VITAMIN: A Compositional Framework for Model Checking of Multi-Agent Systems

Angelo Ferrando[1] [a] and Vadim Malvone[2] [b]

[1]*University of Modena and Reggio Emilia, Italy*

[2]*Telecom Paris, Institut Polytechnique de Paris, France*

Keywords: Formal Verification, Model Checking, Multi-Agent Systems, Software Engineering.

Abstract: The verification of Multi-Agent Systems (MAS) poses a significant challenge. Various approaches and methodologies exist to address this challenge; however, tools that support them are not always readily available. Even when such tools are accessible, they tend to be hard-coded, lacking in compositionality, and challenging to use due to a steep learning curve. In this paper, we introduce a methodology designed for the formal verification of MAS in a modular and versatile manner, along with an initial prototype, that we named VITAMIN. Unlike existing verification methodologies and frameworks for MAS, VITAMIN is constructed for easy extension to accommodate various logics (for specifying the properties to verify) and models (for determining on what to verify such properties).

1 INTRODUCTION

Software and hardware systems are notoriously challenging to verify. This difficulty generally arises from their complexity, making formalisation and proper analysis arduous. At times, it is due to their size, rendering exhaustive verification impractical unless appropriately abstracted or optimised (e.g., through symbolic techniques). Regardless of the cause, formally verifying software and hardware systems is a complex task demanding deep expertise in formal methods. Given that such expertise is often scarce, formal verification techniques find limited usability in real-world software and hardware development.

Moving from monolithic systems to Multi-Agent Systems (MAS), formal verification becomes even more complex to achieve. In fact, the process of testing (Nguyen et al., 2009), debugging (Winikoff, 2017), and verifying (Dennis et al., 2012) such systems can be quite complex. Solutions which make the development more reliable are of uttermost importance. Similar to the challenges mentioned for monolithic systems, MASs encounter the same issues in verification. Moreover, as distributed systems comprising intelligent and independent components (the agents), their verification becomes even more de-

manding. This is due to the fact that MAS properties may rely on the rationality of the agents and on how they interact with each other.

One significant development in formal verification is *Alternating-Time Temporal Logic* (ATL) (Alur et al., 2002), enabling reasoning about agents' strategies with temporal goals as payoff. However, ATL's implicit treatment of strategies limits its suitability for certain concepts, leading to the introduction of more powerful formalisms like *Strategy Logic* (SL) (Mogavero et al., 2014). SL treats strategies as first-order objects, providing a richer framework for strategic reasoning. While SL's expressivity is high, it comes at the cost of non-elementary complete model-checking and undecidable satisfiability (Mogavero et al., 2014). To address this, fragments like Strategy Logic with Simple-Goals (Belardinelli et al., 2019) have been proposed, offering better computational properties while still subsuming ATL. In the context of MAS, considering agents' visibility is crucial. The distinction between *perfect* and *imperfect* information MAS impacts model-checking complexity, with imperfect information scenarios often modelled using indistinguishability relations over MAS states (Reif, 1984). This distinction becomes particularly relevant, for instance, in rendering ATL undecidable in the context of imperfect information and memoryful strategies (Dima and Tiplea, 2011). To overcome this problem, some works have either focused on

[a] https://orcid.org/0000-0002-8711-4670

[b] https://orcid.org/0000-0001-6138-4229

648

an approximation to perfect information (Belardinelli et al., 2023), developed notions of bounded memory (Belardinelli et al., 2022), or developed hybrid techniques (Ferrando and Malvone, 2022; Ferrando and Malvone, 2023).

Even with strong theoretical foundations, the formal verification of MAS heavily depends on tools that support such techniques. Notably, some tools stand out as pillars in this field, including MCMAS (Lomuscio et al., 2017) and STV (Kurpiewski et al., 2019).

