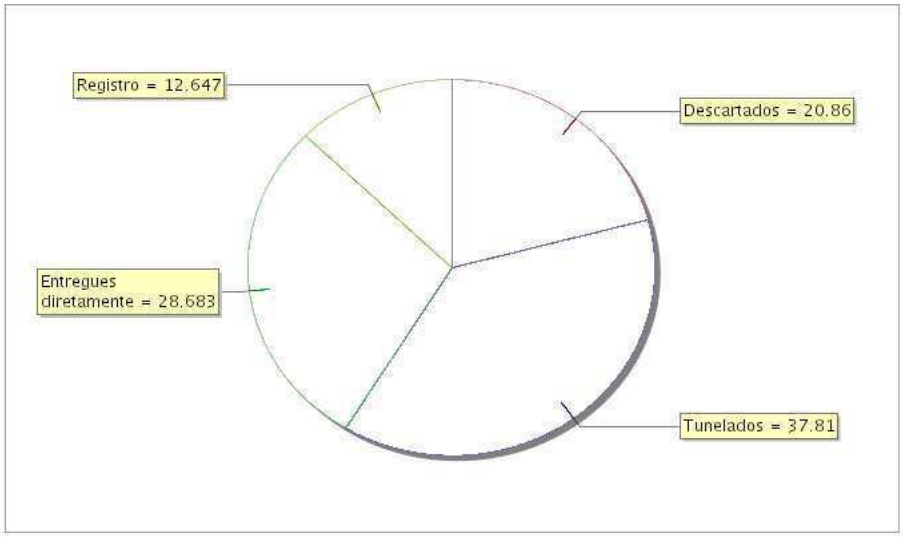

Figura 5.10: Proporção de dados que passaram pelo *home agent*

considerável para um sistema sem armazenamento de datagramas no *home agent*.

A Figura 5.11 ilustra a distribuição de 150 migrações de dispositivos móveis em intervalos de **0.1s**, obtida a partir da simulação de um cenário com taxa de transmissão de 20Kbps por parte dos *correspondent nodes*. A banda desperdiçada neste cenário é ilustrada na Figura 5.12, e chega a atingir a marca de 140Kbps em determinado ponto da simulação.

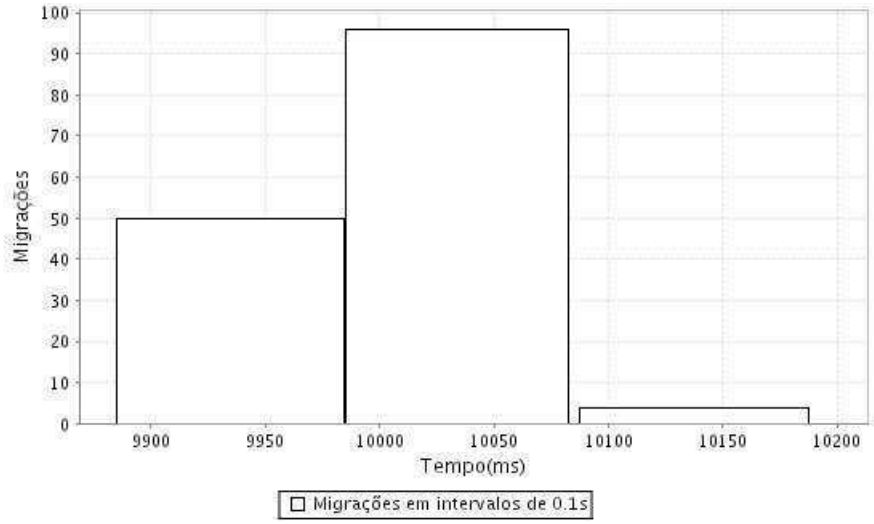

Figura 5.11: Migrações envolvendo 150 dispositivos móveis

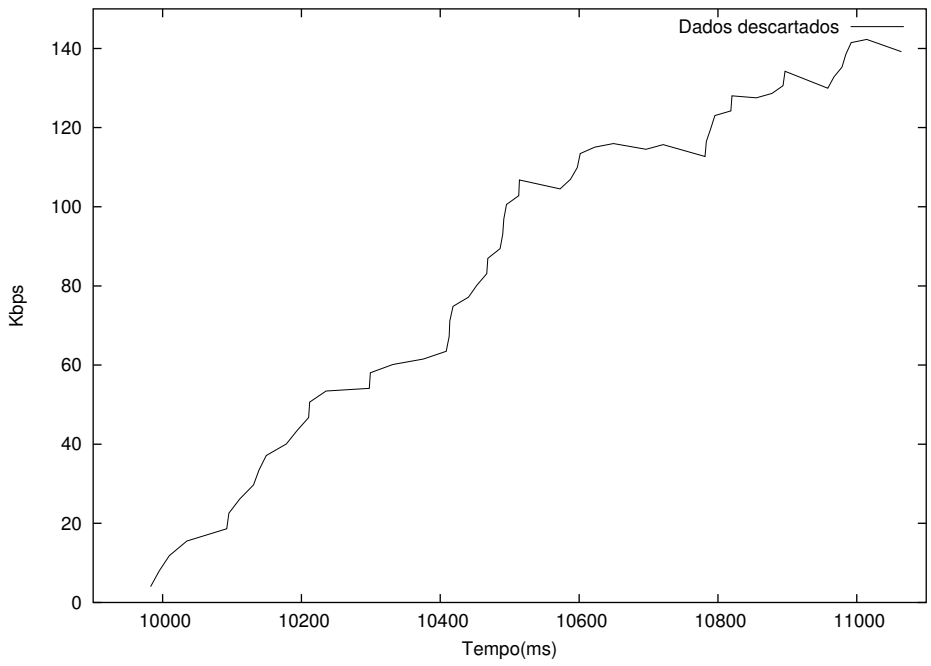

Figura 5.12: Banda desperdiçada

Configurações com Armazenamento de Datagramas no *Home Agent*

Nosso propósito, agora, é o de analisar as simulações de cenários com armazenamento de datagramas em face da queda de desempenho derivada a partir das simulações sem armazenamento de datagramas. Essencialmente, tomaremos o último cenário da subseção anterior, variando o tempo de armazenamento e verificando eventuais ganhos de desempenho, além do tamanho do *buffer* utilizado (que deve variar de acordo com o tempo de armazenamento).

A Figura 5.13 ilustra a banda desperdiçada no *home agent* quando para este foram configurados os tempos de armazenamento de **50**, **100** e **200ms**. Podemos perceber que tais tempos de armazenamento não são adequados para atenuarmos significativamente a perda de datagramas no *home agent*, uma vez que a banda desperdiçada é bem parecida com a que foi apresentada na Figura 5.12 para uma configuração sem armazenamento. Isto ocorre, provavelmente, porque boa parte dos dispositivos móveis não conseguem completar seu processo de registro antes que seus datagramas armazenados expirem.

Contudo, se considerarmos tempos de armazenamento de 500ms e 1s, diminuiremos a perda de datagramas de forma bem mais expressiva. A Figura 5.14 ilustra a banda desperdiçada nestes dois casos, que chega a se aproximar de 90 e 25kbps, respectivamente. A

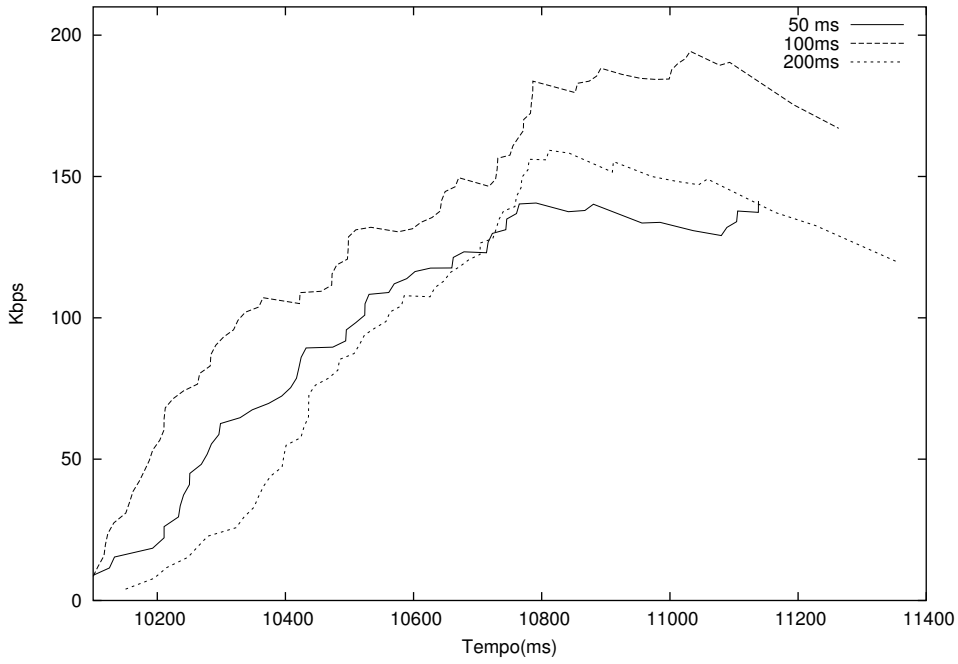

Figura 5.13: Banda desperdiçada para diferentes tempos de armazenamento

proporção dos dados que passam pelo *home agent* durante o intervalo de tempo no qual ocorreram as migrações é ilustrada na Figura 5.15. O *buffer* de armazenamento do *home agent* apresentou um tamanho máximo de 22KB para um tempo de armazenamento igual a 1s.

