UNIVERSIDADE FEDERAL DO RIO GRANDE
CENTRO DE CIÊNCIAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO
CURSO DE MESTRADO EM ENGENHARIA DE COMPUTAÇÃO

Dissertação de Mestrado

Testabilidade de sistemas multiagentes organizados usando o modelo $\mathcal{M}$oise: uma abordagem com Redes de Petri.

Bruno Coelho Rodrigues

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal do Rio Grande, como requisito parcial para a obtenção do grau de Mestre em Engenharia de Computação

Orientador: Prof. Dr. Eder Mateus Gonçalves

Rio Grande, 2018

Ficha catalográfica

R696t Rodrigues, Bruno Coelho.
Testabilidade de sistemas multiagentes organizados usando o modelo Moise: uma abordagem com Redes de Petri / Bruno Coelho Rodrigues. – 2018.
69f.

Dissertação (mestrado) – Universidade Federal do Rio Grande – FURG, Programa de Pós-Graduação em Computação,
Rio Grande/RS, 2018.
Orientador: Dr. Eder Mateus Gonçalves.

1. Testabilidade 2. Sistemas Multiagentes 3. Organização 4. Rede de Petri I. Gonçalves, Eder Mateus II. Título.

CDU 004

Catalogação na Fonte: Bibliotecário José Paulo dos Santos CRB 10/2344

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO RIO GRANDE
CENTRO DE CIÊNCIAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO
CURSO DE MESTRADO EM ENGENHARIA DE COMPUTAÇÃO

ATA DE SESSÃO DE DEFESA DE DISSERTAÇÃO DE MESTRADO

Ata nº 22/2018

Na data de 04 de outubro de 2018, às 10 horas, ocorreu a Sessão de Defesa de Dissertação de Mestrado de Bruno Coelho Rodrigues, que apresentou a dissertação intitulada AVALIAÇÃO DA TESTABILIDADE DO MODELO ORGANIZACIONAL MOISE+ BASEADA EM REDES DE PETRI, realizada sob a orientação do Prof. Dr. Eder Mateus Nunes Gonçalves. A banca examinadora foi constituída pelos Profs. Dr. Jomi Fred Hübner (UFSC), Dr. Cleo Zanella Billa (FURG) e Drª. Diana Francisca Adamatti (FURG), sob a presidência do orientador. Após a apresentação do trabalho, a banca arguiu o candidato e, a seguir, deliberou pela

(X) aprovação da Dissertação
( ) aprovação da Dissertação, sugerindo modificações no texto
( ) reprovação da Dissertação

Rio Grande, 04 de outubro de 2018

Prof. Dr. Dr. Jomi Fred Hübner

Prof. Dr. Cleo Zanella Billa

Prof. Drª. Diana Francisca Adamatti

Prof. Dr. Eder Mateus Nunes Gonçalves
Orientador

RESUMO

RODRIGUES, Bruno Coelho. **Testabilidade de sistemas multiagentes organizados usando o modelo $\mathcal{M}$oise: uma abordagem com Redes de Petri.**. 2018. 69 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação. Universidade Federal do Rio Grande, Rio Grande.

Os Sistemas Multiagentes (SMA) possuem propriedades que dificultam prever completamente seus comportamentos. Para limitar comportamentos atípicos, modelos organizacionais, como o $\mathcal{M}$oise, podem ser empregados para especificar o sistema a partir de um modelo de organização. Estes modelos estruturam os agentes em grupos, onde os membros destes grupos possuem papéis a desempenhar e restrições a obedecer. Mesmo com este nível de controle sobre os SMA, comportamentos inesperados podem surgir. Para garantir que comportamentos imprevistos não prejudiquem o funcionamento do sistema, e assegurar que software cumpra com os requisitos previstos, técnicas de teste de software podem ser empregadas como uma das ferramentas. Entretanto, para garantir uma melhor cobertura possível do sistema, com os recursos disponíveis, estratégias específicas devem ser tomadas, e para isso é necessário avaliar a testabilidade do sistema, ou seja saber o esforço necessário para testar adequadamente um programa. O objetivo principal deste trabalho é desenvolver um método para avaliar a testabilidade de SMA que empregam o modelo de organização $\mathcal{M}$oise, utilizando Rede de Petri (RP) como ferramenta de descrição e análise, onde as especificações do modelo $\mathcal{M}$oise do SMA devem ser mapeado para Redes de Petri para realizar a análise. O resultado indica o número de cenários de testes necessários para garantir através da abordagem todos os caminhos uma boa cobertura testes.

Palavras-chave: Testabilidade, sistemas multiagentes, organização, rede de Petri.

ABSTRACT

RODRIGUES, Bruno Coelho. **Testability of multiagent systems organized using the $\mathcal{M}$oise model: an approach with Petri nets.**. 2018. 69 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação. Universidade Federal do Rio Grande, Rio Grande.

Multiagent Systems (MAS) have properties that make it difficult to fully predict their behavior. To limit atypical behavior, organizational models such as $\mathcal{M}$oise can be employed to specify the system from an organizational model. These models structure the agents into groups, where members of these groups have roles to play and constraints to obey. Even with this level of control over MAS, unexpected behaviors may arise. To ensure that unforeseen behaviors do not adversely affect the operation of the system, and to ensure that the software meets the intended requirements, software testing techniques can be employed as one of the tools. However, in order to ensure a better possible coverage of the system, with the available resources, specific strategies must be taken, and for this, it is necessary to evaluate the system's testability, that is, the effort required to adequately test a program. The main objective of this work is to develop a method to evaluate the testability of MAS using the $\mathcal{M}$oise organization model, using Petri Net (PN) as a description and analysis tool, where the specifications of the $\mathcal{M}$oise model of the MAS should be mapped to PN to carry out the analysis. The result indicates the number of test scenarios required to ensure through the approach all paths, a good test coverage.

Keywords: Testability, multiagent systems, organization, Petri nets.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ED	Especificação Deôntica
EE	Especificação Estrutural
EF	Especificação Funcional
ES	Esquema Social
IA	Inteligência Artificial
IAD	Inteligência Artificial Distribuída
OE	Entidade Organizacional
OS	Especificação Organizacional
RDP	Resolução Distribuída de Problemas
RP	Rede de Petri
RPAN	Rede de Petri de Alto Nível
RPC	Rede de Petri Colorida
SMA	Sistema Multiagente

SUMÁRIO

1 INTRODUÇÃO

Os sistemas complexos e distribuídos são frequentemente utilizados no meio corporativo e industrial tanto para solucionar problemas onde eram possíveis encontrar uma solução ideal em métodos tradicionais de desenvolvimento de software, quanto adotar novas abordagens para problemas já conhecidos e solucionados. Entre estas abordagens encontram-se os sistemas multiagentes (BENFIELD; HENDRICKSON; GALANTI, 2006).

Os agentes possuem propriedades como autonomia, reatividade, proatividade e habilidades sociais (JENNINGS, 2000). Segundo HÜBNER; SICHMAN; BOISSIER (2007) em uma sociedade de agentes esta autonomia pode levar ao comportamento indesejado do sistema. Para resolver este tipo de comportamento os SMA podem ser representados como um grupo através de modelos organizacionais. Estes modelos coordenam os agentes em grupos e hierarquias fazendo com que eles sigam regras comportamentais específicas (VAN DEN BROEK et al., 2005; ARGENTE; JULIAN; BOTTI, 2006).

Mesmo possuindo estruturas organizadas os SMA ainda possuem poucas garantias de que seu funcionamento é correto, sendo este um dos obstáculos para uma maior adoção desta abordagem na indústria (HOUHAMDI, 2011; WINIKOFF, 2010). Um modo de obter essa garantia é através de Teste de Software. Os testes ajudam a medir a qualidade do software em termos de números de defeitos encontrados (GRAHAM; VAN VEENENDAAL; EVANS, 2008). No entanto, sabe-se que testar SMA não é uma tarefa trivial (WINIKOFF, 2010).

Um planejamento para o teste de software é de grande importância, já que é impossível realizar um teste completo devido o limite de recursos como custo e tempo. Neste contexto o que se deve saber é qual a quantidade de testes se deve realizar para ter uma cobertura aceitável para garantir uma boa qualidade de software.

Baseado nesta necessidade, este trabalho propõem um método para avaliar a testabilidade de SMA que utilizam o modelo $\mathcal{M}$oise. Segundo (WINIKOFF; CRANEFIELD, 2014) a testabilidade de um programa é uma métrica que indica o esforço necessário para testar adequadamente um programa. Com esta avaliação se obtém o número de casos de testes necessários para o cobertura do software através da abordagem de todos os cami-

nhos (*All path*).

1.1 Objetivos

O objetivo geral deste trabalho é desenvolver um método para avaliar a testabilidade de sistemas multiagentes levando em consideração a dimensão da organização destes sistemas, utilizando como base o teste de caixa-branca e a Rede de Petri como ferramenta para modelagem e análise dos resultados.

Os objetivos específicos a seguir servem como base para o desenvolvimento da pesquisa:

- Estudar metodologias e modelos para organização de SMA;
- Estudar abordagens de teste de software encontrados na engenharia de software tradicional;
- Modelar Rede de Petri como ferramenta a ser utilizada na modelagem de SMA no nível de organização;
- Desenvolver uma ferramenta para o cálculo de caminhos de uma rede de petri.
- Avaliar a testabilidade de SMA que utilizam $\mathcal{M}$oise como modelo organizacional.

1.2 Organização do Trabalho

O trabalho está organizado da seguinte forma:

- O capítulo 2 apresenta uma revisão bibliográfica relevante para o entendimento do trabalho com temas como: Inteligência artificial e sistemas multiagentes, teste de Software, sendo a primeira parte relativa aos testes na engenharia de software tradicional e a segunda parte uma visão dos testes em SMA, e a última seção apresenta conceitos de Redes de Petri.
- No capítulo 3 é feita uma revisão bibliográfica de trabalhos relacionados ao teste em SMA em diferentes níveis, métodos e comparados com a proposta deste trabalho.
- No capítulo 4 o método para avaliação da testabilidade do modelo organizacional $\mathcal{M}$oise baseada em Redes de Petri é apresentada.
- No capítulo 5 são utilizados exemplos para implementar o método exposto no capítulo anterior.
- O capítulo 6 apresentam as conclusões do trabalho, suas principais contribuições assim como suas limitação, que expõem oportunidades para trabalhos futuros.

2 REFERENCIAL TEÓRICO/ REVISÃO BIBLIOGRÁFICA

2.1 Inteligência Artificial e Sistemas Multiagentes

A Inteligência Artificial (IA), na sua origem, focou na criação de sistemas autônomos com a capacidade de resolver problemas com a ajuda mínima de outros sistemas. Esses sistemas, muitas vezes, não eram suficientes quando a resolução do problema não se adequava a sua especialização. A solução foi colocar o sistema em uma sociedade de sistemas com uma coleção diversificada de habilidades e capacidades, da mesma forma que as pessoas superam as limitações dos indivíduos quando trabalham juntas em uma organização (DURFEE, 1991).

Assim surgiu a Inteligência Artificial Distribuída (IAD), uma subárea da IA, que concentra esforços na pesquisa da compreensão das técnicas de conhecimento e raciocínio necessárias para a coordenação inteligente e sobre a incorporação e avaliação deste entendimento em sistemas de informação. Outros fatores também contribuíram e permitiram o desenvolvimento desta subárea, como o surgimento da computação distribuída e dos *clusters* de computadores (DURFEE, 1991; BOND; GASSER, 2014).

Segundo (DURFEE; ROSENSCHEIN, 1994) da IAD surgiram duas novas subáreas:

- Resolução Distribuída de Problemas (RDP): onde o sistema tem ênfase na resolução de um problema e como fazer com que vários agentes trabalhem juntos para solucioná-lo de forma coerente, robusta e eficiente.

- SMA: baseado na teoria dos jogos e na ciência social tem o pressuposto que o agente deve ser autônomo, racional, ter conhecimento do ambiente e de outros agentes garantindo mecanismos de comunicação e organização indiferente do problema a ser resolvido.

2.1.1 Agente

Atualmente não existe uma definição comum de agentes para toda a comunidade de IAD. Uma das definição mais aceitas foi proposta por FERBER; GASSER (1991):

> "Um agente é uma entidade real, ou virtual, imersa em um dado ambiente onde ela pode executar algumas ações, estar habilitada para perceber

> e representar parcialmente este ambiente, podendo ainda comunicar-se com os demais agentes do ambiente. Este agente apresenta um comportamento autônomo que é uma consequência de suas observações, do conhecimento armazenado e das interações com os demais agentes do ambiente."

Outras definições de agentes são apresentadas por (WOOLDRIDGE; JENNINGS, 1995; FRANKLIN; GRAESSER, 1996; RUSSELL; NORVIG, 2002), mas independente do tipo de agente e ambiente em que ele está inserido. Outro consenso na definição de agente é a propriedade de autonomia, que consiste na capacidade do agente decidir quais decisões assumir para concluir uma meta. A Figura 1 representa um agente simples e sua interação com o ambiente.

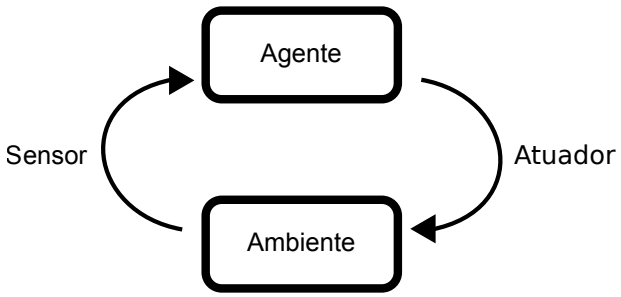


Figura 1: Agente interagindo com o ambiente através dos sensores e atuadores, adaptado de (RUSSELL; NORVIG, 2002).

Para WOOLDRIDGE; JENNINGS (1995), os agentes teriam as seguintes propriedades:

- Autonomia: os agentes operam sem intervenção direta de humanos ou outros agentes, e possuem algum mecanismo para controle de suas ações, o estado interno;
- Habilidade Social: agentes interagem com outros agentes e possivelmente humanos através de uma linguagem de comunicação;
- Reatividade: agentes percebem o ambiente e respondem rapidamente se alguma alteração ocorrer;
- Pró-atividade: agentes podem exibir comportamento orientado a objetivo e tomar a iniciativa, não agindo apenas por reação ao que ocorre no ambiente.

Outros pesquisadores quiseram dar um significado mais forte para agente, além das propriedades já apresentadas acima, incluindo outras características como:

- Mobilidade: habilidade de um agente de se mover de um local para outro na rede;

- Veracidade: um agente não envia conscientemente uma informação falsa, ou seja, não são capazes de mentir ou omitir informações;
- Benevolência: os agentes não devem apresentar um comportamento contra-produtivo, e ele deve sempre executar o que foi solicitado;
- Racionalidade: um agente atuará para alcançar seus objetivos e não agirá de forma a evitar que seus objetivos sejam alcançados, pelo menos na medida em que suas crenças o permitam.

2.1.2 Classificação dos Agentes

Assim como não existe uma definição única para agentes também não existe um único modo de classificação de agentes. Neste trabalho será apresentada a proposta de NWANA (1996) que apresentou sua classificação de agente baseada nas diferentes dimensões. As dimensões expostas foram:

- Mobilidade: capacidade ou não de um agente se movimentar por uma rede ou ambiente;
- Raciocínio: se o agente é deliberativo, agente que possui mecanismos para raciocinar, memorizar ações passadas, ou reativo, aquele que realiza uma ação baseado apenas no que ele percebe do ambiente;
- Autonomia: se os agentes podem operar sem intervenção humana, ou de outro agente;
- Aprendizagem: é a capacidade do agente em melhorar seu desempenho com a interação com o ambiente;
- Cooperação: os agentes possuem habilidade social, e interagem com outros agentes.

Conforme (NWANA, 1996) a classificação dos agentes foi criada combinando as dimensões cooperação, autonomia e aprendizagem, derivando assim sete tipos de agentes: agentes colaborativos, agentes de interface, agentes colaborativos e com aprendizagem e agentes inteligentes sendo os principais, e os tipos agentes móveis, agentes reativos e agentes híbridos como secundários, representados na Figura 2. As distinções não são definitivas e os agentes pode ainda ser agentes heterogêneos, que é quando os agentes combinam duas ou mais categorias.

2.1.3 Sistema Multiagentes

De acordo com (LESSER, 1999) sistemas multiagentes são sistemas computacionais em que dois ou mais agentes interagem ou trabalham em conjunto para executar algum conjunto de tarefas ou para satisfazer um conjunto de metas. O comportamento destes

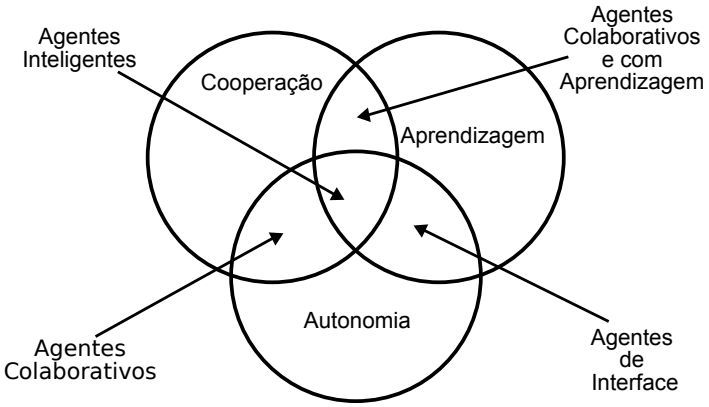


Figura 2: Classificação dos agentes, adaptado de (NWANA, 1996).

agentes pode ser limitado através de políticas nos agentes ou através de uma organização, em um acordo de que agentes específicos exercem certas funções dentro desta sociedade.

Quando os agentes interagem geralmente há algum tipo de organização que define o tipo de relação entre os agentes e influencia o seu comportamento. Estas regras de interação podem ser dinâmicas causando assim alteração no tipo de relação entre os agentes.

Na Figura 3 pode ser observado que os agentes estão formandos pequenos grupos chamados de organização, estas organizações podem ter características que modelam o comportamento dos agentes inclusive como eles interagem entre si, criando regras para determinar quem pode ou não se comunicar com outros agentes ou grupos. Os agentes estão inseridos em um ambiente que provê recursos, limitações ou outras características que podem beneficiar ou limitar os agentes para concluírem seus objetivos (JENNINGS, 2000).

2.1.4 Ambiente

O ambiente é uma abstração que dá condições para que os agentes existam, fornece recursos externos e a infraestrutura de comunicação, dando suporte para a organização e coordenação. O ambiente é uma parte essencial para o SMA, permitindo uma abstração de projeto separando as responsabilidades ajudando a gerenciar a complexidade da engenharia de software em SMA (WEYNS; OMICINI; ODELL, 2007).

Um agente tem um certo número de ações à disposição para cumprir sua meta, mas nem sempre todas estão disponíveis. Alguma das ações podem ter condições prévias informando em que situações estas podem ser aplicadas, e estas limitações podem estar diretamente relacionadas com alguma das propriedades do ambiente. RUSSELL; NORVIG (2002) propõem uma classificação para as propriedades do ambiente.

- **Totalmente observável *vs.* parcialmente observável.**
  Se os sensores do agente dão acesso ao estado completo do meio ambiente, detectando todos os aspectos relevantes para a escolha da ação então a tarefa do ambiente

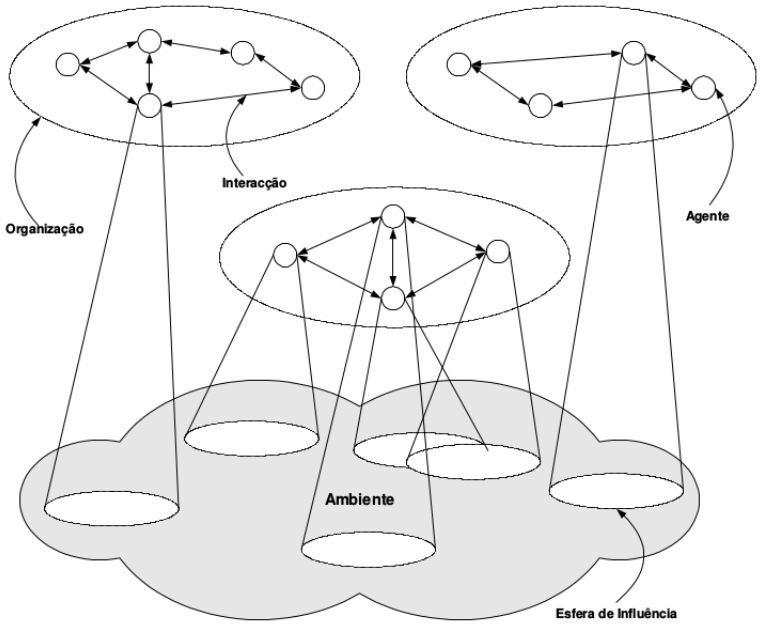


Figura 3: Estrutura de um sistema multiagente, adaptado de (JENNINGS, 2000).