MCMAS is recognized as one of the most widely used model checkers for the strategic verification of multi-agent systems, primarily due to being one of the earliest tools developed, which served as a foundational proof-of-concept for researchers. Despite the widespread use of MCMAS in the academic community, it exhibits issues that hinder its broader adoption, particularly outside the MAS research community itself. Specifically, its verification process is inherently hard-coded. In fact, even though MCMAS has been extended in various ways, it lacks modularity and does not allow a clear separation between different logics and models that causes maintainability issues. That is, MCMAS lacks the capability of being transparently extended with new logics and models for the verification of MASs. Furthermore, while it does offer a graphical interface, users may find its execution challenging, as it requires additional tools, such as Eclipse, for installation. Moreover, the tool lacks comprehensive external documentation to assist developers, and its internal documentation may prove helpful only to those familiar with its original development. It is important to acknowledge that the observed limitations in MCMAS arise from its primary function as a research tool dedicated to proving theoretical contributions. Regarding STV, it is designed to address specific verification goals in a predetermined way. Consequently, the resulting tool lacks compositional nature and only supports certain types of logics and models for the MAS verification. If users wish to verify the MAS against different logics or models, such flexibility is unavailable to them. Moreover, the tool lacks comprehensive documentation to assist users and developers. Additionally, both MCMAS and STV require a strong background in formal methods, making them challenging for non-expert users to employ them. In summary, although widely used and with a history of successes, both MCMAS and STV tend to lack modularity and usability.

These two aspects are the ones we decided to tackle by engineering and developing a formal verification framework for MAS, called VITAMIN (VerIficaTion of A MultI-ageNt system), which aims at being both highly compositional, in terms of the
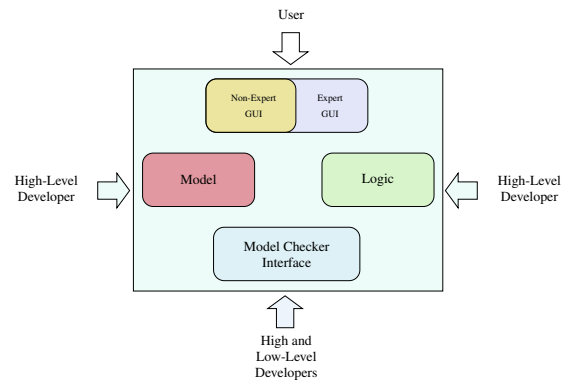

Figure 1: Architecture's overview.

logic and model formalisms that can be employed, and highly usable, in terms of the user experience (from a developer and end-user side). The concept behind our methodology is to generalise MAS verification without being tied to any specific logic or model formalism. VITAMIN, even though still under development, achieves compositionality through its design, minimising assumptions about the types of logics and models that can be employed. Its end-user's usability is enhanced through a user experience that guides the entire verification process. It is worth noting that, VITAMIN's compositionality facilitates straightforward extension of its components by external developers. We refrain from presenting experimental results for two primary reasons. Firstly, an empirical evaluation would be beyond the scope of our work, which focuses on the engineering of VITAMIN and its foundational aspects. Secondly, due to VITAMIN's inherent compositionality, it can readily accommodate the integration of both new and existing verification techniques. Consequently, comparing VITAMIN with existing verification tools for MAS would not be meaningful, as each tool would essentially be compared, in theory, with itself.

2 VITAMIN: ARCHITECTURE

In this section, we focus on the engineering of our approach. To do so, we present an overview of the tool, named VITAMIN, in Figure 1, showcasing its main components. First and foremost, there is a clear separation between the logics and models used in our verification methodology. This separation of concerns is of paramount importance as it enables the tool to evolve independently in both directions. The logics and models should not be tied to each other, providing a more flexible environment where different logics can be verified on distinct models.

Another noteworthy aspect in Figure 1 is the pres-

ence of an interface dedicated to handling the actual formal verification. Further details on this aspect are discussed in the subsequent sections. It is crucial to emphasise that, similar to the logics and models, the verification component foreseen in our methodology is independent and highly compositional. As an illustration, we can choose to verify ATL properties on a Concurrent Game Structure (CGS) (Alur et al., 2002), and such verification can be executed in different ways. For instance, we may opt for an `explicit` verification based on fix-point (similar to what is done in the case of CTL properties), or alternatively, symbolically encode the model as a Binary Decision Diagram (BDD) and perform symbolic (*i.e.*, `implicit`) model checking, or finally, `abstract` the model to cope with its complexity and perform the verification on the resulting abstracted model. These examples underscore how the actual verification, considering a logic and a model, can be achieved in various ways. Importantly, this aspect is kept separate from the tool's ecosystem to avoid hard-coding within it.