5.4.2 *Smooth Handoff*

Quando um datagrama chega a um *foreign agent*, endereçado a um dispositivo móvel que já migrou para outra rede estrangeira, o agente descarta o datagrama por não saber como roteá-lo. Smooth handoff é o mecanismo que define o envio de *binding updates* aos *foreign agents* com o objetivo de minimizar a perda de datagramas nestas situações. Desta forma, ao migrar entre redes estrangeiras, um dispositivo móvel envia um *binding update* ao *foreign agent* de sua última rede, indicando-o seu novo COA. Com isso, os datagramas roteados para o agente errado podem ser redirecionados para o COA atual do dispositivo em vez de descartados.

Observamos a simulação de vários cenários (várias *configurações iniciais*), com e sem *smooth handoff*, no intuito de verificarmos em que situações o mecanismo é realmente necessário e qual o ganho de desempenho decorrente de seu uso. Todos os cenários simulados são compostos por um *home agent* e dois *foreign agent*, todos interligados por um

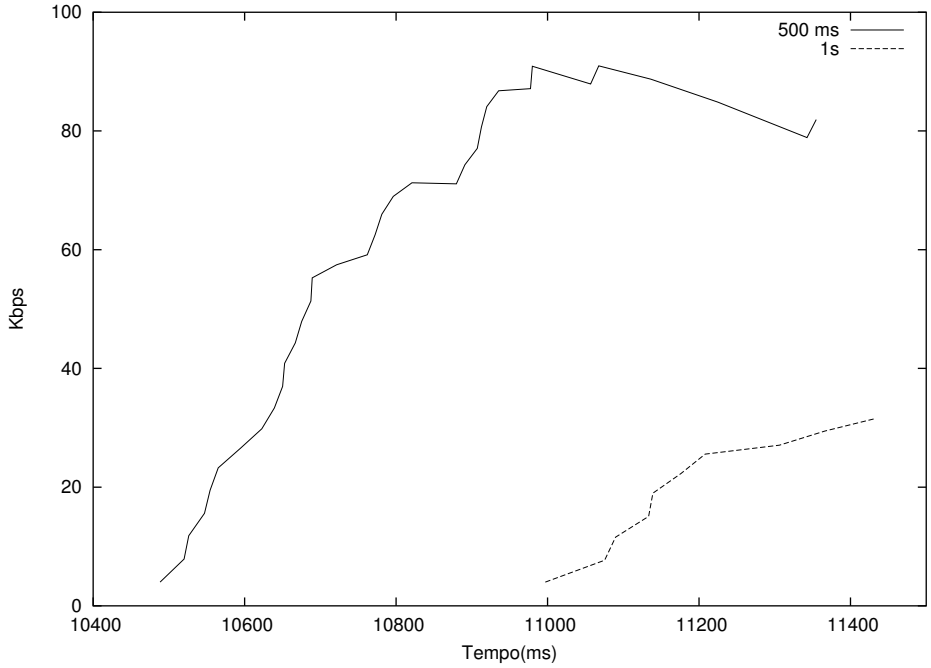

Figura 5.14: Banda desperdiçada para diferentes tempos de armazenamento

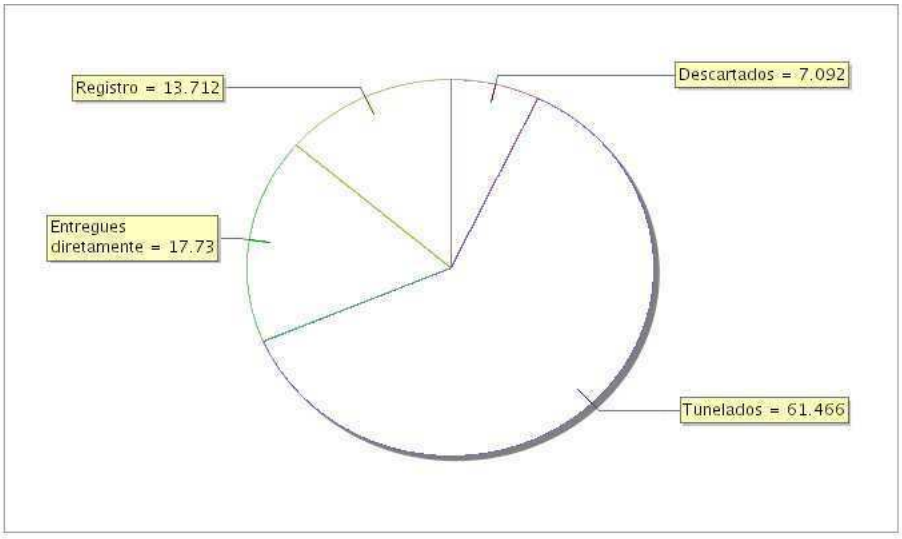

Figura 5.15: Proporção de dados que passaram pelo *home agent*

enlace de 155Mbps. O *home agent* armazena datagramas enviados a dispositivos que não completaram registro de COA por até 1s, o que nos garante um maior fluxo de dados em direção aos *foreign agents*. Nas próximas sub-seções, sumarizamos alguns resultados de simulação.

Simulações com e sem *Smooth Handoff*

Para a observação do desempenho do sistema, variamos, em várias simulações, a taxa de transmissão de dados por parte dos *correspondent nodes*, o número de dispositivos móveis e a freqüência de migrações.

Para cenários com baixa taxa de transmissão de dados e baixo número de migrações por centésimo de segundo, a perda de datagramas no *foreign agent* não é grande o bastante para motivar o uso de um mecanismo como o *smooth handoff*. A banda desperdiçada nos *foreign agents* do sistema durante o período de migrações, considerando dados enviados aos dispositivos móveis numa média de 4Kbps, variou de 1.7 a 5.5Kbps para 100 migrações distribuídas em intervalos de mais de 4s.

No entanto, aumentando a concentração das migrações para dezenas de migrações por centésimo de segundo e a taxa de transmissão para 20kpbs para cada *correspondent node*, pudemos observar um desperdício considerável de dados no *foreign agent* de onde ocorrem as migrações. As simulações deste cenário apresentaram a queda de desempenho de até 45Kbps durante o período de migrações, como podemos conferir na Figura 5.16.

O volume de dados desperdiçados no período de tempo que compreende desde o início das migrações até dois segundos após o seu término equivale a quase 58 por cento dos dados que passaram pelo *foreign agent* em questão, como podemos ver na Figura 5.17.

Simulando o cenário analisado anteriormente com *smooth handoff*, de maneira que o agente estrangeiro possa redirecionar datagramas por até dois segundos após o recebimento de um *binding update*, diminuimos a taxa de dados descartados nas proporções ilustradas na Figura 5.18. Note que a taxa de desperdício de dados diminuiu — atingindo um máximo de aproximadamente 30Kbps — e incidiu sobre um menor intervalo de tempo.

