Verificação de Modelos em Redes de Petri Orientadas a Objetos

Cássio Leonardo Rodrigues

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
Dalton Dario Serey Guerrero
(Orientadores)

Campina Grande, Paraíba, Brasil
©Cássio Leonardo Rodrigues, Fevereiro-2004

UFCG - BIBLIOTECA - CAMPUS I
485 19-04-04

Ficha Catalográfica

R696V 2004

RODRIGUES, Cássio Leonardo
Verificação de Modelos em Redes de Petri Orientadas a Objetos/ Cássio Leonardo Rodrigues. – Campina Grande. UFCG, 2004.

109p. Il.

Dissertação (Mestrado em Informática). UFCG/CCT. Inclui bibliografia. Orientadores: Jorge Figueiredo, D. Sc e Dalton Guerrero, D. Sc

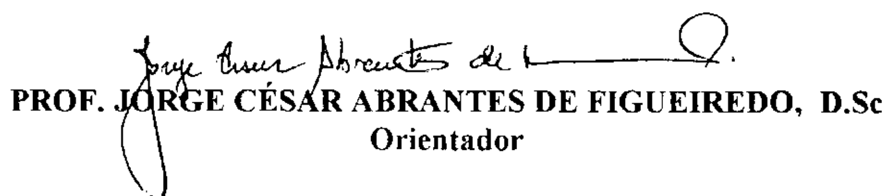
1. Verificação de Modelos, 2. Orientação a Objetos, 3. Rede de Petri, 4. Lógica Temporal CTL
I. Título

CDU: 004.415.5

"VERIFICAÇÃO DE MODELOS EM REDES DE PETRI ORIENTADAS A OBJETOS"

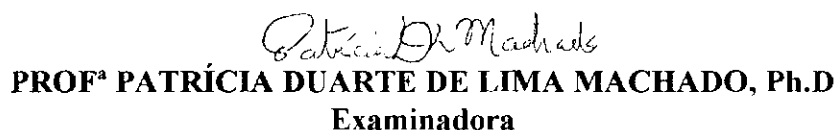
CÁSSIO LEONARDO RODRIGUES

DISSERTAÇÃO APROVADA EM 18.02.2004



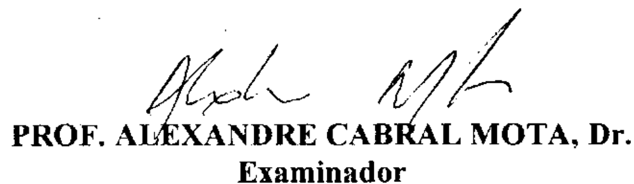
PROF. JORGE CÉSAR ABRANTES DE FIGUEIREDO, D.Sc
Orientador

PROF. DALTON DARIO SEREY GUERRERO , D.Sc
Orientador



PROFª PATRÍCIA DUARTE DE LIMA MACHADO, Ph.D
Examinadora

PROF. ANGELO PERKUSICH, D.Sc
Examinador



PROF. ALEXANDRE CABRAL MOTA, Dr.
Examinador

CAMPINA GRANDE – PB

Agradecimentos

Agradeço aos meus orientadores pelo incentivo, dedicação e envolvimento no trabalho. Outra pessoa que contribuiu muito e que só tenho a agradecer é o aluno de graduação Paulo Eduardo.

Seria injustiça de minha parte não considerar a contribuição de todos os demais integrantes do Grupo de Métodos Formais da UFCG. Em especial, agradeço aos alunos de mestrado Fabrício Guerra e Taciano Morais pela ajuda no estudo de caso realizado no meu trabalho.

Agradeço às agências CAPES e CNPq que financiaram o desenvolvimento do trabalho.

Para fazer este mestrado longe da minha cidade natal e da minha família, eu precisei de muitas pessoas que não necessariamente fizeram parte do meu ambiente acadêmico. Portanto, agradeço a Maria Geneci e Carmem por me acolherem no Pensionato Floriano, de forma especial, nos meus primeiros seis meses em Campina Grande. Agradeço a Nazareno e Danilo pelo companherismo, amizade e bons momentos vividos, principalmente no saudoso tempo em que moramos juntos. Outras pessoas, sem que soubessem, também me ajudaram sendo companheiras e amigas. Sinto-me orgulhoso de poder agradecer a Amâncio, Claudia, Elizeu, Emerson, Glaucimar, Juliana, Lauro e Leandro.

Também não posso me esquecer dos amigos da época da graduação, que remotamente me acompanharam. Muito obrigado, Alessandro, Almeidinha, Javé, Rodrigo, Tatiana e Tiago.

Terno e eternamente devo agradecer a minha família (incluindo Wagner e D. Cida) pelo afeto, apoio e paciência incondicionais em toda minha existência. Em especial, agradeço a Karina por ser a minha fonte de inspiração e companheira nos bons e os maus momentos desta jornada.

Resumo

Nos dias de hoje, os sistemas de software e hardware estão presentes em situações em que falhas são inaceitáveis, por exemplo, em comércio eletrônico, sistemas de telefonia, sistemas bancários, sistemas hospitalares etc. Uma atividade essencial para garantir que estes sistemas funcionem conforme esperado é a aplicação de técnicas formais em seus processos de desenvolvimento. Uma técnica formal cada vez mais utilizada na academia e na indústria é a verificação de modelos. As principais vantagens da verificação de modelos são o poder de automação e a qualidade dos resultados produzidos.

A verificação de modelos foi desenvolvida originalmente para sistemas de hardware. Esta característica pode dificultar a aplicação da técnica em desenvolvimento de software baseado em modelos. Principalmente em software desenvolvido segundo o paradigma OO. Neste trabalho, nós tratamos da técnica de verificação de modelos em Redes de Petri Orientadas a Objetos (RPOO). RPOO é uma linguagem de modelagem formal que integra os conceitos de redes de Petri e OO, de modo a preservar as características originais de cada uma das abordagens. Desde a sua formalização, RPOO tem sido aplicada em vários modelos de sistemas concorrentes e distribuídos. Contudo, antes da realização deste trabalho, a análise destes modelos estava restrita à técnica de simulação. Não existia suporte ferramental adequado para a validação formal de modelos em RPOO.

Para tornar a aplicação desta em técnica no desenvolvimento de softwares baseados em modelos com notação em OO mais viável, definimos um formato para representação de espaço de estados que evidencia a visão OO das modelagens e oculta detalhes das redes de Petri. Também definimos uma estratégia para construção desta estrutura com suporte ferramental. O principal resultado do trabalho é um protótipo de um verificador de modelos capaz de avaliar fórmulas em lógica temporal CTL. Por último, realizamos um estudo de caso em que aplicamos o verificador em uma modelagem do protocolo IP móvel. Nesta atividade, encontramos erros de modelagem não detectados com simulação.

Abstract

Today, hardware and software systems are used in situations where failures are unacceptable, for example, in electronic commerce, telephone systems, banks systems, hospitals systems etc. An essential activity to guarantee that systems work as expected is the application of formal techniques in their development process. A formal technique more and more used in academy and industry is model checking. The main advantages of this technique are power of automation and the quality of results.

Model checking was originally developed to hardware systems. This can difficult the application of model checking in model based software development. Mainly in software developed using the OO paradigm. In this work, we deal with model checking techniques for object oriented Petri nets (Redes de Petri Orientadas a Objetos – RPOO). RPOO is a formal modelling language that integrates Petri nets and OO concepts, and preserves originals features of each one of the approaches. Since its formalization, RPOO has been applied in several concurrent and distributed systems models. Though, before this work, analysis of models was restricted to simulation. There was no supporting tool to deal with formal validation of RPOO models.

To make the application of this technique in model based software development with OO notation more feasible, we define a layout to state space representation that shows up the OO view and holds back the Petri nets details. We define also an approach to construct this structure with tool support. The main result of the work is a prototype of the model checker wich is able to evaluate formulas in CTL temporal logic. Finally, we have conducted a case study in wich we have used the prototype in mobile IP protocol model. In this activity we found modeling mistakes not detected with simulation.

Capítulo 1

Fundamentação Teórica

Neste capítulo, apresentamos os fundamentos teóricos que são a base do trabalho desenvolvido. O capítulo está divido em duas partes principais. Na primeira parte, fazemos uma breve apresentação da técnica de verificação de modelos. Em particular, mostramos a formalização das principais lógicas temporais e discutimos soluções algorítmicas para a avaliação de propriedades de sistemas descritos segundo as mesmas. Na segunda parte, apresentamos informalmente a linguagem de modelagem RPOO. A descrição formal desta linguagem se encontra no Apêndice A.

1.1 Verificação de Modelos

A verificação de modelos é uma técnica automática para analisar o espaço de estados finito de sistemas concorrentes [CWA$^+$96]. Tradicionalmente, a aplicação desta técnica ocorre através de três etapas:

1. Modelagem: esta etapa consiste em construir um modelo formal do sistema e derivar dele *todos* os comportamentos possíveis do sistema. A estrutura que contém todos os comportamentos possíveis é conhecida como *espaço de estados* do sistema.
2. Especificação: esta etapa consiste em especificar os comportamentos *desejáveis* do sistema. Um comportamento se deseja do sistema pode ser descrito formalmente através de lógicas temporais ou máquinas de estado.

3. Verificação: esta etapa consiste em submeter o modelo e as especificações a uma ferramenta chamada *verificador de modelos*. Esta ferramenta produz como resultado um valor verdade que indica se a especificação é satisfeita ou não no modelo. Em caso negativo, o verificador fornece uma seqüência de estados alcançáveis, chamada de *contra-exemplo*, que demonstra que a especificação não é válida no modelo.

A verificação de modelos pode ser aplicada em sistemas reativos, que se caracterizam por uma interação contínua com o ambiente no qual estão inseridos. Sistemas desta natureza tipicamente recebem estímulos do ambiente e quase que imediatamente reagem às entradas recebidas. Tradicionalmente, eles são complexos, distribuídos, concorrentes e não possuem um término de execução, isto é, eles estão constantemente prontos para interagir com o usuário ou outros sistemas. Este conjunto de características exige que as propriedades destes sistemas sejam definidas não apenas em função de valores de entrada e saída, mas também em relação à *ordem* em que os eventos ocorrem.

As lógicas temporais são utilizadas para a especificação de propriedades em verificação de modelos porque elas são capazes de expressar relações de ordem, sem recorrer à noção explícita de tempo. Tradicionalmente, duas formalizações de lógica temporal são utilizadas no contexto de verificação de modelos: LTL (*Linear Temporal Logic*) [Pnu77] e CTL (*Computation Tree Logic*) [Sif90]. A abordagem em LTL considera que uma propriedade pode ser quantificada para todas as execuções do sistema. A abordagem em CTL, por sua vez, considera que uma propriedade pode ser quantificada para para uma ou para todas as execuções do sistema.

O principal desafio à aplicação de verificação de modelos em situações reais é o problema conhecido como *explosão do espaço de estados*. Registrar todos os comportamentos possíveis de um sistema complexo pode esgotar os recursos de memória de uma máquina, mesmo que o número de estados alcançados pelo sistema seja finito. Muitos trabalhos de pesquisa têm sido realizados neste contexto e há, atualmente, um número considerável de técnicas para tratar deste problema. Na área de desenvolvimento de sistemas de hardware, por exemplo, a técnica para a representação simbólica do espaço de estados, desenvolvida por McMillan, viabilizou a aplicação de verificação de modelos no protocolo IEEE Futurebus+ [EOH+93]. Na área de desenvolvimento de sistemas de software, a explosão do espaço de estados também constitui uma barreira à aplicação de verifica-

ção de modelos, porém, a representação simbólica nem sempre pode ser adequadamente aplicada, pois sistemas de software são intrisicamente mais complexos, com comportamento majoritariamente assíncrono. Desta forma, outros trabalhos para tratar deste problema foram desenvolvidos: interpretação abstrata [DHJ+01] e redução de ordem parcial [GPS96; Val98], entre outros.

1.1.1 Lógica Temporal Linear

Sintaxe e semântica A sintaxe da lógica temporal linear (LTL) tem como ponto de partida o conjunto de proposições atômicas, denotado por *AP*. Uma proposição atômica *p* é uma setença que informa algo a respeito de um determinado estado do sistema, ela pode ser interpretada como sendo verdadeira ou falsa. Alguns exemplos são: "x é igual a zero", "o recurso r está alocado", "o processo 1 está na seção crítica". A definição seguinte determina o conjunto de fórmulas que pode ser declarado em lógica temporal linear.

Definição 1.1 (Sintaxe de LTL) *Seja AP o conjunto de proposições atômicas, então*

1. *cada proposição $p \in AP$ é uma fórmula LTL;*
2. *se ϕ e ψ são fórmulas LTL, então $\neg\phi$, $\phi \wedge \psi$, $X\phi$, $\phi U \psi$, $G\phi$ e $F\phi$ também são.*

Os operadores temporais são X (leia-se próximo), U (leia-se até), G (leia-se globalmente) e F (leia-se futuramente). Formalmente, a lógica temporal linear é interpretada em uma seqüência de estados. Intuitivamente, $X\phi$ significa que a fórmula ϕ é deve valer no próximo estado da seqüência, $\phi U \psi$ significa que ϕ deve valer até que ψ seja verdadeira. $G\phi$ significa que ϕ deve ser verdadeira no estado atual e em todos estados alcançáveis. $F\phi$ significa que ϕ deve ser verdadeira no estado atual ou em algum estado do futuro. Os operadores temporais G (globalmente ou sempre) e F (futuramente) podem ser definidos por:

$$G\phi \equiv \neg F \neg \phi$$

e

$$F\phi \equiv true U \phi$$

A noção de "estado" e "próximo estado" é definida em relação à *estrutura de Kripke*, formalizada na definição seguinte:

Definição 1.2 (Estrutura de Kripke) *Uma estrutura de Kripke é uma tupla* $\mathcal{M} = (S, I, R, Label)$ *na qual:*

- S *é um conjunto finito de estados,*
- $I \subseteq S$ *é um conjunto de estados iniciais,*
- $R \subseteq S \times S$ *é uma relação de transição satisfazendo*

$$\forall s \in S.(\exists s' \in S.(s, s') \in R)$$

- $Label : S \to 2^{AP}$, *associando a cada estado* s *de* S, *proposições atômicas* $Label(s)$ *que são válidas em* s.

Portanto, uma estrutura de Kripke é uma máquina de estados finita que representa todas as possíveis execuções de um sistema. Cada estado do sistema é rotulado com proposições atômicas que são verdadeiras nele. Segundo a definição de R, cada estado deve possuir ao menos um sucessor. Desta forma, situações reais nas quais um estado s não possui um sucessor (*deadlock*) devem ser representadas através de $(s, s) \in R$, isto é, através de auto-laço.

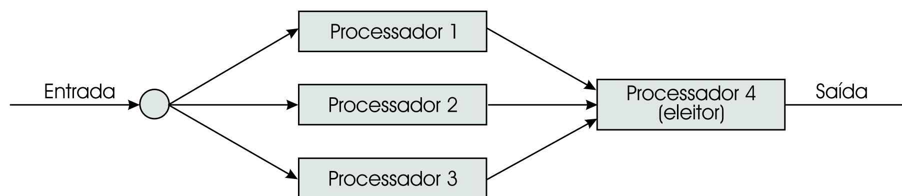


Figura 1.1: Um sistema tolerante a falhas

Para exemplificar uma estrutura de Kripke, considere o sistema tolerante a falhas, ilustrado na Figura 1.1, formado por três processadores que geram resultados para um quarto que é capaz de eleger majoritariamente qual resposta utilizar. Inicialmente todos os componentes estão operacionais, porém sujeitos a falhas durante uma execução. Assim, o estado $S_{i,j}$ modela que i processadores ($0 \leq i < 4$) e j eleitores majoritários ($0 \leq j \leq 1$) estão operacionais. Quando um componente falha ele pode ser reparado e voltar a funcionar. Considere que apenas um componente pode ser reparado por vez. Quando o eleitor falha, todo o sistema pára de funcionar. O conjunto de proposições atômicas deste problema é

$AP = \{up_i | 0 \leq i < 4\} \cup \{down\}$. A proposição up_0 denota que somente o processador eleitor está operacional, up_1 denota que além do processador eleitor, um outro também está operacional e assim por diante. A proposição $down$ informa que todo o sistema não está funcionando. Uma estrutura de Kripke para este sistema tem os seguintes componentes:

$$
\begin{aligned}
S &= \{S_{i,1} |\, 0 \leq i < 4\} \cup \{S_{0,0}\} \\
I &= \{S_{3,1}\} \\
R &= \{(S_{i,1}, S_{0,0} |\, 0 \leq i < 4)\} \cup \{(S_{0,0}, S_{3,1})\} \cup \\
&\quad \{(S_{i,1}, S_{i,1} |\, 0 \leq i < 4)\} \cup \{(S_{i,1}, S_{i+1,1} |\, 0 \leq i < 3)\} \\
&\quad \cup \{(S_{i+1,1}, S_{i,1} |\, 0 \leq i < 3)\} \\
Label(S_{0,0}) &= \{down\}\ e \\
Label(S_{i,1}) &= \{up_i\}\ para\ 0 \leq i < 4
\end{aligned}
$$

Graficamente, a estrutura de Kripke para esse problema é ilustrado pela Figura 1.2.

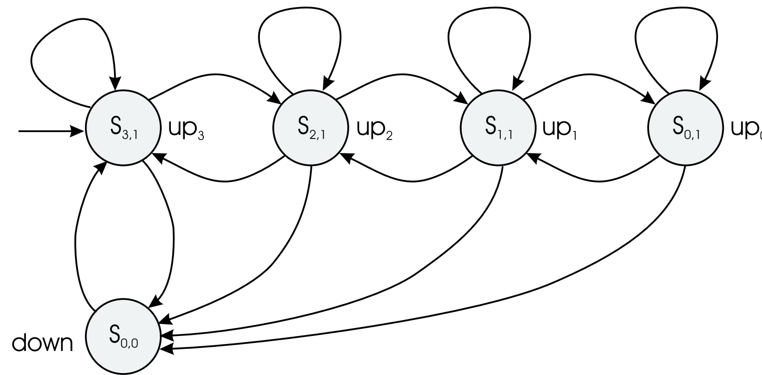


Figura 1.2: Um exemplo de estrutura de Kripke

Para se definir formalmente a semântica de LTL, o conceito de caminho também deve ser formalizado:

Definição 1.3 (Caminho) *Um caminho em* $\mathcal{M}$ *é uma seqüência infinita de estados* $s_0,\ s_1,\ s_2,\ ...$ *tal que* $s_0 \in I$ *e* $(s_i, s_{i+1} \in R$ *para todo* $i \geq 0)$*.*

Portanto, um caminho é uma seqüência infinita de estados que representa uma possível execução do sistema a partir do seu estado inicial. $\sigma[i]$ denota o (i+1)-ésimo estado de σ e

σ^i representa o sufixo de σ obtido pela remoção do(s) i-primeiro(s) estados de σ. A função $Caminhos(s)$ determina todos os possíveis caminhos da estrutura $\mathcal{M}$ que se iniciam no estado s.

Uma vez definida a estrutura na qual LTL é interpretada, sua semântica pode ser então formalmente definida.

Definição 1.4 (Semântica de LTL) *Sejam $p \in AP$ uma proposição atômica, σ caminho infinito e ϕ, ψ fórmulas LTL, a relação "satisfaz", denotada por $\models$, é definida por:*

$\sigma \models p \quad \Leftrightarrow p \in Label(\sigma[0])$

$\sigma \models \neg\phi \quad \Leftrightarrow not(\sigma \models \phi)$

$\sigma \models \phi \wedge \psi \quad \Leftrightarrow (\sigma \models \phi)\ and\ (\sigma \models \psi)$

$\sigma \models X\phi \quad \Leftrightarrow \sigma^1 \models \phi$

$\sigma \models \phi U \psi \quad \Leftrightarrow \exists j \geq 0, (\sigma^j \models \psi\ and (\forall 0 \leq k < j, \sigma^k \models \phi))$

1.1.2 Verifi cação de Modelos usando LTL

O problema da verificação de modelos, dado intuitivamente pela idéia de verificar se uma propriedade é válida para um determinado modelo, pode ser agora definido formalmente da seguinte maneira:

Definição 1.5 (Verificação de modelos usando LTL) *Dado o modelo $\mathcal{M}$ formalmente representado pela estrutura de Kripke $\mathcal{M} = (S, I, R, Label)$ e uma fórmula LTL ϕ:*

$$\mathcal{M} \models \phi\ se\ e\ somente\ se\ \forall s \in I, (\forall\ Caminhos(s), \sigma \models \phi)$$

Em palavras, esta definição diz que a propriedade ϕ vale no modelo $\mathcal{M}$ se e somente se ϕ vale para todos os caminhos que se iniciam em estados iniciais. Em termos de um sistema isto quer dizer que para uma propriedade ser verificada, ela obrigatoriamente tem que valer para todas as possíveis execuções do sistema.

Como a verificação de modelos se aplica a validação de propriedades de sistemas reativos cujo espaço de estados é finito, um verificador de modelos pode ser interpretado como sendo um autômato finito que aceita palavras infinitas, pois a execução de sistemas desta categoria não pára. Um autômato que se comporta deste modo é denominado autômato de *Büchi*. Este tipo de autômato se diferencia de um autômato finito de estados rotulados somente pelo fato

de aceitar palavras infinitas. Sua condição de aceitação só é satisfeita se um estado final (de aceitação) é visitado infinitamente.

As propriedades em LTL também podem ser formuladas através de um autômato de Büchi, mas, por questões de complexidade computacional, da propriedade que se deseja verificar, é construido o autômato de sua negação, ou seja, verifica-se propriamente se uma propriedade indesejada é válida ou não. O motivo de se ter tanto o modelo do sistema quanto as propriedades formalizadas em forma de máquina de estados é transferir a solução do problema para o domínio da teoria dos autômatos. Com esta estratégia o problema se restringe a determinar se a linguagem reconhecida pelo autômato da propriedade é uma sub-linguagem do autômato que modela o sistema. Isto é feito através da operação produto síncrono entre autômatos de palavras infinitas. Se o resultado desta operação é vazio, ou seja, o autômato resultante só reconhece palavras vazias, tem-se que a propriedade se verifica para o modelo.

Resumidamente, esta é uma visão geral do processo de verificação de modelos em LTL. Esta idéia informalmente apresentada implica em várias atividades intermediárias não triviais. O verificador de modelos *SPIN* [Hol97] usa esta abordagem. Como o que se pretende com esta seção é somente fornecer uma visão geral do processo como um todo, detalhes de cada etapa não serão abordados. O detalhamento de cada etapa pode ser encontrado em [Kat99].

1.1.3 Lógica Temporal Ramifi cada

Sintaxe e semântica A lógica temporal linear considera que a cada momento do tempo existe apenas um único estado sucessor e conseqüentemente um único futuro possível. Em meados dos anos 80, Clarke e Emerson [CE81] propuseram uma lógica capaz de considerar diferentes futuros possíveis, através da noção de tempo ramificado. A idéia desta lógica é quantificar as possíveis execuções de um programa através da noção de caminhos que existem no espaço de estados do sistema. Agora as propriedades podem ser avalidas em relação a alguma execução ou então em relação a todas as execuções. Atualmente, existem várias formalizações para lógica temporal ramificada, cada uma com expressividade diferente. Por ser uma lógica utilizada em vários verificadores de modelos [SMV03; ASK03], neste estudo vamos considerar CTL (*Computation Tree Logic*) [CGL92]. A sintaxe

de CTL é dada pela seguinte definição:

Definição 1.6 (Sintaxe de CTL) *Seja AP o conjunto de proposições atômicas, então:*

1. *cada proposição $p \in AP$ é uma fórmula CTL;*
2. *se ϕ e ψ são fórmulas CTL, então $\neg\phi$, $\phi \wedge \psi$, $EX\phi$, $AX\phi$, $E[\phi U\psi]$, $A[\phi U\psi]$, $EG\phi$, $AG\phi$, $EF\phi$ e $AF\phi$ também são.*

Informalmente, $EX\phi$ ($AX\phi$) significa que ϕ deve valer em algum (todo) estado seguinte ao estado atual. $E[\phi U\psi]$ ($A[\phi U\psi]$) significa que, para algum (todo) caminho que se inicia no estado atual, ϕ deve valer até que ψ seja verdadeiro. $EG\phi$ ($AG\phi$) significa que, para algum (todo) caminho que se inicia no estado atual, ϕ deve valer sempre. $EF\phi$ ($AF\phi$) significa que, para algum (todo) caminho que se inicia no estado atual, ϕ deve valer no estado inicial ou em algum estado futuro.

Cada um desses operadores pode ser descrito através dos operadores $EX\phi$, $EG\phi$ e $E[\phi U\psi]$:

- $AX\phi \equiv \neg EX\neg\phi$;
- $EF\phi \equiv E[trueU\phi]$;
- $AG\phi \equiv \neg EF\neg\phi$;
- $AF\phi \equiv \neg EG\neg\phi$;
- $A[\phi U\phi] \equiv \neg E[\neg\psi U(\neg\phi \wedge \neg\psi)] \wedge \neg EG\neg\psi$;

Semanticamente, CTL pode ser definido sobre o mesmo modelo dado pela Definição 1.2, ou seja, através de uma estrutura de Kripke. Porém a relação “satisfaz” agora considera os quantificadores de caminho existencial e universal.

Definição 1.7 (Semântica de CTL) *Sejam AP o conjunto das proposições atômicas, $p \in AP$, o modelo $\mathcal{M}$ formalmente dado pela estrutura de Kripke $\mathcal{M} = (S, R, I, Label)$,*

$s \in S$ e ϕ e ψ fórmulas CTL. A relação satisfaz, denotada por $\models$, é definida por:

$s_0 \models p \quad \Leftrightarrow p \in Label(s_0)$

$s_0 \models \neg\phi \quad \Leftrightarrow s_0 \models \neg\phi$

$s_0 \models \phi \wedge \psi \quad \Leftrightarrow (s_0 \models \phi)\ and\ (s_0 \models \psi)$

$s_0 \models EX\phi \quad \Leftrightarrow$ *para algum estado* t *tal que* $(s_0, t) \in R$, $t \models \phi$

$s_0 \models E[\phi U \psi] \quad \Leftrightarrow \exists\sigma \in Caminhos(s_0),\ (\exists j \geq 0, (\sigma^j \models \psi \wedge (\forall 0 \leq k < j,\ \sigma^k \models \phi))$

$s_0 \models EG\phi \quad \Leftrightarrow \exists\sigma \in Caminhos(s_0)$ *tal que* $\sigma^0, \sigma^1, ..., \sigma^n, \sigma^k,\ 0 \leq k \leq n \wedge (\sigma^n, \sigma^k) \in R, \forall i[0 \leq i \leq n, \sigma^i \models \phi]$

Na Figura 1.3 é apresentada uma estrutura de Kripke e a verificação de algumas fórmulas em CTL. O estado pintado de preto denota que a fórmula é válida nele.

