MATE: An Automated Organization-Driven Testing Environment for JaCaMo Multi-Agent Systems

Guilherme Von Ahn Avila[1]
Ricardo Arend Machado[1]
Diana Adamatti[2]
Eder Mateus Nunes Gonçalves[1]

[1] Centro de Ciências Computacionais (C3) - Universidade Federal do Rio Grande - Rio Grande - RS - Brasil
[2]Campus Serra - Universidade Federal do Rio Grande do Sul - Caxias do Sul - RS - Brasil

guilhermevonahn37, ricardoarend, dianaada, eder.m.goncalves@gmail.com

Abstract. *Systematic validation of the organizational layer of Multi-Agent Systems (MAS) remains a largely unsolved problem. Most existing approaches focus on individual agents or pairwise communication, leaving the social level (which governs collective goals, role assignments, and mission obligations) without systematic validation support. This paper presents MATE (Multi-Agent Testing Enviroment), an automated testing environment that derives executable organizational test cases directly from $\mathcal{M}oise^+$ specifications encoded in XML format, without requiring external modeling tools such as Colored Petri Net simulators. MATE implements an all-paths traversal of the functional scheme's goal decomposition tree, augmented by a deontic variation axis that enumerates alternative role executors for missions with multiple obligated roles. The generated test cases are automatically materialized as AgentSpeak agent files and executed against a running JaCaMo instance, with pass/fail classification based on runtime observation of organizational goal states. Applied to a representative $\mathcal{M}oise^+$ scenario, MATE generated and executed 108 test cases fully automatically, reproducing the coverage predicted by the formal CPN-based method while eliminating its toolchain dependency. The generation stage completed in under 0.5 seconds, compared to the multi-step manual pipeline required by the CPN-based approach. The results exposed a systematic organizational-level fault that was not detectable from the specification alone: all configurations involving concurrent specialization failed to satisfy the root goal.*

Resumo. *A validação da camada organizacional de Sistemas Multiagentes (SMA) permanece um problema em aberto. A maioria das abordagens existentes concentra-se em agentes individuais ou em protocolos de comunicação entre pares, deixando o nível social (que governa metas coletivas, atribuições de papéis e obrigações de missões) sem suporte sistemático de verificação. Este artigo apresenta o MATE, um ambiente de teste automatizado que deriva casos de teste organizacionais executáveis diretamente a partir de especificações $\mathcal{M}oise^+$ codificadas em formato XML, sem exigir ferramentas externas de*

modelagem, como simuladores de Redes de Petri Coloridas. O MATE (Multi-Agent Testing Enviroment)implementa uma travessia all-paths da árvore de decomposição de metas do esquema funcional, complementada por um eixo de variação deôntica que enumera executores alternativos para missões com múltiplos papéis obrigados. Os casos de teste gerados são automaticamente materializados como arquivos de agentes AgentSpeak e executados contra uma instância JaCaMo em execução, com classificação de aprovação/reprovação baseada na observação em tempo de execução dos estados das metas organizacionais. Aplicado a um cenário $\mathcal{M}$oise$^+$ representativo, o MATE gerou e executou 108 casos de teste de forma totalmente automática, reproduzindo a cobertura prevista pelo método formal baseado em RPC e eliminando sua dependência de ferramentas externas. Os resultados expuseram uma falha organizacional sistemática não detectável apenas a partir da especificação: todas as configurações envolvendo especializações concorrentes falharam em satisfazer a meta raiz.

1. Introduction

Organizational-level faults in Multi-Agent Systems (MAS) can propagate across multiple agents and only become observable under particular execution configurations. Examples include a goal assigned to an inappropriate mission, a role bound to an unintended deontic relation, or a cardinality constraint that is satisfied by the specification but violated by the running system. Such faults are difficult to assess because MAS combine autonomous agents, reactive behavior, asynchronous interaction, and organizational constraints whose effects emerge from collective execution, and because existing testing approaches commonly address individual agent behavior or communication protocols rather than the organizational constraints that govern collective goals, role assignments, and mission commitments. A failure at the organizational level may thus remain invisible when agents are tested in isolation, even when every individual agent behaves according to its own plans. This motivates integrated assessment in which organizational scenarios are systematically derived and then exercised by the agents that enact them; in practice, however, manually designing such scenarios is costly, since a single organizational scheme may contain alternative decompositions, parallel activities, and multiple possible role assignments.