é totalmente observável. Nestes tipos de ambientes, o agente não precisa manter nenhum estado interno para acompanhar o mundo. Um ambiente pode ser parcialmente observável devido aos sensores do agente serem ruidosos e imprecisos, ou porque partes do estado do ambiente estão faltando nos dados do sensor, não tendo assim informações completas do estado. Quanto mais observável é um ambiente, mais simples é criar agentes para operar nele.

- **Determinista *vs.* Estocástico**
  O ambiente é determinista se o próximo estado do ambiente for completamente determinado pelo estado atual e pela ação executada pelo agente. Não gerando assim surpresas sobre o estado que resultará a realização de uma ação por um agente. Um ambiente estocástico está associado a uma possibilidade de conhecimento do próximo estado, baseado no estado atual e na ação realizada. Se um ambiente é determinista exceto pela ação dos outros agentes então o ambiente é estratégico.

- **Episódico *vs.* Sequencial**
  Em um ambiente episódico, a escolha da ação em cada episódio depende apenas do próprio episódio, como por exemplo uma tarefa de classificação como detectar peças defeituosas em uma linha de montagem, essa ação não tem interferência de decisões anteriores, e não afetará decisões futuras. Em ambientes sequenciais a decisão atual pode afetar todas as decisões futuras, como por exemplo em um jogo de xadrez. Os ambiente episódicos são mais simples na questão de desenvolvimento.

- **Estático *vs.* dinâmico**
  Um ambiente estático permanece inalterado, exceto por algo que ocorra em con-

sequência da ação do agente. Neste tipo de ambiente o agente não precisa continuar observando o mundo enquanto decide uma ação e não precisa se preocupar com a passagem do tempo. Em um ambiente dinâmico pode ocorrer mudanças enquanto o agente está deliberando, ou seja, outros processos estão operando nele.

- **Discreto *vs.* Contínuo**
  O ambiente discreto tem um número finito de estados distintos, como um jogo de xadrez, que possui um número finito de casas e jogadas possíveis. Já em um ambiente contínuo as ações ocorrem ao longo do tempo, como por exemplo uma corrida de táxi em que as ações e a localização se alteram de acordo com o tempo.

- **Agente único *vs.* Multiagente** O ambiente onde um agente resolve um problema sozinho está em um ambiente de agente único, como é o caso de um agente que retira as peças danificadas na linha de produção. Um agente tem por objetivo concluir sua meta de melhor maneira possível. Em um ambiente multiagente podemos ter agentes atuando com comportamentos cooperativo ou competitivo de acordo com a situação desejada.

2.1.5 Organização

Na nossa sociedade temos diversas organizações, e elas mostram um mecanismo eficaz para coordenar comportamentos de diferentes agentes em uma comunidade. No domínio dos sistemas multiagente a organização é uma coleção de papéis, relacionamentos e estruturas de autoridade que regem seu comportamento. Geralmente os SMA têm alguma forma de organização, embora possa ser implícita e informal. As organizações de agentes orientam o modo como os membros da população interagem uns com os outros, não apenas a curto prazo, mas também a longo prazo. Essa orientação pode influenciar as relações de autoridade, fluxo de dados, alocação de recursos, padrões de coordenação ou qualquer outra série de características do sistema (HORLING; LESSER, 2004).

Existem vários modelos de organização e eles estruturam o comportamento de entidades complexas em uma hierarquia de entidades encapsuladas onde cada membro tem funções a desempenhar. As funções ou papéis estruturam os departamentos, e estes estruturam uma organização. A teoria da organização analisa como estas organizações funcionam, suas principais características, as características mais relevantes dos membros, os papéis gerais que os membros adotam, os relacionamentos entre os membros, a hierarquia, as regras e normas que regem a organização (ARGENTE; JULIAN; BOTTI, 2006). A organização pode ajudar grupos de agentes simples a exibir comportamentos complexos e ajudar agentes sofisticados a reduzir a complexidade de seus raciocínios.

No início, os sistemas tinham apenas uma visão centrada nos aspectos individuais dos agentes de modo que o SMA é projetado em termos de estados mentais dele, como as crenças, intenções e objetivos, conhecida como metodologia orientada a agentes. Desde

então os SMA evoluíram para a metodologia orientada para a organização, levando em conta suas principais metas, estrutura e normas sociais (ARGENTE; JULIAN; BOTTI, 2006).

Um conceito do tema de organização é a coordenação e MALONE; CROWSTON (1994) a definem como gestão de dependências entre atividades independentes. Esta definição para coordenação tem um sentido inclusivo para diferentes áreas de conhecimento, podendo ser aplicado em teoria organizacional, economia, ciência política, biologia, entre outros. Para a computação, as dependências entre diferentes processos computacionais se assemelham a interações entre pessoas, e devem ser estudados como podem ser gerenciadas.

Os modelos de organização tornaram-se populares para coordenar entidades autônomas em sistemas abertos, descentralizados e dinâmicos. Estes modelos propõem uma regulação dos SMA por um conjunto de normas, planos, mecanismos e/ou estruturas formalmente especificadas para alcançar algum objetivo global desejado.

Um modelo organizacional consiste em uma estrutura conceitual e uma sintaxe em que as especificações para organizações de agentes podem ser escritas. A partir destas especificações uma organização pode ser editada em uma plataforma SMA, ou então podem ser utilizadas infraestruturas de gerenciamento de organização, que possuem as especificações para que um agente possa saber como acessar os serviços, e fazer solicitações de acordo com o modelo organizacional disponível, podendo assim fazer parte de uma organização.

Alguns conceitos são recorrentes em diversos modelos que abordam a organização em SMA e são apresentados a seguir:

- **Normas:** (do inglês *Norms*) frequentemente orientam a escolha dos comportamentos nas sociedades humanas (SEN; AIRIAU, 2007). Segundo CIALDINI; TROST (1998) as normas são regras e padrões que são entendidos por membros de um grupo e que orientam e/ou limitam o comportamento social. As normas são utilizadas para caracterizar os comportamentos dos membros de uma organização, vinculando os membros de um grupo e servindo para orientar, controlar ou regular o comportamento adequado e aceitável (JENNINGS, 2000).

- **Papel:** (do inglês *Role*) para ODELL; PARUNAK; FLEISCHER (2002) "um papel é uma classe que define um repertório comportamental normativo de um agente." Um papel deve ser centrado na organização e não no agente, promovendo uma separação de responsabilidades. Os papéis prescrevem um comportamento esperado do agente orientando como interagir na organização (TINNEMEIER; DASTANI; MEYER, 2009).

- **Meta:** (do inglês *Goal*) de acordo com (HÜBNER, 2003) podemos separar em

meta global, que é o estado do mundo desejado pelo SMA e a meta local, que é o objetivo de um único agente.

- **Plano:** (do inglês *Plan*) é o conjunto de ações para cumprir a meta desejada.

Estes conceitos podem variar de acordo com os modelos de organização de SMA, na subseção a seguir será apresentado o $\mathcal{M}$oise$^+$ com mais detalhes.

2.1.5.1 $\mathcal{M}$oise$^+$

O modelo de organização $\mathcal{M}$oise (*Model of Organization for multI-agent SystEms*) foi inicialmente proposto por (HANNOUN et al., 2000) e extensões posteriormente foram desenvolvidas, por (HÜBNER, 2003) e (HÜBNER; SICHMAN; BOISSIER, 2005). Este modelo apresenta uma visão centrada na organização, e a organização é como um conjunto normativo de regras que restringe o comportamento dos agentes. Neste consideram-se três formas de representar as restrições organizacionais: papéis, planos e normas (HANNOUN et al., 2000).

De acordo com (HANNOUN et al., 2000) quando um agente entra em uma organização ele passa a ter que respeitar obrigações e interdições, e tem permissões relacionadas a esta organização. O $\mathcal{M}$OISE é estruturado em três níveis: nível individual, nível coletivo e nível social. No nível individual as restrições são sobre as possibilidades de ação de cada agente. No nível coletivo a restrição está no conjunto de agentes que podem cooperar. No nível social, os enlaces organizacionais restringem os tipos de interação que os agentes podem ter com o sistema.

A Especificação Organizacional (OS) no $\mathcal{M}$oise$^+$ é formada com base nas especificações estrutural, funcional e deôntica (HÜBNER, 2003):

- *Especificação Estrutural (EE)*: no nível individual, os papéis têm a função de ser elo entre o agente e a organização. No nível social os papéis se relacionam com outros papéis através de ligações e compatibilidade. No nível coletivo os grupos representam um conjunto de agentes com maior afinidade e objetivos mais próximos. A Figura 4 representa o grupo *seleção*, os *papéis* (docente, membro, funcionário secretário, presidente e candidato) e a *ligação* entre eles, este grupo forma uma sociedade (soc) de uma comissão de seleção para o ingresso em um curso de pós-graduação.

- *Especificação Funcional (EF)*: é constituída por um conjunto de esquemas sociais e de uma relação entre preferência entre missões. Nos esquemas sociais a meta global é um conceito fundamental. No nível individual um esquema social é constituído por missões. A missão é o conjunto de metas globais que podem ser passadas a um agente através de um de seus papéis. No nível coletivo um esquema social é uma árvore de decomposição de metas globais, a raiz é meta do esquema social

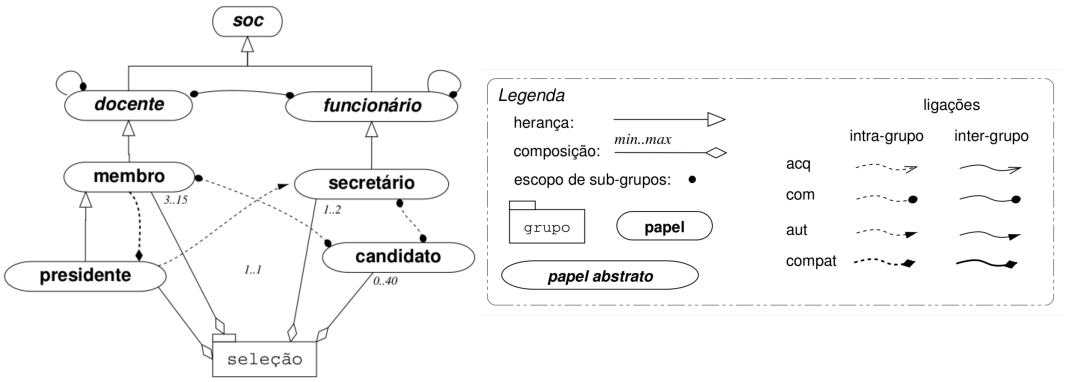


Figura 4: Especificação Estrutural (HÜBNER, 2003).

e a decomposição de metas é feita através de planos. A Figura 5 representa o esquema social para o exemplo. A leitura do esquema social deve ser realizada da esquerda para a direita e de baixo para cima, começando pela meta $g5$ da Figura 5. O operador *sequência* significa que a meta $g2$ só pode ser realizada quando a meta $g1$ for satisfeita. O operador *escolha* significa que a meta $g7$ será satisfeita se uma e somente uma das metas $g8$ ou $g9$ for satisfeita. O operador *paralelismo*, significa que a meta $g4$ será satisfeita quando ambas as metas $g5$ e $g6$ forem alcançadas, mas as duas sub-metas podem ser buscadas em paralelo.

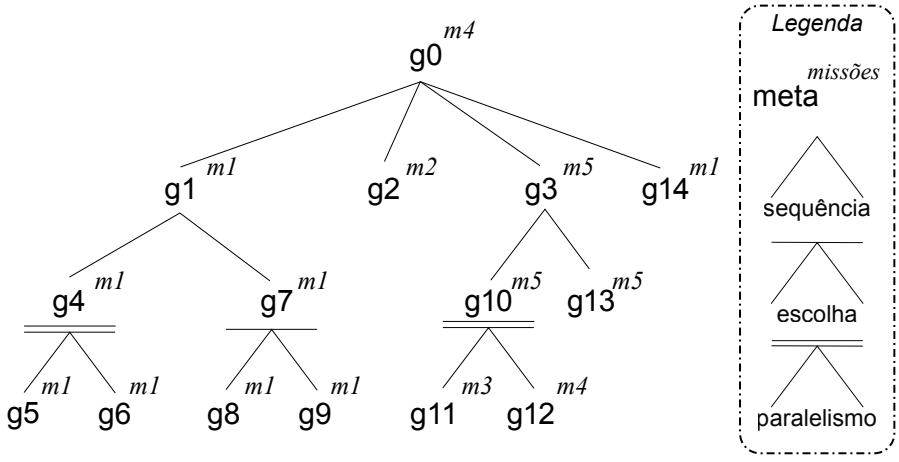


Figura 5: Esquema Social, adaptado de (HÜBNER, 2003).

Os diferentes papéis do exemplo possuem diferentes missões, um agente no papel de candidato, tem a missão $m1$. Para cumprir esta missão o candidato deve ter como objetivo as metas ($g1, g4, g5, g6, g7, g8, g9$ e $g14$), a descrição das metas está na Tabela 1.

Portanto para cumprir a meta $g0$ primeiro o agente no papel candidato tem que ter toda a documentação necessária $g5$ e ter um orientador $g6$, com estas duas metas concluídas então a meta $g4$ é finalizada dando assim liberação para continuar, no caso para as metas $g8$ ou $g9$ já que o operador é de escolha.

Realizando a meta submissão eletrônica ($g8$) ou por correio ($g9$) a meta $g7$ está habilitada para ser concluída. Neste momento a meta $g1$ está liberada para ser satisfeita. A meta $g2$ faz parte da missão $m2$ que corresponde ao papel secretário, então um agente que assumir o papel de secretário da comissão deverá verificar se a documentação está correta.

As metas $g11$ e $g12$ podem ser realizadas em paralelo, e são responsabilidades do papel secretário $m3$ e do papel membro $m4$ respectivamente, assim que a metas estiverem concluídas, o presidente $m5$ assume a responsabilidade que $g10$ seja concluída, e terá que garantir que o projeto do candidato seja avaliado $g13$ e assegurar que seja aprovado pela comissão $g3$, ou reprovado caso ineficiente (um caso de falha na missão do candidato não descrito na especificação). E por fim o candidato tem a meta de garantir que o formulário de matrícula seja recebido pela comissão de seleção $g14$.

As relações das missões com os papéis podem ser observadas no item Especificação Deôntica logo abaixo.

Tabela 1: Descrição das metas (HÜBNER, 2003).

meta	descrição	meta	descrição
g0	candidato é aceito no programa de pós-graduação	g7	a inscrição está submetida
g1(Dt)	a documentação é recebida no prazo	g8	submissão eletrônica
g2	a documentação está correta	g9	submissão por correio
g3	candidato é aprovado pela comissão	g10	metas g11 e g12 são cumpridas
g4	metas g3 e g4 são cumpridas	g11	uma reunião está marcada
g5	candidato tem toda a documentação necessária	g12	um relator está indicado
g6	candidato tem um orientador	g13	o projeto do candidato é avaliado
g14	formulário de matrícula preenchido é recebido		

- *Especificação Deôntica (ED)*: é a especificação que relaciona a EE com a EF no nível individual, especificando quais as missões um papel tem permissão ou obrigação de realizar. A Tabela 2 contém a ED da sociedade comissão de seleção.

Tabela 2: Especificação deôntica. (HÜBNER, 2003)

papel	relação deôntica	missão
candidato	permissão	*m1*
secretário	permissão	*m2*
secretário	permissão	*m3*
membro	permissão	*m4*
presidente	permissão	*m5*

2.2 Teste de Software

O teste faz parte dos métodos de verificação e validação de software onde a verificação investiga se o software atende aos seus requisitos funcionais e não funcionais enquanto a validação é um processo mais amplo no qual o objetivo é garantir que o software atenda às expectativas do cliente, sendo essencial já que nem sempre as especificações dos requisitos refletem os desejos ou necessidades reais dos clientes e usuários do sistema (SOMMERVILLE, 2010). O termo garantir refere-se ao processo que visa obter confiança de que um sistema se comportará adequadamente (WINIKOFF, 2010).

Segundo SOMMERVILLE (2010) o processo de verificação e validação possui outras atividades além dos testes, entre elas as inspeções e revisões de software. Estas atividades analisam e verificam os requisitos do sistema, os modelos de projeto, o código-fonte do programa e os testes propostos para o sistema como descrito na Figura 6.

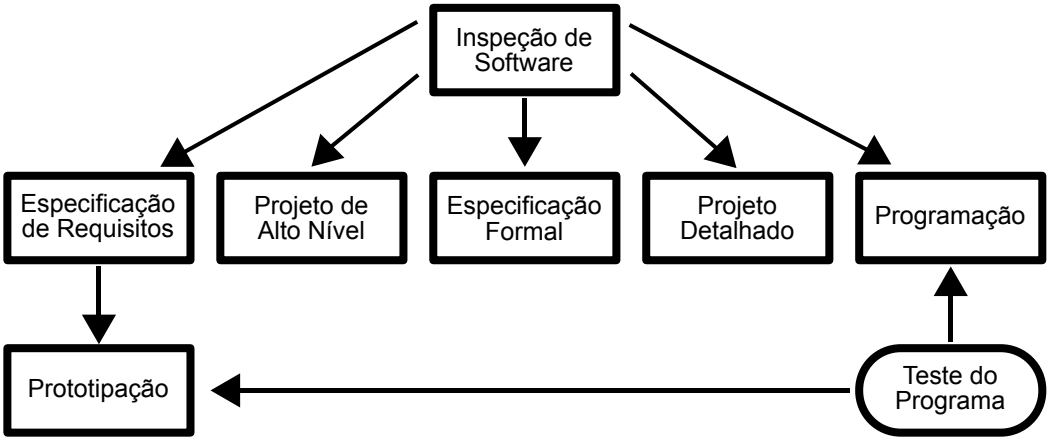


Figura 6: Processo de verificação e validação. Adaptada de (SOMMERVILLE, 2010).

As inspeções podem ser mais eficientes nas descobertas de defeitos do que os testes de software e possuem vantagens, como ser utilizada em versões incompletas do sistema e podem ser inspecionadas sem necessidade de desenvolvimento de um teste específico. Os defeitos encontrados em uma inspeção podem considerar outros atributos de qualidade para um programa. A inspeção é um processo estático e não tem preocupação com as interações entre erros. Em testes dinâmicos muitas vezes um erro pode encerrar o sistema ou o próprio teste, e consequentemente uma única sessão de inspeção pode descobrir muitos erros em um sistema.

No entanto, as inspeções não podem substituir os testes de software, pois não são eficazes para descobrir defeitos que surgem devido a interações inesperadas entre diferentes partes de um programa, problemas de temporização ou problemas com o desempenho do sistema. Além disso, pode ser difícil e dispendioso montar uma equipe de inspeção (SOMMERVILLE, 2010).

Teste de software é um processo, ou uma série de processos, elaborados para garantir que o código do computador realize o que foi projetado para fazer. O software deve

ser previsível e consistente, sem oferecer surpresas aos usuários (MYERS; SANDLER; BADGETT, 2011). Já para (BOURQUE; FAIRLEY et al., 2014) o teste é uma verificação dinâmica onde, em um conjunto de casos de testes finitos adequadamente selecionados dentro de um domínio de execução infinito, é analisado se o programa apresenta o comportamento esperado. Dentro deste conceito dinâmico significa que o teste requer sempre a execução do programa com entradas selecionadas.

O teste tem a função de medir a qualidade do software por pelo menos três termos: do número de defeitos encontrados, pelo rigor dos testes executados e pela cobertura do sistema pelos testes. Um teste deficiente pode revelar defeitos e passar falsa sensação de segurança. Um teste bem planejado irá revelar, em primeiro momento, defeitos se estiverem presentes e, se o teste passar, aumentará a confiança no software e será possível afirmar que o nível geral de risco de usar o sistema foi reduzido. Quando o teste encontra defeitos, a qualidade do sistema de software aumenta quando esses defeitos são corrigidos, desde que as correções sejam realizadas corretamente (GRAHAM; VAN VEENENDAAL; EVANS, 2008).

