SMT-based Safety Checking of Parameterized Multi-Agent Systems

Paolo Felli, Alessandro Gianola, Marco Montali
Free University of Bozen-Bolzano, Bolzano, Italy
{pfelli,gianola,montali}@inf.unibz.it

Abstract

We study the problem of verifying whether a given parameterized multi-agent system (PMAS) is safe, namely whether none of its possible executions can lead to bad states. These are captured by a state formula existentially quantifying over agents. As the MAS is parameterized, it only describes the finite set of possible agent templates, while the actual number of concrete agent instances that will be present at runtime, for each template, is unbounded and cannot be foreseen. We solve this problem via infinite-state model checking based on satisfiability modulo theories (SMT), relying on the theory of array-based systems. We formally characterize the soundness, completeness and termination guarantees of our approach under specific assumptions. This gives us a technique that is implementable on top of third-party, SMT-based model checkers. Finally, we discuss how this approach lends itself to richer parameterized and data-aware MAS settings beyond the state-of-the-art solutions in the literature.

Introduction

The automated verification of Multi-Agent Systems (MASs) typically amounts to check the existence of execution strategies for the achievement of given goals, or to compute counterexamples as evidence of points of potential failure. Model checking (Clarke et al. 2018) is one of the most common approaches to the verification of MASs, often with a focus on strategic abilities (Bulling, Goranko, and Jamroga 2015). However, a common limitation in this literature is the assumption that the system is finite-state and fully specified, which in many applications requires to abstract or propositionalize crucial system features. Other approaches have thus tackled the verification of MASs in settings that are intrinsically infinite-state (Esparza et al. 2017), for which explicit model-checking techniques cannot be used off-the-shelf. In these settings, the model of the MAS is partially specified or some sort of data component is present.

A notable example is that of parameterized MASs (PMASs), addressed by a growing literature (Bloem, Jacobs, and Khalimov 2015; Kouvaros and Lomuscio 2016, 2017; Esparza et al. 2017; De Masellis and Goranko 2020). In PMASs, the number of agents is *unbounded* and *unknown*, so that possibly infinitely many concrete MASs must be considered: the task is to check whether a property is satisfied by any (or all) concrete MASs that adhere to a fixed behavioral structure (such as a known catalogue of possible agent *templates*), without fixing their number a priori.

In particular, the property we consider is that of safety: a PMAS is said to be *safe* if bad states, characterized by a state formula (existentially quantifying on agents), can never be reached. Crucially, this must be verified irrespective of *(i)* the number of agent instances that will be present at runtime, *(ii)* the possible interactions they may have and executions they may induce and *(iii)* the initial setup of the PMAS, i.e., the possible initial interpretation of a read-only, relational data store that is used by PMASs for background data.

Safety checking is a key property not only of MASs but of dynamic systems in general, with a long-standing tradition (see, e.g., (Abdulla et al. 1996; Bloem, Jacobs, and Khalimov 2015)). Applications range from verification of protocols for swarms to applications for Industry 4.0 and 'cloud manufacturing', where an unbounded number of manufacturing resources must be considered (Alechina et al. 2019).

In this paper we illustrate our theoretical result based on the application of satisfiability modulo theories (SMT) (Barrett and Tinelli 2018) techniques for array-based systems (ABSs) (Ghilardi et al. 2008; Ghilardi and Ranise 2010a; Alberti, Ghilardi, and Sharygina 2017; Calvanese et al. 2020), and we characterize its soundness and completeness (and decidability of the task). Specifically, we do so for the notable case of interleaved PMASs. This gives us the theoretical foundation for a practical implementation based on state-of-the-art SMT-based model-checkers, which is showcased in (Felli, Gianola, and Montali 2020b) for a class of PMASs more restricted than the one we study here. The implementation is available at (Felli, Gianola, and Montali 2020a).

The paper is organized as follows. We first state the contributions and discuss related literature. Then, we introduce PMASs, their interleaved execution semantics, and the (un)safety checking problem. We then encode PMASs into ABSs, which allows us to apply SMT-based techniques for their verification. We also show the theoretical guarantees of our verification procedure. Finally we discuss future work.

A simple example. Consider a scenario in which a swarm of robotic agents intends to reach a protected room denoted as `c`. To do so, from their initial location they have to first

move to a room A, then to a room B, then to C. The doors in corridors between these locations are either open or closed. There is no way a robot can move past a closed door, and the information about the status of these doors is not known nor controllable. Moreover, a security system prohibits to move from room B to room C unless the system is first switched off by interacting with a control panel in room A. However, by moving from A to B, the security system automatically turns on again. Moreover, in room C a further security system, when armed, activates an electromagnetic pulse (EMP) to disable robots in that room, and it is armed whenever C is entered. The pulse becomes unarmed after use, but always disables at least one robot. We want to check whether it is possible, after the EMP is activated (hence after a robot entered room C), that there can be robots in C that are not disabled. By careful analysis, we can see that the answer is positive if all doors are open: at least two robots need to move to A, then B. Then, a least one further robot must disable the security system by moving to A and using the control panel in that room. After this, the other robots can move from B to C (this action will arm the EMP). At this point the EMP will be fired in room C, but there are chances that one of the robots will not be disabled, although this is not guaranteed. Note how the existence of this *witness of unsafety* relies on the ability of considering that more robots are at the same time in C, which is not something that is required by the protocol of robots (a protocol describes the executability of actions).

In this paper we address this type of scenarios, and we discuss at the end of the paper how our technique can be directly extended to account for a number of features left as future work. These include, e.g., considering a number of alternative execution semantics (which is possible within this framework *as it is*), and most importantly an extension with read/write relational databases that the agents can use during execution for storing data and exchanging messages.

Related Work and Contribution

Our work is related to the existing literature on parameterized verification, which has however a number of differences with respect to our approach. The verification problem for parameterized systems has been studied extensively, and a number of decidability results are known for various kinds of specifications. For example, it is decidable for forms of regular specifications (Esparza et al. 2017) but undecidable even for stuttering-insensitive properties such as LTL$\setminus$X formulae (Emerson and Kahlon 2003) if asynchronous rendezvous is allowed. As summarized in (Bloem, Jacobs, and Khalimov 2015), decidability results for the verification of parameterized systems are based on reduction to finite-state model checking via abstraction (Pnueli, Xu, and Zuck 2002; John et al. 2012), *cutoff* computations (a bound on the number of instances that need to be verified (Emerson and Namjoshi 2003)), or by proving that they can be represented as well-structured transition systems (Finkel and Schnoebelen 2001; Abdulla et al. 1996). Our verification technique, although limited to safety, is not based on (predicate and counter) abstractions, cutoffs nor reductions to finite-state model checking. Also, the MASs we consider do not assume a particular topology, and the types of disjunctive guards considered in (Emerson and Kahlon 2003) are here extended toward a FO setting by also allowing relation symbols (please refer to the section on future work for more comments on this point). The approach builds on the model-theoretic framework of ABSs (Ghilardi and Ranise 2010a; Calvanese et al. 2020) and can be seen as a declarative, first-order counterpart of theories of well-structured transition systems for which compatible results are known (e.g., (Abdulla et al. 1996; Bloem, Jacobs, and Khalimov 2015)).

Regarding specifically the verification of PMASs, the closest model are those of (Kouvaros and Lomuscio 2016, 2017; Belardinelli, Kouvaros, and Lomuscio 2017) and open MASs (De Masellis and Goranko 2020; Kouvaros et al. 2019). In (De Masellis and Goranko 2020), the authors study homogenous, dynamic MASs that are analogous to our definition of PMASs. There, agents join and leave dynamically during the execution and are partitioned into controllable and uncontrollable, so that the main task is to verify strategic properties of coalitions of at least n controllable agents against coalitions of at most m uncontrollable ones. While a mechanism for joining/leaving the system can be captured natively in our formalization of PMASs, our approach is not currently able to verify strategic, temporal properties.

The framework of (Kouvaros et al. 2019; Kouvaros and Lomuscio 2016) and related papers is also similar. Again, compared with that work, we restrict to the key task of checking safety, instead of tackling model-checking of modal specifications. Their results depend on the chosen execution semantics, hence on the combinations of possible action types that are allowed. The same holds for our technique: although not included in this paper, our approach can be directly extended (with the same soundness and completeness guarantees) to the further variants of execution semantics studied there, with the exception of those involving the global execution of synchronous action types. For these, which we leave as future work due space limitations, we can guarantee a weaker notion of soundness (see (Felli, Gianola, and Montali 2020c)), consistently with the results in (Kouvaros and Lomuscio 2016). Crucially, however, their verification procedure requires to identify a cutoff even for the PMASs considered in this paper (analogous to the class of systems there called $\mathbb{SMR}$): if a cutoff is found then the verification result is correct, otherwise the procedure halts with no result (hence, the procedure is sound but not complete). The existence of a cutoff depends on the existence of a simulation property (between the agent templates and the environment) to be checked on the abstract system, which has to be computed first. Conversely, our technique does not require cutoffs: we are able to characterize our results on soundness and completeness for the execution semantics adopted here, while termination (thus a complete decision procedure) can be directly obtained by a *syntactic* property of the action guards and goal formula.