Enquanto que, sem este mecanismo, tínhamos um desperdício de cerca de 58 por cento do tráfego que passava por um *foreign agent* no período que vai até 2s após o término das migrações, sua utilização nos levou a uma perda de pouco mais de 30 por cento. A Figura 5.19

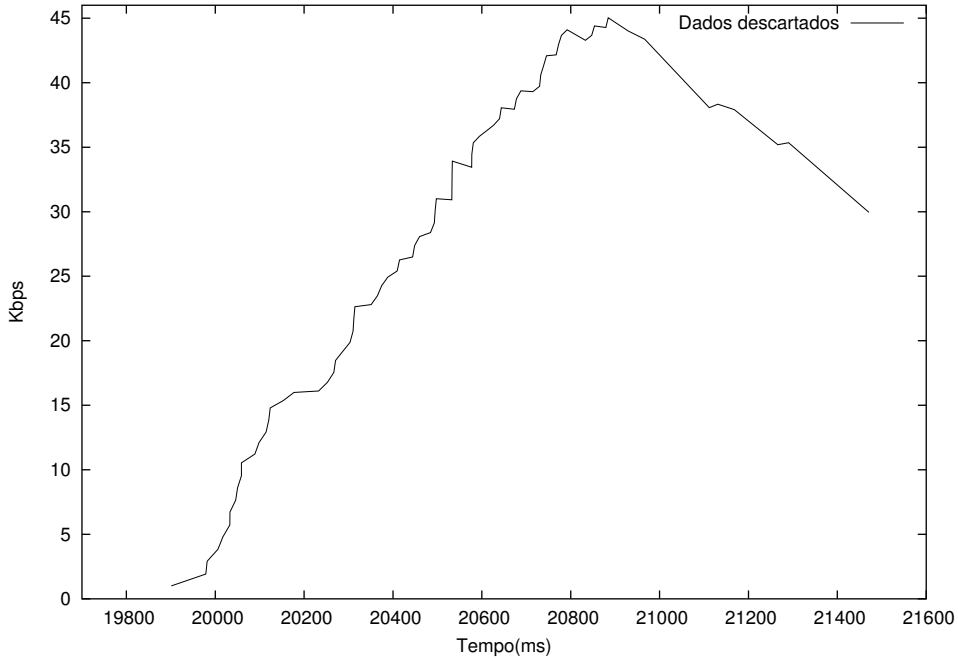

Figura 5.16: Banda desperdiçada

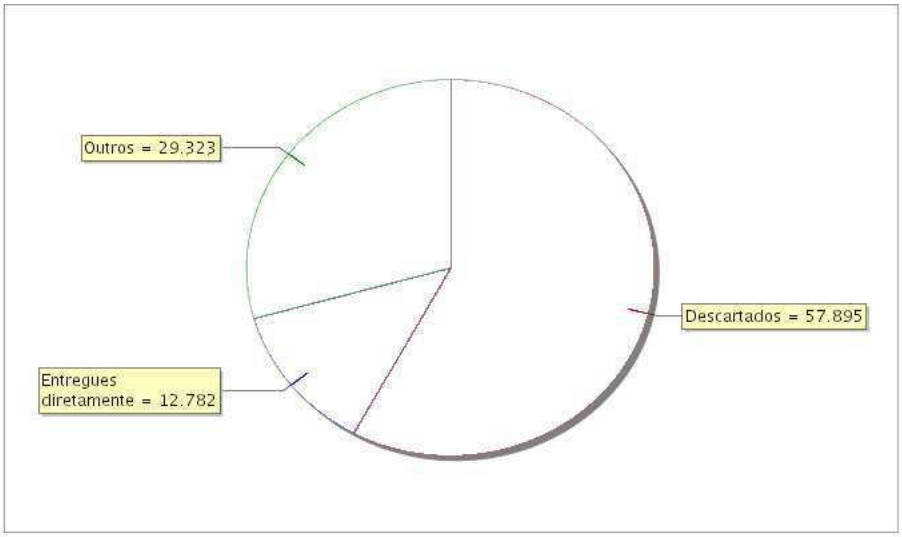

Figura 5.17: Proporção de dados que passaram pelo *foreign agent*

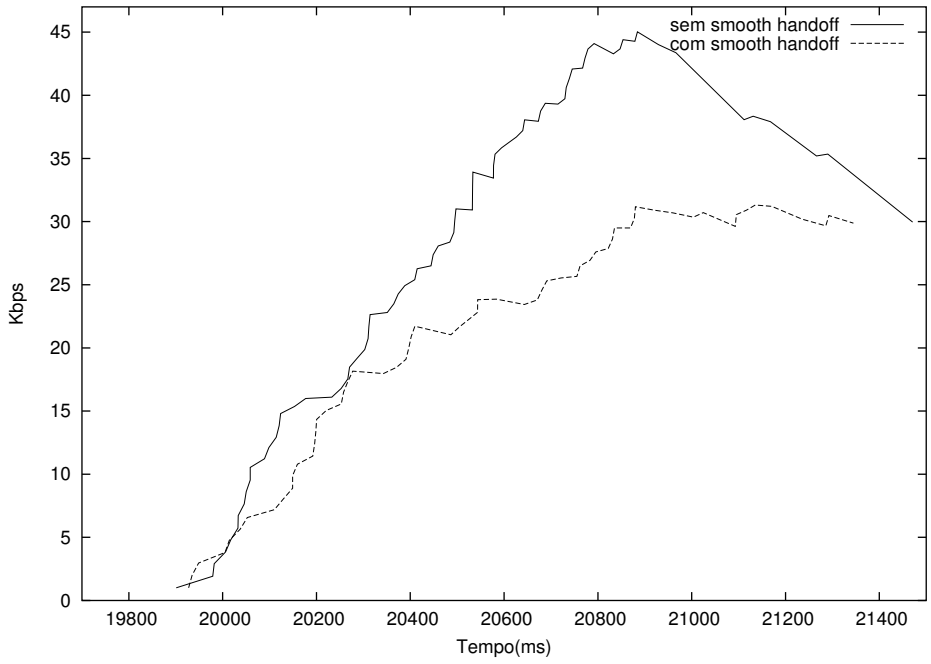

Figura 5.18: Banda desperdiçada com e sem *smooth handoff*

ilustra esses resultados.

5.4.3 Limitações e Considerações Acerca dos Resultados Obtidos

Em nosso trabalho de análise, pudemos observar que a queda de desempenho decorrente de migrações não depende exatamente da quantidade de dispositivos móveis envolvidos, mas da concentração de muitas migrações em curtos intervalos de tempo — mais especificamente, em escalas de milissegundos. Isto se deve especialmente à natureza do tráfego dos *correspondent nodes* (cujas rajadas nem sempre coincidirão com a migração dos dispositivos móveis) e também às suas baixas taxas de transmissão (4Kbps) em alguns cenários de simulação.

A banda desperdiçada, entretanto, pode chegar a aproximar-se de 200Kbps nos piores casos. A perda de datagramas endereçados a dispositivos que não se encontram na rede nativa e não efetuaram registro pode ser atenuada se o *home agent* armazenar tais datagramas em um *buffer* em vez de simplesmente descartá-los. O armazenamento por um período de 1 segundo pode diminuir a perda máxima para cerca de 25Kbps, necessitando, para isso, de um *buffer* de 22KB, numa configuração com até 200 dispositivos móveis. O armazenamento de

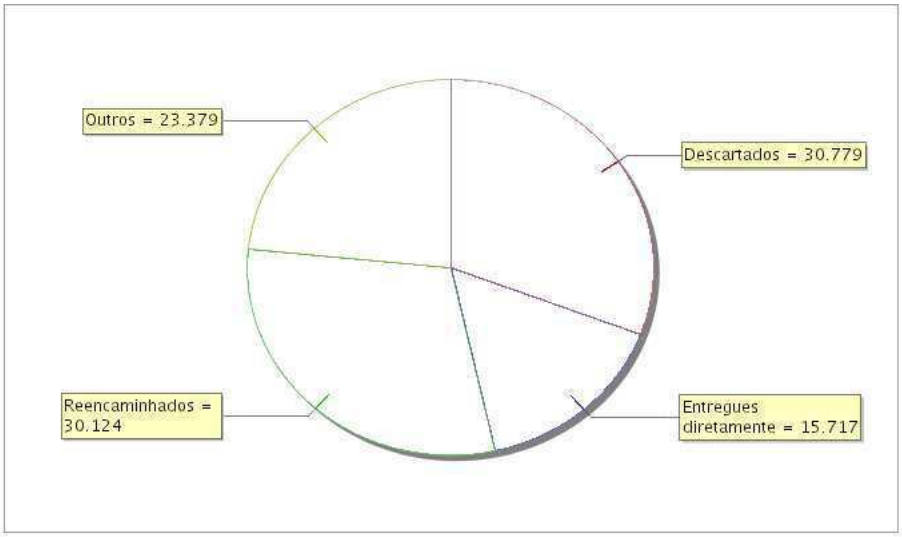

Figura 5.19: Proporção de dados que passaram pelo *foreign agent*

dados no *home agent* não se mostrou extremamente necessário em casos onde as migrações estão dispersas em *grandes* intervalos de tempo (mais de 1 segundo).

