UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

RAQUEL DE MIRANDA BARBOSA

Especificação Formal de Organizações de Sistemas Multiagentes

Tese apresentada como requisito parcial
para a obtenção do grau de
Doutor em Ciência da Computação

Prof. Dr. Antônio Carlos da Rocha Costa
Orientador

Porto Alegre, abril de 2011

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Barbosa, Raquel de Miranda

Especificação Formal de Organizações de Sistemas Multiagentes / Raquel de Miranda Barbosa. – Porto Alegre: PPGC da UFRGS, 2011.

91 f.: il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2011. Orientador: Antônio Carlos da Rocha Costa.

1. Métodos Formais. 2. Organizações de SMA. 3. RSL. 4. Especificação Formal. I. Costa, Antônio Carlos da Rocha. II. Título.

Aos meus amados pais
Afonso e Zelia *(in memorian)*.

AGRADECIMENTOS

Agradeço a todas as pessoas que, de alguma forma, contribuíram para o desenvolvimento desta tese. Em especial agradeço:

- Primeiramente a Deus, presença constante em todos os momentos da minha vida, fortalecendo-me nos momentos difíceis;
- À minha família, que sempre me apoiou na busca de meus objetivos. Em particular, agradeço aos meus pais, Afonso e Zelia, pelos ensinamentos recebidos e por acreditarem nas minhas escolhas; e aos meus irmãos (Ana Paola, Letícia e Junior) que participaram de cada etapa deste percurso, incentivando-me e compreendendo minhas preocupações e ansiedades;
- Ao meu esposo, Marco, pelo constante apoio e incentivo. Por compreender a distância e as ausências que se fizeram necessárias e pela paciência nos momentos de desânimo e *stress*;
- Ao meu orientador, Prof. Antônio Carlos da Rocha Costa, pelos ensinamentos transmitidos e pela amizade e constante incentivo ao desenvolvimento desta tese. Seu apoio foi indispensável à realização deste estudo;
- À professora Patrícia Tedesco (UFPE), por me receber tão bem no período de sanduíche, orientando-me e direcionando minhas atividades. E aos integrantes do VTEAM (*Virtual Team*) que me acolheram junto ao grupo e possibilitaram uma importante experiência durante o período de convivência;
- À professora Graçaliz Dimuro, responsável por despertar o meu interesse para a pesquisa científica, acompanhando meu desenvolvimento desde o curso de graduação;
- À Capes e ao CNPq, pelo suporte financeiro através das bolsas concedidas;
- E aos meus amigos e pessoas especiais que Deus colocou no meu caminho e que, em diferentes contextos, acompanharam as mudanças ocorridas durante este período participando, de alguma forma, desta realização.

“O rio atinge seus objetivos
porque aprendeu a contornar obstáculos.”
LAO-TSÉ

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

AGR	Agent/Group/Role
AOSE	Agent-Oriented Software Engineering
CSP	Communicating Sequential Processes
MaSE	Multiagent Systems Engineering
MOISE	Model of Organization for multI-agent SystEms
OperA	Organizations per Agents
O-MaSE	Organization-based MaSE
OMNI	Organizational Model for Normative Institutions
POA	Programação Orientada a Agentes
POO	Programação Orientada a Objetos
RAISE	Rigorous Approach to Industrial Software
RSL	Raise Specification Language
SMA	Sistema Multiagente
UML	Unified Modeling Language
VDM	Vienna Development Method

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A abordagem de sistemas multiagentes tem sido cada vez mais utilizada para o desenvolvimento de sistemas complexos, o que despertou o interesse das pesquisas na área de engenharia de software orientada a agentes (AOSE) e modelos organizacionais. Neste contexto, esta tese estuda a aplicabilidade de alguns métodos formais tradicionais de engenharia de software para a especificação formal de organizações de sistemas multiagentes, analisando o uso da linguagem de especificação formal RSL para representar o modelo organizacional PopOrg. A escolha da linguagem RSL ocorreu pelo fato de ela ser uma linguagem de especificação formal que cobre amplo espectro de métodos de especificação formal (baseados em modelos e baseados em propriedades, aplicativos e imperativos, sequenciais e concorrentes) e o modelo PopOrg foi escolhido por ser um modelo mínimo de organização de sistemas multiagentes, concebido para representar o conjunto mínimo de aspectos estruturais e operacionais que tais organizações devem ter. O uso da linguagem RSL foi avaliado tanto para a especificação do aspecto estrutural dos sistemas PopOrg, quanto para especificação operacional desses sistemas. Um estudo preliminar realizado com a linguagem CSP para a especificação operacional do modelo PopOrg também é apresentado, visto que serviu como base para a especificação em RSL. Ao final, apresenta-se uma sugestão de extensão da linguagem RSL para sua maior aplicabilidade à especificação de sistemas multiagentes.

Palavras-chave: Métodos Formais, Organizações de SMA, RSL, Especificação Formal.

Formal Specification of Multiagent Systems Organizations

ABSTRACT

The multiagent systems approach have been increasingly used for the development of complex systems, which aroused the interest of research in Agent Oriented Software Engineering (AOSE) and organizational models. In this context, this thesis studies the applicability of some traditional formal methods of software engineering for the formal specification of multiagent systems organizations, analyzing the use of RSL formal specification language to represent the PopOrg organizational model. The choice of RSL language occurred because it is a formal specification language that covers a wide spectrum of formal specification methods (models-based and properties-based, applicative and imperative, sequential and concurrent) and the PopOrg model was chosen because it is a minimal model of multiagent systems organization, designed to represent the minimum set of structural and operational aspects that such organizations should have. The use of RSL language was evaluated both for specifying the structural aspect of PopOrg systems and the operational specification for these systems. A preliminary study carried out with the CSP language for the operational specification of PopOrg model is also presented, as was the basis for the specification in RSL. In the end, a suggestion is given for an extension of the RSL language, to allow for its wider applicability to the specification of multiagent systems.

Keywords: Formal Methods, MAS Organizations, RSL Formal Specification.

1 INTRODUÇÃO

O uso da abordagem de sistemas multiagentes no desenvolvimento de sistemas complexos tem aumentado consideravelmente nos últimos anos. Isto ocorre devido às características desta abordagem e à adequabilidade das técnicas orientadas a agentes para o desenvolvimento de sistemas complexos (JENNINGS, 2001).

Dentre as características fundamentais da noção de modelagem conceitual orientada a agentes está a noção de "autonomia" (WOOLDRIDGE; JENNINGS, 1995). A presença desta característica, no entanto, adiciona uma complexidade considerável às questões de modelagem do sistema, quando comparada à modelagem como sistema de objetos.

Outra questão importante está relacionada aos aspectos sociais dos sistemas de agentes. Eles referem-se aos relacionamentos entre agentes e são importantes devido às questões centrais desta abordagem, como cooperação, competição, negociação, etc. Em particular, a organização social de um sistema deve ser vista como um aspecto central de um modelo orientado a agentes, porque ela define o conjunto de papéis que agentes podem desempenhar no sistema, o conjunto de possíveis relacionamentos que estes papéis podem ter, bem como características reguladoras do sistema, como normas, compromissos, acordos, etc.

Para dar suporte à adoção desta abordagem orientada a agentes, diversas metodologias de desenvolvimento de software têm sido propostas , assim definindo uma área específica da engenharia de software, conhecida como Engenharia de Software Orientada a Agentes (*AOSE - Agent Oriented Software Engineering*) (JENNINGS; WOOLDRIDGE, 2000). Dentre estas metodologias estão GAIA (ZAMBONELLI; JENNINGS; WOOLDRIDGE, 2003), MaSE (DELOACH, 2006), Tropos (GIORGINI et al., 2003) e Prometheus (WINIKOFF; PADGHAM, 2004), descritas na seção 2.4.

Muitas das metodologias AOSE existentes atualmente surgem como modificações de metodologias orientadas a objetos já desenvolvidas, e existem ainda muitas lacunas no que se refere a aspectos particulares de sistemas multiagentes. Poucas metodologias utilizam métodos formais em seu desenvolvimento, e a especificação formal de organizações de sistemas multiagentes não foi, ainda, completamente definida como área de pesquisa na área de Engenharia de Software.

Na área de sistemas multiagentes, encontram-se vários estudos preocupados com a modelagem de sistemas de agentes, tais como MOISE$^+$ (HüBNER; SICHMAN, 2003), AGR (FERBER; GUTKNECHT; MICHEL, 2004), OperA (DIGNUM, 2004), OMNI (DIGNUM; V´AZQUEZ-SALCEDA; DIGNUM, 2004) e ISLANDER (ESTEVA; CRUZ; SIERRA, 2002), apresentados na seção 2.5, e PopOrg (COSTA; DIMURO, 2007a), detalhado no capítulo 4.

Em Coutinho (2009) é apresentada uma classificação de dimensões de modelagem encontradas na maioria dos modelos organizacionais. São elas: funcional, estrutural, dia-

lógica, normativa e ontológica. A dimensão funcional caracteriza-se pela especificação de metas e decomposição de metas ligadas à satisfação do propósito de organização de agentes. A dimensão estrutural está ligada à especificação de papéis, grupos e relacionamentos entre estes, que podem ou não ser definidos a partir de objetivos organizacionais. A modelagem dialógica caracteriza-se pela especificação de estruturas de interação direta entre papéis por troca de mensagens tendo em vista a realização de objetivos organizacionais. Na dimensão normativa são definidas as normas que interrelacionam e regulamentam elementos funcionais, estruturais e dialógicos. E, na modelagem ontológica, são definidas as estruturas conceituais (ontologias de domínio) compartilhadas.

Se, por um lado, na modelagem de sistemas orientada a objetos o uso de métodos formais bem estabelecidos é tido como, no mínimo, conveniente para o projetista do sistema, em termos de prevenção de erros e execução de propriedades desejadas no sistema (BJØRNER, 2007), no caso da modelagem orientada a agentes o uso destes métodos formais torna-se não menos obrigatório, devido à complexidade das questões envolvidas. Neste contexto, o presente trabalho busca uma avaliação crítica da possibilidade de utilização de métodos formais tradicionais de Engenharia de Software na especificação de organizações de SMAs, visando estender para estes sistemas a possibilidade de utilização de ferramentas usuais de verificação formal de sistemas.

Para tanto, tomou-se por base o modelo PopOrg. A escolha por este modelo organizacional ocorreu devido a ele ser considerado um modelo mínimo de organizações de SMAs, visto que ele pode ser pensado como embutido em todos os outros modelos suficientemente completos. Desta forma, o PopOrg utiliza o conjunto mínimo de elementos necessários para representar os aspectos organizacionais trabalhando com as dimensões de modelagem estrutural e dialógica. Dentre os métodos tradicionais, optou-se por utilizar a linguagem RSL para a especificação dos sistemas baseados em PopOrg pelo fato de que ela abrange as principais características das demais linguagens existentes.

1.1 Objetivos

O objetivo do presente trabalho é o estudo da viabilidade da especificação de organizações de sistemas multiagentes, através de métodos formais usuais da Engenharia de Software.

Para atingir este objetivo, busca-se alcançar os seguintes objetivos específicos:

- Estudo das diferentes metodologias existentes para modelagem de organizações de SMA e a comparação dos respectivos modelos com o modelo PopOrg.

- Estudo dos métodos de ES tradicionais, especialmente o método RAISE, de sua linguagem formal RSL e de suas ferramentas de verificação.

- Realização de um estudo de caso para avaliação do uso da linguagem RSL e de suas ferramentas de verificação na especificação formal de organizações de SMA na perspectiva do modelo PopOrg.

- Análise crítica do estudo de caso e possível estabelecimento de extensões à linguagem RSL para torná-la plenamente adequada à modelagem de organizações de SMA na perspectiva do modelo PopOrg.

1.2 Contribuições

Como principais contribuições da presente tese destacam-se:

- Definição de uma semântica PopOrg de CSP, com base em uma extensão do modelo de traces;
- Representação em RSL do modelo PopOrg (estrutural e operacional), possibilitando a aplicação de ferramentas de verificação formal aos sistemas especificados de acordo com esse modelo;
- Avaliação crítica dos resultados obtidos na utilização de métodos formais usuais da ES na especificação de organizações de SMAs.

1.3 Organização

Esta texto está organizado conforme descrito a seguir:

- No capítulo 2 é realizada uma introdução à área de Engenharia de Software Orientada a Agentes, detalhando-se as diferenças entre o paradigma orientado a objetos e o paradigma orientado a agentes. A seguir são apresentadas algumas das principais metodologias propostas para o desenvolvimento de sistemas baseados em agentes e modelos organizacionais de sistemas de agentes.
- O capítulo 3 apresenta a importância do uso de métodos formais para a área de Engenharia de Software, destacando suas principais aplicações e apresentando conceitos relacionados a técnicas, linguagens e ferramentas de especificação formal. Apresenta, ainda, a aplicação de métodos formais na área de Arquitetura de Software e de Sistemas de Agentes, apresentando alguns trabalhos relacionados.
- No capítulo 4 é descrito o modelo PopOrg, escolhido como objeto de estudo desta tese. Apresenta-se a justificativa para tal escolha e os conceitos necessários para o entendimento do trabalho realizado.
- Um trabalho preliminar realizado com a utilização da linguagem CSP e do modelo PopOrg é descrito no capítulo 5. A linguagem CSP foi utilizada para especificar comportamentos e processos de troca de sistemas baseados em PopOrg. O capítulo descreve a representação proposta, seguida de um estudo de caso realizado e as conclusões obtidas.
- O capítulo 6 descreve as principais características do método formal RAISE e sua linguagem de especificação RSL.
- A seguir, no capítulo 7, é apresentada a representação estrutural e operacional do modelo PopOrg utilizando a linguagem RSL nos seus estilos de especificação aplicativo e concorrente.
- Um estudo de caso no contexto do processo de revisão de artigos é descrito no capítulo 8, utilizando-se as representações definidas na tese.
- No capítulo 9 é apresentada uma avaliação do trabalho realizado.

- A conclusão da tese, contribuições e trabalhos futuros são apresentados no capítulo 10, seguidos das referências utilizadas.

Ao final do texto podem ser encontrados os Apêndices A e B que trazem, respectivamente, a especificação estrutural do modelo PopOrg em RSL e a especificação completa do estudo de caso realizado.

2 ENGENHARIA DE SOFTWARE ORIENTADA A AGENTES

A abordagem de agentes tem se mostrado bastante adequada para o desenvolvimento de sistemas complexos. Com o crescimento da programação orientada a agentes, surge a necessidade da existência de modelos, métodos e ferramentas para que este paradigma possa ser adotado e difundido com sucesso. Esta necessidade deu origem às pesquisas na área de Engenharia de Software Orientada a Agentes (AOSE - *Agent-Oriented Software Engineering*).

Este capítulo apresenta alguns conceitos fundamentais relacionados a agentes, bem como as vantagens de uma abordagem orientada a agentes e sua comparação em relação à orientação a objetos. A seguir são descritas algumas metodologias de engenharia de software orientada a agentes e modelos de organizações de sistemas multiagentes.

2.1 Agentes

Não existe uma definição única para o termo agente na área de Sistemas Multiagentes. Em Wooldridge (2002), por exemplo, um agente é definido como "um sistema de computador que está situado em um ambiente, e que é capaz de ação autônoma neste ambiente para atingir seus objetivos de projeto". Resumidamente, agentes são entidades situadas em um ambiente (o qual eles observam e onde eles executam ações), que têm objetivos particulares para alcançar, e são autônomos (com controle sobre seu estado interno e seu comportamento).

São consideradas, na área de SMA, duas noções de agência, a noção fraca e a forte. Na noção fraca, os agentes têm vontade própria (autonomia), são capazes de interagir com os outros (habilidade social), respondem a estímulos (reatividade) e tomam iniciativa (pró-atividade). Na noção forte de agência, estas noções fracas de agência são preservadas e, além disso, os agentes têm capacidade de mobilidade, veracidade, benevolência e racionalidade (eles irão tentar atingir seus objetivos da melhor maneira).

A escolha da abordagem de agentes para sistemas complexos pode ser justificada por diversos argumentos, tais como os apresentados em Jennings (2001): i) as decomposições orientadas a agentes são uma maneira de dividir o espaço do problema de um sistema complexo; ii) as abstrações chave, presentes no modo de pensar orientado a agentes, são um meio natural de modelar sistemas complexos e iii) a filosofia orientada a agentes para identificar e gerenciar relacionamentos organizacionais é apropriada para lidar com as dependências e interações que existem em um sistema complexo.

Em sistemas multiagentes, os conceitos sociais e organizacionais têm grande importância, pois eles representam os papéis que os agentes desempenham no sistema e as

possíveis interações que eles podem realizar, além de definir relações de dependência.

2.2 Agentes X Objetos

Algumas comparações podem ser feitas em relação aos paradigmas "Orientado a Objetos" e "Orientado a Agentes", tais como:

- Na programação orientação a objetos (POO), a unidade básica é o objeto, enquanto na programação orientada a agentes (POA) é o agente.
- A POA fixa o estado (chamado estado mental) dos módulos (chamados agentes) para que eles consistam de crenças, capacidades e decisões.
- O processo de computação em ambas as abordagens é o mesmo, trocas de mensagens e métodos de resposta.
- Os tipos de mensagens utilizados na POA referem-se a informações, requisições, consultas, ofertas, aceitação, rejeição, etc. Estes tipos estão relacionados à teoria dos atos de fala onde, cada tipo de ato de comunicação envolve diferentes pressuposições e tem diferentes efeitos (SEARLE, 1969).

2.3 Programação Orientada a Agentes

Este paradigma foi proposto por Shoham (1993). A ideia principal é a de programar os agentes diretamente em termos de noções mentalísticas e intencionais (como crenças, desejos e intenções), pois da mesma maneira que estas noções auxiliam o raciocínio humano sobre atividades do quotidiano, a programação de agentes desta forma poderia facilitar a concepção de sistemas complexos.

Segundo Shoham (1993), os agentes diferem de objetos convencionais de software por que possuem estados mentais básicos e têm atos de fala nas suas mensagens (isto é, há mensagens que simplesmente informam e há mensagens que solicitam informação).

Tipicamente os agentes são entendidos como um modelo baseado em três atributos: crenças, desejos e intenções; caracterizados pela arquitetura BDI (*Beliefes-Desires-Intentions*). As ações de um agente são determinadas por suas decisões ou escolhas. As decisões são derivadas das crenças do agente. As crenças referem-se ao estado do mundo (passado, presente ou futuro), ao estado mental deste e de outros agentes e às capacidades dos agentes em interação. As decisões são dependentes de decisões anteriores.

Além do nível individual dos agentes (que pode ser expressado através de noções mentalísticas), destaca-se o nível social, onde surgem as noções de papéis, grupos e organizações.

A primeira linguagem de programação orientada a agentes foi Agent0 (SHOHAM, 1993). Nesta linguagem, um agente é especificado em termos de um conjunto de capacidades (coisas que o agente pode fazer), um conjunto de crenças iniciais e compromissos, e um conjunto de regras de compromissos (que é o componente-chave, que indica como o agente age). Depois outras linguagens surgiram, como por exemplo AgentSpeak (BORDINI; VIEIRA, 2003), 3APL.

A utilização deste paradigma de programação pode ser observada em agentes na internet, assistentes pessoais, negociadores, jogos e simulação de sistemas, entre outros.

2.4 Metodologias AOSE

As metodologias AOSE surgiram para tentar sistematizar o desenvolvimento de software orientado a agentes, oferecendo também algumas ferramenta para automatizar este processo. Algumas das principais metodologias são descritas a seguir.

2.4.1 GAIA

A metodologia GAIA, apresentada em Wooldridge, Jennings e Kinny (2000) foi projetada para tratar aspectos de análise e projeto de sistemas baseados em agentes. Esta metodologia trata ambos os aspectos de nível macro (sociedade) e micro (agentes) e mostra-se neutra em relação ao domínio e arquitetura dos agentes.

Gaia é fundamentada na visão de um sistema multiagente como uma organização computacional consistindo de vários papéis de agentes interagindo. O seu objetivo é permitir que o analista possa partir sistematicamente de uma declaração de requisitos a um projeto detalhado o suficiente para ser implementado diretamente. A fase de captura de requisitos é vista como sendo independente do paradigma de análise e projeto.

Em Zambonelli, Jennings e Wooldridge (2003), é apresentada uma extensão à metodologia Gaia para explorar novas abstrações organizacionais necessárias para projetar e construir sistemas complexos.

Os principais conceitos utilizados nesta metodologia podem ser divididos em duas categorias: abstratos e concretos. Os conceitos abstratos são utilizados durante a fase de análise para conceitualizar o sistema (e.g. papéis, permissões, responsabilidades, protocolos, atividades, propriedades *liveness* e propriedades de segurança), enquanto os concretos são utilizados na fase de projeto, correspondendo a elementos que estarão presentes na implementação (e.g. tipos de agentes, serviços e conhecimentos).

Gaia utiliza uma notação baseada no método FUSION (COLEMAN et al., 1996), utilizada na análise e projeto orientado a objetos.

2.4.2 MaSE

Inicialmente proposta em Wood e Delloach (2001), MaSE (*Multiagent Systems Engineering*) é uma metodologia que apresenta um ciclo de vida completo e um ambiente complementar para análise, projeto e desenvolvimento de sistemas multiagentes heterogêneos. MaSE utiliza-se de modelos gráficos para descrever objetivos do sistema, comportamentos, tipos de agentes e interfaces de comunicação entre agentes, fornecendo ainda uma maneira de especificar uma definição detalhada do projeto interno do agente (independente da arquitetura).

Na visão de MaSE, os agentes são pensados como uma especialização dos objetos. Ao invés de simples objetos com métodos que podem ser invocados por outros objetos, os agentes coordenam-se com outros através de diálogos e agem proativamente para atingir objetivos individuais e do sistema (DELOACH; WOOD; SPARKMAN, 2001). Por este motivo, as técnicas utilizadas são baseadas em orientação a objetos e aplicadas na especificação e projeto de sistemas multiagentes.

Juntamente com a metodologia foi desenvolvida a ferramenta agentTool, servindo como plataforma de validação para a MaSE. Esta ferramenta é baseada em gráficos e completamente interativa (DELOACH; WOOD; SPARKMAN, 2001), dando suporte a todas as fases da MaSE e à verificação automática de comunicação entre agentes e geração de código para vários *frameworks* de sistemas multiagentes. Atualmente, esta ferramenta pode ser obtida em http://agenttool.cis.ksu.edu/.

Inicialmente MaSE foi desenvolvida para o projeto de sistemas multiagentes estáticos. Para evitar esta restrição, Delloach (2006) apresenta uma extensão para a metodologia MaSE, chamada O-MaSE, que permite o projeto de uma organização multiagente, com uma estrutura na qual o sistema multiagente pode se adaptar. Muitos dos diagramas utilizados em O-MaSE (*Organization-based MaSE*) são variantes dos diagramas de classes UML usando palavras-chave para descrever a diferença entre objetivos, papéis, capacidades, classes de agentes, etc.

Deloach (2006) apresenta um metamodelo que descreve o conhecimento necessário para definir uma organização. Os elementos que o compõem são: objetivos, papéis, agentes, capacidades, missões, políticas e ontologia. As capacidades são elementos centrais no processo de determinar quais agentes podem assumir quais papéis e o quão bem eles podem fazê-lo, enquanto políticas restringem as missões dos agentes para papéis, controlando os estados permitidos da organização.

2.4.3 Tropos

TROPOS[1] é uma metodologia para desenvolvimento de SMAs, baseada em duas características principais: (i) a noção de agentes e noções mentalísticas relacionadas são utilizadas em todas as fases de desenvolvimento, desde os requisitos iniciais até a implementação; (ii) a metodologia enfatiza a análise de requisitos iniciais, a fase que precede a prescrição de requisitos (o que a torna diferente das outras metodologias).

Tropos adota o modelo i*, de Eric Yu (YU apud (GIORGINI et al., 2003)), que oferece atores (agentes, papéis, ou posições), objetivos e dependências de atores como conceitos primitivos para modelar uma aplicação durante a análise de requisitos iniciais. A metodologia dá suporte a quatro fases do desenvolvimento de software: (i) análise de requisitos iniciais; (ii) análise de requisitos finais; (iii) projeto arquitetural; e (iv) projeto detalhado.

