Verifying Temporal Properties of Stigmergic Collective Systems Using CADP

Luca Di Stefano, Frédéric Lang

▶ **To cite this version:**

Luca Di Stefano, Frédéric Lang. Verifying Temporal Properties of Stigmergic Collective Systems Using CADP. ISoLA 2021 - 10th International Symposium on Leveraging Applications of Formal Methods, Oct 2021, Rhodes, Greece. pp.473-489, 10.1007/978-3-030-89159-6_29 . hal-03385131

HAL Id: hal-03385131
https://inria.hal.science/hal-03385131

Submitted on 19 Oct 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Verifying temporal properties of stigmergic collective systems using CADP

Luca Di Stefano(✉)[0000−0003−1922−3151] and Frédéric Lang

Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP*, LIG, F-38000 Grenoble, France
luca.di-stefano@inria.fr
http://convecs.inria.fr

Abstract. We introduce an automated workflow to verify a variety of temporal properties on systems of agents that interact through virtual stigmergies. By mechanically reducing the property and the system under verification to an MCL query and a sequential LNT program (both MCL and LNT being languages available in the CADP formal verification toolbox), we may reuse efficient model-checking procedures that can give us a verdict on whether the property is satisfied by the system. Among other things, this procedure allows us to verify that a system satisfies a given predicate infinitely often during its execution, which is an improvement over previous verification approaches. We demonstrate the capabilities of this workflow by verifying a selection of example systems. Additionally, we present preliminary results showing that this workflow may also generate parallel LNT programs and exploit compositional verification techniques, which is likely to improve the analysis performance.

Keywords: Collective adaptive systems · Virtual stigmergy · Temporal logics · Model checking.

1 Introduction

Collective adaptive systems (CAS) are composed of several *agents*, with limited capabilities and knowledge, that interact together to reach a common goal [41]. When studying the evolution of these systems, one can often witness global phenomena that arise from the seemingly chaotic interaction of agents, such as the emergence of a coherent pattern of movement in a flock of birds or the spread of a popular opinion on a social network. While collective systems have traditionally been analyzed by looking at their aggregate features (for instance, by using general equilibrium theory to describe an economy), there is a growing interest in understanding what kind of local behaviour and which communication mechanisms may give rise to such global effects. These *individual-based* approaches are increasingly being used in hard and soft sciences alike, with applications ranging from economics [32], to epidemiology [27], to social sciences [18]. Additionally, replicating those mechanisms in man-made systems is likely to make them more adaptive, autonomous, and robust with respect to individual failures [3, 34].

Stigmergies are one example of such a mechanism. A stigmergy is a mode of communication where agents interact by means of traces in a shared environment.

* Institute of Engineering Univ. Grenoble Alpes

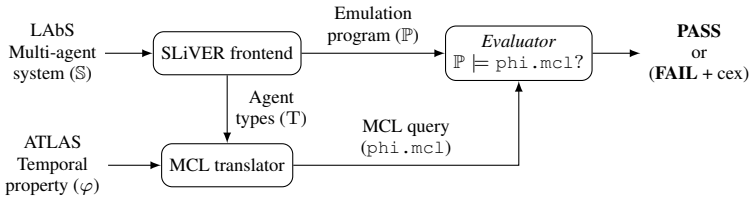

Fig. 1: Our verification workflow.

Stigmergic systems feature agents that can coordinate and reach common goals just by leaving and interpreting these traces, even though no direct communication ever happens between them. While stigmergies were originally introduced to model the construction of termite nests [19], they have found widespread adoption as a conceptual tool to describe several collective systems, from colonies of foraging ants [33] to the Wikipedia collaborative encyclopedia [5]. Thus, they may be useful to scholars across many different disciplines which aim at understanding self-organization and spontaneous coordination [22]. However, the asynchronous nature of stigmergic interaction, coupled with the potentially nondeterministic behaviour and interleaving of agents, results in systems with a large state space. Thus, automated procedures are needed to formally guarantee the correctness of such systems.

In this paper, we introduce ATLAS (A Temporal Logic for Agents with State), a small formalism to express a variety of temporal properties about stateful agents. Then, we put forward an automated workflow to verify properties expressed in this formalism against stigmergic systems specified in the LAbS language [6]. This language allows to describe systems where the stigmergic medium is a distributed data structure [35], and where interaction among agents may be constrained by predicates on their exposed features. Our workflow reduces the system's specification to what we call an *emulation program* in the LNT process language [16], and translates the desired property into a query in the MCL property language [31]. Both LNT and MCL are languages available in the CADP formal verification toolbox[1] [15]. Our workflow can then use CADP tools to model-check the query against the program to obtain a verdict about whether the original specification satisfies the given property (Figure 1). This workflow brings several improvements over previous LAbS verification approaches [11]. For instance, the present work introduces the capability of verifying that a system satisfies a given property infinitely often during its evolution. We demonstrate the effectiveness of this approach by verifying a selection of illustrative stigmergic systems.

As an additional contribution, we describe an alternative approach based on the generation of *parallel* LNT programs, i.e., where components of the system are implemented as separate processes. We then provide preliminary experimental results,[2] showing that these programs can be efficiently verified by means of compositional techniques [28].

[1] http://cadp.inria.fr

[2] Artifacts related to the experiments presented in this work are available as a repository (https://gitlab.inria.fr/ldistefa/labs2lnt-artifacts).

This paper is structured as follows. Section 2 introduces the necessary background information. Section 3 describes recent improvements to our emulation programs, as well as the ATLAS property language and its reduction to MCL queries. In Section 4, we demonstrate our approach by performing verification tasks on a collection of illustrative systems. Section 5 discusses how this workflow can be naturally extended to take advantages of compositional verification techniques, and presents some preliminary experimental results in that direction. Lastly, Section 6 discusses related work and Section 7 contains our conclusions, together with directions for future work.