One additional aspect to note in Figure 1 is the type of users envisioned in our approach. Rather than the standard end-user, we assume the presence of three user categories. The first one corresponds to what we commonly refer to as an end-user: a user who utilises the verification approach solely for verifying a MAS, without the intention or objective of extending or modifying the tool itself. In addition to the standard `User`, we envisage the access to two levels of developer users: the `High-Level Developer`, who concentrates on formal verification aspects, and the `Low-Level Developer`, who focuses on optimisation and low-level implementation.

`High-Level Developer` users possess experience in model checking within MAS and are responsible for developing the `Model` and `Logic` components and can develop the explicit verification via `Model Checker Interface`. They can achieve this by extending existing models and logics or by introducing entirely new ones into the architecture. Importantly, as each model in the `Model` component, each logic in the `Logic` component, and each explicit verification algorithm in the `Model Checker Interface` component is developed as an independent module, the enrichment of these components with new modules does not introduce errors into previously validated modules. These developers are concerned with properly defining and verifying the models and logics in the system, without delving into the intricacies of creating high-performance software solutions.

The task of optimising such implementations is undertaken by `Low-Level Developer` users who possess expertise not only in model checking within

MAS but also in software engineering. They are responsible for enhancing the implementation provided by `High-Level Developers` by leveraging optimisation techniques, which can encompass both algorithmic and data structure improvements. For example, a `High-Level Developer` may propose a logic and its verification on an explicit model, and a `Low-Level Developer`, starting from such an implementation, may offer an optimised solution based on the verification of an implicit model, such as BDDs.

To bridge the gap between these two types of developers, we have introduced the `Model Checker Interface` component in our methodology. This component, developed by low-level developers, is intended to be utilised by end-users and high-level developers to seamlessly use the optimisations.

Thanks to the presented architecture, the strengths of our approach include: (**Modularity**) it allows transparent extension without the need for core engine modifications. (**User-Friendlyness**) it enables end-users to use the software without requiring expertise in formal verification. (**Distribution**) the compositionality of the verification components enables their distribution on different machines. (**Documentation**) we prioritise the development of internal and external documentation to assist users and developers in effectively utilising and extending the tool.

3 VITAMIN: DETAIL ON THE ARCHITECTURE MODULES

We now delve more into the details of the components of the architecture behind our verification methodology. To help us present it, we present a diagram for each of such components. Note that, the colours are consistent with the ones used in Figure 1, which details the more general diagram of the architecture and its components in a whole. Specifically, we can see how the compositionality is achieved through hierarchical structures; where in the root we have the general notion, and going deeper into the resulting hierarchy, we have various instantiations to serve different verification purposes.

3.1 Model

As illustrated in Figure 2, the model component is structured hierarchically. At the root, we find the concept of a model, while further down in the hierarchy, two crucial branches of system models emerge: `Monolithic` and `Multi-Agent`. The former represents the standard model for software and
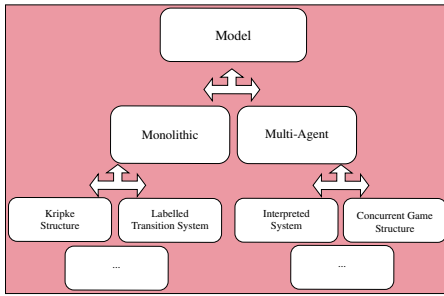
Figure 2: Model component insights.

Figure 3: Logic component insights.