By departing from the literature above, we present here a verification technique grounded in an SMT-based (Barrett and Tinelli 2018) approach for ABSs (Ghilardi et al. 2008; Ghilardi and Ranise 2010a; Alberti, Ghilardi, and Sharygina 2017; Calvanese et al. 2020). This is a very well-

understood SMT-based formalism for which a number of results of practical applicability already exist (Calvanese et al. 2019c,a; Ghilardi et al. 2020). Our approach is the first to establish a theoretical connection between the verification of PMASs and the long-standing tradition of SMT-based model checking for ABSs; moreover, our decidability result, although inspired by an analogous condition in (Calvanese et al. 2020), is a novel contribution for ABSs as well. This result is exploited operationally, by encoding safety-checking of PMASs into the general-purpose model checker MCMT (Ghilardi and Ranise 2010b).

PMASs: parameterised MASs

Let Θ be a set of (semantic) data types (e.g., reals, integers, booleans). Each type $\theta \in \Theta$ comes with a (possibly infinite) domain Δ_θ, and a type-wise equality operator $=_\theta$ (we simply write $=$ when the type is clear). Let $\mathcal{R}$ be a set of relations over Θ, which we treat as *uninterpreted* relations (i.e. simple relation symbols), used to model background information in the MAS which is *never updated during the execution*, thus representing a *read-only* component. E.g., the information about corridors and doors in the example can be modeled via these relations. We consider the usual notion of FO interpretations $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ with $\Delta^{\mathcal{I}} = \bigcup_\Theta \Delta_\theta$ and $\cdot^{\mathcal{I}}$ is an FOL interpretation function for symbols in $\mathcal{R}$.

Definition 1. *An **agent template** is a tuple $T = \langle ID, L, l^0, V, type, val, \mathcal{A}^{loc}, \mathcal{A}^{syn}, P, \delta\rangle$ composed of:*

- *an infinite set ID of unique agent identifiers of sort* ID*;*
- *a finite set L of local states, with initial state $l^0 \in L$;*
- *a finite set V of local (i.e., internal) agent state variables;*
- *a variable-type assignment $type : V \mapsto \Theta$;*
- *a variable-value assignment $val : L \times V \mapsto \bigcup_\Theta \Delta_\theta$, with $val(l, v) \in \Delta_\theta$ for $\theta = type(v)$;*
- *a non-empty, finite set of action symbols $\mathcal{A} \doteq \mathcal{A}^{loc} \cup \mathcal{A}^{syn}$ (described later), s.t. $\mathcal{A}^{loc} \cap \mathcal{A}^{syn} = \emptyset$;*
- *a protocol function specifying the conditions under which each action is* executable*. It is a function $P : \mathcal{A} \mapsto \Psi$, where Ψ are agent formulae, defined below, "querying" the current state of the whole PMAS;*
- *a total transition function $\delta : L \times \mathcal{A} \mapsto L$, describing how the local state is affected by the execution of an action α: the template moves from a state l to a state l' when executing an action α iff $\delta(l, \alpha) = l'$, also denoted $l \xrightarrow{\alpha} l'$.*

The **environment template** is a special agent template T_e with fixed identifier (i.e., $ID = \{e\}$): there is exactly one environment. Intuitively, a **(concrete) agent** is a triple composed of an agent ID, its template and its current local state. Analogously, a **(concrete) environment** is a pair consisting of the template T_e and its current state.

Let $\{T_1, \ldots, T_n, T_e\}$ be a set of agent (and environment) templates, with $T_t = \langle ID_t, L_t, l^0_t, V_t, type_t, val_t, \mathcal{A}^{loc}_t, \mathcal{A}^{syn}, P_t, \delta_t\rangle$ for $t \in \{1, \ldots, n, e\}$. We denote a concrete agent with ID j, template T_t and local state l_j by writing $\langle j, l_j\rangle_t$, and similarly we denote the concrete environment by $\langle e, l_e\rangle_e$. We also denote a vector of k such concrete agents of template T_t as $\langle \vec{I}, \vec{L}\rangle_t$, where $\vec{I} \in ID^k_t$ and $\vec{L} \in L^k_t$ are vectors of IDs and local states. We assume unique agent IDs and disjoint template variables, i.e., $ID_t \cap ID_{t'} = \emptyset$ and $V_t \cap V_{t'} = \emptyset$ for $t, t' \in \{1, \ldots, n, e\}$, $t \neq t'$.

A **PMAS** is a tuple $\mathcal{M} = \langle \{T_1, \ldots, T_n\}, T_e, \mathcal{R}\rangle$ consisting of n agent templates, one environment template and the relations. Note that a PMAS specifies the initial local state of all agents for each template, but *does not specify how many concrete agents exist for each template*. A **configuration** is a tuple $g = \langle \{\langle \vec{I}_1, \vec{L}_1\rangle_1, \ldots, \langle \vec{I}_n, \vec{L}_n\rangle_n\}, \langle e, l_e\rangle_e\rangle$, which thus identifies the number of agent instances (the size of each $\vec{I}_t$, $t \in [1, n]$, may differ). A configuration is *initial* iff all agents are in their initial local state. Infinite possible initial configurations exist, since the number of concrete agents is unbounded. As shorthand, we denote the local state l_j of agent $\langle j, l_j\rangle_t$ in configuration g as $g.j$, thus writing $\langle j, g.j\rangle_t$.

We now define the **agent formulae** used for protocols in Def. 1 as quantifier-free formulae $\psi(\underline{j}, \mathit{self}, e, \underline{v})$ where $\underline{j}$ are the free variables of sort ID, *self* is a special constant used to denote the current agent, e is the special ID (constant) of the concrete environment, $\underline{v}$ are template variables (for any template). These follow the grammar:

$$\psi \doteq (v^{[j]} = k) \mid (v_1^{[j_1]} = v_2^{[j_2]}) \mid \\ \mid R(x_1, \cdots, x_m) \mid j_1 = j_2 \mid \neg\psi \mid \psi_1 \vee \psi_2$$

where $v, v_1 \in V_t$, $v_2 \in V_{t'}$ for templates $t, t' \in \{1, \ldots, n, e\}$, k is a constant in Δ_θ for $\theta = type_t(v)$, R is a relation symbol in $\mathcal{R}$ of arity $m \geq 1$ (over types $\theta_1, \ldots, \theta_m$), each x_i is either a variable $v^{[j]}$ of some template or a constant $k_i \in \Delta_{\theta_i}$, and j, j_1, j_2 are either the variables of sort ID in $\underline{j}$ or the constant *self* or the ID constant e for the environment (with a little abuse of notation, in this paper we use symbols j to denote variables of sort ID or concrete IDs). The usual logical abbreviations apply. Note that, differently from the restricted model in (Felli, Gianola, and Montali 2020b), terms of the form $R(x_1, \cdots, x_m)$ can mention more than one variable of sort ID and make comparisons of the form $v_1^{[j_1]} = v_2^{[j_2]}$, possibly with $j_1 \neq j_2$. This detail will play an important role for our termination guarantees (see Thm. 2).

Intuitively, agent formulae are implicitly quantified existentially over agent IDs. As we will formalize next, they allow to test (dis)equality of agent variables w.r.t. other agent variables and agent constants (IDs), and to check whether a tuple is in a relation (whose elements are agent variables or constants). For instance, $(v^{[j]} = k)$ informally means that there exists an agent ID j so that $v = k$ for such an agent.

An **ID grounding** of a formula $\psi(\underline{j}, \mathit{self}, e, \underline{v})$ in g is an assignment σ which assigns each variable j of sort ID in $\underline{j}$, and the constant *self*, to a concrete agent ID in g (denoted $\sigma(j)$ and $\sigma(\mathit{self})$, resp.). It also assigns the constant e to itself, i.e. $\sigma(e) = e$. Intuitively, for a formula to be true in g, one needs to find a suitable σ that makes the formula true.