2 Background

2.1 The CADP toolbox

CADP [15] is a software toolbox for the analysis of asynchronous concurrent systems, described in languages whose semantics is expressed in terms of an LTS (labelled transition system). It contains a wide range of tools for simulation, test generation, verification (model checking and equivalence checking), performance evaluation, etc.

LNT is one of the input languages available in CADP to formally describe asynchronous concurrent systems [16]. A system is modeled as a process, generally composed of several, possibly concurrent processes, which may perform communication actions on gates and exchange information by multiway (value-passing) rendezvous, in the style of the Theoretical CSP [23] and LOTOS [24] process algebras. The syntax of LNT is inspired from both imperative languages (assignments, sequential composition, loops) and functional languages (pattern matching, recursion), with many static checks, such as binding, typing, and dataflow analysis ensuring the proper definition of variables and function results. A compiler for LNT generates the LTS corresponding to a main process, either as an explicit enumeration of its states and transitions (BCG graph) or in the form of an API (initial state and successor function) for on-the-fly verification.

MCL [31] is an action-based temporal logic based on the alternation-free fragment of the modal μ-calculus [26], extended with regular action formulas and value-passing constructs. It subsumes both branching-time and linear-time formalisms (e.g., CTL [4] and LTL [36]), allowing to express a quite wide range of temporal properties. The MCL language is built upon action formulas α, path formulas β, and state formulas φ. Basically, an action formula is a pattern that is intended to match some of the system's actions. So doing, MCL allows data-values present in the matched system's actions to be captured and stored in data variables, in order to be used in subsequent action or state formulas. For instance, if the system has an action of the form "G !1 !2", then it is matched by the action formula "G *?x : int* !2" and the value of x takes the value 1. A path formula is basically a regular expression built on action formulas, which enables arbitrarily long sequences of actions to be specified. State formulas are built from: Boolean operators; predicates on data variables; the possibility modality $\langle\beta\rangle\varphi$ denoting the states with an outgoing path matching β and leading to a state satisfying φ; the necessity modality $[\beta]\varphi$ denoting the states all outgoing paths of which satisfying β lead to states satisfying φ; and two parameterized fixed point operators: the minimal fixed point operator $\mu X(x_1 : T_1 = e_1, \ldots, x_n : T_n = e_n).\varphi$ (to specify finite sequences recursively), and the maximal parameterized fixed point operator $\nu X(x_1 : T_1 = e_1, \ldots, x_n : T_n = e_n).\varphi$ (to specify

Listing 1: A simple model of flocking behaviour in LAbS.

```
stigmergy Dir {
  link=√((x1−x2)^2 + (y1−y2)^2) ≤ δ
  dx, dy: {-1, 1}, {-1, 1}
}
```

```
agent Bird {
  interface= x:[0..G], y:[0..G]
  stigmergies= Dir
  Behaviour=
    x, y←(x+dx) mod G, (y+dy) mod G;
    Behaviour
}
```

infinite sequences), where X is called a propositional variable and φ usually contains occurrences of X with actual values for the parameters $x_1, \ldots, x_n$ of types $T_1, \ldots, T_n$; the expressions $e_1, \ldots, e_n$ denote the parameter's initial values. MCL also allows action and state formulas to be embedded in parameterized macro definitions, to enable code reuse. Many additional high-level constructs inspired from programming languages (such as loops and conditionals) also exist, mostly introduced as syntactic sugar. A complete description can be found online in the MCL manual page.[3] The *Evaluator* model checker available in CADP can be used to evaluate temporal properties expressed in MCL, either on a BCG graph or on-the-fly through the API representing an LTS. A complete description of *Evaluator*'s features can be found in its online manual page.[4]

2.2 LAbS and SLiVER

In our experimental evaluation, we will consider emulation programs generated from LAbS specifications. The LAbS language [6] allows to describe systems of agents that communicate indirectly via *virtual stigmergies*. A virtual stigmergy [35] is a distributed key-value store that allows an agent to asynchronously diffuse its own (local) knowledge to neighboring agents. An agent can manipulate a stigmergy simply by assigning values to specific local variables (which we call *stigmergic variables*). Whenever this happens, the agent also computes a *timestamp* to record the moment when the assignment took place. Then, the agent asynchronously advertises the new value by sending messages to its neighbors containing the name of the variable, the new value, and the computed timestamp. Receivers will accept *newer* values for a variable (i.e., those with a higher timestamp than their local value) and advertise them as well, while they will reject *older* ones and react by advertising their own, newer, value. Agents will also send similar messages whenever they access the values of these variables (e.g., during evaluation of an expression). While the concept of *neighborhood* is commonly associated with spatial closeness, LAbS allows the user to define it in terms of *predicates* over the exposed features of agents. Furthermore, the user may equip different stigmergic variables with different definitions of neighborhood.

To illustrate LAbS, let us discuss a simplified version of the *Boids* algorithm for flocking agents [39]. An excerpt of the LAbS specification is shown in Listing 1. In this model, N "birds" are scattered on a 2D grid of size G. Every bird has a *position*

[3] http://cadp.inria.fr/man/mcl.html
[4] http://cadp.inria.fr/man/evaluator.html

$(x, y) \in [0, G-1] \times [0, G-1]$ and a *direction* of movement (dx, dy), for which we assume four possible values, i.e., those in the set $\{\pm 1\} \times \{\pm 1\}$. Lastly, the *behaviour* of each bird simply makes it move by repeatedly updating its position according to its direction: specifically, its new position is the sum (modulo G) of its current position and direction. The modulo operation, informally, means that the grid wraps around: for instance, an agent in position $(0, 0)$ can reach $(G-1, G-1)$ in a single step. It is worth noticing that the behaviour does not feature any communication constructs. Rather, we declare (dx, dy) as stigmergic variables, so that agents will advertise their direction every time they use it to update their position. We use a check on the Euclidean distance as the link predicate: two agents a_1 (the sender) and a_2 (the potential receiver) may only exchange a message if their distance is not greater than a given value δ. Intuitively, given a big enough δ and enough time, all birds should eventually have the same values for dx, dy, and thus they should move in the same direction.

