Simulação Automática e Geração de Espaço de Estados de Modelos em Redes de Petri Orientadas a Objetos

Taciano de Morais Silva

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

Área de Concentração: Engenharia de Software

Linha de Pesquisa: Métodos Formais

Jorge César Abrantes de Figueiredo

Dalton Dario Serey Guerrero

(Orientadores)

Campina Grande, Paraíba, Brasil

SILVA, Taciano de Morais

S586

Simulação Automática e Geração de Espaço de Estados de Modelos em Redes de Petri Orientadas a Objetos

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

108p. Il.

Orientadores: Jorge César Abrantes de Figueiredo

Dalton Dario Serey Guerrero

1. Redes de Petri

2. Redes de Petri Orientadas a Objetos

3. JMobile - Simulação Automática e Geração de Espaço de Estados

CDU – 519.711

"SIMULAÇÃO AUTOMÁTICA E GERAÇÃO DE ESPAÇO DE ESTADOS DE MODELOS EM REDES DE PETRI ORIENTADAS A OBJETOS"

TACIANO DE MORAIS SILVA

DISSERTAÇÃO APROVADA EM 31.08.2005

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

PROF. DALTON DARIO SEREY GUERRERO, D.Sc
Orientador

PROFª JOSEANA MACÊDO FECHINE , D.Sc
Examinadora

PROF. ARTURO HERNÁNDEZ DOMÍNGUÉZ, Dr.
Examinador

CAMPINA GRANDE – PB

Resumo

Redes de Petri Orientada a Objetos (RPOO) é um formalismo que faz a composição da semântica das redes de Petri com a semântica da Orientação a Objetos (OO). Esta forma de composição garante a manutenção das características originais de cada um dos paradigmas. A integração dos formalismos em RPOO é um aspecto importante, pois aproveita os recursos suportados pelas redes de Petri e pela orientação a objetos. As características inerentes de RPOO o tornam adequado para a especificação, simulação e verificação de protocolos, sistemas concorrentes e sistemas com características de mobilidade. Contudo, mesmo com o formalismo completamente definido e experimentado, a falta de algumas ferramentas importantes tornam o processo de simulação–validação–verificação difícil e muito trabalhoso. E assim, o uso prático do formalismo RPOO era prejudicado. Este trabalho teve como principal objetivo desenvolver um pacote de software que desse suporte à construção de ferramentas de simulação e geração de espaços de estados para modelos em Redes de Petri Orientadas a Objetos. O pacote de software foi desenvolvido completamente na linguagem de programação orientada a objetos — Java. As redes de Petri e seu mecânismo de evolução foi simulado através de classes e métodos dando origem ao *framework* JMobile. Com a construção do pacote de software foi possível construir modelos maiores e mais complexos, e como os protótipos do simulador e gerador de espaço de estados foi possível simularmos os modelos e gerarmos os seus espaços de estados.

Abstract

Object Oriented Petri Nets (RPOO) is a formalism that merges the Petri Nets and Object Oriented semantics. This kind of merging guarantees the maintaining of original features of each one of the paradigms. The integration of formalisms in RPOO is an important aspect because it uses supported resources in Petri Nets and in object oriented. The inherent features of RPOO make it adequate for the specification, simulation and verification of protocols, concurrent systems and systems with mobile features. However, even with the formalism completely defined and experienced, the absent of some important tools makes the simulation–validation–verification process difficult and harder to be done. In this way, the practical use of RPOO formalism was damaged. This work had like main goal developing a software package that gives support to the building of simulating and state space generating tools for models in Object Oriented Petri Nets. The software package was completely developed in a oriented programing language — Java. Petri Nets and their mechanisms of evolution were simulated by classes and methods. With the software package building for RPOO, it was possible to build bigger and more complex models, and with the simulator and space state generator prototypes it was possible to simulate the models and to generate their space states.

Agradecimentos

Aos meus pais, Tobias Lopes Silva e Josefa Iracema Morais, que sempre me apoiaram e estiveram presentes todos os momentos em que precisei deles. Agradeço a Deus pelos pais tão amorosos e compreensivos. Também tenho um agradecimento especial a fazer para meus irmãos Tacibias e Tacilânia pela força e amizade em todos os momentos e aos amigos da cidade de Serra Branca.

Tenho que fazer um agradecimento especial a todos os meus amigos do Curso de Ciência da Computação, principalmente os do período 2000.2, uma turma muito unida que me ajudaram muito durante os anos de curso e de mestrado. Aos colegas Paulo Eduardo, Rodrigo Tavares, Jairson Cabral e André Figueiredo pela contribuições e a todos os outros membros do Grupo de Métodos Formais.

Aos meus professores e orientadores Jorge César Abrantes de Figueiredo e Dalton Dario Serey Guerrero pelas orientações e ensinamentos que tornaram este trabalho possível.

Aos meus familiares, amigos e funcionários da universidade que de alguma forma participaram de minha vida durante este trabalho de mestrado.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Neste capítulo apresentamos o contexto em que se insere o nosso trabalho, enfatizando a sua motivação e sua relevância. O problema de simulação e geração de espaços de estados é caracterizado dentro do contexto de métodos formais. Finalmente, apresentamos os objetivos e resultados deste trabalho.

1.1 Contextualização

O pouco uso de métodos formais no desenvolvimento de *software* tem sido objeto de estudo e discussão na comunidade acadêmica nos últimos anos. De modo geral, observa-se que seu uso tem se restringido ao desenvolvimento de sistemas críticos, em que se justifica o alto custo da aplicação de métodos formais. Não há, até o momento, um consenso entre pesquisadores e desenvolvedores sobre os reais motivos do problema mencionado. Wordsworth [Wor99] apresenta em seu trabalho vários benefícios e desvantagens do uso de métodos formais. Alega que apesar dos problemas existentes, métodos formais podem ser introduzidos, com vantagens, em várias etapas do processo de desenvolvimento. Um dos principais problemas identificados por Wordsworth é a dificuldade que desenvolvedores e usuários têm de tratar com notações matemáticas e precisas. Ele atribui este fato à pouca ênfase dada à atividade de especificações matemáticas em cursos de ciência e engenharia da computação, em que são formados os desenvolvedores. Robert Glass [Gla04], por sua vez, discorda desta visão do problema. Ele não acredita que falte treinamento aos desenvolvedores, uma vez que métodos formais têm, de uma forma ou de outra, sido ensinados há décadas em diver-

sos cursos de graduação. Um aspecto observado por vários autores, incluindo os citados acima, é o fato de que na prática, especificações evoluem à medida que o desenvolvimento do sistema avança e à medida em que se entendem melhor os requisitos e a própria solução. Nesse sentido, argumenta-se que especificações formais tendem a dificultar a evolução do desenvolvimento por tornar onerosa a sincronização entre artefatos de especificação, projeto e implementação.

Tomando outra perspectiva, Constance Heitmeyer [Hei98] e Stidolph [SW03] discutem a utilização de métodos formais do ponto de vista de processo e de gerência de software. Heitmeyer defende que métodos formais poderiam trazer vantagens mais concretas para o desenvolvimento de sistemas se fosse realizado um esforço no sentido de tornar as técnicas mais fáceis e práticas de serem utilizadas por desenvolvedores sem maior treinamento em métodos formais. Para tanto, argumenta ele, os métodos deveriam utilizar linguagens com sintaxe e semântica familiares às tipicamente usadas pelos desenvolvedores e facilitar sua aplicação, seja pela disponibilidade de ferramentas que automatizem a análise dos artefatos, seja através de formas adequadas de integração a processos existentes e amplamente adotados pelos desenvolvedores. Desta forma, estima o autor, que as habilidades com notações matemáticas requeridas de desenvolvedores e usuários podem ser reduzidas e, apesar disso, as vantagens poderiam ser mantidas.

Stidolph reconhece ainda que métodos formais têm sido pouco utilizados na indústria de *software*, exceto em contextos críticos. No entanto, acredita que o contexto atual está mais receptivo à sua utilização devido à crescente necessidade de se demonstrar a corretude dos sistemas produzidos. Escolher o nível de formalidade, as técnicas, as ferramentas e o que deve ser formalizado, argumenta ele, são os pontos críticos nesse trabalho. Stidolph estabelece ainda critérios que podem ajudar na decisão de adotar ou não métodos formais em projetos de desenvolvimento. Some-se a isto a crescente demanda por sistemas de natureza concorrente e distribuída, cuja análise é reconhecidamente mais difícil que a de sistemas seqüenciais.

Como se vê, o uso prático de métodos formais é um tema bastante discutido nos fóruns de métodos formais. Apesar do pouco uso, a maioria dos autores apresenta fatos que indicam, no contexto atual, uma crescente receptividade à utilização de métodos formais em projetos de forma geral. Com base nestas discussões extraímos, os dois motivos que consideramos

fundamentais:

1. a falta de ferramentas adequadas;

2. e a pouca proximidade das técnicas às práticas de desenvolvimento.

Redes de Petri Orientadas a Objetos Os problemas acima foram percebidos em vários dos métodos e linguagens existentes. Um exemplo são as várias tentativas de aproximar a modelagem de redes de Petri à modelagem e análise de sistemas distribuídos pela adição de conceitos de orientação a objetos a formalismos baseados em redes de Petri (ver [Gue02b]). Guerrero propõe uma variação de redes de Petri Coloridas em que a organização dos modelos se dá mediante os conceitos da orientação a objetos. Tal formalismo permite aproximar modelos e técnicas formais da linguagem que é o padrão *de facto* para a modelagem de sistemas de software, que é a orientação a objetos (OO). Redes de Petri orientadas a objetos, ou simplesmente RPOO, têm sido usadas para modelar e validar formalmente protocolos de comunicação e sistemas distribuídos com características de mobilidade [GGPdF01; Rod04; GS03]. Tais modelos, embora formais, podem ser mais facilmente compreendidos por desenvolvedores do que outros modelos formais devido ao apelo da decomposição OO que lhes é familiar. Também há tentativas de se aproximar métodos formais diretamente com linguagens de programação, por exemplo, a limguagem Java (ver [BHPV00], [HP00]). Mesmo neste caso ainda há problemas, pois este tipo de método gera com maior rapidez o problema da explosão do número de estados possível do modelo do sistema.

Embora tenham se mostrado mais adequadas para a modelagem de sistemas distribuídos, RPOO ainda são um formalismo complexo que envolve vários níveis de abstração. Na prática, isso significa que mesmo modelos pequenos são difíceis de manipular, simular e validar sem o auxílio de ferramentas que automatizem o processo. Em particular, técnicas de verificação baseadas na exploração exaustiva do espaço de estados, como é típico em análise de redes de Petri, tornam-se inviáveis sem o auxílio de ferramentas.

Ferramentas para RPOO Uma forma de editar e simular modelos em RPOO é através do uso de duas outras ferramentas: o Design/CPN [KCJ98] e o SSO, ou Simulador de Sistemas de Objetos [San03]. No Design/CPN é feita a edição e simulação das redes de Petri de que é

composto um modelo em RPOO. Como a evolução dos objetos é representada pelo comportamento de redes de Petri, a simulação das redes expressa o comportamento individual dos objetos. A segunda ferramenta necessária, o SSO, complementa essa simulação permitindo expressar e avaliar a evolução da configuração do sistema de objetos de que é composto um modelo. Assim, a simulação de um modelo RPOO consiste, portanto, em sincronizar duas simulações realizadas em diferentes níveis de abstração, realizada por duas ferramentas independentes.

O artifício descrito permite validar modelos RPOO através de simulações. A idéia é executar o modelo em diversos cenários escolhidos segundo sua relevância, levando-se em conta as propriedades desejadas e não desejadas do sistema. Cada cenário de simulação é acompanhado e verificado pelo próprio desenvolvedor através da animação da evolução dos processos/objetos envolvidos. Embora não possa ser considerado um método formal no sentido estrito, a validação de sistemas através da simulação de cenários específicos é bastante utilizada no desenvolvimento de sistemas e é um método considerado prático para esta atividade. Devido à semelhança com o processo de testes, validar sistemas através da simulação de modelos é uma prática de fácil aceitação. O problema é que somente se o modelo for trivialmente pequeno é que poderemos simular todos os cenários relevantes do sistema e *garantir* que o modelo tenha as propriedades desejadas. Na prática, qualquer modelo tem dimensões suficientemente grandes para tornar impossível a simulação exaustiva. Uma solução seria a verificação automática através de técnicas de exploração exaustiva e automática do comportamento expresso por um modelo. O problema é que o artifício acima descrito para a simulação de modelos em RPOO é manual. Conseqüentemente, todo o processo é lento e sujeito a erros.

Espaços de estados de modelos em RPOO Uma forma de automatizar o processo de verificação automática acima descrito consiste em construir uma representação finita de todo o comportamento descrito pelo modelo. Esse comportamento é tipicamente expresso como um grafo de ocorrência em que os nós representam estados do sistema e transições representam possíveis mudanças de estado—tal grafo também é conhecido como um sistema de transições rotuladas. Neste trabalho, dizemos que esse grafo captura o *espaço de estados* do modelo. Assim, cada caminho no espaço de estados a partir de algum dos seus estados

iniciais é um possível cenário de funcionamento do sistema.

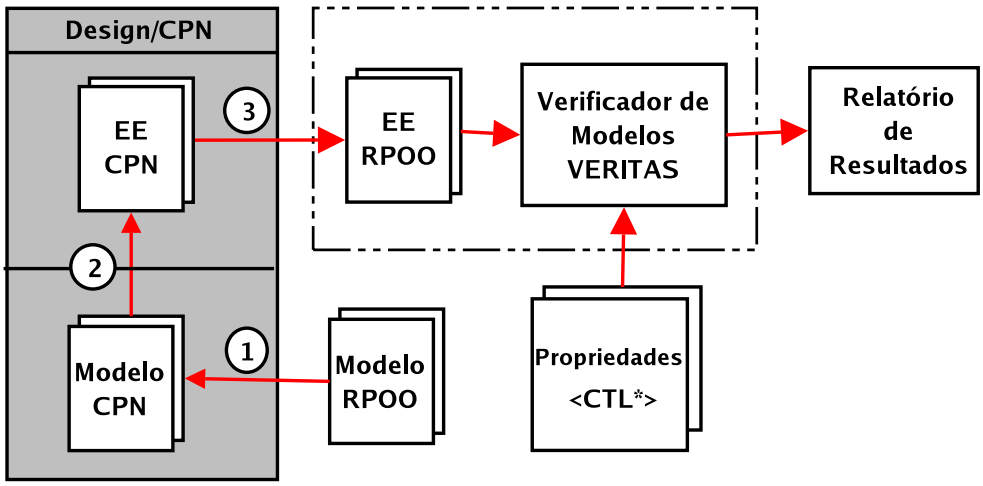


Figura 1.1: Ambiente de verificação RPOO

Diversas técnicas automatizadas de análise partem de uma representação do espaço de estados de um modelo para verificar suas propriedades. Logo, uma forma adequada de verificação formal de um modelo em RPOO consiste exatamente em gerar seu espaço de estados e submetê-lo às ferramentas adequadas de análise. Devido à falta de ferramentas específicas para tratar com modelos em RPOO, o processo tipicamente usado para a geração do espaço de estados de um modelo RPOO é mostrado na Figura 1.1. O processo consiste em três etapas:

1. traduzir o modelo RPOO para um modelo equivalente em redes de Petri coloridas — CPN (Coloured Petri Nets, [Jen92; Jen94; Jen97]);

2. gerar o espaço de estados para o modelo equivalente em CPN usando o Design/CPN;

3. traduzir o espaço de estados do modelo em redes de Petri colorida para o formato de espaço de estados RPOO.

O problema deste processo é que apenas o segundo passo que é completamente automático. Devido à inexistência de ferramentas que automatizem os passos um e três acima, todo o processo torna-se inviável na prática. Pois as duas traduções requerem muito tempo, seguem algoritmos complexos e são de inteira responsabilidade do modelador (Engenheiro de Software).

Em particular, a tradução de modelos em RPOO para redes CPN equivalente é tarefa extremamente complexa para ser executada de forma manual. Este processo de tradução segue o algoritmo de conversão de modelos RPOO para modelos CPN definidos por Guerrero [Gue02b].

JMobile Neste trabalho propomos uma nova notação para redes de Petri orientadas a objetos chamada JMobile cuja sintaxe é baseada na linguagem de programação Java. JMobile é uma evolução de redes de Petri orientadas a objetos baseada nas definições do formalismo RPOO. JMobile permite descrever modelos de sistemas distribuídos e concorrentes através de uma notação gráfica que combina redes de Petri para a descrição de comportamento concorrente, OO como forma de decomposição de modelos e expressões em Java para anotar manipulações de dados.

Na prática, trata-se ao mesmo tempo de uma notação para a modelagem de sistemas distribuídos e concorrentes e de um *framework* [JF88], [JR91], [JF92], que permitirá o desenvolvimento de ferramentas de simulação e geração de espaços de estados para os modelos. A disponibilidade de JMobile aproxima métodos formais da prática de desenvolvimento de software por incorporar notações amplamente utilizadas por desenvolvedores, ao mesmo tempo em que viabiliza a verificação automática dos modelos através de técnicas baseadas na exploração automática de espaços de estados.

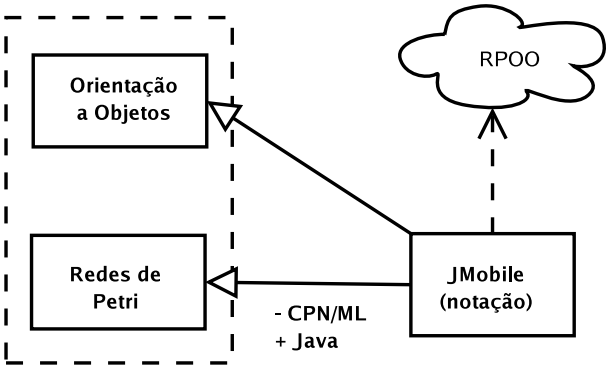


Figura 1.2: Visualização abstrata da composição de linguagens na notação JMobile

A Figura 1.2 apresenta uma visualização abstrata da composição da notação *JMobile*. A notação JMobile combina as redes de Petri com a sintaxe Java nas expressões das redes, substituindo o paradigma funcional usado originalmente nos modelos em RPOO. A

Figura 1.3 apresenta a arquitetura geral do framework JMobile que implementa o suporte para a modelagem, simulação e geração de espaço de estados.

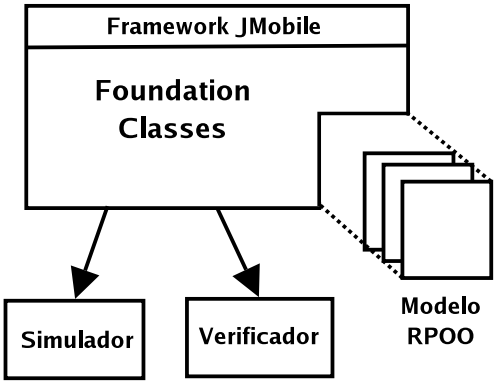


Figura 1.3: Arquitetura do Framework JMobile

1.2 Relevância

O desenvolvimento de suporte ferramental para o formalismo RPOO representa um passo fundamental para a continuação dos trabalhos de pesquisa sobre a inserção de métodos formais na prática de desenvolvimento de software. Em particular, este trabalho tornou possível a continuação da pesquisa sobre o uso de linguagens formais em contextos práticos de desenvolvimento por viabilizar a construção de modelos mais complexos do que os que eram possíveis sem o ferramental desenvolvido. Diversos trabalhos que já se encontram em andamento só foram possíveis devido à existência de JMobile. A título de exemplo, duas linhas de pesquisa, que se beneficiaram dos resultados obtidos neste trabalho, merecem destaque: o projeto Divíduo que investiga a geração distribuída de espaços de estados e a verificação paralela de modelos; e o projeto CompTest que investiga a geração automática de casos de teste de software a partir de modelos formais.

Além das linhas acima citadas, o atual estado do projeto permite iniciar o desenvolvimento de ferramentas de edição e de simulação específicas para JMobile sobre o *framework* desenvolvido. Tais projetos devem focar o desenvolvimento da camada de interface com o usuário e também se encontram em plena atividade. Algumas contribuições adicionais foram desenvolvidas e também são apresentadas como resultados deste trabalho. O pacote destas ferramentas, que chamamos *JMobile Tools*, inclui:

- a descrição da notação JMobile;
- uma representação em XML para modelos JMobile;
- uma representação em Java para modelos JMobile;
- uma API (Application Programming Interface) de um *parser* para a tradução automática da representação XML para a representação Java;
- uma API para a construção e simulação de modelos em JMobile, a partir de sua representação Java. Esta API permite a simulação guiada e a simulação automática de modelos JMobile.
- Uma API para a geração de espaços de estados de modelos JMobile, a partir da API de simulação. Esta API permite gerar o espaço de estados em dois formatos diferentes, com a possibilidade de ampliação no número de formatos.

Comentários sobre JMobile A notação e o *framework JMobile* estão sendo utilizados no ambiente de verificação formal de *software* nos níveis de modelagem, simulação e geração do espaço de estados.

Os modelos feitos com *JMobile* trazem uma maior proximidade com a forma de modelagem da orientação a objetos. Aliada a esta forma de modelagem a utilização da sintaxe Java nos modelos tornou mais intuitivo e direto o processo de modelagem de softwares orientados a objetos.

Os *frameworks* podem ser classificados de acordo com duas dimensões:

- Como o *framework* é usado;
- Onde o *framework* é usado.

Na classificação na Dimensão "Como o Framework é Usado", ele pode ser:

- *Inheritance-focused*

  Também chamado de white-box ou architecture-driven. Estende ou modifica funcionalidade pela definição de sub-classes com override de métodos.

- *Composition-focused*

  Também chamado de black-box ou data-driven. As coisas internas do framework não podem ser vistas ou alteradas. Deve-se usar as interfaces fornecidas As instanciações e composições feitas determinam as particularidades da aplicação.

- *Híbridos*

  A maioria dos frameworks é inheritance-focused com alguma funcionalidade pronta (composition-focused).

Na classificação na Dimensão "Onde o Framework é Usado", ele pode ser:

- *Framework de suporte*

- *Framework de aplicação ou horizontal*

  Encapsula conhecimento ("expertise") aplicável a uma vasta gama de aplicações. Resolve apenas uma fatia do problema da aplicação.

- *Framework de domínio ou vertical*

  Encapsula conhecimento ("expertise") aplicável a aplicações pertencendo a um domínio particular de problema.

O *framework* JMobile é um *framework* híbrido (como será usado), e é um *framework* horizontal (onde será usado).

Ainda na modelagem, a implementação do JMobile como *framework* permitiu a utilização de classes definidas pelo usuário e classes Java. Estas classes precisam estender a hierarquia do framework para serem utilizadas nos modelos. Em versões futuras pretende-se eliminar a esta restrição.

Com a ferramenta de simulação podemos fazer simulações guiadas e simulações automáticas, como em outras ferramentas de simulação de modelos. A inexistência desta ferramenta tornava muito difícil o processo de validação de modelos.

Por último, para completarmos o processo de verificação de software para JMobile era preciso um gerador de espaço de estados. Os espaços de estados são a entrada para ferramentas de verificação como o VERITAS [Rod04] e o Aldebaran [Fer89]. Na Figura 1.4 temos em destaque os artefatos desenvolvidos neste trabalho.

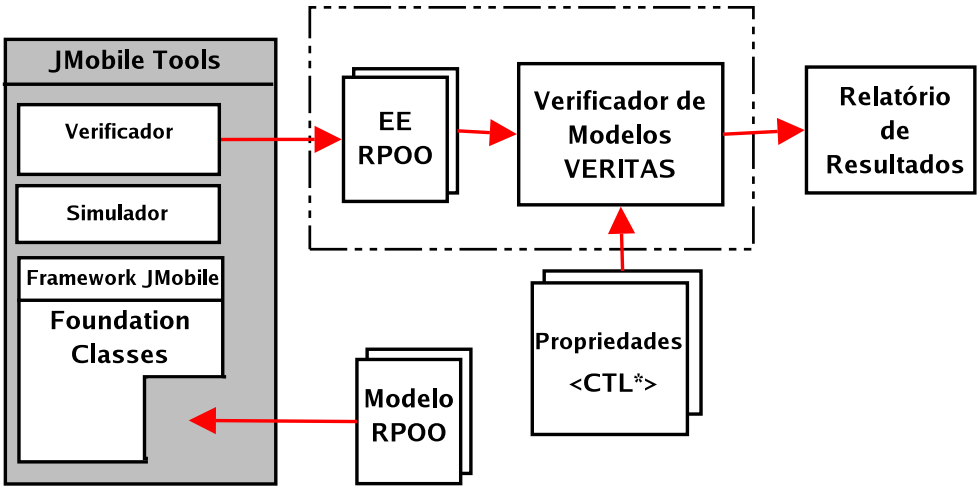


Figura 1.4: Ambiente de verificação RPOO com o *framework JMobile*

As caixas representam os vários artefatos presentes no ambiente de verificação de RPOO. A caixa de cor cinza, que contém outras internas são os artefatos desenvolvidos neste trabalho. As caixas de cor branca, isoladas são artefatos já existentes. As setas representam que os artefatos são ou geram entradas para os outros.

1.3 Estrutura da Dissertação

O restante deste documento está estruturado da seguinte forma:

Capítulo 2: Redes de Petri Orientadas a Objeto Nesse capítulo, apresentamos os fundamentos teóricos que são a base para o entendimento do trabalho desenvolvido. Nele apresentamos o formalismo RPOO que é a base teórica fundamental para o trabalho. Assim como uma breve discussão sobre a simulação de modelos e a geração de espaços de estados.

Capítulo 3: JMobile — Uma notação Java para Redes de Petri Apresentamos a notação JMobile para a concretização em *software* do formalismo RPOO. Apresentamos também, a forma de representação das classes RPOO em XML e em Java. Por último, temos as notações usadas para a representação de espaços de estados para JMobile.

Capítulo 4: A API Framework JMobile Apresentamos a API Java JMobile, sua organização e suas funcionalidades para a modelagem e simulação de modelos JMobile. Também

apresentamos, a arquitetura e as restrições atuais da API, além das possíveis extensões de funcionalidades e suporte a tipos puramente Java. No decorrer deste capítulo, também apresentamos os protótipos das ferramentas para a simulação e geração de espaço de estados de modelos JMobile. Fazemos uso do modelo do problema clássico dos Filósofos mostramos a utilização prática das ferramentas, demonstrando sua integração com o verificador de modelos Veritas, abrangendo todo o processo de validação, deste a construção do modelo até a sua verificação.

Capítulo 5: Validação Neste capítulo, apresentamos uma discussão acerca da aplicação da API JMobile na modelagem, simulação e geração de espaço de estados para alguns sistemas de forma a trazer mais confiança em sua validade.

Capítulo 6: Conclusões Neste capítulo, concluímos a dissertação, apresentando de forma objetiva os resultados e os artefatos produzidos, sumarizando a relevância do nosso trabalho dentro do contexto da verificação de modelos formais de sistemas distribuídos e concorrentes de software. Por fim, discutimos as limitações deste trabalho, bem como os desdobramentos de pesquisa e desenvolvimento que podem se tornar trabalhos futuros.

Capítulo 2

Redes de Petri Orientadas a Objeto

Neste capítulo, apresentamos os fundamentos teóricos que são a base para o trabalho desenvolvido. Na primeira seção, introduzimos o formalismo RPOO. Através de um exemplo, os principais conceitos de RPOO são apresentados. Na seção seguinte, discutimos aspectos relacionados a validação e verificação de modelos RPOO.

2.1 Redes de Petri Orientadas a Objeto

RPOO é um formalismo proposto para unir de forma ortogonal as semânticas de redes de Petri coloridas com a semântica de sistemas de objetos da orientação a objetos. A composição ortogonal de RPOO, trouxe uma alternativa diferente das propostas anteriores de integrar o formalismo das redes de Petri com o paradigma da orientação a objeto. Tentativas anteriores de unir estes dois formalismo implicavam em modificações semânticas em um dos dois níveis. Um estudo comparativo dos formalismos precursores do RPOO estão na tese de Guerrero [Gue02b].

Para RPOO, esta forma de composição, possibilita ao formalismo duas perspectivas de modelagem: i) uma perspectiva de rede de Petri colorida, cuja ênfase é na modelagem do comportamento interno das classes RPOO e, ii) uma perspectiva orientada a objeto com foco nas interações e ligações no nível de objetos.

Para apresentar de maneira informal os fundamentos do formalismo RPOO, vamos fazer uso de um exemplo bem conhecido: o problema do Jantar dos Filósofos. Neste problema clássico temos ao redor de uma mesa cinco filósofos e cada filósofo tem um garfo que é

compartilhado com seu vizinho esquerdo.

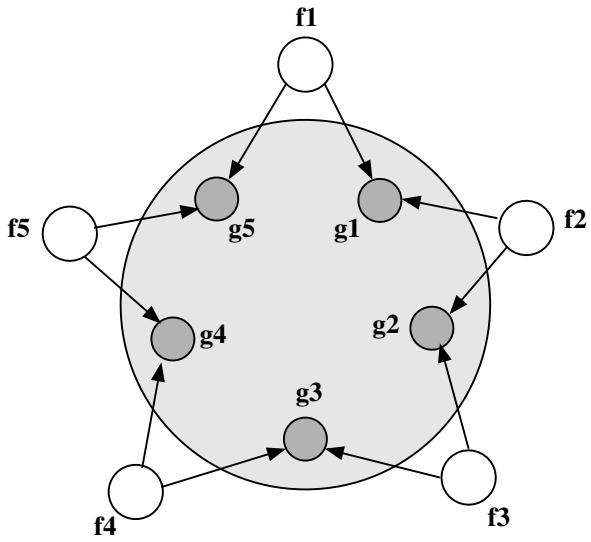


Figura 2.1: Mesa com 5 filósofos e 5 garfos

Na Figura 2.1, os filósofos (f1, f2, f3, f4, f5) são representados por círculos de cor branca na parte mais externa da figura. Já os garfos (g1, g2, g3, g4, g5) são representados por círculos de cor cinza e estão na parte mais interna da figura, em cima da mesa. Cada filósofo tem dois estados possíveis, podendo estar em apenas um deles em cada instante, ou seja, ou está pensando ou está comendo. Assim como o filósofo, um garfo também pode estar em um dois estados: ou ele está livre ou em uso. Os arcos representam que os filósofos podem pegar e soltar os respectivos garfos indicados. Para passar do estado pensando para o estado comendo, o filósofo precisa pegar o seu garfo e o garfo do vizinho da direita, desde que eles estejam livres. Ao passar do estado comendo para o estado pensando, o filósofo libera os dois garfos.

A proposta do formalismo RPOO é usar os artefatos convencionais da Orientação a Objeto para modelar as características externas do sistema. Desta forma, como também para a construção de modelos na Orientação a Objetos, um dos conceitos mais importantes é o conceito de *classe*. No problema dos filósofos, temos duas classes de objetos: a classe *Filósofo* e a classe *Garfo*. O filósofo pode realizar as ações *"pensar"* e *"comer"*, já os garfos podem ser *"pegos"* ou *"largados"* por um dos filósofos. Na Figura 2.2, apresentamos o diagrama de classes UML [OMG03] para o problema do jantar dos Filósofos.

Na modelagem usando RPOO, o comportamento das classes é modelado utilizando uma

Figura 2.2: Diagrama de Classes para o Problema dos Filósofos

rede de Petri colorida modificada. Uma rede de Petri colorida (CP-net) é um grafo bi-partido dirigido usado para representar o comportamento de sistemas. Os principais elementos das rede de Petri são os "*lugares*", as "*transições*" e os **arcos**. Os "*lugares*" são representados graficamente por círculos e são responsáveis por manterem o estado através do acúmulo de dados (chamada de **fichas**). As "*transições*" são representadas graficamente por retângulos e são responsáveis pelas mudanças de estados do sistema. Os **arcos** ligam lugares a transições e transições a lugares. Os arcos contém expressões que definem que fichas serão retiradas ou colocadas nos lugares. Uma transição está habilitada a disparar quando todas as suas pré-condições são avaliadas como verdadeiras. Uma das pré-condições são as expressões nos arcos da transição que podem ser avaliadas para alguma(s) fichas nos lugares ligados a transição. Além das expressões nos arcos, as redes de Petri permitem outras pré-condições para o disparo de uma transição. Podemos acrescentar *guardas* que são expressões *booleanas* relacionadas com as fichas (dados) nos lugares. Em RPOO, as CP-nets são modificadas através de inscrições nas transições que representam ações de interação entre instâncias de objetos das classes do modelo. Existem sete tipos diferentes de inscrições, como detalhado na Tabela 2.1.

As Figuras 2.3 e 2.4 apresentam as redes de Petri orientadas a objeto que descrevem os comportamentos das classes *Filósofo* e *Garfo* do problema do jantar dos filósofos. No modelo da classe *Filósofo*, os dois lugares modelam os dois possíveis estados de um filósofo. Uma ficha de cor ***TOKEN*** em um dos dois lugares define qual o estado corrente de um filósofo. O nó de declaração (***SigmaLocal*** contém a definição dos tipos de fichas e variáveis que são utilizadas no modelo. As duas transições modelam as duas possíveis ações que podem ser executadas por um filósofo. A transição ***comer*** possui duas inscrições de saída síncrona ***ge!pegar();*** e ***gd!pegar()***. De acordo com o nó de declaração, ***ge*** e ***gd*** são objetos do tipo Garfo. As duas inscrições de saída síncrona indicam que a esta transição deve disparar de

Ação	Nome	Pré-Condição	Efeito
τ	ação interna		Nenhuma alteração no sistema de objetos. Ela é referente ao comportamento da rede de Petri do objeto
new x	criação ou instanciação de objetos	Identificador da nova instância não existe	Novo objeto RPOO é adicionado ao sistema de objetos e uma *Ligação* entre "criador" e objeto instanciado é adicionada
$x.m$	Saída Assíncrona	Agente deve possuir ligação com destinatário	Criação de mensagem pendente. Objeto não aguarda consumo da mensagem
$x!m$	Saída Síncrona	Agente deve possuir ligação com destinatário	Criação e consumo de mensagem pendente. Objeto aguarda consumo da mensagem
$x?m$	Entrada de Dados	Deve haver mensagem pendente destinada ao agente da ação	Mensagem pendente consumida é excluída do sistema de objetos
$\tilde{x}$	Desligamento	Deve existir a ligação	Ligação entre o agente da ação e o objeto referenciado é excluída
end	Auto-destruição		Agente se destrói. Ligações cuja origem é o objeto agente são excluídas

Tabela 2.1: Tipos de ações elementares com efeito sobre o modelo

forma síncrona com a transição ***pegar*** de duas instâncias de objetos (***ge*** e ***gd***) da classe Garfo. A transição ***pensar*** possui duas inscrições de saída assíncrona. No modelo da classe *Garfo*, cada uma das trasições tem uma inscrição de entrada de dados (***fil?pegar()*** e ***fil?largar()***).

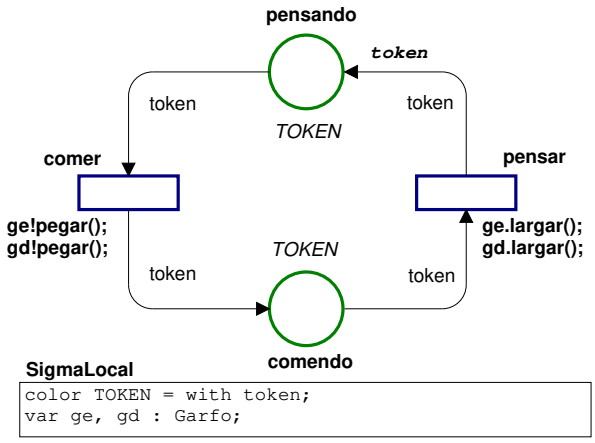


Figura 2.3: Rede para o comportamento dos Filósofos

Para completar a modelagem RPOO, é necessário definir um sistema de objetos. Em um *Sistema de Objetos* (SO) temos a representação de todos os objetos do sistema, ligações

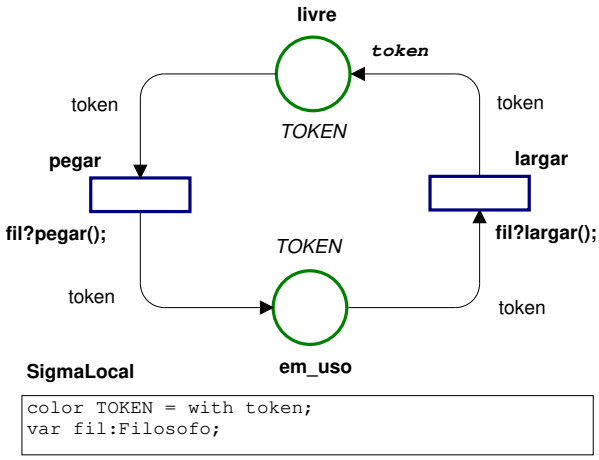


Figura 2.4: Rede para o comportamento dos Garfos

entre eles e eventuais mensagens pendentes representando um estado estrutural do sistema. O Sistema de Objetos representa o estado do sistema em nível de interação externa entre os objetos, ficando a parte interna do comportamento de cada objeto encapsulada nas suas respectivas redes. O termo *Estrutura* será utilizado para denotar um estado do sistema em nível de objeto. O termo *Configuração* será utilizado para denotar um estado do sistema em RPOO, ou seja, estamos interessados na estrutura do SO e no estado interno dos objetos.

Na Figura 2.5 temos a representação gráfica para a estrutura inicial do sistema de objetos para um modelo de solução do problema dos filósofos. Os arcos que partem de ***f1*** para ***g1*** e ***g5*** indicam que o objeto da classe ***f1*** conhece ou tem ligações com os objetos ***g1*** e ***g5***. Os objetos ***g1***, ***g2***, ***g3***, ***g4*** e ***g5*** não têm ligação com outros objetos.

Uma outra forma de apresentar a estrutura do sistema de objetos é utilizar uma representação algébrica. No trabalho de Guerrero [Gue02b] é apresentada uma álgebra para a representação das configurações e das regras para sua evolução. Nesta álgebra temos a definição formal das ações de interação presentes nos modelos RPOO apresentadas na Tabela 2.1.

A representação algébrica para a estrutura da Figura 2.5.

```
Estrutura = f1[g5 g1] + f2[g1 g2] + f3[g2 g3] +
            f4[g3 g4] + f5[g4 g5] + g1 + g2 + g3 + g4 + g5
```

Na representação algébrica, quando escrevemos *f1[g5 g1]*, estamos informando que o objeto ***f1*** da classe *Filósofo* conhece ou tem ligações com os objetos ***g5*** e ***g1*** da classe *Garfo*. Quando escrevemos apenas ***g1***, estamos informando que o objeto não tem ligações para

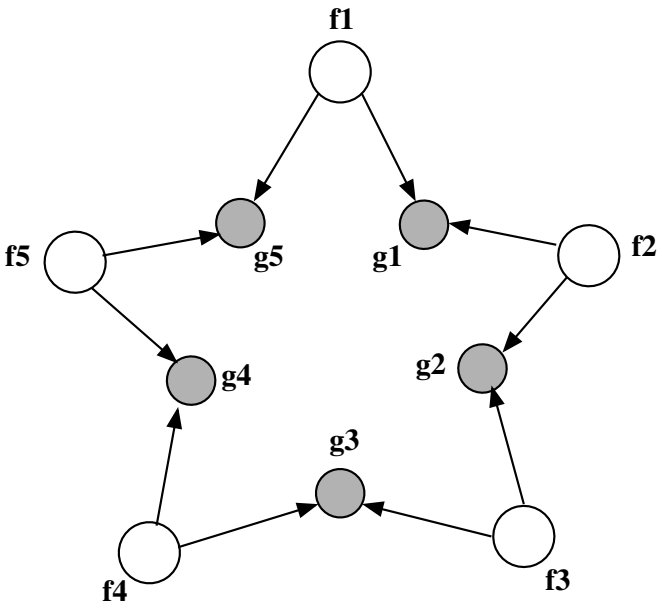


Figura 2.5: Estrutura Inicial do Sistema de Objetos dos Filósofos

outros objetos. O sinal de adição (+) serve apenas como delimitador entre os objetos e indicam que fazem parte da mesma estrutura.