Tropos suporta a aplicação de técnicas de análise formal para a verificação de especificação de requisitos, baseada na Formal Tropos (FT), uma linguagem de especificação que oferece todas as noções mentalísticas padrões de Tropos e suplementos com uma rica linguagem de especificação temporal inspirada em KAOS (LAMSWEERDE apud (GIORGINI et al., 2003).

Formal Tropos permite a descrição dos aspectos dinâmicos dos modelos Tropos, ou seja, não foca apenas nos elementos intencionais, mas também nas circunstâncias em que eles ocorrem e nas condições que levam à sua ocorrência. A especificação descreve elementos relevantes (atores, objetivos, dependências...) de um domínio e relacionamentos entre eles. A descrição de cada um deles é estruturada em duas camadas, sendo a mais externa semelhante a uma declaração de classe (associa elementos ao conjunto de atributos que define sua estrutura), e a mais interna expressa restrições no ciclo de vida dos objetos, através de uma lógica temporal linear de primeira ordem.

Depois de definida a especificação, ela pode ser formalmente verificada a fim de se identificar erros e ambiguidades. Um protótipo de ferramenta para fazer esta verificação é T-Tool [2], baseada em verificação de modelos de estado finito. Com base na especificação, esta ferramenta constrói um modelo finito que representa todos os comportamentos possíveis do domínio que satisfazem as restrições da especificação.

[1] $http://www.troposproject.org.br$
[2] $http://www.dit.unitn.it/\ ft/ft_tool.html$

2.4.4 Prometheus

Segundo Winikoff e Padgham (2004), Prometheus é uma metodologia para desenvolvimento de agentes inteligentes que diferencia-se das demais por ser detalhada e completa abrangendo todas as atividades necessárias no desenvolvimento de sistemas de agentes inteligentes. Uma das preocupações no desenvolvimento desta metodologia foi o detalhamento suficiente para ser usada por não-especialistas, podendo ser utilizada tanto por estudantes de graduação quanto por profissionais na área industrial.

Prometheus foi desenvolvida pelo *Agent Oriented Software Group* (AOS - Estados Unidos, Reino Unido e Austrália) e é composta por três fases:

- Especificação do Sistema: foca na identificação das funcionalidades do sistema junto com suas percepções, ações e quaisquer fontes de dados compartilhados.
- Projeto Arquitetural: utiliza as saídas da fase anterior para determinar quais agentes existirão no sistema e como eles irão interagir. Envolve atividades como definição dos tipos de agentes e definição das interações entre agentes.
- Projeto Detalhado: foca no desenvolvimento da estrutura interna de cada agente e como ele irá realizar suas tarefas dentro do sistema. É nesta etapa que esta metodologia torna-se específica para agentes que usam planos definidos pelo usuário, ativados por objetivos ou eventos, como as implementações de sistemas BDI (*Beliefs, Desires, Intentions*) (RAO; GEORGEFF, 1995).

De acordo com Padgham e Winikoff (2002), uma das vantagens desta metodologia é a quantidade de locais onde ferramentas automatizadas podem ser utilizadas para verificar a consistência entre vários modelos produzidos ou processos de projeto. Dentre elas destaca-se a PDT (*Prometheus Design Tool*)[3], que permite aos usuários criar e modificar projetos Prometheus. Ela assegura que não ocorram certas inconsistências e permite a detecção de outras, além de exportar diagramas de projeto individuais e gerar um relatório com o projeto completo.

Outra ferramenta que dá suporte à metodologia Prometheus é a *JACK Development Environment* (uma plataforma comercial de desenvolvimento de agentes, desenvolvida pelo AOS-*Agent Software Engineering*). Ela permite que os diagramas de visão geral sejam desenhados e pode gerar um esqueleto de código a partir destes diagramas.

2.5 Modelos de Organizações de SMA

Na área de SMA existem diversos trabalhos que preocupam-se com a modelagem de organizações de agentes. Segundo Coutinho, Sichman e Boissier (2005), um modelo organizacional é uma técnica de modelagem utilizada para representar a organização social de (organizações em) um Sistema Multiagente. Neste texto são analisados os modelos AGR, MOISE$^+$, OPERA, OMNI, ISLANDER e PopOrg, sendo este último descrito detalhadamente no capítulo 4, pois é o modelo utilizado como referência nesta tese.

2.5.1 AGR

O modelo AGR (*Agent/Group/Role*) (FERBER; GUTKNECHT; MICHEL, 2004) é uma extensão do modelo Aalaadin (FERBER; GUTKNECHT; MICHEL, 1998). Ele é

[3] $http://www.cs.rmit.edu.au/agents/pdt/$

baseado em três conceitos primitivos: (i) Agente: que é uma entidade ativa, que se comunica e que desempenha papéis dentro de grupos, (ii) Grupo: que é um conjunto de agentes que compartilham características comuns e (iii) Papel: que é a representação abstrata de uma posição funcional de um agente em um grupo.

A Figura 2.1 apresenta o meta-modelo de AGR onde é possível observar a separação entre o nível agente (organização concreta) e o nível organização (organização abstrata).

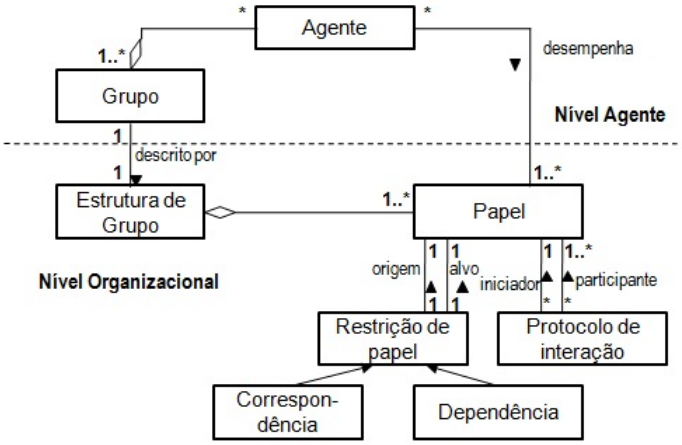

Figura 2.1: Meta-modelo UML de AGR

A especificação deste modelo ocorre de forma semi-formal. Para a especificação de axiomas que definem o aspecto estrutural do modelo AGR é utilizada a Lógica de Primeira Ordem. A Fig.2.2, mostra o axioma que define que dois agentes podem se comunicar apenas se pertencem ao mesmo grupo.

$$\forall\, x, y : Agent, \forall\, m : Message, x.send(y, m) \Rightarrow \exists\, g : Group, member(x, g) \wedge member(y, g)$$

Figura 2.2: Exemplo de Axioma - AGR

A definição das estruturas organizacionais, atividades organizacionais e dinâmicas de grupos é realizada através de diagramas que podem ser utilizados em alguma metodologia AOSE, como por exemplo Gaia (conforme sugere Ferber, Gutknecht e Michel (2004)), onde podem ser preenchidas as características dos papéis e relacionadas com a estrutura organizacional.

2.5.2 MOISE$^+$

O modelo organizacional MOISE$^+$[4] destaca os componentes que formam uma organização e como eles podem contribuir para o SMA. Este modelo considera claramente as três formas de representar restrições organizacionais (papéis, planos e normas).

O MOISE$^+$ é um modelo mais completo que define a organização de um SMA em três dimensões: estrutural, funcional e deôntica (normas), conforme mostra a Figura 2.3. O aspecto estrutural refere-se aos componentes elementares da organização (papéis) e como estão relacionados (ligações entre papéis, grupos de papéis, hierarquias,...). O aspecto funcional especifica como os objetivos globais podem ser atingidos (planos globais,

[4]http://moise.sourceforge.net/

missões,...). Por fim, o aspecto deôntico liga os dois anteriores indicando quais as responsabilidades dos papéis nos planos globais (permissões e obrigações). Uma definição mais detalhada destas três dimensões é encontrada em Hübner e Sichman (2003).

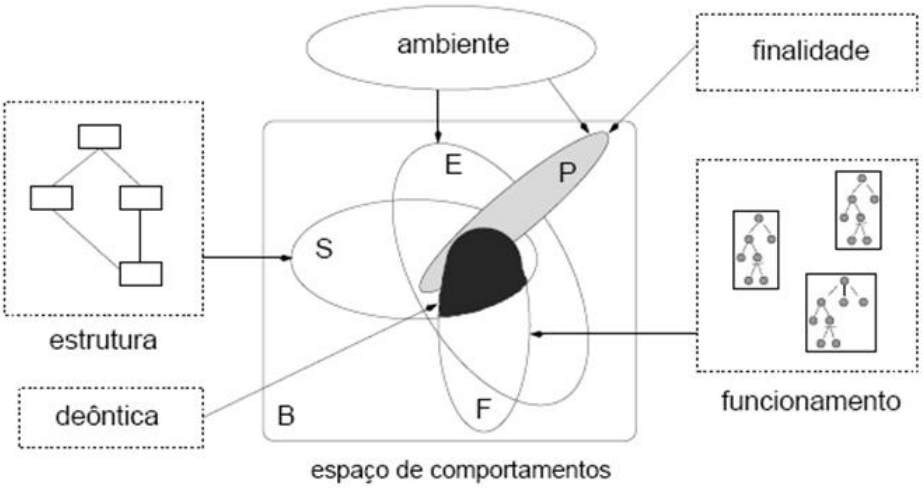

Figura 2.3: Modelo Moise$^+$

A especificação organizacional (EO), composta pelas especificações estrutural, funcional e deôntica, não inclui agentes pois tem um caráter mais abstrato onde os agentes são representados por papéis. Esta especificação pode ser instanciada por um conjunto de agentes, formando uma entidade organizacional (EnO), onde fica estabelecida a posição destes agentes no contexto de uma especificação organizacional (Figura 2.4).

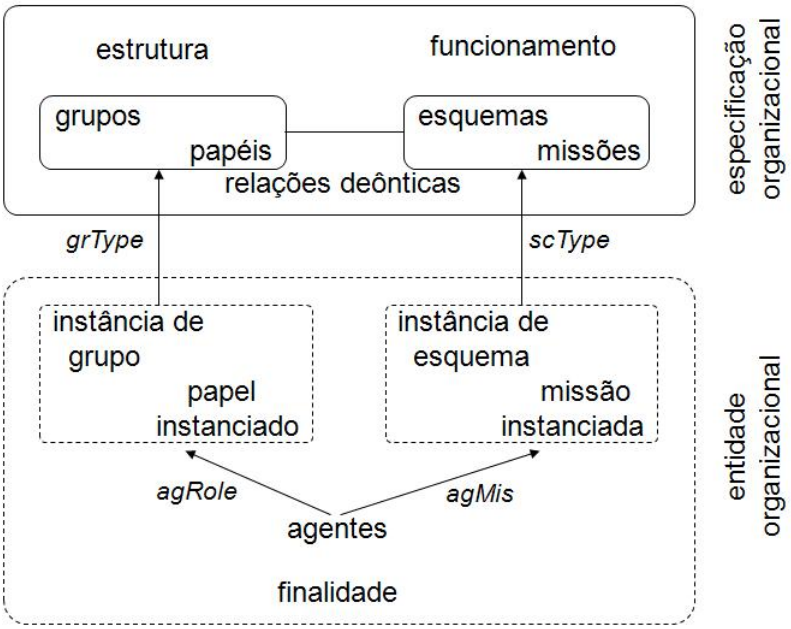

Figura 2.4: Entidade Organizacional - Moise$^+$

Em Hübner et al. (2010) é proposta uma quarta dimensão para o modelo MOISE$^+$, chamada dialógica, focada na comunicação entre papéis, onde são definidos os protocolos utilizados para esta comunicação.

2.5.3 OperA

OperA (***O**rganizations **per A**gents*) é um modelo para sociedades de agentes. De acordo com Dignum (2004), um modelo OperA pode ser pensado como um tipo de protocolo abstrato que determina como agentes membros da sociedade devem agir de acordo com os requisitos sociais. As interações são especificadas em contratos, os quais podem ser traduzidos em expressões formais (usando-se LCR - *Logic for Contract Representation*) e, consequentemente verificadas quanto ao seu cumprimento.

O *framework* Opera representa interações entre agentes de forma que: é independente do projeto interno dos agentes, distingue características organizacionais dos objetivos próprios dos agentes, cria links dinâmicos entre projeto organizacional e populações de agentes, e permite e adaptação de padrões de interação com as características de populações específicas.

O OperA é composto de 3 modelos interrelacionados (Figura 2.5): i) modelo organizacional: que descreve a estrutura organizacional da sociedade, consistindo de papéis e interações, conforme desejado pelos *stakeholders* da organização. ii) modelo social: onde são especificados acordos relativos à representação de papéis por agentes individuais; iii) modelo de interação: dada uma população de agentes, este modelo descreve as possíveis interações entre eles.

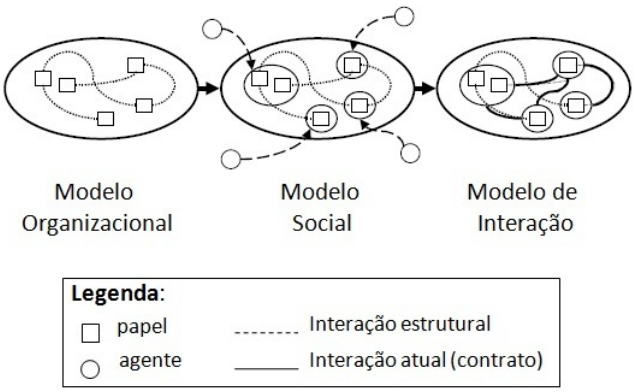

Figura 2.5: Framework organizacional OperA (DIGNUM, 2004)

2.5.4 OMNI

O modelo OMNI (*Organizational Model for Normative Institutions*) (DIGNUM; V´AZQUEZ-SALCEDA; DIGNUM, 2004) é um *framework* integrado para normas, estrutura, interação e ontologias para a modelagem de organizações em SMA. É composto por três dimensões: Normativa, Organizacional e Ontológica.

OMNI é baseado nos modelos OperA (DIGNUM, 2004) e HarmonIA (VáZQUEZ-SALCEDA; DIGNUM, 2003) e está organizado em três níveis de abstração: abstrato, concreto e implementação. No nível abstrato são definidos os estatutos da organização a ser modelada em um alto nível de abstração (similar à primeira etapa na análise de requisitos). Também contém a definição dos termos gerais para qualquer organização e a ontologia do modelo. O nível concreto é onde ocorrem os processos de análise e projeto, partindo dos valores abstratos definidos no nível anterior, refinando seus significados em termos de normas e regras, papéis e conceitos ontológicos concretos. Por fim, o nível de implementação é onde o projeto nas dimensões normativa e organizacional, é implementado em uma arquitetura multiagente, incluindo os mecanismos para decreto de papéis e

aplicação das normas.

O modelo apresenta uma formalização para os diferentes níveis de abstração de normas. No nível normativo abstrato, por exemplo, é utilizada *ANorms* (linguagem para normas abstratas), no nível normativo concreto é utilizada *CNorms* (a linguagem para normas concretas) e, posteriormente, a tradução destas normas para regras em OMNI, marca uma transição de uma perspectiva normativa para uma mais descritiva, onde a linguagem passa a ser a da Lógica Dinâmica Proposicional (uma linguagem mais adequada para expressar ações e restrições de tempo). Em Dignum e Dignum (2007) é apresentada uma formalização lógica para organizações de agentes.

2.5.5 ISLANDER

Uma outra abordagem para trabalhar com a estrutura organizacional de sistemas multiagentes é através da sua modelagem como instituições eletrônicas (SIERRA et al., 2004). Estas instituições definem os papéis participantes, as interações válidas entre os participantes e as normas que irão governá-las.

Uma instituição eletrônica é composta por quatro elementos: (i) *Framework* de Diálogo: que representa o contexto das interações entre agentes que ocorrem através de atos de fala, utilizando um dicionário (Ontologia) bem definido e uma linguagem comum; (ii) Cenas: as interações entre agentes ocorrem por meio de reuniões, chamadas cenas, e são definidas através de protocolos de comunicação, que representam todos os diálogos que os agentes podem ter em cada cena; (iii) Estrutura performativa: especifica as relações entre as cenas, ou seja, quais cenas estão disponíveis para os agentes quando estão em um determinado cena e papel; e (iv) Normas: definem o que é permitido, o que é proibido e o que é obrigatório.

Neste contexto, surgiu o *framework* ISLANDER (ESTEVA; CRUZ; SIERRA, 2002), com o objetivo de desenvolvimento de uma infraestrutura para instituições eletrônicas. O foco principal é dado em relação aos aspectos de nível macro (social) dos agentes, sendo neutro em relação à arquitetura ou linguagem utilizada para os agentes participantes.

Em Esteva et al. (2001) é apresentada a formalização de uma linguagem para instituições e normas.

2.5.6 Comparação entre os modelos

A Tabela 2.1 apresenta um resumo da análise dos modelos organizacionais quanto aos conceitos organizacionais e métodos formais utilizados.

2.6 Considerações

Com a crescente utilização da abordagem de agentes para a modelagem de sistemas, estudos em diferentes áreas buscam contribuir para uma melhora neste desenvolvimento. Por um lado, especialistas da área de Engenharia de Software tentam adequar as metodologias, antes orientadas a objetos, ao desenvolvimento de sistemas de agentes ou ainda criar novas metodologias para atender a este paradigma. Por outro lado, na área de SMA, pesquisadores preocupam-se com a definição de modelos organizacionais onde são representados os aspectos sociais dos sistemas multiagentes.

Em ambos os casos, o que percebe-se é a pouca, ou nenhuma, utilização de métodos formais visto que a definição de métodos específicos para sistemas de agentes seria um trabalho um tanto complexo devido às características destes sistemas, o que reforça a proposta de verificação de possibilidade de utilização de métodos formais tradicionais de

Modelo Organizacional	**Conceitos Organizacionais**	**Métodos Formais**
AGR	Papéis e grupos.	Lógica de Primeira Ordem para a especificação de axiomas do modelo.
MOISE$^+$	Papéis, grupos, hierarquias, normas (obrigações e permissões), dimensões: estrutural, funcional e deôntica.	Linguagem semi-formal para especificação das três dimensões.
OPERA	Papéis, cenas, população.	LCR - *Logic for Contract Representation*.
OMNI	Papéis, direitos, dependências entre papéis, normas, ontologias, dimensões: normativa, organizacional e ontológica.	Linguagem para especificação de normas (*ANorms, CNorms*) e Lógica Dinâmica Proposicional para a representação de regras.
ISLANDER	Instituições eletrônicas, papéis, grupos, diálogos, cenas, normas.	Linguagem formal para instituições e normas.

Tabela 2.1: Utilização de Conceitos Organizacionais e Métodos Formais em Modelos Organizacionais de SMA

Engenharia de Software para este contexto.

3 MÉTODOS FORMAIS PARA ENGENHARIA DE SOFTWARE

Os métodos formais consistem na utilização de técnicas matemáticas para auxiliar na documentação, especificação, projeto, análise e certificação de sistemas computacionais, destacando-se em relação à detecção precoce de falhas, que traz grandes benefícios aos desenvolvedores (PRESSMAN, 2002). Desta forma, são importantes em várias etapas do processo de desenvolvimento, permitindo que sejam criadas especificações mais completas, consistentes e não-ambíguas.

Em Hall (1990) e Bowen e Hinckey (1995) são apresentados e discutidos alguns mitos dos métodos formais, mostrando que há muitos aspectos questionáveis em relação ao seu uso. Dentre as discussões apresentadas nestes artigos, cabe salientar as que se referem à i) "*utilização de métodos formais somente para garantir que um programa é perfeito e correto*": Na verdade, mesmo com a utilização de métodos formais é possível que existam fases do projeto onde ocorram imperfeições. As vantagens dos métodos formais vão além da garantia de sucesso, por exemplo, a elaboração da especificação em uma notação auto-validante (notação que permite estabelecer e validar os requisitos) é um dos maiores benefícios da utilização destes métodos; ii) "*aplicação de métodos formais apenas a sistemas críticos"*: Nestes tipos de sistemas a aplicação se justifica pela parte de verificação do método, mas métodos formais podem ser aplicados a qualquer projeto; iii) *"métodos formais não são suportados por ferramentas"*: Atualmente eles são suportados por diversas ferramentas, inclusive de domínio público; iv) *"métodos formais significam abandonar o método de engenharia de software tradicional"*. Na verdade os métodos formais são parte do processo de engenharia de software. O ideal para um bom desenvolvimento é a integração de métodos formais e tradicionais.

3.1 Formalismo no desenvolvimento de software

Não há uma classificação padrão relacionada ao "grau de formalidade" utilizado em um processo de desenvolvimento de software. Usualmente costuma-se considerar apenas dois tipos: *formal* e *informal*. O desenvolvimento é considerado *formal* se as linguagens utilizadas no processo de desenvolvimento (em todas as suas fases) possuem forma (sintaxe) e significado (semântica) bem definidos. Outra característica é que todas as descrições (especificações, projetos ou implementações) do software, em diferentes níveis de abstração, devem ser provadas equivalentes. De forma resumida, pode-se dizer que um desenvolvimento de software é formal se o seu processo de desenvolvimento está formalmente demonstrado. Caso isto não aconteça, o desenvolvimento é considerado *informal*.

Bjørner (1987) apresenta uma classificação mais detalhada dos métodos de desenvol-

vimento de software em relação ao grau de uso de formalismo, dividindo-o em quatro classes: *informal, sistemático, rigoroso e formal*.

- **Informal** (ou *ad hoc*): Nesta abordagem pouca ou nenhuma regra pré-definida é utilizada para guiar o processo de desenvolvimento. Os documentos utilizados no processo são, em geral, informais, utilizando-se linguagem natural ou diagramas, por exemplo. Esta abordagem acarreta consequências negativas quando utilizada em diferentes fases do processo de desenvolvimento, pois as linguagens informais não possuem o rigor necessário à especificação do software e impedem que verificações e validações formais sejam realizadas. Porém o uso destas linguagens continua tendo um lugar importante no início do processo de desenvolvimento como na análise de requisitos, por exemplo, onde a comunicação com o cliente se dá, na maioria das vezes, informalmente. Outra utilização importante de descrições informais é em comentários (documentação) durante o processo de desenvolvimento.

- **Sistemático:** Nesta abordagem existem algumas regras pré-estabelecidas que guiam a transição entre as fases de desenvolvimento (entre os documentos do processo). Em geral, métodos que permitem desenvolvimento sistemático são conhecidos como *semi-formais* e, apesar de ainda não possibilitarem uma verificação formal do processo de desenvolvimento, são mais estritos que os informais. Mesmo que se utilize métodos formais na fase de especificação, é possível adotar um desenvolvimento sistemático, como ocorre em aplicações que utilizam Z e VDM (JONES, 1986).

- **Rigoroso:** Nesta abordagem todas as fases de desenvolvimento usam algum formalismo e existem regras de transição entre os formalismos. Podem ser utilizados teoremas para representar propriedades que não precisam ser formalmente provadas.

- **Formal:** Esta abordagem assemelha-se à anterior, porém exige que todas as provas sobre as propriedades do programa sejam desenvolvidas formalmente. Neste tipo de desenvolvimento garante-se que a implementação é a realização da especificação, ou seja, que a implementação é um modelo para a teoria que equivale à especificação.

O ideal seria que todo o software fosse desenvolvido através da abordagem formal, porém, devido à relação custo/benefício, o que tem acontecido na maioria dos casos é o desenvolvimento sistemático ou uma mistura entre as abordagens. Em alguns sistemas é possível adotar a abordagem formal para algumas partes do sistema, consideradas críticas, e uma abordagem sistemática para partes menos críticas.

3.2 Método x Metodologia

Estes dois conceitos se confundem na área de Engenharia de Software, porém seus significados são diferentes.

Segundo Bjørner (2007), método é o conjunto de princípios para selecionar e aplicar diversas técnicas e ferramentas de análise e síntese (construção) de forma a construir eficientemente um artefato eficiente, no caso o software (ou seja, um sistema computacional).

Metodologia, para Bjørner, é o estudo de, e o conhecimento sobre um ou mais métodos.