Figure 4: Model Checker Interface insights.

hardware systems, such as Kripke Structures (Chellas, 1980) and Labelled Transition Systems (Keller, 1976). These models are used to depict the behaviour of centralised systems. However, such models typically do not account for the presence of autonomous entities (such as agents) and lack a proper means to characterise their independent and rational behaviours. To address this, the methodology supports the notion of multi-agent models, including Concurrent Game Structures (CGSs) (Alur et al., 2002) and Interpreted Systems (ISs) (Fagin et al., 1995).

It is worth noting that, we envision the possibility of further extending the structure of the hierarchy for the model component. Nevertheless, in its initial phase, we have chosen to consider the two most influential and commonly used branches of models, particularly for specifying system behaviour (at least for verification purposes).

3.2 Logics

As depicted in Figure 3, an independent structure is in place to manage the various logics available. The necessity for such a hierarchy arises from the diverse nature of the logics that can be employed for the verification of Multi-Agent Systems (MAS). Specifically, in our methodology we distinguish between two types of formalisms for denoting properties to be verified: `Temporal` and `Strategic`. The former encompasses more standard temporal logics, such as Linear Temporal Logic (LTL) (Pnueli, 1977) or Computation Tree Logic (CTL) (Clarke and Emerson, 1981). The latter involves more recent strategic logics, with examples including Alternating-Time Temporal Logic (ATL) (Alur et al., 2002) or Strategy Logic (SL) (Mogavero et al., 2014).

Given `VITAMIN`'s intrinsic compositional approach to handling logics, it allows for the addition of branches in the hierarchy. However, in its initial phase, we chose the two most studied and commonly used branches of logics for verification purposes.
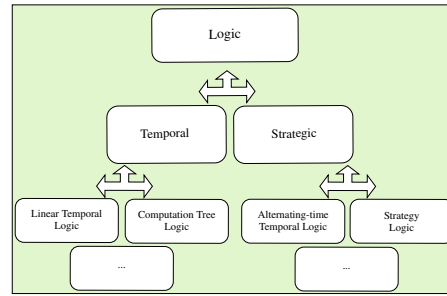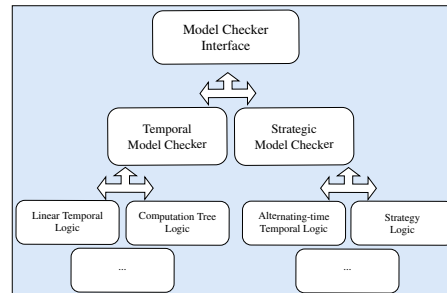
3.3 Model Checker Interface

Once the model and logic are chosen, the next step in our methodology is their verification. This is obtained through the `Model Checker Interface`, presented in Figure 4. Notice that, each leaf of Figure 4 represents a meta-node that can be further decomposed as shown in Figure 5 for the case of Strategy Logic.

To better comprehend the role of such a component, it is noteworthy that our methodology incorporates a selection mechanism connected to the model checker interface. This mechanism enables the choice of the appropriate model checker usage, considering the selected model and logic. The interface is configured to analyse the model description and logical formula, determining the class of model among a set of predefined model classes to which the model belongs (`Model` component) and the class of logic among a set of predefined logic classes to which the formula belongs (`Logic` component). Then, given the model and logic, the model checker interface selects the most efficient verification method. For instance, by assuming the number of states as main parameter of the problem, the model checker interface could select an ex-
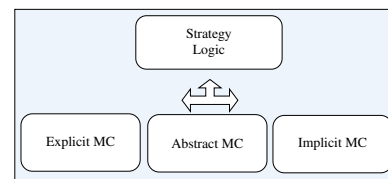

