On the Usability and Extensibility of VITAMIN

Angelo Ferrando[1][0000−0002−8711−4670] and Vadim Malvone[2][0000−0001−6138−4229]

[1] University of Modena and Reggio Emilia, Italy
angelo.ferrando@unimore.it
[2] LTCI, Télécom Paris, Institut Polytechnique de Paris, France
vadim.malvone@telecom-paris.fr

Abstract. The verification of Multi-Agent Systems (MAS) presents a significant challenge, with various approaches and methodologies available to address it. However, tools supporting these methodologies are not always readily accessible. Existing tools often lack compositionality, are hard-coded, and have a steep learning curve. In this paper, we present VITAMIN, a modular and versatile tool designed for the formal verification of MAS. Unlike existing tools, VITAMIN offers high usability and can be easily extended to support various models and logics for MAS.

Keywords: Formal Verification · Model Checking · Multi-Agent Systems

1 Introduction

Verifying software and hardware systems is challenging due to their complexity and size, which often require exhaustive verification impractical without optimisation. This complexity demands deep expertise in formal methods, making formal verification techniques less usable in real-world development. One of the most representative approaches to obtain an exhaustive formal verification is Model Checking (MC) [2]. Transitioning from monolithic systems to Multi-Agent Systems (MAS) exacerbates these challenges. MAS verification involves testing [18], debugging [20], and verifying [10], all of which are complex tasks. MASs, as distributed systems with intelligent agents, face similar verification challenges but with added complexity due to agents' rationality and interactions.

Model Checking for MAS relies heavily on tools supporting these techniques, such as MCMAS [17] and STV [16][3]. MCMAS is widely used for strategic verification of MAS due to its early development and foundational role in research. However, it exhibits issues hindering broader adoption, including a hard-coded verification process and lack of modularity for clear separation between different logics and models. Despite various extensions, MCMAS lacks transparent extension capabilities for new logics and models, causing maintainability issues. Additionally, its execution can be challenging as it requires additional tools like Eclipse for installation, and lacks comprehensive external documentation. These

[3] Note that, we omit the discussion of other works on the formal verification of MAS, like MCAPL [9], as our analysis is centred on the strategic reasoning on MAS.

limitations stem from its primary function as a research tool. STV is designed for specific verification goals, lacking compositional nature and flexibility to support different logics or models for MAS verification. It also lacks comprehensive documentation to assist users and developers. Both MCMAS and STV require a strong background in formal methods, making them challenging for non-expert users. In summary, while widely used, both tools lack modularity and usability.

2 Main Purpose

We tackle the above mentioned challenges by developing VITAMIN (VerIficaTion of A MultI-ageNt system) [12,13,14], a formal verification framework for MAS. VITAMIN aims to be highly compositional, supporting various logic and model formalisms, while also providing a user-friendly experience for both developers and end-users. VITAMIN aims to generalise MAS verification without being tied to specific logic or model formalisms. It achieves compositionality through its design, minimising assumptions about the types of logics and models used. The user experience is enhanced to guide the entire verification process. VITAMIN is still under development, but its compositional nature allows for straightforward extension of its components by external developers. In this paper, we focus on VITAMIN usability and extendability, while refraining from discussing its engineering and architecture (see [13] for an in-depth description on these aspects).

VITAMIN [4] is an open-source model checker designed for verifying MAS. The MAS descriptions are provided through labelled transition systems, and VITAMIN supports a variety of specifications, including Computation Tree Logic (CTL) [8], Alternating-time Temporal Logic (ATL) [1], Natural ATL (NatATL) [15], Natural SL [4], ATL with Fuzzy functions (ATLF) [11], Resource-Bounded ATL (RB-ATL) [19], Resource Action-based Bounded ATL (RAB-ATL) [5], Capacity ATL (CapATL) [3], Obstruction Logic (OL) [6], and Obstruction ATL (OATL) [7]. Our tool is implemented in Python and operates on Streamlit[5], an open-source app framework for creating web apps. Therefore, VITAMIN is cross-platform.

3 Demonstration

3.1 VITAMIN: how to use it

In VITAMIN, two types of users are considered: *expert* and *non-expert* users. The former are those not well-versed in formal methods and require guidance from VITAMIN in the verification process. The latter are users who are knowledgeable about formal methods or familiar with them and want to use VITAMIN to perform formal verification on their MAS. A video demonstration of the tool is available at: https://www.youtube.com/playlist?list=PL7_3IMeFMCaC2dVUkEByoeML5L20nQD2P.