3.3 Especificações de Sistemas e Métodos Formais

A utilização de métodos formais na especificação de sistemas deve-se ao fato de que eles permitem descrições concisas e facilmente compreensíveis, além de aumentar a qualidade de hardware e software, garantindo a integridade dos sistemas (bastante usados em sistemas críticos, onde não podem existir erros).

Os métodos formais podem ser usados em diferentes etapas do processo de desenvolvimento de software e, na maioria das vezes, é utilizada mais de uma linguagem de especificação, visto que cada uma delas é adequada para modelar diferentes aspectos do sistema. Atualmente, poucas linguagens permitem modelar o sistema completo.

A Especificação é uma das etapas iniciais no processo de desenvolvimento de um sistema e tem por objetivo especificar, de forma completa e consistente, os requisitos funcionais do sistema. Em geral, ela é escrita em linguagem natural, estando sujeita a ambiguidades, sensibilidade ao contexto e diferentes interpretações. A Especificação Formal consiste na utilização de notações formais, baseadas em técnicas matemáticas e na lógica formal para a especificação de sistemas permitindo, assim, reduzir erros e ambiguidades cometidas durante este processo.

Em Van (2002) são apresentados alguns dos possíveis usos de especificação formal no processo de desenvolvimento de sistemas. São eles:

- para capturar requisitos para o novo software;
- para realizar desenvolvimento de software a partir de requisitos;
- para formalizar algoritmos e provar sua correção;
- para verificar se o software se comporta corretamente em tempo de execução;
- para construir modelos de domínios de aplicação;
- para relatar modelos descritivos e prescritivos;
- para formalizar notações de desenvolvimento de software;
- para gerar casos de teste;
- pode atuar como um “contrato” preciso entre clientes e projetistas de software, a ser usado em discussões e negociações.

Desta forma, a utilização de técnicas de especificação formal no processo de desenvolvimento de sistemas traz diversas vantagens, como:

- permitem que propriedades e consequências da especificação do sistema não executáveis sejam exploradas através da prova de teoremas nos estados iniciais do seu ciclo de vida;
- atuam como um mecanismo de prevenção de falhas e, ainda, permitem detectar falhas no projeto, geradas nos estados iniciais de desenvolvimento do sistema que provocariam graves defeitos;
- fornecem mecanismos para armazenamento das propriedades fundamentais do sistema a ser desenvolvido;

- podem ser submetidas a várias formas de análise mecânica, que são mais efetivas na detecção de certos tipos de falhas;
- podem ser usadas para testar especificações iniciais do projeto, quando normalmente nenhuma outra forma de validação é possível;
- os requisitos destes métodos impõem um grau de verificação mais exato e completo que outros;
- a verificação formal exige a enumeração completa e explícita de todos as suposições envolvidas no projeto;
- a utilização de ferramentas automatizadas pode reduzir o tempo de desenvolvimento do sistema.

3.3.1 Técnicas, Linguagens e Ferramentas de Especificação Formal

Uma técnica formal, segundo Bjørner (2007), é aquela que tem uma base matemática e, consequentemente, pode ser explicada matematicamente. Desta forma, uma técnica formal requer uma linguagem de especificação formal.

As linguagens nas quais expressamos descrições de domínio e prescrições de requisitos são chamadas linguagens de especificação formal e podem ser classificadas em diferentes categorias, conforme o formalismo utilizado.

Existem várias ferramentas de suporte à aplicação de modelos formais, que fornecem, além da Linguagem de Especificação Formal, recursos para a Validação e Verificação Formal. A escolha destas ferramentas para utilização num determinado domínio de aplicação pode ser feita de acordo com as notações e formalismos matemáticos empregados na especificação.

Em Bjørner (2007) são apresentadas duas categorias de linguagens de especificação: baseadas em propriedades e baseadas em modelos.

- **Baseadas em Propriedades:** Nestas linguagens, o sistema é especificado em termos das suas operações e das relações entre estas operações. *Propriedades algébricas* definem as operações através de uma coleção de relações de equivalência (axiomas equacionais). Algumas linguagens que utilizam esta especificação são OBJ3, CafeOBJ, CASL, Larch... *Propriedades axiomáticas* definem operações através de predicados da lógica de primeira ordem, como, por exemplo, a linguagem Larch.

  Especificações baseadas em propriedades são adequadas para especificar tipos de dados abstratos.

- **Baseadas em Modelos:** Nestas linguagens, o sistema é especificado em termos de um modelo de estado, que é construído através da utilização de construções matemáticas como conjuntos, listas, mapeamentos, funções, etc, e de operações sobre esse estado.

  Algumas linguagens utilizadas para este tipo de especificação são VDM-SL, VDM++ [1], Z, Object Z, ... São adequadas para especificar sistemas de informação.

Além destas é considerada ainda a categoria das linguagens **Mistas**, ou seja, baseadas em modelo e propriedade, como é o caso de RSL.

Em Hinckey, Bowen e Rouff (2006), é apresentada uma outra categoria de linguagens:

[1] http://www.vdmportal.org/twiki/bin/view

- **Álgebra de processos**: que tem evoluído para encontrar as necessidades de sistemas concorrentes, distribuídos e de tempo real. Elas descrevem o comportamento destes sistemas através da descrição de suas álgebras de processos de comunicação. Exemplos destas linguagens são CSP (*Communicating Sequential Processes*) e CCS (*Calculus of Communicating Systems*).

A escolha das linguagens de especificação é realizada de acordo com o tipo de dados que se deseja especificar, da mesma maneira que ocorrem as escolhas de linguagens de programação. Cada uma delas tem características específicas em relação à representação de dados, variáveis e comandos, associações e escopo, abstração, encapsulamento, concorrência, etc.

3.4 *Model Checking*

Model checking é uma técnica para a verificação de modelos usada para assegurar a correção de sistemas de hardware e software. Basicamente, ela consiste em verificar quando um modelo satisfaz uma fórmula lógica e pode ser dividida em três etapas: modelagem (especificação do modelo), especificação de propriedades e verificação (se o modelo satisfaz as especificações) (CLARKE; GRUMBERG; PELED, 1999).

Normalmente o modelo é expressado como um sistema de transição e as propriedades são escritas como especificações formais, geralmente usando fórmulas lógicas (CTL*, CTL ou LTL).

Para realizar a verificação automática de modelos, são utilizadas ferramentas de *model checking* tais como SPIN, SMV e SAL. Para tal, é necessário especificar o modelo e as propriedades através das linguagens definidas para cada ferramenta.

3.5 Métodos Formais e Arquiteturas de Software

Arquitetura de software é o nível do projeto de software que trata da estrutura global e propriedades de sistemas. Ela envolve a descrição de elementos a partir dos quais os sistemas são construídos, interações entre estes elementos, padrões que guiam sua composição e restrições nestes padrões (SHAW; GARLAN, 1996).

Normalmente, as estruturas arquiteturais são descritas de maneira informal usando-se diagramas e textos explicativos. Porém, o uso de formalismos pode ser realizado para fornecer modelos precisos, abstratos e técnicas analíticas baseadas nestes modelos. Podem ser usados para fornecer notações para a descrição de projetos de engenharia específicos.

Em Shaw e Garlan (1996) são listados alguns aspectos que podem ser formalizados no contexto de arquitetura de software, tais como:

- **A arquitetura de um sistema específico:** formalismos podem ser parte da especificação do sistema, ampliando as caracterizações informais da arquitetura do sistema e permitindo análises específicas do sistema.

- **Um estilo arquitetural:** um estilo arquitetural define uma família de sistemas em termos de um padrão de organização estrutural. Estes formalismos têm objetivos como: i) tornar precisos expressões comuns, padrões e arquiteturas de referência que normalmente são usadas informalmente; ii) elucidar uma porção do espaço arquitetural, mostrando como diferentes arquiteturas podem ser tratadas como especializações de uma arquitetura comum.

- **Uma teoria de arquitetura de software:** formalismos deste tipo podem esclarecer o significado de conceitos arquiteturais genéricos, como por exemplo, conexão arquitetural, representação arquitetural hierárquica e estilo arquitetural. Idealmente, estes formalismos devem fornecer uma base dedutiva para análise de sistemas em um nível arquitetural (e.g. pode fornecer regras para determinar quando uma descrição arquitetural está bem formada).

- **Semântica formal para linguagens de descrição arquitetural:** este tipo de formalismo trata a descrição arquitetural como uma questão de linguagem e aplica técnicas tradicionais para a representação de semânticas de linguagens.

3.6 Métodos Formais para Sistemas de Agentes

Quanto mais complexo se torna o software, mais difícil é testá-lo e encontrar erros. No caso de sistemas de agentes este é um problema típico. Erros nesses sistemas raramente podem ser encontrados colocando dados de entrada no sistema e verificando se os resultados estão corretos, pois erros deste tipo são baseados em tempo e podem ocorrer apenas quando o processo envia ou recebe dados em um determinado tempo ou sequência, conforme descrito em Hinckey, Bowen e Rouff (2006).

Para encontrar estes erros, os processos de software envolvidos têm que ser executados em todas as combinações possíveis de estados (espaço de estados) que o processo poderia estar envolvido. Como o espaço de estados é exponencial em relação ao número de estados, ele cresce extremamente rápido com o número de estados no processo e torna-se "não-testável" com um número relativamente pequeno de processos. Uma das soluções que tem sido dada a este problema, chamado de explosão de estados, é o uso de técnicas para a redução de estados e utilização de modelos.

Devido à complexidade envolvida nestes sistemas o uso de técnicas formais auxilia no processo de desenvolvimento trazendo vantagens significativas, principalmente no que se refere à utilização de ferramentas automatizadas para as verificações necessárias.

A seguir, são descritos alguns dos trabalhos que utilizam métodos formais para sistemas baseados em agentes.

3.6.1 Trabalhos Relacionados

Em Luck e d'Inverno (1995), é utilizada a linguagem Z para especificar um *framework* de agentes. Neste *framework* foi especificada uma hierarquia de 4 camadas que consiste de entidades, objetos, agentes e agentes autônomos. Como parte da especificação também encontram-se compromissos, cooperação, relacionamentos interagentes, agentes sociológicos, planos e objetivos.

O trabalho descrito em Wooldridge et al. (2002), apresenta MABLE, uma linguagem para o projeto e verificação automática de sistemas multiagentes. Ela é uma linguagem de programação imperativa, enriquecida por construções do paradigma de programação orientada a agentes. Os agentes em um sistema MABLE têm um estado mental, consistindo de crenças, desejos e intenções e se comunicam através de performativas KQML. MABLE é automaticamente traduzida em Promela (linguagem de entrada do verificador de modelos Spin). Os requisitos do sistema são expressos através de uma linguagem baseada em LORA (Logic of Rational Agents), que são traduzidos em fórmulas LTL, utilizadas no Spin.

Outro trabalho relacionado, pode ser encontrado em Bordini et al. (2003) e Bordini et al. (2004), onde é apresentada uma proposta de verificação de modelos de agentes racionais. Neste trabalho, os autores também propõem a utilização de técnicas de verificação de modelos para verificar automaticamente sistemas multiagentes, porém através do uso de uma linguagem de programação lógica inspirada em sistemas de planejamento reativo. A modelagem dos agentes é feita utilizando-se AgentSpeak(F), uma versão de estados finitos de AgentSpeak(L). Consequentemente, as propriedades verificadas são escritas em lógica BDI, que pode facilmente ser traduzida para LTL (a linguagem de especificação de vários verificadores de modelos). Os autores utilizam um conjunto de ferramentas, chamado CASP(*Checking AgentSpeak Programs*), para fazer a tradução de um agente AgentSpeak em Promela e Java (as linguagens de entrada dos verificadores Spin e Java PathFinder, respectivamente).

Vários tipos de lógicas têm sido utilizados para diferentes aplicações, tais como a lógica proposicional para representar informações factuais (no caso de agentes pode ser utilizada para a base de conhecimento ou ambiente do agente); as lógicas modais, utilizadas para representar diferentes modos de verdade, tais como possivelmente verdade e necessariamente verdade; a lógica deôntica que descreve o que é obrigatório que seja feito; a lógica dinâmica que assemelha-se à modal, porém é baseada em ações; e a lógica temporal que representa informações de tempo. Para dar uma semântica formal à arquitetura BDI, foi desenvolvida a lógica BDI. Através desta lógica arquiteturas BDI podem ser formalmente modeladas e provas de correção em agentes BDI podem ser realizadas. Esta lógica é baseada na lógica modal e descreve crenças, desejos, intenções e planos que um agente pode seguir para atingir suas intenções (WOOLDRIDGE, 2000).

UML (*Unified Modeling Language*) é uma linguagem de especificação, visualização e documentação de modelos de sistemas de software. Ela também tem sido utilizada para a modelagem de agentes através de sua extensão AUML (*Agent UML*), cujo padrão está sendo trabalhado pela FIPA (*Foundation for Intelligent Physical Agents*) introduzindo diagramas para a especificação do comportamento interno dos agentes, ambiente externo e diagramas de interação. Uma das principais mudanças da AUML é a adição de semântica que reflete aspectos relacionados à autonomia, estruturas sociais e comunicação assíncrona de agentes. Os trabalhos recentes de utilização da AUML podem ser encontrados em http://www.auml.org.

Um uso da linguagem RSL para a especificação de sistemas multiagentes pode ser observado em Patra e Moore (2000). Neste trabalho os autores descrevem a especificação de um sistema de mercado eletrônico (*E-Market*), detalhando os agentes envolvidos e as atividades de cada um deles.

Em Barbosa et al. (2010) é descrito o uso de CSP na especificação formal do nível micro-organizacional de sistemas multiagentes, onde a linguagem CSP é utilizada para especificar comportamentos e processos de troca realizados entre papéis organizacionais. O artigo apresenta ainda a utilização do verificador de modelos FDR para a prova de propriedades de *deadlock*, *livelock* e não-determinismo através de refinamentos.

3.7 Considerações

Este capítulo apresentou alguns conceitos importantes da área de métodos formais necessários ao entendimento da importância de sua utilização durante o processo de de-

senvolvimento de software.

Dentre os trabalhos relacionados utilizando métodos formais para a área de agentes, somente um deles utiliza a linguagem RSL e, além do último trabalho descrito (realizado pela autora), nenhum deles aborda os aspectos organizacionais de SMAs.

4 O MODELO POPORG

Neste capítulo é apresentado o modelo PopOrg, escolhido como objeto de estudo desta tese. Sua escolha justifica-se por tratar-se de um modelo mínimo de organização de sistemas multiagentes, concebido para representar o conjunto mínimo de aspectos estruturais e operacionais que estas organizações devem ter.

O modelo População-Organização de um sistema foi introduzido em Demazeau e Costa (1996), como base para um modelo formal de sistemas multiagentes com organizações dinâmicas. Este modelo analisa dois aspectos importantes dos sistemas multiagentes: sua população e sua organização. A população de um SMA consiste do conjunto de agentes que o habitam, juntamente com o conjunto de todos os comportamentos que eles são capazes de executar e o conjunto de todos os processos de interação que eles podem estabelecer entre si. Uma organização é composta por papéis organizacionais e links organizacionais, onde um papel organizacional que um agente pode ter em um sistema é definido em relação a algum processo global no qual o agente participa (processos sociais, ou interação) e os links organizacionais são as influências mútuas entre os agentes que participam em um determinado processo global.

O modelo PopOrg de um sistema multiagente é uma estrutura $PopOrg = (Pop, Org, imp)$ constituída pelas estruturas populacional e organizacional, juntamente com uma relação de implementação estabelecida entre elas.

O modelo PopOrg é baseado na distinção entre as noções de descrições intensionais e extencionais de sistemas (COSTA; DIMURO, 2007a). Descrições intensionais estão relacionadas a aspectos subjetivos, pertencendo ao funcionamento interno dos agentes que operam no sistema modelado (como crenças, objetivos, normas, valores, etc.), enquanto descrições extensionais referem-se a aspectos objetivos, pertencendo ao funcionamento externo dos agentes (como ações executadas, objetos trocados, etc.). O modelo PopOrg concentra-se na representação de aspectos externos do sistema, e considera os aspectos intensionais como estrutura adicional, opcional, que pode ser superimposta na extensional. Desta forma, este modelo é considerado um modelo mínimo de organização de sistemas multiagentes, pois ele representa apenas os componentes-chave de uma organização, permitindo que outros modelos mais complexos possam ser construídos (de maneira modular) em cima dele.

Em Costa e Dimuro (2007a), este modelo foi estendido de forma a contemplar níveis micro-organizacionais e macro-organizacionais (cf. Figura 4.1). O nível micro-organizacional é onde ocorrem as interações organizacionais entre papéis individualizados, enquanto o nível macro-organizacional, é o nível onde unidades organizacionais (grupos de papéis) são introduzidas para permitir a estruturação da organização hierarquicamente, com as interações acontecendo entre estas unidades organizacionais. Um par de relações de implementação define a forma na qual as unidades organizacionais são

implementadas por conjuntos de papéis organizacionais e a forma na qual papéis organizacionais são implementados por agentes individuais da população.

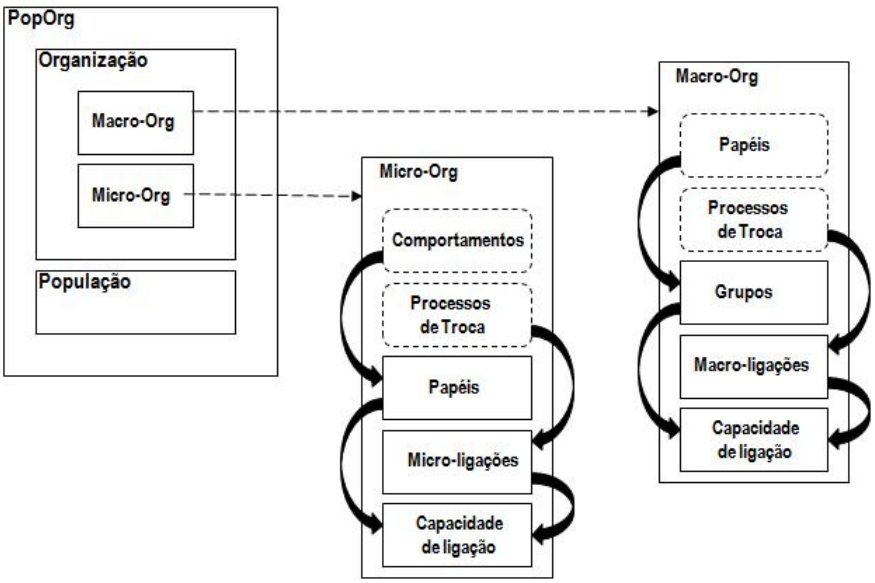

Figura 4.1: Modelo PopOrg

4.1 Estrutura Populacional

A estrutura populacional de um sistema multiagente (considerando T a estrutura do tempo em que este sistema está sendo analisado) é uma tupla $Pop = (Ag, Act, Bh, Ep\,; bc, ec)$, onde:

- Ag é o conjunto de agentes chamado de *população*;
- Act é o conjunto de todas as *ações* (ações de comunicação e ações sobre objetos concretos) que podem ser realizadas pelos agentes do sistema;
- $Bh \subseteq [T \rightarrow \wp(Act)]$ é o conjunto de todos os *comportamentos* (sequências de ações que os agentes são capazes de realizar);
- $Ep \subseteq [T \rightarrow \wp(Act) \times \wp(Act)]$ é o conjunto de todos os *processos de troca* que quaisquer dois agentes podem realizar em conjunto e/ou entrelaçando apropriadamente suas ações;
- $bc : Ag \rightarrow \wp(Bh)$ é uma função de *capacidade comportamental* tal que, para cada agente $a \in Ag, bc(a)$ é o conjunto de comportamentos que o agente é capaz de realizar;
- $ec : Ag \times Ag \rightarrow \wp(Ep)$ é uma função de *capacidade de troca* tal que, para cada $a_1, a_2 \in Ag, ec(a_1, a_2)$ é o conjunto de processos de troca que os agentes a_1 e a_2 podem realizar entre si.
- $\forall\, a_1, a_2 \in Ag\, \forall\, e \in ec(a_1, a_2)\, \forall\, t \in T$:

$$Prj_1(e(t)) \subseteq \bigcup b(t) | b \in bc(a_1) \wedge Prj_2(e(t)) \subseteq \bigcup b(t) | b \in bc(a_2),$$

onde Prj_1 e Prj_2 são *funções de projeção* de modo que as capacidades de troca dos pares de agentes são restringidas pelas suas capacidades de comportamento conjunto.

4.2 Estrutura Organizacional

A estrutura organizacional de um sistema multiagente, definida em Costa e Dimuro (2007b) é uma tupla $Org = (Org_\omega, Org_\Omega)$, onde Org_ω é a estrutura *micro-organizacional* e Org_Ω é a estrutura *macro-organizacional*.

A estrutura *micro-organizacional* é definida por $Org_\omega = (R_\omega, L_\omega, lc_\omega)$, onde:

- $R_\omega \subseteq \wp(Bh)$, é o *conjunto de papéis* que os agentes podem desempenhar no sistema multiagente;
- $L_\omega \subseteq R_\omega \times R_\omega \times E_p$ é o *conjunto de micro-ligações* que podem ser estabelecidas entre papéis, cada micro-ligação especificando um processo de troca que os agentes que desempenham os papéis ligados pela micro-ligação podem ter de realizar;
- $lc_\omega : R_\omega \times R_\omega \to \wp(L_\omega)$ é a *capacidade de micro-ligação* dos pares de papéis, ou seja, o conjunto de micro-ligações que cada par de papéis pode estabelecer entre si;
- $\forall\, l \in L_\omega \,\exists\, r_1, r_2 \in R_\omega : l \in lc_\omega(r_1, r_2)$, isto é, cada ligação possível entre dois papéis tem de estar na capacidade de micro-ligação dos papéis que ela liga.

A estrutura *macro-organizacional* é definida por $Org_\Omega = (\mathbb{G}_\Omega, \mathbb{L}_\Omega)$, onde:

- $\mathbb{G}_\Omega \subseteq \wp(R_\omega) \times \wp(L_\omega)$ é o *conjunto de grupos sociais* do sistema multiagente, cada grupo social $\mathbb{G} = (\mathbb{R}, \mathbb{L}) \in \mathbb{G}_\Omega$ sendo não-vazio e fechado para a sua estrutura micro-organizacional, isto é:

$$\forall\, l \in \mathbb{L} : l = (r_0, r_1, e) \Rightarrow r_0, r_1 \in \mathbb{R}$$

- $\mathbb{L}_\Omega \subseteq \mathbb{G}_\Omega \times \mathbb{G}_\Omega \times \wp(L_\omega)$ é o *conjunto de macro-ligações* que podem ser estabelecidas entre os grupos sociais da estrutura macro-organizacional, cada macro-ligação $(\mathbb{G}_0, \mathbb{G}_1, L_{\mathbb{G}_0,\mathbb{G}_1}) \in \mathbb{L}_\Omega$ respeitando a estrutura micro-organizacional dos grupos sociais que a constituem, isto é
$\forall\, lc \in L_{\mathbb{G}_0,\mathbb{G}_1}$:

$$lc = (r_0, r_1, e) \Rightarrow \exists\, G_0 \in \mathbb{G}_0 \,\exists\, G_1 \in \mathbb{G}_1 : r_0 \in \mathbb{G}_0 \wedge r_1 \in \mathbb{G}_1 \wedge lc \in L_\omega(r_0, r_1)$$

4.3 A relação de implementação

Com a divisão da estrutura organizacional de um sistema nas duas sub-estruturas micro e macro-organizacional, a relação de implementação passa a ser constituída por um par de relações $imp = (imp_\omega, imp_\Omega)$, onde

- imp_ω é a *relação de implementação da estrutura micro-organizacional*, definindo como a estrutura micro-organizacional Org_ω é implementada pela população Pop;
- imp_Ω é a *relação de implementação da estrutura macro-organizacional*, definindo como a estrutura macro-organizacional Org_Ω é implementada pela estrutura micro-organizacional Org_ω.