A Figura 2.6 apresenta a configuração inicial do modelo RPOO, ou seja, a estrutura com estados internos. Cada um dos 5 filósofos está no estado pensando (rótulo ***p***) e cada garfo está livre (rótulo ***l***).

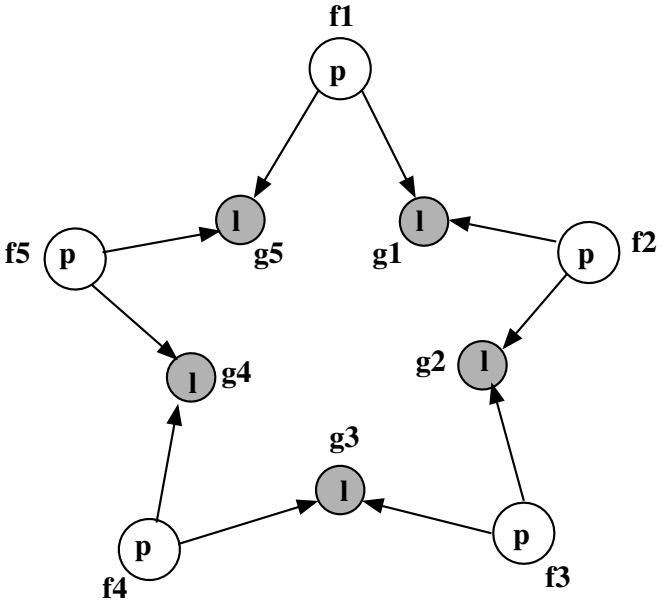


Figura 2.6: Configuração inicial do modelo dos Filósofos

A representação algébrica para a configuração é dada por:

```
Configuração = f1(pensando{token})[g5 g1] + f2(pensando{token})[g1 g2] +
       f3(pensando{token})[g2 g3] + f4(pensando{token})[g3 g4] +
       f5(pensando{token})[g4 g5] + g1(livre{token}) + g2(livre{token}) +
       g3(livre{token}) + g4(livre{token}) + g5(livre{token})
```

Nesta representação algébricam vemos que no lugar *pensando* dos filósofos temos uma ficha tipo *token* e no lugar *livre* dos garfos temos também um *token*. Desta forma, representamos o estado inicial da configuração onde os filósofos estão pensando e os garfos estão livres.

Em alguns casos, estamos interessados em analisar o modelo apenas em nível estrutural e outras vezes queremos analisar em detalhes o estado do sistema modelado. Logo, algumas vezes podemos omitir na representação algébrica o estado interno dos objetos da configuração.

O comportamento de um sistema de objetos é dado pelas modificações que podem ser observadas sobre sua estrutura, em função da ocorrência de ações nos objetos que o compõem. Estas ações são geradas a partir da mudança do estado interno das redes de Petri orientadas a objeto. As alterações na estrutura de um sistema de objetos acontecem de acordo com um conjunto de regras que descrevem cada ação, já apresentada na Tabela 2.1.

No problema dos Filósofos só temos três tipos de ações RPOO: ação de envio de mensagens (ações de saída assíncrona e síncrona) e ação de consumo de mensagens (ação de entrada de dados), como podemos verificar nas Figuras 2.3 e 2.4. Para passar do estado *"pensando"* para o estado *"comendo"*, um filósofo manda mensagens síncronas para os dois garfos. Por exemplo, se o filósofo *"f1"* disparar a transição *comer* que contém as inscrições ***ge!pegar(); gd!pegar()***, como *"f1"* conhece *"g1"* e *"g5"* as variáveis *"ge"* e *"gd"* serão ligadas aos garfos. Desta forma, serão geradas as seguintes mensagens:

f1:g1(pegar()) + f1:g5(pegar())

Como estas mensagens são síncronas, os garfos são obrigados a consumi-las imediatamente (de forma atômica). O consumo destas mensagens pelos garfos se dá através do disparo da transição *pegar* dos garfos "g1" e "g5", que contém a inscrição ***fil?pegar();***. Ou seja, as quatro ações são executadas de uma só vez. O termo *evento* é usado para definir

a composição de ações. Um evento é formado por um conjunto de uma ou mais ações. O evento formado no disparo da transição *comer* do filósofo *"f1"* é:

evt1 = f1!g1.pegar() & f1!g5.pegar() & g1:f1?pegar() & g5:f1?pegar()

Ao executarmos o evento ***evt1***, a estrutura do sistema de objetos da configuração em si não se altera, mas apenas o estado interno do filósofo e do garfo. A nova configuração é mostrada na Figura 2.7. As letras dentro dos círculos que representam os objetos significam respectivamente: **p** – pensando, **c** – comendo, **u** – em uso, e **l** – livre.

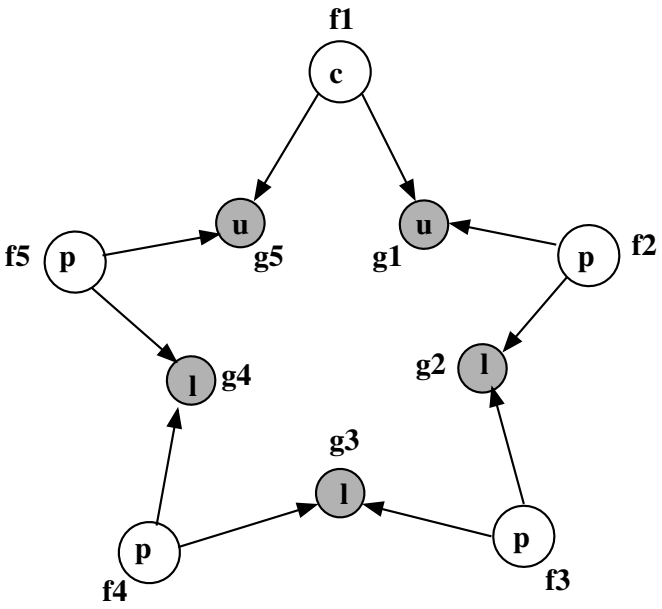


Figura 2.7: Estrutura dos Filósofos com o estado interno dos objetos depois da execução do evento

Depois da execução do evento **evt1**, o filósofo *"f1"* está no estado *"comendo"*, como mostra a Figura 2.7. Se a ação representada pelo disparo da transição *pensar* ocorrer, o filósofo volta ao estado *"pensando"*. Com este disparo a estrutura do sistema de objetos será alterada. A nova configuração contempla duas mensagens pendentes: uma mensagem para *"g1"* e outra para *"g5"* contendo a *string "largar"*, como é mostrado na Figura 2.8.

A representação algébrica para o sistema de objeto da Figura 2.8 é apresentada a seguir:

```
Configuração = f1[g5 g1] + f2[g1 g2] + f3[g2 g3] +
f4[g3 g4] + f5[g4 g5] + g1 + g2 + g3 + g4 + g5 +
f1:g1(largar()) + f1:g2(largar())
```

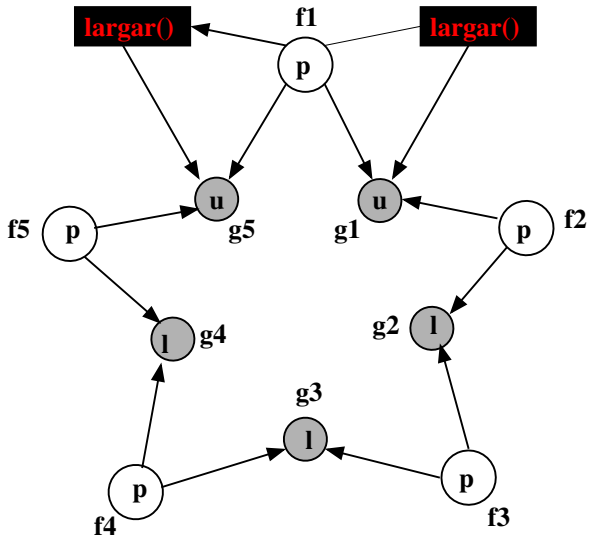


Figura 2.8: Configuração do Sistema de Objetos com mensagens pendentes

Quando escrevemos *f1:g1(largar())*, estamos representando a existência de uma mensagem pendente para *"g1"* vinda de *"f1"* contendo *"largar()"*.

Mesmo com o filósofo *"f1"* voltando ao estado *"pensando"* os estados dos garfos ainda são os mesmos. Com as mensagens *f1:g1(largar())* e *f1:g2(largar())* pendentes, a transição *"largar"* dos garfos ficam habilitadas, pois esta transição contém a inscrição ***fil?largar()***, que é uma ação de entrada (consumo de mensagem). Após o disparo desta transição nos dois garfos, a configuração volta para o estado inicial que foi apresentado na Figura 2.6.

2.2 Ferramentas de Simulação e Verificação de Modelos RPOO

O processo de simulação de modelos é muito importante para a validação de um sistema modelado com redes de Petri. Através da simulação de vários cenários (traces) de comportamento do modelo, podemos observar diversos aspectos do sistema modelado verificando diferentes tipos de propriedades. Desta forma, adquirimos uma maior confiança de que o modelo está correto.

A simulação dos modelos RPOO é o foco principal do desenvolvimento das ferramentas de suporte para RPOO. Com o suporte ferramental para a simulação dos modelos, RPOO passaria a um novo nível de utilização inexistente até a conclusão do trabalho descrito neste

documento.

O processo de simulação de modelos RPOO acontece em dois níveis devido a sua própria definição. Ou seja, temos a simulação em nível de redes de Petri coloridas e a simulação em nível de sistemas de objetos. Desta forma, uma simulação de um Modelo RPOO é a composição destas duas simulações e a simulação em nível de sistema de objetos é o reflexo da simulação das redes de Petri.

Esta composição permite visualizar a simulação apenas no nível de objetos, e desta forma facilita a observação de comportamentos de interesse do engenheiro de software sem a necessidade dos detalhes internos do comportamento dos objetos simulados pelas redes de Petri. A partir da construção do modelo e da marcação inicial (estado inicial) das redes de petri, são determinadas as transições habilitadas e por conseqüência determinamos as ações RPOO habilitadas.

Dado a configuração inicial do modelo e determinadas as ações habilitadas a executar, podemos simular o modelo. Com a simulação do modelo poderemos validar o comportamento do sistema para alguns casos e cenários de interesse.

As ferramentas utilizadas na simulação de modelos RPOO são o Design/CPN e o Simulador de Sistemas de Objetos — SSO. O Design/CPN é uma ferramenta completa para a edição, simulação e análise de Redes de Petri Coloridas. O SSO foi construído para simular a evolução dos sistemas de objetos de modelos RPOO. Como as duas ferramentas foram construídas para propósitos diferentes e não para serem usadas na construção e simulação de modelos RPOO, para simular um modelo RPOO usando o Design/CPN e o SSO, o engenheiro do software tem que executar independentemente as duas ferramentas e atuar como módulo integrador da ferramenta. O engenheiro, ao simular o modelo, deve a cada passo na simulação das Redes de Petri gerar manualmente um evento no sistema de objetos na ferramenta SSO, e depois, voltar ao Design/CPN e gerar os eventos e marcações nas Redes de Petri dos objetos afetados pelo o evento.

2.2.1 Espaços de Estados e Verificação de Modelos

O espaço de estados ou grafo de alcançabilidade é um dos principais meios para a verificação completa do software. Sabemos que para uma validação mais rigorosa precisamos simular o modelo seguindo todas as suas possibilidades de execução e a simulação de alguns poucos

cenários não cobrem todas as possibilidades de estados do modelo de um sistema.

O espaço de estados de um modelo nada mais é do que o conjunto de todos os estados possíveis que um sistema pode assumir. No caso de RPOO, o espaço de estados corresponde ao conjunto de todas as configurações alcançáveis a partir da configuração inicial, considerando a ocorrência de todos os possíveis eventos.

Como exemplo, considere uma instância do modelo dos filósofos utilizado na Seção 2.2, com dois filósofos e dois garfos. Uma representação abstrata do espaço de estados se encontra na Figura 2.9 Para esta instância do problema e, considerando a configuração inicial representada na figura pela configuração *1*, é possível se chegar a 9 configurações diferentes.

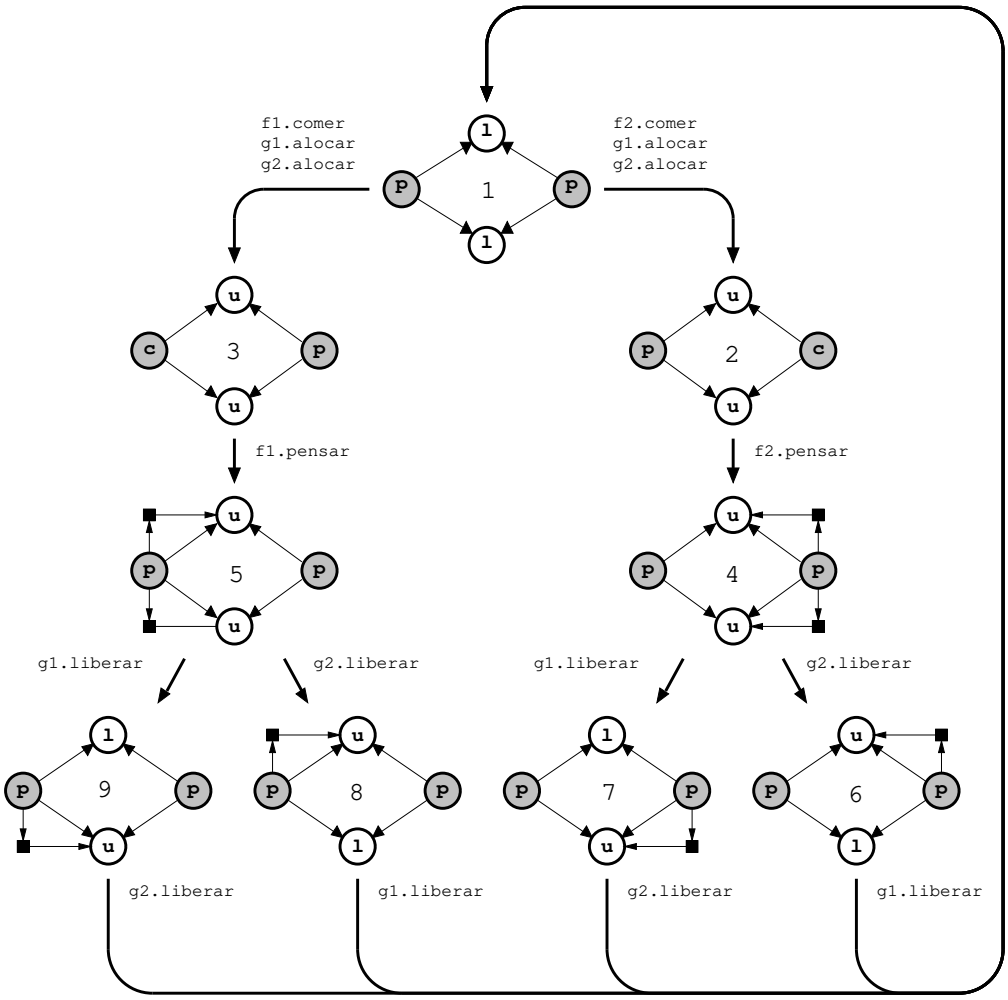


Figura 2.9: Representação gráfica do espaço de estados.

Veritas [Rod04] é um verificador de modelos RPOO. A partir do espaço de estados de um modelo RPOO e propriedades especificadas em lógica temporal, o Veritas permite verificar

se estas propriedades são satisfeitas ou não. A geração automática do espaço de estados de um modelo RPOO é um ponto crucial para que a técnica de verificação de modelos seja usada em sua plenitude no desenvolvimento de sistemas baseados em RPOO.

2.3 Conclusão

A não existência de ferramentas adequadas para a simulação e verificação de modelos RPOO, dificulta o emprego do formalismo na prática. O trabalho manual dificulta a utilização do formalismo RPOO, principalmente em modelos com uma certa quantidade de objetos de diferentes classes (diferentes redes de Petri para o comportamento das classes). Esta dificuldade e a necessidade de uma ferramenta exclusiva para RPOO são os principais motivos para a proposta de nosso trabalho.

Uma ferramenta que unisse as funcionalidade de simulação das redes de Petri e do Sistema de Objetos, permitindo a geração automática do espaço de estados iria contribuir no crescimento da utilização do formalismo. Além disso, teríamos cada vez mais modelos e contribuições para aprimoramento do formalismo RPOO. No Capítulo 4 apresentamos a API de suporte JMobile, sua utilização e os protótipos das ferramentas.

Capítulo 3

JMobile - Uma notação Java para Redes de Petri

Neste capítulo, apresentamos a notação JMobile. Apresentamos também, a forma de representação das classes JMobile em XML e em Java. Por último, temos as notações usadas para a representação de espaço de estados de modelos JMobile. Para apresentar a linguagem e a notação, usamos o problema clássico do jantar dos filósofos.

3.1 Anotando redes de Petri orientadas a objetos com Java

No Capítulo 2 apresentamos o formalismo de RPOO na forma em que foi definido por Guerrero [Gue02b]. Nesta sessão, apresentaremos uma adaptação dessa notação, a que chamamos JMobile[1]. JMobile é uma notação alternativa para redes de Petri OO baseadas em Java.

A formalização de redes de Petri OO depende de uma linguagem formal de suporte que permita descrever os tipos de dados manipulados no modelo e expressões que caracterizam as transformações de dados necessárias. Na formalização abstrata de RPOO, usamos especificações algébricas para a descrição de dados e de expressões. Na versão concreta, optamos por utilizar ML, devido à flexibilidade que uma linguagem funcional oferece ao modelador. Esta escolha permitiu ainda que parte dos modelos fosse facilmente editada e simulada no Design/CPN, ferramenta de simulação de redes CPN, que usam ML para a descrição e

[1]O nome JMobile foi inspirado no nome do projeto de pesquisa *MOBILE*. O "J" é a clássica referência à linguagem de programação Java.

manipulação de dados.

Se por um lado, tal linguagem oferece mais facilidade ao modelador, a linguagem requer treinamento especial e tende a distanciar o modelo do sistema modelado. A adaptação para a sintaxe Java tornou-se interessante pois a linguagem Java de programação orientada a objetos está sendo utilizada nas mais diversas áreas para o desenvolvimento de *software*. Na área comercial, principalmente direcionada para a *Web*, ela é usada em larga escala. Além disso, o paradigma funcional, mesmo sendo um dos paradigmas mais elaborados e utilizados no desenvolvimento rigoroso de software por ser formalmente definido, é pouco utilizado para o desenvolvimento de software fora da academia. Outra questão na escolha de uma notação com sintaxe Java, foi a questão da complexidade de representarmos as redes de Petri usando Java e com uma notação funcional a complexidade de representação aumentava, pois precisaríamos representar a forma de avaliação funcional declarativa com uma linguagem imperativa.

Dentro deste contexto, apresentamos nesta seção a adaptação das redes de Petri orientada a objetos para uma sintaxe baseada na sintaxe da linguagem Java. É importante ressaltar que esta adaptação das redes de Petri é simplificada e com alguns detalhes das definições serão omitidos. Nesta primeira versão da notação restringimos os conceitos tratados das redes de Petri a um subconjunto que permitisse a representação de um bom número de características de modelos RPOO.

Através da concretização de RPOO com a notação JMobile teremos reduzido a lacuna entre o formalismo RPOO e seu uso na prática da Engenharia de Software. Inicialmente apresentamos as modificações da notação nas redes de Petri no JMobile através da comparação dos atributos dos elementos das redes de Petri. Apresentamos os atributos de lugares, transições e arcos que serão modificados e, em seguida, na Seção 3.3, apresentaremos o modelo dos filósofos, definido na página 13 do Capítulo 2, na notação com sintaxe Java. Os atributos dos componentes estruturais das redes de Petri que foram adaptados são apresentados a seguir:

- Lugares
  - place_name — Nome do lugar;
  - ColorSet — Tipo do lugar;

  - marking — Marcação, um MultiSet do tipo ColorSet do lugar;

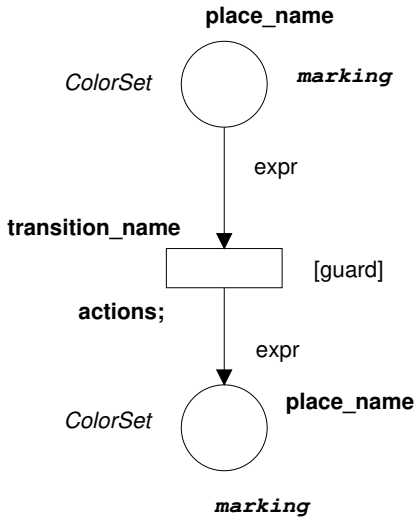
- Transições

  - transition_name — Nome da transição;
  - actions — Ações de interação entre objetos;
  - guard — Guarda, uma expressão booleana de teste;

- Arcos

  - expr — Expressões MultiSet's sobre os tipos e valores dos lugares.

A Figura 3.1 mostra uma rede de Petri colorida com os atributos de lugares, transições e arcos usadas nos modelos RPOO.



SigmaLocal

```
#Definicao de Variaveis e Constantes.
#Definicao de ColorSets e Funcoes.
#Notacao e sematica de programacao funcional (SML).
color colorSetName = "tipos ML"
var var_name: ColorSet;
val const_name : ColorSet;
```

Figura 3.1: Rede de Petri colorida — notação ML

A Figura 3.2 mostra uma rede de Petri colorida com os atributos de lugares, transições e arcos usando a notação JMobile.

Tal como na versão original de RPOO, lugares e transições podem ser rotulados para que melhor expressem seu significado no modelo. Além de um nome, cada lugar da rede deve

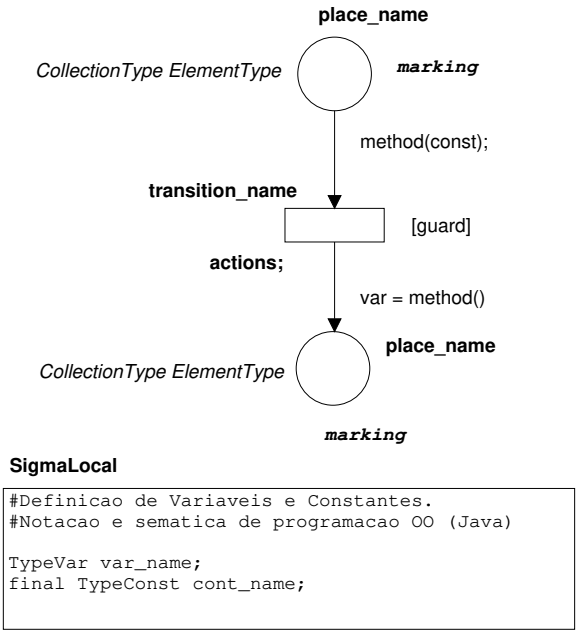


Figura 3.2: Rede de Petri Java — notação Java

ser associado a um tipo de dados que restringe as fichas que poderá armazenar—na versão original de RPOO, tais tipos são chamados de *Color Sets*. Em JMobile, o tipo associado aos lugares deve ser uma coleção de objetos baseada em algum tipo elementar. Transições podem ser inscritas com expressões de interação entre objetos. Tais inscrições restringem o disparo de uma transição, forçando a sincronização do objeto com eventos ou condições externas. (ver Tabela 2.1 da página 2.1 do Capítulo 2).

A diferença mais significativa entre RPOO e JMobile é a sintaxe para a escrita de expressões de arcos. Em JMobile, expressões de arcos devem expressar chamadas a métodos para a coleção representada pelo lugar. Como o lugar é associado a um tipo Java que representa uma coleção podemos chamar métodos definidos em sua interface.

3.2 A notação das ações JMobile

Nesta seção, apresentamos a notação usada para expressar ações. Ações são expressas por inscrições acrescentadas às transições das redes de Petri que modelam o comportamento das classes RPOO. Na Tabela 3.1, temos as notações das ações RPOO que serão apresentadas na Seção 3.3.

O ***this*** é o objeto agente da ação, esta parte fica implícita quando inscrita numa transição,

Ação	Notação	Exemplo
Interna	[this:]#	cf:#
Criação	[this:]varRPOO = new ClasseRPOO(nome);	cf:ge = new Garfo(“g1”);
Saída Assíncrona	[this:]varDestino.nomeMSG(varParametro);	f1:ge.pegar(); ou cf:fe.link(ge);
Saída Síncrona	[this:]varDestino!nomeMSG(varParametro)	f1:ge!pegar(); ou cf:fe!link(ge);
Entrada de Dados	[this:]varOrigem?nomeMSG(varParametro)	g1:fil?pegar(); ou f1:cons?link(ge);
Desligamento	[this:]unlink(varObjetoRPOO)	cf:unlink(ge);
Auto-destruição	[this:]End()	cf:End()

Tabela 3.1: Tipos de ações elementares com efeito sobre uma configuração

pois se refere ao objeto da classe definida pela própria rede de Petri que contém a transição e que está executando a ação. O caractere “: é usado como separador entre o agente e o a ação propriamente dita.

Ação Interna Ações internas são ações que não precisam ser sincronizadas com eventos ou condições internas. Qualquer transição que não seja inscrita com uma expressão de interação expressa uma ação interna.

Ação de Criação A ação de criação ou de instanciação de objetos RPOO é composta de:

- this: - O objeto agente da ação;
- varRPOO - Variável definida no nó de declarações da rede do mesmo tipo do objeto RPOO, que receberá a instância de objeto da classe RPOO;
- new - palavra reservada que defina a operação de instanciação;
- ClasseRPOO - A Classe RPOO do objeto a ser criado;
- nome - String com o nome do objeto.

Ao ser executada, além de criar um novo objeto RPOO e adicioná-lo a configuração, esta ação cria uma ligação (*link*) RPOO entre o objeto agente da ação e o objeto criado.

Ação de Saída Assíncrona

- this: - O objeto agente da ação;
- varDestino - Variável definida no nó declarações da rede de alguma classe RPOO. Esta variável será ligada com um dos objetos RPOO conhecidos pelo objeto agente, ou seja, deve existir uma ligação entre o agente e o destino;
- "." - O "ponto" é o delimitador entre o destino e a mensagem na ação assíncrona;
- nomeMSG - Nome da mensagem a ser enviada para o destino. Esta mensagem deve estar na interface da classe do objeto RPOO destino;
- varParametro - Variável de algum tipo RPOO que guarda o conteúdo da mensagem a ser enviada, pode ser um Objeto RPOO.

Ao ser executada esta ação cria uma mensagem pendente na configuração do sistema. Esta mensagem fica na configuração até que alguma ação de consumo ocorra, podendo esta ação nunca acontecer.

Ação de Saída Síncrona A única diferença na ação de saída síncrona, em relação a ação de saída assíncrona, é o sinal de "exclamação" ("!") que é o delimitador entre o destino e a mensagem da saída síncrona. Ao ser executada esta ação cria uma mensagem pendente na configuração do sistema. Contudo, esta mensagem pendente deve ser consumida imediatamente, ou seja, no mesmo evento através de uma ação de consumo.

Ação de Entrada

- this: - O objeto agente da ação;
- varOrigem - Variável definida no nó declarações da rede de alguma classe RPOO. Esta variável será ligada com um dos objetos RPOO que enviaram uma mensagem para o agente desta ação.
- "?" - O sinal de "interrogação" é o delimitador entre o objeto origem e a mensagem.
- nomeMSG - Nome da mensagem a ser enviada para o destino. Esta mensagem deve estar na interface da classe do objeto RPOO agente.

- varParametro - Variável de algum tipo RPOO, podendo ser até um Objeto RPOO que guarda o conteúdo da mensagem a ser enviada.

Ao ser executada esta ação cria uma mensagem pendente na configuração do sistema. Contudo, esta mensagem pendente deve ser consumida imediatamente, ou seja, no mesmo evento através de uma ação de consumo.

Ação de Desligamento A variável "varObjetoRPOO" é de alguma classe RPOO, onde existe a ligação do agente desta ação para o objeto na variável varObjetoRPOO. Ao ser executada esta ação remove a ligação entre o agente e o objeto RPOO passado na variável.

Ação de Auto-destruição Ao ser executada esta ação o objeto agente é removido da configuração do modelo passando a ser uma referência nula. Além disso, as ligações com origem neste objeto deixam de existir.

3.3 Modelo dos Filósofos na linguagem RPOO/JMobile

Nesta seção, apresentaremos o modelo dos Filósofos (definido em RPOO na Seção 2.1 da Página 12) usando a notação da linguagem JMobile (definida na Seção 3.1). No decorrer da apresentação do modelo iremos detalhar as mudanças entre o formalismo RPOO e sua concretização na linguagem JMobile.

O modelo dos Filósofos que iremos utilizar no decorrer desta seção foi modificado para que exigisse o uso de todas as ações RPOO, detalhadas na Tabela 3.1 da Página 28. Para isso, acrescentamos uma nova classe RPOO, chamada *Construtor*, que através de um conjunto de ações constrói a mesa para o jantar dos filósofos. Ou seja, o construtor constrói a configuração inicial para o modelo dos Filósofos. Assim, ao final da execução do objeto da classe construtor teremos a configuração com dois filósofos e dois garfos.

Na Figura 3.3 temos o diagrama de classes que do modelo dos Filósofos Modificado estendendo o modelo clássico do problema dos Filósofos apresentado na Figura 2.2 da Página 14. Além de acrescentarmos a classe RPOO "Construtor" ao modelo, nós estendemos a classe *Filosofo* na classe *FilosofoModificado* com a adição do método *receberLigacoes* para receber as ligações enviadas pelo *Construtor* através do método *enviarLigacoes*. Este exem-

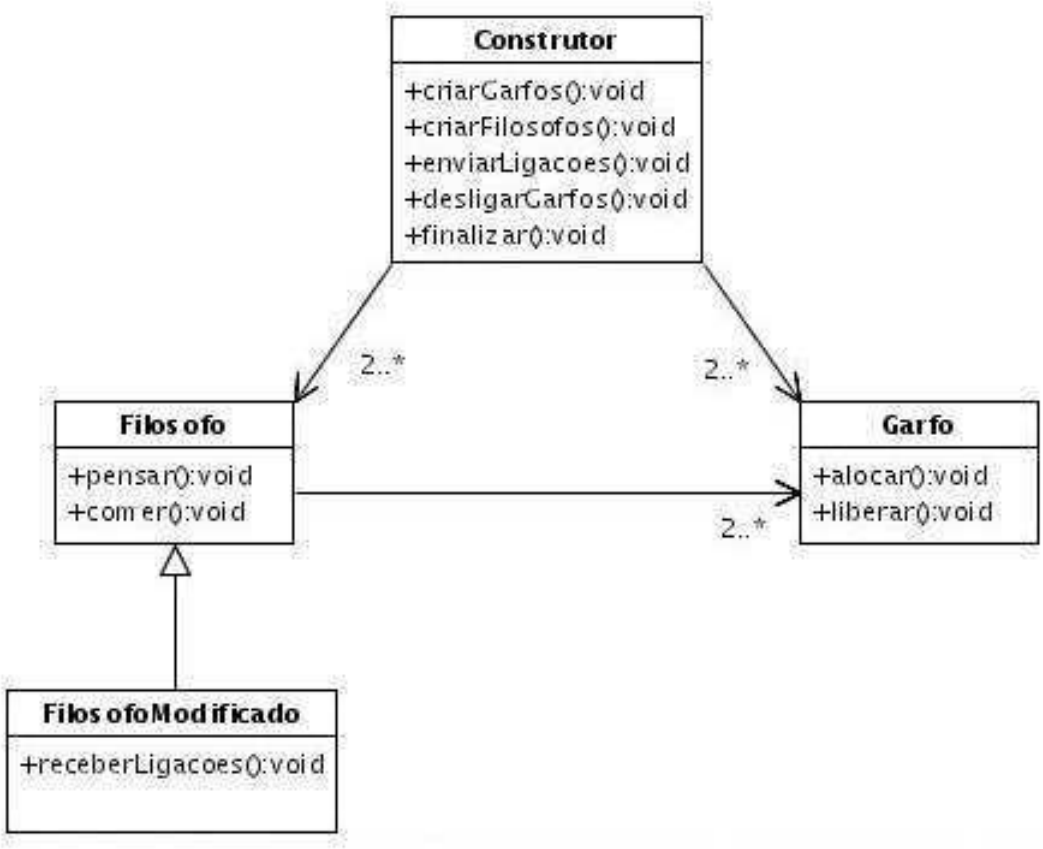


Figura 3.3: Diagrama de Classes para o Problema dos Filósofos Modificado

plo de herança em RPOO, mesmo bem simples, é muito importante. Desta forma, em alguns modelos poderemos aproveitar muitas classes RPOO em novos modelos, sem a necessidade de construção completa de novas classes repetindo comportamentos já modelados em outras classes. O uso de herança ressalta uma das características mais importantes de RPOO, que é uma melhor organização e decomposição dos modelos.

Na Figura 3.4, apresentamos a rede de Petri que modela a classe *Construtor*.

Desta forma a configuração inicial do modelo filósofo modificado é apenas um objeto da classe *Construtor* com um token no lugar *"passo0"*, chamaremos de *cf*. A Figura 3.5 mostra a configuração inicial para o modelo dos filósofos modificado. Acima da linha está a representação gráfica de uma configuração e abaixo da linha está a representação algébrica da mesma configuração. A partir deste ponto, quando a figura for de uma configuração iremos representa-lá desta forma.

A rede da classe Construtor apresenta a maioria das ações RPOO, e a cada disparo de

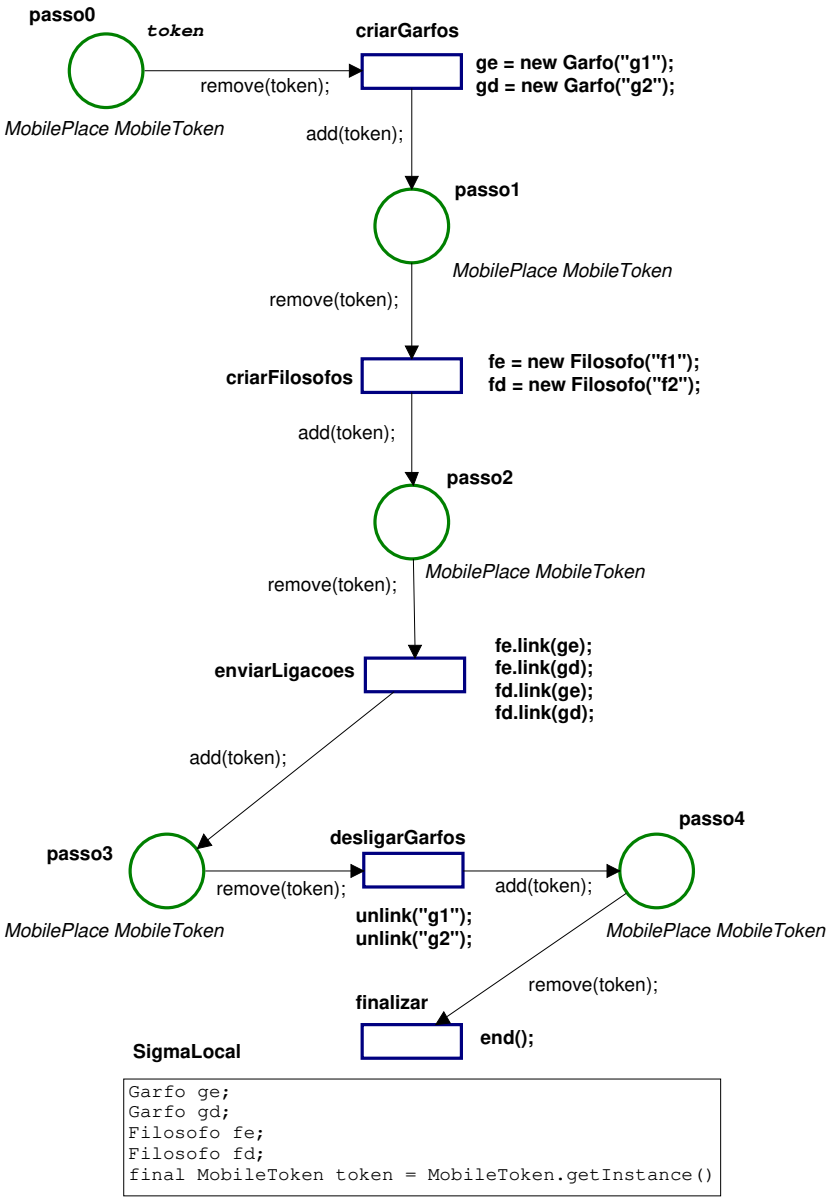


Figura 3.4: Rede para o comportamento do Construtor

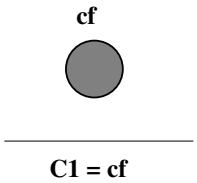


Figura 3.5: C1 - Configuração Inicial dos Filósofos Modificado

transição ele constrói a configuração inicial do modelo dos Filósofos. A classe começa com um ***token*** no lugar *"passo0"*. Note que, as expressões nos arcos da rede de Petri são semelhantes as chamadas a métodos Java, isto significa que estamos chamando um método

da interface do tipo *"Lugar"*. As chamadas nos arcos são feitas para os métodos:

- *remove(elemento)* - Expressão de arco que remove um elemento do lugar de entrada deste arco. O elemento a ser removido deve ser do mesmo tipo do lugar de entrada.
- *add(elemento)* - Expressão de arco que adiciona um elemento ao lugar de saída deste arco. O elemento a ser adicionado deve ser do mesmo tipo do lugar de saída.

No decorrer desta seção apresentaremos outras chamadas de métodos da interface do tipo *"Lugar"* e no final desta seção apresentaremos toda a interface.

A única transição habilitada na rede Construtor Figura 3.4, neste estado inicial, é a transição *"criarGarfos"*. Quando a transição *"criarGrafos"* disparar iremos remover o ***token*** do lugar *"passo0"* e adicionar um ***token*** no *"passo1"*. A ação RPOO no lugar *"criarGarfos"* é uma ação de criação, ou seja, é uma ação de instanciação de objetos RPOO da classe Garfo Figura 3.6.

ge = new Garfo("g1");
gd = new Garfo("g2");

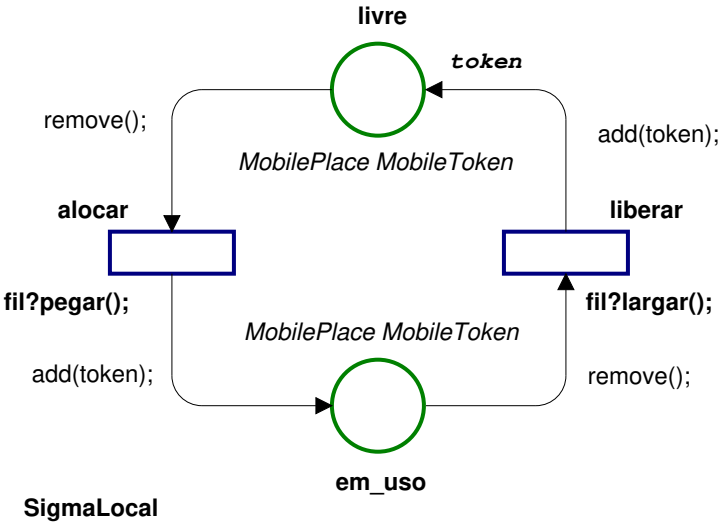


Figura 3.6: Rede para o comportamento dos Garfos

Com esta ação de criação estamos criando dois garfos *g1* e *g2*, e atribuindo-os as variáveis do tipo Garfo *ge* e *gd* definidos no nó de declarações chamado de SigmaLocal no

JMobile. No nó de declarações temos também a definição das variáveis *fe* e *fd*, além da constante *token*. Note que estamos utilizando o estilo e a sintaxe Java para a definição de variáveis e constantes no nó de declarações. Na classe Garfo temos uma outra chamada a métodos, é a chamada para o método *remove()* sem parâmetros, que removerá um elemento aleatoriamente do lugar de entrada. Depois do disparo da transição *"criarGarfos"* temos a seguinte configuração, na representação textual e na representação em sistema de objetos.

