Dynamic multi-agent systems: conceptual framework, automata-based modelling and verification

Rodica Condurache[1], Riccardo De Masellis[2], and Valentin Goranko[2,3]

[1] A.I.Cuza University of Iasi, Romania
rodica.b.condurache@gmail.com
[2] Stockholm University, Sweden
{riccardo.demasellis | valentin.goranko}@philosophy.su.se
[3] University of Johannesburg, South Africa (visiting professorship)

Abstract. We study dynamic multi-agent systems (DMASs). These are multi-agent systems with explicitly dynamic features, where agents can join and leave the system during the evolution. We propose a general conceptual framework for modelling such DMASs and argue that it can adequately capture a variety of important and representative cases. We then present a concrete modelling framework for a large class of DMASs, composed in a modular way from agents specified by means of automata-based representations. We develop generic algorithms implementing the dynamic behaviour, namely addition and removal of agents in such systems. Lastly, we state and discuss several formal verification tasks that are specific for DMASs and propose general algorithmic solutions for the class of automata representable DMASs.

1 Introduction

A multi-agent system (MAS) comprises a set of agents acting and interacting in a common arena. Each agent is equipped with a set of available actions and all agents contribute with simultaneously performed actions, thus generating collectively action profiles causing state transitions in the system.

An overall objective in the area of formal methods for multi-agent systems (MASs), is to design frameworks for modelling of the structure and evolutions of such systems and to develop methods for algorithmic verification of properties of such abstract models specified in suitable formal (usually, logic-based) languages.

Traditionally, the set of agents in such frameworks – including the well-known concurrent game structures [2] and interpreted systems [6] – is assumed fixed during the system evolution. In many settings this is an essential limitation, as witnessed by the increasing interest in *parametric* MASs $[1, 10, 12, 5]$ where the set of agents is taken as a parameter and every instantiation of that parameter gives rise to a MAS. Parametric model-checking amounts to verifying a property irrespectively of the specific value of the parameter. However, even in this case, once the parameter is bound to a specific value, it does not change during the system evolution.

1.1 Our contributions

In this work we take a step forward and consider truly *dynamic* MAS (DMASs), where agents can join and leave the system *during the evolution*. From a practical perspective, the dynamic feature naturally arises in many practical scenarios, such as: markets, where brokers joining and leaving the system dynamically impacts the price of goods and shares; sensor/computer networks where global properties of the network depends dynamically on the connections between components and the network topology; manufacturing [3], where adding/removing machines can enable or disable the capability of producing specific products; and others that we discuss further in the paper.

To the best of our knowledge, this is one of the first efforts to study and analyse such systems using formal methods. Thus, as a first main contribution, we present in detail a conceptual formalisation of DMASs. We believe that our framework is general enough to cover many scenarios and we provide several examples to support this claim. We note that, on the one hand, such a general framework provides a broad conceptual characterisation of DMASs, but on the other hand, such a generality makes it quite difficult to develop algorithmic procedures for them.

As a second main contribution, we ground the abstract modelling framework by proposing an automata-based approach for modelling DMASs and show how it provides automated procedures for adding and removing agents. We then providing concrete algorithms for implementing these dynamic features.

As a third main contribution, we formulate the main dynamic reasoning tasks that are specific for DMASs and propose general algorithmic solutions for the class of automata-representable DMASs.

1.2 Related Work

Our interest in parametric systems is mostly in the way they describe interactions among a bounded, but unknown, number of agents. Here we mention a few related works and lines of research.

Among the first works to consider verification of parametric systems is [1], where a counting abstraction is used and decidability of formal properties is achieved by using vector addition systems with states (VASS). In [12] strategic reasoning is considered but only for a restricted set of properties such as reachability, coverability and deadlock avoidance. In order to achieve decidability, assumptions on the system evolutions are made and, in particular, monotonicity with respect to a well-quasi-ordering. In [10] temporal epistemic properties on parameterized interpreted systems are checked irrespective on the number of agents by using cutoff techniques.

The idea of decoupling the interaction between agents from their internal evolution by means of signals/observations is inspired by modular interpreted systems [9], where however, no truly dynamic behaviour is considered, as the set of agents is fixed.

It is also worth mentioning the work in [3] where a sort of dynamic synthesis problem is solved: given a transducer representing a target behaviour to realize, and a set of transducer types, the output is the minimal number of actual transducers which, suitably (asynchronously) orchestrated, can realize the target behaviour. Another related work, though with different motivation and agenda, is [8] where *term-modal logics* are introduced for reasoning about systems with unspecified number of agents.

Other, more closely related works are discussed in Section 3.

2 Framework

This section is devoted to the formalization of a conceptual framework for dynamic MAS (DMAS). For the sake of readability, we first introduce the main components of a DMAS and defer its formal definition to the end of the section. Also, in order to illustrate the introduced concepts for the reader, we make use of a running example.

Example 1. Let us consider manufacturing plants [3] as dynamic multi-agent systems, where pieces of raw materials are processed by machines – the agents – and assembled in final products.

From a high-level perspective, we aim at providing a modular representation of agents in a DMAS, and we adapt concepts and terminology borrowed from object-oriented programming to our setting. To help the reader, we use different fonts to distinguish the introduced concepts.

- `Agent types` represent the abstract behavior of agents as autonomous and stand-alone entities, e.g., `robotManipulator` or `person`.
- *Agent roles* describe the part that an agent type could play within a DMAS, e.g., in the manufacturing plant agent type `robotManipulator` can play the role of a *roboticArm* with a laser as hand effector, or a the type `person` can play the role of *assemblyLineWorker*.
- Agent instances are actual concrete agents in a DMAS, characterized by a type, a role, and a unique identification, e.g., a `robotManipulator` in the role of a roboticArm, installed in a specific location of the plant.

We emphasize that an agent type is not a concrete agent, as it only describe the high-level features that agent instances of that type have. The same reasoning holds for agent roles. The conceptual difference between a type and a roles is that the former is not related to any specific DMAS, while the latter is DMAS-specific. Also, the same agent type (e.g., `person`) can play several roles depending on the DMAS it is employed in (e.g., *assemblyLineWorker* in the manufacturing plant or *trader* in a financial market) and vice-versa, the same role (*assemblyLineWorker*) can be performed by possibly different agent types (`robotManipulator` or `person`). Also, there may be several concrete instances of the same type and role, for example, several roboticArms in the plant. In the rest of the section we formalize such concepts.

