Compositional Verification of Stigmergic Collective Systems*

Luca Di Stefano[1,2] (✉) [0000−0003−1922−3151] and Frédéric Lang[1] [0000−0002−5221−3353]

[1] Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, Grenoble, France
[2] University of Gothenburg, Gothenburg, Sweden
luca.di.stefano@gu.se

Abstract. Collective adaptive systems may be broadly defined as ensembles of autonomous agents, whose interaction may lead to the emergence of global features and patterns. Formal verification may provide strong guarantees about the emergence of these features, but may suffer from scalability issues caused by state space explosion. Compositional verification techniques, whereby the state space of a system is generated by combining (an abstraction of) those of its components, have shown to be a promising countermeasure to the state space explosion problem. Therefore, in this work we apply these techniques to the problem of verifying collective adaptive systems with stigmergic interaction. Specifically, we automatically encode these systems into networks of LNT processes, apply a static value analysis to prune the state space of individual agents, and then reuse compositional verification procedures provided by the CADP toolbox. We demonstrate the effectiveness of our approach by verifying a collection of representative systems.

1 Introduction

In a collective adaptive system, autonomous individuals or *agents* interact with each other according to simple local rules, which may lead to the emergence of global features and patterns despite the lack of centralized coordination [32]. Using these systems as a modelling framework to study complex phenomena, such as the spread of diseases through a social network [14], the role of spatial constraints in an economy [40], or the evolution of an ecosystem [30], appears to be a trending research methodology. Depending on the field of application, the resulting models are variously referred to as individual- or agent-based models, in silico cell models, or multi-agent systems, but they all share the essential traits of collective adaptive systems.

This increasing popularity owes to the fact that, under such a framework, one can easily specify heterogeneous agents with stateful, nonlinear or discontinuous behavioural rules [3]; additionally, one can easily refine these specifications

* Work partially funded by ERC consolidator grant no. 772459 *D-SynMA* (Distributed Synthesis: from Single to Multiple Agents).

if they turn up to be incomplete or incorrect, e.g., if undesired behaviour is observed when simulating the model. However, simulations are unsuitable to achieve strong confidence in the *correctness* of a model of this kind. In fact, even small collective adaptive systems may evolve in a multitude of different ways, which increases exponentially in the number of agents and the complexity of their behaviour. This *state space explosion* problem means that simulations and testing may only cover a small portion of all feasible evolutions that such models can exhibit: therefore, attempts at uncovering unexpected or problematic behaviour by these means are likely to fail.

Formal verification techniques, in principle, may provide the correctness guarantees that are out of the reach of simulation-based analysis, but they also suffer from complexity issues related to the state space explosion problem. *Compositional* techniques, essentially based on a divide-and-conquer strategy to break down the analysis of large systems, appear to be a general, effective approach to mitigate the state space explosion problem [18]. To support this claim, in this work we introduce a fully-automated workflow to perform compositional verification of *stigmergic* collective systems specified in a high-level language called LAbS [7]. In these systems, agents do not interact directly with each other, but rather share information by manipulating a shared medium called a virtual stigmergy [41]. The concept of stigmergies originates from biology, where it has been used to explain the collective behaviour of social insects such as ants, termites, and bees [47], but appears well-suited to describe a much wider range of phenomena, including the creation and curation of content on the Wikipedia collaborative encyclopedia [6], or the development of open-source software [45]. The indirect and asynchronous nature of this interaction mechanism induces vast state spaces even in modestly-sized systems, making their verification challenging [10, 12]. However, by combining compositional state space generation with a static value analysis that allows us to prune the state space of individual agents, we are able to verify a collection of example systems with significant gains over a non-compositional model checking procedure.

The rest of this paper is organized as follows. Section 2 outlines the specification language that we intend to verify, and provides the necessary background about verification tools, techniques, and abstract domains that are relevant to our work; it also includes an example of a stigmergic system that we will recurringly use to illustrate our approach. Section 3 discusses the encoding of LAbS agents into LNT processes, and how these processes are composed into a parallel program that emulates the agents' evolution and interactions. Section 4 introduces a static value analysis that helps us prevent state-space explosion as we generate the state space of individual agents. Section 5 describes the implementation of our approach and its experimental evaluation over a collection of LAbS examples. Section 6 discusses related work. Lastly, Section 7 contains our conclusions and potential directions for future work.