Antes de prosseguir alguns termos precisam ter suas definições apresentadas para evitar confusão. Conforme (JORGENSEN, 2016) as terminologias apresentadas nesta dissertação estão de acordo com *International Software Testing Qualification Board* (ISTQB) e estas são compatíveis com os padrões do *Institute of Electronics and Electrical Engineers* (IEEE) *Computer Society* (IEEE, 1983).

- Erro (do inglês *error*) - As pessoas cometem erros. Este erros podem ser cometidos durante a codificação por exemplo. Outros sinônimos em português podem ser engano ou equívoco.

- Defeito (do inglês *fault*) - É o resultado de um erro. É mais preciso dizer que um defeito é a representação de um erro, onde a representação é o modo de expressão, sendo uma das formas de expressão um erro no código fonte, gerando uma anomalia (*bug*) no funcionamento no sistema.

- Falha (do inglês *failure*) - É o resultado da execução de um defeito no código. As falhas também podem ser causadas por condições ambientais ou condições de *hardware*.

- Incidente - Ocorrência de evento que requer uma investigação. Um incidente é o sintoma associado a uma falha que alerta o usuário para a sua ocorrência.

- Caso de teste - um caso de teste possui um identificador e está associado ao comportamento de um programa. Ele também possui um conjunto de entradas e saídas esperadas.

Na Figura 7 é apresento um modelo abstrato do processo de teste tradicional, onde os casos de testes são especificações de entradas ao teste e onde é esperado uma saída com os resultados, relacionado com uma declaração do que está sendo testado. Os dados de testes são as entradas que foram planejadas para testar o sistema, os resultados dos testes são automaticamente comparados com os resultados previstos não tendo necessidade de uma pessoa para verificar anomalias nesta etapa (SOMMERVILLE, 2010).

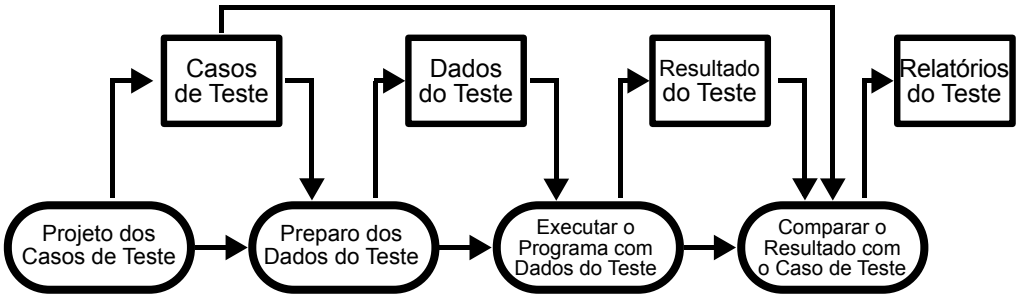


Figura 7: Modelo do processo de teste de software. Adaptada de (SOMMERVILLE, 2010).

Ainda sobre casos de testes, segundo (MYERS; SANDLER; BADGETT, 2011) é de grande importância um bom planejamento, já que é impossível realizar um teste completo. Um boa estratégia é tentar realizar o teste o mais completo possível dadas as restrições de tempo e custo. Levando isso em conta a questão-chave se torna: qual o subconjunto de casos de testes entre todos os casos de testes possíveis tem a maior probabilidade de detectar a maior parte das falhas? Em geral é impraticável, muitas vezes impossível, encontrar todos os erros em um programa. Este problema fundamental, por sua vez, terá implicações para a economia dos testes, os pressupostos que o testador terá que fazer sobre o programa e a maneira como os casos de teste são projetados.

Para encontrar um equilíbrio entre o menor número de casos de testes e uma ótima cobertura do programa são adotadas estratégias de teste de software. Uma estratégia para testes de software fornece um roteiro que descreve as etapas a serem realizadas como parte do teste, quando as etapas são planejadas, realizadas, e quanto esforço, tempo e recursos serão necessários. Portanto, qualquer estratégia de teste deve incorporar o planejamento do teste, a execução do teste e a coleta e avaliação de dados resultantes (PRESSMAN, 2005).

Neste trabalho serão apresentadas duas abordagens que são utilizadas para identificar casos de testes, uma baseada em especificação e tradicionalmente chamada de *testes funcionais*, e outra baseada em código chamada de *testes estruturais* (JORGENSEN, 2016). Os testes funcionais também são conhecidos por *teste de caixa preta*, realizados na interface do software e com pouca consideração à estrutura lógica interna do software. Os testes estruturais, também conhecidos como *teste de caixa branca*, são baseados em uma análise dos detalhes procedurais, os caminhos lógicos através do software e as colabora-

ções entre componentes (PRESSMAN, 2005).

2.2.1 Teste de caixa preta

O termo caixa preta é utilizado fazendo referência ao fato de que o projetista de testes não tem acesso ao código fonte do programa, sendo a parte interna do sistema desconhecida. Assim o desenvolvimento do projeto de teste é realizado apenas tendo conhecimento da especificação do software. Para estes testes, a única informação utilizada é a especificação do software. Portanto os casos de testes são independentes de como o sistema é implementado, não gerando problemas se ocorrerem mudanças na implementação, e o desenvolvimento dos casos de teste pode ocorrer em paralelo com com a implementação (JORGENSEN, 2016).

De acordo com (PRESSMAN, 2005), com nesta abordagem é possível obter um conjunto de condições de entradas que exercerão plenamente todos os requisitos funcionais de um programa. As principais fontes de erros encontrados são funções incorretas ou ausentes, erros de interfaces, erros em estruturas de dados ou acesso externo ao banco de dados, erros de comportamento ou desempenho e erros de inicialização e finalização do sistema.

Alguns dos pontos negativos dos testes baseados em especificação são que podem ocorrer redundâncias entre os casos de testes agravada pela possibilidade de partes do software que não foram testadas. E como os testes são baseados no comportamento especificado, é difícil de imaginar esses métodos identificando comportamentos que não são especificados (JORGENSEN, 2016).

2.2.2 Teste de caixa branca

O nome caixa branca vem da exigência do conhecimento de como o software é implementado. Para executar esta técnica, por exemplo, diferentes casos de testes podem ser derivados para um laço de repetição de um código, sendo independente da funcionalidade do software (GRAHAM; VAN VEENENDAAL; EVANS, 2008). Usando métodos de caixa branca é possível derivar casos de testes que garantem que todos os caminhos lógicos tenham sido executados pelo menos uma vez, como os lados verdadeiros e falso de uma decisão lógica, os laços de repetições e outras estruturas para garantir a sua validade (PRESSMAN, 2005). Com os conceitos de teoria de grafo é possível descrever exatamente o que será testado. Esta técnica serve também para a definição e uso de métricas de cobertura de teste, indicando até que ponto o software foi testado oferecendo assim um melhor gerenciamento de teste (JORGENSEN, 2016).

Entre os testes de caixa branca temos o teste de caminho ou *path testing*, onde um programa é derivado (transformado) em um grafo direcionado em que os nós são fragmentos de declaração e as ligações, chamadas de arestas, representam o fluxo de controle. Para derivar um programa procedural em grafo de fluxo podemos usar a notação da Fi-

gura 8 para cada uma das construções básicas da programação estruturada (JORGENSEN, 2016).

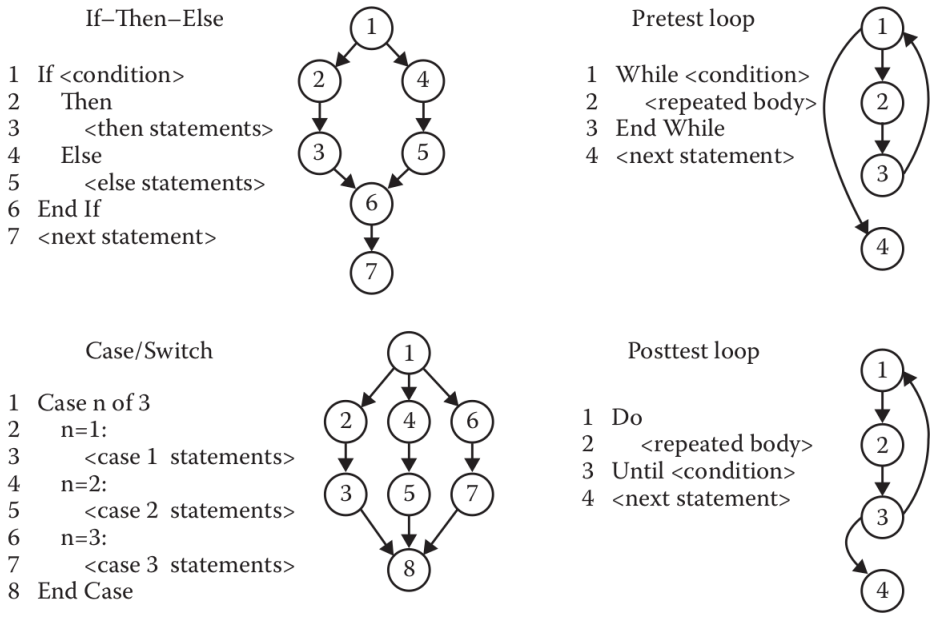


Figura 8: Grafos para as quatro estruturas básicas da programação procedural (JORGENSEN, 2016).

(JORGENSEN, 2016) define que um conjunto de casos de teste para um programa constituem a cobertura de nó. Quando executados no programa, cada nó no grafo é percorrido e constituem cobertura de aresta se, quando executado no programa, cada aresta do nó grafo for percorrida. (WINIKOFF; CRANEFIELD, 2014), em seu trabalho, apresentam um exemplo representado na Figura 9, onde existem 2 caminhos pelo programa (1, 2, 3, 5, 6) e (1, 2, 4, 5, 6). Um conjunto de testes para ser adequado deve ter pelo menos 2 casos de testes, um para exercitar cada caminho possível do programa.

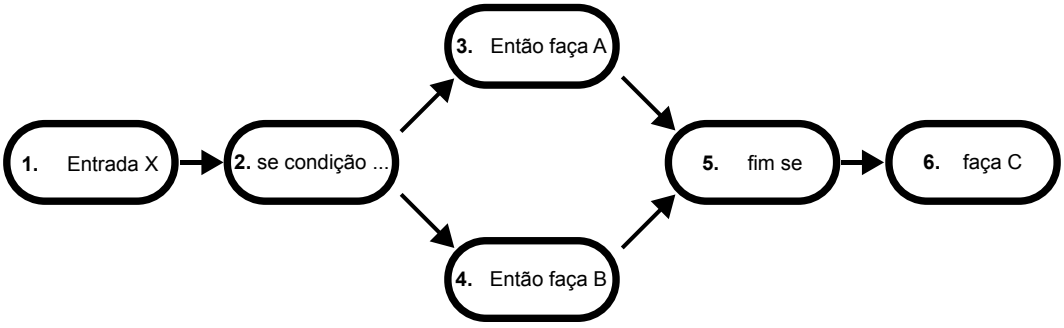


Figura 9: Exemplo de grafo de fluxo de um programa. Adaptado de (WINIKOFF; CRANEFIELD, 2014)

2.2.3 Testes de sistemas no ciclo de desenvolvimento de software

O projeto de testes para um software está diretamente relacionado ao modelo de ciclo escolhido para o desenvolvimento do sistema. Existem vários processos de desenvolvi-

mento de software. Entre eles, modelo cascata, que possui as fases de análise e definição dos requisitos, projeto do sistema, implementação, integração e testes e por último entrega, operação e manutenção do sistema. Este modelo é bastante rígido e em princípio uma nova fase só começa quando a anterior termina (SOMMERVILLE, 2010).

Existem outros modelos como o modelo iterativo, e modelos baseados em metodologias ágeis. Estes modelos especificam as várias etapas do processo e a ordem em que são realizados. A escolha depende dos objetivos e metas do sistema a serem desenvolvidos, mas sempre levando em conta que a qualidade e confiabilidade são os principais fatores (GRAHAM; VAN VEENENDAAL; EVANS, 2008).

Os níveis de teste de software do *V-Model* espelham o modelo cascata do ciclo de vida de desenvolvimento de software. Apesar deste modelo apresentar desvantagens, como ter um ciclo de *feedback* muito longo entre a especificação de requisitos e o teste do software e não prever suporte ao desenvolvimento paralelo no nível de unidade, ele é útil para identificar níveis distintos e esclarecer objetivos e responsabilidades para cada nível de teste. Uma variação do modelo cascata é apresentada na Figura 10 apresentando os níveis de testes relacionados a cada etapa do desenvolvimento (JORGENSEN, 2016).

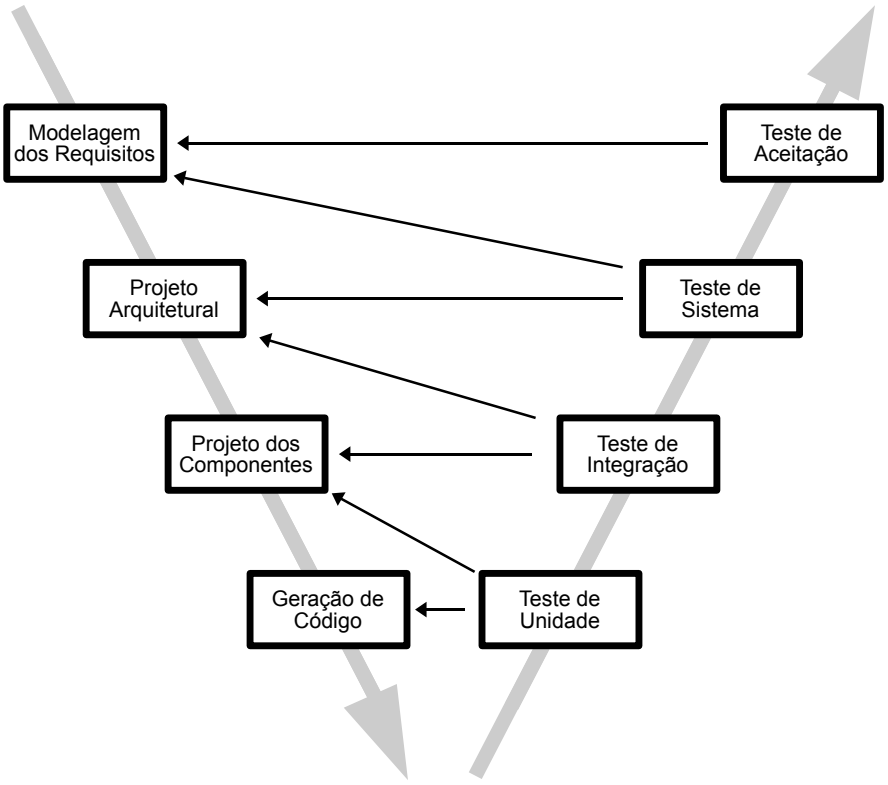


Figura 10: V-Model, adaptado de (PRESSMAN, 2005).

A grande contribuição do *V-Model* foi a orientação de que o teste precisa começar o mais cedo possível no desenvolvimento do projeto. As atividades de testes devem ser realizadas em paralelo com as atividades de desenvolvimento, e toda a equipe deve trabalhar em conjunto para que se possa identificar defeitos em decisões de projeto que de outra

forma, separadamente, dificilmente seriam percebidas e comunicadas. A identificação precoce de defeitos é, de longe, o melhor meio de reduzir o seu custo final (AMMANN; OFFUTT, 2016).

(GRAHAM; VAN VEENENDAAL; EVANS, 2008; AMMANN; OFFUTT, 2016) descrevem os níveis dos testes utilizados e seus objetivos:

- Teste de Aceitação: são projetados para determinar se o software atende aos requisitos e regras de negócio. Os testes são realizados com dados fornecidos pelo cliente do sistema e não com dados de teste simulados, e podem revelar erros e omissões na definição de requisitos do sistema, problemas de requisitos onde as instalações do sistema não atendem às necessidades do usuário ou desempenho do sistema inaceitável. Este nível de teste deve envolver o usuário ou alguém com conhecimento do domínio do sistema.

- Teste de Sistema: se preocupa com o comportamento do sistema e se ele está em conformidade com os requisitos definidos no projeto. Nesta etapa assume-se que os blocos de software estão funcionando de acordo, pois encontrar uma falha de nível inferior pode ter um grande custo para uma correção. Os testes geralmente são realizados por uma equipe específica para esta tarefa e não pelos programadores.

- Teste de Integração: projetado para avaliar se a interação entre as interfaces dos módulos funciona corretamente. É uma técnica sistemática para a construção da arquitetura de software, ao mesmo tempo que se investiga erros associados à interface. Uma das abordagens é construir o programa incrementalmente e cuidadosamente incluindo e testando os módulos de todos os componentes, assim erros são mais fáceis de isolar e corrigir. Geralmente é responsabilidade da equipe de desenvolvimento.

- Teste de Unidade: avalia as unidades produzidas na fase de implementação. É o nível de teste que verifica as menores unidades do software como por exemplo classes, objetos e funções. Os erros encontrados pelos testes são limitados ao escopo estabelecido para o teste unitário, concentrando-se na lógica do processamento interno. É comum os testes unitários serem empacotados juntamente com o código fonte e serem utilizados para testes automáticos em outros momentos.

2.3 Teste de Software em SMA

Os agentes e sistemas multiagentes possuem muitas peculiaridades, que tornam o processo de teste mais complexo e que devem abordar algumas questões que não eram preocupação no desenvolvimento de software orientado a objetos. Algumas dessas dificuldades foram citadas em (ROUFF, 2002; HOUHAMDI, 2011; NGUYEN, 2009) e são

apresentadas a seguir, listando as propriedades específicas de agente ou SMA e o que ela gera de dificuldade em relação aos testes:

- Distribuído/assíncrono: os agentes podem operar de forma simultânea e assíncrona, ou seja, um agente pode ter que aguardar que outros agentes cumpram seus objetivos, e que o contexto permita a execução de seus planos. Um agente pode funcionar corretamente isolado, mas incorretamente quando colocado em uma comunidade de agentes ou vice-versa.

- Autônomos: as mesmas entradas de testes podem resultar em comportamentos diferentes em diferentes execuções, uma vez que os agentes podem modificar sua base de conhecimento entre duas execuções ou podem aprender com entradas anteriores.

- Envio de mensagens: os agentes se comunicam através de envio de mensagens. As técnicas de teste tradicionais, envolvendo chamadas de método, não podem ser aplicadas diretamente.

- Fatores ambientais e normativos: o ambiente e convenções (normas, regras e leis) são fatores importantes que definem ou influenciam os comportamentos dos agentes. Alterações do contexto podem alterar os resultados do teste, e eventualmente, um contexto possui meios para que os agentes se comuniquem ou pode ser utilizado como uma entrada de teste.

- Agentes "selados": os agentes podem fornecer primitivas observáveis ou não ao mundo exterior, resultando em acesso limitado ao estado e ao conhecimento dos agentes internos. Um exemplo poderia ser um SMA aberto que permite que agentes de terceiros acessem os recursos do SMA, semelhante ao funcionamento de APIs. Nestes casos, é difícil garantir que estes agentes de terceiros com o conhecimento limitado sobre suas intenções tenham um comportamento apropriado.

Testar um único agente é diferente de testar uma comunidade de agentes. O teste de agentes pode ser feito de modo incremental durante o desenvolvimento testando funcionalidades à medida que elas são adicionadas. Alguns dos principais erros ao desenvolver agentes podem ser endereçar incorretamente uma mensagem para outro agente, enviar uma solicitação incorreta em uma mensagem ocorrendo que o agente receptor não reconheça a mensagem, analisar incorretamente mensagens recebidas, enviar uma mensagem ao agente errado, ou até mesmo não ter desenvolvido códigos do agente para que ele seja capaz de aceitar todas as mensagens (ROUFF, 2002). Outros testes a serem realizados são verificar se a comunicação com o ambiente está correta e se a aprendizagem está adequada.

