

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO RIO GRANDE
PROGRAMA DE PÓS-GRADUAÇÃO EM MODELAGEM COMPUTACIONAL

**TESTES A NÍVEL ORGANIZACIONAL EM SISTEMAS
MULTIAGENTE UTILIZANDO REDES DE PETRI COLORIDAS**

por

Ricardo Arend Machado

Tese para obtenção do Título de
Doutor em Modelagem Computacional

Rio Grande, Setembro, 2024

Ricardo Arend Machado

“Testes a Nível Organizacional em Sistemas Multiagente utilizando Redes de Petri Coloridas”

Tese apresentada ao Programa de Pós-Graduação em Modelagem Computacional da Universidade Federal do Rio Grande - FURG, como requisito parcial para obtenção do Grau de Doutor. Área de concentração: Modelagem Computacional.

Aprovado em: 11/09/2024

BANCA EXAMINADORA

Documento assinado digitalmente
 **DIANA FRANCISCA ADAMATTI**
Data: 12/09/2024 10:55:01-0300
Verifique em <https://validar.iti.gov.br>

Profa. Dra. Diana Francisca Adamatti (Orientadora - FURG)

Documento assinado digitalmente
 **EDER MATEUS NUNES GONCALVES**
Data: 12/09/2024 16:53:41-0300
Verifique em <https://validar.iti.gov.br>

Prof. Dr. Eder Mateus Nunes Gonçalves (Co-orientador - FURG)

Documento assinado digitalmente
 **Jomi Fred Hubner**
Data: 12/09/2024 12:50:49-0300
CPF: **.334.769-**
Verifique as assinaturas em <https://v.ufsc.br>

Prof. Dr. Jomi Fred Hübner (UFSC)

Documento assinado digitalmente
 **CLEO ZANELLA BILLA**
Data: 12/09/2024 10:57:23-0300
Verifique em <https://validar.iti.gov.br>

Prof. Dr. Cleo Zanella Billa (FURG)

Documento assinado digitalmente
 **GRACALIZ PEREIRA DIMURO**
Data: 13/09/2024 14:08:43-0300
Verifique em <https://validar.iti.gov.br>

Profa. Dra. Graçaliz Pereira Dimuro (FURG)

Ficha Catalográfica

M149t Machado, Ricardo Arend.

Testes a nível organizacional em sistemas multiagente utilizando redes de petri coloridas / Ricardo Arend Machado. – 2024.

78 f.

Tese (doutorado) – Universidade Federal do Rio Grande – FURG, Programa de Pós-Graduação em Modelagem Computacional, Rio Grande/RS, 2024.

Orientadora: Dra. Diana Francisca Adamatti.

Coorientador: Dr. Eder Mateus Nunes Gonçalves.

1. Sistema multiagente 2. Redes de Petri 3. Organização
I. Adamatti, Diana Francisca II. Gonçalves, Eder Mateus Nunes
III. Título.

CDU 004

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

AGRADECIMENTOS

A Deus pela proteção e cuidado.

A toda minha família, especialmente minha mãe Erselha Machado pelo apoio.

À minha namorada Kennah Pestana pelo amor e carinho.

À minha orientadora, Diana Adamatti, que admiro muito como profissional, pela paciência e ensinamentos.

Ao meu co-orientador Eder Gonçalves, pela disponibilidade, contribuições e ideias.

À CAPES pelo apoio financeiro com a bolsa.

A Universidade Federal do Rio Grande (FURG) e ao curso de Modelagem Computacional.

RESUMO

Os Sistemas Multiagente são sistemas distribuídos que consistem em agentes que interagem de forma autônoma entre si em um ambiente para realizar tarefas e atingir objetivos. A fase de teste é uma etapa crucial para buscar a correção de todo o sistema de software para oferecer garantias de funcionamento e segurança aos usuários. No entanto, aplicações baseadas em sistemas multiagente têm maior dificuldade de serem testadas, devido às suas propriedades como autonomia, reatividade, pró-atividade e habilidades sociais, e existe uma lacuna na literatura no que diz respeito ao teste de sistemas com essas características. Assim torna-se importante o desenvolvimento de ferramentas para testes nesses sistemas, visando aumentar as opções disponíveis para garantir uma mais ampla cobertura. Em trabalho anterior foi apresentada uma abordagem para testes em sistemas multiagente que utilizam o modelo organizacional *Moise*⁺, onde são gerados casos de teste utilizando redes de Petri coloridas para mapeamento do sistema em um modelo formal. Diante disso, propõe-se nessa tese de doutorado estender o modelo através da inclusão de grupos organizacionais, as cardinalidades que indicam a quantidade máxima e mínima de agentes para realizar uma missão, realizar uma descrição detalhada das etapas de mapeamento com a inclusão de axiomas, o que permite o desenvolvimento de uma ferramenta que automatize as etapas necessárias desse mapeamento. Ao final, um caso de uso foi utilizado como exemplo de funcionamento do modelo proposto, com a realização de testes de execução, utilizando os casos de testes gerados.

Palavras-chave: Sistema multiagente; Redes de Petri, Organização.

ABSTRACT

Multi-agent systems are distributed systems that consist of agents interacting autonomously with each other in an environment to perform tasks and achieve goals. The testing phase is a crucial step to ensure the correctness of the entire software system, providing guarantees of functionality and security to users. However, applications based on multi-agent systems are more difficult to test due to their properties such as autonomy, reactivity, proactivity, and social abilities, and there is a gap in the literature regarding the testing of systems with these characteristics. Therefore, it is important to develop tools for testing these systems, aiming to increase the available options to ensure broader coverage. In a previous work, an approach was presented for testing multi-agent systems that use the organizational model $\mathcal{Moise}^+$, where test cases are generated using colored Petri nets to map the system into a formal model. In this context, this doctorate dissertation proposes to extend the model by including organizational groups, cardinalities that indicate the minimum and maximum number of agents required to carry out a mission, and present a detailed description of the mapping steps with the inclusion of axioms, which allows for the development of a tool that automates the necessary steps of this mapping. A use case was employed to evaluate the approach as an example of the proposed model's functioning with the execution of tests using the generated test cases.

Keywords: Multi-agent system; Petri Nets, Organization .

ÍNDICE

1	INTRODUÇÃO	13
1.1	Hipótese de pesquisa	16
1.2	Objetivos	17
1.3	Estrutura do trabalho	17
2	EMBASAMENTO TEÓRICO	18
2.1	Agentes	18
2.2	Sistemas Multiagente	19
2.3	Teste de Software	21
2.3.1	Teste de Sistemas Multiagente	23
2.4	Redes de Petri Coloridas	24
2.5	A Plataforma JaCaMo	25
2.6	O Modelo Organizacional $\mathcal{M}oise^+$	25
2.6.1	Especificação Estrutural	27
2.6.2	Especificação Funcional	28
2.6.3	Especificação Deôntica	28
2.7	Trabalhos Relacionados	29
3	DESCRIÇÃO DO MÉTODO	35
3.1	Visão Geral	35
3.2	Mapeamento na Rede de Petri	37
3.2.1	Gerar as Declarações	37
3.2.2	Modelando a Estrutura	39
3.2.3	Incluindo as Inscrições	41
3.3	Contagem dos Casos de Teste	42
3.4	Geração dos Casos de Teste	44
3.5	Execução dos casos de teste	44
3.6	Uma Ferramenta de Automatização do Método	45
4	CASOS DE USO	47
4.1	Cenário do Inspector	47
4.1.1	Mapeamento do Cenário Inspector	47
4.1.2	Contagem dos Casos de Teste	53
4.1.3	Geração dos Casos de Teste	54
4.1.4	Executando os Casos de Teste	55
4.2	Cenário Soccer Team	57
4.2.1	Mapeamento do Cenário Soccer Team	58
4.2.2	Contagem dos Casos de Teste	62
4.2.3	Geração dos Casos de Teste	63
4.2.4	Executando os Casos de Teste	63
5	CONCLUSÃO	68

REFERÊNCIAS 70

6 APÊNDICE 75

LISTA DE FIGURAS

Figura 1:	Ciclo de vida de uma meta do $\mathcal{M}\text{oise}^+$ adaptado de (BOISSIER et al., 2020).	14
Figura 2:	Diferença entre teste a nível Organizacional e Agente, Fonte: Autor.	15
Figura 3:	Classificação dos mecanismos para geração de teste, adaptado de (CLARK; WALKINSHAW; HIERONS, 2021).	16
Figura 4:	Arquitetura BDI genérica, adaptado de (WOOLDRIDGE, 1999).	19
Figura 5:	Uma visão geral dos processos de teste, adaptado de (CLARK; WALKINSHAW; HIERONS, 2021).	22
Figura 6:	RPC: Exemplo do <i>Dining Philosophers</i> .	25
Figura 7:	Visão geral do JaCaMo, adaptado de (BOISSIER et al., 2013).	26
Figura 8:	Exemplo de Especificação Estrutural do $\mathcal{M}\text{oise}^+$ (HÜBNER; SICHMAN, 2007).	27
Figura 9:	Exemplo de Especificação Funcional do $\mathcal{M}\text{oise}^+$ (HÜBNER; SICHMAN, 2007).	28
Figura 10:	Sequência de Etapas do Método, Fonte: autor.	35
Figura 11:	Operadores para mapeamento da estrutura do $\mathcal{CPN}_4M$.	40
Figura 12:	Estrutura de um sistema genérico mapeado para RPC.	40
Figura 13:	Inclusão de Inscrições na RPC.	42
Figura 14:	Exemplo de Grafo de Estado de Espaço do $\mathcal{CPN}_4M$, Fonte: Autor.	43
Figura 15:	Arquitetura da Ferramenta de Automatização do Mapeamento, (MACHADO et al., 2024).	45
Figura 16:	Especificação Estrutural do Cenário Inspector (adaptado de BORN, 2022).	47
Figura 17:	Especificação Funcional do Cenário Inspector (adaptado de BORN, 2022).	48
Figura 18:	Primeira parte da estrutura do cenário Inspector.	50
Figura 19:	Estrutura completa do cenário Inspector.	50
Figura 20:	Inscrições de guarda incluídas no cenário Inspector.	51
Figura 21:	Todas as inscrições incluídas no cenário Inspector.	52
Figura 22:	Diferentes simulações no cenário Inspector.	52
Figura 23:	RPC do modelo Inspector com falha.	53
Figura 24:	Grafo de Espaço de Estado para cenário Inspector.	53
Figura 25:	Rodando o caso de teste <i>A</i> para o cenário Inspector.	57
Figura 26:	Rodando o caso de teste <i>B</i> para o cenário Inspector.	57
Figura 27:	Especificação Estrutural do cenário <i>Soccer Team</i> , adaptado de (HÜBNER; SICHMAN; BOISSIER, 2002).	58
Figura 28:	Especificação Funcional do cenário <i>Soccer Team</i> , adaptado de (HÜBNER; SICHMAN; BOISSIER, 2002).	59
Figura 29:	Primeira parte da estrutura para o cenário Soccer Team.	60
Figura 30:	Segunda parte da estrutura para o cenário Soccer Team.	61

Figura 31: Estrutura Criada para o cenário Soccer Team.	62
Figura 32: Incrições de guarda incluídas no cenário Soccer Team.	63
Figura 33: Todas as inscrições incluídas no cenário Soccer Team.	64
Figura 34: Grafo de Espaço de Estado do cenário Soccer Team.	64
Figura 35: Rodando o caso de teste <i>A</i> para o cenário Soccer Team.	65
Figura 36: Rodando o caso de teste <i>B</i> com erro para o cenário Soccer Team.	67
Figura 37: Rodando o caso de teste <i>B</i> corrigido para o cenário Soccer Team.	67
Figura 38: Código Fonte XML da EE para o cenário Inspector.	75
Figura 39: Código Fonte XML da ED para o cenário Inspector.	75
Figura 40: Código Fonte XML da EF para o cenário Inspector.	76
Figura 41: Código Fonte XML da EE para o cenário Soccer Team.	77
Figura 42: Código Fonte XML da ED para o cenário Soccer Team.	77
Figura 43: Código Fonte XML da EF para o cenário Soccer Team.	78

LISTA DE TABELAS

Tabela 1:	Exemplo de Especificação Deontica do $\mathcal{M}\text{oise}^+$	29
Tabela 2:	Bases de dados bibliográficas utilizadas.	29
Tabela 3:	Sumarização dos trabalhos relacionados.	34
Tabela 4:	Exemplo de caso de teste caminho 1.	44
Tabela 5:	Especificação Deontica do Cenário Inspector.	48
Tabela 6:	Caso de teste A gerado para o cenário <i>Inspector</i>	55
Tabela 7:	Caso de teste B gerado para o cenário <i>Inspector</i>	56
Tabela 8:	Especificação Deontica para o cenário <i>Soccer Team</i>	58
Tabela 9:	Caso de teste A gerado para o cenário <i>Soccer Team</i>	65
Tabela 10:	Caso de teste B gerado para o cenário <i>Soccer Team</i>	66

LISTA DE ABREVIATURAS

SMA	Sistema Multiagente
RP	Redes de Petri
RPC	Redes de Petri Coloridas
EE	Especificação Estrutural
EF	Especificação Funcional
ED	Especificação Deontica
XML	<i>Extensible Markup Language</i>

1 INTRODUÇÃO

Os Sistemas Multiagente (SMA) representam um paradigma inovador para modelar, simular e projetar sistemas distribuídos complexos. Sua eficácia se deve à capacidade dos agentes de representar as entidades do sistema, seus comportamentos e interações. Na verdade, um agente possui características pró-ativas e reativas que são muito úteis na tomada de decisões. Além disso, uma característica fundamental dos SMA é a inteligência coletiva ou distribuída, e a capacidade de funcionar mesmo com a falha de um de seus componentes (agente), sem a necessidade de interrupção do sistema como um todo (BOUCHERIT et al., 2020).

Com o passar do tempo, os SMA vem ganhando importância nas mais diferentes áreas, devido às características únicas dos agentes, como reatividade, proatividade, autonomia e capacidade social (BARNIER et al., 2017). Porém, certas aplicações exigem um mínimo de confiabilidade para que o sistema não apresente nenhum risco ao usuário e ao ambiente em que está operando. Pode-se citar como exemplos: o controle de veículos autônomos, aplicações militares, gerenciamento de sistemas de distribuição de energia elétrica (*Smart Grids*) ou logística.

Para garantir que o sistema seja confiável e possa ser disponibilizado ao usuário, testes de software são necessários. O teste de software consiste na verificação dinâmica do comportamento de um programa em um conjunto de casos de teste selecionados adequadamente (ABRAN et al., 2004). Um caso de teste é uma entrada na qual o programa em teste é executado durante o teste (ZHU; HALL; MAY, 1997), e os testadores geralmente precisam gerar casos de teste para executar todas as instruções no programa pelo menos uma vez.

Existem algumas características dos SMA, como o não-determinismo, que dificultam a realização de testes. Como esses sistemas costumam ser programados para serem autônomos e deliberativos, e operam em um mundo aberto, o que os tornam sensíveis ao contexto (HOUHAMDI, 2011), e o número das diferentes saídas de execução para uma mesma entrada pode ser demasiadamente grande. Uma das maneiras de restringir esse comportamento dos SMA é representá-los como um grupo, através de modelos organizacionais. Tais modelos coordenam os agentes em grupos e hierarquias, além de estabelecer regras comportamentais específicas (ARGENTE; JULIAN; BOTTI, 2006).

Com uma linguagem de modelagem organizacional que decompõe explicitamente uma especificação em dimensões estruturais, funcionais e deônticas, o $\mathcal{Moise}^+$ (HÜBNER; SICHMAN; BOISSIER, 2002) é um modelo organizacional para agentes bastante referenciado. Nesse modelo, cabe a especificação estrutural definir a quais grupos pertencem os papéis, a relação existente entre os diferentes papéis e a quantidade máxima de agentes que podem desempenhar um determinado papel. Uma especificação funcional define a ordem da realização das metas e relaciona cada meta a uma missão. Por fim, uma especificação deôntica é responsável pela ligação entre as outras duas, relacionando papéis com missões.

No modelo $\mathcal{Moise}^+$, o conceito de meta global é utilizado, onde a meta representa o estado do mundo desejado pelo MAS, que difere da meta local que representa o estado desejado por um

único agente. As metas aqui têm quatro estados diferentes: *esperando*, *habilitado*, *impossível* e *alcançada* (BOISSIER et al., 2020). A Figura 1 mostra esses estados, onde toda meta possui um estado inicial de **esperando**, o que significa que as condições para que ela se torne habilitada (satisfação de outras metas) ainda não foram alcançadas, ou o esquema social ainda não está bem-formado. No estado de esperando, um agente já poderia estar comprometido com a meta (e sua respectiva missão), aguardando o próximo estado, **habilitado**, onde pode executar os planos para satisfazer essa meta. Esteja no estado de esperando ou habilitado, agentes comprometidos com as metas podem concluir que não são capazes de alcançá-lo, deixando a meta no estado de **impossível**. Mas, se tudo correr bem, e a meta for **alcançada**, é alcançado o último estado, também conhecido como satisfeito (BOISSIER et al., 2020).

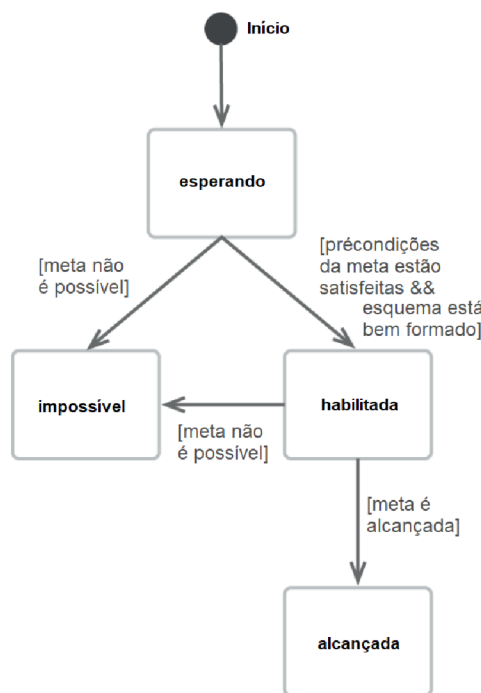


Figura 1: Ciclo de vida de uma meta do $\mathcal{Moise}^+$ adaptado de (BOISSIER et al., 2020).

Pode-se afirmar portanto que o modelo $\mathcal{Moise}^+$ funciona com um paradigma orientado a eventos, que são gerados principalmente quando as ações dos agentes são realizadas no sistema. Independentemente da fonte ou tipo de eventos, a programação orientada a eventos fala de um paradigma de programação em que o fluxo do programa é determinado pelos eventos (DASH et al., 2011). Esse tipo de funcionamento pode ser representado em máquinas de estado finitas que são modelos de comportamento composto por um número finito de estados, transições entre esses estados e ações.

Existem na literatura diferentes trabalhos que tratam de testes em SMA, inclusive na seção 2.7 são apresentados alguns deles. Porém poucos são os trabalhos que levam em consideração a camada organizacional de um SMA. Na Figura 2 apresentamos dois tipos de abordagem de teste, sendo a primeira chamada teste a nível de agente, onde o SMA pode ou não possuir um modelo organizacional, e a realização de testes tem como base a interação dos agentes entre

eles e o ambiente, sem as restrições impostas pela organização. Aqui, é possível identificar erros unitários, de agente, integração e sistema.

Na segunda abordagem tem-se o teste a nível organizacional. O SMA possui um modelo organizacional definido e os testes realizados são orientados por esse modelo. Aqui, além de erros no agente e erros de sistema, é possível encontrar o que seria um erro organizacional, que acontece quando existe uma falha na especificação do modelo. Assim, realizar testes a nível organizacional se torna uma operação importante para o desenvolvimento de SMA organizados, no sentido de ser um complemento para testar o sistema e podem ser efetuados em conjunto com testes a nível de agente.

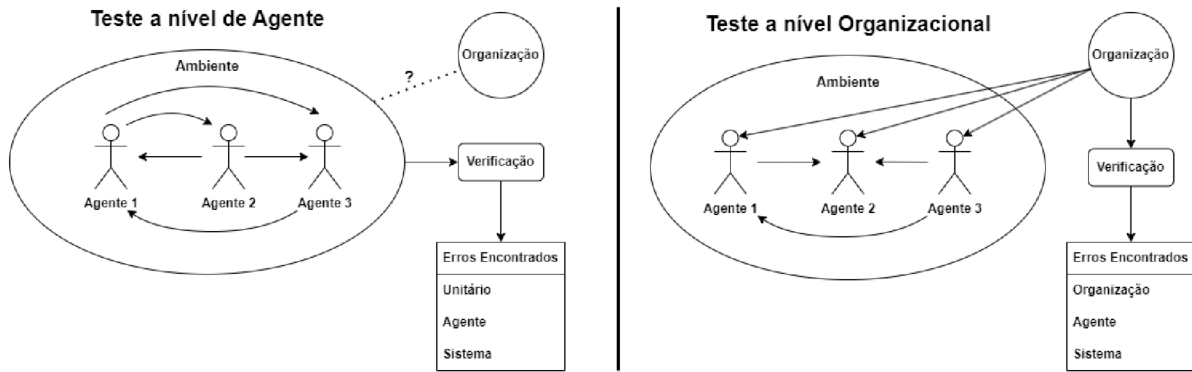


Figura 2: Diferença entre teste a nível Organizacional e Agente, Fonte: Autor.

Segundo CLARK; WALKINSHAW; HIERONS (2021), as abordagens utilizadas para geração de testes em SMA podem ser classificadas de acordo com a Figura 3. Um desses diferentes mecanismos são as Redes de Petri, que pertence ao mesmo grupo de Máquinas de Estado e AURL, denominado Caminho Percorrido, onde diferentes caminhos através do sistema correspondem a diferentes execuções possíveis e, portanto, casos de teste.

Uma Rede de Petri é um modelo abstrato e formal usado para descrever o fluxo de informação em sistemas, onde componentes dinâmicos são capturados por *tokens* que podem se mover entre estados, permitindo que comportamentos assíncronos e concorrentes sejam capturados (HADDAD; POITRENAUD, 1999). Dado que os agentes são frequentemente solicitados a realizar várias tarefas simultaneamente, as redes de Petri são uma ferramenta particularmente eficaz para especificar e testar o comportamento de sistemas baseados em agentes. Além disso, as redes de Petri são um dos formalismos mais confiáveis para simulação e análise de modelagem comportamental e análise de falha de SMA (BOUCHERIT et al., 2020).

Em (MACHADO; GONÇALVES, 2020) e (MACHADO, 2020), um método para testes em SMA foi descrito e formalizado utilizando o modelo organizacional $\mathcal{Moise}^+$. O método inclui um mapeamento do sistema em uma Rede de Petri Colorida (RPC) para dimensionamento dos casos de teste e a identificação e geração desses casos de teste. Uma das limitações deste modelo é que ele não considera o número de agentes comprometidos com uma missão. Porém, a quantidade de agentes precisa ser levada em consideração porque influencia diretamente na funcionalidade do sistema. Por exemplo, uma falha na comunicação entre agentes, que não

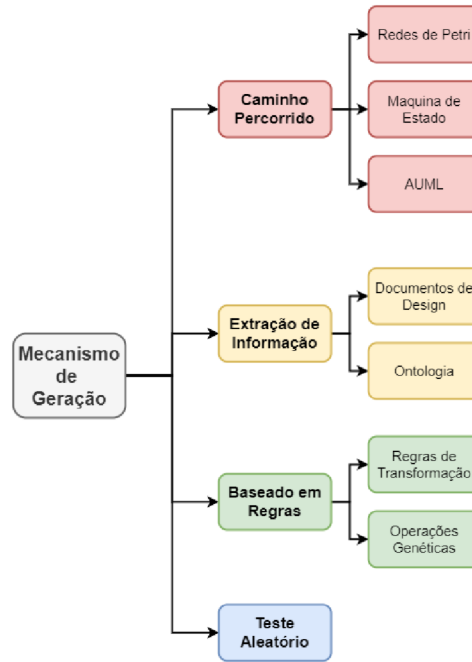


Figura 3: Classificação dos mecanismos para geração de teste, adaptado de (CLARK; WALKINSHAW; HIERONS, 2021).

seria possível detectar em um sistema com um único agente, pode gerar resultados distintos, dependendo do número de agentes no sistema.

Portanto, com objetivo de melhorar o modelo anterior, apresenta-se nessa proposta a inclusão das cardinalidades ao modelo, fazendo com que mais testes possam ser gerados e especificando a quantidade de agentes mínima e máxima permitidos para cada papel. Além disso, a inclusão dos grupos aos quais os agentes fazem parte, melhorando assim a fidelidade do mapeamento com o modelo $\mathcal{Moise}^+$ original. Também está aqui descrita uma descrição precisa para cada etapa do mapeamento utilizando axiomas, o que serviu como guia para a geração de uma ferramenta de automatização de todo o processo de mapeamento.

Nesse contexto, o desenvolvimento de ferramentas que auxiliem o desenvolvedor a realizar testes em SMA é uma necessidade para que sistemas com maior confiabilidade sejam desenvolvidos.

1.1 Hipótese de pesquisa

A hipótese de pesquisa é definida através das seguintes afirmações:

- A realização de testes em SMA tem um grau de dificuldade maior devido as características únicas dos agentes e da maneira com eles interagem;
- A utilização de modelos organizacionais em SMA restringe as ações dos agentes, melhorando a previsibilidade para a execução do sistema;
- Modelos organizacionais podem ser concebidos como modelos baseados em eventos e

máquinas de estado o que viabiliza o uso de técnicas de modelagem como as Redes de Petri, mais apropriada por que é capaz de representar atividades paralelas e distribuídas.

Logo, a hipótese de pesquisa deste trabalho é:

“Redes de Petri Coloridas podem ser utilizadas para testar sistemas multiagentes em nível organizacional.”

1.2 Objetivos

O objetivo geral desta tese é desenvolver um método formal para testes na camada organizacional de sistemas multiagentes. Esse método inclui um mapeamento de um modelo organizacional em uma rede de petri colorida para posterior análise com a contagem e geração do casos de testes.

Os objetivos específicos se resumem em:

- Estender o modelo proposto em (MACHADO, 2020) e (MACHADO; GONÇALVES, 2020), de forma a incluir a concepção de grupos organizacionais e de cardinalidade máxima e mínima de cada missão.
- Realizar uma descrição das variáveis do mapeamento entre conceitos e estruturas do Moise^+ para elementos estruturais e dinâmicos da RPC;
- Desenvolver uma ferramenta que automatize o método de mapeamento, tornando o processo mais rápido e menos suscetível a erros.
- Realizar a contagem, identificação e a posterior execução dos casos de teste no sistema em teste.

1.3 Estrutura do trabalho

Este trabalho está dividido em 5 capítulos, como segue:

1. No Capítulo 1 está a presente introdução do trabalho. Nela estão contidos a motivação, a hipótese de pesquisa, os objetivos e esta estrutura de organização.
2. O embasamento teórico sobre o tema de pesquisa juntamente com trabalhos relacionados é apresentado no Capítulo 2.
3. Uma descrição do método proposto no trabalho é apresentada no Capítulo 3.
4. No Capítulo 4, são apresentados dois casos de uso, bem como a análise dos mesmos.
5. Por fim no Capítulo 5 esta a conclusão do trabalho com considerações finais e trabalhos futuros.

2 EMBASAMENTO TEÓRICO

Na introdução, a hipótese de pesquisa, bem como os objetivos deste trabalho foram apresentados. Para resolver o problema de pesquisa, o embasamento teórico auxilia como fonte de referências, experiências e análises sobre os conceitos.

2.1 Agentes

Um agente é uma entidade real ou virtual, imersa em um determinado ambiente, onde pode realizar determinadas ações, perceber e representar parcialmente esse ambiente, e trocar informações com outros agentes. Esse agente apresenta comportamento autônomo como resultado de observações, conhecimento armazenado e interações com outros agentes no ambiente (FERBER; GASSER, 1991).

Assim, agentes são entidades autônomas, que gerenciam por si mesmos seu próprio estado e comportamento. WOOLDRIDGE; JENNINGS (1995) listaram algumas características que diferem os agentes de outros programas de software, sendo elas:

- **Autonomia:** os agentes são independentes e tomam suas próprias decisões, sem intervenções diretas de outros agentes ou seres humanos;
- **Reatividade:** os agentes reagem às mudanças que ocorrem no ambiente em que estão situados;
- **Pró-atividade:** os agentes agem de maneira direcionada aos objetivos;
- **Sociabilidade:** os agentes interagem com outros agentes através de algum tipo de protocolo de comunicação.