2 Background

In this section, we provide an overview of concepts that will be referred to in the rest of the paper. First, we will introduce the LAbS language for stigmergic collective systems, as well as a running example to demonstrate the peculiarities of these systems. Then, we will describe the *Intervals* and *Powerset of Intervals* abstract domains; the CADP analysis platform and the LNT process calculus; and some notions related to compositional state space generation.

Stigmergic collective systems and LAbS. The LAbS language [7] is a high-level formalism to specify stigmergic collective systems. Agents in a LAbS system cannot explicitly exchange messages with each other: rather, they assign values to specific local variables, which we call *stigmergic variables*. After an assignment to one of these variables, an agent will asynchronously diffuse the assigned value among its neighbours by sending out a *put-message*. All assignments are timestamped, and the receivers of a *put*-message with a newer value will update their own local copy of the variable to that value. Upon receiving a more recent value, agents also help propagate it by forwarding the *put*-message to their own neighbours. Similarly, after accessing the value of a stigmergic variable, an agent will asynchronously check whether someone among its neighbours has a newer value for that variable, by sending a *qry-message*. Neighbours react to the query by sending out a new *put*-message containing either their own value for the variable or the received one, depending on which is newer. These simple mechanisms allow local information to spread from one agent to the others, and new information to replace older data.

In LAbS, the definition of an agent's neighbourhood is not fixed: in fact, the language allows to equip stigmergic variables with *link predicates* to customize this concept. A link predicate is a Boolean function over the state of a sender and a (potential) receiver. Whenever an agent sends a message regarding a given variable x, this message will be received by all the agents that (together with the sender) satisfy x's link predicate. These agents are the *neighbours* of the sender with respect to x's predicate. This feature makes LAbS quite flexible, as it allows modelling different capabilities among the agents, such as their communication range, or having privileged access to some variables, and so on; it also induces neighbourhoods that may vary as the system evolves.

Running example: stigmergic bully election. A bully election is a simple protocol to elect a leader in a distributed system [24]. The protocol assumes that each node in the system has a fixed, unique numeric identifier (id) in the range $0..N-1$, where N is the number of nodes. Intuitively, each node in the system initially considers itself the leader, and advertises this by broadcasting its id to the rest of the system. However, a node that receives a message with an id i lower than their own will instead regard node i as its new leader. When this happens, the node also stops advertising itself, but keeps changing the leader every time it receives a message with a lower id. This protocol eventually makes all nodes agree that the one with the lowest id is the leader.

Replicating such a protocol in a stigmergic system is not immediate, as agents have no primitive to explicitly exchange messages with one another. However, Listing 1: A sketch of a stigmergic election system in LAbS.

```
system {
  spawn = Node: N
}
stigmergy Election {
  link = true
  leader: N
}
agent Node {
  stigmergies = Election
  Behavior =
    leader > id ->
      leader <~ id;
      Behavior
}
```

we can let them manipulate a stigmergic variable `leader` until they reach a consensus on its value. Essentially, each node only needs to check whether `leader` is currently higher than its own id. If it is, it means that the node still has a chance of becoming the leader: so, the node assigns its own id to `leader`. As the link predicate for `leader`, we use the one that is always satisfied: this induces a broadcast communication model, i.e., every time a node assigns a value to `leader`, this will be (asynchronously) diffused to every other node in the system. Every time a value j gets diffused in this way, it immediately puts all nodes with id $i > j$ out of the race. We can speculate that, eventually, every node gets out of the election except the one with the lowest id, and all nodes have that id assigned to `leader`.[3]

Listing 1 shows how such a protocol can be expressed in LAbS. The code specifies that the system is composed of N agents of type `Node`; declares a stigmergy (i.e., a collection of stigmergic variables with the same link predicate) `Election`, equipped with the always-satisfied link predicate `true` and containing a single variable `leader`, which is initialised to N; and finally specifies the `Node` type. Namely, each `Node` participates in the `Election` stigmergy, meaning that it will have a local copy of the `leader` variable. Its behaviour is a guarded recursive process: a guard blocks the agent until the value of `leader` is greater than its identifier `id`. When this is the case, the agent assigns `id` to `leader` and then starts over. (The `<~` operator denotes an assignment to a stigmergic variable).