Definition 2. *Given an interpretation $\mathcal{I}_0$, a configuration g satisfies a formula ψ under $\mathcal{I}_0$, denoted $g \models_{\mathcal{I}_0} \psi$, iff there exists an* ID *grounding σ of ψ in g s.t. $g, \sigma \models_{\mathcal{I}_0} \psi$, with:*

- *$g, \sigma \models_{\mathcal{I}_0} (v^{[j]} = k)$ iff $val_t(g.\sigma(j), v) = k$, where $v \in V_t$; i.e. the concrete agent $\langle \sigma(j), g.\sigma(j)\rangle_t$, i.e. with* ID *$\sigma(j)$ and template T_t, is so that $v = k$;*

- $g, \sigma \models_{\mathcal{I}_0} (v_1^{[j_1]} = v_2^{[j_2]})$ *iff* $val_t(g.\sigma(j_1), v_1) = val_{t'}(g.\sigma(j_2), v_2)$*, where* $v_1 \in V_t$*,* $v_2 \in V_{t'}$*; i.e. the agents* $\langle \sigma(j_1), g.\sigma(j_1)\rangle_t$ *and* $\langle \sigma(j_2), g.\sigma(j_2)\rangle_{t'}$ *with* IDs $\sigma(j_1)$*,* $\sigma(j_2)$ *and templates* T_t *and* $T_{t'}$*, are so that* $v_1 = v_2$*;*
- $g, \sigma \models_{\mathcal{I}_0} R(x_1, \cdots, x_m)$ *iff* $R^{\mathcal{I}_0}(y_i, \ldots, y_m)$*, where* y_i *is as follows for each* $i \in [1, m]$*. If* x_i *is a constant* k*, then* y_i *is* k*; if instead* x_i *is a variable* $v^{[j]}$ *with* $v_i \in V_t$ *for some template* $t \in \{1, \ldots, n, e\}$ *then* y_i *is* $val_t(g.\sigma(j), v)$*. Intuitively,* R *holds under* $\mathcal{I}_0$ *for the constants and values of variables, where the value of* $v^{[j]}$ *is taken from the local state of agent with* ID $\sigma(j)$*;*
- $g, \sigma \models_{\mathcal{I}_0} (j_1 = j_2)$ *iff* $\sigma(j_1) = \sigma(j_2)$*;*
- $g, \sigma \models_{\mathcal{I}_0} \neg\psi$ *iff* $g, \sigma \not\models_{\mathcal{I}_0} \psi$*;*
- $g, \sigma \models_{\mathcal{I}_0} \psi_1 \vee \psi_2$ *iff* $g, \sigma \models_{\mathcal{I}_0} \psi_1$ *or* $g, \sigma \models_{\mathcal{I}_0} \psi_2$*.*

Note that *self* is freely assigned to an agent ID: if g satisfies a formula with *self*, then an agent exists that can be taken as *self*. Hence we write $g \models^j_{\mathcal{I}_0} \psi$, if needed, to denote that there exists σ with $\sigma(\mathit{self}) = j$ so that $g, \sigma \models_{\mathcal{I}_0} \psi$. This informally reads as ψ *is true in* g *for agent with* ID j. E.g., assuming g is s.t. $v_1 = 6$ for agent with ID 3, and $v_1 = 5$ for agent with ID 7, then $g \models^3_{\mathcal{I}_0} (v_1^{[\mathit{self}]} = 6) \wedge (v_1^{[j]} = 5)$.

A **(global) transition** of a PMAS describes its evolution when a vector of actions $\vec{\alpha}$ (one for each concrete agent and one for the environment) are executed from configuration $g = \langle \{\langle \vec{I}_1, \vec{L}_1\rangle_1, \ldots, \langle \vec{I}_n, \vec{L}_n\rangle_n\}, \langle e, l_e\rangle\rangle$, so that a new configuration of the form $g' = \langle \{\langle \vec{I}_1, \vec{L}'_1\rangle_1, \ldots, \langle \vec{I}_n, \vec{L}'_n\rangle_n\}, \langle e, l'_e\rangle\rangle$ is reached. This is denoted by $g \xrightarrow{\vec{\alpha}} g'$.

Since each concrete agent and the environment may either perform an action (in $\mathcal{A}_t^{loc} \cup \mathcal{A}^{syn}$) or remain idle, multiple executions semantics can be defined, depending on the constraints we impose on $\vec{\alpha}$, and in this paper we consider one of these, i.e., *interleaved* MASs. First, we describe the set $\mathcal{A}_t^{loc} \cup \mathcal{A}^{syn}$ used in Def. 1.

Symbols in $\mathcal{A}_t^{loc}$, for each t, are called **local actions**, and those in $\mathcal{A}^{syn}$ **synchronization actions**. Actions in $\mathcal{A}_t^{loc}$ can only affect the local state of the concrete agent which executes them, whereas actions in $\mathcal{A}^{syn}$ represent the synchronization between one or more agents and the environment and thus can affect the local state of each agent involved. Intuitively, synchronization actions in $\mathcal{A}^{syn}$ are used to model explicit communication actions or any action with effects that are not private to the single agent or to the environment.

Therefore, not every vector $\vec{\alpha}$ is meaningful: typically, one wants to constrain the possible evolutions so that synchronization actions and local actions do not happen at the same time, so that we can distinguish those steps in which the PMAS evolves in response to public actions, events, messages, from those in which agents update their local state.

Interleaved PMASs. They are characterized by the following notion of legal transition. First, we consider a special no-op action nop so that $\delta_t(l, nop) = l$ for each template t and local state l. Hence, at each step, either: *(i)* a (possibly not proper) *subset* of concrete agents and the environment perform a (non nop) action in $\mathcal{A}_t^{loc}$ on their local state or *(ii)* the environment and a *subset* of the agents synchronize by executing the same action in $\mathcal{A}^{syn}$. Local (non-nop) and synchronization actions are not mixed. $\vec{\alpha}.j$ denotes the action of the agent with ID j, or of the environment if $j = e$.

Definition 3. *For an interpretation* $\mathcal{I}_0$*,* $g \xrightarrow{\vec{\alpha}} g'$ *is* ***legal*** *iff:*

- $g'.j = \delta_t(g.j, \vec{\alpha}.j)$ *for every* $\langle j, g.j\rangle_t$*,* $t \in \{1, \ldots, n, e\}$*, i.e., agents and environment evolve as per their template;*
- $g \models^j_{\mathcal{I}_0} P_t(\vec{\alpha}.j)$ *for every* $\langle j, g.j\rangle_t$*, i.e., each action is executable and self is replaced by* j *for evaluating protocols;*
- *either only local actions are executed (by some agents and the environment), or the environment and at least one agent synchronize through action* $\alpha \in \mathcal{A}^{syn}$*. All other agents perform nop. Formally, either:*
  - *no* j *exists so that* $\vec{\alpha}.j \in \mathcal{A}^{syn}$*, that is, no synchronization action is executed; or*
  - *the environment and at least one agent synchronize, while other agents can either synchronize as well or freely decide to remain idle. Formally,* $\vec{\alpha}.e = \alpha \in \mathcal{A}^{syn}$ *and* $i \neq e$ *exists with* $\vec{\alpha}.i = \alpha$*, while* $\vec{\alpha}.j \in \{\alpha, nop\}$ *and* $g \models^j_{\mathcal{I}_0} P_t(\vec{\alpha}.j)$ *for every* $\langle j, g.j\rangle_t$*.*

Example 1. *In the scenario, we use a template* T_{att} *for robots, with variables* `room` *(enumeration* `[init,A,B,C]`*) and* `disabled` *(boolean). For the environment template* T_e*,* `secON` *and* `armed` *are used for specifying whether the security system is active, and whether the pulse is armed. Template* T_{att} *has actions* `goA`*,* `goB` *and* `goC`*, plus additional actions* `off` *and* `pulse` *representing the action of switching off the security system and of "being disabled" by the pulse. The environment also has actions* `goB`*,* `goC` *and* `off`*, as the robot template. Indeed, these are synchronization actions which have an effect on the environment as well: respectively, the security system is re-activated, the EMP is armed, the security system is disabled. The fact that corridors between rooms are either open or closed is captured by elements in a binary relation over rooms (e.g.* `Corr(A,B)`*). As we quantify over initial interpretations, this captures the fact that the status of doors is not known, but all possibilities are considered.*

For example, the protocol of action `goC` *in* T_{att} *is* $\mathtt{room}^{[\mathit{self}]} = \mathtt{B} \wedge \mathtt{secON}^{[e]} = \mathtt{false} \wedge \mathtt{Corr}(\mathtt{B}, \mathtt{C})$*. Note that it is not required to specify the template of variables, as these sets are disjoint. Therefore, given a global state in which an agent instance of* T_{att} *(i.e., a robot) is in local state* l *and* T_e *is in local state* l_e*, the robot can execute a transition* $l \xrightarrow{\mathtt{goC}} l'$ *only if* `Corr(B,C)` *is in* $\mathcal{I}_0$*,* $val_{att}(l, \mathtt{room}) = \mathtt{B}$ *and* $val_e(l_e, \mathtt{secON}) = \mathtt{false}$ *in* T_e*. The resulting local state* l' *of the robot is such that* $val_{att}(l', \mathtt{room}) = \mathtt{C}$ *while the environment reaches a local state* l'_e *so that* $val_e(l'_e, \mathtt{armed}) = \mathtt{true}$ *(plus further assignments for inertia). Other actions are defined in a similar manner. E.g., the protocol of* `off` *in* T_{att} *is* $\mathtt{room}^{[\mathit{self}]} = \mathtt{A}$*. The formula expressing the unwanted condition is given at the end of this section.* □

Runs and the Safety Checking Problem. Based on the one-step definition of (legal) global transition, we now define the notion of runs for interleaved PMASs. Given a PMAS $\mathcal{M} = \langle \{T_1, \ldots, T_n\}, T_e, \mathcal{R}\rangle$, a **(global) run** is a pair $\langle \rho, \mathcal{I}_0\rangle$ where ρ is a sequence $\rho = g^0 \xrightarrow{\vec{\alpha}^1} g^1 \xrightarrow{\vec{\alpha}^2} \cdots$ and $\mathcal{I}_0$ is an interpretation for relation symbols as before. We restrict to runs that *(i)* are legal and *(ii)* start from an initial configuration, i.e., with all concrete agents in their initial lo-

cal state. A transition as above specifies how each concrete agent $g.j$ evolves depending on the nature of the action $\vec{\alpha}.j$. As already stated, once fixed at the start of ρ, $\mathcal{I}_0$ *does not change* and is used at each step for evaluating formulae.