No que diz respeito ao *smooth handoff*, a exemplo do que ocorreu com o mecanismo de armazenamento, ele se faz necessário devido ao fato de que, em cenários mais realistas — com uma grande quantidade de dispositivos migrando em menos de um centésimo de segundo — a perda de datagramas pode ser considerável. Em nossas simulações, o uso de *smooth handoff* chegou a diminuir quase que pela metade o desperdício de dados nos *foreign agents* durante o período de migrações.

A principal limitação dos resultados obtidos corresponde à limitação dos modelos de simulação em geral, cujos resultados dizem respeito a simulações de cenários específicos. Procuramos atenuar tais limitações através da análise de diversas simulações, entretanto, nosso objetivo foi, desde o princípio do trabalho, o de mostrar que podemos derivar informações de desempenho através do protótipo de simulador RPOOt que foi desenvolvido.

Capítulo 6

Considerações Finais

Este trabalho teve por objetivo não só a definição formal de uma extensão temporizada para o formalismo RPOO, mas também a construção de um protótipo de ferramenta para a simulação de modelos e análise de desempenho de um protocolo de rede, tomado como estudo de caso.

A idéia de definirmos uma extensão formal temporizada para RPOO surgiu da aplicação deste formalismo à modelagem do protocolo IP Móvel — trabalho que evidenciou algumas deficiências do formalismo com respeito à modelagem de parâmetros temporais e análise de desempenho. De acordo com as deficiências encontradas, o formalismo RPOO foi estendido, dando suporte a interações não-instantâneas entre objetos e integrando suas estratégias de temporização OO com a temporização de ações internas através das redes de Petri usadas na descrição das classes.

Por restrições quanto à duração do trabalho, optamos por limitar o escopo das funcionalidades implementadas na ferramenta, deixando para trabalhos futuros a implementação de algumas características formalmente definidas. As funcionalidades que optamos por implementar — envio e recebimento de mensagens temporizadas — mostraram-se suficientes à modelagem do IP Móvel, protocolo que, além de características inerentes de mobilidade, apresenta diversos parâmetros de tempo. Os parâmetros temporais foram devidamente endereçados pelas ações temporizadas RPOOt e o processo de simulação dos modelos gerou diversas medidas de desempenho com respeito à perda de datagramas durante processo de migração dos dispositivos móveis. Tais medidas de desempenho não poderiam ser obtidas facilmente através de modelos RPOO, como pudemos concluir de trabalhos desenvolvidos

anteriormente com o próprio IP Móvel[9; 36].

6.1 Contribuições

As principais contribuições deste trabalho são o suporte para modelagem de parâmetros de tempo — através das ações RPOOt formalmente definidas — e análise de desempenho — através do protótipo de simulador desenvolvido.

Um ponto importante da estratégia de temporização proposta é a possibilidade de modelagem tanto de interações temporizadas entre objetos quanto de atividades temporizadas internas aos objetos. Isto se torna uma vantagem sobre os modelos bem conhecidos de redes de Petri temporizadas [22; 32; 49], que não oferecem a possibilidade de decomposição OO de seus modelos temporais. Mesmo no mundo da Orientação a Objetos, às vezes a modelagem de interações temporizadas entre objetos é de certa forma negligenciada. Em linguagens como ROOM[40] (cujas principais características foram reaproveitadas em UML-RT[4]) e Real-time UML[4], a temporização se dá na modelagem de atividades internas aos objetos, em diagramas de estado.

O simulador RPOOt que implementamos, apesar de não contemplar todas as ações RPOO/RPOOt, pode ser usado tanto para simulações guiadas pelo usuário, através da interface gráfica, ou para simulações massivas, sem interface gráfica, com grandes quantidades de dados. Simulações guiadas se mostraram importantes na verificação informal da corretude do que foi especificado no modelo. Uma vez que os modelos estejam funcionando corretamente, a simulação massiva de várias configurações com uma grande quantidade de dados de entrada é necessária para a obtenção de medidas de desempenho mais confiáveis. O simulador contempla esse processo, contornando eventuais problemas quanto ao tempo de execução de todas as simulações através da execução dos modelos em grades computacionais.

Desta forma, de acordo com o que foi explanado, podemos dizer que este trabalho difere de outros trabalhos com objetivos similares em diversos sentidos. RPOOt é uma linguagem de modelagem formalmente definida com suporte a temporização intra e inter-objetos e suporte de um protótipo de simulador que pode ser usado em grades computacionais. Além disso, técnicas de verificação formal podem ser usadas na verificação de espaços de estados

RPOOt. Como demos ênfase à simulação de modelos, a exploração dessas técnicas (bem como a geração automática de estados de espaços), tal qual a realizada por Shu [41], se destaca como um interessante trabalho futuro, como discutiremos a seguir.

6.2 Sugestões para Trabalhos Futuros

Diversos trabalhos podem ser derivados deste trabalho de dissertação, tanto em nível teórico (o formalismo em si) como em nível prático (o simulador).

Todos os resultados obtidos com a ferramenta são derivados do processo de simulação, deixando de lado a exploração de espaços de estados temporizados com técnicas de verificação de modelos. Desta forma, um trabalho interessante seria a pesquisa acerca de utilização de lógicas de especificação de propriedades com tratamento explícito de tempo, como CCTL[38] (extensão de CTL), de forma que pudéssemos analisar, em termos de desempenho, quadros mais gerais de execução e não apenas simulações particulares. Este trabalho poderia ser complementado pela extensão de duas ferramentas: o JMobile[12], gerador de espaço de estados RPOO poderia ser estendido para a geração de espaços de estados temporizados RPOOt, ao passo que o verificador de modelos RPOO Veritas[37] poderia ser estendido para a análise de propriedades temporais a partir de espaços de estados temporizados.

Ainda com respeito às simulações, elas se dão, no protótipo da ferramenta, de forma não-determinística. Isto significa que o simulador não oferece meios para que se repitam simulações executando exatamente as mesmas seqüências de ações. Esta característica é extremamente desejável para que simulações com a mesma configuração inicial contemplem diferentes execuções do sistema. Todavia, ao encontrar-se algum erro em um ponto particular da execução do modelo simulado, a repetição exata da simulação que gerou o erro se torna um fator importante para a sua correção. Desta forma, a implementação do suporte a simulações em modo determinístico é um trabalho futuro interessante para a depuração dos modelos, e incide basicamente na extensão do módulo de cálculo de ligações e de simulação da ferramenta.

Trabalhos relativamente mais simples, focados essencialmente em desenvolvimento, podem ser efetivados para incrementar as funcionalidades do simulador Renew/RPOOt. Em

primeira instância, a implementação de todas as ações RPOOt seria o trabalho mais adequado, além da modelagem e simulação de outros protocolos de rede com características temporais (*Bit Torrent*, por exemplo). O compilador, por sua vez, pode ser estendido de maneira a explorar os modelos em busca de *variáveis livres* e outros características indesejadas nos casos mais comuns de modelagem.

No que diz respeito à coleta de dados durante as simulações, esta poderia ser estendida de forma a ser completamente independente das redes que compõem os modelos, como acontece na ferramenta Design'CPN, para redes de Petri coloridas. A coleta de dados através de inscrições *action* pode se dar de maneira completamente independente do funcionamento dos modelos, mas é sujeita a descuidos por parte dos desenvolvedores, uma vez que a coleta de dados é inscrita diretamente nas transições como qualquer outra inscrição.

Por fim, podemos ponderar que a aplicação do protótipo Renew/RPOOt, na prática, à simulação de diversos outros protocolos, poderá evidenciar a necessidade de novos mecanismos de temporização de acordo com as peculiaridades dos sistemas modelados. Isto pode levar, por fim, à realização de extensões do formalismo RPOOt em nível teórico e conseqüentemente, à implementação dos novos mecanismos de temporização no Renew/RPOOt ou eventualmente à implementação de uma nova ferramenta de simulação.