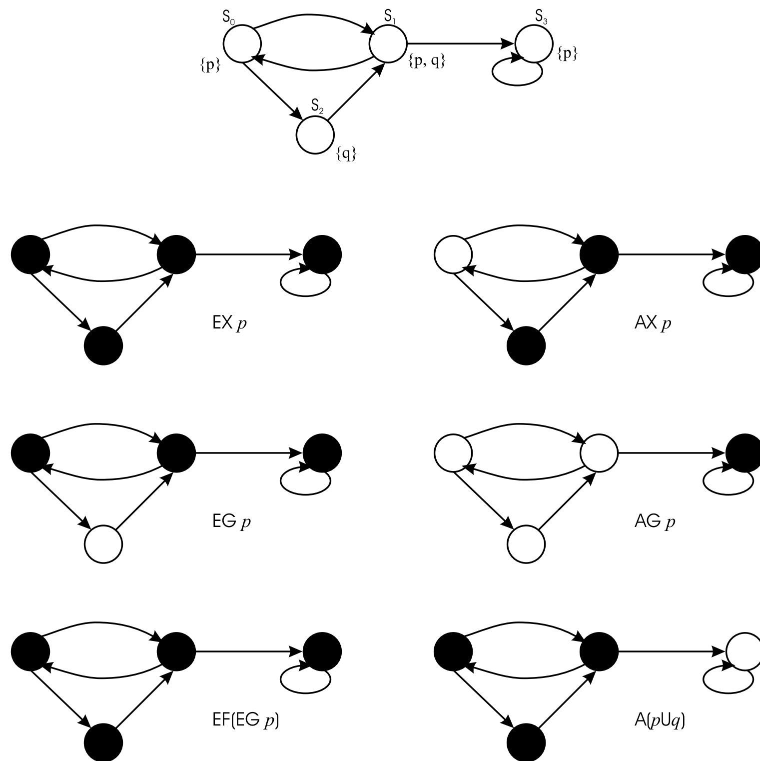


Figura 1.3: Uma estrutura de Kripke e a validação de algumas fórmulas em CTL

1.1.4 Verifi cação de Modelos usando CTL

Para LTL o problema de verificação de modelos foi reduzido a problemas da teoria de autômatos. Embora esta mesma estratégia seja possível para CTL, a verificação de uma fórmula CTL ϕ para um modelo $\mathcal{M}$, se restringe a rotular cada estado s de S com as sub-fórmulas

de ϕ que são válidas para s. O conjunto de sub-fórmulas de ϕ é denotado por $Sub(\phi)$ e indutivamente definido da seguinte maneira:

Definição 1.8 (Sub-fórmulas de uma fórmula CTL) *Sejam AP o conjunto das proposições atômicas, $p \in AP$, ϕ e ψ fórmulas CTL, então:*

$$
\begin{aligned}
Sub(p) &= \{p\} \\
Sub(\neg\phi) &= Sub(\phi) \cup \{\neg\phi\} \\
Sub(\phi \wedge \psi) &= Sub(\phi) \cup Sub(\psi) \cup \{\phi \wedge \psi\} \\
Sub(EX\phi) &= Sub(\phi) \cup \{EX\phi\} \\
Sub(E[\phi U \psi]) &= Sub(\phi) \cup Sub(\psi) \cup \{E[\phi U \psi]\} \\
Sub(EG\phi) &= Sub(\phi) \cup \{EG\phi\}
\end{aligned}
$$

Portanto, o problema da verificação de modelos em CTL para um modelo $\mathcal{M}$ e uma fórmula CTL ϕ pode ser resolvido para qualquer estado s de S através do seu rótulo. Se s for rotulado com ϕ, tem-se que ϕ é verdadeiro no estado s. Formalmente, esta idéia pode ser formulada da seguinte maneira:

$$\mathcal{M}, s \models \phi \;\Leftrightarrow s \text{ é rotulado com } \phi$$

A solução algorítmica para a verificação de uma fórmula CTL ϕ, em um modelo $\mathcal{M}$, consiste em rotular cada estado *s* de *S* com as sub-fórmulas de ϕ que são válidas em *s* [Sif90]. O algoritmo opera de forma iterativa. Na primeira iteração, todas as sub-fórmulas de ϕ de tamanho 1 são avaliadas. Na segunda iteração, todas as sub-fórmulas de tamanho 2 são avaliadas e assim sucessivamente, até a iteração n, onde n é o número de subfórmulas de ϕ. Após a n-ésima iteração, cada estado s do modelo estará rotulado com todas as sub-fórmulas de ϕ que são verdadeiras em s. Seja *label(s)* o conjunto de sub-fórmulas de ϕ que são verdadeiras no estado s, $\mathcal{M}\ s \models \phi \Leftrightarrow \phi \in label(s)$.

A seguir, apresentamos uma subrotina para rotular o espaço de estados de um sistema com a fórmula $f = EU(f_1, f_2)$, denotado por *CheckEU*.

```
1:  procedure CheckEU(f1, f2)
2:      T = {s | f2 ∈ label(s)}
3:      for each s ∈ T do
4:          label(s) = label(s) ∪ {EU[f1, f2]}
5:      while T ≠ ∅ do
6:          choose s ∈ T
7:          T = T \{s}
8:          for each t such that R(t,s) do
9:              if EU[f1, f2] ∉ label(t)  and f1 ∈ label(t)  then
10:                 label(t) = label(t) ∪ {EU[f1, f2]}
11:                 T = T ∪ {t}
12: end
```

Antes da chamada de *CheckEU*, as fórmulas f_1 e f_2 já foram avaliadas. Na linha 2, o algoritmo constrói o conjunto *T* com todo estado *s*, tal que *s* satisfaz a fórmula f_2. Nas linhas 3 e 4, cada estado *s* de *T* é rotulado com *EU*$[f_1, f_2]$ pois *s* satisfaz f_2. O bloco de comandos entre as linhas 5 e 11 vai ser executado enquanto existir estados no conjunto *T*. A idéia é que este bloco percorra os predecessores dos estados contidos em *T* para rotulá-los com *EU*$[f_1, f2]$, caso o predecessor satisfaça f_1. Na linha 6, um estado *s* é escolhido em *T* e, na linha 7, este estado deixa de fazer parte deste conjunto. Na linha 8, cada estado *t* predecessor de *s* é visitado. Na linha 9, se *t* não é rotulado com *EU*$[f_1, f2]$ e é rotulado com f_1, então *t* é rotulado com *EU*$[f_1, f2]$ na linha 10. Na linha 11, *t* é adicionado ao conjunto *T* para que seus predecessores também sejam avaliados. O procedimento *checkEU* tem complexidade $O(|S| + |R|)$

A seguir, apresentamos uma subrotina para rotular o espaço de estados de um sistema com a fórmula $f = EG(f_1)$, denotado por *CheckEG*.

```
1:  procedure CheckEG(f1)
2:      S' = {s | f1 ∈ label(s)}
3:      SCC = {C | C é um SCC não trivial de S'}
```

4: $T = \bigcup_{C \in SCC}\{s \mid s \in C\}$
5: **for each** $s \in T$ **do**
6: label(s) = label(s) $\cup$ {EGf_1}
7: **while** $T \neq \emptyset$ **do**
8: **choose** $s \in T$
9: $T = T \setminus \{s\}$
10: **for each** t **such that** $t \in S'$ **and** R(t,s) **do**
11: **if** EG$f_1 \notin$ label(t) **then**
12: label(t) = label(t) $\cup$ {EGf_1}
13: $T = T \cup \{t\}$
14: **end**

O procedimento *CheckEG* é mais complicado que o procedimento *CheckEU*. Ele se baseia na decomposição do espaço de estados em componentes fortemente conectados não-triviais. Um componente fortemente conectado (SCC, *Strongly Connected Component*) *C* é um subgrafo maximal tal que cada nó em *C* é alcançável a partir de qualquer outro nó em *C* através de um caminho direcionado totalmente contido em *C*. O componente *C* é dito não-trivial se e somente se ele contém mais de um nó ou se contém um nó com auto-laço.

O comando da linha 2 do algoritmo *CheckEG* constrói o conjunto *S'* com todo estado *s* que satisfaz f_1. Na linha 3, ele constrói o conjunto *SCC* com os componentes fortemente conectados não-trivais com os estados do conjunto *S'*. Na linha 4, o conjunto *T* é formado com cada estado *s* de cada componente fortemente conectado de *SCC*. A partir da linha 5, a estratégia para avaliar *EG* é parecida com a utilizada para avaliar *EU*. Uma vez que cada estado *s* de *T* faz parte de um componente fortemente conectado não-trivial, EGf_1 vale em cada estado *s*. Assim, nas linhas 5 e 6, cada estado *s* de *T* é rotulado com EGf_1.

O bloco de comandos entre as linhas 7 e 13 vai ser executado enquanto existir estados no conjunto *T*. A idéia que este bloco percorra os predecessores dos estados contidos em *T* para rotulá-los com EGf_1, caso o predecessor satisfaça f_1. Na linha 8, um estado *s* é escolhido em *T* e, na linha 9, este estado deixa de fazer parte deste conjunto. Na linha 10, cada estado *t* predecessor de *s* que satisfaz f_1, isto é, que está contido em *S'*, é visitado. Nas linhas 11 e 12, se *t* não é rotulado com EGf_1, então *t* é rotulado com com $EG[f_1]$. Na linha

13, *t* é adicionado ao conjunto *T* para que seus predecessores também sejam avaliados. Este procedimento tem complexidade $O(|S| + |R|)$

Para avaliar uma fórmula *f* em CTL, os algoritmos apresentados rotulam cada estado *s* do espaço de estados com as subfórmulas de *f*, iniciando o processo pela menor e mais aninhada fórmula em direção às maiores e mais externa, até chegar em *f*. Desta maneira, uma fórmula *f* só pode ser avaliada se todas as suas subfórmulas já foram avaliadas. Considerando-se que a avaliação de uma subfórmula pode ter complexidade $O(|S| + |R|)$, para se avaliar toda a fórmula a complexidade se torna $O(|f| \times (|S| + |R|))$.

1.1.5 Problema da Explosão do Espaço de Estados

Como mencionado anteriormente, representar o comportamento de uma sistema através da construção de seu espaço de estados nem sempre é factível, mesmo que o número de estados alcançados por ele seja finito. Esta estrutura pode assumir uma dimensão que pode esgotar os recursos de memória de uma máquina. Diante deste problema, vários pesquisadores desenvolveram técnicas para representação mais concisas destas estruturas. Nesta seção, apresentamos uma visão geral de três técnicas: representação simbólica, redução de ordem parcial e interpretação abstrata.

Representação simbólica A aplicação da técnica de representação simbólica para descrever o espaço de estados foi originalmente desenvolvida na tese de doutorado de Ken McMillan [McM92]. Esta técnica fundamenta-se na utilização de diagramas binários de decisão ordenados (*OBDDs*, Ordered Binary Decision Diagrams [Bry86]) para a representação das estruturas de Kripke. *OBDDs* são formas canônicas de se representar fórmulas booleanas que, segundo Clarke e McMillan [JEK+90], são extremamente adequadas para se obter representações concisas de relações sobre domínios finitos.

Para ilustrar a aplicação de *OBDDs* vamos considerar primeiro a representação de fórmulas booleanas. Seja $f(a_1, a_2, b_1, b_2) = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$, uma forma de se representar todas as possíveis avaliações de *f* é através de árvores binárias de decisão, conforme ilustrado na Figura 1.4. Em uma árvore binária de decisão, cada nó não-terminal representa uma variável e cada arco que sai do nó representa o valor booleano assumido pela variável. Cada nó terminal é um valor booleano que representa o resultado da avaliação da fórmula

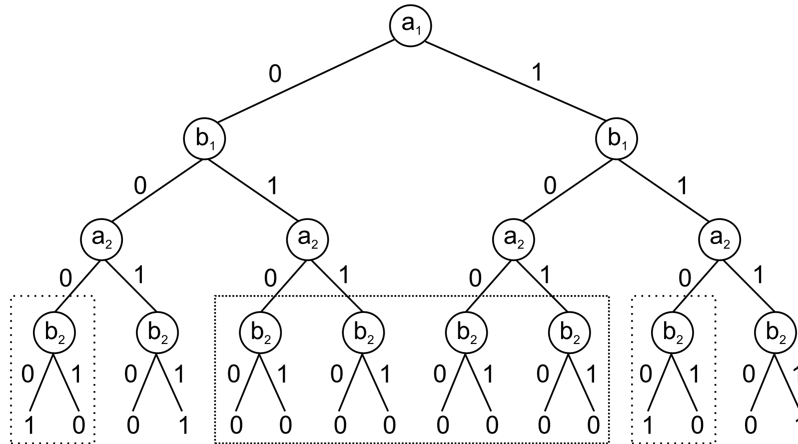


Figura 1.4: Árvore binária de decisão para $f(a_1, a_2, b_1, b_2) = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$

percorrendo-se a árvore da raiz até o nó terminal. Por exemplo, percorrer a árvore pelo ramo $a_1 = 1,\ b_1 = 1,\ a_2 = 0,\ b_2 = 1$ resulta no nó rotulado com 0, isto é, a fórmula f é falsa para estes valores. As árvores binárias de decisão, contudo, não são formas concisas de se representar funções booleanas. Tradicionalmente, eles apresentam várias subárvores redundantes. Por exemplo, na Figura 1.4, existem oito subárvores com a raiz rotulada por b_2, porém, somente três são distintas. Desta forma, a estratégia para se obter *OBDDs* é unificar as subárvores isomórficas. Na Figura 1.5, mostramos o *OBDD* da fórmula f.

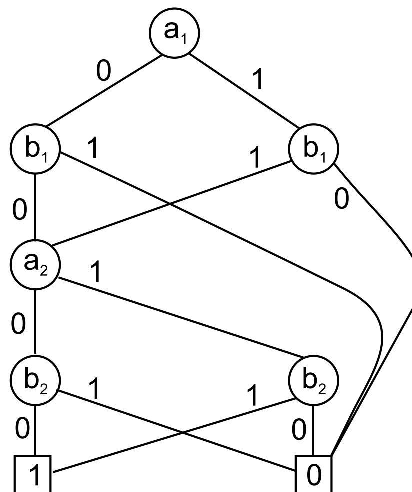


Figura 1.5: Diagrama binário de decisão da fórmula $f(a_1, a_2, b_1, b_2) = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$

Usar representação simbólica em verificação de modelos consiste em registrar os estados e a relações de transição da estrutura de Kripke através de expressões booleanas para depois calcular o OBDD da expressão obtida. Por exemplo, a máquina de estados da Figura 1.6

possui dois estados, s_1 e s_2, e duas variáveis de estado, a e b. Para representar o seu espaço de estados usando representação simbólica, duas variáveis adicionais, a' e b', devem ser definidas para se representar o estado sucessor de cada uma delas. E a mudança do sistema do estado s_1 para s_2 pode ser codificada pela fórmula $(a \wedge b \wedge a' \wedge \neg b')$. Assim, todas transições do sistema são definidas pela fórmula seguinte:

$$(a \wedge b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge b')$$

Cada transição é representada por uma disjunção. Uma vez que esta fórmula descreve o sistema, ela pode ser convertida para um *OBDD* para se obter uma representação concisa. A verificação de modelos sobre este tipo de representação requer algoritmos especiais, diferentes daqueles que apresentamos anteriormente. Uma vez que *OBDDs* representam os estados e transições do sistema, os algoritmos não podem mais operar sobre um estado em individual, eles devem operar sobre todo o *OBDD*. Mais detalhes sobre representação simbólica podem ser encontrados nos trabalhos de McMillan e Clarke [McM93]

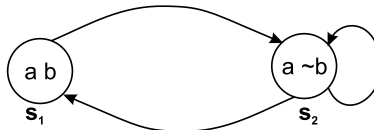


Figura 1.6: Máquina de estados para ilustrar o uso de OBDDs em estruturas de kripkes

Redução de ordem parcial As técnicas que exploram relação de ordem parcial são utilizadas em sistemas concorrentes com semântica de ações intercaladas (*interleaving*). Sistemas desta natureza podem possuir um conjunto de ações que levam a um estado comum, mas as possíveis ordens em que estas ações ocorrem geram estados intermediários que podem ser suprimidos se for escolhida uma única ordem para representar todas as demais. Ou seja, os sistemas são representados por seqüências de ações parcialmente ordenadas. Mais informações sobre redução por relação de ordem parcial podem ser encontradas nos trabalhos de Godefroid [GPS96], Peled [Pel94] e Valmari [Val98]

Interpretação abstrata A interpretação abstrata é uma das principais técnicas para redução do espaço de estados. Tradicionalmente, a abstração pode ocorrer através de abstração

de dados ou por redução por cone de influência. A abstração de dados consiste em definir mapeamentos de valores concretos do sistema para valores abstratos contidos em uma faixa de valores reduzida, mas que preservam as informações para a avaliação da corretude. Na redução por cone de influência a redução ocorre através da eliminação de variáveis que não influenciam as variáveis da especificação. Desta forma as características necessárias a verificação são preservadas e o espaço de estados é reduzido. Mais detalhes sobre interpretação abstrata podem ser obtidos nos trabalhos de Dwyer [DHJ$^+$01], Clarke [CGL94] e Hatcliff [HDZ00].

1.2 Redes de Petri Orientadas a Objetos

Nesta seção apresentamos uma visão informal de RPOO através da modelagem do famoso problema do jantar dos filósofos. Neste jantar, os filósofos se sentam em uma mesa redonda, de forma que entre um filósofo e o seu vizinho existe apenas um garfo. Cada filósofo pode assumir dois estados possíveis: *pensando* e *comendo*. O estado inicial de cada filósofo é pensando. Para passar a comer, cada filósofo deve pegar dois garfos, um que está a sua esquerda e outro que está a sua direita, simultaneamente. Depois de comer, o filósofo devolve os garfos à mesa e volta para o estado pensando. Este problema é comumente utilizado para ilustrar situações em que mais de um processo concorre pela utilização de um recurso.

Se interpretarmos este problema como um sistema distribuído e concorrente, duas entidades independentes podem ser idenficadas: *filósofo* e *garfos*. Em RPOO, as entidades e relacionamentos de um modelo são descritas através de diagramas de classes, assim como em outras metodologias orientadas a objeto. Na Figura 4.1 é apresentado o diagrama de classes da modelagem do problema em questão. Cada retângulo é uma classe e cada arco entre as classes é uma associação. Desta forma, este diagrama registra que cada filósofo deve se associar a dois garfos, um que executa o papel de garfo esquerdo, dado pelo seletor *gesq* e outro que executa o papel de garfo direito, dado pelo seletor *gdir*.

Uma vez registradas as entidades macro de um modelo, através do diagramas de classe, a formalização de RPOO prescreve a construção de diagramas que detalham o comportamento de cada classe. O detalhamento de uma classe em RPOO é definido por redes de Petri colorida [Jen92] e *incrições de interação*. A uma transição da rede, é permitido associar

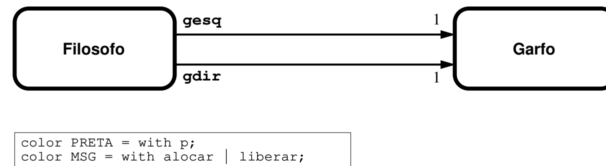


Figura 1.7: O diagrama de classes do jantar dos filósofos

inscrições que descrevem a relação daquela ação com outros objetos que compõem o sistema.

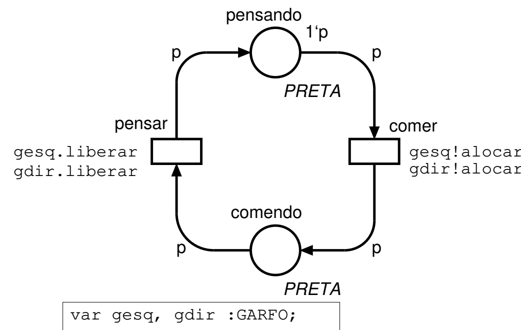


Figura 1.8: O detalhamento classe Filósofo

Na Figura 1.8 é apresentado o detalhamento da classe Filósofo. Nesta rede há dois lugares que modelam os possíves estados de um filósofo – *pensando* e *comendo* – e duas transições que modelam as ações de passar a comer e passar a pensar. Associada a transição *comer* há duas inscrições de interação, *gesq!alocar e gdir!alocar*. O operador *!* destas inscrições significa que estas ações estão condicionadas a uma sincronização com as ações complementares, que ocorrem nos outros objetos da associação. Assim, um filósofo só pode passar a comer quando os objetos referenciados por *gesq* e *gdir* sincronizarem com a recepção da mensagem *alocar*. Como mostrado na Figura 1.9, esta sincronização ocorre mediante o disparo da transição *alocar*, de cada objeto do tipo garfo. O operador *?* da inscrição de interação daquela transição, indica que ela está condicionada pelo recebimento de uma mensagem *alocar*, enviada por um objeto do tipo Filósofo. As inscrições de interação da transição pensar, *gesq.liberar* e *gdir.liberar*, da classe Filósofo, modelam a liberação dos garfos pelo filósofo. O operador . destas inscrições significa o envio assíncrono das mensagens, isto é, significa que a ação de enviar a mensagem não depende da sincronização com ação de recebê-la. O envio da mensagem *liberar* consiste em torná-la disponível para o objeto garfo. A recepção da mensagem, neste caso, vai ocorrer mediante o disparo da transição

liberar, do objeto da classe Garfo, mostrada na Figura 1.9.

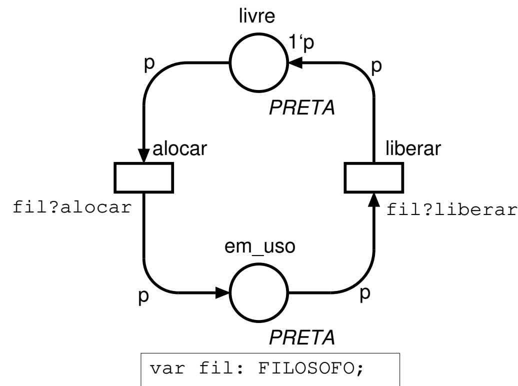


Figura 1.9: O detalhamento classe Garfo

Sistemas de Objetos Até agora, nós definimos o diagrama de classe para descrever as entidades principais que compõem o sistema, seus relacionamentos e definimos também o detalhamento de cada entidade através de redes de Petri. Com estas informações podemos finalmente instanciar o modelo que descreve o sistema em questão. Em RPOO, instanciar um modelo significa definir uma *configuração inicial* para o sistema. Embora o estado inicial de cada entidade seja definido em sua rede de Petri, precisamos especificar quais instâncias e relacionamentos devem existir para representar o comportamento inicial do sistema. No nosso jantar do filósofos, por exemplo, a especificação do problema nos permitir concluir que os filósofos estão sentados em volta de uma mesa redonda e que entre um filósofo e outro deve existir somente um garfo, isto é, que o garfo esquerdo de um deve ser o garfo direito de outro. Não existe regra para especificar quantos filósofos devem existir no jantar. Uma configuração inicial válida, por exemplo, pode ser um jantar composto por dois filósofos e, conseqüentemente, dois garfos. Esta configuração é representanda na Figura 1.10. Ela possui dois objetos da classe filósofo, denotados por *f1* e *f2*, e dois objetos da classe garfo, denotados por *g1* e *g2*. O garfo *g1* esta à esquerda do filósofo *f1* e à direita do filósofo *f2*. Enquanto o garfo *g2* esta à direita do filósofo *f1* e à esquerda do filósofo *f2*.

A partir desta configuração, nós podemos analisar o comportamento do sistema. Podemos observar quais ações podem ocorrer e também calcular o novo estado resultante da execução de uma ação por um objeto. Na configuração inicial que definimos para o nosso exemplo, os dois filósofos podem passar do estado comendo para o estado pensando. Caso

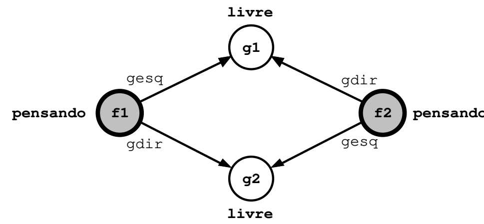


Figura 1.10: O jantar do filósofos com dois filósofos

o filósofo *f2* resolva fazê-lo, temos uma nova configuração em que *f2* está pensando e os garfos *g1* e *g2* estão alocados. Nesta configuração, o filósofo *f1* não pode mais comer, pois não há como ele sincronizar suas ação de comer com a ação de alocação do objetos do tipo garfo. O que pode acontecer é o filósofo *f2* voltar a pensar. Isto significa que, mediante o disparo da transição *pensar*, ele vai enviar de forma assíncrona uma mensagem para cada garfo, notificando que ele não está mais em uso. Nesta configuração, as mensagens enviadas pelo filósofo *f2* estão pendentes para cada garfo. Mediante o disparo da transição *liberar* do garfo *g1*, esta mensagem é consumida e ele se torna livre novamente. Em seguida, mediante o disparo da transição *liberar*, o garfo *g2* também se torna livre novamente e sistema volta para a sua configuração inicial.

Podemos perceber, portanto, que o comportamento do sistema pode ser analisado através das configurações alcançadas pelo mesmo. Uma mudança de configuração é efeito de ações executadas pelos objetos. Em RPOO, são definidos sete tipos de ações e seus respectivos efeitos sobre a configuração de um sistema. Essa parte da formalização é denominada *sistema de objeto*. Existem ações para instanciar novos objetos, para auto-destruição dos mesmos, para criar e destruir ligações, para enviar e receber mensagens, etc. Para cada tipo de ação definida para o sistema de objetos, há um conjunto de regras que pode restringir a mudança de configuração de uma instância do modelo. Um objeto, por exemplo, não pode enviar uma mensagem para um outro objeto que ele não mentém uma ligação. No Apêndice A, apresentamos a formalização dos sistemas de objetos.