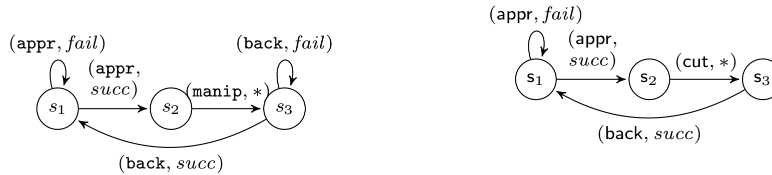
2.1 Agent types

Informally, an agent type describes the abstract behavior of an agent by itself, independently from the system it could be deployed in. Mathematically, it is akin to a Mealy machine, i.e. a finite state automaton, the transition function and outputs of which depends not only on its own actions, but also on the inputs, or *observations*, that come from the system in which it could be deployed to operate.

Definition 1 (Agent type).

$$T = \langle S_T, \mathit{Act}_T, \mathit{avt}_T, \mathcal{O}, \delta_T \rangle \text{ where:}$$

- S_T *is a finite set of* internal *states;*
- Act_T *is a finite set of* action types*;*
- $\mathit{avt}_T : S_T \rightarrow 2^{\mathit{Act}_T} \setminus \emptyset$ *maps to each state a set of available action types;*
- $\mathcal{O}$ *is a set of* observations*;*
- $\delta_T : S_T \times \mathit{Act}_T \times \mathcal{O} \dashrightarrow S_T$ *is the partial transition function such that for each* $s \in S_T$, $\mathit{act} \in \mathit{avt}_T(s)$ *and* $o \in \mathcal{O}$ *we have that* $\delta_T(s, \mathit{act}, o)$ *is defined.*



(a) Graphical representation of agent type `robotManipulator`.

(b) Graphical representation of the automaton instance of type `robotManipulator` with role *roboticArm* and parameter *handEffector* assigned to value *laser*.

Fig. 1: Graphical representation of an agent type and instance.

Example 2. Figure 1a shows a graphical representation of agent type `robotManipulator`. The circles are states and the transition function is rendered by arrows labeled with pairs (`action`, *observation*), where $*$ represent any value. The set of actions is $\mathtt{Act_T} = \{\mathtt{appr}, \mathtt{manip}, \mathtt{back}, \mathtt{noop}\}$. Also, we assume a self-looping transition $(\mathtt{noop}, *)$ in any state, not represented in the picture for the sake of readability. The set of observations is $\mathcal{O} = \{\mathit{fail}, \mathit{succ}\}$. Intuitively, the abstract behavior of `robotManipulator` is as follows. In state s_1 it tries to reach for the object to manipulate by performing action `appr`. Its success in doing so depends on the system it is currently acting: if, e.g., the object is not available or there are obstacles in its path, then it observes *fail*, thus looping in s_1. If it observes *success*, then it successfully moves to s_2, where it can perform

its (still abstract) action of manipulating the object. Finally, in state s_3 it goes back in its original position or not by performing action `back` depending on the observation, analogously to what happens in s_1.

We remark that observations are the interface with the external world and they allow to decouple the internal behavior of the agent from the possible events coming from the outside. To be more precise, we distinguish between *signals* that a DMAS produces, which will be formally defined later, and *observations* that agents have of signals coming from the other agents and the system. In other words, signals are global events that a DMAS generates while observations are the interpretation that agents have of signals. From now on, we denote by Σ the set of signals of a DMAS.

2.2 Agent role, agent instances and instance creation function

Unlike agent types, *roles* are conceptually related to a specific DMAS. Formally, a role is a set of parameter names $R = \{p_1, \ldots, p_m\}$, where each parameter can be thought as a characteristic element, or a feature, assigned to an agent in role R. As it will be shown later on, parameter names are assigned values when creating agent instances.

Example 3. In the manufacturing plant example, the role *roboticArm* is characterized by one parameter name only, *handEffector*. Any instance of *roboticArm* will have a specific value assigned to it, such as *laser* or *rotaryTool*.

When a new agent joins a DMAS, first an agent instance is created for it and then that instance is incorporated into the system. The latter operation is described in the next section, while here we focus on the former. The creation of the instance is an operation specific to each DMAS, called *instance creation function*: it takes as inputs a `type T`, a *role* R and an assignment β of values to parameters in R and returns an agent instance. An agent instance can be thought of as an automaton that *specializes* `T`. Indeed, there is a mapping between states of the automaton instance to states of the automaton type, and analogously for actions.

Definition 2. *An agent instance of type* $T = \langle S_T, Act_T, avt_T, \mathcal{O}, \delta_T \rangle$ *with role* R *and assignment* β *is an automaton:*

$$\mathsf{ag} = \langle \mathsf{S}, \mathsf{Act}, \alpha, \mathsf{avt}, \mathcal{O}, \mathsf{obs}, \delta \rangle \text{ where:}$$

- S *is the set of* local *states;*
- Act *is the set of actions;*
- $\alpha : \mathsf{S} \cup \mathsf{Act} \to S_T \cup Act_T$ *is the abstraction function such that for each* $\mathsf{s} \in \mathsf{S}$ *we have* $\alpha(\mathsf{s}) \in S_T$ *and for each* $\mathsf{act} \in \mathsf{Act}$ *we have* $\alpha(\mathsf{act}) \in Act_T$*;*
- $\mathsf{avt} : \mathsf{S} \to 2^{\mathsf{Act}}$ *is consistent with* avt_T*, namely, for each* $\mathsf{act} \in \mathsf{Act}$ *and* $\mathsf{s} \in \mathsf{S}$*, if* $\mathsf{act} \in \mathsf{avt}(\mathsf{s})$*, then* $\alpha(\mathsf{act}) \in avt_T(\alpha(\mathsf{s}))$*;*
- $\mathsf{obs} : \mathsf{S} \times \Sigma \to \mathcal{O}$ *is the observation function, mapping local states and signals coming from the system to observations;*

– $\delta : \mathsf{S} \times \mathsf{Act} \times \mathcal{O} \dashrightarrow \mathsf{S}$ *it is such that:*
  - *it is defined for each available action in each state, namely for each* $\mathsf{s} \in \mathsf{S}$ *and* $\mathsf{act} \in \mathsf{avt}(\mathsf{s})$ *there exists* $o \in \mathcal{O}$ *such that* $\delta(\mathsf{s}, \mathsf{act}, o)$ *is defined;*
  - *it is consistent with transition function* δ_T *through mapping* α*, viz. if* $\delta(\mathsf{s}, \mathsf{act}, o)$ *is defined and equal to* s'*, then* $\delta_T(\alpha(\mathsf{s}), \alpha(\mathsf{act}), o)$ *is defined and equal to* $\alpha(\mathsf{s}')$*.*

We remark that all components of an agent instance depend not only on the type, but also on the role and assignment. This is hidden in the definition because such a dependency is domain-specific and cannot, in general, be formally generalized.