The features described above lead to compact specifications of systems which may nonetheless display very large state spaces. Thus, formal verification becomes essential to prove that the specified systems do behave correctly and do not reach unwanted states. A possible way to verify a LAbS system $\mathbb{S}$ is by crafting and analyzing a sequential emulation program [10], i.e., a program $\mathbb{P}$ that can reproduce all possible executions of $\mathbb{S}$, without introducing spurious executions. $\mathbb{P}$ is *sequential* in the sense that the concurrent execution of the agents in $\mathbb{S}$ is reproduced by means of a nondeterministic, but non-concurrent scheduler, similarly to the way sequentialization is used to verify a piece of concurrent software by reducing it to an equivalent sequential program [37]. The SLiVER tool[5] aims at verifying LAbS systems by generating emulation programs in several languages, including LNT [11]. In fact, we will rely on a slightly modified version of SLiVER as part of our workflow.

3 Model checking sequential LNT emulation programs

As stated earlier, the automated workflow of Figure 1 translates a LAbS system and an ATLAS property into an LNT program and an MCL query, respectively. Then, it uses CADP's Evaluator to determine whether the program satisfies the query. This gives us a verdict on whether the system satisfies the property. In this section, we describe our workflow in more detail: we first outline several improvements in the generation of LNT programs (Section 3.1), and then introduce ATLAS (Section 3.2) and describe how we can encode ATLAS properties into MCL queries (Section 3.3).

3.1 Improving LNT emulation programs for LAbS

The SLiVER tool already demonstrated the feasibility of building and analyzing LNT emulation programs for LAbS specifications, by means of a provably correct structural encoding procedure [10]. In a previous version of SLiVER [11], the emulation program contained a *monitor* that encoded the property under verification; then, one simply used *Evaluator* to check the behaviour of the monitor and obtain a verdict on the satisfaction

[5] https://github.com/labs-lang/sliver

of the property. The emulation program also contained "diagnostic" transitions that were used to extract a counterexample in case of a property violation. However, these transitions did not follow a rigorous scheme: for instance, transitions resulting from assignments were different from those caused by stigmergic messages. In contrast, the current LNT code generator does not introduce a monitor, and its diagnostic transitions have been systematically revised to allow for MCL-based property verification (see Section 3.3). Namely, we let the emulation program signal every change of state (i.e., any change to one of the program's variables) by means of a visible transition, labelled as `assign !<id> !<varName> !<value>`, where `<id>` is the agent's unique identifier, `<varName>` the name of the changed variable, and `<value>` its new value. MCL queries are generally more efficient than the previous approach, and allow to verify multiple properties on the same program. Furthermore, this change allows us to progressively increase the fragment of supported properties without altering the LAbS-to-LNT code generator.

Additionally, we introduce a more efficient encoding of virtual stigmergies, resulting in emulation programs with a smaller state space that are thus easier to verify. While the original encoding explicitly modelled timestamps as natural numbers, we observed that the semantics of stigmergic interaction only ever needs to *compare* the timestamps of two agents. Thus, the new encoding just records these comparisons by associating to each stigmergic variable `var` a matrix $M_{\texttt{var}}$ of symbolic values `GREATER`, `SAME`, `LESS`. The intuition is that $M_{\texttt{var}}(i, j)$ is `GREATER` (resp. `LESS`) iff the value of `var` stored by the the i-th agent is newer (resp. older) than that of the j-th agent, and it is `SAME` iff the two values have the same timestamp. To enforce this rule, we must maintain the matrix in two ways. First, whenever agent i assigns a new value to `var`, we set $M_{\texttt{var}}(i, j)$ to `GREATER` for all agents $j \neq i$. Respectively, we set $M_{\texttt{var}}(j, i)$ to `LESS`. Furthermore, whenever an agent j successfully receives a value for `var` from another agent i, we update both $M_{\texttt{var}}(i, j)$ and $M_{\texttt{var}}(j, i)$ to `SAME`.

3.2 A basic property language for systems of stateful agents

We are interested in verifying temporal properties on an emulation program that represents a system of stateful agents. Naturally, we could encode any such property by manually writing an MCL query. However, doing so requires a good knowledge of both MCL and the structure of our emulation programs, and thus would be unsuitable for most users without a strong background in model checking. Furthermore, since agents are stateful, users may want to express *state-based* properties about them. Even though action- and state-based logics are essentially interchangeable [9], these users may feel at odds with the action-based MCL. What we propose, instead, is that the user should define properties in a higher-level language that trades off some of the expressiveness of MCL in exchange for more compact and intuitive properties, and encode such properties into MCL by means of a mechanizable procedure. We call this language ATLAS (A Temporal Logic for Agents with State).

The syntax of ATLAS is presented in Figure 2, where $\circ$ is a generic binary arithmetic operator ($+$, $-$, ...), $\bowtie$ is a generic comparison ($=$, $>$, ...), and $|e|$ is the absolute value of e. Furthermore, we assume that the agents in the system are partitioned into user-defined *types*, ranged over by T. A *value expression* is an arithmetic expression

$$
\begin{array}{lll}
e ::= \kappa \mid x.var \mid e \circ e \mid |e| & & \text{(value expression)} \\
p ::= e \bowtie e \mid x = x \mid \neg p \mid p \wedge p & & \text{(predicate)} \\
\psi ::= p \mid \exists x \in \mathrm{T} \bullet \psi \mid \forall x \in \mathrm{T} \bullet \psi & & \text{(quantified predicate)} \\
\phi ::= \mathsf{always}\, \psi \mid \mathsf{fairly}\, \psi \mid \mathsf{fairly}_\infty\, \psi & & \text{(temporal property)}
\end{array}
$$

