A SMT-based Implementation for Safety Checking of Parameterized Multi-Agent Systems

Paolo Felli, Alessandro Gianola, and Marco Montali

Free University of Bozen-Bolzano, Bolzano, Italy
{pfelli,gianola,montali}@inf.unibz.it

Abstract. We address the problem of verifying whether unwanted states, characterized as a given state formula, are reachable in a given parameterized multi-agent system (PMAS), i.e., whether the PMAS is unsafe. As the multi-agent system is parameterized, it only describes the finite set of possible agent templates, while the actual number of concrete agent instances for each template is unbounded and cannot be foreseen. However, as safety depends in general on the number of agent instances, the verification result must be correct irrespective of such a number. After having defined two distinct execution semantics of PMASs, in this paper we focus on an implemented approach for checking safety, which is composed of two steps. First, we have implemented a modeling tool, called *SAFE*, that allows to specify the agent templates in the PMAS and their possible interactions, and to automatically translate this model into a textual encoding of an array-based system (ABS). Second, we check safety via infinite-state model checking based on satisfiability modulo theories (SMT), by using the general purpose SMT-based model checker MCMT, which accepts ABS specifications as input. We show the correctness guarantees of this approach by relying on the theory of ABSs. 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, using SMT-based results now available thanks to this work.

Keywords: Multi-Agent Systems · Safety · Parameterized Verification · Satisfiability Modulo Theories

1 Introduction

Multi-agent systems (MASs) are commonly used in many complex, real-life domains, so it has become crucial to be able to verify such systems against given specifications. This 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 [15] is one of the most common approaches to verification of MASs, often with a focus on strategic abilities [10]. 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 propositionalize crucial system features. Other approaches have thus tackled the verification of MASs in settings that are intrinsically infinite-state [21], for which explicit model-checking techniques cannot be used off-the-shelf. These are the settings in which either some sort of *data component* is present or where the concrete component instances of the MAS are not explicitly listed beforehand. Our work falls into the latter category, that is the one of verification of parameterized MASs (PMASs), recognized as a key reasoning task and addressed by a growing literature [29, 9, 21, 17]. In PMASs, the number of agents is *unbounded* and *unknown*, so that possibly infinite concrete MASs need to be considered: the task is to check whether the specification is met by any (or all) concrete MASs that adhere to some behavioral structure (typically a set of agent *templates*), without fixing the number of actual agents a priori. Here, we focus on checking *safety*, namely that no state satisfying a state formula (existentially quantifying on agent instances) is reachable for any number of agent instances. E.g., checking that there will never be two agents in the restricted area. Note that this differs from checking that a strategy (for some agent) exists to prevent unsafe states. Safety checking (and reachability) is a crucial property of MASs as well as finite and infinite dynamic systems, with a long-standing

tradition (e.g., [3]). Applications are numerous, from the verification of properties of swarms to industry 4.0 [6], where one wants to check that instances of a product family will be manufacturable by robots from a fixed model catalogue.

In this paper we present our verification technique based on an SMT [7] approach for array-based systems (ABSs) [24, 25, 5, 4, 12, 14]. We provide a formal model of PMASs with two different execution semantics and we discuss how the safety problem for these systems can be correctly handled in the ABS framework. We detail our solution and comment on its implementation based on the well-established SMT-based model-checker MCMT [26]: we develop a tool called *SAFE*, an intuitive user interface that allows to directly encode PMASs. *SAFE* returns a textual representation of ABS that can be used as input specification file for MCMT. This implementation makes available the third-party model checker MCMT for checking the safety of PMASs. The remainder of this paper is organized as follows. In Section 2, we state the contributions of this work and relate our results to the previous literature on parameterized verification and, in particular, verification of parameterized multi-agent systems. Then, in Section 3 we provide the definition of Parameterized MAS (PMAS) and we present two different semantics for PMASs, i.e. the concurrent and the interleaved ones: this distinction gives rise to two corresponding classes of PMASs. In Section 3.2 the (un)safety checking problem for PMASs is introduced. In Section 4 we present our user-interface tool, called SAFE, illustrating the implementation of our approach, based on the state-of-the-art MCMT model checker. We conclude and discuss future work in Section 6, where we comment on how this framework lends itself to accommodate a further source of infinity, i.e. the *data* dimension. We now introduce our running example.

Example scenario. Imagine a robotic swarm attacking a defence position, protected by a robot cannon. There are only two possible paths to reach the position: an attacker must first move to waypoint A or B, then move again to reach the target. Attackers can only move to either waypoint if the paths to A or B are not covered in snow, and similarly for reaching the target. The snow condition is not known in advance. The defensive cannon can target either waypoint with a blast or with an EMP pulse. The cannon program is so that a blast can only be fired if there are robots in that waypoint, and at least one robot under fire is hit. If instead the EMP pulse is directed towards A or B, no robot can move there. The cannon can use either the blast or activate the pulse at the same time but, while the EMP is active, the cannon can continue firing blasts. The EMP can cover either A or B, not both. The number of attackers is not known.

It appears that, even if all paths are free of snow, an effective defensive plan exists: at the beginning, use the pulse (say against A); let the enemy robots make their moves to B (the pulse remains active on A), then use the blast against B to destroy robots there. Whenever further attackers move to B, use again the blast, otherwise wait. If one path is not viable the plan is even simpler. Question: does this strategy work? Answer: only if the blast destroys all robots in the waypoint against which it is fired. Q: if blasts do not hit all robots under fire, how many attackers may have a chance of reaching the target? A: at least two, if they move to the waypoint B together, since blasts always hit at least one robot. Q: is any attack plan for at least two robots guaranteed to work? A: no. This scenario is trivial, however a complex network of waypoints or cannons capable of targeting multiple waypoints can make this arbitrarily complex, also given that the snow conditions cannot be foreseen, requiring to reason by cases. If "playing" as attackers, computing an attack plan that has chances against any number of cannons is even more complex.

In this paper we tackle this type of scenarios, showing how they can be modeled and solved, but also that our solution technique is powerful enough to be used to account for a number of features that cannot be included in this preliminary work, e.g., the inclusion of full fledged relational database storing public and private agent data information with read and write access.

2 Related Work and Contribution

The related work is constituted by the literature on parameterized verification [9] and more specifically verification of PMASs.