4.4 Relacionamentos entre os componentes PopOrg

Para um melhor entendimento sobre os componentes do modelo PopOrg e seus relacionamentos, a Figura 4.2 descreve estes elementos, destacando os tipos envolvidos no modelo e suas instâncias.

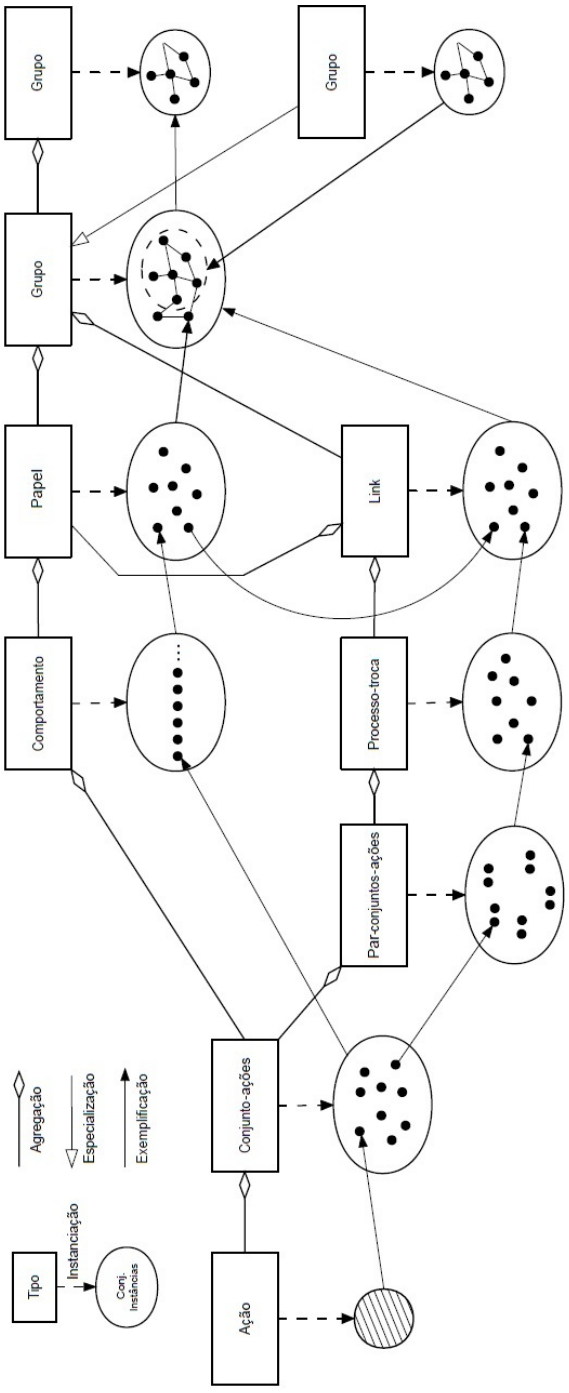

Figura 4.2: Relacionamentos entre componentes PopOrg

Observa-se por exemplo, que um comportamento é formado por um conjunto de ações, um papel é um conjunto de comportamentos, etc. O tipo grupo apresenta 2 relacionamentos possíveis, pois um grupo pode fazer parte de outro (agregação) ou ser um caso particular de um outro grupo (especialização).

A compreensão destes relacionamentos torna-se necessária para o acompanhamento da especificação realizada na Seção 7.1.

4.5 Considerações

O modelo PopOrg, escolhido como base para o trabalho desenvolvido nesta tese, abrange o conjunto mínimo de elementos estruturais e operacionais que organizações de SMAs devem ter.

Neste modelo são representados os aspectos externos do sistema, considerando os aspectos intensionais (relacionados a aspectos subjetivos) como uma estrutura adicional. Desta forma outros modelos mais complexos de organizações podem ser modelados através dele.

5 TRABALHO PRELIMINAR PARA A TESE

Este capítulo descreve um estudo preliminar realizado utilizando-se a linguagem CSP para a representação de comportamentos e processos de troca PopOrg.

5.1 A linguagem CSP

CSP *(Communicating Sequential Processes)* é uma notação para a descrição de sistemas concorrentes nos quais os processos componentes interagem através de comunicação (ROSCOE, 1998).

O modelo conceitual usado por CSP considera componentes, ou processos, como entidades independentes (autônomas) com interfaces particulares através das quais eles interagem com seu ambiente. A interface de um processo é descrita como um conjunto de eventos, cada um deles descrevendo um tipo particular de ação atômica que pode ser executada ou sofrida pelo processo. Esta interface pode ser relacionada à especificação estática de um processo, enquanto sua especificação dinâmica descreve como ele irá se comportar nesta interface.

CSP tem um sistema de provas associado para a verificação de propriedades corretas ou incorretas. Esta verificação de propriedades é baseada em traces de eventos que podem ser produzidos por uma especificação.

Como linguagem formal, CSP pode ser entendida através das semânticas: operacional, denotacional e algébrica (ROSCOE, 1998). No estilo denotacional, CSP pode ser interpretada em três níveis de detalhes: *traces*, falhas e falhas-divergências. Nesta proposta é utilizado o modelo de *traces* (que mostra a história dos eventos de cada processo em um dado sistema) para relacionar CSP com as estruturas organizacionais de sistemas multiagentes baseados em PopOrg. Mais especificamente, a linguagem CSP é utilizada para representar comportamentos entre papéis no nível micro-organizacional e processos de troca realizados entre estes papéis.

5.2 A relação entre *Traces* CSP e Comportamentos PopOrg

Para fazer uso significativo da linguagem CSP como formalismo para a especificação de papéis e processos de troca da estrutura micro-organizacional de modelos PopOrg, deve-se ter certeza de que as propriedades dos programas CSP escritas como uma especificação organizacional são preservadas quando transformadas em implementações PopOrg.

Para se ter essa garantia, observa-se como é possível tratar o modelo comportamental PopOrg como um espalhamento temporal do modelo de traces de CSP, que não afeta a relação de ordem entre os eventos dos traces. Isto é suficiente para estabelecer que

qualquer propriedade de segurança (*safety*) provada verdadeira na especificação CSP é também verdadeira para qualquer implementação PopOrg daquela especificação.

A figura 5.1 apresenta a relação entre a linguagem CSP e o modelo PopOrg. As setas curvas mostram que papéis e links são compostos de comportamentos e processos de troca e estão incluídos nas capacidades de ligações.

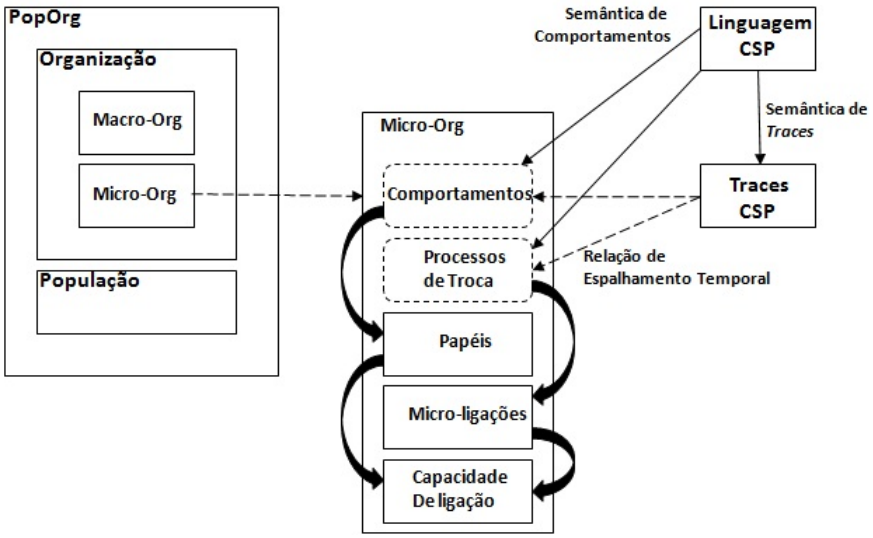

Figura 5.1: Conexão entre CSP e PopOrg

5.3 Comportamentos e Processos de Troca PopOrg

Assume-se T, uma estrutura de tempo linear discreta e A, o universo finito de ações a partir das quais são definidos os comportamentos organizacionais.

Um comportamento de um papel organizacional PopOrg é uma sequência indexada no tempo de subconjuntos de ações $b : T \rightarrow \wp(A)$ da forma $b = \{0 \rightarrow \alpha, 1 \rightarrow \beta, ...\}$, cada subconjunto $\alpha, \beta, ... \in \wp(A)$ indicando um possível conjunto de ações que o papel organizacional que executa o comportamento b pode executar no tempo $t \in T$. O conjunto de todos os comportamentos organizacionais é denotado por $Bh = [T \rightarrow \wp(A)]$.

Um processo de troca PopOrg, entre dois papéis organizacionais, é uma sequência, indexada no tempo, de pares de subconjuntos de ações, $e : T \rightarrow \wp(A) \times \wp(A)$, cada par de subconjuntos $e(t) \in \wp(A) \times \wp(A)$ indicando o conjunto de ações que cada papel organizacional pode executar no tempo t, quando eles estão interagindo. O conjunto de todos os processos de troca organizacionais é denotado por $Ep = [T \rightarrow \wp(A) \times \wp(A)]$.

Assim, por exemplo, assumindo que $A = \{a, b, c, ...\}$ e que $T = (0, 1, 2, ...)$, tem-se que $b = \{0 \rightarrow \{a\}, 1 \rightarrow \emptyset, 2 \rightarrow \{b, c\}, 3 \rightarrow \{b\}, 4 \rightarrow \emptyset, 5 \rightarrow \{a, c\}, ...\}$ pode ser um comportamento PopOrg que escolhe aleatoriamente executar no instante zero, algumas ações obtidas do subconjunto $\{a, b, c\}$ e que $e = \{0 \rightarrow (\{a\}, \emptyset), 1 \rightarrow (\emptyset, \{b, c\}), 2 \rightarrow (\{a\}, \emptyset), 3 \rightarrow (\emptyset, \{b, c\}), ...\}$ pode ser um processo de troca onde os dois papéis organizacionais envolvidos alternam suas ações, o primeiro papel sempre executando a ação a quando ele age e o segundo executando simultaneamente as ações b e c quando age.

5.3.1 O subconjunto de CSP usado para especificar comportamentos organizacionais

Para esta análise foi considerado apenas um subconjunto dos construtores da linguagem CSP (HOARE, 1985) (ROSCOE, 1998). O único programa primitivo é o programa de *deadlock* STOP. Para introduzir sequencialidade em programas CSP, usa-se o construtor de prefixo $\rightarrow$. Para introduzir não-determinismo e concorrência entre dois ou mais processos, são utilizados os operadores de escolha externa ($\Box$) e entrelaçamento ($|||$).

No modelo PopOrg, os comportamentos organizacionais são sequências de conjuntos de ações que papéis organizacionais podem realizar. Desta forma, especificações CSP devem ter o alfabeto $\sum = \wp(A)$. Neste texto, usa-se o construtor de prefixo na forma $\alpha \rightarrow P$, com $\alpha \in \wp(A)$. Entretanto, por conveniência, permite-se que $\{a\} \rightarrow P$ seja denotado por $a \rightarrow P$.

5.3.2 A semântica de Comportamentos de CSP

A semântica PopOrg de CSP é uma semântica de *timed traces*. Entretanto, ela difere da semântica de *timed traces* de *Timed* CSP (DAVIES; SCHNEIDER, 1995) , pois *timed traces* PopOrg são completos, no sentido em que eles denotam explicitamente todos os conjuntos de ações que são possíveis em cada instante de tempo, enquanto os *traces* de *Timed* CSP apenas denotam explicitamente os tempos em que as ações ocorreram.

Por outro lado, quando define-se o conjunto de comportamentos PopOrg que correspondem a um programa CSP, sabe-se que o sequenciamento abstrato de ações prescritas pelo programa CSP não é traduzido em um sequenciamento temporal rígido. Ou seja, garante-se que um número arbitrário de eventos ociosos pode ser inserido entre quaisquer dois eventos concretos, tal que qualquer expressão seja interpretada como um conjunto infinito de comportamentos PopOrg.

Desta forma, a semântica dos programas CSP que modelam comportamentos PopOrg é dada pela função $bhs^t : CSP \mapsto \wp(Bh)$, que modela um programa CSP como um conjunto de comportamentos organizacionais PopOrg, partindo do tempo 0.

Para definir bhs^t, são adotadas as seguintes notações:

- para qualquer comportamento $b = \{t \mapsto \alpha, t+1 \mapsto \beta, ...\}$, o deslocamento de t por k unidades de tempo é definido por:
  $b^k = \{t+k \mapsto \alpha, t+k+1 \mapsto \beta, t+k+2 \mapsto \gamma, ...\}$, para todo k tal que $t+k \geq 0$.

- para qualquer evento comportamental $\{t \mapsto \alpha\}$, a exponenciação $\{t \mapsto \alpha\}^n$ do evento é dada por:
  $\{t \mapsto \alpha\}^0 = \emptyset$
  $\{t \mapsto \alpha\}^n = \{t \mapsto \alpha\} \cup \{t+1 \mapsto \alpha\} \cup ... \cup \{t+n-1 \mapsto \alpha\} =$
  $\{t \mapsto \alpha, t+1 \mapsto \alpha, ..., t+n \mapsto \alpha\}$, se $n \geq 1$

- para quaisquer comportamentos b_1 e b_2, iniciando no tempo t, seu entrelaçamento $b_1 ||| b_2$ é definido por:
  $\langle\rangle ||| b_2 = b_2$
  $b_1 ||| \langle\rangle = b_1$
  $\{t \mapsto \alpha\} \cup b'_1 ||| \{t \mapsto \beta\} \cup b'_2 =$
  $\{\{t \mapsto \alpha\} \cup b | b \in (b'_1 ||| \{t+1 \mapsto \beta\} \cup b_2'^{(+1)})\} \cup$
  $\{\{t \mapsto \beta\} \cup b | b \in (\{t+1 \mapsto \alpha\} \cup b_1'^{(+1)} ||| b'_2)\}$

A função semântica bhs^t é definida por:

$bhs^t(STOP) = \{\{t \mapsto \emptyset, t+1 \mapsto \emptyset, t+2 \mapsto \emptyset, ...\}\}$
$bhs^t(\alpha \to P) = \{\{t \mapsto \emptyset\}^n \cup \{t+n \mapsto \alpha\} \cup b | b \in bhs^{t+n+1}(P), n \geq 0\}$
$bhs^t(\alpha \to P \,\Box\, \beta \to Q) =$
$\quad \{\{t \mapsto \emptyset\}^n \cup \{t+n \mapsto \alpha\} \cup b | b \in bhs^{t+n}(P \,\Box\, \beta \to Q), n \geq 0\} \cup$
$\quad \{\{t \mapsto \emptyset\}^n \cup \{t+n \mapsto \beta\} \cup b | b \in bhs^{t+n}(\alpha \to P \,\Box\, Q), n \geq 0\}$
$bhs^t(P \,|||\, Q) = \bigcup\{b_P \,|||\, b_Q | b_P \in bhs^t(P), b_Q \in bhs^t(Q)\}$

Denota-se $bhs^0(P)$ simplesmente por $bhs(P)$. Alguns exemplos são apresentados a seguir:

$bhs(a \to STOP) = \{$
$\quad \{0 \mapsto \{a\}, 1 \mapsto \emptyset, 2 \mapsto \emptyset, ...\},$
$\quad \{0 \mapsto \emptyset, 1 \mapsto \{a\}, 2 \mapsto \emptyset, 3 \mapsto \emptyset, ...\},$
$\quad \{0 \mapsto \emptyset, 1 \mapsto \emptyset, 2 \mapsto \{a\}, 3 \mapsto \emptyset, 4 \mapsto \emptyset, ...\},$
$\quad ...$
$\quad \}$

$bhs(\{a, b\} \to c \to STOP) = \{$
$\quad \{0 \mapsto \{a, b\}, 1 \mapsto \{c\}, 2 \mapsto \emptyset, 3 \mapsto \emptyset, ...\},$
$\quad \{0 \mapsto \{a, b\}, 1 \mapsto \emptyset, 2 \mapsto \{c\}, 3 \mapsto \emptyset, 4 \mapsto \emptyset, ...\},$
$\quad \{0 \mapsto \{a, b\}, 1 \mapsto \emptyset, 2 \mapsto \emptyset, 3 \mapsto \{c\}, 4 \mapsto \emptyset, 5 \mapsto \emptyset ...\},$
$\quad ...,$
$\quad \{0 \mapsto \emptyset, 1 \mapsto \{a, b\}, 2 \mapsto \{c\}, 3 \mapsto \emptyset, 4 \mapsto \emptyset, ...\},$
$\quad \{0 \mapsto \emptyset, 1 \mapsto \{a, b\}, 2 \mapsto \emptyset, 3 \mapsto \{c\}, 4 \mapsto \emptyset, 5 \mapsto \emptyset, ...\},$
$\quad ...,$
$\quad \{0 \mapsto \emptyset, 1 \mapsto \emptyset, 2 \mapsto \{a, b\}, 3 \mapsto \{c\}, 4 \mapsto \emptyset, 5 \mapsto \emptyset ...\},$
$\quad \{0 \mapsto \emptyset, 1 \mapsto \emptyset, 2 \mapsto \{a, b\}, 3 \mapsto \emptyset, 4 \mapsto \{c\}, 5 \mapsto \emptyset, ...\},$
$\quad ...,$
$\quad \}$

$bhs(a \to STOP \,\Box\, b \to STOP) = \{$
$\quad \{0 \mapsto \{a\}, 1 \mapsto \emptyset, 2 \mapsto \emptyset, ...\},$
$\quad \{0 \mapsto \{b\}, 1 \mapsto \emptyset, 2 \mapsto \emptyset, ...\},$
$\quad \}$

$bhs(a \to STOP \,|||\, b \to STOP) = \{$
$\quad \{0 \mapsto \{a\}, 1 \mapsto \emptyset, 2 \mapsto \emptyset, ...\},$
$\quad \{0 \mapsto \{b\}, 1 \mapsto \emptyset, 2 \mapsto \emptyset, ...\},$
$\quad \{0 \mapsto \{a\}, 1 \mapsto \{b\}, 2 \mapsto \emptyset, ...\},$
$\quad \{0 \mapsto \{b\}, 1 \mapsto \{a\}, 2 \mapsto \emptyset, ...\},$
$\quad \{0 \mapsto \{a\}, 1 \mapsto \emptyset, 2 \mapsto \{b\}, ...\},$
$\quad \{0 \mapsto \{b\}, 1 \mapsto \emptyset, 2 \mapsto \{a\}, ...\},$
$\quad ...,$
$\quad \{0 \mapsto \emptyset, 1 \mapsto \{a\}, 2 \mapsto \emptyset, ...\},$
$\quad \{0 \mapsto \emptyset, 1 \mapsto \{b\}, 2 \mapsto \emptyset, ...\},$
$\quad \{0 \mapsto \emptyset, 1 \mapsto \{a\}, 2 \mapsto \{b\}, ...\},$
$\quad \{0 \mapsto \emptyset, 1 \mapsto \{b\}, 2 \mapsto \{a\}, ...\},$
$\quad \{0 \mapsto \emptyset, 1 \mapsto \{a\}, 2 \mapsto \emptyset, 3 \mapsto \{b\}, ...\}],$

$\{0 \mapsto \emptyset, 1 \mapsto \{b\}, 2 \mapsto \emptyset, 3 \mapsto \{a\}, ...\},$
$...,$
$\}$

A notação [.] é utilizada para o conjunto de todas as diferentes associações temporais de um dado comportamento, tal que $bhs(\{a, b\} \rightarrow c \rightarrow STOP)$ é denotado simplesmente por:

$\{[0 \mapsto \{a, b\}, 1 \mapsto \{c\}]\}$

E, sendo $P = a \rightarrow P$ e $Q = b \rightarrow Q$, tem-se:

$bhs(P) = \{$
$[0 \mapsto \{a\}, 1 \mapsto \{a\}, 2 \mapsto \{a\}, ...],$
$\}$

$bhs(Q) = \{$
$[0 \mapsto \{b\}, 1 \mapsto \{b\}, 2 \mapsto \{b\}, ...],$
$\}$

$bhs(P\,;Q) = \{$
$[0 \mapsto \{a\}, 1 \mapsto \{a\}, 2 \mapsto \{a\}, ...],$
$\}$

$bhs(P \,\square\, Q) = \{$
$[0 \mapsto \{a\}, 1 \mapsto \{a\}, 2 \mapsto \{a\}, ...],$
$[0 \mapsto \{b\}, 1 \mapsto \{b\}, 2 \mapsto \{b\}, ...],$
$\}$

$bhs(P \,|||\, Q) = \{$
$[0 \mapsto \{a\}, 1 \mapsto \{a\}, 2 \mapsto \{a\}, ...],$
$[0 \mapsto \{b\}, 1 \mapsto \{a\}, 2 \mapsto \{a\}, ...],$
$[0 \mapsto \{a, b\}, 1 \mapsto \{a\}, 2 \mapsto \{a\}, ...],$
$[0 \mapsto \{a\}, 1 \mapsto \{b\}, 2 \mapsto \{a\}, ...],$
$[0 \mapsto \{a\}, 1 \mapsto \{a, b\}, 2 \mapsto \{a\}, ...],$
$[0 \mapsto \{b\}, 1 \mapsto \{b\}, 2 \mapsto \{a\}, ...],$
$[0 \mapsto \{a, b\}, 1 \mapsto \{a, b\}, 2 \mapsto \{a\}, ...],$
$...,$
$[0 \mapsto \{b\}, 1 \mapsto \{b\}, 2 \mapsto \{b\}, ...],$
$[0 \mapsto \{a\}, 1 \mapsto \{b\}, 2 \mapsto \{b\}, ...],$
$[0 \mapsto \{a, b\}, 1 \mapsto \{b\}, 2 \mapsto \{b\}, ...],$
$[0 \mapsto \{b\}, 1 \mapsto \{a\}, 2 \mapsto \{b\}, ...],$
$[0 \mapsto \{b\}, 1 \mapsto \{a, b\}, 2 \mapsto \{b\}, ...],$
$[0 \mapsto \{a\}, 1 \mapsto \{a\}, 2 \mapsto \{b\}, ...],$
$[0 \mapsto \{a, b\}, 1 \mapsto \{a, b\}, 2 \mapsto \{b\}, ...],$
$...,$
$[0 \mapsto \{a, b\}, 1 \mapsto \{a, b\}, 2 \mapsto \{a, b\}, ...],$
$\}$

Claramente, para cada programa CSP existe um mapeamento do seu conjunto de comportamentos PopOrg para seu conjunto de *traces*, abstraindo-se a vinculação temporal dos eventos de comportamento: para cada comportamento do programa CSP, da forma $\{0 \mapsto \alpha_0, 1 \mapsto \alpha_1, 2 \mapsto \alpha_2, ...\}$, existe um *trace* daquele programa, da forma $\langle \alpha_{i0}, \alpha_{i1}, \alpha_{i2}, ... \rangle$, tal que todos os eventos α_k no *trace* são não-vazios, $\alpha_k \neq \emptyset$, e o *trace* é tal que, para quaisquer dois de seus eventos α_j e α_k, se k<j, então os eventos de comportamento $t_k \mapsto \alpha_k$ e $t_j \mapsto \alpha_j$ estão no comportamento e $t_k < t_j$.

Nota-se, então, que a semântica PopOrg de CSP dá uma semântica de programa que preserva todas as propriedades da semântica de traces, dependentes da relação de ordem entre os eventos, porque envolve apenas o espalhamento dos eventos no tempo. Assim, todas as propriedades de programas CSP provadas usando a semântica de traces, são herdadas pela interpretação PopOrg daquele programa, mostrando que programas CSP podem ser usados para a especificação de partes operacionais (comportamentos de papéis e processos de troca) do modelo micro-organizacional PopOrg.

5.3.3 A semântica de Processos de Troca de CSP

Da mesma forma que os papéis foram definidos como um processo CSP, agora define-se os processos de troca.

Um processo de troca é uma sequência, indexada no tempo, de pares de subconjuntos de ações $e : T \to \wp(A) \times \wp(A)$, e o conjunto de processos de troca é denotado por Ep (conforme definido na seção 5.3).