Os agentes podem ser divididos em duas classes: agentes reativos e agentes cognitivos. Os agentes reativos têm funcionamento simples pois escolhem suas ações baseados unicamente nas percepções que têm do ambiente. Já os agentes cognitivos são mais complexos, pois possuem representações internas do ambiente e funcionam de forma a raciocinar para construir um plano de ações que leva a realização do objetivo pretendido (WOOLDRIDGE, 1999).

Quanto a arquitetura dos agentes, que modela seu comportamento, pode-se destacar a arquitetura BDI (*Belief Desire Intention*), um modelo clássico da literatura que foi criada por RAO; GEORGEFF (1991), onde agentes trabalham com a ideia de crenças, desejos e intenções. Na Figura 4 pode-se ver um diagrama representando a arquitetura BDI (WOOLDRIDGE, 1999), contendo sete componentes principais:

- **Crenças** (do inglês *Beliefs*): um conjunto de crenças atuais representando a informação que o agente tem sobre seu ambiente;

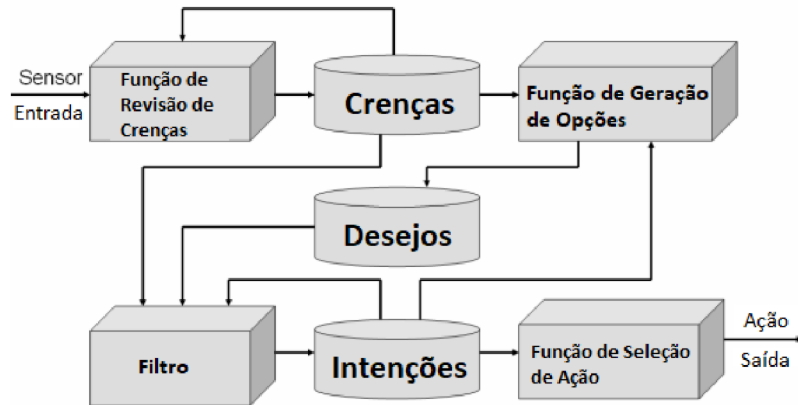


Figura 4: Arquitetura BDI genérica, adaptado de (WOOLDRIDGE, 1999).

- **Função de Revisão de Crenças:** determina um novo conjunto de crenças a partir da percepção da entrada e das crenças do agente;
- **Função de Geração de Opções:** determina as opções disponíveis ao agentes (seus desejos), com base nas suas crenças sobre seu ambiente e nas suas intenções;
- **Desejos** (do inglês *Desires*): um conjunto de estados ou objetivos que um agente deseja alcançar;
- **Filtro:** uma função de filtro, a qual representa o processo de deliberação do agente, que determina as intenções do agente com base nas suas crenças, desejos e intenções atuais;
- **Intenções** (do inglês *Intentions*): um conjunto de intenções atual, que representa o foco atual do agente, isto é, aqueles estados que o agente está determinado a alcançar;
- **Função de Seleção de Ação:** determina uma ação a ser executada com base nas suas intenções atuais.

2.2 Sistemas Multiagente

Um Sistema Multiagente pode ser visto com um conjunto de agentes que interagem entre si de forma cooperativa com a finalidade de atingir um determinado objetivo global. Como esse tipo de sistema não depende apenas de único agente para a realização dos objetivos, mesmo que ocorra um eventual problema em um agente, o funcionamento não necessariamente será prejudicado, uma vez que os outros agentes podem assumir o lugar desse agente. Por esse motivo, a utilização de SMA é vantajosa quando existe a necessidade de uma maior robustez e tolerância a falhas.

Existem algumas características que diferem um SMA de outros sistemas mais tradicionais (VLASSIS, 2007):

- **Design do agente:** Frequentemente os agentes de um mesmo SMA são projetados de maneiras diferentes como na parte comportamental. A heterogeneidade do agente pode afetar todos os aspectos funcionais de um agente, desde a percepção até a tomada de decisão.
- **Ambiente:** Os agentes precisam lidar com ambientes que podem ser estáticos ou dinâmicos. Em um SMA, a mera presença de múltiplos agentes faz o ambiente parecer dinâmico do ponto de vista de cada agente.
- **Percepção:** O fato de os agentes poderem observar coisas diferentes torna o mundo parcialmente observável para cada agente, o que tem várias consequências na tomada de decisão dos agentes.
- **Controle:** Ao contrário dos sistemas de agente único, o controle em um SMA é tipicamente descentralizado. O controle descentralizado é preferível ao controle centralizado por razões de robustez e tolerância a falhas.
- **Conhecimento:** Em um SMA, os níveis de conhecimento de cada agente sobre o estado atual do mundo podem diferir substancialmente. Assim, cada agente deve também considerar o conhecimento de cada outro agente em sua tomada de decisão.
- **Comunicação:** A interação é frequentemente associada a alguma forma de comunicação. A comunicação pode ser usada em vários casos, por exemplo, para coordenação entre agentes cooperativos ou para negociação entre agentes de interesse próprio.

A classificação de um SMA é uma questão complexa, pois pode ser feita com base em vários atributos diferentes como arquitetura, aprendizagem, comunicação e coordenação. Considerando apenas a arquitetura interna dos agentes que formam um SMA, esse sistema pode ser classificado em dois tipos (PARASUMANNA GOKULAN; SRINIVASAN, 2010):

- **Estrutura Homogênea:** todos os agentes que formam o sistema multiagente têm a mesma arquitetura interna. A arquitetura interna refere-se aos objetivos locais, capacidades do sensor, estados internos, mecanismos de inferência e ações possíveis.
- **Estrutura Heterogênea:** os agentes podem diferir em capacidade, estrutura e funcionalidade. A arquitetura heterogênea ajuda a tornar os aplicativos de modelagem muito mais próximos do mundo real.

A comunicação é um dos componentes cruciais em sistemas multiagente e pode ser classificada em dois tipos: comunicação local (*peer to peer*) ou quadro negro (*blackboard*). Na comunicação local, os agentes trocam mensagens entre si diretamente sem a necessidade de um local para armazenamento da informação, esse tipo de comunicação é indicada para aplicações descentralizadas. Já na comunicação por quadro negro, um grupo de agentes compartilha um

mesmo repositório de dados para enviar e receber mensagens (PARASUMANNA GOKULAN; SRINIVASAN, 2010).

Existem diferentes aplicações para os SMA, por exemplo na indústria de jogos (HAGELBÄCK; JOHANSSON, 2008), na indústria da automação (HERRERA et al., 2008), na indústria cinematográfica¹, ou mesmo para modelar uma epidemia como foi utilizado mais recentemente no Covid-19 (SHARMA et al., 2020), ou seja, para situações onde estão envolvidos aspectos sociais e ambientais.

2.3 Teste de Software

O teste é uma fase de desenvolvimento de software com o objetivo de avaliar a qualidade do produto e aprimorá-lo, detectando **erros** e **problemas**. O teste preocupa-se em estabelecer que um sistema desenvolvido funcione adequadamente, levando em consideração os requisitos do usuário e a especificação do sistema associado. Estes dois aspectos essenciais dos testes são comumente referidos como validação e verificação (ABDULLAH; BURNSTEIN, 2003).

“**Validação** é processo de avaliar um sistema ou componente durante ou no final do processo de desenvolvimento para determinar se ele satisfaz os requisitos especificados.

Verificação é o processo de avaliar um sistema ou componente para determinar se os produtos de uma determinada fase de desenvolvimento satisfazem as condições impostas no início dessa fase.” (ABDULLAH; BURNSTEIN, 2003)

Durante o procedimento de teste, localizam-se erros no código ou especificação. Nesse sentido, erros podem levar a falhas na execução do sistema, desviando entre os resultados reais e esperados. Um método de teste geralmente se baseia em quatro componentes principais: um artefato de informação, um mecanismo de geração, critérios de adequação de dados de teste e um oráculo de teste, que podem ser vistos na Figura 5.

Esses componentes são descritos da seguinte maneira (CLARK; WALKINSHAW; HIERONS, 2021):

- **Artefato de Informação:** Uma fonte de informação usada como guia para gerar casos de teste. Pode ser uma especificação formal, um documento de projeto ou código-fonte e fornece informações sobre a funcionalidade do sistema ou os objetivos necessários para produzir casos de teste significativos.
- **Mecanismo de Geração:** Um algoritmo ou estratégia adotada para gerar um caso de teste a partir de um artefato de informação.
- **CrITÉrios de Adequação de Dados de Teste:** Uma medida da qualidade do caso de teste em relação a uma meta de teste, como cobertura. Um critério de adequação de

¹<https://www.massivesoftware.com>

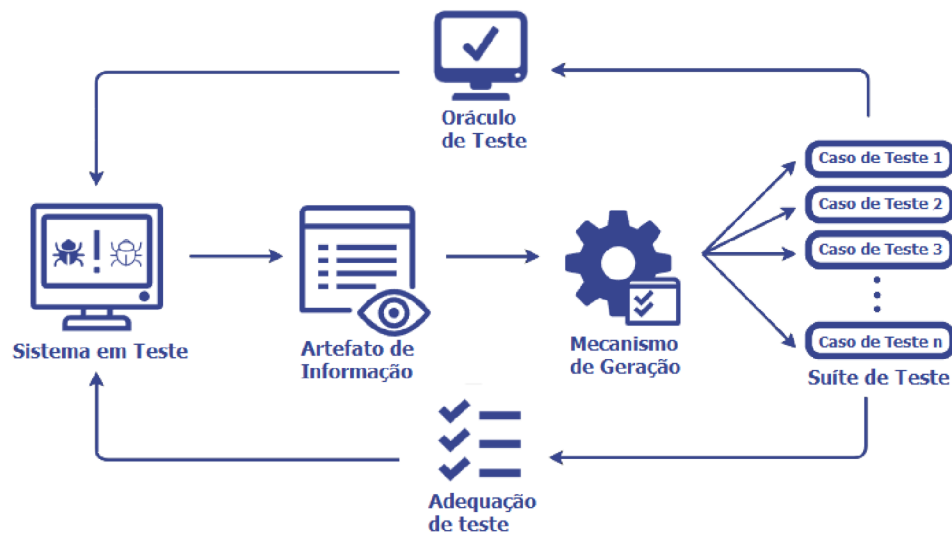


Figura 5: Uma visão geral dos processos de teste, adaptado de (CLARK; WALKINSHAW; HIERONS, 2021).

dados de teste é frequentemente usado para orientar a geração de casos de teste para atingir a meta definida.

- **Oráculo de Teste:** Um procedimento para avaliar se a execução de um teste está correta ou não. Pode ser especificado, derivado, implícito ou humano.

Durante o processo de teste são gerados casos de teste para executar todas as instruções no programa pelo menos uma vez. Um caso de teste é uma entrada na qual o programa em teste é executado durante o teste. Um conjunto de testes é um conjunto de casos de teste para testar um programa (ZHU; HALL; MAY, 1997).

Para alcançar uma boa cobertura de teste é necessária a escolha de um critério de adequação de teste apropriado para o sistema a ser testado. Alguns tipos de critérios de adequação utilizados são:

- **Instruções (*statement*):** O critério de adequação por instruções é atendido por um conjunto de testes específico para um programa específico se cada instrução executável no programa for executada por pelo menos um caso de teste no conjunto de testes.
- **Ramificações (*branch*):** Esse critério mede um programa pelo número de ramificações que ele contém e informa aos usuários quais dessas ramificações não foram obtidas durante a execução do teste.
- **Caminhos (*path*):** O critério todos os caminhos exige que todos os caminhos de execução desde a entrada do programa até sua saída sejam executados durante o teste.

2.3.1 Teste de Sistemas Multiagente

No caso de SMA, existem algumas características que tornam o teste mais complexo do que em sistemas convencionais. A realização de testes nesses sistemas requer novas técnicas que lidem com sua natureza específica. Segundo HOUHAMDI (2011), as principais razões para esse grau de dificuldade são:

- **Maior complexidade:** existem vários processos distribuídos que são executados de forma autônoma e simultânea;
- **Quantidade de dados:** os sistemas podem ser compostos por milhares de agentes, cada um com seus próprios dados;
- **Não-determinismo:** não há garantia que duas execuções do mesmo sistema com as mesmas entradas levem ao mesmo estado, dificultando a procura de erros;
- **Alcançabilidade:** uma vez que não é possível determinar antecipadamente todas as interações de um agente durante sua execução.

Existem diferentes níveis de testes que podem ser aplicados nos SMA. Uma classificação foi descrita por NGUYEN et al. (2009), que separa os testes em:

- **Unidade:** testar unidades de código e módulos que compõem os agentes como metas, planos, crenças, sensores, mecanismo de raciocínio e assim por diante;
- **Agente:** testar a integração dos diferentes módulos dentro de um agente; testar os recursos dos agentes para preencher seus objetivos e detectar o ambiente;
- **Integração:** testar a interação de agentes, protocolos de comunicação e semântica, interação de agentes com o ambiente, integração de agentes com recursos compartilhados, cumprimento de normas; observar propriedades emergentes; certificar que um grupo de agentes e os recursos do ambiente funcionem corretamente juntos;
- **Sistema:** testar o SMA como um sistema em execução no ambiente operacional de destino; teste para propriedades de qualidade que o sistema pretendido deve atingir, como adaptação, abertura, tolerância a falhas, desempenho;
- **Aceitação:** Testar o SMA no ambiente de execução do cliente e verificar se este atende aos objetivos das partes interessadas.

Embora existam diferentes níveis de teste, as dificuldades para se testar um SMA fazem com que existam poucas ferramentas disponíveis para testes nestes sistemas. Na Seção 2.7 são apresentadas algumas dessas ferramentas. A criação de novas metodologias é necessária para que essa área evolua e consiga tornar os SMA competitivos no mercado atual.

2.4 Redes de Petri Coloridas

Uma Rede de Petri (RP) é uma ferramenta para modelagem gráfica e matemática aplicável a muitos sistemas. Elas são usadas como um auxílio de comunicação visual similar a fluxogramas. As RP são divididas em duas classes: RP de baixo nível e RP de alto nível.

Nas RP de baixo nível (MURATA, 1989), existe apenas um tipo de marca, o que não permite a diferenciação de recursos em um lugar, sendo necessários diferentes lugares para expressar os recursos similares. Essa limitação faz com que essas redes sejam inadequadas para a aplicação de sistemas reais e complexos já que os modelos obtidos possuem grandes dimensões além de serem não-hierárquicos dificultando sua compreensão.

Quanto as RP de alto nível tem-se como exemplo as chamadas Redes de Petri Coloridas (RPC), que combinam as capacidades de uma RP ordinária com as capacidades de uma linguagem de programação. Nas RPC, as fichas possuem um valor de dados anexado que são chamados de cores. Essas cores podem representar diferentes processos ou recursos de uma rede, reduzindo assim o tamanho dos modelos. Para um determinado lugar, todas as fichas devem ter cores que pertençam a um tipo específico, que é chamado de conjunto de cores do lugar. Esse uso de conjunto de cores é equivalente aos tipos de dados nas linguagens de programação (inteiro, caractere, booleano) e a especificação dos conjuntos de cores e das variáveis da rede é feita por declarações (JENSEN, 1997).

Para gerar uma RPC é preciso modelar a estrutura e em seguida incluir inscrições e declarações. A estrutura é composta por um grafo dirigido com dois tipos de vértices (lugares e transições). Os lugares são representados por círculos ou elipses, e as transições por retângulos. Cada lugar, transição e arco da rede possui inscrições associadas. Por exemplo, os lugares tem três tipos de inscrições: nomes, conjunto de cores ou expressão de inicialização (é a marcação inicial). Já as declarações são as especificações dos conjuntos de cores e das variáveis.

Um exemplo que ilustra os elementos de uma RPC é o *Dining Philosophers*, Figura 6, que constitui uma representação na rede para um problema clássico de sincronização que é usado para avaliar situações em que há necessidade de alocar vários recursos a vários processos. Os elementos contidos nesse exemplo são: três lugares chamados *Think*, *Eat* e *UnusedChopsticks*; duas transições *TakeChopsticks* e *PutDownChopsticks*; os tokens (ou cores) *ph* e *cs*, e os respectivos conjuntos de cores *PH* e *CS*; uma variável *p* e uma função de arco chamada *Chopsticks()*. Na Figura 6 também é possível visualizar as declarações que estão localizadas na parte esquerda. Esse exemplo foi gerado utilizando a ferramenta de modelagem para RPC chamado *CPNTools*² (JENSEN; KRISTENSEN; WELLS, 2007), sendo este um dos exemplos contidos no conjunto de modelos da ferramenta.

²<http://cpnTools.org>

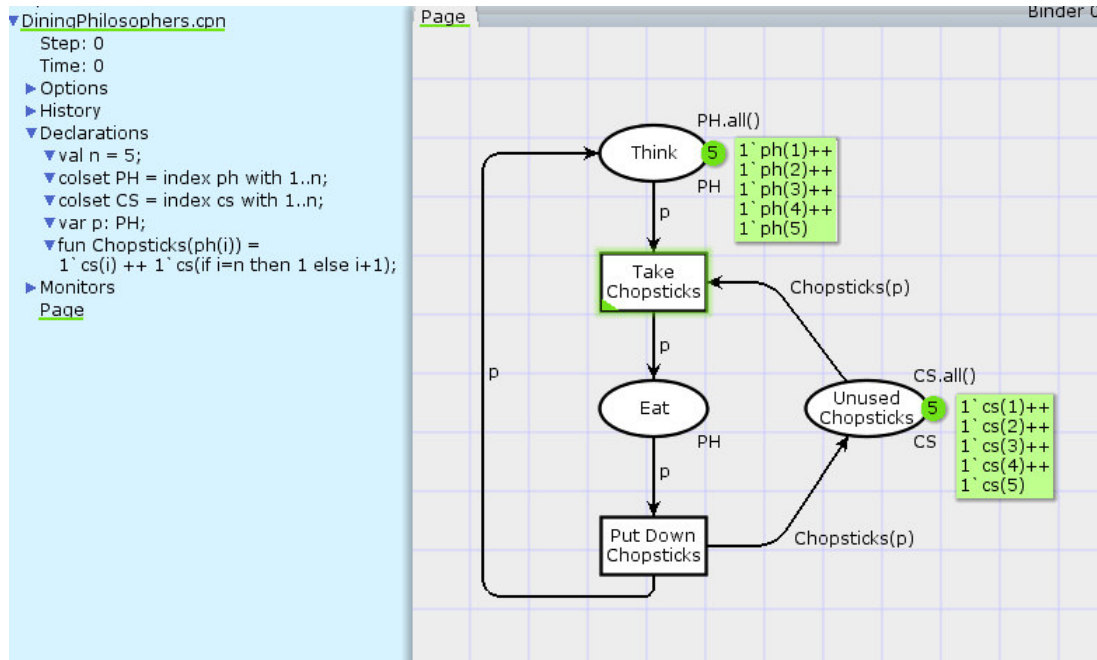


Figura 6: RPC: Exemplo do *Dining Philosophers*.

2.5 A Plataforma JaCaMo

JaCaMo (BOISSIER et al., 2013) é um *framework* para programação de SMA que combina três diferentes tecnologias: (i) Jason, uma plataforma para desenvolvimento de SMA que incorpora uma programação orientada a agentes; (ii) CArTAgO, um *framework* para a programação do ambiente; (iii) O *Moise+* que implementa um modelo de programação para a dimensão organizacional.

A plataforma JaCaMo oferece um recurso para escalabilidade de aplicações complexas, permitindo sua distribuição em vários nós. Além disso, cada uma das ferramentas que compõem a plataforma utiliza um conjunto de abstrações, modelos e metamodelos para programação. O metamodelo define as dependências, conexões e mapeamentos conceituais entre as três ferramentas que integram a plataforma (CASTRO et al., 2018).

De acordo com BOISSIER; HÜBNER; RICCI (2016), pode-se usar o JaCaMo em diversas áreas, como: aplicações de computação ambiental, gestão inteligente de construção, ambientes virtuais inteligentes, gestão do conhecimento ou gestão de crises. A Figura 7 apresenta a visão principal desta plataforma, onde pode-se ver as três dimensões de um SMA (organização, agente e ambiente).

2.6 O Modelo Organizacional *Moise+*

Inicialmente, o desenvolvimento de SMA teve como foco o comportamento individual do agente, que ficou conhecido como abordagem centrada no agente. Com o tempo, tornou-se necessário um novo método onde a unidade conceitual básica é uma organização multiagente, ao invés do nível de agentes individuais. Então orientação deixou de ser o comportamento do

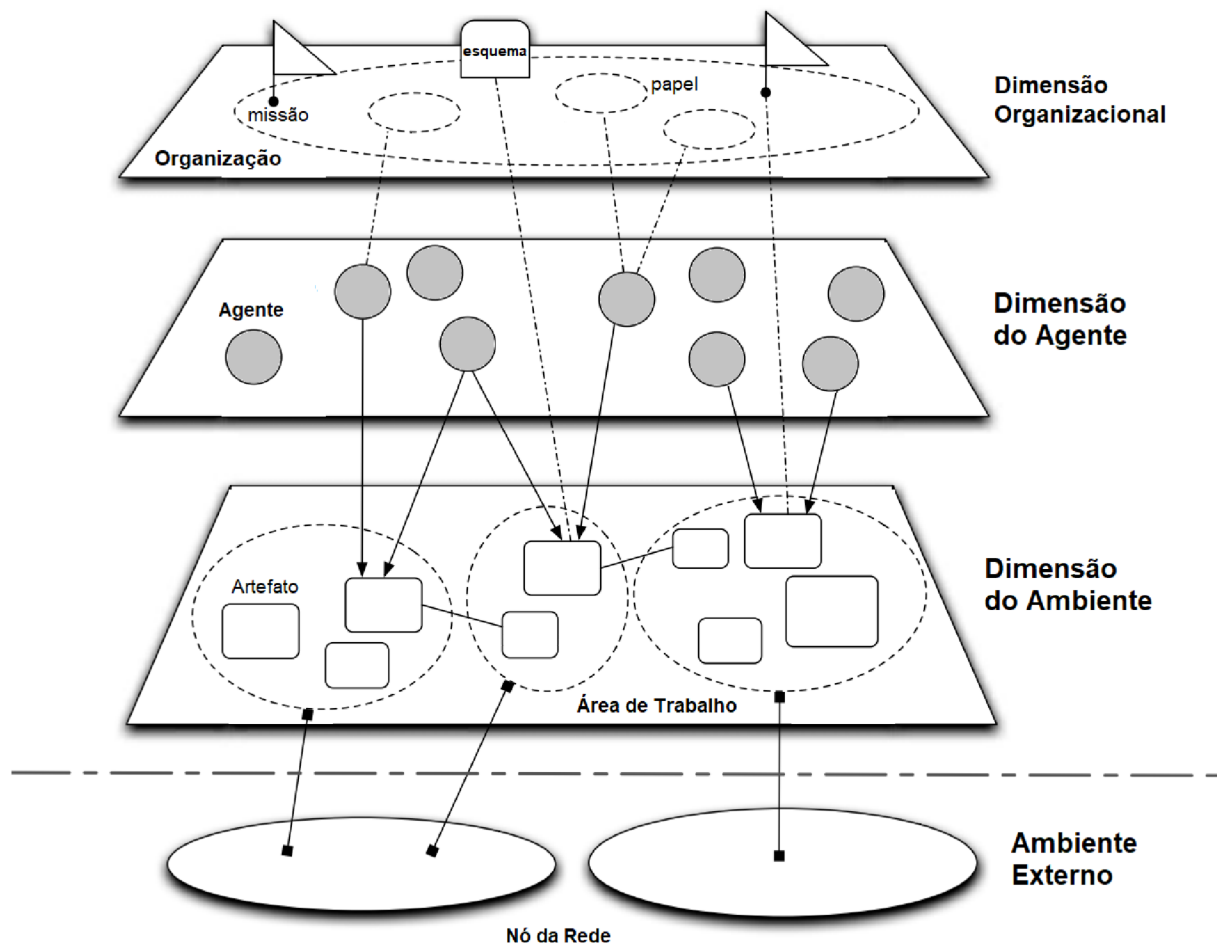


Figura 7: Visão geral do JaCaMo, adaptado de (BOISSIER et al., 2013).

agente e passou a ser a organização, sendo esse método conhecido como abordagem centrada na organização (LEMAÎTRE; EXCELENTE; EL FALLAH-SEGHRUCHNI, 1999).

Segundo WOOLDRIDGE; JENNINGS; KINNY (2000), uma organização é como uma coleção de papéis, que possuem relacionamentos uns com os outros, e que participam de padrões sistemáticos institucionalizados de interações com outros papéis. A organização de SMA pode ser entendida a partir de duas perspectivas: organização como um processo e organização como uma entidade. Ou seja, a organização é considerada tanto como o processo de organização de um conjunto de indivíduos, ou como uma entidade em si, com seus próprios requisitos e objetivos (DIGNUM, 2009).

Existem atualmente diferentes modelos de organização para SMA, um deles é o *Moise*⁺, que implementa um modelo de programação para a dimensão organizacional de um SMA. Essa abordagem inclui uma linguagem para a organização do sistema e especificação de sua infraestrutura, com suporte para mecanismos de raciocínio baseados na organização no nível do agente. Esse modelo contém uma Especificação Estrutural (EE) e uma Especificação Funcional (EF) independentes, que estão ligadas por uma Especificação Deontica (ED) (HÜBNER;

SICHMAN, 2007).

2.6.1 Especificação Estrutural

A Especificação Estrutural (EE) do $\mathcal{Moise}^+$ é formada por três níveis (HübNER; SICHMAN, 2007):

1. **Individual:** os comportamentos pelos quais um agente é responsável quando adota um papel;
2. **Social:** as ligações de conhecimento, comunicação e autoridade entre os papéis;
3. **Coletivo:** a agregação dos papéis em grupos.

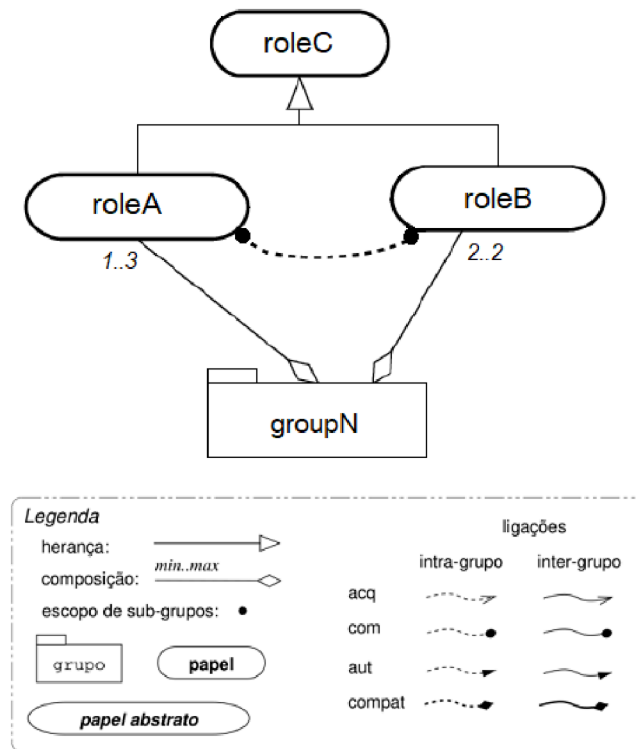


Figura 8: Exemplo de Especificação Estrutural do $\mathcal{Moise}^+$ (HübNER; SICHMAN, 2007).

A Figura 8 apresenta um exemplo com os elementos da EE e as ligações entre eles. Nesse exemplo existem três papéis, sendo que os dois primeiros com nome de *roleA*, *roleB* possuem uma relação de herança com o terceiro papel chamado *roleC*. Ambos *roleA* e *roleB* fazem parte de um mesmo grupo denominado *groupN*, que na sua composição aceita no mínimo 1 e no máximo 3 agentes *roleA*, e somente 2 agentes *roleB*. Entre *roleA* e *roleB* também existe uma ligação de comunicação, indicando que dois agentes possam se comunicar, caso pertençam a um desses papéis.

2.6.2 Especificação Funcional

A Especificação Funcional (EF) no $\mathcal{Moise}^+$ é constituída por um conjunto de esquemas sociais e uma relação de preferência de missões. Para o esquema social é utilizado o conceito de meta global, que representa um estado do mundo desejado pelo SMA. Em outro contexto, a meta local representa o objetivo de um único agente. A EF possui dois níveis sendo o nível individual onde o esquema social é constituído por missões e o nível coletivo onde o próprio esquema social é uma árvore de decomposição de metas globais, onde a raiz é a meta do esquema social e a decomposição das metas é feita através de planos (HÜBNER, 2003).