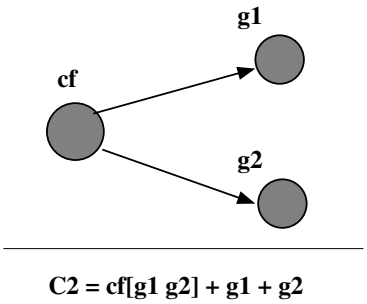


Figura 3.7: C2 - Configuração após o disparo da transição *"criarGarfos"*

Em seguida, temos a transição *"criarFilosofos"* habilita na rede *Construtor*, da mesma forma que a transição *"criarGrafos"*, contém duas ações de criação para objetos RPOO da classe Filósofo Figura 3.8:

fe = new Filosofo("f1");
fd = new Filosofo("f2");

O efeito do disparo da transição *"criarFilosofos"* é a criação dos dois filósofos e a criação das ligações entre o *Construtor* **cf** e os filósofos. Cuja Configuração resultante é mostrada na Figura 3.9.

Na configuração C3, na Figura 3.9, temos a transição *"enviarLigacoes"* habilitada. Esta transição contém inscrições RPOO de envio assíncrono das referências dos garfos para os filósofos.

fe = link(ge); fe = link(gd); fd = link(ge); fd = link(gd);

Depois do disparo da transição *"enviarLigacoes"*, temos quatro mensagens pendentes, como é mostrado na Figura 3.10.

Na configuração C4, na Figura 3.10, temos a transição *"desligarGarfos"* da rede da classe *Construtor*, Figura 3.4 habilitada e as transições *"receberLigacoes"* da rede da classe

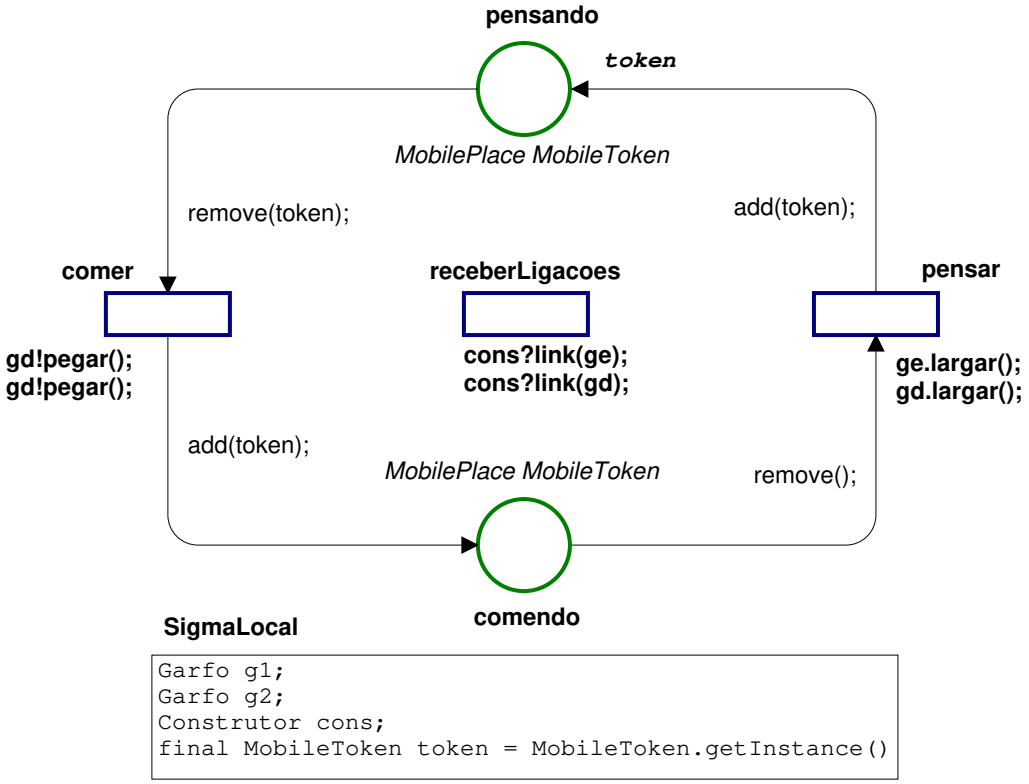


Figura 3.8: Rede para o comportamento dos Filósofos

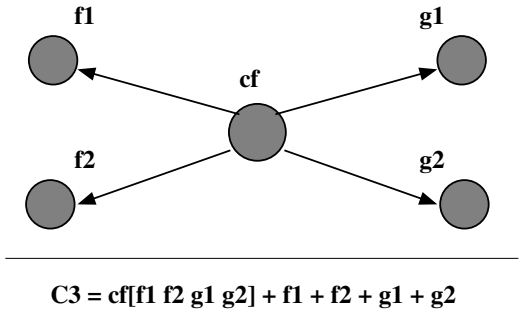


Figura 3.9: C3 - Configuração após o disparo da transição *"criarFilosofos"*

Filósofo, Figura 3.8. A transição *"receberLigacoes"* foi acrescentada a classe filósofo neste modelo modificado, para consumir as mensagens pendentes no sistema criadas pelo construtor **cf**. Em RPOO transições isoladas, como a transição *"receberLigacoes"*, podem ser utilizadas pois o acréscimos das inscrições de interação fazem com que elas tenham sentido. Podemos disparar qualquer uma das duas transições descritas. Iremos considerar o disparo da transição *"desligarGarfos"* do construtor, que contém a ação RPOO de desligamento de referências aos objetos RPOO Garfos com os nomes *"g1"* e *"g2"*.

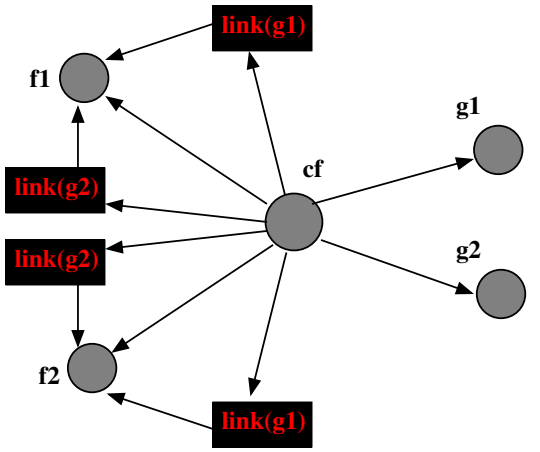


Figura 3.10: C4 - Configuração após o disparo da transição *"enviarLigacoes"*

unlink("g1");

unlink("g2");

Nesta configuração não existe mais as ligações entre o construtor ***cf*** e os garfos ***g1*** e ***g2***, como é mostrado na Figura 3.11.

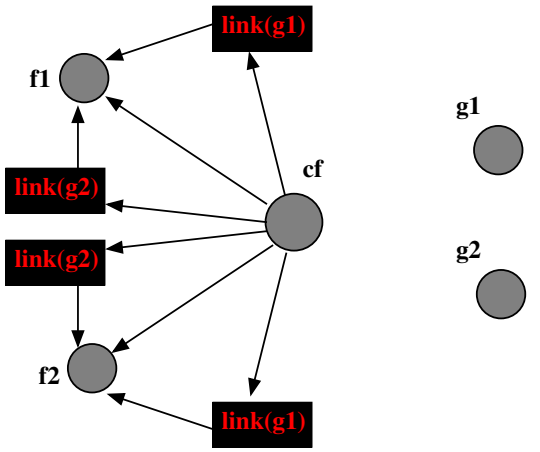


Figura 3.11: C5 - Configuração após o disparo da transição *"desligarGarfos"*

Na configuração C5, na Figura 3.11, temos a transição *"finalizar"* da rede da classe *Construtor* (Figura 3.4) habilitada e as transições *"receberLinks"* da rede da classe Filósofo, Figura 3.8. Podemos disparar qualquer uma das duas transições descritas. Iremos

considerar o disparo da transição *"finalizar"* do construtor, que contém a ação RPOO de auto-destruição. O efeito da ação de auto-destruição irá remover o objeto e todas as suas ligações. O objeto passa para um estado morto, ou seja, passa para um estado que representa uma referência nula (ele não existe). A referência nula é representada por um círculo branco de linha pontilhada como mostra a Figura 3.12. A ação de auto-destruição é representada pela inscrição:

end();

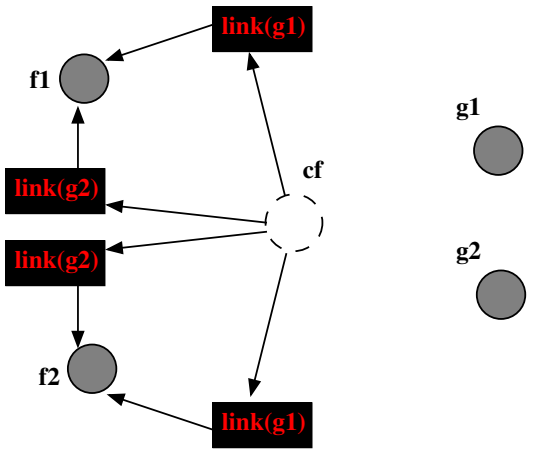


Figura 3.12: C6 - Configuração após o disparo da transição *"finalizar"*

Na Figura 3.12, vemos a configuração com o efeito da ação de auto-destruição, que transformou o objeto RPOO construtor ***cf*** em um objeto morto.

Na configuração C6, na Figura 3.12, o objeto construtor não existe mais, sua representação como objeto morto, ainda é mantida porque ainda existe mensagens pendentes onde ele é a origem. Desta forma, as únicas ações habilitadas no modelo dos filósofos modificado são as transições *"receberLigacoes"* dos dois filósofos. A transição *"receberLigacoes"* da classe filósofo tem as seguintes ações de entrada de dados:

cons?link(ge);

cons?link(gd);

Iremos considerar primeiramente o disparo da transição *"receberLigacoes"* do filósofo f1. Poderiamos escolher qualquer umas das transições ou até mesmo as duas formando um único evento. A configuração C7, resultante é mostrada na Figura 3.13.

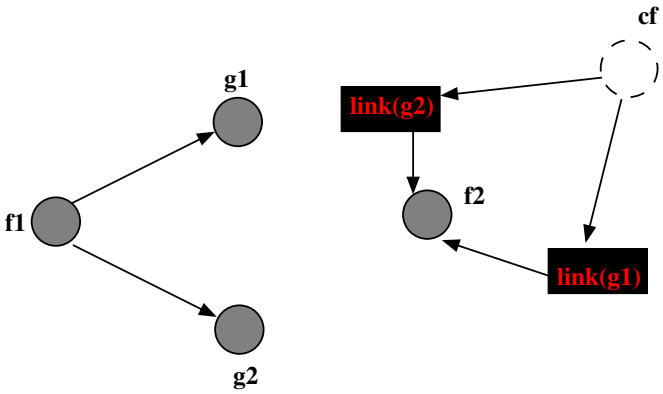


Figura 3.13: C7 - Configuração após o disparo da transição *"receberLigacoes"* de f1

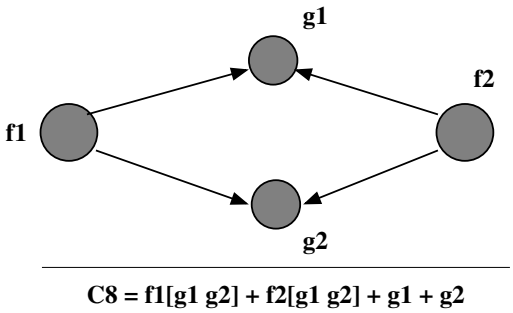


Figura 3.14: C8 - Configuração após o disparo da transição *"receberLinks"* de f2

Na configuração C8, na Figura 3.14, repetimos a operação anterior e disparamos a transição *"receberLigacoes"* do filósofo f2.

Neste ponto da simulação do modelo dos filósofos modificado, obtemos a configuração original da mesa do jantar dos filósofos. A partir de agora, o comportamento do modelo será o mesmo que o modelo original apresentado no Capítulo 2.

O modelo do problema do Jantar dos Filósofos Modificado que será descrito em detalhes e simulado nesta subseção, apresentam a proposta para as principais modificações na notação a ser usada nas redes que descrevem o comportamento das classes RPOO. Temos algumas diferenças entre as redes de Petri coloridas apresentadas nas Figuras 2.4 e 2.3 e as

apresentadas nas Figuras 3.6 e 3.8, que iremos explicar a seguir.

A primeira diferença está na declaração do tipo do lugar, como podemos ver o lugar pensando é do tipo *JMobilePlace JMobileToken*. Todo lugar de nossas redes é de algum tipo coleção. A interface da classe *JMobilePlace*, definida para representar o tipo "Lugar" é apresentada a seguir:

- *add(elemento)* - Expressão de arco que adiciona um elemento ao lugar de saída deste arco. Um exemplo do uso desta operação se encontra na rede filósofo da Figura 3.8, no arco que liga a transição "comer" ao lugar "comendo". O "elemento" deve ser de um tipo válido para a API.

- *get()* - Expressão de arco que retorna um elemento qualquer do lugar de entrada do arco, mas não o remove do lugar. O elemento é escolhido de forma aleatória.

- *get(chave)* - Expressão de arco que retorna um elemento do lugar de entrada do arco que tenha a chave passada como parâmetro. A chave pode ser um string ou inteiro que identifica o elemento a ser retornado. No caso do tipo do lugar for um "JMobileObject" a chave é o seu nome e no caso de "JMobileType" é o valor do elemento que deve ser removido.

- *isEmpty()* - Expressão de arco que retorna verdadeiro se o lugar está vazio ou falso caso contrário. Este método representa o arco inibidor das redes de Petri.

- *remove()* - Expressão de arco que remove um elemento do lugar de entrada deste arco. Se o lugar tem um conjunto de elementos, um deles será escolhido aleatoriamente e será removido.

- *remove(chave)* - Expressão de arco que remove o elemento do lugar de entrada deste arco cuja chave foi passada como parâmetro. Esta chave, como já foi dito pode ser um string ou um inteiro.

Podemos usar estas operações de forma semelhante ao código Java. Poderemos pegar o retorno das operações e guardá-lo em variáveis que poderão ser utilizadas dentro do escopo previsto para variáveis em RPOO. Outros métodos existentes na interface da classe MobilePlace foram omitidos, porque não tem semântica relacionada com as redes de Petri e são usados internamente.

Na declaração do tipo de lugares na rede da Figura 3.8 temos logo após a definição do tipo do lugar, a definição do tipo dos elementos contidos naquele conjunto. Por exemplo no lugar *"pensando"* da rede filósofo na temos o tipo MobileToken. O tipo MobileToken (ficha) é um tipo que só contém um valor, que é gerado estaticamente através da chamada *MobileToken.getInstance()*.

3.4 Representação de Classes RPOO

Desde o surgimento de RPOO havia a necessidade de representação dos modelos em uma forma textual que facilitasse a integração com outras ferramentas de suporte ao formalismo RPOO. Para isso, na definição da linguagem JMobile que faz a adaptação da sintaxe usada nas redes de Petri para sintaxe Java, foram definidas duas formas de representação textual uma em XML e outra em Java.

A representação textual usando XML tem como principal objetivo a comunicação entre ferramentas, em particular, entre um editor de redes de Petri para RPOO e as ferramentas para simulação e geração de espaço de estados desenvolvidas e descritas neste documento.

Já a representação textual usando Java, é usada internamente pela API JMobile. Esta representação é usada para descrever as redes de Petri das classes presentes nos modelos RPOO. Este código Java de descrição é usado pela API para a simulação dos modelos e para a geração de seus espaços de estados.

A idéia por traz da concretização de RPOO, através da linguagem JMobile é permitir que o modelador (o engenheiro de software) possa modelar seu sistema tanto graficamente quanto textualmente. A representação textual usada é a representação XML, e isso permite a definição de vários outros formatos. Desde que, estes novos formatos possam ser traduzidos para XML ou para Java que são as representações aceitas pela API. Contudo, se for de interesse do modelador (engenheiro de software), ele pode até mesmo já modelar seu sistema usando a representação textual Java e utilizar a API JMobile (apresentado no Capítulo 4) para simulação e geração de espaço de estados.

3.4.1 Representação XML

O formalismo das redes de Petri Orientadas a Objeto apresenta várias facilidades, uma delas é a representação gráfica herdada das redes de Petri Coloridas, úteis para discussão e para melhor entendimento do sistema e de seu funcionamento. E para manter a possibilidade de integração das ferramentas desenvolvidas a partir da API JMobile, com ferramentas de edição de redes de Petri coloridas e de verificação de modelos foi desenvolvida uma representação textual para manter um padrão de comunicação entre estas e outras ferramentas.

Para definição da representação textual das classes RPOO foi escolhido a linguagem XML (*Extensible Markup Language*) [Con04]). A linguagem XML é uma linguagem de marcação que permite a definição de outras linguagens a partir de extensões e definições de DTD (*Document Type Definitions*). O DTD na realidade é a definição de uma gramática para uma linguagem de extensão do XML. A seguir, na Figura 3.15, apresentamos o DTD que define a gramática para a escrita de classes RPOO com XML.

A partir desta representação XML para a descrição das classes de um modelos RPOO, foi definida um forma de tradução destas classes para código Java de descrição. O código Java será utilizado como base de tipos para o simulador. Ou seja, a API faz uso desse código Java para completar os tipos e facilitar os mecanismos de simulação implementados. Assim, especificamos um *Parser XML-Java*, com a função de traduzir as descrições das classes RPOO em XML para uma descrição em código Java. É importante ressaltar, que a responsabilidade de geração da descrição XML será de um editor de redes de Petri orientadas a objeto (não desenvolvido neste trabalho), e de posse deste XML o *parser* faz a tradução para código Java. Todo este processo será feito pelas ferramentas e não haverá a necessidade obrigatório do engenheiro de software editá-las. Contudo, ainda não temos uma ferramenta de edição de redes de Petri, ela se encontra em fase inicial de desenvolvimento em outro projeto. Como nosso objetivo não foi o desenvolvimento do editor, fizemos a edição gráfica de nossos modelos usando o Design/CPN e as descrições textual foram feitas diretamente nas representações textuais XML e Java.

Apresentaremos um exemplo do uso do DTD para a descrição das classes RPOO, a seguir temos o XML da rede de Petri da classe Garfo, Figura 3.6. Ele está dividido em duas partes nas Figuras 3.16 e 3.17, por ser extenso.

Na primeira parte do XML, Figura 3.16, temos as descrições do nó de declaração e dos

```
<!ELEMENT class_rpoo (c_name,local_sigma?, place+, transition+)>
<!ELEMENT c_name (#PCDATA)>
<!ELEMENT local_sigma ((var | const)+)>
<!ELEMENT var (v_name,v_type,v_value?)>
<!ELEMENT v_name (#PCDATA)>
<!ELEMENT v_type (#PCDATA)>
<!ELEMENT v_value (#PCDATA)>
<!ELEMENT const (ct_name,ct_type,ct_value?)>
<!ELEMENT ct_name (#PCDATA)>
<!ELEMENT ct_type (#PCDATA)>
<!ELEMENT ct_value (#PCDATA)>
<!ELEMENT place (p_name,modify?,p_type,marking)>
<!ELEMENT p_name (#PCDATA)>
<!ELEMENT modify (#PCDATA)>
<!ELEMENT p_type (#PCDATA)>
<!ELEMENT marking (#PCDATA)>
<!ELEMENT transition (t_name,input?,action?,guard?,output?)>
<!ELEMENT t_name (#PCDATA)>
<!ELEMENT input (in_place+)>
<!ELEMENT in_place (inp_name,inp_exp)>
<!ELEMENT inp_name (#PCDATA)>
<!ELEMENT inp_exp (#PCDATA)>
<!ELEMENT action (#PCDATA)>
<!ELEMENT guard (#PCDATA)>
<!ELEMENT output (in_place+)>
<!ELEMENT out_place (outp_name,outp_exp)>
<!ELEMENT outp_name (#PCDATA)>
<!ELEMENT outp_exp (#PCDATA)>
```

Figura 3.15: DTD que define a linguagem de escrita de Classes RPOO

lugares. No *tag local_sigma* descrevemos as variáveis (*tag var*) e as constantes (*tag const*) que serão utilizadas na classe, com seus respectivos tipos e valores iniciais. Já nos *tags place* temos as descrições dos lugares que compõem a rede, neles temos o atributos nome (*tag*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE class_rpoo SYSTEM "rpoo_class.dtd">

<class_rpoo>
    <c_name>Garfo</c_name>
    <local_sigma>
        <var>
            <v_name>fil</v_name>
            <v_type>Filosofo</v_type>
        </var>
        <const>
            <ct_name>token</ct_name>
            <ct_type>JMobileToken</ct_type>
            <ct_value>JMobileToken.getInstance()</ct_value>
        </const>
    </local_sigma>

    <place>
        <p_name>livre</p_name>
        <p_type>JMobileToken</p_type>
        <marking>token</marking>
    </place>
    <place>
        <p_name>em_uso</p_name>
        <p_type>JMobileToken</p_type>
        <marking>empty</marking>
    </place>
```

Figura 3.16: XML que descreve a classe Garfo - parte 1 de 2

p_name), o tipo (*tag p_type*) e a marcação inicial do lugar (*tag marking*).

Na segunda parte do XML, Figura 3.17, temos as descrições das transições. Nos *tags transition* temos as descrições das transições que compõem a rede, neles temos os atributos que descrevem respectivamente o nome (*tag t_name*), os lugares de entrada (*tag in_place*, os arcos são as chamadas a métodos), as ações RPOO (*tag action*), as guardas (*tag guard*) e os lugares de saída (*tag out_place*). Como podemos notar, não há nenhuma dúvida, que a descrição XML nas Figuras 3.16 e 3.17 representam fielmente a classe RPOO descrita

```
    <transition> <t_name>alocar</t_name>
        <input> <in_place>
                    <inp_name>livre</inp_name>
                    <inp_exp>remove()</inp_exp>
                </in_place>
        </input>
        <action>fil?pegar();</action> <guard>true</guard>
        <output> <out_place>
                     <outp_name>em_uso</outp_name>
                     <outp_exp>add(token)</outp_exp>
                 </out_place>
        </output>
    </transition>
    <transition> <t_name>liberar</t_name>
        <input> <in_place>
                    <inp_name>em_uso</inp_name>
                    <inp_exp>remove()</inp_exp>
                </in_place>
        </input>
        <action>fil?largar();</action> <guard>true</guard>
        <output> <out_place>
                     <outp_name>livre</outp_name>
                     <outp_exp>add(token)</outp_exp>
                 </out_place>
        </output>
    </transition>
</class_rpoo>
```

Figura 3.17: XML que descreve a classe Garfo - parte 2 de 2

graficamente na Figura 3.6, usando as redes de Petri.

3.4.2 Representação Java

Neste subseção, iremos demonstrar como seria feita a tradução das classes RPOO descritas em XML, como foi mostrado na Subseção 3.4.2, anterior a esta. A representação Java para as classes RPOO segue o mesmo princípio, da descrição XML. Ou seja, para cada parte da

descrição em XML teremos uma representação Java, assim como, a representação XML tem um *tag* específico para cada parte da representação gráfica de redes de Petri.

Nas Figuras 3.18(na Página 46), e 3.19 (na Página 47) temos a representação Java para a classe Garfo da Figura 3.6, com base no XML nas Figuras 3.16 e 3.17. É importante ressaltar que como nós queremos descrever a classe RPOO com Java, então os métodos que representam as transições devem descrever as expressões sem executá-las. Ou seja, devemos descrever as operações nos arcos, na guarda, e nas ações RPOO de maneira que elas não sejam executadas no momento da instanciação dos objetos. Para isso, a classe *MobileTransition* dá suporte a esta descrição fazendo uma espécie de agendamento de operações que serão executadas posteriormente.

Na Figura 3.18, temos a primeira parte do arquivo com o código Java para a descrição da classe RPOO. Nela temos a definição do pacote desta classe, os *import's* e o construtor. A classe Garfo estende a classe abstrata *AbstractJMobileObject*, que é a classe que implementa a representação e o comportamento das classes RPOO, ou seja, das rede de Petri.

O construtor da classe é bem simples, e como toda classe ele segue um padrão bem definido. Ele recebe como parâmetro o nome do objeto e o atribui a chave (*key*) e ao tipo do objeto. Por último o construtor faz chamadas a métodos de construção de lugares e transições. Todos os lugares e transições são armazendos em coleções chamadas *placeInfo* e *transitionInfo*, essas informações são utilizadas nos métodos da classe *AbstractJMobileObject* que da suporte a avaliação de habilitação e a simulação. A última linha do construtor traz a chamada ao método que define a marcação inicial (estado inicial) das instâncias desta classe.

Logo abaixo do construtor temos os métodos de construção dos lugares. São métodos bem simples que fazem uma chamada ao construtor do tipo *JMobilePlace* e depois retorna o lugar criado. O construtor, da classe que representa os lugares (*JMobilePlace*), recebe como parâmetros uma *string* com o nome do lugar e um *class* que define os tipos dos elementos que poderão estar contidos no lugar.

Na Figura 3.19, temos a segunda parte do arquivo com o código Java para a descrição da classe RPOO. Nela temos os métodos para a definição da marcação inicial e para a construção das transições. O método que define a marcação inicial, adiciona algum elemento aos lugares através da chamada a um método *add* definido na classe abstrata a qual o Garfo

```
package jmobile.examples.filosofo;
import jmobile.model.SingleVariable;
import jmobile.types.*;
import jmobile.types.petrinet.*;
public class Garfo extends AbstractJMobileObject {
   private SingleVariable fil = new SingleVariable("fil",Filosofo.class);
   private SingleVariable t = new SingleVariable("t", MobileToken.class);
   private SingleVariable content;
   public Garfo(String name) {
      this.name = name;
      this.key = name;
      this.type = this.getClass().getName();
      this.placesInfo.put("livre", createPlace_livre());
      this.placesInfo.put("em_uso", createPlace_em_uso());
      try {
         this.trasitionInfo.put("alocar", createTransition_alocar());
         this.trasitionInfo.put("liberar", createTransition_liberar());
         this.initialMarking();
      } catch (Exception e) {
         e.printStackTrace();
      }
  }
  private JMobilePlace createPlace_em_uso() {
     return new JMobilePlace("em_uso", JMobileToken.class);
  }
  private JMobilePlace createPlace_livre() {
     return new JMobilePlace("livre", JMobileToken.class);
  }
```

Figura 3.18: Código Java que descreve a classe Garfo - parte 1 de 2

estende.

Os métodos de criação de transição são os maiores e podemos até dizer os mais complexos. Neles temos as descrições para os lugares de entrada e saída, feitas através dos métodos *addInputPlace* e *addOutputPlace* da transição. Em seguida, temos a descrição das expressões de arcos que neste caso, foram representadas como a adição à transição de uma chamada (*Call*) a algum método da interface do lugar (*JMobilePlace*). Como podemos ver nas linhas

```
   public void initialMarking()
      throws InvalidElementTypeException, NoSuchPlaceException {
         this.add("livre", JMobileToken.getInstance());
   }
   private JMobileTransition createTransition_alocar()
      throws NotDefinedOperationException {
         JMobileTransition alocar = new JMobileTransition("alocar");
         alocar.addInputPlace("livre");
         alocar.CallRemInVar("livre",t);
         alocar.addInputAction(this,fil,"pegar",content);
         alocar.addOutputPlace("em_uso");
         alocar.CallAdd("em_uso", t);
         return alocar;
   }
   private JMobileTransition createTransition_liberar()
      throws NotDefinedOperationException {
         JMobileTransition liberar = new JMobileTransition("liberar");
         liberar.addInputPlace("em_uso");
         liberar.CallRemInVar("em_uso",t);
         liberar.addInputAction(this,fil,"largar",content);
         liberar.addOutputPlace("livre");
         liberar.CallAdd("livre", t);
         return liberar;
   }
}
```

Figura 3.19: Código Java que descreve a classe Garfo - parte 2 de 2

com os métodos *CallRem("livre")* e *CallAdd("em_uso", token)*, estas e outras chamadas a métodos serão explicados em detalhes no capítulo 4, que apresentam a API JMobile de suporte ao formalismo RPOO.

Nas linhas de código centrais, na Figura 3.19, dos métodos de descrição das transições temos as adições de ações RPOO à transição. Na transição *alocar* temos a adição de uma ação de entrada (ou de consumo) de uma mensagem pendente no sistema. Esta mensagem pendente, como demostra a ação, deve ter como origem um filósofo como determina a va-

riável *fil*, deve ter o nome *"pegar"*, o agente deve ser este objeto (*this*[2]) e o conteúdo será atribuído a variável *content*. Já na transição *liberar* temos uma outra transição de entrada de dados bem semelhante a anterior, cujo o nome deve ser *largar*.

3.5 Representação de Modelos RPOO

Além da representação textual para a classes RPOO, detalhadas na Seção 3.4 da Página 40. Temos definida uma representação textual em XML e em Java para os modelos RPOO.

3.5.1 Representação XML

Para definição da representação textual dos modelos RPOO foi definido o DTD para a descrição dos modelos em XML. A seguir, na Figura 3.20, apresentamos o DTD que define a gramática para a escrita de modelos RPOO com XML.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT model_rpoo (m_name, classes+, global_sigma?)>
<!ELEMENT m_name (#PCDATA)>
<!ELEMENT classes (class+)>
<!ELEMENT class (#PCDATA)>
<!ELEMENT global_sigma (types?, vars?, consts?)>
<!ELEMENT types (type+)>
<!ELEMENT type (ty_name, ty_extends, ty_values)>
<!ELEMENT ty_name (#PCDATA)>
<!ELEMENT ty_extends (#PCDATA)>
<!ELEMENT ty_values (#PCDATA)>
<!ELEMENT vars (var+)>
<!ELEMENT var (v_name, v_type, v_value?)>
<!ELEMENT v_name (#PCDATA)>
<!ELEMENT v_type (#PCDATA)>
<!ELEMENT v_value (#PCDATA)>
```

Figura 3.20: DTD que define a linguagem de escrita de Modelos RPOO

3.5.2 Representação Java

Para construirmos uma representação do estado inicial de um modelo RPOO nós traduzimos a descrição do modelo em XML para uma classe *Main* do Java, usando os tipos definidos na

[2]A própria instância de objeto que está fazendo a chamada ao método de construção de transições.

API e os tipos definidos para o modelo: tipos rpoo e tipos para representar dados.

A classe *Main* contém um método *main* do Java. Dentro deste método construimos nosso modelo e instanciamos um simulador para a simulação guida do modelo. Um exemplo da classe *Main* é apresentada na Figura 3.21.

```
01: import jmobile.model.*;
02: public class Main {
03:     public static void main(String[] args) {
04:         //Construindo o Modelo
05:         JMobileModel modelo = new JMobileModel(``Modelo'');
06:         //Construindo a Configuração ``Vazia''
07:         Configuration confInicial = new Configuration();
08:         //Construindo os objetos RPOO
09:         Classe1 obj1 = new Classe1(``obj1'');
10:         Classe2 obj2 = new Classe2(``obj2'');
11:         Classe2 obj3 = new Classe3(``obj3'');
12:         //Adicionando os objetos a configuração
13:         confInicial.add(obj1);
14:         confInicial.add(obj2);
15:         confInicial.add(obj3);
16:         //Adicionando as ligações entre os objetos
17:         confInicial.addLink(``obj1'',``obj2'');
18:         confInicial.addLink(``obj1'',``obj3'');
19:         //Adicionando a configuração incial ao modelo.
20:         modelo.setConfiguration(confInicial);
21:     }
22: }
```

Figura 3.21: Código da classe Main

A configuração do estado inicial para o modelo construido na Figura 3.21, é a seguinte:

obj1[obj2 obj3] + obj2 + obj3

Na próxima seção apresentaremos os dois principais formatos para a representação do espaço de estados dos modelos RPOO.

3.6 A notação para representar o Espaço de Estados

O objetivo desta Seção é descrever duas formas de representação externa para espaços de estados de sistemas modelados através de RPOO [Gue02b]. A existência de um formato de referencia permite independizar atividades referentes aos diversos processos que manipulam representações comportamentais de modelos RPOO. Em particular, independiza os processos de geração e de verificação do espaço de estados.

O primeiro formato, apresentado na subseção 3.6.1, é o formato de estrada para o espaço de estados do verificador de modelos em RPOO — Veritas, [Rod04]. Além do espaço de estados a ferramenta Veritas recebe fórmulas em Lógica temporal com a descrição de propriedade a serem verificados no espaço de estados pela ferramenta. Apresentando ao final deste processo um *trace* exemplo ou contra-exemplo, demonstrando assim, a validade ou não da propriedade.

O segundo formato, apresentado na subseção 3.6.2, é um formato definido e utilizado por uma ferramenta chamada *Aldebaran*, [Fer89]. Este formato também será utilizado no trabalho sobre geração de casos de testes, desenvolvido por André Figueiredo [FM04].

3.6.1 Formato do Espaço de Estados para o Veritas

O formato consiste em uma linearização do espaço de estados, definido sobre arquivos texto (arquivos ASCII). Além de facilitar a leitura humana do espaço de estados, o formato é especialmente adequado para a manipulação direta através de ferramentas convencionais de tratamento de texto disponíveis, por exemplo, em sistemas *unix*.

O texto está dividido em duas partes. Na primeira, descrevemos o formato do arquivo, usando gramáticas—de fato, usamos EBNFs *Extended Backus-Naur Form*. Na segunda parte, apresentamos um exemplo de representação do espaço de estados de um sistema simples.

O Formato

Espaços de estados são representados sobre arquivos de texto. As regras abaixo caracterizam o formato geral do *space* de estados, como uma seqüência não vazia de nós. Cada *node* caracteriza uma configuração alcançável do sistema.

$$
\begin{aligned}
\textit{space} &::= (\textit{node})^{+} \\
\textit{node} &::= \textbf{id}\ \textbf{:}\ \textit{struct} \mid \textit{events} \mid \textit{predecessors}\ \textbf{;}
\end{aligned}
$$

Podemos pensar em cada *node* como um registro do arquivo. A seguir, descrevemos detalhadamente, cada um dos campos que o compõem.

Campo 1: Estrutura O primeiro campo de um nó, o identifica unicamente dentro do espaço de estados e caracteriza a estrutura da configuração. Tem a forma:

id : *struct*

Cada nó é identificado unicamente por um identificador composto por caracteres alfanuméricos, que na prática pode ser gerado automaticamente para representar a ordem em que o nó foi gerado no espaço de estados. A estrutura da configuração é representada por sua expressão algébrica (ver [Gue02b]), segundo a seguinte gramática[3]

$$
\begin{aligned}
\textit{struct} &::= \textit{object}\ (\textbf{+}\ \textit{object})^{*}\ [\ \textit{msg\_destination\_dead}\ ] \mid \epsilon \\
\textit{object} &::= \textbf{id}\ [\ \textit{references}\ ]\ [\ \textbf{:}\ \textit{messages}\ ] \\
\textit{references} &::= \textbf{[}\ (\textit{id})^{*}\ \textbf{]} \\
\textit{messages} &::= \textbf{[}\ \textit{msg}\ (\textbf{++}\ \textit{msg})^{*}\ \textbf{]} \\
\textit{msg} &::= \textbf{number `\ id . id} \\
\textit{msg\_destination\_dead} &::= \textbf{\#}\ \textit{messages}
\end{aligned}
$$

O componente *msg_destino_morto* é um campo de mensagens especial. Deverão constar neste campo todas as mensagens cujo objeto consumidor, ou destino, já foi destruído.

[3] 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.

Campo 2: Eventos O segundo campo de um registro caracteriza os eventos habilitados naquela configuração e os nós que são alcançados no caso de ocorrência dos eventos. Os eventos são separados por "+" e devem ser descritos segundo a seguinte gramática:

$$
\begin{aligned}
\textit{events} &::= \textit{event}\ (\textbf{+}\ \textit{event})^* \mid \epsilon \\
\textit{event} &::= \textit{action}\ (\textbf{\&}\ \textit{action})^*\ >\ \textbf{id} \\
\textit{action} &::= \textbf{id :}\ \textit{elementary\_action}\ (\textbf{@}\ \textit{elementary\_action})^* \\
\textit{elementary\_action} &::= \textbf{id . id} \mid \textbf{id ! id} \mid \textbf{id ? id} \mid \textbf{new id} \mid \textbf{del id} \mid \textbf{end}
\end{aligned}
$$

Campo 3: predecessores O terceiro e último campo de cada registro identifica os nós predecessores da configuração. O campo consiste na simples enumeração dos identificadores dos nós, separados por espaços.

$$\textit{predecessors} ::= (\textbf{id})^*$$

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 naturais. Os demais terminais usados nas gramáticas acima devem ser considerados literalmente como as seqüências de caracteres que os compõem (p. exemplo, **new** e **@**).

Separadores de campos Os campos de cada registro são separados pelo caractere "|" que, portanto, não deve ser usado na informação contida nos demais campos. Logo, cada registro tem a forma geral:

$$\langle \textit{struct} \rangle \mid \langle \textit{events} \rangle \mid \langle \textit{predecessors} \rangle$$

Um Exemplo

Para exemplificar o formato descrito, representamos o espaço de estados de um modelo para o clássico problema do jantar dos filósofos de Dijkstra. O modelo completo em RPOO pode ser encontrado em [Gue02b, Seção 5.3.1]. Uma representação gráfica do espaço de estados desse sistema para uma configuração inicial contendo apenas 2 filósofos é apresentada na Figura 3.22.

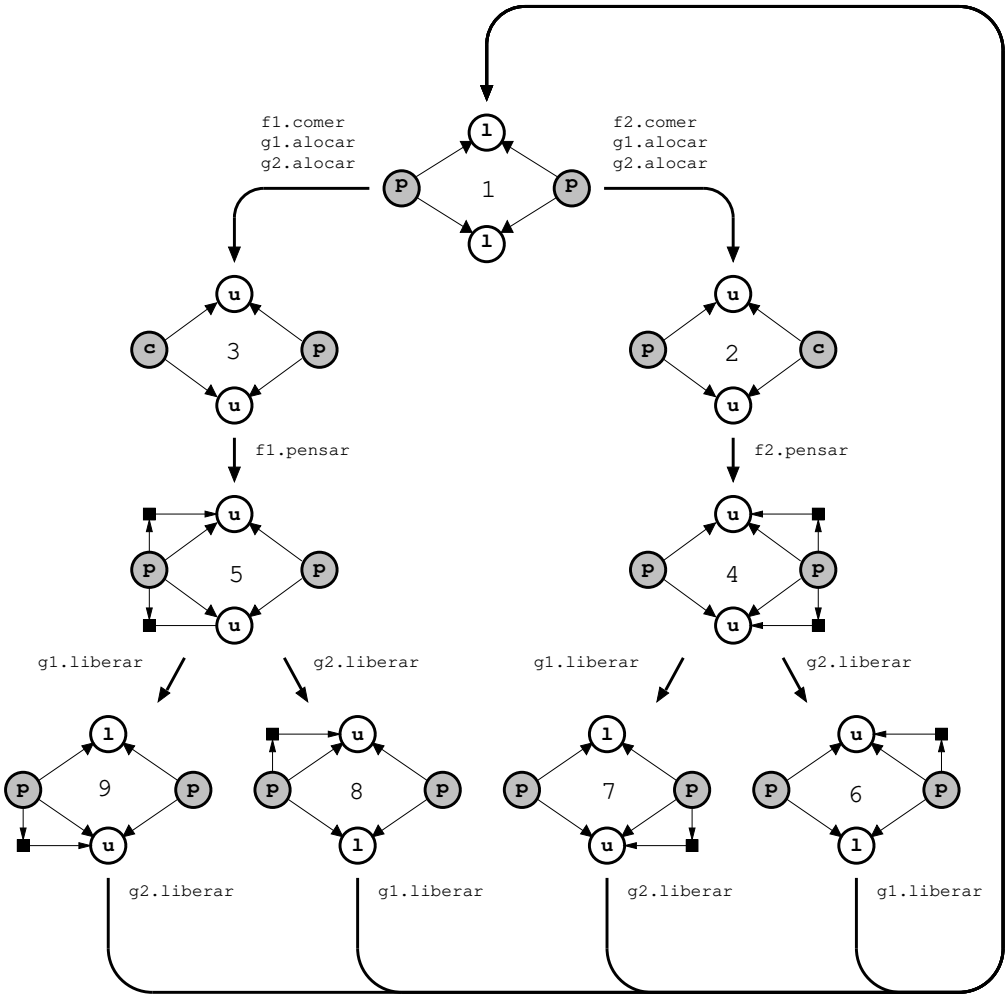


Figura 3.22: Representação gráfica do espaço de estados.