[4] The full project can be found here: https://github.com/VadimMalvone/VITAMIN.
[5] https://vitamin-app.streamlit.app/

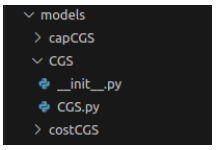
Non-Expert Users. When a non-expert user interacts with VITAMIN, it gathers all the necessary information from the user to generate the model of the MAS and the formal property to verify. The system prompts the user to specify the number of agents to include in the MAS for analysis, giving users the option to assign names to each agent created during this step. Following agent creation, the system requests the number of states to be included in the model, allowing users the opportunity to assign names to the states added to the model. Finally, the system asks users to define the actions to be performed by the agents. Users can customise these actions according to their requirements, and additionally, they need to specify which actions, in which states, the agents will be able to perform, thus defining the transition function among states in the model. Once the system has gathered all the above-mentioned information, a graph will be shown to the user for verification. The last step is to perform the actual verification of the MAS created previously. This is achieved by deciding which formalism, among the available ones in VITAMIN, to use to specify the formula of interest. After that, users can insert the formula into a specific text box according to the chosen formalism. The process concludes with the verification of the formula and the reporting of the verification result in VITAMIN's GUI. The result includes the states from which the formula holds, the model checking result (*i.e.*, the verification outcome considering the initial state(s) of the model), and the time the verification process required to conclude.

Expert Users. When an expert user interacts with VITAMIN, the system does not proactively gather any information but waits for the user's input. In VITAMIN, the required input consists of a text file describing the model of the MAS to verify. The file given in input is a text file with the following attributes: `Transition`, a matrix where row and columns denote the transitions involved in the MAS. `Unknown_by`, a matrix where the indistinguishability relations are defined (it is optional). `Name_State`, a list of states that follows the same order of the transition matrix (*i.e.*, the i-th element represents the i-th row and i-th column in the matrix). `Initial_State`, the unique initial state of the MAS. `Atomic_propositions`, a list of atomic propositions used in the MAS. `Labelling`, a matrix where the rows represent the states, while the columns represent the atomic propositions. Also in this case the columns are ordered w.r.t. the atomic propositions (*i.e.*, the i-th atomic proposition truth values can be derived by the i-th column in the labelling matrix). `Number_of_agents`, the number of agents in the MAS. Once the model is given and properly parsed, VITAMIN expects the user to insert the formal property to verify on the model. This is done in the form of a string of characters describing the formula according to the chosen syntax of one of the logics supported in VITAMIN. Once the property has been passed to VITAMIN and properly parsed as well, the verification can be concluded on the model, and the result returned to the user.

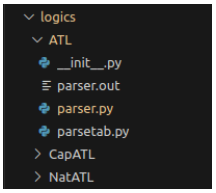
3.2 VITAMIN: how to extend it

Here, we illustrate that the usability of our tool extends not only to end-users but also to developers. In particular, to help developers, VITAMIN is divided in three main components: model component, logic component, and model checker interface component. In the rest of the section we present the project structure.

Model component. In the `models` folder, all the Python scripts regarding the support of all the models in VITAMIN are present. In case a developer wants to extend VITAMIN with a new model formalism, all that is needed is to add a corresponding folder inside the `models` folder. Inside such a folder then, the developer can implement – as they prefer – the parser for the new models to be included in VITAMIN.



Logic component. In the `logics` folder, all the Python scripts regarding the support of all the logics in VITAMIN are present. In case a developer wants to extend VITAMIN with a new logic formalism, all that is needed is to add a corresponding folder inside the `logics` folder. Inside such a folder then, the developer can implement – as they prefer – the parser for the new logics to be included in VITAMIN.



Model Checker Interface component. In the `model_checker_interface` folder, all the Python scripts to perform the actual verification of the logics and models supported in VITAMIN are present. In case a developer wants to extend VITAMIN with a new verification mechanism, all that is needed is to add a corresponding folder inside the `model_checker_interface` folder. Inside such a folder then, the developer can implement – as they prefer – the algorithms to achieve the verification of models against formulas. Notice that, in this folder there is a directory for each verification approach. For example, directories for implicit, explicit, and abstract approaches can be found. Note that, the addition of a verification mechanism may not be tied to the addition of a corresponding new logic or model formalism. Indeed, new verification mechanisms can be added to handle existing logics and models in VITAMIN as well.



How to integrate the modification in VITAMIN. To guarantee a high software quality, all VITAMIN extensions need to be validated before being integrated in the tool and used by the end-users. To achieve this, an auxiliary repository has been created that contains only the skeleton architecture of VITAMIN. Such repository can be used as basis for future extensions. Once each extension is

validated it can be integrated in the deployed version of VITAMIN and served through Streamlit. The skeleton repository can be found at https://github.com/VITAMIN-organisation/VITAMIN-public. After the model, logic, and verification modules are added in VITAMIN, the last step is to make such modules available to end-users (both non-expert and expert ones). To do so, it is necessary to modify the front_end_CS.py Python script. Such a script handles the GUI used by the end-users to interact with VITAMIN and is the script that needs to be updated in order to make the new modules available in the interface. Specifically, such modification requires to update a callback function where such selection is taken into consideration. That is, through a switch case, the newly introduced module can be activated (as reported in the following Python snippet).