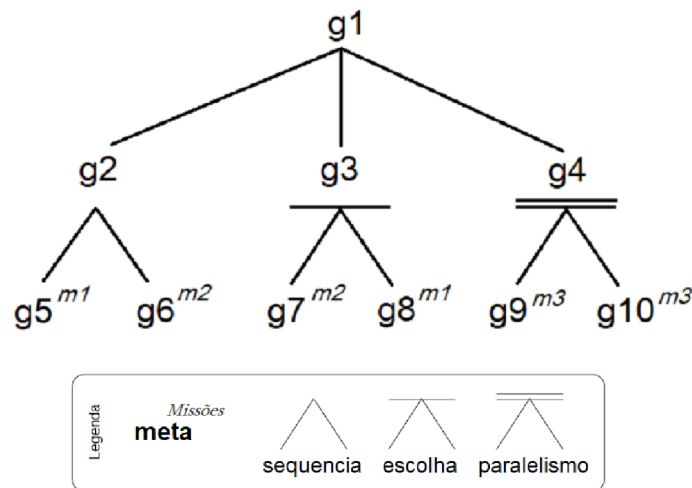


Figura 9: Exemplo de Especificação Funcional do $\mathcal{Moise}^+$ (HÜBNER; SICHMAN, 2007).

A Figura 9 apresenta um exemplo de uma árvore de decomposição de metas para um esquema social. A leitura é feita da esquerda para a direita e de baixo para cima. Por exemplo, nesse esquema, um agente que assuma a missão $m1$ compromete-se a realizar a meta $g5$ e na sequência um agente com a missão $m2$ executa a meta $g6$. Quando ambas as metas $g5$ e $g6$ forem executadas com sucesso, a meta $g2$ automaticamente será considerada concluída. Na EF existem três tipos de operadores usados para a construção de um plano sendo eles:

- **Sequência** “,”: No plano “ $g2 = g5, g6$ ” a meta $g2$ será satisfeita se as metas $g5$ e $g6$ forem satisfeitas nessa ordem;
- **Escolha** “|”: No plano “ $g3 = g7|g8$ ” a meta $g3$ será satisfeita caso uma, e apenas uma, das metas $g7$ ou $g8$ forem satisfeitas;
- **Paralelismo** “||”: No plano “ $g4 = g9||g10$ ” a meta $g4$ será satisfeita se ambas $g9$ e $g10$ forem satisfeitas paralelamente.

2.6.3 Especificação Deontica

A Especificação Deontica (ED) descreve as permissões e obrigações dos papéis para as missões, tendo assim uma função de ligação entre a EE e a EF. Uma *permissão*(p, m) afirma

que um agente que desempenha o papel p tem permissão para se comprometer para a missão m . Além disso, uma *obrigação*(p, m) afirma que um agente com o papel p deve se comprometer com a missão m (HÜBNER, 2003). Por exemplo, a ED mostrada na Tabela 1, onde pode-se dizer que o papel *roleA* tem permissão para realizar a missão *m1*, então qualquer agente com o papel *roleA* pode realizar uma meta relacionada a *m1*.

Tabela 1: Exemplo de Especificação Deontica do *Moise*⁺.

papel	relação deontica	missão
roleA	<i>permissão</i>	<i>m1</i>
roleB	<i>obrigação</i>	<i>m2</i>
roleA	<i>obrigação</i>	<i>m3</i>

2.7 Trabalhos Relacionados

Nessa seção é apresentada uma lista de trabalhos que tenham relação com a área de testes em SMA. Para a busca foram utilizadas as palavras-chave no idioma inglês: “multi-agent system”, “test”, “verification”, “validation”, “error”, “fail”. Foram selecionados artigos de diferentes bases de dados, conforme apresenta a Tabela 2. A consulta foi realizada dentro do período de 2012 à 2024.

Tabela 2: Bases de dados bibliográficas utilizadas.

Nome	URL
ACM Digital Library	https://dl.acm.org/
ArXiv	https://arxiv.org/
IEEE Xplore	https://ieeexplore.ieee.org/
ScienceDirect	https://www.sciencedirect.com/
Scopus (Elsevier)	https://www.scopus.com/
SpringerLink	https://link.springer.com/

Entre os resultados encontrados foram selecionados 18 trabalhos com base nos critérios de inclusão baseado em: testes em SMA, geração de casos de teste de SMA ou utilização de redes de petri para modelagem de SMA.

WANG; ZHU (2012) propuseram um *framework* de automação de testes chamado CA-Test, utilizando a linguagem de especificação formal baseada em agentes SLABS (ZHU, 2001). Durante a execução do programa de teste o comportamento dos agentes é monitorado e gravado. Então esses comportamentos gravados são checados através de um verificador de correção e um verificador de adequação para garantir que estejam funcionando de acordo com a especificação. Uma limitação é a não existência de um gerador de casos de teste, eles precisam ser definidos manualmente pelo usuário, porém segundo o autor uma ferramenta de geração automática pode ser facilmente integrada.

EASSA et al. (2014) desenvolveram uma ferramenta de teste dinâmico que usa uma linguagem de asserção, declarativa, de lógica temporal para detectar erros em tempo de execução

dos agentes. A linguagem foi uma proposta dos próprios autores e através das declarações inseridas, a ferramenta pode identificar erros dinâmicos durante a execução do programa de teste. Essa ferramenta gera agentes de teste para monitorar, controlar e gerar os testes que podem ser tanto a nível de agente quanto de integração.

Em **KERRAOUI et al. (2016)** é apresentada uma abordagem de teste em SMA baseada em Modelos. Primeiramente, é gerado e validado um modelo que represente o comportamento do sistema através de uma Rede de Petri de alto nível. Em seguida, os casos de teste são gerados automaticamente tendo como entrada o modelo comportamental do sistema. Por fim, uma versão instrumentada do modelo é gerada para a execução dos casos de teste e geração dos resultados.

No trabalho de **FRASHERI et al. (2017)**, um conceito inicial de um agente autônomo adaptativo, foi analisado com relação à tolerância a falhas por meio de Redes de Petri. A análise de falha é realizada modelando os agentes através de Redes de Petri Coloridas. Os resultados da simulação indicam a quantidade de tarefas que é descartada ou completada pelos agentes. A partir dos resultados experimentais, a análise mostrou a correlação da probabilidade de falha do sistema (aumento do número de tarefas perdidas) com a falha de um agente.

Já em **WINIKOFF (2017)**, com foco em programas de agentes BDI, analisa sua testabilidade em relação ao critério de adequação de testes de todas as arestas “*all edges*”. Para isso, ele faz uma comparação de quantidades de testes necessários para os critérios todas as arestas e todos os caminhos “*all paths*”, e mostram que o número de testes necessários para cobrir todas as arestas é bem menor do que todos os caminhos. Esse artigo também faz uma comparação para mostrar o quanto programas BDI são mais difíceis de testar do que programas processuais de tamanho equivalente. Portanto, ele realiza uma avaliação de certos critérios para que possam ser analisados e utilizados em futuras aplicações. Segundo o autor, as comparações que foram feitas podem ser representadas a nível de agente, partes de um agente (unidade) ou o sistema todo.

REHMAN; NADEEM; SINDHU (2019) apresentam uma abordagem de teste baseada em modelo que utiliza a ferramenta *Prometheus* para a modelagem do SMA. Para a geração de casos de teste foram utilizados sete diferentes critérios de cobertura definido pelos próprios autores. A geração de caminhos de teste é automatizada com a ajuda de uma ferramenta que recebe um modelo de teste como entrada, aplica os diferentes critérios de cobertura e gera caminho de teste para cada critério de cobertura.

BENAC EARLE; FREDLUND (2019) descrevem um *framework* para testar SMA escritos na linguagem de programação Jason, usando a técnica de teste conhecida como teste baseado em propriedades. Essa técnica consiste em gerar automaticamente casos de teste, a partir de uma descrição mais abstrata do comportamento do sistema em teste: as propriedades. A abordagem substitui um subconjunto dos agentes no sistema multiagente por uma máquina de estados. Por sua vez a máquina de estados desempenha o papel desses agentes, interagindo com os agentes reais restantes, enviando mensagens, modificando o ambiente, julgando se esses agentes reais restantes são implementados corretamente, examinando as mensagens enviadas

para qualquer agente substituído e as percepções de crença que eles recebem.

NASCIMENTO et al. (2020) apresentam uma abordagem baseada em metadados e no paradigma de publicação-assinatura para desenvolver aplicativos de teste que abordam o processo de diagnóstico de falhas em um SMA auto-organizado no contexto do comportamento local e global. Para ilustrar o uso dessa abordagem, foi desenvolvido, utilizando a plataforma JADE, um SMA auto-organizado baseado na internet das coisas, que simula um conjunto de luzes de rua inteligentes. Para validar os casos de teste, a aplicação de teste gera automaticamente máquinas de estado, onde cada estado representa uma ação.

Em **BOUCHERIT et al. (2020)**, os autores utilizam Redes de Petri e a lógica de reescrita para facilitar a formalização de aplicações críticas de segurança baseadas em multiagentes. Apresentam um algoritmo que permite a geração automática de especificações de Maude para modelos de sistemas multiagente baseados em Redes de Petri. Além disso, é feita uma integração das técnicas de verificação de modelo e teste baseado em propriedade nas fases de verificação e teste.

MIHRET; LIU (2021) apresentam uma abordagem para a geração de casos de teste, utilizando um verificador de modelos chamado PAT (SUN et al., 2009), para sistemas colaborativos. Para viabilizar oscasos de teste gerados, foi utilizada a estrutura de simulação JADE para um cenário de pouso de aeronaves no domínio de controle de tráfego aéreo.

ENGELMANN et al. (2022) apresentam uma abordagem de verificação em tempo de execução para sistemas multiagente utilizando a plataforma *framework* JaCaMo. A verificação é feita através de uma extensão para um SMA que tenha sido implementado usando JaCaMo. Essa extensão realiza um monitoramento para extrair e verificar execuções do SMA em relação às propriedades formais relacionadas a comunicação dos agentes.

Em **(GONÇALVES et al., 2022)** é descrita a formalização para geração de casos de teste partindo de um modelo organizacional $\mathcal{Moise}^+$ e mapeando o modelo em uma RPC para gerar os casos de teste a partir de dois diferentes critérios de cobertura. No final foram executados os casos de teste através da plataforma JaCaMo, identificando possíveis erros do sistema.

EL HOUDA; SOUFIENE; DJABER (2022) propõe uma abordagem de teste baseado em modelo para testar as interações entre agentes. A abordagem usa um diagrama de sequência AUML como modelo e restrições expressas na linguagem OCL (*Object Constraint Language*). A abordagem gera um conjunto de casos de teste capazes de, individualmente, abranger interações entre agentes, bem como possíveis cenários que podem ser executados de maneira inclusiva, exclusiva ou paralela. Além disso foram utilizados algoritmos genéticos para acompanhar a avaliação da introdução de novas interações no sistema em teste.

Em **(HE; LIU; ZHOU, 2022)** são utilizadas redes de petri ordinárias para descrever formalmente tanto o processo de interação e colaboração dos SMA quanto suas evoluções epistêmicas. Ao se deparar com um problema de explosão de estados da RP o que tornava a verificação inviável, foi adotando um diagrama de decisão binário ordenado reduzido (ROBDD) para codificar e armazenar todos os estados alcançáveis, e ao mesmo tempo projetado um algoritmo para garantir que as relações de transição e de equivalência específicos de alguns estados

também fosse armazenada reduzindo substancialmente o tempo de verificação. Para o experimento foi utilizado um *benchmark* chamado protocolo dos criptógrafos jantando, que trata do problema de privacidade de SMA.

ALDAHASH; KING; BENABID (2023) Apresenta uma modelagem de SMA com foco na camada de comunicação, onde uma RPC é usada para modelar a comunicação baseada em mensagens regida por Protocolos de Interação de agentes. A RPC permite a detecção de erros comuns de comunicação, tais como: mensagens perdidas, atraso, espera ocupada e término de agente. Para isso um protocolo de interação é especificado como uma marcação da rede de Petri onde as fichas são instâncias de mensagens e os lugares representam os agentes.

FAKHIR et al. (2023) utiliza RPC para verificação formal de sistemas adaptativos complexos. Para isso um sistema concorrente multi-agente auto-adaptativo é modelado em uma RPC para análise do espaço de estados e com isso verificar as propriedades comportamentais do modelo como a não existência de *deadlocks* ou marcações mortas. Também foi formalmente verificado que o trabalho proposto é adequado para provar a correção de sistemas com comportamento concorrente complexo.

Em **(ODEH; JAYOUSI; RATROT, 2023)** são usados conceitos de sistema multiagente para construir uma gestão competente de recursos urbanos, mais especificamente a entidade de energia de uma cidade. A rede de Petri é usada para modelar o sistema matematicamente. Ao mesmo tempo, utiliza-se o software *CPNTools* para simular o comportamento do sistema na vida real. O modelo matemático e os resultados da simulação provaram que o SMA de energia projetado alcançaria seus objetivos quando aplicado na vida real.

MENASSEL et al. (2024) apresenta uma nova metodologia de geração de casos de teste para sistemas multiagentes normativos (NorMAS). A metodologia é baseada em um perfil operacional normativo e um conjunto de sequências de teste normativas geradas a partir dos diagramas de sequência e atividade da linguagem *NorMAS-ML* (FREIRE et al., 2019). Utilizando um algoritmo proposto e um conjunto de regras normativas o resultado é usado para gerar casos de teste normativos. A metodologia proposta é demonstrada com um estudo de caso prático que leva em consideração as características únicas dos sistemas multiagentes normativos e distingue entre os níveis de agente e sistema.

Além dos trabalhos acima, foram encontradas duas revisões sistemáticas da literatura. A primeira de **BAKAR; SELAMAT (2018)** tem como tema a verificação de sistemas de agentes, com foco nas técnicas de verificação de propriedades de agentes e detecção de falhas durante as etapas do processo de desenvolvimento baseado em agentes. Neste trabalho foram investigadas técnicas de teste para sistemas de agentes em geral, mapeando as diferentes propriedades e falhas que as técnicas existentes podem detectar.

A segunda revisão possui um escopo mais específico, apresentada por **CLARK; WALKINSHAW; HIERONS (2021)**, onde o foco está na geração de casos de teste para modelos baseados em agentes e revisa os artefatos usados para formar casos de teste, mecanismos de geração, oráculos de teste, métricas de adequação de teste e níveis de abstração de teste alcançados.

Na Tabela 3 apresenta uma sumarização entre os trabalhos relacionados. Entre os itens comparativos consta um breve resumo do objetivo, se o trabalho utiliza Sistemas Multiagente, se utiliza ou não Redes de Petri, se realiza testes e, por fim, se foi implementado algum mecanismo para geração de casos de teste. Todos os trabalhos possuem algum tipo de relação aos SMA, e 8 desses trabalhos também utilizam redes de petri o que é justificado pelas capacidades das RP em especificar e testar SMA, qualidades que foram anteriormente mencionadas na introdução desse trabalho e justificam o uso de RPC.

De uma perspectiva mais ampla do desenvolvimento de SMA, considerando as três principais dimensões: o nível social, os aspectos de comunicação e o nível do agente GONÇALVES; QUADROS; SALDANHA (2016). Tem-se que alguns dos trabalhos listados realizam testes a nível de agente, analisando as interações entre os agentes e o ambiente, entre outros agentes ou seu funcionamento individual. Outros trabalhos focam na comunicação entre estes agentes, mas apenas (GONÇALVES et al., 2022), que constitui um trabalho anterior ao aqui apresentado, examina a camada social através testes a nível organizacional do SMA, levando em consideração um modelo organizacional do sistema e as restrições ao comportamento do agente que esse modelo pode proporcionar. Dessa forma, pode-se afirmar que este trabalho representa uma abordagem inovadora, pois é o único a testar modelos organizacionais.

Nessa seção foi apresentada uma síntese de alguns trabalhos publicados relacionados com a área de teste em SMA, que mais se relacionam a questão de pesquisa desta tese. Cada um desses trabalhos trouxe uma contribuição tanto para a geração de casos de testes, como para a criação de ferramentas de teste. Como é possível observar, existe um crescente aumento de publicações especialmente nos últimos quatro anos, o que demonstra um maior interesse da comunidade científica em testes de SMA.

Tabela 3: Sumarização dos trabalhos relacionados.

Trabalho	Objetivo	SMA	RP	Testes	Casos de Teste
WANG; ZHU (2012)	Automatizar testes em tempo de execução	✓	✗	✓	✗
EASSA et al. (2014)	Detectar erros de tempo de execução	✓	✗	✓	✗
KERRAOUI et al. (2016)	Realizar testes em SMA	✓	✓	✓	✓
FRASHERI et al. (2017)	Teste de agentes autônomos	✓	✓	✓	✓
WINIKOFF (2017)	Análise da testabilidade de SMA	✓	✗	✗	✓
REHMAN; NADEEM; SINDHU (2019)	Realizar testes em SMA	✓	✗	✓	✓
BENAC EARLE; FREDLUND (2019)	Realizar testes em SMA	✓	✗	✓	✓
NASCIMENTO et al. (2020)	Realizar testes em SMA	✓	✗	✓	✓
BOUCHERIT et al. (2020)	Análise formal para SMA	✓	✓	✓	✓
MIHRET; LIU (2021)	Geração de casos de testes para sistemas colaborativos	✓	✗	✓	✓
ENGELMANN et al. (2022)	Testes em Tempo de execução para SMA	✓	✗	✓	✓
GONÇALVES et al. (2022)	Realizar testes em SMA a nível organizacional	✓	✓	✓	✓
EL HOUDA; SOUFIENE; DJABER (2022)	Gerar casos de teste	✓	✗	✗	✓
HE; LIU; ZHOU (2022)	Verificação de Modelo de SMA	✓	✓	✓	✗
ALDAHASH; KING; BENABID (2023)	Testes na camada de comunicação de SMA	✓	✓	✓	✗
FAKHIR et al. (2023)	Verificação formal de SMA	✓	✓	✓	✗
ODEH; JAYOUSI; RATROT (2023)	Modelagem de SMA EM RPC	✓	✓	✗	✗
MENASSEL et al. (2024)	Geração de casos de teste para SMA normativos	✓	✗	✓	✓
BAKAR; SELAMAT (2018)	Revisão sistemática da literatura	✓	✗	✓	✓
CLARK; WALKINSHAW; HIERONS (2021)	Revisão sistemática da literatura	✓	✗	✓	✓

3 DESCRIÇÃO DO MÉTODO

3.1 Visão Geral

Nesta seção é apresentada uma descrição detalhada para o método de teste para SMA que utilizam o modelo organizacional $\mathcal{M}\text{oise}^+$. Esse método teve sua primeira versão apresentada em (MACHADO, 2020), e mais recentemente em (GONÇALVES et al., 2022), tendo como finalidade gerar os casos de teste necessários para a verificação de um sistema multiagente.

Este trabalho foi incremental. Assim, comparando a versões anteriores do método, a versão atual possui novas funcionalidades, como a inclusão no conteúdo da ficha mais uma informação, relacionada ao grupo ao qual o papel pertence. Mas a modificação mais significativa foi a inclusão da cardinalidade máxima e mínima, o que exigiu ferramentas para o controle do número de fichas, que agora representam o número de agentes para um determinado papel, resultando numa série de mudanças, tanto nas declarações como nas inscrições. Além disso, todas as formalizações anteriores foram refeitas e também foram incluídas novas definições formais, levando em conta os novos elementos incluídos no modelo.

Outra modificação foi a remoção de caminhos para um lugar final de falha. Na versão anterior do método para cada lugar da rede que representa uma meta global vinculada a uma missão, existia um caminho extra na rede que tinha como destino um lugar de falha, indicando assim que um problema na execução ocorreu. Percebeu-se que esses caminhos de falha geravam casos de teste equivocados, uma vez que levavam a um estado final que presumia uma falha antes mesmo do sistema ser testado. Nesta nova versão são gerados apenas casos de teste que levam a uma execução completa do sistema e a falha, caso exista, poderá então ser identificada ao executá-los.

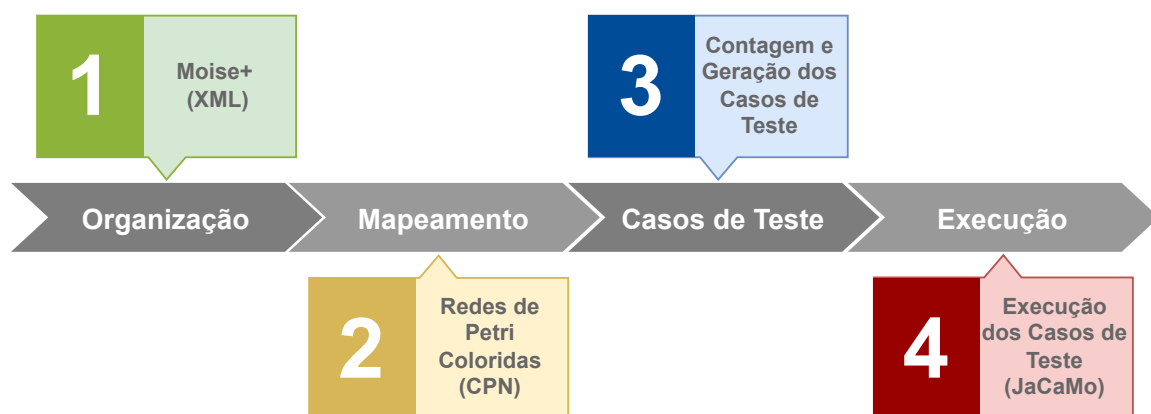


Figura 10: Sequência de Etapas do Método, Fonte: autor.

Na Figura 10 é apresentada uma visão geral das etapas de aplicação do método, onde: (i) um Modelo Organizacional, neste caso o arquivo XML do $\mathcal{M}\text{oise}^+$, é selecionado; (ii) em seguida, o arquivo XML é mapeado para uma RPC, gerando um arquivo de extensão CPN, utilizando

o método de mapeamento aqui descrito; em (iii) ocorre a contagem e geração de casos de teste para o mapeamento gerado; por fim, a última etapa (iv) consiste na execução de cada caso de teste para verificar o sistema, utilizando a plataforma de SMA JaCaMo.

Considerando agora a Figura 5, onde foram descritos diferentes processos necessários para o procedimento de teste, o método de teste proposto pode ser descrito da seguinte forma:

- **Sistema em teste:** é uma especificação $\mathcal{Moise}^+$ que descreve o comportamento social de um SMA;
- **Artefatos de informação:** a especificação $\mathcal{Moise}^+$ é composta por três artefatos de informação que devem ser integrados: a Especificação Funcional, a Especificação Estrutural e a Especificação Deôntica. A principal causa de falhas em uma especificação $\mathcal{Moise}^+$ é a ausência de uma integração formal entre essas especificações;
- **Mecanismo de geração:** Para o mecanismo de geração de casos de teste é proposto um modelo de RPC, a ser formalmente descrito nas linhas a seguir. O modelo proposto pode integrar as três especificações do modelo $\mathcal{Moise}^+$;
- **Suíte de teste:** O conjunto de casos de teste é aqui gerado utilizando um critério de cobertura denominado “*caminho de transição de estados*”, onde cada caso de teste gerado corresponde a uma sequência de execução dos estados contidos no grafo de alcançabilidade da RPC, conhecido como estado de espaço.
- **Oráculo de teste:** é a execução do modelo de RPC, cobrindo todos os caminhos do seu grafo de alcançabilidade, avaliando se nas possibilidades de execução é possível chegar ao local e estado final, que corresponde a meta raiz da especificação $\mathcal{Moise}^+$, sem utilizar um caminho de falha.

Antes de apresentar cada etapa do mapeamento, é apresentada uma descrição formal de cada elemento da RPC mapeada, que aqui foi denominada CPN_4M , no Axioma 1.

Em resumo, pode-se afirmar que os locais da rede, formados pelas elipses, representam as diferentes metas do sistema, e os tokens descrevem os papéis e seus grupos, onde o número de *tokens* representa o número de agentes para um papel específico. Todos os demais elementos, incluindo variáveis, constantes e funções, servem para garantir que os tokens corretos sejam encaminhados para seus devidos locais com a quantidade correta, obedecendo assim à cardinalidade mínima e máxima do sistema.

Axioma 1: (Rede de Petri Colorida para $\mathcal{Moise}^+$)

Uma Rede de Petri Colorida para o $\mathcal{Moise}^+$ (CPN_4M) é uma nove-tupla:

$$CPN_4M = (P, T, A, C, V, N, F, E, K)$$

onde:

1. P é um conjunto finito de lugares com dimensão n que representam as metas e suas respectivas missões em uma organização $\mathcal{Moise}^+$. É obrigatório que P inclua um local de início (representando o estado inicial), um local de término (representando o estado final).
2. T é um conjunto finito de transições de dimensão m estabelecendo a sequência de metas a partir de eventos a nível de agentes.
3. A é um conjunto de arcos, e $A \subseteq (P \times T) \cup (T \times P)$.
4. C é um conjunto de cores finito não vazio que representa os papéis e os grupos em uma organização $\mathcal{Moise}^+$.
5. V é um conjunto de variáveis tipadas tal que $Type[v] \in C$ para todas variáveis $v \in V$.
6. N é um conjunto de constantes usadas para especificar o número de agentes para cada papel e os valores máximo e mínimo das cardinalidades da missão em uma organização $\mathcal{Moise}^+$.
7. $F : T \rightarrow EXPR_V$ é uma função de guarda que atribui uma guarda a cada transição t tal que $Type[G(t)] = Bool$. É utilizado para verificar se o número de agentes atende à cardinalidade mínima da missão e também como ferramenta de monitoramento da Especificação Deontica.
8. $E : A \rightarrow EXPR_V$ é uma função de expressão de arco que associa uma expressão de arco a cada arco a tal que $Type[E(a)] = C(p)$, onde p é o lugar conectado ao arco a . Representa um verificador de cardinalidade máxima do $\mathcal{Moise}^+$, definindo o limite superior de agentes para uma determinada missão.
9. $K : P \rightarrow EXPR_\emptyset$ é uma função de inicialização que atribui uma expressão de inicialização um determinado lugar p tal que $Type[K(p)] = Bool(start)$, representando o estado inicial da rede sendo por padrão o lugar booleano $start$.

Pode-se utilizar agora o Axioma 1 como base para os procedimentos necessários para realizar todas as etapas apresentadas na Figura 10, que serão detalhadamente explicado nas subseções seguintes.

3.2 Mapeamento na Rede de Petri

A primeira etapa do método consiste no mapeamento do sistema em uma RPC, mais especificamente no formato $CPN4M$. Para que seja realizado esse mapeamento é necessário seguir as três etapas apresentadas na Figura 10: gerar as declarações, modelar a estrutura e incluir inscrições. A seguir, descreve-se cada uma dessas etapas e uma definição formal específica para cada elemento delas.

3.2.1 Gerar as Declarações

O primeiro passo é gerar as declarações. Essas declarações possuem um papel importante no funcionamento da RPC. Antes de iniciar, destaquemos dois tipos de declarações importantes: as variáveis que são declaradas com o comando *var*, e as constantes que é utilizado o comando *val*. Nas variáveis não é possível declarar com um valor pré-estabelecido, pois essa atribuição só pode ocorrer durante a execução do modelo (JENSEN; KRISTENSEN, 2009). Ao declarar variáveis

devemos apenas indicar que tipo de cor elas pertencem, que podem ser tipos já existentes como inteiros, reais, booleanos, strings ou um conjunto de cor criado pelo usuário como o que será aqui utilizado, o conjunto RG . Já as constantes, essas devem receber um valor ao serem declaradas, porém esse valor não pode ser alterado. A seguir na Lista 1 tem-se um exemplo de declarações feitas no mesmo formato padrão da ferramenta CPNTools, porém aqui numeradas, que será utilizado como base para nossas definições:

Lista 1: Exemplo de Declarações.

```

1. colset R = with roleA | roleB;
2. colset G = with groupA | groupB;
3. colset RG = product R*G;
4. var v1,v2 : RG;
5. val NroleA = 3;
6. val NroleB = 2;
7. val Cmin_m1 = 2;
8. val Cmax_m1 = 4;
9. val Cmin_m2 = 1;
10. val Cmax_m2 = 2;
11. fun Fcmax_m1(v1)= case v1 of (roleA ,groupA)=>
    (if NroleA <= Cmax_m1 then NroleA‘v1 else Cmax_m1‘v1);
12. fun Fcmax_m2(v2)= case v2 of (roleB ,groupB)=>
    (if NroleB <= Cmax_m2 then NroleB‘v2 else Cmax_m2‘v2);
13. fun Fcmin_m1()= NroleA>=Cmin_m1;
14. fun Fcmin_m2()= NroleB>=Cmin_m2;
15. fun Fdeon_m1()= v1=(roleA ,groupA);
16. fun Fdeon_m2()= v2=(roleB ,groupB);

```

A primeira declaração apresentada é o conjunto de cor (*colset*) R que representa os papéis existentes no $\mathcal{Moise}^+$. No exemplo estão os papéis *roleA* e *roleB*, que são separados por uma barra vertical. Para cada papel é declarada uma constante que representa o número de agentes para aquele papel, aqui $NroleA$ e $NroleB$, onde foram inseridos os valores 3 e 2 respectivamente mas poderiam receber qualquer número inteiro positivo. Portanto, para cada papel no $\mathcal{Moise}^+$ tem-se dois tipos de declarações fundamentais sendo elas o conjunto de cor R e as constantes $Nrole$. Assim a declaração dos elementos dos papéis satisfaz o Axioma 2.