Intervals and their powersets. For our purpose, an *interval* is either the empty interval $\perp$ or a pair $[a, b]$, with $a \in \mathbb{R} \cup \{-\infty\}$, $b \in \mathbb{R} \cup \{\infty\}$, and $a \leq b$; we do not need open-bounded intervals, which are excluded from our definition. Intuitively, an interval-based value analysis [5] starts from an initial *abstract state* s_0 of the program under analysis, i.e., a mapping from program variables to intervals. The precise way in which s_0 is computed depends on the semantics of the language; generally, variables that are initialized to a constant κ are mapped to the singleton interval $[\kappa, \kappa]$, while nondeterministic variables, e.g., those representing inputs to the program, are mapped to $[-\infty, \infty]$, meaning that they may initially assume any value. The analysis then explores the abstract states that are reachable from s_0 by performing an abstract interpretation of the program. As an example, Fig. 1a shows a function $[\![e]\!](s)$, defined by induction on the structure of a very simple expression language, to evaluate an expression e

[3] In Section 5, we will prove by model-checking that this speculation is correct.

$$
\begin{aligned}
[\![x]\!](s) &= s(x)\\
[\![\kappa]\!](s) &= [\kappa, \kappa], \quad \kappa \in \mathbb{Z}\\
[\![e_1 \circ e_2]\!](s) &= [\![e_1]\!](s) \circ^\sharp [\![e_2]\!](s)
\end{aligned}
$$

(a) Abstract evaluation of expressions.

$$
\begin{aligned}
[a, b] +^\sharp [c, d] &= [a + c, b + d]\\
[a, b] -^\sharp [c, d] &= [a - d, b - c]\\
[a, b] \times^\sharp [c, d] &= [\min(ac, bc, ad, bd),\\
&\qquad \max(ac, bc, ad, bd)]
\end{aligned}
$$

(b) Examples of abstract operators.

$$
[\![x \leftarrow e]\!](s) = s\,[x \mapsto [\![e]\!](s)]
$$

(c) Abstract evaluation of assignments.

$$
\begin{aligned}
[a, b] \sqcup [c, d] &= [\min(a, c), \max(b, d)]\\
[a, b] \sqcup \bot &= [a, b]\\
\bot \sqcup [a, b] &= [a, b]\\
\bot \sqcup \bot &= \bot
\end{aligned}
$$

(d) The join operator.

$$
\begin{aligned}
[a, b] \cap [c, d] &= \begin{cases} \bot & \text{iff } a > d \text{ or } b < c\\ [\max(a, c), \min(b, d)] & \text{otherwise} \end{cases}\\
\bot \cap [a, b] = [a, b] \cap \bot &= \bot\\
\bot \cap \bot &= \bot
\end{aligned}
$$

(e) The intersection operator.

Fig. 1: Definitions related to the interval abstraction.

on an abstract state s. An integer constant κ evaluates to the single-element interval $[\kappa, \kappa]$. A reference to a variable x evaluates to the interval $s(x)$. For a binary operation $e_1 \circ e_2$, one evaluates e_1, e_2 to obtain two intervals, and then composes the intervals according to an abstract version $\circ^\sharp$ of the operation. As a minimal example, in Fig. 1b we show the usual definition of abstract addition, subtraction, and multiplications over integers. Lastly, we can slightly abuse our notation and write $[\![x \leftarrow e]\!](s)$ to denote the abstract evaluation of an *assignment* statement on state s, where variable x will receive the result of expression e. This operation returns a new abstract state that is identical to s, except that x maps to the abstract evaluation of e on s (Fig. 1c).

