A Graphical Interface for Visualising and Debugging MASPY Agents

Gabriel G. Neres[1 [0009−0008−1426−1507]], Alexandre L. L. Mellado[1 [0009−0004−1802−9546]], Rafael C. Cardoso[2 [0000−0001−6666−6954]], André P. Borges[1 [0000−0002−1716−8614]], and Gleifer V. Alves[1 [0000−0002−5937−8193]]

[1] Federal University of Technology, Ponta Grossa (UTFPR), Brazil
neres@alunos.utfpr.edu.br, mellado@alunos.utfpr.edu.br, apborges@utfpr.edu.br, gleifer@utfpr.edu.br
[2] University of Aberdeen, Aberdeen, United Kingdom
rafael.cardoso@abdn.ac.uk

Abstract. MASPY is a Python multi-agent system framework grounded in the BDI model, designed to lower the barrier to entry for developers building and experimenting with multi-agent systems. Although MASPY provides rich internal information about agent reasoning and system execution, its visualisation support is limited to a command-line interface, hindering the inspection and analysis of complex behaviours. To address this, we present MASPY-GUI, a graphical interface for visualising and debugging MASPY systems. MASPY-GUI receives execution data directly from the framework and presents it through four interactive views: a dashboard summarising system state and intentions, an agents view detailing beliefs, goals, and intention histories, an environment view visualising perceptions and their evolution over time, and a messages view showing inter-agent communication and supporting message exchange diagrams. Empirical results show that the interface scales well with increasing numbers of agents and intentions, and to a lesser extent with message volume. A user study also reports a high System Usability Scale score and positive qualitative feedback.

Keywords: MASPY · MAS Graphical User Interface · BDI Agents.

1 Introduction

Debugging agent systems is one of the most time-consuming tasks during the development of cognitive agents [1]. This difficulty arises from the inherent characteristics of multi-agent systems. Multiple autonomous agents executing simultaneously, continuously changing internal states, and emergent behaviours resulting from agent interactions contribute to the difficulty of mapping execution traces [17,20]. In Belief-Desire-Intention (BDI) agents [7,9], this challenge is compounded by the dynamic nature of beliefs, goals, and intentions, which evolve throughout execution [19,3].

Existing agent development platforms provide different levels of debugging support. Tools such as JADE's Introspector [4] and Jason's Mind Inspector [6]

offer snapshot-based inspection, but lack temporal navigation, making it difficult to trace how an agent's mental state evolves over time. They also lack comprehensive debugging support that accounts for agents, messages, and environment abstractions in a multi-agent system.

MASPY is a Python framework for developing BDI multi-agent systems, designed to facilitate MAS development for new programmers [13]. MASPY currently lacks graphical tools for visualising and debugging agent execution. In this paper, we propose MASPY-GUI, a graphical debugging interface for MASPY that provides real-time visualisation of agent beliefs, goals, intentions, message exchanges, and environment states. MASPY-GUI is designed for BDI multi-agent systems supporting the inspection of beliefs, goals, intentions, messages, and environment states. It is particularly suited for development and debugging workflows where understanding agent reasoning and system behaviour is the priority. The main contributions of MASPY-GUI are: *i.* a visualisation aid tool to guide developers; *ii.* a complete debugging support interface for MAS development; and *iii.* a timeline-based navigation system that allows moving backwards and forward through the execution history.

2 Related Work

In this section, we analyse visualisation and debugging tools for multi-agent systems, focusing on platforms that support the BDI paradigm. The development of graphical interfaces for agents remains a challenge, as debugging autonomous agents requires specialised visualisation of mental states, message exchanges, and the environment. We examine three agent platforms: JADE, SPADE, and Jason, focusing specifically on their graphical tools for runtime visualisation and debugging.

JADE (Java Agent DEvelopment Framework) is recognised as one of the most used FIPA-compliant MAS platforms [4,11]. Created as a distributed middleware in Java, JADE simplifies agent management through a container-based architecture in which a Main Container hosts FIPA services such as the Agent Management System (AMS) and the Directory Facilitator (DF) [4]. However, JADE does not natively support the BDI architecture, so extensions such as BDI4JADE are required to incorporate belief-desire-intention reasoning. As a result, JADE's visualisation and debugging tools do not reflect BDI-specific reasoning elements such as beliefs, desires, and intentions [14].

JADE's graphical interface provides debugging and management tools developed with Java Swing. Key components include the Remote Management Agent (RMA) for topological monitoring, the Introspector Agent for internal state inspection, and the Sniffer Agent, which intercepts and visualises message exchanges as UML-style sequence diagrams [4,10,18] (Fig. 1). This message visualisation feature inspired a similar capability in MASPY-GUI. However, JADE's tools lack features such as timeline navigation for reviewing past states and environment monitoring.