The literature on parameterized verification is related but nonetheless distinct from our approach. The problem has been studied in the context of verification of reactive systems, e.g., for the analysis of broadcast protocols. The problem of checking whether a specification is true in a parameterized system is, e.g., shown to be decidable for forms of regular specifications [21] but undecidable even for stuttering-insensitive properties such as LTL\X formulae [19] if asynchronous rendezvous is allowed. As summarized in [9], decidability results for these systems are based on reduction to finite-state model checking via abstraction [32, 28], *cutoff* computations (i.e. a bound on the number of instances that need to be verified [18, 20]) or by proving that they can be represented as well-structured transition systems [22, 3]. Similarly to these settings, we are tackling verification of parameterized PMASs by relating our decidability results with the assumed MAS execution semantics and shape of the allowed guards, relating our modeling choices to a MAS setting. However, we here focus on safety and our technique is not based on (predicate and counter) abstractions, cutoffs or reductions to finite-state model checking. Also, the multi-agent systems we consider do not assume a particular topology, and the conjunctive and disjunctive guards considered in [19] are here extended toward a FO setting by also allowing relation symbols. Our theoretical results are based on the model-theoretic framework of ABS [25, 14] and can be seen as a declarative, first-order counterpart of theories of well-structured transition systems for which compatible results are known in the community (see, e.g., [3, 9]). Indeed, our focus is not on providing a characterization of the decidability boundaries for safety checking of PMASs, but to demonstrate the effectiveness of employing backward reachability with soundness, completeness, and termination guarantees (thus decidability). Finally, as argued below, the application of our results is novel, effective and yields decidability results of direct and immediate applicability to a clear class of PMASs.

Regarding specifically the verification of PMASs, the closest model is that of [29, 8] and open MASs [17, 30]. In [17, 30], the authors study MASs where agents can join and leave dynamically. As in our work, agents are characterized by a type and their number is not bounded. They employ synchronous composition operations over automata on infinite words and their procedure can verify strategic abilities for LTL goals by reduction to synthesis. Notwithstanding the fact that we only look at safety, a mechanism for joining/leaving the system can be captured natively in our formalization of PMASs. A similar framework is in [30], sharing the same model of [29] and related papers, with agent templates similar to those considered here. Compared with that work, we restrict to the key task of checking safety (reachability), instead of tackling model checking of arbitrary specifications. Safety checking (and, conversely, reachability analysis) is a crucial task with a long tradition in AI and in the field of reasoning about actions. Although not included here, we can capture many variants of execution semantics considered in [29]. As in our case, the results in that work depend on the chosen execution semantics, hence on the combinations of possible action types that are allowed. In the general case, their procedure requires to check the existence of a cutoff; if it exists, the verification result is correct, otherwise the procedure halts with no result. 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 nor any notion alike: we directly obtain a complete procedure in general and we get a full decision procedure for interleaved PMASs.

By departing from the literature mentioned above, we present here a verification technique based on an SMT-based [7] approach for ABSs [24, 25, 5, 14]. This is a very well-understood SMT-based formalism for which a number of results of practical applicability already exist [13, 11, 23]. Our approach is the first to establish a formal connection between the verification of PMASs and the long-standing tradition of SMT-based model checking for ABSs. Also, leveraging SMT-techniques makes the framework directly extendible in multiple directions: for example, we can easily introduce theories constraining agent data, i.e. elements can be retrieved by agents from relational databases (both shared or private) with constraints such as key dependencies, in the line with [12, 11, 14]. On this, it is our aim to combine this framework with the RAS formalism in [12, 14]. Adding theories, data-aware extensions, restricted arithmetics, cardinality constraints, are all now concretely viable directions for checking safety of PMASs.

Finally, we exploit *operationally* the connection between the PMAS and the SMT tradition, encoding safety checks of PMASs into the general-purpose model checker MCMT [26]. Since both the modeling of PMASs and their translation into input files for MCMT is particularly laborious, we have realized the intuitive web tool (*SAFE* [2]) for modeling and encoding PMASs into MCMT.

3 PMASs: parameterized MAS

In this section we give our definition of PMASs and of two alternative execution semantics. This model, although novel, shares many similarities with known parametric systems in the literature.

We consider a set Θ of (semantic) data types, used for variables. Each type $\theta \in \Theta$ comes with a (possibly infinite) domain Δ_θ, and a type-wise equality operator $=_\theta$. For instance, types are reals, integers, booleans, etc. We simply write $=$ when the type is clear. We also consider a set $\mathcal{R}$ of relations over types in Θ, which we treat as *uninterpreted* relations (i.e. simple relation symbols). These are used to model background information in the MAS but *are never updated during its execution*, constituting a *read-only* component. E.g., the snow condition in the scenario can be modeled via these relations, as we will show. 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* $T = \langle ID, L, l^0, V, type, val, \mathcal{A}^{loc}, \mathcal{A}^{syn}, P, \delta \rangle$ *having:*

- *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 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 in the next section, that allow to "query" the current state of the whole PMAS;*
- *a 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 $l \xrightarrow{\alpha} l'$. We assume δ to be total.*

An **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 (again, e is a constant).

Example 1. We use a template T_{att} for robots, with variables `loc` (enumeration [`init,A, B,target`]) and `destroyed` (boolean). The former variable is used for storing an agent's location, whereas the latter is for specifying whether the agent is destroyed. The actions are `gotoA`, `gotoB` and `gotoT` for moving to waypoints (from the initial location) and to the target (from either waypoint), plus additional actions `blastA`, `blastB` representing the action of "being destroyed" by a shot fired at position `A` or `B`, respectively. For instance, a transition $l \xrightarrow{\texttt{gotoA}} l'$ exists in δ for this template when $val(l, \texttt{loc}) = \texttt{init}$ and $val(l, \texttt{destroyed}) = \texttt{false}$, and the resulting local state l' is such that $val(l', \texttt{loc}) = \texttt{A}$ (plus further assignments for inertia). Other actions are defined in a similar manner. Figure 1 depicts this template, represented as a labeled finite-state transition system. The cannon is modeled as (part of) the environment, whose template T_e has actions `pulseA`,`pulseB`,`blastA`,`blastB` and variables `pulse-loc` (enumeration [`A`, `B`, `nil`]). The former is used to store the location (waypoint `A` or `B`) toward which the pulse is currently directed. The snow is captured by a binary relation over locations (e.g. $Snow(\texttt{init,A})$), whose interpretation is unbounded. Protocols are given later. □

Let $\{T_1, \ldots, T_n, T_e\}$ be a set of agent (and environment) templates, with $T_t = \langle ID_t, L_t, l_t^0, V_t, type_t, val_t, \mathcal{A}_t^{loc}, \mathcal{A}^{syn}, P_t, \delta_t \rangle$ for $t \in \{1, \ldots, n, e\}$. We denote a concrete agent of type T_t, $t \in [1, n]$, and ID 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

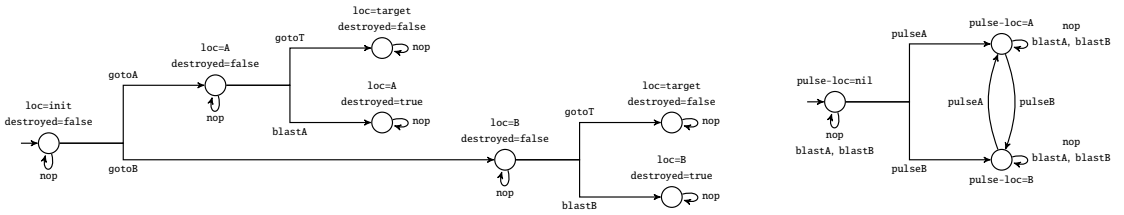


Fig. 1. A depiction of the templates T_{att} (left) and T_e (right). The label next to each local state l specifies the value $val(l, v)$ of each $v \in V$.