Estrutura: A configuração inicial do sistema, representada pelo nó mais acima no grafo, consiste nos filósofos f_1 e f_2 ligados aos garfos g_1 e g_2. Podemos representar essa estrutura como a soma formal: $f_1[g_1\ g_2] + f_2[g_1\ g_2] + g_1 + g_2$ (trata-se de uma variação da notação usada em [Gue02b]). Para representar essa estrutura no formato textual descrito, contudo,

é necessário abrir mão de facilidades gráficas como índices. Assim, a estrutura acima é representada por[4]:

```
f1[g1 g2] + f2[g1 g2] + g1 + g2
```

Eventos: No estado inicial, os filósofos estão pensando. Logo, há apenas dois eventos possíveis: ou f_1 deixa de pensar e passa a comer; ou o mesmo para f_2. Devido ao uso compartilhado de garfos, os eventos são mutuamente exclusivos. Formalmente, o primeiro evento corresponde ao seguinte conjunto de ações:

$$\{f_1{:}g_1!pegar \circ g_2!pegar, g_1{:}f_1?pegar, g_2{:}f_1?pegar\}.$$

No formato descrito, o evento é representado por:

```
f1:g1!pegar@g2!pegar & g1:f1?pegar & g2:f1?pegar
```

e o segundo evento por:

```
f2:g1!pegar@g2!pegar & g1:f2?pegar & g2:f2?pegar
```

Identificadores: Em princípio, qualquer identificador pode ser usado em cada nó. Do ponto de vista prático, contudo, é conveniente que os identificadores sejam prefixados por uma seqüência de letras e terminados por um número de série. Isto, como veremos na próxima seção, facilita a análise de arquivos de espaços de estados através de ferramentas de manipulação de arquivos de texto. Neste exemplo, usamos o prefixo `no` e numeramos os estados a partir de 0. Assim, `no0` identifica o estado inicial. Observe também que não há nenhuma imposição sobre a ordem em que os nós do espaço de estados devem aparecer na linearização. Contudo, é conveniente que o estado inicial seja o primeiro registro.

Um registro: Considerando o que foi exposto, podemos *montar* o primeiro registro do espaço de estados. Basta concatenar a estrutura aos eventos e aos nós predecessores, complementado os campos com os identificadores dos nós sucessores e predecessores. O registro referente à configuração inicial do sistema é representado por:

[4]Em todo o documento, usaremos fonte `courier` para denotar textos que correspondem à representação no formato descrito.

```
1 : f1[g1 g2] + f2[g1 g2] + g1 + g2
    | f1:g1!pegar@g2!pegar & g1:f1?pegar & g2:f1?pegar > 3
    + f2:g1!pegar@g2!pegar & g2:f1?pegar & g2:f2?pegar > 2
    | 9 8 7 6 ;
```

O registro abaixo ilustra o estado 5 da Figura 3.22:

```
5 : f1[g1 g2] + f2[g1 g2] + g1:[f1.liberar] + g2:[f1.liberar]
    | g1:f1?liberar > 9
    + g2:f1?liberar > 8
    | 3 ;
```

No último exemplo, diferentemente do primeiro, temos mensagens pendentes a serem consumidas. O termo g1:[f1.liberar] tem a seguinte semântica:
Há uma mensagem pendente para ser consumida por g_1, cujo conteúdo é *liberar* e o agente é o objeto f_1. O problema do jantar dos filósofos não apresenta nenhum estado contendo alguma mensagem cujo objeto consumidor tenha sido destruído, contudo podemos imaginar que no estado 5 g_1 foi destruído. A estrutura seria então:

```
f1[g1 g2] + f2[g1 g2] + g2:[f1.liberar] #[f1.liberar]
```

O termo #[f1.liberar] tem a seguinte semântica:
Há uma mensagem pendente para ser consumida por algum objeto que foi destruído, cujo conteúdo é *liberar* e o agente é o objeto f_1. É evidente que esta mensagem jamais será consumida, e portanto permanecerá em todos os estados sucessores.
OBS: Mesmo que no futuro um outro objeto venha a ser criado com o mesmo id, as referências serão diferentes, logo o novo objeto não terá ligação com a mensagem pendente.

3.6.2 Formato do Espaço de Estados para o Aldebaran

Nesta subseção descriveremos o formato textual para o espaço de estados utilizado pela a ferramenta *Aldebaran*. Esta ferramenta permite a minimização e comparação de Sistema

de Transições Rotuladas ou Labelled Transitions Systems (LTS), com respeito à relações de equivalência e pré-ordem. Este formato está sendo utilizado no trabalho de mestrado sobre a Geração Automática de Casos de Teste para Sistemas Baseados em Agentes Móveis ([FM04]).

Formato Aldebaran: espaço de estados para LTS O arquivo com o formato Aldebaran para espaço de estados é um arquivo de texto com extensão **.aut**. Onde para cada estado do espaço de estados é representado por um número natural. A primeira linha do nomearquivo.aut, chamado descritor tem a seguinte estrutura:

des (<primeiro-estado>, <número-de-transições>, <número-de-estados>)

O primeiro estado é sempre igual a 0. Cada linha restantes do arquivo representam um arco; estas linhas têm a seguinte estrutura:

(<estado-origem>, <evento>, <estado-destino>) Onde <estado-origem> e <estado-destino> são números e <evento> é um string de caracteres entre aspas duplas (com no máximo 5000 caracteres), que representa o evento de transição entre os estados. Note: <evento> é *case-sensitive*.

Um Exemplo

Para exemplificar o formato descrito, representamos o espaço de estados de um modelo para o clássico problema do jantar dos filósofos de Dijkstra, o mesmo apresentado na subseção 3.6.1. O modelo completo em RPOO pode ser encontrado em [Gue02b, Seção 5.3.1].

```
(0,12,9)
(0,"f2:g2!pegar() & f2:g1!pegar() & g2:f2?pegar() & g1:f2?pegar()",1)
(0,"f1:g1!pegar() & f1:g2!pegar() & g1:f1?pegar() & g2:f1?pegar()",5)
(1,"f2:g2.largar() & f2:g1.largar()",2)
(2,"g1:f2?largar()",3)
(2,"g2:f2?largar()",4)
(3,"g2:f2?largar()",0)
(4,"g1:f2?largar()",0)
(5,"f1:g1.largar() & f1:g2.largar()",6)
(6,"g1:f1?largar()",7)
```

```
(6,"g2:f1?largar()",8)
(7,"g2:f1?largar()",0)
(8,"g1:f1?largar()",0)
```

Capítulo 4

A API JMobile

Neste capítulo, apresentamos a API Java JMobile. Em primeiro lugar, damos uma visão geral da organização da API e de suas funcionalidades para a modelagem e simulação. Também apresentamos a arquitetura da API em termos de pacotes (*packages*) com a descrição de suas funcionalidades. Por último, apresentamos as restrições atuais, as possíveis extensões de funcionalidades.

4.1 Visão Geral

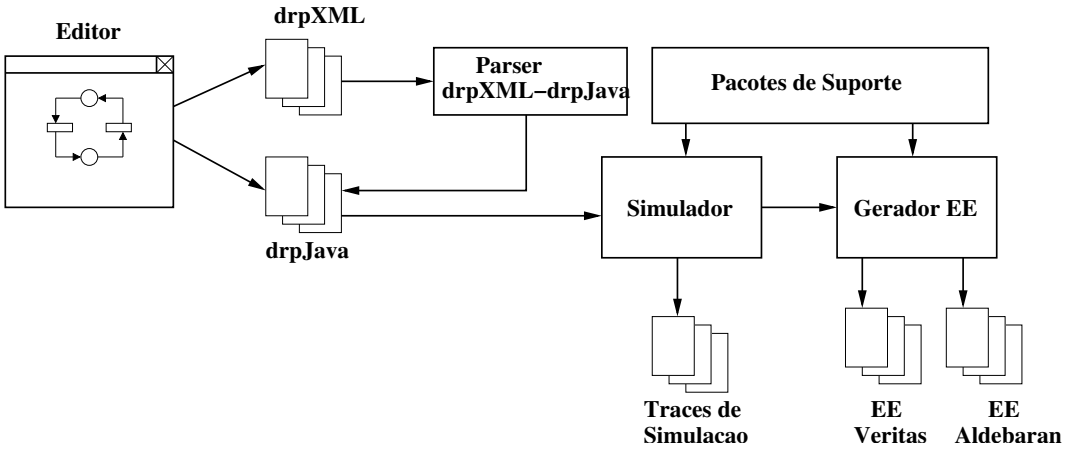


Figura 4.1: Ambiente e relacionamento das ferramentas de suporte a RPOO

A API JMobile implementa tipos e funcionalidades para a representação de configurações RPOO e de seu mecanismo de evolução. Na Figura 4.1, apresentamos o ambiente no qual se insere a API e o relacionamento das ferramentas de simulação e geração de espaço de

estados, que são protótipos de ferramentas desenvolvidas a partir da API. A seguir, listamos as ferramentas presentes no ambiente de suporte ao formalismo RPOO. No decorrer do capítulo apresentamos as funcionalidades e a forma de utilização de cada uma das ferramentas. A seguir temos a lista de ferramentas presentes no ambiente apresentado na Figura 4.1.

Editor — Editor de Redes de Petri Orientadas a Objeto que será utilizado para a descrição das redes graficamente e a partir desta descrição gerarmos os arquivos de descrição das classes em XML (chamado drpXML — descrição de redes de Petri em XML) e os arquivos de descrição das classes em Java (chamado drpJava — descrição de redes de Petri em Java). As descrições em XML e Java foram apresentadas no Capítulo 3 e são responsáveis pela comunicação do Editor com o Simulador e com o Parser. Ressaltamos que o Editor de rede de Petri OO, não é nosso objeto de desenvolvimento.

Parser drpXML-drpJava — A partir dos arquivos XML com a descrição das classes do modelo, o *package* Parser faz uma tradução do XML para a representação da rede com uma descrição em código Java. Estes novos arquivos de descrição em Java, são usados pelo simulador no momento da simulação.

Simulador — O simulador de modelos em *JMobile/RPOO* faz uso dos arquivos de descrição em código Java e de uma API (*packages*) de suporte as funcionalidades de representação, de simulação de modelos e de suas configurações possíveis. O simulador tem como entrada principal uma configuração inicial, que nada mais é, que um estado do sistema com os relacionamentos entre os objetos e seus estados internos. Através dos mecanismos de evolução o simulador gera novas configurações do sistema.

Gerador EE — O módulo de geração de espaço de estados traz implementações de algoritmos de geração em profundidade, fazendo uso dos *packages* de suporte e dos mecanismos de simulação. O espaço de estados pode ser gerado em dois formatos: formato Veritas e formato Aldebaran. O formato Veritas para o espaço de estados é utilizado pela ferramenta Veritas [RdFG03] de verificação de software para a análise de propriedades a partir de lógica temporal. O formato Aldebaran é utilizado pela ferramenta Aldebaran [Fer89], que é uma ferramenta para a verificação de processos de comunicação.

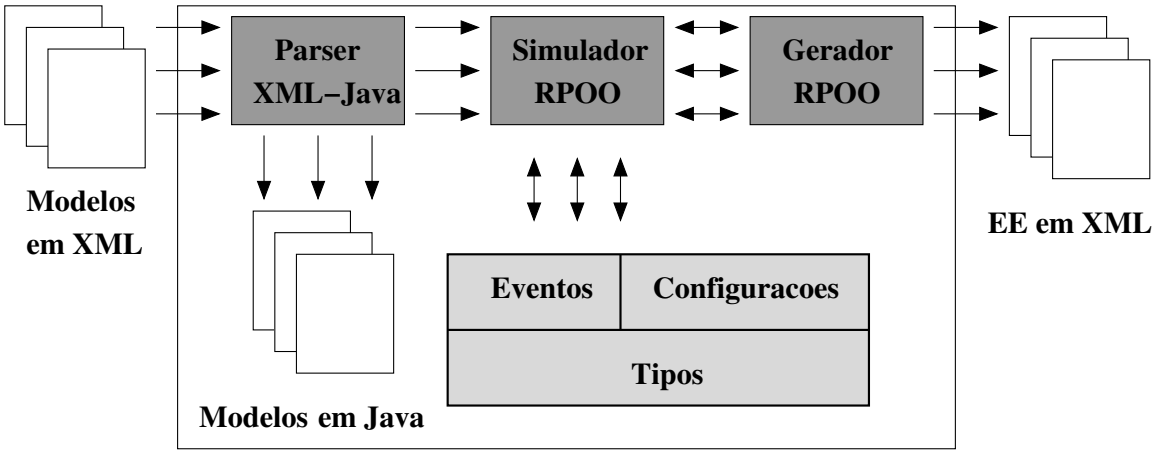


Figura 4.2: Arquitetura geral da API JMobile

Na Figura 4.2 é apresentada a arquitetura geral da API JMobile de simulação e geração de espaço de estados. A API JMobile é a junção dos *packages* de suporte (pacotes com as representações para eventos, configurações e tipos) com os *packages* que contém o Parser, o Simulador e o Gerador de espaço de estados. A API está dividida em vários sub-pacotes (*subpackages*) e cada um deles é responsável por um conjunto de funcionalidades relacionadas.

Os *subpackages* da API são responsáveis pela representação de Configurações (classe *Configuration*), pela representação dos eventos, ou seja, pela representação das ações RPOO. O *subpackage* de tipos traz a representação de tipos básicos a serem utilizados nos modelos, além dos tipos usados para representar e simular o comportamento das redes de Petri orientadas a objeto.

Na seção seguinte vamos apresentar em mais detalhes a organização dos *packages*, com maior enfoque nas classes principais classes da API e seus relacionamentos com as outras.

4.2 Estruturação Interna

Nesta seção, iremos apresentar a estruturação interna da API JMobile e iremos detalhar algumas de suas partes. As partes mais relevantes e de interesse do modelador são: as classes de representação de modelos, as classes de representação de tipos básicos e as de suporte ao mecanismo de simulação e de geração de espaço de estados das redes de Petri orientada a objetos.

Na Figura 4.2 da Página 60 apresentamos de forma geral como estão relacionadas as funcionalidades da API através dos *packages*. Agora estamos interessados em uma visão mais interna da API e como já foi dito usaremos parte do diagrama de classes. O diagrama de classes completo está no apêndice A na seção A.5.

A API JMobile está dividida em 5 grandes *packages*: *jmobile.examples*, *jmobile.rpoo*, *jmobile.parser*, *jmobile.tools* e *jmobile.types*. A seguir descrevemos os *packages*:

package jmobile.examples Este *package* traz três modelos RPOO construídos usando a API JMobile. Um dos modelos, é o modelo dos Filósofos, que implementa uma solução para o problema clássico dos Filósofos, onde temos um problema de concorrência e disputa por recursos. Outro modelo é o dos Filósofos Modificados, que é uma variação do modelo clássico dos Filósofos, utilizado para exemplificar o uso e o funcionamento de todos os tipos de ações RPOO. O terceiro exemplo é o modelo para o sistema Conferência, este modelo foi utilizado como estudo de caso deste trabalho. Além disso, o sistema conferência foi utilizado na dissertação de Figueiredo [FMdF05].

package jmobile.rpoo Neste *package* temos as classes que são responsáveis pela representação dos modelos RPOO.

package jmobile.tools Neste *package* temos os protótipos para o simulador automático e para o gerador de espaço de estados de modelos.

package jmobile.types No *package types* temos a implementação dos tipos básicos utilizados nos modelos e no mecanismo de simulação das redes de Petri orientadas a objetos.

package jmobile.parser O *package parser* contém o protótipo para o Parser de tradução dos modelos RPOO descritos em XML para a sua representação em código Java. Este *parser* se encontra em uma versão inicial e terá ampla utilização quando for integrada à uma interface gráfica para a edição de modelos.

Os *packages* mais relevantes são os *packages jmobile.rpoo*, *jmobile.types* e *jmobile.tools* que trazem a representação dos modelos RPOO, os tipos elementares, as expressões e os protótipos das ferramentas.

4.2.1 *Package jmobile.rpoo*

Neste *package* temos as classes Java para a representação dos modelos RPOO. Apresentamos na Figura 4.3 as classes mais relevantes do *package jmobile.rpoo*. As classes para a representação de uma Configuração RPOO são: a classe *Configuration*, a classe *Link*, a classe *Message*, a interface *JMobileObject* e a classe *AbstractJMobileObject*.

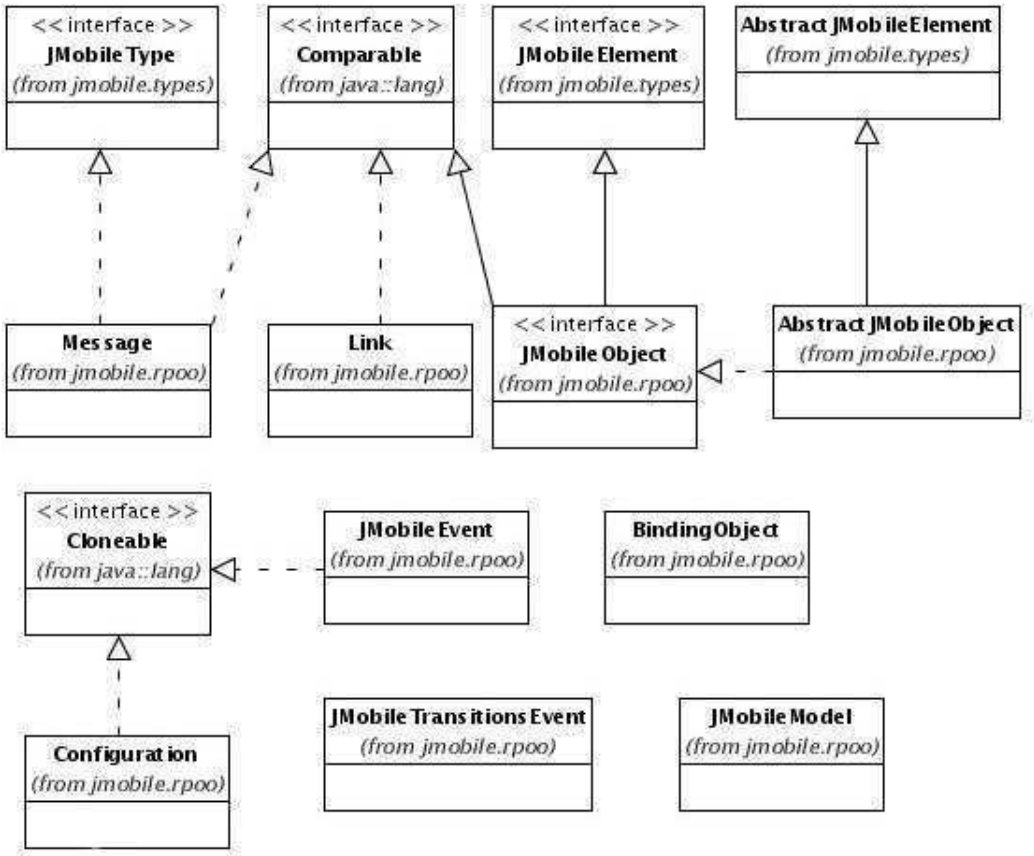


Figura 4.3: Diagrama de Classes do *package jmobile.rpoo*

As classes *Configuration*, *Message*, *Link* e as classes que estendem a classe abstrata *AbstractJMobileObject* formam a representação de uma Configuração RPOO. Para construirmos um modelo específico, como o modelo dos Filósofos, as classes RPOO do modelo estendem a classe AbstractJMobileObject que contém a implementação genérica para os métodos necessários para a API JMobile utilizar para a simulação do modelo.

A classe *JMobileModel* representa o modelo RPOO, que contém a configuração inicial para o estado inicial do modelo. A partir desta configuração podemos obter o Eventos de Transição (classe *JMobileTransitionsEvent*) que é composto por um conjunto de objetos da

classe *BindingObject*. Um *BindingObject* é um par que relaciona um objeto RPOO com uma Transição e seu *Binding* (Um binding de uma transição é a atribuição de valores à todas as variáveis da transição). O *BindingObject* identifica unicamente uma forma para o disparo de uma transição de um objeto. A classe *JMobileEvent* representa os eventos RPOO que são um conjunto de ações inscritas nas transições dos *BindingObject's* do *JMobileTransitionEvent*.

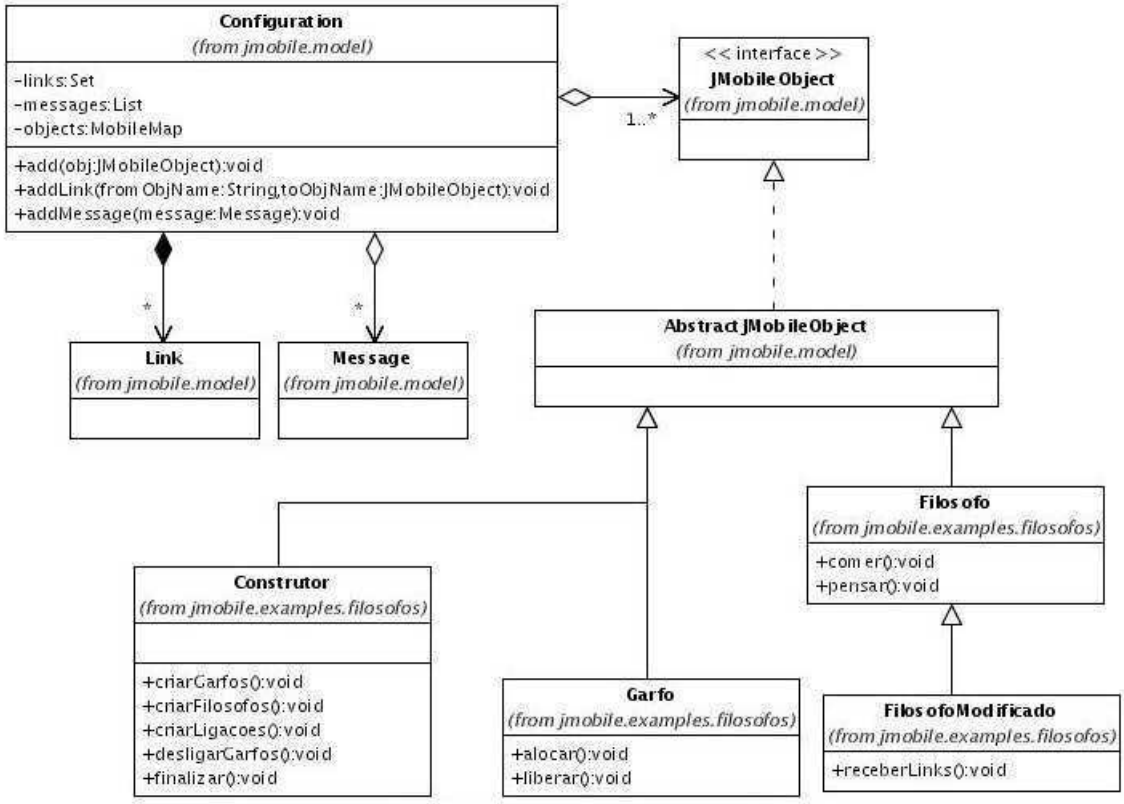


Figura 4.4: Diagrama parcial do *package jmobile.rpoo* com relacionamentos

Na Figura 4.4, além das classes para a representação da configuração e de seus componentes, temos também as classes: *Construtor*, *Filosofo*, *FilosofoModificado* e *Garfo*. Estas classes são componentes dos modelos dos Filósofos e dos Filósofos Modificados que fazem parte do *package jmobile.examples* e ilustram o relacionamento destas classes com as classes do *package jmobile.rpoo*. Todas as classes RPOO para os modelos estendem a classes *AbstractJMobileObject*.

4.2.2 *Package jmobile.types*

O *package jmobile.types* como já foi dito, define o conjunto de tipos que são utilizados como base para a representação dos estados de uma configuração. Além dos elementos na raiz do *package types*, ele é sub-dividido em três outros *subpackages*. Na Figura 4.5 temos o diagrama de classes para a raiz do *package jmobile.types* e nas Figuras 4.6, 4.7 e 4.8 temos os seus *sub-packages*.

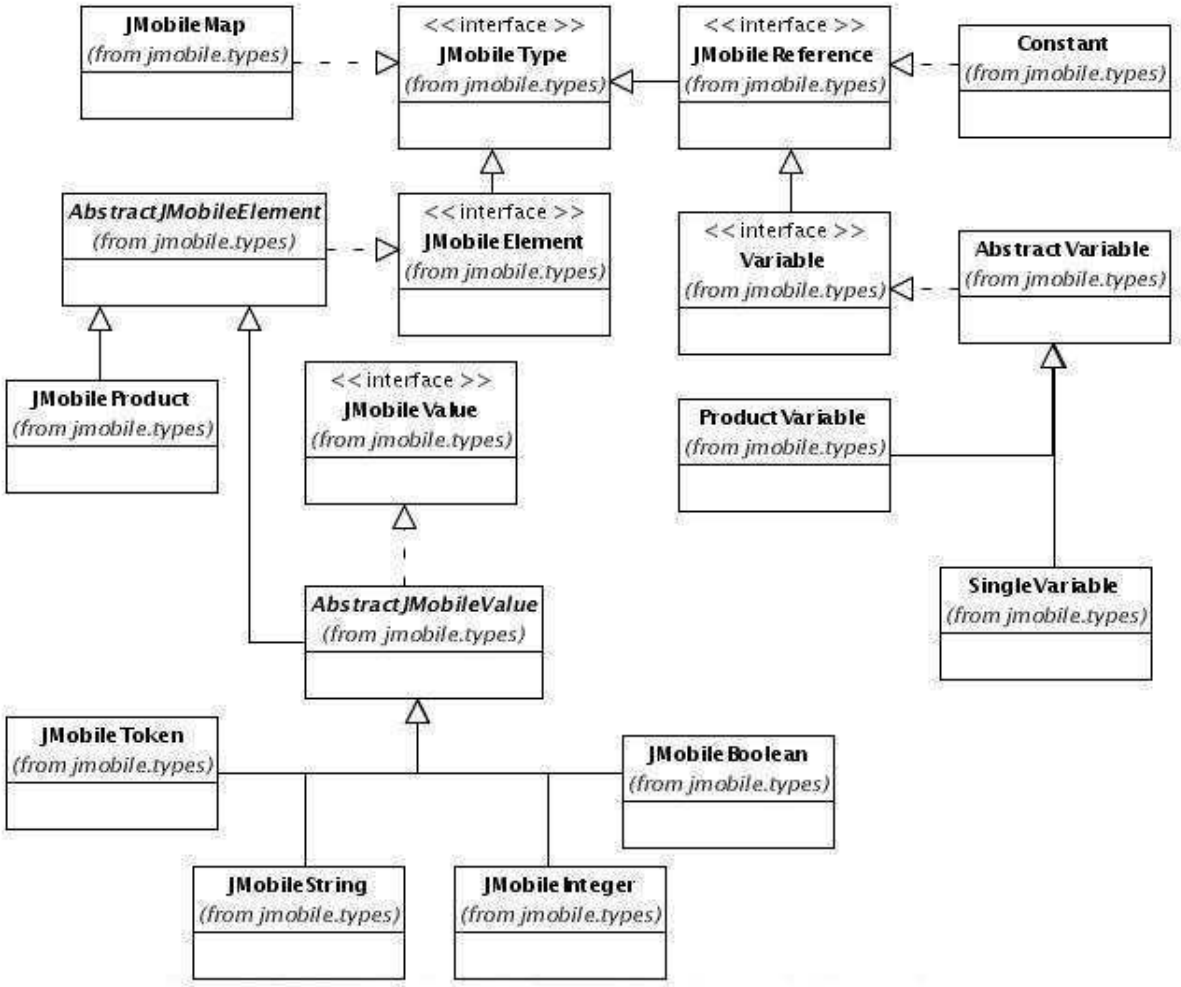


Figura 4.5: Diagrama de Classes do *package jmobile.types*

O topo da hierarquia de tipos é a interface *JMobileType*, todos os tipos usados na API devem implementar esta interface ou estender uma das classes que implementam a interface.

A API JMobile conta com os tipos básicos de dados *JMobileToken*, *JMobileString*, *JMobileInteger*, *JMobileBoolean* e o tipo de dado composto *JMobileProduct*. Estes tipos esten-

dem a classe abstrata *AbstractJMobileValue* que implementam os métodos genéricos para o tratamento de valores na API. Além disso, as classes *JMobileString*, *JMobileInteger* e *JMobileBoolean* encapsulam as classes *String*, *Integer*, *Boolean* do Java, acrescentando apenas os métodos definido para um *JMobileValue* e suas operações.

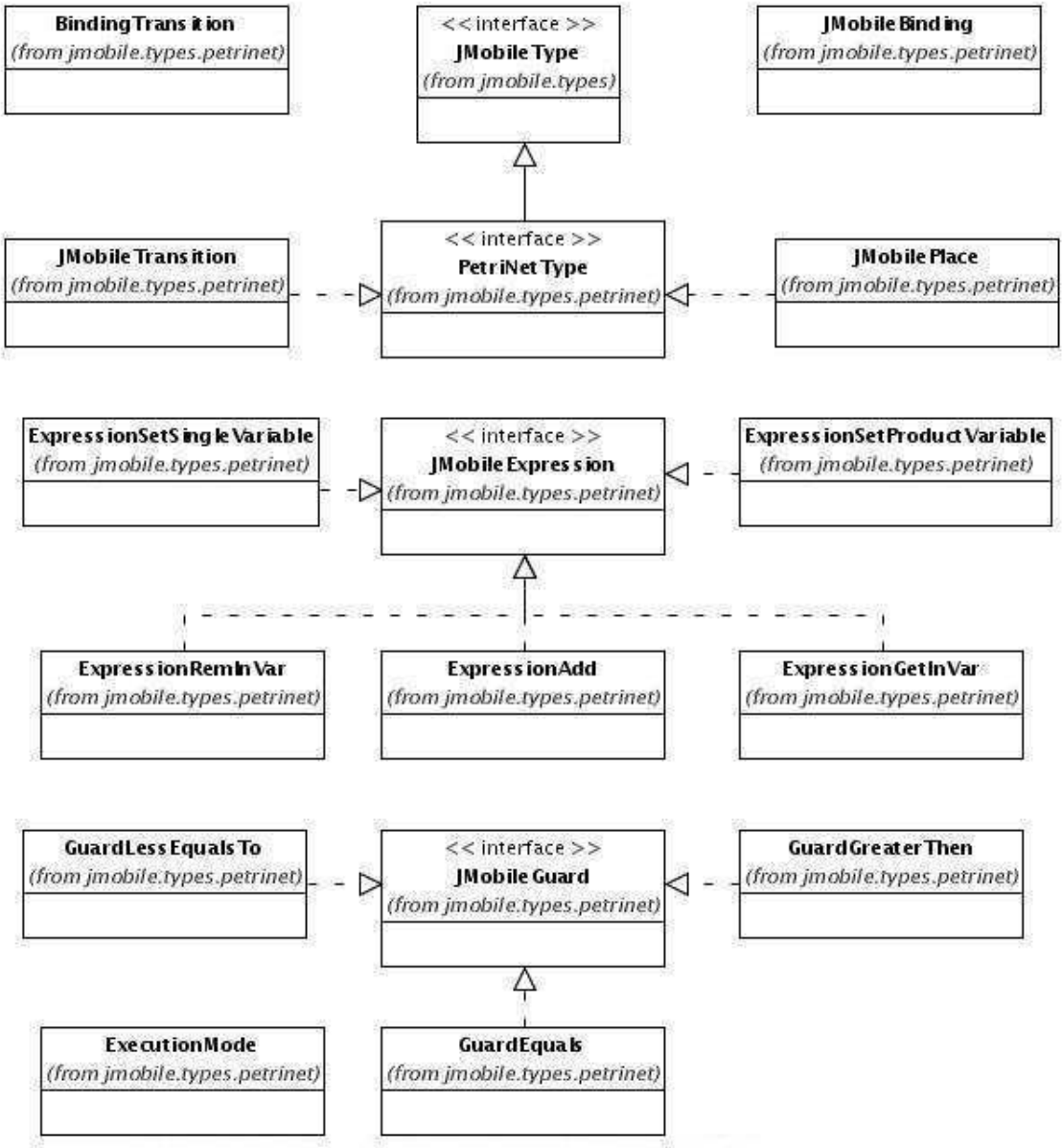


Figura 4.6: Diagrama de Classes do *package jmobile.types.petrinet*

Na Figura 4.6 temos o *sub-package jmobile.types.petrinet* que contém classes para a representação da estrutura e do mecanismo de simulação das redes de Petri. Os lugares e transições são representados pelas classes *JMobilePlace* e *JMobileTransition*, que imple-

mentam a interface *PetriNetType*. As expressões de arcos são representadas pelas classes que implementam a interface *JMobileExpression* e as guardas pelas classes que implementam a interface *JMobileGuard*. Todos esses elementos de uma rede de Petri são utilizados nas classes que estendem a classe *AbstractJMobileObject* do *package jmobile.rpoo*. A classe *AbstractJMobileObject* tem os métodos para manipular estes elementos e é também a classe que representam o tipo rede de Petri, não representado explicitamente na API.

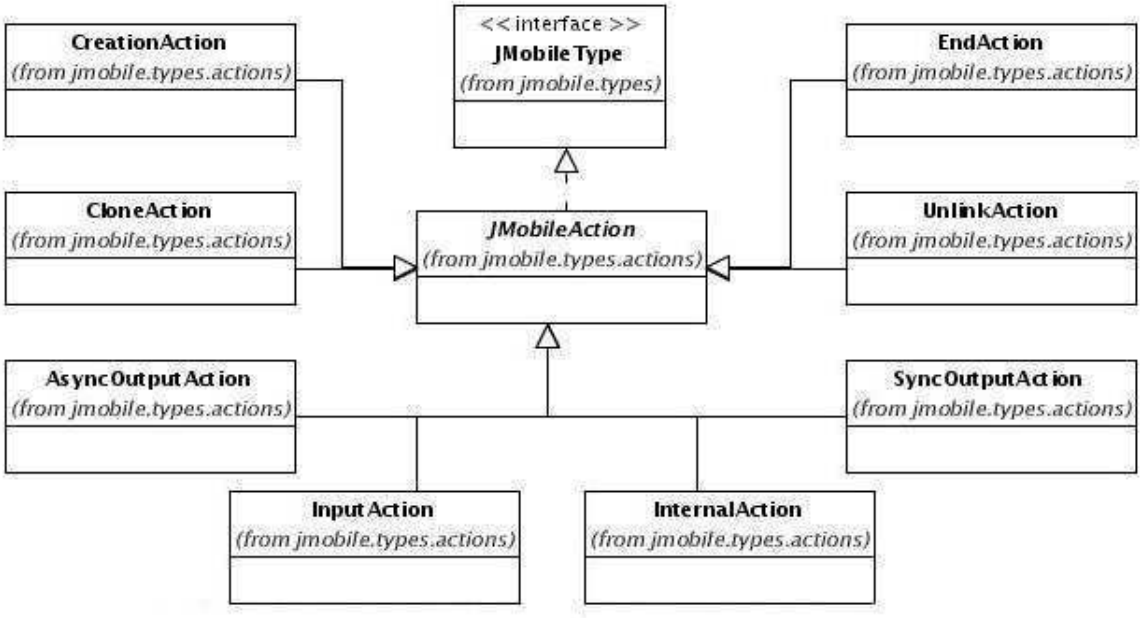


Figura 4.7: Diagrama de Classes do *package jmobile.types.actions*

Já na Figura 4.7 temos o *sub-package* que traz as classes de representação das ações RPOO, que foram definidas na Tabela 3.1 da Página 28. A única diferença para as ações definidas para RPOO é o acréscimo da ação de clonagem de objetos, classe *CloneAction*, que pode ser definida como uma instanciação de objeto (CreateAction) onde o estado inicial da nova instância é determinado por um dos objetos RPOO da mesma classe já existente. Esta *nova* ação foi necessária para a criação e simulação do modelo do sistema de conferência, Capítulo B.

Por último, na Figura 4.8 na Página 4.8, temos o *sub-package jmobile.types.ss* que contém a classe *JMobileStatesSpace* para a representação de espaço de estados dos modelos RPOO. Esta classe é na realidade uma estrutura de dados em forma de grafo para o armazenamento dos estados, configurações (*Configuration*), possíveis de serem alcançados.

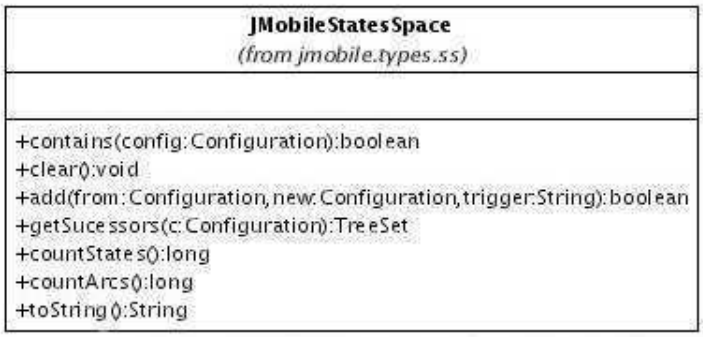


Figura 4.8: Diagrama de Classes do *package jmobile.types.ss*

4.2.3 *Package jmobile.tools*

Na Figura 4.9 temos as classes que implementam os protótipos para o simulador e para o gerador de espaço de estados de modelos RPOO.

A classe *JMobileSimulator* tem apenas um método que permite a simulação de um modelo (JMobileModel) e guarda o trace de simulação, ou seja, guarda a seqüencia de estados alcançados durante o processo de simulação. Detalharemos mais esta classe quando falarmos de simulação na Seção 4.4.

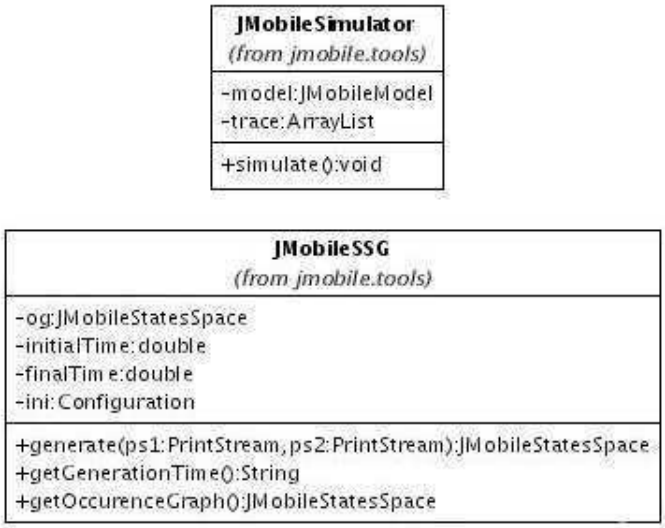


Figura 4.9: Diagrama de Classes do *package jmobile.tools*

Já a classe *JMobileSSG* implementa o algoritmo de geração de espaço de estados. Este protótipo gera o espaço de estados em pelo menos três formatos: Um formato próprio da

ferramenta, o formato para o verificador de modelos Veritas e um no formato Aldebaran. Os espaços de estados são gerados em arquivo de texto determinados pelos *PrintStream* passados como parâmetro no método *"generate(PrintStream, PrintStream)"*. Iremos detalhar um pouco mais o gerador na Seção 4.5.

4.2.4 *Package jmobile.parser*

Na Figura 4.10 temos as classes para o *package jmobile.parser*. A classe *XmlToJava* lê uma arquivo XML com a descrição da classe RPOO e usando o Parser XML Document Object Model (XML DOM) que define um padrão para o acesso e a manipulação de documentos XML. Fazendo uso das funcionalidades oferecidadas pela classe *XmlToJava*, a classe *JavaFileCreator* cria um arquivo Java com a descrição da classe RPOO.