Definition 4. *An agent formula ψ_{goal} is **reachable** in $\mathcal{M}$ iff $\mathcal{I}_0$ and an initial configuration g^0 exist such that a configuration g with $g \models_{\mathcal{I}_0} \psi_{goal}$ is reachable through a run $\langle \rho, \mathcal{I}_0 \rangle$ from g^0.*

If a formula is not reachable then it is so for any number of agents and all possible interpretations. Finally, we can formally state the task at hand.

Definition 5. *Given a PMAS $\mathcal{M}$ and an agent formula ψ_{goal}, the safety checking problem is to check whether ψ_{goal} is* not *reachable in $\mathcal{M}$. If this is the case then $\mathcal{M}$ is said to be **safe** w.r.t. ψ_{goal}, otherwise it is **unsafe** w.r.t. ψ_{goal}.*

Example 2. *In our running example, the agent formula expressing the unwanted condition is* $\texttt{room}^{[j]} = \texttt{c} \wedge \texttt{destroyed}^{[j]} = \texttt{false} \wedge \texttt{armed}^{[e]} = \texttt{false}$.

PMASs as Array-based Systems

"Array-based Systems" (ABSs) is a generic term used to refer to *infinite-state transition systems* implicitly specified using a declarative, logic-based formalism in which arrays are manipulated via logical formulae. They are described using a multi-sorted theory: one kind of sorts for the indexes of arrays and another for the elements stored therein. The content of an array is unbounded and updated during the evolution. Nonetheless, note that our technique does not distinguish between unboundedness and infiniteness, and ABSs enjoy the finite model property (Calvanese et al. 2020).

In order to introduce verification problems in the symbolic setting of ABSs, one first has to specify the FO theories T_{Ind} and T (equipped with FO signatures Σ_{Ind} and Σ, resp.) for *array indexes* and for the *array elements*. In this paper, T_{Ind} will be the empty theory where Σ_{Ind} contains only equality, and T will be EUF (the theory of uninterpreted symbols), i.e., the empty theory with signature Σ containing sorts $\mathcal{S}$, relation symbols Rel and constants C. This is a standard, common setting in the ABS literature. Then, one needs to write the formulae representing the set of initial states and the system evolution. We denote by $\underline{z}$ a tuple $\langle z_1, \ldots, z_m \rangle$ and by $\phi(\underline{x}, \underline{a})$ the formula with $\underline{x}$ as free individual variables and $\underline{a}$ as free array variables. In the following we use the notation $F(\underline{x}) := \texttt{case of } \{\kappa_1(\underline{x}) : t_1; \cdots \kappa_n(\underline{x}) : t_n\}$ (where $\kappa_i(\underline{x})$ are quantifier-free Σ-formulae and t_i are generic terms), or, equivalently, nested if-then-else expressions: we call one such F *case-defined function*. We also use λ-abstractions like $b = \lambda j.F(j, \underline{z})$ in place of $\forall j.\, b(j) = F(j, \underline{z})$, where typically F is a case-defined function or a constant assignment. We consider three types of formulae:

- An *initial formula* $\iota(\underline{x}, \underline{a})$ initializes individual and array variables via assignments and λ-abstractions: $(\bigwedge_{i=1}^{m} x_i = c_i) \wedge (\bigwedge_{i=1}^{k} a_i = \lambda j.d_i)$, with c_i,d_i constants from Σ;
- A *state formula* of the form $\exists \underline{j}\, \phi(\underline{j}, \underline{x}, \underline{a})$ specifies conditions on variables, where ϕ is a quantifier-free Σ-formula and $\underline{j}$ are individual variables of the index sort;
- A *transition formula* $\hat{\tau}$ relates current and new (primed) values of individual and array variables: $\exists \underline{e}\, (\gamma(\underline{e}, \underline{x}, \underline{a}) \wedge (\bigwedge_{i=1}^{m} x_i' := c) \wedge (\bigwedge_{i=1}^{k} a_i' = \lambda j.F_i(j, \underline{e}, \underline{x}, \underline{a})))$, where $\underline{e}$ are individual variables (of *both* element and index sorts); γ (the 'guard') is a quantifier-free Σ-formula; $\underline{x}'$ and $\underline{a}'$ are renamed copies of $\underline{x}$ and $\underline{a}$; c is a constant from Σ and F_j (the "conditional update") is a case-defined function.

We now give a general definition of array-based systems, one that helps us to narrow down the scope and consider the kind that is suitable for our purposes (e.g. having the notion of action), in place of a generic notion of array-based system, that is extremely general. Then we show how a PMAS can be encoded as a special case of such definition (in Def. 7). Known results on ABS (Ghilardi and Ranise 2010a; Calvanese et al. 2020) can directly be adapted to this variant.

Definition 6. *An **abstract AB-PMAS** is a tuple:*

$$\langle \Sigma, \mathcal{S}_{ind}, \underline{x}, \underline{arr}_Y, \underline{act}, \iota_a, \tau_a \rangle$$

(i) $\Sigma := \langle \mathcal{S}, Rel, C \rangle$ is a multi-sorted FO signature as before, such that there exists a specific sort $\mathcal{A}^a \in \mathcal{S}$ called 'actions sort' ; (ii) $\mathcal{S}_{ind}$ is a set of sorts of index type; (iii) $\underline{x}$ is a set of individual variables (containing the global variables encoding the states of the environment); (iv) $\underline{arr}_Y$ is a set of arrays, one for each variable $y \in Y$, where Y is an abstract set of variables; (v) $\underline{act}$ is a set of arrays, with codomain of type $\mathcal{A}^a$; (vi) ι_a is an initial formula, whose individual variables are $\underline{x}$ and whose array variables are $\underline{arr}_Y$ and $\underline{act}$; (vii) τ_a is a disjunction of transition formulae, with individual and array variables $\underline{x}, \underline{x}'$ and $\underline{arr}_Y, \underline{act}, \underline{arr}'_Y, \underline{act}'$, resp.

Formally, a FO interpretation of Σ can be thought as an *instance* of the *'elements' domain* of an abstract AB-PMAS, the individual variables are assigned to values taken from this interpretation, $\mathcal{A}^a$ is interpreted over a finite set of elements called 'actions' and the sorts $\mathcal{S}_{ind}$ are interpreted over disjoint sets of concrete indexes. The array variables are assigned to functions from these sets of indexes to the instance of the elements domain.

In what follows, we show how a specific abstract AB-PMAS (simply called AB-PMAS) can be used to model a PMAS as in the previous section. To this end, we consider the different sorts and relations as in that section, and we encode the set of agent and environment templates. Instead of the abstract set $\underline{arr}_Y$ of arrays, for each template T_t with $t \in \{1, \ldots, n, e\}$ we consider a set $\underline{arr}_{V_t}$ of arrays, one for each variable in V_t, that is used to store the current value of that variable for each concrete agent of type t. Intuitively, the 'cell' for index j in the array for variable $v \in V_t$ of template T_t holds the value of v for the agent with ID (in correspondence to) j in the current global state (see Fig. 1). Since only one concrete environment exists, instead of arrays $\underline{arr}_{V_e}$ we use individual global variables of the form env_{V_e}. Accordingly, the set $\mathcal{A}^a$ of generic actions is now $\cup_t \mathcal{A}_t^{loc} \cup \mathcal{A}^{syn}$.

Additional global variables $\underline{x}$, which we now denote by $\underline{glob}$ for readability, are needed for book-keeping (that is, to model any required low-level detail in the PMAS that is needed to encode its execution, such as flags, counters, turn indicators, etc). The global variable *Phase*, discussed later, is an example of such variables.