Para ilustrar a construção de *espaço de estados* de modelos em RPOO, vamos considerar o nosso jantar partindo da configuração inicial mostrada na Figura 1.10. No início, os dois filósofos estão habilitados a comer, pois cada um está no estado pensando e os garfos à direita e a esquerda de cada um estão disponíveis. Quando o filósofo *f1* decide comer, por exemplo, seu estado passa para comendo. Nesta nova configuração, as ações possíveis no modelo

se restringem a liberação do garfos pelo filósofo *f1*. Embora o filósofo *f2* esteja no estado pensando, ele não pode passar a comer, porque não há como sincronizar as ações de pegar garfo, com os garfos que estão a sua esquerda e a sua direita. Quando o filósofo *f1* decide voltar a pensar, ele envia de forma assíncrona as mensagens para cada um dos garfos. Estas ações significam que o filósofo *f1* voltou a pensar, porém não significam que os garfos estão disponíveis. Desta forma, nesta configuração tem-se a situação em que existem mensagens *liberar* pendentes, prontas para serem recebidas por seus respectivos objetos destinatários. Quando isto ocorrer, os garfos efetivamente estarão disponíveis e o sistema retorna a sua configuração inicial. A Figura 1.11 é representação gráfica do espaço de estados do nosso modelo. A configuração inicial é aquela que aparece mais acima na figura. Arcos marcados com pequenos quadrados denotam mensagens pendentes.

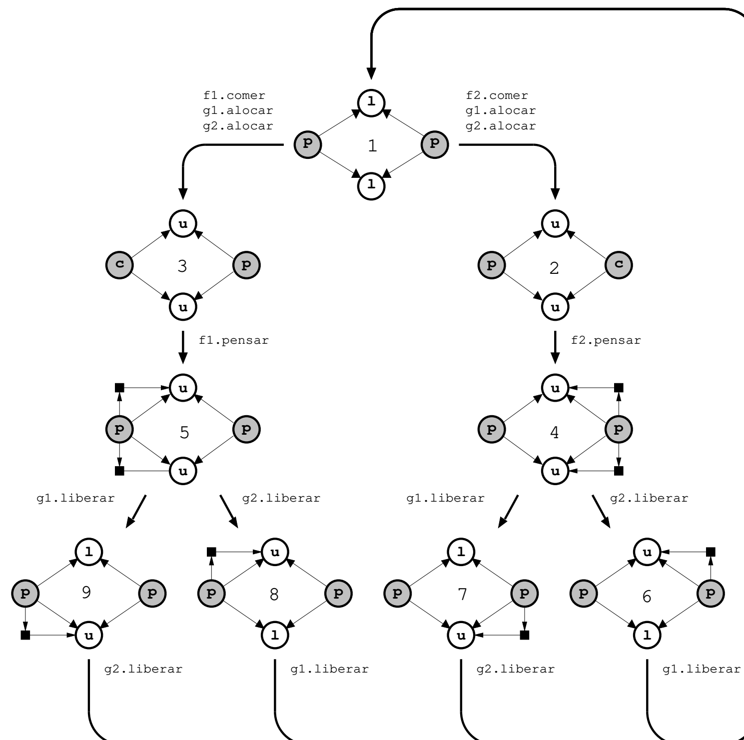


Figura 1.11: O espaço de estados do problema jantar dos filósofos

No Apêndice A, apresentamos formalmente os elementos que compõem a linguagem RPOO. A apresentação se restringe aos elementos necessários à compreensão do trabalho que propomos. A descrição completa da linguagem pode ser encontrada no trabalho de Guerrero [Gue02].

1.3 Considerações finais

Neste capítulo, nos preocupamos em apresentar a fundamentação teórica que serve de embasamento para o trabalho desenvolvido. Na primeira parte, discutimos a verificação de modelos abordando lógicas temporais e soluções algorítmicas para a validação de fórmulas em lógica temporal. Sobre a explosão de espaço de estado, nós apresentamos a idéia básica de três soluções para este problema. Não demoramos mais nesta discussão porque o desenvolvimento de trabalhos teóricos e práticos neste sentido não está no escopo deste trabalho. Atualmente, muitos pesquisadores estão trabalhando para aplicação de verificação de modelos na validação de código em Java [DHJ+01]. Tais trabalhos também não foram considerados neste capítulo porque acreditamos que o potencial de RPOO é contribuir com o desenvolvimento de sistemas baseados em modelos.

Capítulo 2

Verifi cação de Modelos em Redes de Petri Orientadas a Objetos

Neste capítulo, tratamos de pontos importantes para o desenvolvimento da técnica de verificação de modelos para modelagens em RPOO. O primeiro ponto abordado é a arquitetura do verificador de modelos. Nesta parte, apresentamos os módulos previstos na composição do protótipo, discutindo suas funcionalidades e seus relacionamentos. Em seguida, definimos algoritmos para a validação de fórmulas em CTL.

Por último, tratamos do espaço de estados de modelos RPOO. Sobre este assunto, primeiro discutimos as características desejáveis da notação para descrição desta estrutura, em seguida, definimos um formato de referência que tenta contemplar estas características. Ainda sobre espaço de estados, tratamos da sua construção. Como não existe ferramenta própria para esta atividade, apresentamos um procedimento para a construção do espaço de estados usando a ferramenta *Design/CPN* [Jen92].

2.1 Arquitetura do Verificador de Modelos

A arquitetura do verificador de modelos em RPOO está apresentada na Figura 2.1. O módulo principal da ferramenta é o módulo de verificação. Ele é responsável por avaliar fórmulas CTL. Se uma fórmula existencialmente quantificada for avaliada como verdadeira em um modelo, este módulo deve mostrar um caminho no espaço de estados que prova que a fórmula é verdadeira. De forma análoga, se uma fórmula universalmente quantificada for

avaliada como falsa, este módulo deve mostrar um caminho que prova que a fórmula é falsa.

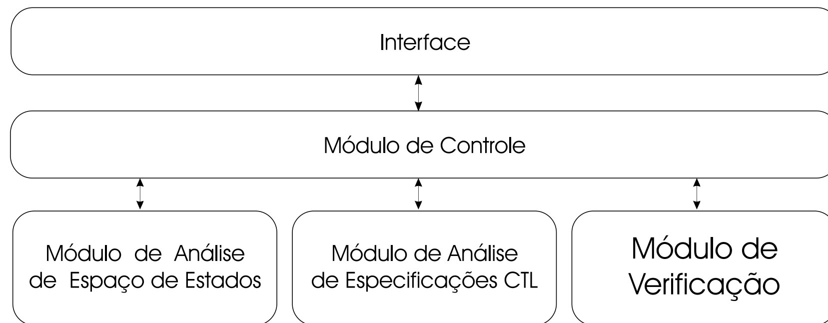


Figura 2.1: A arquitetura do verificador de modelos RPOO

Nossa ferramenta não deve construir o espaço de estados do modelo em RPOO. Nós tomamos esta decisão para desacoplar o verificador de modelos da ferramenta de geração de espaço de estados. Assim, a estrutura que contém o espaço de estados deve ser fornecida como entrada e deve respeitar a gramática definida na Seção 2.4.1. O módulo de análise de espaço de estados é responsável por garantir que a estrutura fornecida é sintaticamente válida segundo a gramática definida. O mesmo cuidado é tomado com as especificações em CTL através do módulo de análise de especificações CTL. O módulo de controle existe para interpretar as solicitações do usuário e coordenar a interação entre os módulos.

2.2 Algoritmos

Para avaliar qualquer fórmula CTL, nós adotamos os seguintes operadores básicos: *AP*, *NOT*, *AND*, *EX*, *EG* e *EU*. A estratégia que utilizamos para avaliação destes operadores é baseada nos trabalhos de Heljanko e Vergauwen [Hel97; VL93].

No Capítulo 1, mostramos os algoritmos de Clarke [CGP98] para avaliar uma fórmula f em CTL. Estes algoritmos primeiro avaliam as subfórmulas de f para em seguida avaliar f. Ao final do processo tem-se f avaliada para todos os estados do espaço de estados. Uma vez que estamos interessado em avaliar f para um estado específico – o estado inicial do sistema, por exemplo –, tais algoritmos podem realizar trabalho desnecessário. Heljanko e Vergauwen desenvolveram uma estratégia para avaliar fórmulas CTL que pode evitar isto. A idéia básica, neste caso, é iniciar a validação de f por um estado específico. O tipo de f é que

Identificador	Descrição
marked	*Array* de booleanos com *l* linhas e *c* colunas, onde *l* é o número de estados do espaço de estados e *c* é tamanho da fórmula CTL a ser avaliada. Cada posição indica se o estado *c* foi visitado pela fórmula *l*. No início, todas as posições armazenam valor *falso*. Ela é utilizada para a detecção de estados que foram visitados.
info	*Array* de booleanos com *l* linhas e *c* colunas, onde *l* é o número de estados do espaço de estados e *c* é tamanho da fórmula CTL a ser avaliada. Cada posição armazena o resultado da avaliação da fórmula indicada por *c* no estado indicado por *l*. No início, todas as posições armazenam valor *falso*.
pi	*Array* de listas de inteiros com *p* posições, onde *p* é o tamanho da fórmula CTL a ser avaliada. Cada posição serve para armazenar os estados que são exemplo ou contra-exemplo da fórmula indicada por cada posição. No início, cada posição contém uma lista vazia.
continue	*Array* de booleanos com *p* posições, onde *p* é o tamanho da fórmula CTL a ser avaliada. Cada posição serve de *flag* para controlar a avaliação da fórmula indicada pelo índice do *array*. No início, cada posição contém valor *verdadeiro*. Quando um caminho que prova uma fórmula cujo índice é *i* for encontrado, *continue*[*i*] vai receber valor *falso*.

Tabela 2.1: Descrição das principais variáveis utilizadas nos algoritmos.

orienta a pesquisa no espaço de estados. Se f é do tipo *EX(g)*, por exemplo, onde g é uma proposição atômica, os estados percorridos serão o estado de referência e os seus sucessores. A estratégia de Heljanko e Vergauwen é mais interessante para a produção de exemplos e contra-exemplos, uma vez que os algoritmos de Clarke não têm esta característica. Justamente por esta facilidade de se prover exemplos e contra-exemplos que nossos algoritmos são baseados nos de Heljanko e Vergauwen.

Antes de iniciar a apresentação dos algoritmos, nas tabelas 2.1 e 2.2 nós apresentamos uma descrição resumida das variáveis e funções utilizadas pelos mesmos.

A seguir, nós apresentamos o algoritmo principal do verificador, denotado por *model_checker*. Antes da chamada deste procedimento, consideramos que o espaço de esta-

Identificador	Descrição
arg1(f)	Função que retorna a primeira subfórmula de *f*.
arg2(f)	Função que retorna a segunda subfórmula de *f*.
size_f(f)	Função que calcula e retorna o tamanho da fórmula *f*.
get_index2(f,i_f)	Função que calcula o índice da segunda subfórmula de *f* em relação ao índice de *f*, dado por *i_f*.
update_pi(s,i_f)	Função que apaga o conteúdo atual de *pi* na posição *i_f* e armazena *s*.
insert_pi(s,i_f)	Função que armazena *s* no iníco da lista contida na posição *i_f* de *pi*.

Tabela 2.2: Descrição das funções auxiliares utilizadas nos algoritmos.

dos do sistema está carregado em memória.

```
1:  proc model_checker(f, s)
2:      i_f = size_f(f)
3:      pi = empty_path(i_f)
4:      continue = new_continue(i_f)
5:      info = new_info(i_f)
6:      marked = new_marked(i_f)
7:      check(f, s, i_f)
8:      print(info[s,  i_f])
9:      print(pi)
10: end
```

O procedimento *model_checker* possui dois parâmetros: uma fórmula f em CTL e o estado s em que f deve ser avaliada. A partir destes parâmetros, preparamos as variáveis globais apresentadas na Tabela 2.1 e delegamos o processo de avaliação de f ao procedimento *check*. Terminada a avaliação, o valor verdade do resultado vai estar contido na variável *info*, na posição dada por s e *i_f*. Se f é uma fórmula existencialmente quantificada e verdadeira no modelo para o estado s, a variável *pi* vai conter um caminho que prova isto, mas se for falsa, *pi* vai conter um valor vazio. Se f é uma fórmula universalmente quantificada e falsa

no modelo, então *pi* vai conter um caminho que prova isto, mas se for verdadeira, *pi* vai conter um valor vazio.

A seguir, apresentamos o algoritmo *check*. Os seus parâmetros são a fórmula *f*, o estado *s* em que *f* deve ser avaliada e o tamanho de *f*, denotado por *i_f*. A finalidade deste procedimento é invocar o módulo adequado à avaliação de *f*. Contudo, como estamos utilizando uma solução recursiva, a primeira coisa que este procedimento faz é verificar se o estado *s* já foi visitado pela fórmula *f*. Quando isto ocorre, a variável *marked*[*s, i_f*] guarda o valor verdadeiro. Isto significa que não precisamos avaliar *f* neste estado, é suficiente retornar o valor contido na variável *info*[*s, i_f*]. Quando *f* é binária, tais como *AND* e *EU*, calculamos os índices de suas subfórmulas antes de invocar seus respectivos módulos para avaliação. Terminado a avaliação de *f*, o valor verdade retornado por *check* é aquele contido na variável *info*[*s, i_f*].

```
1:  proc check(f, s, i_f)
2:      if marked[s, i_f] then   (* s foi visitado anteriormente pela fórmula f?*)
3:          return info[s, i_f]
4:      f_type = formula_type(f)
5:      continue[i_f] := true
6:      if f_type = AP then
7:          check_AP(f, s, i_f)
8:      elseif f_type = NOT then
9:          check_NOT(f, s, i_f)
10:     elseif f_type = AND then
11:         i_f2 = get_index2(f, i_f);
12:         check_AND(f, s, i_f, i_f - 1, i_f2)
13:     elseif f_type = AX then
14:         check_AX(f, s, i_f)
15:     elseif f_type = EU then
16:         i_f2 = get_index2(f, i_f);
17:         check_EU(f, s, i_f, i_f - 1, i_f2)
18:     elseif f_type = EG then
```

```
19:         check_EG(f, s, i_f)
20:     return info[s, i_f]
21: end
```

A seguir, apresentamos o procedimento para avaliar expressões booleanas, denotado por *check_AP*. Este procedimento corresponde ao operador mais básico da lógica CTL. Ele possui três parâmetros: a expressão booleana denotada por f, um estado s e o índice de f. Como estamos interessado em produzir um caminho que prova o valor verdade da avaliação de f, nós guardamos o estado s na variável *pi*, índice *i_f*. Por último, calculamos a expressão booleana f, no estado s e armazenamos o resultado na variável *info*[*s, i_f*].

```
1: proc check_AP(f, s, i_f)
2:     update_pi(s, i_f)
3:     info[s, i_f] = evaluate_f(f, s)
4:     marked[s, i_f] = true
5:     return
6: end
```

Nos algoritmos seguintes, para se obter as subfórmulas que compõem uma fórmula f em CTL, nós utilizamos as funções *arg1(f)* e *arg2(f)*, apresentadas na Tabela 2.1. Agora vamos mostrar o procedimento para avaliar fórmulas com o operador *NOT*, denotado por *check_NOT*. Primeiro, guardamos o estado s na variável *pi* e o marcamos como visitado. Em seguida, avaliamos a subfórmula que compõem f e o complemento do resultado obtido é armazenado na variável *info*[*s, i_f*].

```
1: proc check_NOT(f, s, i_f)
2:     update_pi(s, i_f)
3:     marked[s, i_f] = true
4:     info[s, i_f] = not check(arg1(f), s, i_f - 1)
```

```
5:         return
6:     end
```

A seguir, apresentamos o procedimento para avaliar uma fórmula *f* com o operador *AND*, denotado por *check_AND*. Por se tratar de um operador binário, além da fórmula *f* e do estado *s*, este procedimento recebe como parâmetros os índices das subfórmulas de *f*. Na estrutura condicional da linha 4, avaliamos a primeira subfórmula de *f*. Para evitar trabalho desnecessário, a segunda subfórmula será avaliada somente se a primeira fórmula tiver sido avaliada como verdadeira. Assim, caso as duas subfórmulas tenham sido avaliadas como verdadeira, a variável *info*[*s, i_f*] vai guardar valor verdadeiro.

```
1:     proc check_AND(f, s, i_f, i_f1, i_f2)
2:         update_pi(s, i_f)
3:         marked[s, i_f] = true
4:         if check(arg1(f), s, i_f1) then
5:             info[s, i_f] = check(arg2(f), s, i_f2)
6:         return
7:     end
```

A seguir, apresentamos o procedimento para avaliar uma fórmula *f* com operador *EX*, denotado por *check_EX*. No laço representado na linha 3, visitamos todo estado *t* que é sucessor de *s*. Se em algum estado *t* vale a subfórmula de *f*, temos um exemplo que prova *EX*. Isto significa que devemos atribuir valor verdadeiro à variável *info*[*s, i_f*] e armazenar o estado *s* na variável *pi*[*i_f*]. Note que armazenamos somente o estado *s* em *pi*[*i_f*]. O próximo estado em que a subfórmula de *f* vale vai estar armazenado em *pi*[*i_f - 1*], isto é, no índice referente a sua subfórmula. Assim, todo o caminho que prova *f* é dado pela concatenação dos conteúdos de *pi* nos índice de *f* e de sua subfórmula.

```
1:     proc check_EX(f, s, i_f)
```

```
2:         marked[s, i_f] = true
3:         for each t ∈ successors(s) do
4:             if check(arg1(f), t, i_f) then
5:                 info[s, i_f] = true
6:                 update_pi(s, i_f)
7:                 return
8:         return
9:     end
```

A seguir, apresentamos o procedimento para avaliar uma fórmula com operador *EU*, denotado por *check_EU*. Pelo fato deste operador ser binário, o procedimento recebe como parâmetro, também, os índices de suas subfórmulas, denotados por *i_f1* e *i_f2*. O algoritmo deste operador é mais complexo que os apresentados anteriormente, ele é baseado no axioma que especifica que $EU(f1, f2) \equiv f2 \vee (f1 \wedge EX(EU(f1, f2)))$. A idéia básica da nossa avaliação é realizar uma busca em profundidade no espaço de estados. A partir do estado *s*, nós procuramos por um caminho em que a fórmula *f1* é válida até que a fórmula *f2* seja válida.

A primeira preocupação do nosso algoritmo, denotado por *check_EU*, é saber se o estado *s* foi visitado anteriormente pela fórmula *f*. Neste caso, não precisamos avaliar novamente *f* no estado *s*, basta consultar o valor armazenado na variável *info*[*s, i_f*]. Se esta variável guarda valor verdadeiro, encontramos um caminho em que é possível alcançar um estado em que *EU* vale. Note que segundo os testes das linhas 2 e 3, caso o estado revisitado *s* não faça parte de um caminho em que potencialmente é válido *EU*, nós não fazemos chamadas recursivas de *check_EU* referente à segunda visita.

```
1:     proc check_EU(f, s, pi, i_f, i_f1, i_f2)
2:         if marked[s, i_f] then
3:             if info[s, i_f] then
4:                 continue[i_f] = false
5:         else
```

```
6:          marked[s, i_f] = true
7:          if check(arg2(f), s, pi, i_f2) then
8:              info[s, i_f] = true (* um exemplo foi encontrado *)
9:              insert_pi(s, i_f)
10:             continue[i_f] = false
11:         else
12:             if check(arg1(f), s, pi, i_f1) then
13:                 for each t ∈ successors(s) and continue[i_f] do
14:                     check_EU(f, t, pi, i_f, i_f1, i_f2)
15:                 if not continue[i_f] then
16:                     insert_pi(s, i_f)
17:                     info[s, i_f] = true
18:     return
19: end
```

Na linha 6 do procedimento *check_EU*, marcamos o estado *s* como visitado. De acordo com o axioma apresentado anterioriormente, primeiro devemos saber se o estado *s* satisfaz a segunda subfórmula de *f*. Nós tratamos este caso no bloco entre as linha 7 e 10. Se a avaliação da segunda subfórmula é verdadeira, temos que o estado *s* também satisfaz *f*. Para construir o caminho de estados que prova *EU*, inserimos *s* na variável *pi*[*i_f*]. Em seguida, atribuimos valor falso à variável *continue*[*i_f*] para indicar que não precisamos continuar a pesquisa.

O fato da segunda subfórmula de *f* não valer em *s* não significa que *f* não vale em *s*. O estado *s* ainda pode fazer parte de uma caminho em que *f* é válida. Neste caso, basta que a primeira subfórmula de *f* seja válida em *s*. No bloco entre as linhas 12 e 17, tratamos este caso. Primeiro, pesquisamos recursivamente cada estado *t*, sucessor de *s*, enquanto o caminho que prova *EU* não tiver sido encontrado. Após estas chamadas recursivas, se a variável *continue*[*i_f*] é falsa, significa que *f* vale no estado *t* e, portanto, vale em *s*. Assim, inserimos *s* no início da lista contida na variável *pi*[*i_f*].

Para ilustrar o funcionamento do algoritmo *check_EU*, na Figura 2.2, mostramos a execução passo a passo da avaliação de *EU(p,q)*. Antes do procedimento ser invocado, o sistema

está como representado no item *(a)* e cada item seguinte representa um passo do algoritmo. A seta negra de cada item indica o estado atualmente visitado. Durante a execução do algoritmo, cada estado pode assumir três cores: branca, cinza e preta. A cor branca denota que o estado não foi visitado, a cinza denota que o estado foi visitado e a preta que a fórmula *EU(p,q)* é válida no estado. Em cada item, há também o detalhamento do conteúdo da variável *pi*, destinada a guardar o caminho que prova *EU(p,q)*.

Por último, apresentamos o algoritmo para verificar uma fórmula *f* com operador *EG*, denotado por *check_EG*. A solução que apresentamos é parecida com a que usamos para avaliar *EU* e se baseia no axioma que especifica que $EG(f) \equiv f \wedge (EX(EG(f))$. A nossa primeira preocupação é saber se o estado *s* já foi visitado anteriormente durante a avaliação de *f*. Quando for este o caso, não precisamos avaliar *f* novamente neste estado, basta consultar o resultado armazenado em *info*[*s, i_f*].

Quando o estado *s* é visitado pela primeira vez na avaliação de *f*, nós testamos se *s* satisfaz a subfórmula de *f*. Se satisfaz, supomos que *s* faz parte de um caminho que prova *EG*. Em seguida, fazemos chamadas recursivas à *check_EG* para percorrer cada estado *t*, sucessor de *s*. Depois destas chamadas, se o valor de *continue*[*i_f*] é falso, significa que *f* vale no estado *t* e, portanto, vale em *s*. Assim, inserimos *s* no início da lista contida na variável *pi*[*i_f*] e atribuimos valor verdadeiro a variável *info*[*s, i_f*]. Porém, se o valor de *continue*[*i_f*] é verdadeiro, significa que a suposição de que *s* faz parte de um caminho que prova foi equivocada. Logo, devemos voltar o valor para *info*[*s, i_f*] para falso.

```
1:  proc check_EG(f, s, pi, i_f)
2:      if marked[s, i_f] then
3:          if info[s, i_f] then
4:              continue[i_f] = false
5:              insert_pi(s, i_f)
6:      else
7:          marked[s, i_f] = true
8:          if check(arg1(f), s, pi, i_f - 1) then
9:              info[s, i_f] = true   (* Supomos que EG é válido. *)
10:             for each t ∈ sucessors(s) and continue[i_f] do
```

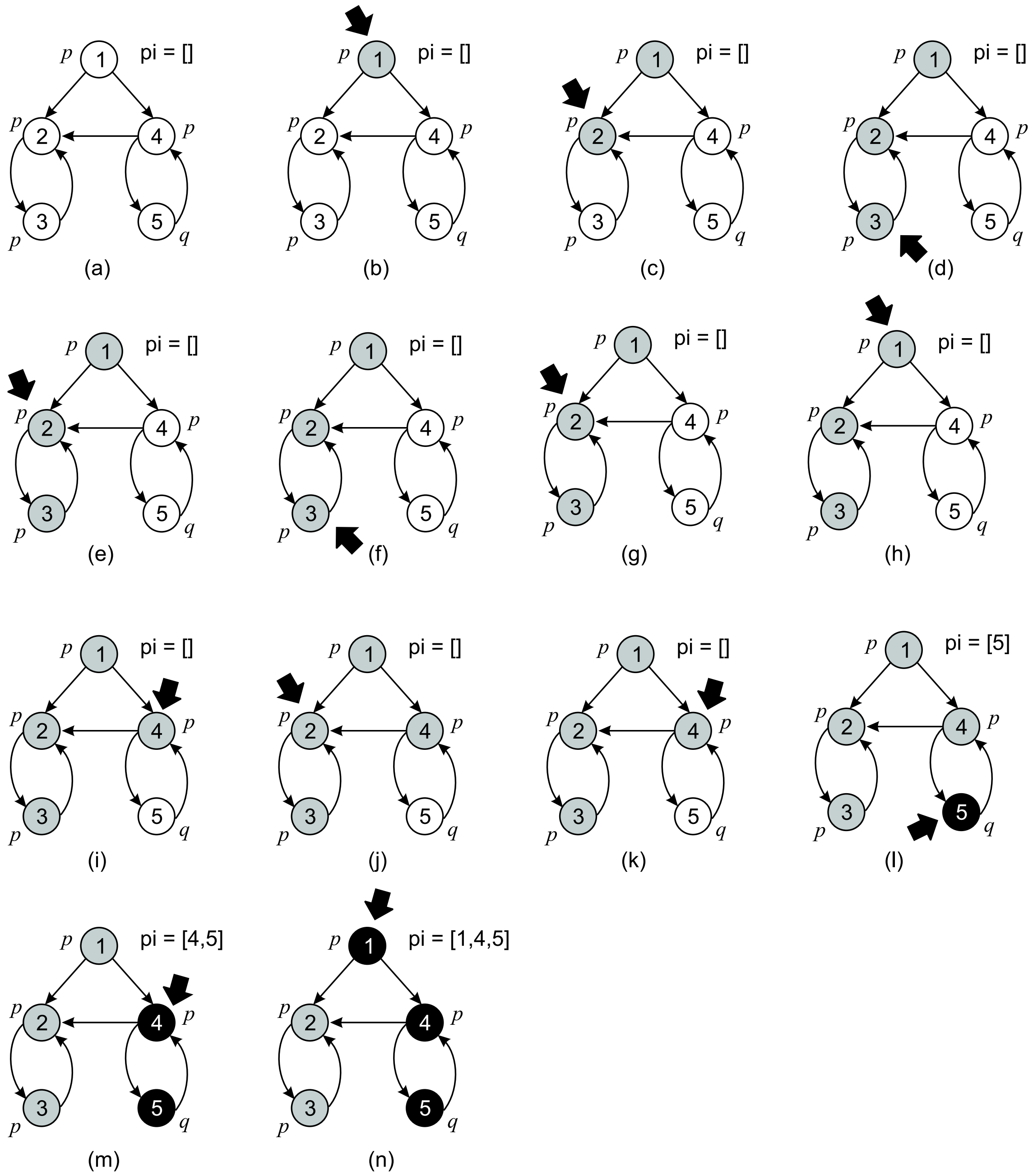


Figura 2.2: Execução passo a passo de *check_EU* para a fórmula *EU(p,q)*

.

```
11:             check_EG(f, t, pi, i_f)
12:         if not continue[i_f] then
13:             insert(s, pi, i_f)
14:         else info[s, i_f] = false  (* A suposição na linha 9 foi equivocada *)
15:     return
16: end
```

Para ilustrar a aplicação do algoritmo *check_EG*, apresentamos a avaliação da fórmula *EG(p)* na Figura 2.2. Antes do procedimento ser invocado, o sistema está como representado no item *(a)* e cada item seguinte representa um passo do algoritmo. A seta negra de cada item indica o estado atualmente visitado. Durante a execução do algoritmo, cada estado pode assumir três cores: branca, cinza e preta. A cor branca denota que o estado não foi visitado, a cinza denota que o estado foi visitado e a preta que a fórmula *EG(p)* é válida no estado. Em cada item, há também o detalhamento do conteúdo da variável *pi*, destinada a guardar o caminho que prova *EG(p)*.