The need for systematic organizational testing is not new: early work identified the challenges of testing systems whose behavior emerges from autonomous components [Jennings et al. 1998], and later surveys and frameworks continued to focus largely on agent- and communication-level concerns [Houhamdi 2011], later extending to normative MAS [Menassel et al. 2024]. This motivates a more specific question addressed here: how can the organizational level be assessed systematically and automatically, rather than through manually constructed scenarios?

Organizational models provide a natural basis for addressing this question because they explicitly encode the collective structure and constraints that the running MAS is expected to follow. In the $\mathcal{M}$oise$^+$ model [Hübner et al. 2007], for example, roles, missions, goals, decomposition structures, and deontic relations are represented explicitly and can be processed from a machine- readable XML specification. This makes the organizational specification a potential source for automatically deriving test

scenarios. The contribution of MATE is to exploit this property as an automated testing workflow: instead of requiring developers to manually translate organizational alternatives into executable agent scenarios, the tool derives those scenarios from the $\mathcal{M}$oise$^+$ specification and executes them against a running JaCaMo system.

MATE (Multi-Agent Testing Environment) addresses this gap by providing an automated pipeline that derives executable organizational test cases directly from a $\mathcal{M}$oise$^+$ XML specification. The generator traverses the functional goal decomposition to enumerate execution paths and analyzes deontic constraints to introduce role-assignment variations. The resulting scenarios are materialized as AgentSpeak agents and executed against JaCaMo, where runtime organizational states are observed to classify each case as pass or fail. Thus, the approach combines specification-derived coverage with execution of the actual organizational implementation.

The paper makes three main contributions: (i) automated generation of organizational test cases directly from $\mathcal{M}$oise$^+$ XML, eliminating the need for an intermediate CPN model; (ii) automatic synthesis of the required AgentSpeak agents and execution of the suite against a running JaCaMo instance, connecting test generation to runtime validation; and (iii) an evaluation showing the approach reproduces prior CPN-based coverage while also exposing a runtime fault tied to concurrent role specializations, undetectable from the specification alone.

The remainder of this paper is organized as follows. Section 2 reviews testing approaches relevant to the organizational level, including native Jason/JaCaMo testing and CPN-based organizational testing. Section 3 describes the organizational modeling background in JaCaMo and $\mathcal{M}$oise$^+$. Section 4 details the MATE pipeline, from XML parsing to runtime validation. Section 5 presents and discusses the experimental results, including the relation between testing, formal verification, and simulation. Section 6 concludes with final considerations and directions for future work.

2. Related Work

Testing Multi-Agent Systems (MAS) is challenging because autonomy, reactivity, asynchronous interaction, and emergent behavior make it difficult to predict system behavior from individual components alone [Bordini et al. 2007]. For the organizational level targeted by MATE, two lines of work are particularly relevant: native testing support for JaCaMo agents and formal approaches for testing $\mathcal{M}$oise$^+$ organizations.

Jason and JaCaMo provide a native testing infrastructure based on Goal-Oriented Test-Driven Development (GOTDD), which allows developers to write tests in the agent language, mock artifacts, and check agent-level interactions [**?**]. GOTDD is complementary to MATE rather than a direct alternative. Its tests target individual agents, plans, and their interactions, whereas MATE derives scenarios from the organizational specification and evaluates whether the collective organizational objective is achieved. Consequently, a MATE-generated scenario could be complemented by GOTDD tests to investigate the internal behavior of the agents responsible for satisfying its organizational goals.

The work most closely related to MATE at the organizational level is the CPN-based line of research on testing the social level of MAS under the $\mathcal{M}$oise$^+$ organizational

model [Gonçalves et al. 2022]. The framework maps a $\mathcal{M}$oise$^+$ specification to CPN4M, a Colored Petri Net representation tailored to $\mathcal{M}$oise$^+$ organizations, using all-paths and state-transition-path coverage criteria. Subsequent work incorporated minimum and maximum mission cardinalities into the model [Machado et al. 2023], and the MAMTCPN tool was developed to automate parsing of $\mathcal{M}$oise$^+$ XML specifications and their mapping to CPN [Machado et al. 2024]. The resulting methodology and its formal definitions were further consolidated in [Machado 2024] and [Machado et al. 2025].