Definition 7. *Given a PMAS $\mathcal{M} := \langle \{T_1, \ldots, T_n\}, T_e, \mathcal{R} \rangle$ and a set of initial states, its **AB-PMAS** is a tuple:*

$$\langle \Sigma, \{\mathcal{S}_{ID_t}\}_{t\in[1,n]}, \underline{glob}, \{\underline{arr}_{V_t}\}_{t\in[1,n]}, \{arr_{\mathcal{A}_t}\}_{t\in[1,n]}, \iota, \tau \rangle$$

where: (i) $\Sigma := \langle \mathcal{S}, \mathcal{R}, C \rangle$, where $\mathcal{S}$ are sorts (including sorts $\mathcal{S}_{\mathcal{A}_t^{loc}}$ and $\mathcal{S}_{\mathcal{A}^{syn}}$), $\mathcal{R}$ are the relation symbols of $\mathcal{M}$ and C a set of constants (including all values $val_t(l, v)$ for every t, l, v); (ii) $\{\mathcal{S}_{ID_t}\}_{t\in[1,n]}$ is a set of sorts of indexes type, one for each ID_t. (iii) $\underline{glob}$ is a set of individual variables used to encode the local state of the environment plus any book-keeping info (see later); (iv) $\{\underline{arr}_{V_t}\}_{t\in[1,n]}$ is a set of sets of arrays, one for each variable in V_t of each T_t, whose elements range over Δ_θ for $\theta = type(v)$; (v) $\{arr_{\mathcal{A}_t}\}_{t\in[1,n]}$ is a set of arrays, one for each T_t, whose codomain has type $\mathcal{S}_{\mathcal{A}_t^{loc}}$ or $\mathcal{S}_{\mathcal{A}^{syn}}$; (vi) ι is an initial formula, with individual variables $\underline{glob}$ and array variables $\underline{arr}_{V_t}, arr_{\mathcal{A}_t}$; (vii) τ is a disjunction of transition formulae, with individual variables $\underline{glob}, \underline{glob}'$, and array variables $\underline{arr}_{V_t}, arr_{\mathcal{A}_t}, \underline{arr}'_{V_t}, arr'_{\mathcal{A}_t}$.

A model of an AB-PMAS is a FO interpretation of Σ accounting for the 'elements' domain, equipped with an assignment of the individual variables to elements of that interpretation; the action sorts $\mathcal{S}_{\mathcal{A}_t^{loc}}$ and $\mathcal{S}_{\mathcal{A}^{syn}}$ are resp. interpreted over sets $\mathcal{A}_t^{loc}$ and $\mathcal{A}^{syn}$; the index sorts $\{\mathcal{S}_{ID_t}\}_{t\in[1,n]}$ are interpreted over the disjoint sets of concrete agents IDs ID_t, for every $t \in [1, n]$. Array variables are assigned to functions from these sets of IDs to the 'elements' domain.

Next, we finally encode a PMAS into an AB-PMAS, which constitutes our first main contribution. As shown in Fig. 1, for each $t \in [1, n]$ there are k_t arrays for local variables $\{v_1^t, \cdots, v_{kt}^t\} = V_t$ plus one array storing the current chosen action (in $\mathcal{A}^{syn} \cup \mathcal{A}_t^{loc}$ or nop as default value). The environment is modeled with global variables: there is one global variable for each template variable and one action variable storing the current synchronization action in $\mathcal{A}^{syn}$ (or nop). The initial formula is trivial to write (it has the very same shape given before). The formula τ in Def. 7 is the disjunction of transition formulae $\hat{\tau}$, shown next.

Encoding interleaved PMAS as ABSs

In this section we present the ABS formulae that are needed for capturing the execution semantics of our interleaved PMASs. These formulae can be directly implemented into MCMT, a infinite-state model checker that can verify ABSs: the encoding into MCMT input files *matches exactly* the reduction from PMASs to AB-PMAS described here. Each global transition of the PMAS is encoded as a sequence of 'steps' of the AB-PMAS, each specified by a disjunction of transition formulae, ordered by means of an additional global variable *Phase* used to guide the progression. This is intuitively shown in Figure 1 (Below). The variables in each agent template are encoded by array variables, so that the value of each v for an agent with ID j can be written in the position j of that array arr_v (the length and content of arrays is unbounded). An additional array $arr_{\mathcal{A}_t}$, one for each template T_t, stores the action currently "declared" by agents. So the local state l of a concrete agent $\langle j, l \rangle_t$ is encoded by the values written for index j in the arrays for template T_t (e.g., the red area in Figure 1). The environment variables

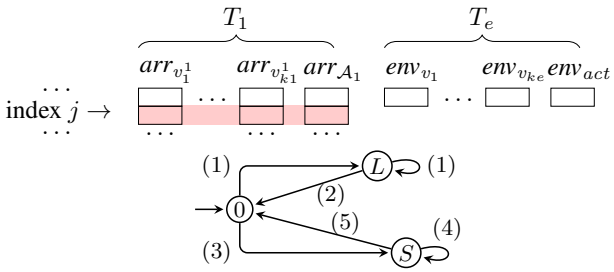

Figure 1: Above, a depiction of the encoding of an agent template T_1 and of the environment template T_e. Below, the phases used to capture the execution of an AB-PMAS.

are instead encoded via global variables, because only one concrete environment exists. In the same figure (Below), we show the how transition formulae of the ABS are used to encode the execution semantics of interleaved PMASs. Nodes correspond to phases and edges to (disjunctions of) transition formulae. There are two kinds of progressions, corresponding to the two semantics in Def. 3: either some non-empty subset of concrete agents execute local actions on their local state (upper branch) or a synchronization action is performed by the environment and at least one concrete agent (lower branch). The former case is realized by a non-empty sequence of steps following formula (1), in which local actions are written ('declared') in the appropriate position j of array $arr_{\mathcal{A}_t}$ for some t, followed by a single step in which the local state of all concrete agents that declared an action is updated in bulk (by applying the function in the λ-abstraction) as in (2). As transitions ($\hat{\tau}$ formulae) are taken nondeterministically, this will capture all possible sequences of (1)-steps followed by a (2)-step. The latter case is analogous: the formula in (3) makes sure that the environment and at least one concrete agent with ID j and type t can execute a synchronization action, which is then written in a global variable env_{act} as well as in the array position $arr_{\mathcal{A}_t}[j]$. Then, a number of concrete agents can declare the same action, updating their action array as specified by (4). Nondeterministically, a bulk update is performed as in formula (5), which also updates the environment. In both cases, when the initial phase is reached again, the action arrays $arr_{\mathcal{A}_t}$ of each template T_t are reset to contain nop values. As a result a possible evolution of an AB-PMAS template corresponds to a possible path in this intuitive diagram.

We now list the transition formulae, numbered as in Figure 1 (Below). For encoding the first step of the upper branch we use a disjunction of transition formulae as (1) below, for each $t \in \{1, \ldots, n, e\}$ and local action $\alpha \in \mathcal{A}_t^{loc}$:

$$\begin{pmatrix} Phase = 0 \vee \\ Phase = L \end{pmatrix} \wedge \exists j_{self} \begin{pmatrix} arr_{\mathcal{A}_t}[j_{self}] = nop \ \wedge \ \overline{P_t}(\alpha) \ \wedge \\ Phase' := L \ \wedge \ arr'_{\mathcal{A}_t}[j_{self}] := \alpha \end{pmatrix} \quad (1)$$

where a disjunction is used for compactness: we can write this as two distinct formulae. Here, as in the rest of the paper, when a primed array variable a' does not appear explicitly, the current content of a is propagated, as is, to the next state.

Remark 2. Above we denoted by $\overline{P_t}(\alpha)$ the transformation of the (quantifier-free) agent formula $P_t(\alpha) =$

$\psi(\underline{j}, \mathit{self}, e, \underline{v})$ into a (quantifier-free) formula $\phi(\underline{j}, \underline{x}, \underline{a})$ for AB-PMASs, with the same syntactic shape of state formulae. This is required because agent formulae, e.g., make use of local template variables, whereas array-based formulae make use of individual and array variables. Such transformation is trivial and it is not formalized further: we simply need to replace the variables of sort ID in φ with index variables from the sort used for agents in the AB-PMAS. As a special case, we impose that *self* is always replaced with a special index j_{self} that is existentially quantified in (1): there must exist an agent with index j_{self} and template T_t that we can take as *self* to evaluate $P_t(\alpha)$. If other index variables j are present, these are existentially quantified as well.