Complexidade dos Algoritmos A avaliação de uma fórmula *f* em um estado s implica, inicialmente, em uma chamada à função *check*. Se *f* não foi avaliada anteriormente no estado *s*, temos que a complexidade de *check* vai ser dada pelos procedimentos para a avaliação de *f*. No caso de *check_EU*, uma vez que s não foi visitado na avaliação *f* este procedimento pode ser chamado para cada estado *t* sucessor de *s*, isto é, $|sucessores(s)|$ vezes, segundo a estrutura de repetição entre as linhas 13 e 14. No pior caso, contudo, estas chamadas podem ocorrer para cada estado *s* do espaço de estados. Este é o caso, por exemplo, em que iniciamos a avaliação pelo estado inicial do sistema e em cada estado do espaço de estados vale a primeira subfórmula sem valer a segunda. Isto significa que para cada estado s nós devemos visitar todos os seus sucessores, sem que o sistema possua sequer um caminho que prove a fórmula *f*. Logo, a complexidade de *check_EU* é $O(|S| + |E|)$, onde $|S|$ é a quantidade de estados do espaço de estados e $|E|$ é a quantidade de eventos.

O procedimento para avaliar uma fórmula *f* com operador *EG*, no estado *s*, é caso análogo. Segundo os comandos das linhas 11 e 12 de *check_EG*, este procedimento pode ser chamado para cada estado *t* sucessor de *s*, isto é, $|sucessores(s)|$ vezes. O pior caso é

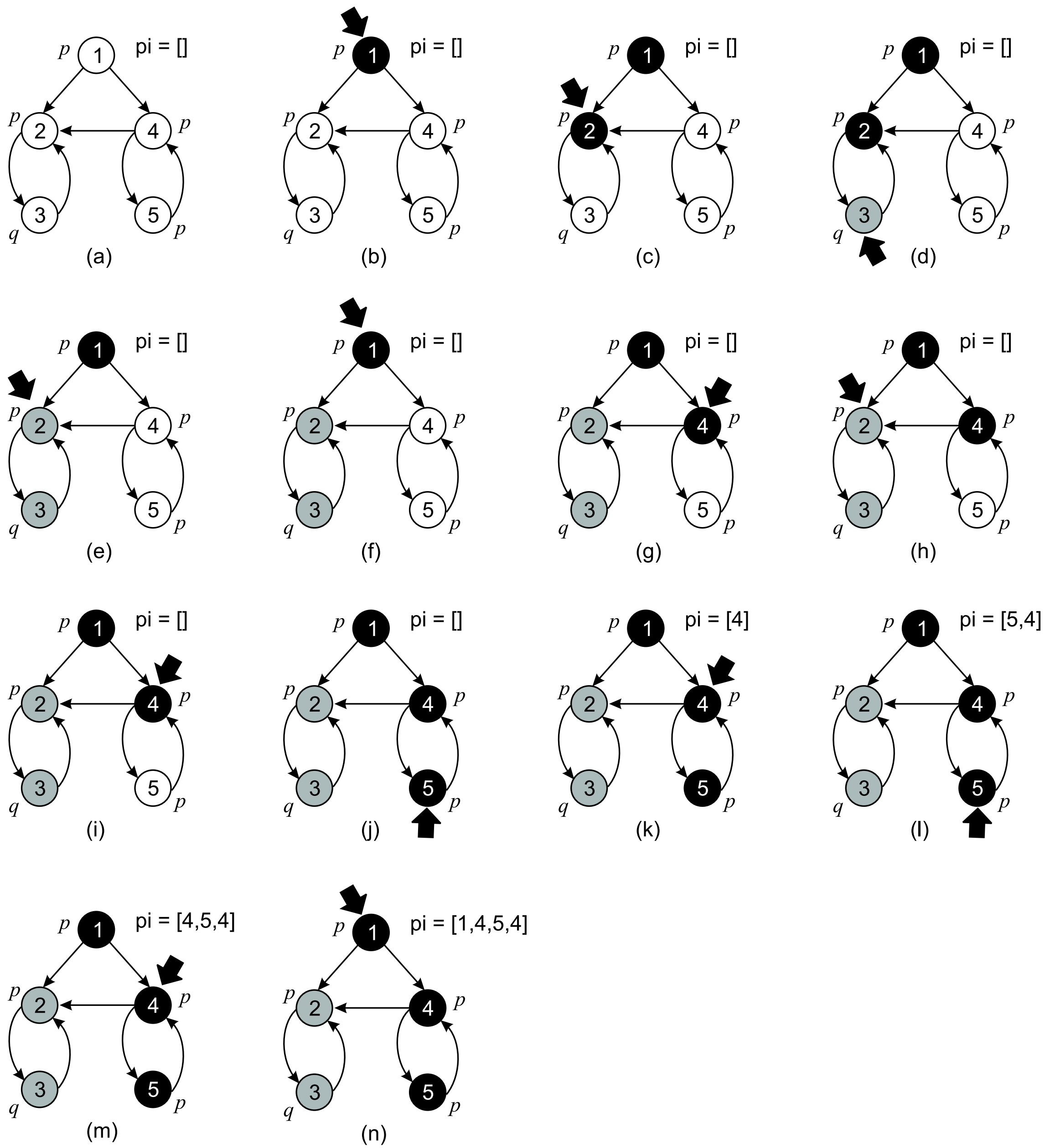


Figura 2.3: Execução passo a passo de *check_EG* para a fórmula *EG(p)*

aquele em devemos visitar cada estado do espaço de estado sem que nele exista um caminho que prove *EG*. Portanto, temos que a complexidade de *check_EG* também é *check_EU* é $O(|S| + |E|)$

Seja $|f|$ o tamanho de uma fórmula *f*, todo o processo de avaliação de *f*, no estado *s*, pode implicar em $|f|$ chamadas ao procedimento *check*, isto é, *check* pode ser invocado para cada subfórmula de *f*. Desta forma, a complexidade dos nossos algoritmos é $(|f| \times (|S| + |E|))$.

2.3 Corretude do Algoritmo

Nesta seção, apresentamos indícios formais de que o algoritmo *check_EU* está correto. Os argumentos que utilizamos se baseiam nos argumentos originalmente utilizados por Clarke em [Sif90]. Para conduzir a prova deste procedimento, nós inserimos uma pilha, denotada por *ST*, que não altera o resultado final do procedimento. Esta pilha serve somente para armazenar os estados visitados pelo algoritmo e desta forma auxiliar a nossa demonstração.

O novo algoritmo para validar fórmulas com operador *EU* é apresentado a seguir. As alterações decorrentes da inclusão da pilha *ST* são as novas linhas 12a e 17a. A linha 12a é responsável por armazenar em *ST* o estado *s* que satisfaz *arg1(f)*. A linha 17a é responsável por desempilhar este estado que foi armazenado. Assim, *ST* é vazia antes da chamada do procedimento *check_EU* e assim permanece após todas as chamadas recursivas deste procedimento.

```
1:   proc check_EU(f, s, pi, i_f, i_f1, i_f2)
2:       if marked[s, i_f] then
3:           if info[s, i_f] then
4:               continue[i_f] = false
5:       else
6:           marked[s, i_f] = true
7:           if check(arg2(f), s, pi, i_f2) then
8:               info[s, i_f] = true (* um exemplo foi encontrado *)
9:               insert_pi(s, i_f)
10:              continue[i_f] = false
```

```
11:          else
12:             if check(arg1(f), s, pi, i_f1) then
12a:               push(s, ST)
13:                for each t ∈ successors(s) and continue[i_f] do
14:                   check_EU(f, t, pi, i_f, i_f1, i_f2)
15:                if not continue[i_f] then
16:                   insert_pi(s, i_f)
17:                   info[s, i_f] = true
17a:               pop(ST)
18:       return
19:  end
```

Seja A o conjunto de asserções sobre o comportamento de *check_EU* dado a seguir:

- **A1:** Os procedimentos para avaliar as subfórmulas de *f* estão corretos.
- **A2:** Os estados que compõem a pilha *ST* formam um caminho no espaço de estados do sistema.
- **A3:** O estado *s* é descendente do estado que está no topo da pilha *ST*.
- **A4:** Para todo estado *r* da pilha *ST*, *r* satisfaz *arg1(f)* e não satisfaz *arg2(f)*.
- **A5:** Para todo estado *r* da pilha *ST*, *r* é marcado como visitado.
- **A6:** Se a fórmula *arg2(f)* é válida no estado *s* , então *f* é válida em *s*.
- **A7:** Se a fórmula *f* é válida no estado *s* , então *s* está contido em *pi*[*i_f*].
- **A8:** Se a fórmula *f* é válida no estado *s* , então *f* é válida em *r*, onde *r* é o estado que está no topo de *ST*.
- **A9:** Se o estado *s* está marcado pela fórmula *f*, então se *info*[*s, i_f*] é verdadeiro, *s* satisfaz *f* e *continue*[*i_f*] é falso.

- **A10:** Se o estado corrente *s* está marcado, então se *info*[*s, i_f*] é falso e *s* não está contido em *ST*, *s* não satisfaz *f* e *continue* é verdadeiro .

- **A11:** Se a fórmula *f* não é válida no estado *s*, então o caminho formado por *s* e os estados que estão em *ST* não é uma exemplo que prova *f*.

- **A12:** Quando o procedimento *check_EU(f, s, pi, i_f, marked, continue)* for invocado pela primeira vez, a pilha *ST* deve estar vazia. Após todas as chamadas a este procedimento, a pilha também deve estar vazia.

Provar *check_EU(f, s, pi, i_f, marked, continue)* significa mostrar que o conjunto de asserções *A* é valido antes e depois da execução deste procedimento. Em uma notação segundo as triplas de Hoare, temos que a hipótese a ser provada é dada por **H1**:

H1 : $\{A\}check_EU(f, s, pi, i_f, marked, continue)\{A \wedge \neg continue[i_f] \wedge s \in pi \leftrightarrow s \models f\}$

Quando a pilha *ST* está vazia temos que:

1. $\forall s\ \{marked[s,\ i_f] \rightarrow \{info[s,\ i_f] \wedge s \in pi \rightarrow s \models f \wedge \neg continue[i_f]\}\}$ (de A9 e A6).

2. $\forall s\ \{marked[s,\ i_f] \rightarrow \{\neg info[s,\ i_f] \wedge s \notin pi \rightarrow s \not\models f \wedge continue[i_f]\}\}$ (de A10, A6 e A12).

A partir de 1 e 2 temos que

$$\forall s\ \{marked[s, i_f] \rightarrow \{\neg continue[i_f] \wedge s \in pi \leftrightarrow s \models f\}\}$$

Supondo que **H1** é verdadeira quando a pilha *ST* possui *n* estados e que **H1** também é verdadeira até que este *n*-ésimo estado seja alcançado, vamos provar que **H1** é verdadeira para a pilha *ST* contendo *n* + 1 estados. Neste cenário, quatro casos podem ocorrer.

C1: O estado corrente *s* satisfaz *right(f)*. Neste caso, para todo estado *r* na pilha *ST*, *r* satisfaz *arg1(f)*. De A2, A3 e A4, temos o caminho *K* formado pelos estados em *ST* e *s*, $K = r_1, r_2, ..., r_n, r_{n+1}, s$ tal que $1 \leq i \leq n+1$, $\forall i[r_i \models arg1(f) \wedge r_i \not\models arg2(f) \wedge s \models arg2(f)]$, em que *EU* vale em cada um dos estados, segundo o Lema 1:

Lema 1: Seja C um caminho no espaço de estados, $C = r_1, r_2, ..., r_n, s\ tal\ que\ 1 \leq i \leq n,\ \forall i[r_i \models arg1(f) \land r_i \not\models arg2(f) \land s \models arg2(f) \rightarrow s \models f \land r_i \models f]$.

C2: O estado corrente *s* foi visitado anteriormente. Neste caso, se *s* já existir na pilha *ST*, *s* não satisfaz *f* segundo o seguinte lema:

Lema 2: Seja C um caminho no espaço de estados, $C = s_1, s_2, ..., s_m, s_k$ tal que $1 \leq k \leq m$ e $\forall i[1 \leq i \leq m \rightarrow s_i \not\models right(f)]$ então C não é exemplo que prova *f*.

Quando isto ocorre, o procedimento simplesmente prossegue em busca de um caminho que prova *f*.

C3: O estado corrente *s* não foi visitado anteriormente e ele satisfaz *arg1(f)*. Isto significa que o algoritmo vai inserir *s* em *ST* (A4), marcar este estado (A5) e prosseguir em busca de um estado em que *right(f)* seja verdadeiro (A3). Neste caso o estado *s* pertence a pilha ST, sem pertencer a *pi* e *continue*[*i_f*] permanece com valor verdadeiro.

C4: O estado corrente *s* não foi visitado anteriormente e ele não satisfaz *arg1(f)*. Isto significa que o algoritmo vai marcar este estado como visita e retroceder para o estado anterior e prosseguir em busca de um caminho que prove *f*.

Nos casos C2, C3 e C4 o algoritmo sempre vai terminar, pois as chamadas recursivas ocorrem somente se um caminho que comprova *f* ainda não foi encontrado e o estado corrente não foi visitado anteriormente pela fórmula *f*. □

Como vimos na apresentação dos algoritmos, para avaliar uma fórmula com operador *EG*, utilizamos uma estratégia semelhante ao caso do operador *EU*. Portanto, a forma como conduzimos a prova formal de *EU* pode ser aplicada para o prodedimento *check_EG*. A diferença básica são as novas asserções sobre o procedimento. Porém, para não tornar a apresentação do capítulo entediante, restringimos nossa apresentação ao caso do procedimento *check_EU*.

2.4 Espaço de Estados em RPOO

2.4.1 Representação Externa do Espaço de Estados

Segundo Jackson [JR00], dois níveis de abstração são utilizados para a descrição de sistemas de software: uma abstração global e uma abstração local. Na abstração global são registradas as entidades principais do modelo e seus relacionamentos através de diagramas de classe. De forma complementar, na abstração local é registrado o comportamento interno de cada entidade, geralmente através de diagramas de transição de estados. Embora estes dois níveis estejam amarrados para a descrição abstrata do sistema, do ponto de vista da análise cada nível possui relevância distinta. Durante a etapa inicial de modelagem, o engenheiro de software está mais interessado em analisar o comportamento global do sistema. Desta forma, um verificador de modelos idealmente deve suportar a especificação de propriedades globais. Segundo Jackson, isto não acontece devido à limitação das linguagens para descrição dos modelos.

Segundo a compreensão deste problema, o nosso verificador de modelos deve operar sobre um espaço de estados que registra o comportamento global do sistema e não sobre o espaço de estados das redes de Petri que detalham cada entidade. Na prática isto significa que ele deve esperar um espaço de estados definido em função das informações descritas pelo sistema de objetos, tais como topologia da rede, troca de mensagens e eventos. A gramática que apresentaremos a seguir caracteriza as informações e o formato que espaços de estados de modelos em RPOO devem possuir:

<*espaço*> ::= <*nó*>$^{+}$

<*nó*> ::= <*topologia*> | <*mensagens*> | <*eventos*> | <*predecessores*>

A seguir, descrevemos detalhadamente cada um dos campos da gramática.

Campo 1: Topologia O primeiro campo de um nó é formado pelo seu identificador único dentro do espaço de estados e pela topologia da rede na configuração. A topologia da rede

descreve os objetos que estão vivos e os seus relacionamentos. O formato do campo 1 é:

$$\langle \textit{identificador} \rangle \ \mathbf{:} \ \langle \textit{topologia} \rangle$$

O identificador deve ser um inteiro maior que zero. A topologia da configuração é representada por sua expressão algébrica (ver [Gue02]), segundo a seguinte gramática[1]

$$<\textit{topologia}> \quad ::= \quad <\textit{objeto}> \ (\mathbf{+} <\textit{objeto}>)^{*} \mid \epsilon$$
$$<\textit{objeto}> \quad ::= \quad \mathbf{id} \ [\ \mathbf{[} \ \mathbf{id}^{*} \ \mathbf{]} \ ] \mid \epsilon \ ]$$

Campo 2: Mensagens O segundo campo de um espaço de estados descreve as mensagens que foram enviadas, mas que ainda não foram consumidas pelos objetos destinários. Uma mensagem é formada pelo identificador do objeto remetente, um separador, o identificador do objeto destinatário e o identificador da mensagem. Como podem existir mensagens repetidas em uma configuração, todo o campo é dado em uma notação de multi-conjunto segundo a seguinte gramática:

$$<\textit{mensagens}> \quad ::= \quad <\textit{msg}> \ (\mathbf{+} <\textit{msg}>)^{*}$$
$$<\textit{msg}> \quad ::= \quad \mathbf{num\ `\ id : id\ (\ id\ \ )}$$

Campo 3: Eventos O terceiro campo de um espaço de estados descreve os eventos que estão habilitados naquela configuração e os nós que são alcançados no caso da ocorrência dos mesmos. Os eventos são separados por "+" e devem ser descritos segundo a seguinte gramática:

[1]Variáveis são denotadas em <*itálico*> e terminais pelo uso de caracteres helvéticos em negrito, como em **id**. O símbolo ϵ é usado para denotar a regra de produção vazia. Os demais símbolos têm o significado convencional: parênteses para agrupamento, colchetes determinam opcionais, "*" e "$^{+}$" indicam repetições.

<*eventos*> ::= <*evento*> (**+** <*evento*>)* | ϵ

<*evento*> ::= <*ação*> (**&** <*ação*>)$^+$ > **id**

<*ação*> ::= **id :** <*ação_elementar*> (**&** <*ação_elementar*>)*

<*ação_elementar*> ::= **id . id** | **id ! id** | **id ? id** | **new id** | **del id** | **end**

Campo 4: predecessores O último campo de cada registro identifica os nós predecessores da configuração. O campo consiste na simples enumeração dos identificadores dos nós, separados por espaços.

Símbolos terminais Os símbolos terminais seguem as convenções usuais. Identificadores (**id**s) consistem em seqüências de letras, números e sublinhas ("_"). Números (**num**s) são apenas números inteiros. Os demais terminais usados nas gramáticas acima devem ser considerados literalmente como as seqüências de caracteres que os compõem (p. exemplo, **new** e **&**).

Exemplo Seja o espaço de estados do modelo do jantar dos filósofos, apresentado na Figura 1.11. O estado 6 desta estrutura fica representado da seguintes forma:

```
6 : f1[g1 g2] + f2[g2 g1] + g1 + g2 | 1‘f2 : g1(largar) | g1 : f2?largar > 1 | 4
```

No estado 6, há dois filósofos, *f1* e *f2*, e ambos estão associados aos garfos *g1* e *g2*, que também estão ativos. Note que neste estado há uma mensagem com conteúdo *largar* pendente. O objeto remetente desta mensagem é o filósofo *f2* e o destinatário é o objeto *g1*. Quando o evento correspondente ao consumo desta mensagem ocorre, o novo estado alcançado é o estado 1. Por último, o predecessor de 6 é o estado 4.

2.4.2 Geração do Espaço de Estados

Embora não exista ferramenta própria para a geração de espaço de estados de modelos em RPOO, este processo pode ser automatizado através da ferramenta para redes de Petri colori-

das *Design CPN*. O princípio básico que garante este processo é a equivalência entre RPOO e redes de Petri coloridas. Em sua tese de doutorado, Guerrero [Gue02, Capítulo 6] mostra que para cada modelo em RPOO é possível construir um modelo equivalente em rede de Petri colorida[2]. O processo de conversão está descrito de forma resumida no Apêndice A.

A partir da rede equivalente, nós podemos gerar o espaço de estados usando a biblioteca para geração de grafo de ocorrência. Porém, a estrutura produzida pela ferramenta é obtida em notação de redes coloridas, sem as características da OO. Desta forma, para completar o processo, devemos converter as informações que estão nesta notação para a notação em RPOO que definimos anteriormente. O procedimento que propomos para automatizar esta atividade, denotado por *espaço_de_estados_RPOO*, é apresentado a seguir. Na ferramenta *Design/CPN*, o espaço de estados é denotado por *grafo de ocorrência*. Para facilitar a compreensão do texto utilizaremos este termo para nos referirmos à estrutura original gerada por esta ferramenta.

```
1:  proc espaço_de_estados_RPOO(GO)
2:      for each nó ∈ GO do
3:          Obter_topologia(nó);
4:          Obter_mensagens(nó);
5:          Obter_predecessores(nó);
6:      for each arco ∈ GO do
7:          Obter_evento(arco);
8:  end
```

O grafo de ocorrência da rede equivalente é isomórfico ao espaço de estados do sistema de objetos. A idéia do procedimento para se obter espaço de estados na notação desejada consiste, portanto, em converter as informações que estão em notação de rede coloridas da rede equivalente para a notação que definimos anteriormente.

Obter a estrutura da rede consiste em analisar as informações que estão representadas como fichas na rede equivalente. Nós fazemos isto através do laço que se inicia na linha

[2] Esta equivalência pode sugerir que é dispensável o desenvolvimento de ferramentas para RPOO. O leitor deve compreender, porém, que esta conversão implica na perda das características da OO.

2. O comando da linha 3 é uma chamada ao procedimento responsável por obter os objetos vivos da configuração e seus relacionamentos. Na prática, o que este procedimento faz é analisar os lugares que controlam os ciclos de vida dos objetos. Abordagem semelhante é utilizada pelo procedimento *Obter_mensagens*, invocado na linha 4. Ele analisa o lugar utilizado para armazenar as mensagens e gera as informações na notação desejada. Na linha 5, nós registramos os nós predecessores.

Precisamos ainda obter as informações sobre os eventos. O fato de que as estruturas são isomórficas não significa somente que cada nó do grafo de ocorrência é um estado alcançável do espaço de estados do modelo RPOO. Significa também que cada arco do grafo de ocorrência é um evento em RPOO. Portanto, para obtermos estas informações, devemos analisar os arcos do grafo de ocorrência. Cada um deles contém informações sobre os eventos, mas em notação de redes coloridas. O laço que se inicia na linha 6 é responsável por obter as informações sobre os eventos na notação desejada e também por obter os identificadores dos respectivos estados alcançados.

2.5 Considerações Finais

Neste capítulo, tratamos de pontos fundamentais para aplicação da técnica de verificação de modelos em RPOO. Tendo como base os trabalhos de Heljanko e Vergauwen [Hel97; VL93], definimos algoritmos para validação de fórmulas CTL. Optamos por não adotar a abordagem tradicional proposta por Clarke [CGP98] porque ela não trata da produção de exemplos e contra-exemplos.

Ao definir uma notação para construção do espaço de estados, levamos em consideração a realidade cotidiana da engenharia de software e enfatizamos a visão OO dos modelos RPOO. A construção do espaço de estados requer suporte ferramental. Como não existe ferramenta RPOO para a construção desta estrutura, propomos a conversão de modelos para CPN equivalente [Gue02, Capítulo 6]. Esta conversão torna possível o uso da ferramenta *Design/CPN*.

Capítulo 3

O Protótipo de Verificação de Modelos

Neste capítulo, tratamos do desenvolvimento do protótipo do verificador de modelos. Na primeira parte, discutimos as principais características do código do verificador, enfocando dois pontos: apresentação do módulo responsável por avaliar fórmulas CTL e apresentação do módulo de suporte à construção de propriedades do modelo.

Na segunda parte, explicamos como utilizar o nosso verificador. Desenvolvemos a explicação através da verificação do modelo do famoso jantar dos filósofos, conforme apresentado na Seção 1.2. Escolhemos exemplo tão simples porque a finalidade deste capítulo é apresentar as características do verificador, com foco no protótipo e não na modelagem. Para ilustrar a aplicação do protótipo em um modelo mais complexo, discutimos validação de um modelo do protocolo IP Móvel no Capitulo 4.

3.1 Implementação

Para implementar os algoritmos apresentados na Seção 2.2, nós utilizamos a linguagem de programação funcional *Moscow ML* [RS95]. Esta decisão foi tomada considerando-se o trabalho futuro de integração do verificador de modelos com outras ferramentas do ambiente RPOO, tais como ferramenta de simulação e geração de espaço de estados. Da arquitetura apresentada na Figura 2.1, o nosso protótipo contempla integralmente os módulos de verificação e de controle. O módulo para tratamento de espaço de estados está parcialmente implementado, contemplando somente a leitura do espaço de estados em arquivo textual.

