Towards the Combination of Model Checking and Runtime Verification on Multi-Agent Systems

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

[1] University of Genova, Italy
angelo.ferrando@unige.it
[2] Télécom Paris, France
vadim.malvone@telecom-paris.fr

Abstract. Multi-Agent Systems (MAS) are notoriously complex and hard to verify. In fact, it is not trivial to model a MAS, and even when a model is built, it is not always possible to verify, in a formal way, that it is actually behaving as we expect. Usually, it is relevant to know whether an agent is capable of fulfilling its own goals. One possible way to check this is through Model Checking. Specifically, by verifying Alternating-time Temporal Logic (ATL) properties, where the notion of strategies for achieving goals can be described. Unfortunately, the resulting model checking problem is not decidable in general. In this paper, we present a verification procedure based on combining Model Checking and Runtime Verification, where sub-models of the MAS model belonging to decidable fragments are verified by a model checker, and runtime monitors are used to verify the rest. We present our technique and show experimental results.

Keywords: Model Checking · Runtime Verification · Strategic reasoning

1 Introduction

Intelligent systems, such as Multi-Agent Systems (MAS), can be seen as a set of intelligent entities capable of proactively decide how to act to fulfill their own goals. These entities, called generally agents, are notoriously autonomous, *i.e.*, they do not expect input from a user to act, and social, *i.e.*, they usually communicate amongst each other to achieve common goals.

Software systems are not easy to trust in general. Because of this, we need verification techniques to verify that such systems behave as expected. More specifically, in the case of MAS, it is relevant to know whether the agents are capable of achieving their own goals, by themselves or by collaborating with other agents by forming a coalition. This is usually referred to as the process of finding a strategy for the agent(s).

A well-known formalism for reasoning about strategic behaviours in MAS is Alternating-time Temporal Logic (ATL) [1]. Before verifying ATL specifications, two questions need to be answered: (i) *does each agent know everything about the system?* (ii) *does the property require the agent to have memory of the system?* The first question concerns the model of the MAS. If each agent can distinguish each state of the model, then we have *perfect information*; otherwise, we have *imperfect information*. The second question concerns the ATL property. If the property can be verified without the need for the agent to remember which states of the model have been visited before, then we have *imperfect recall*; otherwise, we have *perfect recall*.

The model checking problem for ATL giving a generic MAS is known to be undecidable. This is due to the fact that the model checking problem for ATL specifications under imperfect information and perfect recall has been proved to be undecidable [12]. Nonetheless, decidable fragments exist. Indeed, model checking ATL under perfect information is PTIME-complete [1], while under imperfect information and imperfect recall is PSPACE [23]. Unfortunately, MAS usually have imperfect information, and when memory is needed to achieve the goals, the resulting model checking problem becomes undecidable. Given the relevance of the imperfect information setting, even partial solutions to the problem are useful.

Given an ATL formula φ and a model of MAS M, our procedure extracts all the sub-models of M with perfect information that satisfy a sub-formula of φ. Then, runtime monitors are used to check if the remaining part of φ can be satisfied at execution time. If this is the case, we conclude at runtime the satisfaction of φ for the corresponding system execution. Note that, this does not imply that the system satisfies φ, indeed future executions may violate φ. The formal result over φ only concerns the current execution, and how it has behaved in it. However, we will present preservation results on the initial model checking problem of φ on the model of the system M, as well.

Related Work. Several approaches for the verification of specifications in ATL and ATL^* under imperfect information and perfect recall have been recently put forward. In one line, restrictions are made on how information is shared amongst the agents, so as to retain decidability [10,11]. In a related line, interactions amongst agents are limited to public actions only [6,7]. These approaches are markedly different from ours as they seek to identify classes for which verification is decidable. Instead, we consider the whole class of iCGS and define a general verification procedure. In this sense, existing approaches to approximate ATL model checking under imperfect information and perfect recall have either focused on an approximation to perfect information [5,8] or developed notions of bounded recall [4].

Differently from these works, we introduce, for the first time, a technique that couples model checking and runtime verification to provide results. Furthermore, we always concludes with a result. Note that the problem is undecidable in general, thus the result might be inconclusive (but it is always returned). When the result is inconclusive for the whole formula, we present sub-results to give at least the maximum information about the satisfaction/violation of the formula under exam.

Runtime Verification (RV) has never been used before in a strategic context, where monitors check whether a coalition of agents satisfies a strategic property. This can be obtained by combining Model Checking on MAS with RV. The combination of Model Checking with RV is not new [17]; even though focused only on LTL. Instead, in here, we focus on strategic properties, such as ATL^*. Because of this, our work is closer in spirit to [17]; in fact, we use RV to support Model Checking in verifying at runtime what the model checker could not at static time. Finally, in [14], a demonstration paper presenting the tool deriving by this work may be found. Specifically, in this paper we present the theoretical foundations and experimental results behind the tool.