The step above is repeated an unbounded number of times (see the loop on state L in Fig. 1), as long as a new index j_{self} exists. Then, nondeterministically, a further step can be executed, characterized by the transition formula below. Here, as in the remainder of the section, we write $\underline{arr}_{V_t}[j] = l$ as a shorthand to denote that, for each variable $v \in V_t$ for some $t \in \{1, \ldots, n, e\}$, $arr_v[j] = val(l, v)$, namely the set of arrays $\underline{arr}_{V_t}[j]$ for the concrete agent with ID j of type t encode the local state $l \in L_t$ (see Fig. 1, in red). The same for $\underline{arr}'_{V_t} := \lambda j.\ \texttt{case of}\ \{\ldots, \kappa_i : val(l, v), \ldots\}$. In this formula, we perform a bulk update of all instances (indexes j) by applying the transition function of each T_t (below, one case is listed for each couple of local state-action):

$$\mathit{Phase} = L \wedge \mathit{Phase}' := 0 \wedge \bigwedge_{t \in \{1,\ldots,n,e\}} arr'_{\mathcal{A}_t} := \lambda j.\ \mathit{nop} \wedge \bigwedge_{t \in \{1,\ldots,n,e\}} \underline{arr}'_{V_t} := \lambda j. \left(\begin{array}{c} \texttt{case of} \\ \left\{ \begin{array}{c} \underline{arr}_{V_t}[j] = l_1 \wedge arr_{\mathcal{A}_t}[j] = \alpha_1 : \delta_t(l_1, \alpha_1) \\ \ldots \\ \underline{arr}_{V_t}[j] = l_m \wedge arr_{\mathcal{A}_t}[j] = \alpha_k : \delta_t(l_m, \alpha_k) \end{array} \right\} \end{array} \right) \tag{2}$$

Above, for each j one possible case applies. E.g., if α_1 was declared and the state is l_1, then $\underline{arr}_{V_t}[j] = \delta_t(l_1, \alpha_1)$.

For synchronization actions, for each $\alpha \in \mathcal{A}^{syn}$ and template T_t we have the following formula, making sure that at least one concrete agent and the environment can perform the same action, then written in the global variable env_{act}:

$$\mathit{Phase} = 0 \wedge \exists j_{\mathit{self}} \left(\begin{array}{c} arr_{\mathcal{A}_t}[j_{\mathit{self}}] = \mathit{nop} \wedge \mathit{env}_{act} = \mathit{nop} \wedge \\ \overline{P_t}(\alpha) \wedge \overline{P_e}(\alpha) \wedge \\ arr'_{\mathcal{A}_t}[j_{\mathit{self}}] := \alpha \wedge \mathit{env}'_{act} := \alpha \wedge \mathit{Phase}' := S \end{array} \right) \tag{3}$$

Then two possibilities exist: either an unbounded number of agents participate in the synchronization (via (4)), or the bulk progression is performed by applying the formula (5) further below. Again, this is encoded as a disjunction of formulae. For each $t \in [1, n]$ and $\alpha \in \mathcal{A}^{syn}$:

$$\mathit{Phase} = S \wedge \exists j_{\mathit{self}} \left(\begin{array}{c} \mathit{env}_{act} = \alpha \wedge \overline{P_t}(\alpha) \wedge arr_{\mathcal{A}_t}[j_{\mathit{self}}] = \mathit{nop} \\ \wedge\ \mathit{Phase}' := S \wedge arr'_{\mathcal{A}_t}[j_{\mathit{self}}] := \alpha \end{array} \right) \tag{4}$$

$$\mathit{Phase} = S \wedge \mathit{env}_{act} = \alpha \wedge \mathit{env}_{V_e} = l_e \wedge \mathit{env}'_{V_e} := \delta_e(l_e, \alpha) \wedge \mathit{env}'_{act} := \mathit{nop} \wedge arr'_{\mathcal{A}_t} := \lambda j.\ \mathit{nop} \wedge \mathit{Phase}' := 0 \wedge \bigwedge_{t \in [1,n]} \underline{arr}'_{V_t} := \lambda j. \left(\begin{array}{c} \texttt{case of} \\ \left\{ \begin{array}{c} \underline{arr}_{V_t}[j] = l_1 \wedge arr_{\mathcal{A}_t}[j] = \alpha : \delta_t(l_1, \alpha) \\ \ldots \\ \underline{arr}_{V_t}[j] = l_m \wedge arr_{\mathcal{A}_t}[j] = \alpha : \delta_t(l_m, \alpha) \} \end{array} \right\} \end{array} \right) \tag{5}$$

Verification

We denote by $ab(\mathcal{M})$ the AB-PMAS obtained by encoding a PMAS $\mathcal{M}$ as in the previous section. An **unsafety formula** for $ab(\mathcal{M})$ is a state formula ϕ of the form $\exists \underline{j}.\phi(\underline{j}, \underline{x}, \underline{a})$. These formulae are used to characterize undesired states of $ab(\mathcal{M})$.

By adopting a customary terminology for array-based systems, we say that $ab(\mathcal{M})$ is **safe with respect to** ϕ if intuitively the system has no finite run leading from ι to ϕ. Formally, this means that there is no interpretation $\mathcal{I}_0$ of relations, no $k \geq 0$ and no possible assignment to the individual and array variables $\underline{x}^0, \underline{a}^0, \ldots, \underline{x}^k, \underline{a}^k$ such that the formula

$$\iota(\underline{x}^0, \underline{a}^0) \wedge \tau(\underline{x}^0, \underline{a}^0, \underline{x}^1, \underline{a}^1) \wedge \tau(\underline{x}^{k-1}, \underline{a}^{k-1}, \underline{x}^k, \underline{a}^k) \wedge \phi(\underline{x}^k, \underline{a}^k)$$

is valid in any model of $ab(\mathcal{M})$. The safety problem for $ab(\mathcal{M})$ is the following: *Given an unsafety formula ϕ as before, decide whether $ab(\mathcal{M})$ is safe with respect to ϕ*. It is immediate to see that this matches Def. 4: the AB-PMAS cannot be safe w.r.t ϕ if there exists an initial interpretation $\mathcal{I}_0$ so that a global state g with $g \models_{\mathcal{I}_0} \psi$, where ψ is an agent formula with $\overline{\psi} = \phi$, is reachable through a run $\langle \rho, \mathcal{I}_0 \rangle$, and vice-versa (recall the description of $\bar{\cdot}$ in Remark 2).

Proposition 1. *An agent formula ψ_{goal} is reachable in a PMAS $\mathcal{M}$ iff the AB-PMAS $ab(\mathcal{M})$ is unsafe w.r.t. $\overline{\psi_{goal}}$.*

Soundness, Completeness, Termination

The algorithm described in Fig. 1 shows the *SMT-based backward reachability procedure* (or, **backward search**) for handling the safety problem for an AB-PMAS $ab(\mathcal{M})$. An integral part of the algorithm is to compute *symbolic* preimages. For that purpose, for any $\tau(\underline{z}, \underline{z}')$ and $\phi(\underline{z})$ (where $\underline{z}'$ are renamed copies of $\underline{z}$), we define $\mathit{Pre}(\tau, \phi)$ as the formula $\exists \underline{z}'(\tau(\underline{z}, \underline{z}') \wedge \phi(\underline{z}'))$. The *preimage* of the set of states described by a state formula $\phi(\underline{x})$ is the set of states described by $\mathit{Pre}(\tau, \phi)$ (notice that, when $\tau = \bigvee \hat{\tau}$, then $\mathit{Pre}(\tau, \phi) = \bigvee \mathit{Pre}(\hat{\tau}, \phi)$). Backward search computes iterated preimages of an unsafety formula ϕ, until a fixpoint is reached (in that case, $ab(\mathcal{M})$ is *safe* w.r.t. ϕ) or until a set intersecting the initial states (i.e., satisfying ι) is found (in that case, $ab(\mathcal{M})$ is *unsafe* w.r.t. ϕ) . *Inclusion* (Line 2) and *disjointness* (Line 3) tests can be discharged via proof obligations to be handled by SMT solvers. The fixpoint is reached when the test in Line 2 returns *unsat*: the preimage of the set of the current states is included in the set of states reached by the backward search so far (represented as the iterated application of preimages to the unsafety formula ϕ). The test at Line 3 is satisfiable when the states visited so far by the backward search includes a possible initial state (i.e., a state satisfying ι). If this is the case, then $ab(\mathcal{M})$ is unsafe w.r.t. ϕ. We call **PMAS-Backward search** the procedure that takes as input a PMAS $\mathcal{M}$ and a goal formula ψ_{goal}, transforms them into its corresponding array-based system $ab(\mathcal{M})$ and into $\overline{\psi_{goal}}$ resp., and then applies the backward search $\mathit{BReach}(ab(\mathcal{M}), \overline{\psi_{goal}})$. This procedure either does not terminate or returns a SAFE/UNSAFE result. Given a PMAS $\mathcal{M}$ and an agent formula ψ_{goal}, a SAFE (resp. UNSAFE) result is *correct* iff $\mathcal{M}$ is safe (resp. unsafe) w.r.t. ψ_{goal}.