3.1.1 Módulo de Verifi cação

A seguir, mostramos a assinatura ML do módulo de verificação, denotado por *CTL*. Os operadores CTL são especificados de forma semelhante à biblioteca *ASK-CTL* [CCM96] para *Design CPN*. Cada operador deve ter o tipo *F*. Os valores verdade são representados por *TT* e *FF*. Na linha 7, *AP* é o operador para representar proposições atômicas. Ele deve receber como parâmetro uma função contendo dois parâmetros: um inteiro e um espaço de estados. O segundo parâmetro é o espaço de estados do sistema e o primeiro é o identificador do estado em que a expressão booleana deve ser avaliada. O operador *NOT*, na linha 8, deve ter como argumento um elemento do tipo *F*, isto é, seu argumento também deve ser uma fórmula *F*. Os demais operadores unários, *EX*, *AX*, *EF*, *AF*, *EG* e *AG*, são especificados de forma semelhante ao operador *NOT*. Operadores binários, tais como *AND*, *EU* e *AU*, naturalmente devem receber como parâmetro duas outras fórmulas do tipo *F*.

```
1: signature CTL =
2:  sig
3:      exception invalidCTLFormat
4:      type F
5:      val TT : F
6:      val FF : F
7:      val AP : (int * SS.ss_table- > bool)- > F
8:      val NOT : F- > F
9:      val OR : (F * F)- > F
10:     val AND : (F * F)- > F
11:     val IMP : (F * F)- > F
12:     val EQUIV : (F * F)- > F
13:     val EX : F- > F
14:     val AX : F- > F
15:     val EU : (F * F)- > F
16:     val EF : F- > F
17:     val AU : (F * F)- > F
```

```
18:     val AF : F-> F
19:     val EG : F-> F
20:     val AG : F-> F
21:     val check : int * ss_table * F-> bool * int list
22: end;
```

A especificação *check*, na linha 21, é a função que inicia o processo de avaliação. Ela deve receber como parâmetros um inteiro identificador do estado de referência, um espaço de estados e uma fórmula do tipo *F*. O resultado que ela retorna é uma tupla contendo o valor verdade resultante da avaliação da fórmula passada como parâmetro e uma lista de inteiros que pode conter o caminho que prova o resultado da avaliação da fórmula.

Na estrutura que implementa a assinatura *CTL*, os operadores básicos que definimos para representar qualquer fórmula válida em CTL, isto é, *AP*, *NOT*, *AND*, *EX*, *EG* e *EU*, são definidos como constrututores. Todos os demais são definidos como funções cujo valor de retorno são fórmulas equivalentes, descritas utilizando-se somente os operadores básicos. O operador $AX(f)$, por exemplo, é definido como uma função cujo valor de retorno é $NOT(EX(NOT(f)))$.

Na Figura 3.1.1, mostramos o código que implementa o procedimento *check_eu*. Por se tratar de um paradigma funcional, não foi possível utilizar o conceito de variável global. Assim, as variáveis definidas como globais nos algoritmos são parâmetros da função. O estado *s*, do espaço de estados *t*, é o estado em que vai ser avaliado a fórmula *EU* cujo índice é dado por *i_f*. *f1* e *f2* são as subfórmulas. O parâmetro *pi* é utilizado para guardar o exemplo que prova a fórmula, quando for o caso. A variável *continue* é indicador de que devemos continuar a pesquisa. Quando ele assume valor falso significa que um exemplo foi encontrado.

Em vários trechos do código, usamos funções auxiliares não apresentadas nos algoritmos. A função *sub2* serve para acessar o conteúdo das variáveis *marked* e *info*, definidas como *arrays* booleanos bidimensionais. A função *set2*, por sua vez, serve para atribuir valores a estas variáveis. As funções *sub* e *set* têm as mesmas finalidades, porém, se aplicam a *arrays* unidimensionais.

A estrutura do código não se difere muito da estrutura do algoritmo. Para guardar os sucessores de um estado *s*, utilizamos uma lista de inteiros. Esta decisão nos permitiu a utilização da função *exists*, da biblioteca *List*, para realizar as chamadas recursivas de *check_eu*. Entre as linhas 17 e 20, para cada estado *x* sucessor de *s* será aplicada a função *check_eu*. Se existir um estado sucessor que satisfaça a fórmula em avaliação, o *flag* indicador de que a pesquisa deve prosseguir se tornará falso e não haverá mais chamadas recursivas.

```
 1 fun check_eu(s,t,f1,f2,pi,continue,i_f,marked,info) = if sub2(marked, s, i_f)
 2  then if sub2(info,s,i_f) (* f é subformula  que ja visitou s  *)
 3       then let val u = set(continue,i_f,false)
 4                val u = empty_pi(i_f,pi)
 5            in ignore true
 6            end
 7       else ignore true
 8  else let val u = set2(marked,s,i_f,true) (* Marcamos o estado como visitado *)
 9       in if check'(s,t,f2,pi,continue,i_f-1-size_f(f1),marked,info)
10          then let val u = insert(s,i_f,pi) (* Encontramos um caminho que prova EU *)
11                   val u = set2(info,s,i_f,true)
12                   val u = set(continue,i_f,false)
13               in ignore true
14               end
15          else if check'(s,t,f1,pi,continue,i_f-1,marked,info)
16               then let val u = empty_pi(i_f,pi) (* Nos certificamos que pi esta vazio *)
17     b = List.exists (fn x => let val u = check_eu(x,t,f1,f2,pi,continue,i_f,marked,info)
18                                in not(sub(continue,i_f))
19                                end
20                         ) (SS.getSuc(s,t))
21                      in if not(sub(continue,i_f))
22                         then let val u = set2(info,s,i_f,true)
23                                  val u = insert(s,i_f,pi)
24                              in ignore true
25                              end
26                         else ignore true
27                      end
28               else ignore true (* nada *)
29          end
```

Figura 3.1: Código ML da função para avaliar fórmulas com operador *EU*

3.1.2 Módulo de Suporte à Especifi cação de Propriedades

Para construir as proposições atômicas sobre os estados de modelos RPOO, precisamos de uma linguagem com sintaxe e semântica bem definidas. Para não sobrecarregar o usuário do verificador com a definição de uma nova linguagem, optamos por utilizar a linguagem ML. Assim, a linguagem do verificador é a mesma utilizada para descrever as funções, valores e expressões das redes de Petri que detalham cada entidade. Conseqüentemente, para descrever as propriedades, podemos reutilizar o código originalmente definido no detalhamento das classes.

Para dar suporte à especificação de propriedades, consideramos os elementos básicos que constituem a notação para representação de espaço de estados, apresentada na Seção 2.4.1,

e criamos um conjunto de funções para construção de expressões booleana. Estas funções estão disponíveis ao usuário através da biblioteca *VRTS*. O *funcionamento* do protótipo não depende desta biblioteca. Por enquanto, ela existe para ajudar o usuário no processo de verificação. Quando for o caso, por exemplo, de se desenvolver uma interface gráfica para a ferramenta, esta biblioteca pode ser substituída sem compromenter o funcionamento do verificador.

Na biblioteca *VRTS*, o acesso aos operadores CTL ocorre da mesma forma que na biblioteca *CTL*, exceto pelo operador de proposições atômicas *AP*. Além de uma expressão booleana, agora precisamos de uma descrição textual da proposição atômica. Na apresentação do resultado de uma verificação, esta descrição é utilizada para descrever textualmente a fórmula CTL. Para facilitar a construção de proposições atômicas, esta biblioteca tem as funções apresentadas a seguir:

```
1: signature VRTS =
2:   sig
3:     val getConf : int * SS.ss_table-> SS.ss_node;
4:     val getLiveObj : int * SS.ss_table-> string list;
5:     val getMsgs : int * SS.ss_table-> string list;
6:     val getEvents : int * SS.ss_table-> string list;
7:     val getMsgsObj : string * int * SS.ss_table-> string list;
8:     val getLink : string * int * SS.ss_table-> string list;
9:     val canOccurr : string * int * SS.ss_table-> bool;
10:    val isLiveObj : string * int * SS.ss_table-> bool;
11:    val areLinked : string * string * int * SS.ss_table-> bool;
12:    val isReady : string * int * SS.ss_table-> bool;
13:    val isDeadConf : int * SS.ss_table-> bool;
14:    val load : string-> SS.ss_table;
15:    val check : int * SS.ss_table * CTL.F-> unit;
16: end;
```

A especificação da linha 3 é a função básica desta biblioteca. Ela deve receber como parâmetros um identificador de estado, um espaço de estado e retornar a respectiva estrutura. As demais funções servem para acessar informações específicas da estrutura. Na linha 4, por exemplo, podemos obter somente os objetos que estão ativos em um dado estado. O valor retornado é uma lista de *strings* em que cada elemento é um objeto ativo, sem informações sobre as suas ligações. De forma analóga, na linha 5, podemos extrair todas as mensagens que estão pendentes. Uma mensagem contém o objeto remetente, o objeto destinário e a mensagem propriamente. Como a notação que adotamos faz referência aos eventos que estão habilitados em uma configuração, a função da linha 6, *getEvents*, deve retornar a lista destes eventos. Para saber se um evento específico pode ocorrer, podemos usar função especificada na linha 9, onde o primeiro parâmetro é o evento que queremos consultar.

Como em algumas situações pode ser útil consultar as mensagens que estão pendentes para um objeto específico, a função especificada na linha 7 deve retornar tais mensagens em uma lista de *strings*. O primeiro parâmetro da função é o objeto para o qual a consulta deve ser realizada. Ainda sobre um objeto específico de uma configuração, podemos realizar mais três consultas: *isLiveObj*, *getLink* e *areLinked*. A primeira consulta informa se o objeto passado como parâmetro é um objeto ativo em um determinado estado. A segunda retorna as referências que o objeto passado como parâmetro possui, isto é, informa as suas ligações. A terceira informa se o objeto do primeiro parâmetro está ligado ao objeto identificado pelo segundo parâmetro. Podemos, também, obter informações sobre uma mensagem específica. Através da função *isReady* podemos consultar se a mensagem passada no primeiro parâmetro da função está pendente. Por último, na linha 13, podemos consultar se um estado específico é um estado que não possui pelo menos um estado sucessor, ou seja, se o estado representa um bloqueio no sistema.

Diferentemente da biblioteca *CTL*, a função *check* desta biblioteca não retorna valores. Agora, o resultado da avaliação de uma fórmula é retornado para a saída padrão. Para cada propriedade verificada, os seguintes dados são gerados para a saída padrão: descrição textual da fórmula CTL, estado inicial da validação, valor verdade resultante da validação, possível seqüência de estados que formam um exemplo ou contra-exemplo e o tempo consumido para a validação .

3.2 Exemplo de Verificação

Nesta seção, vamos mostrar a aplicação prática do verificador em uma modelagem RPOO. O modelo que escolhemos para a verificação é o famoso jantar dos filósofos, conforme apresentado na Seção 1.2. Escolhemos exemplo tão simples para conduzir a apresentação porque nosso objetivo agora é apresentar as características do verificador. A utilização de um modelo mais complexo poderia compremeter o desenvolvimento da apresentação.

3.2.1 Propriedades do modelo

Antes de discutir a utilização do verificador, vamos sumarizar as propriedades desejáveis do modelo, isto é, vamos especificar o comportamento que esperamos da modelagem. Tradicionalmente, o problema dos filósofos é utilizado para ilustrar a ocorrência de bloqueios (*deadlock*) e inanição (*starvation*). Contudo, para enfatizar o potencial de verificação de propriedades que são extraídas da visão global do modelo, vamos especificar outras propriedades envolvendo a topologia da rede e trocas de mensagens. As propriedades que vamos verificar são referentes ao estado inicial do sistema e são as seguintes:

1. Em algum momento futuro, o filósofo *f1* pode vir a comer. No estado inicial do sistema, cada filósofo está pensando. Assim, esta propriedade especifica que deve existir pelo menos um caminho de execução em que se alcança um estado no qual o filósofo *f1* está comendo e, conseqüentemente, os garfos à sua esquerda e à sua direita estão alocados. Seja *f1Comendo* a proposição atômica para especificar que o filósofo *f1* está comendo, toda a propriedade em CTL é dada for $P1 = EF(f1Comendo)$.

2. Inevitavelmente, o filósofo *f1* irá comer. Agora estamos especificando que para todo caminho de execução deve ser possível alcançar um estado no qual o filósofo *f1* está comendo. O que muda em relação à propriedade anterior é o quantificador de caminho, que agora é universal. Logo, $P2 = AF(f1Comendo)$.

3. Não importa o que ocorra, o filósofo *f1* sempre pode passar a comer. Esta propriedade especifica que *f1* potencialmente *pode* passar a comer após uma seqüência finita de ações, não importa o estado em que se inicia esta seqüência ações. Em CTL, $P3 = AG(EF(f1Comendo))$.

4. Não importa o que ocorra, o filósofo *f1* sempre passará a comer. Esta propriedade especifica que *f1 sempre* passa a comer após uma seqüência finita de ações, não importa o estado em que se inicia esta seqüência ações. Em CTL, $P4 = AG(AF(f1Comendo))$.

5. O sistema não possui bloqueios. Seja *MarcaçãoMorta* a proposição atômica para especificar que um estado alcançável não possui pelo menos um estado sucessor que não seja ele próprio, toda a propriedade *CTL* é dada por $P5 = AG(NOT(MarcaoMorta))$.

6. O filósofo *f1* sempre se mantém à mesa. Como estamos considerando o filósofo como uma entidade autônoma, para expressar que o mesmo faz parte do sistema, vamos utilizar a proposição atômica *Ativo*. Logo, $P6 = AG(Ativo)$.

7. Os garfos do filósofo *f1* sempre serão *g1* e *g2*. Seja *f1g1g2* a proposição atômica para descrever que o filósofo *f1* está ligado ao seu garfo esquerdo e direito, a propriedade 7 é dada por $P7 = AG(f1g1g2)$.

8. Inevitavelmente, o filósofo *f1* vai liberar o garfo *g1*. Nesta propriedade, especificamos que para todas as execuções, em algum estado futuro, o filósofo *f1* vai enviar uma mensagem para o garfo *g1* se liberar. Seja *liberarG1* a proposição atômica que indica o envio desta mensagem, a propriedade 8 é dado por $P8 = AF(liberarG1)$.

3.2.2 Verifi cação do Modelo

Processo de Verificação A idéia do processo de validação está representado na Figura 3.2. As entradas do processo são dois arquivos: um contendo as propriedades do modelos e outro contendo o espaço de estados. Com estas informações, o verificador gera um relatório para a saída padrão.

Os recursos de software que precisamos para realizar a verificação são os seguintes:

- Ferramenta *Moscow ML*: por enquanto, para que o verificador funcione fora do ambiente RPOO, o código ML que escrevemos para verificação deve ser interpretado pela ferramenta *Moscow ML*[RS95].

Figura 3.2: O processo de verificação de modelos em RPOO

- Biblioteca para tratar espaço de estados: biblioteca dedicada à manipulação de espaços de estados fornecidos como entrada no processo de verificação. Ela é composta pelos arquivos *SS.sig* e *SS.sml*.

- Biblioteca para avaliaçao de fórmulas CTL: biblioteca principal do protótipo, ela é responsável por avaliar as fórmulas que expressam propriedades do sistema e fornecer os respectivos resultados. Ela é composta pelos arquivos *CTL.sig* e *CTL.sml*.

- Biblioteca para construção de expressões booleanas: denotada por *VRTS*, esta biblioteca tem como finalidade prover funções básicas para a especificação de propriedades do sistema.

Especificando propriedades Agora que já sabemos quais propriedades desejamos verificar e também como construir expressões boolenas sobre as informações contidas nos estados alcançáveis, vamos descrever as propriedades do modelo do jantar dos filósofos em um formato esperado pelo verificador. A propriedade 1 faz uso de uma proposição atômica que informa se o filósofo *f1* está comendo ou não. De acordo a nossa modelagem, quando um filósofo está comendo os dois garfos da sua vizinhança estão alocados. Conseqüentemente, podemos identificar que o filósofo *f1* está comendo se ele estiver habilitado a enviar as mensagens *largar* para os garfos *g1* e g2. Logo, a expressão booleana pode ser construída através da função *canOccurr*, da seguinte maneira:

```
fun f1Comendo(cfg, ss) = canOccurr("f1 : g1.largar & f1 : g2.largar", cfg, ss);
```

Para transformá-la em uma proposição atômica CTL, nós usamos o construtor *AP*, da

biblioteca *VRTS*:

```
val f1c = VRTS.AP("f1Comendo", f1Comendo);
```

Toda a propriedade fica assim:

```
val P1 = VRTS.EF(f1c);
```

As propriedades 2, 3 e 4 ficam da seguinte forma:

```
val P2 = VRTS.AF(f1e);
val P3 = VRTS.AG(VRTS.EF(f1e));
val P4 = VRTS.AG(VRTS.AF(f1e));
```

Para descrever a proposição atômica da propriedade 5, a proposição atômica *marcaçãoMorta* pode ser obtida diretamente da função *isDeadConf*, da biblioteca *VRTS*. Logo, toda a fórmula fica da seguinte forma:

```
val P5 = VRTS.AG(VRTS.NOT(VRTS.AP("MarcaçãoMorta", isDeadConf)));
```

Para descrever a propriedade 6, precisamos de uma proposição atômica para expressar que o filósofo *f1* está ativo. A expressão booleana desta proposição pode definida com auxílio da função *isLiveObj*, da biblioteca *VRTS*:

```
fun f1Ativo(cfg, ss) = VRTS.isLiveObj("f1", cfg, ss);
```

Toda a propriedade 6 fica assim:

```
val P6 = VRTS.AG(VRTS.AP("f1 a mesa", f1Ativo));
```

A proposição atômica da propriedade 7 indica se o filósofo *f1* possui as referências dos garfos *g1* e *g2*. Na biblioteca *VRTS*, a função que que determina se um objeto está ligado a outro é denotada por *areLinked*. Como devemos testar se *f1* está ligado com dois outros objetos, a nossa expressão pode ser obtida pela conjunção dos resultados da aplicação de *areLinked* com *f1* e os dois garfos. Assim, a expressão fica da seguinte forma:

```
fun f1g1g2(cfg, ss) = VRTS.areLinked("f1", "g1", cfg, ss) andalso
   VRTS.areLinked("f1", "g2", cfg, ss);
```

Logo, a propriedade 7 pode ser descrita da seguinte forma:

```
val P7 = VRTS.AG(VRTS.AP("f1 conhece g1 e g2", f1g1g2));
```

Por último, na propriedade 8 há uma proposição que testa se a mensagem *largar* está pendente para o garfo *g1*. Podemos expressar esta proposição com auxílio da função *isReady* da biblioteca *VRTS*. Logo, a proposição *liberarG1* vai ser composta pela seguinte função:

```
fun liberarG1(cfg, ss) = VRTS.isReady("1`f1 : g1(largar())", cfg, );
```

A última propriedade pode ser definida da seguinte forma:

```
val P8 = VRTS.AF(VRTS.AP("f1 libera g1", liberarG1));
```

Para verificar cada propriedade que definimos, primeiro precisamos carregar o espaço de estados do sistema. O verificador espera esta estrutura em um arquivo textual em um formato que respeita a gramática definida na Seção 2.4.1. O comando que devemos declarar para carregar o espaço de estado é o seguinte:

```
val t = SS.load("arvore_de_diretorios/f2.eee");
```

Após a execução deste comando, o espaço de estados contido no arquivo *f2.eee* estará guardado no valor *t*. Agora que já preparamos o ambiente, declaramos as propriedades e carregamos o espaço de estados, finalmente podemos declarar os comandos para realizar a verificação propriamente. Conforme a assinatura da biblioteca *CTL*, a verificação ocorre através do comando *check*. Como queremos verificar as propriedades em relação ao estado inicial, denotado pelo valor inteiro 1, o comando para verificar a propriedade 1, por exemplo, fica da seguinte forma:

```
val u = VRTS.check(1,t,P1);
```

Seja *filosofos.sml* o arquivo que contém todos os comandos que decrevemos anteriormente, o proceso de verificação pode ser iniciado através da seguinte linha de comando no sistema operacional:

```
vrts filosofos.sml
```

Relatório da avaliação O resultado da avaliação é um relatório gerado para a saída padrão. Para cada propriedade contida no arquivo de especificações, as seguintes informações são geradas: descrição textual da fórmula, estado inicial da avaliação, resultado da avaliação,

Propriedade	CTL	Resultado	Exemplo ou Contra-exemplo
P1	EF(f1Comendo)	Verdadeiro	1, 3
P2	AF(f1Comendo)	Falso	1, 2, 4, 6, 1
P3	AG(EF(f1Comendo))	Verdadeiro	
P4	AG(AF(f1Comendo))	Falso	1, 2, 4, 6, 1
P5	AG(NOT(Marcacao Morta))	Verdadeiro	
P6	AG(f1 a mesa)	Verdadeiro	
P7	AG(f1g1g2)	Verdadeiro	
P8	AF(liberarG1)	Falso	1, 2, 4, 6, 1

Tabela 3.1: Resultados das avaliações das propriedades

exemplo ou contra-exemplo produzido e tempo consumido para avaliação. Sobre a propriedade 1, por exemplo, as informações geradas são as seguintes:

```
Propriedade em CTL : EF(f1Comendo)
Estado Inicial : 1
Valor Verdade : Verdadeiro
Exemplo ou Contra − exemplo : 1, 3
Tempo(ms) : 0
```

Na Tabela 3.1, mostramos os resultados da avaliação de cada propriedade.

3.3 Desempenho do protótipo

Para analisar o desempenho do verificador, instanciamos o problema do jantar dos filósofos com mesas compostas de 2 até 10 filósofos e verificamos o conjunto de propriedades apresentada na seção anterior. O computador que usamos possui um processador *AMD Athlon(tm) XP 2200+* e 1 GB de memória RAM. A variação do tamanho de cada espaço de estado obtido está apresentado na Figura 3.3. No eixo x do gráfico temos a representação do número de filósofos no jantar e no eixo y temos a quantidade de elementos do espaço de estados, dada pela soma do total de arcos e do total de estados.

Para ilustrar os resultados obtidos na avaliação, apresentamos dois gráficos que relacionam o número de filósofos com o tempo consumido para a avaliação. A Figura 3.4

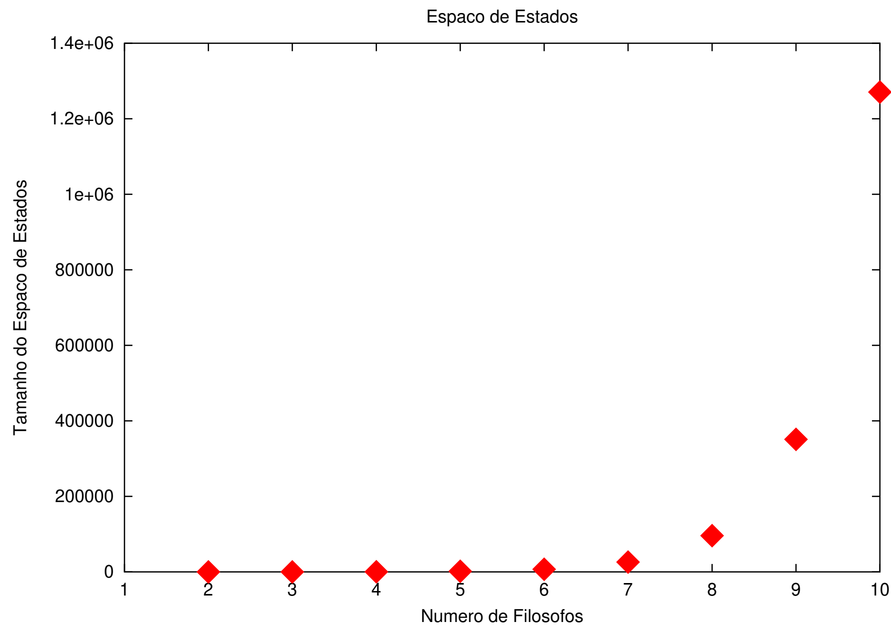


Figura 3.3: Tamanho de espaços de estados do jantar dos filósofos

refere-se à validação da propriedade 3. Para o jantar com 2 filósofos, a validação desta fórmula levou algumas dezenas de milisegundos e para um jantar com 9 filósofos, cerca 25 segundos. Na validação do modelo com 10 filósofos tivemos problemas com falta de memória. Este resultado não nos surpreende pois sabemos que este problema é decorrente do tamanho exorbitante que as estruturas de espaço de estados podem assumir caso não ocorra nenhum tratamento para sua redução.

Na Figura 3.5, temos os resultados obtidos com a propriedade 4. Neste caso não ocorreu o problema da falta de memória. Uma possível justificativa é que está propriedade é falsa no modelo e o contra-exemplo foi encontrado sem ser necessário percorrer todo o espaço de estados.

3.4 Considerações Finais

Neste capítulo, mostramos as características do verificador. Em especial, apresentamos a biblioteca *VRTS*, que provê acesso às funcionalidades do protótipo ao usuário. A linguagem que adotamos para o verificador é ML, ou seja, a mesma linguagem utilizada para descrever funções, tipos e expressões no detalhamento das redes de Petri. Acreditamos que desta maneira o usuário não ficará sobrecarregado com o aprendizado de uma nova linguagem.