2 Preliminaries

In this section we recall some preliminary notions. Given a set U, $\overline{U}$ denotes its complement. We denote the length of a tuple v as $|v|$, and its i-th element as v_i. For $i \leq |v|$, let $v_{\geq i}$ be the suffix $v_i, \ldots, v_{|v|}$ of v starting at v_i and $v_{\leq i}$ the prefix $v_1, \ldots, v_i$ of v. We denote with $v \cdot w$ the concatenation of the tuples v and w.

2.1 Models for Multi-agent systems

We start by giving a formal model for Multi-agent Systems by means of concurrent game structures with imperfect information [1,18].

Definition 1. *A* concurrent game structure with imperfect information (iCGS) *is a tuple* $M = \langle Ag, AP, S, s_I, \{Act_i\}_{i \in Ag}, \{\sim_i\}_{i \in Ag}, d, \delta, V \rangle$ *such that:* $Ag = \{1, \ldots, m\}$ *is a nonempty finite set of agents (or players);* AP *is a nonempty finite set of atomic propositions (atoms);* $S \neq \emptyset$ *is a finite set of* states, *with* initial state $s_I \in S$*; for every* $i \in Ag$, Act_i *is a nonempty finite set of* actions *where* $Act = \bigcup_{i \in Ag} Act_i$ *is the set of all actions and* $ACT = \prod_{i \in Ag} Act_i$ *is the set of all joint actions; for every* $i \in Ag$, $\sim_i$ *is a relation of* indistinguishability *between states, that is, given states* $s, s' \in S$, $s \sim_i s'$ *iff* s *and* s' *are observationally indistinguishable for agent* i*; the* protocol function $d : Ag \times S \to (2^{Act} \setminus \emptyset)$ *defines the availability of actions so that for every* $i \in Ag$, $s \in S$, *(i)* $d(i, s) \subseteq Act_i$ *and (ii)* $s \sim_i s'$ *implies* $d(i, s) = d(i, s')$*; the* (deterministic) transition function $\delta : S \times ACT \to S$ *assigns a successor state* $s' = \delta(s, \vec{a})$ *to each state* $s \in S$*, for every joint action* $\vec{a} \in ACT$ *such that* $a_i \in d(i, s)$ *for every* $i \in Ag$*, that is,* $\vec{a}$ *is* enabled *at* s*; and* $V : S \to 2^{AP}$ *is the* labelling function.

By Def. 1 an iCGS describes the interactions of a group Ag of agents, starting from the initial state $s_I \in S$, according to the transition function δ. The latter is constrained by the availability of actions to agents, as specified by the protocol function d. Furthermore, we assume that every agent i has imperfect information of the exact state of the system; so in any state s, i considers epistemically possible all states s' that are i-indistinguishable from s [13]. When every $\sim_i$ is the identity relation, *i.e.*, $s \sim_i s'$ iff $s = s'$, we obtain a standard CGS with perfect information [1]. Given a set $\Gamma \subseteq Ag$ of agents and a joint action $\vec{a} \in ACT$, let $\vec{a}_\Gamma$ and $\vec{a}_{\overline{\Gamma}}$ be two tuples comprising only of actions for the agents in Γ and $\overline{\Gamma}$, respectively. A history $h \in S^+$ is a finite (nonempty) sequence of states. The indistinguishability relations are extended to histories in a synchronous, point-wise way, *i.e.*, histories $h, h' \in S^+$ are *indistinguishable* for agent $i \in Ag$, or $h \sim_i h'$, iff (i) $|h| = |h'|$ and (ii) for all $j \leq |h|$, $h_j \sim_i h'_j$.

2.2 Syntax

To reason about the strategic abilities of agents in iCGS with imperfect information, we use Alternating-time Temporal Logic ATL^* [1].

Definition 2. *State* (φ) *and path* (ψ) *formulas in* ATL^* *are defined as follows, where* $q \in AP$ *and* $\Gamma \subseteq Ag$*:*

$$\varphi ::= q \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\langle \Gamma \rangle\rangle \psi$$
$$\psi ::= \varphi \mid \neg\psi \mid \psi \wedge \psi \mid X\psi \mid (\psi U \psi)$$

Formulas in ATL^* are all and only the state formulas.

As customary, a formula $\langle\langle\Gamma\rangle\rangle\Phi$ is read as "the agents in coalition Γ have a strategy to achieve Φ". The meaning of linear-time operators *next* X and *until* U is standard [2]. Operators $[\![\Gamma]\!]$, *release* R, *finally* F, and *globally* G can be introduced as usual. Formulas in the ATL fragment of ATL^* are obtained from Def. 2 by restricting path formulas ψ as follows (where φ is a state formula and R is the *release* operator):

$$\psi ::= X\varphi \mid (\varphi U \varphi) \mid (\varphi R \varphi)$$

We will also consider the syntax of ATL* in negation normal form (NNF):

$$\varphi ::= q \mid \neg q \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle\langle\Gamma\rangle\rangle\psi \mid [\![\Gamma]\!]\psi$$
$$\psi ::= \varphi \mid \psi \wedge \psi \mid \psi \vee \psi \mid X\psi \mid (\psi U \psi) \mid (\psi R \psi)$$

where $q \in AP$ and $\Gamma \subseteq Ag$.