Algorithm 1: backward search $BReach(ab(\mathcal{M}), \phi)$

1 $B \longleftarrow \bot$;
2 **while** $\phi \wedge \neg B$ *is* satisfiable **do**
3 | **if** ($\iota \wedge \phi$ is satisfiable) **then return** UNSAFE;
4 | $B \longleftarrow \phi \vee B$;
5 | $\phi \longleftarrow Pre(\tau, \phi)$; // τ is as in Def. 7
6 **return** SAFE;

Theorem 1. *PMAS-Backward search for the reachability problem for* interleaved *PMASs is so that,* (i) *when it terminates, it returns a correct result, and* (ii) *whenever* UNSAFE *is the correct result, then* UNSAFE *is indeed returned.*

Proof Sketch. Thanks to Proposition 1, establishing reachability of ψ_{goal} in $\mathcal{M}$ is equivalent to establishing the unsafety of $ab(\mathcal{M})$ w.r.t. $\overline{\psi_{goal}}$: hence, we can focus on the latter. For both (i) and (ii) we need to prove, by taking inspiration from (Ghilardi and Ranise 2010a; Calvanese et al. 2019b), that every step in Alg. 1 can be effectively executed: for doing so, we need suitable decision procedures for the satisfiability tests at Lines 2-3. First, in order to guarantee the *regressability* of the procedure, the fact that ϕ is a state formula needs to be a loop invariant: this is obtained by showing that the pre-image of a state formula can be converted to a state formula. Then, to perform the tests at Lines 2-3, we can show that entailment between state formulae can be decided via finite instantiation techniques: this is possible thanks to the specific shape of these formulae (which are called $\exists\forall$-formulae in (Felli, Gianola, and Montali 2020c)). For (ii), we conclude the proof by noticing that finite unsafe traces are found after finitely many steps. The full proof is reported in (Felli, Gianola, and Montali 2020c). □

Backward search for interleaved AB-PMAS is thus a *semi-decision* procedure for checking that a PMASs is unsafe. However, there is no guarantee of termination because the pre-image computation can diverge on a safe AB-PMAS.

We show under which (sufficient) condition we can guarantee termination of the backward search, which will gives us a decision procedure for unsafety. Although technical proofs are quite involved at the syntactic level, they can be intuitively understood as based on this *locality condition*: the states "visited" by the backward search can be represented by state formulae which do not include direct/indirect comparisons and "joins" of distinct state variables for different agent IDs. E.g., we cannot use direct comparisons of the form $(v_1^{[j_1]} = v_2^{[j_2]})$, i.e., comparing template variables for agent IDs $j_1 \neq j_2$. Similarly, we cannot (indirectly) correlate $v_1^{[j_1]}$ and $v_2^{[j_2]}$ by writing $R(v_1^{[j_1]}, v_2^{[j_2]})$ for some relation R. We can however write $v_1^{[j_1]} = k$, $v_2^{[j_2]} = k$ for a constant k.

Of course, if this property is true for ϕ, it does not necessary hold for the formula obtained by "regressing" ϕ w.r.t. some transition formula $\hat{\tau}$, i.e., $Pre(\hat{\tau}, \phi)$: $\hat{\tau}$ includes translations $\overline{P_t}(\alpha)$ of template protocols $P_t(\alpha)$ for action α. Formally, we call a state formula **local** if it is a disjunction of the formulae of the form:

$$\exists j_1 \cdots \exists j_m \, (Eq(j_1, \ldots, j_m) \wedge \bigwedge_{k=1}^{m} \phi_k(j_k, \underline{x}, \underline{a})) \qquad (6)$$

Here, Eq is a conjunction of variable (dis)equalities, ϕ_k are quantifier-free formulae, and $j_1, \ldots, j_m$ are individual variables of index sort. Moving all the existential quantifiers externally, it is easy to see that a local state formula is a state formula. Note how each ϕ_k in (6) can contain only the existentially quantified index variable j_k. As said before, this limitation has an impact on transition formulae as well: we say that a transition formula $\hat{\tau}$ is *local* if whenever a formula ϕ is local, so is $Pre(\hat{\tau}, \phi)$.

Theorem 2. $BReach(ab(\mathcal{M}), \overline{\psi_{goal}})$ *always* terminates *for an interleaved AB-PMAS if its transition formula* $\tau = \bigvee \hat{\tau}$ *is a disjunction of* local *transition formulae and* $\overline{\psi_{goal}}$ *is local.*

Proof Sketch. Although involved, the proof relies on the use of a suitable *well-quasi-order (wqo)* definable thanks to locality. After introducing the algebraic notion of *cyclic* FO structure (see (Felli, Gianola, and Montali 2020c) for the formal definition), one can notice that, since our language is relational, there are only finitely many cyclic FO structures that can be built. The particular format of local formulae implies that it is sufficient to check the validity of a local formula in cyclic structures in order to know if a local formula holds in a generic model of AB-PMASs. This allows us to introduce a specific wqo based on counting how many copies of cyclic FO-structures every model of AB-PMASs contains. We can show that a non-terminating backward search would destroy the well-foundedness of this wqo. The full proof is in (Felli, Gianola, and Montali 2020c). □

Theorem 2 (with Thm. 1) gives a sufficient condition for termination, inspired to a condition well-known in the literature of verification of data-aware processes (Calvanese et al. 2020). There, an analogous result holds, and decidability in the general case is still unknown. Our model of PMASs does not require the restriction imposed by locality, and indeed we proved only soundness and completeness of backward search (Thm. 1), making it a semi-decision procedure for checking unsafety. This is consistent with our focus, which is not on decidability, but on the effectiveness of employing the backward reachability procedure. Nevertheless, when locality is imposed, Thm. 2 proves that backward search, thanks to termination, becomes a full decision procedure. In particular, when relations and agent state variables comparisons are disregarded, locality (thus decidability) is always guaranteed: this in line with the models/results in (Emerson and Kahlon 2000, 2003; Emerson and Namjoshi 1996) when only reachability is considered. The structure of the proof of the theorem follows the schema of that of Thm. 5.4 in (Calvanese et al. 2020), with a significant difference: decidability is there based on locality and applies to ABSs whose FO-signatures do *not* have relational symbols, while ABSs have free relational symbols.

It is immediate to verify that the transition formulae and goal formula $\overline{\psi_{goal}}$ for our running scenario are both lo-

cal, hence checking safety is decidable. The PMAS can be proved to be unsafe w.r.t. ψ_{goal} as in Example 2.

Implementation

In this section, we illustrate our implementation approach and the tool called *SAFE*, which makes available the third-party model checker MCMT (Ghilardi and Ranise 2010b) for checking the safety of PMASs. A thorough description of *SAFE* and of its use in connection with MCMT for the safety checking of PMASs is presented in (Felli, Gianola, and Montali 2020b): we report here a brief discussion on this implementation in order to show the feasibility of our theoretical approach, and what follows is not intended either as a proper tool description or as an experimental validation.

MCTM is a symbolic model checker for safety properties of infinite-state systems, based on backward reachability and fixpoints computations (computed relying on an SMT solver): every ABS can be processed by MCMT (with the correct syntax). By using this tool-chain, the user is able to (i) model a PMAS $\mathcal{M}$ in a concrete syntax using *SAFE*, (ii) generate its encoding into MCMT (this encoding follows *exactly* the formalization presented in this paper) (iii) check the safety of $\mathcal{M}$ w.r.t. a given agent formula via MCMT (which also provides a *witness* in case of an unsafe verdict).

For lack of space, in this paper we do not describe the MCMT input files, as these are only used as the internal representation in our implementation. Instead, we briefly describe *SAFE* (Felli, Gianola, and Montali 2020a), i.e., our implementation of a user interface allowing to use in practice the results presented in this paper. *SAFE* automatizes the textual encoding of the PMAS into MCMT input files, by relying on a *MAS-oriented* modeling approach. This allows the user to focus on modeling the PMAS, i.e., the agent and environment templates, without worrying about how their constructs can be encoded for MCMT. The tool also allows to convert the witnesses for unsafety that MCMT returns (for unsafe ABSs) back into executions of the original PMAS.

***SAFE* Agent templates.** In its present version, the representation of agent and environment templates used by *SAFE* differs slightly from the one used here, although it is equivalent. Instead of an explicit state-transition representation as in Def. 1, *SAFE* assumes a more succinct representation, namely a STRIPS-like approach where actions are specified by means of pre- and post- conditions.

Execution of *SAFE*-MCMT. We run here our tool-chain over the example. As the PMAS is unsafe, this allows us to comment in practice on how a witness of unsafety is returned by MCMT. More examples (including safe ones) are available through the GUI of *SAFE* (Felli, Gianola, and Montali 2020a) and in (Felli, Gianola, and Montali 2020b), and they are all reproducible running *SAFE* and then MCMT. The textual encoding of the ABS corresponding to the PMAS in the running example is solved by MCMT v.3.0, on a machine with Ubuntu 18.04, 3.60 GHz Intel Core i7-7700 CPU, in 2 minutes and 22 seconds and in 56 seconds respectively using Yices (version 1.0.40) and Z3 (version 4.8.9.0) as background SMT solvers. MCMT correctly reports that the system is unsafe.[1] The generated input file ("download MCMT input") contains 501 lines of code and has 3 local variables for $T_{\mathtt{att}}$, 4 global variables and 15 transitions formulae. MCMT returns this witness for unsafety: `[t1_3][t2_2][t2_1][t3][t5_2][t9_1][t13][t6_3][t14][t4_2][t8_1][t12][t7_2][t15]`, where each `tn_m` represents the execution of the n-th transition in the input file, in the order in which they appear, with m instantiated index variables. In this case, this is sequence of actions `goA,goA,goA,goB,goB,off,goC,goC,pulse`, where each action is executed by one agent (and/or by the environment).