Testar uma comunidade de agentes torna os objetivos de testes mais amplos, tendo que verificar se os agentes da comunidade trabalham juntos como projetado, atestando a

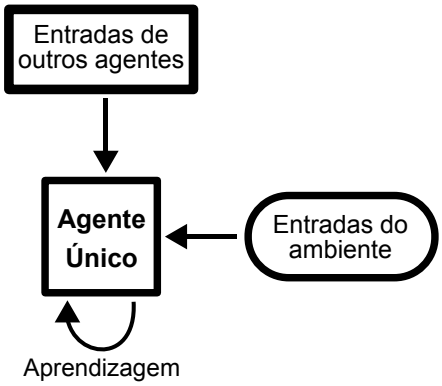


Figura 11: Teste em um Agente, adaptado de (ROUFF, 2002).

troca de comunicação entre eles, averiguando se essa troca de mensagem realmente está ocorrendo entre os agentes previstos e com o ambiente, mas agora com um agravante de ter um número muito maior de interações. (ROUFF, 2002) cita os erros mais observados em programadores desenvolvendo comunidades de agentes:

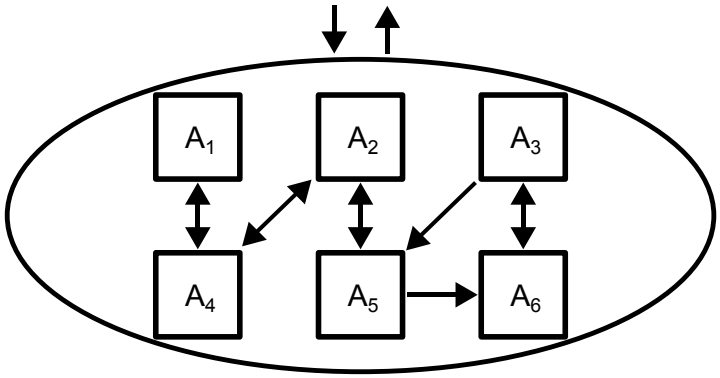


Figura 12: Teste em comunidade de agentes, adaptado de (ROUFF, 2002).

- Não documentar adequadamente as interações do agente, de modo que diferentes desenvolvedores implementam as interações de forma diferente.
- Erro na mensagem, não comunicando o remetente, o conteúdo, entre outros, nas mensagens do agente.
- Projetando impasses nas trocas de mensagens.
- Não implementando mudanças nas mensagens para todos os agentes ao mesmo tempo (o que dificulta o teste da comunidade).

Para uma melhor organização, assim como os testes no ciclo de vida de desenvolvimento de software tradicional, os testes nos SMA também possuem vários níveis. O teste de unidade possui o mesmo nome nas duas abordagens, já o segundo nível é o teste de

agente e não possui um relacionado. O teste de grupo se relaciona com o teste de integração, o teste de sociedade está relacionado com o teste de sistema. E por último e com o mesmo nome tem o teste de aceitação (HOUHAMDI, 2011).

Os testes de unidade certificam-se que as menores unidades que compõem o agente estejam funcionando de acordo com o que foi planejado. Essas unidades incluem os blocos de código que implementam as metas, planos, base de conhecimento, mecanismos de raciocínio, especificações de regras entre outros. Já os testes de agentes testam a integração destas pequenas unidades dentro do agente, verificando se os agentes são capazes de cumprir suas metas e se interagem corretamente com o ambiente. O teste de grupo avalia a interação entre agentes, protocolo de comunicação e semântica, integração com o ambiente, integração de agentes com recursos compartilhados, observa propriedades emergentes e comportamentos coletivos. Este teste certifica que um grupo de agentes e o ambiente funcionem corretamente juntos. O teste de sociedade avalia o SMA em execução no ambiente operacional de destino, testa as propriedades emergentes e macroscópicas esperadas do sistema. E o teste de aceitação testa o SMA no ambiente de execução do cliente e verifica se ele atende aos objetivos esperados das partes interessadas (NGUYEN, 2009).

2.4 Rede de Petri

Redes Petri (RP) é uma ferramenta gráfica e matemática para a descrição e análise de processos concorrentes, assíncronos e paralelos que surgem em sistemas distribuídos. Como ferramenta gráfica, ela pode auxiliar na comunicação visual como um fluxograma e sendo uma ferramenta matemática, é possível estabelecer equações de estados, equações algébricas e outros modelos matemáticos que regem o comportamento dos sistemas (MURATA, 1989).

Este modelo foi proposto por Carl Petri para modelar a comunicação entre autômatos, utilizados para representar sistemas a eventos discretos. Um sistema discreto é caracterizado pelas alterações de estados que ocorrem em instantes precisos. Os conceitos básicos para a modelagem de sistemas discretos são (CARDOSO; VALETTE, 1997):

- Eventos: são momentos de observação e de mudanças de estado do sistema.
- Atividades: são as caixas-pretas utilizadas para abstrair a evolução do sistema entre dois eventos.
- Processos: sequência de eventos e atividades.

A Figura 13 representa uma RP simples. O grafo da rede modela as propriedades estáticas de um sistema, assim como um fluxograma representa as propriedades estáticas de um programa de computador. O grafo contém dois tipos de nós: os círculos chamados

de *lugares* e as barras, chamadas de *transições*. Os nós são conectados por arcos direcionados de *lugares* para *transições* e de *transições* para *lugares* (PETERSON, 1977).

Além destes dois elementos, uma RP ainda possui propriedades dinâmicas que são resultantes da sua execução. O elemento que atribui esta propriedade dinâmica é a *ficha* (PETERSON, 1977). As fichas são indicadas por um ponto em um lugar. A *ficha* pode representar um recurso em uma posição, ou uma estrutura de dados que se manipula por exemplo (CARDOSO; VALETTE, 1997).

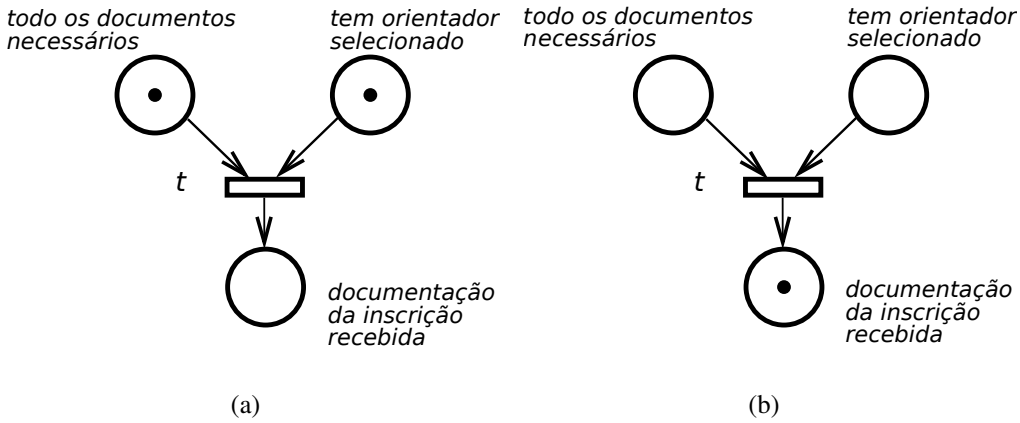


Figura 13: Rede de Petri

A dinâmica da RP atua da seguinte maneira. As fichas são movidas pelo disparo das transições associadas da rede. Na Figura 13(a) podemos observar que temos uma ficha no lugar *todos os documentos necessários* e outra ficha em *tem orientador selecionado*. A ocorrência do evento, associado à transição *t* só pode ocorrer se houver ao menos uma ficha em cada um destes lugares. O disparo da transição *t* retira uma das fichas vinculadas a cada um dos lugares de entrada e as posiciona no lugar de saída *documentação da inscrição recebida* Figura 13(b). A Figura 13 representa estados diferentes do mesmo sistema, uma evolução da rede, os arcos não apresentam nenhuma numeração indicando que o peso para disparar a transição é um, caso o peso para disparar a transição seja maior é necessário indicar o numeral no arco.

As definições formais para Rede de Petri e Rede de Petri Marcada segundo CARDOSO; VALETTE (1997) são:

Definição 1 (Rede de Petri)**.** Graficamente, uma Rede de Petri é uma n-tupla

$$R = \langle P, T, Pre, Post \rangle \tag{1}$$

onde:

- P é um conjunto finito de lugares de dimensão *n*;
- T é um conjunto finito de transições de dimensão *m*;

- $Pre : P \times T \rightarrow \mathbb{N}$ é a aplicação de *entrada* (lugares precedentes ou incidência anterior), com $\mathbb{N}$ sendo o conjunto dos números naturais;
- $Post : P \times T \rightarrow \mathbb{N}$ é a aplicação de saída (lugares seguintes ou incidência posterior).

Definição 2 (Rede marcada)**.** Uma rede marcada N é uma dupla

$$N = \langle R, M \rangle \tag{2}$$

onde:

- R é uma rede de Petri,;
- M é a marcação inicial dada pela aplicação;

$$M : P \rightarrow \mathbb{N} \tag{3}$$

Para modelar sistemas complexos, onde por exemplo várias máquinas usam recursos diversos e podem trabalhar paralelamente fabricando peças diferentes, é necessário dividir em várias RP ordinárias. Por não ter como diferenciar recursos e peças, já que as fichas são indiferenciáveis por obter apenas valores inteiros, as RP podem modelar o comportamento geral sem identidade de cada processo, faltando informação, ou modelar cada um dos processos individualmente e então modelar a interação entre eles, podendo ser muito trabalhoso ou tornando até mesmo ilegível (CARDOSO; VALETTE, 1997; JENSEN, 2013).

Extensões para as RP ordinárias foram propostas para modelar sistemas complexos, cada uma com suas características, estes modelos são chamados de Redes de Petri de Alto Nível (RPAN). Neste trabalho o foco será nas Redes de Petri Coloridas (RPC), exposta na subseção seguinte.

2.4.1 Redes de Petri Coloridas

O benefício das RPC em relação as RP ordinárias é a possibilidade de utilização de marcas individualizadas (cores), que pode representar diferentes processos ou recursos de uma rede, permitindo a redução do tamanho de modelos (MACIEL; LINS; CUNHA, 1996). O uso dos conjuntos de cores nas RPC é semelhante aos tipos de dados das linguagens de programação (inteiro, real, caractere, lista) (JENSEN, 2013).

Nas RPC, cada lugar está associado a um conjunto de cores das fichas que podem pertencer a este lugar, cada transição se associa um conjunto de cores que podem disparalá e os arcos não possuem apenas o peso (valor inteiro), é necessário descrever quais cores de fichas serão retiradas do local inicial e quais cores de fichas serão colocadas nos lugares de saída.

Segundo (MACIEL; LINS; CUNHA, 1996), as RPC são compostas por três partes distintas.

- Estrutura: é um grafo dirigido composto por vértices do tipo lugar, representado graficamente por círculos ou elipses e por vértices do tipo transição representado retângulos.
- Declarações: são as especificações dos conjuntos de cores e declaração das variáveis.
- Inscrições: os lugares possuem as inscrições nome, conjunto de cores e expressão de inicialização, estas são as fichas iniciais. As transições possuem as inscrições de nome e guarda que são restrições impostas pela transição. E os arcos possuem a inscrição de expressão, representando a ficha que será deslocada.

Para a modelagem das RPC será utilizada a ferramenta CPN Tools.

2.4.2 CPN Tools

CPN Tools é uma ferramenta para editar, simular e analisar RPC hierárquicas e temporais, ela possui uma interface flexível, técnicas de interação detalhadas e retornos gráfico que informam ao usuário sobre o status das verificações, simulações e sintaxes (RATZER et al., 2003). Na Figura 14 é apresentada a interface gráfica da ferramenta com um exemplo de RPC.

A interface do programa possui um menu à esquerda. O item *Tool Box* possui as caixas com as ferramentas para utilização do sistema, é possível colocar as caixas na área de trabalho clicando nelas e arrastando até o local desejado. Na Figura 14 é possível observar, à direita, duas caixas, uma delas com a aba *Sim* selecionada mostrando as ferramentas de controle da simulação da RPC, e a aba *View* em segundo plano, a outra caixa tem a aba *Create* selecionada com as opções de criação da modelagem da RPC, e as abas *Style* e *Net* ambas em segundo plano, responsáveis respectivamente por estilização da rede e por operações de gerenciamento de arquivo como salvar a rede atual.

O item RPC.cpn do menu é o nome do arquivo e dentro dele o item *Declarations* se destaca, é no subitem *Standard Declarations*, os conjuntos de cores *UNIT*, *INT*, *STRING*, entre outras já são tipos declarados por padrão, e é possível definir outros tipo de declarações. Neste exemplo o código abaixo exibe por completo as declarações.

```
1 colset Role = with candidato | secretario | membro | presidente;
2 colset ROLES = list Role;
3 var c, s, m, p: ROLES;
```

É definido um conjunto de cores *Role* que possui quatro valores possíveis (candidato, secretario, membro, presidente), um conjunto de cores *ROLES* que é uma lista de *Role*, ou seja, aceita mais de uma cor por vez e a declaração das variáveis c, s, m, p to tipo *ROLES*.

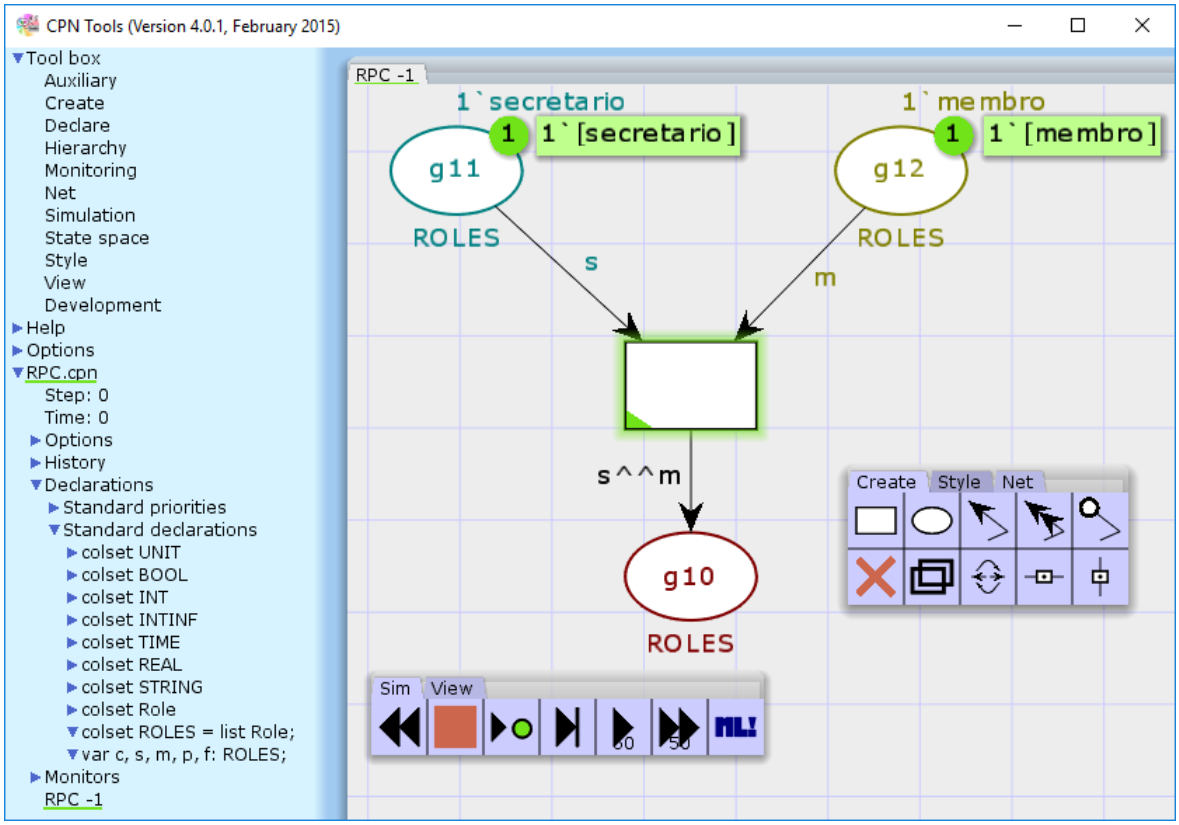


Figura 14: Interface do CPN Tools.

À direita é encontrada a área para modelagem da rede e o local onde as ferramentas de interação podem ser posicionadas. A rede modelada possui três *lugares* cada um com a inscrição de seu nome $g11$, $g12$ e $g13$, todos inscritos com o conjunto de cores *ROLES* e os lugares $g11$ e $g12$ possuem expressão de inicialização, o primeiro uma ficha *secretario* e $g12$ uma ficha *membro*. Os lugares estão estilizados com cores diferentes mas esta configuração é apenas para melhor visualização da rede.

Os arcos possuem a inscrição de expressão que determina as fichas que irão sair dos lugares iniciais e que irão para o lugar final, a expressão $s-$ significa que a transição só vai ser ativada quando chegar na transição uma variável s e e outra m e ao sair da transição elas se tornarão uma lista. A transição $T1$ possui apenas a inscrição de nome, a borda e canto inferior esquerdo verde é um retorno que o CPN Tools dá de que a transição está ativada, ou seja, tem todos os requisitos para ser disparada.

Nativamente RP no CPN Tools não possui uma avaliação para contagem de caminhos, a solução para mensurar quantos caminhos tem uma RPC é avaliá-la como um grafo, construindo um grafo de marcações acessíveis. Os grafos possuem diversos algoritmos para contagem de caminhos, menor caminho, entre outros, esta solução foi desenvolvida para este trabalho.

As RP são grafos direcionados onde $G = (V, E)$ consiste em um conjunto V de nós e um conjunto E de arcos tal que cada arco $e \in E$ é associado a um par de nós não

ordenados. Os caminho em grafos tem a seguinte definição de: considere v_o e v_n nós de um grafo. Um caminho de v_o a v_n de comprimento n é uma sequência alternada de n+1 nós e n arcos começando em v_o e terminando em v_n (CARDOSO; VALETTE, 1997).

3 TRABALHOS RELACIONADOS

A busca por trabalhos relacionados teve início em novembro de 2016 até julho 2017. As palavras chaves *verification and validation of multi-agent system*, *agent systems verification* foram o ponto inicial da pesquisa na busca de encontrar trabalhos relatando formas de garantir a execução correta de SMA. Com o avanço da pesquisa novos termos foram considerados, *"testing multi-agent system"* e *assurance agent system*.

Um novo entendimento sobre o caminho da pesquisa foi ao entender que antes de testar um software é necessário saber o quanto testá-lo para garantir uma cobertura suficiente do sistema. Foi então que a a palavra chave *testability agent system*. Outras palavras chaves foram relacionadas para a utilização em conjunto de redes de petri e SMA: *petri nets multi-agent system*, *agent system petri nets testing*.

No trabalho *Multi-agent system testing: A survey* HOUHAMDI (2011) faz uma investigação e descreve os trabalhos relacionados ao tema de teste em agentes. A separação em níveis de teste facilita a compreensão, também acontece de alguns trabalhos citados ocuparem mais de um nível de teste. A Tabela 3 mostra a relação de trabalhos apresentados neste artigo e seus respectivos níveis. Mais detalhes sobre cada nível de teste pode ser visto na seção 2.3.

O autor conclui que os trabalhos em testes em SMA se concentram principalmente no nível de agente e integração e expõe áreas de investigações de teste que necessitam de atenção como: desenvolver um processo de teste completo para SMA, testes no nível de aceitação, testar propriedades emergentes no nível de sistema macroscópico, métricas para avaliar a qualidade dos testes em SMA e reduzir ou remover os efeitos colaterais na execução e monitoramento dos testes.

Em (ATHAMENA; HOUHAMDI, 2012), os autores propõem uma abordagem baseada em RP para o teste de comportamento de SMA. Para isso, modelos de análise e projeto criados com base na metodologia MaSE são convertidos em um modelo UML 2.0 padrão, e então os modelos UML são transformados em RP para testes formais. A Figura 15 apresenta as atividades da abordagem.

Neste trabalho os autores desenvolveram um modelo para a transformação dos diagrama de classe de agente e diagrama de função do MaSE, para o diagrama de sequência

Tabela 3: Relação entre níveis de teste e trabalhos relacionados (HOUHAMDI, 2011).

Tipo de Teste	Descrição	Trabalhos
Teste de Unidade	Teste nos menores blocos de construção de um SMA.	(ZHANG; THANGARAJAH; PADGHAM, 2007)
		(EKINCI et al., 2009)
Teste do Agente	Teste de integração dos diferentes módulos dos agentes.	(CAIRE et al., 2004)
		(LAM; BARBER, 2004)
		(NÚÑEZ; RODRÍGUEZ; RUBIO, 2005)
		(COELHO et al., 2006)
		(GOMEZ-SANZ et al., 2008)
		(HOUHAMDI, 2011)
Teste de Integração	Testa a interação entre agentes e a interação destes com o ambiente.	(KNUBLAUCH, 2002)
		(BOTÍA; LÓPEZ-ACOSTA; SKARMETA, 2004)
		(PADGHAM; WINIKOFF; POUTAKIDIS, 2005)
		(RODRIGUES et al., 2005)
		(EKINCI et al., 2009)
		(NGUYEN; PERINI; TONELLA, 2009)
		(HOUHAMDI; ATHAMENA, 2011)
Teste de Sistema	Testa o SMA no ambiente de operação.	(SUDEIKAT; RENZ, 2008)
		(HOUHAMDI; ATHAMENA, 2011)
Teste de Aceitação	Verifica se atende aos objetivos do cliente.	Sem trabalhos relacionados

do modelo UML 2.0, e então propuseram um modelo para transformar os diagramas de sequência em RP equivalentes e enfim podem ser aplicados as técnicas de testes automáticos no SMA.