2.3 Semantics

When giving a semantics to ATL^* formulas we assume agents are endowed with *uniform strategies* [18], *i.e.*, they perform the same action whenever they have the same information.

Definition 3. *A* uniform strategy *for agent $i \in Ag$ is a function $\sigma_i : S^+ \to Act_i$ such that for all histories $h, h' \in S^+$, (i) $\sigma_i(h) \in d(i, last(h))$; and (ii) $h \sim_i h'$ implies $\sigma_i(h) = \sigma_i(h')$.*

By Def. 3 any strategy for agent i has to return actions that are enabled for i. Also, whenever two histories are indistinguishable for i, then the same action is returned. Notice that, for the case of CGS (perfect information), condition (ii) is satisfied by any strategy σ. Furthermore, we obtain memoryless (or imperfect recall) strategies by considering the domain of σ_i in S, *i.e.*, $\sigma_i : S \to Act_i$.

Given an iCGS M, a *path* $p \in S^\omega$ is an infinite sequence $s_1 s_2 \dots$ of states. Given a joint strategy $\sigma_\Gamma = \{\sigma_i \mid i \in \Gamma\}$, comprising of one strategy for each agent in coalition Γ, a path p is σ_Γ*-compatible* iff for every $j \geq 1$, $p_{j+1} = \delta(p_j, \vec{a})$ for some joint action $\vec{a}$ such that for every $i \in \Gamma$, $a_i = \sigma_i(p_{\leq j})$, and for every $i \in \overline{\Gamma}$, $a_i \in d(i, p_j)$. Let $out(s, \sigma_\Gamma)$ be the set of all σ_Γ-compatible paths from s.

Definition 4. *The satisfaction relation $\models$ for an iCGS M, state $s \in S$, path $p \in S^\omega$, atom $q \in AP$, and ATL^* formula ϕ is defined as follows (clauses for Boolean connectives are immediate and thus omitted):*

$(M, s) \models q$ *iff* $q \in V(s)$
$(M, s) \models \langle\langle\Gamma\rangle\rangle\psi$ *iff for some* σ_Γ, *for all* $p \in out(s, \sigma_\Gamma)$, $(M, p) \models \psi$
$(M, p) \models \varphi$ *iff* $(M, p_1) \models \varphi$
$(M, p) \models X\psi$ *iff* $(M, p_{\geq 2}) \models \psi$
$(M, p) \models \psi U \psi'$ *iff for some* $k \geq 1, (M, p_{\geq k}) \models \psi'$, *and for all* $1 \leq j < k \Rightarrow (M, p_{\geq j}) \models \psi$

We say that formula ϕ is *true* in an iCGS M, or $M \models \phi$, iff $(M, s_I) \models \phi$.

Definition 5. *Given an iCGS M and a formula ϕ, the model checking problem concerns determining whether $M \models \phi$.*

Since the semantics provided in Def. 4 is the standard interpretation of ATL^* [1,18], it is well known that model checking ATL, *a fortiori* ATL^*, against iCGS with imperfect information and perfect recall is undecidable [12]. In the rest of the paper we develop methods to obtain partial solutions to this by using Runtime Verification (RV).

2.4 Runtime Verification and Monitors

The standard formalism to specify formal properties in RV is Linear Temporal Logic (LTL) [21]. The syntax and semantics of LTL is the same of ATL^* (Def. 2-4), with state formulas $\varphi ::= q$ (*i.e.*, no strategic operators are allowed).

Definition 6 (Monitor). *Let M be an iCGS and ψ be an LTL property. Then, a monitor for ψ is a function $Mon_\psi^M : S^+ \to \mathbb{B}_3$, where $\mathbb{B}_3 = \{\top, \bot, ?\}$:*

$$Mon_\psi^M(h) = \begin{cases} \top & \forall_{p \in S^\omega} \ (M, h \cdot p) \models \psi \\ \bot & \forall_{p \in S^\omega} \ (M, h \cdot p) \not\models \psi \\ ? & otherwise. \end{cases}$$

where the path p is a valid continuation of the history h in M.