Conclusion and Future Work

We have presented a model of PMASs, defined the verification task of checking safety, and provided a custom, MAS-oriented tool that allows to make use of a generic SMT-model checker off-the-shelf. Our tool supports a concrete syntax for PMASs, and automatically encodes it into the input format accepted by the MCTM model checker for ABSs. The generality of our approach allows in principle any tool accepting ABS inputs to be used, e.g. Cubicle (Conchon et al. 2012). Cubicle, however, crucially does not support data-aware verification, like relations with primary and foreign keys: MCMT, instead, will allow us to exploit these features when, as commented below, data-aware extensions of PMASs are considered. To the best of our knowledge, this is the first paper that establishes a theoretical connection between the verification of PMASs and SMT-based model checking for ABSs, opening up the possibility of solving the former via advanced techniques and tools for the latter. This allows a number of interesting extensions.

From the foundational perspective, *data-aware* extensions of our framework can be directly incorporated, along the line studied in (Calvanese et al. 2020). This supports finite action signatures with infinite number of possible parameter values, and also to store and inspect infinite data values. For instance, this will allow us to model and check for safety extended models of PMASs where agents are given *read and write* access to private and public databases, hence allowing us to model complex systems in which data is stored and exchanged. Also, the background theories employed by the SMT-solver in this paper are only the empty theory or EUF, whereas further ones may be considered. Adding data extensions, theories, possibly arithmetics and cardinality constraints, are all future directions.

From the applied perspective, the SMT technology brings effective techniques for symbolic reasoning, like decision procedures for combined theories or quantifier handling through instantiation and quantifier elimination. At the same time, it features advanced heuristics and approximation techniques, like acceleration, predicate abstraction and invariant synthesis. This triggers a natural extension of our implementation, tailored for efficiency. In fact, it is well-known that the performance of symbolic verification techniques can be substantially improved when such techniques are suitably developed for the domain at hand (Conchon et al. 2013).

[1] The example, modeled with *SAFE*, is publicly available at the address: http://safeswarms.club/page/mcmt/rooms

Acknowledgements

This work was supported by the Unibz projects SMARTEST, VERBA and REKAP.

References

Abdulla, P. A.; Cerans, K.; Jonsson, B.; and Tsay, Y.-K. 1996. General decidability theorems for infinite-state systems. In *Proc. of LICS*, 313–321. IEEE.

Alberti, F.; Ghilardi, S.; and Sharygina, N. 2017. A Framework for the Verification of Parameterized Infinite-state Systems. *Fund. Inform.* 150(1): 1–24.

Alechina, N.; Brázdil, T.; De Giacomo, G.; Felli, P.; Logan, B.; and Vardi, M. Y. 2019. Unbounded Orchestrations of Transducers for Manufacturing. In *Proc. of AAAI*, 2646–2653.

Barrett, C. W.; and Tinelli, C. 2018. Satisfiability Modulo Theories. In *Handbook of Model Checking.*, 305–343.

Belardinelli, F.; Kouvaros, P.; and Lomuscio, A. 2017. Parameterised Verification of Data-aware Multi-Agent Systems. In *Proc. of IJCAI*, 98–104.

Bloem, R.; Jacobs, S.; and Khalimov, A. 2015. *Decidability of Parameterized Verification*. Morgan & Claypool Publishers.

Bulling, N.; Goranko, V.; and Jamroga, W. 2015. Logics for Reasoning About Strategic Abilities in Multi-player Games. In *Models of Strategic Reasoning - Logics, Games, and Communities*, 93–136.

Calvanese, D.; Ghilardi, S.; Gianola, A.; Montali, M.; and Rivkin, A. 2019a. Formal Modeling and SMT-Based Parameterized Verification of Data-Aware BPMN. In *Proc. of BPM*, 157–175. Springer.

Calvanese, D.; Ghilardi, S.; Gianola, A.; Montali, M.; and Rivkin, A. 2019b. From model completeness to verification of data aware processes. In *Description Logic, Theory Combination, and All That*, 212–239. Springer.

Calvanese, D.; Ghilardi, S.; Gianola, A.; Montali, M.; and Rivkin, A. 2019c. Model Completeness, Covers and Superposition. In *Proc. of CADE*, 142–160. Springer.

Calvanese, D.; Ghilardi, S.; Gianola, A.; Montali, M.; and Rivkin, A. 2020. SMT-based Verification of Data-Aware Processes: a Model-Theoretic Approach. *Math. Struct. Comp. Sci.* 30(3): 271–313.

Clarke, E. M.; Henzinger, T. A.; Veith, H.; and Bloem, R., eds. 2018. *Handbook of Model Checking*. Springer.

Conchon, S.; Goel, A.; Krstic, S.; Mebsout, A.; and Zaïdi, F. 2012. Cubicle: A Parallel SMT-Based Model Checker for Parameterized Systems - Tool Paper. In *Proc. of CAV*, LNCS, 718–724. Springer.

Conchon, S.; Goel, A.; Krstic, S.; Mebsout, A.; and Zaïdi, F. 2013. Invariants for finite instances and beyond. In *Proceedings of FMCAD*, 61–68. IEEE.

De Masellis, R.; and Goranko, V. 2020. Logic-based specification and verification of homogeneous dynamic multi-agent systems. *Auton. Agents Multi Agent Syst.* 34(2): 34.

Emerson, E. A.; and Kahlon, V. 2000. Reducing Model Checking of the Many to the Few. In *Proc. of CADE*, 236–254. Springer.

Emerson, E. A.; and Kahlon, V. 2003. Model Checking Guarded Protocols. In *Proc. of LICS*, 361–370. IEEE.

Emerson, E. A.; and Namjoshi, K. S. 1996. Automatic Verification of Parameterized Synchronous Systems (Extended Abstract). In *Proc. of CADE*, LNCS, 87–98. Springer.

Emerson, E. A.; and Namjoshi, K. S. 2003. On Reasoning About Rings. *International Journal of Foundations of Computer Science* 14(4): 527–550.

Esparza, J.; Ganty, P.; Leroux, J.; and Majumdar, R. 2017. Verification of population protocols. *Acta Inf.* 54(2): 191–215.

Felli, P.; Gianola, A.; and Montali, M. 2020a. SAFE: the Swarm Safety Detector. www.safeswarms.club. Accessed: 2020-12-15.

Felli, P.; Gianola, A.; and Montali, M. 2020b. A SMT-based Implementation for Safety Checking of Parameterized Multi-Agent Systems. In *Proc. of PRIMA*, LNCS, 259–280. Springer.

Felli, P.; Gianola, A.; and Montali, M. 2020c. SMT-based Safety Verification of Parameterised Multi-Agent Systems. Technical Report arXiv:2008.04774, arXiv.org.

Finkel, A.; and Schnoebelen, P. 2001. Well-structured transition systems everywhere! *Theoretical Computer Science* 256(1): 63–92. ISSN 0304-3975.

Ghilardi, S.; Gianola, A.; Montali, M.; and Rivkin, A. 2020. Petri Nets with Parameterised Data: Modelling and Verification. In *Proc. of BPM*, 55–74. Springer.

Ghilardi, S.; Nicolini, E.; Ranise, S.; and Zucchelli, D. 2008. Towards SMT Model Checking of Array-Based Systems. In *Proc. of IJCAR*, 67–82.

Ghilardi, S.; and Ranise, S. 2010a. Backward Reachability of Array-based Systems by SMT Solving: Termination and Invariant Synthesis. *Log. Methods Comput. Sci.* 6(4).

Ghilardi, S.; and Ranise, S. 2010b. MCMT: A Model Checker Modulo Theories. In *Proc. of IJCAR*, 22–29. Springer.

John, A.; Konnov, I.; Schmid, U.; Veith, H.; and Widder, J. 2012. Counter Attack on Byzantine Generals: Parameterized Model Checking of Fault-tolerant Distributed Algorithms. Technical Report arXiv:1210.3846, arXiv.org.

Kouvaros, P.; and Lomuscio, A. 2016. Parameterised verification for multi-agent systems. *Artif. Intell.* 234: 152–189.

Kouvaros, P.; and Lomuscio, A. 2017. Parameterised Verification of Infinite State Multi-Agent Systems via Predicate Abstraction. In *Proc. of AAAI*, 3013–3020.

Kouvaros, P.; Lomuscio, A.; Pirovano, E.; and Punchihewa, H. 2019. Formal Verification of Open Multi-Agent Systems. In *Proc. of AAMAS*, 179–187.

Pnueli, A.; Xu, J.; and Zuck, L. 2002. Liveness with (0,1, infty)- Counter Abstraction. In *Proc. of CAV*, 107–122. Springer.