Axioma 2: Declaração dos Elementos dos Papéis

- Para todo r , se $r \in R$ então existem: uma cor C ; uma constante N , todas associadas a r .

Onde: R é o conjunto dos papéis em um modelo organizacional.

Continuando as declarações, tem-se o conjunto de cores G que representa os grupos da especificação, onde para cada grupo do $\mathcal{Moise}^+$ vai existir uma cor diferente, aqui por exemplo foram declarados dois grupos: *groupA* e *groupB*. Já o conjunto de cores RG representa o produto cartesiano entre os dois conjuntos anteriores R e G e relaciona cada função com seu respectivo grupo, sendo essa uma declaração padrão que não depende da especificação.

Axioma 3: Declaração dos Elementos dos Grupos

- Para todo g , se $g \in G$ então existe: uma cor C associada a g .

Onde: G é o conjunto dos grupos em um modelo organizacional.

Algumas variáveis também precisam ser declaradas usando o comando *var*. Para cada missão na rede, é necessário criar uma nova variável. Neste exemplo, a variável $v1$ é usada para a missão $m1$ e a variável $v2$ para a missão $m2$, ambas do tipo RG porque as variáveis carregam os tokens, que neste caso são o produto entre o papel (R) e o grupo (G). Além das

variáveis, são declaradas constantes, representando a cardinalidade máxima e mínima de cada missão. $Cmin_m1$ é um exemplo desse tipo de constante que indica a cardinalidade mínima para a missão $m1$, já $Cmax_m1$ indica a cardinalidade máxima para $m1$. A cardinalidade mínima e máxima estão escritas no XML do $\mathcal{Moise}^+$ como valores inteiros positivos.

As últimas declarações são as funções para controle das fichas usadas nos arcos e nas guardas. Existem três tipos de funções, a primeira tem o nome de $Fcmax$ e é usada para gerar as fichas nos arcos, garantindo que não exceda a cardinalidade máxima. Assim, mesmo com mais agentes disponíveis, a missão usa apenas o número informado na cardinalidade máxima. Por outro lado, se o número de agentes for menor ou igual à cardinalidade máxima, o valor não muda. Essa função utiliza como entrada a variável v correspondente da sua missão, e esta variável contém a ficha que especifica um papel e o grupo que este papel pertence. A segunda função, $Fcmin$, garante que o número de fichas, representando aqui quantidade de agentes, atenda à cardinalidade mínima. Se uma missão tiver mais de um papel, todos devem ser descritos nesta função. A terceira função, $Fdeon$, é uma função de controle da especificação deôntica que faz a conexão entre os papéis e a missão garantindo que as fichas que a variável v utilizada na função $Fcmax$ possuam o papel e o grupo corretos, obedecendo a especificação deôntica do $\mathcal{Moise}^+$. Assim, para cada missão, há uma função de cada tipo atribuída a ela. Por exemplo, aqui tem-se a missão $m1$, então suas três funções são $Fcmax_m1$, $Fcmin_m1$ e $Fdeon_m1$.

No Axioma 4 estão incluídos os elementos relacionados a uma missão. Assim, para cada missão existente no $\mathcal{Moise}^+$ ter-se-á, obrigatoriamente, a declaração de uma variável, duas constantes sendo uma para a cardinalidade mínima e outra para a cardinalidade máxima, uma função deôntica, uma função de cardinalidade mínima e por último uma função de cardinalidade máxima.

Axioma 4: Declaração dos Elementos das Missões

- Para todo m , se $m \in M$ então existem: uma função $Fcmin$; uma função $Fcmax$; uma função $Fdeon$; duas constantes $Cmin$ e $Cmax$; e uma variável v , todos associados a m .

Onde: M é o conjunto das missões em um modelo Organizacional; $Fcmin$ é uma função de guarda de cardinalidade mínima na RPC; $Fdeon$ é uma função de guarda deôntica na RPC; $Fcmax$ é uma função de arco de cardinalidade máxima na RPC; $Ncmin$ e $Ncmax$ são constantes na RPC; v é uma variável na RPC;

3.2.2 Modelando a Estrutura

Na modelagem da estrutura, as metas do $\mathcal{Moise}^+$ são representados pelos lugares da RPC (elipses), e os diferentes tipos de operadores de plano do $\mathcal{Moise}^+$ (sequência, escolha e paralelismo) são transformados em estruturas de rede através do modelo de mapeamento apresentado na Figura 11, que define a forma como os lugares são distribuídos na rede.

Além desses elementos, um lugar inicial chamado *start* e um lugar final chamado *end* são incluídos na rede, indicando onde o sistema começa e termina. Na Figura 12 é apresentado um exemplo de uma estrutura que representa uma sequência entre dois objetivos, *goal1* e *goal2*, e cada um desses objetivos pertence a uma missão, com *goal1* pertencendo a $m1$ e *goal2* a $m2$.

A estrutura sequencial é basicamente uma conexão ordenada entre n lugares, onde a cada dois lugares existem uma transição e dois arcos usados para conectá-los. Além de ser o tipo mais simples de estrutura da rede, também é o mais comumente utilizado.

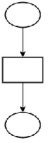
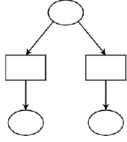
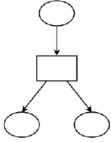
	Operador Moise+	Estrutura do CPN4M
Sequência		
Escolha		
Paralelismo		

Figura 11: Operadores para mapeamento da estrutura do *CPN4M*.

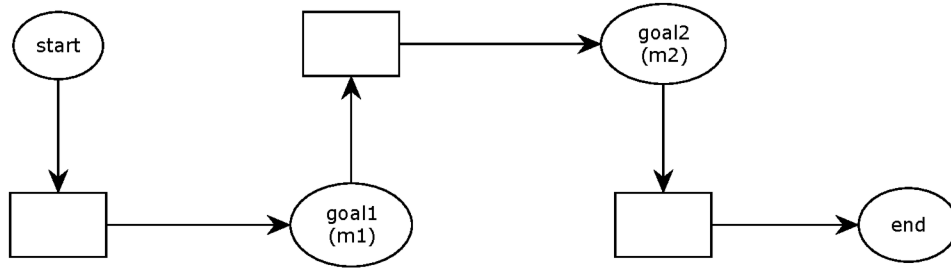


Figura 12: Estrutura de um sistema genérico mapeado para RPC.

Axioma 5: Estrutura Sequência

- Para todo operador **sequência** para n metas existe uma estrutura composta por:

$$p \Rightarrow a \Rightarrow t \Rightarrow a \Rightarrow \dots p_n$$

Onde: n é um número inteiro positivo; p é um lugar na RPC; a é um arco na RPC; t é uma transição na RPC; $\Rightarrow$ é uma ligação direcionada entre dois elementos.

A estrutura de escolha é a representação de n caminhos que partem de um único lugar, onde cada caminho tem seu próprio lugar de transição e arcos. A possibilidade de diferentes transições a partir do mesmo lugar significa que apenas um caminho pode ser utilizado.

Axioma 6: Estrutura Escolha

- Para todo operador **escolha** para n metas existe uma estrutura composta por:

$$p \Rightarrow n(a \Rightarrow t \Rightarrow a \Rightarrow p)$$

Onde: n é um número inteiro positivo; p é um lugar na RPC; a é um arco na RPC; t é uma transição na RPC; $\Rightarrow$ é uma ligação direcionada entre dois elementos.

A estrutura de paralelismo possui diferentes caminhos que partem da mesma transição, com diferentes arcos e lugares, tornando possível que a rede siga todos os caminhos ao mesmo tempo.

Axioma 7: Estrutura Paralela

- Para todo operador **paralelo** para n metas existe uma estrutura composta por:

$$p \Rightarrow a \Rightarrow t \Rightarrow n(a \Rightarrow p)$$

Onde: n é um número inteiro positivo; p é um lugar na RPC; a é um arco na RPC; t é uma transição na RPC; $\Rightarrow$ é uma ligação direcionada entre dois elementos.

3.2.3 Incluindo as Inscrições

Uma vez que a estrutura da RPC está finalizada, deve-se incluir as inscrições nos elementos estruturais da rede. Nesta etapa, são incluídos os nomes dos lugares, suas marcações iniciais e seus tipos, bem como os nomes das transições, guardas e funções de arco.

A Figura 13 mostra o mesmo exemplo anterior com todas as inscrições incluídas. No lugar *start*, o tipo de cor é **BOOL** com o token inicial sendo *true*. No arco que conecta *start* à primeira transição *t1*, a expressão incluída por padrão é um token *true*. Uma observação é que tanto lugar inicial *start* como o lugar final *end* não fazem parte da especificação do $\mathcal{Moise}^+$, mas definem de maneira clara a entrada e a saída do sistema. Esses lugares são também fundamentais para que existam duas transições importantes, que são a primeira e última transição (neste exemplo *t1* e *t3*) e essas transições e seus respectivos arcos recebem funções operacionais importantes que serão vistas a seguir.

Ainda na Figura 13, na transição *t1* há algumas chamadas para funções de guarda, incluindo a função deôntica $Fdeon_m1(v1)$, uma função que garante que a variável *v1* contém o token com as informações corretas de papel e grupo para a missão *m1*. Esta função *Fdeon* deve ser incluída tanto na guarda da transição de entrada, aqui *t1*, quanto na transição de saída *t2*. Nessa primeira transição *t1* também são incluídas as chamadas para a função de guarda da cardinalidade mínima de todas as missões (aqui $Fcmin_m1$ e $Fcmin_m2$).

Axioma 8: Inscrição de Chamada de Função de Cardinalidade Mínima

- Para todo m , se $m \in M$ então existe: uma inscrição da chamada da função $Fcmin$ de m na guarda da transição t_1 .

Onde: M é o conjunto das missões em um modelo organizacional; $Fcmin$ é uma função de guarda de cardinalidade mínima na RPC; t_1 é a primeira transição da RPC e $t_1 \in T$ o conjunto de transições.

Axioma 9: Inscrição de Chamada de Função Deôntica

- Para todo t , se t está conectado ao lugar p e p possui uma missão m inscrita, então existe: uma inscrição da chamada da função $Fdeon$ de m na guarda da transição t .

Onde: $m \in M$ e M é o conjunto das missões em um modelo organizacional; $t \in T$ e T é o conjunto de transições na RPC; $p \in P$ e P é o conjunto dos lugares na RPC; $Fdeon$ é uma função de guarda deôntica na RPC.

Chamadas de funções são usadas também nos arcos de entrada e saída dos lugares com o tipo *RG*. Por exemplo, entre a transição *t1*, o lugar *goal1* e a transição *t2*, temos nos arcos a inscrição $Fcmax_m1(v1)$, que é a chamada para uma função que indica o número de agentes atribuídos à missão *m1* usando a variável *v1*. Por sua vez, *goal2* já pertence a *m2*, e, portanto, a função $Fcmax_m2(v2)$ é usada nos arcos.

Axioma 10: Inscrição de Chamada de Função de Cardinalidade Máxima

- Para todo a , se a conecta t em p ou p em t e p possui uma missão m inscrita, então existe: uma inscrição da chamada da função $Fcmax$ de m em a .

onde: $a \in A$ e A é o conjunto de arcos na RPC; $t \in T$ e T é o conjunto de transições da RPC; $m \in M$ e M é o conjunto de missões em um modelo organizacional; $p \in P$ e P é o conjunto de lugares na RPC; $Fcmax$ é uma Função de arco de cardinalidade máxima da RPC.

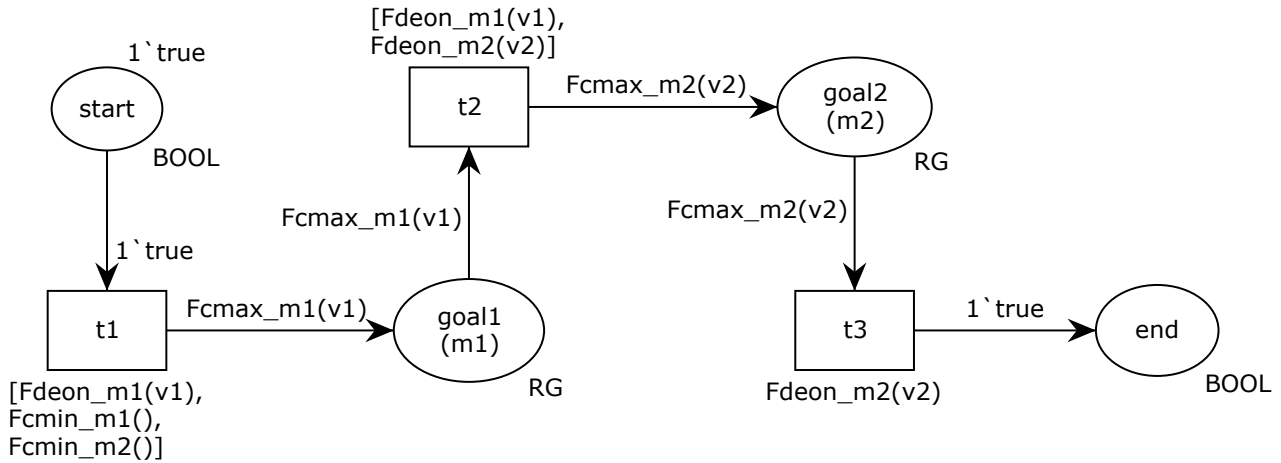


Figura 13: Inclusão de Inscrições na RPC.

Se uma meta não estiver relacionado a uma missão, o lugar na rede que a representa será do tipo booleano *bool*. Outros lugares que recebem esse tipo de cor são *start* e *end*, e nesses lugares, tanto os arcos de entrada quanto os de saída carregam um token booleano *true*.

Axioma 11: Inscrição de Arco Booleana

- Para todo a , se a conecta t em p ou p em t e p possui uma inscrição *Bool*, então existe: uma inscrição booleana *true* em a .

onde: $a \in A$ e A é o conjunto de arcos na RPC; $t \in T$ e T é o conjunto de transições da RPC; $p \in P$ e P é o conjunto de lugares na RPC; *true* é um valor booleano para o tipo de cor *Bool* da RPC.

3.3 Contagem dos Casos de Teste

Nessa etapa, com o sistema já mapeado para o *CPN4M*, é realizada a avaliação da testabilidade do modelo. WINIKOFF (2017) definiu testabilidade de um programa como sendo uma métrica que indica o esforço necessário para testar adequadamente esse programa. Por exemplo, para um programa simples como “**if** x **then** $t1$ **else** $t2$ ”, são necessários dois testes para cobrir todos os caminhos de um grafo de controle de fluxo correspondente a esse programa.

Para realizar a contagem dos casos de teste é necessário escolher um critério de adequação de teste. Como visto na seção 2.3, existem diferentes critérios de adequação que podem ser utilizados. Esses critérios definem a forma como os casos de teste serão extraídos do sistema a ser testado. Por exemplo, em (WINIKOFF, 2017) os critérios utilizados para avaliação foram o de **todas as arestas** (*all edges*) e **todos os caminhos** (*all paths*).

Neste trabalho, será utilizado um critério chamado **caminho de transição dos estados** (*state transition path*), que pode ser visto em (ZHU; HE, 2002), onde foi definido também com o critério **todos os estados** (*all states*). A vantagem do critério escolhido é a possibilidade de contabilizar não apenas todos os caminhos da rede mas também os diferentes estados que a RPC pode alcançar. Mais especificamente, pode-se ter em um lugar da rede diferentes fichas que são atribuídas na especificação *CPN4M* a diferentes papéis que uma determinada meta pode possuir, e essas distintas possibilidades não seriam contabilizadas utilizando o critério **todos os caminhos**.

Para utilizar o critério de adequação **caminho de transição de estados**, diferentes marcações da RPC são transformadas em um grafo direcionado conhecido como espaço de estados. Os nós neste grafo indicam marcadores da rede, enquanto os arcos representam os elementos de conexão entre eles. O critério *caminho de transição de estados* considera todos os caminhos de execução possíveis para um grafo de espaço de estados em uma RPC, portanto, deve-se identificar cada um desses caminhos.

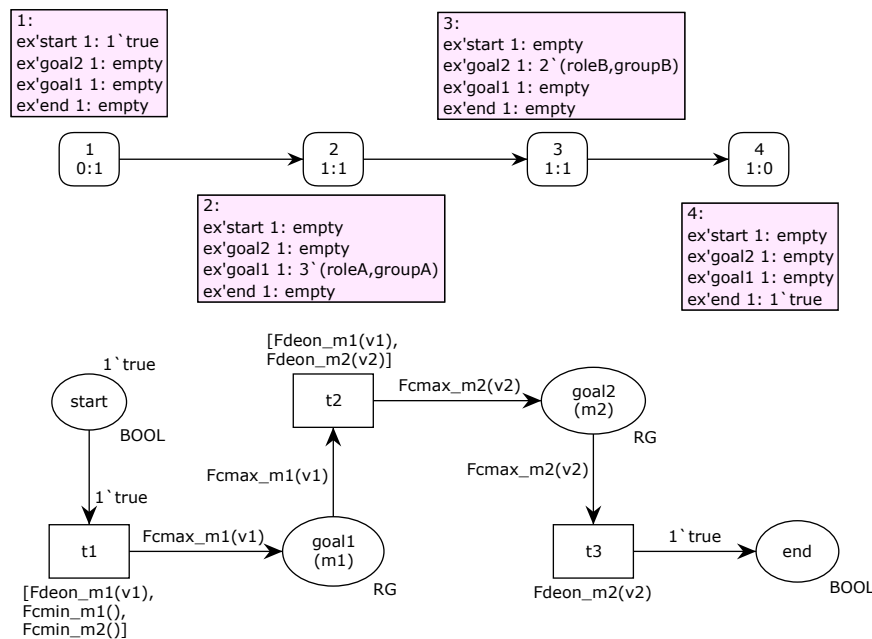


Figura 14: Exemplo de Grafo de Estado de Espaço do *CPN4M*, Fonte: Autor.

Como resultado, é construído um grafo de espaço de estado correspondente ao método *CPN4M* para contar os casos de teste. A contagem de caminhos do nó inicial para todos os nós finais possíveis deste grafo deve então ser realizada. Como cada caminho de teste do espaço de estados corresponde a um caso de teste, o número de caminhos encontrados é igual ao número de casos de teste.

Na Figura 14 pode-se ver o mesmo exemplo da Figura 13 mas agora com o grafo de espaço de estado gerado a partir da RPC. Neste grafo estão os quatro estados para esse exemplo, sendo que é possível ver também uma descrição de cada um desses estados. Por exemplo, no estado numero 1 na descrição tem-se o lugar *start*, com uma ficha *true*. No estado 2 tem-se o lugar *goal1* com 3 fichas (*roleA,groupA*) e no estado 3 o lugar *goal2* com 2 fichas (*roleB,groupB*) e o estado 4 com a ficha *true* no lugar *end*. Um detalhe: em cada estado, além do número do estado existe uma numeração que indica o número de arcos de entrada e saída. Por exemplo, para o estado 1 temos 0 : 1 sendo 0 arcos de entrada e 1 arco de saída.

Com o grafo de espaço de estado pronto, para contar os casos de teste basta realizar todos os possíveis caminhos que levam do primeiro ao último estado. Nesse exemplo tem-se um

único caminho que gera um caso de teste. Porém, também é necessário incluir na contagem as diferentes possibilidades de entradas do número de agente para cada papel. Assim, o resultado final será o número de caminhos do grafo de espaço de estado, multiplicado pelo número de possibilidades de entrada de cada papel, obedecendo a cardinalidade máxima e mínima da missão.

Para realizar este calculo de maneira objetiva basta calcular a quantidade de números entre a cardinalidade máxima e mínima para cada missão. Por exemplo, tem-se que o valor de C_{max_m1} é 4 e C_{min_m1} é 2, então é possível para $m1$ ter 2, 3 ou 4 agentes o que gera uma quantidade de possíveis entradas de 3 elementos, que deve ser multiplicado pelo total de caminhos de teste anteriormente encontrado. No caso 1, ficando o total de 3 casos de teste. O mesmo calculo deve ser feito para todas as missões, então calculando a quantidade da missão $m2$ encontra-se o valor 2, agora multiplica-se ele pelo 3 que é o valor atual, ficando um total de 6 casos de teste.

3.4 Geração dos Casos de Teste

Com os casos de teste necessários para testar o modelo proposto devidamente identificados e quantificados, agora o passo é especificar o cenário de teste que compõe cada caso. Assim, o objetivo dessa seção é propor um método para especificar detalhadamente cada um desses casos de teste. Mas antes de descrever o método, é necessário definir o que será testado e a maneira como serão realizados esses testes.

O que será testado: O teste do sistema será realizado para cada diferente caminho de teste que juntos formam um conjunto de teste. Cada caminho é uma sequência de execução de metas e são essas metas, ou seja, as ações necessárias para executá-las que serão testadas.

Como será testado: No ambiente de teste serão realizadas simulações onde um agente com o papel x , estando comprometido com a meta y deve executar ações que satisfaçam o plano necessário para cumprir a meta y , seguindo a sequência do caminho de metas até o final ou até apresentar algum erro.

Os casos de teste contabilizados devem ser descritos individualmente para serem utilizados como guia na execução de testes do sistema. Cada caminho de teste devidamente identificado pelo método já apresentado, será um caso de teste. A descrição do caso de teste é feita através de uma tabela contendo o número do nó, a meta, o papel e uma breve descrição dessa meta. Com essa tabela, o caso de teste estará montado e pronto para ser utilizado como guia na execução do teste no sistema. Para exemplificar, a Tabela 4 apresenta um caso de teste gerado para o grafo da Figura 14.

Tabela 4: Exemplo de caso de teste caminho 1.

Nó	Meta	Agentes	Papel	Descrição
1	<i>start</i>	0	<i>true</i>	Teste em espera para iniciar
2	<i>goal1</i>	3	<i>roleA</i>	O papel <i>roleA</i> realiza a tarefa <i>X</i>
3	<i>goal2</i>	2	<i>roleB</i>	<i>roleB</i> realiza a tarefa <i>Y</i>
4	<i>end</i>	0	<i>true</i>	O teste encerra.

3.5 Execução dos casos de teste

Com todos os casos de teste gerados, já é possível executar testes. Para isto será preciso agora configurar para que os papéis, com a quantidade de agentes corretos, realizem as missões

definidas para eles. Além disso, para que os caminhos de execução obedecem a ordem definida no caso de teste, também é necessário que os agentes façam as escolhas corretas quando existirem tornando necessário configurar no código dos agentes de teste os parâmetros para que as metas corretas sejam escolhidas.

Ao realizar testes a nível organizacional é possível encontrar certos tipos de falhas que são geradas, especialmente por erros na especificação do modelo. Como exemplo de erros, pode-se ter um papel erroneamente incluído no grupo errado, ou um operador de plano trocado, o que dependendo do modelo, pode gerar uma dependência equivocada na execução de metas.

Outro erro possível é a atribuição errada de uma meta para uma missão, fazendo com que o papel errado seja responsável por ela e como consequência isto gera uma falha de execução. Isto também pode ocorrer na Especificação Deontica, caso o papel seja relacionado para a missão errada, por exemplo.

Esses são alguns exemplos de possíveis erros que podem ser encontrados no modelo organizacional do $\mathcal{Moise}^+$, porém ao executar os testes aqui propostos, além desses erros também é possível identificar falhas no código do agente, realizando testes em tempo de execução. Nesta seção não será apresentada uma execução de caso de teste, deixando esta tarefa para o próximo capítulo, que apresentará o caso de uso para o modelo.

3.6 Uma Ferramenta de Automatização do Método

O método descrito neste trabalho foi implementado em uma ferramenta que automatiza o mapeamento de um modelo $\mathcal{Moise}^+$ em uma RPC (MACHADO et al., 2024), denominada MAMTCPN³. Essa automatização foi elaborada em parceria com o aluno de graduação Arthur Zelindro, sendo desenvolvida utilizando a linguagem de programação Python. Entre as vantagens de automatizar o processo pode-se citar uma redução significativa do tempo em comparação com a elaboração manual, além de eliminar a probabilidade de uma falha de modelagem por erro humano.

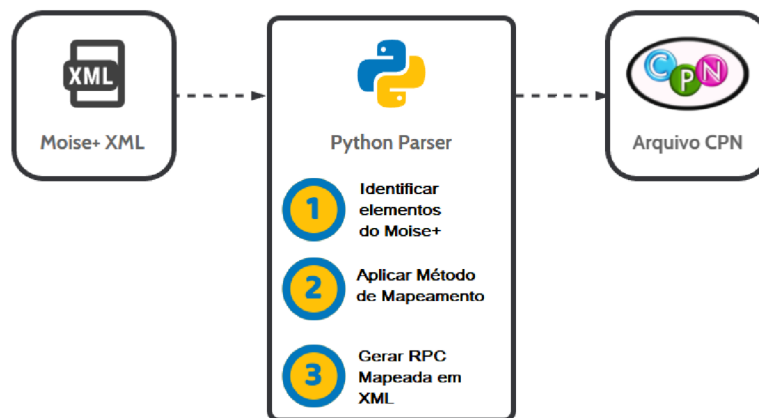


Figura 15: Arquitetura da Ferramenta de Automatização do Mapeamento, (MACHADO et al., 2024).

Na Figura 15 pode-se ver uma ilustração de como esta ferramenta opera. Primeiramente, utilizando duas bibliotecas nativas do Python chamadas MiniDom e ElementTree, essa ferramenta faz a leitura do arquivo XML do $\mathcal{Moise}^+$. Depois identifica os elementos do $\mathcal{Moise}^+$ necessários para o mapeamento, aplica o mapeamento utilizando o método aqui descrito e gera

³<https://github.com/zelindro/MAMTCPN>

os elementos na RPC que também em XML. Por fim, um arquivo no formato CPN, padrão da ferramenta CPNTools é gerado e pode ser utilizado para leitura da rede.

Além das vantagens já descritas aqui, essa ferramenta ajuda a validar o método de mapeamento descrito neste trabalho, mostrando que os modelos formais aqui declarados possuem uma lógica operacional padronizada que pode ser utilizada na elaboração de futuras ferramentas de teste.

4 CASOS DE USO

Neste capítulo é demonstrada a utilização do método através dois cenários para apresentar seu funcionamento na prática. Aqui, além das etapas de mapeamento também serão gerados os casos de teste e executados testes no sistema para validar a usabilidade do modelo. Os exemplos foram escolhidos de modo a apresentar as possibilidades de combinações de elementos funcionais de um modelo organizacional *Moise*⁺. Todos os exemplos aqui apresentados e seus respectivos arquivos CPN (que também são um XML) gerados, estão alocados em um repositório⁴.

4.1 Cenário do Inspector

O exemplo de cenário que será utilizado para demonstrar o método de mapeamento se chama *Inspector* e faz parte do projeto GORIM, uma modelagem a nível organizacional de agentes para estudo da complexidade e funcionalidade dos personagens em um *Role-Playing Game* (RPG), implementado em (BORN, 2022). O exemplo do *Inspector* tem como objetivo específico descrever um ambiente onde existe um inspetor responsável por fiscalizar determinados produtores. No modelo aqui apresentado foram feitas modificações comparado ao modelo original, entre elas a divisão do papel *Farmer* em três sub papéis, dando uma maior complexidade ao modelo.

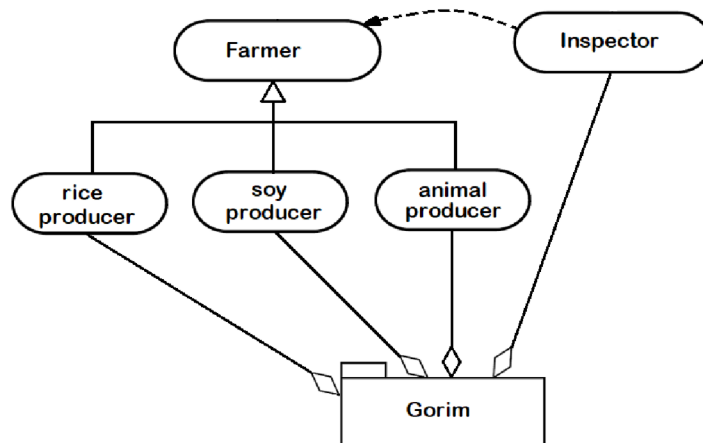


Figura 16: Especificação Estrutural do Cenário Inspector (adaptado de BORN, 2022).