Intuitively, a monitor returns $\top$ if all continuations of h satisfy ψ; $\bot$ if all continuations of h violate ψ; $?$ otherwise. The first two outcomes are standard representations of satisfaction and violation, while the third is specific to RV. In more detail, it denotes when the monitor cannot conclude any verdict yet. This is closely related to the fact that RV is applied while the system is still running, and not all information about it are available. For instance, a property might be currently satisfied (resp., violated) by the system, but violated (resp., satisfied) in the (still unknown) future. The monitor can only safely conclude any of the two final verdicts ($\top$ or $\bot$) if it is sure such verdict will never change. The addition of the third outcome symbol $?$ helps the monitor to represent its uncertainty w.r.t. the current system execution.

2.5 Negative and Positive Sub-models

Now, we recall two definitions of sub-models and some preservation results, defined in [15], that we will use in our verification procedure.

Definition 7 (Negative and Positive sub-models). *Given an iCGS $M = \langle Ag, AP, S, s_I, \{Act_i\}_{i \in Ag}, \{\sim_i\}_{i \in Ag}, d, \delta, V\rangle$, we denote with $M' = \langle Ag, AP, S', s_I, \{Act_i\}_{i \in Ag}, \{\sim'_i\}_{i \in Ag}, d', \delta', V'\rangle$ a negative sub-model of M, formally $M' \subseteq M$, such that: $S' = S^\star \cup \{s_t\}$, where $S^\star \subseteq S$ and $s_I \in S^\star$; $\sim'_i$ is defined as the corresponding $\sim_i$ restricted to $S^\star$; the protocol function is defined as $d'(i,s) = d(i,s)$, for every $s \in S^\star$ and $d'(i, s_t) = Act_i$, for all $i \in Ag$; given a transition $\delta(s, \vec{a}) = s'$, if $s, s' \in$*

$S^\star$ then $\delta'(s,\vec{a}) = \delta(s,\vec{a}) = s'$ else if $s' \in S \setminus S^\star$ and $s \in S'$ then $\delta'(s,\vec{a}) = s_t$; for all $s \in S^\star$, $V'(s) = V(s)$ and $V'(s_t) = \emptyset$. Furthermore, we denote with $M^* = \langle Ag, AP, S', s_I, \{Act_i\}_{i\in Ag}, \{\sim_i'\}_{i\in Ag}, d', \delta', V^*\rangle$ a positive sub-model of M, formally $M^* \subseteq M$, such that: $V^*(s) = V(s)$ and $V^*(s_t) = AP$.

The intuition behind the above sub-models is to remove the imperfect information by replacing each state involved in $\sim$ with a sink state s_t that under (resp., over) approximates the verification of ATL formulas in negative (resp., positive) sub-models. We conclude this part by recalling two preservation results presented in [15].

Lemma 1. *Given a model M, a negative (resp., positive) sub-model with perfect information M' (resp., M^*) of M, and a formula φ of the form $\varphi = \langle\langle A\rangle\rangle\psi$ (or $[\![A]\!]\psi$) for some $A \subseteq Ag$. For any $s \in S' \setminus \{s_t\}$, we have that:*

$$M', s \models \varphi \Rightarrow M, s \models \varphi \qquad\qquad M^*, s \not\models \varphi \Rightarrow M, s \not\models \varphi$$

3 Our procedure

In this section, we provide a procedure to handle games with imperfect information and perfect recall strategies, a problem in general undecidable. The overall model checking procedure is described in Algorithm 1. It takes in input a model M, a formula φ, and a trace h (denoting an execution of the system) and calls the function $Preprocessing()$ to generate the NNF of φ and to replace all negated atoms with new positive atoms inside M and φ. After that, it calls the function $FindSub\text{-}models()$ to generate all the positive and negative sub-models that represent all the possible sub-models with perfect information of M. Then, there is a while loop (lines 4-7) that for each candidate checks the sub-formulas true on the sub-models via $CheckSub\text{-}formulas()$ and returns a result via $RuntimeVerification()$. For additional details on $Preprocessing()$, $FindSub\text{-}models()$, and $CheckSub\text{-}formulas()$ see [15].

Algorithm 1 $ModelCheckingProcedure$ (M, φ, h)

```
1: $Preprocessing(M, \varphi)$;
2: $candidates = FindSub\text{-}models(M, \varphi)$;
3: $finalresult = \emptyset$;
4: while $candidates$ is not empty do
5:     extract $\langle M_n, M_p\rangle$ from $candidates$;
6:     $result = CheckSub\text{-}formulas(\langle M_n, M_p\rangle, \varphi)$;
7:     $finalresult = RuntimeVerification(M, \varphi, h, result) \cup finalresult$;
8: return $finalresult$;
```