denote a vector of k such concrete agents of type T_t as $\langle \boldsymbol{I}, \boldsymbol{L} \rangle_t$, where $\boldsymbol{I} \in ID_t^k$ and $\boldsymbol{L} \in L_t^k$ are vectors of IDs and local states, respectively. Importantly, we assume that agent IDs are unique and template variables disjoint, 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 **snapshot** is a tuple $g = \langle \{\langle \boldsymbol{I}_1, \boldsymbol{L}_1 \rangle_1, \ldots, \langle \boldsymbol{I}_n, \boldsymbol{L}_n \rangle_n\}, \langle e, l_e \rangle_e \rangle$, which thus identifies the number of agent instances (the size of each $\boldsymbol{I}_t$, $t \in [1, n]$, may differ). A snapshot is *initial* iff all agents are in their initial local state. Clearly, *infinite possible initial snapshots exist*, since the number of concrete agents for each template is unbounded and not known a priori. As shorthand, we denote the local state l_j of agent $\langle j, l_j \rangle_t$ in snapshot g as $g.j$, thus writing $\langle j, g.j \rangle_t$.

3.1 Agent formulae

Here we 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:

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

where $v \in V$, k is a constant in Δ_θ for $\theta = \mathit{type}(v)$, R is a relation symbol in $\mathcal{R}$ of arity $m \geq 1$ (defined over types $\theta_1, \ldots, \theta_m$), each x_i is either a variable $v_i^{[j]}$ or a constant $k_i \in \Delta_{\theta_i}$, and j, j_1, j_2 are 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, as in the previous section, concrete IDs). Note that in a relation term we are restricted to use only one variable j of sort ID. The usual logical abbreviations apply. Intuitively, these formulas are implicitly quantified existentially over agent IDs. As we will formalize next, they allow to test (dis)equality of agent variables with respect to agent constants (IDs), and to check whether a tuple is in a relation (whose elements are agent variables). For instance, $(v^{[j]} = k)$ informally means that there exists an agent ID j so that $v = k$ for the such 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 $\underline{j}$, as well as the constant *self*, to a concrete agent ID in g (denoted $\sigma(j)$ and $\sigma(\mathit{self})$, respectively). It also assigns the constant e to itself, i.e. we impose $\sigma(e) = e$. 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 snapshot 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 such that $g, \sigma \models_{\mathcal{I}_0} \psi$, 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 $v = k$;*
- *$g, \sigma \models_{\mathcal{I}_0} R(x_1, \cdots, x_m)$ 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*. 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 $g \models^{3}_{\mathcal{I}_0} (v_1^{[\mathit{self}]} = 6) \wedge (v_1^{[j]} = 5)$.

Example 2. (cont.d) In the running example, the program of the cannon is so that the cannon can fire on a waypoints only if there is at least one attacker (not already destroyed) in that location. Hence, we have $P_e(\texttt{blastA}) = (\texttt{loc}^{[j]} = \texttt{A}) \wedge (\texttt{destroyed}^{[j]} = \texttt{false})$ (recall that $V_t \cap V_{t'} = \emptyset$ for $t \neq t'$, hence given a variable we know to which template it belongs). Similarly for `blastB`. Moreover, only positions that are not targeted by the pulse and are clear of snow can be accessed: $P_{att}(\texttt{gotoA}) = (\texttt{pulse-loc}^{[e]} \neq \texttt{A}) \wedge \neg \mathit{Snow}(\texttt{init}, \texttt{A})$ (similarly for `gotoB`). Note that we are assuming that the interpretation $\mathcal{I}_0$ is fixed, hence the snow condition (the relation Snow) is fixed and is not affected by the execution. As we are going to show in Section 3.2, however, we define our safety check so as to check that a condition cannot be reached irrespective of the initial interpretation, so as to consider any possible snow condition (we do so by checking whether there exists an interpretation so that the condition can be reached).

3.2 Concurrent and interleaved PMASs

In this section we introduce the two main execution semantics, hence defining two distinct types of PMASs, called concurrent and interleaved. These are distinguished by how the (single-step) transitions of the system are defined, which has to do with the types of interactions that are allowed between the agents and the environment.

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

Since each concrete agent and the environment may either perform an action (in $\mathcal{A}^{loc}_t \cup \mathcal{A}^{syn}$) or remain idle, multiple executions semantics can be defined, depending on the constraints we impose on $\boldsymbol{\alpha}$. We now describe more in detail the sets $\mathcal{A}^{loc}_t$ and $\mathcal{A}^{syn}$ introduced in Def. 1.

Symbols in $\mathcal{A}^{loc}_t$, for each t, are called **local actions**, and those in $\mathcal{A}^{syn}$ **synchronization** actions. Actions in $\mathcal{A}^{loc}_t$ 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, the synchronization actions are used to model explicit communication actions or any action with effects that are not private to the single agent or to the environment.

As a consequence, not every vector $\boldsymbol{\alpha}$ is meaningful: synchronization actions in $\mathcal{A}^{syn}$ are shared across all templates and are used to model global events that are (potentially) observable by any agent, whereas local actions in $\mathcal{A}^{loc}$ are private and can be freely executed. We 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 or messages from those steps in which agents update their local state.

Concurrent PMASs. First, we focus on those PMASs in which either all agents perform local actions if possible or, whenever a synchronization action is executed, then all agents and the environment are forced to synchronize (whenever this is possible), as formalized below. Intuitively, synchronization actions are seen as public events, affecting the local state of each concrete agent. For capturing this execution semantics, we adopt the following definition to characterize the global transitions that are said to be *legal*.

Definition 3. *Given $\mathcal{I}_0$, a global transition $g \xrightarrow{\boldsymbol{\alpha}} g'$ is said to be **legal** iff:*
- *$g'.j = \delta_t(g.j, \boldsymbol{\alpha}.j)$ for every $\langle j, g.j\rangle_t$, $t \in \{1, \ldots, n, e\}$, i.e., all the agents and the environment evolve as per their template;*
- *$g \models^j_{\mathcal{I}_0} P_t(\boldsymbol{\alpha}.j)$ for every $\langle j, g.j\rangle_t$, i.e., each action is executable and the constant self is substituted by j when evaluating the protocol of the chosen action;*
- *either only local actions are executed (by all agents and environment), or the environment and at least one agent synchronize with action $\alpha \in \mathcal{A}^{syn}$. In the former case, however, agents are allowed to remain idle iff they cannot execute any local action, and in the latter iff they cannot execute the synchronization action α. Formally, either:*
  - *no j exists so that $\boldsymbol{\alpha}.j \in \mathcal{A}^{syn}$, and for every $\langle j, g.j\rangle_t$ if $\boldsymbol{\alpha}.j = nop$ then no $\alpha \in \mathcal{A}^{loc}_t$ exists with $g \models^j_{\mathcal{I}_0} P_t(\alpha)$;*
  - *$\boldsymbol{\alpha}.e = \alpha \in \mathcal{A}^{syn}$ and at least one $j \neq e$ exists so that $\boldsymbol{\alpha}.j = \alpha$. Moreover, for every agent $\langle j, g.j\rangle_t$ either* (i) *$g \models^j_{\mathcal{I}_0} P_t(\alpha)$ and $\boldsymbol{\alpha}.j = \alpha$ or* (ii) *$g \not\models^j_{\mathcal{I}_0} P_t(\alpha)$ and $\boldsymbol{\alpha}.j = nop$.*

Interleaved PMASs. In these systems, at each step either *(i)* a subset of concrete agents (and the environment) perform a (non *nop*) action in $\mathcal{A}^{loc}_t$ on their local state or *(ii)* the environment and *a subset* of the agents synchronize by executing the same action in $\mathcal{A}^{syn}$. *nop* is a special no-op action: $\delta_t(l, nop) = l$ for all t, l. Local and synchronization actions cannot be mixed. We denote by $\boldsymbol{\alpha}.j$ the action of the agent with ID j, or of the environment if $j = e$.