Segundo os autores, as principais contribuições do trabalho foram propor um processo de teste completo e abrangente para o SMA e reduzir os efeitos colaterais na execução e monitoramento do teste, não utilizando agentes testadores ou agentes oráculos, o que pode influenciar no comportamento dos agentes ou desempenho do sistema.

No estudo de (WINIKOFF; CRANEFIELD, 2014), que avaliaram o quão difícil é testar um programa de agente na arquitetura BDI. Para isso, os autores utilizam a técnica de testes de caixa-branca, que é baseada no fluxo de controle, um critério básico e muito longo para avaliar a adequação de um conjunto de testes em que todos os caminhos (*All-Path*) do programa sejam cobertos. O motivo da escolha de utilizar todos os caminhos é que os SMA geralmente envolvem ambientes não-episódicos, sendo que o comportamento de um determinado plano ou meta é geralmente sensível ao histórico do agente,

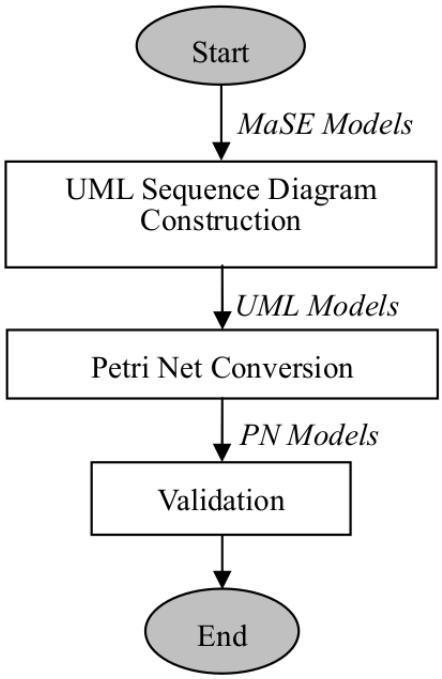


Figura 15: Fluxograma do projeto de teste no SMA (ATHAMENA; HOUHAMDI, 2012).

precisando assim considerar as diferentes histórias possíveis.

Neste trabalho, para analisar o processo de execução BDI é utilizada uma visão declarativa. Os eventos e planos podem ser visualizados como uma árvore, em que cada objetivo tem como filhos as instâncias do plano que são aplicáveis a ele, e cada instância do plano tem como filhos os sub-objetivos que ele publica. Esta forma de visualização facilita a análise do número de caminhos através de um programa BDI.

Para a geração de árvore um programa em Prolog foi implementado, onde uma árvore de plano-meta é representada por termos Prolog com a gramática apresentada na Figura 16, onde árvore de plano-meta (*Goal-Plan Tree*) é representado por GPT, o AoGL abreviou lista de ação ou meta (*Action or Goal List*) e A é um símbolo. Por exemplo,ç a Figura 17 mostra a árvore de plano-meta simplificada modelada pelo termo Prolog $goal\left(\left[plan\left(\left[act\left(a\right)\right]\right), plan\left(\left[act\left(b\right)\right]\right)\right]\right)$.

$$
\begin{aligned}
\langle GPT\rangle &::= goal\left(\left[\right]\right) | goal(\left[\langle PlanList\rangle\right]) \\
\langle PlanList\rangle &::= \langle Plan\rangle \mid \langle Plan\rangle, \langle PlanList\rangle \\
\langle Plan\rangle &::= plan(\left[\right]) | plan(\left[\langle AoGL\rangle\right]) \\
\langle AoGL\rangle &::= act(A) | \langle GPT\rangle | act(A), \langle AoGL\rangle \mid \langle GPT\rangle, \langle AoGL\rangle
\end{aligned}
$$

Figura 16: Termos Prolog (WINIKOFF; CRANEFIELD, 2014).

Com isso em vez de ver a execução de um programa BDI como um processo, o programa é visualizado como uma transformação de dados de uma árvore de planos-metas (finita) em uma sequência de execuções de ações. Assim, a questão de quão grande é o

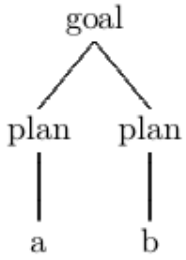


Figura 17: Árvore de plano-meta (WINIKOFF; CRANEFIELD, 2014).

espaço de comportamento para os agentes BDI é respondida derivando fórmulas que permitem calcular o número de comportamentos, bem-sucedidos e mal sucedidos, ou seja, falhou para uma determinada planta de plano de meta.

Após testes em modelos de execução BDI abstratos, os autores verificaram se os dados obtidos se aplicavam também à modelos reais. A aplicação real escolhida foi o trabalho de (BURMEISTER et al., 2008) que preenchia os requisitos para a comparação. Os valores obtidos são apresentados na Tabela 4 onde $n^{\checkmark}(g)$ são os caminhos em que os testes são bem sucedidos, $n^{\chi}(g)$ são os caminhos onde os teste falham e ℓ são ações antes, depois e entre sub-metas em um plano.

Tabela 4: Resultado do teste realizado em uma aplicação BDI (WINIKOFF; CRANEFIELD, 2014)

Árvore com 57 metas	Sem manipulação de Falha		Com manipulação de Falha	
	$n^{\checkmark}(g)$	$n^{\chi}(g)$	$n^{\checkmark}(g)$	$n^{\chi}(g)$
$\ell = 4$	294,912	3,250,604	$\approx 2.98 \times 10^{20}$	$\approx 9.69 \times 10^{20}$
$\ell = 2$	294,912	1,625,302	$\approx 6.28 \times 10^{15}$	$\approx 8.96 \times 10^{15}$
$\ell = 1$	294,912	812,651	$\approx 9.66 \times 10^{11}$	$\approx 6.27 \times 10^{11}$

(WINIKOFF; CRANEFIELD, 2014) concluíram que um teste completo em um sistema BDI não é viável. O tamanho de espaços de comportamento é muito grande e ainda se torna significativamente maior quando o sistema tem suporte à manipulação de falhas. Os autores também chegaram à conclusão de que o mecanismo de recuperação de falhas é eficaz para alcançar uma baixa taxa de falha real. E novas abordagens foram propostas para lidar com a testabilidade do sistema.

O trabalho (WINIKOFF, 2017) retorna com o objetivo de avaliar se é possível obter garantias em um sistema multiagente através de testes, verificando a testabilidade de um programa, dando continuidade ao trabalho publicado anteriormente (WINIKOFF; CRANEFIELD, 2014), mas utilizando novas métricas. A testabilidade de um programa é um métrica que indica o esforço necessário para testar adequadamente um programa. Este trabalho tem por objetivo quantificar quantos testes são necessários para testar um um

programa de agente BDI para satisfazer um critério, considerando o critério de adequação do teste de todas as arestas, que é considerado como o mínimo geralmente aceito (JORGENSEN, 2016).

Uma das grandes contribuições dos autores é que a análise é genérica permitindo a aplicação a todos os programas que utilizam a arquitetura BDI. Para esta análise, equações são derivadas para chegar a conclusão de quantos casos de teste (caminhos) são necessários para cobrir todas as arestas no gráfico de fluxo de controle correspondente a um determinado programa BDI.

A Figura 18 apresenta um exemplo de um grafo de controle de fluxo de um programa BDI, este programa tem início em "S" e possui quatro ações $\alpha_1, \alpha_2, \alpha_3$ e α_4. Caso alguma das ações forem bem-sucedidas então o programa executa "Y" e é encerrado em "E", concluindo-se com sucesso. Se a ação α_1 falhar ela avança para ação α_2 que pode ter sucesso ou a falha pode ocorrer novamente e assim a próxima ação seria executada. Este programa exigiria 5 testes para cobrir todas as bordas, um teste é onde as quatro ações falham ($S \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4 \rightarrow N \rightarrow E$) e os outros 4 testes é para quando uma ação é bem-sucedida e a anterior falhou.

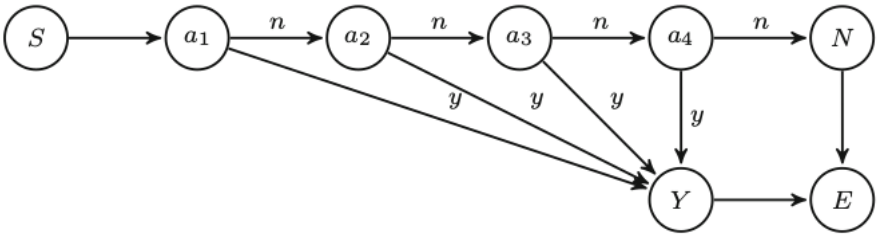


Figura 18: Controle de Fluxo (WINIKOFF, 2017).

Para derivar equações que calculem o menor número de caminhos exigidos de um programa começando em S para chegar em E é necessário descobrir quantos destes caminhos são bem-sucedidos (passando por Y) e quantos falharam (passando por N). Os autores definiram $p(P)$ como o número de caminhos necessários para cobrir todas as arestas do grafo de fluxo de controle correspondente ao programa P, $y(P)$ para os caminhos que vão por Y e $n(P)$ para os caminhos que vão por N, então $p(P) = y(P) + n(P)$.

Logo após, os autores consideram $P_1; P_2$, onde um subprograma P_1 é colocado em sequência com P_2 (Figura 19). O subprograma P_1 requer $p(P_1)$ testes para cobrir todas as arestas com $n(P_1)$ testes levando até a falha e $y(P_1)$ levando à uma execução com sucesso.

A partir do exemplo da Figura 19, diversas equações são derivadas para os diferentes casos. Estas equações servem para determinar quantos testes são necessários para garantir uma cobertura adequada em relação ao critério todas as arestas. Então, os autores implementam as equações em um programa Prolog que calcula os valores de p(p), y(P) e n(P) para qualquer programa BDI. A Tabela 5 contém a comparação dos resultados do trabalho anterior dos autores (WINIKOFF; CRANEFIELD, 2014) com os novos resultados

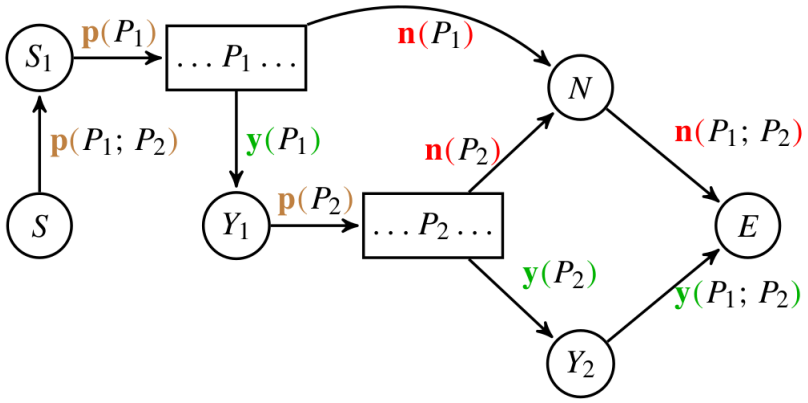


Figura 19: Controle de Fluxo de $P_1; P_2$ (WINIKOFF, 2017).

encontrados.

Tabela 5: Comparação entre os resultados dos trabalhos (WINIKOFF; CRANEFIELD, 2014; WINIKOFF, 2017)

d = j = k = 3	Todos Caminhos		Todas Arestas			
	$n^{\checkmark}(g)$	$n^{X}(g)$	p(g)		p(g)	
			Rel.	Aplic.	Rel.	Aplic.
62 (j = k = 2)	6.33×10^{12}	1.82×10^{13}	141	78	85	64
363	1.02×10^{107}	2.56×10^{107}	6391	2961	469	378
776 (j = 2, d = 4)	1.82×10^{157}	7.23×10^{157}	1585	808	1037	778
627 (k = 4)	3.13×10^{184}	7.82×10^{184}	10,777	4767	799	642

Sendo: *d* é profundidade da árvore planos-meta, *j* as instâncias de planos aplicáveis e *k* as submetas. Para *Todos Caminhos* $n^{\checkmark}(g)$ e $n^{X}(g)$ fazem referência ao número de testes que tiveram sucesso e testes que falharam respectivamente. Para *Todas Arestas* o primeiro caso, p(g), é para os testes com tolerância a falhas em uso e o segundo caso é para quando a tolerância a falhas está desativada. As colunas com *Rel.* e *Aplic.* são onde os planos associados a um objetivo são, respectivamente, os planos relevantes e os planos aplicáveis

(WINIKOFF, 2017) concluiu que o número de testes necessários para *Todas Arestas* é muito menor que para a abordagem de *Todos Caminhos*, encontrando resultados onde é possível realizar os testes na prática. Outra conclusão é que permitir o tratamento de exceções não fez diferença significativa no número de testes como pode ser observado na Tabela 5.

Outras buscas foram realizadas com o objetivo de encontrar relação entre RP e SMA, e diversos trabalhos fazem essa associação. Em (KÖHLER; MOLDT; RÖLKE, 2001) RPC executáveis foram utilizadas para modelar a estrutura e o comportamento dos agentes. WEYNS; HOLVOET (2002) relataram como principais motivos da utilização de RPC como ferramenta de modelagem a visão conceitual clara sobre os agentes e o ambiente e o ótimo suporte a verificação e formalização.

Em (BAI; ZHANG; WIN, 2004), os autores apresentaram uma abordagem baseada

em RPC para formar protocolos de interação flexíveis entre agentes. No trabalho (ALMEIDA et al., 2004) é introduzido um modelo formal para verificar os planos em um SMA, baseadas na modelagem, simulação e verificação do modelo de RP Coloridas Hierárquicas (RPCH). (POUTAKIDIS et al., 2009) apresentam ferramentas para geração de casos de teste para de teste de unidade e outra para depuração e monitoramento de sistemas de agentes em execução.

GONCALVES (2010) expõe um modelo de RP desenvolvida para especificar o conhecimento em agentes e SMA, independentemente de estruturas e formalismos de representação do conhecimento. (MILLER; PADGHAM; THANGARAJAH, 2011) especifica, e mostra como medir, o grau de detalhes de um conjunto de casos de teste através de um agente de deputação que age como um oráculo, para avaliar a correção de um teste e utilizar a representação da RP do agente como suporte para medidas de cobertura de teste.

Neste capítulo, trabalhos relacionados ao teste de software em SMA foram apresentados. As abordagens dos trabalhos, na grande maioria, tratam apenas do nível de agentes ou na testabilidade em nível de agentes. O nível de organização de agentes é uma evolução natural no desenvolvimento de SMA.

4 APRESENTAÇÃO DO MÉTODO

A Figura 20, representa o fluxograma das etapas do processo para a avaliação da testabilidade de um SMA utilizando $\mathcal{M}$oise, através da contagem de caminhos da RPC modelada do sistema. O método é composto por cinco etapas, são elas:

- Declaração: etapa onde são definidas as cores utilizadas na modelagem.
- Estrutura: modelagem inicial da RPC.
- Inscrições: definição das inscrições da RPC.
- Falha: inclusão dos caminhos das falhas na RPC.
- Caminhos: contagem dos caminhos

Figura 20: Etapas do processo.

As etapas são descritas em sessões neste capítulo. Antes é apresentado como saindo dos trabalhos (WINIKOFF; CRANEFIELD, 2014) e (WINIKOFF, 2017) é possível fazer a conexão com a testabilidade de SMA utilizando RP.

Em (WINIKOFF; CRANEFIELD, 2014) um grafo de controle de fluxo foi utilizado para avaliar a testabilidade de um sistema BDI. A proposta deste trabalho é avaliar a testabilidade de um SMA utilizando o $\mathcal{M}$oise$^+$ como modelo de organização e empregando RP como ferramenta de descrição e análise.

Partindo como base o trabalho de (WINIKOFF, 2017) o grafo de controle de fluxo apresentado na Figura 18 será utilizado para apresentar o método de transformação de controle de fluxo para RP. Na conversão de controle de fluxo para RP é considerado que cada nó de ação, a minúsculo, será transformado em um lugar, A maiúsculo, e as ligações entre os nós, as arestas, são convertidas em transições que são conectados aos lugares por arcos. Logo o nó **S** que é o início do controle de fluxo é transformado no lugar **S**. Os lugares **S** e **A1** são conectados por dois arcos e pela transição **t0**, que é a transição disparada ao começar o programa.

O nó **a1**, da Figura 18, está ligado a outros dois nós em uma estrutura de divisão de caminhos, onde uma das arestas significa a falha e liga com o nó **a2** e outra quando a ação tem sucesso acessando o nó *Y* que representa que o programa foi realizado com sucesso. Para a transformação na RP é necessário incluir a transição **t1**, que é disparada quando a ação **a1** falha, unindo o lugar **A1** a **A2**, e também a transição **t6** unindo o lugar **A1** ao lugar **Y**, transição que é disparada quando a ação **a1** é bem sucedida. Esta primeira etapa é apresentada na Figura 21.

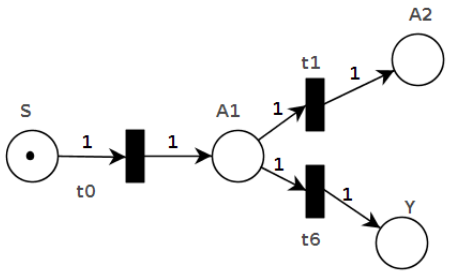


Figura 21: RP inicial.

Como pode ser visto na Figura 22 a dinâmica da RP inicia pela indicação da *ficha* no lugar **S**, quando a transição **t0** é disparada a ficha é movida para o lugar **A1**. Os arcos possuem o valor 1, significando que uma ficha já tem a capacidade de disparar aquela transição.

Em **A1** há uma estrutura de divisão entre diferentes sequências, onde e a transição **t6** é disparada quando é possível realizar a ação, indo assim para um lugar **Y** que corresponde à execução bem sucedida do programa, e finalmente para **E** que é o encerramento do programa. A transição **t1** é acionada quando o agente não consegue executar a ação

a1, direcionando assim para o lugar **A2**, esta ação pode ser bem sucedida, indo para a transição **t7**, ou falhar e ir para a transição **t2**, operando semelhante ao apresentado no *lugar* **A1**.

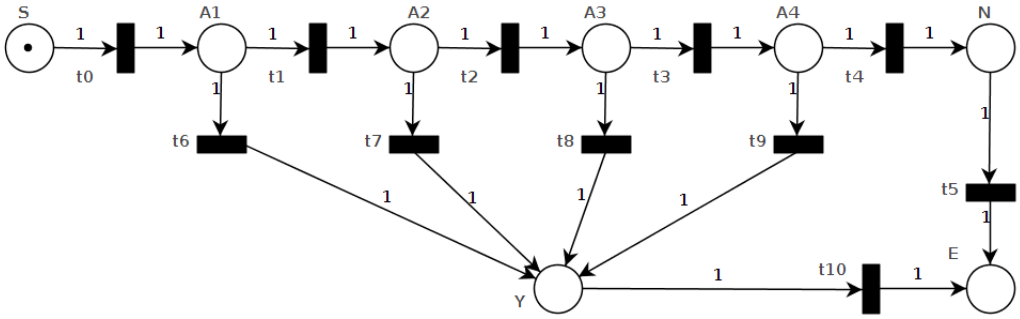


Figura 22: RP do grafo de controle de fluxo.

No método de (WINIKOFF; CRANEFIELD, 2014; WINIKOFF, 2017), a avaliação de testabilidade é baseada na árvore de planos-metas, onde os objetivo tem como filhos os plano que são aplicáveis a ele, e cada instância do plano tem como filhos os sub-objetivos que ele publica, mas ainda apenas no nível de metas e ações. No contexto deste trabalho é necessário avaliar em múltiplos níveis que não estavam previsto em outros trabalhos, como missões, papéis e metas.

Em (WINIKOFF; CRANEFIELD, 2014) a árvore de planos-metas é o pilar para seu trabalho. Neste, a avaliação da testabilidade no $\mathcal{M}$oise será baseada no esquema social da especificação social, que é uma árvore de decomposição de metas. Para realizar a conversão do esquema social para RP é necessário convencionar a relação dos operadores apresentados na legenda da Figura 5 e sua RP equivalente.