The CPN-based approach enables systematic generation of organizational test cases, but its workflow requires an intermediate CPN model and an external modeling/simulation toolchain before the generated cases can be obtained. MATE instead traverses the $\mathcal{M}$oise$^+$ specification directly and produces executable AgentSpeak scenarios without CPN construction or simulation. The two approaches share the objective of systematic organizational coverage but differ in where the generated scenarios are validated: the CPN-based method operates on an abstract formal model, whereas MATE executes them against the running JaCaMo implementation. MATE does not replace the formal guarantees of CPN-based verification; rather, it complements them by connecting specification-derived coverage to runtime execution. As shown in Section 5, MATE reproduces the case count predicted by the CPN-based method and additionally exposes a failure that appears only when multiple obligated role specializations run concurrently – motivating MATE's direct execution stage.

3. Organizational Modeling in JaCaMo

JaCaMo is a fully operational programming platform for Multi-Agent Systems that integrates four complementary dimensions of MAS development [Boissier et al. 2016]: agents, programmed in AgentSpeak using the Jason interpreter [Bordini et al. 2007]; environments, modeled as artifacts using the CArtAgO framework; interaction, through which agents communicate and coordinate using an ACL-based messaging infrastructure; and organizations, specified through the $\mathcal{M}$oise$^+$ organizational model [Hübner et al. 2007].

3.1. The $\mathcal{M}$oise$^+$ Organizational Model

The $\mathcal{M}$oise$^+$ model structures a MAS organization along three dimensions [Hübner et al. 2007]:

- **Structural Specification (SS):** defines the static structure of the organization in terms of roles, groups, and links between roles. Roles represent behavioral patterns that agents can adopt, while groups define the context in which roles are played.
- **Functional Specification (FS):** defines the dynamic aspects of the organization through schemes, missions, and goals. A scheme coordinates agents toward a collective objective by assigning missions (sets of goals) to roles.
- **Deontic Specification (DS):** defines the deontic constraints of the organization, establishing obligations and permissions that relate roles to missions within a given context.

3.2. Organizational Specifications in XML

In JaCaMo, $\mathcal{M}$oise$^+$ organizational specifications are encoded in XML format, allowing both human readability and programmatic parsing. Listing 1 illustrates a representative fragment of a $\mathcal{M}$oise$^+$ specification, showing the declaration of roles within a group, missions within a scheme, and deontic constraints.

Listing 1. Fragment of a $\mathcal{M}$oise$^+$ XML specification.

```
<!-- Example fragment -->
<structural-specification>
  <group-specification id="myGroup">
    <roles>
      <role id="roleA" min="1" max="3"/>
      <role id="roleB" min="1" max="1"/>
    </roles>
  </group-specification>
</structural-specification>

<functional-specification>
  <scheme id="myScheme">
    <goal id="g1"/>
    <mission id="m1" min="1" max="1">
      <goal id="g1"/>
    </mission>
  </scheme>
</functional-specification>

<normative-specification>
  <norm id="n1" type="obligation"
        role="roleA" mission="m1"/>
</normative-specification>
```

3.3. Organization Management Infrastructure

At runtime, JaCaMo enforces organizational constraints through the Organization Management Infrastructure (OMI), which monitors agent behavior and ensures compliance with the $\mathcal{M}$oise$^+$ specification. Agents perceive organizational facts as beliefs and act upon the organization by adopting roles (`adoptRole`), committing to missions (`commitMission`), and achieving goals, making the organizational specification directly observable and testable during execution.

4. Methods and Materials

The proposed approach automatically derives executable organizational test cases directly from a $\mathcal{M}$oise$^+$ specification, preserving its semantics while ensuring reproducibility, execution isolation, and deterministic validation of each generated scenario.

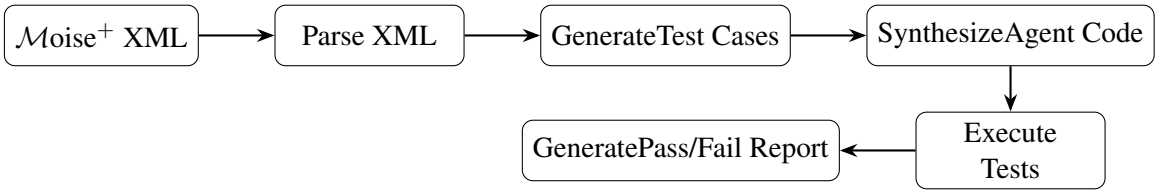

Figure 1. Overview of the MATE pipeline, from parsing the $\mathcal{M}$oise$^+$ XML specification to the structured Pass/Fail report; the Web-based orchestration layer driving these stages is described in Section 4.4.