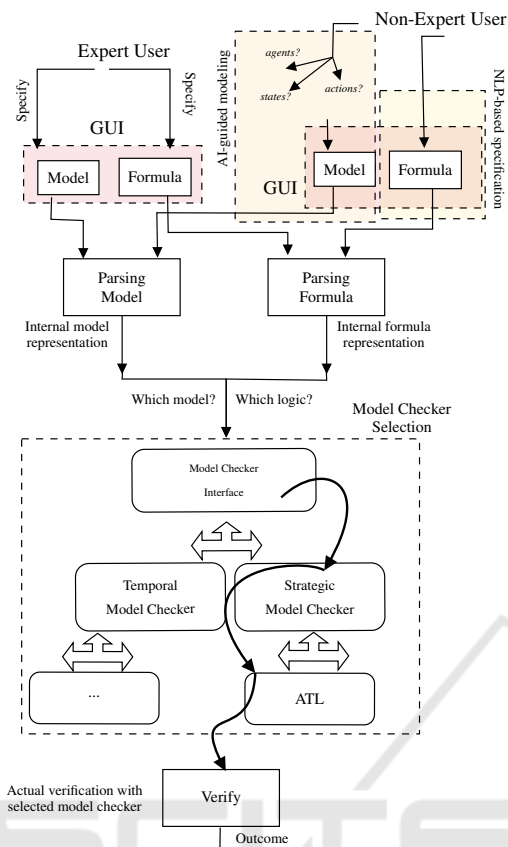Figure 5: Meta-node for Strategy Logic.

Figure 6: Our methodology's flowchart for a ATL.

plicit method for models with less than fifty states, an implicit method for models with less than one-hundred states, and an abstract method with more than one-hundred states.

3.4 User Interface

Here, we focus on the way the end-user interacts with VITAMIN. In VITAMIN, we categorise end-users based on their knowledge of formal methods. Specifically, we distinguish between Expert and Non-Expert users. In both cases, being end-users of the system, they would benefit from a Graphical User Interface (GUI) to guide them through the verification.

In Figure 6, we present an ideal interaction flow for both types of users.

On the left, we depict the interaction involving an expert user, which is more straightforward and direct. Since the user is experienced with formal methods, they can simply upload the corresponding files for the model and formula through a GUI. These files must be compatible with the format expected by the specific instantiation. Once these files are provided as input, the verification continues.

On the right, the interaction is guided, as the non-

expert user lacks experience with formal methods. To address this, our methodology solicits details and information about the model and the formula. These information can be gathered through a sequence of questions to the user. For instance, the user may be asked about the number of agents he/she thinks to employ in the MAS, or, how many and which kind of actions are available for such agents. These examples serve only as illustrations, demonstrating the system's potential to guide non-expert users. By following such a constrained step-by-step process, a non-expert user can interact in a more natural way. Once information about the model is supplied, we may continue by seeking additional details about the formula. The communication in this phase can also be handled through natural language. Questions during this step may inquire about the specific property of interest, which might involve considering temporal information. It is worth noting that this step is inspired by what is commonly done in the FRET framework (Giannakopoulou et al., 2020) (and similar ones), where users can describe formal properties in a constrained natural language, and it is the system's responsibility to generate the corresponding formal property in the chosen formalism.

At the conclusion of the guided process, akin to the expert user, the non-expert user can proceed with the verification steps illustrated in Figure 6. This process begins with a parsing step, in which the model and formula are parsed, leading to the creation of an internal representation. Following the parsing step, the verification process advances with the selection of an appropriate model checker to address the verification of the model against the given formula. Subsequently, the verification result is returned to the user.

4 IMPLEMENTATION

In this section, we focus on the instantiation of VITAMIN and what the tool currently supports. VITAMIN is implemented in Python and its GUI is accessible through a web browser (https://vitamin-app. streamlit.app/), which makes the tool cross-platform. This accessibility is facilitated by utilising the Streamlit[1] Python library, which supports the transparent sharing of Python programs via HTTP protocol. Furthermore, the source code of the architecture of VITAMIN can be found in https://github.com/ VITAMIN-organisation/VITAMIN-public. It is important to note that this repository contains only the architecture, while the source codes of the individual

---

[1]https://streamlit.io/

modules are not publicly available. However, they can be freely utilised through VITAMIN's browser GUI. It is worth mentioning that the actual implementation of the modules is beyond the scope of this paper, as its goal is to present the engineering and architecture of the VITAMIN framework. As mentioned in the paper, VITAMIN supports the interaction with both expert and non-expert users.