- Operador *sequência*: representa uma cadeia de metas em sequência. A meta **G10** inicia a sequência, caso a transição **t1** for iniciada a ficha é retirada de G10 e é colocada em **G13**, permitindo assim o seguimento da cadeia.

- Operador *escolha*: apenas uma das metas, **G8** ou **G9** podem ser adotadas, para isso o lugar **G7** recebe uma restrição, quando uma transição for disparada e a ficha for inserida no lugar **G7**, a outra transição será desabilitada, não permitindo assim que a outra meta seja realizada.

- Operador *paralelismo*: significa que as metas **G5** e **G6** podem progredir paralelamente e de modo assíncrono, mas a transição **t1** só será disparada quando as duas metas estiverem concluídas.

Com a relação para a transformação dos operadores do esquema social para RP é possível criar a estrutura básica da rede, onde agora cada meta será um *lugar*, semelhante ao que foi realizado na Figura 8 onde cada ação foi transformada em um *lugar*.

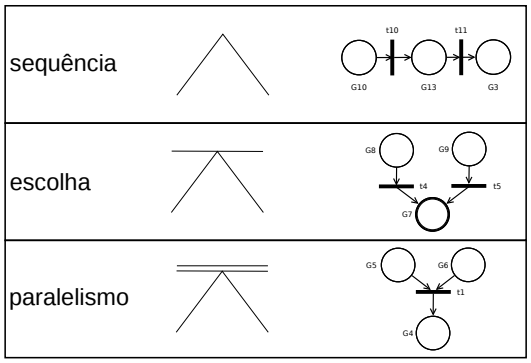


Figura 23: Relações dos operadores para Rede de Petri.

Para atender ao nível organizacional é necessário utilizar recursos que não são disponíveis nas RP ordinárias, mas que podem ser modelados através de RP Coloridas. No método, as fichas coloridas são etiquetas empregadas na representação dos papéis da organização. Os lugares se associam ao conjunto de cores das fichas, ou seja, sendo os lugares metas. Então cada lugar terá a cor associada ao conjunto de papéis relacionados e de acordo com a especificação deôntica é possível correlacionar com a missão que representa aquele lugar.

Os arcos que ligam os lugares as transições recebem agora uma variável de cor (*colset*) com valor pertencente ao papel, descrevendo qual cor está saindo do lugar e qual cor estará indo para o lugar de saída. Para o exemplo da Figura 5 deve ter um *colset* que receberá os papéis, um *colset* papéis, que será uma lista de papéis, e as variáveis que retratam as cores das transições. O código para declarar as cores na ferramenta CPN Tools encontra-se abaixo.

```
1 colset Role = with candidato | secretario | membro | presidente;
2 colset ROLES = list Role;
3 var c, s, m, p: ROLES;
```

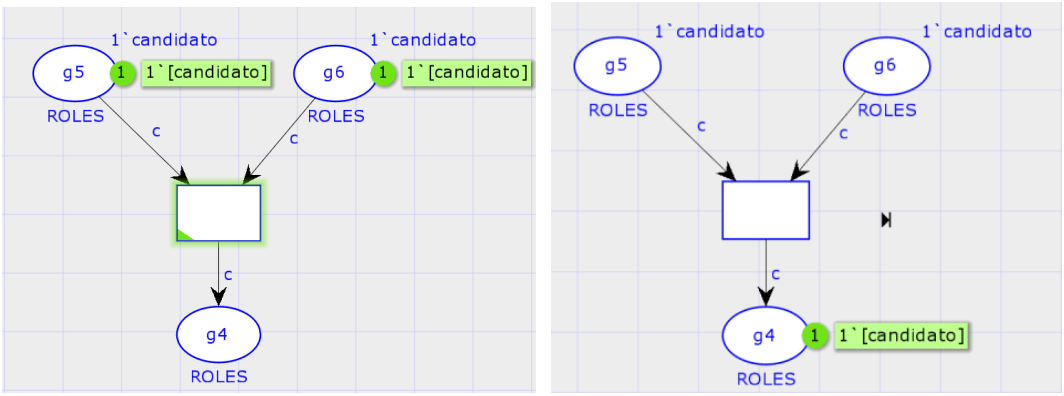
A linha 1 declara o *colset* Role, que deverá conter todos os papéis presentes no modelo $\mathcal{M}$oise$^+$ utilizado. A linha 2 é declarada ROLES que é uma lista de papéis (*Role*), ela será a cor padrão dos lugares que em alguns casos recebem mais de uma ficha, ou então mais de um papel. E *var* são as variáveis que representarão os papéis *ROLES* nos arcos, limitando a maneira de como disparar as transições e quais cores devem ser colocadas nos lugares de saída.

Após inserir as declarações padrões para as novas cores é possível criar a RPC na ferramenta CPN Tools. Para o ES da Figura 5 começando do canto esquerdo inferior da árvore. A modelagem da RPC pode ser dividida em 3 etapas.

1. Definição da estrutura da RPC: como primeiro conjunto de metas tem-se $g5$, $g6$ e $g4$ ligadas pelo operador paralelismo, logo utilizamos a Figura 23 para escolher a estrutura da RP para este conjunto de metas.

2. Inscrições da RPC: como todas as metas são da mesma missão, todos os lugares receberão o mesmo conjunto de cor (ROLES), e sendo a missão $m1$ papel do candidato, os lugares iniciais recebem a ficha candidato. E as transições recebem c como inscrição representante dos candidatos.

O resultado pode ser visto na Figura 24. Na Figura 24(a) é apresentada a RPC com as fichas de candidato nos lugares iniciais. É possível observar que a transição está ativada pela marcação verde que há nela, sendo assim todos os requisitos para disparar estão contemplados. Na Figura 24(b) a transição foi disparada, ela está desativada e o lugar $g4$, lugar final, recebeu uma ficha, que é o valor de saída da transição mostrado pela variável c do arco. Este exemplo mostra que, apesar de ter duas fichas candidato nos lugares iniciais, e apenas uma no lugar final, simboliza que o mesmo agente candidato se comprometeu com as metas $g5$ e $g6$.

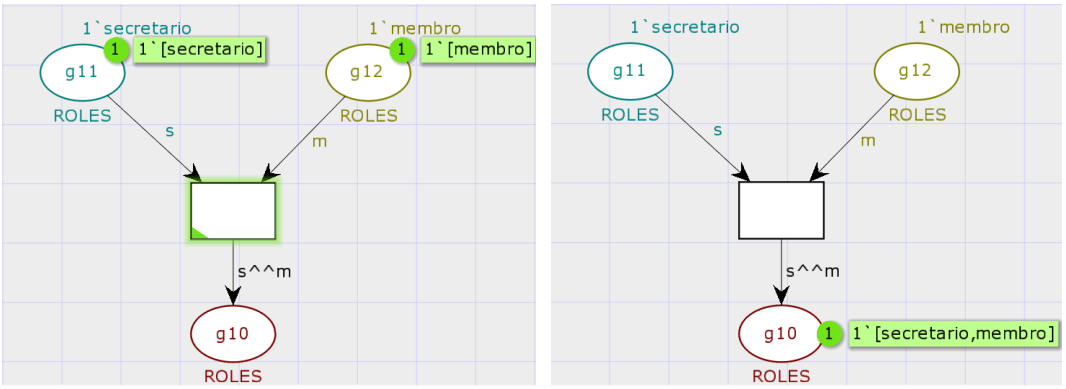


(a) RPC (b) RPC com transição disparada

Figura 24: RPC das primeiras metas do ES

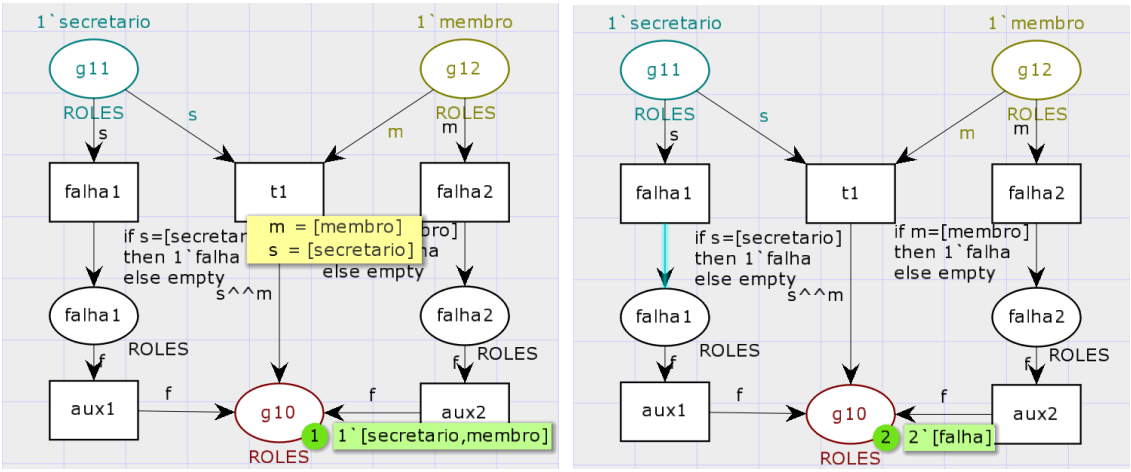
Na Figura 25, a RPC representa as metas $g11$, $g12$ e $g10$ da ES. Este conjunto de metas tem diferentes missões. A meta $g11$ em Verde-azulado (cerceta) é da missão $m3$ e recebeu a ficha "secretario", a meta $g12$ em oliva recebeu a ficha "membro"e faz parte da missão $m4$ e por último a meta $g10$, em vermelho escuro, faz parte da missão $m5$, representa a conclusão das metas $g11$ e $g12$, e é de responsabilidade do presidente. A opção de manter as fichas "secretario"e "membro"na missão $g10$ foi para simbolizar que um agente pode delegar as metas para outros papéis realizarem, e o responsável pela meta ter apenas que garantir que ela seja concluída.

Para completar a modelagem da RPC, deve-se incluir transições que serão ativadas quando uma meta falhar. É necessário incluir uma transição e um lugar de falha para cada lugar até então do modelo, uma transição auxiliar é necessária para realizar a ligação com o lugar final. Na Figura 26(a) a simulação da RPC foi concluída pelo caminho esperado, já na Figura 26(b) ocorreu uma falha. É possível observar que a inscrição do arco após a transição de falha possui uma função onde caso alguma das fichas entrar na transição a ficha resultante no lugar $g10$ será a de falha.



(a) RPC (b) RPC com transição disparada

Figura 25: RPC metas $g11$, $g12$ e $g10$



(a) RPC com sucesso na execução (b) RPC com falha na execução

Figura 26: RPC com uma transição de falha

A última etapa do processo é a contagem dos caminhos da RPC modelada. O programa CPN Tools não possui uma ferramenta para esta avaliação, para obter este resultado é utilizado, um algoritmo de contagem de caminhos para grafos dirigidos. Para executar o algoritmo na rede é preciso a conversão em um grafo dirigido.

Foi desenvolvida uma ferramenta Web[1] para a conversão do arquivo do CPN Tool, a RPC, para um grafo dirigido. A ferramenta consiste no carregamento do arquivo com extensão *.cpn*, então este é convertido em um objeto *JSON* para facilitar a obtenção das informações necessárias pelo Javascript.

Através da leitura do arquivo *JSON*, são convertidos os lugares, transições e arcos da RPC para os nós e arcos do grafo. Com estes dados é possível gerar um grafo e apresentar em tela através do biblioteca Cytoscape.js[2].

Com as mesmas informações utilizadas para gerar o grafo em tela é gerado um *array* nó-origem, nó destino que permite o algoritmo realizar a contagem dos caminhos.

[1] disponível em https://github.com/brunocoelhor/PetriNet2Graph
[2] disponível em http://js.cytoscape.org/

Logo, o resultado da ferramenta é apresentado, as informações são: o número de lugares, transições, arcos e o total de caminhos e a estrutura do grafo gerado na transformação, o resultado pode ser visto na Figura 27.

Figura 27: Resultados da RP.

Os resultados número de lugares, transições e arcos servem para informar o tamanho da modelagem para possíveis comparações futuras com outros modelos ou modelagem do mesmo sistema com abordagens diferentes. O resultado total de caminhos apresenta o total de cenários de testes necessários para cobrir o SMA modelado.

Não é o objetivo deste trabalho propor cenários de testes, entretanto para exemplificar cenários de teste em relação ao exemplo da Figura 26 são apresentados a seguir. Um dos caminhos, ou cenário de teste, que deve ser testado é quando a meta $g11$, uma reunião está marcada, e a meta $g12$, um relator está indicado, são bem sucedidas, encaminhando para a meta $g10$. Outro cenário de teste é quando a meta $g11$ falha e o outro cenário é quando a meta $g12$ falha, ambos levando o meta $g10$ a falha.

No capítulo seguinte o método é implementado em dois estudos de caso.

5 IMPLEMENTAÇÃO

Após a apresentação do método, este capítulo tem por objetivo demonstrar a aplicação desta abordagem em diferentes exemplos. Para esta demonstração foram escolhidos cenários apresentados em trabalhos que SMA foram modelados de acordo com a organização $\mathcal{M}$oise^{+}. A escolha destes cenários se baseou na relevância dos autores, apresentarem a documentação das especificações necessárias para a aplicação do método e nível de complexidade.

5.1 Writing paper

5.1.1 Descrição do cenário

Este primeiro exemplo foi apresentado no trabalho (KITIO et al., 2008) e estendido em (HÜBNER; BOISSIER; BORDINI, 2011), e representa um grupo de agentes com objetivo de escrever um artigo. A especificação estrutural desta organização descreve que ela é composta por apenas um grupo *wpgroup* e este grupo possui dois papéis *escritor* e *editor* e ambos são sub-papéis de *autor* Figura 28.

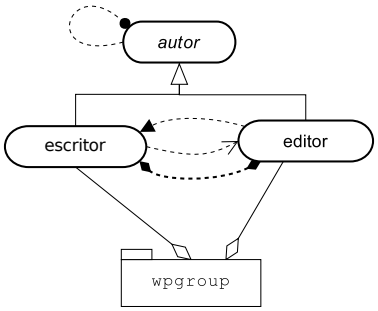


Figura 28: Esquema social *Writing paper*. Adaptado de (HÜBNER; BOISSIER; BORDINI, 2011)

Para coordenar este grupo de agentes a atingir sua meta, um esquema funcional é definido Figura 29. Neste esquema, primeiro um agente que assume a missão *mMan* deve escrever uma versão de rascunho do artigo (*fdv*), contendo como sub-metas escrever um título (*wtitle*), um resumo (*wabs*) e o título das seções (*wsectitle*), sendo necessário atingir

os objetivos nesta sequência. A segunda ramificação denominada *sv*, versão de submissão é composta pelas metas *wsec* escrever seção, e a finalização do artigo composta por duas metas em paralelo, escrever a conclusão *wcon* e escrever as referências *wref*.

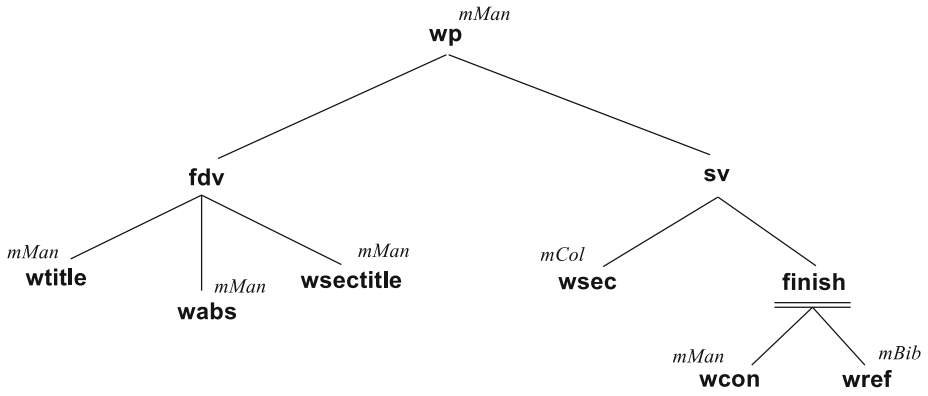


Figura 29: Especificação funcional *Writing paper*. Adaptado de (HÜBNER; BOISSIER; BORDINI, 2011)

A Tabela 6 representa a especificação deôntica que define as permissões e obrigações dos papéis que assumem as missões. Estas missões são definidos por *mMan* gerenciamento geral do projeto, composta por quatro metas, *mCol* colaboração na escrita do conteúdo e a missão *mBib* em que o agente que assumir deverá reunir e escrever as referências do artigo.

Tabela 6: Especificação deôntica *Writing paper*. (HÜBNER; BOISSIER; BORDINI, 2011)

papel	relação deôntica	missão
editor	permissão	*mMan*
escritor	obrigação	*mCol*
escritor	obrigação	*mBib*

5.1.2 Implementação

Seguindo a etapas do método, a ordem é:

5.1.2.1 Declarações

Baseado na especificação estrutural os papéis são editor e Escritor, com estas informações é possível codificar as *Standard declaration* de acordo com o código abaixo.

```
1 colset Role = with writer | editor | falha;
2 colset ROLES = list Role;
3 var w, e, f: ROLES;
```

5.1.2.2 Estrutura

A árvore de decomposição de metas do esquema social da Figura 29 é a base para a estrutura da RPC. Utilizando a relação dos operadores a estrutura da RPC é modelada de acordo com a Figura 30.

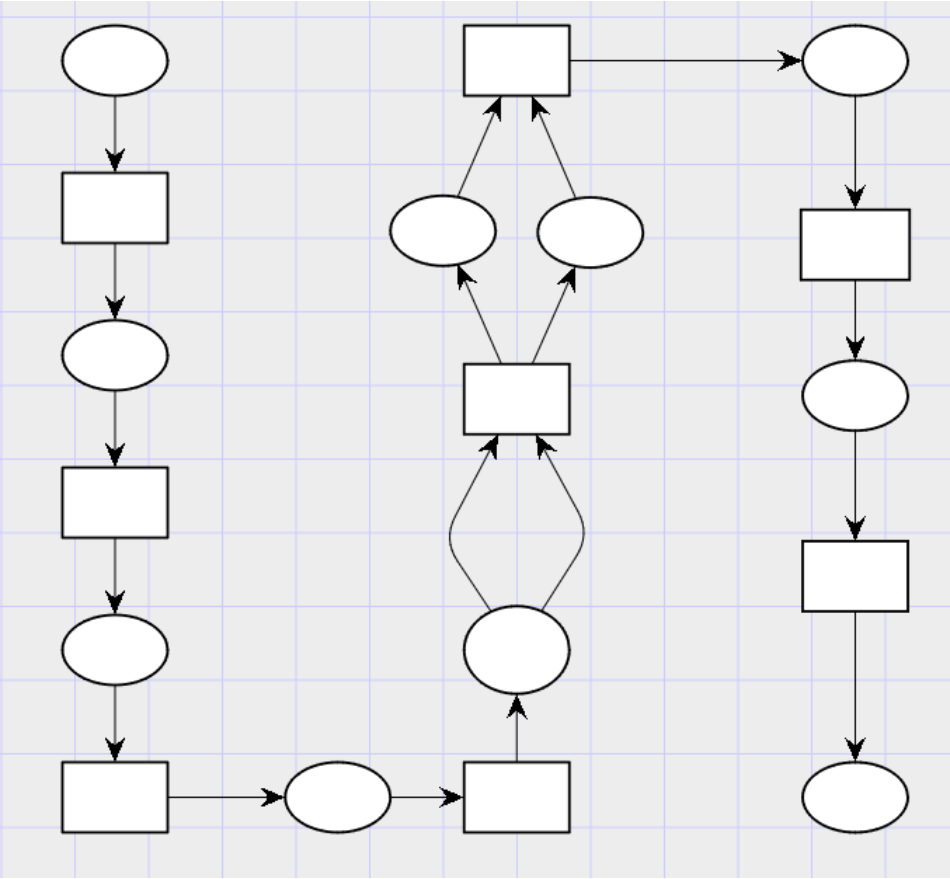


Figura 30: Estrutura da RPC

5.1.2.3 Inscrições

Através do esquema social e da especificação deôntica da Tabela 6 em que o papel editor é responsável pela missão *mMan* e o papel escritor é responsável pelas missões *mCol* e *mBib* é possível determinar as inscrições da RPC. No nível folha em que um agente diferente começa uma missão deve conter uma ficha de inicialização neste lugar, para determinar que a transição seguinte tem como requisito a atuação do novo papel como pode ser visto na Figura 31 no lugar *wsec*.