Bibliografia

[1] M. Dell Abate, M. de Marco, e V. Trecordi. Performance Evaluation of Mobile IP Protocols in a Wireless Environment. *IEEE*, 1998.

[2] M. A. Azgomi e A. Movaghar. Towards an Object-oriented Extension for Stochastic Activity Networks. *10th Workshop on Algorithms and Tools for Petri Nets*, páginas 144–155, Setembro 2003.

[3] K. Bergner, R. Grosu, A. Rausch, A. Schmidt, P. Cholz, e M. Broy. Focusing on Mobility. *Proceedings of the 32nd Annual Hawaii International Conference on Systems Science*, 1999.

[4] L. Bichler, A. Radermacher, e A. Schurr. Evaluating UML Extensions for Modeling Real-time Systems. 2002.

[5] C. Blondia, N. Van den Wijngaert, G. Willems, e O. Casals. Performance Analysis of Optimized Smooth Handoff in Mobile IP. Em *Proceedings of the 5th ACM international workshop on Modeling analysis and simulation of wireless and mobile systems*, páginas 22–29. ACM Press, 2002.

[6] M. Boyer e M. Diaz. Non Equivalence Between Time Petri Nets and Time Stream Petri Nets. Em *Proceedings of the 8th Int. Workshop on Petri Net and Performance Models (PNPM'99), 8-10 October 1999, Zaragoza, Spain*, páginas 198–207, 1999.

[7] E. Canedo, J. A. Santos, J. C. de Figueiredo, e D. D. S. Guerrero. Experimenting a Notation Based on Petri Nets and Object-oriented Concepts. *Proceedings of the I Brazilian Petri-Nets Meeting*, 2002.

[8] Z. Dang e R. A. Kemmerer. Using the Astral Model Checker to Analyze Mobile IP. *ACM*, páginas 132–141, 1999.

[9] F. V. de A. Guerra, T. S., J. C. A. de Figueiredo, e D. D. S. Guerrero. Formal Object-oriented Modeling and Validation of Mobile IP Protocol. Setembro 2003.

[10] A. L. L. de Figueiredo. Geração Automática de Casos de Teste para Sistemas Baseados em Casos de Testes. Dissertação de Mestrado, UFCG, 2005.

[11] P. E. e S. Barbosa, T. de M. Silva, J. C. A. de Figueiredo, e D. D. S. Guerrero. Simulação de Modelos RPOO - Entrada de Dados e Parsers, Interface Gráfica, Simulador de Redes de Petri e Integração de Ferramentas. *ENIC 2003*, Dezembro 2003.

[12] T. M. e Silva. Simulação Automática e Geração de Espaço de Estados de Modelos RPOO. Dissertação de Mestrado, Coordenação de Pós-graduação em Informática - COPIN, UFCG, Campina Grande - PB, 2005.

[13] K. Jensen (ed.). Second Workshop on Practical Use of Coloured Petri Nets and Design/CPN.

[14] M. A. Escalante, L. Lavagno, e N. Dimopoulos. Performance Analysis of an Arbiter Using Probabilistic Timed Petri Nets. Em *Proceedings of the International Workshop on Logic Synthesis*, páginas 12–15, maio de 1997.

[15] I. Foster, C. Kesselman, e S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Lecture Notes in Computer Science*, 2001.

[16] H. Giese e S. Burmester. Real-time Statechart Semantics. Technical report, Computer Science Department, University of Paderborn, junho 2003. Relatório técnico trri -03-239.

[17] D. Gilbert. Jfreechart Library. URL: www.jfree.org, 2002.

[18] D. D. S. Guerrero. *Redes de Petri Orientadas a Objetos*. Tese de Doutorado, Curso de Pós-graduação em Engenharia Elétrica, Universidade Federal da Paraíba – Campus II, Campina Grande, Paraíba, Brasil, abril de 2002.

[19] H. Hermanns, J. P. Katoen, J. Meyer-Kayser, e M. Siegle. Towards Model Checking Stochastic Process Algebra. *Lecture Notes in Computer Science*, 1945:420, janeiro de 2000.

[20] P. H. M. Jacobs. *DSOL: An Open Source, Java Based, Suite for Continuous and Discrete Event Simulation*. Technische Universiteit Delft, Delft, Holanda, 2005.

[21] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis, Methods and Practical Use. Volume 1*. mtcs. Springer-Verlag, 1992.

[22] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use.* Volume 2. mtcs. Springer-Verlag, 1994.

[23] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use - Volume 3*. Springer Verlag, 1997.

[24] W. Kaiser. Become a Programming Picasso with JHotDraw – Use the Highly Customizable GUI Framework to Simplify Draw Application Development. *JavaWorld*, fevereiro de 2001.

[25] H. Klaudel e F. Pommereau. Asynchronous Links in the PBC and M-nets. *Proceedings of Asian 99*, páginas 190–200, 1999.

[26] I. Kruger, W. Prenninger, e R. Sandner. Deriving Architectural Prototypes for a Broadcasting System Using UML-RT. Em P. Kruchten, editor, *Proceedings 1st ICSE Workshop on Describing Software Architecture with UML*, 2001.

[27] O. Kummer, F. Wienberg, M. Duvigneau, J. Schumacher, M. Köhler, D. Moldt, H. Rölke, e R. Valk. An Extensible Editor and Simulation Engine for Petri Nets: Renew. Em Jordi Cortadella and Wolfgang Reisig, editors, *Proceedings of the International Conference on Theory and Application of Petri Nets, Bologna, Italy*, volume 3099 / 2004 of *LNCS (Lecture Notes in Computer Science)*, páginas 484–493. Springer Verlag, junho de 2004.

[28] F. Marotta, A. Merzenti, e D. Mandrioli. Modeling and Analyzing Real-time Corba and Supervision & Control Framework and Applications. Em *Proceedings of the The*

21st International Conference on Distributed Computing Systems, página 567. IEEE Computer Society, 2001.

[29] M. A. Marsan e G. Chiola. On Petri Nets with Deterministic and Exponentially Distributed Firing Times. *Lecture Notes in Computer Science: Advances in Petri Nets 1987*, 266:132–145, 1987.

[30] G. R. Mateus e A. A. F. Loureiro. Introdução a Computação Móvel. Relatório técnico, DCC/UFMG, 2000.

[31] P. J. McCann e G. Roman. Modeling Mobile IP in Mobile Unity. *ACM Transactions on Software Engeneering and Methodology*, 8(2):115–146, abril de 1999.

[32] M. K. Molloy. Fast Bounds for Stochastic Petri Nets. Em *International Workshop on Timed Petri Nets, Torino, Itália, 1–3 de julho, 1985*, páginas 244–249. IEEE Computer Society Press, 1985.

[33] E. Pelz e H. Fleishhack. Compositional High Level Petri Nets with Timing Constraints - a Comparison. 2003.

[34] C. Perkins. RFC 3344:IP Mobility Support for IPv4, agosto de 2002. Status: Proposed Standard.

[35] R. R. Razouk. The Derivation of Performance Expressions for Communication Protocols from Timed Petri Net Models. Em *Proceedings of the ACM SIGCOMM symposium on Communications architectures and protocols*, páginas 210–217. ACM Press, 1984.

[36] C. L. Rodrigues, F. V. de A. Guerra, J. C. A. de Figueiredo, e D. D. S. Guerrero. Modeling and Verification of Mobility Issues Using Object-oriented Petri Nets. *3rd International Information and Telecommunication Technologies Simposium*, 2004.

[37] C. L. Rodrigues, J. C. A. de Figueiredo, e D. D. S. Guerrero. Verificação de Modelos em Redes de Petri Orientadas a Objetos. Em *VII Workshop de Teses e Dissertacoes*, Manaus - Brasil, outubro de 2003.

[38] J. Ruf e T. Kropf. Symbolic Verification and Analysis of Discrete Timed Systems. *Form. Methods Syst. Des.*, 23(1):67–108, 2003.

[39] J. A. Santos. Suporte a Análise e Verificação de Modelos RPOO. Dissertação de Mestrado, Universidade Federal de Campina Grande, Campina Grande, Brasil, 2003.

[40] B. Selic. Tutorial: Real-time Object-oriented Modeling (ROOM). Em *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, 1996.

[41] G. Shu, C. Li, Q. Wang, e M. Li. Validating Objected-oriented Prototype of Real-time Systems with Timed Automata. *IEEE International Workshop on Rapid System Prototyping*, 2002.