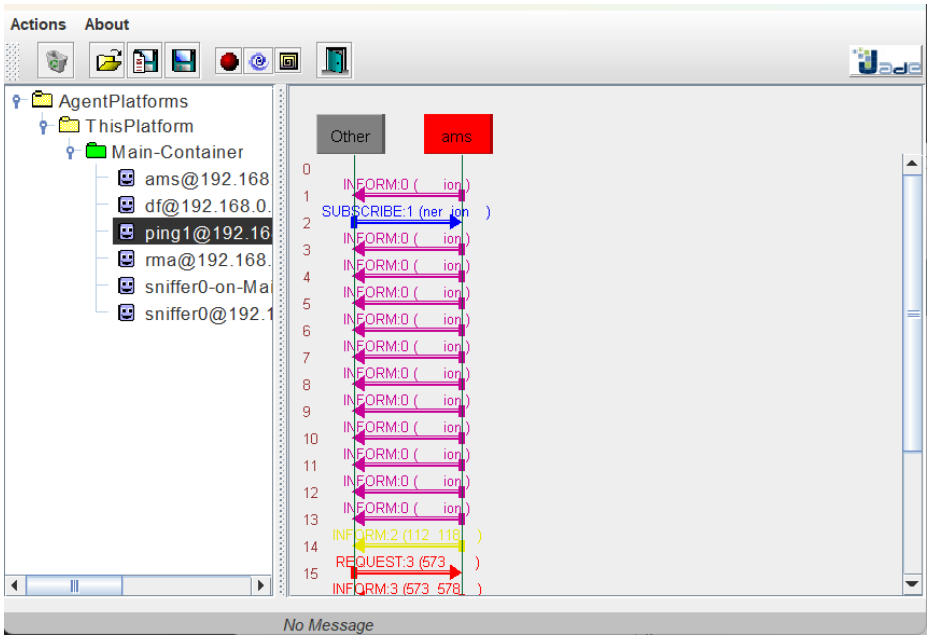

Fig. 1. JADE Sniffer Agent interface displaying message exchanges as a sequence diagram [12].

SPADE (Smart Python Agent Development Environment) is a multi-agent platform written in Python that leverages XMPP (eXtensible Messaging and Presence Protocol) for agent communication [15,18]. Unlike other platforms, SPADE uses instant messaging to deliver real-time notifications and secure messages between agents. The platform is built on an asynchronous architecture using Python's `asyncio` library and adopts a behaviour-based agent model in which agents execute predefined behaviours such as cyclic, periodic, one-shot, and finite-state machine patterns [15,10]. However, similarly to JADE, SPADE does not natively implement the BDI architecture, requiring the *spade_bdi plugin* to incorporate AgentSpeak-based reasoning. Consequently, its interface does not contain BDI reasoning elements such as beliefs, desires, intentions, and plans.

SPADE's graphical interface provides a built-in web-based dashboard accessible via any browser (Fig. 2). This interface displays each agent's list of behaviours and their contacts with their respective presence statuses. Additionally, users can inspect detailed information about each behaviour, including mailbox size, message templates, and state machine structure for FSM behaviours. The interface also provides a chat box for visualising exchanged messages and allows direct interaction with contacts [15]. However, since SPADE does not provide a native environment abstraction, the interface lacks the capability to monitor the environment. Furthermore, it does not offer timeline navigation for analysing past simulation states.

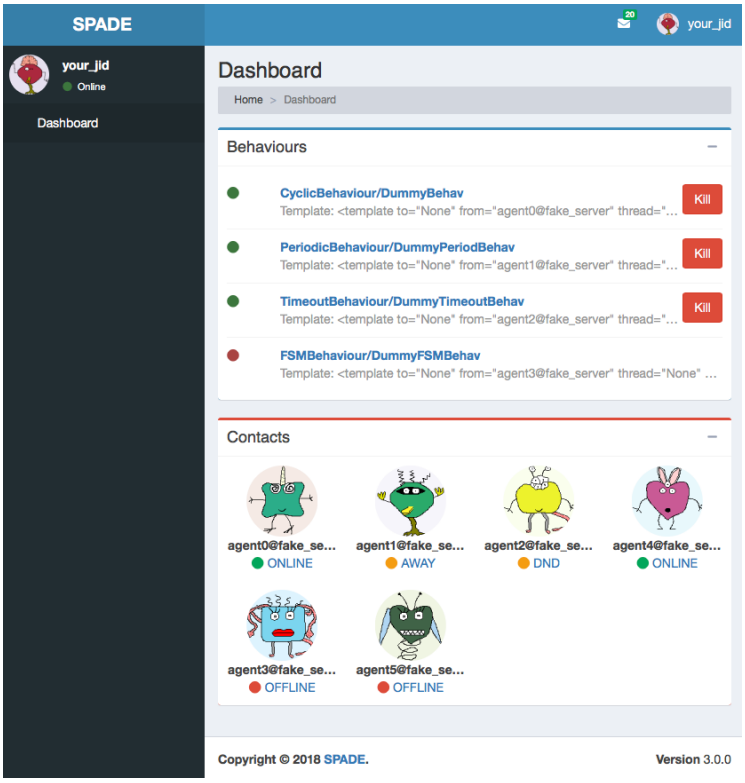

Fig. 2. SPADE web interface [16].

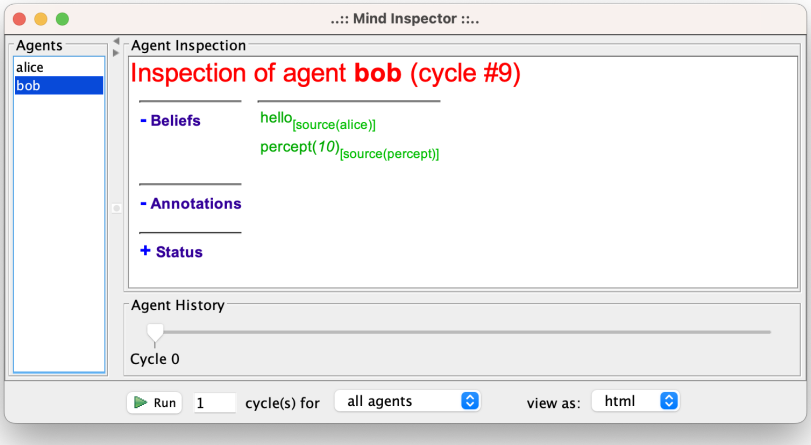

Fig. 3. Jason Mind Inspector [5].