Now, we focus on the last step, the procedure $RuntimeVerification()$. It is performed at runtime, directly on the actual system. In previous steps, the sub-models satisfying (resp., violating) sub-properties φ' of φ are generated, and listed into the set $result$. In Algorithm 2, we report the procedure performing runtime verification on the system. Such algorithm gets in input the model M, an ATL property φ to verify, an execution trace h of events observed by executing the system, and the set $result$ containing the sub-properties of φ that have been checked on sub-models of M. First, in lines 2-3, M and φ are updated according to the atoms listed in $result$. This step is used to identify in M and φ which sub-formulas have already been verified through $CheckSub\text{-}formulas()$. The two resulting functions are not reported for space constraints, but their full description can be found in [15]. Note that, $UpdateFormula()$ produces two new ATL formulas (ψ_n, ψ_p), which correspond to the updated version of φ for the negative and positive sub-models, respectively. Once ψ_n and ψ_p have been generated, they need to be converted into their corresponding LTL representation to be verified at runtime. This translation is obtained by removing the strategic operators, while leaving the temporal ones (and the atoms). The resulting two new LTL properties φ_n and φ_p are so obtained (lines 4-5). Finally, by having these two LTL properties, the algorithm proceeds generating (using the standard LTL monitor generation algorithm [3]) the corresponding monitors $Mon^{M'}_{\varphi_n}$ and $Mon^{M'}_{\varphi_p}$. Such monitors are then used by Algorithm 2 to check φ_n and φ_p over an execution trace h given in input. The latter consists in a trace observed by executing the system modelled by M' (so, the actual system). Analysing h the monitor can conclude the satisfaction (resp., violation) of the LTL property under analysis (w.r.t. the model M'). However, only certain results can actually be considered valid. Specifically, when $Mon^{M'}_{\varphi_n}(h) = \top$, or when $Mon^{M'}_{\varphi_p}(h) = \bot$. The other cases, which may include the inconclusive verdict (?), are considered undefined, since nothing can be concluded at runtime. The reason why the conditions in lines 8-9 are enough to conclude $\top$ and $\bot$ directly follow from the following lemmas. The rest of the algorithm is only for storing how the sub-formulas have been verified, whether at runtime (*i.e.*, stored in φ_{rv}), at static time (*i.e.*, stored in φ_{mc}), or not at all (*i.e.*, stored in φ_{unchk}).

Algorithm 2 $RuntimeVerification$ (M, φ, h, $result$)

```
1:  $k = ?$;
2:  $M' = UpdateModel(M, result)$;
3:  $\langle\psi_n, \psi_p\rangle = UpdateFormula(\varphi, result)$;
4:  $\varphi_n = FromATLtoLTL(\psi_n, n)$;
5:  $\varphi_p = FromATLtoLTL(\psi_p, p)$;
6:  $Mon^{M'}_{\varphi_p} = GenerateMonitor(\varphi_p)$;
7:  $Mon^{M'}_{\varphi_n} = GenerateMonitor(\varphi_n)$;
8:  **if** $Mon^{M'}_{\varphi_n}(h) = \top$ **then** $k = \top$;
9:  **if** $Mon^{M'}_{\varphi_p}(h) = \bot$ **then** $k = \bot$;
10: $\varphi_{unchk} = \emptyset$;
11: **for** $\varphi' \in \varphi_{rv}$ **do**
12:     $Mon^{M'}_{\varphi'} = GenerateMonitor(\varphi')$;
13:     **if** $Mon^{M'}_{\varphi'}(h) = ?$ **then** $\varphi_{rv} = \varphi_{rv} \setminus \varphi'$; $\varphi_{unchk} = \varphi_{unchk} \cup \varphi'$;
14: **return** $\langle k, \varphi_{mc}, \varphi_{rv}, \varphi_{unchk}\rangle$;
```

We present the preservations results to provide the correctness of our algorithm.

Lemma 2. *Given a model M and a formula φ, for any history h of M starting in s_I, we have that:*

$$\begin{aligned} Mon_{\varphi_{LTL}}(h) = \top &\implies M, s_I \models \varphi_{Ag} \\ Mon_{\varphi_{LTL}}(h) = \bot &\implies M, s_I \not\models \varphi_{\emptyset} \end{aligned}$$

where φ_{LTL} is the variant of φ where all strategic operators are removed, φ_{Ag} is the variant of φ where all strategic operators are converted into $\langle\langle Ag\rangle\rangle$, $\varphi_{\emptyset}$ is the variant of φ where all strategic operators are converted into $\langle\langle\emptyset\rangle\rangle$.

Due to the limited space, the proof is omitted. It can be found in [16]. However, it is important to evaluate in depth the meaning of the lemma presented above.