4.1. Parsing the $\mathcal{M}$oise$^+$ Specification

The organizational model is encoded in a $\mathcal{M}$oise$^+$ XML file, which serves as the single authoritative source for the generation pipeline. A Python-based parser interprets the functional and deontic dimensions of the organization, extracting the complete goal decomposition structure (including sequential, parallel, and choice operators) as well as the missions associated with each goal and their respective role obligations.

4.2. Test Case Generation

After extracting the organizational structure, all valid execution paths of the scheme are enumerated according to the All-Paths coverage criterion. Each path represents a distinct organizational execution flow, preserving the ordering constraints imposed by the goal decomposition tree. In addition to the structural paths, the generation process also considers the dynamic permutations of role assignments associated with missions containing variable participants. The generator is designed to handle an arbitrary number of dynamic missions simultaneously: for each mission in which more than one role carries an obligation, the tool identifies the top-level goals of that mission (those that are not children of any other goal within the same mission) and assigns an independent actor to each such group. The final set of role assignment headers is obtained by computing the Cartesian product across all dynamic missions and all groups within each mission. This combination enables the derivation of a complete set of executable organizational scenarios, serialized into a structured $JSON$ file.

Each generated test case contains the ordered sequence of goals to be satisfied, the missions involved in the execution, and the roles required to achieve each organizational objective. This intermediate representation acts as a formal execution blueprint, allowing the organizational specification to be transformed into concrete executable agents.

4.3. ASL Agent Synthesis

A second Python module automatically synthesizes the corresponding $AgentSpeak$ (ASL) agent source files used by the JaCaMo platform. The synthesis stage is intentionally selective: only agents associated with roles effectively required in a given execution path are synthesized. Roles not participating in the current scenario are omitted from the execution environment to prevent interference caused by idle agents, unsolicited commitments, or unintended goal satisfaction, ensuring that each execution remains fully aligned with the organizational path represented by the test case. The generated agents are instantiated together with a predefined JaCaMo execution environment containing the organizational configuration and the base infrastructure shared across all experiments.

4.4. Web-Based Orchestration Interface

MATE exposes all pipeline stages through a lightweight web interface backed by a Python HTTP server (`server.py`). The server provides a REST API over which the browser-based frontend can: upload a $\mathcal{M}$oise$^+$ XML specification; invoke the test case generator, the ASL suite generator, and the test runner as background jobs; and receive real-time log output via Server-Sent Events (SSE). All generated artefacts are written to an isolated `Runtime/` directory. Execution parameters such as timeout values and the JaCaMo project path are externalized in a `config.json` file editable through the web interface.

4.5. Execution and Validation

For each generated test case, the Python orchestration script (`run_tests.py`) copies only the agents associated with the current scenario into the `agt` directory of the JaCaMo project before execution, preventing interference from unrelated agents.

Before launching each case, the orchestrator checks that no residual validation signal from the previous execution remains visible in the Mind Inspector API, polling until the signal disappears or a configurable guard timeout (default: 30 seconds) elapses. This step prevents a stale signal produced by the preceding run from being incorrectly attributed to the current case.

After preparing the execution environment, the orchestrator launches the JaCaMo platform and allows a configurable initialization interval (25 seconds by default) before beginning result polling. This interval gives the platform and its organizational infrastructure time to become available before the validation stage starts. The system then continuously polls the JaCaMo Mind Inspector REST API at 2-second intervals, checking whether the root goal has reached the state `goalState(gN,...,...,satisfied)`. A per-case timeout of 120 seconds is enforced after initialization; cases that do not produce the validation signal within this window are classified as *invalid*. This validation strategy focuses on the root goal because, under $\mathcal{M}$oise$^+$ semantics, it is satisfied only after all subordinate goals have been achieved.

By comparing the observed goal state with the expected outcome, the complete pipeline transforms a high-level $\mathcal{M}$oise$^+$ specification into executable JaCaMo scenarios, automatically configuring the participating agents, executing the corresponding organizational path, and classifying each case from the monitored organizational goal states.

The results of all executed cases are persisted in two complementary formats: a structured $JSON$ report (`results.json`) containing summary statistics, configuration metadata, and per-case details including the agent composition and execution duration; and a flat CSV file (`results.csv`) that provides a one-row-per-case view suitable for analysis in spreadsheet tools.