Um link especifica um conjunto de processos de troca que os agentes, que desempenham os papéis ligados pelo link, podem ter de realizar. Desta forma:

Um link é $l = (r_1, r_2, E)$ onde $E \subseteq Ep$.

$E = \{e_1, e_2, \ldots\}$

$e_1 = [0 \mapsto (\{a\}, \{b\}), 1 \mapsto (\{a\}, \{b, c\}), 2 \mapsto (\{b, c\}, \emptyset), 3 \mapsto (\{a\}, \{b\}), \ldots]$

A semântica de um programa CSP que especifica um conjunto de processos de troca PopOrg é dada pela função $eps^t : CSP \to \wp(Ep)$, a qual modela o programa CSP como um conjunto de processos de troca, todos iniciando no tempo t. Os processos CSP que determinam os processos de troca são da forma $P = (\alpha, \beta) \to (\gamma, \delta) \to ...$, sendo o seu alfabeto representado por pares de conjuntos de ações $\Sigma \subseteq \wp(A) \times \wp(A)$, cada conjunto executado por um papel.

$eps^t(STOP) = \{[t \mapsto (\emptyset, \emptyset), t+1 \mapsto (\emptyset, \emptyset), t+2 \mapsto (\emptyset, \emptyset), ...]\}$
$eps^t(P) = \{[t \mapsto (\alpha, \beta), t+1 \mapsto (\gamma, \delta), ...]\}$

Denota-se $eps^0(P)$ simplesmente por $eps(P)$.
Por exemplo, seja $P = (\{a\}, \{b\}) \to (\{c\}, \{d\}) \to STOP$
Aplicando-se a função eps ao processo P, obtém-se o conjunto com as diferentes maneiras de distribuir os pares de ações $(\{a\}, \{b\})$ e $(\{c\}, \{d\})$ no tempo:

$eps(P) = \{$
$[0 \to (\{a\}, \{b\}), 1 \to (\{c\}, \{d\}), ...],$
$[0 \to (\emptyset, \emptyset), 1 \to (\{a\}, \{b\}), 2 \to (\{c\}, \{d\}), ...],$

$[0 \to (\{a\},\{b\}), 1 \to (\emptyset,\emptyset), 2 \to (\{c\},\{d\}), ...],$
$[0 \to (\emptyset,\emptyset), 1 \to (\{a\},\{b\}), 2 \to (\emptyset,\emptyset), 3 \to (\{c\},\{d\}), ...],$
$...$
$\}$
$= \{[0 \to (\{a\},\{b\}), 1 \to (\{c\},\{d\}), ...]\}$

Para verificar a compatibilidade de um processo de troca com o par de papéis envolvidos, é necessário verificar se cada papel pode executar o conjunto de comportamentos que lhe corresponde no processo de troca. Para isto define-se duas funções de projeção $prj_1(e)$ e $prj_2(e)$, onde $e \in Ep$, que recuperam os comportamentos desejados para cada um dos papéis envolvidos em um processo de troca. Considerando e como o processo de troca e_1 definido anteriormente, tem-se:

$$prj_1(e) = \{0 \mapsto \{a\}, 1 \mapsto \{a\}, 2 \mapsto \{b,c\}, 3 \mapsto \{a,b\}, \ldots\}$$

$$prj_2(e) = \{0 \mapsto \{b\}, 1 \mapsto \{b,c\}, 2 \mapsto \emptyset, 3 \mapsto \{b\}, \ldots\}$$

Desta forma pode-se verificar a compatibilidade entre papéis e processos de troca, necessária para a formação do link $l = (r_1, r_2, E)$, analisando as restrições de compatibilidade a seguir:

$$\forall\, e \in E\, \forall\, t\, \exists\, b \in r_1 : prj_1(e)(t) \subseteq b(t)$$

$$\forall\, e \in E\, \forall\, t\, \exists\, b \in r_2 : prj_2(e)(t) \subseteq b(t)$$

Chama-se essa restrição de *compatibilidade fraca*, pois em cada instante pode existir um comportamento diferente em cada papel satisfazendo a restrição.

No exemplo do link $l = (r_1, r_2, e_1)$, considerando os papéis r_1 e r_2:

$$r_1 = \{[0 \mapsto \{a,d\}, 1 \mapsto \{a,e\}, 2 \mapsto \{a,b,c\}, 3 \mapsto \{a,d\}, \ldots], \ldots\}$$

$$r_2 = \{[0 \mapsto \{a,b\}, 1 \mapsto \{b,c,d\}, 2 \mapsto \{c\}, 3 \mapsto \{b,d\}, \ldots], \ldots\}$$

é possível verificar que os papéis satisfazem esta restrição e, portanto, formam o link $l = (r_1, r_2, P)$.

Uma restrição de *compatibilidade forte* exige que em cada papel exista um comportamento responsável por viabilizar, sozinho, todo o processo de troca, sendo definida como:

$$\forall\, e \in E\, \exists\, b \in r_1\, \forall\, t : prj_1(e)(t) \subseteq b(t)$$

$$\forall\, e \in E\, \exists\, b \in r_2\, \forall\, t : prj_2(e)(t) \subseteq b(t)$$

5.4 Considerações

A realização deste trabalho mostrou a possibilidade de representação de comportamentos e processos de troca entre agentes através do uso de CSP. Um estudo de caso foi realizado, conforme descrito em Barbosa et al. (2010) e, com ele foram verificadas algumas limitações com a utilização da ferramenta FDR2. Em relação à especificação de processos de troca, ela não permite que o alfabeto de eventos seja definido em termos

de operações de conjuntos das partes de um alfabeto básico finito, embora isto seja formalmente possível na estrutura de CSP. Além disto, FDR2 não permite a definição de novas relações entre símbolos do alfabeto, tornando impossível verificar as restrições de compatibilidade exigidas no modelo PopOrg.

CSP mostrou-se um formalismo útil para a representação de interações, porém não adequado para a representação estrutural de uma organização de SMAs.

6 O MÉTODO RAISE E A LINGUAGEM RSL

RAISE (*Rigorous Approach to Industrial Software Engineering*) é um método formal que fornece facilidades para o uso industrial de métodos formais no desenvolvimento de sistemas de software. Dentre seus objetivos estão a construção de softwares mais confiáveis, com menos erros, melhor documentados e de fácil manutenção. A tecnologia RAISE foi desenvolvida como trabalho coletivo dos projetos RAISE (ESPRIT I 315, 1985 - 1990) e LaCoS (ESPRIT II 5383, 1990 - 1995, *LArge scale COrrect Systems using formal methods*) [1].

RAISE foi inicialmente desenvolvido com o objetivo de fornecer uma melhoria em relação a outros métodos como VDM, Z, CSP, CCS, Larch e OBJ. Este método é composto pela linguagem *RSL- Raise Specification Language*, que é uma poderosa linguagem de especificação e projeto, e um método de desenvolvimento.

O método de desenvolvimento RAISE, apresentado em (GROUP, 1995), compreende as seguintes atividades:

- formulação de especificações abstratas;
- desenvolvimento destas especificações em outras mais concretas sucessivamente;
- demonstração da correção do desenvolvimento;
- tradução da especificação final em uma linguagem de programação.

Existem duas principais atividades envolvidas no método RAISE: escrever uma especificação inicial, e desenvolvê-la em direção a algo que possa ser implementado em uma linguagem de programação (VAN et al., 2002). Escrever a especificação inicial é a tarefa mais crítica no desenvolvimento de software. Se ela estiver errada, ou seja, se falhar em relação aos requisitos, então grande parte do trabalho feito a partir dela será desperdiçado.

Além da linguagem de especificação e do método de desenvolvimento, existem diversas ferramentas que dão suporte à utilização de RAISE. Elas são utilizadas para a edição, apresentação de justificativas (demonstrações), tradução em linguagens imperativas e suporte à documentação. A linguagem e ferramentas focam no suporte às etapas de especificação, projeto e implementação do processo de desenvolvimento de software.

6.1 RSL

A linguagem formal RSL foi projetada para i) dar suporte a especificações grandes, modulares, ii) fornecer diversos estilos de especificação (axiomática e baseada em modelos, aplicativa e imperativa, sequencial e concorrente) e iii) suportar especificações

[1] http://www.iist.unu.edu/www/raise

variando do abstrato (próximo aos requisitos) ao concreto (próximo à implementação) (GEORGE, 1991) (GROUP, 1992).

RSL é inspirada em diferentes linguagens de especificação (como VDM, Z e CSP). Oferece uma notação rica, baseada em matemática, na qual requisitos, especificações e etapas do projeto de software podem ser formulados e verificados. RSL é uma linguagem de amplo espectro, podendo ser utilizada para expressar tanto especificações abstratas, de alto nível, quanto projetos concretos, de baixo nível. Permite ainda a especificação e projeto modularizados de sistemas de grande porte e o desenvolvimento separado de sub-sistemas.

Em (GROUP, 1995), são discutidas as diferenças entre os estilos aplicativo e imperativo, e sequencial e concorrente. A linguagem RSL disponibiliza quatro opções:

- Sequencial Aplicativo: é um estilo de "programação funcional", sem variáveis ou concorrência.
- Sequencial Imperativo: com variáveis, atribuições, sequenciamento, *loops*, etc. , mas sem concorrência.
- Concorrente Aplicativo: programação funcional, mas com concorrência.
- Concorrente Imperativo: com variáveis, atribuições, sequenciamento, *loops*, etc. , e concorrência.

Os três maiores tipos de módulos geralmente utilizados são Sequencial Aplicativo, Sequencial Imperativo e Concorrente Imperativo, normalmente abreviados como Aplicativo, Imperativo e Concorrente.

No processo de modelagem de um sistema em RSL, uma especificação pode ser realizada combinando estes diferentes estilos.

6.2 Conceitos Básicos da Linguagem RSL

Um conceito chave da linguagem RSL é o de módulos, pois é através deles que as especificações são decompostas em unidades compreensíveis e reusáveis. Um módulo é basicamente uma coleção de declarações identificada por um nome. Um módulo M_1 pode ser usado para definir um módulo M_2, significando que as declarações de M_1 são usadas para definir M_2.

Em geral, a definição de um módulo tem a forma:

$id =$
class
declaração$_1$
$\vdots$
declaração$_n$
end

para n $\geq$ 0, onde uma declaração começa com uma palavra-chave (**type, value, axiom...**) que indica o tipo de declaração seguinte, seguida por uma ou mais definições deste tipo, separadas por vírgulas.

Cada declaração de classe pode incluir as seguintes declarações:

- **Types**: tipos pré-definidos e tipos compostos;
- **Objects**: módulos embutidos;
- **Channels**: para a entrada e saída de dados;
- **Values**: constantes, funções e processos;
- **Variables**: para o armazenamento de valores;
- **Axioms**: propriedades lógicas que devem ser mantidas;
- **Test cases**: expressões a serem avaliadas por outras ferramentas de tradução;

Nenhuma destas declarações é obrigatória e é permitida qualquer ordem e mais de uma ocorrência de um tipo de declaração.

Declarações de Tipos

Um tipo é uma coleção de valores logicamente relacionados. RSL apresenta alguns tipos pré-definidos, (e.g. **Nat, Int, Bool, Real, Char, Text e Unit**) e permite que novos tipos sejam definidos, através das declarações *type*. Uma declaração *type* tem a forma

type
definição_de_tipo$_1$,
⋮
definição_de_tipo$_n$

para n $\geq$ 1. Uma definição de tipo pode ter a forma

id

que define tipos abstratos, onde não são definidos operadores pré-definidos para a geração e operação de seus valores, exceto para a igualdade (=) que compara dois valores do tipo.

Outra forma de definição de tipo é

id = *expressão_tipo*

onde o nome id é especificado para ser uma abreviação para a expressão de tipo que ocorre do lado direito de =.

Ex:

type
$Author$,
$Paper$,
$Database = Paper$**-set**

Declarações de Valores

Valores podem ser nomeados em declarações de valores. Uma definição de valor pode ser uma constante, uma função ou um processo (uma função com acesso a canais). Uma declaração *value* tem a forma

value
definição_de_valor$_1$,
⋮
definição_de_valor$_n$

para n $\geq$ 1.

Uma definição de valor tem, no caso mais simples, a forma

id : *expressão_tipo*

onde o identificador id é definido para representar um valor dentro do tipo representado pela expressão de tipo. Esta definição é também chamada de assinatura do valor.

Ex:

value
$empty : Database = \{\}$,
$register : Paper \times Database \rightarrow Database$,
$author : Index \rightarrow$ **in** $count, notif$ **out** $submitPaper$ **Unit**

Declarações de Axiomas

Axiomas expressam propriedades de nomes de valores. Uma declaração *axiom* (no caso mais simples) tem a forma

axiom
expressão_de_valor$_1$,
⋮
expressão_de_valor$_n$

para n$\geq$ 1.

Ex:

axiom
$empty \equiv \{\}$,
$\forall\, p : Paper, db : Database \bullet register(p, db) \equiv \{p\} \cup db$

Declaração de Canais

RSL fornece meios para a especificação de sistemas concorrentes. Primitivas de comunicação são fornecidas tal que expressões avaliadas concorrentemente possam se comunicar através de canais.

Uma declaração *channel* tem a forma

channel
definição_de_canal$_1$,
⋮
definição_de_canal$_n$

para n$\geq$ 1. Uma definição de canal tem a forma

id : *expressão_tipo*

ou seja, o canal id é definido para transportar valores do tipo representado por $expressao\_tipo$.

Ex:

channel
$submitPaper : Paper$

6.3 Ferramentas

A ferramenta mais utilizada para RSL é a *rsltc* (*RSL Type Checker*), desenvolvida pela UNU/IIST. Esta ferramenta tem sido constantemente estendida pelos seus pesquisadores para abranger outras funcionalidades (GEORGE, 2008).

Atualmente as extensões disponíveis compreendem:

- Um *pretty printer*.
- Um gerador de "*confidence condition*".
- Um gerador de gráfico de dependência de módulos.
- Um tradutor para as linguagens SML, C++, Visual C++, CSP.
- Um tradutor para PVS.
- Um tradutor de UML para RSL e para SAL.

Outras extensões importantes referem-se ao uso de **"test_cases"** e **"ltl_asserts"** (GEORGE, 2008), que já estão incorporadas à ferramenta *rsltc*.

Os **test_cases** servem para que verificações sejam feitas na linguagem SML e os **ltl_asserts** permitem a verificação de propriedades escritas em LTL no *model checker* SAL. Ambos são utilizados como declarações (como **type**, **value**, **axiom**), seguidos de uma ou mais definições de testes ou propriedades (respectivamente) identificadas por um rótulo (opcional).

Os **test_cases** não são parte oficial de RSL. Podem ser pensados como comentários (embora o verificador de tipos reporte erros neles). Mas para um interpretador eles significam expressões que serão avaliadas. Por exemplo, se houver o caso de teste

test_case
$[t_1]\ 1 + 2$

pode-se esperar a seguinte saída do interpretador

$[t_1]\ 3$

Os casos de teste são interpretados em ordem e o resultado de um pode afetar os resultados dos outros.

Para escrever asserções dentro de uma especificação RSL é necessário seguir a seguinte gramática:

LTL_Property_decl ::= "ltl_assertion" {LTL_assertion}$^+$,
LTL_assertion ::= "["id"]" id " $\vdash$ " LTL_expr
LTL_expr ::= logical_value_expr

Os operadores temporais LTL **G** (*globaly*), **F** (*in the future*) e **X** (*in the next state*) são permitidos em LTL_exprs como símbolos de função.

6.4 Considerações

A linguagem RSL destaca-se pelos seus diferentes estilos de especificação e pela variedade de ferramentas disponíveis. Para este trabalho, em particular, outro ponto importante é a compatibilidade com CSP, bastante utilizada para sistemas concorrentes.

Como não há suporte a *model checking* em RSL, as traduções disponíveis para outras linguagens e outras ferramentas são de grande importância para que verificações adicionais possam ser realizadas.

7 POPORG X RSL

Este capítulo apresenta a utilização da linguagem RSL para a especificação de organizações de sistemas multiagentes baseados no modelo PopOrg. No seu estilo de especificação concorrente, RSL é similar à linguagem CSP onde, além do operador de sequenciamento, outros operadores como escolha interna e externa, concorrência e entrelaçamento, são usados para expressar uma especificação de forma a permitir a interação entre múltiplas tarefas (*threads*) implementadas como processos, conforme descrito em Tapia e George (2008).

Sendo assim, a linguagem RSL foi utilizada para a i) a representação da dimensão operacional do modelo PopOrg, utilizando-se o estilo concorrente de RSL e a semântica de comportamentos e interações definidas anteriormente em CSP (cf. apresentado no Capítulo 5) e ii) a representação da dimensão estrutural do modelo PopOrg, utilizando-se o estilo aplicativo. A representação da dimensão estrutural faz uso essencial do sistema de tipos da linguagem RSL.

7.1 Representação da Estrutura PopOrg em RSL

Para representar a estrutura do modelo PopOrg, foi utilizada a linguagem RSL no modo aplicativo (GROUP, 1995). Este capítulo descreve os módulos criados relacionando-os com as definições do modelo PopOrg. Esta especificação pode ser utilizada como base para a representação formal de sistemas baseados em PopOrg.

A verificação de tipos dos módulos definidos foi realizada com o apoio da ferramenta *rsltc (RSL Type Checker)* (UNU/IIST, 2008) e a especificação completa dos módulos pode ser observada no Apêndice A.

7.2 Especificação

Inicialmente foi definido o módulo principal (chamado PopOrgTypes) contendo os tipos utilizados na representação da dimensão estrutural do modelo PopOrg (cf. Fig.4.1).

scheme PopOrgTypes =
  class
    type
      $Agent$,
      $Action =$ **Text**,
      $T =$ **Nat**,
      $Behavior = T \xrightarrow{m} Action$**-set**
      $ExchProc = T \xrightarrow{m} Action$**-set** $\times$ $Action$**-set**,

$Role = Behavior$**-set**,
$Link = Role \times Role \times ExchProc$**-set**,
$Group = Role$**-set** $\times$ $Link$**-set**,
$MacroLink = Group \times Group \times ExchProc$**-set**

Na definição **type**, assume-se o tipo abstrato *Agent* e define-se os tipos *Action* e *T* como os tipos pré-definidos **Text** e **Nat**, respectivamente. A partir destes, são definidos os demais tipos *Behavior*, *ExchProc*, *Role*, *Link*, *Group* e *MacroLink*.

A seguir são definidas as funções comuns a todos os módulos que utilizam o esquema PopOrg. São elas:

value
$prj_1 : ExchProc \times T \rightarrow Action$**-set**
$prj_1(e, t) \equiv first(e(t))$,

$prj_2 : ExchProc \times T \rightarrow Action$**-set**
$prj_2(e, t) \equiv second(e(t))$,

$first : (Action$**-set** $\times$ $Action$**-set**$) \rightarrow Action$**-set**
$first(x, y) \equiv x$,

$second : (Action$**-set** $\times$ $Action$**-set**$) \rightarrow Action$**-set**
$second(x, y) \equiv y$,

$empty : Behavior$**-set** $= \{\}$,

$actTime : Behavior$**-set** $\times$ $Action$**-set** $\times T \rightarrow Action$**-set**
$actTime(b, x, t) \equiv$
case b **of**
$empty \rightarrow x$,
$\_ \rightarrow actTime(b \backslash \{$**hd**$(b)\}, ($**hd**$(b))(t)$ **union** $x, t)$
end

$actSet : Behavior$**-set** $\times$ $Action$**-set** $\rightarrow Action$**-set**
$actSet(b, x) \equiv$
case b **of**
$empty \rightarrow x$,
$\_ \rightarrow actSet(b \backslash \{$**hd**$(b)\},$ **rng**$($**hd**$(b))$ **union** $x)$
end

As funções prj_1 e prj_2 retornam o conjunto de ações realizadas pelos agentes, papéis ou grupos envolvidos em um processo de troca, utilizando-se das funções auxiliares *first* e *second* que retornam o primeiro e o segundo elemento, respectivamente, em um par de conjuntos de ações.

A definição de *empty* é utilizada para a representação do conjunto vazio. A função *actTime* retorna o conjunto de ações de um agente, papel, ou grupo em um determinado instante de tempo (utilizada para verificar a compatibilidade de processos de troca) e a função *actSet* gera o conjunto de todas as ações existentes em um comportamento.

A partir deste módulo inicial, foram definidas as estruturas do modelo PopOrg (macro-organizacional, micro-organizacional e populacional) nos correspondentes módulos **MacroOrg**, **MicroOrg** e **Pop**.

7.2.1 Módulo MacroOrg

Os componentes individualizados da estrutura macro-organizacional PopOrg são definidos como valores em MacroOrg.

MacroOrg =
extend PopOrgTypes **with**
class
value
$G_\Omega : Group$**-set**,
$Act_\Omega : Action$**-set**,
$Bh_\Omega : Behavior$**-set**,
$Ep_\Omega : ExchProc$**-set**,
$bc_\Omega : Group \xrightarrow{m} Behavior$**-set**,
$ec_\Omega : Group \times Group \xrightarrow{m} MacroLink$**-set**,
$L_\Omega : MacroLink$**-set**,
$lc_\Omega : Group \times Group \to MacroLink$**-set**,

$unionbh_\Omega : T \times Group \to Action$**-set**
$unionbh_\Omega(t, g) \equiv actTime(bc_\Omega(g), \{\}, t)$

A função $unionbh_\Omega$ é utilizada para gerar o conjunto de todos os comportamentos possíveis de um grupo em um determinado instante de tempo (utilizada para verificar a compatibilidade de processos de troca).

As propriedades dos valores são definidas através de axiomas em RSL, que podem ser identificados através de nomes entre colchetes ([ ]) para fins de documentação. No módulo MacroOrg foram definidos os seguintes axiomas:

axiom
$[prj1]$
$\forall\, e : ExchProc, t : T \bullet$
$prj_1(e, t)$ **as** a **post** $a \subseteq Act_\Omega$
pre $e \in Ep_\Omega$,

$[prj2]$
$\forall\, e : ExchProc, t : T \bullet$
$prj_2(e, t)$ **as** a **post** $a \subseteq Act_\Omega$
pre $e \in Ep_\Omega$,

Os axiomas prj_1 e prj_2 declaram que as funções prj_1 e prj_2 são definidas apenas para os processos de troca que estão no conjunto Ep_Ω (conforme definido na pré-condição) e retorna um elemento a que satisfaz a pós-condição, ou seja, faz parte do conjunto Act_Ω.

O axioma $[bc_\Omega]$ declara que a função bc_Ω é definida apenas para o conjunto de grupos que estão em G_Ω (pré-condição) e retorna um elemento b que é um elemento do conjunto Bh_Ω (pós-condição).

axiom
$[bc_\Omega]$
$\forall\, g : Group \bullet$
$bc_\Omega(g)$ **as** b **post** $b \subseteq Bh_\Omega$
pre $g \in G_\Omega$,

De forma semelhante, os axiomas $[ec_\Omega]$ e $[lc_\Omega]$ são utilizados para garantir que i) a capacidade de troca entre dois grupos g_1 e g_2, pertencentes a G_Ω (pré-condição), é um elemento do conjunto Ep_Ω (pós-condição); e ii) a capacidade de ligação entre dois grupos g_1 e g_2, pertencentes a G_Ω (pré-condição), é um elemento do conjunto L_Ω (pós-condição), respectivamente.

axiom
$[ec_\Omega]$
$\forall\, g_1, g_2 : Group \bullet$
$ec_\Omega(g_1, g_2)$ **as** e **post** $e \subseteq Ep_\Omega$
pre $g_1 \in G_\Omega \land g_2 \in G_\Omega$,

$[lc_\Omega]$
$\forall\, g_1, g_2 : Group \bullet$
$lc_\Omega(g_1, g_2)$ **as** l **post** $l \subseteq L_\Omega$
pre $g_1 \in G_\Omega \land g_2 \in G_\Omega$,

O axioma $[exchange\_capability]$ define as restrições para uma capacidade de troca, ou seja, verifica se o conjunto de ações que o grupo g_1 realiza num processo de troca (obtido pela função prj_1) está contido no conjunto de ações possíveis do grupo g_1 no instante de tempo em que ocorre a troca (obtido pela função $unionbh_\Omega$). A mesma verificação é feita para o grupo g_2, sendo g_1 e g_2 elementos de G_Ω e e um processo de troca pertencente a $ec_\Omega(g_1, g_2)$.

axiom
$[exchange\_capability]$
$\forall\, g_1, g_2 : Group, e : ExchProc, t : T \bullet$
$prj1(e, t) \subseteq unionbh_\Omega(t, g_1) \land prj2(e, t) \subseteq unionbh_\Omega(t, g_2)$
pre $g_1 \in G_\Omega \land g_2 \in G_\Omega \land e \in ec_\Omega(g_1, g_2)$

Por fim, o axioma $[macrolink\_capability]$ serve para verificar a capacidade de macro-ligação, definindo que, se existe um link entre g_1 e g_2 no conjunto L_Ω, este link deve estar presente também na capacidade de ligação destes grupos.

axiom
$[macrolink\_capability]$
$\forall\, l : MacroLink \bullet \exists\, g_1, g_2 : Group \bullet$
$(l \in L_\Omega \land (g_1 \in G_\Omega \land g_2 \in G_\Omega)) \Rightarrow l \in lc_\Omega(g_1, g_2)$

7.2.2 Módulo MicroOrg

A segunda estrutura do modelo PopOrg especificada foi a micro-organizacional. Foi criado um novo módulo com os componentes individualizados desta estrutura, redefinindo-se a função $unionbh$ para tratar de papéis.