5.1.2.4 Falhas

A etapa seguinte, inclusão das transições de falha, pode ser visto na Figura 32. Deve-se incluir uma transição, um lugar e uma transição auxiliar de falha para verificar se cada meta falhou, este processo é semelhante ao de (WINIKOFF, 2017), onde na Figura 18 ele verificava se cada ação tinha falhado.

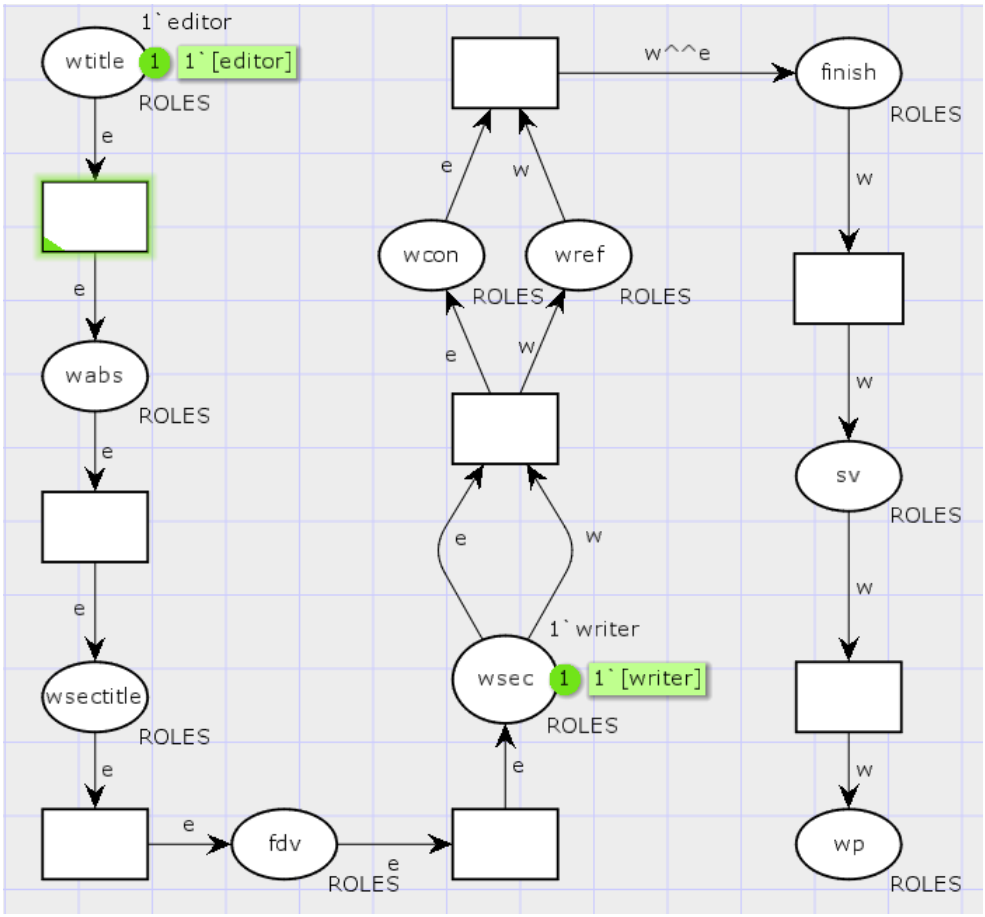


Figura 31: RPC com as inscrições.

5.1.2.5 Caminhos

Para a avaliação de testabilidade do modelo, realizando a contagem de todos caminhos é necessário obter o arquivo gerado pelo CPN Tools ao salvar a RPC e inserí-lo na ferramenta. A ferramenta gera o grafo e apresenta como saída a contagem de todos os lugares, transições, arcos e caminhos, este último, o total de caminhos (Figura 33).

Os resultados número de caminhos, dezoito, números de transição, vinte e quatro e número de arcos, cinquenta e três são resultados da RPC e não do grafo. O resultado número de caminhos, representa o número de cenários de testes necessários para a cobertura do sistema.

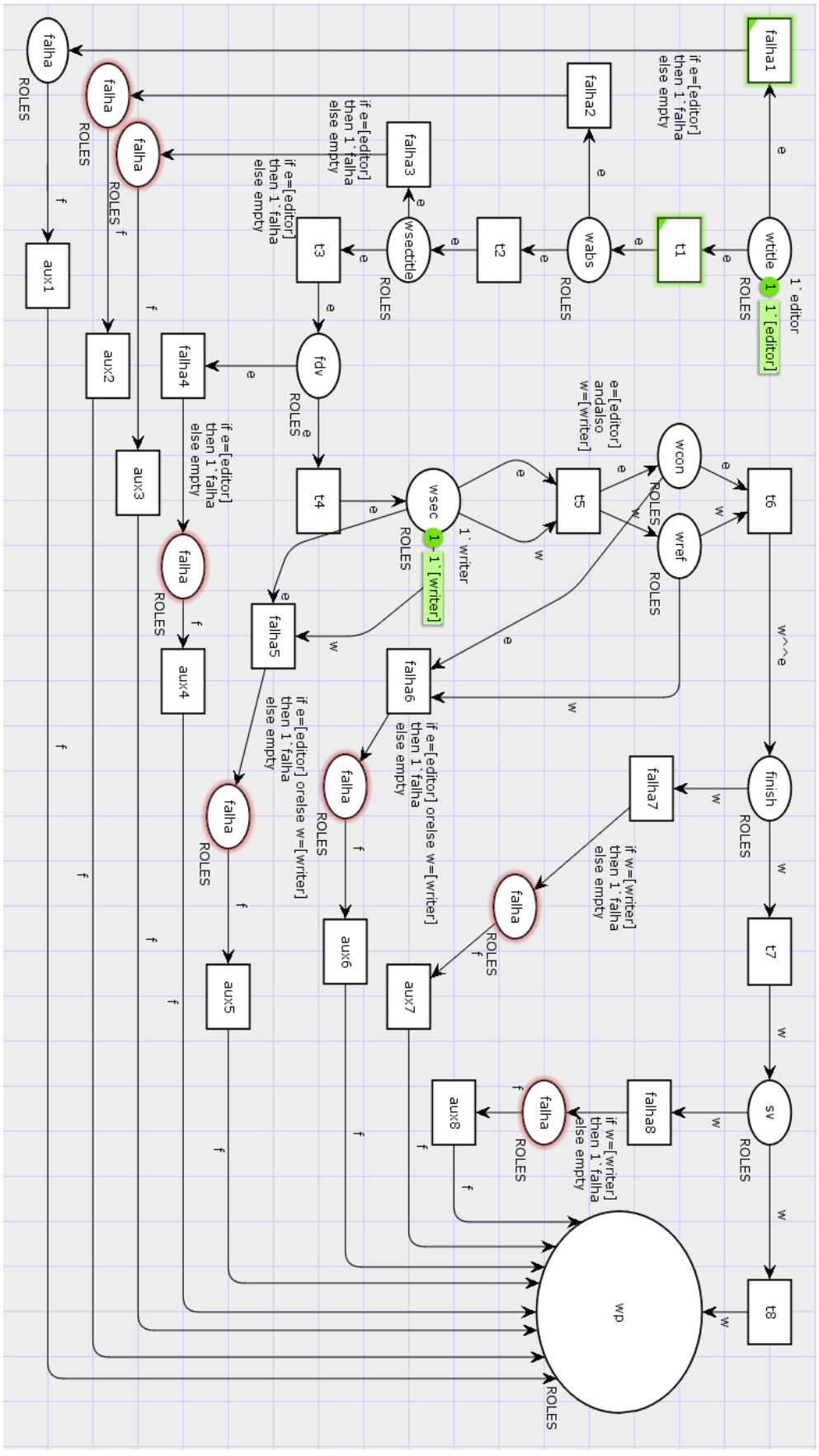


Figura 32: RPC com as transições de falha.

Figura 33: Resultados da RPC *Writing paper*.

5.2 *Multi-Agent Programming Contest- Agents on Mars*

5.2.1 Descrição do cenário

Este cenário faz parte da *Multi-Agent Programming Contest* [1], um evento anual que tem o objetivo de estimular a pesquisa na área de desenvolvimento e programação de SMA. Através da competição, é possível avaliar e comparar diferentes aspectos dos sistemas desenvolvidos, identificando problemas, coletando *benchmarks* e reunindo casos de teste que podem servir como referência para testar linguagens de programação multiagente, plataformas e ferramentas (KÖSTER; SCHLESINGER; DIX, 2012; AHLBRECHT et al., 2013).

Neste ano de 2013, o cenário da competição é Marte, onde os competidores deverão desenvolver agentes inteligentes autônomos para localizar poços de água, ocupar as melhores zonas de Marte e sabotar seus rivais para conseguir seu objetivo ou para defender a si mesmos. Neste contexto, foi utilizado a solução da equipe SMADAS-UFSC que venceu a competição no ano de 2013 e desenvolveu a solução utilizando Jason, Cartago e $\mathcal{M}$oise$^+$ (ZATELLI et al., 2013; AHLBRECHT et al., 2013).

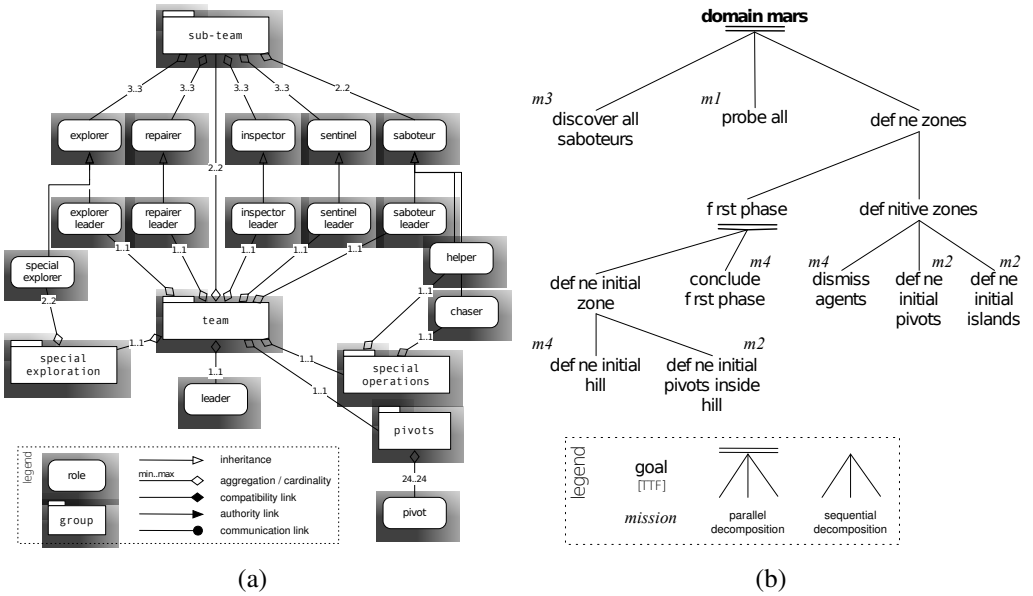


Figura 34: Agentes em Marte

5.2.2 Implementação

5.2.2.1 Declaração

Este exemplo como pode ser visto na Figura 34(a) possui mais papéis. Para esta modelagem, são declarados apenas os papéis utilizados na especificação social, e também presentes na especificação deôntica. A declaração encontra-se no código abaixo.

[1] https://multiagentcontest.org/

Tabela 7: Especificação deôntica *Agentes em Marte*. (ZATELLI et al., 2013)

norma	papel	relação deôntica	missão
n1	explorer	obrigação	*m1*
n2	sentinelLeader	obrigação	*m2*
n3	inspector	obrigação	*m3*
n4	explorerLeader	obrigação	*m4*

```
colset Role = with explorer | sentinelLeader | inspector |
   explorerLeader | falha;
colset ROLES = list Role;
colset Sequence = product Role*Role;
var e, s, i, el, sl, seq: ROLES;
```

5.2.2.2 *Estrutura*

A criação de estrutura tem início pelo modo de leitura do esquema social, e obedecendo a relação entre operadores e as estruturas da RP. As metas *discover all saboteurs*, *probe all* e *define zones* estão em paralelo, portanto elas são independentes para começar, mas a meta global *domain mars* só poderá ser concluída quando todas as metas também estiverem concluídas. Logo, a estrutura da RPC para este modelo, foi definida na Figura 35. O posicionamento dos lugares importa apenas para a RP ficar mais legível, o que determina por onde ela começa são as fichas, inscrições inseridas na próxima etapa.

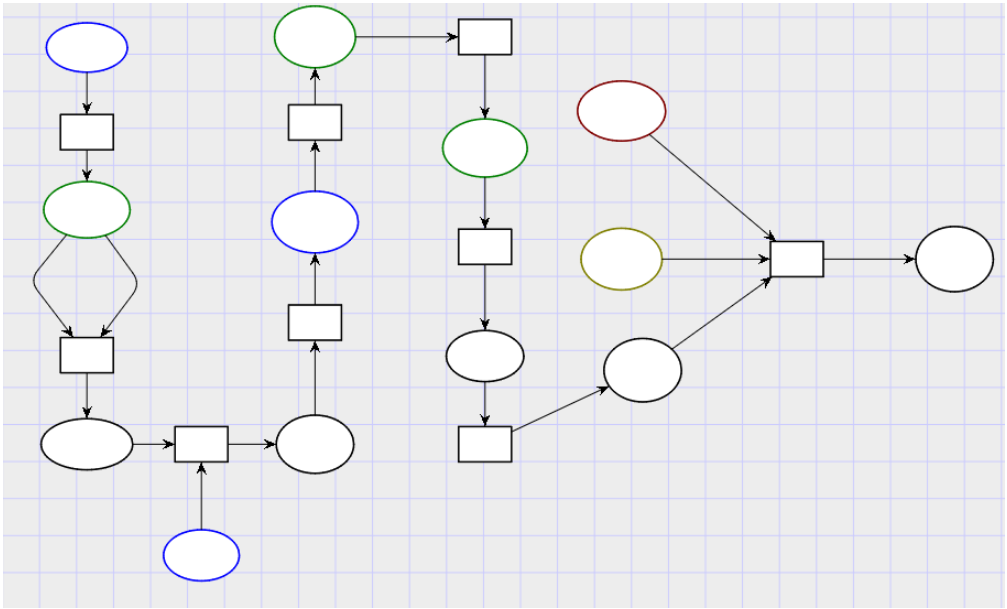


Figura 35: Estrutura da RPC *Agents on Mars*.

5.2.2.3 *Inscrições*

Na Figura 36 é possível ver as inscrições dos nomes dos lugares identificando as metas, as fichas identificando os papéis que atuam naquelas metas/lugares e também as inscrições nos arcos que determina que fichas são retiradas dos lugares iniciais e que fichas vão para o lugar final quando uma transição é disparada. As cores são apenas estilizações que podem ser utilizadas para identificar a que missão aquela meta/lugar está alocado.

Apesar dos lugares *discover all saboteurs*, *probe all* estaram no final da RP, é possível identificar pelas fichas que eles já estão prontos para serem disparados desde o início, tendo como dependência apenas a conclusão da meta *define zones*. A dependência é criada na inscrição do arco de saída, exigindo que as três fichas estejam disponível para a transição ser disparada.

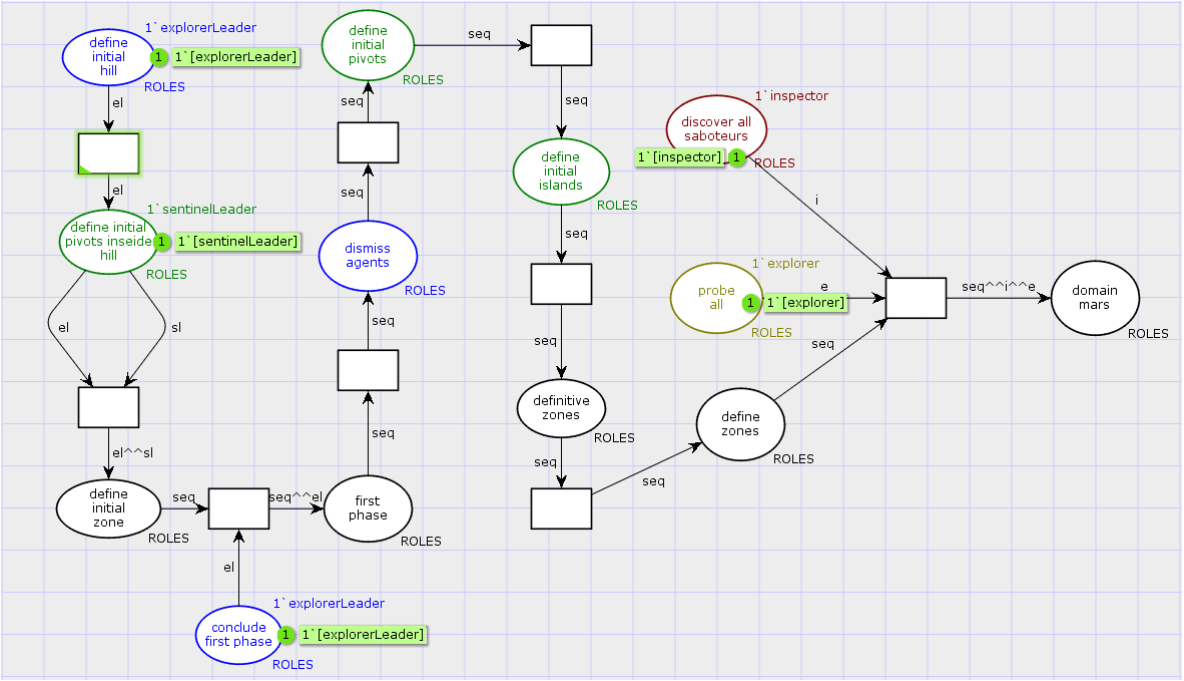


Figura 36: RPC *Agents on Mars* com as inscrições.

5.2.2.4 *Falha*

As transições de falha são inseridas obedecendo os operadores. Para uma melhor visualização elas são identificadas com a inscrição $ficha_n$. Elas recebem uma inscrição de arco que é uma função dependente da ficha recebida, caso a condição for verdadeira para a ficha o lugar final receberá uma ficha de falha, dependendo do modelo uma falha já poderá comprometer todo o sistema, caso não tenha nenhum laço de reparação. A Figura 37 apresenta a RPC com as transições de falha.

5.2.2.5 Caminhos

Nesta etapa, é necessário a inserção do arquivo gerado pela ferramenta CPN Tools ao salvar a rede, na ferramenta desenvolvida para a contagem de caminhos. O grafo gerado e os resultados são apresentados na Figura 38.

Entre os resultados são apresentadas as informações referente ao modelo de RPC, vinte e cinco lugares, trinta e três transições e 71 arcos. Estas informações podem ser utilizadas para comprar outros modelos para a mesma solução do problema. E o último resultado, número de caminho igual a treze, é o número de cenários de testes previstos para contemplar toda a cobertura do sistema.