[42] A. Stephane e A. H. Aghvami. Fast Handover Schemes for Future Wireless IP Networks: A Proposal and Analysis. *VTC spring*, maio de 2001. Rhodes Island Greece.

[43] F. Taiani, M. Paludetto, e J. Delatour. Composing Real-time Objects: A Case for Petri Nets and Girard's Linear Logic. Em *4th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, páginas 298–305, Magdeburg, Alemanha, maio de 2001. IEEE Computer Society.

[44] W.M.P. van der Aalst. Interval Timed Coloured Petri Nets and Their Analysis. Em M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, páginas 453 – 472. Springer-Verlag, junho de 1993.

[45] P. C. Vinh. Formal Specification and Verification of Protocol-based Handover in a Mobile Process. *Proceedings of the Joint 4th IEEE International Conference on ATM and High Speed Intelligent Internet Symposium*, páginas 354–358, 2001.

[46] Y. Wang, S. Yao, Y. Zhao, e M. Zhou. CPN Modeling and Analysis of L2TP. *Proceedings of the International Conference on Computer Networks and Mobile Computing*, páginas 281–288, 2001.

[47] L. Wells. *Performance Analysis Using Coloured Petri Nets*. Tese de Doutorado, University of Aarhus, Aarhus, Dinamarca, agosto de 2002.

[48] A. Zenie. Colored Stochastic Petri Nets. Em *International Workshop on Timed Petri Nets, Torino, Itália, 1–3 de junho de 1985*, páginas 262–271. IEEE Computer Society Press, 1985.

[49] W. M. Zuberek. Timed Petri Nets and Preliminary Performance Evaluation. Em *Proceedings of the 7th Annual Symposium on Comp. Architecture, 6–8 de maio de 1980, La Baule, França*, páginas 88–96, 1980.

[50] W. M. Zuberek. On Generation of State Space for Timed Petri Nets. Em *Proceedings of the 1988 ACM sixteenth annual conference on Computer science*, páginas 239–248. ACM Press, 1988.

[51] W. M. Zuberek. Timed Petri Nets, Definitions, Properties, and Applications. *Microelectronics and Reliability*, 31(4):627–644, 1991.

Apêndice A

Validação do Uso da Ferramenta Renew/RPOOt para Modelos RPOO

A extensão da ferramenta Renew para a simulação de modelos RPOOt baseia-se na hipótese de que as ações de saída síncrona e entrada de dados (originalmente existentes na ferramenta) e as ações para comunicação assíncrona (implementadas posteriormente) correspondem a ações RPOO. Apesar de uma observação superficial de simulações com a ferramenta indicar com certa nitidez que a execução das ações implementadas na ferramenta estão de acordo com a semântica das ações RPOO de saída síncrona/assíncrona e entrada de dados, não havia, após a fase de implementação, nenhuma evidência concreta desta possibilidade.

Este documento tem o objetivo de descrever o processo de validação das ações implementadas na ferramenta Renew/RPOOt. A estratégia adotada é a de comparar seqüências de ações a partir de simulações com o Renew/RPOOt com espaços de estados obtidos a partir do JMobile, ferramenta de geração de espaços de estados RPOO. Caso alguma seqüência de ações não conste nos espaços de estados, a semântica das ações implementadas certamente não será a mesma das ações RPOO. Caso contrário, teremos um forte indício de que as ações implementadas são, de fato, ações deste formalismo.

Para este processo, dois modelos diferentes serão estudados com respeito às seqüências de ações e espaços de estados. Esta fora do escopo deste documento o detalhamento da semântica dos problemas modelados. Desta forma, ilustraremos os modelos em estudo e concentraremos esforços em comparar suas respectivas simulações com seus respectivos espaços de estados.

A.1 Modelo dos Filósofos

Nosso processo de validação iniciar-se-á pela análise do largamente conhecido problema do jantar dos filósofos. Primeiramente, geramos o espaço de estados para o modelo numa configuração inicial específica, através do JMobile. Depois disso, simulamos o mesmo modelo (com a mesma configuração inicial) diversas vezes no Renew/RPOOt, para construirmos um espaço de estados através da simulação de todas as possibilidades de execução do modelo.

No sistema, filósofos compartilham garfos e precisam alocá-los para poder *comer* ou desalocá-los para voltar a *pensar*. A alocação de garfos se dá através da troca síncrona de mensagens enquanto que o processo complementar se dá através da troca assíncrona de mensagens.

O sistema consiste de apenas duas classes, **Filosofo** e **Garfo**, ilustradas nas Figuras A.1 e A.2 (a sintaxe das inscrições ilustradas é descrita de acordo com a sintaxe utilizada na ferramenta JMobile).

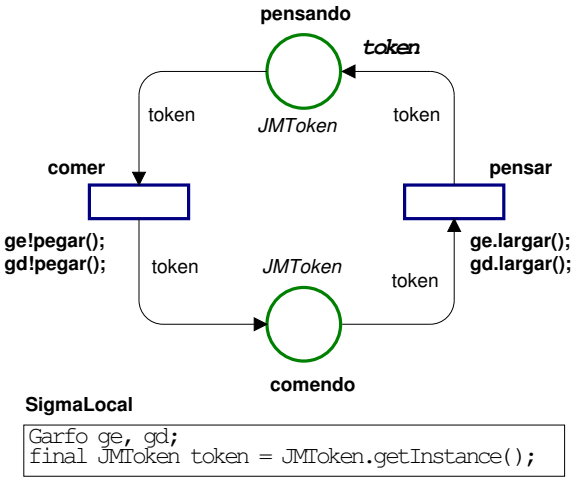

Figura A.1: A rede Filosofo

Para uma configuração inicial com dois filósofos e dois garfos, temos o espaço de estados ilustrado na Figura A.3. Cada nó do espaço de estados é representado por uma linha, que é dividida em quatro partes pelo símbolo $|$. A primeira parte indica o número do estado e os objetos do sistema; a segunda indica as mensagens pendentes; a terceira indica as ações que podem ser executadas e a que estado sua execução leva e a última parte indica os estados antecessores.

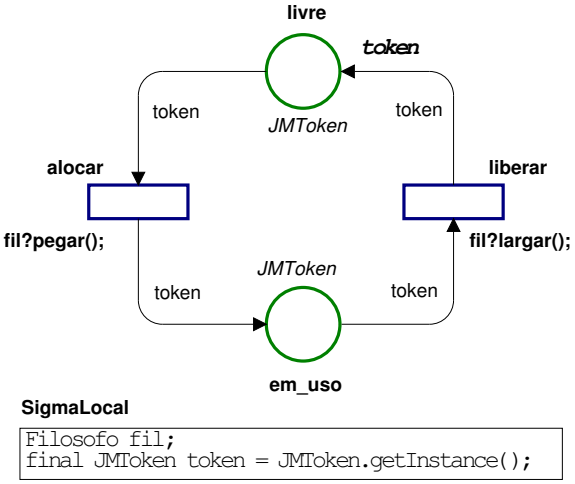

Figura A.2: A rede Garfo

```
File  Edit  Tools  Syntax  Buffers  Window  Help
0 : f1[g1 g2] + f2[g1 g2] + g1 + g2  |  | f2:g2!pegar() & f2:g1!pegar() & g1:f2?p
egar() & g2:f2?pegar() > 1  + f1:g2!pegar() & f1:g1!pegar() & g1:f1?pegar() & g2:
f1?pegar() > 2 | 5 6 7 8
1 : f1[g1 g2] + f2[g1 g2] + g1 + g2  |  | f2:g2.largar() & f2:g1.largar() > 3 | 0
2 : f1[g1 g2] + f2[g1 g2] + g1 + g2  |  | f1:g2.largar() & f1:g1.largar() > 4 | 0
3 : f1[g1 g2] + f2[g1 g2] + g1 + g2  | f2:g1(largar()) + f2:g2(largar()) | g1:f2?
largar() > 5  + g2:f2?largar() > 6 | 1
4 : f1[g1 g2] + f2[g1 g2] + g1 + g2  | f1:g1(largar()) + f1:g2(largar()) | g1:f1?
largar() > 7  + g2:f1?largar() > 8 | 2
5 : f1[g1 g2] + f2[g1 g2] + g1 + g2  | f2:g2(largar()) | g2:f2?largar() > 0 | 3
6 : f1[g1 g2] + f2[g1 g2] + g1 + g2  | f2:g1(largar()) | g1:f2?largar() > 0 | 3
7 : f1[g1 g2] + f2[g1 g2] + g1 + g2  | f1:g2(largar()) | g2:f1?largar() > 0 | 4
8 : f1[g1 g2] + f2[g1 g2] + g1 + g2  | f1:g1(largar()) | g1:f1?largar() > 0 | 4

~
                                                                1,1           All
```

Figura A.3: A rede Garfo

Partindo da mesma configuração ilustrada no estado **0**, na Figura A.3, iniciamos a simulação do mesmo modelo, com a sintaxe adaptada para a utilizada em Renew/RPOOt, evidentemente. Internamente, a ferramenta identifica por **Filosofo[369]**, **Filosofo[362]**, **Garfo[371]** e **Garfo[364]** os objetos identificados no JMobile por **f1**, **f2**, **g1** e **g2**.