Example 4. Figure 1b shows a graphical representation of the automaton instance of type `robotManipulator` with role *roboticArm* and parameter *handEffector* assigned to value *laser* in the DMAS manufacturing plant. The abstraction function is such that $\alpha(\mathsf{appr}) = \mathtt{appr}$, $\alpha(\mathsf{cut}) = \mathtt{manip}$ and $\alpha(\mathsf{back}) = \mathtt{back}$. We do not present here the whole observation function but we point out that, for each signal $\sigma \in \Sigma$, it is such that $\mathsf{obs}(\sigma, \mathsf{s}_3) = \textit{success}$. This is because robotic arms in the plant are placed in an obstacle-free area which makes the action of going back in the initial position always successful (as opposed to action appr that can fail if the object to manipulate is not present). For this reason, there is no $(\mathtt{loop}, \textit{fail})$ transition in state s_3. The diligent reader can easily verify that transition function δ satisfies the constraints of Definition 2.

We now have all the concepts needed to define a DMAS.

2.3 Dynamical multi-agent systems

In a DMAS, agents can join and leave during the evolution. From a high-level perspective, at each time instant the system (global) state is characterized by the tuple of local states of the agent instances that are currently part of the system. At each step, either the instances concurrently perform an action that results in the system evolving in a new global state, or a single agent is added/removed to the system. In the first case, the step can be conceptually understood as a sequence of micro-steps:

1. each agent chooses and performs an action, thus generating an *action profile* π (formally defined later);
2. given π and the current global state, the system generates a signal σ;
3. the local state of each agent instance ag is updated according to the observation of σ and ag's own action, which results in the new global state being computed.

When an agent is added, first its instance is generated by calling the system's *instance creation function* described before (by providing a type, a role, and an assignment of values to the role parameters) and then the instance is incorporated into the system by adding its initial state to the current global state. When

an agent is removed from the system, its local state is simply removed from the system's global state.

There is a special agent instance in every DMAS that we call the *arena*: it is of type $\mathtt{T}_0$ and role R_0 and it is *unique*, namely: there is only one instance of type $\mathtt{T}_0$ with role R_0 in every DMAS; no other agent instances are of type $\mathtt{T}_0$ or role R_0 and it cannot be removed, thus it is always present. In order to explain its purpose, let us consider the actual usage we intend to make of dynamic MAS, which is verifying formal properties. In a traditional MAS (global) system states are labeled with atomic propositions, but in our dynamic setting the states themselves are subject to change (as it will be clear later in this section) given that they depend on which, or how many, agents are currently present. Thus, the issue of how to express properties naturally arises, and we solve it by labelling the arena states with atomic propositions. The arena is like any other agent type and role but conceptually represents the ground where the agents play (or equivalently, the behavior of the fixed components that cannot be removed from the system). We note that such arena can be designed so as to incorporate properties that we are interested to check in a DMAS: it is sufficient to design specific arena states that are reached when the (un)desired conditions are met. E.g., in the factory example, we can model arena states that are reached whenever an agent instance fails an action, i.e., when it performs a transition with observation *fail*. Such a choice makes our framework suitable to model the effects that dynamic agents cause on the environment/arena they live in.

A DMAS $\mathfrak{D}$ has the following components:

- a set $\{\mathtt{T}_0, \mathtt{T}_1, \ldots, \mathtt{T}_n\}$ of agent `types`;
- a set $\{R_0, R_1, \ldots, R_m\}$ of *roles*, where each R_i is a set of parameter names for $i \in \{0, \ldots, m\}$;
- a set of *domain* values for each parameter name;
- a finite set Σ of *signals*;
- a function sig mapping system (global) states and action profiles π to signals;
- an *instance creation function*;
- an *add protocol* add;
- a *remove protocol* rem.

We have already introduced the instance creation function: it takes a type $\mathtt{T}_j$ a role R_i and an assignment β for the parameters in R_i and produces an agent instance. More precisely, β assigns to each parameter in R_i a value from its domain. It remains to formalize the function sig and the add and remove protocols.

Signal function. It intuitively defines the core behavior of the system as it tells which signals are produced for every possible joint action of the agent instances currently present in the system, regardless of how many they are and of which type and role. Given that the number of agent instances is unbounded, such a function is usually defined implicitly.

Let ST be the union of local states for every agent instance that may be present in the system. We note that such a set is finite and bounded by the

number of roles, parameters and assignments. We denote $\mathsf{ST}^n := \mathsf{ST} \times ... \times \mathsf{ST}$ (n times) and likewise ACT^n. The function sig^n outputs a signal for n agent instances in any state performing any action, as follows: $\mathsf{sig}^n : \mathsf{ST}^n \times \mathsf{ACT}^n \dashrightarrow \Sigma$. Lastly, we define $\mathsf{sig} := \bigcup_1^\infty \mathsf{sig}^n$ and with this definition at hand, we can now describe the behavior of a DMAS $\mathfrak{D}$ for a specific set of agent instances, which we call a *modular multi-agent structure* (MMAS).