Remark 1. Lemma 2 shows a preservation result from RV to ATL* model checking that needs to be discussed. If our monitor returns true we have two possibilities: (1) the procedure found a negative sub-model in which the original formula φ is satisfied then it can conclude the verification procedure by using RV only by checking that the atom representing φ holds in the initial state of the history h given in input; (2) a sub-formula φ' is satisfied in a negative sub-model and at runtime the formula φ_{Ag} holds on the history h given in input. While case (1) gives a preservation result for the formula φ given in input, case (2) checks formula φ_{Ag} instead of φ. That is, it substitutes Ag as coalition for all the strategic operators of φ but the ones in φ'. So, our procedure approximates the truth value by considering the case in which all the agents in the game collaborate to achieve the objectives not satisfied in the model checking phase. That is, while in [5,8] the approximation is given in terms of information, in [4] is given in terms of memory of strategies, and in [15] the approximation is given by generalizing the logic, here we give results by approximating the coalitions. So, the main limitation of our approach concerns this aspect. Furthermore, we recall that our procedure produces always results, even partial. This aspect is strongly relevant in concrete scenario in which there is the necessity to have some sort of verification results. For example, in the context of swarm robots [19], with our procedure we can verify macro properties such as "the system works properly" since we are able to guarantee fully collaboration between agents because this property is relevant and desirable for each agent in the game. The same reasoning described above, can be applied in a complementary way for the case of positive sub-models and the falsity.

Theorem 1. *Algorithm 1 terminates in double-exponential time. Moreover, Algorithm 1 is sound: if the value returned is different from $?$, then $M \models \varphi$ iff $k = \top$.*

Due to the limited space, the proof is omitted (see [16] for details).

4 Our tool

The algorithms presented previously have been implemented in Java[3]. The resulting tool implementing Algorithm 1 allows to extract all sub-models with perfect information

[3] The tool can be found at https://github.com/AngeloFerrando/StrategyRV

that satisfy a strategic objective from a model given in input. The extracted sub-models, along with the corresponding sub-formulas, are then used by the tool to generate and execute the corresponding monitors over a system execution (Algorithm 2). In more detail, as shown in Figure 1, the tool expects a model in input formatted as a Json file. This file is then parsed, and an internal representation of the model is generated. After that, the verification of a sub-model against a sub-formula is achieved by translating the sub-model into its equivalent ISPL (Interpreted Systems Programming Language) program, which then is verified by using the model checker MCMAS[4][20]. This corresponds to the verification steps performed in $CheckSub\text{-}formulas()$ (*i.e.*, where static verification through MCMAS is used). For each sub-model that satisfies this verification step, the tool produces a corresponding tuple; which contains the information needed by Algorithm 2 to complete the verification at runtime. The entire manipulation, from parsing the model formatted in Json, to translating the latter to its equivalent ISPL program, has been performed by extending an existent Java library [9]; the rest of the tool derives directly from the algorithms presented in this paper. The monitors are obtained using LamaConv [22], which is a Java library capable of translating expressions in temporal logic into equivalent automata and generating monitors out of these automata. For generating monitors, LamaConv uses the algorithm presented in [3].

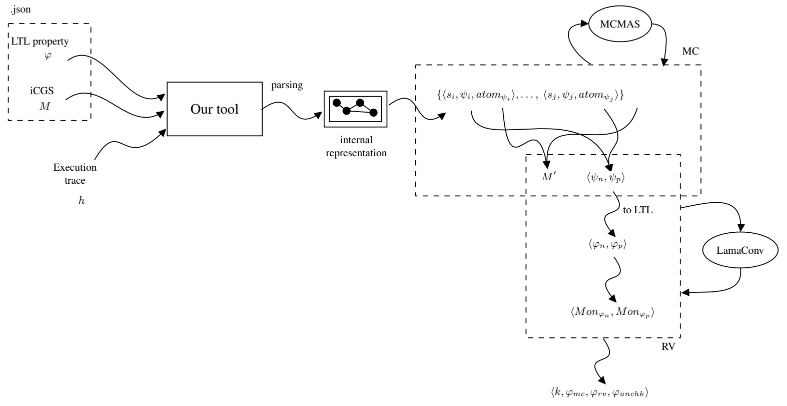
.json
LTL property
φ
iCGS
M
Execution trace
h
Our tool
parsing
internal representation
MCMAS
MC
$\{\langle s_i, \psi_i, atom_{\psi_i}\rangle, \ldots, \langle s_j, \psi_j, atom_{\psi_j}\rangle\}$
M'
$\langle \psi_n, \psi_p \rangle$
to LTL
$\langle \varphi_n, \varphi_p \rangle$
LamaConv
$\langle Mon_{\varphi_n}, Mon_{\varphi_p} \rangle$
RV
$\langle k, \varphi_{mc}, \varphi_{rv}, \varphi_{unchk} \rangle$

Fig. 1: Overview of the implemented tool.