Interval-based reasoning provides a rather coarse approximation of the concrete set of values that a variable may assume. For instance, interval $[0, 10]$ is a sound abstraction of the concrete set $\{0, 3, 10\}$, but includes several elements that do not belong to the set. To enjoy a tighter approximation while still relying on the (computationally cheap) domain of intervals, we consider the *powerset of intervals* [5,16] domain, commonly denoted by $P(I)$. Intuitively, an element in $P(I)$ is a set of disjoint intervals; we say that two intervals i, j are disjoint when their intersection $i \cap j$, as defined in Fig. 1e, is the empty interval. Given any set of intervals S, possibly including non-disjoint intervals, we can find its *normal form* $n(S) \in P(I)$, defined by Eq. 3 below, which replaces subsets of continuous (Eq. 1) intervals disjoint (Eq. 2) from the rest by their *join*, where the join of two intervals $i \sqcup j$ is the smallest interval that entirely contains both i and j, and is computed as shown in Fig. 1d. Lastly, we can use $P(I)$ as a domain for abstract interpretation of expressions by lifting the abstract operators already defined over intervals. Namely, if S_1, S_2 are elements of $P(I)$ and $\circ$ is a binary operator, one can soundly define $S_1 \circ^\sharp S_2$ as in Eq. 4 below, by evaluating the operation pairwise over the elements of S_1 and S_2, and then finding the normal form of the resulting set.

$$\begin{aligned}
\mathit{cont}(S) &= (\forall x \in \bigsqcup S)\ (\exists i \in S)\ x \in i && (1)\\
\mathit{disj}(S, S') &= (\forall i \in S, j \in S')\ i \cap j = \bot && (2)\\
n(S) &= \{\bigsqcup S' \mid S' \subseteq S \wedge \mathit{cont}(S') \wedge \mathit{disj}(S', S \setminus S')\} && (3)\\
S_1 \circ^\sharp S_2 &= n(\{i_1 \circ^\sharp i_2 \mid i_1 \in S_1, i_2 \in S_2\}) && (4)
\end{aligned}$$

Compositional state space generation. The systems we are interested in analysing may be imagined as trees of parallel processes, branching out from a root parallel composition and whose leaves correspond to sequential processes. To generate the state space of these systems, we may apply a divide-and-conquer approach where we first generate the state spaces of each leaf, and then compose them together [49]. What makes this approach appealing is that, under appropriate assumptions and depending on our goals (e.g., on which properties we want to verify on the system), we can also perform hiding and minimization steps on the components, facilitating their composition.

Several compositional *strategies* have been put forward to outline the order in which these steps are carried out [18]. In this work we exploit one such strategy, namely *root leaf reduction.* Under this strategy, first, hiding operators are propagated as far down the tree as allowed; then, the state spaces of the leaves are generated and minimized modulo some equivalence relation R; lastly, the state spaces are composed together according to the structure of the tree and the resulting state space is further minimized modulo R.

The CADP toolbox and LNT. CADP [19] is a software toolbox for the analysis of asynchronous concurrent systems. It provides a wide range of tools for simulation, test generation, verification, performance evaluation, etc., and accepts system descriptions in several languages whose semantics can be expressed in the form of an LTS (labelled transition system). CADP provides efficient model-checking procedures for a data-aware extension of the modal μ-calculus called MCL [37], and allows declaring complex verification tasks by means of an ad-hoc scripting language called SVL [17]. SVL natively supports several compositional strategies, including root leaf reduction.

In this work we will use LNT [21] to describe networks of *processes* that interact by means of *offers.* We will use the following subset of LNT *communication actions*: $G(x_1, \ldots, x_n)$ denotes an *output offer*, i.e., an action by which a process is willing to output the values $x_1, \ldots, x_n$ through a gate G; on the other hand, $G(?x_1, \ldots, ?x_n)$ denotes an *input offer*, i.e., an action where a process is willing to receive any n values from gate G and bind them to variables $x_1, \ldots, x_n$. Finally, $G(?x_1, \ldots, ?x_n)$ ***where*** $p(x_1, \ldots, x_n)$ is also an input offer, but the process is only willing to receive those values that satisfy a given Boolean predicate p. The semantics of a process is an LTS in which each label corresponds to an offer.

To make LNT processes synchronize on offers, we have to compose them in parallel and specify a *synchronization set* for the composition. Specifically, the syntax ***par*** $G_1, \ldots, G_m$ ***in*** $P_1 \| \cdots \| P_n$ ***end par*** denotes a parallel composition of n processes where an offer on any of the gates $G_1, \ldots, G_m$ may only take place if *all* processes are willing to perform it simultaneously; all other offers may happen freely. *Partial* synchronization sets may also be defined by using the syntax ***par*** $\Gamma_1 \rightarrow P_1 \| \cdots \| \Gamma_n \rightarrow P_n$ ***end par***, where $\Gamma_1, \ldots, \Gamma_n$ are sets of gates. In this case, synchronization over a gate G is only required among those processes that have G in their set of gates [23].