4.1 Non-Expert User Experience

Figure 7 displays a screenshot of VITAMIN's GUI where the user is asked about the number of agents to employ in the MAS to verify. In this specific example, the user wants to create a MAS comprising two agents, named A0 and A1.

Figure 7: VITAMIN asks the user for the agents of the MAS.

After the number of agents has been given to the system, the process continues with the number of states. As reported in Figure 8, the user inserts the number of states he/she wishes to add to the MAS model under analysis. In this specific case, the user chooses to add four states, which are then named: S0, S1, S2, and S3. Once both agents and states have been collected, VITAMIN asks for the actions that the agents can perform in the states. Figure 9 reports the step where VITAMIN asks the user to assign the actions to the agents created previously, in the states added previously. In this specific scenario, the user decides that the agents can perform three actions, which are named: A, B, and C.

Now that the agents have actions to perform, VITAMIN requires the user to specify which actions can be performed by which agent in which state. This process allows a natural definition of transitions amongst states through agents' actions. The transitions are reported by the user through VITAMIN's GUI. In Figure 10, we report only a subset of A1's transitions to improve readability. In this specific scenario, agent A1 can perform only action A in state S1 to move to state S2, and actions B and C to move to state S3. For the remaining transitions, both for A0 and A1, the
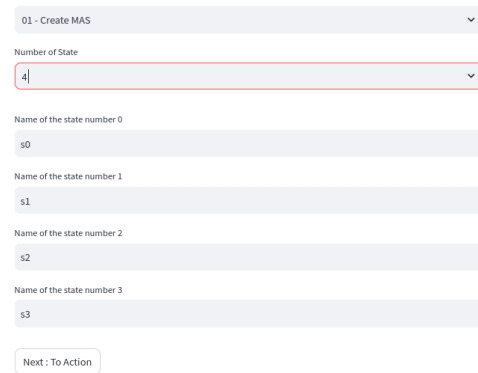

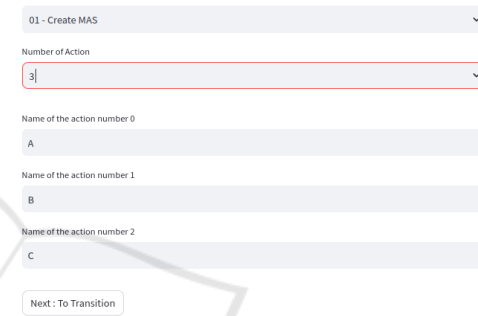Figure 8: VITAMIN asks the user the states of the model.

Figure 9: VITAMIN asks the user for the agents' actions.
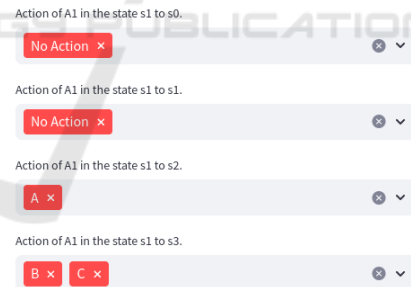
reader can refer to Figure 11.

Figure 10: VITAMIN asks the user for transitions.

At this point, VITAMIN has all the information it needs to represent the model graph created by the user. To allow the user to validate the resulting model, VITAMIN shows the graph result to the user, as reported in Figure 11. VITAMIN supports the graphical visualisation of the model and allows the user to validate it before moving on in the verification process.

Finally, given the model produced step-by-step by guiding the user through all the details needed to populate the model, VITAMIN can conclude the process by asking the formula to verify on the obtained model. Figure 12 reports the last step of the VITAMIN's process where the user can specify the formal property to
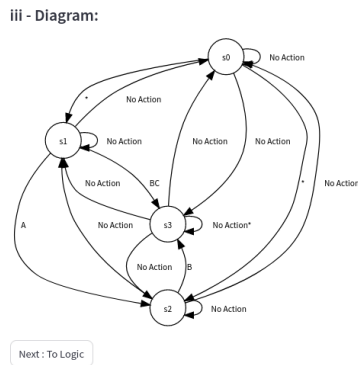
iii - Diagram:

Figure 11: VITAMIN displays the graph from the gathered information for user validation.

verify on the model. In this specific case, the user decides to specify an ATL formula and, in particular, to verify whether the agents can reach state S3 by collaborating. Such a property is verified then by VITAMIN and concluded as satisfied on the model of the MAS.