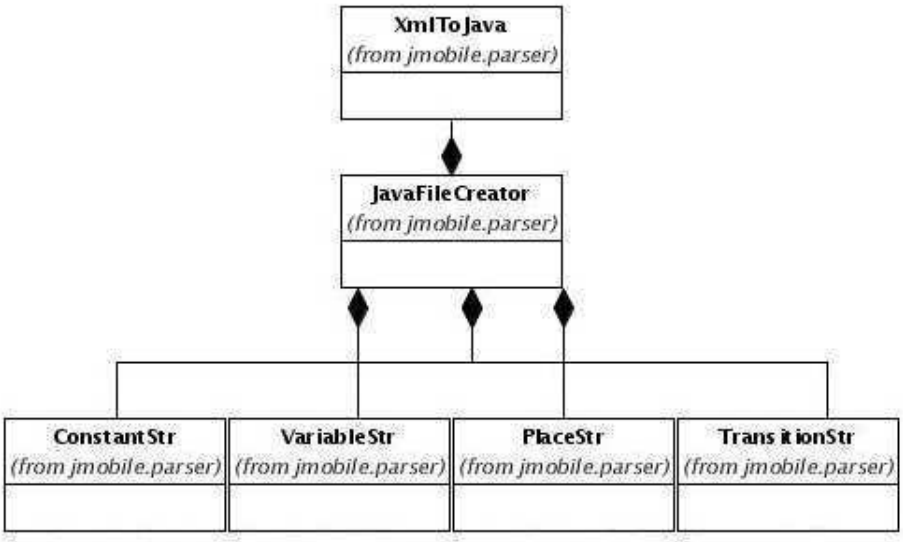


Figura 4.10: Diagrama de Classes do *package jmobile.parser*

4.3 Construindo Modelos

Nesta seção, apresentaremos algumas funcionalidades da API JMobile e como utiliza-lá na construção de modelos RPOO. Para isto, usaremos o modelo dos filósofos modificados apresentado em detalhes na Seção 3.3 do Capítulo 3.

4.3.1 Funcionalidades da API para a Construção de Modelos

Para construirmos um modelo RPOO usando a API JMobile precisamos definir uma configuração inicial do sistema modelado. Na realidade a configuração inicial é o modelo, ou seja, ela representa o estado de interesse do sistema. Desta forma, para construirmos uma representação do sistema em RPOO construímos uma *Configuração*.

Uma configuração, como já foi dito no Capítulo 2, é composta de um conjunto de Objetos RPOO, de ligações entre estes objetos e de mensagens pendentes. Cada objeto é de alguma classe RPOO, que é descrita por redes de Petri (representadas pela classe *AbstractJMobileObject*). A classe que representa uma Configuração RPOO do sistema é a classe *Configuration* do *package jmobile.rpoo* que contém as classes utilizadas nas definições dos modelos.

Antes de construirmos a configuração inicial do sistema temos que descrever as classes através das redes de Petri, como apresentamos no Capítulo 3, é necessária a descrição da rede usando-se um arquivo XML. Para cada classe de objetos RPOO presentes no modelo escrevemos um arquivo XML que a sua descrição. Esta representação será traduzida pela ferramenta *Parser* para código Java, para ser utilizada pela API na construção de instâncias de objetos RPOO.

O arquivo *.java* de descrição das classes RPOO deve estender a classe *AbstractJMobileObject* que é uma classe abstrata que implementa a interface *JMobileObject* com a definição de métodos necessário para a representação das redes. A classe *AbstractJMobileObject* e a interface *JMobileObject* são apresentadas na Figura 4.11 na Página 70. A interface *JMobileObject* define os principais métodos de suporte para a representação da estrutura e do comportamento de uma rede de Petri orientada a objetos. Já a classe abstrata *AbstractJMobileObject* tem as implementações genéricas necessárias para a simulação do mecanismo de disparo das redes de Petri acrescidas das representações para as ações de interação RPOO.

4.3.2 Construindo modelos RPOO

Para demonstrarmos a utilização das funcionalidades da API para a construção de modelos, iremos construir os modelos dos filósofos e o dos filósofos modificados, ambos já apresentados nos Capítulos 2 e 3. O diagrama de classes da Figura 4.4 contém as classes RPOO

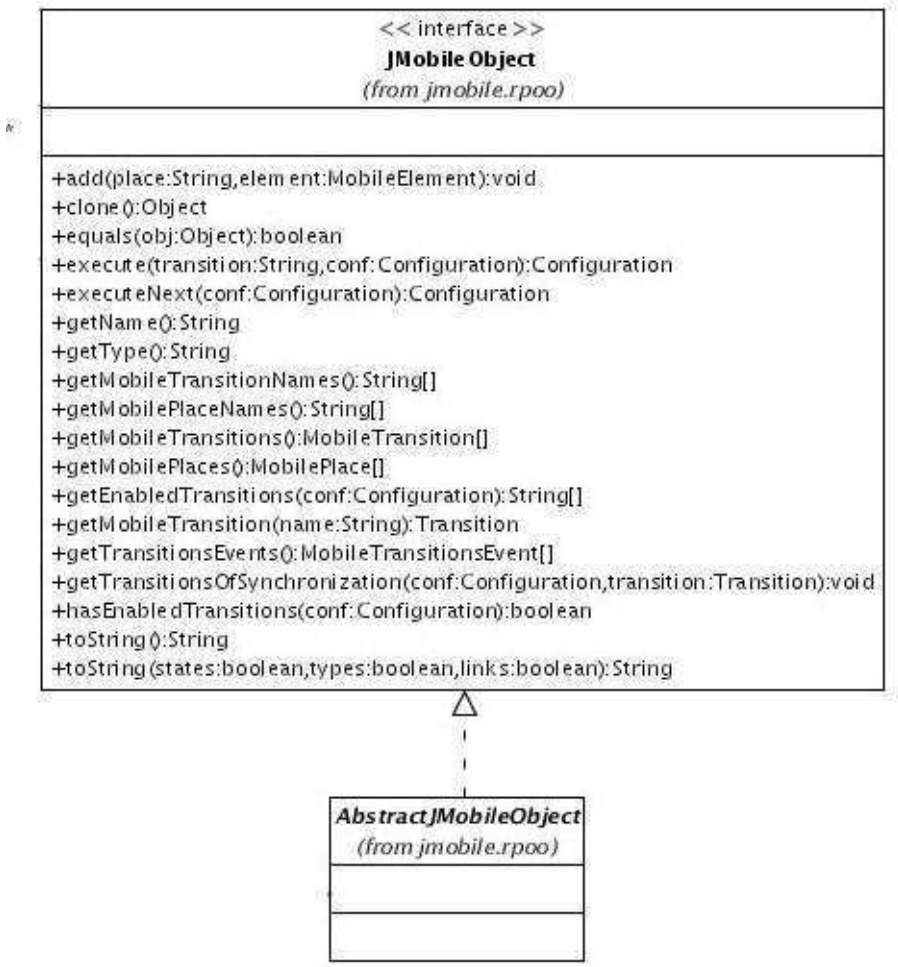


Figura 4.11: Interface *JMobileObject* e classe *AbstractJMobileObject*

presentes nos dois modelos para os filósofos. As classes para o modelo dos clássico dos filósofos são as classes Filósofo e Garfo e, para o modelo dos filósofos modificado são as classes FilosofoModificado, Construtor e Garfo. A classe garfo é a mesma, não é necessário fazer nenhuma modificação, a não ser a importação dos pacotes (*packages*) corretos.

Para construirmos uma representação do estado inicial de um modelo RPOO nós podemos editar uma classe *Main* do Java, usando os tipos definidos na API e os tipos definidos pelo modelador: tipos rpoo e tipos para representar dados. Uma classe *Main* deve conter pelo menos um método estático chamado *main*. Dentro deste método construímos nosso modelo e instanciamos um simulador para a simulação guiada do modelo.

Modelo dos Filósofos

As classes componentes do modelo dos Filósofos são as classes *Filósofo* e *Garfo*. A configuração inicial do modelo dos filósofos que iremos construir tem dois filósofos e dois garfos. O estado inicial para o modelo clássico dos filósofos com dois filósofos, na representação algébrica da configuração, é a seguinte:

$$confInicial = f1[g1\ g2] + f2[g1\ g2] + g1 + g2$$

Na Figura 4.12 temos um diagrama de seqüencia que ilustra o processo de construção do modelo dos filósofos utilizando as classes da API JMobile. Para construirmos a configuração inicial para o modelo temos que ter gerado os arquivos Java para a classes RPOO presentes no modelo.

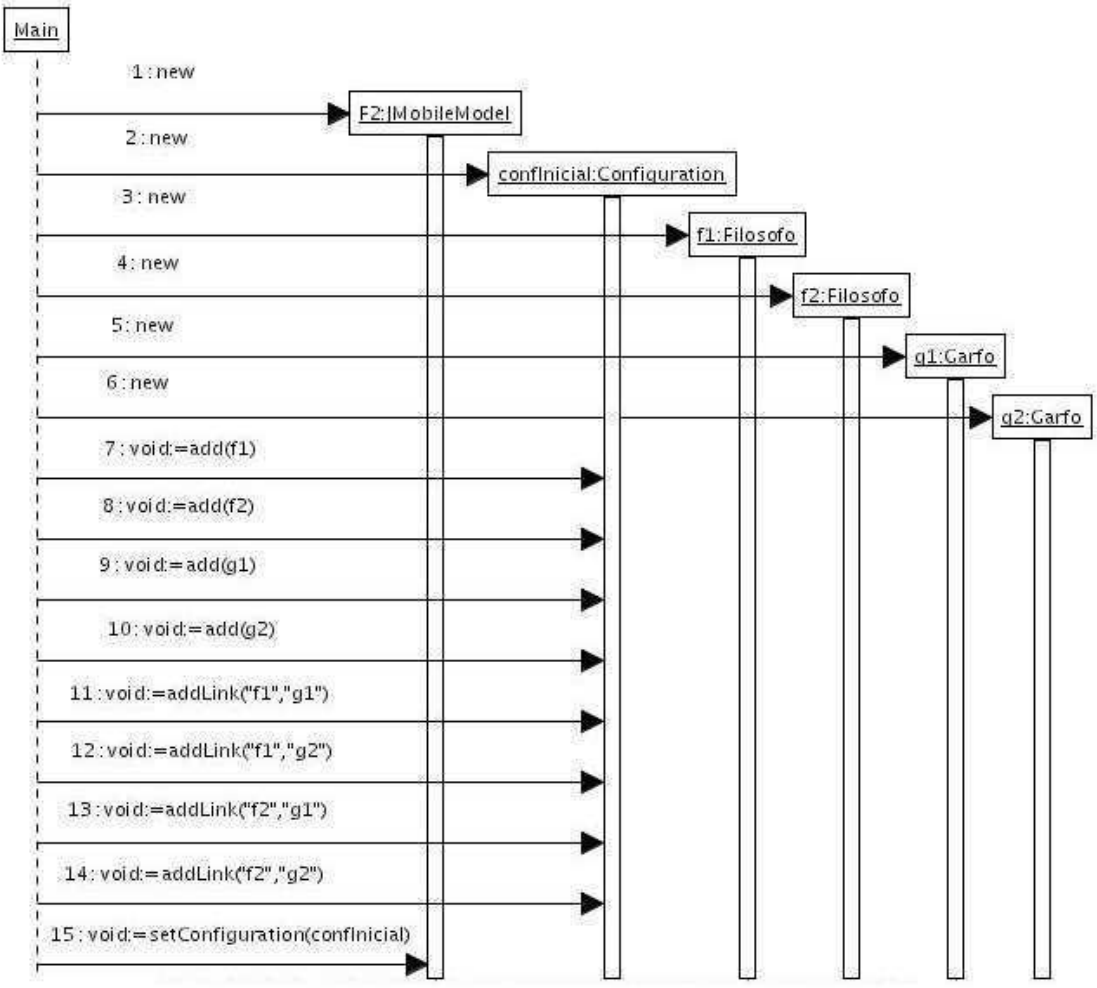


Figura 4.12: Diagrama do processo de construção do modelo clássico dos Filósofos

A classe Java *MainF2*, na Figura 4.13, traz a implementação do processo de construção para o modelo Clássico dos Filósofos apresentado na Figura 4.12.

```
01: import jmobile.rpoo.*;
02:
03: public class MainF2 {
04:     public static void main(String[] args) {
05:
06:         //Construindo o Modelo
07:         System.out.println(``<-- Modelo Filósofos Modificados - 2 filósofos -->\n'');
08:         JMobileModel f2model = new JMobileModel(``F2'');
09:
10:         //Construindo a Configuração ``Vazia''
11:         Configuration confInicial = new Configuration();
12:
13:         //Construindo os objetos RPOO das Classes ``Filosofo'' e ``Garfos''
14:         Filosofo f1 = new Filosofo(``f1'');
15:         Filosofo f2 = new Filosofo(``f2'');
16:         Garfo g1 = new Garfo(``g1'');
17:         Garfo g2 = new Garfo(``g2'');
18:
19:         //Adicionando os objetos a configuração
20:         confInicial.add(f1);
21:         confInicial.add(f2);
22:         confInicial.add(g1);
23:         confInicial.add(g2);
24:         //Adicionando as ligações entre os objetos
25:         confInicial.addLink(``f1'',``g1'');
26:         confInicial.addLink(``f1'',``g2'');
27:         confInicial.addLink(``f2'',``g1'');
28:         confInicial.addLink(``f2'',``g2'');
29:
30:         //Adicionando a configuração inicial ao modelo
31:         f2model.setConfiguration(confInicial);
32:     }
33: }
```

Figura 4.13: Código da classe MainF2

Modelo dos Filósofos Modificados

As classes componentes do modelo dos Filósofos Modificados são as classes *Construtor*, *FilósofoModificado* e *Garfo*. A configuração inicial do modelo que iremos construir tem apenas um objeto da classe *Construtor*. O estado inicial para o modelo clássico dos filósofos

com dois filósofos, na representação algébrica da configuração, é a seguinte:

confInicial = construtor

Na Figura 4.14 temos o diagrama de seqüencia para o processo de construção do modelo dos filósofos modificados.

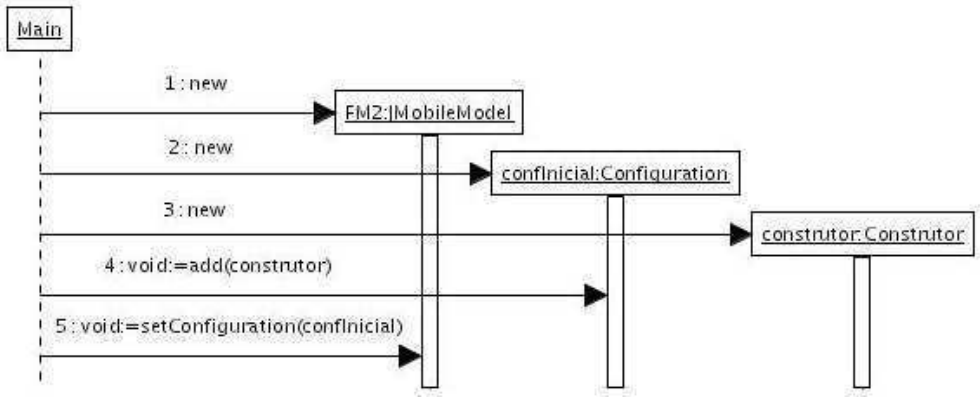


Figura 4.14: Diagrama do processo de construção do modelo dos Filósofos Modificados

A classe *MainFM2* para o modelo dos Filósofos Modificados é apresentada na Figura 4.15.

```
01: import jmobile.rpoo.*;
02: public class MainFM2 {
03:     public static void main(String[] args) {
04:
05:         //Construindo o Modelo
06:         System.out.println(``<-- Modelo Filósofos Modificados - 2 filósofos -->\n'');
07:         JMobileModel fm2model = new JMobileModel(``FM2'');
08:         Configuration confInicial = new Configuration();
09:         //Construindo o Objeto RPOO da Classe ``Construtor''
10:         Construtor construtor = new Construtor(``construtor'');
11:
12:         //Adicionando o objeto ``construtor'' a configuração
13:         confInicial.add(construtor);
14:
15:         //Adicionando a configuração inicial ao modelo.
16:  inicial fm2model.setConfiguration(confInicial);
17:     }
18: }
```

Figura 4.15: Código da classe MainFM2

Os modelos *"F2"* e *"FM2"* construídos nas classes *"MainF2"* e *"MainFM2"*, Figuras 4.13 e 4.15, representam a configuração inicial de cada um dos modelos. Como as classes Java são executáveis, pois contém o método estático *"main"* e, assim, podemos imprimir na tela a configuração dos modelos criados. Na Figura 4.16 temos a classe que representa um modelo RPOO.

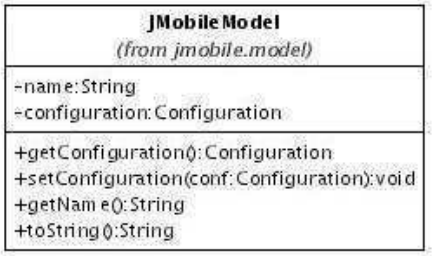


Figura 4.16: Classe *JMobileModel* do *package jmobile.model*

As Figuras 4.17 e 4.18 mostram as saídas da execução destas classes executáveis em um terminal *"shell"* no sistema operacional *"GNU/Linux"*.

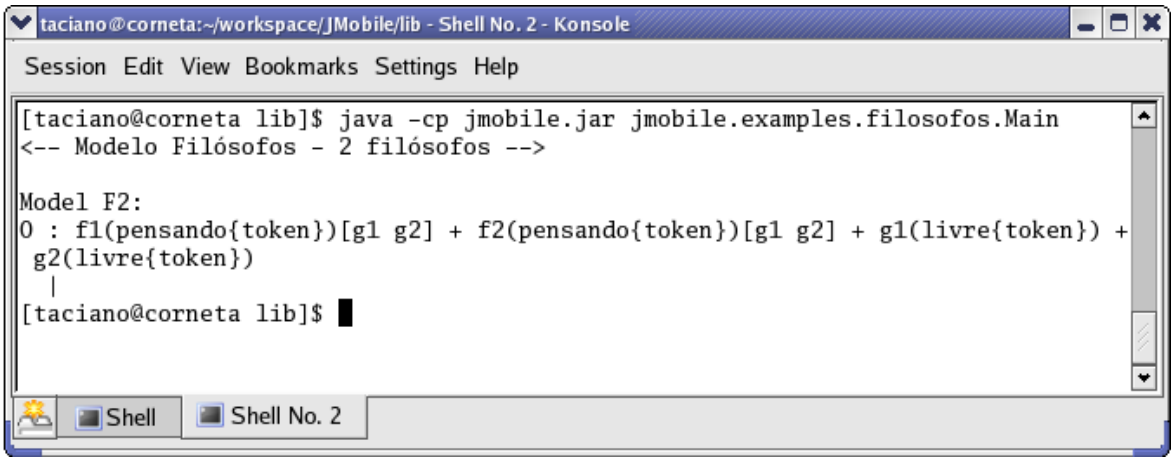


Figura 4.17: Executando a classe *MainF2*

Na representação textual apresentada nas saídas da execução das classes *MainF2* e *MainFM2*, nas Figuras 4.17 e 4.18 trazem um representação textual detalhada da configuração. Este representação além de mostrar o identificador e a estrutura, também trás os estados internos de cada objeto RPOO e as mensagens pendentes. O estado interno de um objeto, na mais é do que a lista de seus lugares e suas fichas. Note que a separação entre o string de

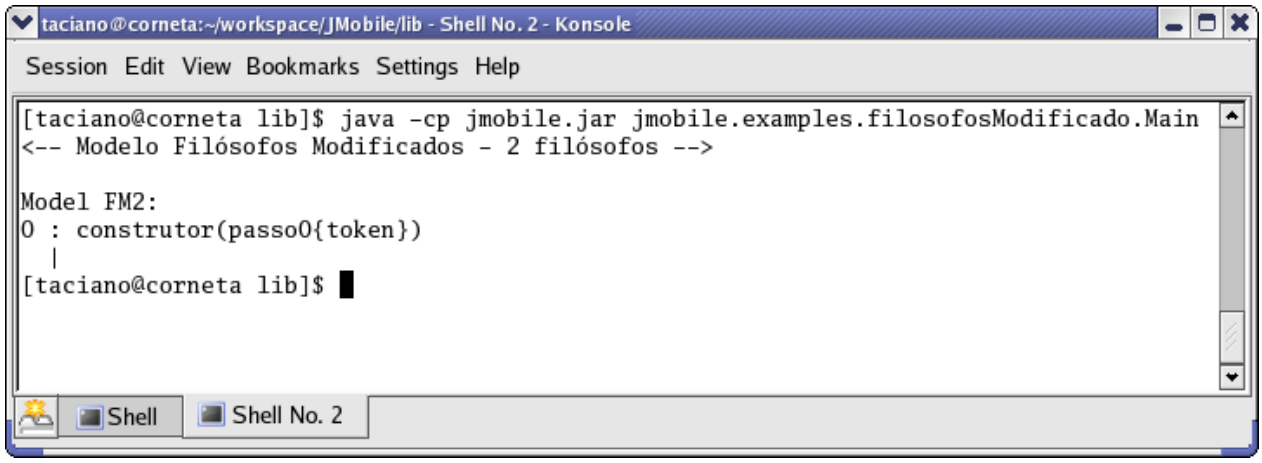


Figura 4.18: Executando a classe *MainFM2*

representação da estrutura e das mensagens é o símbolo "pipe" (|) ou barra vertical. Observe que não temos mensagens pendentes nas configurações iniciais de nossos modelos.

4.4 Simulando Modelos

Nesta seção, apresentaremos algumas funcionalidades da API JMobile e como utiliza-lá na simulação de modelos RPOO. Para isto, usaremos o modelo dos filósofos construído na Seção 4.3.

4.4.1 Funcionalidades da API JMobile para a Simulação

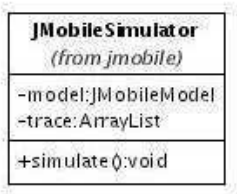


Figura 4.19: Classe *JMobileSimulator* do *package jmobile*

Para simularmos um modelo RPOO usando a API JMobile precisamos construir um *JMobileModel* como mostrado na Seção 4.3. Como o modelo construído, nós instanciamos da classe *JMobileSimulator*, que é um protótipo de simulador para RPOO, que faz uso das funcionalidades da API para simulação. O simulador é uma classe que fornece um *prompt*

de comando através do método *simulate()*. Nas Figuras 4.19 e 4.20, apresentamos a classe *JMobileSimulator* e o seu método *simulate()*.

```
01: public void simulate() {
02:     int nEvents = 0;
03:     int cid = 0;
04:     while (true) {
05:         cid=trace.size()-1;
06:         JMobileTransitionsEvent[] mtEvents;
07:         System.out.println("-----------------------------------------------");
08:         Configuration c = (Configuration)trace.get(cid);
09:         System.out.println(c);
10:         System.out.println("\n<<< Eventos Conf "+cid+" >>>\n");
11:         mtEvents = c.getMobileTransitionsEvent();
12:         for (int i = 0; i < mtEvents.length; i++) {
13:             System.out.println(i + ": " + mtEvents[i]);
14:         }
15:         System.out.print("\nDisparar evento n.º? ");
16:         nEvents = Integer.parseInt(
17:             (new BufferedReader(new InputStreamReader(System.in))).readLine()
18:         );
19:         System.out.println();
20:         if (mtEvents.length >= 0 && nEvents < mtEvents.length && nEvents >= 0) {
21:             Configuration confTemp = c.executeNew(mtEvents[nEvents]);
22:             trace.add(confTemp);
23:             System.out.println("-----------------------------------------------");
24:         } else {
25:             System.err.println("\n Não existe evento para o número digitado.");
26:         }
27:     }
28: }
```

Figura 4.20: Método *simulate()* da Classe *JMobileSimulator* do *package jmobile*

4.4.2 Simulando modelos RPOO

Para simularmos os modelos Filósofos e Filósofos modificados precisaremos acrescentar as seguintes linhas nos códigos das classes executáveis *MainF2* e *MainFM2*.

JMobileSimulator sim = new JMobileSimulator(f2model);

sim.simulate();

Ao iniciarmos a execução desta classe obtemos o *prompt* mostrado na Figura 4.21. O prompt imprime a representação algébrica da um configuração com o estado interno dos

objetos RPOO. Em seguida, logo depois do texto <<< ***Eventos Conf 0*** >>> são apresentados os eventos possíveis para a esta configuração. No processo de cálculo dos eventos, cada variável presente no modelo pode assumir um dos valores possíveis para ela. No caso de variáveis que referenciam objetos RPOO, ou seja, ligações, a ferramenta pode variar as atribuições. Por exemplo, os eventos *"0"* e *"1"* e os eventos *"2"* e *"3"* são praticamente iguais. Estes eventos geram os mesmos eventos RPOO como podemos ver logo depois do símbolo ===>>, a única diferença é a ordem de atribuição das variáveis.

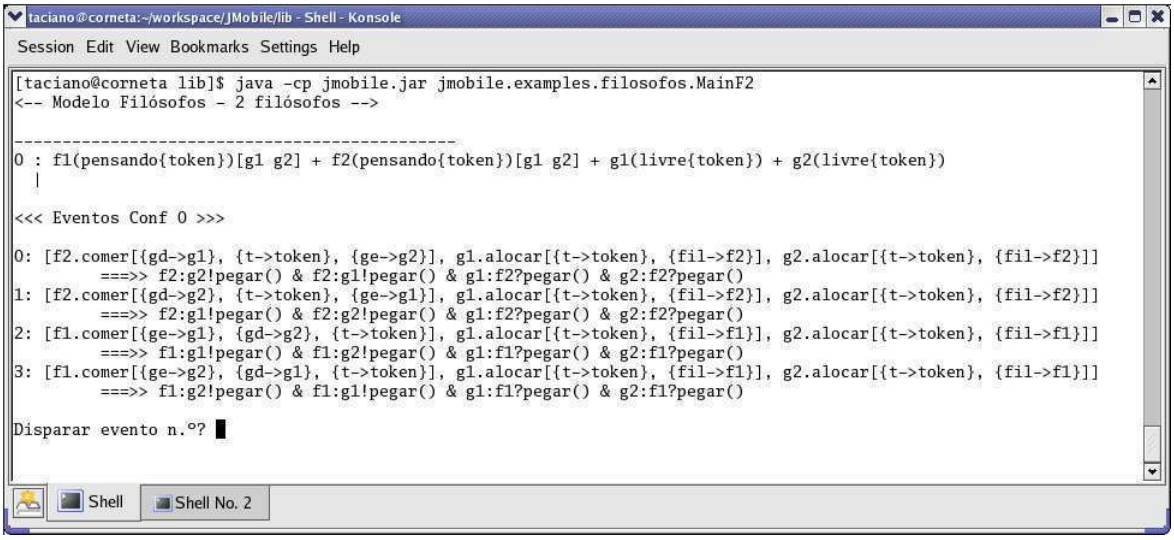


Figura 4.21: Executando a classe *MainF2* com a criação de um simulador

Logo abaixo dos eventos, o *prompt* imprime o texto ***Disparar Evento n.º?*** , onde podemos digitar o número de um dos quatro eventos possíveis. O simulador guarda as configurações alcançadas em um trace de execução e desta forma o engenheiro de software poderá analisar os traces para a identificação de problemas.

4.5 Gerando Espaço de Estados

Apresentaremos nesta seção o gerador de espaço de estados e sua utilização, para isso vamos utilizar o modelo do filósofo e gerar o espaço de estados para várias instâncias do modelo. Será utilizado para a geração do espaço de estados um computador com processador com 1.8Ghz e 1Gb de memória RAM. Todos os espaços de estados foram gerados no mesmo computador e cronometramos o tempo para a geração.

Para gerarmos o espaço de estado acrescentaremos as seguintes linhas de código na Figura 4.22 a classe *MainF2* na Página 72:

```
34: JMobileSSG ssg = new JMobileSSG(confInicial, true);
35: JMobileStatesSpace og = ssg.generate
36: (
37:    new PrintStream(
38:       new  FileOutputStream("/home/taciano/workspace/JMobile/filosofo2.eee")
39:    ),
40:    new PrintStream(
41:       new FileOutputStream("/home/taciano/workspace/JMobile/filosofo2.aut")
42:    )
43: );
44:
45: System.setOut(
46:    new PrintStream(
47:       new FileOutputStream("/home/taciano/workspace/JMobile/filosofo2.veritas")
48:    )
49: );
50: System.out.print(og.toStringVeritas());
```

Figura 4.22: Executando o gerador de espaço de estados

O código na Figura 4.22 gera o espaço de estados do modelo em três formatos em arquivos separados. Na linha 34 instanciamos o gerador, classe *JMobileSSG*, passando a configuração inicial do modelo e um booleano informando se queremos um espaço de estados ordenado ou não pelo identificador dos estados. Dá linha 35 à linha 43, temos a chamada ao método *"ssg.generate"* passando como parâmetro dois *PrintStream's*(fluxo de saída para gravação em arquivo) para os arquivos "filosofo2.eee" e "filosofo2.aut". O arquivo "filosofo2.eee" contém o espaço de estados padrão, onde a representação textual traz as configurações alcançáveis. Já o arquivo "filosofo2.aut" contém o espaço de estados no formato aldebaran. Estes dois formatos são gerados em tempo de execução do gerador, ou seja, a cada nova configuração alcançada é gravada uma linha nos arquivos para esta configuração.

Para gerar o espaço de estado no formato Veritas, é preciso gerar todo o espaço de estados na estrutura de dados da classe *JMobileStatesSpace*. Isto é necessário porque o formato Veritas necessita de informações de estados predecessores e sucessores para fazer a verificação e para obtermos estas informações necessitamos do espaço de estados completo.

4.6 Alguns Números

Executamos o gerador para o modelo dos Filósofos com 2, 3, 4, 5, 6, 7, 8 e 9 filósofos e medimos o tempo para a geração total do espaço de estados. Para cada caso, foi gerado o espaço de estados. Na Tabela 4.6 apresentamos os resultados da geração, informando o número de estados, o número de arcos e o tempo médio decorrido.

Qtd Filósofos	N.º Estados	Nº Arcos	Tempo Médio
2	9	12	0.00308 min
3	34	72	0.10855 min
4	117	336	0.03210 min
5	391	1410	0.11861 min
6	1296	5616	0.64061 min
7	4285	21672	1.63293 min
8	14157	81840	7.22157 min
9	46762	304128	4.65000 horas

Tabela 4.1: Estatísticas da geração de espaço de estados — Modelo Filósofos

É conhecido o problema da explosão do espaço de estados no modelo dos filósofos e até mesmo ferramentas para Redes de Petri com anos de desenvolvimento apresentam dificuldades em tratar este problema inerente de alguns sistemas. O nosso trabalho não buscou respostas para este problema contudo vários outros trabalhos, dentro do Grupo de Métodos Formais/UFCG, estão atacando o problema tendo como base a API JMobile.

A tentativa de geramos o espaço de estados para um modelo com 10 filósofos resultou em estou de memória. O número de estados para 10 instâncias de filósofo é aproximadamente 153.000 estados, onde já temos um crescimento considerável.

Na geração do espaço de estados do Modelo do Sistema de Conferência obtivemos os seguintes números, onde temos 9 classes e 11 instâncias de objetos no modelo:

- Estados = 85.325
- Arcos = 175.573
- Tempo = 20 horas

Foram utilizados computadores do tipo PC com 2.000 MegaHetz e 512 MB de memória.

4.7 Restrições Atuais e Futuras Extensões

A API JMobile como foi apresentada é funcional e atende aos objetivos propostas para ela. Seu tempo de geração de espaço de estados é aceitável. Contudo, ela ainda necessita de extensões para a melhoria no suporte a outros tipos de dados e também uma melhora na eficiência do gerador.

A API JMobile suporta modelos com todos os tipos de ações RPOO, mas no caso de ações de saída síncronas e entradas síncronas só é possível um nível de sincronização. Ou seja, só é possível sincronizar pares de transições. Nas transições só podemos ter uma inscrição de consumo de mensagens síncronas, entretanto podemos ter qualquer quantidade de ações de saída síncronas.

Até o momento a API suporta os tipos elementares *JMobileBoolean*, *JMobileInteger*, *JMobileString* e *JMobileToken*. Em extensões futuras do JMobile teremos suporte a tipos enumerados, números reais (Double) e possívelmente, será permitido utilizar qualquer tipo Java nos modelos. O tipo tupla (JMobileProduct) necessita de uma melhor estruturação para facilitar o seu uso nas expressões.

O suporte a operações e expressões é outro ponto importante. A API já conta com um bom suporte a expressões e como está estruturada em forma de *framework* a inclusão de novas expressões é relativamente fácil. Já o suporte a operações sobre os tipos e a inclusão de código nos modelos precisa de uma análise e um planejamento para ser completamente funcional.

De forma geral, A API JMobile apresenta-se pronta e funcional para uma boa variedade de modelos. Sua estruturação em forma de *framework* facilitou a inclusão de funcionalidades de forma rápida para atender, por exemplo, ao estudo de caso. Permite a utilização de tipos Java definidos pelo usuário através de extensões de classes específicas da API.

Capítulo 5

Validação

A API JMobile conta com um conjunto de funcionalidades para a simulação e geração de espaço de estados de modelos JMobile com uma variedade de característica, entre elas as de concorrência e de distribuição.

Alguns destes modelos foram convertidos de RPOO para JMobile antes mesmo de termos uma versão do simulador e do gerador em condições adequadas. As ferramentas de simulação e geração do JMobile são usadas desde o início de seu desenvolvimento em vários modelos nos trabalhos de outros pesquisadores [FMdF05], e [FM04] e no trabalho sobre geração de espaço de estados distribuído. Com esta utilização obtivemos uma grande evolução do *framework* JMobile.

O mais importante nisto foi à confiança adquirida pelas ferramentas desenvolvidas a partir do *framework*. Com o uso do *framework* desde as primeiras versões a validação através de experimentos ocorreu de forma contínua, sendo feita com modelos definidos e utilizados em outros trabalhos já citados.

Cada um dos trabalhos de pesquisa que utilizou o nosso framework, fez uso de modelos diferentes tendo impacto positivo sobre a ferramenta. Isto, possibilitou uma evolução significativa e a implementação de funcionalidades pertinentes referentes aos sistemas modelados.

5.1 Os modelos

Apresentaremos nas subseções a seguir os modelos e as suas contribuições para a validação do *framework* Jmobile.

5.1.1 Modelo do Problema do Jantar dos Filósofos

Um dos modelos que utilizamos no JMobile foi o modelo que soluciona o problema do jantar dos filósofos, apresentado nos Capítulos 2 e 3.

Mesmo este modelo sendo bem simples foi muito util no desenvolvimento do *framework* e das primeiras versões para o simulador e o gerador. Como já sabemos, neste problema clássico temos ao redor de uma mesa um certo número de filósofos e cada filósofo tem um garfo que é compartilhado com seu vizinho esquerdo.

Construimos o modelo JMobile para o problema dos filósofos de forma genérica para que pudessemos alterar a quantidade de filósofos facilmente. Com este modelo fizemos simulações e geramos o espaço de estados para diversas configurações iniciais do modelo. A Tabela 4.6 mostra estatísticas referentes aos espaços de estados gerados para o modelo dos filósofos em um computador com processador com 1.8Ghz e 1Gb de memória RAM. Para podermos avaliar a ferramenta de geração de espaço de estados do JMobile, também geramos os espaços de estados para o modelo na ferramenta Design/CPN. Como foi dito no anteriormente, existe um algoritmo para a conversão de modelos RPOO para um modelo equivalente em redes de Petri coloridas. Como JMobile não alterou a estrutura das redes de Petri orientadas a objetos o algoritmo de conversão pode ser usado tranquilamente.

O número de estados e arcos do espaço de estados gerado no Design/CPN deve ser exatamente igual ao do modelo em JMobile. Este resultado é muito importante para a confiança na validade do espaço de estados gerado no JMobile, quando um modelo RPOO é convertido corretamente para um modelo CPN, o número de estados e arcos é o mesmo. Os espaços de estados gerados no Gerador JMobile e no Design/CPN para o modelo dos filósofos continham o mesmo número de estados e de arcos.

5.1.2 Modelo do Protocolo *Stop And Wait*

O modelo JMobile para o protocolo de comunicação *Stop And Wait* utilizado em pesquisas dentro do Grupo de Métodos Formais - DSC/UFCG, sobre geração de espaço de estados distribuídos, também está sendo simulado na ferramenta.

Ao gerarmos o espaço de estados para o modelo *Stop And Wait*, para algumas configurações, também obtivemos resultados significativos e o número de estados dos do espaço de

estados também foi exatamente igual ao do modelo equivalente gerado no Design/CPN.

5.1.3 Modelo do Sistema Conferência

Para obtermos uma confiança ainda maior sobre as ferramentas desenvolvidas sobre o *JMobile* para a geração e simulação de espaço de estados utilizamos o modelo para o Sistema de Conferência. O sistema conferência foi utilizado por Figueiredo [FMdF05] em seu trabalho de mestrado.

O Sistema Conferência foi desenvolvida por Guedes e pode ser encontrado em sua dissertação de Mestrado [Gue02a]. Trata-se de um Sistema de Apoio às Atividades de Comitês de Programa em Conferências. A aplicação gerencia atividades de comitês de programas de conferências, tais como submissão de artigos, processo de avaliação e notificação de aceitação ou rejeição de artigos aos autores.

A especificação e o código fonte podem ser encontrados em [Gue02a]. A aplicação foi desenvolvida em Java sobre a plataforma Grasshopper. Diagramas de classe, de seqüência e de colaboração de UML foram utilizados por Guedes para a especificação do sistema. A partir destes diagramas, foram construídos os modelos RPOO em XML e em Java usando a API JMobile, que serviram de entrada para o simulador e o gerador de espaço de estados.

Uma vez de posse dos modelos RPOO, podemos converte-lo para as representações em XML e em Java, que são as entradas para as ferramentas de simulação e geração do JMobile. Este processo de conversão foi feito manualmente e não iremos apresentá-lo aqui uma vez que, com o advento de uma ferramenta de edição de modelos RPOO, este processo seja realizado de forma automática por tal ferramenta.

É importante ressaltar aqui que o processo de conversão dos modelos para XML e Java, foi realizado em conjunto com Figueiredo [FMdF05]. A API JMobile foi utilizada por Figueiredo para a geração de espaço de estados do modelo conferência que é a entrada para o processo de geração automática de casos de teste definido em seu trabalho.

Sendo assim, mostraremos como a simulação foi realizado sobre os modelos e seus resultados, e, em seguida, mostraremos como foi realizada a geração de espaço de estados, apresentando algumas considerações sobre este espaço de estados.

Tanto as simulações quanto a geração do espaço de estados foram feitas partindo do cenário inicial. A representação algébrica para configuração inicial do sistema de conferência

é apresentada na Figura 5.1 e na Figura 5.2 temos a configuração inicial com a representação do estado interno cada um dos objetos JMobile.

```
0 : agcCoord[agentCoord net] + agcMemb[net] + agcRev1[net] + agcRev2[net]
  + agentCoord[agcCoord conf guicoord] + conf[agentCoord] + guicoord[agentCoord]
  + net[agcCoord agcMemb agcRev1 agcRev2]
  |
```

Figura 5.1: Configuração Inicial do Sistema de Conferência

```
0 : agcCoord(EveryThingGoesFromHere{token} and address{'agcCoord'})[agentCoord net]
  + agcMemb(EveryThingGoesFromHere{token} and address{'agcMemb'})[ net]
  + agcRev1(EveryThingGoesFromHere{token} and address{'agcRev1'})[ net]
  + agcRev2(EveryThingGoesFromHere{token} and address{'agcRev2'})[ net]
  + agentCoord(Criado{token})[ agcCoord conf guicoord]
  + conf(Executando{token} and dadosConferencia{'dadosconf1'})[ agentCoord]
  + guicoord(EndMembro{'agcMemb'} and QntCopias{1,2})[ agentCoord]
  + net(AgenciesAddress{
              ('agcCoord',agcCoord),('agcMemb',agcMemb),('agcRev1',agcRev1),('agcRev2',agcRev2)
        }
        and
        EverythingHere{token}
    )[agcCoord agcMemb agcRev1 agcRev2]
|
```

Figura 5.2: Configuração Inicial do Sistema de Conferência com Estados Internos

Simulação dos Modelos JMobile para o Sistema Conferência

A simulação dos modelos foi realizada através da ferramenta *JMobileSimulator* apresentado na Página 75 que disponibiliza informações sobre o estado atual do sistema e as transições habilitadas para disparo.

Durante a simulação do modelo, realizada em conjunto com Figueiredo [FMdF05], alguns erros de modelagem foram encontrados. Uma vez que os modelos em UML podem conter ambigüidades e serem incompletos, eles pode inserir falhas no modelo. Estas falhas foram encontradas durante o processo de simulação, e suas relativas correções foram feitas tanto no modelo UML como, conseqüentemente, no modelo JMobile.