MicroOrg =
extend PopOrgTypes **with**
class
value
$R_\omega : Role$-**set**,
$Act_\omega : Action$-**set**,
$Bh_\omega : Behavior$-**set**,
$Ep_\omega : ExchProc$-**set**,
$bc_\omega : Role \xrightarrow{m} Behavior$-**set**,
$ec_\omega : Role \times Role \xrightarrow{m} ExchProc$-**set**,
$L_\omega : Link$-**set**,
$lc_\omega : Role \times Role \rightarrow Link$-**set**,

$unionbh_\omega : T \times Role \rightarrow Action$-**set**
$unionbh_\omega(t, r) \equiv actTime(bc_\omega(r), \{\}, t)$

Os axiomas definidos para esta estrutura referem-se à verificação de propriedades do nível micro-organizacional. O axioma $[bc_\omega]$ verifica a aplicação da função bc_ω apenas para os papéis pertencentes a R_ω (pré-condição), retornando um elemento b pertencente ao conjunto Bh_ω (pós-condição).

axiom
$[bc_\omega]$
$\forall\, r : Role \bullet$
$bc_\omega(r)$ **as** b **post** $b \subseteq Bh_\omega$
pre $r \in R_\omega$,

Os axiomas $[ec_\omega]$ e $[lc_\omega]$ verificam que: i) a capacidade de troca entre dois papéis r_1 e r_2, pertencentes a R_ω (pré-condição), é um elemento do conjunto Ep_ω (pós-condição); e ii) a capacidade de ligação entre dois papéis r_1 e r_2, pertencentes a R_ω (pré-condição), é um elemento do conjunto L_ω (pós-condição), respectivamente.

axiom
$[ec_\omega]$
$\forall\, r_1, r_2 : Role \bullet$
$ec_\omega(r_1, r_2)$ **as** e **post** $e \subseteq Ep_\omega$
pre $r_1 \in R_\omega \wedge r_2 \in R_\omega$,

$[lc_\omega]$
$\forall\, r1, r2 : Role \bullet$
$lc_\omega(r1, r2)$ **as** l **post** $l \subseteq L_\omega$
pre $r_1 \in R_\omega \wedge r_2 \in R_\omega$,

O axioma $[exchange\_capability]$ define as restrições para uma capacidade de troca entre papéis, ou seja, verifica se o conjunto de ações que o papel r_1 realiza num processo de troca (obtido pela função prj_1) está contido no conjunto de ações possíveis do papel r_1 no instante de tempo em que ocorre a troca (obtido pela função $unionbh_\omega$). A mesma verificação é feita para o papel r_2, sendo r_1 e r_2 elementos de R_ω e e um processo de troca pertencente a $ec_\omega(r_1, r_2)$.

axiom
$[exchange\_capability]$
$\forall\, r_1, r_2 : Role, e : ExchProc, t : T \bullet$
$prj1(e,t) \subseteq unionbh_\omega(t, r_1) \wedge prj2(e,t) \subseteq unionbh_\omega(t, r_2)$
pre $r_1 \in R_\omega \wedge r_2 \in R_\omega \wedge e \in ec_\omega(r_1, r_2)$

Por fim, o axioma $[macrolink\_capability]$ serve para verificar a capacidade de microligação entre papéis, definindo que, se existe um link entre os papéis r_1 e r_2 no conjunto L_ω, este link deve estar presente também na capacidade de ligação destes papéis.

axiom
$[microlink\_capability]$
$\forall\, l : MicroLink \bullet \exists\, r_1, r_2 : Role \bullet$
$(l \in L_\omega \wedge (r_1 \in R_\omega \wedge r_2 \in R_\omega)) \Rightarrow l \in lc_\omega(r_1, r_2)$

7.2.3 Módulo Pop

Depois de definidas as estruturas organizacionais, foi criado o módulo **Pop**, que especifica a estrutura populacional do modelo PopOrg, com os seus componentes individualizados representados pelos valores a seguir e a função $unionbh$ sendo redefinida para o nível de agentes.

Pop =
extend PopOrg **with**
class
value
$Ag : Agent$**-set**,
$Act : Action$**-set**,
$Bh : Behavior$**-set**,
$Ep : ExchProc$**-set**,
$bc : Agent \xrightarrow{m} Behavior$**-set**,
$ec : Agent \times Agent \xrightarrow{m} ExchProc$**-set**,

$unionbh : T \times Agent \rightarrow Action$**-set**
$unionbh(t, ag) \equiv actTime(bc(ag), \{\}, t)$

Os axiomas criados para a estrutura populacional refletem propriedades desejáveis para os valores desta estrutura. Por exemplo, o axioma $[bc]$ define a aplicação da função bc apenas para os agentes pertencentes ao conjunto Ag (pré-condição), retornando um elemento b pertencente ao conjunto Bh (pós-condição).

axiom
$[bc]$
$\forall\, a : Agent \bullet$
$bc(a)$ **as** b **post** $b \subseteq Bh$
pre $a \in Ag$,

O axioma $[ec]$ declara que a função que determina a capacidade de troca entre dois agentes a_1 e a_2, pertencentes a Ag (pré-condição), retorna um elemento do conjunto Ep (pós-condição).

axiom
$[ec]$
$\forall\, a1, a2 : Agent \bullet$
$ec(a_1, a_2)$ **as** e **post** $e \subseteq Ep$
pre $a_1 \in Ag \wedge a_2 \in Ag$,

E o axioma $[exchange\_capability]$ define as restrições para uma capacidade de troca entre agentes, verificando se o conjunto de ações que o agente a_1 realiza num processo de troca (obtido pela função $prj1$) está contido no conjunto de ações possíveis do agente a_1 no instante de tempo em que ocorre a troca (obtido pela função $unionbh$). A mesma verificação é feita para o agente a_2, sendo a_1 e a_2 elementos de Ag e e um processo de troca pertencente a $ec(a_1, a_2)$.

axiom
$[exchange\_capability]$
$\forall\, a_1, a_2 : Agent, e : ExchProc, t : T \bullet$
$prj1(e, t) \subseteq unionbh(t, a_1) \wedge prj2(e, t) \subseteq unionbh(t, a_2)$
pre $a_1 \in Ag \wedge a_2 \in Ag \wedge e \in ec(a_1, a_2)$

A Tabela 7.1 apresenta a relação entre os elementos estruturais do modelo PopOrg e seus respectivos tipos em RSL, com alguns exemplos de valores que podem ser atribuídos a cada um deles.

PopOrg	**RSL**	**Exemplo**
Agente	$Agent$	
Ação	$Action = Text$	"ação1"
Tempo	T	{0,1,2,...}
Comportamento	$Behavior = T \xrightarrow{m} Action$-**set**	$[0 \mapsto \{a_1\}, 1 \mapsto \{a_2\}]$
Processo de Troca	$ExchProc = T \xrightarrow{m} Action$-**set** $\times$ $Action$-**set**	$[0 \mapsto (\{a_1\}, \{\}), 1 \mapsto (\{\}, \{a_2\})]$
Papel	$Role = Behavior$-**set**	$\{[0 \mapsto \{a_1\}, 1 \mapsto \{a_2\}], ...\}$
MicroLigação	$Link = Role \times Role \times ExchProc$-**set**	$\{(r_1, r_2, \{e_1, e_2, e_3\}), (r_1, r_3, \{e_1\})\}$
Grupo	$Group = Role$-**set** $\times$ $Link$-**set**	$\{(\{r_1, r_2, r_3\}, \{l_1, l_2\}), (\{r_2, r_3\}, \{l_4\})\}$
MacroLigação	$MacroLink = Group \times Group \times ExchProc$-**set**	$\{(g_1, g_2, \{e_1, e_3\}), (g_4, g_5, \{e_2\})\}$

Tabela 7.1: Relação entre elementos estruturais PopOrg e RSL

7.3 Representação de Comportamentos PopOrg e Processos de Troca em RSL

Para representar os aspectos operacionais do modelo PopOrg, foi utilizada a linguagem RSL nos modos aplicativo e concorrente. Esta seção descreve a semântica utilizada para esta representação.

7.3.1 Comportamentos em RSL

Uma ação PopOrg pode ser definida como uma função em RSL (adotando-se o estilo aplicativo abstrato), ou pode ser representada por um canal (estilo concorrente) por onde

são transferidos dados, podendo-se diferenciar uma ação de consulta (identificada pelo símbolo *?*) de uma ação de escrita no canal (identificada por *!*).

Para verificar a compatibilidade de representação da dimensão operacional em relação às restrições estruturais definidas anteriormente, deve-se garantir que cada uma das ações esteja no conjunto de ações Act_w. Considerando-se que a ferramenta utilizada não oferece recursos para a geração de um conjunto de todas as funções utilizadas para confrontar com o conjunto Act_w, uma alternativa é a utilização de pré-condições, tais como

value
$acao_1 : Tipo \rightarrow Tipo$
pre "$acao_1$" $\in Act_w$

Para a representação de sequências de ações, que caracterizam os comportamentos e processos de troca PopOrg, optou-se por utilizar o estilo de especificação concorrente de RSL. Neste estilo, processos (semelhante a CSP) podem ser utilizados para representar os comportamentos dos papéis e os canais podem ser utilizados para representar as ações de trocas.

Um exemplo deste tipo de especificação é o processo P que acessa os canais c_1 e c_2, obtendo informações através de c_1 e escrevendo no canal c_2. Considerando-se P um papel organizacional (representado por seus comportamentos), a verificação de compatibilidade da especificação operacional de P em relação à especificação estrutural é realizada através da pré-condição que garante que todas as ações realizadas no processo estejam no conjunto de ações Act_w e também no conjunto de comportamentos do papel P (obtidos pela função *actSet*).

scheme P =
hide c_2 in
class
channel
$c_1, c_2 : Tipo$
value
P : **Unit** $\rightarrow$ **in** c_1 **out** c_2 **Unit**
axiom
$P() \equiv$ **let** $v = c_1?$ **in** $c_2!v$ **end** ; $P()$
pre "$c_1?$", "$c_2!$", "$acao_1$" $\in Act_w \wedge$
"$c_1?$", "$c_2!$", "$acao_1$" $\in actSet(bc_w(P), \{\})$

O processo P pode ser ilustrado como mostra a Figura 7.1.

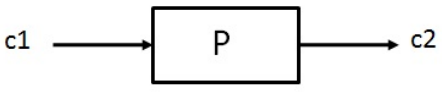

Figura 7.1: Exemplo de processo

Para destacar a diferença entre a definição de funções (no estilo aplicativo) e processos (estilo concorrente) é apresentado o exemplo a seguir relacionado a uma base de dados com suas operações *insert* e *remove*.

Os tipos utilizados em ambos os exemplos são

type
$Key, Data,$
$Database : Key \xrightarrow{m} Data$

No estilo aplicativo são construídas funções que recebem como parâmetro a base de dados e a modificam:

value
$insert : Key \times Data \times Database \to Database,$
$remove : Key \times Database \to Database$
axiom
$insert(k, d, db) \equiv db \dagger [k \mapsto d],$
$remove(k, db) \equiv db \backslash \{k\}$

Já no estilo concorrente, um processo (*database*) é definido juntamente com canais para a comunicação com este processo.

channel
$insert : Key \times Data,$
$remove : Key$
value
$database : Database \to$ **in** $insert, remove$ **Unit**,
axiom
$database(db) \equiv$
let $(k, d) = insert?$ **in** $database(db \dagger [k \mapsto d])$ **end**
[]
let $k = remove?$ **in** $database(db \backslash \{k\})$ **end**

7.3.2 Processos de Troca em RSL

Um processo de troca PopOrg corresponde à especificação de uma interação entre dois papéis (ou seja, dois conjuntos de comportamentos), podendo ser representado em RSL também como um processo, definido como

value
$troca_1 : Role \times Role \to$ **in** $c_1?$ **out** $c_1!$ **Unit**,
$troca_1(R_1, R_2) \equiv$
let $v = c_1?$ **in** ... **end** $||c_1!\_$
pre "$c_1?$" $\in actSet(bc_w(R_1)) \wedge$
"$c_1!$" $\in actSet(bc_w(R_2)) \wedge$
("$c_1?$", "$c_1!$") $\in alphaEp(ec_w(R_1, R_2), \{\})$

Neste exemplo é definida uma troca entre os papéis R_1 e R_2, onde as ações realizadas pelos papéis na troca são $c_1?$ e $c_1!$, respectivamente. Para verificar a compatibilidade desta especificação em relação à estrutural, deve-se garantir que as respectivas ações realizadas pelos papéis R_1 e R_2 estejam em seus conjuntos de comportamentos (gerados pela função *actSet*) e que o par de ações realizadas esteja no conjunto de processos de troca possíveis entre os papéis envolvidos (gerados pela função *alphaEp*).

Para representar uma interação entre dois papéis (processos em RSL), deve-se observar apenas as ações de troca que eles realizam (identificadas através de canais). Então,

a interação que dois comportamentos realmente efetivam, isto é $P()||Q()$ com as ações locais ocultas, deve ser refinada pelo processo de troca (que é visto como o conjunto mínimo de trocas que os papéis devem executar). Para exemplificar, considere o processo Q que utiliza o canal c_1 para enviar dados e executa também a ação a_1.

scheme Q =
  hide a_1 in
  class
    channel
      $c_1 : Tipo$
    value
      Q : **Unit** $\rightarrow$ **out** c_1 **Unit**
      $a_1 : Tipo_1 \rightarrow Tipo_2$
    axiom
      $Q() \equiv a_1 \; ; c_1!v \; ; Q()$

Observa-se que, no processo P, o canal c_2 está oculto e, no processo Q, a ação a_1 está oculta (através da declaração **hide**), porém as ações $c_1?$ e $c_1!$, envolvidas na troca, estão visíveis.

Desta forma, uma interação entre os processos P (definido na seção 7.3.1) e Q é dada por

axiom
  $troca_1(P, Q) \equiv P() \; || \; Q()$

Para verificar se a troca executada pelos dois processos satisfaz os requisitos mínimos dados pela $troca_1$, deve-se garantir que a interação real entre P e Q (que é este caso, onde as ações internas a_1 em P, e c_2 em Q já estão escondidas, é simplesmente $P||Q$) é refinada pelo processo de troca, representado por:

$$P||Q \sqsubseteq troca_1(P, Q)$$

Esta é uma verificação externa que pode ser feita através da ferramenta FDR2.

Observa-se, no entanto, que a noção PopOrg de especificação de comportamentos e processos de troca (sequência temporal mínima de conjuntos de ações a serem executadas) é o oposto da noção de especificação de *trace* (sequência temporal máxima de conjuntos de ações a serem executadas), de forma que as ideias de refinamentos de traces e refinamentos (implementação) de processos (de CSP, cf. (ROSCOE, 1998)) não se aplicam.

A Tabela 7.2 apresenta a relação entre os elementos operacionais do modelo PopOrg e suas respectivas representações em RSL, com alguns exemplos de aplicação.

7.4 Considerações

A linguagem RSL apresenta capacidade para a representação estrutural e operacional do modelo PopOrg. A estrutura do modelo representa uma restrição sobre o comportamento, representado através de funções e canais, necessitando-se de uma verificação destas duas especificações para garantir sua compatibilidade.

Conceito PopOrg	**Representação RSL**	**Exemplo**
Ação	função canal	$acao_1 : Tipo_1 \rightarrow Tipo_2$ $acao_2!v,\ acao_3?$
Comportamento	processo	P : **Unit** $\rightarrow$ **in** c_1 **out** c_2, c_3 **Unit** $P \equiv$ **let** $v = c_1?$ **in** $c_2!v$ **end** ; $acao_1$
Processo de Troca	processo	$EP \equiv P \|\| Q$ **hide** $(\alpha P \cup \alpha Q) - \alpha EP$, onde α corresponde ao alfabeto do processo

Tabela 7.2: Relação entre elementos operacionais PopOrg e RSL

Neste sentido a ferramenta apresentou algumas limitações, como por exemplo, não permitir a definição de um conjunto de funções utilizadas no módulo de forma automática (identificando seu alfabeto), porém outras alternativas foram encontradas, como a utilização de pré-condições nas funções.

Desta forma, pode-se dizer que RSL permite verificações internas tanto estrutural (através da análise dos alfabetos utilizados) quanto operacional (através de axiomas, canais e interações). Além disso, devido à sua compatibilidade com a linguagem CSP, uma verificação externa em relação a refinamentos, *deadlock*, *livelock* e não-determinismo, pode ser realizada.

8 ESTUDO DE CASO

Um estudo de caso foi realizado utilizando o exemplo do processo de revisão de artigos em uma conferência (também analisado em Ferber, Gutknecht e Michel (2004) e Dignum (2004)).

Neste exemplo observam-se 3 grupos papéis de agentes: o grupo de **comitê de programa** (*program committee*), o grupo de **submissão** (*submission*) e o grupo de **avaliação** (*evaluation*). O grupo Comitê de Programa é composto pelos papéis *ProgramChair* e *PC-Member*. O grupo de Submissão é composto pelos papéis *SubmissionReceiver* e *Author* e, finalmente, o grupo de avaliação é composto pelos papéis *Reviewer* e *ReviewingManager*.

As interações observadas neste exemplo podem ser descritas por:

- distribuir arquivos para a revisão
- obter avaliações
- notificação de aceitação/rejeição
- submeter artigo
- distribuir artigos para revisão
- negociar avaliação

Este capítulo descreve algumas partes da especificação do nível micro-organizacional. A especificação completa pode ser encontrada no Apêndice B.

8.1 Especificação Estrutural

A especificação estrutural apresentada refere-se à estrutura micro-organizacional. Para isso é utilizado o módulo PopOrg onde são definidos os tipos básicos do modelo e são acrescentados os valores referentes ao sistema a ser especificado.

O módulo MicroOrg_RP descreve, então, os aspectos estruturais do nível mico-organizacional do modelo PopOrg para o sistema que modela o Processo de Revisão de Artigos. Inicialmente são criados os novos tipos, identificando-se os papéis presentes no sistema.

MicroOrg_RP =
extend PopOrg **with**
class
type
$ProgramChair = Role,$

$PCMember = Role,$
$SubmReceiver = Role,$
$Author = Role,$
$Reviewer = Role,$
$RevManager = Role$

As instâncias dos papéis são representadas pelos valores *pc, pcm, sr, aut, rev* e *rm*, onde, a cada uma delas está associado o conjunto de comportamentos que pode ser executado pelo papel.

value
$pc : ProgramChair =$
$\{bh_3\},$
$pcm : PCMember =$
$\{bh_4, bh_5, bh_6\},$
$sr : SubmReceiver =$
$\{bh_2\},$
$aut : Author =$
$\{bh_1\},$
$rev : Reviewer =$
$\{bh_7, bh_8, bh_9, bh_{10}\},$
$rm : RevManager =$
$\{bh_8, bh_{10}, b_{11}\}$

Cada elemento bh_i corresponde a um comportamento possível do papel ao qual ele pertence. Por exemplo o comportamento bh_1 (o único comportamento de *author*) poderia ser descrito como segue, adotando-se a representação teórica de conjuntos usada para a definição do modelo PopOrg

$$bh_1 = \{[0 \mapsto \{\text{“}geraArtigo\text{”}\}, 1 \mapsto \{\text{“}count?\text{”}\}, 2 \mapsto \{\text{“}submitPaper!\text{”}\},$$
$$3 \mapsto \{\text{“}notif?\text{”}\}, 4 \mapsto \{\text{“}register\text{”}\}]\}$$

Quando se representa este comportamento em RSL, no entanto, utiliza-se a definição de processos de forma semelhante a CSP(cf. Seção 8.2).

A partir dos comportamentos pode ser definido o conjunto de papéis R_w:

value
$R_w : Role$**-set** $= \{pc, pcm, sr, aut, rev, rm\}$

O conjunto de ações do sistema consiste em todas as ações possíveis que os agentes podem realizar, considerando ações e ações de troca, onde informações são passadas através dos canais. No exemplo RP, o conjunto destas ações é representado por:

value
$Act_w : Action$**-set** $=$
$\{\text{“}geraArtigo\text{”}, \text{“}count!\text{”}, \text{“}count?\text{”}, \text{“}submitPaper!\text{”}, \text{“}submitPaper?\text{”},$
$\text{“}mk\_paper\text{”}, \text{“}notif!\text{”}, \text{“}notif?\text{”}, \text{“}register\text{”}, \text{“}empty?\text{”}, \text{“}insert!\text{”},$
$\text{“}insert?\text{”}, \text{“}getr!\text{”}, \text{“}getr?\text{”}, \text{“}lookup\_PCMember\text{”}, \text{“}selectMember\text{”},$
$\text{“}ins\_membDB!\text{”}, \text{“}ins\_membDB?\text{”}, \text{“}getAval!\text{”}, \text{“}getAval?\text{”},$
$\text{“}getPCMember!\text{”}, \text{“}getPCMember\text{”}, \text{“}lookup\_member\text{”},$

$$\text{“}paper\_toReview!\text{”}, \text{“}paper\_toReview\text{”}, \text{“}lookup\_rev!\text{”}, \text{“}lookup\_rev?\text{”},$$
$$\text{“}createGroup\text{”}, \text{“}ins\_revDB!\text{”}, \text{“}ins\_revDB?, \text{“}lookup\_rev!\text{”},$$
$$\text{“}lookup\_rev?\text{”}, \text{“}getReviewer!\text{”}, \text{“}getReviewer?\text{”}, \text{“}lookup\_reviewer\text{”},$$
$$\text{“}aval!\text{”}, \text{“}aval?\text{”}, \text{“}aval2!\text{”}, \text{“}aval2\text{”}, \text{“}submitPaper2!\text{”}, \text{“}submitPaper2?\text{”}\}$$

E o conjunto de todos os comportamentos do sistema pode ser representado por

value
$Bh_w : Behavior$**-set** $=$
$\{pc$ **union** pcm **union** sr **union** aut **union** rev **union** $rm\}$

Por fim, a representação do conjunto que representa os processos de trocas possíveis entre pares de papéis é dada por

value
$Ep_w : ExchProc$**-set** $=$
$\{ep_1, ep_2, ep_3\}$

onde cada ep_n corresponde a um processo de troca. Por exemplo, o processo ep_1 poderia ser descrito como (cf. Seção 8.2)

$$ep_1 = \{[0 \mapsto (\{\text{“}submitPaper!\text{”}\}, \{\text{“}submitPaper?\text{”}\})]\},$$

8.2 Especificação Operacional

Para iniciar a especificação operacional do sistema RP foram definidos os tipos

Types =
extend MicroOrg_RP **with**
type
$Paper ::$
$id : Int,$
$aut : Int,$
$title : Text,$
$area : Area,$
$Area == IA|ES|Redes|TeoComp,$
$Acceptance == Accept|Reject|NotEvaluated|InEvaluation,$
$Papers = Paper \xrightarrow{m} Acceptance,$
$Reviewers = Reviewer \xrightarrow{m} Papers,$
$PCMembers = PCMember \xrightarrow{m} Papers$**-set**,
$RevManagers = RevManager \xrightarrow{m} Papers$**-set**

A seguir, as ações foram desenvolvidas no modo aplicativo, como funções. Alguns exemplos destas ações são descritas a seguir:

value
$distributeToReview : PCMembers \times Paper \times PCMember \to PCMembers$
$distributeToReview(ms, p, m) \equiv$
ms **union** $[m \mapsto \{[p \mapsto InEvaluation]\}]$

$getResult : Papers \times Paper \to Acceptance$
$getResult(ps, p) \equiv ps(p)$

$accept : Papers \times Paper \to Papers$
$accept(ps, p) \equiv ps \dagger [p \mapsto Accept]$

Como este sistema necessita trocar informações e atualizar bases de dados (base de artigos, revisores, etc) foi necessária a utilização do modo concorrente com a criação de processos e canais. Algumas ações foram redefinidas, principalmente as que referem-se ao acesso às bases de dados atualizando as informações através dos canais.