Na Figura 16 tem-se a EE do modelo, onde como papéis principais tem-se *Farmer* e *Inspector*, e os subpapéis *ricep* (rice producer), *soyp* (soy producer) e *animalp* (animal producer). Todas esses papéis pertencem ao mesmo grupo chamado *gorim*.

Na EF, existem 24 metas, que podem ser vistas na Figura 17, onde 20 metas pertencem à missão *m3* e 4 metas à *m4*. Por fim, tem-se a ED na Tabela 5, onde o papel *inspector* possui uma obrigação com a missão *m3* e os papéis *soyp*, *ricep* e *animalp* com *m4*.

4.1.1 Mapeamento do Cenário Inspector

- Gerando as Declarações do cenário Inspector

Como primeira etapa do mapeamento apresenta-se as declarações. Utilizando inicialmente a Definição 2, são declarados os papéis *inspector*, *soyp*, *ricep* e *animalp*, pertencendo ao conjunto de cor *R* na linha 1. As constantes de cada papel *Ninspector*, *Nsoyp*, *Nricep*,

⁴<https://github.com/ricardoarend/ModelagemComp>

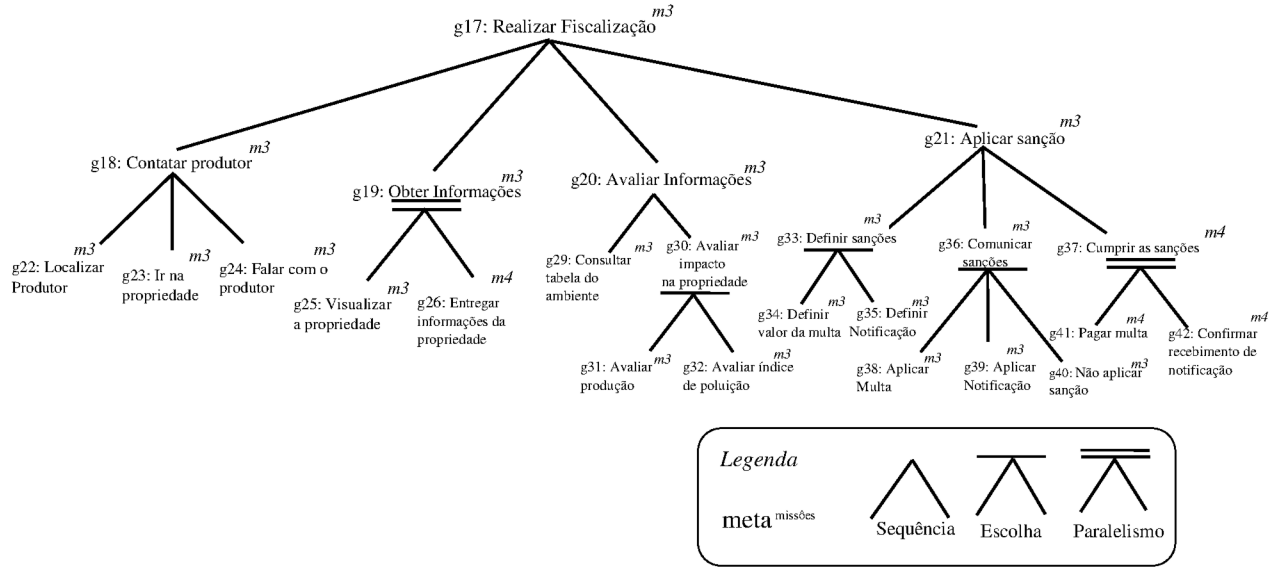


Figura 17: Especificação Funcional do Cenário Inspeção (adaptado de BORN, 2022).

Tabela 5: Especificação Deontica do Cenário Inspeção.

papel	relação deontica	missão
<i>inspector</i>	obrigação	m3
<i>soyp</i>	obrigação	m4
<i>ricep</i>	obrigação	m4
<i>animalp</i>	obrigação	m4

Nanimalp e seus respectivos valores que correspondem ao número de agentes disponível estão nas linhas 5, 6, 7 e 8. A seguir, com base na Definição 3, o grupo *gorim*, aos quais os papéis fazem parte, foi declarado no conjunto de cor G , na linha 2. Também foi feita a declaração padrão do conjunto RG na linha 3.

Após, parte-se para a Definição 4, onde são feitas as declarações com base nas missões do modelo $m3$ e $m4$. As primeiras são duas variáveis $v1$ e $v2$, uma para cada missão na linha 4. Uma constante de cardinalidade mínima e máxima, $Cmin$ e $Cmax$ para cada missão nas linhas 9, 10, 11 e 12. Já nas linhas 13 e 14 tem-se as funções de cardinalidade máxima, na 15 e 16 as de cardinalidade mínima, e nas linhas 17 e 18 as funções deonticas.

Com todas estas definições preenchidas, agora deve-se analisar um pouco a função de cardinalidade máxima da linha 14, feita para a missão $m4$. Aqui tem-se, de acordo com a especificação deontica, três possibilidades de papéis: *soyp*, *ricep* e *animalp*, para uma mesma missão, o que exigiu que nessas funções fosse colocado, por padrão, um *case* para cada papel. Assim, a rede vai contabilizar todos os possíveis papéis na hora de gerar os espaços de estados. Já na função deontica da missão $m4$ (linha 18), tem-se também a inclusão dos três papéis, mas agora utilizando um *else* entre eles, uma vez que essa função tem como objetivo filtrar as possíveis entradas geradas.

Lista 2: Declarações do cenário Inspeção.

1. `colset R = with inspector | soyp | ricep | animalp;`
2. `colset G = with gorim;`
3. `colset RG = product R*G;`

```

4. var v1,v2 : RG;
5. val Ninspector = 1;
6. val Nsoyp = 6;
7. val Nricep = 4;
8. val Nanimalp = 2;
9. val Cmin_m3 = 1;
10. val Cmax_m3 = 1;
11. val Cmin_m4 = 1;
12. val Cmax_m4 = 4;
13. fun Fmax_m3(v1)= case v1 of (inspector,gorim)=>
    (if Ninspector <= Cmax_m3 then Ninspector'v1 else Cmax_m3'v1);
14. fun Fmax_m4(v2) = case v2 of
    (soyp,gorim)=>(if Nsoyp <= Cmax_m4 then Nsoyp'v2 else Cmax_m4'v2)|
    (ricep,gorim)=>(if Nricep <= Cmax_m4 then Nricep'v2 else Cmax_m4'v2)|
    (animalp,gorim)=>(if Nanimalp <= Cmax_m4 then Nanimalp'v2 else Cmax_m4'v2);
15. fun Fmin_m3()= Ninspector >= Cmin_m3;
16. fun Fmin_m4()= Nsoyp >= Cmin_m4 andalso Nricep>=Cmin_m4 andalso Nanimalp>=Cmin_m4;
17. fun Fdeon_m3(v1)=v1=(inspector,gorim);
18. fun Fdeon_m4(v2)=v2=(soyp,gorim) orelse v2=(ricep,gorim) orelse v2=(animalp,gorim);

```

- Gerando a Estrutura

Nessa etapa será gerada a estrutura da RPC para o mapeamento do cenário *Inspector*. Aqui, utilizando como base as Definições 5, 6 e 7, que correspondem as estruturas sequência, escolha e paralelismo, respectivamente, foi gerada a estrutura da rede. Também está incluída na estrutura os lugares *start* e *end*.

A estrutura será aqui apresentada em duas etapas e é feita com base na EF do modelo que está na Figura 17, sendo que a primeira etapa será até a meta *g20*. Uma observação, na estrutura aqui apresentada tem-se algumas inscrições que estão foram incluídas para melhor localização dos elementos na rede, mas a inclusão delas não faz parte dessa etapa. Inicialmente tem-se uma sequência de três metas, *g22*, *g23* e *g24* que seguem para a meta *g18*, assim após o lugar inicial *start* colocamos essas quatro metas em sequência como visto na Figura 18. Após *g18* existe uma sequência para *g19* e *g20*, porém *g19* ramifica para um operador de plano paralelismo entre *g25* e *g26* que deve ser incluído antes. Terminado o paralelismo tem-se *g19* e depois para as ramificações de *g20* sendo a primeira a meta *g29*. Após *g29* tem-se a primeira escolha do modelo que é entre *g31* e *g32*, a escolha termina e são alcançadas *g30* e *g20* em sequência, encerrando a primeira etapa.

Agora na segunda etapa será incluído o restante da estrutura da rede. Continuando na Figura 17 agora partindo de *g20* tem-se uma sequência para *g21*, que se ramifica em três metas, sendo a primeira delas *g33*. Para chegar a *g33* primeiro tem-se uma escolha entre duas metas *g34* e *g35*, que são incluídos após *g20*. Terminada a escolha segue-se para *g33* e tem-se novamente outra escolha agora entre três metas: *g38*, *g39* e *g40*. Após esta escolha tem-se a sequência para *g36* e depois um paralelismo entre *g41* e *g42* e nova sequência para *g37*. Agora após *g37* restam apenas duas metas, a primeira *g21* e a segunda *g17* ambas em sequência, por fim se inclui o lugar final *end* e a estrutura esta pronta e pode-se ver ela na Figura 19.

- Incluindo as Inscrições

Com as declarações e a estrutura pronta, já é possível incluir as inscrições na rede. Primeiramente, deve-se incluir as inscrições que identificam os lugares, as transições e os tipos de cor de cada lugar. Com base na Definição 1, tem-se que os lugares da rede representam as metas e missões do *Moise*⁺, na Figura 19 já foram incluídas essas informações, juntamente com o lugar inicial e final e também foram incluídos os tipos de cor para cada lugar, sendo *RG* para os lugares com missões e *BOOL* para os demais.

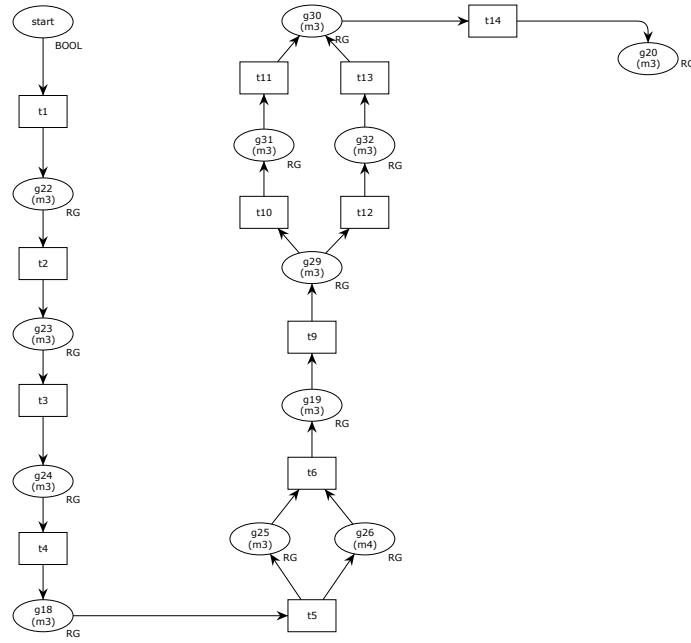


Figura 18: Primeira parte da estrutura do cenário Inspector.

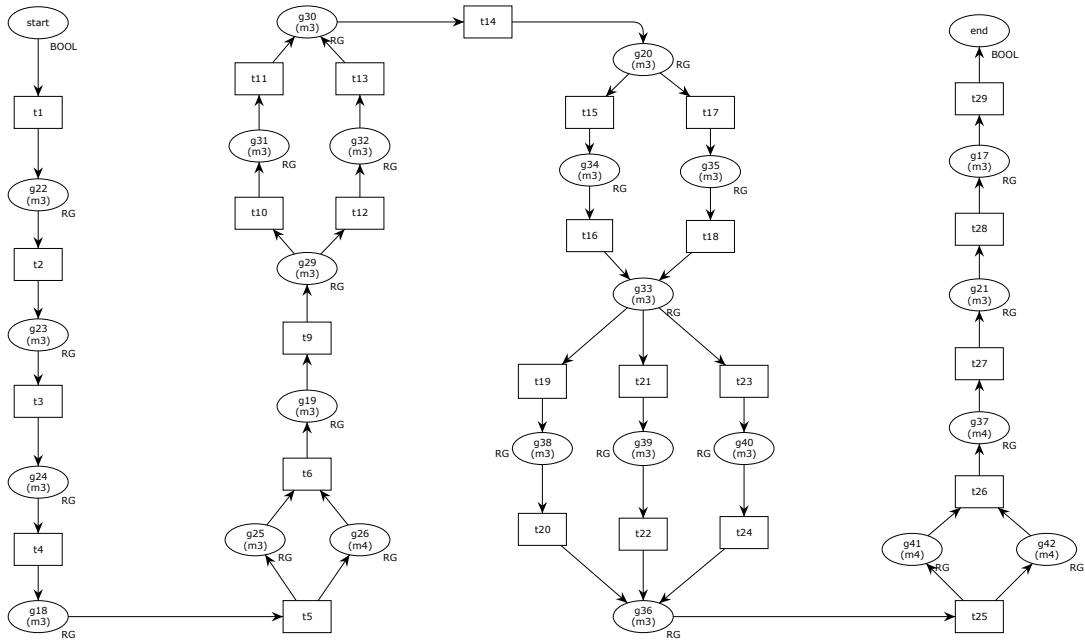


Figura 19: Estrutura completa do cenário Inspector.

Agora que todos os lugares estão identificados com os nomes das metas e suas respectivas missões, e as transições estão nomeadas em ordem crescente, são incluídas as funções que são utilizadas nas guardas das transições da rede. Essas funções correspondem a Definição 8, que diz que para cada missão do $\mathcal{M}oise^+$ deve ser inscrita uma chamada de função de cardinalidade mínima na primeira transição da rede. Na Figura 20 pode-se ver essas inscrições, onde $t1$ recebeu as chamadas das funções $Fmin_m3$ e $Fmin_m4$.

Ainda com relação as chamadas das funções de guarda, tem-se na Definição 9 a inclusão das funções deonticas nas transições. Deve-se analisar para cada transição os lugares anteriores e posteriores: mais especificamente, se nesse lugar existe uma missão inscrita, deve-se ter uma inscrição da chamada da função deontica dessa missão na respectiva

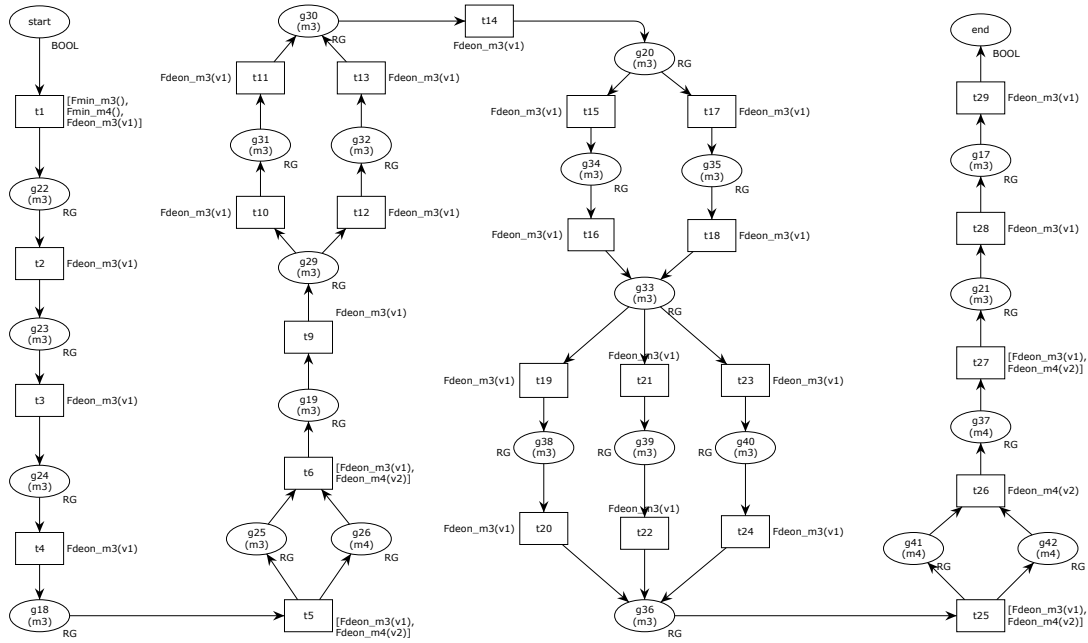


Figura 20: Inscrições de guarda incluídas no cenário Inspector.

transição.

Analisando a Figura 20, pode-se conferir como foram realizadas essas inscrições, na primeira transição $t1$ o lugar anterior a ela é o $start$ que não possui nenhuma missão. Já o lugar posterior a $t1$ é o $g22$ que possui a missão $m3$. Portanto, em $t1$ foi incluída a chamada de função $Fdeon_m3(v1)$. Isso é feito sucessivamente para as restantes transições até a transição final em $t29$, que também recebe a função $Fdeon_m3(v1)$, já que o lugar anterior $g17$ pertence a missão $m3$. Caso a transição precise chamar mais de uma função, elas precisam ser incluídas em forma de lista, utilizando colchetes no início e fim da inscrição e vírgulas separando cada função, na transição $t1$ tem-se um exemplo dessa lista.

As inscrições estão quase todas preenchidas, falta apenas incluir as inscrições nos arcos. Para isso, primeiro deve-se incluir as chamadas das funções de cardinalidade máxima de acordo com a Definição 10. Deve-se analisar cada arco da rede e como os arcos da RPC são utilizados sempre como ligação entre um lugar e uma transição, se o lugar ligado a esse arco possui uma missão, esse arco recebe a chamada da função de cardinalidade máxima referente à esta missão. Por exemplo no arco entre $t1$ e $g22$ temos a chamada da função $Fmax_m3(v1)$. Caso o lugar que está ligado ao arco não possua nenhuma missão, então automaticamente ele é do tipo *Bool* e vai entrar na regra descrita na Definição 11, onde receberá a inscrição *true*.

Na Figura 21 estão incluídas todas as inscrições nos arcos, e é possível observar que no primeiro e no último arcos da rede estão a ficha *true*, uma vez que esses arcos estão ligados a lugares Booleanos. Todos os outros arcos da rede possuem uma chamada de função. Outra observação é que deve ser incluída apenas uma ficha inicial na rede, que é a ficha *true* no lugar $start$ sendo isso por padrão nesse modelo de mapeamento.

Com as inscrições incluídas, é possível executar simulações na rede. A Figura 22 apresenta três capturas de tela de uma mesma parte da rede sendo executada em diferentes simulações. Observa-se que em cada simulação a rede carregou fichas diferentes para o lugar $g26$, obedecendo a especificação deôntica de $m4$ que previa essa possibilidade. A

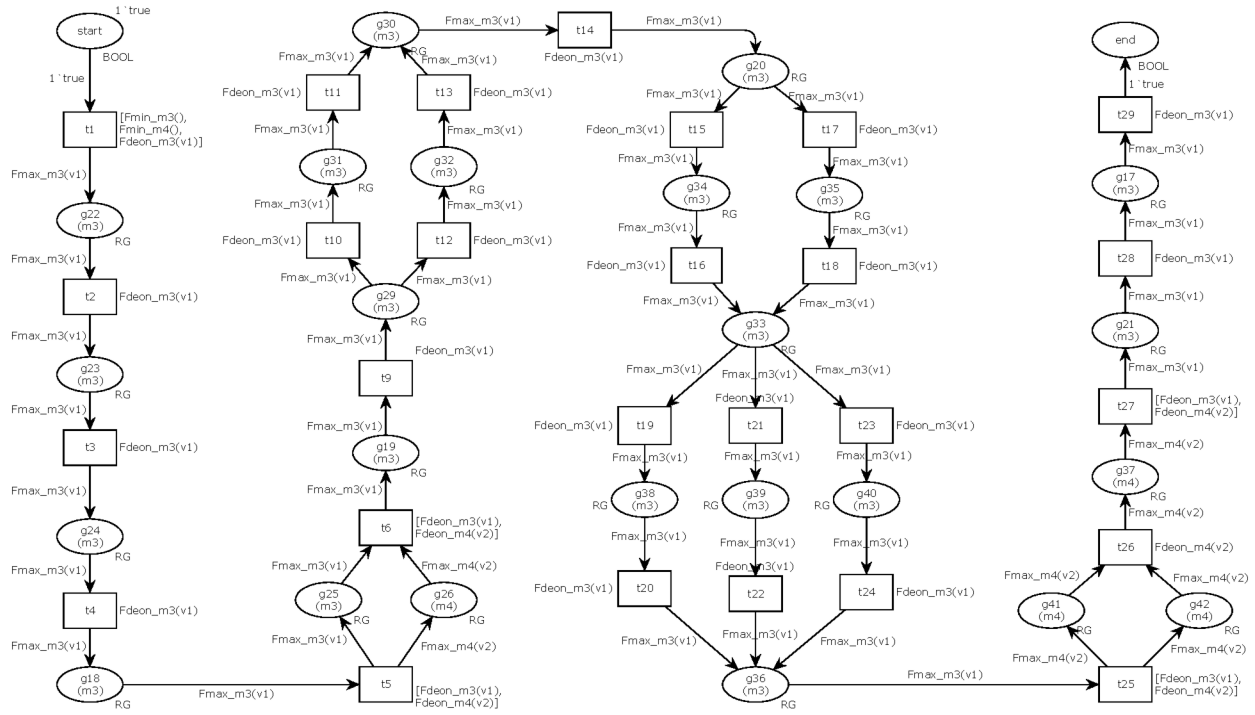


Figura 21: Todas as inscrições incluídas no cenário Inspector.

cardinalidade máxima também foi mantida: por exemplo na imagem (a) tem-se 4 fichas (*soyp*, *gorim*), uma vez que o valor de *Nsoyp* foi declarado com o número 6 e a rede utilizou apenas a quantidade máxima permitida de fichas definida na função de cardinalidade máxima para a missão *m4*. Já em *g25* foi carregada 1 ficha com o papel *inspector*, o que está de acordo com a especificação deôntica de *m3*.

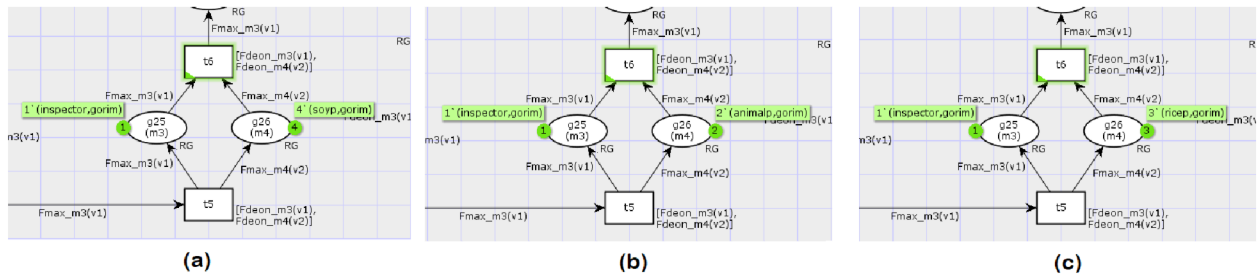


Figura 22: Diferentes simulações no cenário Inspector.

Nesta etapa do mapeamento, demonstramos a capacidade do método em gerar uma RPC fiel ao modelo *Moise*⁺ original. Agora, com o objetivo de identificar erros na modelagem da organização, o que não seria possível com o modelo usado para gerar a figura 21 já que ele não possui nenhuma falha, o modelo XML utilizado foi alterado. Mais especificamente, foi removida a meta *g40* na lista de metas da missão *m3*. Assim gerou-se uma nova RPC para esse XML com falha que pode ser visto na figura 23 em que o lugar com a meta *g40* acabou virando um booleano, e foi destacado com um círculo vermelho.

Nas etapas seguintes usaremos o modelo da Figura 23 como base para contar, gerar e executar os casos de teste, e verificar se é possível identificar esta falha.

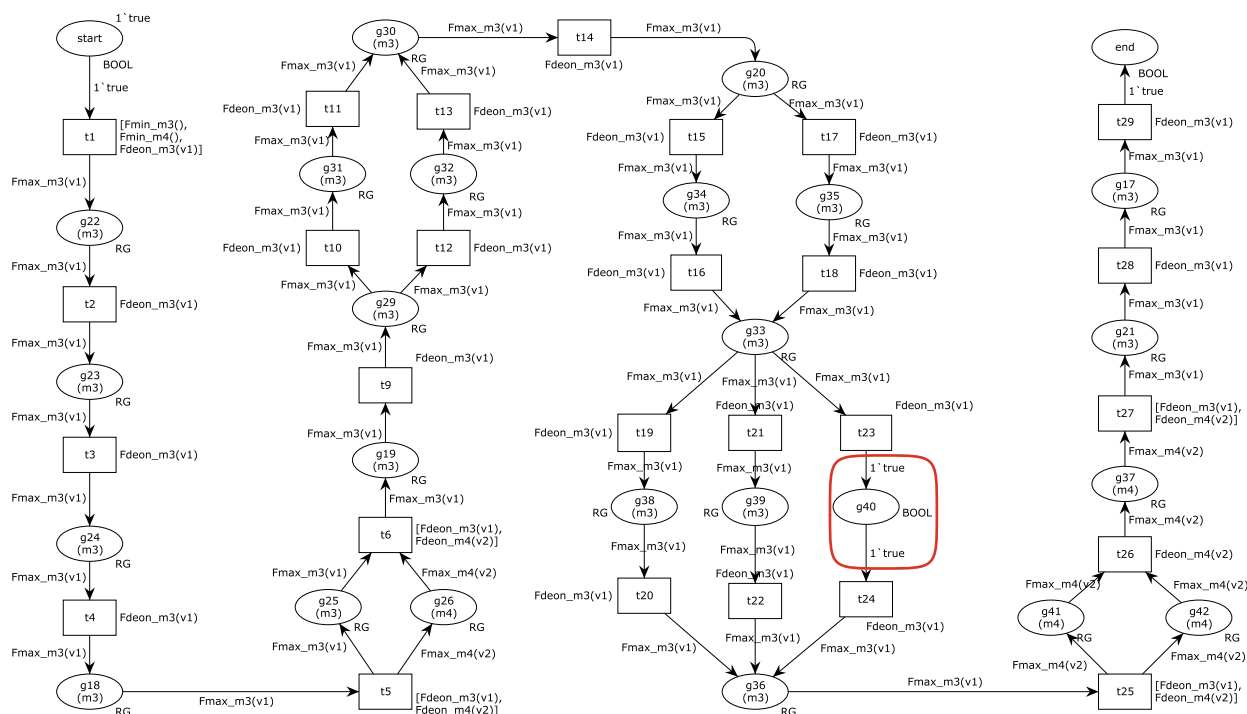


Figura 23: RPC do modelo Inspector com falha.

4.1.2 Contagem dos Casos de Teste

A RPC gerada na Figura 23 será usada para analisar o grau de testabilidade do sistema, retornando o número de casos de teste. Diferentes critérios de cobertura podem ser usados para gerar casos de teste, por exemplo, utilizando um critério bem conhecido chamado critério de cobertura *all-paths* (BANG-JENSEN; GUTIN, 2008). O critério *all-paths* exige que todos os caminhos de execução, desde a entrada até a saída do programa, sejam executados durante o teste. Neste exemplo, usando este critério, foram identificados 12 caminhos de teste diferentes, resultando em 12 casos de teste.

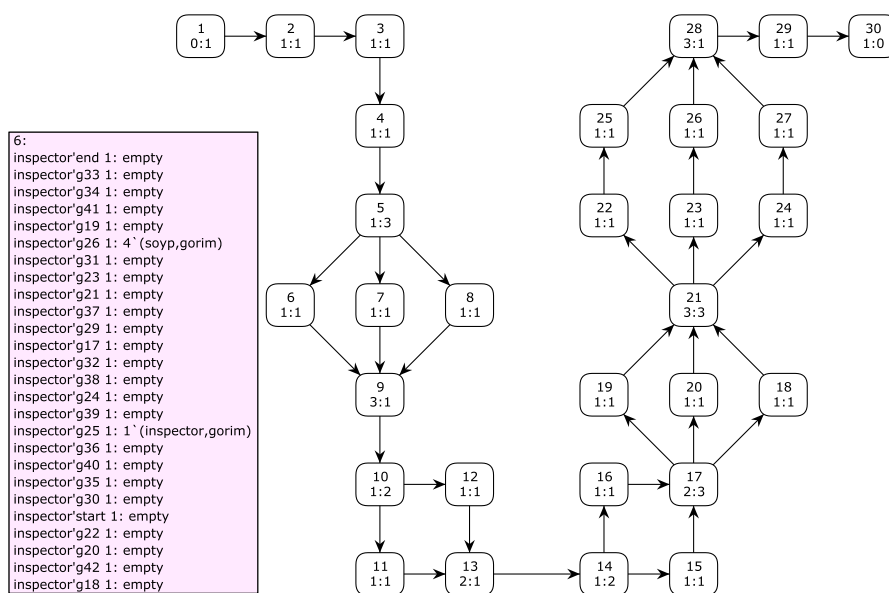


Figura 24: Grafo de Espaço de Estado para cenário Inspector.