Definition 4. *Given an interpretation $\mathcal{I}_0$, $g \xrightarrow{\boldsymbol{\alpha}} g'$ is **legal** iff:*
- *$g'.j = \delta_t(g.j, \boldsymbol{\alpha}.j)$ for every $\langle j, g.j\rangle_t$, $t \in \{1, \ldots, n, e\}$, i.e., agents and environment evolve as per their template;*
- *$g \models^j_{\mathcal{I}_0} P_t(\boldsymbol{\alpha}.j)$ 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 environment) or the environment and at least one agent synchronize via $\alpha \in \mathcal{A}^{syn}$. Other agents perform nop. Formally, either:*
  - *no j exists so that $\boldsymbol{\alpha}.j \in \mathcal{A}^{syn}$, 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, $\boldsymbol{\alpha}.e = \alpha \in \mathcal{A}^{syn}$ and $i \neq e$ exists with $\boldsymbol{\alpha}.i = \alpha$, while $\boldsymbol{\alpha}.j \in \{\alpha, nop\}$ and $g \models^j_{\mathcal{I}_0} P_t(\boldsymbol{\alpha}.j)$ for every $\langle j, g.j\rangle_t$.*

Example 3. (cont.d) The actions `blastA/B` are synchronization actions (modeling the firing action and the 'being hit' action of robots). According to the definition above, when the example is modeled as an interleaved PMAS, then `blastA` is not guaranteed to destroy all targets because not all agents in location `A` are forced to synchronize with such action. In fact, the two cases in which a blast destroys all agents in the location, or just a subset, are elegantly captured by simply assuming a concurrent or interleaved semantics. In the former, a global transition including `blastA` (by the environment) is legal iff all agents execute `blastA` as well if this action is executable, that is, if they can be hit because they are in the location at which the cannon is firing.

Runs of PMASs and the Reachability Problem. Based on the one-step definition of (legal) global transition, we now define the notion of runs for concurrent and 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 $\rho = g^0 \xrightarrow{\boldsymbol{\alpha}^1} g^1 \xrightarrow{\boldsymbol{\alpha}^2} \cdots$ 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 snapshot, i.e., with all concrete agents in their initial local state. A global transition as above specifies how each concrete agent $g.j$ evolves depending on the nature of the action $\boldsymbol{\alpha}.j$. 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 5. *An agent formula ψ_{goal} is **reachable** in $\mathcal{M}$ iff there exists $\mathcal{I}_0$ and an initial snapshot g^0 of $\mathcal{M}$ s.t. a snapshot g with $g \models_{\mathcal{I}_0} \psi_{goal}$ is reachable through a run $\langle \rho, \mathcal{I}_0\rangle$ from g^0.*

The verification task at hand is to assess whether ψ_{goal} is reachable, i.e. $\mathcal{M}$ is **unsafe** w.r.t. ψ_{goal}. If a formula is unreachable then it is so for any number of agents and all possible interpretations. In such a cases, $\mathcal{M}$ is said to be **safe** w.r.t. ψ_{goal}.

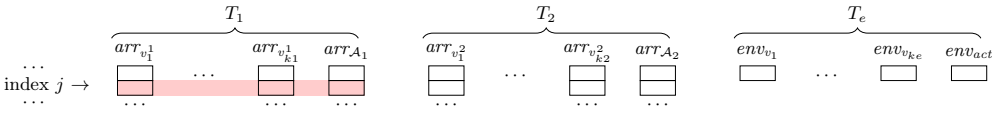


Fig. 2. A depiction of the encoding of agent templates and environment.

4 A Practical Solution to the Reachability Problem for PMASs

In this section, we illustrate our implementation approach and the tool called *SAFE*, which makes available the model checker MCMT [26] for checking the safety of PMASs. Through this tool-chain, one can (*i*) specify a PMAS $\mathcal{M}$ with *SAFE* and then (*ii*) check its safety with MCMT.

Theorem 1. *Given a PMAS $\mathcal{M}$ and a formula ψ_{goal}:*
- *If $\mathcal{M}$ is interleaved then MCMT always terminates with the correct answer (i.e., MCMT provides a full decision procedure);*
- *If $\mathcal{M}$ is concurrent, if MCMT terminates reporting that the PMAS is safe, then the answer is correct. If the PMAS is unsafe, then MCMT will terminate with the correct answer.*

The results stated in the previous theorem follow from the fact that MCMT implements the formal framework of array-based systems [25, 14]: this formalism allows us to encode PMASs into suitable state and transition formulae conforming the format required by MCMT and respecting the desired execution semantics. The proof should be adapted following the line of reasoning from [14] (for interleaved PMASs) and [27] (for concurrent PMASs). Note that, if a concurrent PMAS is safe, MCMT may still report an incorrect answer.

These results are consistent with results in the literature (see Section 2). Indeed, the model checking solution in [29] guarantees a correct verification outcome when a cutoff exists, otherwise the procedure halts with no result. 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 abstracted system, which has to be computed first. In our approach, the theorem above provides guarantees based only on the assumptions on the type of PMAS (whether it is interleaved or concurrent).

4.1 MCMT: Model Checker Modulo Theories

MCMT [26] is a declarative and deductive symbolic model checker for safety properties of infinite-state systems, based on backward reachability and fix-points computations (with calls to an SMT solver). The input to the software is a textual representation of an *Array-based System* (ABS), briefly introduced below, which essentially are a symbolic representation for infinite-state systems. Given an ABS and an existential formula (as the agent formulae in Section 3.1), MCMT is able to check whether a state satisfying the formula is reachable or not, which exactly matches our verification task (Def. 5). If the system is unsafe, a *witness* is provided.

ABSs. "Array-based Systems" [25] 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. Intuitively, they describe a system that, starting from an initial configuration (specified by an initial formula), is progressed through transitions (specified by transition formulae). The precise definition depends on the specific application, and it makes use of a multi-sorted theory with 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. We exploit this to capture the possible evolutions of a PMAS through suitable transition formulae so that we can use MCMT for verification, as it supports both forms of existential and universal quantification needed for expressing interleaved and concurrent execution semantics of PMASs. If universal quantification is used, a warning is issued, as this may generate spurious runs (see Thm. 1).