4.1 Experiments

We tested our tool on a large set of automatically and randomly generated iCGSs. The objective of these experiments was to show how many times our algorithm returned a conclusive verdict. For each model, we ran our procedure and counted the number of times a solution was returned. Note that, our approach concludes in any case, but since the general problem is undecidable, the result might be inconclusive (*i.e.*, ?). In Figure 2, we report our results by varying the percentage of imperfect information (x axis) inside the iCGSs, from 0% (perfect information, *i.e.*, all states are distinguishable for

[4] https://vas.doc.ic.ac.uk/software/mcmas/

all agents), to 100% (no information, *i.e.*, no state is distinguishable for any agent). For each percentage selected, we generated 10000 random iCGSs and counted the number of times our algorithm returned with a conclusive result (*i.e.*, $\top$ or $\bot$). As it can be seen in Figure 2, our tool concludes with a conclusive result more than 80% of times. We do not observe any relevant difference amongst the different percentage of information used in the experiments. This is due to the random nature of the iCGSs used. Moreover, the results we obtained depend on the topology of the iCGSs, so it is very hard to precisely quantify the success rate. However, the results obtained by our experiments using our procedure are encouraging. Unfortunately, no benchmark of existing iCGSs exists, thus these results may vary on more realistic scenarios. Nonetheless, considering the large set of iCGSs we experimented on, we do not expect substantial differences.

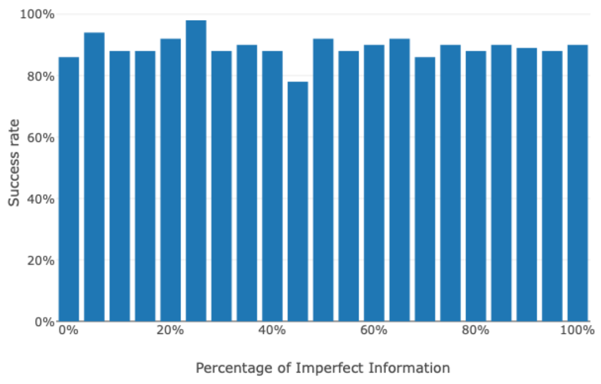

Fig. 2: Success rate of our tool when applied to a set of randomly generated iCGSs.

Other than testing our tool w.r.t. the success rate over a random set of iCGSs, we evaluated the execution time as well. Specifically, we were much interested in analysing how such execution time is divided between $CheckSub\text{-}formulas()$ and Algorithm 2. *I.e.*, how much time is spent on verifying the models statically (through model checking), and how much is spent on verifying the temporal properties (through runtime verification). Figure 3 reports the results we obtained on the same set of randomly generated ICGSs used in Figure 2. The results we obtained are intriguing, indeed we can note a variation in the percentage of time spent on the two phases (y-axis) moving from low percentages to high percentages of imperfect information in the iCGSs (x-axis). When the iCGS is close to have perfect information (low percentages on x-axis), we may observe that most of the execution time is spent on performing static verification (∼70%), which corresponds to $CheckSub\text{-}formulas()$. On the other hand, when imperfect information grows inside the iCGS (high percentage on x-axis), we may observe that most of the execution time is spent on performing runtime verification (∼90% in occurrence of absence of information). This behaviour is determined by the number of candidates extracted by the $FindSub\text{-}models()$ function. When the iCGS has perfect information, such function only extracts a single candidate (*i.e.*, the entire model), since $FindSub$-

$models()$ generates only one tuple. Such single candidate can be of non-negligible size, and the resulting static verification, time consuming; while the subsequent runtime verification is only performed once on the remaining temporal parts of the property to verify. On the other hand, when the iCGS has imperfect information, $FindSub\text{-}models()$ returns a set of candidates that can grow exponentially w.r.t. the number of states of the iCGS. Nonetheless, such candidates are small in size, since $FindSub\text{-}models()$ splits the iCGS into multiple smaller iCGSs with perfect information. Thus, the static verification step is applied on small iCGSs and require less execution time; while the runtime verification step is called for each candidate (so an exponential number of times) and is only influenced by the size of the temporal property to verify.

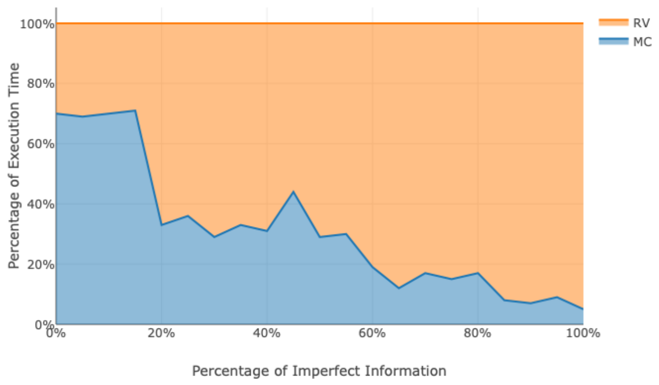

Fig. 3: How the execution time of our tool when applied to randomly generated iCGSs is divided.