Identificamos as ações potenciais através da verificação dos *bindings* das transições às quais as ações RPOO estão inscritas. As Figuras A.4 e A.5 nos ilustram os *bindings* para as transições passíveis de disparo na configuração inicial. De acordo com as inscrições destas transições, poderemos executar as mesmas ações descritas no estado inicial ilustrado na Figura A.3, guardadas as diferenças na identificação dos objetos por parte das duas ferramentas.

Figura A.4: Ligações possíveis

Figura A.5: Ligações possíveis

Executando a transição **comer** (e ações subjacentes) do objeto **Filosofo**[**362**], o sistema é levado a um estado onde apenas a transição **pensar** daquele objeto estará habilitada. A Figura A.6 ilustra a ligação possível para o disparo da transição, neste estado. O disparo da transição executa duas chamadas RPOO de saída de dados, enviando para cada um dos garfos uma mensagem assíncrona.

Isto leva o sistema a um estado onde duas mensagens estarão pendentes, uma para cada garfo. As Figuras A.7 e A.8 ilustram as mensagens pendentes para cada objeto do tipo Garfo

Figura A.6: Ligações possíveis

(note que as mensagens estão de acordo com o estado **3** do espaço de estados ilustrado na Figura A.3).

Figura A.7: Mensagem pendente

Figura A.8: Mensagem pendente

O estado do sistema, neste momento, habilita a transição **liberar** nos dois garfos. A transição inscreve uma ação de entrada de dados e sua habilitação, em ambos os objetos, é ilustrada nas Figuras A.9 e A.9. O disparo da transição *liberar* em ambos os objetos **Garfo[371]** e **Garfo[364]** leva o sistema de volta ao estado inicial. Repetindo todo este processo para o objeto **Filosofo**[**369**], verificamos que o comportamento do modelo simulado é exatamente o mesmo descrito em seu espaço de estados gerado com o JMobile.

A.2 Modelo *Stop and Wait*

Nesta seção, utilizamos um abordagem diferente da seção anterior. Desta feita, partimos de um modelo temporizado, geramos (através de simulações) seqüências de ações, na forma de arquivos de texto e, por fim verificamos se a seqüência de ações *temporizadas* está presente

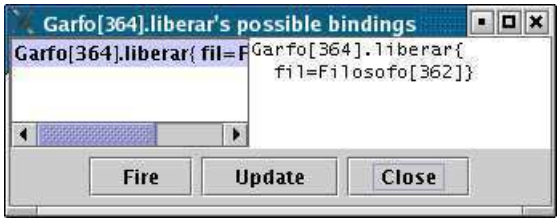

Figura A.9: Ligações possíveis

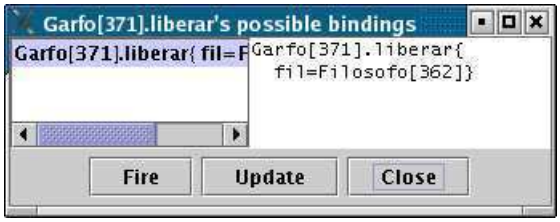

Figura A.10: Ligações possíveis

no espaço de estados. A verificação é levada a cabo por um pequeno módulo Java, recebe como entrada arquivos textuais correspondentes a seqüências de ações e espaços de estados RPOO. O programa percorre o espaço de estados em busca da seqüência de ações recebida como entrada. Sua saída é simplesmente um valor *booleano*, indicando se a seqüência foi ou não encontrada no espaço de estados.

O modelo em estudo é uma versão simplificada do *Stop and Wait*. este protocolo envolve três entidades, servidor, rede e cliente, modeladas pelas classes, Server, Network e Client. O objetivo do protocolo é especificar a troca de uma (pré-estabelecida) seqüência de pacotes entre servidor e cliente. De acordo com o protocolo, o servidor endereça um pacote numerado para o cliente e este, ao recebê-lo, endereça ao servidor um pacote de confirmação, numerado com o número do próximo pacote que o cliente espera receber. Recebendo a confirmação do cliente, o servidor envia o próximo pacote da seqüência. Este processo se repete até que todos os pacotes da seqüência sejam recebidos pelo cliente e devidamente confirmados. As classes do sistema estão ilustradas nas Figuras A.11, A.12 e A.13.

A comunicação entre servidor e cliente se dá através da rede, que simplesmente repassa os pacotes aos seus destinatários com um determinado atraso. As comunicações Server/Network e Client/Network também são temporizadas. Na geração do espaço de estados do mesmo modelo, as informações temporais serão simplesmente ignoradas, e a comunicação se dará de acordo com a semântica da troca de mensagens assíncronas RPOO.