For lack of space, in this paper we do not provide a thorough formal treatment of ABS nor of how a PMAS can be encoded into an ABS, but we directly show and comment the resulting textual encoding, in the next section. Consistently with this, ABSs are only used as the internal representation of our implementation, and we do not require users of our implementation to know how these work nor how to manually encode a PMAS into an ABS.

In Figure 2 we depict the main intuition behind the overall encoding approach: an ABS features array variables and simple variables (called global variables). The variables in each agent template are encoded by array variables, so that the value of each variable 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 for the arrays for template T_t (e.g., the red area in Figure 2). The environment variables are instead encoded via global variables, because only one concrete environment exists.

4.2 MCMT input files for interleaved and concurrent PMASs

We now exemplify the textual encoding by listing the salient portions of the input file for our running example. An input file for MCMT is a textual representation of an ABS that includes *(i)* the declaration of individual and array variables, *(ii)* the initial state formula, *(iii)* the goal state formula, *(iv)* the list of transition formulae that specify how the system evolves. As stated in the previous section, these files are only used as the internal representation of our implementation, but we do not require users to know how these work nor how to manually encode a PMAS.

With the exception of minor syntactic details, the textual encoding is the same for the interleaved and concurrent cases, hence in this paper we focus on the former case only.

First, we define the sorts that are used in the ABS, which intuitively represent the variables types that we are going to use. For the running example, we have actions, booleans, strings used for locations (`init`,`A`,`B`,`target`), one `turn` type used to implement the alternation between the cannon and the robots, and finally an internal book-keeping type `Phase` that we need for the encoding. The encoding captures all possible evolutions of the PMAS by restricting how global and array variables are be updated at each step, using a flag of type `PhaseSort` to guide the progression.

```
:smt (define-type Action)                    :smt (define-type turnSort)                  :smt (define-type BOOLE)
:smt (define-type StringLoc)                 :smt (define-type PhaseSort)
```

Then, constants and relations are declared with their corresponding sort:

```
:smt (define Snow ::(-> StringLoc StringLoc )):smt (define blastA ::Action)               :smt (define target ::StringLoc)
:smt (define pulseA ::Action)                :smt (define blastB ::Action)               :smt (define P0 ::PhaseSort)
:smt (define pulseB ::Action)                :smt (define TRUE ::BOOLE)                  :smt (define PL ::PhaseSort)
:smt (define gotoA ::Action)                 :smt (define FALSE ::BOOLE}                 :smt (define PS ::PhaseSort)
:smt (define gotoB ::Action)                 :smt (define A ::StringLoc)                 :smt (define turnTEMPS ::turnSort)
:smt (define goTargetA ::Action)             :smt (define B ::StringLoc)                 :smt (define turnREST ::turnSort)
:smt (define goTargetB ::Action)             :smt (define init ::StringLoc)
```

Then, local and global variables are declared (see Figure 2 and its description):

```
:local locATT StringLoc                      :local actATT Action                        :global actEnv Action
:local destroyedATT BOOLE                    :global pulseLoc StringLoc                  :global phase PhaseSort
```

The three arrays (the local variables) are used, respectively, to hold the location of robots, their destroyed (boolean) state, the actions they currently selected for execution (array `actAtt`). The remaining three (global) variables are used to hold the location toward which the pulse is currently directed, the action selected for execution by the environment, the phase value needed for encoding the transitions (i.e., a variable `phase` of type `PhaseSort`).

Further, the initial states are represented by means of a conjunctive formula, where `x` is a variable of sort *array index*, implicitly quantified universally:

```
:initial
:var x
:cnj (= pulselocE NULL_StringLoc) (= actE NULL_Action) (= locATT[x] init) (= destroyedATT[x] FALSE)
     (= actATT[x] NULL_Action) (= turn turnTEMPS) (= phase P0)
```

The conditions above require that: the EMP pulse is not directed towards any waypoint, the environment has declared no action, all robots are in their initial state, no robot is destroyed, no robot declared yet an action, the turn is of the cannon (and the book-keeping variable `phase` is equal to 0). Note that nothing is said about the *Snow* relation, as we want to check that the system is safe irrespective of the snow conditions.

Further, we specify the safety formula to verify. For our example, this formula requires that at least one robot exists (we use an index `z1`, implicitly quantified existentially) so that the cell of array `locATT` has value equal to `target`, namely the robot is in the target location:

```
:u_cnj (= locATT[z1] target)
```

The remaining portion of the file is constituted by a list of transition formulae, which essentially encode the possible ways in which the ABS can evolve. These are constituted by a *guard* condition and a sequence of *if-then-else* updates.

For instance, the transition formula below specifies the conditions (guard) for allowing a robot to declare an action (in this case, action `gotoB`). The effect of this transition is to write `gotoB` in a suitable cell of the array `actAtt`. The guard is composed of six conjuncts: action `gotoB` can be declared by a robot (identified by the existentially quantified index variable `x` – i.e., agent *self* in Section 3.1) when the robot in the initial location, no action was already declared by the robot, the robot is not destroyed, the EMP pulse is not directed at waypoint `B` and there is no snow on the path between the initial location and `B`. The two cases in the second and third column, capturing an *if-then-else* update, make sure that only the cell corresponding to index `x` (i.e., *self*) is updated, while all other cells in the arrays remain unchanged (namely, the first case is applied for the array index `j` $=$ `x` and the second to all array indexes `j` $\neq$ `x`). By simply repeating a variable name, we specify that the variable is unchanged: in the second column, the value of array `actATT` (that holds the actions declared by all attacking robots for the current round) is updated to `gotoB` for the array index `j=x`, whereas in the third column we repeat `actATT[j]`.

```
:transition                          :numcases 2
:var j
:var x                               :case (= x j)                    :case
:guard (= phase 0)                      :val locATT[j]                   :val  locATT[j]
       (= actATT[x] NULL_Action)        :val destroyedATT[j]             :val destroyedATT[j]
       (= locATT[x] init)               :val gotoB                       :val actATT[j]
       (= destroyedATT[x] FALSE)        :val pulseLoc                    :val pulseLoc
       (not (= pulseLoc B))             :val actEnv                      :val actEnv
       (= Snow (start B) FALSE)         :val L                           :val L
```

The following is an example of transition in which a synchronization action (`blastA`) is declared by the environment together with one robot (see Definition 4), capturing the fact that that robot will be hit by the blast when the effects of the action will be applied in a subsequent transition (before that, more robots will be allowed to participate in the synchronization, i.e., to declare the same action). The guard makes sure that an agent index exists (variable `x`) so that the agent is in waypoint `A` and it has not yet declared an action (this will be the robot that will be hit), and that a further index exists (variable `y`, with value possibly equal to the value of `x`) that is in `A` (this is the robot that satisfies the protocol function of the action `blastA`, requiring the existence of a possible target). In the first case (second column), the value of `actAtt[x]` is updated to `blastA`.

```
:transition                          (= locATT[x] A)                  :numcases 2
:var x                               (= locATT[y] A)
:var y                               (= destroyedATT[y] FALSE)        :case (= x j)
:var j                                                                   :val locATT[j]
:guard (= phase 0)                                                       :val destroyedATT[j]
       (= actATT[x] NULL_Action)                                         :val blastA
```

```
:val pulselocE
:val actE
:val S
                                    :case
                                       :val locATT[j]
                                       :val destroyedATT[j]
                                                                    :val actATT[j]
                                                                    :val pulselocE
                                                                    :val actE
                                                                    :val S
```

The encoding of the rest of the transitions follows the same approach, although one has to manually write all the transitions, which is a delicate and cumbersome task, prone to error (the running example, for the interleaved execution semantics, requires 26 transitions). This justifies the need of a user-oriented approach, which we comment in Section 4.3.