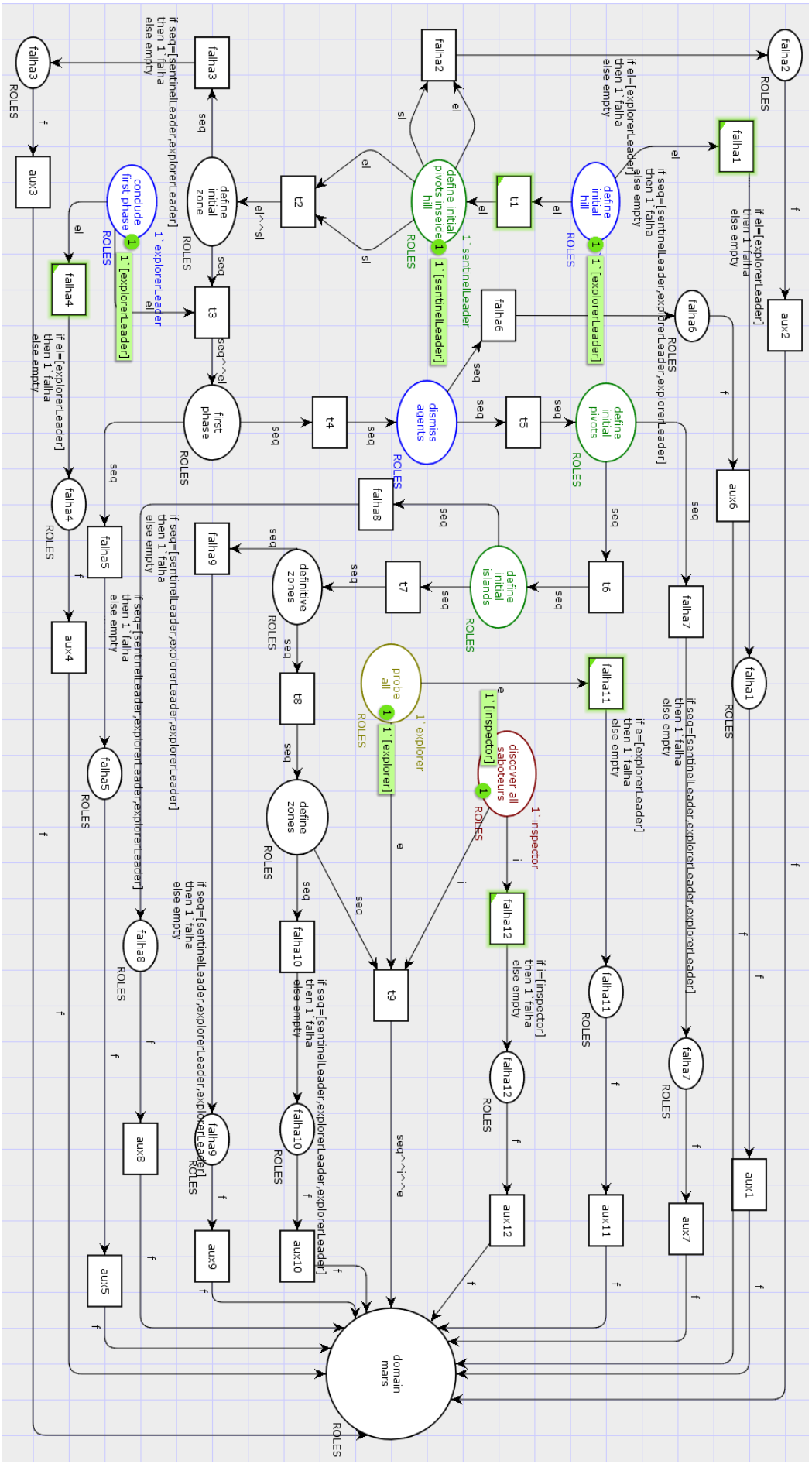


Figura 37: RPC *Agents on Mars* com as transições de falha.

Figura 38: Resultados da RPC *Agents on Mars*.

6 CONCLUSÃO

Avaliar o número de casos de testes para softwares tradicionais com uma cobertura em um nível adequado não é uma tarefa fácil, avaliar este número de casos de testes com uma boa cobertura é ainda mais complicado devido a natureza autônoma dos agentes, que podem realizar seus objetivos de forma independente.

Esta dissertação propõem um método para a avaliação da testabilidade de SMA cuja organização é modelada através do $\mathcal{M}$oise. Este método permite modelar uma organização $\mathcal{M}$oise para uma RPC através de cinco etapas com os processos: declarações, estrutura, inscrições, falha e caminhos. Para avaliar a testabilidade contando os caminhos apresentados na RPC foi desenvolvido uma ferramenta que utiliza algoritmo de contagem de caminhos em grafos.

A principal contribuição deste trabalho é o método desenvolvido, que permite modelar em RPC sistemas multiagentes modelados pelo $\mathcal{M}$oise. Através das cinco etapas do processo e da relação dos operadores da Figura 23 é possível modelar RPC que podem ser simuladas e avaliadas pelos recursos que as redes de petri oferecem. Uma contribuição secundária é uma ferramenta que realiza a contagem de caminhos de RP modeladas pela ferramenta CPN Tools, por padrão esta ferramenta não disponibiliza este tipo de análise.

Para os exemplos apresentados e modelados pelo método foi possível avaliar a testabilidade e obter números de cenários de testes necessários para a cobertura dos sistemas. Pela simplicidade dos exemplos, o número de cenários de teste é factível para realização dos testes, mas para exemplos mais complexos é possível que haja necessidade de escolha de áreas mais críticas do sistema para a criação de casos de uso focando por necessidade.

No entanto uma limitação apresentada pela pesquisa foi a modelagem das restrições de comunicação, característica do $\mathcal{M}$oise$^+$ apresentada na especificação estrutural. Para incluir a modelagem de restrição de comunicação é necessário um nível onde os agentes estejam incluídos, assim permitindo uma simulação mais completa da RPC.

6.1 Trabalhos Futuros

Para um aperfeiçoamento e extensão do método além da correção da limitação descrita anteriormente são apontadas possibilidades de trabalhos futuros.

- Extensão para o nível de agentes: para uma análise mais completa se um SMA, é necessário avaliar o comportamento dos agentes. Este comportamento poderia ser realizado utilizando ainda o $\mathcal{M}$oise$^+$ como modelo de organização e a base do método apresentada neste trabalho, mas utilizando em conjunto das RPC as RP Hierárquicas. Cada meta teria um subnível hierárquico que teria uma RP que faria a análise dos agentes e as ações que estes estão comprometidos.

- Utilização do método em métodos de modelagem de sistemas orientados em agentes: Avaliar a utilização do método em métodos como o Prometheus ou outros que já provê diretrizes para o desenvolvimento de SMA.

REFERÊNCIAS

AHLBRECHT, T.; DIX, J.; KÖSTER, M.; SCHLESINGER, F. Multi-agent programming contest 2013. In: INTERNATIONAL WORKSHOP ON ENGINEERING MULTI-AGENT SYSTEMS, 2013. **Anais...** [S.l.: s.n.], 2013. p.292–318.

ALMEIDA, H. O. de; SILVA, L. D. da; PERKUSICH, A.; BARROS COSTA, E. de. A formal approach for the modelling and verification of multiagent plans based on model checking and Petri nets. In: INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING FOR LARGE-SCALE MULTI-AGENT SYSTEMS, 2004. **Anais...** [S.l.: s.n.], 2004. p.162–179.

AMMANN, P.; OFFUTT, J. **Introduction to software testing**. [S.l.]: Cambridge University Press, 2016.

ARGENTE, E.; JULIAN, V.; BOTTI, V. Multi-agent system development based on organizations. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.150, n.3, p.55–71, 2006.

ATHAMENA, B.; HOUHAMDI, Z. A petri net based multi-agent system behavioral testing. **Modern Applied Science**, [S.l.], v.6, n.3, p.46, 2012.

BAI, Q.; ZHANG, M.; WIN, K. T. A colored petri net based approach for multi-agent interactions. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS ROBOTS AND AGENTS, PALMERSTON NORTH, NEW ZEALAND, 2., 2004. **Proceedings...** [S.l.: s.n.], 2004. p.152–157.

BENFIELD, S. S.; HENDRICKSON, J.; GALANTI, D. Making a strong business case for multiagent technology. In: AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, 2006. **Proceedings...** [S.l.: s.n.], 2006. p.10–15.

BOND, A. H.; GASSER, L. **Readings in distributed artificial intelligence**. [S.l.]: Morgan Kaufmann, 2014.

BOTÍA, J. A.; LÓPEZ-ACOSTA, A.; SKARMETA, A. G. ACLAnalyser: a tool for debugging multi-agent system. In: EUROPEAN CONFERENCE ON ARTIFICIAL INTELLIGENCE, 16., 2004. **Proceedings...** [S.l.: s.n.], 2004. p.967–968.

BOURQUE, P.; FAIRLEY, R. E. et al. **Guide to the software engineering body of knowledge (SWEBOK (R))**: Version 3.0. [S.l.]: IEEE Computer Society Press, 2014.

BURMEISTER, B.; ARNOLD, M.; COPACIU, F.; RIMASSA, G. BDI-agents for agile goal-oriented business processes. In: AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS: INDUSTRIAL TRACK, 7., 2008. **Proceedings...** [S.l.: s.n.], 2008. p.37–44.

CAIRE, G.; COSSENTINO, M.; NEGRI, A.; POGGI, A.; TURCI, P. **Multi-agent systems implementation and testing**. [S.l.]: na, 2004.

CARDOSO, J.; VALETTE, R. **Redes de petri**. [S.l.]: Editora da UFSC, 1997.

CIALDINI, R. B.; TROST, M. R. Social influence: Social norms, conformity and compliance. , [S.l.], 1998.

COELHO, R.; KULESZA, U.; STAA, A. von; LUCENA, C. Unit testing in multi-agent systems using mock agents and aspects. In: SOFTWARE ENGINEERING FOR LARGE-SCALE MULTI-AGENT SYSTEMS, 2006., 2006. **Proceedings...** [S.l.: s.n.], 2006. p.83–90.

DURFEE, E. H. The distributed artificial intelligence melting pot. **Ann Arbor**, [S.l.], v.1001, p.48109, 1991.

DURFEE, E. H.; ROSENSCHEIN, J. S. Distributed problem solving and multi-agent systems: Comparisons and examples. **Ann Arbor**, [S.l.], v.1001, n.48109, p.29, 1994.

EKINCI, E. E.; TIRYAKI, A. M.; ÇETIN, Ö.; DIKENELLI, O. Goal-oriented agent testing revisited. In: **Agent-Oriented Software Engineering IX**. [S.l.]: Springer, 2009. p.173–186.

FERBER, J.; GASSER, L. Intelligence artificielle distribuée. **Tutorial Notes of the 11th Conference on Expert Systems and their Aplications**, [S.l.], 1991.

FRANKLIN, S.; GRAESSER, A. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In: INTERNATIONAL WORKSHOP ON AGENT THEORIES, ARCHITECTURES, AND LANGUAGES, 1996. **Anais...** [S.l.: s.n.], 1996. p.21–35.

GOMEZ-SANZ, J. J.; BOTÍA, J.; SERRANO, E.; PAVÓN, J. Testing and debugging of MAS interactions with INGENIAS. In: INTERNATIONAL WORKSHOP ON AGENT-ORIENTED SOFTWARE ENGINEERING, 2008. **Anais...** [S.l.: s.n.], 2008. p.199–212.

GONCALVES, E. M. N. An approach to specify knowledge in multi-agent systems using petri nets. In: NETWORK AND SYSTEM SECURITY (NSS), 2010 4TH INTERNATIONAL CONFERENCE ON, 2010. **Anais...** [S.l.: s.n.], 2010. p.456–461.

GRAHAM, D.; VAN VEENENDAAL, E.; EVANS, I. **Foundations of software testing**: ISTQB certification. [S.l.]: Cengage Learning EMEA, 2008.

HANNOUN, M.; BOISSIER, O.; SICHMAN, J.; SAYETTAT, C. MOISE: An organizational model for multi-agent systems. **Advances in Artificial Intelligence**, [S.l.], p.156–165, 2000.

HORLING, B.; LESSER, V. A survey of multi-agent organizational paradigms. **The Knowledge Engineering Review**, [S.l.], v.19, n.4, p.281–316, 2004.

HOUHAMDI, Z. Multi-agent system testing: A survey. **International Journal of Advanced Computer Science and Applications**, [S.l.], v.2, n.6, p.135–141, 2011.

HOUHAMDI, Z.; ATHAMENA, B. Structured integration test suite generation process for multi-agent system. **Journal of Computer Science**, [S.l.], v.7, n.5, p.690, 2011.

HÜBNER, J. F. **Um modelo de reorganização de sistemas multiagentes.** 2003. Tese (Doutorado em Ciência da Computação) — Universidade de São Paulo.

HÜBNER, J. F.; BOISSIER, O.; BORDINI, R. H. A normative programming language for multi-agent organisations. **Annals of Mathematics and Artificial Intelligence**, [S.l.], v.62, n.1-2, p.27–53, 2011.

HÜBNER, J. F.; SICHMAN, J. S.; BOISSIER, O. \mathcal {SM} oiseˆ{+}: A Middleware for Developing Organised Multi-agent Systems. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, 2005. **Anais...** [S.l.: s.n.], 2005. p.64–77.

HÜBNER, J. F.; SICHMAN, J. S.; BOISSIER, O. Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels. **International Journal of Agent-Oriented Software Engineering**, [S.l.], v.1, n.3-4, p.370–395, 2007.

JENNINGS, N. R. On agent-based software engineering. **Artificial intelligence**, [S.l.], v.117, n.2, p.277–296, 2000.

JENSEN, K. **Coloured Petri nets**: basic concepts, analysis methods and practical use. [S.l.]: Springer Science & Business Media, 2013. v.1.

JORGENSEN, P. C. **Software testing**: a craftsman's approach. [S.l.]: CRC press, 2016.

KITIO, R.; BOISSIER, O.; HÜBNER, J. F.; RICCI, A. Organisational artifacts and agents for open multi-agent organisations:“giving the power back to the agents”. In: **Coordination, Organizations, Institutions, and Norms in Agent Systems III**. [S.l.]: Springer, 2008. p.171–186.

KNUBLAUCH, H. Extreme programming of multi-agent systems. In: AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS: PART 2, 2002. **Proceedings...** [S.l.: s.n.], 2002. p.704–711.

KÖHLER, M.; MOLDT, D.; RÖLKE, H. Modelling the structure and behaviour of Petri net agents. **Applications and Theory of Petri Nets 2001**, [S.l.], p.224–241, 2001.

KÖSTER, M.; SCHLESINGER, F.; DIX, J. The multi-agent programming contest 2012. In: INTERNATIONAL WORKSHOP ON PROGRAMMING MULTI-AGENT SYSTEMS, 2012. **Anais...** [S.l.: s.n.], 2012. p.174–195.

LAM, D. N.; BARBER, K. S. Debugging Agent Behavior in an Implemented Agent System. In: PROMAS, 2004. **Anais...** [S.l.: s.n.], 2004. p.104–125.

LESSER, V. R. Cooperative multiagent systems: A personal view of the state of the art. **IEEE Transactions on knowledge and data engineering**, [S.l.], v.11, n.1, p.133–142, 1999.

MACIEL, P. R.; LINS, R. D.; CUNHA, P. R. **Introdução às redes de Petri e aplicações**. [S.l.]: UNICAMP-Instituto de Computacao, 1996.

MALONE, T. W.; CROWSTON, K. The interdisciplinary study of coordination. **ACM Computing Surveys (CSUR)**, [S.l.], v.26, n.1, p.87–119, 1994.

MILLER, T.; PADGHAM, L.; THANGARAJAH, J. Test coverage criteria for agent interaction testing. In: **Agent-oriented Software Engineering XI**. [S.l.]: Springer, 2011. p.91–105.

MURATA, T. Petri nets: Properties, analysis and applications. **Proceedings of the IEEE**, [S.l.], v.77, n.4, p.541–580, 1989.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The art of software testing**. [S.l.]: John Wiley & Sons, 2011.

NGUYEN, C. D.; PERINI, A.; TONELLA, P. Goal-oriented testing for MASs. **International Journal of Agent-Oriented Software Engineering**, [S.l.], v.4, n.1, p.79–109, 2009.

NGUYEN, D. C. **Testing techniques for software agents**. 2009. Tese (Doutorado em Ciência da Computação) — University of Trento.

NÚÑEZ, M.; RODRÍGUEZ, I.; RUBIO, F. Specification and testing of autonomous agents in e-commerce systems. **Software Testing, Verification and Reliability**, [S.l.], v.15, n.4, p.211–233, 2005.

NWANA, H. S. Software agents: An overview. **The knowledge engineering review**, [S.l.], v.11, n.3, p.205–244, 1996.

ODELL, J. J.; PARUNAK, H. V. D.; FLEISCHER, M. The role of roles in designing effective agent organizations. In: INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING FOR LARGE-SCALE MULTI-AGENT SYSTEMS, 2002. **Anais...** [S.l.: s.n.], 2002. p.27–38.

PADGHAM, L.; WINIKOFF, M.; POUTAKIDIS, D. Adding debugging support to the Prometheus methodology. **Engineering Applications of Artificial Intelligence**, [S.l.], v.18, n.2, p.173–190, 2005.

PETERSON, J. L. Petri nets. **ACM Computing Surveys (CSUR)**, [S.l.], v.9, n.3, p.223–252, 1977.

POUTAKIDIS, D.; WINIKOFF, M.; PADGHAM, L.; ZHANG, Z. Debugging and testing of multi-agent systems using design artefacts. **Multi-Agent Programming:**, [S.l.], p.215–258, 2009.

PRESSMAN, R. S. **Software engineering**: a practitioner's approach. [S.l.]: Palgrave Macmillan, 2005.

RATZER, A. V.; WELLS, L.; LASSEN, H. M.; LAURSEN, M.; QVORTRUP, J. F.; STISSING, M. S.; WESTERGAARD, M.; CHRISTENSEN, S.; JENSEN, K. CPN tools for editing, simulating, and analysing coloured Petri nets. In: INTERNATIONAL CONFERENCE ON APPLICATION AND THEORY OF PETRI NETS, 2003. **Anais...** [S.l.: s.n.], 2003. p.450–462.

RODRIGUES, L.; CARVALHO, G.; PAES, R.; LUCENA, C. Towards an integration test architecture for open MAS. In: SOFTWARE ENGINEERING FOR AGENT-ORIENTED SYSTEMS/SBES, 1., 2005. **Anais...** [S.l.: s.n.], 2005. p.60–66.

ROUFF, C. A test agent for testing agents and their communities. In: AEROSPACE CONFERENCE PROCEEDINGS, 2002. IEEE, 2002. **Anais...** [S.l.: s.n.], 2002. v.5, p.5–2638.

RUSSELL, S. J.; NORVIG, P. **Artificial intelligence**: a modern approach (International Edition). [S.l.]: {Pearson US Imports & PHIPEs}, 2002.

SEN, S.; AIRIAU, S. Emergence of norms through social learning. In: IJCAI, 2007. **Anais...** [S.l.: s.n.], 2007. v.1507, p.1512.

SOMMERVILLE, I. **Software Engineering**. 9th.ed. USA: Addison-Wesley Publishing Company, 2010.

SUDEIKAT, J.; RENZ, W. A systemic approach to the validation of self–organizing dynamics within MAS. In: INTERNATIONAL WORKSHOP ON AGENT-ORIENTED SOFTWARE ENGINEERING, 2008. **Anais...** [S.l.: s.n.], 2008. p.31–45.

TINNEMEIER, N.; DASTANI, M.; MEYER, J.-J. Roles and norms for programming agent organizations. In: THE 8TH INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS-VOLUME 1, 2009. **Proceedings...** [S.l.: s.n.], 2009. p.121–128.

VAN DEN BROEK, E. L.; JONKER, C. M.; SHARPANSKYKH, A.; TREUR, J. et al. Formal modeling and analysis of organizations. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, 2005. **Anais...** [S.l.: s.n.], 2005. p.18–34.

WEYNS, D.; HOLVOET, T. A colored Petri-net for a multi-agent application. In: MOCA'02, 2002. **Proceedings...** [S.l.: s.n.], 2002. v.561, p.121–141.

WEYNS, D.; OMICINI, A.; ODELL, J. Environment as a first class abstraction in multiagent systems. **Autonomous agents and multi-agent systems**, [S.l.], v.14, n.1, p.5–30, 2007.

WINIKOFF, M. Assurance of agent systems: What role should formal verification play? **Specification and Verification of Multi-agent systems**, [S.l.], p.353–383, 2010.

WINIKOFF, M. BDI agent testability revisited. **Autonomous Agents and Multi-Agent Systems**, [S.l.], p.1–39, 2017.

WINIKOFF, M.; CRANEFIELD, S. On the Testability of BDI Agent Systems. **Journal of Artificial Intelligence Research**, [S.l.], v.51, p.71–131, 2014.

WOOLDRIDGE, M.; JENNINGS, N. R. Intelligent agents: Theory and practice. **The knowledge engineering review**, [S.l.], v.10, n.2, p.115–152, 1995.

ZATELLI, M. R.; DE BRITO, M.; SCHMITZ, T. L.; MORATO, M. M.; DE SOUZA, K. S.; UEZ, D. M.; HŘBNER, J. F. SMADAS: A Team for MAPC Considering the Organization and the Environment as First-class Abstractions. In: INTERNATIONAL WORKSHOP ON ENGINEERING MULTI-AGENT SYSTEMS, 2013. **Anais...** [S.l.: s.n.], 2013. p.319–328.

ZHANG, Z.; THANGARAJAH, J.; PADGHAM, L. Automated Unit Testing for Agent Systems. **ENASE**, [S.l.], v.7, p.10–18, 2007.