Uma vez que os erros encontrados através da simulação foram simples, onde basicamente refletiam erros inseridos pelo modelo por questões de mal entendimento de especificações, etc., não entraremos em detalhes a respeito dos mesmos. No entanto, a descoberta de erros no

modelo e suas correções nos mostraram que a manutenção de dois modelos (UML e RPOO) foi simples.

Geração do Espaço de Estados para o Sistema Conferência

Para a geração do espaço de estados do modelo JMobile do sistema de conferência foi utilizado o gerador da API JMobile. O gerador de espaço de estados da API JMobile pode gerar o espaço de estados em dois formatos como mostrado na Seção 3.6 da Página 50. O processo de geração é totalmente automático e não exige qualquer tipo de intervenção humana.

Para o Sistema Conferência foi gerado o espaço de estados tanto no formato para o Veritas quanto para o formato Aldebaran. Os dois formatos são gerados ao mesmo tempo em arquivos separados. O espaço de estados do modelo contém 85.325 estados e 175.573 transições e foi gerado em um tempo aproximado de 20 horas e 25 minutos, em um computador com processador com 1.8Ghz e 1Gb de memória RAM. Este espaço de estados foi gerado para um modelo onde:

- um artigo poderia ter uma ou duas revisões, conseqüentemente um ou dois agentes de formulário de revisão estaria executando ao mesmo tempo;
- além do membro de comitê, um revisor poderia redirecionar o formulário para um outro revisor, solicitando ou não o seu retorno.

No Apêndice B o Sistema de Conferência é detalhado.

Uma vez que o espaço de estados do sistema sem quaisquer restrições, ou seja, para um número ideterminado de formulários de revisão e um número ideterminado de revisores, seria infinito, houve a necessidade de se assumir tais restrições apresentadas acima. Entretanto, tais restrições não comprometem o estudo realizado neste capítulo, pois preservam o comportamento dos agentes envolvidos.

5.2 Considerações Finais

A utilização do modelo para o Sistema Conferência, um sistema de maior porte e mais complexo que os utilizado até então, foi de grande utilidade para o desenvolvimento, consoli-

dação e validação da API JMobile, a partir da qual foram desenvolvidas as ferramentas de simulação e geração de espaço de estados para modelos JMobile.

Um outro ponto importante foi que o modelo para o Sistema Conferência era necessário em outro trabalho e foi neste trabalho que foram utilizadas as ferramentas de suporte a RPOO da API JMobile. Ou seja, a api foi utilizada na prática por um engenheiro que não estava diretamente ligado ao nosso trabalho de desenvolvimento.

Validação da API JMobile através de experimentos Além dos nossos próprios experimentos, apresentados neste trabalho de dissertação, a utilização das ferramentas JMobile em outros trabalhos de pesquisa trouxe maior confiança para a validade das ferramentas desenvolvidas a partir do *framework* JMobile. No caso do Sistema Conferência não foi possível a geração do espaço de estados no Design/CPN devido a alta demanda de tempo para a conversão do modelo em JMobile para o modelo equivalente em CPN. Contudo, o método de geração de casos de teste foi utilizado e foram obtidos casos de teste válidos. Estes casos de testes quando aplicados sobre a implementação encontraram erros na implementação.

Validação da API JMobile através de testes Após o trabalho de Rodrigues [Rod04] desenvolvedor do Verificador de Modelos para RPOO — Veritas, foi desenvolvidos testes para a validação do espaço de estados de modelos RPOO [MOBR04]. Todos os espaço de estados gerados através da ferramenta de geração JMobile passaram nos testes feitos para garantir a validade do espaço de estados de entrada do Veritas.

Equivalência em os espaços de estados JMobile e Design/CPN Os modelos para o problema dos Filósofos e para o protocolo de comunicação *Stop and Wait* nos confirmaram a validade do método (algoritmo) para a simulação e geração de espaço de estados.

Desta forma, temos confiança em dizer que o simulador e o gerador geram estados válidos para o sistema modelado. E que alguns problemas no espaço de estados são ocasionados pelos modelos e simples modificações nos modelos podem remover estes erros. Contudo, ainda estamos em um processo de evolução da ferramenta e temos outros trabalhos a iniciar que irão utilizar a API JMobile em seu experimentos e isso trará um evolução e uma confiança ainda maior sobre o JMobile.

Capítulo 6

Conclusões

Neste capítulo, apresentamos os principais resultados do trabalho, sumarizando sua relevância dentro do contexto da verificação de modelos em RPOO. Por fim, discutimos as limitações das soluções propostas para a simulação e geração de espaço de estados e identificamos possíveis desdobramentos de pesquisa e desenvolvimento para trabalhos futuros.

O problema central atacado por este trabalho foi a construção de um simulador e de um gerador de espaço de estados para RPOO. Em outras tentativas de solucionar este problema, buscou-se representar de forma separada cada uma das perspectivas do RPOO, ou seja, tentou-se construir um simulador de sistemas de objetos e um simulador de redes de Petri. Um terceiro módulo seria necessário para fazer a integração destes dois simuladores.

Em nossa solução atacamos o problema de forma integrada, ou seja, fizemos através do desenvolvimento de um pacote de software (API JMobile) uma representação completa para RPOO. Esta abordagem do problema, aliada a linguagem Java, permitiu-nos uma construção incremental partindo de problemas e modelos particulares para problemas e modelos mais genéricos.

O simulador da API JMobile permite dois tipos de simulação: guiada e automática. A simulação guiada permite ao engenheiro de software simular cenários específicos e problemáticos do sistema modelado, visando uma validação de tais modelos através da análise do comportamento do sistema nestes caminhos (*traces*) de simulação. A simulação automática possibilita gerar um ou mais caminhos, que como foram gerados automaticamente podem encontrar cenários não pensados durante a fase de especificação do sistema.

Antes da conclusão do trabalho apresentado neste documento, não havia suporte ferra-

mental adequado para a simulação e geração de espaço de estados de modelos RPOO. Ao concluirmos nosso trabalho podemos dizer que RPOO, através de sua evolução para a notação JMobile, agora tem um suporte para todas as etapas do processo de simulação, geração de espaço de estados e verificação de modelos. Foram desenvolvidos os seguintes artefatos e as seguintes ferramentas como resultados de nosso trabalho:

- Uma representação em XML para modelos JMobile;
- Uma representação em Java para modelos JMobile;
- Uma API de Parser para a tradução automática da representação XML para a representação Java.
- Uma API para a construção e simulação de modelos em JMobile, a partir de sua representação Java.
- Um protótipo do simulador de modelos JMobile.
- Uma API para a geração de espaço de estados de modelos JMobile.
- Um protótipo para o gerador de espaço de estados de modelos JMobile, que gera em três formatos diferentes.

A união das API's, acima, formam os *packages* que compõem a API Java JMobile de suporte a simulação automática e a geração de espaço de estados de modelos em JMobile.

Com a utilização de um suporte ferramental desenvolvido a partir da API JMobile, o formalismo poderá ser utilizado para a validação e a verificação de sistemas maiores e mais complexos do que os tratados até então. Além disso, podemos agora fazer a verificação automática dos modelos existentes. Desta forma, o engenheiro de software poderá usar estas ferramentas de mais alto nível para um desenvolvimento mais rigoroso de seus sistemas, através de uma avaliação mais concreta e realista da qualidade do software em desenvolvimento.

Além disso, a disponibilidade de ferramentas de suporte à simulação permitirá automatizar o processo de validação desses modelos, que até o momento era feito manualmente. A possibilidade de gerar automaticamente o espaço de estados permite a verificação dos modelos em relação a especificações em CTL através do verificador de modelos. Finalmente,

devemos observar que o suporte ferramental será de fundamental importância para permitir futuros avanços na teoria de redes de Petri orientadas a objetos.

Ao desenvolvermos avanços na notação e no suporte ferramental, estamos contribuindo para diminuir a distância e as barreiras existentes entre a teoria e o uso de métodos formais para a prática da engenharia de software.

Bibliografia

[BHPV00] G. Brat, K. Havelund, S. Park, e W. Visser. Java pathfinder - a second generation of a java model checker. In *Workshop on Advances in Verification*, Julho 2000.

[Con04] W3C World Wide Web Consortium. Extensible markup language (xml). Online: http://www.w3.org/XML, 2004.

[Fer89] Jean-Claude Fernandez. Aldebaran: A tool for verification of communicating processes. Relatório técnico, Rapport SPECTRE, C14, Laboratoire de Génie Informatique - Institut IMAG, Grenoble - França, Setembro 1989.

[FM04] A. L. L. Figueiredo e P. D. L. Machado. Geração automática de casos de teste para sistemas baseados em agentes móveis. In *III Workshop de Teses e Dissertações em Qualidade de Software*, Brasília - Brasil, Junho 2004.

[FMdF05] A. L. L. Figueiredo, P. D. L. Machado, e J. C. A. de Figueiredo. Geração automática de casos de teste para sistemas baseados em agentes móveis. Dissertação (Mestrado), COPIN - Universidade Federal de Campina Grande, 2005.

[GGPdF01] E. L. Gallindo, D. D. S. Guerrero, A. Perkusich, e J. C. A. de Figueiredo. Aplicações de uma Notação baseada em Redes de Petri e Orientação a Objetos: Um Experimento de Modelagem. In *IV Workshop de Métodos Formais*, Rio de Janeiro - Brasil, Outubro 2001.

[Gla04] Robert L. Glass. The mystery of formal methods disuse. *Commun. ACM*, 47(8):15–17, 2004.

[GS03] F. V. A. Guerra e T. M. Silva. Formal object-oriented modeling and validation

of mobile ip. Relatório técnico, Universidade Federal de Campina Grande, Outubro 2003.

[Gue02a] F. P. Guedes. Um modelo para o desenvolvimento de aplicações baseadas em agentes móveis. Dissertação (Mestrado), Universidade Federal de Campina Grande, 2002.

[Gue02b] D. D. S. Guerrero. *Redes de Petri Orientadas a Objetos*. Tese, Curso de Pós-graduação em Engenharia Elétrica, Universidade Federal da Paraíba – Campus II, Campina Grande, Paraíba, Brasil, Abril 2002.

[Hei98] C. L. Heitmeyer. On the need for practical formal methods. In *FTRTFT '98: Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 18–26, London, UK, 1998. Springer-Verlag.

[HP00] K. Havelund e T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer STTT*, 2(4), Abril 2000.

[Jen92] K. Jensen. *Coloured Petri Nets 1: Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer-Verlag, Berlin, Alemanha, 1992.

[Jen94] K. Jensen. *Coloured Petri Nets 2: Basic Concepts, Analysis Methods and Practical Use*, volume 2. Springer-Verlag, Berlin, Alemanha, 1994.

[Jen97] K. Jensen. *Coloured Petri Nets 3: Basic Concepts, Analysis Methods and Practical Use*, volume 3. Springer-Verlag, Berlin, Alemanha, 1997.

[JF88] R. E. Johnson e B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[JF92] R. E. Johnson e B. Foote. Documenting Frameworks Using Patterns. *Proc of OOPLSA*, 1992.

[JR91] R. E. Johnson e V. F. Russo. Reusing Object-Oriented Design. 1991.

[KCJ98] L. M. Kristensen, S. Christensen, e K. Jensen. The practitioners Guide to Coloured Pertri Nets. *International Journal on Software Tools for Technology Transfer STTT*, 2(2):98–132, 1998.

[MOBR04] P. D. L. Machado, E. Oliveira, P. E. S. Barbosa, e C. L. Rodrigues. Algebraic specification-based testing: The veritas case study. *Simpósio Brasileiro de Métodos Formais — SBMF*, 2004.

[OMG03] OMG. Omg unified modeling language specification 1.5. PDF on-line: http://www.omg.org, Março 2003.

[RdFG03] C. L. Rodrigues, J. C. A. de Figueiredo, e D. D. S. Guerrero. Verificação de modelos em redes de petri orientadas a objetos. In *VII Workshop de Teses e Dissertações*, Manaus - Brasil, Outubro 2003.

[Rod04] C. L. Rodrigues. Verificação de modelos em redes de petri orientadas a objetos. Dissertação (Mestrado), COPIN - Universidade Federal de Campina Grande, 2004.

[San03] J. A. M. Santos. Suporte à análise e verificação de modelos rpoo. Dissertação (Mestrado), COPIN - Universidade Federal de Campina Grande, 2003.

[SW03] Donna C. Stidolph e James Whitehead. Managerial issues for the consideration and use of formal methdos. *FME 2003: Formal Methods, International Symposium of Formal Methods Europe*, 2003.

[Wor99] John B. Wordsworth. Getting the best from formal methods. In *Information and Software Technology*, volume 41, pages 1027–1032. 1999.

Apêndice A

Manual JMobile

A.1 Como usar?

Para se utilizar o *framework JMobile* precisamos ter o arquivo *jmobile.jar* que contém todos os binários e fontes do *framework*.

Com o arquivo jmobile.jar da API no diretório onde estarão os modelos ou no *classpath* podemos iniciar o processo de modelagem e construção das classes componentes do modelo que usaram e extenderam a API JMobile.

A.2 Como modelar?

Os modelos JMobile são compostos de artefatos de modelagem da orientação a objetos e artefatos de modelagem das redes de Petri.

A.2.1 Artefatos do modelo

O primeiro artefato a ser construído é o diagrama de classes do sistema. Como exemplo usaremos o problema clássico do jantar do Filósofos. Neste problema clássico temos ao redor de uma mesa cinco filósofos e cada filósofo tem um garfo que é compartilhado com seu vizinho esquerdo.

Na Figura A.1, os filósofos (f1, f2, f3, f4, f5) são representados por círculos de cor branca na parte mais externa da figura. Já os garfos (g1, g2, g3, g4, g5) são representados

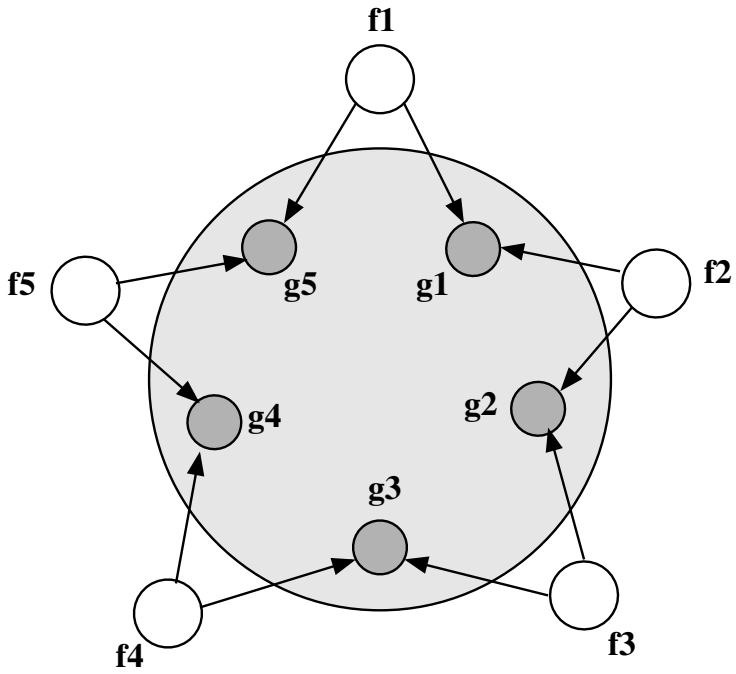


Figura A.1: Mesa com 5 filósofos e 5 garfos

por círculos de cor cinza e estão na parte mais interna da figura, em cima da mesa. Cada filósofo tem dois estados possíveis, podendo estar em apenas um deles em cada instante, ou seja, ou está pensando ou está comendo. Assim como o filósofo, um garfo também pode estar em um dois estados: ou ele está livre ou em uso. Os arcos representam que os filósofos podem pegar e soltar os respectivos garfos indicados. Para passar do estado pensando para o estado comendo, o filósofo precisa pegar o seu garfo e o garfo do vizinho da direita, desde que eles estejam livres. Ao passar do estado comendo para o estado pensando, o filósofo libera os dois garfos.

A proposta do formalismo RPOO é usar os artefatos convencionais da Orientação a Objeto para modelar as características externas do sistema. Desta forma, como também para a construção de modelos na Orientação a Objetos, um dos conceitos mais importantes é o conceito de *classe*. No problema dos filósofos, temos duas classes de objetos: a classe *Filósofo* e a classe *Garfo*. O filósofo pode realizar as ações *"pensar"* e *"comer"*, já os garfos podem ser *"pegos"* ou *"largados"* por um dos filósofos. Na Figura A.2, apresentamos o diagrama de classes UML [OMG03] para o problema do jantar dos Filósofos.

Como o diagrama de classes definido, modelaremos agora as redes de Petri que representem o comportamento de cada uma das classes. As redes de Petri usadas no JMobile são

Figura A.2: Diagrama de Classes para o Problema dos Filósofos

modificadas através de inscrições nas transições que representam ações de interação entre instâncias de objetos das classes do modelo.

Existem sete tipos diferentes de inscrições, como detalhado na Tabela A.1:

Ação	**Notação**	**Exemplo**
Interna	[this:]#	cf:#
Criação	[this:]varRPOO = new ClasseRPOO(nome);	cf:ge = new Garfo(“g1”);
Saída Assíncrona	[this:]varDestino.nomeMSG(varParametro);	f1:ge.pegar(); ou cf:fe.link(ge);
Saída Síncrona	[this:]varDestino!nomeMSG(varParametro)	f1:ge!pegar(); ou cf:fe!link(ge);
Entrada de Dados	[this:]varOrigem?nomeMSG(varParametro)	g1:fil?pegar(); ou f1:cons?link(ge);
Desligamento	[this:]unlink(varObjetoRPOO)	cf:unlink(ge);
Auto-destruição	[this:]End()	cf:End()

Tabela A.1: Tipos de ações elementares com efeito sobre uma configuração

As Figuras A.4 e A.3 apresentam as redes de Petri orientadas a objeto que descrevem os comportamentos das classes *Filósofo* e *Garfo* do problema do jantar dos filósofos. No modelo da classe *Filósofo*, os dois lugares modelam os dois possíveis estados de um filósofo. Temos a mesma situação para a classe *Garfo*.

Os lugares (*places*) em JMobile são todos do tipo *MobilePlace* e podem conter qualquer tipo elementar (tipos que extendem a classe *MobileValue*) definido na API ou definidos pelo modelador.

O nó de declaração (***SigmaLocal*** contém a definição dos tipos de fichas e variáveis que são utilizadas no modelo. As duas transições modelam as duas possíveis ações que podem ser executadas por um filósofo. A transição ***comer*** possui duas inscrições de saída síncrona

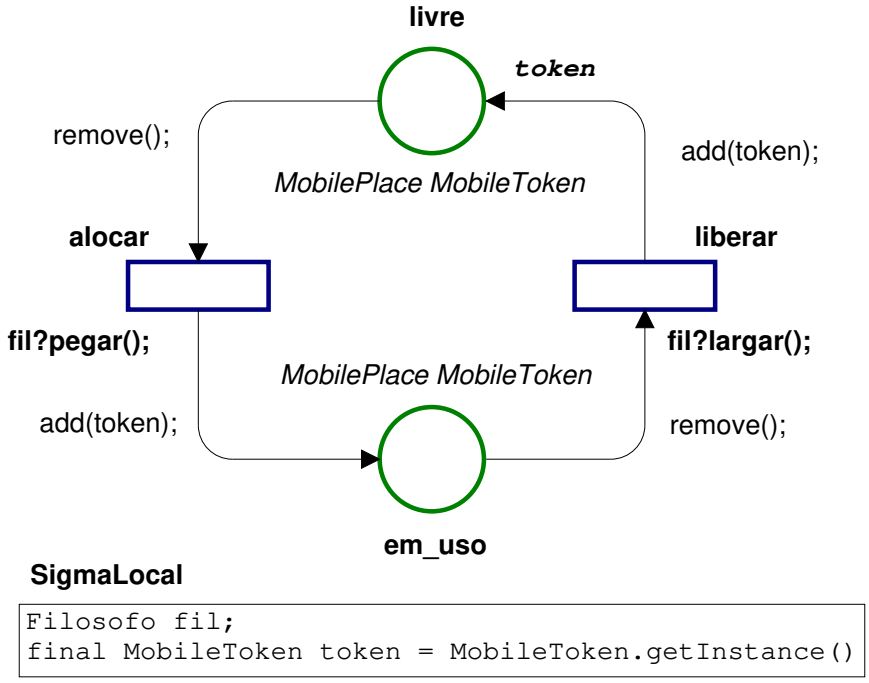


Figura A.3: Rede para o comportamento dos Garfos

ge!pegar(); e ***gd!pegar()***. De acordo com o nó de declaração, ***ge*** e ***gd*** são objetos do tipo Garfo. As duas inscrições de saída síncrona indicam que a esta transição deve disparar de forma síncrona com a transição ***pegar*** de duas instâncias de objetos (***ge*** e ***gd***) da classe Garfo. A transição ***pensar*** possui duas inscrições de saída assíncrona. No modelo da classe *Garfo*, cada uma das transições tem uma inscrição de entrada de dados (***fil?pegar()*** e ***fil?largar()***).

Para completar a modelagem RPOO, é necessário definir um sistema de objetos. Em um *Sistema de Objetos* (SO) temos a representação de todos os objetos do sistema, ligações entre eles e eventuais mensagens pendentes representando um estado estrutural do sistema. O Sistema de Objetos representa o estado do sistema em nível de interação externa entre os objetos, ficando a parte interna do comportamento de cada objeto encapsulada nas suas respectivas redes. O termo *Estrutura* será utilizado para denotar um estado do sistema em nível de objeto. O termo *Configuração* será utilizado para denotar um estado do sistema em RPOO, ou seja, estamos interessados na estrutura do SO e no estado interno dos objetos.

Na Figura A.5 temos a representação gráfica para a estrutura inicial do sistema de objetos para um modelo de solução do problema dos filósofos. Os arcos que partem de ***f1*** para ***g1*** e ***g5*** indicam que o objeto da classe ***f1*** conhece ou tem ligações com os objetos ***g1*** e ***g5***. Os objetos ***g1***, ***g2***, ***g3***, ***g4*** e ***g5*** não têm ligação com outros objetos.

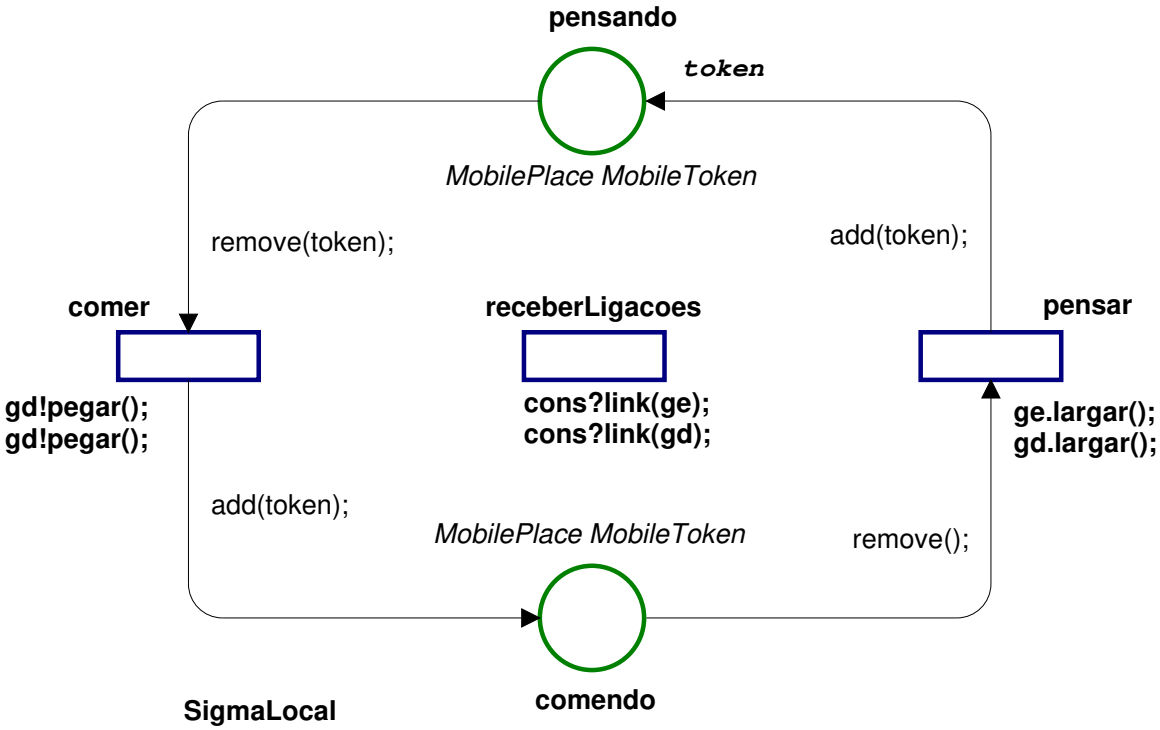


Figura A.4: Rede para o comportamento dos Filósofos

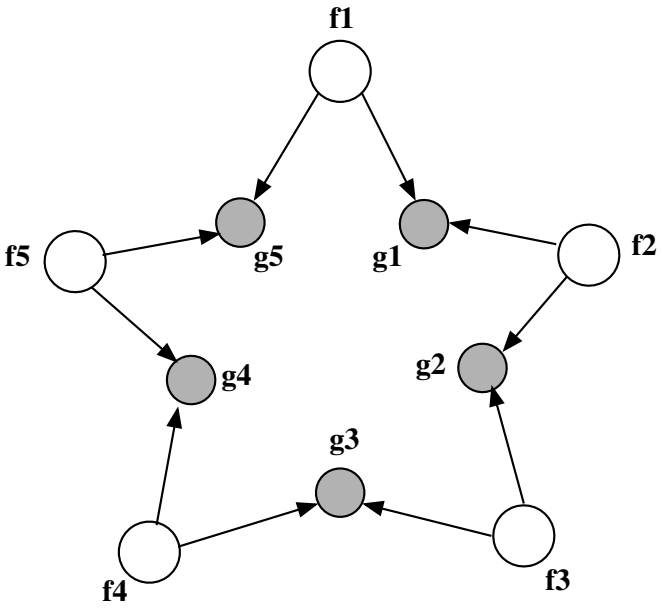


Figura A.5: Estrutura Inicial do Sistema de Objetos dos Filósofos

Uma outra forma de apresentar a estrutura do sistema de objetos é utilizar uma representação algébrica. A representação algébrica para a estrutura da Figura A.5.

```
Estrutura = f1[g5 g1] + f2[g1 g2] + f3[g2 g3] +
            f4[g3 g4] + f5[g4 g5] + g1 + g2 + g3 + g4 + g5
```

Na representação algébrica, quando escrevemos *f1[g5 g1]*, estamos informando que o objeto ***f1*** da classe *Filósofo* conhece ou tem ligações com os objetos ***g5*** e ***g1*** da classe *Garfo*. Quando escrevemos apenas ***g1***, estamos informando que o objeto não tem ligações para outros objetos. O sinal de adição (+) serve apenas como delimitador entre os objetos e indicam que fazem parte da mesma estrutura.

A Figura A.6 apresenta a configuração inicial do modelo RPOO, ou seja, a estrutura com estados internos. Cada um dos 5 filósofos está no estado pensando (rótulo ***p***) e cada garfo está livre (rótulo ***l***).

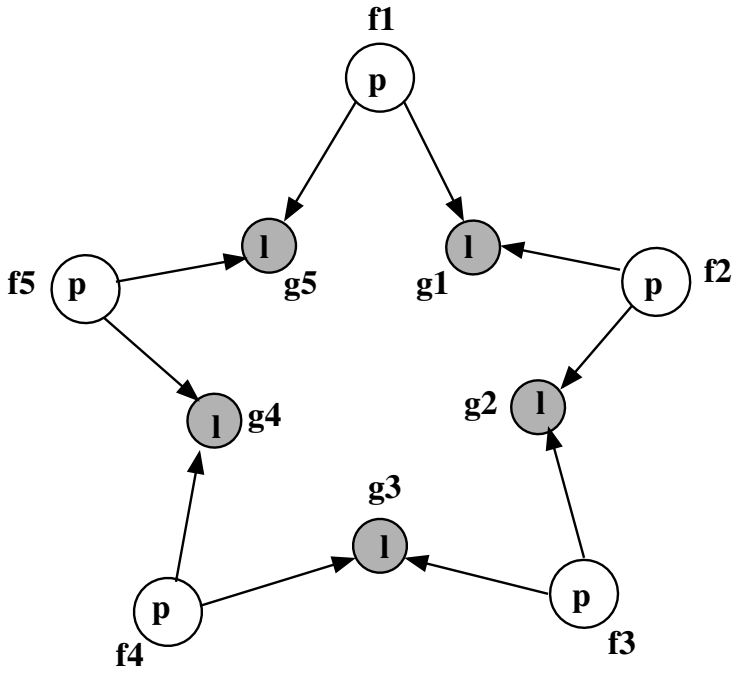


Figura A.6: Configuração inicial do modelo dos Filósofos

A representação algébrica para a configuração é dada por:

```
Configuração = f1(pensando{token})[g5 g1] + f2(pensando{token})[g1 g2] +
       f3(pensando{token})[g2 g3] + f4(pensando{token})[g3 g4] +
       f5(pensando{token})[g4 g5] + g1(livre{token}) + g2(livre{token}) +
       g3(livre{token}) + g4(livre{token}) + g5(livre{token})
```

Nesta representação algébricam vemos que no lugar *pensando* dos filósofos temos uma ficha tipo *token* e no lugar *livre* dos garfos temos também um *token*. Desta forma, representamos o estado inicial da configuração onde os filósofos estão pensando e os garfos estão livres.

A.2.2 Modelo em XML e Java

Como os artefatos de modelagem definidos, como foi apresentado na subseção anterior, escreveremos arquivos XML e arquivos Java. Os arquivos XML e Java, são arquivos que representam o modelo JMobile de forma textual e que serviram de entrada para as ferramentas da API JMobile.

Para cada classe (rede de Petri) do modelo escrevemos um arquivo XML que descreve a classe. A partir do XML a API JMobile criar arquivos .java, com a representação em código java das redes de Petri Orientadas a Objetos. Contudo, no estado atual de desenvolvimento da API esta tradução ainda depende da construção de um Editor para as redes de Petri. Desta forma, escrevemos diretamente os arquivos .java, mas em versões seguintes do JMobile este processo será automático e usará como entrada os artefatos de modelagem.

Assim, teremos dois arquivos .java, uma para a rede Filósofo e outro para a rede Garfo. Na API as classes do modelo são classes que extendem a classe *AbstractMobileObject*, pois definem o comportamento de objetos RPOO. A classe *AbstractMobileObject* implementa a interface *MobileObject*. A interface *MobileObject* define todos os métodos utilizados pela API para extrair informações sobre o estado e sobre as ações de cada objeto RPOO, ou seja, de cada mobileObject.

Com as classes Java Filósofo e Garfo para os MobileObject's presentes no modelo dos Filósofos, iremos construir uma instância da classe *Configuration* e uma da classe *JMobileModel*. A instância da classe *Configuration* representa a configuração definida na Figura A.6, só que neste caso construimos uma configuração só com dois Filósofos e dois Garfos. Na Figura A.7 temos a classe que representa um modelo JMobile.

A classe Java *MainF2*, na Figura A.8, traz a implementação do processo de construção para o modelo dos Filósofos.

A Figura A.9 mostra a saída da execução desta classe executável em um terminal *"shell"* no sistema operacional *"GNU/Linux"*.

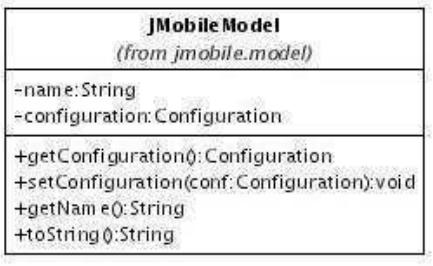


Figura A.7: Classe *JMobileModel* do *package jmobile.model*

```
01: import jmobile.rpoo.*;
02:
03: public class MainF2 {
04:     public static void main(String[] args) {
05:
06:         //Construindo o Modelo
07:         System.out.println(``<-- Modelo Filósofos Modificados - 2 filósofos -->\n'');
08:         JMobileModel f2model = new JMobileModel(``F2'');
09:
10:         //Construindo a Configuração ``Vazia''
11:         Configuration confInicial = new Configuration();
12:
13:         //Construindo os objetos RPOO das Classes ``Filosofo'' e ``Garfos''
14:         Filosofo f1 = new Filosofo(``f1'');
15:         Filosofo f2 = new Filosofo(``f2'');
16:         Garfo g1 = new Garfo(``g1'');
17:         Garfo g2 = new Garfo(``g2'');
18:
19:         //Adicionando os objetos a configuração
20:         confInicial.add(f1);
21:         confInicial.add(f2);
22:         confInicial.add(g1);
23:         confInicial.add(g2);
24:         //Adicionando as ligações entre os objetos
25:         confInicial.addLink(``f1'',``g1'');
26:         confInicial.addLink(``f1'',``g2'');
27:         confInicial.addLink(``f2'',``g1'');
28:         confInicial.addLink(``f2'',``g2'');
29:
30:         //Adicionando a configuração inicial ao modelo
31:         f2model.setConfiguration(confInicial);
32:     }
33: }
```

Figura A.8: Código da classe MainF2

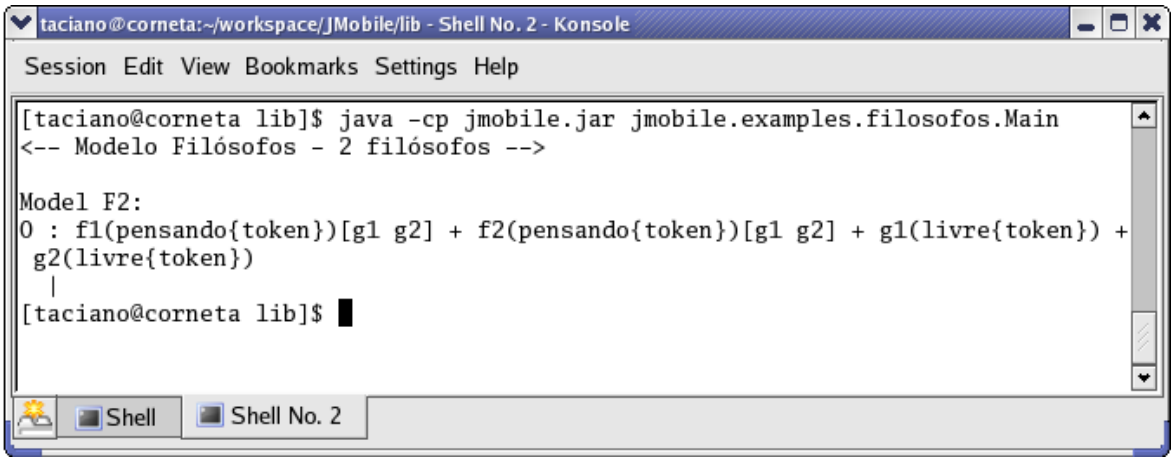


Figura A.9: Executando a classe *MainF2*

A.3 Como simular?

Nesta seção, apresentaremos algumas funcionalidades da API JMobile e como utiliza-lá na simulação de modelos RPOO. Para isto, usaremos o modelo dos filósofos construído na Seção 4.3.

A.3.1 Funcionalidades da API JMobile para a Simulação

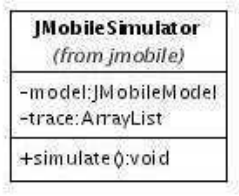


Figura A.10: Classe *JMobileSimulator* do *package jmobile*

Para simularmos um modelo RPOO usando a API JMobile precisamos construir um *JMobileModel* como mostrado na Seção 4.3. Como o modelo construído, nós instanciamos da classe *JMobileSimulator*, que é um protótipo de simulador para RPOO, que faz uso das funcionalidades da API para simulação. O simulador é uma classe que fornece um *prompt* de comando através do método *simulate()*. Nas Figuras A.10 e A.11, apresentamos a classe *JMobileSimulator* e o seu método *simulate()*.

```
01: public void simulate() {
02:     int nEvents = 0;
03:     int cid = 0;
04:     while (true) {
05:         cid=trace.size()-1;
06:         JMobileTransitionsEvent[] mtEvents;
07:         System.out.println("-----------------------------------------------");
08:         Configuration c = (Configuration)trace.get(cid);
09:         System.out.println(c);
10:         System.out.println("\n<<< Eventos Conf "+cid+" >>>\n");
11:         mtEvents = c.getMobileTransitionsEvent();
12:         for (int i = 0; i < mtEvents.length; i++) {
13:             System.out.println(i + ": " + mtEvents[i]);
14:         }
15:         System.out.print("\nDisparar evento n.º? ");
16:         nEvents = Integer.parseInt(
17:             (new BufferedReader(new InputStreamReader(System.in))).readLine()
18:         );
19:         System.out.println();
20:         if (mtEvents.length >= 0 && nEvents < mtEvents.length && nEvents >= 0) {
21:             Configuration confTemp = c.executeNew(mtEvents[nEvents]);
22:             trace.add(confTemp);
23:             System.out.println("-----------------------------------------------");
24:         } else {
25:             System.err.println("\n Não existe evento para o número digitado.");
26:         }
27:     }
28: }
```

Figura A.11: Método *simulate()* da Classe *JMobileSimulator* do *package jmobile*

A.3.2 Simulando modelos

Para simularmos os modelos Filósofos e Filósofos modificados precisaremos acrescentar as seguintes linhas nos códigos das classes executáveis *MainF2* e *MainFM2*.

JMobileSimulator sim = new JMobileSimulator(f2model);

sim.simulate();

Ao iniciarmos a execução desta classe obtemos o *prompt* mostrado na Figura A.12. O prompt imprime a representação algébrica da um configuração com o estado interno dos objetos RPOO. Em seguida, logo depois do texto $<<<$ ***Eventos Conf 0*** $>>>$ são apresentados os eventos possíveis para a esta configuração. No processo de cálculo dos eventos, cada variável presente no modelo pode assumir um dos valores possíveis para ela. No caso

de variáveis que referenciam objetos RPOO, ou seja, ligações, a ferramenta pode variar as atribuições. Por exemplo, os eventos *"0"* e *"1"* e os eventos *"2"* e *"3"* são praticamente iguais. Estes eventos geram os mesmos eventos RPOO como podemos ver logo depois do símbolo ===>>, a única diferença é a ordem de atribuição das variáveis.

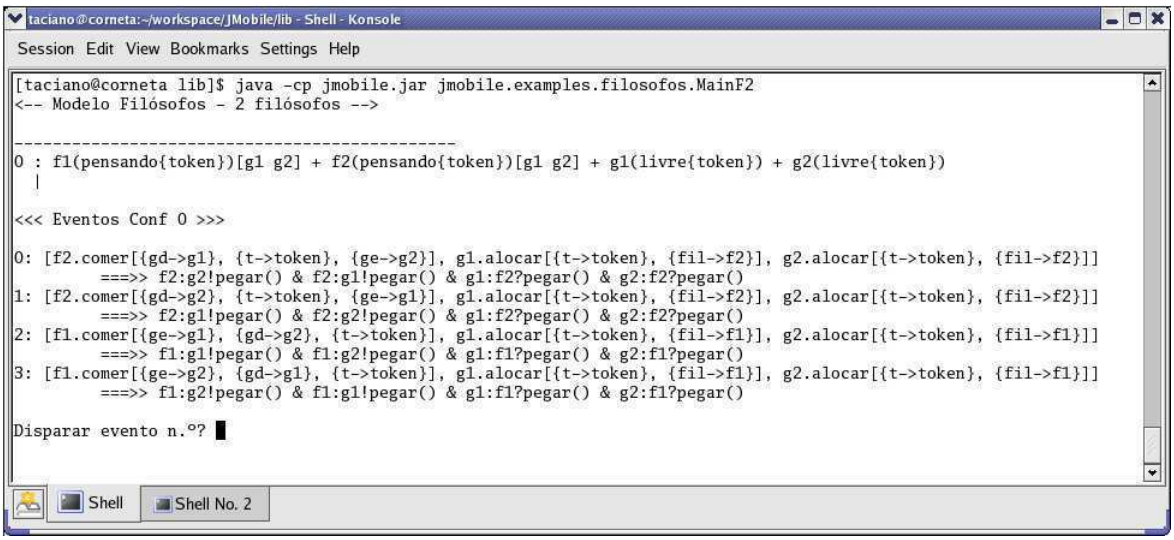


Figura A.12: Executando a classe *MainF2* com a criação de um simulador

Logo abaixo dos eventos, o *prompt* imprime o texto ***Disparar Evento n.º?*** , onde podemos digitar o número de um dos quatro eventos possíveis. O simulador guarda as configurações alcançadas em um trace de execução e desta forma o engenheiro de software poderá analisar os traces para a identificação de problemas.