Definition 3 (MMAS). *Let $\mathfrak{D}$ be a* DMAS *and let* $\mathsf{Ag} = \langle \mathsf{ag}_0, \mathsf{ag}_1, \ldots, \mathsf{ag}_x \rangle$ *be a tuple of agent instances where* ag_0 *is the arena and each* $\mathsf{ag}_i \in \mathsf{Ag}$ *is* $\langle \mathsf{S}_i, \mathsf{Act}_i, \alpha_i, \mathsf{avt}_i, \mathcal{O}_i, \mathsf{obs}_i, \delta_i \rangle$. *A modular multi-agent structure for* $\mathfrak{D}$ *with agents* Ag *is the tuple:*

$$\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag}) = \langle Q, \Pi, \Delta \rangle \text{ where:}$$

- $Q \subseteq \mathsf{S}_0 \times \ldots \times \mathsf{S}_x$ *is the set of global states. Given a state* $q \in Q$ *we denote by* $q(i)$ *the state of agent* ag_i.
- $\Pi \subseteq \mathsf{Act}_0 \times \ldots \times \mathsf{Act}_x$. *An* action profile *for* $q \in Q$ *is any* $\pi \in \Pi$ *such that* $\pi(i) \in \mathsf{avt}_i(q(i))$ *for each* $i \in \{0, \ldots, x\}$.
- *Partial transition function* $\Delta : Q \times \Pi \dashrightarrow Q$ *is such that* $\Delta(q, \pi)$ *is defined and equal to* q' *iff:*
  - π *is an action profile for* q *and*
  - *for each* $i \in \{1, \ldots, x\}$ *the next state of agent* ag_i *is given by transition function* δ_i *from:* ag_i *current state* $q(i)$*;* ag_i *performed action* $\pi(i)$ *and from* ag_i *observation of the signal computed by* sig *function (which in turns depends on the whole current system state* q *and action profile* π*). Formally we have that* $q'(i) = \delta_i(q(i), \pi(i), \mathsf{obs}_i(q(i), \mathsf{sig}(q, \pi)))$ *for each* $i \in \{1, \ldots, x\}$.

Note how the evolution of the system is described by means of transition function Δ: the local state of each agent instance is updated according to its current local state, its action and the observation of the signal that is generated by the system, which in turn depends on the whole action profile.

We also remark that a modular multi-agent structure is parameterized by a tuple of agents Ag and thus it suffices to specify how to update such a tuple to handle the joining and leaving of agents. This is precisely the purpose of add and rem protocols.

Add and remove protocols. Let $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ be a MMAS, the protocols define a new MMAS that is the result of adding/removing an agent to/from $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$. We remark that such protocols do not provide yet actual procedures for incorporating or removing an agent to an existing MMAS, as such procedures are domain-specific and may not be even computable, in general. In Section 4 we will be more concrete and will provide an actual algorithm for the add and rem protocols for a specific class of DMAS, namely those composed by using automata-based agent representation and techniques.

Definition 4 (Add protocol). *Let* $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ *be a* MMAS *for a* DMAS $\mathfrak{D}$ *with* $\mathsf{Ag} = \langle \mathsf{ag}_0, \ldots, \mathsf{ag}_x \rangle$ *and let* $q \in Q$ *be the current state of* $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$*. The addition of agent instance* $\mathsf{ag} \notin \mathsf{Ag}$ *is defined by function:*

$$\mathsf{add}(\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag}), q, \mathsf{ag}) \text{ which returns } (\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag}) \oplus \mathsf{ag}, q') \text{ where:}$$

– $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag}) \oplus \mathsf{ag} = \mathcal{G}_{\mathfrak{D}}(\langle \mathsf{ag}_0, \ldots, \mathsf{ag}_x, \mathsf{ag}\rangle)$ *and*
– q' *is the new current state after the addition of* ag *and it is such that* $q'(i) = q(i)$ *for every* $i \in \{0, \ldots, x\}$.

Note that, in general, no constraint is put on the current state of the new agent after the addition. This is because such a state depends on the specific application, but intuitively, it functionally depends on q and ag.

Definition 5 (Remove protocol). *Let* $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ *be a* MMAS *with* $\mathsf{Ag} = \langle \mathsf{ag}_0, \ldots, \mathsf{ag}_x\rangle$ *and let* $q \in Q$ *the current state of* $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$*. The removal of agent* $\mathsf{ag}_i \neq \mathsf{ag}_0$ *is defined by function*

$$\mathsf{rem}(\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag}), q, \mathsf{ag}_i) \text{ which returns } (\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag}) \ominus \mathsf{ag}_i, q') \text{ where:}$$

– $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag}) \ominus \mathsf{ag}_i = \mathcal{G}_{\mathfrak{D}}(\langle \mathsf{ag}_0, \ldots, \mathsf{ag}_{i-1}, \mathsf{ag}_{i+1}, \ldots, \mathsf{ag}_x\rangle)$ *and*
– q' *is the new current state after the removal of* ag_i *and it is obtained from* q *by dropping the element in position* i.

We note that if Ag and Ag' differ only on the order of agents, they still give rise to different $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ and $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag}')$. However, since the behavior of a MMAS does not depend on the order of the agents, we say that $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ and $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag}')$ are order-isomorphic iff Ag is a rearrangement of Ag' (we omit the details for lack of space). With such a notion at hand, we notice that the add and rem protocols enjoy the following property: for every set of agent instances Ag, every agent instance ag_1, ag_2 and every state $q \in Q$, adding an agent and then removing it results in the same MMAS: $\mathsf{rem}(\mathsf{add}(\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag}), q, \mathsf{ag}_1), \mathsf{ag}_1) = (\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag}), q)$.

3 Some concrete examples

In what follows we informally show how to model some existing frameworks as MMASs. The multi-agent systems that we model, although not all necessarily *dynamic*, are characterized by being parametric, namely systems whose evolution can be described symbolically regardless how many concrete agents are part of it. The purpose is solely to show the expressiveness of MMASs, therefore the translations are by no means efficient, nor unique.

Homogeneous Dynamic Multi-Agent Systems (HDMAS*).* We refer to the framework in [4], where homogeneity means that all agents have the same available actions at any given state and the actions have the same effects regardless of which agents perform them. The global state transitions are therefore determined only by the vector of numbers of agents performing each action and are specified symbolically, by means of conditions on these numbers, called *guards*. The internal states of the arena are the global states of the HDMAS. Conceptually, here the arena is passive, meaning that it does not perform actions. Technically, we allow for the same spurious action in every state. All agents are thus represented by the same agent type and the same agent role (with no parameters), i.e. the

set of agents is homogeneous. The internal states of the agent type are again the global states of the HDMAS, whose action availability function follows from the HDMAS and so does the transition function. Signals and observations coincides. We have one for each guard and they are used to tell which guard is satisfied at each step. The global state of the MMAS is a tuple of agents states, but, being agent-homogeneous, such tuples are of the kind $\langle s, s, \ldots, s \rangle$, i.e., composed by the same state. The protocol for adding one agent appends one more element (the current state) to the tuple of the global states and updates the transitions to account for the new agent, according to the HDMAS transition function. The protocol for removing an agent takes no inputs and removes one state from the global MMAS state tuple and updates the transitions according to the HDMAS transition function, analogously to the addition.