Jason is an interpreter for an extended version of AgentSpeak, a logic-based agent-oriented programming language inspired by the BDI paradigm [6]. Jason is implemented in Java and provides a platform for developing MAS based on reactive planning. Unlike JADE and SPADE, Jason natively implements the BDI architecture, where agents are defined in terms of beliefs, goals, and plans written in AgentSpeak. The platform supports speech-act-based inter-agent communication, customisable selection functions, and distributed network operation.

Jason's graphical interface provides the Mind Inspector debugging tool, accessible via the Jason console (Fig. 3). This tool allows developers to inspect an agent's internal states during execution, displaying the belief base, plan library, event set, and current intentions [6]. It operates in debugging mode, where execution can be paused to examine the mental state of each agent in the system. However, the *Mind Inspector* does not provide environment monitoring, timeline navigation, or message-exchange diagrams between agents.

Based on analyses of the three platforms and comparative studies of MAS tools [11], Table 1 presents a feature comparison of graphical interface capabilities, including general platform characteristics, agent inspection features, system monitoring capabilities, and debugging functionalities. Unlike JADE and SPADE, which support distributed execution across multiple machines, MASPY's current version requires that all agents run on a single machine. The major points of Table 1 that make MASPY-GUI different from other approaches are the *Timeline Navigation*, which enables analysis of the simulation at different points in time, and the *Environment Monitoring*, which allows observation of the environment state and percepts.

Table 1. Comparison of graphical interface features for agents

Feature	**JADE**	**SPADE**	**Jason**	**MASPY-GUI**
Programming Language	Java	Python	Java	Python
Native BDI Support	No	No	Yes	Yes
Distributed Execution	Yes	Yes	Yes	No
GUI Technology	Swing	Web	Web	PyQt5
Agent Inspection				
Belief Visualisation	No	No	Yes	Yes
Goal Visualisation	No	No	Yes	Yes
Intention Visualisation	No	No	Yes	Yes
Plan Monitoring	Yes	Yes	Yes	Yes
System Monitoring				
Message Visualisation	Yes	Yes	No	Yes
Environment Monitoring	No	No	No	Yes
Debugging Capabilities				
Timeline Navigation	No	No	No	Yes
History Tracking	No	No	No	Yes
Pause/Resume Execution	Yes	No	Yes	Yes
Diagram generation	Yes	Yes	No	Yes

3 MASPY-GUI Process

This section describes the MASPY-GUI process, presenting an overview of the MASPY framework, the details of the MASPY-GUI process, the data management structure, and the timeline feature.

3.1 MASPY

MASPY [13] is a Python framework for developing multi-agent systems based on the BDI paradigm, designed to facilitate the development of MAS, particularly for new programmers. The framework can be used to model agents and the environments where they act. The framework is structured with five main classes. The *Agent* class manages the agent's beliefs, goals, and plans, implementing the BDI reasoning cycle where agents perceive the environment, process received messages, send messages, and decide which plan to execute. The *Environment* class models entities that agents can perceive and interact with through percepts and actions. The *Channel* class handles message exchange between agents and supports directives such as *tell*, *achieve*, and *askOne*. The *Admin* class provides configuration and control of the system, including agent registration, connection management, and system startup. Finally, the *Learning* class enables integration with machine learning techniques. Note that support for learning aspects in MASPY-GUI is future work.

A key feature of MASPY is the logging system available through the Admin class. When enabled, the framework generates detailed logs of agent activities, including belief updates, goal adoption, plan execution, message exchanges, and interactions with the environment. These logs provide the foundation for the graphical interface presented in this paper, enabling runtime monitoring and execution analysis of the multi-agent system.

3.2 MASPY-GUI

MASPY-GUI[3] is designed as a complementary layer of the MASPY framework, providing real-time visualisation and debugging capabilities without requiring significant modifications to existing MASPY applications. The graphical interface can be enabled with simple configuration adjustments. Once activated, the interface runs as a separate process alongside MASPY, capturing and displaying system events in real time.

Creating a separate process is necessary to avoid interfering with the main execution. MASPY's main process generates logs and sends them to the GUI process via a shared queue. In the GUI process, a `QueueListener` receives and passes them to the `LogStore`, the class responsible for reading all logs, extracting all information, classifying them by type, and finally sending them to the graphical interface pages.

[3] Available at: https://github.com/laca-is/MASPY-GUI (Last accessed 2025/02/22)

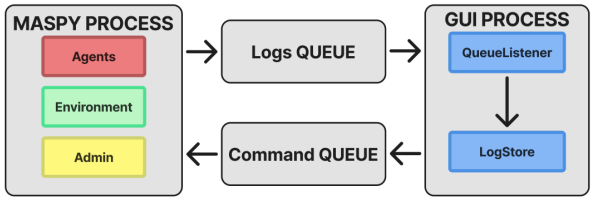

Fig. 4. MASPY-GUI overview diagram.

Fig 4 provides an overview of the communication between the two processes and the two existing queues. The separation into two distinct processes means that if the interface freezes, the simulation can continue running without interruption. It also reduces the GUI's impact on execution, since the interface does not close abruptly when the MASPY code finishes. Instead, the user can inspect the system state and debug or analyse the corresponding code more carefully.

There are two communication queues between the processes. The Logs Queue is where all information about what is happening in the system will be. The Command Queue currently supports the pause command, which can be used via a button in the interface menu at any time during the simulation. This is a MASPY resource that, when called, stops log generation and freezes the system.