The code is available for review and reproducibility at `github.com/guivonahn/MATE`

5. Results and Discussion

The evaluation of MATE was conducted using the Inspector organizational specification [Machado et al. 2025], a scenario derived from the GORIM role-playing

game that models an agricultural inspection process, selected because it combines multiple decomposition operators, missions with variable role obligations, and an established CPN-based baseline [Machado et al. 2025]. The specification defines five roles (`inspector`, `farmer`, `ricep`, `soyp`, and `animalp`), two missions (`m3` and `m4`), and a functional scheme (`inspSch`) comprising 22 goals organized through sequence, parallel, and choice decomposition operators. The deontic specification obliges the `inspector` role to commit to `m3`, while the three specializations of the `farmer` role are each individually obliged to commit to `m4`.

The specification file used for testing is available at `github.com/guivonahn/MATE/Inspector`

5.1. Test Case Generation

MATE's parser extracted the goal decomposition tree and identified three `choice` operators within the scheme: one in goal `g30` (two alternatives), one in goal `g33` (two alternatives), and one in goal `g36` (three alternatives). Under the All-Paths coverage criterion, these operators produce $2 \times 2 \times 3 = 12$ distinct execution paths per role configuration.

Beyond the structural paths, the deontic analysis identified `m4` as a mission with multiple obligated roles (`ricep`, `soyp`, and `animalp`), triggering the role-permutation axis of the generator. Mission `m4` involves two participation phases: the *preparation* phase, in which a farmer role commits to the mission, and the *execution* phase, in which a farmer role carries out the inspection goals. Because the deontic specification obliges each of the three farmer specializations independently, either phase may be fulfilled by any of the three roles. The Cartesian product of the three obligated roles across these two phases yields $3 \times 3 = 9$ distinct role configurations. Combined with the 12 structural paths, the generator produced a total of $12 \times 9 = 108$ test cases, serialized as a structured $JSON$ file. Each test case encodes the ordered goal sequence, the responsible role per goal, and the mission binding, constituting a complete executable blueprint for the corresponding organizational scenario.

This result is consistent with those previously reported for the same specification by the CPN-based method in [Machado et al. 2025], confirming that MATE's direct XML traversal preserves the semantics of the original formal approach. The generation stage completed in under 0.5 seconds on the evaluation machine, requiring no manual CPN modeling, no external toolchain, and no state-space simulation.

5.2. ASL Agent Synthesis and Execution

From the 108 generated test cases, MATE's synthesis module produced the corresponding ASL agent files. Only roles effectively participating in a given execution path are materialized as agent files, preventing idle agents from inadvertently satisfying organizational goals or interfering with commitment resolution. For example, test cases in which `m4` is executed exclusively by `soyp` do not include `ricep.asl` or `animalp.asl` in the execution environment, ensuring that the runtime reflects only the configuration under test.

The 108 test cases were executed against a running JaCaMo instance configured with the scenario's organizational model. For each case, the orchestrator deployed

the generated agents, launched the platform, and monitored the Mind Inspector API for the validation signal `goalState(s1,g17,[ins1],[ins1],satisfied)`, indicating that the root goal of the scheme had been satisfied by the designated validation agent.

5.3. Execution Results

Of the 108 executed test cases, 33 were classified as *valid* (30.6%) and 75 as *invalid* (69.4%). The distribution of results across agent compositions is summarized in Table 1.

Table 1. Execution results by agent composition.

Agent Composition	Valid	Total
`inspector` + `soyp`	12	13
`inspector` + `animalp`	12	13
`inspector` + `ricep`	9	12
`inspector` + `ricep` + `soyp`	0	12
`inspector` + `animalp` + `soyp`	0	24
`inspector` + `animalp` + `ricep`	0	12
`inspector` + `ricep` + `soyp` + `animalp`	0	22
Total	**33**	**108**

The results reveal a clear and consistent pattern: all valid executions correspond to configurations involving exactly two agents (`inspector` and a single `farmer` specialization), while every test case involving three or more agents timed out without satisfying the root goal. Valid cases completed execution in 6 to 11 seconds, whereas invalid cases uniformly reached the 120-second per-case timeout, indicating that the failures were not marginal or timing-related, but structural.