Open Multi-agent systems (OMAS*) [11].* These are dynamic MAS where agents can be added or removed during the system evolution by means of special transitions and their behaviour is described by means of automata similar to our agent types. At each time instant, the system is described by the tuple of local states of agents. However, we point out two main differences between OMAS and DMAS. The first concern the way the interaction between a possibly unbounded number of agents is achieved: in OMAS, agents evolutions depends on the projection of the joint action of all agents into a set rather than the joint action itself. Secondly, the (global) transition function of the whole system in OMAS is asynchronous, but synchronization between agents can be achieved by means of special actions. Each OMAS agent type can be modeled as a MMAS agent type. As the OMAS global transition function take into account only if every action has been performed by at least one agent, the corresponding MMAS transition function can be described with a finite number of signals and observations. The add, resp., rem protocols simply add, resp., remove, the agent local state from the tuple of global states.

Population protocols [5]. These are homogeneous asynchronous systems where at each step only a pair of agents change state. The global transition relation thus comprises nondeterministically all possible interactions that can happen at each step among pairs of agents. As agents in a population are homogeneous, we need one agent type and one role only. We also equip the system with actions and signals to "resolve" the intrinsic nondeterminism of the protocol: at each step is the system that decides which pair of agents interact (and thus change state). In order to do so, however, each agent instance in a MMAS should be formally distinguishable, or otherwise the environment cannot specify *which* pair of agents change state (notice indeed that agent instances of the same type and role react to signals in the very same way). We can make agent instances distinguishable from each other by assigning for each of them different values to a special role parameter, although this requires an unbounded domain for that parameter whenever we want to model an unbounded number of agents to join/leave the system.

4 Automata-based representation

Here we use infinite word automata in order to model dynamic MMAS, following the general framework outlined in Section 2.

Definition 6. *A deterministic infinite word automaton is a tuple* $WA = \langle \Gamma, S, \gamma, Acc\rangle$ *where* Γ *is the input alphabet,* S *is the set of states,* $\gamma : S \times \Gamma \to S$ *is the transition function, and* Acc *is the accepting condition.*

For the chosen accepting condition, we denote by $\mathcal{L}(WA, s)$ the set of words accepted by the automaton WA starting from the state s. Here we consider Büchi accepting condition for the automata. We represent an agent as being a deterministic infinite word automaton as follows.

Let $\mathsf{ag} = \langle \mathsf{S}, \mathsf{Act}, \alpha, \mathsf{avt}, \mathcal{O}, \mathsf{obs}, \delta\rangle$ be an agent instance of type $\mathtt{T}$, role R and assignment β. The interaction between the agent and the other agents in the system, as well as the arena, materialises in an infinite sequence of actions of the agent and observations over the signals sent by the system. Then, the automaton WA_{ag} corresponding to the agent ag is such that it accepts all words over actions and observations that describe possible interactions. That is, the alphabet of the automaton is $\Gamma = \mathsf{Act} \times \mathcal{O}$, $S = \mathsf{S} \cup \{\bot\}$ and γ is defined as

$$\gamma(\mathsf{s}, \mathsf{act} \bullet o) = \begin{cases} \delta(\mathsf{s}, \mathsf{act}, o) & \text{if } \mathsf{act} \in \mathsf{avt}(\mathsf{s}) \\ \bot & \text{otherwise} \end{cases}$$

and $\gamma(\bot, \mathsf{act} \bullet o) = \bot$ for any action act and observation o, where $\bullet$ denotes, from now on, the concatenation of two elements or tuples. The set of accepting states is $Acc = S \setminus \{\bot\} = \mathsf{S}$. Note that the state $\bot$ in the automaton mimics the fact that the agent ag played an unauthorized action.

Lemma 1. *The automaton* WA_{ag} *accepts exactly the paths in* ag.

Proof. Note that the transitions of WA_{ag} are equivalent to the ones in ag when the transition function in ag is defined, otherwise the sink state $\bot$ is reached. Therefore, since the accepting condition in WA_{ag} asks to not visit $\bot$ infinitely often, WA_{ag} accepts only executions corresponding to executions in ag.

4.1 Modular multi-agent structures as composition of automata

Let $\mathfrak{D}$ be a dynamic MAS, $\mathsf{Ag} = \langle \mathsf{ag}_0, \mathsf{ag}_1, \ldots, \mathsf{ag}_x\rangle$ be a set of agent instances where ag_0 is the arena and each $\mathsf{ag}_i \in \mathsf{Ag}$ is $\langle \mathsf{S}_i, \mathsf{Act}_i, \alpha_i, \mathsf{avt}_i, \mathcal{O}_i, \mathsf{obs}_i, \delta_i\rangle$. Let moreover $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ the MMAS for Ag and $WA_i = \langle \Gamma_i, S_i, \gamma_i, Acc_i\rangle$ the word automaton for ag_i for $i \in \{1, \ldots, x\}$.

We model the MMAS $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ by the synchronous composition of the automata WA_{ag_i} and we call the resulting automaton $WA_{\mathsf{Ag}} = \langle \Gamma, W, \gamma, Acc\rangle$. Note that $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ is already by definition the synchronous (commutative) product of the agent instances ag from the set Ag of agents. Transitions are labelled with tuples of actions of the agent instances together with signals. Such signals are not

only those generated by tuples of actions of agents in Ag, but they might be generated by other (longer) tuples of actions. This technical solution is used to solve the dynamic verification Problem 5 in Section 5. We therefore say that a signal σ is *compatible* with a state $w \in S_0 \times \ldots \times S_x$ and a tuple of actions $\pi \in \mathsf{Act}_0 \times \ldots \times \mathsf{Act}_x$ if there is some tuple $\pi' \in \mathsf{ACT}^*$ such that $\sigma = \mathsf{sig}(w, \pi \bullet \pi')$. Technically a state of WA_{Ag} is composed of local states of the agent instances together with symbol $\star$ when the state is reached by a signal generated by the currently present agents in Ag, or with symbol $+$ when the state is reached by a (compatible) signal generated when other agents are added. Also, if agent instances do not play available actions or the signal is not compatible with the current state and the actions played, then the next state in the automaton is the sink state $\perp$. In what follows, we use the usual notation $\pi(i)$ for the i-th component of the action profile. More precisely, $WA_{\mathsf{Ag}} = \langle \Gamma, W, \gamma, Acc \rangle$ is defined as follows:

- alphabet $\Gamma = \mathsf{Act}_0 \times \ldots \times \mathsf{Act}_x \times \Sigma$ consisting of action profiles paired with signals;
- the set of states is $W = S_0 \times \ldots \times S_x \times \{\star, +\} \cup \{\perp\}$.
  States in W are denoted by $w \bullet c$, where $w \in S_0 \times \ldots \times S_x$, $w(i)$ is the i-th component of w and $c \in \{\star, +\}$.
- the transition function $\gamma : W \times \Gamma \to W$ is defined as follows:
  - $\gamma(\perp, \pi \bullet \sigma) = \perp$;
  - $\gamma(w \bullet c, \pi \bullet \sigma) = \perp$ if there exists $i \in \{0, \ldots, x\}$ such that $\gamma_i(w(i), \pi(i) \bullet \mathsf{obs}_i(w(i), \sigma)) = \perp$ (instance ag_i is performing a non-available action) or σ is not compatible with w and π;
  - otherwise $\gamma(w \bullet c, \pi \bullet \sigma) = w' \bullet c'$ where $w'(i) = \gamma_i(w(i), \pi(i) \bullet \mathsf{obs}_i(w(i), \sigma))$ for $i \in \{0, \ldots, x\}$ and $c' = \star$ if $c = \star$ and $\sigma = \mathsf{sig}(w, \pi)$ and $c' = +$ otherwise;
- accepting set $Acc = W \setminus \{\perp\}$ consisting of all states but $\perp$.

A *run* in the automaton WA_{Ag} is a (infinite) sequence $\rho = (w_0 \bullet c_0)(\pi_0 \bullet \sigma_0)(w_1 \bullet c_1)(\pi_1 \bullet \sigma_1)(w_2 \bullet c_2)(\pi_2 \bullet \sigma_2)\ldots$ of states and actions and signals such that $c_0 = \star$ and $(w_{i+1} \bullet c_{i+1}) = \gamma(w_i \bullet c_i, \pi_i \bullet \sigma_i)$ for any $i \in \mathbb{N}$. It is then accepted if it visits infinitely often the states in Acc. That is, since the transitions in WA_{Ag} are such that, once in $\perp$ the automaton stays there, the run is accepted if it never visits the state $\perp$.

Adding protocol. Let $WA_{\mathsf{Ag}} = \langle \Gamma, W, \gamma, Acc \rangle$ be the automaton modelling the MMAS $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ for a given set of agents Ag, let $\overline{w} \bullet \star$ be a state in W, and let $WA_{\mathsf{ag}} = \langle \Gamma, S, \gamma, Acc \rangle$ be the automaton for agent ag to be added to WA_{Ag}. The protocol for adding ag is then defined as: $\mathsf{add}(WA_{\mathsf{Ag}}, WA_{\mathsf{ag}}, \overline{w} \bullet \star) = (WA'_{\mathsf{Ag}^+}, \overline{w}' \bullet \star)$ where $WA'_{\mathsf{Ag}^+} = \langle \Gamma', W', \gamma', Acc' \rangle$ is the new word automaton obtained after the addition of WA_{ag}, which intuitively is composed from the (old) word automaton WA_{Ag} by "attaching" to state $\overline{w} \bullet \star$ the word automaton $WA_{\mathsf{Ag} \cup \{\mathsf{ag}\}} = \langle \Gamma_{\mathsf{Ag} \cup \{\mathsf{ag}\}}, W_{\mathsf{Ag} \cup \{\mathsf{ag}\}}, \gamma_{\mathsf{Ag} \cup \{\mathsf{ag}\}}, Acc_{\mathsf{Ag} \cup \{\mathsf{ag}\}} \rangle$, that is, the automaton for agents $\mathsf{Ag} \cup \{\mathsf{ag}\}$. In other words, from state $\overline{w} \bullet \star$ the new automaton WA'_{Ag^+} "jumps" to $WA_{\mathsf{Ag} \cup \{\mathsf{ag}\}}$ in state $\overline{w}' \bullet \star$ and start behaving like it. The new current

state $\overline{w}' = \overline{w} \bullet s$ is obtained by adding a state $s \in S$ of WA_{ag} to the old global state. Notice that the choice of the specific state s is up to the protocol itself, and depends on its inputs. More precisely, $WA'_{\mathsf{Ag}^+} = \langle \Gamma', W', \gamma', Acc' \rangle$ where:

- $\Gamma' = \Gamma \cup \Gamma_{\mathsf{Ag} \cup \{\mathsf{ag}\}}$;
- $W' = W \cup W_{\mathsf{Ag} \cup \{\mathsf{ag}\}}$;
- $\gamma' : (W \cup W_{\mathsf{Ag} \cup \{\mathsf{ag}\}}) \times (\Gamma \cup \Gamma_{\mathsf{Ag} \cup \{\mathsf{ag}\}}) \to (W \cup W_{\mathsf{Ag} \cup \{\mathsf{ag}\}})$ is such that:
  - $\gamma'(w \bullet c, \pi \bullet \sigma) = \gamma(w \bullet c, \pi \bullet \sigma)$ for each w in W and every $\pi \bullet \sigma \in \Gamma$ such that $\gamma(w \bullet c, \pi \bullet \sigma) \neq \overline{w} \bullet \star$;
  - $\gamma'(w \bullet c, \pi \bullet \sigma) = \overline{w}' \bullet \star$ for each w in W and every $\pi \bullet \sigma \in \Gamma$ such that $\gamma(w \bullet c, \pi \bullet \sigma) = \overline{w} \bullet \star$;
  - $\gamma'(w \bullet c, \pi \bullet \sigma) = \gamma(w \bullet c, \pi \bullet \sigma)$ for each w in $W_{\mathsf{Ag} \cup \{\mathsf{ag}\}}$ and every $\pi \bullet \sigma \in \Gamma$.
- $Acc' = WA'_{\mathsf{Ag}^+} \setminus \{\bot\}$.