Fig. 2: Syntax of ATLAS, a temporal property language for collective adaptive systems.

that may contain constants (κ) or refer to of agents' local variables: given a quantified *agent variable* x, $x.var$ evaluates to the value that x currently gives to its local variable var. A *predicate* can either compare two expressions ($e \bowtie e$), or check whether two agent variables refer to the same agent ($x = x$). Predicates can be negated, or can contain a conjunction of other predicates ($\neg p$, $p \wedge p$). Other compound predicates ($x \neq x$, $p \vee p$, ...) can be derived from these ones in the usual way. A *quantified predicate* is a predicate preceded by zero or more universal or existential (typed) quantifiers. We will only consider *sentences*, i.e., predicates where all agent variables are in the scope of some quantifier. Every sentence has a definite truth value in every state of a system: that is, a state either satisfies a sentence or not. Finally, a *temporal property* specifies *when* a quantified predicate should hold during the execution of a system. Property $\mathsf{always}\, \psi$ states that ψ should be satisfied in every state of the system: $\mathsf{fairly}\, \psi$ says that every *fair* execution of the system should contain at least one state satisfying ψ; lastly, $\mathsf{fairly}_\infty\, \psi$ says that every fair execution should satisfy ψ infinitely many times. The precise definition of a fair execution will be given below.

Informal semantics of quantified predicates. We think of a *state* of a given system as a function $\sigma : A \to V \cup \{id, type\} \to \mathbb{Z}$ that maps every agent $a \in A$ to a valuation function that, in turn, maps each local variable $var \in V$ to its current value, which we assume to be an integer. This valuation function also defines two special variables, id and $type$. The value of id is unique to each agent, while the value of $type$ will be the same for agents of the same type. Both values are constant across all states of the system.

To check whether a state σ satisfies a quantified predicate, we repeatedly apply quantifier elimination until we obtain a propositional predicate. Thus, $\forall x \in \mathrm{T} \bullet p$ reduces to $\bigwedge_{a \in A} a \in \mathrm{T} \Rightarrow p[a/x]$ and $\exists x \in \mathrm{T} \bullet p$ reduces to $\bigvee_{a \in A} a \in \mathrm{T} \wedge p[a/x]$, where predicate $a \in \mathrm{T}$ holds iff the type of agent a is T (and can be implemented as a check on $\sigma(a)(type)$), and $p[a/x]$ is the predicate p where all occurrences of agent variable x are replaced by the actual agent a. Then, we replace all expressions of $a.var$, and all predicates of the form $a = b$ (where a and b are agents), by $\sigma(a)(var)$ and $\sigma(a)(id) = \sigma(b)(id)$. The truth value of the resulting Boolean formula tells whether the original predicate holds in σ.

Informal semantics of temporal properties. We can give an informal explanation of our temporal modalities by means of CTL operators [4]. A predicate $\mathsf{always}\, \psi$ holds iff all reachable states in the system satisfy ψ. Thus, it corresponds to the CTL property $\mathsf{AG}\psi$. The fairly modality represents a form of *fair reachability* [38]. An execution is *unfair* if

it ends in an infinite cycle, such that there is at least one state along the cycle from which one could break out of the loop by performing some transition. The fairness assumption here is that such a transition is enabled infinitely often, and thus it should be fired at least once during the evolution of the system. Therefore, $\mathsf{fairly}\,\psi$ holds if every path such that ψ does not hold leads to a state from which we may reach a state that satisfies ψ. We can slightly abuse CTL's notation and express such a property as $\mathsf{AG}_{\neg\psi}\mathsf{EF}\psi$, where $\mathsf{AG}_{\neg\psi}$ represents those paths on which ψ is never satisfied. Lastly, $\mathsf{fairly}_{\infty}\,\psi$ holds if every fair execution of the system contains infinitely many states where ψ holds, and may be represented in CTL as $\mathsf{AGEF}\psi$. This modality was not supported by previous verification workflows for LAbS specifications.

3.3 Encoding state-based properties as MCL queries

We now focus on the problem of mechanically translating a temporal property espressed in the (state-based) ATLAS formalism into the (action-based) MCL language. Regardless of the temporal modality used in the formula, we will always have to reduce ψ to a propositional predicate and encode the result as a *parameterized macro*. For instance, consider the quantified predicate $\forall x \in \mathrm{T} \bullet x.var > 0$, and assume that our system of interest contains 3 agents of type T (which we will denote by $a_{1,2,3}$). Then, we can reduce such predicate to the conjunction $a_1.var > 0 \wedge a_2.var > 0 \wedge a_3.var > 0$, which we encode as the following macro:

```
macro Predicate(a1_var, a2_var, a3_var) =
  (a1_var > 0) and (a2_var > 0) and (a3_var > 0)
end_macro
```

To mechanically perform this translation we only need ψ and some information about agent types featured in ψ, which can be provided by SLiVER. In general, a formula may have to consider n agents and m variables, resulting in a macro with nm parameters. For simplicity, from now on we assume that the identifiers of these n agents are $1, 2, \ldots, n$, and that the variables relevant to ψ are named $\mathtt{v}_1, \ldots, \mathtt{v}_m$. We will denote by x_{ij} the value that the i-th agent gives to variable $\mathtt{v}_j$, and we will write $x_{11}, \ldots, x_{nm}$ to range over all nm such values.