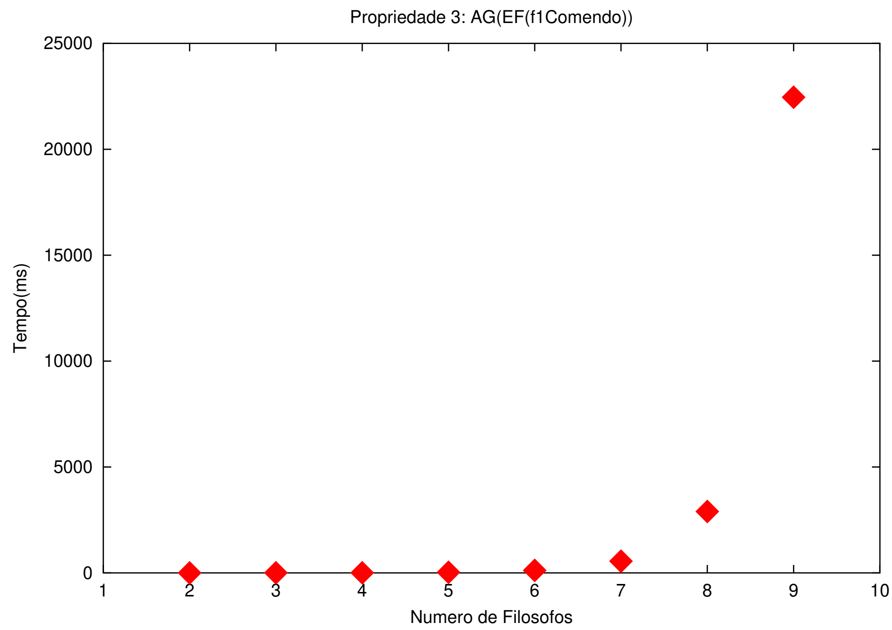


Figura 3.4: Tempo de validação da propriedade 3

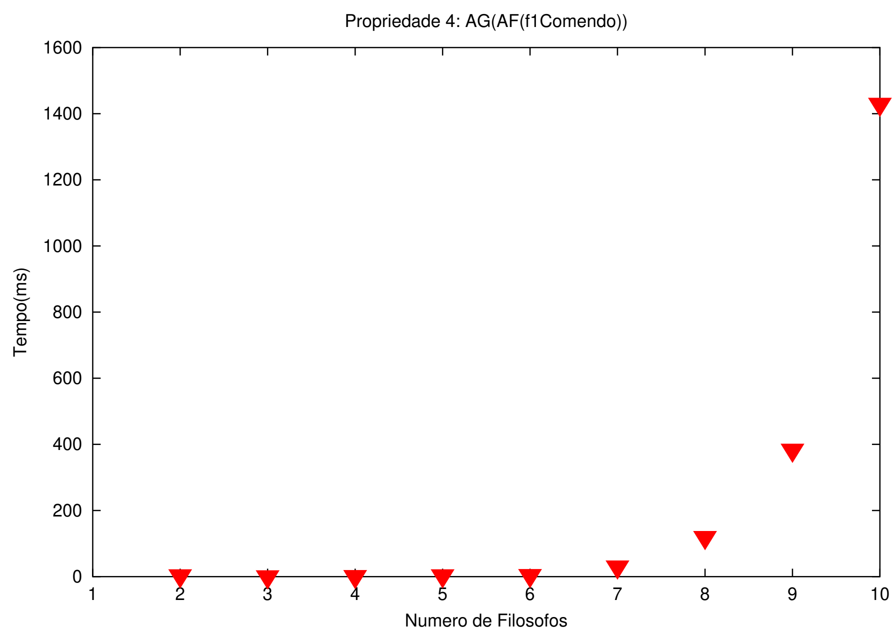


Figura 3.5: Tempo de validação da propriedade 4

Para fornecer indícios do desempenho do protótipo, fizemos experiências com modelagens do problema do filósofo e analisamos o tempo consumido para a verificação de propriedades. Conforme esperado, o principal problema que tivemos foi em relação à manipulação de espaço de estados de tamanhos exorbitantes. Nestes casos, tivemos problema de falta de memória na avaliação de alguma propriedades.

Capítulo 4

Estudo de caso: Validação da Modelagem do Protocolo IP Móvel

Neste capítulo, apresentamos a aplicação do protótipo na modelagem do protocolo IP móvel. Esta modelagem foi realizada como atividade do projeto Móbile [dFGM03]. O objetivo desta aplicação é complementar o processo de validação do modelo, originalmente realizado através de simulação.

Antes de tratar da validação, nós explicamos o funcionamento do protocolo IP móvel ao mesmo tempo em que apresentamos a modelagem em RPOO. Conduzimos a apresentação mostrando as entidades principais do modelo. Como estamos interessados em analisar o comportamento, também discutimos as propriedades desejadas do modelo. Primeiro, analisamos o modelo através de simulação. Para complementar esta atividade, aplicamos a técnica de verificação de modelos através do protótipo desenvolvido. Por último, fazemos a análise do resultados obtidos. Mais detalhes sobre o protocolo podem ser encontrados no trabalhos de Perkins [PM94].

4.1 Apresentação do modelo

O foco da modelagem é a caracterização do suporte à mobilidade prevista pelo protocolo. Segundo este interesse, cinco entidades foram caracterizadas: host móvel, denotado por *Mobile Node*; agente de mobilidade, denotado por *MobilityAgent*; meio físico, denotado por *Medium*; host fixo, denotado por *Correspondent Node* e a internet, denotada por *Internet*.

Na Figura 4.1, definimos como estas entidades se relacionam. A entidade responsável por garantir o endereçamento correto de pacotes para os hosts móveis é o agente de mobilidade. Assim, cada host móvel deve ser servido por um agente de mobilidade. Quando um host móvel está em sua rede local ele é servido pelo agente de mobilidade nativo e quando ele se move para uma outra rede ele é servido por um agente de mobilidade estrangeiro. Desta forma, no diagrama apresentado na Figura 4.1 a classe *Mobility Agent* é uma classe abstrata que pode ser especializada como *Foreign Agent* ou como *Home Agent*. A classe *Medium* representa o meio físico através do qual esta mobilidade ocorre.

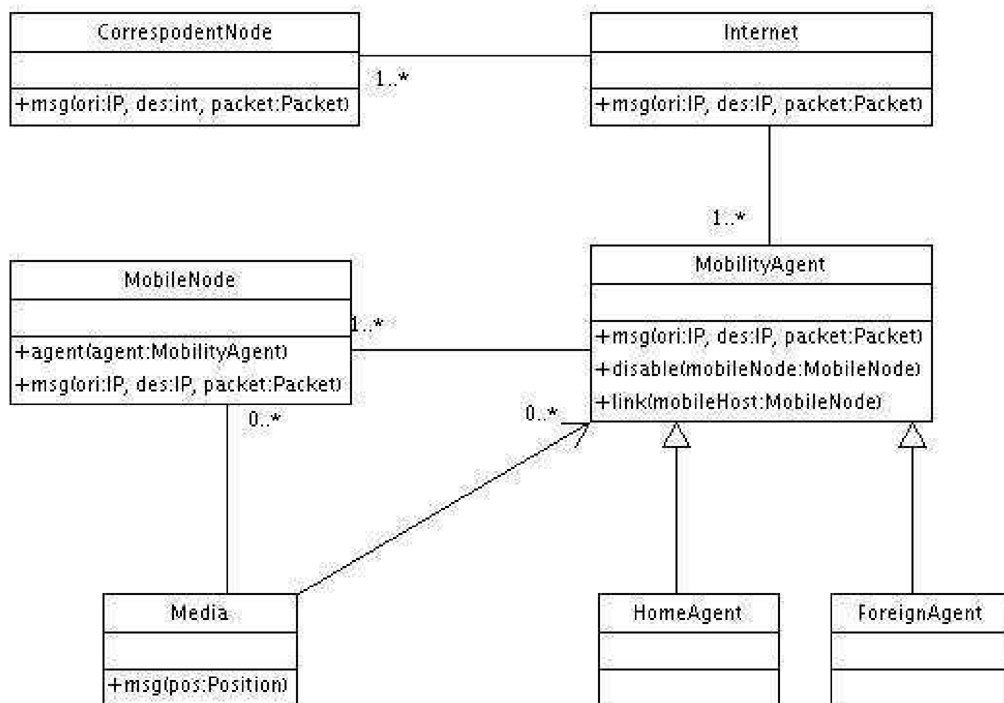


Figura 4.1: Diagrama de classes do protocolo IP Móvel

Na Figura 4.2, apresentamos o detalhamento da classe *Home Agent*. A rede possui três lugares: *Input Buffer*, *Mobile Node* e *Tunnel*. O lugar *Mobile Node* é utilizado para guardar as referências dos hosts móveis que o agente nativo deve servir. Se um desses hosts muda de rede, sua referência é retirada deste lugar mediante o disparo da transição *Disconnect*. O objeto do tipo *Medium* é quem notifica o agente nativo de que o host móvel se mudou para uma rede estrangeira. Em termos de sistema de objetos isto implica em uma ação de desligamento, através da ação *unlink*. Para que as mensagens cheguem corretamente a este

host que não está mais em sua rede nativa, o agente nativo deve armazenar o endereço do host móvel na rede estrangeira. Isto ocorre mediante o disparo da transição *Rcv Reg Request*. O lugar *Input Buffer* armazena temporariamente as mensagens que chegam da internet. Se o host móvel que deve receber a mensagem armazenada estiver em sua rede nativa, o agente nativo a encaminha para o host móvel através do disparo da transição *Delivering*. Se o host móvel estiver em uma rede estrangeira, a mensagem é encaminhada através da transição *Tunneling*.

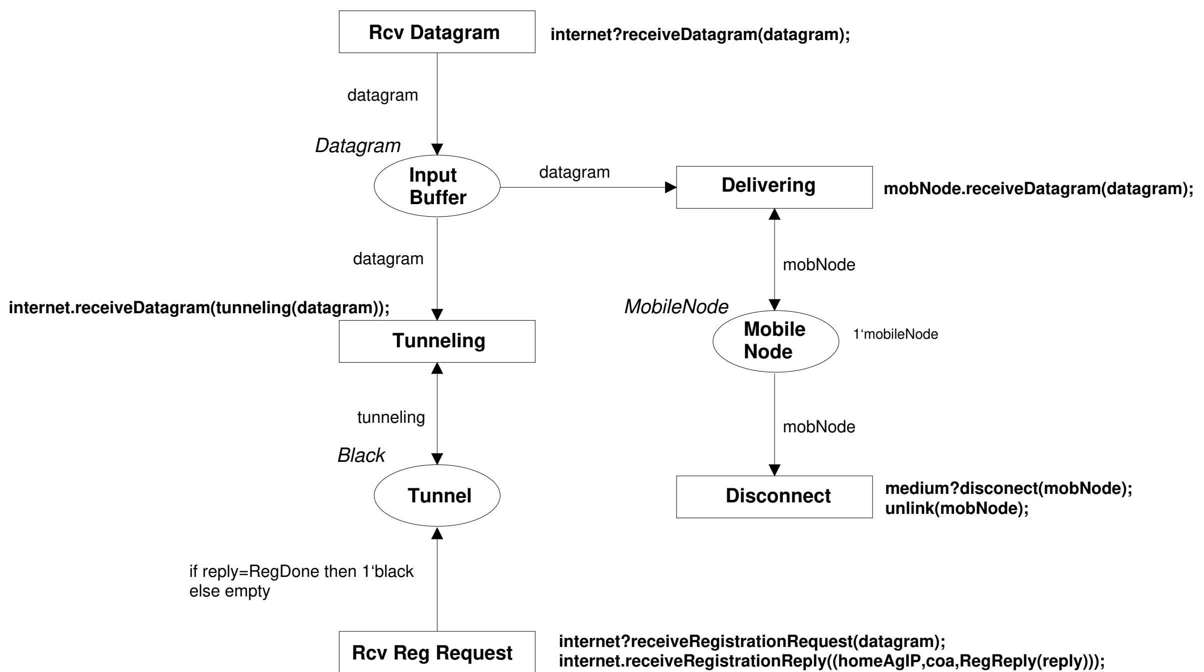


Figura 4.2: Detalhamento da classe HomeAgent

Na Figura 4.3, apresentamos o detalhamento da classe *ForeignAgent*. A rede possui dois lugares: *DtgrmsToMobNode* e *Visitors*. O lugar *Visitors* armazena referências para os hosts móveis que ele deve servir. Quem notifica ao agente estrangeiro a inclusão de um host móvel visitante é o objeto do tipo *Medium* (ver Figura 4.6). Para isto ocorrer este objeto deve enviar uma mensagem *addVisitor(mobNode)* para o agente estrangeiro, onde *mobNode* é uma referência para o host visitante. Quando este envio ocorre, a inclusão por parte do agente estrangeiro ocorre através do disparo da transição *Receive Visitor*. Uma vez que o agente armazena uma referência para o host móvel, ele pode notificá-lo de que está sendo servido por um agente estrangeiro. Esta notificação ocorre através do disparo da transição *Send Agent Advertisement*. Para que as mensagens sejam endereçadas corretamente, o host móvel deve

solicitar uma atualização de seu registro de *COA* a sua rede nativa. Esta solicitação ocorre por intermédio do agente estrangeiro através do disparo da transição *Forward Reg Request*. Ainda sobre este processo de atualização de *COA*, o agente estrangeiro encaminha a resposta do pedido de registro através da transição *Forward Reg Reply*. As mensagens de dados chegam para o agente estrangeiro através do disparo da transição *Receive Gen Datagrams* e são armazenadas no lugar *DtgrmsToMobNode*. Elas são encaminhadas para host destinatário mediante o disparo da transição *Deliver to Mob Node*.

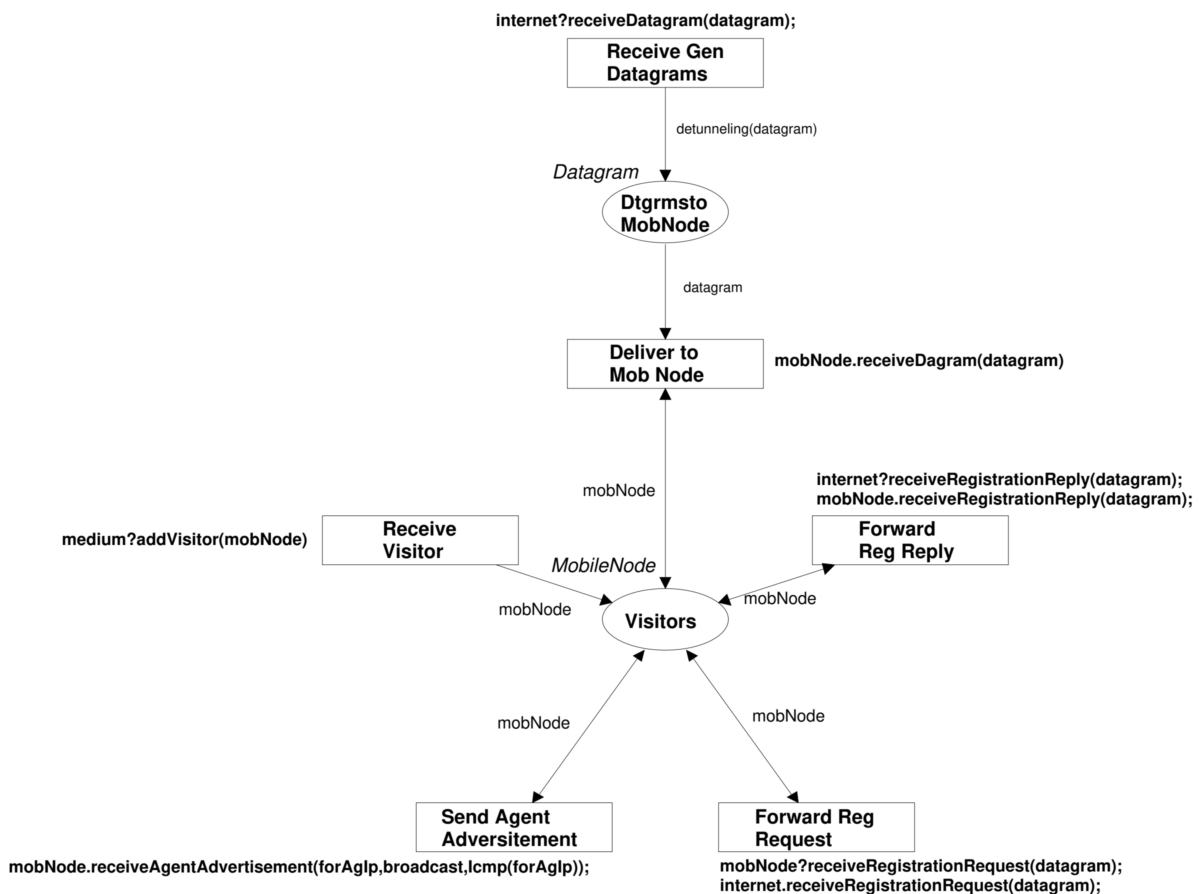


Figura 4.3: Detalhamento da classe ForeignAgent

Na Figura 4.4, apresentamos o detalhamento da classe *Mobile Node*. A rede possui quatro lugares: *Mobility Agent*, *Status*, *InputBuffer1* e *InputBuffer2*. O lugar *Mobility Agent* armazena uma referência para o agente de mobilidade que o serve. Quando ele muda de rede a atualização do agente ocorre através do disparo da transição *Change Agent*. Ele é notificado da mudança mediante o disparo da transição *Reg Request*. O lugar *Status* serve para controlar o estado da host móvel. Seus possíveis valores são *InHomeNetwork* para indicar que ele

está na sua rede nativa, *WaitingReply* para indicar que ele está em um estado intermediário, aguardando a resposta de pedido de registro e *RegDone* para indicar que ele está em rede estrangeira com registro de *COA* realizado. A resposta de pedido de registro é consumida através do disparo da transição *Receive Reg Reply*. Os lugares *InputBuffer1* e *InputBuffer2* servem para guardar as mensagens de dados recebidas. *InputBuffer1* armazena as mensagens de dados que chegam para ele, quando ele está em sua rede nativa. *InputBuffer2* armazena as mensagens que chegam para ele, quando ele está em uma rede estrangeira.

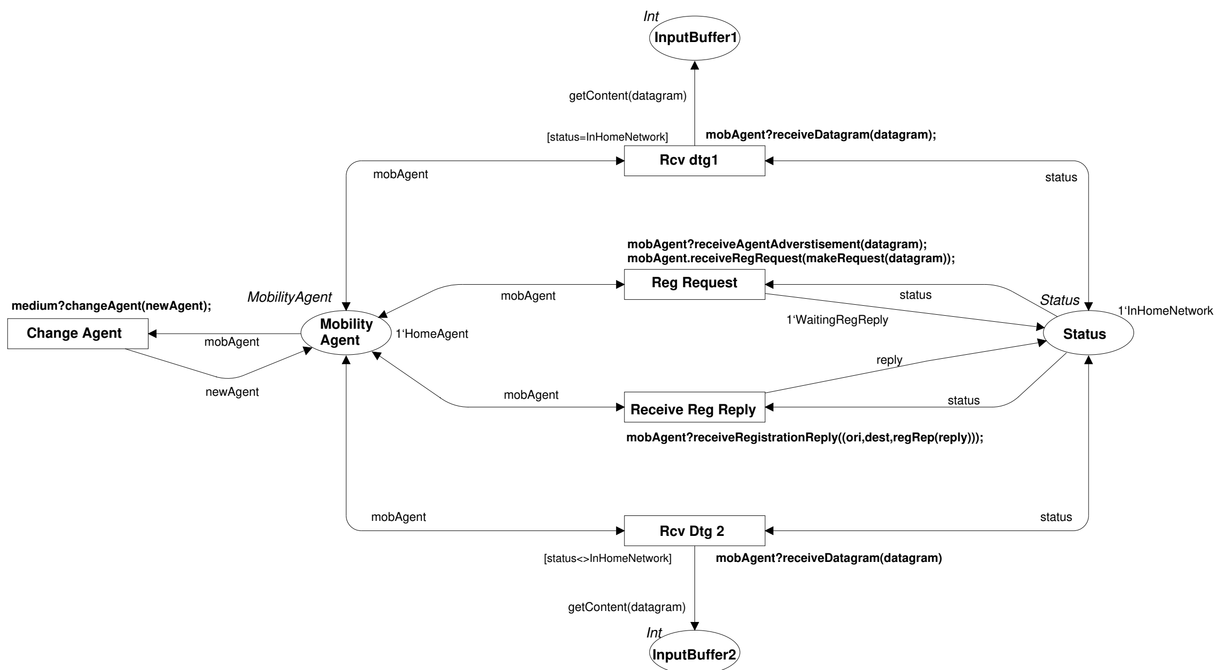


Figura 4.4: Detalhamento da classe MobileNode

Na Figura 4.5, apresentamos o detalhamento da classe *Internet*. Esta rede modela de forma simplificada a rede internet. Basicamente, ela recebe mensagens e as encaminha para seus respectivos destinatários. Por exemplo, através da transição *Datag. From Corr. Node* ela recebe as mensagens de dados do host fixo e as armazena no lugar *General Datagrams*. A partir deste lugar, elas são encaminhadas para o agente nativo através da transição *Forward Datagram*.

Por último, nós aprensentamos o detalhamento da classe *Medium* na Figura 4.6. Esta entidade representa o meio físico através do qual um host pode se mover. De forma simplificada, esta rede de Petri modela a mudança de rede através do disparo da transição *Network changing*. Para esta transição ocorrer, ela deve ser sincronizada com o recebimento da men-

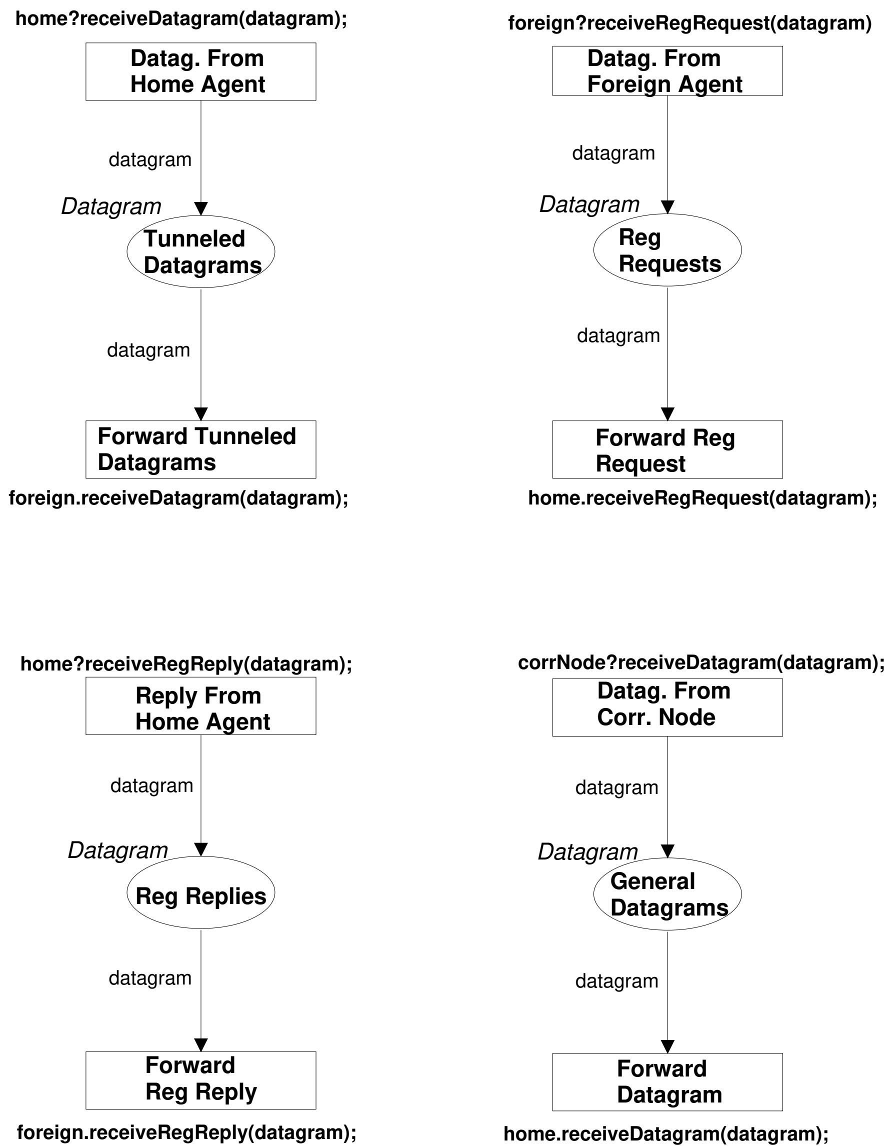


Figura 4.5: Detalhamento da classe Internet

sagem *disconnect(mobNode)* por parte agente estrangeiro, com o recebimento da mensagem *addVisitor(mobNode)* por parte do agente estrangeiro e com o recebimento da mensagem *changeAgent(foreignAgent)* por parte host móvel.

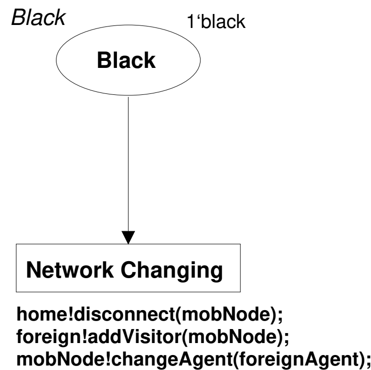


Figura 4.6: Detalhamento da classe Medium

4.2 Simulação

Ferramentas Embora não exista uma ferramenta completa para simulação de modelos em RPOO, é possível realizar esta atividade por meio de um processo semi-automático. Como o comportamento de um modelo em RPOO é definido em termos de um sistema de objetos e de redes de Petri, a simulação pode ser obtida pela coordenação da simulação de cada uma dessas visões. Para simular as redes de Petri do protocolo IP móvel utilizou-se a ferramenta *Design/CPN*[Jen92] para redes de Petri coloridas. Para simular o sistema de objetos utilizou-se a ferramenta de simulação de sistema de objetos, SSO [San03]. Para representar as simulações foi utilizada uma notação baseada em diagramas de seqüência.

Estratégia de validação A estratégia de validação consiste em construir diagramas de sequências a partir de cenários pré-definidos e verificar se eles satisfazem a especificação do protocolo IP móvel. Cada cenário parte de uma configuração inicial contendo os seguintes objetos:

- 1 host fixo, denotado por *cn*.
- 1 agente estrangeiro, denotado por *ha*.

- 1 agente nativo, denotado por *fa*.
- 1 agente móvel, denotado por *mn*.
- 1 internet, denotada por *net*.
- 1 meio físico, denotado por *m*.