Intuitively, the jump into the automaton $WA_{\mathsf{Ag} \cup \{\mathsf{ag}\}}$ is modelled by identifying states $\overline{w} \bullet \star$ and $\overline{w}' \bullet \star$ and defining the transitions at that merged state so that the incoming ones are those for state $\overline{w} \bullet \star$ in the automaton WA_{Ag} and the outgoing transitions are those from the state $\overline{w}' \bullet \star$ in $WA_{\mathsf{Ag} \cup \{\mathsf{ag}\}}$.

Removing protocol. The removing protocol is modelled here by projecting away the state of the removed agent ag. Formally, we define the function $\mathsf{rem}(WA_{\mathsf{Ag}}, \mathsf{ag}, \overline{w} \bullet \star) = (WA'_{\mathsf{Ag}^-}, \overline{w}' \bullet \star)$ where, analogously as before, WA'_{Ag^-} is composed from the old automaton WA_{Ag} where we attach to state $\overline{w} \bullet \star$ the word automaton $WA_{\mathsf{Ag} \setminus \{\mathsf{ag}\}}$. Intuitively, WA'_{Ag^-} behaves like WA_{Ag} until state $\overline{w} \bullet \star$ is reached, and then it "jumps" to $WA_{\mathsf{Ag} \setminus \{\mathsf{ag}\}}$ in state $\overline{w}' \bullet \star$ and start behaving like it. More precisely, $WA'_{\mathsf{Ag}^-} = \langle \Gamma', W', \gamma', Acc' \rangle$ where:

- $\Gamma' = \Gamma \cup \Gamma_{\mathsf{Ag} \setminus \{\mathsf{ag}\}}$;
- $W' = W \cup W_{\mathsf{Ag} \setminus \{\mathsf{ag}\}}$;
- $\gamma' : (W \cup W_{\mathsf{Ag} \setminus \{\mathsf{ag}\}}) \times (\Gamma \cup \Gamma_{\mathsf{Ag} \setminus \{\mathsf{ag}\}}) \to (W \cup W_{\mathsf{Ag} \setminus \{\mathsf{ag}\}})$ is such that:
  - $\gamma'(w \bullet c, \pi \bullet \sigma) = \gamma(w \bullet c, \pi \bullet \sigma)$ for each w in W and every $\pi \bullet \sigma \in \Gamma$ such that $\gamma(w \bullet c, \pi \bullet \sigma) \neq \overline{w} \bullet \star$;
  - $\gamma'(w \bullet c, \pi \bullet \sigma) = \overline{w}' \bullet \star$ for each w in W and every $\pi \bullet \sigma \in \Gamma$ such that $\gamma(w \bullet c, \pi \bullet \sigma) = \overline{w} \bullet \star$;
  - $\gamma'(w \bullet c, \pi \bullet \sigma) = \gamma_{\mathsf{Ag} \setminus \{\mathsf{ag}\}}(w \bullet c, \pi \bullet \sigma)$ for each $w \in W_{\mathsf{Ag} \setminus \{\mathsf{ag}\}}$ and every $\pi \bullet \sigma \in \Gamma_{\mathsf{Ag} \setminus \{\mathsf{ag}\}}$;
- $Acc' = WA'_{\mathsf{Ag}^-} \setminus \{\bot\}$.

5 Dynamic verification of MMASs

Modelling dynamic multi-agent systems using automata enables us to solve some relevant and important decision problems that arise in the context of MMASs in a uniform way. Here we state and sketch solutions for the most important ones. All these problems can be solved using automata and game theory approaches. In what follows, we assume properties to be expressed in Linear-time Temporal Logic (LTL) where the atomic propositions are labels on the arena states, given that it is the only agent instance that is always present and never removed.

A classical verification problem is the one asking whether any interaction of the agents present in the model satisfies some required property:

Problem 1 (Verification). Let $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ be a MMAS. Does the MMAS $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ satisfy the property φ at a state q?

Problem 1 was already studied and proved to be solvable in PSPACE [13], thus the same technique can be used here by considering only states of $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ marked with $\star$. In the following, we introduce some problems that are related to the changes that may appear in a MMAS when agents leave or new agents join.

5.1 Addition of agents

Problem 2 (Verification of additions of agents). Let $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ be a MMAS with its current state q and $\mathsf{ag} \notin \mathsf{Ag}$ be an agent instance. Does the addition of the agent ag in $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ at the current state satisfy the property φ?

Theorem 1. *There is an algorithm that solves Problem 2 for all LTL properties in* ExpTime*.*

Proof. The problem is solved using the automata approach outlined here as follows. First, we build the automata WA_{Ag} and WA_{ag} for $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ and ag and then apply the function add to compute the automaton WA'_{Ag^+} and its current state $\overline{w}' \bullet \star$. Also, we set the accepting states for WA'_{Ag^+} as being all states $w \bullet \star$ with $w \in W'$. That is, we only accept executions that correspond to the interaction of agents in the set Ag and call those "good" executions. Then, the problem is reduced to verifying whether $\mathcal{L}(WA'_{\mathsf{Ag}^+}, \overline{w}' \bullet \star) \subseteq \mathcal{L}(\varphi)$ holds. The automaton WA'_{Ag^+} has a size polynomial in the size of the input. However, the verification of the LTL formula φ on all "good" executions takes ExpTime. ☐

Problem 3 (Existence of an agent satisfying a requirement). Let $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ be the MMAS for $\mathfrak{D}$ and agents Ag. Is there an agent $\mathsf{a} \notin \mathsf{Ag}$ such that its addition in the current state $w \bullet \star$ of the MMAS $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ guarantees the property φ?

Theorem 2. *There is an algorithm that solves Problem 3 for all LTL properties in* NExpTime*.*

Proof. There is a nondeterministic algorithm that guesses an agent type $\mathtt{T}$, a role R, and an assignment β, such that for the resulting agent ag we can answer positively Problem 2. Note that, since the set of allowed types, roles, and domains are fixed in the DMAS $\mathfrak{D}$, when the domains are finite there is a finite number of agent instances that may be added to $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$. The complexity bound is a consequence of the nondeterministic choice and the verification in exponential time. ☐

Problem 4 (Satisfiability of a property after addition of an agent). Let $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ be the MMAS for $\mathfrak{D}$ and agents Ag. Is there a state in the MMAS WA_{Ag} and an agent $\mathsf{a} \notin \mathsf{Ag}$ such that the agents in $\mathsf{Ag} \cup \{\mathsf{a}\}$ can cooperate and ensure φ in the resulting MMAS?