Foi criado um processo para cada um dos papéis identificados no sistema, representando seu conjunto de comportamentos possíveis. O processo *author* é definido da seguinte maneira (especificando o comportamento bh_1, mencionado na Seção 8.1)

value
$author : IndexAut \to$ **in** $count, notif$ **out** $submitPaper$ **Unit**
axiom
$author(i) \equiv$
$geraArtigo()$;
let $c = count?$ **in** $submitPaper!mk\_paper(c, i, \text{“ ”}, \_)$ **end**;
let $(p, a) = A[i].notif?$ **in**
if $a = Accept$ **then** $register(p)$ **end**
end;
$author(i)$
pre “$geraArtigo$”, “$count?$”, “$submitPaper!$”, “mk_paper”,
“$notif?$”, “$register$” $\in Act_w \wedge$ “$geraArtigo$”, “$count?$”, “$submitPaper!$”,
“mk_paper”, “$notif?$”, “$register$” $\in bc_w(aut)$

Nota-se que este processo requer algumas especificações adicionais. As funções chamadas (*geraArtigo, mk_Paper e register*) são definidas por

value
$geraArtigo$: **Unit** $\to$ **in** $count$ **out** $count$ **Unit**;
$geraArtigo() \equiv$
let $c = count?$ **in** $count!c + 1$ **end**,
mk_paper : **Int** $\times$ **Int** $\times$ **Text** $\times Area \to Paper$,
$register : Paper \to Unit$

Ainda no processo *author* observa-se a presença da expressão $A[i].notif?$, que existe para indexar a informação no canal. Para possibilitar esta utilização é necessário a criação de um *array* de objetos da forma

object $A[i : IndexAut]$:
class
channel
$submitPaper : Paper$,
$notif : Paper \times Acceptance$,
$count$: **Int**
end
type
$IndexAut = \{|i : \textbf{Int} \bullet i \in 1..30|\}$

Através da pré-condição é feita uma verificação estrutural onde todas as ações devem estar no conjunto Act_w e na capacidade comportamental do papel $bc_w(aut)$.

O processo ***submRec***, que corresponde ao papel ***SubmissionReceiver*** tem a seguinte especificação:

value
$submRec$: **Unit** $\rightarrow$ **in** $submitPaper2, getr$ **out** $notif, insert$ **Unit**
axiom
$submRec() \equiv$
let $p = submitPaper?$ **in**
$insert!p$;
$submitPaper2!p$;
let $(p, r) = getr?$ **in** $notif!(p, r)$ **end**
end;
$submRec()$
pre "$submitPaper?$", "$insert!$", "$submitPaper2?$", "$getr?$", "$notif!$" $\in Act_w$
$\wedge$"$submitPaper?$", "$insert!$", "$submitPaper2?$", "$getr?$", "$notif!$" $\in bc_w(submRec)$

Este processo acessa a base de dados de artigos através do canal ***insert*** enviando informações do artigo para serem cadastradas. A especificação desta base de dados é dada por

value
$papersDB$:
$Papers \rightarrow$
in $empty, insert, remove, defined$
out $defined\_res$ **Unit**
axiom
$\forall\, db : Papers \bullet$
$papersDB(db) \equiv$
$empty?$; $papersDB([\,])$
[]
let $p = insert?$ **in**
$papersDB(db \dagger [p \mapsto NotEvaluated])$
end
[]
let $k = remove?$ **in**
$papersDB(db \backslash \{k\})$
end

```
    []
    let k = defined? in
      defined_res!(k ∈ dom(db)) ; papersDB(db)
    end
```

De forma semelhante, foram criados os processos pChair(), PCMember(), Reviewer() e RevManager() e as bases de dados membersDB e ReviewersDB.

O sistema principal é definido, então, como

```
value
  system : Unit → out empty Unit
axiom
  system() ≡
    empty!() ; count!0 ; all_authors()
    ||
    submRec()
    ||
    pChair()
    ||
    all_PCMembers()
    ||
    revManager()
    ||
    all_reviewers()
```

onde

$$all\_authors() \equiv ||\{author(i) \mid i : IndexAut\},$$
$$all\_pcMembers() \equiv ||\{pcMember(i) \mid i : IndexPCMember\},$$
$$all\_reviewers() \equiv ||\{reviewer(i) \mid i : IndexReviewer\}$$

8.3 Considerações

Através da realização deste estudo de caso foi possível aplicar a representação e definições propostas para a utilização de RSL e suas ferramentas na especificação de aspectos estruturais e operacionais de organizações de sistemas multiagentes baseadas no modelo PopOrg (cf. Cap. 7).

A especificação operacional descrita neste capítulo mostra a aplicação da linguagem ao nível micro-organizacional do modelo. Esta representação pode ser utilizada da mesma forma para representar o nível macro, onde o foco são os grupos (ao invés de papéis) e as ligações entre eles.

9 AVALIAÇÃO

Nesta tese foram avaliadas as linguagens de especificação formal CSP e RSL (tradicionais de Engenharia de Software) para a especificação de organizações de sistemas multiagentes. Para sua realização, optou-se por um modelo específico de organizações, PopOrg, visto que ele contém os requisitos mínimos necessários para a sua representação. Alguns pontos foram observados durante esta experiência no que se refere à expressividade de cada linguagem e suas ferramentas a fim de avaliar a sua possibilidade de aplicação.

9.1 Especificação Estrutural e Operacional

Inicialmente a linguagem CSP foi escolhida por ser uma linguagem de especificação própria para sistemas concorrentes e permitir a comunicação entre processos. Além disso, esta linguagem oferece uma ferramenta para a verificação baseada em refinamentos. No estudo de caso realizado, verificou-se que, para a parte operacional a linguagem mostra-se adequada pois permite a representação de ações, comportamentos e processos de troca entre papéis de agentes, embora a ferramenta FDR tenha apresentado algumas limitações em relação à representação do alfabeto necessário e verificações de compatibilidades entre as especificações.

Enquanto CSP é uma linguagem voltada principalmente para a modelagem de processos e suas comunicações, RSL é uma linguagem mais completa, que permite a especificação de dados e concorrência.

Outras características de RSL também contribuíram para que ela fosse utilizada neste trabalho, tais como a possibilidade de utilização de diferentes estilos de especificação e a utilização de uma mesma linguagem em diferentes níveis de desenvolvimento, além da vasta disponibilidade de ferramentas disponíveis para o seu trabalho. Sua compatibilidade com a linguagem CSP permitiu, ainda, a reutilização da semântica definida para os comportamentos e processos de troca e a detecção da possibilidade de realizar *model checking* nas especificações realizadas.

Na utilização de RSL foi observada sua capacidade tanto para a representação estrutural quanto para a operacional do modelo PopOrg, utilizando-se os estilos de especificação aplicativo e concorrente. Neste modelo, observa-se que a estrutura representa uma restrição sobre os aspectos operacionais, visto que os comportamentos e processos de troca especificados no nível operacional devem estar incluídos na especificação estrutural dos respectivos papéis que eles representam. Esta verificação pôde ser realizada através do uso de pré-condições (no caso de comportamentos) e através da operação de ocultamento de ações (no caso de processos de troca). Desta forma, pode-se dizer que RSL permite verificações internas tanto estrutural (através da análise dos alfabetos utilizados) quanto

operacional (através de axiomas, canais e interações). Além disso, devido à sua compatibilidade com a linguagem CSP, uma verificação externa em relação a refinamentos, *deadlock*, *livelock* e não-determinismo, pode ser realizada.

Outro aspecto importante que buscou-se verificar no desenvolvimento deste trabalho foi a possibilidade de realização de *model checking* nas especificações obtidas, verificando sua aplicação em organizações de SMAs.

A utilização de *model checking* para especificações em RSL pode ser feita através do verificador de modelos SAL (uma ferramenta de tradução RSLxSAL é incluída na ferramenta rsltc), porém ele não aplica-se a especificações concorrentes, pois não suporta canais. Nestes casos, geralmente o que tem ocorrido é a tradução das especificações para a linguagem CSPM de FDR (*model checker* para CSP). O trabalho de Tapia e George (2008) mostra as equivalências das duas linguagens (RSL e CSP) que provam a compatibilidade das mesmas e possibilita a tradução (uma ferramenta para automatizar este processo é fornecida juntamente com com rsltc) (GEORGE, 2008).

Em relação às propriedades LTL que podem ser representadas em RSL, uma possibilidade é a tradução de fórmulas LTL para processos de teste CSP, que podem também ser verificados com FDR, conforme descrito em Vargas et al. (2009).

Além das ferramentas citadas, em George (2008) são descritas diversas outras extensões que foram desenvolvidas para dar suporte a especificações RSL, suas verificações e traduções. Novas extensões estão em constante desenvolvimento e são disponibilizadas para os usuários através do site do projeto (http://www.iist.unu.edu/newrh/III/3/1/page.html).

9.2 Normas

No contexto social de sistemas multiagentes, em particular, de organizações de SMAs, existe uma atenção voltada ao uso de normas como forma de especificar estes sistemas. As normas ditam como os agentes devem se comportar no sistema, o que eles devem ou não fazer. E, ainda, podem ser considerados os casos em que as normas são violadas para se atingir algum objetivo mais importante (cf. (DIGNUM, 1999)).

Um sistema multiagente que utiliza normas (ou baseado em normas) é chamado sistema multiagente normativo, como apresentado em Boella e Torre (2006). Segundo Boella, sistemas normativos são um exemplo do uso de teorias sociológicas em sistemas multiagentes, i.e., a relação entre teoria de agentes e as ciências sociais como sociologia, filosofia, economia, e ciência jurídica. Estes conceitos sociais são importantes em SMAs porque existe um grande interesse entre os comportamentos de agentes tanto a nível individual quanto social, o que envolve os níveis micro e macro em organizações de agentes.

A representação de normas, geralmente é feita através de fórmulas da lógica deôntica, usando-se operadores do tipo Obrigação (**O**), Permissão (**P**), Proibição (**F**) e conceitos relacionados (WRIGHT, 1951). Por exemplo, uma norma que diz que "um agente que é autor de um artigo não pode ser revisor deste artigo" pode ser representada da seguinte forma:

$$\mathbf{O}(author(x, y) \Rightarrow \neg reviewer(x, y))$$

que é equivalente a

$$\mathbf{F}(author(x, y) \wedge (reviewer(x, y)))$$

e

$$\neg\, \mathbf{P}(author(x, y) \wedge (reviewer(x, y)))$$

Estes operadores são aplicados a fórmulas lógicas, da mesma maneira que os quantificadores. Porém, as linguagens de especificação de software não costumam fazer uso de operadores deônticos.

Logo, acredita-se que uma importante extensão necessária às linguagens de especificação de software seria a inclusão de conceitos deônticos e operadores modais. Em Lomuscio e Sergot (2001) é descrita uma proposta de extensão de um modelo semântico de especificação de sistemas distribuídos com conceitos deônticos, permitindo fórmulas com um operador semelhante ao operador de obrigação, mas que representa a ideia de comportamento ideal de um agente.

10 CONCLUSÃO

A presente tese apresentou um estudo sobre a utilização de métodos formais para a especificação de organizações de sistemas multiagentes. Foram analisadas as linguagens CSP e RSL aplicadas ao modelo organizacional PopOrg.

A linguagem CSP mostrou-se muito útil para a representação do nível operacional de organizações de SMAs, enquanto RSL apresenta maior expressividade podendo ser utilizada também para especificar o nível estrutural e para a realização de verificações de compatibilidades entre as especificações, devido às restrições impostas pelo modelo. Por fim, com a identificação da compatibilidade entre as duas linguagens trabalhadas, observou-se a possibilidade de realização de *model checking* nas especificações RSL.

A escolha de um modelo mínimo de organizações e uma linguagem de amplo espectro permitiu a verificação da viabilidade de utilização destes métodos formais para a área de SMA trazendo grandes vantagens no que se refere à detecções de erros em tempo de projeto dos sistemas e garantia de manutenção da estrutura organizacional de SMAs.

10.1 Resultados Obtidos

Dentre os resultados obtidos com a realização da presente tese, destacam-se:

- Definição de uma semântica PopOrg de CSP, com base em uma extensão do modelo de traces ;
- Representação em RSL do modelo PopOrg (estrutural e operacional), possibilitando a aplicação de ferramentas de verificação formal aos sistemas especificados de acordo com esse modelo;
- Avaliação crítica dos resultados obtidos na utilização de métodos formais usuais da ES na especificação de organizações de SMAs.
- Identificação de possibilidade de realização de *model checking* para organizações de SMAs devido à compatibilidade das linguagens CSP e RSL e ferramentas disponíveis.

10.2 Trabalhos Futuros

Alguns possíveis trabalhos futuros foram identificados no decorrer deste trabalho. São eles:

- Desenvolvimento de uma ferramenta que faça a verificação da especificação operacional em relação à estrutural de forma automática.

- Descrição de uma metodologia para a especificação de sistemas PopOrg utilizando-se a linguagem RSL.
- Desenvolvimento de uma ferramenta de tradução de especificações RSL em linguagens de programação para agentes.
- Estudo da possibilidade de inclusão de operadores deônticos em RSL.

REFERÊNCIAS

BARBOSA, R. M. et al. Using CSP in the Formal Specification of the Micro-organizational level of Multiagent Systems. In: CYBERNETICS AND SYSTEMS, Vienna. **Anais...** Austrian Society for Cybernetic Studies, 2010. p.459–464.

BJØRNER, D. On the Use of Formal Methods in Software Development. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING - ICSE, Monterey, California. **Anais...** IEEE, 1987. p.17–29.

BJØRNER, D. **Software Engineering I**. [S.l.]: Springer Science, 2007.

BOELLA, G.; TORRE, L. V. D. Introduction to Normative Multiagent Systems. **Computational and Mathematical Organization Theory**, [S.l.], v.12, p.71–79, 2006.

BORDINI, R. et al. Model checking AgentSpeak. In: INTERNATIONAL JOINT CONFERENCE ON AUTONOMOUS AGENTS AND MULTI-AGENT SYSTEMS (AAMAS2003), 2. **Proceedings...** ACM Press:, 2003. p.409–416.

BORDINI, R. et al. Model checking multi-agent programs with CASP. In: FIFTEENTH CONFERENCE ON COMPUTER-AIDED VERIFICATION (CAV-2003), Boulder, CO. **Proceedings...** Berlin: Springer-Verlag, 2003. p.110–113. (Lecture Notes in Commputer Science).

BORDINI, R. H. et al. Model Checking Rational Agents. **IEEE Intelligent Systems**, Piscataway, NJ, USA, v.19, n.5, p.46–52, 2004.

BORDINI, R. H.; VIEIRA, R. Linguagens de Programação Orientadas a Agentes: uma introdução baseada em agentspeak(l). **RITA**, [S.l.], p.7–38, 2003.

BOWEN, J. P.; HINCHEY, M. G. Seven More Myths of Formal Methods. **IEEE Software**, [S.l.], v.12, n.4, p.34–41, 1995.

CLARKE, E. M. J.; GRUMBERG, O.; PELED, D. A. **Model Checking**. Cambridge, USA: MIT Press, 1999.

COLEMAN, D. et al. **Desenvolvimento Orientado a Objetos**: o método fusion. Rio de Janeiro: Campus, 1996.

COSTA, A. C. R.; DIMURO, G. P. Semantical Concepts for a Formal Structural Dynamics of Situated Multiagent Systems. In: COIN@DURHAM, Durham, UK. **Proceedings...** Durham: University of Durham, 2007. p.41–52.

COSTA, A. C. R.; DIMURO, G. P. A Basis for an Exchange Value-Based Operational Notion of Morality for Multiagent Systems. In: FOURTH WORKSHOP ON MULTI-AGENT SYSTEMS: THEORY AND APPLICATIONS (MASTA 2007), Berlin. **Anais...** Springer Verlag, 2007. p.580–592. (Lecture Notes in Artificial Intelligence, v.4874).

COUTINHO, L. R. **Interoperabilidade Organizacional em Sistemas Multiagentes Abertos baseada em Engenharia Dirigida por Modelos.** 2009. Tese (Doutorado em Ciência da Computação) — Universidade de São Paulo (USP), São Paulo, SP.

COUTINHO, L. R.; SICHMAN, J. S.; BOISSIER, O. Modeling organization in MAS: a comparison of models. In: WORKSHOP ON SOFTWARE ENGINEERING FOR AGENT-ORIENTED SYSTEMS (SEAS'05), 1., Uberlândia, MG. **Proceedings...** Berlin: Springer, 2005.

DAVIES, J.; SCHNEIDER, S. A brief history of Timed CSP. **Theoretical Computer Science**, Essex, UK, v.138, n.2, p.243–271, 1995.

DELOACH, S. A. Engineering Organization-Based Multiagent System. In: INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING FOR LARGE-SCALE MULTI-AGENT SYSTEMS (SELMAS'05), St. Louis, MO. **Anais...** Berlin: Springer, 2006. p.109–125. (Lecture Notes in Computer Science, v.3914).

DELOACH, S. A.; WOOD, M. F.; SPARKMAN, C. H. Multiagent System Engineering. **International Journal of Software Engineering and Knowledge Engineering**, [S.l.], v.11, n.3, p.231–258, 2001.

DEMAZEAU, Y.; COSTA, A. C. R. Populations and Organizations in Open Multi-Agent Systems. In: NATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED AI (PDAI'96), 1., Hyderabad, India. **Proceedings...** [S.l.: s.n.], 1996.

DIGNUM, F. Autonomous agents with norms. **Artificial Intelligence and Law**, Netherlands, v.7, p.69–79, 1999.

DIGNUM, V. **A model for organizational interaction**: based on agents, founded in logic. 2004. Tese (Doutorado em Ciência da Computação) — Utrecht University, Utrecht.

DIGNUM, V.; DIGNUM, F. A Logic for Agent Organization. In: FAMAS@AGENTS'07, Durham. **Anais...** [S.l.: s.n.], 2007.

DIGNUM, V.; V´AZQUEZ-SALCEDA, J.; DIGNUM, F. OMNI: introducing social structure, norms and ontologies into agent organizations. In: PROGRAMMING MULTI-AGENT SYSTEMS: SECOND INTERNATIONAL WORKSHOP PROMAS 2004, New York, NY. **Anais...** Berlin Heidelberg: Springer, 2004. p.181–198. (Lecture Notes in Artificial Intelligence, v.3346).

ESTEVA, M.; CRUZ, D. de la; SIERRA, C. ISLANDER: an electronic institutions editor. In: FIRST INTERNATIONAL JOINT CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS (AAMAS 2002), Bologna. **Proceedings...** [S.l.: s.n.], 2002. p.1045–1052.

ESTEVA, M. et al. On the Formal Specification of Electronic Institutions. In: AGENT-MEDIATED ELECTRONIC COMMERCE: THE EUROPEAN AGENTLINK PERSPECTIVE,. **Anais...** Springer Verlag, 2001. p.126–147. (Lecture Notes in Artificial Intelligence, v.1991).

FERBER, J.; GUTKNECHT, O.; MICHEL, F. A meta-model for the analysis and design of organizations in multi-agent systems. In: THIRD INTERNATIONAL CONFERENCE ON MULTI-AGENT SYSTEMS, Paris, France. **Proceedings...** IEEE, 1998. p.128–135.

FERBER, J.; GUTKNECHT, O.; MICHEL, F. From Agents to Organizations: an organizational view of multi-agent systems. In: AGENT-ORIENTED SOFTWARE ENGINEERING VI. **Anais...** Springer Verlag, 2004. p.214–230. (Lecture Notes in Computer Science, v.2935).

GEORGE, C. The RAISE Specification Language: a tutorial. In: VDM'91. **Proceedings...** Springer Berlin: Heidelberg, 1991. p.238–319. (Lecture Notes in Computer Science, v.551).

GEORGE, C. **RAISE Tool User Guide**. Macau: UNU/IIST, 2008. (227).

GIORGINI, P. et al. The Tropos Methodology: an overview. In: **Anais...** Kluwer Academic Press: New York, 2003. p.505. (Methodologies And Software Engineering For Agent Systems).

GROUP, T. R. L. **The RAISE Specification Language**. [S.l.]: TERMA A/S, Denmark, 1992.

GROUP, T. R. M. **The RAISE Development Method**. [S.l.]: TERMA A/S, Denmark, 1995.

HALL, A. Seven Myths of Formal Methods. **IEEE Software**, Los Alamitos, CA, USA, v.7, p.11–19, 1990.

HüBNER, A. et al. A dialogic dimension for the moise+ organizational model. In: THE MULTI-AGENT LOGICS, LANGUAGES, AND ORGANISATIONS FEDERATED WORKSHOPS (MALLOW 2010), Lyon. **Anais...** Aachen : Sun SITE Central Europe/ RWTH Aachen University/Tilburg University, 2010. p.23/21–23/26. (CEUR-WS Workshop Proceedings Series, v.627).

HüBNER, J. F.; SICHMAN, J. S. Organização de sistemas multiagentes. In: III JORNADA DE MINI-CURSOS DE INTELIGêNCIA ARTIFICIAL (JAIA'03), Campinas, Brasil. **Anais...** Campinas: SBC, 2003. p.247–296. (III Jornada de Mini-Cursos de Inteligência Artificial (JAIA'03), v.8).

HINCHEY, M.; BOWEN, J.; ROUFF, C. Introduction to Formal Methods. In: AGENT TECHNOLOGY FROM A FORMAL PERSPECTIVE. **Anais...** Springer-Verlag, 2006. p.360.

HOARE, C. A. R. **Communicating Sequential Processes**. New York: Prentice Hall, 1985.

JENNINGS, N. R. An agent-based approach for building complex software systems. **Communications ACM**, New York, NY, USA, v.44, n.4, p.35–41, 2001.

JENNINGS, N.; WOOLDRIDGE, M. Agent-oriented software engineering. In: HANDBOOK OF AGENT TECHNOLOGY. **Anais...** AAAI/MIT Press, 2000.

JONES, C. B. **Systematic Software Development using VDM**. UK: Prentice-Hall International, 1986.

LOMUSCIO, A.; SERGOT, M. Extending Interpreted Systems with Some Deontic Concepts. In: TARK 2001. **Proceedings...** Morgan Kauffman, 2001. p.207–218.

LUCK, M.; D'INVERNO, M. Structuring a Z Specification to Provide a Formal Framework for Autonomous Agent Systems. In: ZUM '95: THE Z FORMAL SPECIFICATION NOTATION, 9TH INTERNATIONAL CONFERENCE OF Z USRES, LIMERICK, IRELAND, SEPTEMBER 7-9. **Anais...** Springer, 1995. p.47–62. (Lecture Notes in Computer Science, v.967).

PADGHAM, L.; WINIKOFF, M. Prometheus: a methodology for developing intelligent agents. In: AAMAS '02: PROCEEDINGS OF THE FIRST INTERNATIONAL JOINT CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, New york, NY, USA. **Anais...** ACM, 2002. p.37–38.

PATRA, M. R.; MOORE, R. **A Formal Model of an Agent-mediated Electronic Market**. Macau: UNU/IIST, 2000. (211).