Graficamente, esta estrutura está representada na Figura 4.7. Em notação algébrica, a estrutura da configuração inicial é:

$$\begin{aligned} E_i \;=\; & net[ha\ \ fa\ \ cn]\ + \\ & ha[net\ \ mn]\ + \\ & fa[net]\ + \\ & mn[ha]\ + \\ & m[ha\ \ fa\ \ mn]\ + \\ & cn[net] \end{aligned}$$

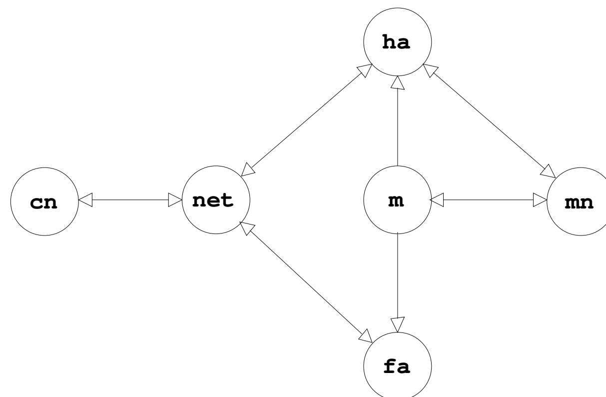


Figura 4.7: Configuração inicial do protocolo IP móvel

Os seguintes cenários foram investigados:

Cenário 1 O host fixo transmite pacotes para o host móvel que está em sua rede nativa. A seqüência é válida quando o pacote é entregue ao host móvel pelo agente nativo. A seqüência em que este cenário ocorre é apresentado na Figura 4.8.

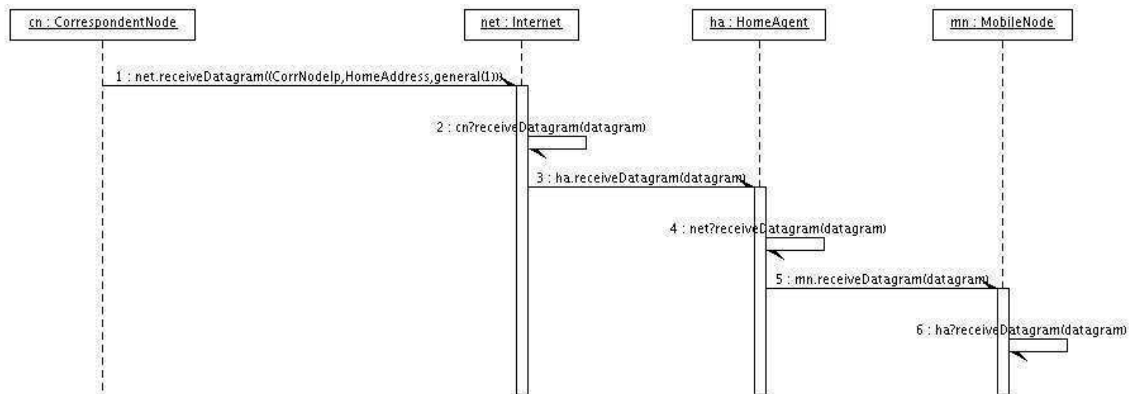


Figura 4.8: Diagrama de seqüência do cenário 1

Cenário 2 O host móvel se muda da rede nativa para a rede estrangeira. O ponto de partida deste processo é a mudança do agente de mobilidade, induzida pelo objeto meio físico. A seqüência em que este cenário ocorre é apresentado na Figura 4.9.

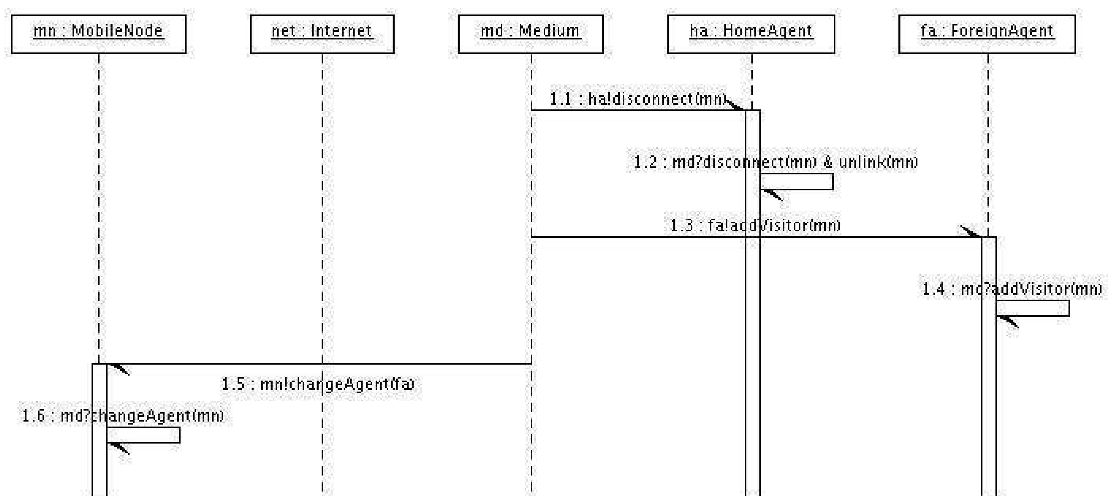


Figura 4.9: Diagrama de seqüência do cenário 2

Cenário 3 O host móvel se muda da rede nativa para a rede estrangeira e se registra para que possa receber pacotes originalmente enviados para a sua rede nativa. O processo termina quando o host móvel recebe o aviso de registro efetuado. A seqüência em que este cenário ocorre é apresentado na Figura 4.10.

Cenário 4 O host fixo transmite pacotes para o host móvel que está em uma rede estrangeira. A seqüência é válida quando o pacote é entregue ao host móvel pelo agente es-

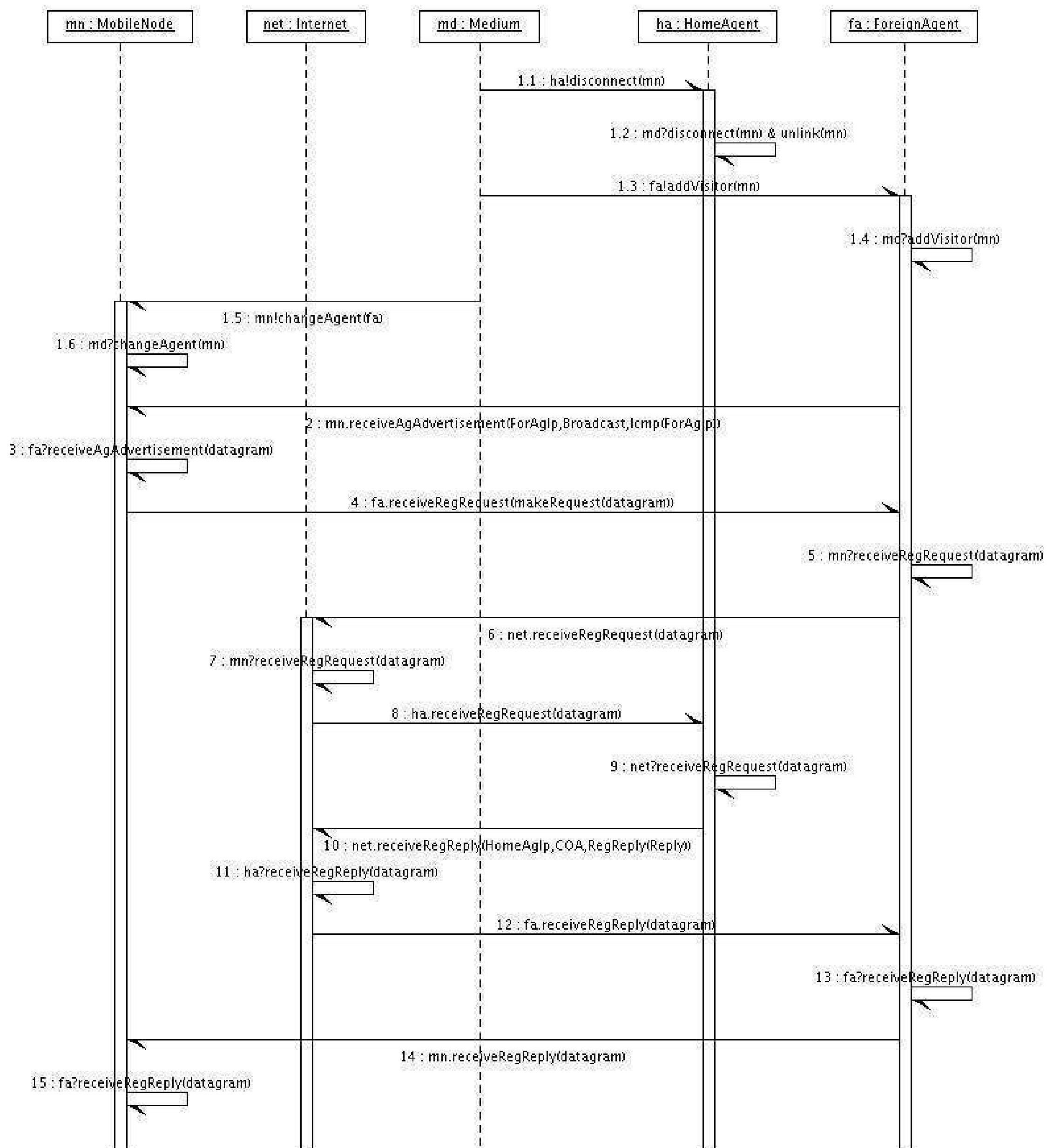


Figura 4.10: Diagrama de seqüência do cenário 3

trangeiro. A seqüência em que este cenário ocorre é apresentado na Figura 4.11.

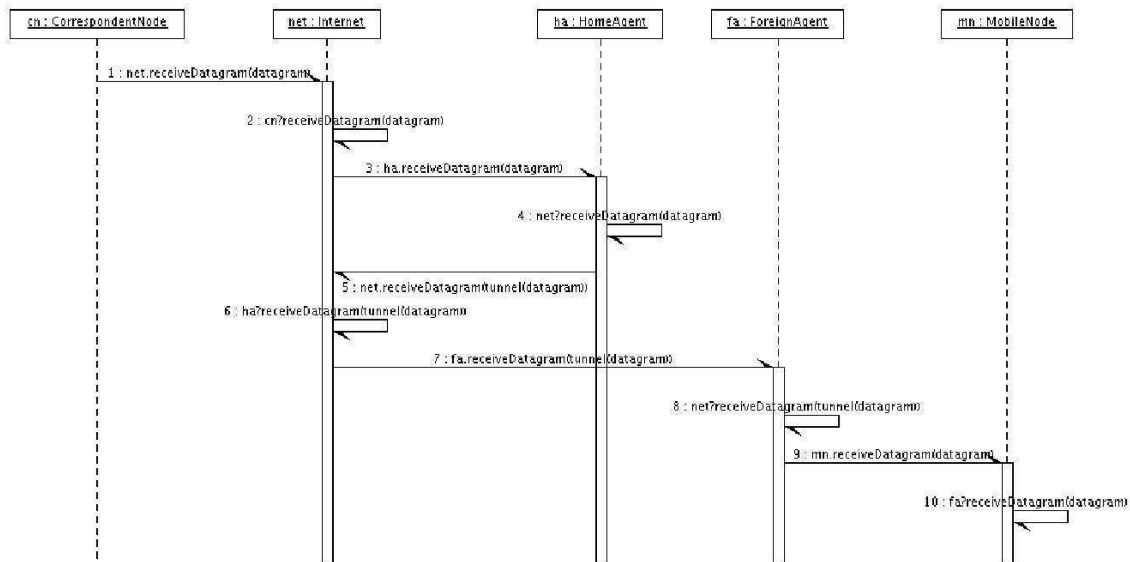


Figura 4.11: Diagrama de seqüência do cenário 4

4.3 Verificação de modelos

Para complementar a validação do protocolo IP móvel, utilizamos o protótipo desenvolvido na verificação do modelo. Sete propriedades foram definidas para verificar a corretude do modelo. Nas subseções seguintes, trataremos de cada uma delas:

Propriedade 1 A propriedade 1 especifica que nenhum datagrama se perde, isto é, todo datagrama enviado para o host móvel deve chegar até ele. Para expressar que o host fixo enviou um datagrama para o host móvel, podemos utilizar a função *ap1a*. Na prática, um host fixo não envia o datagrama diretamente para o host móvel. Primeiro, a mensagem que contém o datagrama é enviada para a internet.

$$\begin{aligned}\texttt{fun ap1a(s, ipm)} &= \texttt{VRTS.isReady("1`cn : net(receiveDatagram(CorrNodeIp,}\\ &\quad \texttt{HomeAddress, general(1)))", s, ipm);}\end{aligned}$$

Para avaliar que este datagrama sempre será enviado, podemos utilizar a propriedade *P1a*.

```
val AP1a = VRTS.AP("Host fixo envia datagrama", ap1a);
val P1a = VRTS.AF(AP1a);
```

Um vez que a propriedade *P1a* foi avaliada como verdadeira, tem-se que o datagrama enviado poderá chegar ao host móvel pelo agente nativo ou agente estrangeiro. Para expressar este recebimento, podemos utilizar as funções definidas por *ap1b* e *ap1c*.

```
fun ap1b(s, ipm) = VRTS.isReady("1`ha : mn(receiveDatagram(CorrNodeIp,
                   HomeAddress, general(1)))", s, ipm);
fun ap1c(s, ipm) = VRTS.isReady("1`fa : mn(receiveDatagram(CorrNodeIp,
                   HomeAddress, general(1)))", s, ipm);
val AP1b = VRTS.AP("Host movel recebe do agente nativo", ap1b);
val AP1c = VRTS.AP("Host movel recebe do agente estrangeiro", ap1b);
```

Quando um datagrama é encaminhado ao host móvel ele deve ser consumido em algum estado futuro. Em CTL, nós podemos expressar isso através da fórmula *P1b* e *P1c*. Mas também temos que em algum momento futuro sempre este datagrama será encaminhado, como especificado por *P1d* e *P1e*.

```
val P1b = VRTS.IMP(AP1b, VRTS.EX(VRTS.NOT(AP1b)));
val P1c = VRTS.IMP(AP1c, VRTS.EX(VRTS.NOT(AP1c)));
val P1d = VRTS.AG(VRTS.AF(P1b));
val P1e = VRTS.AG(VRTS.AF(P1c));
```

A avaliação da propriedade *P1d* foi positiva. A propriedade *P1e* falhou. No caso do host móvel se mudar de rede depois que um host fixo ter enviado datagramas, estes datagramas podem ficar pendentes para sempre para o host móvel.

Propriedade 2 Uma mensagem não pode chegar ao host móvel por duas rotas diferentes. Isto significa que uma mensagem deve chegar ao host móvel ou pelo agente nativo ou pelo

agente estrangeiro, mas nunca através dos dois. Em CTL, isto pode ser representado por *P2*, definida pela conjunção das propriedades *P2a* e *P2b*. A avalição de *P2* foi positiva.

```
val P2a = VRTS.AG(VRTS.IMP(AP1b, VRTS.NOT(VRTS.AF(AP1c))));
val P2b = VRTS.AG(VRTS.IMP(AP1c, VRTS.NOT(VRTS.AF(AP1b))));
val P2 = VRTS.AND(P2a, P2b);
```

Propriedade 3 Um host móvel não pode ser servido simultaneamente por um agente nativo e um agente estrangeiro. Em termos de sistemas de objetos, isto significa que um host móvel não pode estar associado simultaneamente a um agente nativo e a um agente estrangeiro. Em CTL, isto pode ser representado através da fórmula *P3*. A avaliação desta fórmula foi positiva.

```
fun ap2a(s, ipm) = VRTS.areLinked("fa","mn", s, ipm)
                   andalso VRTS.areLinked("mn","fa", s, ipm);
fun ap2b(s, ipm) = VRTS.areLinked("ha","mn", s, ipm)
                   andalso VRTS.areLinked("mn","ha", s, ipm);

val AP2a = VRTS.AP("Host movel na rede nativa", ap2a);
val AP2b = VRTS.AP("Host movel na rede estrangeira", ap2b);
val P3 = VRTS.AG(VRTS.NOT(VRTS.AND(AP2a, AP2b)));
```

Propriedade 4 Toda mensagem enviada pelo host fixo deve ser roteada para o agente nativo. Em termos de sistema de objeto, isto significa que sempre no futuro a mensagem enviada pelo host fixo vai estar pendente para o agente nativo. Em CTL, isto pode ser representado pela propriedade *P4*. A avaliação desta propriedade foi positiva.

```
fun ap3(s, ipm) = VRTS.isReady("1`net : ha(receiveDatagram(CorrNodeIp,
                  HomeAddress, general(1)))", s, ipm);

val AP3 = VRTS.AP("Datagrama enviado para o agente nativo", ap3);
```

$$\texttt{val P4} = \texttt{VRTS.AF(VRTS.AP(ap3));}$$

Propriedade 5 O host móvel deve mudar de agente de mobilidade. Nós podemos extrair esta informação observando a topologia da rede. No estado inicial do sistema, o host móvel está em sua rede nativa, assim, sempre no futuro o host móvel deve estar em uma rede estrangeira. Em CTL isto fica como apresentado na propriedade *P5*. *P5* foi avaliada como verdadeira.

$$\texttt{val P5} = \texttt{VRTS.AF(AP2a);}$$

Propriedade 6 Sempre que um host móvel se mudar de rede ele deve ser identificado por um agente estrangeiro e ser notificado da mudança pelo o agente que o identificou. Isto implica em verificar se sempre no futuro o host móvel recebe uma mensagem de notificação. A função que avalia se este recebimento ocorreu é dado por *ap4*. Toda a fórmula é dada por *P6*. *P6* foi avaliada como verdadeira.

$$\begin{aligned}\texttt{fun ap4(s,ipm)} &= \texttt{VRTS.isReady("1`fa:mn(receiveAgAdvertisement(ForAgIp,} \\ &\quad \texttt{Broadcast,icmp(ForAgIp)))",s,ipm);}\end{aligned}$$

$$\texttt{val AP4} = \texttt{VRTS.AP("Hostmovelfoinotificado",ap4);}$$
$$\texttt{val P6} = \texttt{VRTS.AF(AP4);}$$

Propriedade 7 Sempre que um host móvel se mudar de rede ele deve registrar seu novo endereço junto ao seu agente nativo. Através de *P5* nós sabemos que um host móvel sempre no futuro se muda para uma rede estrangeira. Portanto, agora precisamos saber se ele consegue se registrar junto ao seu agente nativo. O predicado atômico que avalia se ele recebeu a confirmação de registro é dado por *ap4*.

$$\begin{aligned}\texttt{fun ap5(s,ipm)} &= \texttt{VRTS.isReady("1`fa:mn(receiveRegistrationReply(HomeAgIp,} \\ &\quad \texttt{ForAgIp,regRep(RegDone)))",s,ipm);}\end{aligned}$$

Para que o host efetue seu registro, em algum momento do futuro ele deve consumir a confirmação representada por *AP4*. Este consumo é representado pela propriedade *p6a*. Em CTL, nós podemos expressar toda a propriedade através da fórmula *P7*.

```
val p7 = VRTS.IMP(VRTS.AP5, VRTS.EX(VRTS.NOT(VRTS.AP5)));
val P7 = VRTS.AG(VRTS.AF(p7));
```

4.4 Considerações Finais

Neste capítulo, apresentamos a aplicação do protótipo na verificação do modelo protocolo IP móvel. Inicialmente, a validação desta modelagem ocorreu através de simulação. Para isto nós utilizamos as ferramentas SSO e *Design/CPN*. Dos cenários abordados, nós construimos diagramas de seqüência. Esta primeira etapa foi fundamental para depurar e adquirir confiança no modelo. Somente após a estabilização do modelo que realizamos a verificação de modelos. Nesta segunda etapa, definimos setes propriedades que o modelo deve satisfazer. A verificação do modelo em relação a estas propriedades mostrou inconsistências não detectadas através de simulação.

A respeito do protótipo, esta atividade nos permite concluir que até tratarmos o problema da explosão de espaço de estados, o nosso verificador pode analisar modelos que a primeira vista se mostram simples, mas que são naturalmente complexos justamente por se tratar de sistemas concorrentes.

Capítulo 5

Conclusão

Nesta dissertação, tratamos do problema da falta de ferramentas para a verificação formal de modelos em RPOO. Dentre as possíveis técnicas de verificação formal, escolhemos a técnica de verificação de modelos. Esta decisão foi tomada considerando-se o potencial de automação, qualidade dos resultados produzidos e a sua crescente aplicação tanto na academia quanto na indústria.

Nossa preocupação principal ao desenvolver este trabalho foi tornar mais viável a verificação de modelos em processos de desenvolvimento baseados em modelos OO. Ao definirmos a gramática para formalizar a representação do espaço de estados de modelos em RPOO, por exemplo, priorizamos uma notação que torna evidente as características de OO, tais como as entidades de um modelo, as associações e a troca de mensagens. A definição deste formato de referência para espaço de estados, por sua vez, nos permitiu definir uma arquitetura independente das demais ferramentas do ambiente RPOO. Com este formato, foi possível desacoplar o verificador da ferramenta da geração do espaço de estados.

A lógica temporal que escolhemos para o nosso verificador foi CTL. Optamos por esta lógica porque ela tem se mostrado adequada para expressar propriedades de sistemas concorrentes e também eficiente para a verificação de modelos. Neste trabalho, definimos algoritmos para a solução de fórmulas nesta lógica, com capacidade de produzir exemplos e contra-exemplos. Nossos algoritmos têm complexidade $O(|f| \times (|S| + |E|))$, onde $|f|$ é o tamanho da fórmula em lógica temporal CTL, $|S|$ é a quantidade de estados do espaço de estados e $|E|$ é a quantidade de eventos.

A concretização do trabalho resultou em um protótipo do verificador de modelos em

RPOO. Durante a aplicação prática da ferramenta, nos deparamos com o problema da explosão de espaço de estados. Na verificação de modelos com cerca de 10^6 elementos, tivemos problemas com falta de memória. Esta ocorrência era esperada, pois a explosão de espaço de estado é um problema bem conhecido na literatura de verificação de modelos.

Em relação à aplicação prática dos resultados obtidos, realizamos a validação de uma modelagem do protocolo IP móvel. A validação deste modelo ocorreu através de simulação e verificação de modelos. A aplicação de verificação de modelos permitiu encontrar inconsistências não detectadas durante a simulação.

5.1 Contribuições

Antes da realização deste trabalho não existia ferramentas para a validação formal de modelos RPOO. Os experimentos com as modelagens nesta linguagem estavam restritos à edição e simulação de modelos. Embora sejam atividades significativas e que podem trazer bons resultados, elas não exploram todo o potencial que o formalismo possui. Com o nosso trabalho, agora é possível realizar verificação de modelos RPOO. Ao construir o modelo, o usuário pode descrever as propriedades esperadas usando lógica temporal CTL. O modelo pode ser avaliado segundo estas propriedades usando a verificação de modelos com suporte ferramental que considera as características notacionais do formalismo.

Os resultados da aplicação do verificador no modelo do protocolo IP Móvel exemplificam a contribuição do trabalho realizado. Esta atividade mostrou que até tratarmos o problema da explosão de espaço de estados, o nosso verificador pode analisar modelos que a primeira vista se mostram simples, mas que são naturalmente complexos justamente por se tratar de sistemas concorrentes.

Este trabalho também é motivação para um projeto de desenvolvimento para realizar a construção de um ambiente RPOO completo, com ferramentas para edição, simulação e verificação formal de modelos. Este novo ambiente tem como ponto de partida a utilização da linguagem de programação Java para descrever os tipos de dados, funções e expressões das redes de Petri. A arquitetura do nosso verificador, os algoritmos desenvolvidos e a notação para representação do espaço de estados serão utilizados neste novo ambiente.

Os algoritmos que definimos também são resultados relevantes para o projeto RPOO.

Na arquitetura atual, estes algoritmos dependem da representação explícita do espaço de estados do sistema. Contudo, nada impede que eles sejam utilizados em uma abordagem de verificação mais eficiente, conhecida como *On-the-fly Model Checking*. Nesta abordagem, o espaço de estados é construído sob demanda da propriedade a ser verificada. Ela é mais eficiente porque em muitos casos o exemplo ou contra-exemplo pode ser encontrado sem ser necessário percorrer todo o espaço de estados. Os nossos algoritmos estão prontos para esta abordagem porque eles dependem somente da informação contida no estado e da relação de sucessor.

Durante o desenvolvimento do trabalho a nossa preocupação principal foi desenvolver a verificação de modelos para RPOO de forma a viabilizar a sua aplicação em desenvolvimento de sistemas baseados em modelos segundo o paradigma da orientação a objetos. Alguns autores [DHJ$^+$01; JR00] argumentam que a aplicação de verificação de modelos pode não ocorrer nestes casos porque as linguagens para descrição de modelos utilizadas pelos verificadores não são capazes de caracterizar com rigor e naturalidade as características da OO. Por definirmos uma notação para representação do espaço de estados que enfatiza as características OO dos modelos, acreditamos que o nosso trabalho tem grande potencial para contribuir neste domínio de aplicação.

5.2 Trabalhos Futuros

Neste trabalho, desenvolvemos um protótipo de um verificador de modelos para RPOO. Um futuro trabalho que deve ser realizado é a transformação deste protótipo em uma ferramenta. Isto significa desenvolver interface gráfica, instalador, tutorial, mecanismos de configuração de ambiente, tais como quantidade de memória utilizada, entre outros.

Outro trabalho futuro importante é a validação da implementação através de testes formais. A adoção de linguagem de programação funcional para a implementação do protótipo pode facilitar sobremaneira a realização de testes formais usando especificação CASL [BM03]. As assinaturas SML dos módulos que construímos e os axiomas utilizados na prova dos algoritmos constituem uma base para especificação e geração automática de testes usando CASL .

A aplicação prática de verificação de modelos em sistemas complexos depende de

soluções para o problema da explosão de espaço de estados. Naturalmente, é fundamental que trabalhos neste sentido sejam realizados para permitir a aplicação do verificador de modelos para RPOO em situações reais. Com os resultados obtitdos neste trabalho, podemos finalmente realizar estudos neste sentido. Os algoritmos que definimos, a notação que escolhemos e a aplicação do verificador em outros modelos RPOO podem embasar a adequação de técnicas existentes como relação de ordem parcial, interpretação abstrata e verificação modular.