Notice that an input offer is semantically equivalent to an output offer of a nondeterministic value. For instance, if x is a Boolean variable, $G(?x)$ is the same as a choice between $G(\texttt{true})$ and $G(\texttt{false})$, followed by an assignment of the offered value to x. Thus, even though the LNT syntax might suggest asymmetrical interactions (*à la* CCS [39]), its synchronization semantics makes no difference between senders and receivers, and naturally supports multi-party rendezvous.

3 Parallel emulation programs

Given a specification $\mathbb{S}$ of a collective system, an emulation program $\mathbb{P}$ for it is a program, written in some target programming language, that may reproduce all feasible executions of $\mathbb{S}$ without introducing spurious ones. Thus, one may check whether a given temporal property holds in $\mathbb{S}$ by verifying an adequate encoding of the property against $\mathbb{P}$. A *sequential* emulation program replaces agent concurrency with nondeterminism, essentially applying sequentialization [43] to $\mathbb{S}$. Sequential emulation programs may be written in any imperative language and enable verification of collective systems by means of several analysis techniques for sequential programs [9]. In this section, instead, we show how we can exploit LNT's native constructs for parallelism to construct a *parallel* emulation program, where each agent is encoded into a separate process and communication is described via process synchronization. This encoding preserves the structure of the original system and enables compositional analysis.

The structure of an `Agent` LNT process is summarized in Fig. 2. We assume that each agent has a unique identifier, denoted by id. First, the agent performs an initialization ($init$), where its (potentially nondeterministic) internal state is set up according to the specifications. This state is entirely contained into two arrays L_{id} (the *local stigmergy* of id) and I_{id} (the *interface* of id), which respectively contain the values of stigmergic variables and of other internal variables.

After the initialization, the agent enters a loop during which it may repeatedly choose between six alternative behaviours. Namely, it may perform an individual action itself ($step$), signalled by an offer `tick(`id`)`, or let another agent do

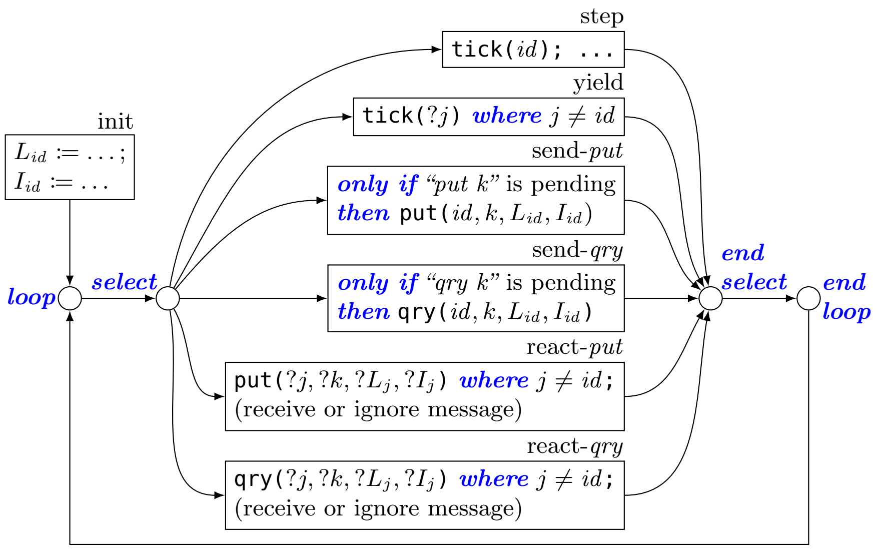
step
tick(id); ...
yield
tick(?j) where j ≠ id
init
L_{id} := ...;
I_{id} := ...
send-put
only if "put k" is pending
then put(id, k, L_{id}, I_{id})
send-qry
only if "qry k" is pending
then qry(id, k, L_{id}, I_{id})
loop
select
end
select
end
loop
react-put
put(?j, ?k, ?L_j, ?I_j) where j ≠ id;
(receive or ignore message)
react-qry
qry(?j, ?k, ?L_j, ?I_j) where j ≠ id;
(receive or ignore message)

Fig. 2: Structure of an `Agent` process with identifier id.

the same (*yield*), signalled by an input offer `tick`$(?j)$ for some $j \neq id$. It may also send one of its pending stigmergic messages, if any (*send-put*, *send-qry*); or it may react to a message sent by another agent (*react-put*, *react-qry*). By *reacting* to a message, we mean that the agent either accepts or ignores it, based on the semantic rules outlined in Section 2. To take this decision, we need the sender to evaluate the link predicate for the variable k within the message. To do so, it needs to know the state of the sender j, which is why L_j and I_j are part of the input offers (and L_{id}, I_{id} are part of the corresponding output offers).