Executing MCMT. Once the textual encoding is done, MCMT can be simply executed via command line, specifying as argument the textual file: `./mcmt file.txt`. For more details and options, please refer to the MCMT manual [1].

4.3 *SAFE*: the Swarm Safety Detector

Here we present and illustrate *SAFE* [2], i.e., our own implementation of a user interface that allows to directly employ in practice the results presented in this paper. The tool automatizes the textual encoding of the PMAS into MCMT input files (i.e., ABS files), by relying on a *MAS-oriented* modeling framework. This allows the user to focus on modeling the PMAS, i.e., the agent templates and the environment template, without worrying about how their constructs can be encoded for MCMT under the distinct execution semantics. The tool also allows to convert the witnesses for unsafety that MCMT returns (when the input ABS is unsafe) back into executions of the original PMAS. *SAFE* is a user-friendly, effective, implemented approach for modeling and verifying the safety of PMASs. Some preliminary experimental evaluation is discussed in Section 5.

The *SAFE* modeling interface. The interface of *SAFE*: the Swarm Safety Detector is available at [2] in its *Base* version, which allows to model and interleaved PMASs and encode them into ABSs. Further versions, called *SAFE For all* and *SAFE Data*, allow to model concurrent PMASs and their *data-aware* extensions, and are currently in development. Figure 3 shows the running example of this paper as it appears in the *SAFE* GUI.

***SAFE* 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. Indeed, Definition 1 defines an agent templates as a labeled, finite-state transition system, whereas *SAFE* assumes a more succinct representation, where instead of listing explicitly the local states and transitions of the templates, we assume a STRIPS-like approach. Therefore, instead of an explicit finite-state machine labeled by actions, actions are specified by means of pre- and post- conditions;

Another difference, with respect to the formal model presented here, is that the current version of *SAFE* Base does not allow to use disjunction in action preconditions. This implies that, whenever we want to model an action label a that is available under two distinct conditions (i.e., a precondition of the form $\phi_1 \vee \phi_2$), we need instead to use a distinct copy a' of the action and assign to these the preconditions ϕ_1 and ϕ_2, respectively. Future versions will avoid this restriction.

At the same time, *SAFE* Base includes some minor features that were not included in the formal model discussed here, as these are primarily implementation details:

- Apart from local actions and synchronization actions, the tool allows a further type of actions, called *individual* synchronization actions, that can only be executed by the environment and by exactly another agent at the same time;
- The *turn-based* mechanics, which allows to alternate the actions of two distinct subsets of the templates (as in our running example), is a core-feature of the tool and can be easily enabled or disabled without the need of manually implementing the alternation logic. In order to make this compatible with the execution semantics for synchronization actions, *SAFE* Base allows an additional annotation of these actions, by which it is possible to specify the *initiator* template of each synchronization action.

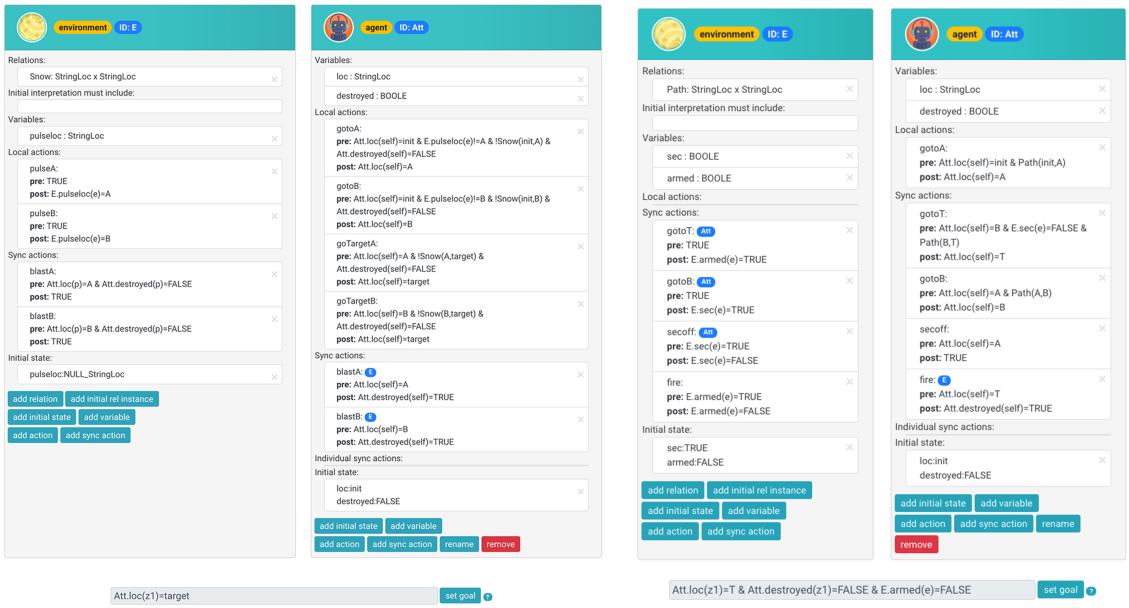


Fig. 3. The main part of the *SAFE Base* interface, showing (on the left) the PMAS model for the running example and (on the right) the example in Section 5.1. The GUI provides intuitive buttons to either add or remove any component from the templates, so that there is no manual coding required.

5 Execution of *SAFE*-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 1.67 seconds using Z3 (version 4.8.9.0) as background SMT solver. MCMT correctly reports that the system is unsafe. The input file, of which some parts are listed and commented in the previous section, has 3 local variables for the robot template *att*, 3 global variables and 26 transitions. MCMT gives in output this sequence as witness of unsafety: `[t2][t17][t3_1][t15][t1][t16][t5_1][t15]`, where each `t`n represents the execution of the n-th transition in the input file, following the order in which they appear. In this case, they correspond to the following sequence of actions: `pulseB, gotoA, pulseA, goTarget`. Trivially, the robots reach the target while avoiding the EMP pulse used by the cannon, which in this instance does not even attempt to use the blasts to destroy robots.