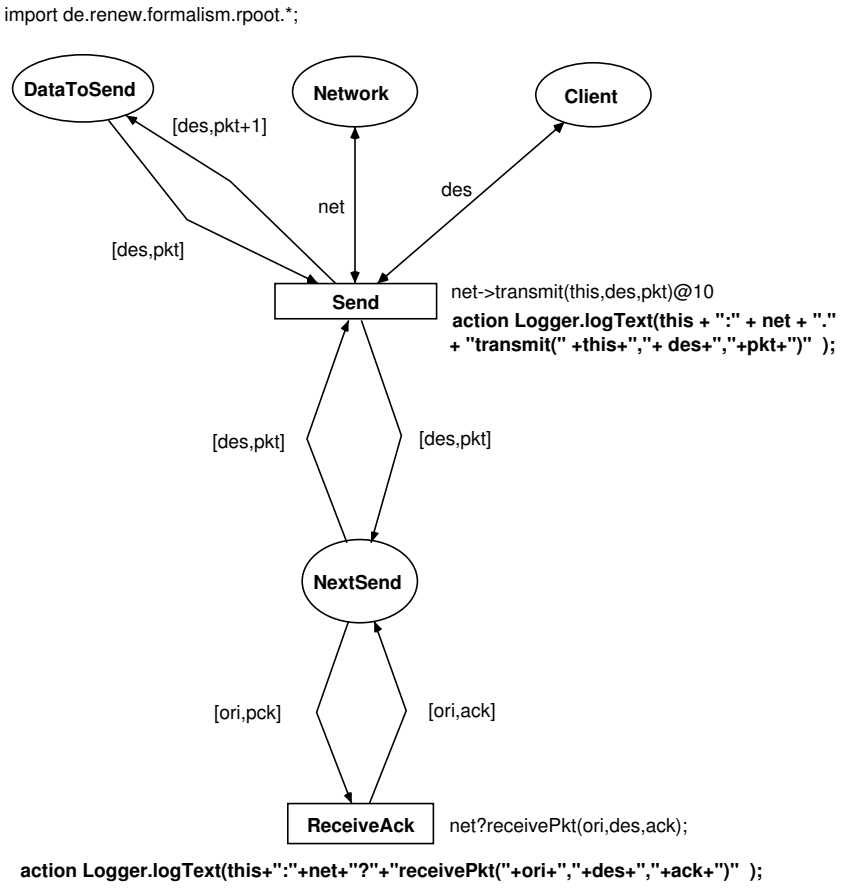

Figura A.11: A rede Server

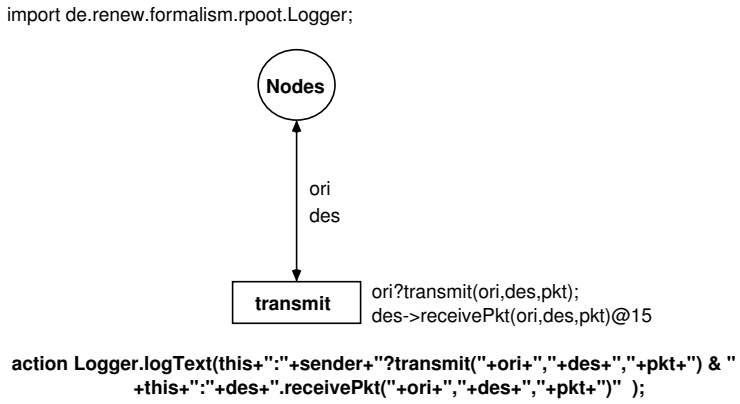

Figura A.12: A rede Network

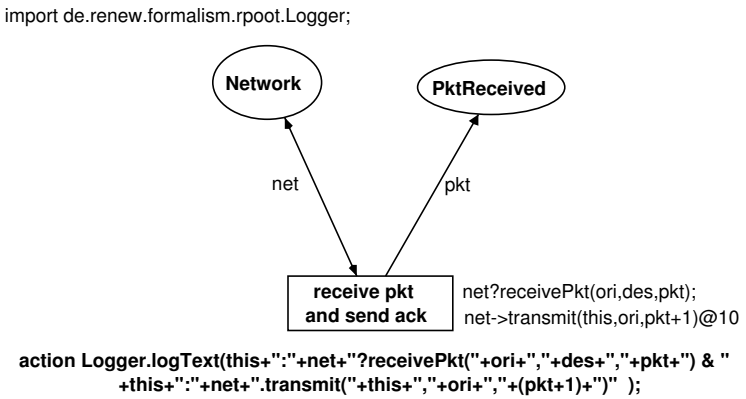

Figura A.13: A rede Client

As Figuras A.14 e A.15 ilustram um arquivo de seqüencia de ações e um espaço de estados para o modelo, ambos considerando uma configuração inicial com dois pacotes a serem enviados do servidor ao cliente. Em face da pouca quantidade de ações/nós nos arquivos nesta configuração, podemos ver, sem o auxílio do módulo Java de verificação, que a seqüência de ações está presente no espaço de estados. Note que as instâncias de Server, Network e Client são identificadas no Renew/RPOOt por **Server**[**5**], **Network**[**6**] e **Client**[**11**] (os números não têm nenhuma relação com a quantidade de instâncias presentes na simulação) ao passo que, no JMobile, as instâncias são identificadas por **s1**, **net** e **c1**.

File Edit Tools Syntax Buffers Window Help

```
 1 Server[5]:Network[6].transmit(Server[5],Client[11],1)
 2 Network[6]:Server[5]?transmit(Server[5],Client[11],1) & Network[6]:Clien
   t[11].receivePkt(Server[5],Client[11],1)
 3 Client[11]:Network[6]?receivePkt(Server[5],Client[11],1) & Client[11]:Ne
   twork[6].transmit(Client[11],Server[5],2)
 4 Network[6]:Client[11]?transmit(Client[11],Server[5],2) & Network[6]:Serv
   er[5].receivePkt(Client[11],Server[5],2)
 5 Server[5]:Network[6]?receivePkt(Client[11],Server[5],2)
 6 Server[5]:Network[6].transmit(Server[5],Client[11],2)
 7 Network[6]:Server[5]?transmit(Server[5],Client[11],2) & Network[6]:Clien
   t[11].receivePkt(Server[5],Client[11],2)
 8 Client[11]:Network[6]?receivePkt(Server[5],Client[11],2) & Client[11]:Ne
   twork[6].transmit(Client[11],Server[5],3)
 9 Network[6]:Client[11]?transmit(Client[11],Server[5],3) & Network[6]:Serv
   er[5].receivePkt(Client[11],Server[5],3)
10 Server[5]:Network[6]?receivePkt(Client[11],Server[5],3)
~
~
                                                          1,1           All
```

Figura A.14: Execução de ações: cada linha representa a execução de um evento RPOO

Nosso programa verificou a que todas as seqüências geradas através de simulações no

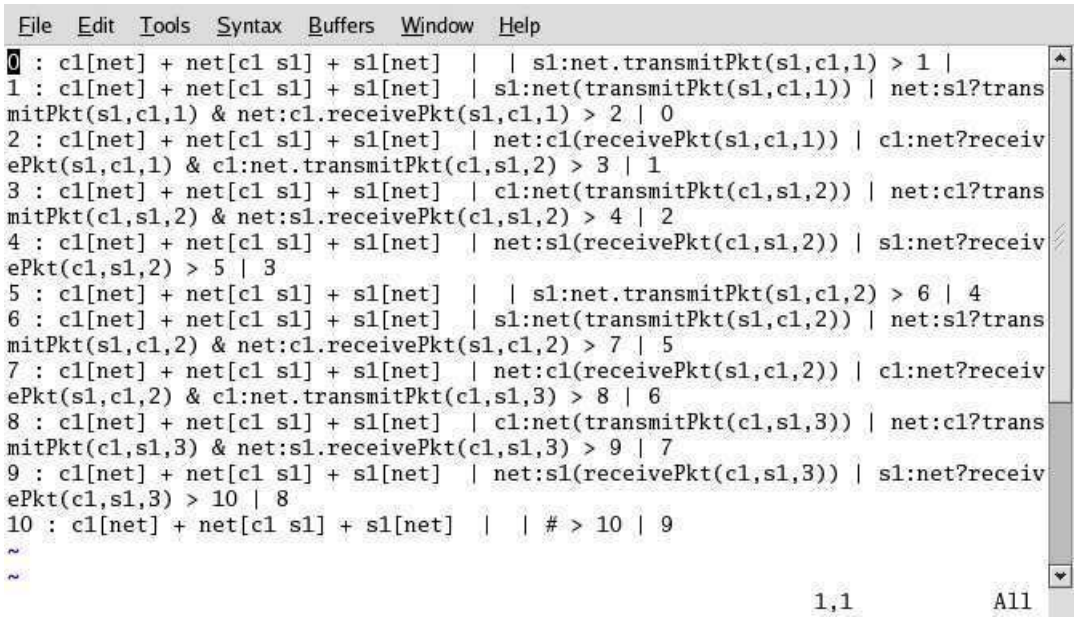

Figura A.15: Espaço de estados: cada linha representa um estado

Renew/RPOOt estão presentes no espaço de estados gerado através do JMobile. Este experimento considerou configurações iniciais com até quatro servidores e seqüências de quatro pacotes, gerando espaços de estados de até 65000 nós.

A.3 Resultados

Na Seção A.1 analisamos e validamos a troca síncrona e assíncrona de mensagens através do modelo dos filósofos, no qual constam estes dois tipos de interação entre objetos. Uma série de simulações guiadas construiu um espaço de estados equivalente ao gerado para o mesmo modelo por uma ferramenta RPOO específica (JMobile). Os experimentos descritos na Seção A.2 complementam os resultados da Seção A.1, analisando a troca de mensagens assíncronas em um contexto diferente, com temporização.

De acordo com a comparação entre seqüências de ações geradas a partir de simulações e seqüências de ações presentes em espaços de estados, consideramos válida a afirmação de que as ações RPOOt implementadas na ferramenta Renew/RPOOt estão de acordo com a semântica das ações formalmente definidas para o formalismo.

Temos, também, um indício de que o esquema de temporização proposto para troca assíncrona de mensagens não acrescenta *à lógica* de um determinado modelo nenhuma ação

diferente das ações que constam em seu espaço de estados não-temporizado. Em outras palavras, podemos dizer que a temporização definida acrescenta *restrições* ao comportamento de um modelo onde os aspectos temporais fossem ignorados. Um espaço de estados temporizado seria, desta forma, um sub-conjunto do espaço de estados não-temporizado[1].

Salientamos, todavia, que uma afirmação definitiva de que as ações de interações entre objetos implementadas na ferramenta Renew/RPOOt correspondem a ações RPOOt compreenderia uma prova formal de equivalência entre as inscrições de interação síncrona de RPOO e das *Redes de Referência*, já que a troca síncrona de mensagens implementada originalmente em Renew (estendida em Renew/RPOOt) advém deste formalismo.

[1]Uma relação similar ocorre entre as CPN's e as TCPN's. Pode-se verificar formalmente que os espaços de estados destas constituem um sub-conjunto do espaço de estados daquelas