MASPY generates three types of logs during system execution, depending on the active components: Agent Log, Message Log, and Environment Log. The GUI receives these logs through a message queue and parses them into typed objects. Additionally, to facilitate the process, the GUI extracts agents' intentions from the Agent Log and creates a separate Intention Log. These four log categories are sent to their respective interface views and are accessible through the graphical user interface menu.

3.3 Data Management and Timeline

The GUI's central component and data hub is the `LogStore`. It receives raw data, analyses and classifies it, distributes it to the appropriate page logs, and stores it for later use. An overview of the internal process of MASPY-GUI is presented in Fig. 5. There are four types of logs: Agent, Message, Intention and Environment.

- *AgentLog:* Records the complete state of an agent at each cycle, including: cycle number, beliefs, goals, running intentions, connected environments and channels, perceptions, and the state of the agent, for example, if it is idle or executing a plan.
- *MessageLog:* Register the communication between the agents, storing: the sender, the receiver, the performative used (example *tell* for beliefs or *achieve* for goals), and the message content.
- *IntentionLog:* Created by the GUI from AgentLog data. It records when an agent changes their intention, storing the agent's name, the intention

description, the triggering event, and the cycle number. This separate log type is necessary because intentions are displayed not only on the agent page but also on the menu page.
– *EnvironmentLog:* Records changes in the environment's percepts, including: the environment name, the type of change (CREATE, DELETE, or CHANGE), and the percept content that was modified.

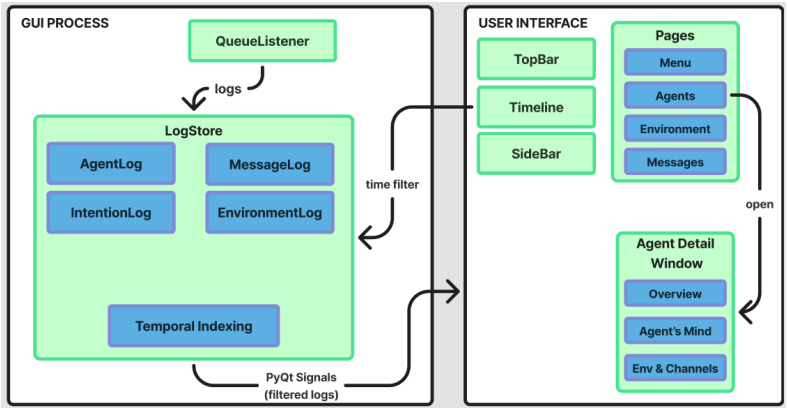

Fig. 5. MASPY-GUI internal diagram.

To ensure efficient performance when handling large volumes of data, the `LogStore` employs two optimisation strategies: memory reduction and the use of efficient data structures. For memory optimisation, all log classes use Python's `__slots__`[4] mechanism. For time efficiency, the implementation uses specific data structures to ensure optimal performance. New logs are stored in a `deque` (double-ended queue), allowing O(1) insertion time. Logs are indexed by agent and environment using `defaultdict` (hash table), providing O(1) access time.

In Fig. 5, it is important to note the *Timeline* widget in the user interface. This component is located at the top of all pages and allows users to navigate through the simulation timeline. The developer can see how much time has passed and control it to return to previous moments, enabling analysis of different simulation states over a specific period.

This is an important resource because it provides a debugging tool that covers the entire simulation history, allowing users to return to the beginning and replay it at speeds from 0.25x to 8x. Since the GUI continues to run after the MASPY simulation ends, it is particularly valuable to re-examine specific moments along the simulation timeline.

The Timeline operates in two modes: *Live mode* and *Historical mode*. In Live mode, the interface displays data in real time as new logs arrive from

[4] `__slots__` is a Python feature that restricts the attributes of a class to a predefined set, reducing memory overhead by avoiding the creation of a per-instance `__dict__`.

MASPY, and the slider automatically follows the latest events. When the user interacts with the Timeline—by dragging the slider backwards or clicking navigation buttons—the interface switches to Historical mode, displaying the system state at the selected point in time. It is important to note that while in Historical mode, new logs continue to be received and stored normally.

Internally, the `LogStore` maintains a `deque`, `timeline_ms_map`, that stores the timestamp of each log in milliseconds (ms) in chronological order. When the user moves the slider to a specific time, the Timeline emits a signal containing the selected timestamp. The `LogStore` then performs a binary search using `bisect.bisect_right(timestamps, target_ms)` to find the corresponding index in $O(logn)$ time.

Once the correct index is determined, the `LogStore` notifies all pages via PyQt signals, prompting them to synchronise their content with the selected moment in the simulation. The Timeline widget provides the user with several control options: buttons to play/pause playback, buttons to step forward or backward (100ms per step), buttons to jump to the beginning or end of the simulation, and a speed selector to control playback speed.

4 User Interface

This section describes the interface pages, outlining their structure and explaining each page.

4.1 Navigation Structure

The GUI includes a left sidebar that lists available pages, allowing the user to switch between them. On the right side, there are three horizontal sections: the top bar, which contains the settings menu and a guide explaining the interface and timeline; the timeline, which allows navigation through the simulation time; and, finally, the page content area, displaying the currently selected page.

The navigation system uses a `QButtonGroup` in the sidebar, with only one button active at a time. When a page is selected, the top bar title updates accordingly, and the active button is highlighted with a different colour to indicate the current page. In the top bar, the guide is composed of three different sections: the first provides a brief explanation about the GUI, describing the composition of each part, the second is a more detailed guide about the interface, explaining each page and what information can be found there, and the third section provides MASPY and MAS terminology explanations.