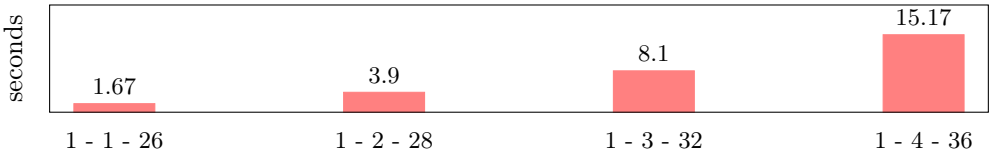
When a PMAS is determined to be unsafe, *SAFE* does not provide support for embedding into the model the witness provided by MCMT, so it is the responsibility of the user to update their MAS by taking insights from the witness and then check the new model again.

By increasing the number of agent templates and number of waypoints required to reach the target on either of the two paths (i.e., by having waypoints $\mathtt{A}_1, \ldots, \mathtt{A}_n$ and $\mathtt{B}_1, \ldots, \mathtt{B}_n$), we can test the scalability of our approach (i.e., the use of *SAFE* and MCMT for checking safety of PMASs) with respect to the minimum *length* of possible runs of the PMAS that achieve the goal formula. In these versions of the running example, cannon blasts hit all locations on the same path simultaneously, the EMP pulse can block all robots on the entire path at which it is directed, and the protocol of the cannon is so that it can freely fire at either path without checking that there are available robot targets. This is achieved by adding a further `path` variable to robot templates. The number of transitions in the encoding also increases.[1]

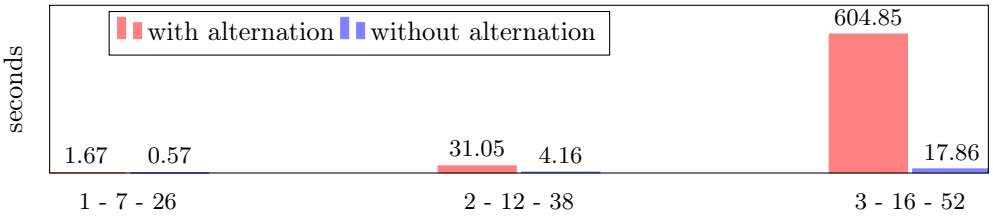
As it can be seen from the experiments, the number of transitions and execution time increase slightly. The length of the shortest unsafe runs also increases. There is of course a direct relationship between the length of the PMAS runs that must be checked and the *depth* of the state-space

[1] These examples are available at: http://safeswarms.club/page/mcmt/exZb with z=1..4

exploration of MCMT, although the number of possible combinations of actions, for the agents and environment, remains the same at each step.



If instead we increase the number of templates, by introducing further *copies* of the robot template from the original problem instance (preserving only waypoints `A` and `B`), then both the number of transitions and variables increase substantially, leading to much longer execution times. This was expected, as the possible global states of the PMAS, as well as the number of possible combinations of actions, grow exponentially. We report below (in red color) on the average execution times for the case of 1-3 robot templates (in addition to the environment template).[2] This shows how the cost of the verification is greatly affected by the number of templates. Interestingly, if we remove the assumption that the agents and the environment alternate their moves (thus also removing the global variable `turn` that is automatically added by *SAFE*), determining the unsafety of the PMAS becomes much faster (times shown in blue).[3]



Contrary to intuition, the verification becomes more challenging as the PMAS becomes more specified (that is, with more detailed pre- and post-conditions of actions, turn alternation flags, goal conjuncts). This may constitute a limitation to the use of *SAFE* and MCMT for larger examples, although the examples in the literature involve typically at most two agent templates. Moreover, we should keep in mind that these problem instances are intrinsically computationally demanding. For example, for the case of 3 agent templates, more than 3M calls are made to Z3, and more than 2.5k nodes were explored.

5.1 A further example

We present in this subsection a second example, which we encode in *SAFE* to obtain the corresponding MCMT input file. This example differs from the previous in that the unsafety of the system does not depend on the possibility that a proper strategy is not enacted, but rather on the ability of the agents to coordinate so that the goal formula is reached. Note that the problem specification does not contain hints on how many agent instances are needed for this to happen.

Description of the example. A swarm of robotic agents wants to reach a protected room denoted as C. To do so, they have to first move to a room A, then to a room B, then to C. The corridors between these locations are either open or closed, and this is not known nor controllable. Moreover, a security system prohibits to move from B to C unless the system is first switched off 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 EMP pulse to disable robots in the 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, that there can be robots in C which are not disabled. By careful analysis, we can see that the

[2] These examples are available at: http://safeswarms.club/page/mcmt/exZ with `z`=1..3
[3] To replicate these experiments, it is sufficient to disable the template alternation through the GUI.

answer is positive if all corridors are open: at least two robots need to move to A, then B. At least one further robot can then disable the security system after moving to A, so that the others can enter C. Then, after the EMP is activated, there are chances that one robot will not be disabled. Note how this relies on the ability of considering that more robots are in C.

We use a template T_{att} for robots, with variables `room` (enumeration $\lceil$`init,A,B,C`$\rceil$) and `disabled` (boolean). For the environment template T_e, `secON` and `armed` are used for specifying whether the security system is on, and whether the pulse is armed. Template T_{att} has actions `goA`, `goB` and `goC`, plus additional actions `off`, `pulse` representing the action of switching off the security system and of "being disabled" by the pulse. The environment has actions `goB`, `goC`, `off`, as T_{att} has, because these are synchronization actions which have an impact on the environment: 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 its interpretation.

For example, given a global state in which an agent instance of T_{att} is in local state l and T_e is in local state l_e, the agent can execute a transition $l \xrightarrow{\texttt{goC}} l'$ only if `Corr(B,C)` is in $\mathcal{I}_0$, $val_{att}(l, \texttt{room}) =$ `B` and $val_e(l_e, \texttt{secON}) =$ `false` in T_e, and the resulting local state l' of the agent is such that $val_{att}(l', \texttt{room}) =$ `C` while the environment reaches a local state l'_e so that $val_e(l'_e, \texttt{armed}) =$ `true` (plus further assignments for inertia). Other actions are defined in a similar manner. For instance, the protocol of `off` in T_{att} is $\texttt{room}^{self} =$ `A`.

Execution in MCMT. Figure 3 shows this example modeled in *SAFE*. The resulting MCMT input file (i.e., the textual encoding of the ABS corresponding to the PMAS) is solved by MCMT v.3.0, on the same machine as before, 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.[4] The generated input file ("download MCMT input") contains 501 lines of code and has 3 local variables for T_{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 `t_n` represents the execution of the n-th transition in the input file, following the order in which they appear (subscripts after the first one refer to the number of instantiated index variables). In this case, they correspond to the following sequence of actions, where each action is executed by one agent (or by the environment): `goA,goA,goA,goB,goB,off,goC,goC,fire`.

6 Conclusions and Future Work

In this paper, we have presented a model of parameterized multi-agent systems, defined the verification task of checking whether the model is safe, and provided a custom, MAS-oriented tool that allows to make use of a generic SMT-model checker off-the-shelf.