VITAMIN's current instantiation requires the formula to be specified, however, a step-by-step mechanism could be employed as well. Nonetheless, differently from the model scenario, the formula may require additional engineering since it may largely depend on the formalism chosen by the user.

Figure 12: VITAMIN asks the user for the formula to verify on the previously constructed model of the MAS.

4.2 Expert User Experience

Here, we show how an expert user interacts with VITAMIN. Specifically, this is achieved by the definition of the model of the MAS and the formal property the user wishes to verify. Differently from the non-expert user, VITAMIN does not guide the expert user, but it expects the model and formula as input.

VITAMIN is born to handle different types of models and logics. Due to its design, it is not limited to any specific model (resp., logic) since each model (resp., logic) can be seen as an independent component of the system. However, to give an example, we show how to define a MAS model as a CGS in VITAMIN. Figure 13 displays a screenshot of VITAMIN's GUI where the user can upload the CGS of the MAS to verify. The model has to follow a specific format that has been chosen for the definition of

CGSs in VITAMIN. Naturally, because of their compositionality, these format choices are exclusive to how VITAMIN handles CGSs and do not concern the development of other formalisms for model representation. So, if VITAMIN supported Interpreted Systems (like MCMAS), it would be free to choose the format that best suits such models, without being concerned about how CGSs are modelled, and *vice versa*.
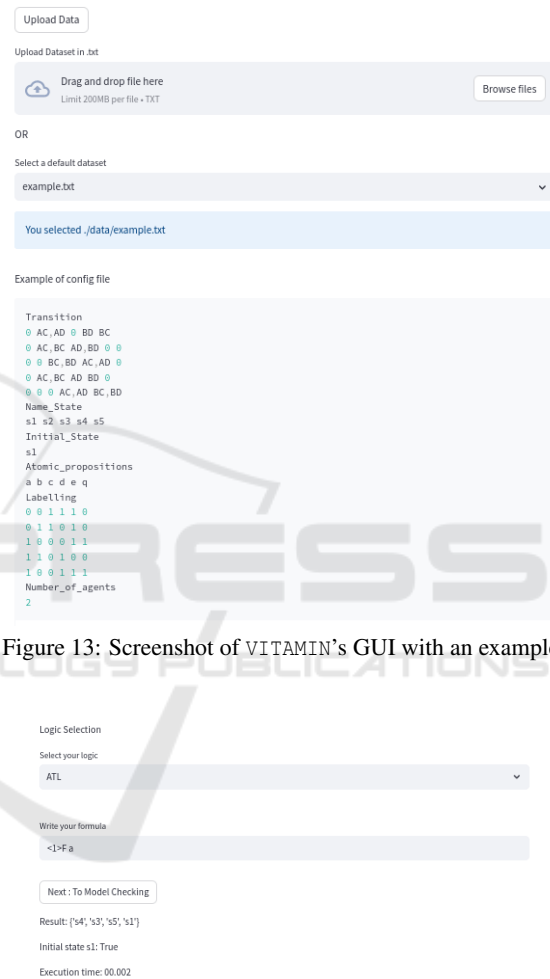

Figure 13: Screenshot of VITAMIN's GUI with an example.

Figure 14: Screenshot of VITAMIN's GUI.

After the model has been uploaded by the user, VITAMIN expects the formula to be verified on the latter. This step, similarly to the non-expert user scenario, is obtained by letting the user fill a field box in the VITAMIN's GUI. Figure 14 displays a screenshot of the GUI where the user fills the box with the formula of interest to verify on the model. Once both the model and formula are given, the verification process may start and the result of the verification is returned back to the user.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we introduced VITAMIN, a comprehensive and versatile framework designed for model checking of Multi-Agent Systems and beyond. Our emphasis was on the engineering aspects and decisions made during the development of VITAMIN.