PRESSMAN, R. S. **Engenharia de Software**. 5.ed. Rio de Janeiro: McGraw-Hill, 2002.

RAO, A. S.; GEORGEFF, M. P. BDI agents: from theory to practice. In: FIRST INTERNATIONAL CONFERENCE ON MULTI-AGENT SYSTEMS (ICMAS'95), San Francisco, CA. **Proceedings...** CA:MIT Press, 1995. p.312–319.

ROSCOE, A. W. **The theory and practice of concurrency**. Englewood Cliffs NJ: Prentice Hall, 1998.

SEARLE, J. R. **Speech acts** : an essay in the philosophy of language. Cambridge: Cambridge University Press, 1969.

SHAW, M.; GARLAN, D. **Software Architecture**: perspectives on an emerging discipline. UK: Prentice-Hall, 1996.

SHOHAM, Y. Agent-oriented programming. **Artificial Intelligence**, [S.l.], n.60, p.51–92, 1993.

SIERRA, C. et al. Engineering multi-agent systems as electronic institutions. **European Journal for the Informatics Professional**, [S.l.], v.4, p.33–39, 2004.

TAPIA, L.; GEORGE, C. **Model Checking Concurrent RSL with CSP$_M$ and FDR2**. Macau: UNU/IIST, 2008. (393).

UNU/IIST. **III/3/1 RAISE tools**. 2008.

VAN, H. D. et al. **Specification Case Studies in RAISE**. London Berlin Heidelberg: Springer-Verlag, 2002.

VARGAS, A. P. et al. Model Checking LTL Formulae in RAISE with FDR. In: IFM 2009. **Anais...** Springer-Verlag, 2009. p.231–245. (Lecture Notes in Computer Science, v.5423).

VáZQUEZ-SALCEDA, J.; DIGNUM, F. Modelling electronic organizations. In: MULTI-AGENT SYSTEMS AND APPLICATIONS III. **Anais...** Springer-Verlag, 2003. p.584–593. (Lecture Notes in Artificial Intelligence, v.2691).

WINIKOFF, M.; PADGHAM, L. The Prometheus Methodology. In: METHODOLOGIES AND SOFTWARE ENGINEERING FOR AGENT SYSTEMS. THE AGENT-ORIENTED SOFTWARE ENGINEERING HANDBOOK. **Anais...** Kluwer Publishing, 2004. p.217–234. (Multiagent Systems, Artificial Societies, and Simulated Organizations, v.11).

WOOD, M. F.; DELOACH, S. A. An Overview of the Multiagent Systems Engineering Methodology. In: FIRST INTERNATIONAL WORKSHOP ON AGENT-ORIENTED SOFTWARE ENGINEERING, Limerick, Ireland. **Proceedings...** Berlin: Springer, 2001. p.207–221. (Lecture Notes in Computer Science, v.1957).

WOOLDRIDGE, M. et al. Model checking multiagent systems with MABLE. In: AUTONOMOUS AGENTS AND MULTI-AGENT SYSTEMS (AAMAS 02), Bologna, Italy. **Proceedings...** New York: ACM Press, 2002. p.952–959.

WOOLDRIDGE, M. J. **Reasoning about rational agents**. New York: MIT Press, 2000.

WOOLDRIDGE, M. J. **An introduction to multiagent systems**. New York: John Wiley, 2002.

WOOLDRIDGE, M.; JENNINGS, N. R. Intelligent Agents: theory and practice. **Knowledge Engineering Review**, [S.l.], v.10, p.115–152, 1995.

WOOLDRIDGE, M.; JENNINGS, N. R.; KINNY, D. The Gaia Methodology for Agent-Oriented Analysis and Design. **Journal of Autonomous Agents and Multi-Agent Systems**, [S.l.], v.3, n.3, p.285–312, 2000.

WRIGHT, G. H. von. Deontic Logic. In: MIND. **Anais...** Oxford University Press, 1951. p.1–15. (New Series, v.60 No.237).

ZAMBONELLI, F.; JENNINGS, N. R.; WOOLDRIDGE, M. Developing Multiagent Systems: the gaia methodology. **ACM Transactions on Software Engineering and Methodology**, New York, NY, USA, v.12, n.3, p.317–370, 2003.

APÊNDICE A ESPECIFICAÇÃO POPORG - RSL

Os módulos a seguir foram criados na ferramenta *rsltc (RSL Type Checker)* (UNU/IIST, 2008), levando-se em consideração a conversão de símbolos de acordo com a Tabela A.1.

Sym	ASCII	Sym	ASCII	Sym	ASCII
$\times$	><	*	-list	$^\omega$	-inflist
$\rightarrow$	->	$\xrightarrow{\sim}$	-~->	$\overrightarrow{\text{m}}$	-m->
$\overset{\sim}{\overrightarrow{\text{m}}}$	-~m->	$\leftrightarrow$	<->		
$\wedge$	/\	$\vee$	\/	$\Rightarrow$	=>
$\forall$	all	$\exists$	exists	$\cdot$	:-
$\Box$	always	$\equiv$	is	$\neq$	~=
$\leq$	<=	$\geq$	>=	$\uparrow$	**
$\in$	isin	$\notin$	~isin	$\subset$	<<
$\subseteq$	<<=	$\supset$	>>	$\supseteq$	>>=
$\cup$	union	$\cap$	inter	$\dagger$	!!
$\langle$	<.	$\rangle$	.>	$\mapsto$	+>
$\parallel$	||	$\#$	++	$\square$	|=|
$\prod$	|^|	λ	-\	$\circ$	#
$\vdash$	|-	$\preceq$	{=	$\sqsubseteq$	[=

Tabela A.1: Símbolos RSL em ASCII

Módulo PopOrg

Este é o módulo principal, onde são definidos os tipos utilizados nos diferentes níveis do modelo (populacional, micro e macro-organizacional).

```
scheme PopOrg =
  class

    type
      Agent,
      Action = Text,
      T = Nat,
      Behavior = T -m-> Action-set,
      ExchProc = T -m-> Action-set >< Action-set,
      Role = Behavior-set,
      Link = Role >< Role >< ExchProc-set,
      Group = Role-set >< Link-set,
      MacroLink = Group >< Group >< ExchProc-set
```

```
    value
      first : (Action-set >< Action-set) -> Action-set
      first(x, y) is x,

      second : (Action-set >< Action-set) -> Action-set
      second(x, y) is y,

      prj1 : ExchProc >< T -> Action-set
      prj1(e, t) is first(e(t)),

      prj2 : ExchProc >< T -> Action-set
      prj2(e, t) is second(e(t)),

      empty : Behavior-set = {},

      -- gera o conjunto de ações que podem ser executadas em t
      actTime :
        Behavior-set >< Action-set >< T -> Action-set
      actTime(b, x, t) is
        case b of
          empty -> x,
          _ -> actTime(b \ {hd (b)}, (hd (b))(t) union x, t)
        end,

      -- gera o conjunto de todas as ações existentes em um comportamento
      actSet : Behavior-set >< Action-set -> Action-set
      actSet(b, x) is
        case b of
          empty -> x,
          _ -> actTime(b \ {hd (b)}, rng (hd (b)) union x)
        end
  end
```

Módulo População

```
PopOrg

scheme Pop =
  extend PopOrg with
  class

    value
      Ag : Agent-set,
      Act : Action-set,
      Bh : Behavior-set,
      Ep : ExchProc-set,
      bc : Agent -m-> Behavior-set,
      ec : Agent >< Agent -m-> ExchProc-set,

      unionbh : T >< Agent -> Action-set
      unionbh(t, ag) is actTime(bc(ag), {}, t)

    axiom
      [bc]
        all a : Agent :-
          bc(a) as b post b <<= Bh pre a isin Ag,

      [ec]
        all a1, a2 : Agent :-
          ec(a1, a2) as e post e <<= Ep
          pre a1 isin Ag /\ a2 isin Ag,

      [exchange_capability]
        all a1, a2 : Agent, e : ExchProc, t : T :-
          prj1(e, t) <<= unionbh(t, a1) /\
          prj2(e, t) <<= unionbh(t, a2)
        pre a1 isin Ag /\ a2 isin Ag /\ e isin ec(a1,a2)

    end
```

Módulo Micro-Organização

```
PopOrg

scheme MicroOrg =
  extend PopOrg with
  class

    value
      Rw : Role-set,
      Actw : Action-set,
      Bhw : Behavior-set,
      Epw : ExchProc-set,
      bcw : Role -m-> Behavior-set,
      ecw : Role >< Role -m-> ExchProc-set,
      Lw : Link-set,
      lcw : Role >< Role -> Link-set,

      -- específica para papéis
      unionbhw : T >< Role -> Action-set
      unionbhw(t, r) is actTime(bcw(r), {}, t)

    axiom
      [bcw]
        all r : Role :-
          bcw(r) as b post b <<= Bhw pre r isin Rw,

      [ecw]
        all r1, r2 : Role :-
          ecw(r1, r2) as e post e <<= Epw
          pre r1 isin Rw /\ r2 isin Rw,

      [exchange_capability]
        all r1, r2 : Role, e : ExchProc, t : T :-
          prj1(e, t) <<= unionbhw(t, r1) /\
          prj2(e, t) <<= unionbhw(t, r2),
        pre r1 isin Rw /\ r2 isin Rw /\ e isin ec(r1,r2)

      [lcw]
        all r1, r2 : Role :-
          lcw(r1, r2) as l post l <<= Lw
          pre r1 isin Rw /\ r2 isin Rw,

      [microlink_capability]
        all l : Link :-
          exists r1, r2 : Role :-
            (l isin Lw /\ (r1 isin Rw /\ r2 isin Rw)) =>
              l isin lcw(r1, r2)

  end
```

Módulo Macro-Organização

```
PopOrg

scheme Pop =
  extend PopOrg with
  class

    value
      Ag : Agent-set,
      Act : Action-set,
      Bh : Behavior-set,
      Ep : ExchProc-set,
      bc : Agent -m-> Behavior-set,
      ec : Agent >< Agent -m-> ExchProc-set,
```

```
    unionbh : T >< Agent -> Action-set
    unionbh(t, ag) is actTime(bc(ag), {}, t)

  axiom
    [bc]
      all a : Agent :-
        bc(a) as b post b <<= Bh pre a isin Ag,

    [ec]
      all a1, a2 : Agent :-
        ec(a1, a2) as e post e <<= Ep
        pre a1 isin Ag /\ a2 isin Ag,

    [exchange_capability]
      all a1, a2 : Agent, e : ExchProc, t : T :-
        prj1(e, t) <<= unionbh(t, a1) /\
        prj2(e, t) <<= unionbh(t, a2)
      pre a1 isin Ag /\ a2 isin Ag /\ e isin ec(a1,a2)
end
```

APÊNDICE B ESTUDO DE CASO - RSL

Os módulos a seguir foram criados na ferramenta *rsltc (RSL Type Checker)* (UNU/IIST, 2008), levando-se em consideração a conversão de símbolos de acordo com a Tabela A.1.

Módulo MicroOrg_RP

Este módulo representa os aspectos estruturais do nível micro-organizacional do sistema RP (*Reviewing Process*), com base no módulo geral PopOrg.

```
PopOrg

scheme MicroOrg_RP =
  extend PopOrg with
  class

    type
      ProgramChair = Role,
      PCMember = Role,
      SubmReceiver = Role,
      Author = Role,
      Reviewer = Role,
      RevManager = Role

    value
      pc : ProgramChair =
        {[0 +> {"geraArtigo"}, 1 +> {"count?"},
          2 +> {"submitPaper!"}, 3 +> {"notif?"}, 4+> {"register"}]},
      pcm : PCMember =
        {[0 +> {"paper_toReview?"}, 1 +> {"lookup_rev!"},
          2 +> {"createGroup"}, 3 +> {"ins_revDB!"},
          4 +> {"getr?"}, 5 +> {"getAval!"}]},
      sr : SubmReceiver =
        {[0 +> {"submitPaper?"}, 1 +> {"insert!"},
          3 +> {"submitPaper2!"}, 4 +> {"getr?"},
          5 +> {"notif!"}]},
      aut : Author =
        {[0 +> {"geraArtigo"}, 1 +> {"count?"},
          2 +> {"submitPaper!"}, 3 +> {"mk_paper"},
          3 +> {"notif?"}, 4 +> {"register"}]},
      rev : Reviewer =
        {[0 +> {"paper_toReview?"}, 1 +> {"aval!"}, 2 +> {"aval2!"}]},
      rm : RevManager =
        {[0 +> {"submitPaper2?"}, 1 +> {"lookup_rev"},
          3 +> {"createGroup"}, 4 +> {"getr!"}]},

      Rw : Role-set = {pc, pcm, sr, aut, rev, rm},

      Actw : Action-set =
        {"geraArtigo", "count!", "count?", "submitPaper!",
         "submitPaper?", "mk_paper", "notif!", "notif?",
         "register", "empty?", "insert!", "insert?",
         "getr!", "getr?", "lookup_PCMember",
```

```
        "selectMember", "ins_membDB!", "ins_membDB?",
        "getAval!" , "getAval?", "getPCMember!",
        "getPCMember", "lookup_member", "paper toReview!",
        "paper_toReview", "lookup_rev!", "lookup rev?",
        "createGroup", "ins revDB!", "ins revDB?",
        "lookup rev!", "lookup rev?", "getReviewer!",
        "getReviewer?", "lookup reviewer", "aval!",
        "aval?", "aval2!", "aval2", "submitPaper2!",
        "submitPaper2?"},

    Bhw : Behavior-set =
      pc union pcm union sr union aut union rev union rm,

    Epw : ExchProc-set,

    bcw : Role -m-> Behavior-set,

    ecw : Role >< Role -m-> ExchProc-set,

    Lw : Link-set,

    lcw : Role >< Role -m-> Link-set

  axiom
    [lcw]
      all r1, r2 : Role :-
        lcw(r1, r2) as l post l <<= Lw
        pre r1 isin Rw /\ r2 isin Rw,

    [microlink_capability]
      all l : Link :-
        exists r1, r2 : Role :-
          (l isin Lw /\ (r1 isin Rw /\ r2 isin Rw)) =>
            l isin lcw(r1, r2)
end
```

Módulo Types

Este módulo define os principais tipos específicos do sistema RP e a definição de algumas funções no estilo aplicativo.

```
MicroOrg_RP

scheme Types =
  extend MicroOrg_RP with
  class

    type
      Quantity = Int,
      Paper ::
        id : Int   aut : Int   title : Text   area : Area,
      Area == IA | ES | Redes | TeoComp,
      Acceptance == Accept | Reject | NotEvaluated | InEvaluation,
      Papers = Paper -m-> Acceptance,
      Reviewers = Reviewer -m-> Papers,
      PCMembers = PCMember -m-> Papers-set,
      RevManagers = RevManager -m-> Papers-set

    value
      quote : Quantity = 3,

      sum :
        Papers -> Quantity -- retorna a quantidade de papers
      sum(ps) is card (dom ps),

      sum :
        Reviewers >< Reviewer ->
          Quantity -- qtdade papers por revisor
```

```
    sum(rs, r) is card dom (rs(r)),

    -- submitPaper : Paper -> Acceptance,

    -- receivePaper : Papers >< Paper -> Papers
    -- receivePaper(ps, p) is ps union [p +> NotEvaluated],

    mk_paper : Int >< Int >< Text >< Area -> Paper,

    mk_reviewer : Int >< Area -> Reviewer,

    register : Paper -> Unit,

    -- distribui para os PCMembers
    distributeToReview :
      PCMembers >< Paper >< PCMember -> PCMembers
    distributeToReview(ms, p, m) is
      ms union [m +> {[p +> InEvaluation]}],

    -- distribui para os revisores
    distributePaper :
      Reviewers >< Paper >< Reviewer -> Reviewers
    distributePaper(rs, p, r) is
      if sum(rs, r) < quote
      then rs !! [r +> rs(r) !! [p +> InEvaluation]]
      else distributePaper(rs, p, selectReviewer(rs))
      end -- tirar este revisor da procura
    pre r isin dom rs,

    selectMember : PCMembers -> PCMember,
    -- seleciona um dos membros do comite de programa

    selectReviewer : Reviewers -> Reviewer,

    -- selectReviewer(r) is
    -- random dom r -- ver como faz o random

    getResult : Papers >< Paper -> Acceptance
    getResult(ps, p) is ps(p),

    accept : Papers >< Paper -> Papers
    accept(ps, p) is ps !! [p +> Accept],

    reject : Papers >< Paper -> Papers
    reject(ps, p) is ps !! [p +> Reject]
  end
```

Módulo RP

Este módulo apresenta a especificação concorrente do sistema RP, estendedno os módulos anteriores, com a definição de canais e processos necessários ao funcionamento do sistema.

```
Types

scheme RP =
 extend Types with
  class

    object
      A[i: IndexAut]:
         class
            channel submitPaper : Paper, notif : Paper >< Acceptance,
                    count : Int end,
      M[i: IndexPCMember]:
         class
            channel paper_toReview: Paper >< Reviewer, lookup_rev : Paper,
                    ins_revDB : Paper >< Reviewer-list end
```

```
type
  IndexAut = {| i : Int :- i isin {1..3} |}, -- indices para Autores
  IndexPCMember = {|i:Int :- i isin {1..3}|}, -- indices para PCMembers
  IndexReviewer = {|i:Int :- i isin {1..3}|}  -- indices para Revisores

channel
  remove, defined, lookup, lookup_PCMember, lookup_rev : Paper,
  submitPaper, submitPaper2 : Paper,
  notif : Paper >< Acceptance,
  count : Int,
  insert : Paper,
  getAval,getr : Paper >< Acceptance,
  ins_membDB : Paper >< PCMember,
  ins_revDB : Paper >< Reviewer-list,
  getReviewer: Reviewer-set,
  getPCMember : PCMember-set,
  lookup_res : Acceptance,
  paper_toReview : Paper >< Reviewer,
  aval: Reviewer >< Paper >< Acceptance,
  aval2: Paper >< Acceptance,
  defined_res : Bool,
  empty : Unit

value
  submRec :
    Unit -> in submitPaper2, getr out notif, insert Unit,

  pChair :
     Unit -> in submitPaper2, getr out lookup_PCMember, ins_membDB Unit,

  pcMember : IndexPCMember -> in paper_toReview out lookup_rev, ins_revDB Unit,

  author : IndexAut -> in count, notif out submitPaper Unit,

  geraArtigo : Unit -> in count out count Unit,

  reviewer : IndexReviewer -> out aval,aval2 Unit,

  selectMember: Paper -> PCMember,

  lookup_member: Paper >< PCMembers -> PCMember-set,

  lookup_reviewer: Paper >< Reviewers -> Reviewer-set,

  createGroup : Paper -> Reviewer-list,

  papersDB :
    Papers ->
      in empty, insert, remove, defined
      out lookup_res, defined_res Unit,

  membersDB :
    PCMembers ->
      in empty, ins_membDB, lookup_PCMember out getPCMember Unit,

  reviewersDB : Reviewers ->
      in empty, lookup_rev, aval
      out getReviewer Unit,

  all_authors : Unit -m-> Unit,
  all_submRec: Unit -m->Unit,
  all_pcMembers : Unit -m-> Unit,
  all_reviewers : Unit -m-> Unit,

  system :
    Unit ->
      in submitPaper, getr, notif
      out submitPaper, notif, empty Unit,

  not_found : Acceptance
```

```
    axiom
      system() is
        empty!() ; count!0 ; all_authors() || submRec()
        ||
        pChair()
        ||
        all_pcMembers()
        ||
        revManag()
        ||
        all_reviewers(),

      geraArtigo() is let c = count? in count!c + 1 end,

      all i : IndexAut :-
      author(i) is
        geraArtigo() ;
        let c = count? in submitPaper!mk_paper(c, i, "", IA) end ;
        let (p,a) = A[i].notif? in if a = Accept then register(p)end end;
        author(i)
      pre {"geraArtigo", "count?", "submitPaper!", "mk_paper",
        "notif?", "register"} <<= Actw /\
         {"geraArtigo", "count?", "submitPaper!", "mk_paper",
         "notif?", "register"} <<= actSet(bcw(aut),{}),

      all db : Papers :-
        papersDB(db) is
          empty? ; papersDB([])
          |=|
          let p = insert? in
            papersDB(db !! [p +> NotEvaluated])
          end ,
          |=|
          let k = remove? in papersDB(db \ {k}) end
          |=|
          let k = defined? in
            defined_res!(k isin dom db) ; papersDB(db)
          end
          |=|
          let k = lookup? in
            if k isin dom db
            then lookup_res!(db(k)) ; papersDB(db)
            else lookup_res!not_found ; papersDB(db)
            end
          end,

      submRec() is
        let p = submitPaper? in
          insert!p ; -- corresponde ao receive
          submitPaper2!p;
          let (p, r) = getr? in notif!(p,r) end
        end ;
        submRec()
        pre {"submitPaper?", "insert!", "submitPaper2!", "getr?", "notif!"} <<= Actw /\
         {"submitPaper?", "insert!", "submitPaper2!", "getr?", "notif!"} <<=
         actSet(bcw(sr),{}),

      pChair() is
         let p = submitPaper? in
           lookup_PCMember!p;   -- procura um PCMember da mesma área do artigo
           let m = selectMember(p) in
             ins_membDB!(p,m)
         end end;
         let (p,r) = getAval? in getr!(p,r) end;  -- pega o resultado (negocia*)
         pChair()
         pre {"submitPaper?", "lookup_PCMember!", "selectMember",
            "ins_membDB!", "getAval?", "getr!"} <<= Actw /\
             {"submitPaper?", "lookup_PCMember!", "selectMember",
            "ins_membDB!", "getAval?", "getr!" } <<= actSet(bcw(pc),{}),

--      selectMember(p) is
--         let x = getPCMember? in m post m isin x end,
```

```
      all db : PCMembers :-
        membersDB(db) is
          empty? ; membersDB([])
          |=|
          let (p,m) = ins_membDB? in
            membersDB(db !! [m+> {[p +> InEvaluation]}])
           -- paper_toReview!  -- inicializa o PCMember
          end
          |=|
          let k = lookup_PCMember? in
            getPCMember!lookup_member(k,db)
          end,

--      lookup_member(k,db) as
--         x post x <<= dom db /\ x.area = k.area,

      all i: IndexPCMember :-
      pcMember(i) is
          let (p,r) = M[i].paper_toReview? in
            lookup_rev!p;
            let g = createGroup(p) in
               ins_revDB!(p,g)end
           end;
        let (p,a) = getr? in
              getAval!(p,a) end  -- envia avaliacao para o PChair
      pre {"paper_toReview?", "lookup_rev!", "createGroup",
         "ins_revDB!", "getr?", "getAval!"} <<= Actw /\
          {"paper_toReview?", "lookup_rev!", "createGroup",
         "ins_revDB!", "getr?", "getAval!" } <<= actSet(bcw(pcm),{}),

      all db : Reviewers :-
        reviewersDB(db) is
          empty?; reviewersDB([])
          |=|
          let (p,g) = ins_revDB? in
             for r in g do
                reviewersDB(db !! [r +> db(r) !! [p+> InEvaluation]])
             end
          end
          |=|
          let k = lookup_rev? in
             getReviewer!lookup_reviewer(k,db)
          end
          |=|
          let (r,p,a) = aval?  in
               reviewersDB(db !! [r +> db(r) !! [p+>a]])
          end,

      all i: IndexReviewer :-
      reviewer(i) is
          let (p,r) = paper_toReview? in
             aval!(r,p,Accept);  -- adiciona avaliacao no reviewersDB
             aval2!(p,Accept) -- adiciona avaliacao no PCMembers
             |=|
             aval!(r,p,Reject);
             aval2!(p,Reject)
          end
       pre {"paper_toReview?", "aval!", "aval2!"} <<= Actw /\
         {"paper_toReview?", "aval!", "aval2!"} <<= actSet(bcw(pcm),{}),

      all_authors() is ||{author(i) | i:IndexAut},
      all_pcMembers() is ||{pcMember(i)| i:IndexPCMember},
      all_reviewers() is ||{reviewer(i)| i:IndexReviewer}

  end
```