Our technique is based on a very well-understood SMT-based theory for which a number of results of practical applicability already exist, and research is active. To the best of our knowledge, the usage of SMT techniques in the context of multi-agent systems is novel: for the first time in order to verify safety of parameterized MASs we exploit a state-of-the-art model checker (MCMT) that discharges safety tests via proof obligations to an efficient SMT-solver. The advantages of using the SMT technology are several: *(i)* first of all, it allows us to exploit a full-fledged *declarative* and symbolic framework where all system specifications can be entirely given with only logical formulae; *(ii)* a plethora of effective techniques for symbolic reasoning are available, like decision procedures for combined theories or quantifier handling through instantiation and quantifier elimination; *(iii)* the theories underlying the SMT-solvers guarantee a large expressivity and flexibility, as well as the possibility of integrating techniques like acceleration, predicate abstraction and invariant synthesis; *(iv)* one of the most important features of an SMT-based approach is the generality, since there is a large spectrum of other classes of systems (distributed, timed, fault tolerant, sequential systems etc.) which this framework can be integrated with.

[4] The example, modeled via *SAFE*, is publicly available at: http://safeswarms.club/page/mcmt/rooms

Finally, note that the background theories employed by the SMT-solver in this paper are only the empty theory or EUF (although, these theories are customary and very significant in the SMT literature): more involved theories are not used because the empty theory or EUF suffices to formalize the uninterpreted symbols characterizing FO interpretations we deal with. Nevertheless, the possibility of extensions is readily available thanks to our work. We can easily introduce theories with axioms for constraining array elements: e.g., elements can be retrieved from full-fledged relational databases with constraints (e.g. key dependencies), as studied in [14, 11, 31]. This proves beneficial both foundationally and practically, and opens up a number of interesting continuations of our work. From the foundational perspective, thanks to the connection between PMASs and ABSs, *data-aware* extensions of our framework can be directly incorporated, along the line studied in [14]. This supports finite action signatures with infinite number of possible parameter values, and also to store and inspect infinite data values. In addition, there are several other already available extensions of ABS, like the use of restricted arithmetics, cardinality constraints, that are all now concretely usable directions for checking safety of PMASs.

From the applied perspective, as argued before, any advancements for SMT on array-based systems can in principle have a direct application: the effective techniques provided by SMT technology can be exploited for performing symbolic reasoning in these natural extensions. At the same time, the existence of advanced heuristics and approximation techniques triggers a natural continuation of our tool-chain, tailored to efficiency issues. In fact, it is well-known that the performance of symbolic verification techniques can be improved by orders of magnitude if such techniques are suitably developed for the domain at hand [16]. A thorough experimental evaluation of this approach is left as future work.

References

1. MCMT: Model Checker Modulo Theories. http://users.mat.unimi.it/users/ghilardi/mcmt, accessed: 2020-09-01
2. SAFE: the Swarm Safety Detector. http://www.safeswarms.club, accessed: 2020-09-01
3. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: Proc. of 11th Annual IEEE Symposium on Logic in Computer Science. pp. 313–321 (1996)
4. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: An extension of lazy abstraction with interpolation for programs with arrays. Form. Methods Syst. Des. **45**(1), 63–109 (2014)
5. Alberti, F., Ghilardi, S., Sharygina, N.: A framework for the verification of parameterized infinite-state systems. Fund. Inform. **150**(1), 1–24 (2017)
6. Alechina, N., Brázdil, T., De Giacomo, G., Felli, P., Logan, B., Vardi, M.Y.: Unbounded orchestrations of transducers for manufacturing. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence. pp. 2646–2653 (2019)
7. Barrett, C.W., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Model Checking., pp. 305–343 (2018)
8. Belardinelli, F., Kouvaros, P., Lomuscio, A.: Parameterised verification of data-aware multi-agent systems. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017. pp. 98–104 (2017)
9. Bloem, R., Jacobs, S., Khalimov, A.: Decidability of Parameterized Verification. Morgan & Claypool Publishers (2015)
10. 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)
11. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Formal modeling and SMT-based parameterized verification of data-aware BPMN. In: Proc. of BPM. LNCS, vol. 11675. Springer (2019)
12. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: From model completeness to verification of data aware processes. In: Description Logic, Theory Combination, and All That. LNCS, vol. 11560. Springer (2019)
13. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Model completeness, covers and superposition. In: Proc. of CADE. LNCS, vol. 11716. Springer (2019)
14. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: SMT-based verification of data-aware processes: a model-theoretic approach. Mathematical Structures in Computer Science **30**(3), 271–313 (2020)

15. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018)
16. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaïdi, F.: Invariants for finite instances and beyond. In: Proceedings of FMCAD. pp. 61–68. IEEE (2013)
17. Condurache, R., De Masellis, R., Goranko, V.: Dynamic multi-agent systems: Conceptual framework, automata-based modelling and verification. In: Proc. of PRIMA 2019: Principles and Practice of Multi-Agent Systems - 22nd International Conference. pp. 106–122 (2019)
18. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: McAllester, D.A. (ed.) Automated Deduction - CADE-17, 17th International Conference on Automated Deduction. Lecture Notes in Computer Science, vol. 1831, pp. 236–254. Springer (2000)
19. Emerson, E.A., Kahlon, V.: Model checking guarded protocols. In: 18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003. pp. 361–370. IEEE Computer Society (2003)
20. Emerson, E.A., Namjoshi, K.S.: On reasoning about rings. International Journal of Foundations of Computer Science **14**(4), 527–550 (2003)
21. Esparza, J., Ganty, P., Leroux, J., Majumdar, R.: Verification of population protocols. Acta Inf. **54**(2), 191–215 (2017)
22. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science **256**(1), 63–92 (2001), iSS
23. Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Petri nets with parameterised data: Modelling and verification. In: Proc. of BPM. LNCS, Springer (2020)
24. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Towards SMT model checking of array-based systems. In: Proc. of IJCAR. pp. 67–82 (2008)
25. Ghilardi, S., Ranise, S.: Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. Log. Methods Comput. Sci. **6**(4) (2010)
26. Ghilardi, S., Ranise, S.: MCMT: A model checker modulo theories. In: Automated Reasoning, 5th International Joint Conference, IJCAR 2010. pp. 22–29 (2010)
27. Ghilardi, S., Ranise, S., Valsecchi, T.: Light-weight smt-based model checking. Electron. Notes Theor. Comput. Sci. **250**(2), 85–102 (2009)
28. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Counter attack on byzantine generals: Parameterized model checking of fault-tolerant distributed algorithms (2012)
29. Kouvaros, P., Lomuscio, A.: Parameterised verification for multi-agent systems. Artif. Intell. **234**, 152–189 (2016)
30. Kouvaros, P., Lomuscio, A., Pirovano, E., Punchihewa, H.: Formal verification of open multi-agent systems. In: Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, 2019. pp. 179–187 (2019)
31. Li, Y., Deutsch, A., Vianu, V.: VERIFAS: A practical verifier for artifact systems. Proceedings of the VLDB Endowment **11**(3), 283–296 (2017)
32. Pnueli, A., Xu, J., Zuck, L.: Liveness with (0,1, infty)- counter abstraction. In: Brinksma, E., Larsen, K.G. (eds.) Computer Aided Verification. pp. 107–122. Springer Berlin Heidelberg (2002)