Encoding always. An MCL query that encodes $\mathsf{always}\,\psi$ should evaluate ψ (actually, the `Predicate` macro encoding it) on every reachable state of the emulation program, and report a property violation iff it finds a state where this evaluation yields *false*. To do so, it should first capture the initial values for all mn variables that are relevant to ψ, and then use a *parameterized fixed point* formula to check that ψ is indeed an invariant of the system, while capturing those assignments that may affect its satisfaction by means of parameters. The structure of such a query is shown on Listing 2. Informally, the initial "box" operator captures the initial values of $\mathtt{v}_1, \ldots, \mathtt{v}_m$ for all agents, while also specifying that the subsequent formula should hold for all paths. Then, we pass these values to a parameterized maximal fixed point formula `Inv`, which is satisfied in a given state iff the current parameters satisfy ψ, and additionally:

Listing 2: Checking invariance of `Predicate` in MCL.

```
[{assign !1 !"v1" ?x̄11:Int} . ... . {assign !n !"vm" ?x̄nm:Int}]
nu Inv (x11:Int=x̄11, ..., xnm:Int=x̄nm) . (
  Predicate(x11,...,xnm) and
  [not {assign ...} or {assign to other variables}] Inv(x11,...,xnm) and
  [{assign !1 !"v1" ?v:Int}] Inv(v,x12,...,xnm) and ... and
  [{assign !n !"vm" ?v:Int}] Inv(x11,x12,...,v) )
```

Listing 3: An MCL macro to check that a state satisfying `Predicate` is fairly reachable from the current state.

```
macro Reach (v11,...,vnm) =
  mu R (x11:Int=v11, ..., xnm:Int=vnm) . (
  Predicate(x11,...,xnm) or
  <not {assign ...}> or <{assign to other variables}>R(x11,...,xnm) or
  <{assign !1 !"v1" ?v:Int}>R(v,...,xnm) or ... or
  <{assign !n !"vm" ?v:Int}>R(x11,...,v) )
end_macro
```

1. Every assignment that does not affect ψ leads to a state that satisfies `Inv` (without changing its parameters);
2. If an assignment changes the value of the ij-th relevant variable, the resulting state still satisfies `Inv` (after using the new value as its ij-th parameter).

Encoding fairly_∞ *and* fairly. To encode "infinitely-often" properties ($\mathsf{fairly}_\infty\,\psi$) in MCL, we similarly start by encoding ψ as a macro `Predicate`. Then, we need to write another macro `Reach` that says "A state satisfying `Predicate` is reachable from this state", by means of a *minimal* fixed point formula (Listing 3). Lastly, we check that `Reach` is an invariant of the system by using the same query shown in Listing 2, but with `Reach` instead of `Predicate`.

Checking fair reachability of ψ ($\mathsf{fairly}\,\psi$) is just a variation on the "infinitely-often" case. Informally, we want to stop exploring an execution of our system as soon as we find a state where ψ holds. To do so, we check `Predicate` within the body of the `Inv` formula, so that its satisfaction is sufficient to satisfy the whole formula.

4 Verification of sequential emulation programs

In this section, we evaluate our suggested verification workflow (Figure 1) by performing a selection of verification tasks on a selection of LAbS specifications. For each specification, we generated a corresponding LNT emulation program and verified one or more temporal properties against it.

Description of the benchmark and properties. Here, we briefly describe our systems of interest and the temporal property (or properties) that we have verified on them.

`formation` describes a system of line-forming agents. $N = 3$ robots are randomly placed on a segment of length $L = 10$. They can asynchronously inform other agents about their position, but their communication range is limited to $\delta = 2$. Property `safety` asks that the position of all robots is always in the interval $[0, L - 1]$. ($\texttt{safety} \triangleq \mathsf{always}\, \forall x \in \text{Robot} \bullet x.pos \geq 0 \wedge x.pos < L$). Property `distance` states that, eventually, the distance between any two robots should not be smaller than δ. ($\texttt{distance} \triangleq \mathsf{fairly}_\infty\, \forall x \in \text{Robot} \bullet \forall y \in \text{Robot} \bullet x \neq y \implies |a.pos - b.pos| \geq \delta$).

`flock` is the simple flocking model described in Section 2.2. In our experiments we used $N = 3$ birds, a 5×5 arena, and a communication radius $\delta = 5$.

Property `consensus` states that the system reaches a state where every bird moves in the same direction infinitely often. ($\texttt{consensus} \triangleq \mathsf{fairly}_\infty\, \forall x \in \text{Bird} \bullet \forall y \in \text{Bird} \bullet x.dir = y.dir$).

`leader` is a bully algorithm for leader election [17]. N agents must elect one of them as their leader, or coordinator. Each agent starts by advertising itself as the leader, by sending asynchronous messages containing its own identifier id. However, an agent will stop doing so if they receive a message with an id lower than their own. Property `consensus0` states that all agents eventually agree to choose the one with identifier 0 as their leader. ($\texttt{consensus0} \triangleq \mathsf{fairly}_\infty\, \forall x \in \text{Agent} \bullet x.leader = 0$).

`twophase` is a two-phase commit system [20]. A coordinator asks N workers if they agree to commit a transaction or not. If all of them agree, the coordinator commits the transaction; otherwise, it performs a rollback operation. We represent those operations by equipping the coordinator with two local variables `commit`, `rollback`, initially set to 0, and later assigning a value of 1 to either of them. After either operation has been performed, the coordinator resets both variables to 0 and the system starts over. In our specification, workers always agree to perform the commit: thus, we expect the system to perform infinitely many commits. This expectation is encoded by the `infcommits` property ($\texttt{infcommits} \triangleq \mathsf{fairly}_\infty\, \exists x \in \text{Coordinator} \bullet x.commit = 1$).

Experimental setup and results. Each verification task consists of two steps. First, we generate an LTS in BCG format from the given program. Then, we check that this LTS satisfies the requested property. This procedure is generally faster than asking *Evaluator* to model-check the LNT program on the fly, and also allows us to quantify the complexity of a program by looking at the size of its LTS. All the experiments were performed with CADP version 2021-e on the Grid'5000 testbed[6], specifically on the *Dahu* cluster in Grenoble. Each node in this cluster is equipped with two Intel Xeon Gold 6130 CPUs. We set a timeout of 3 hours and a memory limit of 16 GiB for all experiments.