4.2 Interface Pages

The GUI provides four main pages, each focused on a specific aspect of the MAS. All pages respond to the global timeline, updating their content according to the selected simulation time.

Menu: The first screen that appears when the interface is opened is the Menu Page, which displays a dashboard of the simulation containing the number of agents, messages, intentions, and environments. Below the dashboard, you can see all the intentions that occurred in the system. This screen includes a help button that lets the user read a brief summary of the intention report to better understand it. The intention report may contain a large number of entries, so it is paginated and includes navigation buttons for the next and previous page

Agents: The Agents Page, Fig. 6, displays all agents in the system as a grid of cards. Each card shows the agent's name, current cycle, number of beliefs and goals, and the current action being executed. A search bar allows filtering agents by name. Selecting "View Details" opens the Agent Detail Window, which provides more detailed information organised in four tabs: (i) Overview, showing the real time status with running intentions, current beliefs, and goals; (ii) Agent's Mind, showing a history of all beliefs, goals, and intentions that changed during execution; (iii) Environment and Channels, listing the environments and communication channels the agent is connected to, along with the current percepts.

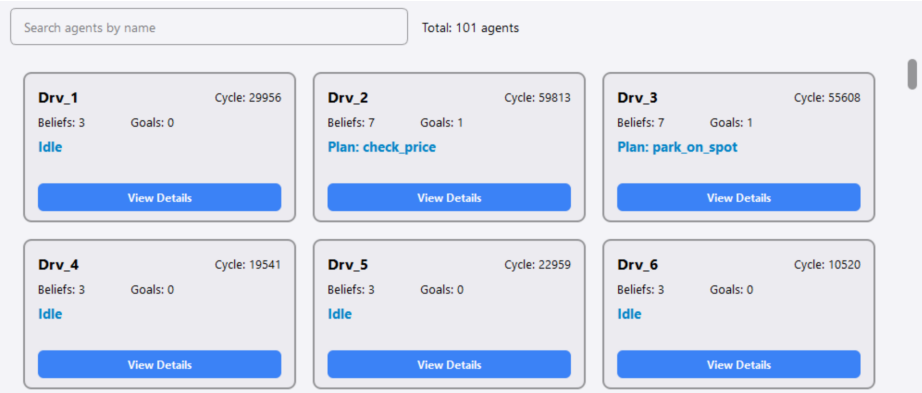

Fig. 6. Agent's page containing information about each agent.

Environment: The Environment Page allows monitoring MASPY environments. MASPY can support multiple agent environments, allowing different groups of agents to operate in separate contexts, each with its own state, rules, and interactions. A list on the left shows all available environments, and selecting one displays its details on the right side. The details section contains three components: Connected Agents, showing which agents are currently linked to the environment at that specific moment; the percepts, displaying a drop-down table with all percepts that exist in the system; (Fig. 7) and Changes History, a

chronological log of all modifications in the perception system, including CREATE, DELETE, or CHANGE (Fig. 8).

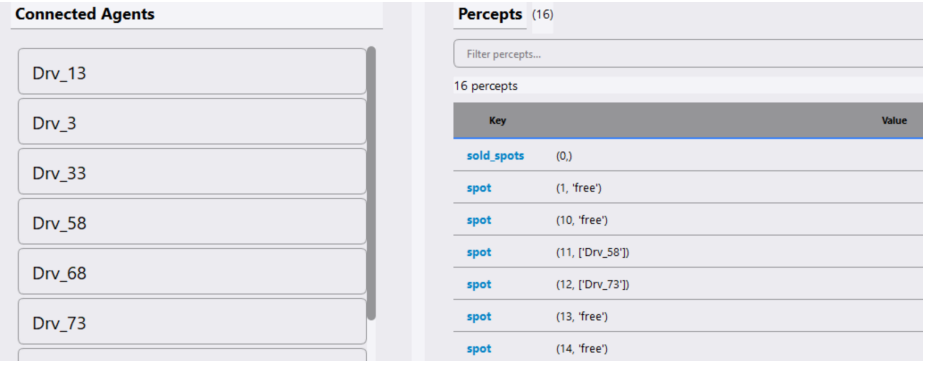

Fig. 7. Part of Environment's page containing information about the percept

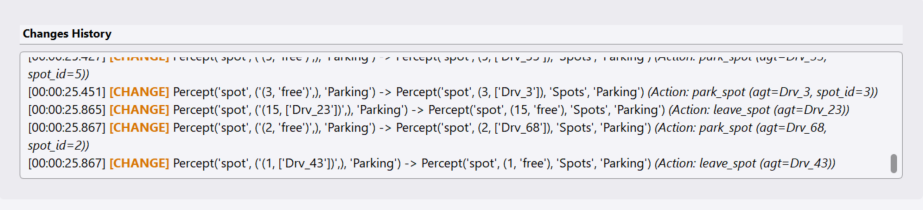

Fig. 8. Part of Environment's page containing the history of the percepts

Messages: provides visualisation of all messages exchanged between agents. On the left side, a panel lists agents, allowing users to filter messages by selecting a specific agent. The main area displays the filtered messages as cards, each showing the sender, receiver, timestamp, performative, and content.

A key feature is the Message Sequence Diagram Generator, which creates UML-style sequence diagrams of agent communication, as shown in Fig 9. Users can select which agents to include, between two or all agents, and the tool generates a visual representation of the message over the simulation, with arrows indicating direction and showing the performative and content of the message. The diagrams can be exported as PNG or SVG for documentation purposes.