A skeleton of the code to add is the following:

```
# NEWLOGIC is the name of the logic introduced
elif Logic == "NEWLOGIC":
  from model_checker_interface.NEWLOGIC import NEWLOGIC
  result = NEWLOGIC.model_checking(formula, filename)
  del NEWLOGIC
  st.write(result["res"]) # states satisfying formula
  st.write(result["initial_state"]) # MC result
```

4 Conclusions

We have presented VITAMIN, an extensive and adaptable framework crafted for the model checking of MAS and beyond. Our main emphasis has lied in the usability and extendability aspects. We are aware that VITAMIN is an ongoing project requiring further refinement, yet we acknowledge that its current stage already signifies a notable progression in the domain of tools for formally verifying MAS. This is particularly significant owing to its potential for further exploration and enhancement, facilitated by its inherent compositional nature.

Our forthcoming endeavours involve the continual expansion of VITAMIN and its dissemination within the MAS community. Furthermore, we plan to present its extensions in future research pursuits, exploring potential implementations of models and logics within the framework. While this manuscript primarily concentrates on the usability and extendability of VITAMIN, future studies will investigate into specific implementations and applications of what we have termed on the VITAMIN's components.

References

1. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM 49(5), 672–713 (2002).
2. Baier, C., Katoen, J.P.: Principles of model checking. MIT press (2008).

3. Ballot, G., Malvone, V., Leneutre, J., Laarouchi, Y.: Strategic reasoning under capacity-constrained agents. In: AAMAS 2024, pp. 123–131.
4. Belardinelli, F., Jamroga, W., Malvone, V., Mittelmann, M., Murano, A., Perrussel, L.: Reasoning about human-friendly strategies in repeated keyword auctions. In: AAMAS 2022, pp. 62–71.
5. Catta, D., Ferrando, A., Malvone, V.: Resource action-based bounded atl: a new logic for mas to express a cost over the actions. In: PRIMA 2024, pp. 206–223.
6. Catta, D., Leneutre, J., Malvone, V.: Obstruction logic: A strategic temporal logic to reason about dynamic game models. In: ECAI 2023, pp. 365–372.
7. Catta, D., Leneutre, J., Malvone, V., Murano, A.: Obstruction alternating-time temporal logic: A strategic logic to reason about dynamic models. In: AAMAS 2024, pp. 271–280.
8. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logics of Programs, pp. 52–71.
9. Dennis, L.A.: The MCAPL framework including the agent infrastructure layer an agent java pathfinder. J. Open Source Softw. **3**(24), 617 (2018).
10. Dennis, L.A., Fisher, M., Webster, M.P., Bordini, R.H.: Model checking agent programming languages. Autom. Softw. Eng. **19**(1), 5–63 (2012).
11. Ferrando, A., Luongo, G., Malvone, V., Murano, A.: Theory and practice of quantitative atl. In: PRIMA 2024, pp. 231–247.
12. Ferrando, A., Malvone, V.: Hands-on VITAMIN: A compositional tool for model checking of multi-agent systems. In: WOA 2024, pp. 148–160.
13. Ferrando, A., Malvone, V.: VITAMIN: A Compositional Framework for Model Checking of Multi-Agent Systems. In: ICAART 2025, pp. 648–655.
14. Ferrando, A., Malvone, V.: VITAMIN: VerIficaTion of A MultI ageNt system. In: AAMAS 2025 – Demo Track. To appear.
15. Jamroga, W., Malvone, V., Murano, A.: Natural strategic ability. Artif. Intell. 277 (2019).
16. Kurpiewski, D., Jamroga, W., Knapik, M.: STV: model checking for strategies under imperfect information. In: AAMAS 2019, pp. 2372–2374.
17. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: an open-source model checker for the verification of multi-agent systems. Int. J. Softw. Tools Technol. Transf. 9–30.
18. Nguyen, C.D., Perini, A., Bernon, C., Pavón, J., Thangarajah, J.: Testing in multi-agent systems. In: AOSE 2009, pp. 180–190.
19. Nguyen, H.N., Alechina, N., Logan, B., Rakib, A.: Alternating-time temporal logic with resource bounds. J. Log. Comput. 631–663 (2018).
20. Winikoff, M.: Debugging agent programs with why?: Questions. In: AAMAS 2017, pp. 251–259.