Table 1 shows the results of our experimental evaluation. Columns from left to right contain, respectively, the name of the system under verification; the number of states and transitions of its corresponding LTS; the name of the property we are checking; and the resources (time and memory) that were used to check the property. Specifically, the *Time* column reports the total amount of time spent on generating and model-checking the LTS, while the *Memory* column reports the maximum amount of memory that was

[6] https://www.grid5000.fr

Table 1: Experimental results for sequential emulation programs.

System	States	Transitions	Property	Time (s)	Memory (kiB)
formation-rr	8786313	160984587	safety	1931	1723492
			distance	2147	2062872
flock-rr	58032296	121581762	consensus	3772	14122756
flock	60121704	223508430	consensus	4375	14108608
leader5	41752	1630955	consensus0	11	43456
leader6	421964	27873756	consensus0	240	224392
leader7	4438576	497568001	consensus0	3963	3240356
twophase2	19246	1124680	infcommits	17	54584
twophase3	291329	22689137	infcommits	849	145904

used during the whole task. In the first column, an `-rr` suffix denotes that we have assumed round-robin scheduling of agents, i.e., we only verified those executions where agents performed their action in circular order. In systems without the suffix, we have instead assumed free interleaving of agents. We have verified increasingly larger versions of `leader` and `twophase`. For `leader` systems, the numerical suffix denotes the number of agents; For `twophase` systems, it denotes the number of workers.

Every task resulted in a positive verdict. For `formation` and `flock`, this is consistent with previous verification results [10]. For `leader` and `twophase`, the positive verdict is consistent with the existing literature on the two algorithms [20, 17].

In the `leader` and `twophase` systems, we can observe that adding one agent makes the LTS grow by roughly one order of magnitude. This is likely the effect of two factors, namely the free interleaving of agents and the asynchronous nature of stigmergic messages. We can similarly appreciate the effect of using free interleaving instead of round-robin scheduling by comparing the `flock` and `flock-rr` experiments. The number of *states* is not particularly affected, but the *transitions* nearly double: respectively, they increase by roughly 3.6% and 83.8%. Informally, free interleaving mainly allows for alternative ways of moving between one state and the next. This has an obvious impact on the overall verification time, which increases by roughly 15%.

5 Compositional verification of parallel emulation programs

Compositional verification is a family of *divide and conquer* approaches, which exploit the parallel structure of a concurrent model to palliate state space explosion. In this work, we consider compositional (property-dependent) state space reduction (available in CADP [14, 28]) which, given a property of interest, identifies both a maximal set of actions that can be hidden and a reduction with respect to an equivalence relation, in a way that guarantees to preserve the truth-value of the property. In general, the equivalence relation is based on bisimulations (strong, divbranching[7], or sharp bisimulation [28]), which are congruences with respect to parallel composition. It is hence possible to apply action hiding and bisimulation reduction incrementally, first to individual agents and

[7] We use divbranching as shorthand for divergence-preserving branching.

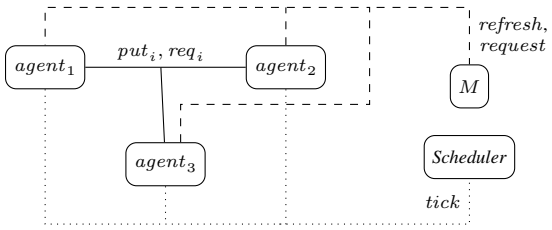

Fig. 3: Structure of a 3-agent parallel emulation program.

then to intermediate compositions. Choosing an appropriate composition order is a key of success of the approach.

To exploit the potential of compositional verification in our context, we consider in this section the feasibility of creating *parallel* emulation programs from LAbS specifications, and verify them compositionally. Figure 3 shows a simplified network diagram of a 3-agent parallel emulation program. Each agent is implemented as its own process $agent_i$, which can send stigmergic messages to the others via gates put_i and req_i. This means that a system with n agents would require $2n$ gates. Information about *timestamps* is stored in a separate process, which essentially maintains the $M_{\texttt{var}}$ matrices introduced in Section 4. Each agent can signal that they have updated a stigmergic variable by means of a $refresh$ gate, or can ask how their timestamp for a value compares to that of another agent via a $request$ gate. Lastly, a *scheduler* process ensures that agents perform their actions either in round-robin or nondeterministic fashion, without producing interleavings that are forbidden by the semantics of LAbS. Notice that a program with such an architecture can be mechanically generated from a LAbS specification by simply altering the "program template" used by the SLiVER code generator.

A potential issue that arises when using a compositional approach is that, when a process P can receive values of type T over a gate `G` (expressed in LNT as `G(?x)`, with `x` a variable of type T), the associated LTS as generated by CADP will have to consider *every* possible value of type T. If P is part of a larger system, it may be the case that only a small subset of those values are ever sent to P by other processes, and an LTS that only considered such subset would be sufficient.

To generate such an LTS from our emulation program, we had to constrain the values accepted over the communication gates depicted in Figure 3, by decorating LNT reception statements with `where` clauses. Thus, a statement `G(?x)` becomes `G(?x) where f(x)` (with `f(x)` a predicate over `x`). In this preliminary experiment we added such `where` clauses manually, but this step may be mechanized, for instance, by computing the clauses via an interval analysis.

5.1 Experimental evaluation

As a preliminary evaluation of this approach, we considered the `flock-rr` system from Section 4, and reduced it to a parallel emulation program whose architecture matches that of Figure 3. Then, we asked CADP to generate the individual LTSs of each process, reduce them with respect to divbranching bisimulation, and finally compose them. We

Table 2: Compositional verification of `flock-rr`.

Process	States	Transitions	Time (s)	Memory (kiB)
M	13	234	3	34360
Scheduler	6	12	2	34212
$Agent_1$	25637	1989572	537	3102904
$Agent_2$	25637	1989572	537	3102908
$Agent_3$	25637	1989572	538	3100408
Main	28800	74906	73	70828
Main $\models$ `consensus`	–	–	2	43428
Total time, max memory			1692	3102908

used the same experimental setup described in Section 4. Finally, we asked to evaluate the `consensus` property on the resulting LTS.