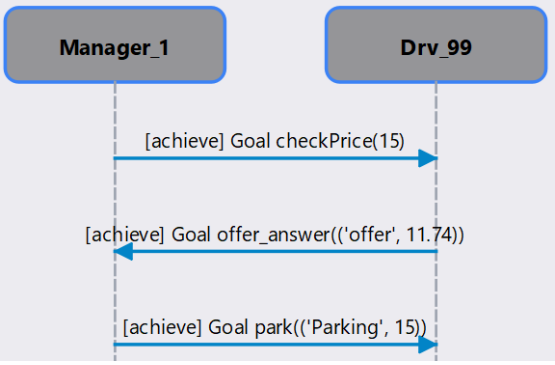

Fig. 9. Message's page showing a sequence diagram.

5 Experimental Validation

This section presents the validation of MASPY-GUI. First, we present a technical performance benchmark to assess system scalability. Second, we present the results of a user study evaluating usability and interaction quality.

5.1 Performance Evaluation

The goal of this evaluation is to test the interface under critical conditions, identify the GUI's capabilities, and measure RAM consumption across different scenarios. The experiments were conducted on a machine equipped with an Intel Core i7-1165G7 (4 cores, 2.80 GHz), 8GB of RAM, running Windows 11.

To conduct these tests, we simulated 14 different scenarios. For greater precision, we recorded the average from ten executions of each scenario to obtain more accurate metrics. To measure the GUI's efficiency, we recorded the average RAM usage during the experiments and the time required to complete each execution.

In Table 2, our experiments are categorised into two types: the first type consists only of intentions and zero messages, and the other consists of combined messages and intentions. The experiment's settings include three variables: the number of agents, the number of messages, and the number of intentions per simulation. It is important to note that these are the totals per agent. For example, in experiment number eight, there are 10 agents with 10 messages and 10 intentions each, totalling 100 messages and 100 intentions.

The last three columns contain the results. RAM is the experiment's average peak RAM, not the average RAM consumed during the test, but the highest critical moment. The same applies to CPU cache, but for CPU cache use. The last column shows the average execution time.

Based on the results, experiments 3 and 5 are interesting to analyse because both have 10,000 intentions, and the difference between them is the number of agents (1,000 and 10). Looking at the results, CPU cache usage is 4 times higher,

Table 2. Performance experiments settings and results. Containing the number of agents, messages, and intentions in the simulation. The results are shown in terms of RAM and CPU consumption, and the duration of the simulation.

Id	# agents	# msg	# int	RAM (Mb)	CPU cache (Mb)	Time
1	10	0	100	182.11	22.09	4.1s
2	100	0	1000	195.12	119.88	4.7s
3	1000	0	10000	362.16	532.84	5.0min
4	10	0	1000	188.77	140.16	5.1s
5	10	0	10000	228.16	164.25	21.5s
6	10	0	100000	630.11	175.13	2.9min
7	10	0	1000000	3913.43	187.47	27.4min
8	10	100	100	181.55	47.11	4.1s
9	100	1000	1000	221.46	191.15	10.2s
10	1000	10000	10000	457.32	692.55	9.3min
11	10	1000	1000	215.75	165.34	7.8s
12	10	10000	10000	545.51	170.38	2.1min
13	10	100000	100000	2472.01	166.82	2.71h
14	100	100000	100000	2189.67	244.19	2.44h

and execution time is 14 times longer. The conclusion we can draw from this is that, for MASPY-GUI, the number of agents is the most resource-intensive factor of our interface.

Fig. 10 shows the impact of varying the number of agents on RAM and execution time, comparing scenarios with intentions only against combined scenarios (messages and intentions). When examining RAM consumption, the difference between the two test types remains relatively small across all agent counts, with the combined scenarios showing only a slight overhead. However, the execution time shows a different result: while both test types perform similarly with 10 and 100 agents, the gap becomes substantial with 1000 agents, where the combined scenario takes 558 seconds compared to 300 seconds for intentions only.

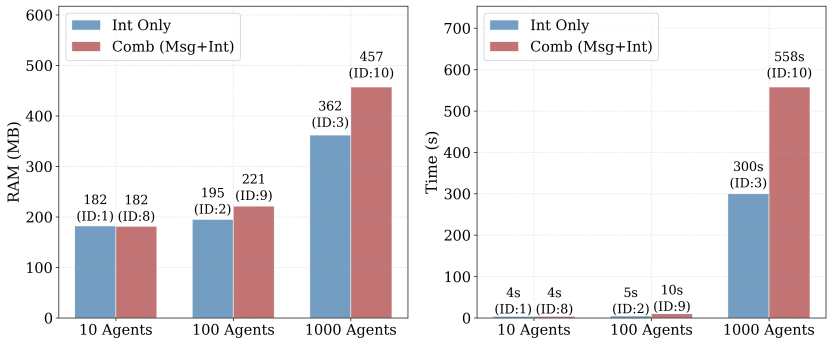

Fig. 10. RAM and execution time comparison when increasing the number of agents.

The other three experiments that call our attention are experiments 7, 13, and 14. These three represent the most critical conditions of each test type. Experiment 7 has 1 million intentions, and what stands out most is the RAM and CPU usage, which are by far the highest in our table. On the other hand, the time was technically accessible when compared to experiments 13 and 14.

Fig. 11 shows the scalability analysis as the number of items (intentions and messages) increases, while the number of agents remains fixed at 10. The RAM consumption shows a growth pattern for both test types, with the combined scenarios requiring more memory due to the additional message storage. The most striking observation is in execution time: at 100k of each item, the combined scenario requires 9756 seconds (approximately 2.7 hours), while the intentions-only scenario completes in 174 seconds.