Note that, while the definition of most blocks depends only on the variables of the input LAbS agent, the *step* block is the only one whose definition depends on its behaviour: generating the *step* block essentially amounts to transforming the LAbS behaviour into LNT fragments, by means of an automated procedure [9]. All other blocks simply implement the semantic rules of LAbS and are the same for every specification.

To model a LAbS system of n agents, we construct an LNT parallel emulation program following the structure shown in Listing 2. Specifically, we instantiate n `Agent` processes, each with a unique identifier from 0 to $n - 1$ (lines 3–7). These processes are composed in parallel, with a global synchronization set containing gates `tick`, `put`, and `qry`, so that all agents must synchronize in order to perform an input/output offer over these gates. This ensures that individual actions never overlap with each other nor with message-passing steps, and also that messages are always visible to all agents (which then decide whether to accept or ignore them). These restrictions are necessary to avoid spurious executions, i.e., computations of the programs that do not correspond to a trace in the original specifications. The program also features a `Timestamps` process

Listing 2: A parallel emulation program for a system of n agents.

```
 1 process Main [...] is
 2 par refresh, request in
 3    par tick, put, qry in
 4       Agent [...] (0)
 5    || ···
 6    || Agent [...] (n − 1)
 7    end par
 8 || Timestamps [refresh, request]
 9 end par
10 end process
```

Listing 3: An emulation program with round-robin scheduling.

```
 1 process Sched [tick] is loop
 2   tick(0); tick(1); ... ;
 3   tick(n − 1)
 4 end loop end process
 5
 6 process Main [...] is
 7 par
 8    tick, refresh, request ->
 9      par tick, put, qry in
10         Agent [...] (0)
11      || ···
12      || Agent [...] (n − 1)
13      end par
14 || refresh, request ->
15      Timestamps [refresh, request]
16 || tick -> Sched [tick]
17 end par
```

Listing 4: Sketch of the LNT translation of the `Node` process from Listing 1.

```
 1 process Agent [...] (id: Nat) is
 2   -- init
 3   L[0] := N; -- leader
 4   pending := ∅;
 5   ...;
 6   loop
 7   select
 8     -- step
 9     tick(id);
10     if L[0] > id then
11       L[0] := id;
12       updateTimestamp(0);
13       addPendingMsg("put", 0)
14     else loop spurious end loop
15     end if
16   []
17     -- yield
18     tick(?j) where j <> id
19   []
20     -- other behaviours
21     -- (react-put, react-qry,
22     -- send-put, send-qry)
23   end select
24   end loop
25 end process
```

(line 8) that tracks timestamping information about stigmergic variables. This information, in turn, determines how agents will react when they receive a stigmergic message. Each agent may independently contact this process through two gates: namely, they can either `refresh` their timestamp for a variable (which they do after assigning a new value to it), or `request` a comparison between their timestamp for a variable and the one of another agent (which they do when they process an incoming message).

Round-robin scheduling. In the emulation program shown so far, agents can freely interleave their actions. In some cases, however, it makes sense to only consider *round-robin* executions, i.e., those where agents perform one *step* at a time, in a fixed sequence given by their identifiers. (Message passing actions can still happen at any time). We may enforce this restriction by adding to our program a *scheduler* process that constrains `tick` offers, so that the agent with identifier 0 is forced to act first, followed by the one with id 1, and so on (Listing 3).