No entanto, o critério de cobertura *all-paths* não é eficiente para usar em RPC, pois não

considera os diferentes valores que as fichas podem representar. Portanto, para contar os casos de teste, deve-se usar o critério de *caminho de transição de estado* (ZHU; HE, 2002), onde as diferentes marcações de uma RPC são transformadas em um grafo direcionado, chamado *espaço de estados*, que corresponde ao seu grafo de alcançabilidade.

Usando a ferramenta de modelagem *CPNTools*, é possível gerar um grafo de espaço de estados, e a Figura 24 mostra este grafo, onde os nós representam as marcações da rede, e os arcos representam os elementos de conexão entre essas marcações. Neste exemplo, o nó número 6 foi expandido, mostrando suas marcações. Assim, o nó 6 representa uma ficha (*inspector, gorim*) no lugar *g25* e quatro fichas (*soyp, gorim*) no *g26*. O critério de *caminho de transição de estado* considera todos os possíveis caminhos de execução para um grafo de espaço de estados em uma RPC, então deve-se identificar cada um desses caminhos correspondentes aos casos de teste.

Usando um algoritmo de *backtracking* para contar o número total de caminhos entre dois vértices no grafo direcionado da Figura 24, do nó 1 ao nó 36, encontram-se 108 caminhos diferentes que resultaram em 108 casos de teste. No entanto, essa quantidade de casos de teste abrange um cenário com o mesmo número de agentes em cada papel. Para que o número de casos de teste seja de acordo com as dimensões dadas pela cardinalidade de cada missão, acrescenta-se ao cálculo a variável das possíveis entradas em cada missão. Por exemplo, para a missão *m3* tem-se o $Cmin_m3 = 1$ e $Cmax_3 = 1$ e para a missão *m4* tem-se $Cmin_m4 = 1$ e $Cmax_m4 = 4$, o que retorna os valores 1 para *m3* e 4 para *m4*. Multiplicando os valores encontrados temos $108 * 1 * 4 = 432$ casos de teste.

4.1.3 Geração dos Casos de Teste

Com o número total de caminhos de teste calculados usando um diagrama de controle de fluxo do espaço de estados do *CPN4M*, é necessário gerar uma descrição para cada um desses caminhos, que serão os casos de teste.

O objetivo dessa etapa é criar uma tabela individual para cada caso de teste contendo todo o caminho de execução com as informações como quais metas são selecionadas, a quantidade de agentes e o papel. Também uma breve descrição de cada meta pode ser incluída, sendo que essa informação pode ou não estar na especificação funcional do sistema.

Para exemplificar, serão apresentados dois casos de teste, o primeiro está na Tabela 6 e chama-se de caso de teste *A*. Neste exemplo, pode-se ver todos os nós do grafo mostrando a sequência de execução. Neste caso de teste tem-se a quantidade de 1 agente com o papel *inspector* responsável pela missão *m3* e 2 agentes com o papel *soyp* responsáveis pela missão *m4*. Nas metas que possuem escolha ficaram selecionadas as metas *g31*, *g34* e *g39*. Uma observação, onde existe execução paralela, como nos nós número 6 e 23, as metas são colocadas em linhas separadas.

O segundo caso de teste, denominado *B*, está na Tabela 7, tem-se novamente 1 agente no papel *inspector*, porém agora são 3 agentes no papel *ricep*. As metas nos 3 operadores escolha também mudaram, agora são elas *g32*, *g35* e *g40*. Como a missão da meta *g40* foi removida, o lugar na rede se tornou booleano então no caso de teste ela também aparece desta forma ficando com numero zero de agentes e papel do tipo *true*. Comparando com o caso de teste *A*, a sequência de nós também mudou, isso porque a tabela é feita com base no grafo de espaço de estados, e o caminho dela foi realizado de forma diferente, bem como a sequência de nós.

Com os casos de teste gerados todas as fases do método de testes aqui definidos estão concluídos. Falta efetuar a execução dos testes utilizando os casos de teste definidos. Para isto, é necessário preparar o ambiente de teste com as entradas devidamente definidas, de forma que todo o caminho de execução para cada caso de teste seja alcançado. A seguir é apresentada a execução para os dois casos de teste, para melhor entendimento da proposta.

Tabela 6: Caso de teste *A* gerado para o cenário *Inspector*.

Nó	Meta	Agentes	Papel	Descrição
1	start	0	<i>true</i>	Iniciando
2	g22	1	<i>inspector</i>	Localizar produtor
3	g23	1	<i>inspector</i>	Ir na propriedade
4	g24	1	<i>inspector</i>	Falar com o produtor
5	g18	1	<i>inspector</i>	Contatar produtor
6	g25	1	<i>inspector</i>	Visualizar propriedade
6	g26	2	<i>soyp</i>	Entregar informações da propriedade
9	g19	1	<i>inspector</i>	Obter informações.
10	g29	1	<i>inspector</i>	Consultar tabela do ambiente
12	g31	1	<i>inspector</i>	Avaliar produção
13	g30	1	<i>inspector</i>	Avaliar impacto na propriedade
14	g20	1	<i>inspector</i>	Avaliar informações
16	g34	1	<i>inspector</i>	Definir valor da multa
17	g33	1	<i>inspector</i>	Definir sanções
19	g39	1	<i>inspector</i>	Aplicar notificação
21	g36	1	<i>inspector</i>	Comunicar sanções
23	g41	2	<i>soyp</i>	Pagar multa
23	g42	2	<i>soyp</i>	Confirmar recebimento da notificação
26	g37	2	<i>soyp</i>	Cumprir as sanções
28	g21	1	<i>inspector</i>	Aplicar sanção
29	g17	1	<i>inspector</i>	Realizar fiscalização
30	end	0	<i>true</i>	finalizado

4.1.4 Executando os Casos de Teste

Esta seção apresenta como são feitas as execuções dos casos de teste no sistema *Moise*⁺. Para isto, o cenário *inspector* foi implementado na plataforma JaCaMo. No Apêndice 6 tem-se o código XML da Especificação Estrutural, Especificação Funcional e da Especificação Deontica do cenário *Inspector* (Figuras 38, 39 e 40).

Primeiramente, executa-se o sistema utilizando as definições do caso de teste *A* da Tabela 6. Assim ficaram definidos 2 agentes de teste para o papel *soyp*, chamados *soy1* e *soy2* e 1 agente para o papel *inspector* chamado *ins1*.

Na Figura 25 pode-se visualizar a execução do teste *A*, onde a parte (a) constitui a preparação do ambiente, mostrando o estado de cada meta e qual agente está comprometido à satisfazê-la. Pode-se ver que para o agente *ins1* está disponível as 20 metas pertencentes à missão *m3* e para os 2 agentes com o papel *soyp* estão disponíveis as 4 metas pertencentes à *m4*. Porém como algumas metas estão dentro de um operador de escolha, e nem todas serão executadas (as metas *g40*, *g32*, *g35* e *g38*, não serão escolhidas). Já na Figura 25 (b) está o restante da execução do teste com cada agente identificado a meta executada (para facilitar o entendimento, foram destacadas as metas *g31*, *g34* e *g39*, que foram as opções dos operadores de escolha desse caso de teste). Como é mostrado esse caso de teste foi executado com sucesso sem apresentar erros.

Já na Figura 26 tem-se a execução do caso de teste *B*, onde é possível conferir que a sequência de execução das metas corresponde à ordem colocada no caso de teste da Tabela 7. Na parte (a) temos todas as 24 metas em espera com seus respectivos agentes alocados a elas. Uma importante mudança está nos agentes, agora temos três agentes *ric1*, *ric2* e *ric3*

Tabela 7: Caso de teste B gerado para o cenário *Inspector*.

Nó	Meta	Agentes	Papel	Descrição
1	start	0	<i>true</i>	Iniciando
2	g22	1	<i>inspector</i>	Localizar produtor
3	g23	1	<i>inspector</i>	Ir na propriedade
4	g24	1	<i>inspector</i>	Falar com o produtor
5	g18	1	<i>inspector</i>	Contatar produtor
8	g25	1	<i>inspector</i>	Visualizar propriedade
8	g26	3	<i>ricep</i>	Entregar informações da propriedade
9	g19	1	<i>inspector</i>	Obter informações.
10	g29	1	<i>inspector</i>	Consultar tabela do ambiente
11	g32	1	<i>inspector</i>	Avaliar índice de poluição
13	g30	1	<i>inspector</i>	Avaliar impacto na propriedade
14	g20	1	<i>inspector</i>	Avaliar informações
15	g35	1	<i>inspector</i>	Definir notificação
17	g33	1	<i>inspector</i>	Definir sanções
18	g40	0	<i>true</i>	Não aplicar sanção
21	g36	1	<i>inspector</i>	Comunicar sanções
22	g41	3	<i>ricep</i>	Pagar multa
22	g42	3	<i>ricep</i>	Confirmar recebimento da notificação
25	g37	3	<i>ricep</i>	Cumprir as sanções
28	g21	1	<i>inspector</i>	Aplicar sanção
29	g17	1	<i>inspector</i>	Realizar fiscalização
30	end	0	<i>true</i>	finalizado

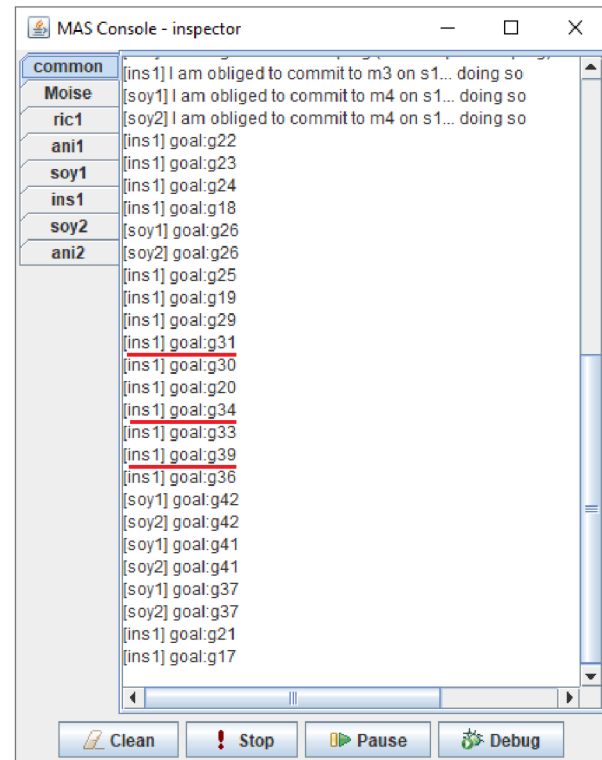
do papel *ricep*. E nas escolhas, agora temos $g32$, $g35$ e $g40$, mas como pode ser observado, a execução parou na meta $g33$, e de acordo com a Tabela 7, $g33$ pertence ao nó 17, sendo o próximo nó o 18, onde a meta $g40$ está localizada. Portanto, o sistema parou exatamente na meta de falha, identificando assim o erro, que neste caso é a falha ao atribuir $g40$ à missão $m3$. Como consequência, o objetivo $g40$ não está relacionado deonticamente ao papel de *inspector*, e não há nenhum agente comprometido com a meta.

Esse erro de implementação usado como exemplo, onde a meta não foi atribuída à missão designada, é apenas um dos possíveis problemas que podem ocorrer em um modelo organizacional como o $\mathcal{M}oisse^+$. Outras possibilidades incluem atribuir o objetivo à missão errada, uma missão estar relacionada ao papel errado no DS, ou até mesmo alguma falha na estrutura de decomposição dos objetivos.

As etapas de geração e execução dos casos de teste ainda são feitas de maneira manual uma vez que a ferramenta de automatização apenas gera a RPC mas não realiza a interpretação por isso essas duas etapas são ainda suscetíveis a erros e requerem um maior cuidado. Por exemplo na geração dos casos de teste dependendo da quantidade é possível alguns casos de teste não sejam gerados corretamente, mas mesmo com todos os casos de teste gerados quando realizamos a execução o sistema precisa ser configurado um a um para cada caso de teste, garantindo que a quantidade correta de agentes em cada papel e a decomposição das metas obedeça a ordem estabelecida em cada tabela, o que demanda tempo e uma maior atenção.

Com a execução dos casos de teste gerados, confirma-se a eficácia do modelo de mapeamento proposto neste trabalho para a realização de testes em SMA organizados. Aqui foi demonstrado a capacidade de configurar a plataforma de testes com os casos de teste gerados, o que é fundamental para que possamos garantir uma ampla cobertura de testes.

goal changed: goalState(Sch,g42,[soy1,soy2],[],waiting)
 goal changed: goalState(Sch,g41,[soy1,soy2],[],waiting)
 goal changed: goalState(Sch,g26,[soy1,soy2],[],waiting)
 goal changed: goalState(Sch,g37,[soy1,soy2],[],waiting)
 goal changed: goalState(Sch,g40,[ins1],[],waiting)
 goal changed: goalState(Sch,g20,[ins1],[],waiting)
 goal changed: goalState(Sch,g31,[ins1],[],waiting)
 goal changed: goalState(Sch,g30,[ins1],[],waiting)
 goal changed: goalState(Sch,g22,[ins1],[],enabled)
 goal changed: goalState(Sch,g33,[ins1],[],waiting)
 goal changed: goalState(Sch,g32,[ins1],[],waiting)
 goal changed: goalState(Sch,g21,[ins1],[],waiting)
 goal changed: goalState(Sch,g24,[ins1],[],waiting)
 goal changed: goalState(Sch,g35,[ins1],[],waiting)
 goal changed: goalState(Sch,g23,[ins1],[],waiting)
 goal changed: goalState(Sch,g34,[ins1],[],waiting)
 goal changed: goalState(Sch,g25,[ins1],[],waiting)
 goal changed: goalState(Sch,g36,[ins1],[],waiting)
 goal changed: goalState(Sch,g39,[ins1],[],waiting)
 goal changed: goalState(Sch,g17,[ins1],[],waiting)
 goal changed: goalState(Sch,g38,[ins1],[],waiting)
 goal changed: goalState(Sch,g19,[ins1],[],waiting)
 goal changed: goalState(Sch,g18,[ins1],[],waiting)
 goal changed: goalState(Sch,g29,[ins1],[],waiting)

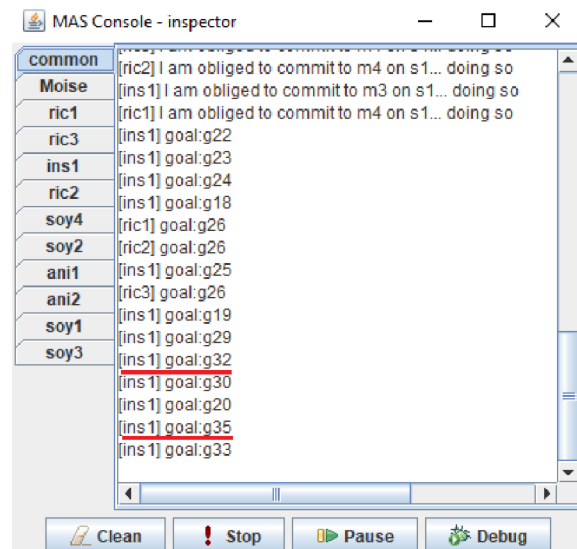


(a)

(b)

Figura 25: Rodando o caso de teste A para o cenário Inspector.

goal changed: goalState(Sch,g42,[ric1,ric2,ric3],[],waiting)
 goal changed: goalState(Sch,g41,[ric1,ric2,ric3],[],waiting)
 goal changed: goalState(Sch,g26,[ric1,ric2,ric3],[],waiting)
 goal changed: goalState(Sch,g37,[ric1,ric2,ric3],[],waiting)
 goal changed: goalState(Sch,g40,[ins1],[],waiting)
 goal changed: goalState(Sch,g20,[ins1],[],waiting)
 goal changed: goalState(Sch,g31,[ins1],[],waiting)
 goal changed: goalState(Sch,g30,[ins1],[],waiting)
 goal changed: goalState(Sch,g22,[ins1],[],enabled)
 goal changed: goalState(Sch,g33,[ins1],[],waiting)
 goal changed: goalState(Sch,g32,[ins1],[],waiting)
 goal changed: goalState(Sch,g21,[ins1],[],waiting)
 goal changed: goalState(Sch,g24,[ins1],[],waiting)
 goal changed: goalState(Sch,g35,[ins1],[],waiting)
 goal changed: goalState(Sch,g23,[ins1],[],waiting)
 goal changed: goalState(Sch,g34,[ins1],[],waiting)
 goal changed: goalState(Sch,g36,[ins1],[],waiting)
 goal changed: goalState(Sch,g39,[ins1],[],waiting)
 goal changed: goalState(Sch,g17,[ins1],[],waiting)
 goal changed: goalState(Sch,g38,[ins1],[],waiting)
 goal changed: goalState(Sch,g19,[ins1],[],waiting)
 goal changed: goalState(Sch,g18,[ins1],[],waiting)
 goal changed: goalState(Sch,g29,[ins1],[],waiting)



(a)

(b)

Figura 26: Rodando o caso de teste B para o cenário Inspector.

4.2 Cenário Soccer Team

O segundo cenário utilizado como caso de uso chama-se Soccer Team que simula um time de futebol e tem como objetivo principal chegar ao campo adversário e marcar um gol. Ori-

ginalmente esse cenário foi apresentado em (HÜBNER; SICHMAN; BOISSIER, 2002), e mais recentemente em (MACHADO; GONÇALVES, 2020) com algumas modificações.

Primeiramente, são apresentadas a Especificação Estrutural do *Soccer Team*, que pode ser visualizada na Figura 27, onde pode-se identificar os papéis *goalk* (goleiro) e *back* (zagueiro) que pertencem ao grupo *defense* (defesa), *middle* (meia) e *attacker* (atacante) do grupo *attack* (ataque). Para conseguir marcar o gol, os quatro papéis *goalk*, *back*, *middle* e *attacker* que são sub-papéis de *player* (jogador), são utilizados para conduzir a bola do campo de defesa, passar pelo meio campo e ir até o campo de ataque.

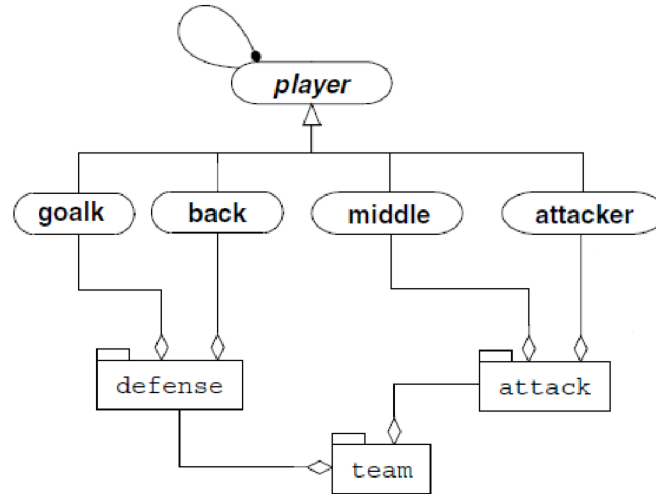


Figura 27: Especificação Estrutural do cenário *Soccer Team*, adaptado de (HÜBNER; SICHMAN; BOISSIER, 2002).

Agora apresentando a Especificação Funcional, tem-se a Figura 28, onde se encontra a árvore de decomposição de metas para o *Soccer Team*. Pode-se ver um total de 17 metas onde a meta *g6* pertence à missão *m1*, as metas *g7*, *g8* e *g9* à *m2*, as metas *g14*, *g16*, *g17* e *g19* pertencem à *m3* e as metas *g1*, *g18*, *g20* e *g21* à *m4*. Tem-se ainda algumas metas sem missão: *g2*, *g3*, *g4*, *g11* e *g13*, o que é uma situação possível no *Moise⁺*, caso essa não seja uma meta folha, e portanto possua sub-metas (BOISSIER et al., 2020).

Na Tabela 8 está a especificação deôntica com a relação entre papéis e missões. Nesse exemplo ambos papéis *goalk* e *back* possuem permissão com a missão *m1*. O papel *back* também possui obrigação com *m2*. Já o papel *middle* possui obrigação com *m3* e o *attacker* obrigação com *m4*.

Tabela 8: Especificação Deôntica para o cenário *Soccer Team*.

papel	relação deôntica	missão
goalk	permissão	<i>m1</i>
back	permissão	<i>m1</i>
back	obrigação	<i>m2</i>
middle	obrigação	<i>m3</i>
attacker	obrigação	<i>m4</i>

4.2.1 Mapeamento do Cenário Soccer Team

- Gerando as Declarações do Cenário Soccer Team

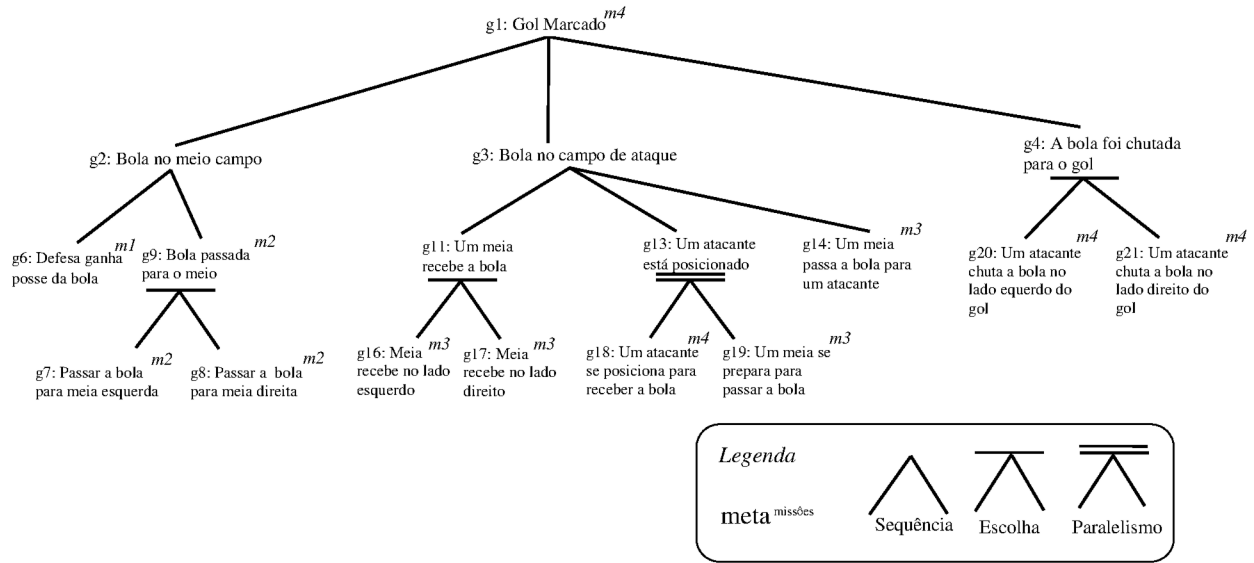


Figura 28: Especificação Funcional do cenário *Soccer Team*, adaptado de (HÜBNER; SICHMAN; BOISSIER, 2002).

Inicia-se o mapeamento apresentando a geração das declarações que constitui a primeira etapa. De acordo com a Definição 2, para cada papel no *Moise+* deve-se incluir as respectivas cores ao conjunto de cor *R* que aqui serão *goalk*, *back*, *middle* e *attacker* declaradas na linha 1. Ainda na Definição 2, tem-se as constantes com referência ao número de agentes para cada papel, portanto tem-se na linha 5 *Ngoalk*, na 6 *Nback*, na 7 *Nmiddle* e na linha 8 *Nattacker*. Agora com base na Definição 3 são declarados os grupos ao conjunto de cor *G*, sendo eles 2 grupos, *defense* e *attack*. Na linha 3 tem-se a declaração do produto entre os conjuntos de cores *R* e *G* criando o conjunto *RG*, sendo essa uma declaração padrão para todos os modelos.

Partindo para as declarações dos elementos das missões, usando a Definição 4, onde para cada missão tem-se uma série de declarações. Inicia-se com a criação de quatro variáveis na linha 4: *v1*, *v2*, *v3* e *v4*, uma para cada missão do exemplo. Também são criadas as constantes *Cmin* e *Cmax* para cada missão, aqui declaradas entre as linhas 9 e 16. As funções de cardinalidade máxima estão declaradas nas linhas 17, 18, 19 e 20. Já as funções de cardinalidade mínima nas linhas 21, 22, 23 e 24. Por fim, tem-se as funções deônticas que aqui estão nas linhas 25, 26, 27 e 28. Com isso fecham-se as declarações e agora o próximo passo é gerar a estrutura da rede.

Lista 3: Declarações do cenário Soccer Team.

```

1. colset R = with goalk | back | middle | attacker;
2. colset G = with defense | attack;
3. colset RG = product R*G;
4. var v1,v2,v3,v4 : RG;
5. val Ngoalk = 1;
6. val Nback = 3;
7. val Nmiddle = 4;
8. val Nattacker = 2;
9. val Cmin_m1 = 1;
10. val Cmax_m1 = 1;
11. val Cmin_m2 = 1;
12. val Cmax_m2 = 4;
13. val Cmin_m3 = 1;
14. val Cmax_m3 = 4;

```

```

15.val Cmin_m4 = 1;
16.val Cmax_m4 = 2;
17.fun Fmax_m1(v1)= case v1 of
(goalk,defense) => (if Ngoalk <= Cmax_m1 then Ngoalk'v1 else Cmax_m1'v1)|
(back,defense) => (if Nback <= Cmax_m1 then Nback'v1 else Cmax_m1'v1);
18.fun Fmax_m2(v2)= case v2 of
(back,defense) => (if Nback <= Cmax_m2 then Nback'v2 else Cmax_m2'v2);
19.fun Fmax_m3(v3)= case v3 of
(middle,attack)=> (if Nmiddle <= Cmax_m3 then Nmiddle'v3 else Cmax_m3'v3);
20.fun Fmax_m4(v4)= case v4 of
(attacker,attack) => (if Nattacker <= Cmax_m4 then Nattacker'v4 else Cmax_m4'v4);
21.fun Fmin_m1()= Ngoalk >=Cmin_m1 andalso Nback >=Cmin_m1;
22.fun Fmin_m2()= Nback>=Cmin_m2;
23.fun Fmin_m3()= Nmiddle>=Cmin_m3;
24.fun Fmin_m4()= Nattacker>=Cmin_m4;
25.fun Fdeon_m1(v1)= v1=(back,defense) orelse v1=(goalk,defense);
26.fun Fdeon_m2(v2)= v2=(back,defense);
27.fun Fdeon_m3(v3)= v3=(middle,attack);
28.fun Fdeon_m4(v4)= v4=(attacker,attack);

```

- Gerando a Estrutura do Cenário Soccer Team

Para gerar esta estrutura deve-se partir da Figura 28 onde está a EF do modelo. Esse processo está dividido em partes, sendo a primeira parte todas as metas que estão abaixo da meta $g2$ e incluindo ela. Portanto, aqui tem-se uma sequência entre $g6$ e $g9$, porém entre essas metas existe também uma escolha entre $g7$ e $g8$. Após $g9$ será então incluída a meta $g2$. Além desses elementos, deve-se incluir também o lugar inicial *start*, esse primeiro bloco pode ser visto na Figura 29.

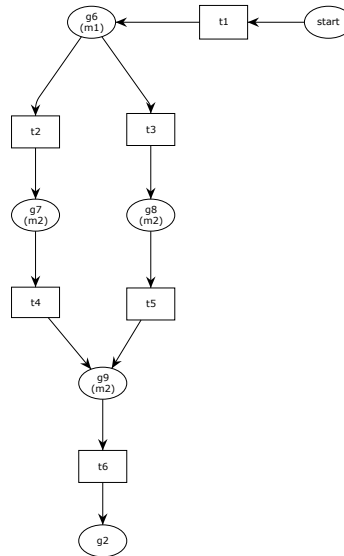


Figura 29: Primeira parte da estrutura para o cenário Soccer Team.