In conclusion, it is important to emphasise that, even though the monitor synthesis is computationally hard (*i.e.*, $2EXPTIME$), the resulting runtime verification process is polynomial in the size of the history analysed. Naturally, the actual running complexity of a monitor depends on the formalism used to describe the formal property. In this work, monitors are synthesised from LTL properties. Since LTL properties are translated into Moore machines [3], the time complexity w.r.t. the length of the analysed trace is linear. This can be understood intuitively by noticing that the Moore machine so generated has finite size, and it does not change at runtime.

5 Conclusions and Future work

The work presented in this paper follows a standard combined approach of formal verification techniques, where the objective is to get the best of both. We considered the model checking problem of MAS using strategic properties that is undecidable in general, and showed how runtime verification can help by verifying part of the properties at execution time. The resulting procedure has been presented both on a theoretical (theorems and algorithms) and a practical level (prototype implementation). Note that this is the first attempt of combining model checking and runtime verification to verify strategic properties on MAS. Thus, even though our solution might not be optimal, it is a milestone for the corresponding lines of research. Additional works will be done to improve the technique and, above all, its implementation. For instance, we are planning to extend this work by considering a more predictive flavour.

References

1. Alur, R., Henzinger, T., Kupferman, O.: Alternating-time Temporal Logic. J. ACM **49**(5), 672–713 (2002).
2. Baier, C., Katoen, J.P.: Principles of Model Checking, 2008.
3. Bauer, A., Leucker, M., Schallhart, C.: Runtime Verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14:1–14:64 (2011).
4. Belardinelli, F., Lomuscio, A., Malvone, V.: Approximating Perfect Recall when Model Checking Strategic Abilities. In KR. pp. 435–444 (2018).
5. Belardinelli, F., Lomuscio, A., Malvone, V.: An Abstraction-based Method for Verifying Strategic Properties in Multi-agent Systems with Imperfect Information. In AAAI (2019).
6. Belardinelli, F., Lomuscio, A., Murano, A., Rubin, S.: Verification of Multi-agent Systems with Imperfect Information and Public Actions. In AAMAS 2017. pp. 1268–1276 (2017).
7. Belardinelli, F., Lomuscio, A., Murano, A., Rubin, S.: Verification of Multi-agent Systems with Public Actions Against Strategy Logic. Artif. Intell. **285**, 103302 (2020).
8. Belardinelli, F., Malvone, V.: A Three-valued Approach to Strategic Abilities under Imperfect Information. In KR. pp. 89–98 (2020).
9. Belardinelli, F., Malvone, V., Slimani, A.: A Tool for Verifying Strategic Properties in MAS with Imperfect Information (2020), https://github.com/VadimMalvone/A-Tool-for-Verifying-Strategic-Properties-in-MAS-with-Imperfect-Information
10. Berthon, R., Maubert, B., Murano, A.: Decidability Results for ATL* with Imperfect Information and Perfect Recall. In AAMAS (2017).
11. Berthon, R., Maubert, B., Murano, A., Rubin, S., Vardi, M.Y.: Strategy Logic with Imperfect Information. ACM Trans. Comput. Log. **22**(1), 5:1–5:51 (2021).
12. Dima, C., Tiplea, F.: Model-checking ATL under Imperfect Information and Perfect Recall Semantics is Undecidable. CoRR **abs/1102.4225** (2011).
13. Fagin, R., Halpern, J., Moses, Y., Vardi, M.: Reasoning about Knowledge. MIT (1995).
14. Ferrando, A., Malvone, V.: Strategy rv: A Tool to Approximate ATL Model Checking under Imperfect Information and Perfect Recall. In AAMAS (2021).
15. Ferrando, A., Malvone, V.: Towards the Verification of Strategic Properties in Multi-agent Systems with Imperfect Information. CoRR **abs/2112.13621** (2021).
16. Ferrando, A., Malvone, V.: Towards the Combination of Model Checking and Runtime Verification on Multi-agent Systems. CoRR **abs/2202.09344** (2022).
17. Hinrichs, T.L., Sistla, A.P., Zuck, L.D.: Model Check What You Can, Runtime Verify the Rest. In Voronkov, A., Korovina, M.V. (eds.) HOWARD-60:, pp. 234–244. EasyChair (2014).
18. Jamroga, W., van der Hoek, W.: Agents that Know How to Play. Fund. Inf. **62**, 1–35 (2004)
19. Kouvaros, P., Lomuscio, A.: Parameterised Verification for Multi-agent Systems. Artif. Intell. **234**, 152–189 (2016).
20. Lomuscio, A., Raimondi, F.: Model Checking Knowledge, Strategies, and Games in Multi-agent Systems. In AAMAS. pp. 161–168. (2006).
21. Pnueli, A.: The Temporal Logic of Programs. In SFCS pp. 46–57. (1977).
22. Scheffel, T., et al., M.S.: LamaConv- Logics and Automata converter library (2016).
23. Schobbens, P.: Alternating-Time Logic with Imperfect Recall. ENTCS **85**(2), 82–93 (2004).