5.4. Discussion

The predominance of failures in multi-agent configurations points to a limitation at the organizational level rather than at the agent programming level. Under the evaluated specification, `m4` carries a single obligation for each of the three `farmer` specializations. When more than one such specialization is instantiated simultaneously, the $\mathcal{M}$oise$^+$ runtime must resolve concurrent mission commitments for the same scheme. Inspection of the JaCaMo console output for representative failing cases confirmed that the platform did not raise exceptions, but that the commitment resolution for `m4` entered a state in which no agent progressed toward the scheme's goals, consistent with a deadlock in mission adoption rather than a plan-level failure. This class of fault is precisely the kind that static inspection of the specification cannot detect and that MATE is designed to expose through systematic execution: errors that do not arise from incorrect agent plans, but from mismatches between the organizational configuration and the actual agent composition at runtime.

The two isolated failures in two-agent configurations (case_5 with `soyp` and case_102 with `animalp`) also merit attention: they share the same structural path and role composition as their valid counterparts, yet failed to produce the validation signal in time. Since all other cases with identical compositions succeeded consistently, these

isolated failures are likely attributable to non-deterministic scheduling in the JaCaMo runtime rather than to organizational faults – a reminder that executing the full suite without manual intervention also surfaces intermittent failures that selective testing would miss.

Beyond the reduction in generation time and the removal of the external CPN toolchain, the systematic organizational-level fault reported above (Section 5) constitutes a concrete substantive advantage of the approach: because the CPN-based method validates test cases against a Petri-net *model* of the organization rather than against the running JaCaMo implementation, a mismatch between the deployed agent composition and the organizational specification – such as the concurrent-specialization deadlock identified here – is not something that model-level analysis alone would be expected to expose. MATE's execution-based validation against the live platform is what surfaces this class of fault, which is orthogonal to, and not merely faster than, what the CPN-based method reports.

From an engineering standpoint, the results validate the core contribution of MATE: the automated pipeline from $\mathcal{M}$oise$^+$ XML specification to executable JaCaMo test suite operated without manual intervention across all 108 cases. The generation, synthesis, execution, and reporting stages were fully automated, and the results were reproducible across repeated runs. The tool demonstrated that All-Paths coverage of an organizational scheme (including deontic variation axes) can be achieved systematically and at a scale that would be impractical through manual test design.

MATE combines specification-derived coverage with execution-based validation: its test cases are systematically generated from the organizational model and then executed against the running JaCaMo implementation. Unlike pure formal verification, it does not provide correctness guarantees over all reachable states, but it exposes runtime faults that may not appear in model-level analysis.

5.5. Threats to Validity

Several threats to the validity of the reported results should be acknowledged. With respect to *internal validity*, the validation mechanism relies on a single root-goal signal observed through the Mind Inspector API. Although this is sound under $\mathcal{M}$oise$^+$ execution semantics (the root goal is satisfied only after all subordinate goals have been achieved), intermediate goal failures masked by alternative plan resolution would not be independently detected. Additionally, the two isolated failures in two-agent configurations suggest residual sensitivity to JaCaMo's non-deterministic scheduling, which could affect reproducibility in environments with different hardware or JVM configurations.

With respect to *external validity*, the evaluation used a single organizational scenario that, while representative of prior work, may not capture the full diversity of $\mathcal{M}$oise$^+$ specifications; replication on larger specifications is a key direction for future work.

A further threat concerns the coverage represented by the 12 structural paths. Winikoff [Winikoff 2017] argues that testing BDI agents is inherently harder than testing procedural programs, since agent behavior depends on runtime-selected plan applicability and belief-revision effects not visible from the plan source code. MATE's All-Paths

criterion is defined over the *organizational* goal decomposition tree, not over each agent's internal plan library: the 12 structural paths guarantee exhaustive coverage of the scheme's sequence, choice, and parallel operators, but say nothing about the internal branching of the AgentSpeak plans that realize each goal. Two agents executing the same organizational path could still expose different behavior depending on plan selection, a variability that MATE's coverage does not, by construction, exercise. Combining MATE with agent-level approaches, such as the GOTDD framework discussed in Section 2, would be necessary for confidence across both levels, and is left as future work.

6. Final Considerations

This paper presented MATE, an automated testing environment for JaCaMo multi-agent systems that derives executable test cases directly from $\mathcal{M}$oise^{+} organizational specifications encoded in XML format. The tool addresses a practical gap in the literature: while the CPN-based method established a formal foundation for organizational test case generation [Machado et al. 2025], its dependence on an external modeling toolchain limited accessibility and scalability. MATE reproduces the same coverage semantics through direct XML traversal, eliminating the need for CPN construction or state-space simulation, and completes generation in under 0.5 seconds.