Após, deve-se incluir na estrutura o segundo bloco que são as metas abaixo de $g3$ incluindo ela. Assim, após $g2$ que foi a última meta incluída, tem-se uma escolha entre $g16$ e $g17$, após essa escolha segue para $g11$. Depois de $g11$ tem-se um paralelismo entre $g18$ e $g19$ seguido de uma sequência entre $g13$ e $g14$. Após $g14$ a rede deve seguir para $g3$ e assim encerra o segundo bloco, que está na Figura 30.

Agora resta apenas incluir a terceira e última parte, que inclui uma escolha entre $g20$ e $g21$ após $g3$. Terminada essa escolha a rede segue para $g4$ e $g1$, por fim é também incluído o local final *end*. Assim com essa parte incluída, a estrutura da RPC está completa e pode ser vista na Figura 31.

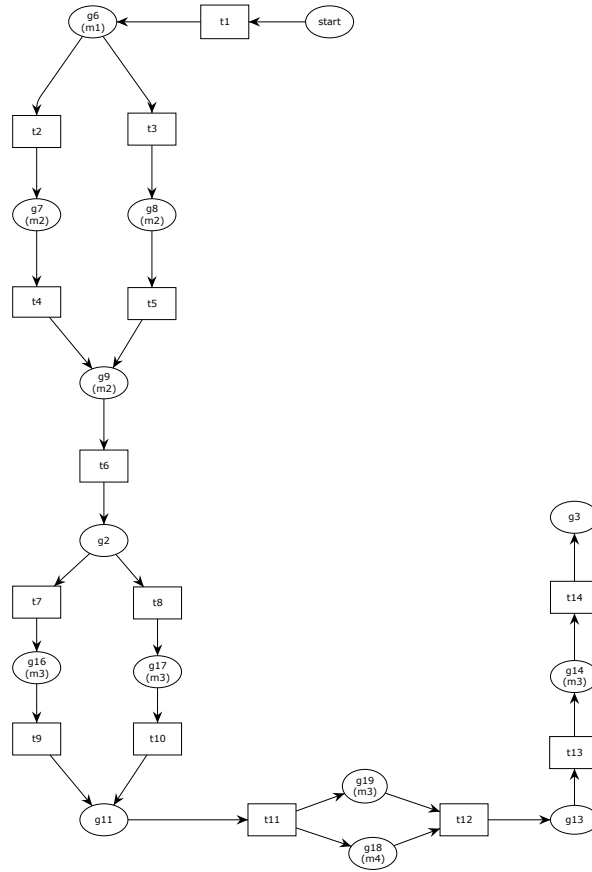


Figura 30: Segunda parte da estrutura para o cenário Soccer Team.

- Gerando as Inscrições no Cenário Soccer Team

Nesta etapa o mapeamento será finalizado com a inclusão das inscrições na estrutura da RPC. Primeiramente, devem ser incluídos as informações básicas como os nomes dos lugares, que de acordo com a Definição 1, são as metas e suas missões. Também são incluídos o tipo de cor de cada lugar, sendo RG para lugares com missões e BOOL para os demais, por fim os nomes de cada transição em ordem crescente. Essas informações já foram incluídas durante a criação da estrutura para facilitar a localização, e estão na Figura 31.

Agora serão incluídas as funções de guardas nas transições da rede. As primeiras funções correspondem a Definição 8 que diz que para cada missão do $\mathcal{Moise}^+$ deve ser inscrita uma chamada de função de cardinalidade mínima na primeira transição da rede. Na Figura 32 pode-se ver estas inscrições onde a transição $t1$ recebeu as chamadas das funções $Fmin_m1$, $Fmin_m2$, $Fmin_m3$, e $Fmin_m4$.

Ainda com relação as chamadas das funções de guarda, tem-se na Definição 9 a inclusão das funções deônticas nas transições. Deve-se analisar para cada transição os lugares anteriores e posteriores: mais especificamente, se nesse lugar existe uma missão inscrita, deve-se ter uma inscrição da chamada da função deôntica dessa missão na respectiva transição. Alguns lugares na rede são metas sem missões portanto nesses não é necessário funções deônticas, pois estão aqui representado com o tipo booleano.

Ainda na Figura 32 pode-se ver essas transições por exemplo tem-se na transição $t1$ a chamada da função $Fdeon_m1$ pois o lugar posterior a $t1$ que é $g6$ pertence à $m1$. Já em $t2$ tem-se duas chamadas de funções deônticas, uma relacionada a $g6$ que é novamente

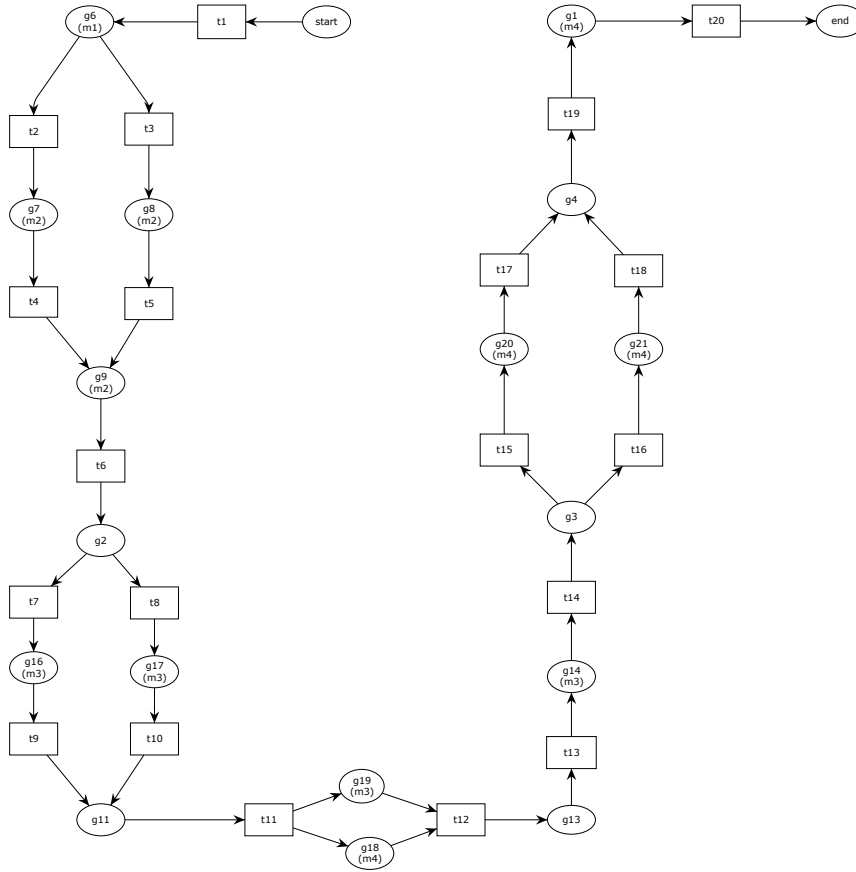


Figura 31: Estrutura Criada para o cenário Soccer Team.

$Fdeon_m1$ e outra relacionada a $g7$ que é $Fdeon_m2$ já que $g7$ pertence à missão $m2$, ambas incluídas utilizando colchetes e separadas por vírgulas, o que representa uma lista.

Agora para finalizar a etapa de inclusão das inscrições falta incluir as inscrições de arcos. Para isso, primeiro deve-se incluir as chamadas das funções de cardinalidade máxima de acordo com a Definição 10. Ao analisar cada arco da rede e sabendo que os arcos da RPC são utilizados sempre como ligação entre um lugar e uma transição, se o lugar ligado a esse arco possui uma missão esse arco recebe a chamada da função de cardinalidade máxima referente à esta missão. Caso o lugar que está ligado ao arco não possua nenhuma missão, então automaticamente ele é do tipo *Bool* e vai entrar na regra descrita na Definição 11, onde receberá a inscrição *true*. Na Figura 33 podemos ver a rede com o mapeamento concluído e com todas as inscrições incluídas.

4.2.2 Contagem dos Casos de Teste

Nessa seção apresenta-se o método para contagem dos casos de teste utilizado no cenário Soccer Team. O critério de cobertura utilizado será o *caminho de transição de estado* (ZHU; HE, 2002), e portanto se gerou o grafo de espaços de estados da RPC, utilizando a ferramenta *CPNTools*, este grafo pode ser visto na Figura 34. Realizando a contagem dos caminhos para o espaço de estado foram identificados 16 caminhos. Agora é preciso multiplicar esse número pelas possíveis entradas em cada missão, respeitando as cardinalidades mínima em máxima. Assim tem-se os seguintes valores: 1 para $m1$, 4 para $m2$, 4 para $m3$ e 2 para $m4$. Multiplicando todos os valores dá um total de 32 possibilidades, e por fim multiplica-se esse valor 32 pelos 16 caminhos encontrados dando um total de 512 casos de teste para o cenário *Soccer Team*.

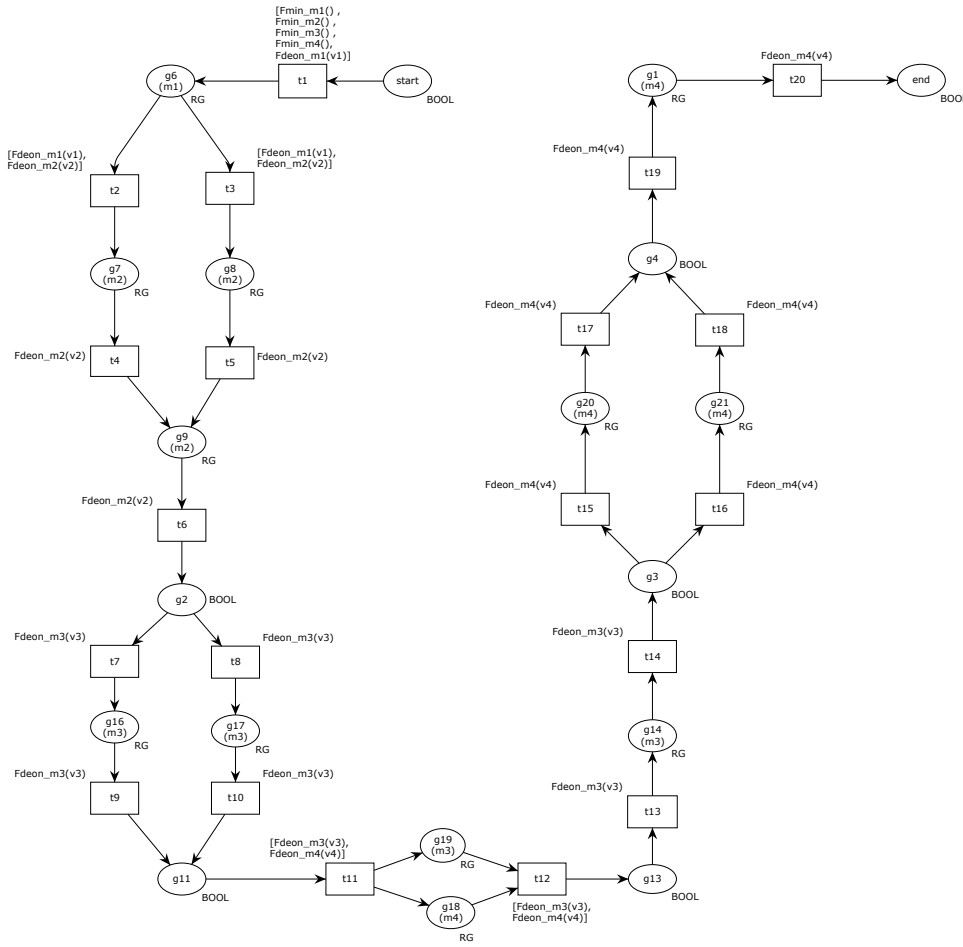


Figura 32: Inicialização de guardas incluídas no cenário Soccer Team.

4.2.3 Geração dos Casos de Teste

Nessa etapa deve-se fazer a geração de dois casos de teste para este cenário, através de uma tabela individual que descreve o caminho de execução, quais metas são selecionadas, a quantidade de agentes e o tipo de papel. Além dessas informações também é incluída uma breve descrição para cada meta, sendo que essa informação pode ou não estar na especificação funcional do sistema.

O primeiro caso de teste é apresentado na Tabela 9 e é chamado caso de teste A, onde é utilizada a seguinte quantidade de agentes nos papéis: *goalk* 0 agentes, *back* 1 agente para *m1* e 3 agentes para *m2*, *middle* 4 agentes e *attacker* 2 agentes. Como particularidades podemos destacar a meta *g6* aqui é dada ao papel *back*, na escolha entre *g7* e *g8* é escolhida *g8*, já entre *g16* e *g17* temos *g16* e entre *g20* e *g21* tem-se a escolha de *g21*.

O segundo caso de teste B, está na Tabela 10, ele possui o mesmo número de agentes do caso de teste A porém com algumas alterações como na meta *g6* que aqui é atribuída ao papel *goalk*, e também algumas escolhas mudaram como *g7* no lugar de *g8* e *g20* no lugar de *g21*. Em ambos casos de teste as metas sem missões estão com *true* no lugar do papel, o que representa a ficha para os lugares booleanos.

4.2.4 Executando os Casos de Teste

Agora com os casos de teste gerados, apresenta-se sua execução utilizando cada um dos dois exemplos apresentados. Para isso o exemplo foi implementado na plataforma JaCaMo e é

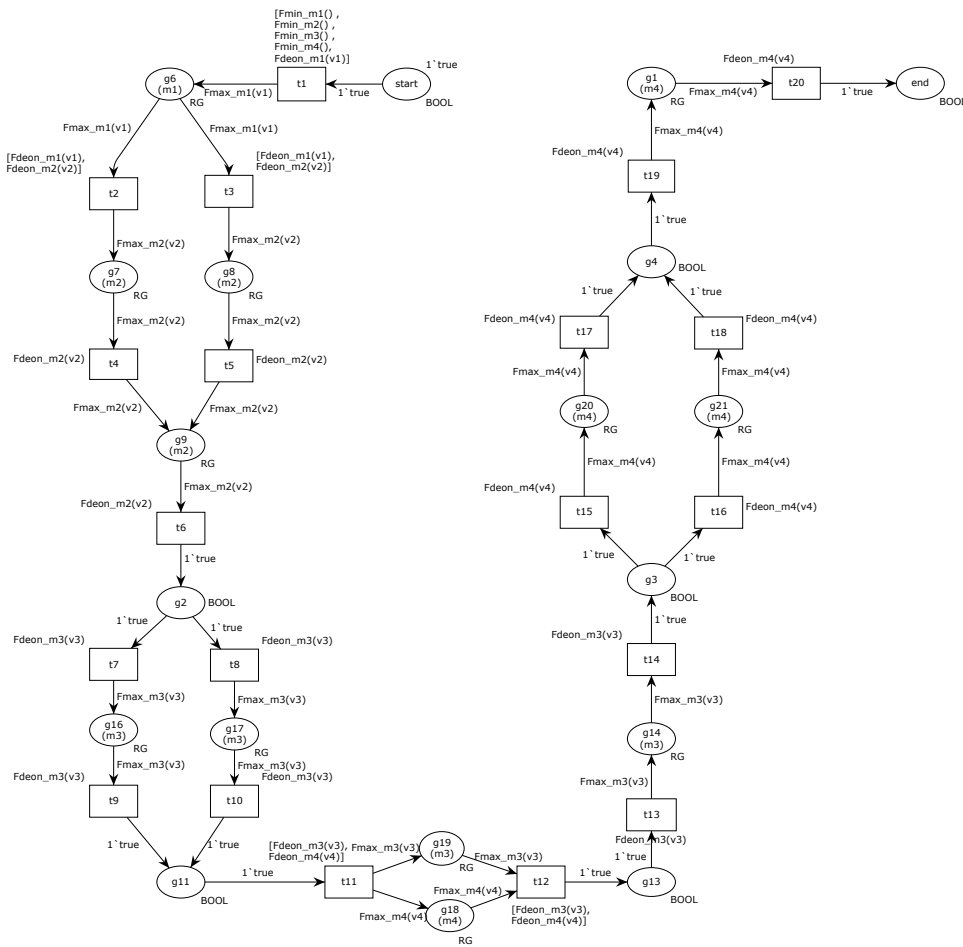


Figura 33: Todas as inscrições incluídas no cenário Soccer Team.

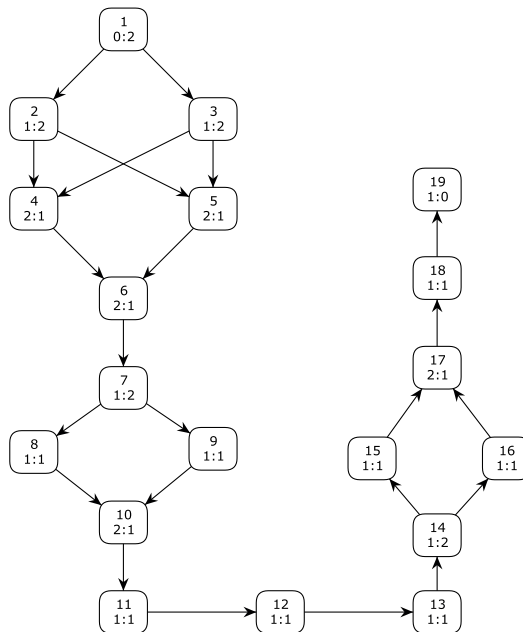


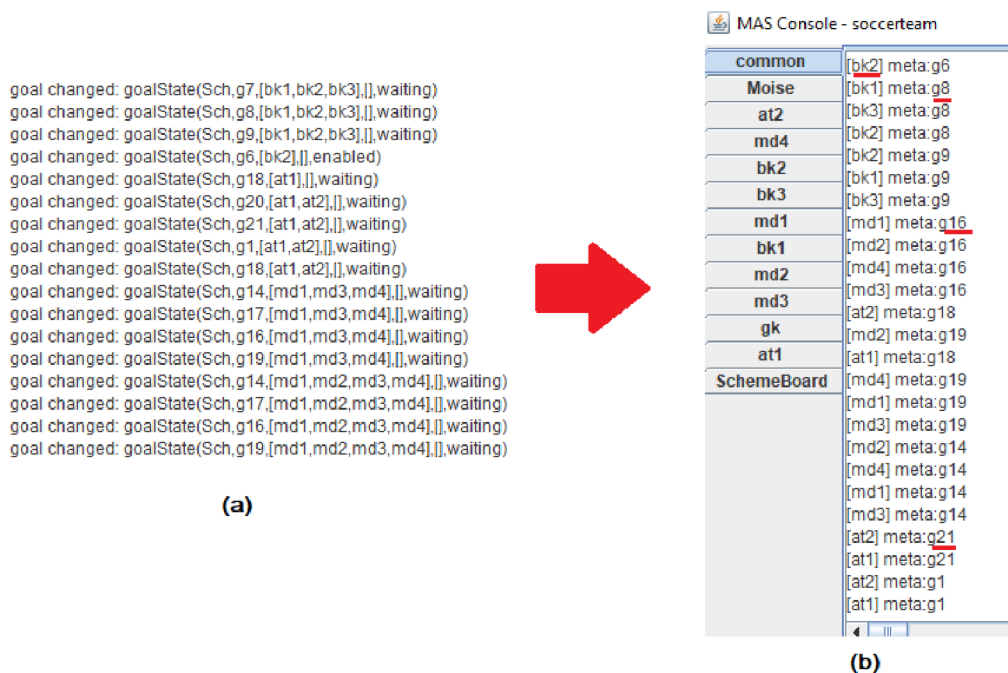
Figura 34: Grafo de Espaço de Estado do cenário Soccer Team.

possível visualizar no Capítulo 6 (Apêndice) o código XML fonte da Especificação Estrutural,

Tabela 9: Caso de teste *A* gerado para o cenário *Soccer Team*.

Nó	Meta	Agentes	Papel	Descrição
1	start	0	<i>true</i>	Iniciando
2	g6	1	<i>back</i>	Defesa ganha posse da bola
5	g8	3	<i>back</i>	Passar a bola para meia direita
6	g9	3	<i>back</i>	Bola passada para o meio
7	g2	0	<i>true</i>	Bola no meio campo
8	g16	4	<i>middle</i>	Meia recebe no lado esquerdo
10	g11	0	<i>true</i>	Um meia recebe a bola
11	g18	2	<i>attacker</i>	Um atacante se posiciona para receber a bola
11	g19	4	<i>middle</i>	Um meia se prepara para passar a bola
12	g13	0	<i>true</i>	Um atacante está posicionado
13	g14	4	<i>middle</i>	Um meia passa a bola para um atacante
14	g3	0	<i>true</i>	Bola no campo de ataque
16	g21	2	<i>attacker</i>	Um atacante chuta a bola para o lado direito do gol
17	g4	0	<i>true</i>	Abola foi chutada para o gol
18	g1	2	<i>attacker</i>	Gol foi marcado
19	end	0	<i>true</i>	Finalizado

Especificação Funcional e Especificação Deôntica. para o cenário *SoccerTeam* (Figuras 41, 42 e 43) .

Figura 35: Rodando o caso de teste *A* para o cenário *Soccer Team*.

Na Figura 35 tem-se a execução do caso de teste *A*, na parte (a) está a informação do estado das metas antes dos agentes começarem a execução, sendo possível ver quais agentes estão comprometidos com essas metas. Por exemplo na primeira linha temos a meta *g7* e os seguintes agentes: *bk1*, *bk2* e *bk3* todos com o papel *back*. Os agentes *at1* e *at2* com o papel *attacker* e os agentes *md1*, *md2*, *md3* e *md4* com o papel *middle*. Nesse caso de teste não

foi utilizado o agente *gk* que é um *goalk* pois a meta *g6* foi atribuída a um *back* no caso *bk2* como é possível ver na Figura 35 (b), também destacam-se as escolhas que foram feitas aqui obedecendo ao caso de teste temos *g8*, *g16* e *g21*.

A execução do caso de teste *B* é apresentado na Figura 36 onde podemos ver que ocorreu um erro quando o agente *gk* tentou satisfazer a meta *g6*, neste caso o programa indicou que a falha se trata da falta de um plano no agente para satisfazer esta meta, o que demonstra a capacidade do método de identificar não somente falhas na organização mas também no código dos agentes.

Após realizada a correção do problema o caso de teste *B* foi rodado novamente, agora sem erros, o que pode ser visto na Figura 37 onde na parte (a) mostra a preparação para execução com os agentes alocados em suas respectivas metas, e na (b) temos a execução de todas as metas. Destacamos em vermelho que o agente *gk* satisfaz a meta *g6* e também que foram realizadas as escolhas das metas *g7*, *g16* e *g20* conforme descrito na Tabela 10.

Para que ambos os casos de teste sejam executados conforme sua especificação é necessário efetuar algumas configurações: como criar os agentes na quantidade correta, ajustar o código do agente para que ele faça as escolhas corretas e garantir que o papel correto se comprometa com a missão quando houver mais de um papel para ela. Por exemplo, a missão *m1* e dos papéis *goalk* e *back*.

Tabela 10: Caso de teste *B* gerado para o cenário *Soccer Team*.

Nó	Meta	Agentes	Papel	Descrição
1	start	0	<i>true</i>	Iniciando
3	<i>g6</i>	1	<i>goalk</i>	Defesa ganha posse da bola
4	<i>g7</i>	3	<i>back</i>	Passar a bola para meia esquerda
6	<i>g9</i>	3	<i>back</i>	Bola passada para o meio
7	<i>g2</i>	0	<i>true</i>	Bola no meio campo
8	<i>g16</i>	4	<i>middle</i>	Meia recebe no lado esquerdo
10	<i>g11</i>	0	<i>true</i>	Um meia recebe a bola
11	<i>g18</i>	2	<i>attacker</i>	Um atacante se posiciona para receber a bola
11	<i>g19</i>	4	<i>middle</i>	Um meia se prepara para passar a bola
12	<i>g13</i>	0	<i>true</i>	Um atacante está posicionado
13	<i>g14</i>	4	<i>middle</i>	Um meia passa a bola para um atacante
14	<i>g3</i>	0	<i>true</i>	Bola no campo de ataque
15	<i>g20</i>	2	<i>attacker</i>	Um atacante chuta a bola para o lado esquerdo do gol
17	<i>g4</i>	0	<i>true</i>	Abola foi chutada para o gol
18	<i>g1</i>	2	<i>attacker</i>	Gol foi marcado
19	end	0	<i>true</i>	Finalizado

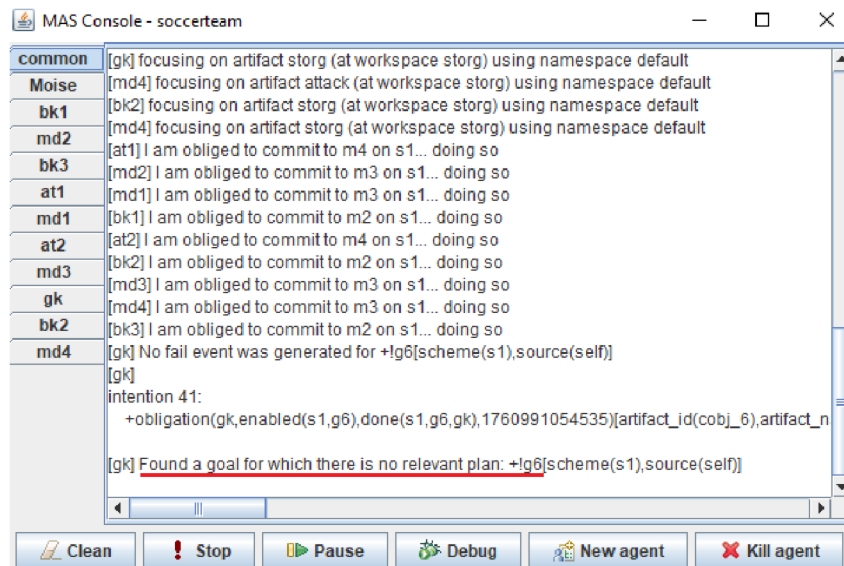


Figura 36: Rodando o caso de teste *B* com erro para o cenário Soccer Team.

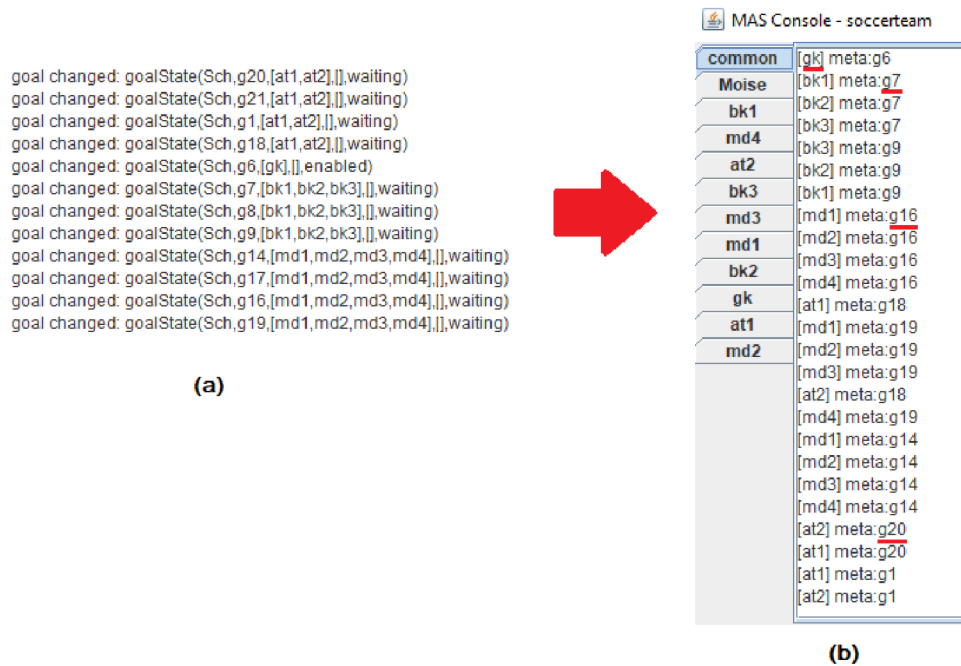


Figura 37: Rodando o caso de teste *B* corrigido para o cenário Soccer Team.

5 CONCLUSÃO

Sistemas multiagentes são artefatos computacionais criados para resolver problemas com base nas interações de seus agentes autônomos. Assim como os sistemas distribuídos com comportamento emergente, uma tarefa desafiadora no desenvolvimento desses sistemas é a verificação. Isso é especialmente verdadeiro para sistemas críticos, como aqueles empregados na fabricação, automação ou sistemas ciber-físicos. Testes e testabilidade são considerados partes de um quebra-cabeça para buscar uma confiabilidade suficientemente alta para o software (VOAS; MILLER, 1995). No entanto, eles podem apresentar um comportamento não determinístico, uma vez que a mesma entrada pode gerar diferentes saídas.