We acknowledge that VITAMIN is an ongoing project that will require additional refinement, but recognise that its current state already represents a noteworthy advancement in the realm of tools for the formal verification of MAS. This is especially notable given its potential for further study and extension, facilitated by its inherent compositionality.

Note that VITAMIN is currently in a prototype stage. Certain aspects presented in this paper such as Natural Language Processing (NLP) support for non-expert users and the Model Checker Interface part, are still in development. However, everything related to verification and compositional representation in VITAMIN has already been implemented and tested across various scenarios, each highlighting different models and formulas for verification.

Our future plans include the continued expansion of VITAMIN, along with sharing it with the MAS community. Additionally, we intend to present its extensions in future research endeavours, exploring possible instantiations of models and logics within the tool. While this work has primarily focused on VITAMIN's engineering and architecture, future research will delve into specific instantiations (of what we called the VITAMIN's components).

REFERENCES

Alur, R., Henzinger, T. A., and Kupferman, O. (2002). Alternating-time temporal logic. *J. ACM*, 49(5):672–713.

Belardinelli, F., Ferrando, A., and Malvone, V. (2023). An abstraction-refinement framework for verifying strategic properties in multi-agent systems with imperfect information. *Artif. Intell.*, 316:103847.

Belardinelli, F., Jamroga, W., Kurpiewski, D., Malvone, V., and Murano, A. (2019). Strategy logic with simple goals: Tractable reasoning about strategies. In IJCAI 2019, pages 88–94.

Belardinelli, F., Lomuscio, A., Malvone, V., and Yu, E. (2022). Approximating perfect recall when model checking strategic abilities: Theory and applications. *J. Artif. Intell. Res.*, 73:897–932.

Chellas, B. F. (1980). *Modal Logic - An Introduction*. Cambridge University Press.

Clarke, E. M. and Emerson, E. A. (1981). Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs*, pages 52–71.

Dennis, L. A., Fisher, M., Webster, M. P., and Bordini, R. H. (2012). Model checking agent programming languages. *Autom. Softw. Eng.*, 19(1):5–63.

Dima, C. and Tiplea, F. L. (2011). Model-checking ATL under imperfect information and perfect recall semantics is undecidable. *CoRR*, abs/1102.4225.

Fagin, R., Halpern, J. Y., Moses, Y., and Vardi, M. Y. (1995). *Reasoning About Knowledge*. MIT Press.

Ferrando, A. and Malvone, V. (2022). Towards the combination of model checking and runtime verification on multi-agent systems. In PAAMS, pages 140–152.

Ferrando, A. and Malvone, V. (2023). Towards the verification of strategic properties in multi-agent systems with imperfect information. In AAMAS, pages 793–801.

Giannakopoulou, D., Pressburger, T., Mavridou, A., Rhein, J., Schumann, J., and Shi, N. (2020). Formal requirements elicitation with FRET. In REFSQ.

Keller, R. M. (1976). Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384.

Kurpiewski, D., Jamroga, W., and Knapik, M. (2019). STV: model checking for strategies under imperfect information. In AAMAS, pages 2372–2374.

Lomuscio, A., Qu, H., and Raimondi, F. (2017). MCMAS: an open-source model checker for the verification of multi-agent systems. *Int. J. Softw. Tools Technol. Transf.*, 19(1):9–30.

Mogavero, F., Murano, A., Perelli, G., and Vardi, M. Y. (2014). Reasoning about strategies: On the model-checking problem. *ACM Trans. Comput. Log.*, 15(4):34:1–34:47.

Nguyen, C. D., Perini, A., Bernon, C., Pavón, J., and Thangarajah, J. (2009). Testing in multi-agent systems. In AOSE, pages 180–190.

Pnueli, A. (1977). The temporal logic of programs. In FOCS, pages 46–57.

Reif, J. H. (1984). The complexity of two-player games of incomplete information. *J. Comput. Syst. Sci.*, 29(2):274–301.

Winikoff, M. (2017). Debugging agent programs with why?: Questions. In AAMAS, pages 251–259.