Correctness of the encoding. Our argument for the correctness of the LAbS-to-LNT encoding is essentially the same as the one we put forward for generic emulation programs [9], the main difference being that the sequentializing scheduler in that work is now replaced by multi-party synchronizations where the agents decide who should act next. First, we assume there exists a translation from each LAbS action α (e.g., as an assignment) to an LNT fragment that respects the operational semantics of α. By forcing synchronizations over the `tick` gate, we prevent agents from overlapping their actions with other actions or with message exchanges. Therefore, for every sequence of actions allowed by the original specification $\mathbb{S}$, the emulation program $\mathbb{P}$ features some execution in which the corresponding LNT code fragments are invoked in the correct order. Vice versa, if $\mathbb{P}$ allows a sequence of code fragments to be executed, then there is a feasible execution of $\mathbb{S}$ where agents perform the corresponding LAbS actions in the same order.

Running example. Listing 4 contains a simplified sketch of how the `Node` agent from Listing 1 would get translated into LNT. First, the agent's stigmergic variable `leader` (stored in `L[0]`) and its set of pending messages are initialized to N and to the empty set, respectively. We omit the rest of the initialization. Then, the agent enters the loop we graphically depicted in Figure 2. For sake of brevity, we only include a simplified version of the *step* and *yield* behaviours. We encode the guarded action of Listing 1 by means of an `if` statement. If the guard is not satisfied, we send the agent to a sink state where it repeatedly performs a special `spurious` action (line 14). We will use this action to detect and ignore invalid traces during our analysis. If the guard holds, the agent can proceed with the stigmergic assignments, which consists of three steps: updating the value, setting its timestamp to the current instant, and adding a *put*-message to the list of pending messages (lines 11–13).

Notice that, for an agent with a more elaborate behaviour, the `step` block would be a nondeterministic choice over the feasible actions that the agent can perform at the present time. This also requires keeping track of the agent's execution point: for instance, an agent whose behaviour is a sequence of two actions $a; b$ must necessarily perform a before it is able to perform b. To maintain track of this information, we constrain the agent's choice of actions by a dedicated variable that acts as a program counter, and is updated after every action [9].

4 Value analysis of LAbS specifications

As seen in Fig. 2, the exchange of stigmergic messages requires the sender to offer its local stigmergy and interface to all other agents, so that they can perform a corresponding input offer to receive this information and evaluate whether they should receive or ignore the message. These input offers make generating the state space of individual agents problematic. In fact, each agent may expect to receive *any* possible L_j and I_j over the `put` and `qry` gates, meaning that its transition system has to enumerate all potential offers. This easily makes the agent's state space explode, even when we assume that variables range over relatively modest intervals, such as the 8-bit representation range $(-128, \ldots, 127)$.

We work around this issue by observing that, in typical systems, agents will only ever see a rather small subset of those offers. We can over-approximate this subset by performing an automated value analysis on the input specifications,

```
1  $\varsigma_0 \leftarrow$ initial abstract state of $\mathbb{S}$
2  $A \leftarrow$ all assignments within $\mathbb{S}$
3  $\sigma \leftarrow \{\varsigma_0\}$
4  while true do
5      $\sigma' \leftarrow \{[\![a]\!](\varsigma) \mid \varsigma \in \sigma, a \in A\}$
6      if $\sigma' \subseteq \sigma$ then
7          break
8      else
9          $\sigma \leftarrow \sigma \cup \sigma'$
10     end
11 end
12 return $\bigsqcup \sigma$
```

Fig. 3: Value analysis of a LAbS specification $\mathbb{S}$.

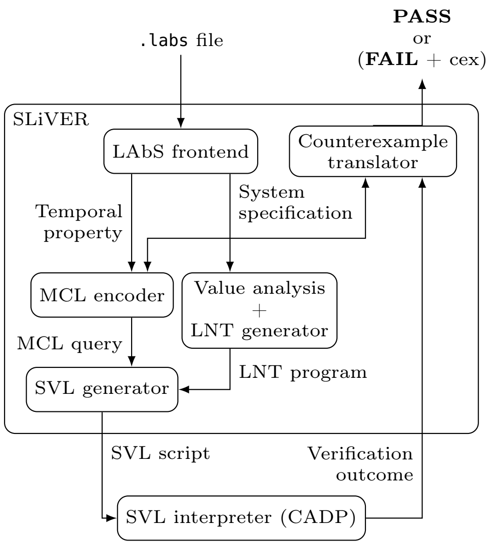

Fig. 4: Our compositional verification workflow.

and then constrain input offers based on the result of this analysis. Notice that this over-approximation will not lead our procedure to produce spurious counterexamples: during composition, spurious input offers will find no matching output offers, and will therefore be pruned away.