A.4 Como gerar o espaço de estados?

Apresentaremos nesta seção o gerador de espaço de estados e sua utilização, para isso vamos utilizar o modelo do filósofo e gerar o espaço de estados para várias instâncias do modelo. Será utilizado para a geração do espaço de estados um computador com processador com 1.8Ghz e 1Gb de memória RAM. Todos os espaços de estados foram gerados no mesmo computador e cronometramos o tempo para a geração.

Para gerarmos o espaço de estado acrescentaremos as seguintes linhas de código na Figura A.13 a classe *MainF2* na Página 72:

O código na Figura A.13 gera o espaço de estados do modelo em três formatos em ar-

```
34: JMobileSSG ssg = new JMobileSSG(confInicial, true);
35: JMobileStatesSpace og = ssg.generate
36: (
37:    new PrintStream(
38:       new  FileOutputStream("/home/taciano/workspace/JMobile/filosofo2.eee")
39:    ),
40:    new PrintStream(
41:       new FileOutputStream("/home/taciano/workspace/JMobile/filosofo2.aut")
42:    )
43: );
44:
45: System.setOut(
46:    new PrintStream(
47:       new FileOutputStream("/home/taciano/workspace/JMobile/filosofo2.veritas")
48:    )
49: );
50: System.out.print(og.toStringVeritas());
```

Figura A.13: Executando o gerador de espaço de estados

quivos separados. Na linha 34 instanciamos o gerador, classe *JMobileSSG*, passando a configuração inicial do modelo e um booleano informando se queremos um espaço de estados ordenado ou não pelo identificador dos estados. Dá linha 35 à linha 43, temos a chamada ao método *"ssg.generate"* passando como parâmetro dois *PrintStream's*(fluxo de saída para gravação em arquivo) para os arquivos "filosofo2.eee" e "filosofo2.aut". O arquivo "filosofo2.eee" contém o espaço de estados padrão, onde a representação textual traz as configurações alcançáveis. Já o arquivo "filosofo2.aut" contém o espaço de estados no formato aldebaran. Estes dois formatos são gerados em tempo de execução do gerador, ou seja, a cada nova configuração alcançada é gravada uma linha nos arquivos para esta configuração.

Para gerar o espaço de estado no formato Veritas, é preciso gerar todo o espaço de estados na estrutura de dados da classe *JMobileStatesSpace*. Isto é necessário porque o formato Veritas necessita de informações de estados predecessores e sucessores para fazer a verificação e para obtermos estas informações necessitamos do espaço de estados completo.

A.5 Diagrama de Classes Detalhado

Nesta seção temos as figuras detalhadas dos *packages* que compoem a API JMobile.

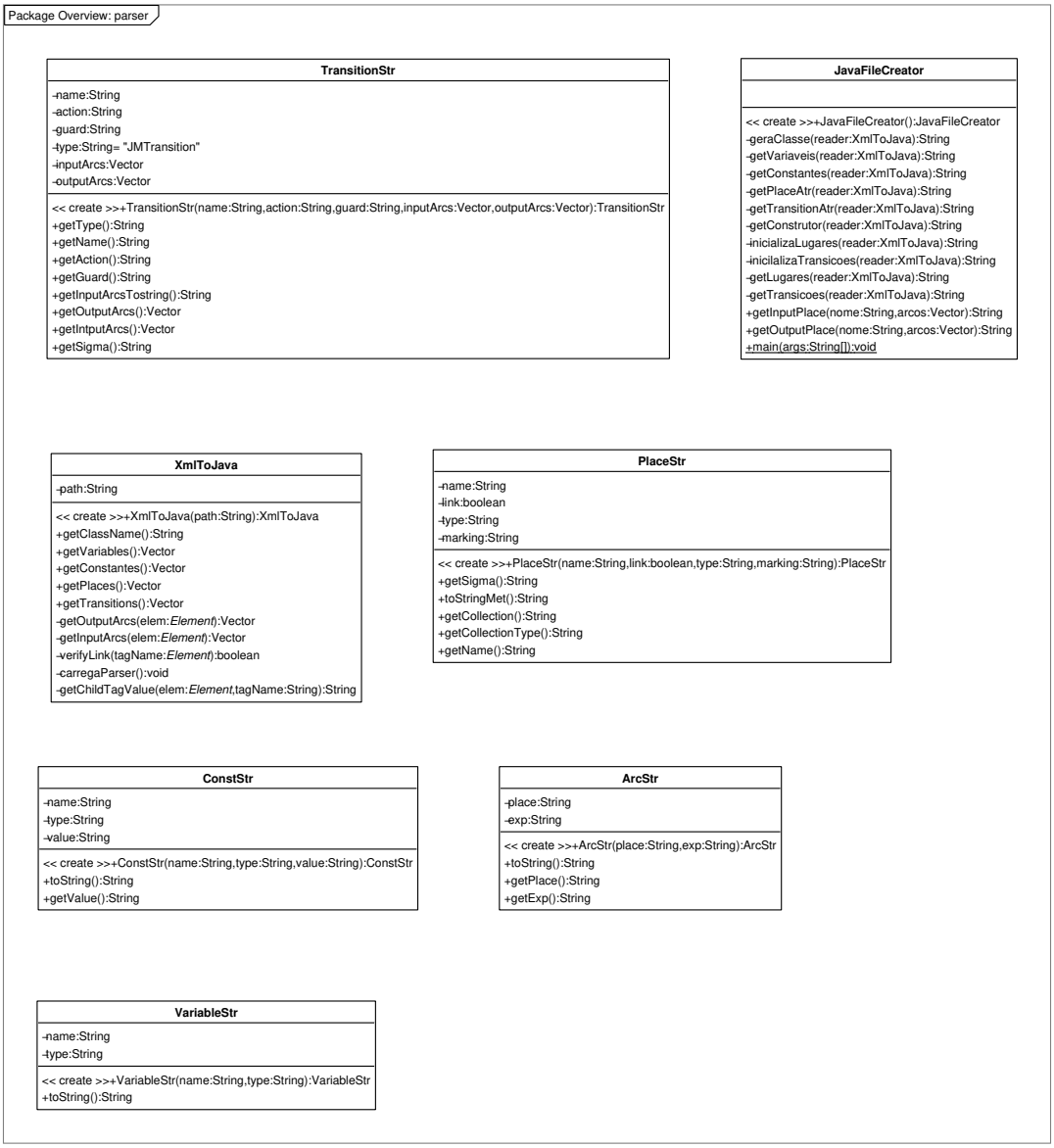


Figura A.14: Classes do Package Parser

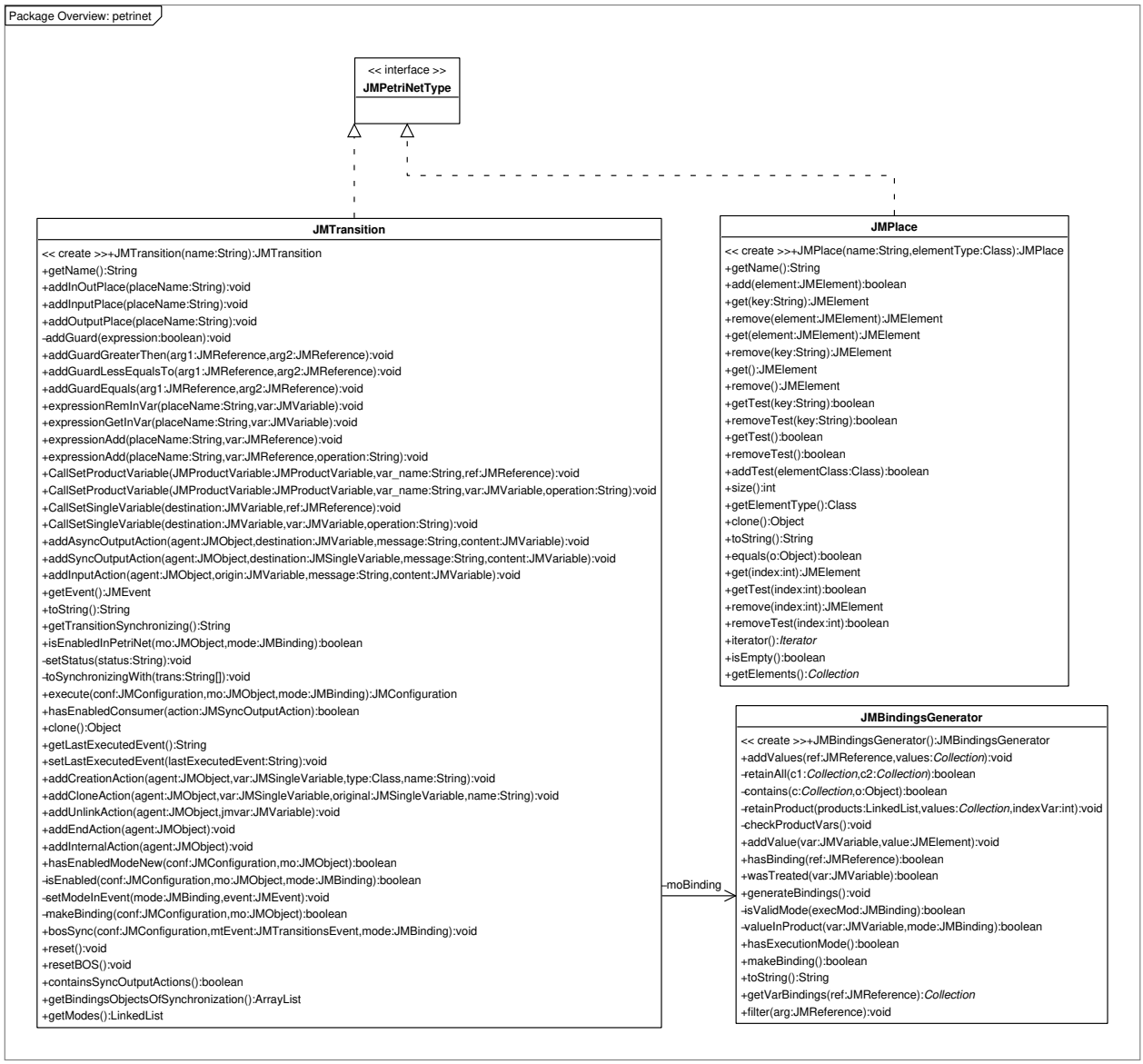


Figura A.15: Classes do Package PetriNet - Parte 1

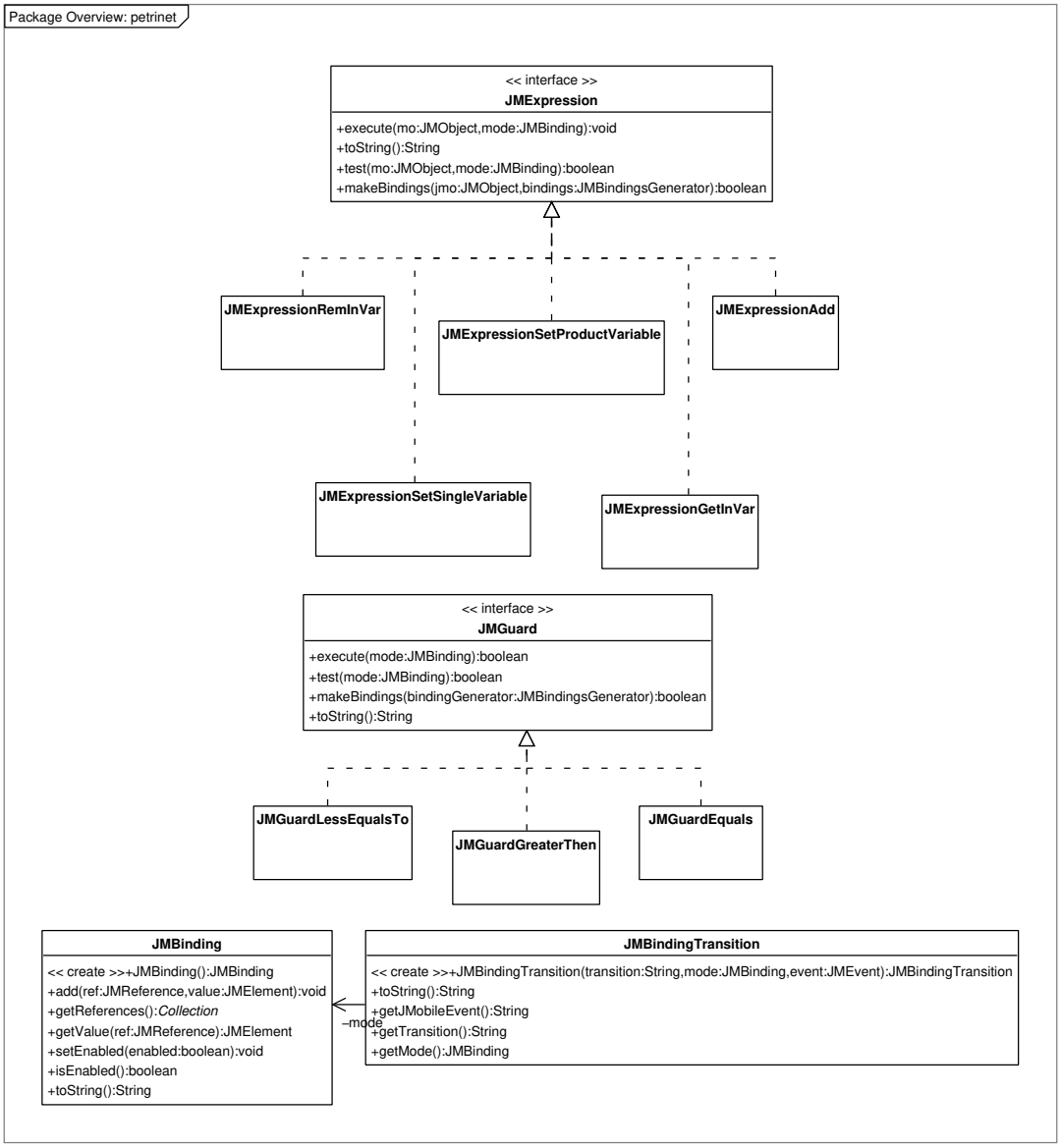


Figura A.16: Classes do Package PetriNet - Parte 2

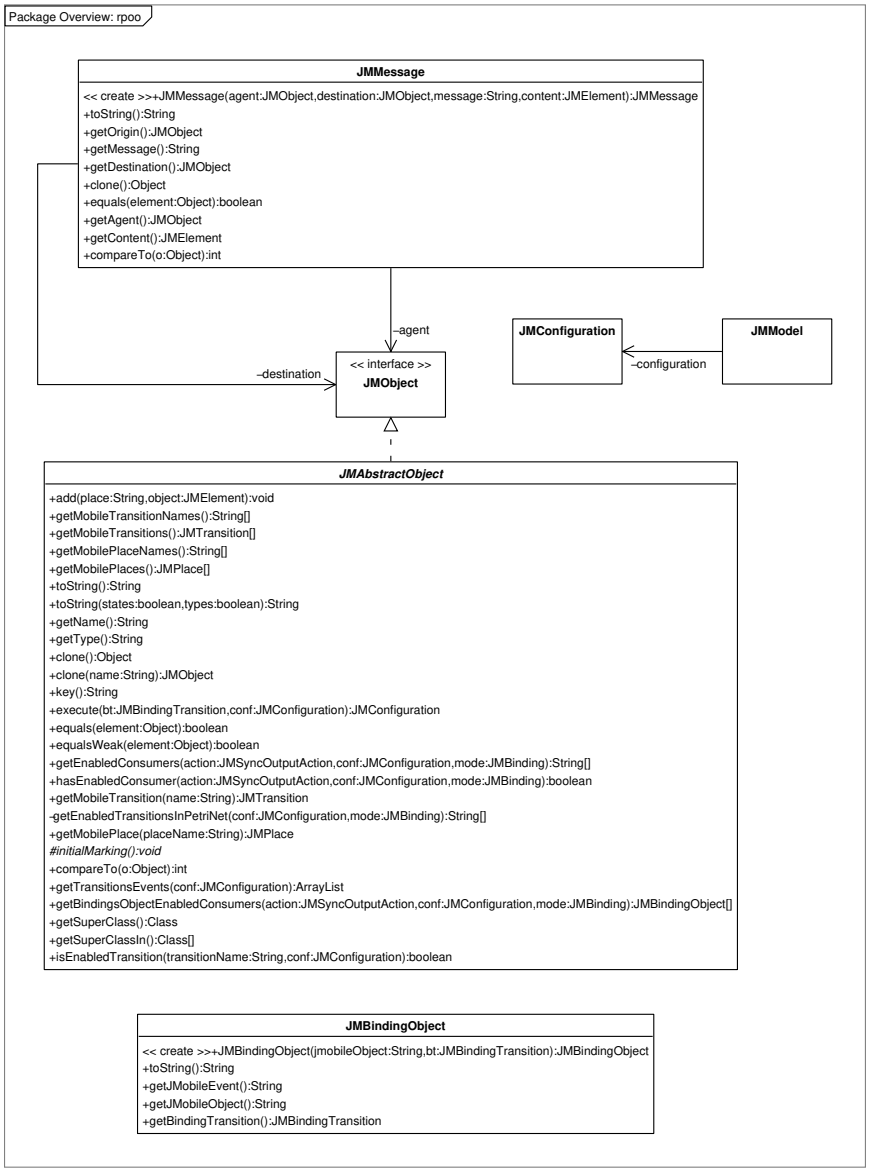


Figura A.17: Classes do Package RPOO

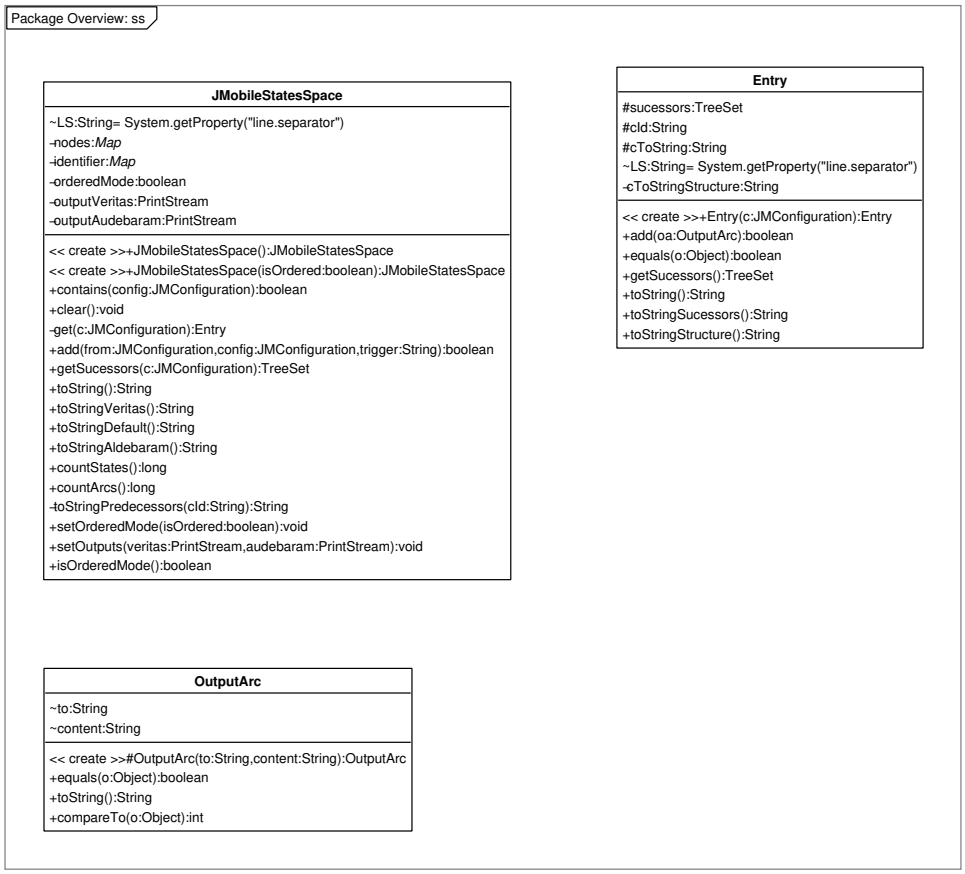


Figura A.18: Classes do Package SS

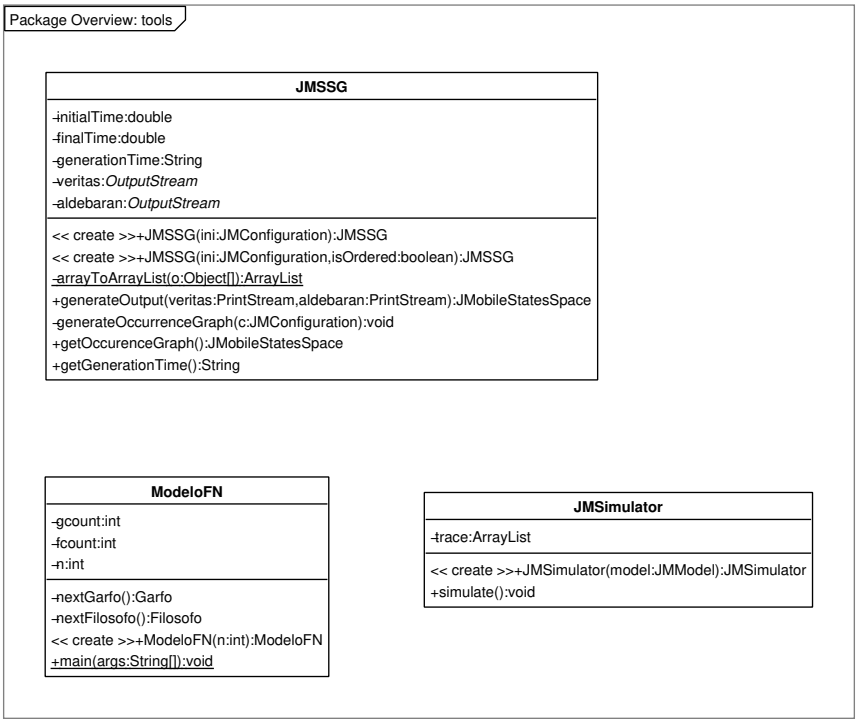


Figura A.19: Classes do Package Tools

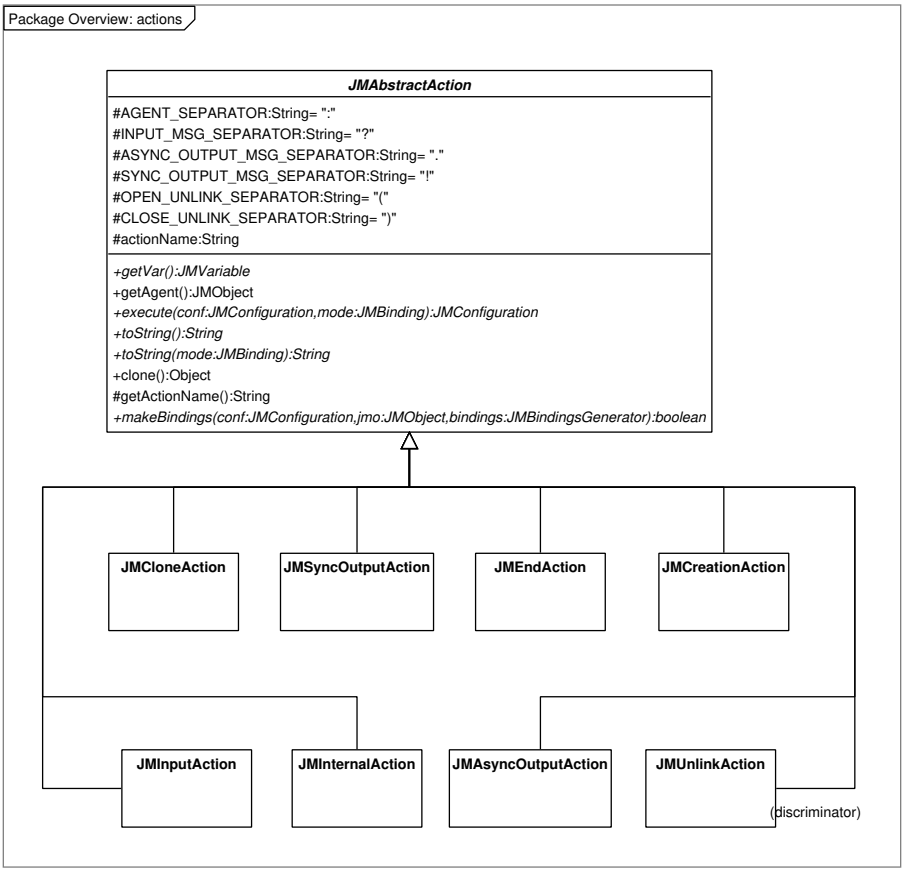


Figura A.20: Classes do Package Action

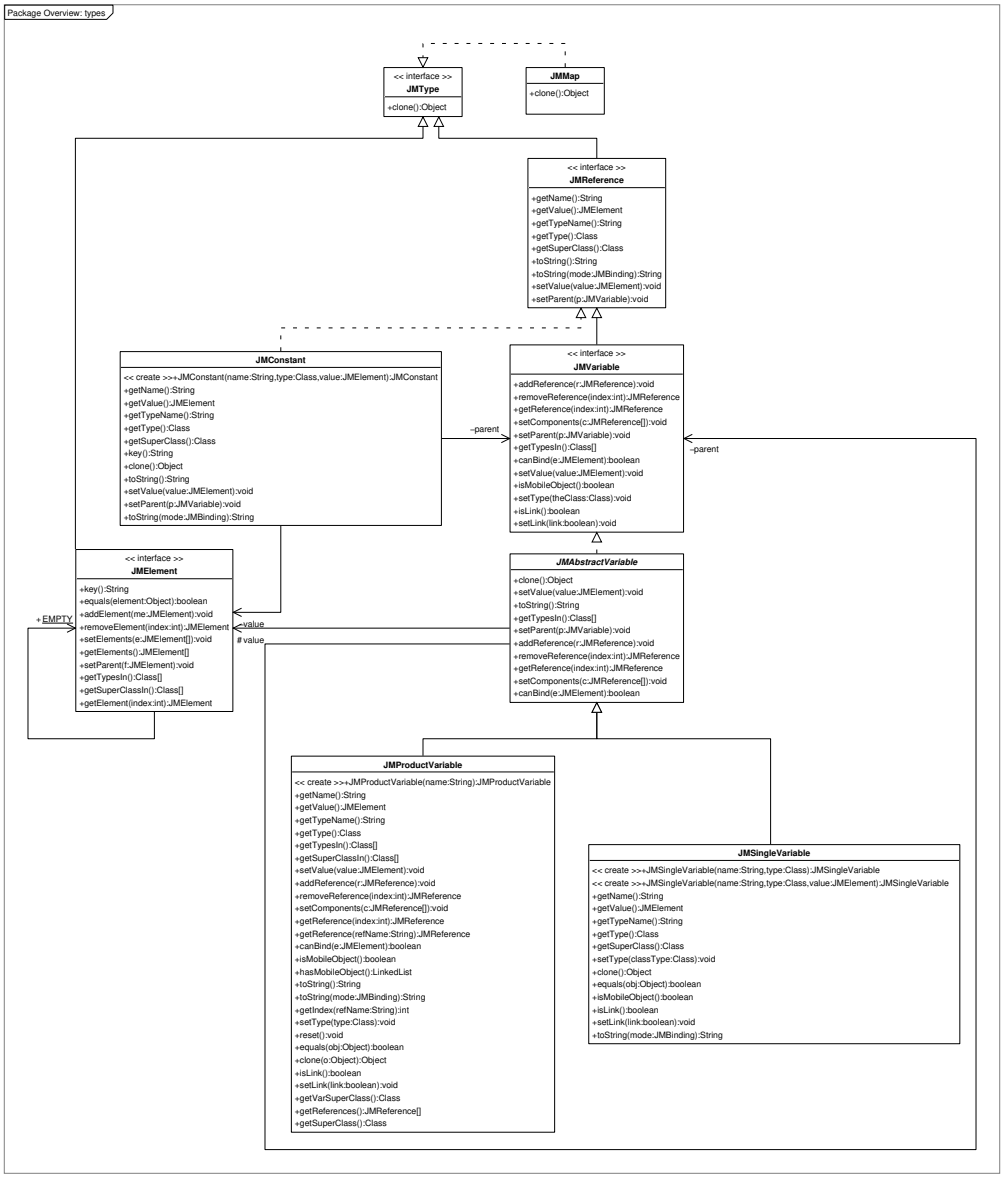


Figura A.21: Classes do Package Types - Parte 1

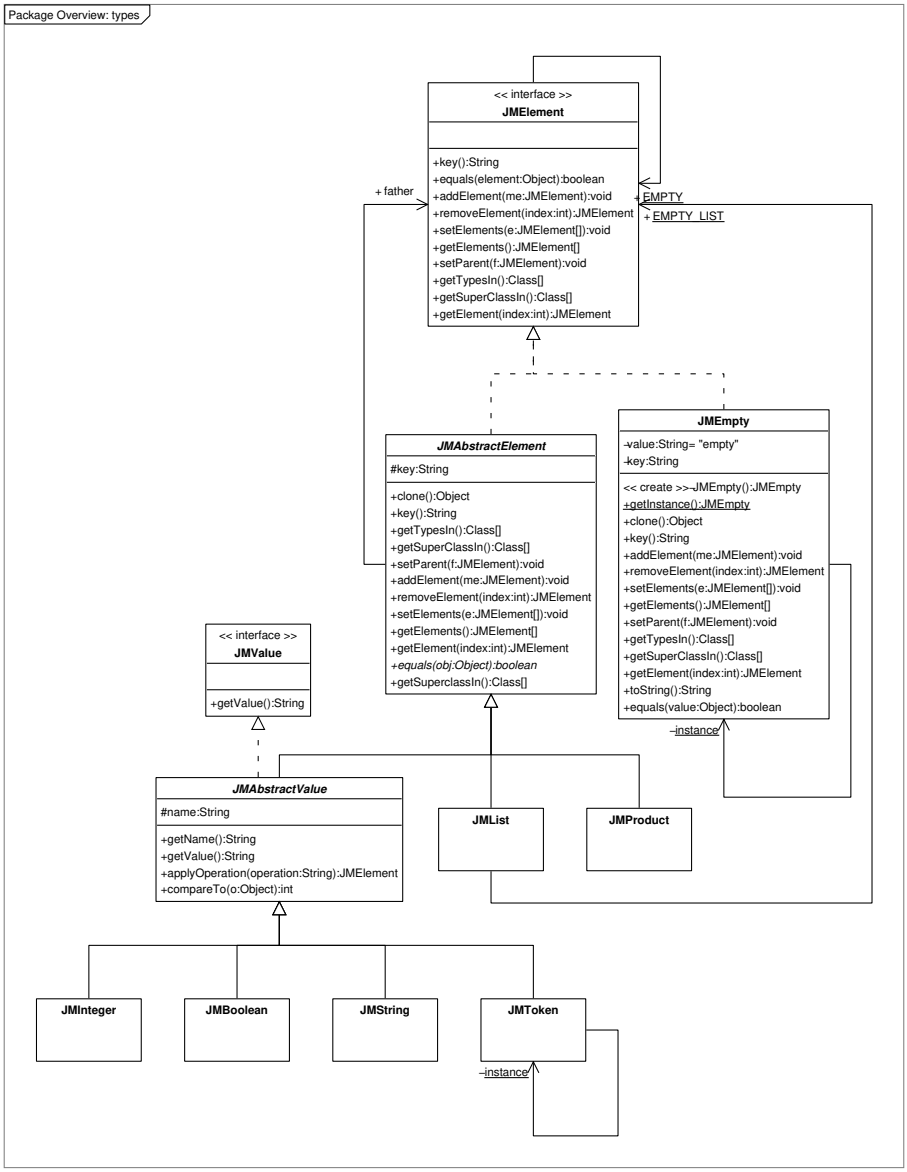


Figura A.22: Classes do Package Types - Parte 2

Apêndice B

Modelagem do Sistema Conferência

Este capítulo tem como objetivo mostrar uma aplicação da API JMobile apresentada no Capítulo 4 sobre um determinado sistema. O sistema escolhido para o estudo de caso foi um sistema de maior porte do que os sistemas utilizados até então. Além disso, este sistema apresenta características de concorrência e mobilidade.

A aplicação selecionada para realizar o estudo de caso foi desenvolvida por Guedes e pode ser encontrada em sua dissertação de Mestrado [Gue02a]. Trata-se de um Sistema de Apoio às Atividades de Comitês de Programa em Conferências. A aplicação gerencia atividades de comitês de programas de conferências, tais como submissão de artigos, processo de avaliação e notificação de aceitação ou rejeição de artigos aos autores.

A especificação e o código fonte podem ser encontrados em [Gue02a]. A aplicação foi desenvolvida em Java sobre a plataforma Grasshopper. Diagramas de classe, de seqüência e de colaboração de UML foram utilizados por Guedes para a especificação do sistema. A partir destes diagramas, foram construídos os modelos RPOO em XML e em Java usando a API JMobile, que serviram de entrada para o simulador e o gerador de espaço de estados. O modelo JMobile para o sistema conferência foi construido em conjunto com Figueiredo [FMdF05] que utilizou o nosso *framework* em seu trabalho.

Na Seção B.1, iremos apresentar as entidades do sistema de conferência e descreveremos o seu comportamento. Na Seção B.2 iremos apresentar a partir do diagrama de classes as redes de Petri Orientadas a Objetos as principais entidades que compõe o sistema conferência. Em seguida, descrevemos o processo de geração do seus espaço de estados.

B.1 Sistema Conferência

Nesta seção, apresentaremos a descrição do sistema através de um diagrama de comportamento que descreve a forma com que os agentes móveis envolvidos no sistema irão migrar entre as agências e se comunicar com os atores externos e entre si. Esta descrição tem como fonte a descrição apresentada por Figueiredo, aqui apresentamos apenas um resumo onde foram retiradas partes específicas relacionadas aos trabalhos de Figueiredo.

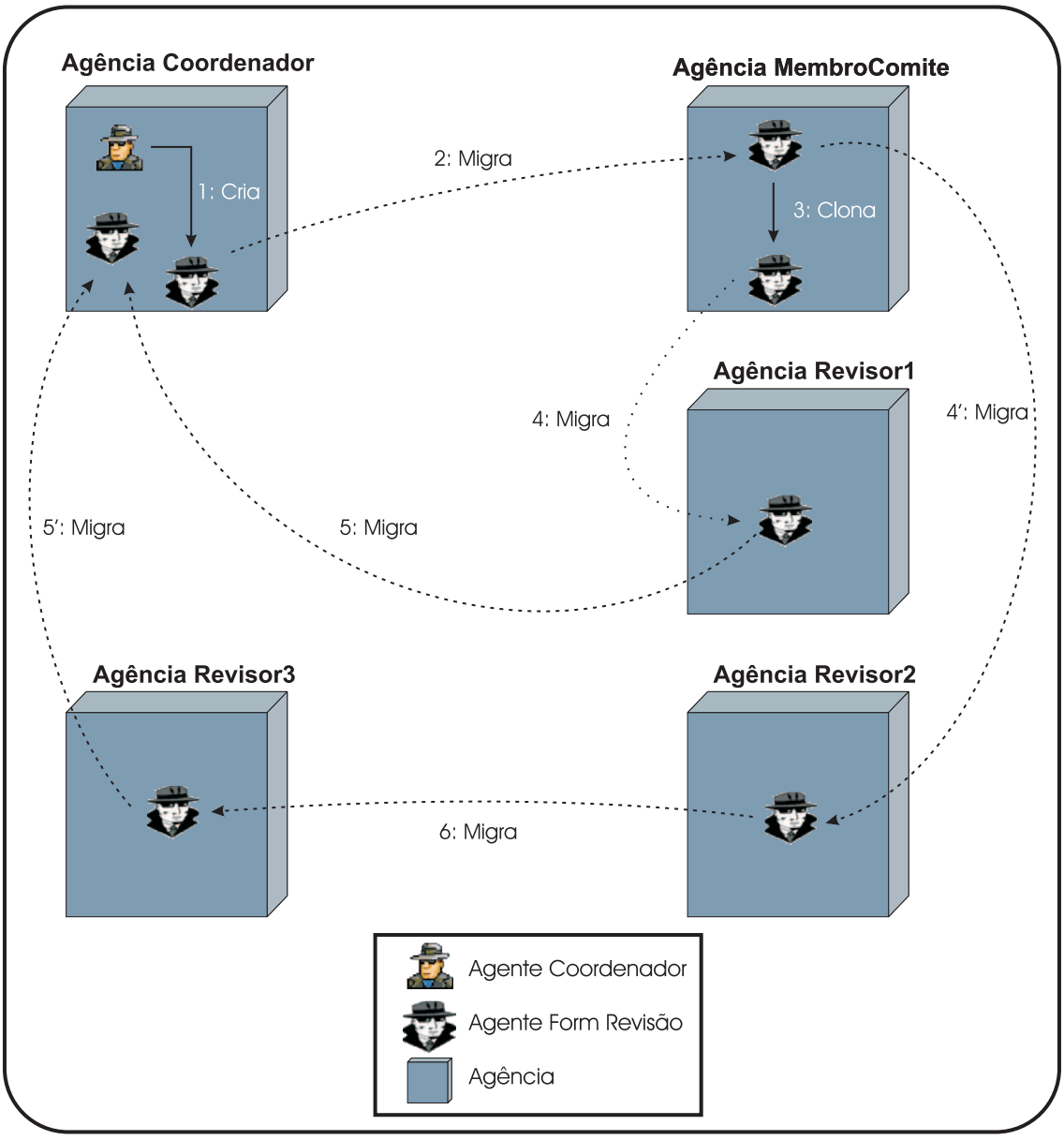


Figura B.1: Diagrama de Comportamento dos Agentes

A Figura B.1 [FMdF05] mostra o comportamento dos principais agentes do sistema para

um determinado cenário. Nela, vemos a presença de um *Agente Coordenador* (agente estacionário) na *Agência Coordenador*. Por meio de uma interface gráfica, provida pelo *Agente Coordenador*, o coordenador do programa seleciona um artigo a ser revisado, o membro do comitê responsável por encontrar revisores para este artigo e a quantidade de cópias deste formulário desejada. O *Agente Coordenador*, então, cria um *Agente Form Revisao* (agente móvel) contendo o artigo a ser revisado (1). O *Agente Form Revisao* migra para a máquina do membro do comitê (*Agência MembroComite*) (2). O *Agente Form Revisao* gera um clone para cada formulário solicitado pelo coordenador menos um (já que ele mesmo é um *Agente Form Revisao*) (3). O membro do comitê, através de suas interfaces gráficas, redireciona os agentes *Agente Form Revisao* existentes em sua máquina para outros revisores, informando se deseja que eles retornem quando tiverem sido revisados ou não. Cada *Agente Form Revisao* migra para a máquina de um revisor (4 e 4'). O revisor recebe o *Agente Form Revisao* e pode optar por revisar o artigo diretamente ou redirecioná-lo para um outro revisor (6), informando se deseja que ele retorne quando tiver sido revisado. Quando o processo de revisão é finalizado o *Agente Form Revisao* retorna diretamente para a máquina do coordenador de programa caso nenhum dos remetentes tenha solicitado o seu retorno (5 e 5'). Do contrário, o *Agente Form Revisao* volta para o último remetente que solicitou o seu retorno. Caso ainda exista um remetente anterior a este, que também tenha solicitado o seu retorno, o *Agente Form Revisao* retornará para ele. Após ter sido aprovado por todos os remetentes que solicitaram o seu retorno, o *Agente Form Revisao* volta para a máquina do coordenador de programa.

Na Figura B.2, apresentamos o diagrama de classes para o sistema de conferências.