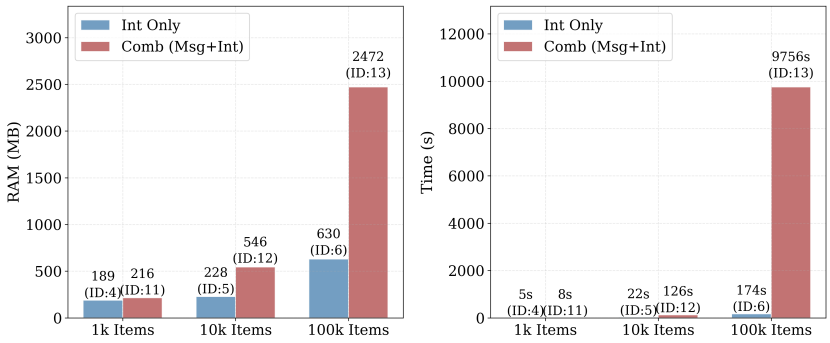

Fig. 11. RAM and execution time scalability when increasing the number of items.

Experiments 13 and 14 also stand out because their execution times are much higher than the others: the third-longest execution time (test 7) is 27 minutes, while these took 2.71 and 2.44 hours. The difference between them and test 7 is that we added messages to the system. It is important to compare them with other combined experiments, showing that when both variables are present, the time will not necessarily deviate from the pattern.

5.2 User Study

To obtain user validation of our interface, we conducted two studies in which users used the GUI and then evaluated it. For both studies, we created a form divided into six steps. The first is a questionnaire about the participant's background: their academic degree, Python knowledge, and whether they have used MASPY or MASPY-GUI before. The second, third, and fourth steps constitute the actual experiment; in each session, we provided code with the interface already working and gave the user three or four tasks to perform. The last two steps constitute the evaluation phase: the fifth step uses the System Usability

Scale (SUS) [8], and the final step consists of open-ended questions to gather user feedback on their experience with MASPY-GUI.

The first study was conducted with students from the Federal University of Technology-Paraná (UTFPR) in Brazil, while the second study was conducted with students from the University of Aberdeen in the United Kingdom.

In the UTFPR experiment, we received 12 answers from participants in Computer Science, Systems Analysis, and Master's degrees in Computer Science. Most of them have intermediate knowledge of Python, while their MAS knowledge is split between intermediate and basic, nine of them had used MASPY before and three of them had used the MASPY-GUI before.

The SUS scale consists of 10 questions, each scored on a 1–5 scale. After receiving the answers, we performed a simple calculation: for the odd questions, we subtracted 1 from the score; for the even questions, we subtracted the score from 5. After this, we sum the results and multiply the sum by 2.5 to obtain the final SUS score. Doing this for our questions, we have:

$$SUS = 2.5(x + y) \tag{1}$$

$$SUS = 2.5(17.67 + 17.67) = 2.5(35.34) = 88.35 \tag{2}$$

The results indicate an A+ SUS grade in this first user study. In the final step, we conducted an open questionnaire about our interface, including both positive and negative points, what users wanted to change, and features they felt were missing. Analysing the answers we received, we identified two major negative points: the timeline and the visualisation of perceptions on the message and agents page. Following this review, we reformulated the pages and made the following changes: in the timeline, we added the control buttons and the playback speed controller; in the perception list, we made changes from a single line containing all perceptions (which was difficult to read when many perceptions were present) to the drop-down list that we have now.

The most common comment was about the poor visibility of perceptions. Other changes were made based on user feedback. For example, the interface guide and help buttons, which did not exist before. The users felt lost while using the interface or did not know where to find specific information. Other micro-changes occurred after the user study, demonstrating the benefits of collecting this type of feedback.

From the University of Aberdeen user study, we received 16 responses. A notable difference from the UTFPR study was the participants' level of MAS knowledge. This study was conducted with students at an earlier stage of their MAS studies: 10 reported having basic knowledge, 4 had none, and only 2 had intermediate knowledge. The major difference is that only one participant had used MASPY before, and none had used MASPY-GUI previously. Using the same SUS calculation formula:

$$SUS = 2.5(x + y) \tag{3}$$

$$SUS = 2.5(15.5 + 13.94) = 2.5(29.44) = 73.6 \tag{4}$$

The B- SUS score of 73.6 is still considered high. The negative feedback from Aberdeen participants was similar to that in the UTFPR study, reinforcing the need for the previously described interface improvements.

Considering both studies together, the weighted average SUS score across all 28 participants is calculated as follows:

$$SUS_{combined} = \frac{(n_1 \times SUS_1) + (n_2 \times SUS_2)}{n_1 + n_2} \tag{5}$$

$$SUS_{combined} = \frac{(12 \times 88.35) + (16 \times 73.6)}{12 + 16} = \frac{1060.2 + 1177.6}{28} = 79.92 \tag{6}$$

The combined weighted average SUS score is 79.92, corresponding to a grade B, which is still considered a good usability result.

6 Conclusion

The MASPY-GUI interface provides real-time visualisation of agent beliefs, goals, intentions, message exchanges, and environment states, including a history of all of them for the MASPY framework. Its main feature is a timeline-based navigation system that allows developers to move backwards and forward through the execution history.