Details about this experiment are shown in Table 2. For each row except the last two, the first three columns contain the name of a process in the emulation program and the size of its corresponding LTS (in terms of its states and transitions). Then, the *Time* column contains the total time spent by CADP to generate and reduce the LTS with respect to divbranching bisimulation. The *Memory* column, instead, contains the maximum amount of memory required by these two operations. Notice that *Main* is the process encoding the whole system. The last rows, instead, show the resources used by CADP to evaluate whether the LTS of `Main` satisfies `consensus`, and an overall account of the resources (total time and maximum amount of physical memory used) needed for the whole experiment. CADP provided a positive verdict after 1692 seconds (roughly half an hour), by spending roughly 3 GiB of memory. Most of the time was spent in the generation and reduction of the LTSs for the three agents. These numbers appear to improve over the sequential approach, which by comparison requires 3772 seconds and around 14 GiB of memory (Table 1).

6 Related work

Several other works investigate the use of on-the-fly model checking to verify multi-agent systems [2, 30]. Recently, formal verification of *open* systems (where countably many agents may join or leave during its evolution) has been addressed in [25], which also uses properties that are quantified over the agents. So far, SLiVER does not support open systems, and the problem appears to be decidable only for a specific class of systems. It would be interesting to investigate whether LAbS systems belong to such a class.

Verifying higher-level languages by means of mechanized translations to process algebras or similar kinds of formal models has been explored by several other works, allowing to verify diverse classes of systems, including web choreographies [13], agents with attribute-based communication [8], information systems with trace-dependent attributes [40], or component-based ensembles [21]. Compositional verification has also been shown to be effective in dealing with component-based [1] and asynchronous concurrent systems [14]. An alternative approach relies on encodings into some general-purpose programming language, such as C: the resulting emulation programs can then

be verified by any off-the-shelf analysis tool supporting said language [7]. SLiVER's modular architecture allows to support both approaches, exemplified respectively by LNT and C, and to add new modules implementing additional translations [10].

Our encoding of temporal properties bears some resemblance to the *specification pattern system* of [12]. Similarly to us, the authors of these patterns contend that higher-level logical formalisms are needed to facilitate the adoption of formal verification. While their approach focuses on capturing temporal relationships between states or actions (which are left unspecified), we focused on specifying state properties by means of quantified predicate and then implemented some of these patterns on top of a data-aware, action-based logic. Extending our approach to other patterns may be a worthwhile effort.

7 Conclusions

We have introduced an automated workflow for the verification of a variety of temporal properties on stigmergic systems, which are encoded as sequential emulation programs in the LNT language. This workflow reuses an expressive property language and state-of-the-art procedures for on-the-fly model checking. We have demonstrated our approach by carrying out a selection of verification tasks. Then, we have presented some preliminary results about the feasibility of using parallel emulation programs, combined with compositional verification techniques. This direction of research appears to be promising for large systems and should be investigated as part of our future work. Another possible approach would consist in generating the LTS of a sequential emulation program in a distributed fashion, using other tools provided by CADP.[8]

We should stress that our proposed property language and verification workflow can be applied to other types of concurrent systems, as long as they can be reduced to emulation programs. This aspect should also be analysed in further depth, by finding other formalisms and case studies. Lastly, our property language could be improved in multiple ways, e.g., by supporting predicates about shared memory; introducing the capability to count [29] the agents that satisfy some predicate; or extending the range of temporal properties supported by our procedure.

Acknowledgments

The authors wish to thank Radu Mateescu for his precious insights into MCL. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations.

References

1. Bensalem, S., Bozga, M., Sifakis, J., Nguyen, T.H.: Compositional verification for component-based systems and application. In: ATVA. LNCS, vol. 5311. Springer (2008). https://doi.org/10.1007/978-3-540-88387-6_7