Os SMA tem em sua arquitetura componentes distribuídos entre três níveis: (i) o nível individual, onde os agentes são especificados e implementados; (ii) os aspectos de interação, onde são especificados e formalizados os protocolos de comunicação; (iii) e o nível social, que pode ser realizado em uma formalização explícita como um modelo organizacional (GONÇALVES; QUADROS; SALDANHA, 2016).

Com foco no nível social dos SMA, foi apresentada nesse trabalho uma proposta de reformulação para o método de mapeamento de um sistema multiagente que utiliza o modelo organizacional Moise^+ , para uma Rede de Petri Colorida denominada CPN4M , anteriormente proposta em (MACHADO, 2020). Esta reformulação visa incluir elementos como a cardinalidade, grupos, e o número de agentes existentes para um dado papel, gerando assim um modelo em RPC mais fiel ao sistema original.

Além da reformulação, cada etapa de mapeamento agora apresenta definições formais individuais baseados na lógica de predicados, o que traz uma transparência ao processo, evitando ambiguidades já que utiliza um vocabulário técnico e objetivo.

Comparado ao trabalho anterior, essa nova abordagem traz algumas melhorias, como a inclusão de valores de cardinalidade para cada missão, o que resultou na adição de vários elementos que anteriormente não existiam na rede de Petri, como constantes com o número de agentes, constantes de cardinalidade mínima e máxima, e três novas funções. A primeira função, F_{max} , é utilizada nos arcos e garante que o número de agentes não exceda a cardinalidade máxima. A segunda função, F_{min} , é utilizada na primeira transição e impede que a rede inicie se não houver agentes suficientes. A terceira função, F_{deon} , é uma função deontica que atribui o papel correto a uma determinada missão.

Pode-se afirmar que como principal contribuição deste trabalho tem-se a elaboração de um método formal capaz de gerar casos de teste a nível organizacional para um SMA através de um mapeamento em RPC. Afirma-se também que o modelo mapeado se encontra fiel ao modelo organizacional Moise^+ original, contendo os elementos papéis, grupos, missões, as especificações estruturais, funcionais e deontica e as cardinalidades das missões. Ficaram de fora apenas as cardinalidades dos papéis e o tipo de relação deontica como permissão ou obrigação, uma vez que a inclusão dessas informações não alteraria os resultados obtidos.

O método aqui apresentado e suas definições formais serviram também como base para a criação de uma ferramenta que automatiza o processo. A ferramenta de código aberto para interpretação de arquivos XML do Moise^+ para RPC chama-se MAMTCPN (MACHADO et al., 2024), e ainda encontra-se em estágio de desenvolvimento.

Um conjunto de casos de teste também foi gerado para um critério de adequação chamado *caminho de transição de estados*. Isso possibilitou confirmar que o mapeamento pode gerar os possíveis casos de teste para o modelo de SMA em teste. Esses casos de testes foram executados utilizando a plataforma de desenvolvimento JaCaMo para demonstrar seu uso em uma aplicação real. Para demonstrar a eficácia do método em detectar uma falha de implementação na organização, uma meta específica foi removida do conjunto de metas da missão. Ao executar

os casos de teste, o sistema não executou totalmente em um deles, o que diferiu do resultado esperado se tudo estivesse correto, possibilitando assim a identificação da falha. Além disso também foi provocada uma falha em um agente e ao executar o caso de teste o sistema acusou o problema que foi corrigido e apresentada a execução completa demonstrando a capacidade do método em identificar também erros a nível de agentes.

Entre os problemas encontrados, cita-se a dificuldade de lidar com variáveis na RPC, o que impossibilitou por exemplo a geração aleatória da quantidade de agentes com base nas cardinalidades mínima e máxima. Por exemplo foi atribuído à uma constante o parâmetro *discrete*($Cmin, Cmax$), o que gera um número inteiro aleatório no intervalo previsto nas cardinalidades, sendo esta constante utilizada nas funções de arcos. Porém mesmo ao reiniciar a rede este valor permanecia o mesmo o que dificultada realizar simulações com diferentes parâmetros. Para contornar este problema, a quantidade de agentes deve ser informada manualmente, e foram criados mecanismos de controle de cardinalidade através de funções específicas ($Fmin$ e $Fmax$) para cada missão.

Em relação à escalabilidade dos modelos organizacionais, acreditamos que eles não são muito maiores do que dezenas de papéis, e podem ter centenas de objetivos. No caso de $\mathcal{Moise}^+$, não encontramos modelos muito maiores do que o testado neste trabalho. No entanto, acreditamos que nossa abordagem é capaz de realizar os casos de teste de tradução e geração.

Como trabalhos futuros tem-se a sugestão da inclusão de uma constante às declarações. Esta constante receberá o valor da multiplicação dos intervalos de cardinalidade de todas as missões automaticamente, valor este que é utilizado para a contagem dos casos de teste. Por exemplo tem-se para um cenário qualquer duas missões cada uma com duas possibilidades de entrada no número de agentes, essa constante então recebe o valor da multiplicação delas que seria o número 4.

Ao mesmo tempo tem-se a possibilidade de automatizar também as etapas de contagem e geração de casos de teste, agilizando o processo e eliminando possíveis falhas humanas. Para isso, existe um *plug-in* chamado *accessCPN* (WESTERGAARD, 2011), que permite que programas Java incorporem programas inteiros como sub-páginas de modelos de RPC ou incorporem esses modelos como partes de programas, facilitando a construção de ambientes para testes sistemáticos de componentes de programa usando modelos de RPC. Por fim, é interessante a criação de uma interface gráfica amigável para integrar todo o ferramental computacional criado, de forma a facilitar o uso do método proposto.

REFERÊNCIAS

- ABDULLAH, D.; BURNSTEIN, I. **Practical Software Testing**. [S.l.]: Springer, 2003.
- ABRAN, A.; MOORE, J. W.; BOURQUE, P.; DUPUIS, R.; TRIPP, L. Software engineering body of knowledge. **IEEE Computer Society, Angela Burgess**, [S.l.], 2004.
- ALDAHASH, N.; KING, S.; BENABID, A. Interaction Petri Nets for Modeling Multiagent Interactions. In: CONGRESS IN COMPUTER SCIENCE, COMPUTER ENGINEERING, & APPLIED COMPUTING (CSCE), 2023., 2023. **Anais...** [S.l.: s.n.], 2023. p.2061–2068.
- 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.
- BAKAR, N. A.; SELAMAT, A. Agent systems verification: systematic literature review and mapping. **Applied Intelligence**, [S.l.], v.48, n.5, p.1251–1274, 2018.
- BANG-JENSEN, J.; GUTIN, G. Z. **Digraphs: theory, algorithms and applications**. [S.l.]: Springer Science & Business Media, 2008.
- BARNIER, C.; MERCIER, A.; JAMONT, J.-P. et al. Toward an Embedded Multi-agent System Methodology and Positioning on Testing. In: IEEE INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING WORKSHOPS (ISSREW), 2017., 2017. **Anais...** [S.l.: s.n.], 2017. p.239–244.
- BENAC EARLE, C.; FREDLUND, L.-Å. A Property-based Testing Framework for Multi-Agent Systems. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, 18., 2019. **Proceedings...** [S.l.: s.n.], 2019. p.1823–1825.
- BOISSIER, O.; BORDINI, R. H.; HÜBNER, J. F.; RICCI, A.; SANTI, A. Multi-agent oriented programming with JaCaMo. **Science of Computer Programming**, [S.l.], v.78, n.6, p.747–761, 2013.
- BOISSIER, O.; BORDINI, R. H.; HUBNER, J.; RICCI, A. **Multi-agent oriented programming: programming multi-agent systems using JaCaMo**. [S.l.]: Mit Press, 2020.
- BOISSIER, O.; HÜBNER, J. F.; RICCI, A. The JaCaMo Framework. In: **Social coordination frameworks for social technical systems**. [S.l.]: Springer, 2016. p.125–151.
- BORN, M. B. **Modelagem a nível organizacional de agentes em um jogo do tipo RPG: estudo da complexidade de seus personagens e de suas funcionalidades**. 2022. Tese (Doutorado em Ciência da Computação) — Universidade Federal de Pelotas.
- BOUCHERIT, A.; CASTRO, L. M.; KHABABA, A.; HASAN, O. Petri net and rewriting logic based formal analysis of multi-agent based safety-critical systems. **Multiagent and Grid Systems**, [S.l.], v.16, n.1, p.47–66, 2020.
- CASTRO, L. F. S. d. et al. **Desenvolvimento de uma arquitetura multiagente holônica reconfigurável aplicada ao cenário de smart parking**. 2018. Dissertação (Mestrado em Ciência da Computação) — Universidade Tecnológica Federal do Paraná.

- CLARK, A.; WALKINSHAW, N.; HIERONS, R. M. Test case generation for agent-based models: A systematic literature review. **Information and Software Technology**, [S.l.], p.106567, 2021.
- DASH, N. P.; DASGUPTA, R.; CHEPADA, J.; HALDER, A. Event driven programming for embedded systems-a finite state machine based approach. In: THE SIXTH INTERNATIONAL CONFERENCE ON SYSTEMS, 2011. **Anais...** [S.l.: s.n.], 2011. p.23–28.
- DIGNUM, V. **Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models: Semantics and Dynamics of Organizational Models**. [S.l.]: IGI Global, 2009.
- EASSA, F. E.; OSTERWEIL, L. J.; FADEL, M. A.; SANDOKJI, S.; EZZ, A. DTTAS: A Dynamic Testing Tool for Agent-based Systems. **Pensee Journal**, [S.l.], v.76, n.5, 2014.
- EL HOUDA, D. N.; SOUFIENE, B.; DJABER, G. Towards a new dynamic model-based testing approach for multi-agent systems. In: INTERNATIONAL CONFERENCE ON PATTERN ANALYSIS AND INTELLIGENT SYSTEMS (PAIS), 2022., 2022. **Anais...** [S.l.: s.n.], 2022. p.1–6.
- ENGELMANN, D. C.; FERRANDO, A.; PANISSON, A. R.; ANCONA, D.; BORDINI, R. H.; MASCARDI, V. RV4JaCa–Runtime Verification for Multi-Agent Systems. **arXiv preprint arXiv:2207.09708**, [S.l.], 2022.
- FAKHIR, I.; KAZMI, A. R.; QASIM, A.; ISHAQ, A. Smacs: A framework for formal verification of complex adaptive systems. **Open Computer Science**, [S.l.], v.13, n.1, p.20220275, 2023.
- FERBER, J.; GASSER, L. Intelligence artificielle distribuée.(1991). **Tutorial Notes of the 11th Conference on Expert Systems and their Applications Avignon, France**, [S.l.], 1991.
- FRASHERI, M.; TRINH, L. A.; CÜRÜKLÜ, B.; EKSTRÖM, M. Failure analysis for adaptive autonomous agents using petri nets. In: FEDERATED CONFERENCE ON COMPUTER SCIENCE AND INFORMATION SYSTEMS (FEDCSIS), 2017., 2017. **Anais...** [S.l.: s.n.], 2017. p.293–297.
- FREIRE, E. S. S.; CORTÉS, M. I.; JÚNIOR, R. M. D. R.; GONÇALVES, Ê. J. T.; DE LIMA, G. A. C. NorMAS-ML: Supporting the modeling of normative multi-agent systems. **ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal**, [S.l.], v.8, n.4, p.49–81, 2019.
- GONÇALVES, E. M. N.; MACHADO, R. A.; RODRIGUES, B. C.; ADAMATTI, D. CPN4M: Testing Multi-Agent Systems under Organizational Model M oise+ Using Colored Petri Nets. **Applied Sciences**, [S.l.], v.12, n.12, p.5857, 2022.
- GONÇALVES, E. M.; QUADROS, C.; SALDANHA, J. Uma análise comparativa da especificação formal em sistemas multi-agente: os desafios e as exigências uma década depois. In: X WORKSHOP-ESCOLA DE SISTEMAS DE AGENTES, SEUS AMBIENTES E APLICAÇÕES, 2016, Maceio, Brazil. **Anais...** [S.l.: s.n.], 2016. v.1.
- HADDAD, S.; POITRENAUD, D. Theoretical aspects of recursive Petri nets. In: INTERNATIONAL CONFERENCE ON APPLICATION AND THEORY OF PETRI NETS, 1999. **Anais...** [S.l.: s.n.], 1999. p.228–247.

- HAGELBÄCK, J.; JOHANSSON, S. J. Demonstration of Multi-agent Potential Fields in Real-time Strategy Game. In: SEVENTH INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTI-AGENT SYSTEMS (AAMAS), MAY 12-16, 2008, ESTORIL, 2008. **Anais...** [S.l.: s.n.], 2008. p.1687–1688.
- HE, L.; LIU, G.; ZHOU, M. Petri-net-based model checking for privacy-critical multiagent systems. **IEEE Transactions on Computational Social Systems**, [S.l.], v.10, n.2, p.563–576, 2022.
- HERRERA, V. V.; BEPPERLING, A.; LOBOV, A.; SMIT, H.; COLOMBO, A.; LASTRA, J. M. Integration of multi-agent systems and service-oriented architecture for industrial automation. In: IEEE INTERNATIONAL CONFERENCE ON INDUSTRIAL INFORMATICS, 2008., 2008. **Anais...** [S.l.: s.n.], 2008. p.768–773.
- HOUHAMDI, Z. Multi-agent system testing: A survey. **International Journal of Advanced Computer**, [S.l.], 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.; SICHMAN, J. S.; BOISSIER, O. A model for the structural, functional, and deontic specification of organizations in multiagent systems. In: BRAZILIAN SYMPOSIUM ON ARTIFICIAL INTELLIGENCE, 2002. **Anais...** [S.l.: s.n.], 2002. p.118–128.
- HübNER, J.; SICHMAN, J. Developing organised multi-agent systems using the *Moise*⁺ model: Programming issues at the system and agent levels. In: INT. J. ACCOUNTING, AUDITING AND PERFORMANCE EVALUATION, 2007. **Anais...** [S.l.: s.n.], 2007. p.1–10.
- JENSEN, K. **Coloured Petri nets: basic concepts, analysis methods and practical use**. [S.l.]: Springer Science & Business Media, 1997. v.1.
- JENSEN, K.; KRISTENSEN, L. M. **Coloured Petri nets: modelling and validation of concurrent systems**. [S.l.]: Springer Science & Business Media, 2009.
- JENSEN, K.; KRISTENSEN, L. M.; WELLS, L. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. **International Journal on Software Tools for Technology Transfer**, [S.l.], v.9, n.3-4, p.213–254, 2007.
- KERRAOUI, S.; KISSOUM, Y.; REDJIMI, M.; SAKER, M. MATT: Multi Agents Testing Tool Based Nets within Nets. **Journal of Information and Organizational Sciences**, [S.l.], v.40, n.2, p.165–184, 2016.
- LEMAÎTRE, C.; EXCELENTE, C.; EL FALLAH-SEGHRUCHNI, A. Multi-agent organization approach to electronic business automation. In: INTERNATIONAL CONFERENCE ON DECISION SCIENCE INSTITUTE, ATHÈNES, 5., 1999. **Anais...** [S.l.: s.n.], 1999.
- MACHADO, R. A. **Testes em sistemas multiagentes sob modelo organizacional *Moise*⁺**. 2020. Dissertação (Mestrado em Ciência da Computação) — .
- MACHADO, R. A.; GONÇALVES, E. M. Testing Multiagent Systems Under Organizational Model *Moise* Using a Test Adequacy Criterion Based on State Transition Path. In: BRAZILIAN CONFERENCE ON INTELLIGENT SYSTEMS, 2020. **Anais...** [S.l.: s.n.], 2020. p.154–168.

- MACHADO, R.; ZELINDRO, A.; FARIAS, G.; ADAMATTI, D.; GONÇALVES, E. MAMTCPN: M oise+ Automated Mapping to Colored Petri Net. In: INTELLIGENT SYSTEMS CONFERENCE, 2024. **Anais...** [S.l.: s.n.], 2024. p.24–37.
- MENASSEL, Y.; MARIR, T.; MOKHATI, F.; GUPTA, V. Operational Profile-Based Test Case Generation for Normative Mas. **Available at SSRN 4853428**, [S.l.], 2024.
- MIHRET, Z.; LIU, L. Attack-driven Test Case Generation Approach using Model-checking Technique for Collaborating Systems. In: IEEE/ACM 2ND INTERNATIONAL WORKSHOP ON ENGINEERING AND CYBERSECURITY OF CRITICAL SYSTEMS (ENCYCRIS), 2021., 2021. **Anais...** [S.l.: s.n.], 2021. p.1–8.
- MURATA, T. Petri nets: Properties, analysis and applications. **Proceedings of the IEEE**, [S.l.], v.77, n.4, p.541–580, 1989.
- NASCIMENTO, N.; ALENCAR, P.; LUCENA, C.; COWAN, D. A metadata-driven approach for testing self-organizing multiagent systems. **IEEE Access**, [S.l.], v.8, p.204256–204267, 2020.
- NGUYEN, C. D.; PERINI, A.; BERNON, C.; PAVÓN, J.; THANGARAJAH, J. Testing in multi-agent systems. In: INTERNATIONAL WORKSHOP ON AGENT-ORIENTED SOFTWARE ENGINEERING, 2009. **Anais...** [S.l.: s.n.], 2009. p.180–190.
- ODEH, A.; JAYOUSI, R.; RATROT, A. Toward Competent City Management using Multi-agent System (MAS). In: INTERNATIONAL CONFERENCE ON ADVANCED TECHNOLOGIES, 2023. **Proceedings...** [S.l.: s.n.], 2023. v.11, p.177–188.
- PARASUMANNA GOKULAN, B.; SRINIVASAN, D. **An Introduction to Multi-Agent Systems**. [S.l.: s.n.], 2010. v.310, p.1–27.
- RAO, A. S.; GEORGEFF, M. P. Modeling rational agents within a BDI-architecture. **KR**, [S.l.], v.91, p.473–484, 1991.
- REHMAN, S. U.; NADEEM, A.; SINDHU, M. Towards automated testing of multi-agent systems using prometheus design models. **Int. Arab J. Inf. Technol**, [S.l.], v.16, n.1, p.54–65, 2019.
- SHARMA, A.; BAHL, S.; BAGHA, A. K.; JAVAID, M.; SHUKLA, D. K.; HALEEM, A. et al. Multi-agent system applications to fight COVID-19 pandemic. **Apollo Medicine**, [S.l.], v.17, n.5, p.41, 2020.
- SUN, J.; LIU, Y.; DONG, J. S.; PANG, J. PAT: Towards flexible verification under fairness. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED VERIFICATION, 2009. **Anais...** [S.l.: s.n.], 2009. p.709–714.
- VLASSIS, N. A concise introduction to multiagent systems and distributed artificial intelligence. **Synthesis Lectures on Artificial Intelligence and Machine Learning**, [S.l.], v.1, n.1, p.1–71, 2007.
- VOAS, J. M.; MILLER, K. W. Software testability: The new verification. **IEEE software**, [S.l.], v.12, n.3, p.17–28, 1995.
- WANG, S.; ZHU, H. CATest: a test automation framework for multi-agent systems. In: COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE (COMPSAC), 2012 IEEE 36TH ANNUAL, 2012. **Anais...** [S.l.: s.n.], 2012. p.148–157.

- WESTERGAARD, M. Access/CPN 2.0: a high-level interface to coloured petri net models. In: APPLICATIONS AND THEORY OF PETRI NETS: 32ND INTERNATIONAL CONFERENCE, PETRI NETS 2011, NEWCASTLE, UK, JUNE 20-24, 2011. PROCEEDINGS 32, 2011. **Anais...** [S.l.: s.n.], 2011. p.328–337.
- WINIKOFF, M. BDI agent testability revisited. **Autonomous Agents and Multi-Agent Systems**, [S.l.], v.31, n.5, p.1094–1132, 2017.
- WOOLDRIDGE, M. Intelligent agents. **Multiagent systems**, [S.l.], v.35, n.4, p.51, 1999.
- WOOLDRIDGE, M.; JENNINGS, N. R. Intelligent agents: Theory and practice. **The knowledge engineering review**, [S.l.], v.10, n.2, p.115–152, 1995.
- WOOLDRIDGE, M.; JENNINGS, N. R.; KINNY, D. The Gaia methodology for agent-oriented analysis and design. **Autonomous Agents and multi-agent systems**, [S.l.], v.3, n.3, p.285–312, 2000.
- ZHU, H. SLABS: A formal specification language for agent-based systems. **International Journal of Software Engineering and Knowledge Engineering**, [S.l.], v.11, n.05, p.529–558, 2001.
- ZHU, H.; HALL, P. A.; MAY, J. H. Software unit test coverage and adequacy. **Acm computing surveys (csur)**, [S.l.], v.29, n.4, p.366–427, 1997.
- ZHU, H.; HE, X. A methodology of testing high-level Petri nets. **Information and Software Technology**, [S.l.], v.44, n.8, p.473–489, 2002.

6 APÊNDICE

```

<structural-specification>
  <role-definitions>
    <role id="inspector" />
    <role id="farmer"/>
    <role id="ricep"> <extends role="farmer"/> </role>
    <role id="soyp"> <extends role="farmer"/> </role>
    <role id="animalp"> <extends role="farmer"/> </role>
  </role-definitions>
  <group-specification id="gorim" >
    <roles>
      <role id="inspector" />
      <role id="ricep" />
      <role id="soyp" />
      <role id="animalp" />
    </roles>
    <links>
      <link from="inspector" type="authority" to="farmer" scope="intra-group" />
    </links>
  </group-specification>
</structural-specification>

```

Figura 38: Código Fonte XML da EE para o cenário Inspector.

```

123@  <normative-specification>
124    <norm id = "n1" role="inspector" type="obligation" mission="m3" />
125    <norm id = "n2" role="soyp" type="obligation" mission="m4" />
126    <norm id = "n3" role="ricep" type="obligation" mission="m4"/>
127    <norm id = "n4" role="animalp" type="obligation" mission="m4"/>
128  </normative-specification>

```

Figura 39: Código Fonte XML da ED para o cenário Inspector.

```

36<functional-specification>
37<scheme id="inspSch">
38  <goal id="g17" ds="Realizar Fiscalização">
39    <plan operator="sequence">
40      <goal id="g18" ds="Contatar produtor">
41        <plan operator="sequence">
42          <goal id="g22" ds="Localizar produtor" />
43          <goal id="g23" ds="Ir na propriedade"/>
44          <goal id="g24" ds="Falar com o produtor"/>
45        </plan>
46      </goal>
47      <goal id="g19" ds="Obter Informações">
48        <plan operator="parallel">
49          <goal id="g25" ds="Visualizar propriedade" />
50          <goal id="g26" ds="Entregar informações da propriedade"/>
51        </plan>
52      </goal>
53      <goal id="g20" ds="Avaliar Informações">
54        <plan operator="sequence">
55          <goal id="g29" ds="Consultar tabela do ambiente" />
56          <goal id="g30" ds="Avaliar impacto na propriedade">
57            <plan operator="choice">
58              <goal id="g31" ds="Avaliar produção"/>
59              <goal id="g32" ds="Avaliar índice de poluição"/>
60            </plan>
61          </plan>
62        </goal>
63      </goal>
64      <goal id="g21" ds="Aplicar Sanção">
65        <plan operator="sequence">
66          <goal id="g33" ds="Definir Sanções">
67            <plan operator="choice">
68              <goal id="g34" ds="Definir valor da multa"/>
69              <goal id="g35" ds="Definir notificação"/>
70            </plan>
71          </goal>
72          <goal id="g36" ds="Comunicar Sanções">
73            <plan operator="choice">
74              <goal id="g38" ds="Aplicar multa" />
75              <goal id="g39" ds="Aplicar notificação"/>
76              <goal id="g40" ds="Não aplicar notificação"/>
77            </plan>
78          </goal>
79          <goal id="g37" ds="Cumprir Sanções">
80            <plan operator="parallel">
81              <goal id="g41" ds="Pagar multa"/>
82              <goal id="g42" ds="Confirmar recebimento" />
83            </plan>

```

```

90</mission id="m3" min="1" max="1">
91  <goal id="g17" />
92  <goal id="g18" />
93  <goal id="g19" />
94  <goal id="g20" />
95  <goal id="g21" />
96  <goal id="g22" />
97  <goal id="g23" />
98  <goal id="g24" />
99  <goal id="g25" />
100  <goal id="g29" />
101  <goal id="g30" />
102  <goal id="g31" />
103  <goal id="g32" />
104  <goal id="g33" />
105  <goal id="g34" />
106  <goal id="g35" />
107  <goal id="g36" />
108  <goal id="g38" />
109  <goal id="g39" />
110  <goal id="g40" />
111</mission>
112
113<mission id="m4" min="1" max="4">
114  <goal id="g26" />
115  <goal id="g37" />
116  <goal id="g41" />
117  <goal id="g42" />
118</mission>

```

Figura 40: Código Fonte XML da EF para o cenário Inspetor.

```

15  <structural-specification>
16
17    <role-definitions>
18      <role id="player" />
19      <role id="goalk" <extends role="player"/> </role>
20      <role id="back" <extends role="player"/> </role>
21      <role id="middle" <extends role="player"/> </role>
22      <role id="attacker" <extends role="player"/> </role>
23    </role-definitions>
24
25    <group-specification id="team" >
26      <subgroups>
27        <group-specification id="defense" >
28          <roles>
29            <role id="back"/>
30            <role id="goalk"/>
31          </roles>
32        </group-specification>
33
34        <group-specification id="attack" >
35          <roles>
36            <role id="middle"/>
37            <role id="attacker"/>
38          </roles>
39        </group-specification>
40      </subgroups>
41    </group-specification>
42
43  </structural-specification>

```

Figura 41: Código Fonte XML da EE para o cenário Soccer Team.

```

113  <normative-specification>
114
115    <norm id = "n1" role="goalk" type="permission" mission="m1" />
116    <norm id = "n2" role="back" type="permission" mission="m1" />
117    <norm id = "n3" role="back" type="obligation" mission="m2" />
118    <norm id = "n4" role="middle" type="obligation" mission="m3" />
119    <norm id = "n5" role="attacker" type="obligation" mission="m4" />
120
121  </normative-specification>

```

Figura 42: Código Fonte XML da ED para o cenário Soccer Team.

```

45<functional-specification>
46<scheme id="marcarGol" >
47<goal id="g1" ds="Gol Marcado">
48<plan operator="sequence">
49<goal id="g2" ds="Bola no meio campo">
50<plan operator="sequence">
51<goal id="g6" ds="Defesa ganha posse da bola" />
52<goal id="g9" ds="Bola passada para o meio">
53<plan operator="choice">
54<goal id="g7" ds="Passar a bola para a meia esquerda"/>
55<goal id="g8" ds="Passar a bola para a meia direita"/>
56</plan>
57</goal>
58</plan>
59</goal>
60<goal id="g3" ds="Bola no campo de ataque">
61<plan operator="sequence">
62<goal id="g11" ds="Um meia recebe a bola">
63<plan operator="choice">
64<goal id="g16" ds="Um meia recebe no lado esquerdo"/>
65<goal id="g17" ds="Um meia recebe no lado direito"/>
66</plan>
67</goal>
68<goal id="g13" ds="Um atacante está posicionado">
69<plan operator="parallel">
70<goal id="g18" ds="Um atacante se posiciona para receber a bola"/>
71<goal id="g19" ds="Um meia se prepara para passar a bola"/>
72</plan>
73</goal>
74<goal id="g14" ds="Um meia passa a bola para um atacante"/>
75</plan>
76</goal>
77<goal id="g4" ds="A bola foi chutada para o gol">
78<plan operator="choice">
79<goal id="g20" ds="Um atacante chuta a bola no lado esquerdo do gol"/>
80<goal id="g21" ds="Um atacante chuta a bola no lado direito do gol"/>
81</plan>
82</goal>
83</plan>
84</goal>
85</functional-specification>
86<mission id="m1" min="1" max="1">
87<goal id="g6"/>
88</mission>
89
90<mission id="m2" min="1" max="3">
91<goal id="g7"/>
92<goal id="g8"/>
93<goal id="g9"/>
94</mission>
95
96<mission id="m3" min="1" max="4">
97<goal id="g14"/>
98<goal id="g16"/>
99<goal id="g17"/>
100<goal id="g19"/>
101</mission>
102
103<mission id="m4" min="1" max="2">
104<goal id="g1"/>
105<goal id="g18"/>
106<goal id="g20"/>
107<goal id="g21"/>
108</mission>
109
110</scheme>
111</functional-specification>

```

Figura 43: Código Fonte XML da EF para o cenário Soccer Team.