B.2 Modelo RPOO para o Sistema Conferência

Nesta seção, apresentaremos uma parte do modelo RPOO para o Sistema Conferência. A parte principal do modelo é a que representa a classe *AgenteFormRevisao* e a partir dela vamos explicar o modelo RPOO. O modelo RPOO para o Sistema Conferência foi construido por Figueiredo [FMdF05] e em conjunto o colocamos na notação JMobile.

A Figura B.3 apresenta a página principal da rede de Petri da classe *AgenteFormRevisao*. A rede da classe *AgenteFormRevisao* é uma rede de Petri hierarquizada, pois devido ao

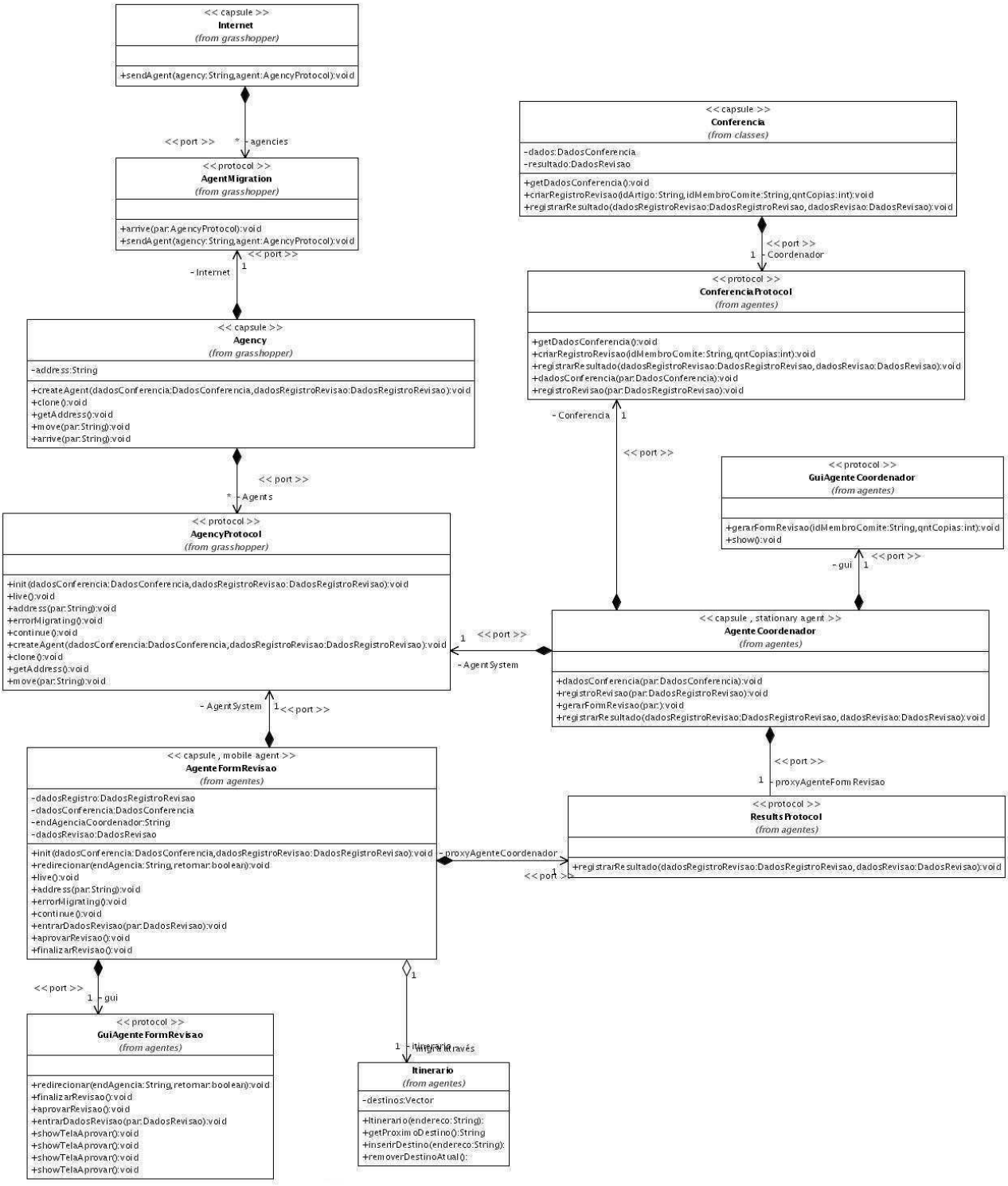


Figura B.2: Diagrama de Classes do Sistema de Conferências

seu tamanho e complexidade foi necessário acrescentarmos transições de substituição. Uma transição de substituição é uma transição cujo comportamento abstrai a execução de uma

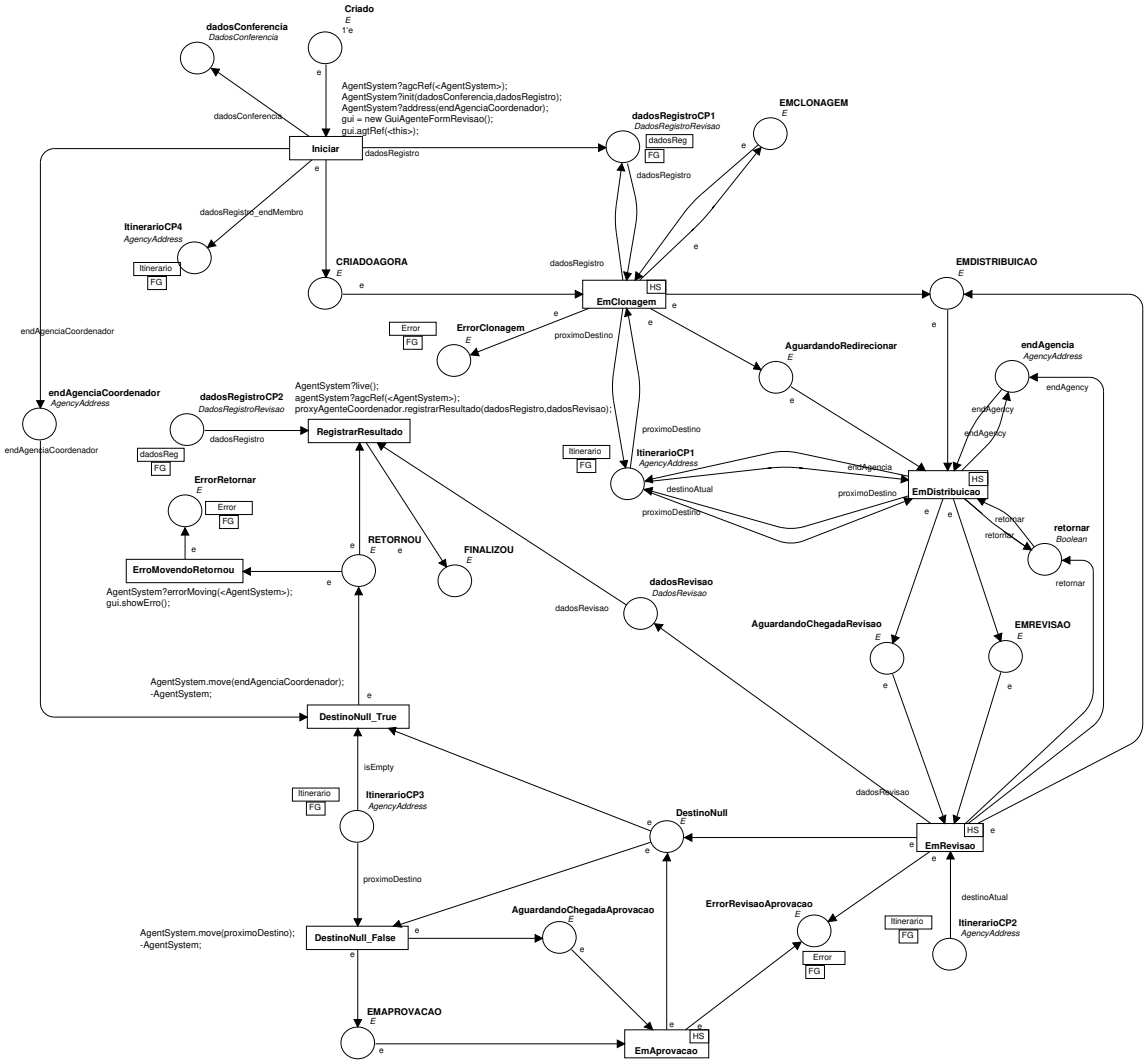


Figura B.3: Página principal da Rede de Petri que descreve a Classe *AgenteFormRevisao*

outra rede de Petri, de forma semelhante a um estado em um diagrama de estados que possui um outro diagrama de estados que detalhe o comportamento de um objeto neste estado. As transições de substituição são as transições "*EmClonagem*", "*EmDistribuicao*", "*EmRevisao*" e "*EmAprovacao*", e as suas relativas redes de Petri são as redes apresentadas nas Figuras B.4, B.5, B.6 e B.7.

No diagrama de classes apresentado na Figura B.2, podemos ver que a classe *Itinerario* está agregada à classe *AgenteFormRevisao*. Esta classe abstrai um lista de endereços de agências por onde o agente *AgenteFormRevisao* deverá migrar. Por ser um comportamento simples, sua modelagem em RPOO foi feita através de um lugar na rede de Petri da classe *AgenteFormRevisao* chamado "*Itinerario*", e cuja tipo é *AgencyAddress*. Desta forma, os

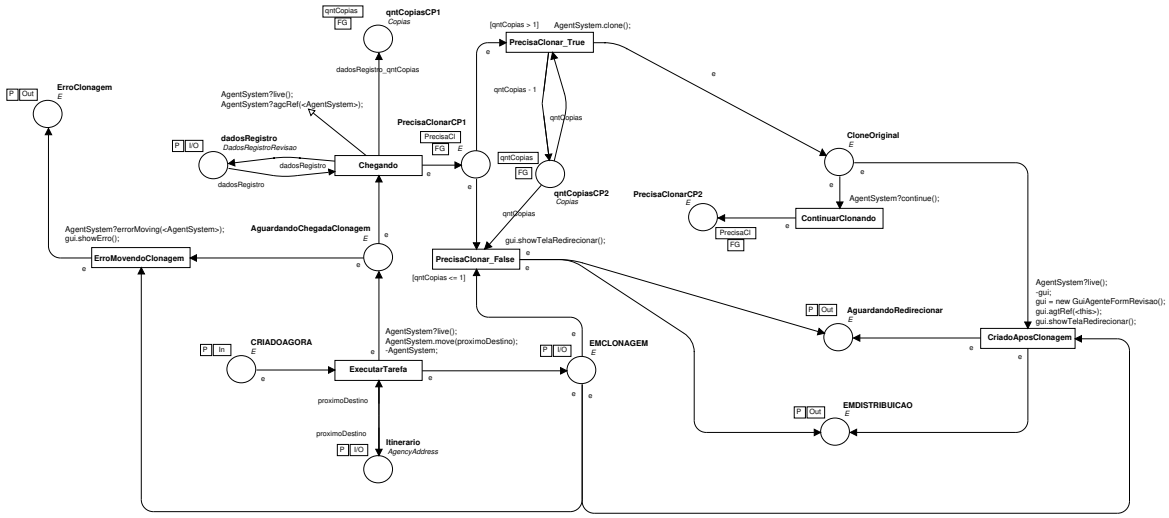


Figura B.4: Página *EmClonagem* da Rede de Petri que descreve a Classe *AgenteFormRevisao*

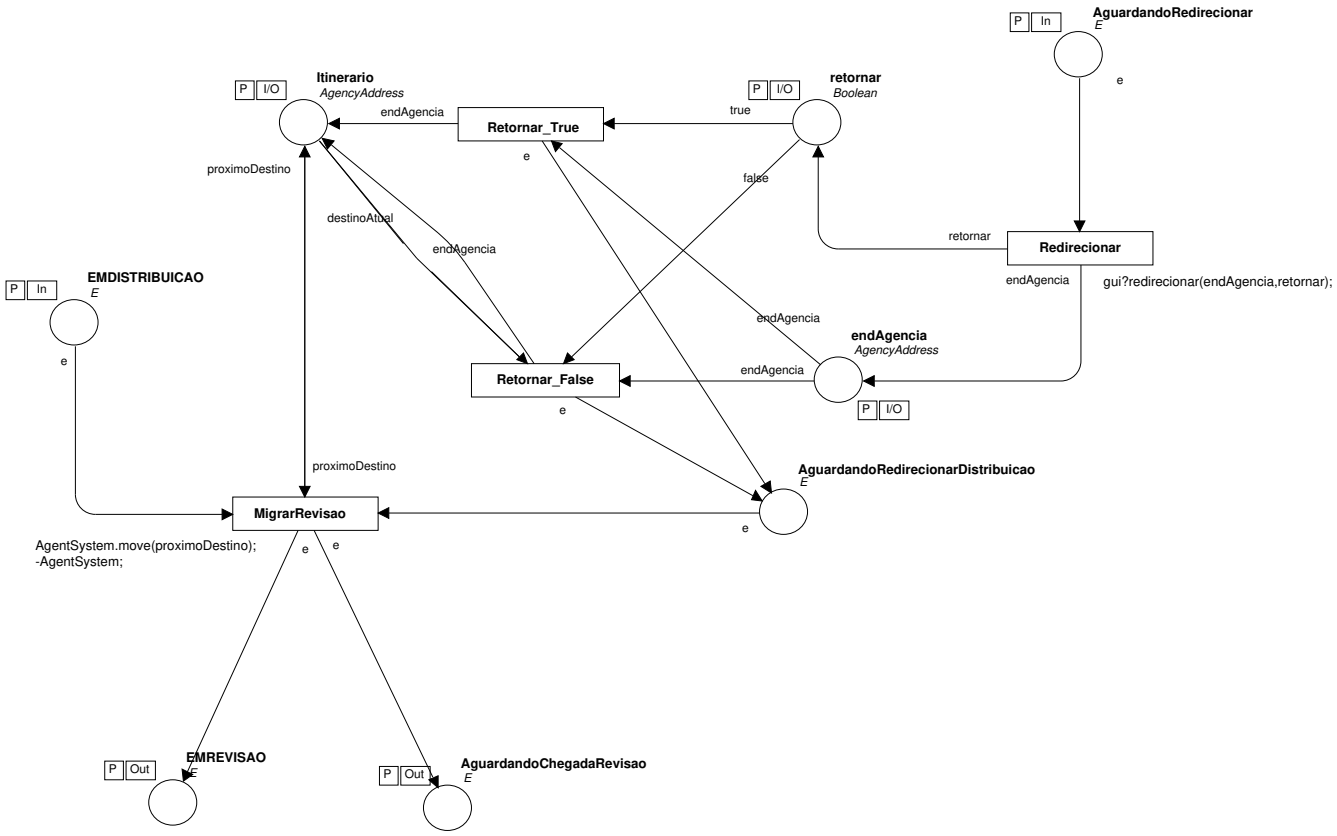


Figura B.5: Página *EmDistribuicao* da Rede de Petri que descreve a Classe *AgenteFormRevisao*

endereços de agências que estiverem neste lugar deverão ser percorridos pelo agente e os arcos de entrada e de saída originados ou destinados a este lugar representarão as operações

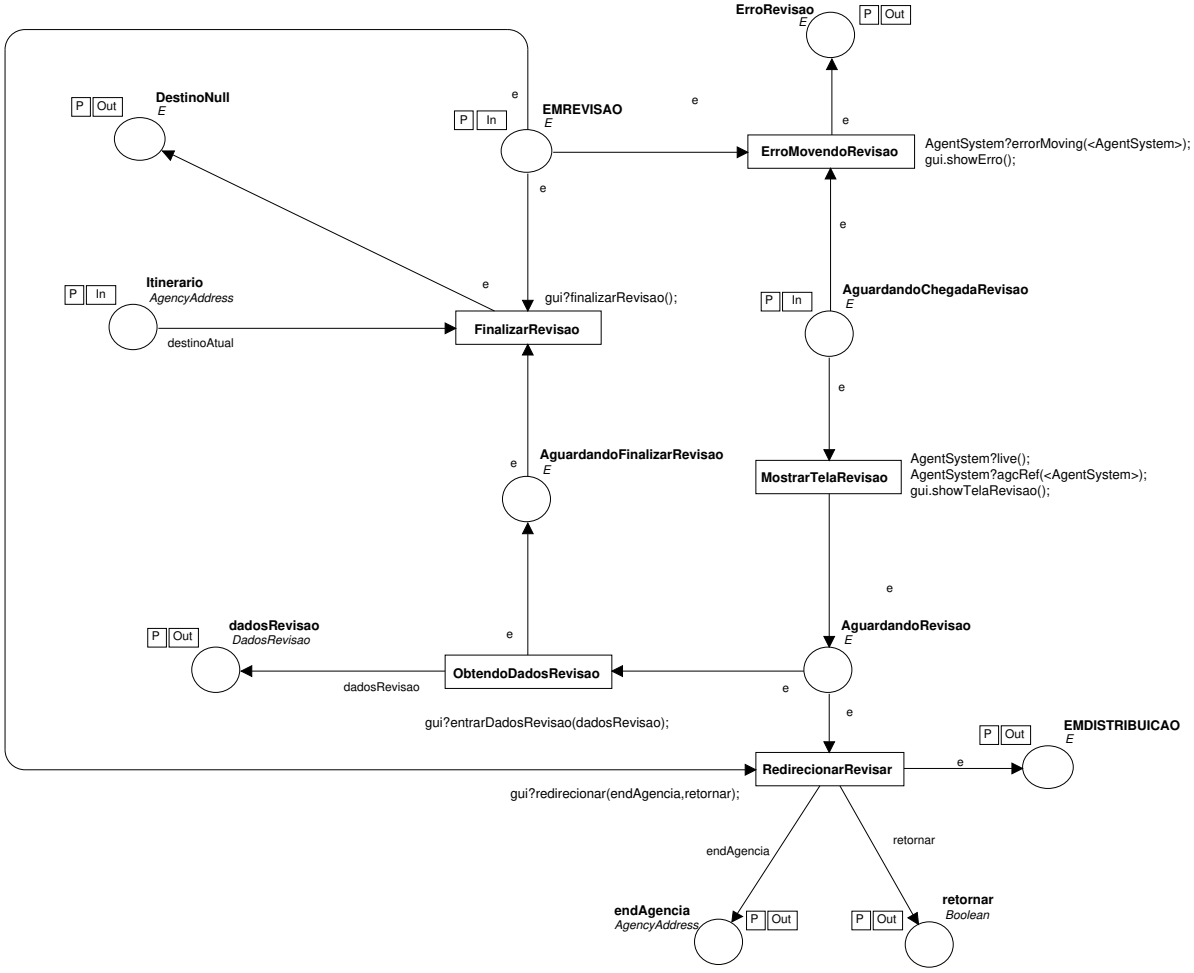


Figura B.6: Página *EmRevisao* da Rede de Petri que descreve a Classe *AgenteFormRevisao*

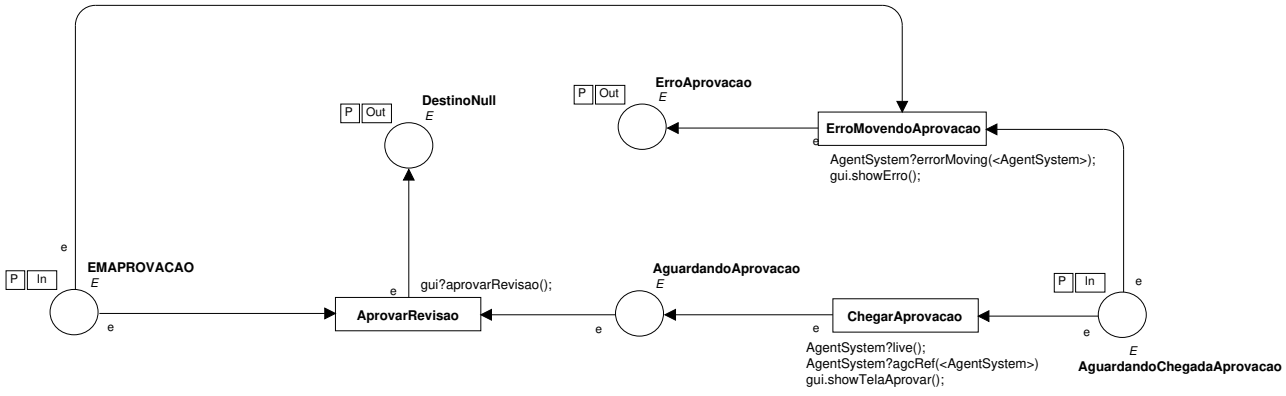


Figura B.7: Página *EmAprovacao* da Rede de Petri que descreve a Classe *AgenteFormRevisao*

da classe *Itinerario*.

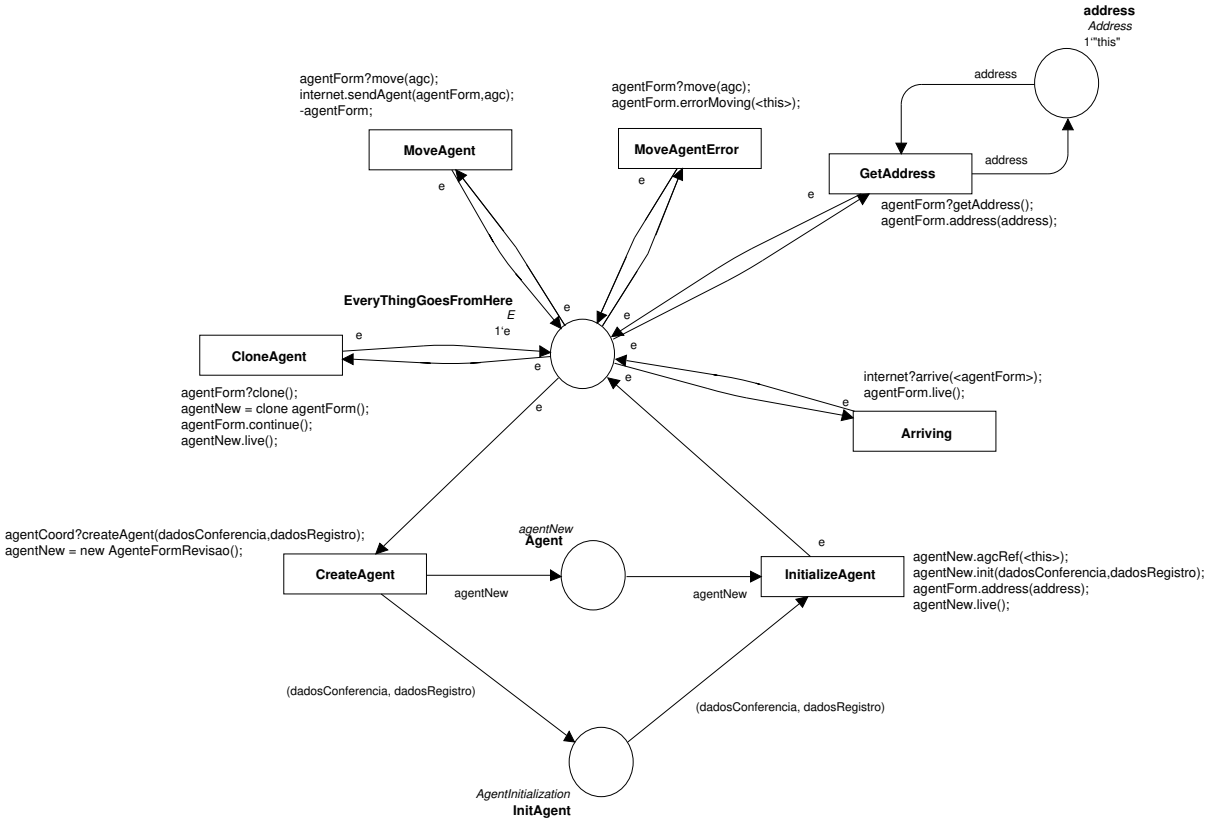
Ao disparar a transição "*Iniciar*", dados serão colocados nos lugares "*Itinerario*", "*da-*

dosConferencia" e "*dadosRegistro*", além de uma ficha no lugar "*CRIADOAGORA*", significando que o agente passou para o estado **CRIADOAGORA**. Neste estado, a transição de substituição "*EmClonagem*" poderá disparar e isto resultará em uma ficha colocada no lugar "*Erro*" ou em fichas nos lugares "*EMDISTRIBUICAO*" e "*AguardandoRedirecionar*". Com este último modo de disparo, a transição "*EmDistribuicao*" estará habilitada para o disparo. Após seu disparo, a transição "*EmRevisao*" disparará e colocará uma ficha no lugar "*DestinoNull*". Caso a transição "*DestinoNull_False*" dispare, em seguida a transição de substituição "*EmAprovacao*" estará habilitada para disparo, e, caso a transição "*True*" dispare, a transição "*RegistrarResultado*" estará habilitada para disparo, registrando o resultado da execução do agente e o levando ao estado **FINALIZOU**.

No modelo original do sistema as interfaces gráficas dos agentes não estavam modeladas, e sim a forma com que elas irão interagir com os agentes através dos protocolos. No entanto, a geração do espaço de estados para tal modelo resultaria em uma explosão do espaço de estados[1], uma vez que como foi dito no começo desta seção, não apresentamos as redes de Petri pra todas as classes e sim a rede de Petri principal da classe *AgenteFormRevisao*, uma vez que o intuito da seção, bem como o do capítulo, é o de ilustrar uma aplicação da API JMobile para um sistema de maior porte.

Apresentamos a seguir o restante das redes que compõem o modelo do Sistema de Conferência sem maiores detalhes.

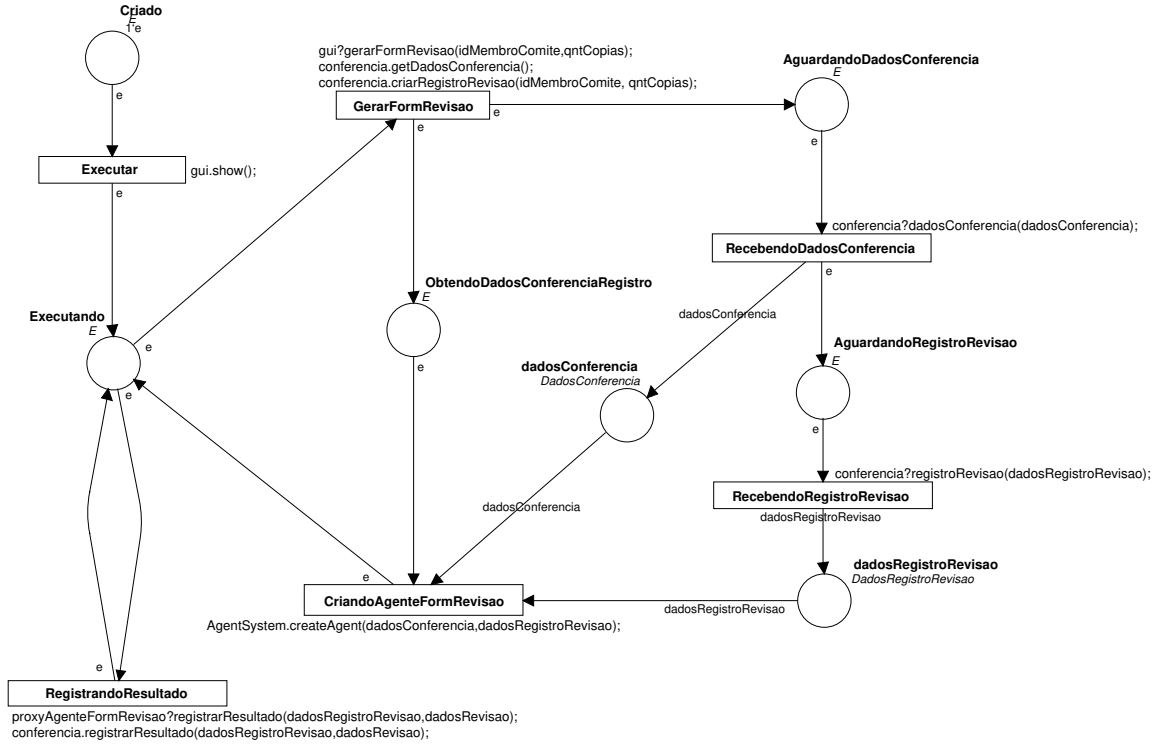
[1]Este termo é utilizado quando o espaço de estados é tão grande a ponto que as ferramentas de geração, verificação de modelos e outras não conseguem tratá-lo ou o tempo e a capacidade de processamento é inviável.




```
color E = with e;
color Address = string;
color AgenteCoordenador = with agtC1 | agtC2 | agtC3;
color AgenteFormRevisao = with agtF1 | agtF2 | agtF3;
color DadosConferencia = with dadosconf1 | dadosConf2 | dadosConf3;
color DadosRegistroRevisao = with dadosReg1 | dadosReg2 | dadosReg3;


var agentForm, agentNew : AgenteFormRevisao;
var agentCoord : AgenteCoordenador;
var dadosRegistro : DadosRegistroRevisao;
var dadosConferencia : DadosConferencia;
var address : Address;
```

Figura B.8: Rede para a classe Agência




```
color E = with e;
color DadosConferencia = with conf1 | conf2 | conf3;
color DadosRegistroRevisao = with regRev1 | regRev2 | regRev3;

var dadosConferencia : DadosConferencia;
var dadosRegistroRevisao : DadosRegistroRevisao;
```

Figura B.9: Rede para a classe AgenteCoordenador

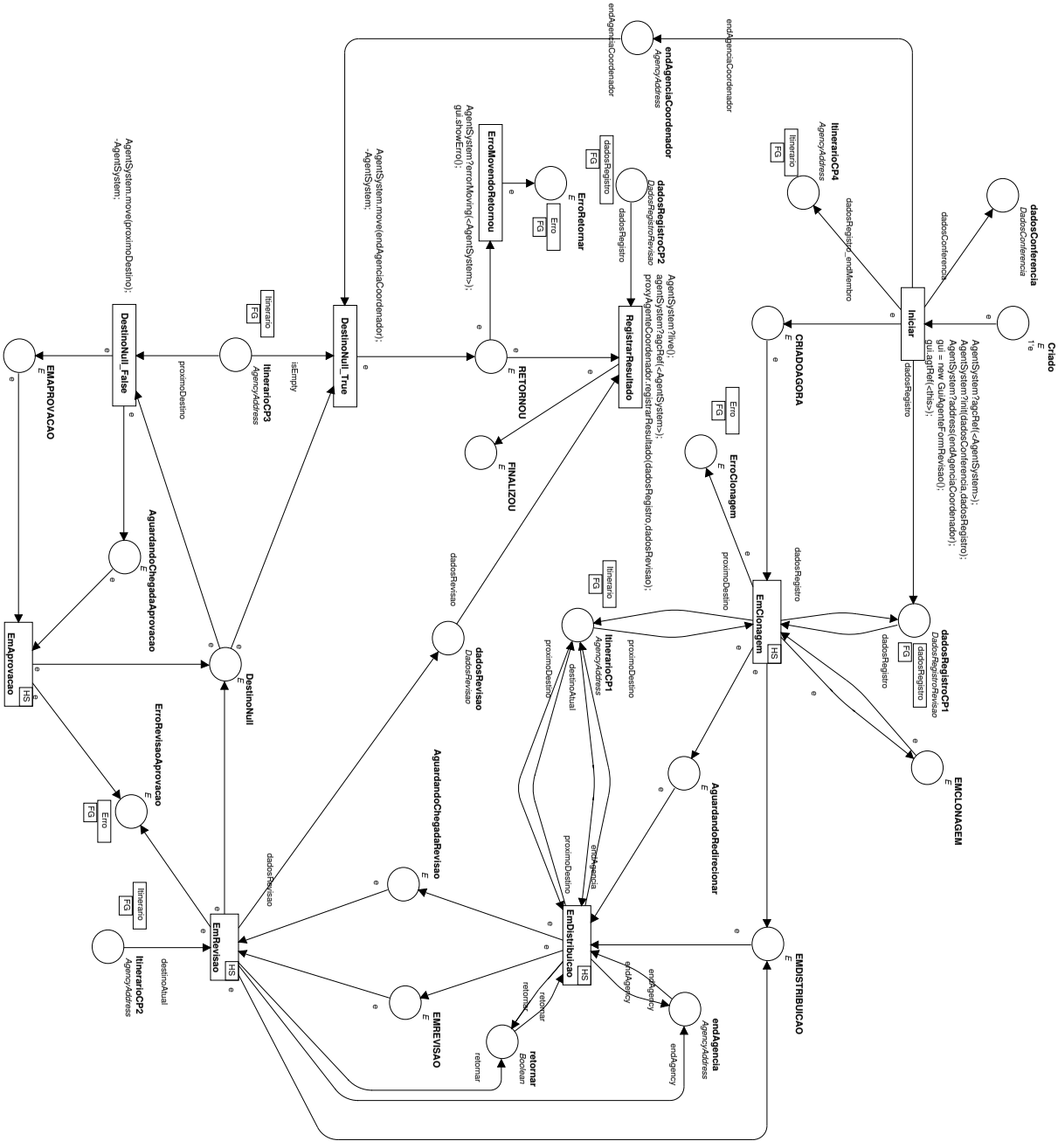


Figura B.10: Rede para a classe AgenteFormRevisao

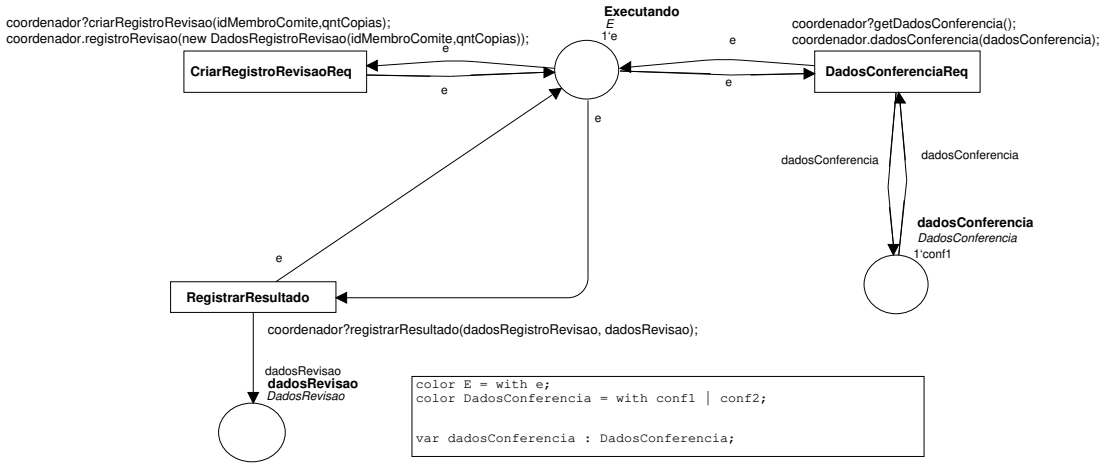


Figura B.11: Rede para a classe Conferencia

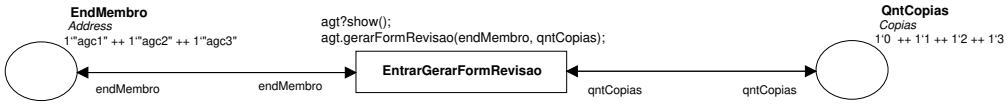


Figura B.12: Rede para a classe GuiAgenteCoordenador

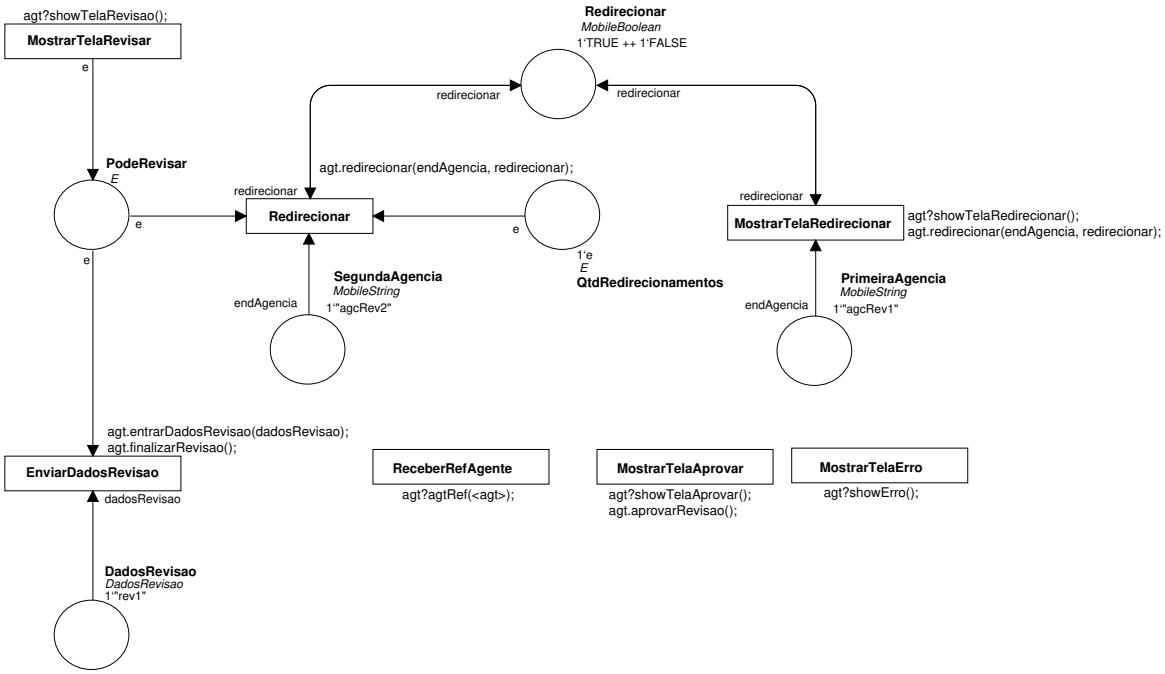


Figura B.13: Rede para a classe GuiAgenteFormRevisao

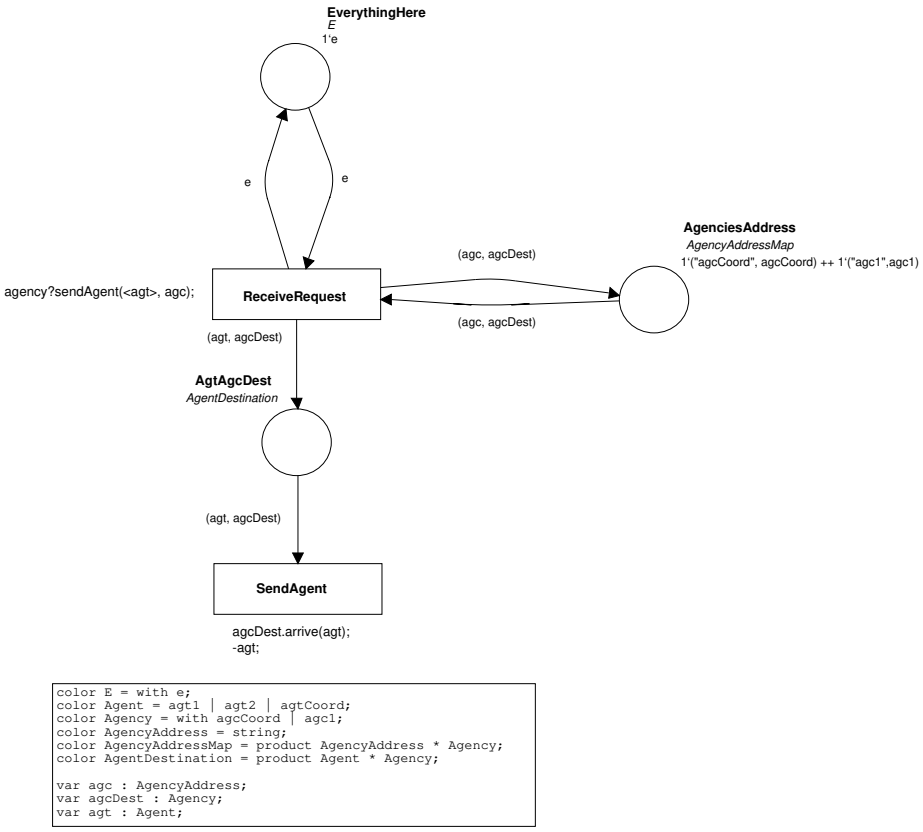


Figura B.14: Rede para a classe Internet