[8] E.g., the *Distributor* tool: see https://cadp.inria.fr/man/distributor.html

2. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifying multi-agent programs by model checking. Autonomous Agents and Multi-Agent Systems **12**(2) (2006). https://doi.org/10.1007/s10458-006-5955-7
3. Brooks, R.A., Flynn, A.M.: Fast, cheap and out of control: A robot invasion of the solar system. Journal of the British Interplanetary Society **42** (1989)
4. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logics of Programs Workshop. LNCS, vol. 131. Springer (1981). https://doi.org/10.1007/BFb0025774
5. Crowston, K., Rezgui, A.: Effects of stigmergic and explicit coordination on wikipedia article quality. In: HICSS. ScholarSpace (2020)
6. De Nicola, R., Di Stefano, L., Inverso, O.: Multi-agent systems with virtual stigmergy. Science of Computer Programming **187** (2020). https://doi.org/10.1016/j.scico.2019.102345
7. De Nicola, R., Duong, T., Inverso, O.: Verifying AbC specifications via emulation. In: ISoLA. LNCS, vol. 12477. Springer (2020). https://doi.org/10.1007/978-3-030-61470-6_16
8. De Nicola, R., Duong, T., Inverso, O., Mazzanti, F.: A systematic approach to programming and verifying attribute-based communication systems. In: From software engineering to formal methods and tools, and back. LNCS, vol. 11865. Springer (2019). https://doi.org/10.1007/978-3-030-30985-5_22
9. De Nicola, R., Vaandrager, F.W.: Action versus state based logics for transition systems. In: LITP 1990: Semantics of Systems of Concurrent Processes. LNCS, vol. 469. Springer (1990). https://doi.org/10.1007/3-540-53479-2_17
10. Di Stefano, L.: Modelling and Verification of Multi-Agent Systems via Sequential Emulation. PhD Thesis, Gran Sasso Science Institute (2020)
11. Di Stefano, L., Lang, F., Serwe, W.: Combining SLiVER with CADP to analyze multi-agent systems. In: COORDINATION. LNCS, vol. 12134. Springer (2020). https://doi.org/10.1007/978-3-030-50029-0_23
12. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: ICSE. ACM (1999). https://doi.org/10.1145/302405.302672
13. Foster, H., Uchitel, S., Magee, J., Kramer, J.: WS-Engineer: A model-based approach to engineering web service compositions and choreography. In: Test and Analysis of Web Services. Springer (2007)
14. Garavel, H., Lang, F., Mateescu, R.: Compositional verification of asynchronous concurrent systems using CADP. Acta Informatica **52**(4) (2015)
15. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: A toolbox for the construction and analysis of distributed processes. Software Tools for Technology Transfer **15**(2) (2013). https://doi.org/10.1007/s10009-012-0244-z
16. Garavel, H., Lang, F., Serwe, W.: From LOTOS to LNT. In: ModelEd, TestEd, TrustEd. LNCS, vol. 10500. Springer (2017). https://doi.org/10.1007/978-3-319-68270-9_1
17. Garcia-Molina, H.: Elections in a distributed computing system. IEEE Transactions on Computers **31**(1) (1982). https://doi.org/10.1109/TC.1982.1675885
18. Geller, A., Moss, S.: Growing qawm: An evidence-driven declarative model of Afghan power structures. Advances in Complex Systems **11**(2) (2008). https://doi.org/10.1142/S0219525908001659
19. Grassé, P.P.: La reconstruction du nid et les coordinations interindividuelles chez Bellicositermes natalensis et Cubitermes sp. la théorie de la stigmergie: Essai d'interprétation du comportement des termites constructeurs. Insectes Sociaux **6**(1) (1959). https://doi.org/10.1007/BF02223791
20. Gray, J.: Notes on data base operating systems. In: Operating Systems, An Advanced Course. LNCS, vol. 60. Springer (1978). https://doi.org/10.1007/3-540-08755-9_9

21. Hennicker, R., Klarl, A., Wirsing, M.: Model-checking Helena ensembles with Spin. In: Logic, Rewriting, and Concurrency. LNCS, vol. 9200. Springer (2015). https://doi.org/10.1007/978-3-319-23165-5_16
22. Heylighen, F.: Stigmergy as a universal coordination mechanism I: Definition and components. Cognitive Systems Research **38** (2016). https://doi.org/10.1016/j.cogsys.2015.12.002
23. Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall (1985)
24. ISO/IEC: Information processing systems - Open systems interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour. International Standard 8807, ISO (1989)
25. Kouvaros, P., Lomuscio, A., Pirovano, E., Punchihewa, H.: Formal verification of open multi-agent systems. In: AAMAS. IFAAMAS (2019)
26. Kozen, D.: Results on the propositional mu-calculus. Theoretical Computer Science **27** (1983)
27. Kuylen, E., Liesenborgs, J., Broeckhove, J., Hens, N.: Using individual-based models to look beyond the horizon: The changing effects of household-based clustering of susceptibility to measles in the next 20 years. In: ICCS. LNCS, vol. 12137. Springer (2020). https://doi.org/10.1007/978-3-030-50371-0_28
28. Lang, F., Mateescu, R., Mazzanti, F.: Sharp congruences adequate with temporal logics combining weak and strong modalities. In: TACAS. LNCS, vol. 12079. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_4
29. Libkin, L.: Logics with counting and local properties. ACM Transactions on Computational Logic **1**(1) (2000). https://doi.org/10.1145/343369.343376
30. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: An open-source model checker for the verification of multi-agent systems. Software Tools for Technology Transfer **19**(1) (2017). https://doi.org/10.1007/s10009-015-0378-x
31. Mateescu, R., Thivolle, D.: A model checking language for concurrent value-passing systems. In: FM. LNCS, vol. 5014. Springer (2008). https://doi.org/10.1007/978-3-540-68237-0_12
32. Olner, D., Evans, A.J., Heppenstall, A.J.: An agent model of urban economics: Digging into emergence. Computers, Environment and Urban Systems **54** (2015). https://doi.org/10.1016/j.compenvurbsys.2014.12.003
33. Panait, L.A., Luke, S.: Ant foraging revisited. In: ALIFE. MIT Press (2004). https://doi.org/10.7551/mitpress/1429.003.0096
34. Parunak, H.V.D.: "Go to the ant": Engineering principles from natural multi-agent systems. Annals of Operations Research **75**(0) (1997). https://doi.org/10.1023/A:1018980001403
35. Pinciroli, C., Beltrame, G.: Buzz: An extensible programming language for heterogeneous swarm robotics. In: IROS. IEEE (2016). https://doi.org/10.1109/IROS.2016.7759558
36. Pnueli, A.: The temporal logic of programs. In: FOCS. IEEE (1977). https://doi.org/10.1109/SFCS.1977.32
37. Qadeer, S., Wu, D.: KISS: Keep it simple and sequential. In: PLDI. ACM (2004). https://doi.org/10.1145/996841.996845
38. Queille, J.P., Sifakis, J.: Fairness and related properties in transition systems - A temporal logic to deal with fairness. Acta Informatica **19** (1983). https://doi.org/10.1007/BF00265555
39. Reynolds, C.W.: Flocks, herds and schools: A distributed behavioral model. In: SIGGRAPH. ACM (1987). https://doi.org/10.1145/37402.37406
40. Vekris, D., Lang, F., Dima, C., Mateescu, R.: Verification of EB3 specifications using CADP. Formal Aspects of Computing **28**(1) (2016). https://doi.org/10.1007/s00165-016-0362-6
41. Wirsing, M., Banâtre, J.P., Hölzl, M., Rauschmayer, A. (eds.): Software-Intensive Systems and New Computing Paradigms - Challenges and Visions, LNCS, vol. 5380. Springer (2008)