The experimental evaluation confirmed that MATE correctly derives all test cases predicted by the All-Paths criterion, including the variation axis for missions with multiple obligated roles. Beyond reproducing the formal coverage, execution of the generated suite uncovered a structurally significant fault not visible from the specification alone: configurations with a single `farmer` specialization alongside the `inspector` consistently satisfied the root goal, whereas two or more specializations failed without raising any agent-level exception. This shows that MATE's value lies not only in test case generation, but in systematic runtime observation of the full organizational scenario space.

From an engineering standpoint, the pipeline operated without manual intervention and was reproducible across all evaluated cases. These properties make MATE suitable for integration into development workflows where organizational specifications are revised iteratively and regression testing of the social layer is required.

Three primary extensions are identified for future work. First, support for additional coverage criteria (such as boundary-value analysis of mission cardinalities or pairwise role combinations) would broaden the fault-detection capability of the framework. Second, the current validation mechanism relies on a single root-goal signal; extending it to support per-goal and per-mission assertions would enable finer-grained diagnosis of execution failures. Third, applying MATE to larger and more complex $\mathcal{M}$oise^{+} specifications will be necessary to characterize the practical scalability of the XML traversal approach and to identify structural limitations of the all-paths enumeration strategy under specifications with high branching factors or deep goal trees.

Acknowledgements

The authors declare that AI-assisted tools were used during the preparation of this work. Claude (Anthropic) assisted in writing and refining portions of the manuscript text and in the development of the front-end UI component.

References

Boissier, O., Hübner, J. F., and Ricci, A. (2016). The JaCaMo framework. In Aldewereld, H., Boissier, O., Dignum, V., Noriega, P., and Padget, J., editors, *Social Coordination Frameworks for Social Technical Systems*, volume 30 of *Law, Governance and Technology Series*, pages 125–151. Springer, Cham.

Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming Multi-Agent Systems in AgentSpeak Using Jason*. Wiley.

Gonçalves, E. M. N., Machado, R. A., Rodrigues, B. C., and Adamatti, D. (2022). CPN4M: Testing multi-agent systems under organizational model $\mathcal{M}$oise$^+$ using colored Petri nets. *Applied Sciences*, 12(7):3508.

Houhamdi, Z. (2011). Multi-agent system testing: A survey. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 2(6):135–141.

Hübner, J. F., Sichman, J. S., and Boissier, O. (2007). Developing organised multiagent systems using the $\mathcal{M}$oise$^+$ model: Programming issues at the system and agent levels. *International Journal of Agent-Oriented Software Engineering (IJAOSE)*, 1(3–4):370–395.

Jennings, N. R., Sycara, K., and Wooldridge, M. (1998). A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38.

Machado, R., Zelindro, A., Farias, G., Adamatti, D., and Gonçalves, E. (2024). MAMTCPN: Moise automated mapping to colored Petri net. In *Proceedings of the Intelligent Systems Conference (IntelliSys 2024)*, volume 1067 of *Lecture Notes on Data Engineering and Communications Technologies*, pages 24–37. Springer Nature Switzerland AG.

Machado, R. A. (2024). *Testes a Nível Organizacional em Sistemas Multiagente utilizando Redes de Petri Coloridas*. Tese (Doutorado em Modelagem Computacional), Universidade Federal do Rio Grande (FURG), Rio Grande, RS, Brasil.

Machado, R. A., Adamatti, D. F., and Gonçalves, E. M. (2023). Improving the mapping of Moise to colored Petri nets. In *Anais do XVII Workshop-Escola de Sistemas de Agentes, seus Ambientes e Aplicações (WESAAC 2023)*, pages 91–100, Porto Alegre, RS, Brasil. SBC.

Machado, R. A., Cardoso, A. d. S. Z., Farias, G. P., Gonçalves, E. M. N., and Adamatti, D. F. (2025). A formal testing method for multi-agent systems using colored Petri nets. *Autonomous Agents and Multi-Agent Systems*, 39(1):10.

Menassel, Y., Marir, T., Mokhati, F., and Gupta, V. (2024). Operational profile-based test case generation for normative MAS. *SSRN Electronic Journal*.

Winikoff, M. (2017). Bdi agent testability revisited. *Autonomous Agents and Multi-Agent Systems*, 31(5):1094–1132.