Outros trabalhos relevantes que também podem ser realizados são os seguintes:

1. Verificação de modelos UML: é muito comum que processos de desenvolvimento utilizem UML como linguagem para descrição de modelos. Como esta linguagem não é formal, a verificação de modelos não pode ser diretamente aplicada. Realizar a verificação de modelos UML requer a definição de uma semântica formal para diagramas UML. Esta definição pode ocorrer usando RPOO. Depois disto, nós poderemos utilizar o nosso verificador de modelos.

2. Verificação de código Java: tradicionalmente, a verifição de código em Java pode ser obtida através de abstrações derivadas de programas escritos nesta linguagem. Dentro do projeto RPOO, existe um trabalho cuja essência consiste em anotar o código de programas com uma linguagem baseada em Java. Destas anotações, são extraídos modelos abstratos em RPOO e também especificações para permitir a verificação. O suporte ferramental desenvolvido pode ser utilizado para viabilizar a verificação destes modelos.

Bibliografi a

[ASK03] Ask-ctl / design cpn, 2003". http://www.daimi.au.dk/designCPN/libs/askctl/.

[BM03] M. Bidoit and P. D. Mosses. *Casl User Manual*. The Common Framework Initiave, 2003.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *ieeetc*, C-35(8):677–691, aug 1986.

[CCM96] A. Cheng, S. Christensen, and K. H. Mortensen. Model checking coloured petri nets exploiting strongly connected components. In *M.P. Spathopoulos, R. Smedinga, P. Kozak (eds.), International Workshop on Discrete Event Systems, Edinburg, Scotland, UK*, pages 169–177, aug 1996".

[CE81] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *Lecture Notes in Computer Science*, 131, 1981.

[CGL92] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 343–354. ACM Press, 1992.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[CGP98] E. Clarke, O. Grumber, and D. Peled. *Model Checking*. MIT Press, 1998.

[CWA+96] E. M. Clarke, J. M. Wing, R. Alur, R. Cleaveland, D. Dill, A., S. Garland, S. German, J. Guttag, A. Hall, T. Henzinger, G. Holzmann, C. Jones, R. Kurshan, N. Leveson, K. McMillan, J. Moore, D. Peled, Amir, J. Rushby, N. Shankar,

J. Sifakis, P. Sistla, B. Steffen, P. Wolper, J. Woodcock, and P. Zave. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996".

[dFGM03] J. C. A. de Figueiredo, D. D. S. Guerrero, and P. D. L. Machado. Técnicas e ferramentas para a validação rigorosa de sistemas de software com características de mobilidade, 2003. http://www.dsc.ufcg.edu.br/ mobile.

[DHJ+01] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasareanu, H. Zheng, and W. Visser. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd international conference on Software engineering*, pages 177–187. IEEE Computer Society, 2001.

[EOH+93] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In D. Agnew, L. Claesen, and R. Camposano, editors, *The Eleventh International Symposium on Computer Hardware Description Languages and their Applications*, pages 5–20, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[GPS96] P. Godefroid, D. Peled, and M. Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. In *Proceedings of the 1996 international symposium on Software testing and analysis*, pages 261–269. ACM Press, 1996.

[Gue02] D. D. S. Guerrero. Redes de petri orientadas a objeto. *Tese de Doutorado - COPELE, UFCG*, 2002.

[HDZ00] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher Order Symbol. Comput.*, 13(4):315–353, 2000.

[Hel97] K. Heljanko. Model checking the branching time temporal logic ctl. Technical Report A45, Helsinki University of Technology, 1997.

[Hol97] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997".

[JEK^{+}90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[Jen92] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1992.

[JR00] D. Jackson and M. Rinard. Software analysis: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 133–145. ACM Press, 2000.

[Kat99] J. P. Katoen. *Concepts, Algorithms, and Tools for Model Checking*. Lectures Notes of the Course "Mechanised Validation of Parallel Systems. Friedrich-Alexander Universitat Erlangen-Nurnberg, 1999.

[McM92] K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, 1992.

[McM93] K. L. McMillan. Symbolic model checking: An approach to the state explosion problem. *Kluwer Academic Publishers*, 1993".

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390. Springer-Verlag, 1994.

[PM94] C. E. Perkins and A. Myles. Mobile IP. *Proceedings of International Telecommunications Symposium*, pages 415–419, 1994.

[Pnu77] A. Pnueli. The temporal logic of programs. *Proceedings 18th IEEE Symposium on Foundations of Computer Science*, 1977.

[RS95] S. Romanenko and P. Sestoft. Moscow ml owner's manual, 1995.

[San03] J. A. M. Santos. Suporte ï¿½anï¿½ise e verificaï¿½o modelos rpoo. *Dissertaï¿½o de mestrado - COPIN, UFCG*, 2003.

[Sif90] J. Sifakis, editor. *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings*, volume 407 of *Lecture Notes in Computer Science*. Springer, 1990.

[SMV03] Model checking smv, 2003". http://www-2.cs.cmu.edu/ modelcheck/index.html.

[Val98] A. Valmari. The state explosion problem. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:429–528, 1998.

[VL93] B. Vergauwen and J. Lewi. A linear local model checking algorithm for ctl. In E. Best, editor, *CONCUR'93: Proc. of the 4th International Conference on Concurrency Theory*, pages 447–461. Springer, Berlin, Heidelberg, 1993.

Apêndice A

Formalização de RPOO

Nesta apêndice, definimos formalmente os elementos que compõem a linguagem RPOO. Nossa apresentação se restringe aos elementos necessários à compreensão do trabalho que propomos. A descrição completa da linguagem pode ser encontrada no trabalho de Guerrero [Gue02]. Por se tratar de uma apresentação formal, a maior parte das definições foram transcristas do formalismo original. A notação que utilizamos é a mesma utilizada no formalismo original.

A.1 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, vamos utilizar a notação

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

Para representar objetos, vamos utilizar a notação

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

Para descrever que um objeto o_i pertence a uma classe C_j nós 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, denotamos o identificador do objeto o_i por $\vec{o_i}$. Também assumimos 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,

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

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 Grande parte da da modelagem consiste em descrever os tipos de dados que devem ser manipulados pelos objetos. A modelagem de dados que adotamos consiste em utilizar 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.

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, devemos utilizar 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 da 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, dizemos que "o tipo de m é $\mathbb{S}_i$" se $m \in \mathcal{T}^{\mathsf{ms}}_{\mathbb{S}_i}$.

A.2 Modelo RPOO

Nesta seção, vamos apresentar os elementos básicos necessários à composição de modelos 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. Nós definimos formalmente cada um desses elementos 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 de 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 comportamento dos objetos que compõem uma classe, utilizamos redes de Petri colorida, ou CPN ([Jen92]). Desta forma, esta parte de RPOO será apresentada com a mesma notação utilizada na formalização de CPN em [Jen92]. 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.

Apresentamos a definição da estrutura de uma rede de Petri:

Definição A.1 (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 A.2 (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:*

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

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

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 A.3 (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 multiconjunto sobre* F*. O conjunto das marcações de uma rede é denotado por* M*.*

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 A.4 (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 A.5 (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 A.6 (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*

$$m'(p) = m(p) - [\![E(p,t)]\!]_a + [\![E(t,p)]\!]_a \qquad \textit{para todo } p \in P. \tag{A.1}$$

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 a 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 não tem efeito externo, isto é, que alteram apenas o estado interno dos objetos, são denominadas ações internas e não necessitam de inscrições de interação.

Definição A.7 (Inscrições de interação) *Sejam* 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*

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

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 $v : \mathbb{C}$ são chamadas de *instanciações*. A variável objeto v é ligada ao identificador do objeto instanciado; logo, os valores de $v \in \mathsf{N}$. Inscrições do tipo $\widetilde{s}$ 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 A.8 (Classe) *Seja* $\mathbb{C}_i$ *um nome de classe,* K *uma rede de Petri e* $I : T \rightarrow \mathsf{R}$ *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.*

Nossa apresentação do elementos que compõem um modelo termina aqui. Não vamos definir as relações entre as classes pois acreditamos que conteúdo apresentado até agora é suficiente para a construção de modelos RPOO. Mais informações podem ser encontradas na versão completa do formalismo.

A.3 Sistema de Objetos

Nesta seção, vamos tratar finalmente do comportamento de um modelo RPOO. A parte do formalismo que trata deste assunto é denominada: *sistema de objetos*. No 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. Sabemos, também, que cada objeto que compõem um sistema pode executar um conjunto de ações condicionadas pelo seu comportamento e estado. Assim, as várias configurações alcançáveis, caracterizadas pelos objetos, ligações, mensagens, comportamento e estado, definem o sistema de objetos. Por se tratar de uma linguagem formal em que estados podem ser calculados, para composição do modelo basta-nos a definição de uma configuração inicial que represente os elementos ativos no momento da inicialização do sistema. As demais configurações passam a ser definidas pelo comportamento dos objetos descritos através das redes de Petri.

Definição A.9 (Estrutura do sistema de objetos) *A estrutura de um sistema de objetos é uma tupla* $\langle O, L, M \rangle$*, onde* O *é um conjunto de identificadores formado por elementos de* N*, de objetos ativos no sistema,* L *é o conjunto de ligações que são pares de identificadores da forma* $\langle a1, a2 \rangle$ *e* M *é um conjunto de mensagens pendentes. As mensagens são tuplas* $\langle m, a1, a2 \rangle$*, onde* m *é um termo e* $a1$ *e* $a2$ *são identificadores de objetos.*

As ligações descritas na forma $\langle a1, a2\rangle$ definem que o objeto $a1$ possui uma referência para o objeto $a2$. 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, $a2$ pode ser um objeto morto. Para descrever que existe uma mensagem m pendente, enviada do objeto $a1$ para o objeto $a2$, utilizamos o formato $\langle m, a1, a2\rangle$. 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 estrutura inicial a uma classe. A partir de uma estrutura 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 A.10 (Instanciação) *Seja $E = \langle O, L, M\rangle$ a estrutura de um sistema de objetos, e $\mathbb{C}$ a representação de uma classe, definimos a* função instância $I : O \rightarrow C$ *mapeia cada objeto da estrutura para uma das classes definidas no modelo.*

A partir da definição da função instância anteriormente apresentada, nós podemos formalizar o conceito de uma configuração inicial.

Definição A.11 (Configuração Inicial) *Seja E a estrutura de um sistema de objetos, e I uma função instância que mapeia cada instância da estrutura a sua respectiva classe. Chamamos de* configuração inicial *a tupla formada por $\langle E, I\rangle$.*

Comportamento 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 ocorram em função da ocorrência de ações nos objetos que o compõem, as alterações na estrutura de um sistema de objetos devem obedecer um conjunto de regras que descrevem cada ação. Na Tabela A.1 apresentamos os tipos de ações elementares que cada objeto pode executar em um sistema de objetos, bem como a notação usada para descrevêlas.

Ação	Nome
τ	Ação interna (ou local)
new x	Criação ou instanciação de objetos
$x?m$	Entrada de dados
$x.m$	Saída assíncrona de dados
$x!m$	Saída síncrona de dados
$\tilde{x}$	Desligamento ou remoção de ligação
end	Ação final (ou auto-destruição)

Tabela A.1: Ações Elementares

Seja O o conjunto de objetos ativos na estrutura de um sistema de objetos, as ações são executadas por objetos $o \in O$. Na tabela, x é um identificador de objeto, isto é, $x \in \mathsf{N}_i$, e m é um elemento de qualquer um dos domínios de dados.

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. Cada objeto de uma estrutura é representado pelo seu rótulo. A estrutura $E = a + b$, por exemplo significa que na estrutura E, nós temos dois objetos, a e b. As ligações são descritas no formato $\vec{ab}$. Desta forma, a estrutura $E = a + b + \vec{ab}$, possui dois objetos, a e b, e o objeto a conhece o objeto b. Por último, as mensagens são representadas no formato: m^a_b. Assim, se $E = a + b + \vec{ab} + m^a_b$, a estrutura E possui dois objetos, a e b, o objeto a conhece o objeto b e existe 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 estrututura resultante da aplicação da ação a sobre a estrutura E. Para representar as ações, vamos utilizar os símbolos apresentados na Tabela A.1. Nas definições seguintes, o elemento que precede o símbolo ":" representa o objeto agente da ação.

Definição A.12 (Ação interna) *Se E é uma estrutura e $x : \tau$ representa 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 A.13 (Ação de criação) *Se E é uma estrutura e* x:new y *representa uma ação de criação em que o objeto x cria o objeto y, definimos*

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

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 $\notin!$ significa que a ação só pode ser executada se o objeto criado não fizer parte da estrutura anterior à execução da ação.

Definição A.14 (Ação de entrada de dados) *Seja E uma estrutura e $x : y?m$ uma ação de entrada de dados em que o objeto x consome a mensagem m enviada pelo objeto y, temos*

$$\langle E + m_y^x, x{:}y?m \rangle \rightarrow E + \vec{xz}, \textit{se m} = \bar{z}\ (\forall z \in O)$$

$$\langle E + m_y^x, x{:}y?m \rangle \rightarrow E, \textit{se m} \neq \bar{z}\ (\forall z \in O).$$

Onde $\bar{z}$ indica que z *é uma referência a um objeto e O é o conjunto de todos os possíveis objetos do sistema (ativos ou não).*

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 A.15 (Ação de saída de dados) *Seja E uma estrutura e $x{:}y.m$ ou $x{:}y!m$ uma ação de saída assíncrona e síncrona de dados, respectivamente, em que o objeto x envia a mensagem m para objeto y, definimos que*

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

$$\langle E, x{:}y.m\rangle \rightarrow E + m^x_y, \text{se } m = \bar{z} \text{ e } \vec{xy}, \vec{xz} \in E.$$

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 mesmo do consumo da mensagem pelo objeto destinatário.

Definição A.16 (Ação de desligamento) *Seja E uma estrutura e $x{:}\tilde{y}$ uma ação de desligamento em que o objeto x se desliga do objeto y, definimos que*

$$\langle E + \vec{xy}, x{:}\tilde{y}\rangle \rightarrow E, \text{se } \vec{xy} \notin E.$$

Em uma ação de desligamento um objeto se desliga de outro objeto.

Definição A.17 (Ação de auto-destruição) *Seja E uma estrutura e $x{:}end$ uma ação final em que x se destrói, temos*

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

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. Este efeito é representado pelo símbolo $\ominus$.

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 A.18 (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 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:

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

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 A.19 (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:*

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

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 A.20 (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 caracteiza 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 a 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 épossível 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. A Figura 2.4.1, por exemplo, pode ser considerado o grafo de alcançabilidade de uma instância do modelo do jantar dos filósofos, em que que cada configuração alcançada é um vértice.

Definição A.21 *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 $C_0 \models^* C_i$. 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. É exatamente através da exploração dos caminhos do grafo de alcançabilidade que analisamos nossos modelos.

A.4 CPN Equivalente

Na seção anterior definimos grafos de alcançabilidade como forma de representação do comportamento de modelos em sistemas objetos. Na prática, gerar e representar grafos de alcançabilidade é um problema que por si só requer suporte computacional adequado. Para fins de experimentação, contudo, podemos utilizar as ferramentas disponíveis para gerar espaços de estados de redes de Petri coloridas.

Para construirmos uma rede de Petri cujo grafo de alcançabilidade seja "semelhante" ao grafo de alcançabilidade do sistema de interesse, partimos da definição de uma relação de equivalência entre sistemas de objetos e redes de Petri. Dizemos que uma rede de Petri e um sistema de objetos são equivalentes se seus grafos de alcançabilidade são isomórficos.

Definição A.22 *Seja N uma rede de Petri e C_0 a configuração inicial de um sistema. Dizemos que a rede e o sistema expressam comportamentos* equivalentes*, denotado por $N \equiv C_0$, se seus grafos de alcançabilidade $\mathrm{G}(N)$ e $\mathrm{G}(C_0)$ são isomórficos.*

Do ponto de vista prático, a relação acima nos permite reduzir o problema de gerar o espaço de estados de um sistema de objetos à geração de uma rede de Petri equivalente do sistema. Desta forma, enquanto não houver ferramentas específicas para a notação proposta, podemos construir o espaço de estados de um sistema de objetos através de ferramentas para redes de Petri, como *Design CPN* [Jen92] por exemplo.

O restante desta seção é destinado a uma breve apresentação do algoritmo para conversão de modelos RPOO para CPN. Objetivo desta apresentação é permitir que o leitor com-

preenda como podemos construir grafos de ocorrência (ou espaço de estados) de modelos RPOO. O algoritmo original é apresentado em uma notação de pseudo-código, sem modularização explícita. Para facilitar a compreensão do texto, em vez de reproduzir o algoritmo original, vamos ocultar pseudo-código através de chamadas a procedimentos. Cada chamada representa uma etapa da apresentação original do algoritmo. Mais detalhes sobre o processo podem ser obtidos no trabalho de Guerrero [Gue02]

O algoritmo, denotado por *CPN_Equivalente* recebe como entrada um modelo RPOO, denotado por M, contendo as suas n classes. A saída produzida é uma única rede CPN, denotada por K, que representa a rede CPN equivalente. Para tornar a conversão mais genérica, a nova rede obtida não possui marcação inicial.

```
1:  procedure CPN_Equivalente(M)
2:      Cria_Novas_Cores(M)
3:      Altera_Cores(M)
4:      Cria_Rede_Ciclo_De_Vida(M)
5:      K = Unifica_Redes(M)
6:      Resolve_Saída_Assíncrona(K)
7:      Resolve_Entrada_de_Dados(K)
8:      Inicia_Composição_de_Eventos(K)
9:      Trata_Ações_Internas(K)
10:     Trata_Ações_Criação_Destruição(K)
11:     Trata_Sincronização(K)
12:     return K
13: end
```

Uma abordagem válida para se obter uma rede equivalente seria replicar múltiplas estruturas da rede, uma para cada objeto. Contudo, podemos obter uma solução mais concisa utilizando-se a abordagem convencional de redes de alto nível. Assim, basta-nos utilizar uma estrutura uma única vez e codificar as informações referentes aos objetos individuais nas fichas manipuladas pela rede. O procedimento chamado na linha 2 se encarrega de mo-

dificar cada cor utilizada para descrever o domínio de lugares da rede de modo que a cor resultante descreva também informações sobre o domínio de nomes das classes. As novas cores podem ser obtidas através da definição de duplas em que o primeiro elemento é a cor original e o segundo é um identificador de objeto. Na linha 3, o procedimento chamado se encarrega de fazer as mudanças para que cada lugar da rede tenha a nova cor correspondente. Para manter a rede consistente, este procedimento também modifica cada expressão de arco e de guarda para que seja considerado as novas cores dos lugares.

Em um modelo RPOO, os objetos podem executar uma ação de auto-destruição e podem também instanciar novos objetos. Na linha 4, nós tratamos justamente do ciclo de vida dos objetos. A idéia do procedimento chamado é compor cada rede atual com uma nova rede que modela o ciclo de vida dos objetos. Esta rede basicamente contém um lugar e duas transições. O lugar existe para guardar os objeto ativos do sistema. Para permitir a instanciação de novos objetos, uma das transições tem por finalidade inserir um novo objeto no lugar que guarda os objetos ativos Para manter a rede consistente, as demais transições da rede devem possuir um arco de teste para este lugar. Desta forma, garantimos que objeto só vai executar ações se ele estiver ativo. Para tratar da auto-destruição, a outra transição deve retirar objetos do lugar que guarda objeto ativos

Para tratar futuramente dos efeitos externos das ações de cada rede, na linha 5, criamos uma única rede, denotada por k, através da união disjunta das redes resultantes até este momento. Na linha 6, o primeiro efeito externo tratado é a saída assíncrona de dados. O resultado equivalente desta ação é obtido adicionando-se lugares para comportar mensagens pendentes e ligando as transições que modelam saídas assíncronas a esses lugares.

Em seguida, na linha 7, tratamos parcialmente da entrada de dados. Apenas pela ação de entrada não é possível saber se a entrada é complemento de uma saída de dados assíncrona ou síncrona. Por esta razão, nós devemos substituir cada transição que modela uma entrada de dados por duas: uma que resolve entradas assíncronas e outra para entradas síncronas. Somente as que tratam de entrada assíncrona devem ser ligadas aos lugares que guardam as mensagens pendentes. As que modelam entradas síncronas são “marcadas ” para serem tratadas futuramente.

Os procedimentos chamados nas linhas seguintes se resumem à composição des eventos a partir das ações de cada rede. Isto implica que a partir de agora cada transição vai ser

tratada para que o seu disparo represente a ocorrência de um evento no sistema de objetos. O procedimento chamado na 8 tem a finalidade de iniciar esse o processo. Ele basicamente faz uma cópia de todas as transições da rede K para um conjunto temporário e define o conjunto das transições de K como vazio. À medida que o restante do algoritmo obtém os eventos das ações, as novas transições obtidas são inseridas em K e as respectivas transições são retiradas do conjunto temporário. Assim, quando este conjunto estiver vazio, teremos que todas as ações foram tratadas.

Na linha 9, o procedimento chamado se encarrega de identificar os eventos correspondentes às ações internas. Isto implica em remover cada transição do conjunto do temporário e devolvê-la à rede K. Na linha 10, o algoritmo chama o procedimento responsável por tratar das ações instanciação de novos objetos e de auto-destruição. Para o caso de instanciação, a transição que está anotada como uma ação deste tipo deve ser fundida com a transição para instanciar novos objetos, criada na linha 4. Se ação corresponde à auto-destruição do objeto, a respectiva transição deve ser fundida com a transição para a finalização de objetos, também criada pelo procedimento invocado na linha 4.

Para finalizar, o procedimento chamado na linha 11 é responsável por tratar das sincronizações. O algoritmo busca as ações complementares de cada de saída síncrona. Se uma ação é encontrada, uma nova transição é criada fundindo as transições originais. Se a nova transição não tiver mais restrições de interação, isto é, não tiver mais inscrições, a transição modela um evento do sistema e é adicionada à rede K. Caso contrário ela é adicionada no conjunto temporário para que interações seguintes do algoritmo a restrição possa ser resolvida.

Para ilustrar aplicação do algoritmo de conversão, na Figura A.1 mostramos o resultado obtido na modelagem do jantar dos filósofos, apresentada na Seção 1.2. Por questões de simplificação, a modelagem equivalente não contém os lugares e transições para tratar do ciclo de vida dos objeto, já que neste modelo não há ações de instanciação e auto-destruição. Na Figura A.2, mostramos o grafo de ocorrência produzido pela ferramenta *Design CPN*. Na Figura 2.4.1, se considerarmos que cada configuração alcançada é um vértice, podemos notar que as estruturas obtidas são isomórficas.

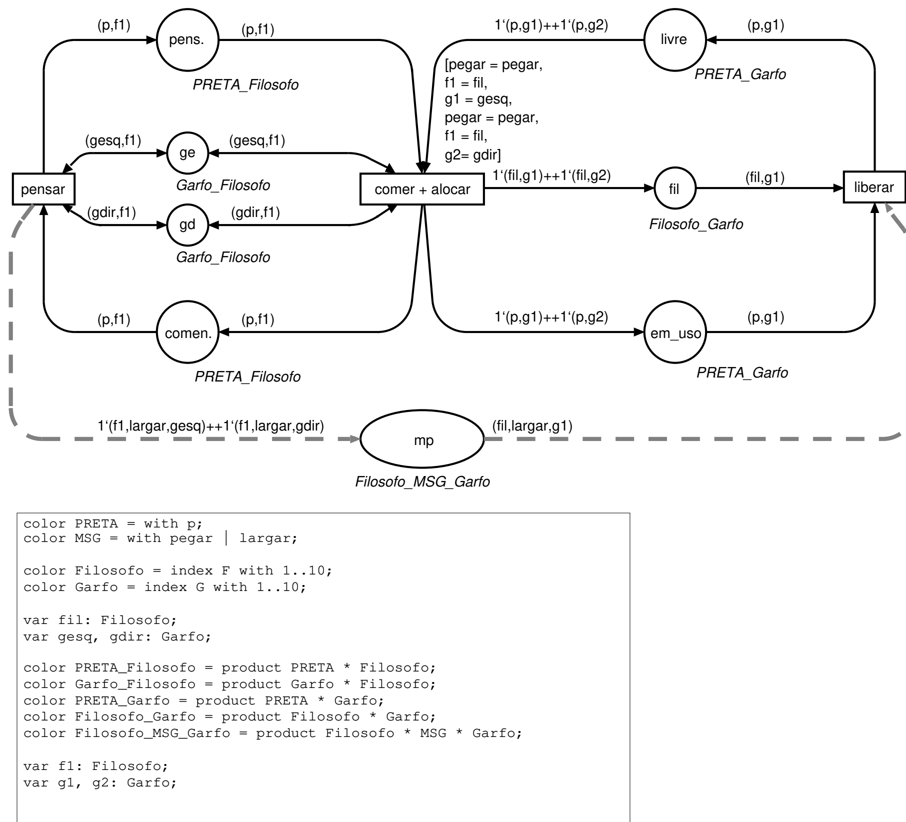


Figura A.1: O modelo jantar dos filósofos em CPN equivalente

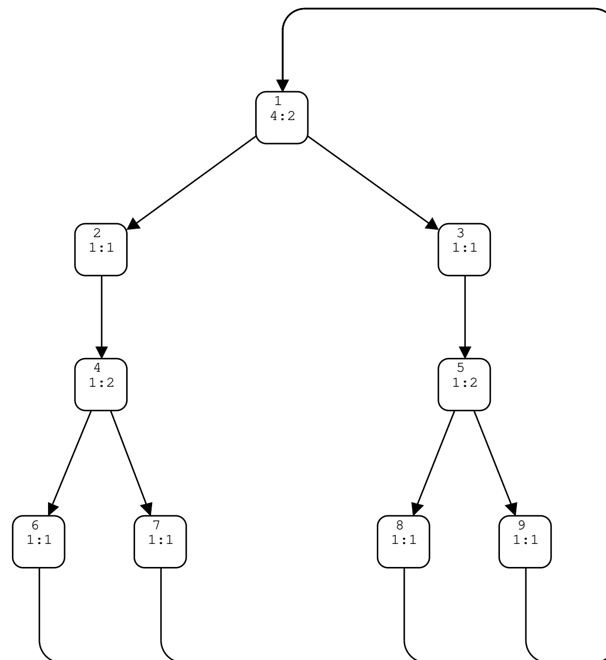


Figura A.2: O espaço de estados do modelo jantar dos filósofos em CPN equivalente