Theorem 3. *There is an algorithm that solves Problem 4 for all LTL properties in* NExpTime*.*

Proof. The nondeterministic algorithm first guesses an agent instance (an agent type, a role, and an assignment) and a state s in WA_{Ag}, then it builds the automaton corresponding to the addition of the chosen agent, then guess a path in the resulting automaton and, finally, verifies in ExpTime that it satisfy the LTL condition.

Problem 5 (Stability). Let $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ be the MMAS for $\mathfrak{D}$ and agents Ag. Can the agents in Ag ensure φ whenever an arbitrary number of agents of any type and role are added to the MMAS?

Theorem 4. *There is an algorithm that solves Problem 5 for all LTL properties in* 2ExpTime*.*

Proof. Solving Problem 5 can be reduced to solving a two-players game between a constructor and a spoiler on WA_{Ag}. The constructor proposes actions of the agents and spoiler plays signals. We remark that the spoiler in any state $w \bullet c$ can play signals that leads to a state marked with $+$, say $w' \bullet +$. This intuitively means that there exist agents which, added in $w \bullet c$, can perform actions leading to $w' \bullet +$. The objective of the constructor is therefore to play a strategy profile for the players in Ag such that for any actions of the spoiler, the state $\perp$ is never reached and the LTL formula is satisfied. Note, that this problem is equivalent to LTL synthesis for the agents in Ag against an environment where the objective is $\varphi \wedge \Box\neg\mathsf{ff}$, where φ is the LTL formula given as input, $\Box$ is the "always" modality, and the property ff is true only in the state $\perp$. This problem is solved in 2ExpTime.

5.2 Removal of agents

Problem 6 (Verification of removals of agents). Let $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ be the MMAS for $\mathfrak{D}$ and agents Ag. Does the removal of agent $\mathsf{a} \in \mathsf{Ag}$ from the MMAS WA_{Ag} at its current state ensure the truth of formula φ at that state?

Theorem 5. *There is an algorithm that solves Problem 6 for LTL properties in* ExpTime*.*

Proof. The algorithm runs as follows: first it builds the automaton for the system and then verifies in ExpTime that all its executions satisfy the LTL formula.

Problem 7 (Satisfiability of a property after removal of an agent). Let $\mathcal{G}_{\mathfrak{D}}(\mathsf{Ag})$ be the MMAS for $\mathfrak{D}$ and agents Ag. Is there a state in WA_{Ag} and an agent $\mathsf{a} \in \mathsf{Ag}$ such that its removal from that state guarantees the property φ?

Theorem 6. *There is an algorithm that solves Problem 7 for LTL properties in* NExpTime*.*

Proof. The alorithm guesses the state in WA_{Ag} and the agent to be removed. Then, after building the automaton corresponding to the removal of the chosen agent, it verifies if all paths in it verify the LTL property φ.

6 Concluding remarks

In this work we propose DMAS, a conceptual framework for modeling dynamic MAS the main features of which are being modular and automata-based. The latter enables using techniques and results from automata-based verification. We argue that DMASs are expressive enough to capture a wide range of scenarios and other frameworks in literature. However, we note that not all types of multi-agent systems can be modeled as DMASs. For instance, concurrent game structures (CGS), not being modular, make the removal of agents not generally implementable. Also, in dynamic reactive modules [7] the available actions of a module m depend not only on the state of variables of m, but also on the state of the variables of any other module m' to which m has access. Therefore, the behavior of m modeled as an agent instance may become undefined when module m' is removed, as DMASs do not feature a dynamic "availability function" of actions. Extending our framework to cover such cases is left for future work.

Acknowledgements

The work of Valentin Goranko and Riccardo De Masellis was supported by a research grant 2015-04388 of the Swedish Research Council.

We thank the reviewers for some helpful comments and suggestions.

References

1. Bloem, R., Jacobs, S., Khalimov, A., Konnov, I., Rubin, S., Veith, H., Widder, J.: Decidability in parameterized verification. SIGACT News **47**(2), 53–64 (2016)
2. Bulling, N., Goranko, V., Jamroga, W.: Logics for reasoning about strategic abilities in multi-player games. In: Models of Strategic Reasoning: Logics, Games, and Communities. pp. 93–136 (2015)
3. De Giacomo, G., Vardi, M., Felli, P., Alechina, N., Logan, B.: Synthesis of orchestrations of transducers for manufacturing. In: Proc. AAAI-18. pp. 6161–6168 (2018)
4. De Masellis, R., Goranko, V.: Logic-based specification and verification of homogeneous dynamic multi-agent systems. arXiv:1905.00810 [cs.LO] (2019)
5. Esparza, J., Ganty, P., Leroux, J., Majumdar, R.: Model checking population protocols. In: 36th IARCS Ann. Conf. on FSTTCS. pp. 27:1–27:14 (2016)
6. Fagin, R., Halpern, J., Moses, Y., Vardi, M.: Reasoning about Knowledge. MIT Press: Cambridge, MA (1995)
7. Fisher, J., Henzinger, T.A., Nickovic, D., Piterman, N., Singh, A.V., Vardi, M.Y.: Dynamic reactive modules. In: Proc of CONCUR 2011. pp. 404–418 (2011)
8. Fitting, M., Thalmann, L., Voronkov, A.: Term-modal logics. Studia Logica **69**(1), 133–169 (2001)
9. Jamroga, W., Ågotnes, T.: Modular interpreted systems. In: Proceedings of AAMAS. pp. 131:1–131:8. ACM (2007)
10. Kouvaros, P., Lomuscio, A.: Parameterised verification for multi-agent systems. Artif. Intell. **234**, 152–189 (2016)
11. Kouvaros, P., Lomuscio, A., Pirovano, E., Punchihewa, H.: Formal verification of open multi-agent systems. In: Proc. of AAMAS-19. pp. 179–187 (2019)
12. Raskin, J., Samuelides, M., Van Begin, L.: Games for counting abstractions. Electr. Notes Theor. Comput. Sci. **128**(6), 69–85 (2005)
13. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. J. ACM **32**(3), 733–749 (Jul 1985)