The evaluation was conducted in two phases. Performance tests demonstrated that the interface can handle systems with up to 1000 agents and 100000 messages and intentions while maintaining acceptable RAM consumption. User studies conducted at UTFPR (Brazil) and the University of Aberdeen (UK) resulted in SUS scores of 88.35 and 73.6, respectively, indicating excellent usability.

As future work, we plan to add support for monitoring the learning class, enabling the visualisation of reinforcement learning processes within MASPY agents. Additionally, we intend to implement a mechanism that allows the user not only to visualise the system but also to interact with it, changing the agent's beliefs, goals, and plans, similar to the proposal of JaCaMo-Web [2].

We also plan further optimisations to both MASPY and MASPY-GUI to improve interface performance and support larger numbers of agents, messages, and intentions. To improve usability when handling large-scale systems, we intend to introduce advanced filtering. Future versions of MASPY are expected to support distributed agents, which could significantly improve MASPY's performance but may also introduce new bottlenecks in the centralised graphical interface.

Acknowledgments. This work is supported in part by CNPq/MCTI/FNDCT N° 22/2024 grant number 444568/2024-7, "Engineering Neuro-Symbolic Agents".

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Ahlbrecht, T.: An algorithmic debugging approach for belief-desire-intention agents. Annals of Mathematics and Artificial Intelligence **92**(4), 797–814 (May 2023). https://doi.org/10.1007/s10472-023-09843-4
2. Amaral, C.J., Hübner, J.F.: Jacamo-web is on the fly: An interactive multi-agent system IDE. In: Engineering Multi-Agent Systems - 7th International Workshop, EMAS 2019, Montreal, QC, Canada, May 13-14, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12058, pp. 246–255. Springer (2019). https://doi.org/10.1007/978-3-030-51417-4_13
3. Amaral, C.J., Hübner, J.F., Kampik, T.: TDD for AOP: test-driven development for agent-oriented programming. In: Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2023, London, United Kingdom, 29 May 2023 - 2 June 2023. pp. 3038–3040. ACM (2023). https://doi.org/10.5555/3545946.3599165
4. Bellifemine, F., Poggi, A., Rimassa, G.: Developing multi-agent systems with JADE. In: Proceedings of the 7th International Workshop on Intelligent Agents VII. Agent Theories Architectures and Languages. p. 89–103. ATAL '00, Springer-Verlag, Berlin, Heidelberg (2000). https://doi.org/10.1007/3-540-44631-1_7
5. Bordini, R.H., Hübner, J.F.: Jason documentation: Getting started. http://jason-lang.github.io/jason/tutorials/getting-started/readme.html (2024), last accessed 2025/02/22
6. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology). John Wiley & Sons, Inc., Hoboken, NJ, USA (2007)
7. Bratman, M.E., Israel, D.J., Pollack, M.E.: Plans and resource-bounded practical reasoning. Computational Intelligence **4**(4), 349–355 (1988). https://doi.org/10.1111/j.1467-8640.1988.tb00284.x
8. Brooke, J.: Sus: A quick and dirty usability scale. Usability Eval. Ind. **189** (11 1995)
9. Cardoso, R.C., Ferrando, A.: A review of agent-based programming for multi-agent systems. Computers **10**(2), 16 (2021). https://doi.org/10.3390/computers10020016
10. Donâncio, H., Casals, A., Brandão, A.A.F.: Exposing agents as web services: a case study using JADE and SPADE. In: Proceedings of the 13th Workshop-Escola de Sistemas de Agentes, seus Ambientes e apliCações (WESAAC). Florianópolis, Brazil (may 2019)
11. Kravari, K., Bassiliades, N.: A survey of agent platforms. Journal of Artificial Societies and Social Simulation **18** (01 2015). https://doi.org/10.18564/jasss.2661
12. Lab, T.I.: Spade documentation: Web graphical interface. https://jade.tilab.com/documentation/tutorials-guides/ (2024), last accessed 2025/02/22
13. Mellado, A.L.L., Pinz Borges, A., Alves, G.V.: Maspy: A python-based framework for developing bdi multi-agent systems. In: Advances in Practical Applications of Agents, Multi-Agent Systems, and Computational Social Science: The PAAMS Collection. pp. 216–227. Springer Nature Switzerland, Cham (2026). https://doi.org/10.1007/978-3-032-07638-0_18
14. Nunes, I., Lucena, C., Luck, M.: BDI4JADE: a BDI layer on top of JADE. In: Proc. of the 9th WS on Programming Multiagent Systems. pp. 88–103 (2011)
15. Palanca, J., Terrasa, A., Julián, V., Carrascosa, C.: Spade 3: Supporting the new generation of multi-agent systems. IEEE Access **8**, 182537–182549 (01 2020). https://doi.org/10.1109/ACCESS.2020.3027357

16. Palanca, J.: Spade documentation: Web graphical interface. https://spade-mas.readthedocs.io/en/latest/web.html (2024), last accessed 2025/02/22
17. Poutakidis, D., Padgham, L., Winikoff, M.: Debugging multi-agent systems using design artifacts: the case of interaction protocols. In: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2. p. 960–967. AAMAS '02, Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/544862.544966
18. Radhakrishnan, G., K L, S.: Comparative study of JADE and SPADE multi agent system. International Journal of Advanced Research **6**, 1035–1042 (10 2018). https://doi.org/10.21474/IJAR01/8090
19. Winikoff, M.: BDI agent testability revisited. Auton. Agents Multi Agent Syst. **31**(5), 1094–1132 (2017). https://doi.org/10.1007/S10458-016-9356-2
20. Wooldridge, M.: An Introduction to MultiAgent Systems. Wiley Publishing, 2nd edn. (2009)