Given a specification $\mathbb{S}$, let us denote by $\mathcal{V}$ the set of its variables. From now on, we define an *abstract state* for $\mathbb{S}$ as a mapping from $\mathcal{V}$ to $P(I)$. Furthermore, we define the *merge* of two abstract states $\varsigma_1 \sqcup \varsigma_2$ as the state ς such that, for every $x \in \mathcal{V}$, $\varsigma(x) = n(\varsigma_1(x) \cup \varsigma_2(x))$. Our value analysis (Fig. 3) is straightforward. Initially, we compute the initial abstract state ς_0 for the given specification, and create a set σ that only contains this state. Computing ς_0 is immediate, as every LAbS specification must specify one or more feasible initial values for each declared variable.[4] We also extract from $\mathbb{S}$ a set A of all assignment statements that appear in it (lines 1–3). Then, we run a loop in which we abstractly evaluate every assignment $a \in A$ on every state $\varsigma \in \sigma$ and add the resulting states to σ (lines 4–11). If, at some point, we fail to find any new states, then we break out of this loop, and return as the final value analysis the merge of all states in σ (line 12).

The result of running this algorithm is an abstract state $\bar{\varsigma}$, mapping every variable name x to a powerset of intervals $\bar{\varsigma}(x)$. We can easily see that $\bar{\varsigma}(x)$ over-approximates the set of all values that x may actually assume across all feasible executions of $\mathbb{S}$. In fact, our analysis simply performs every possible assignment

[4] LAbS allows variables with an *undefined* initial value `undef`, but we currently do not support that feature in our analysis.

at every iteration, without considering the order in which they appear in the specifications, or whether they are guarded or not. Thus, we may say that we are considering the *chaos automaton* $chaos(A)$, i.e., the automaton that can always perform any of the assignments in $\mathbb{S}$. Every sequence of assignments from A, including those that are actual executions of $\mathbb{S}$, is a feasible execution in $chaos(A)$. Therefore, the set of values that a variable x can ever assume in the state space of $chaos(A)$ is a superset of those it may assume in the state space of $\mathbb{S}$, meaning that our analysis is sound but potentially over-approximating.

It is also nonterminating on infinite-state specifications, which are out of the scope of this work. This over-approximation also takes into account the exchange of values through stigmergic messages. To understand this, it suffices to notice that a value (say, κ) may be sent in a message only if it has been previously computed and stored in a stigmergic variable (say, x) by some agent. That is, messages cannot include values that are not the result of some sequence of assignments. But then, $chaos(A)$ will necessarily allow every agent to perform that same sequence of assignments and assign κ to x. Thus, there is no need to explicitly model message passing within our algorithm.

After computing $\bar{\varsigma}$, we can easily derive a Boolean function `goodL` that takes a local stigmergy L and returns `true` if and only if, for every stigmergic variable x, $L(x)$ is in $\bar{\varsigma}(x)$. Likewise, we can derive a function `goodI` that does the same for interfaces. Then, we force agents to only consider these objects as valid by constraining all their input offers, such as $\texttt{put(}?j, ?k, ?L_j, ?I_j\texttt{)}$ (as in the *recv-put* block of Fig. 2), by the predicate $j \neq id \wedge \texttt{goodL}(L_j) \wedge \texttt{goodI}(I_j)$.
Running example. Our bully election system (Listing 1) contains one variable `leader`, initialized to N for every agent. Plus, it refers to a special variable `id` that stores the agent's identifier. LAbS guarantees that identifiers are unique, contiguous and start at 0: so, the initial abstract state for our analysis will be $\varsigma_0 = \{\texttt{leader} \mapsto [N, N]; \texttt{id} \mapsto [0, N-1]\}$. Then, interpreting the assignment `leader <~ id` over ς_0 yields a new abstract state $\varsigma_1 = \{\texttt{leader} \mapsto [0, N-1]; \texttt{id} \mapsto [0, N-1]\}$. It is plain to see that our analysis cannot find any other states beyond this, since $[\![a]\!](\varsigma_1) = \varsigma_1$. Thus, the result of the analysis is just the merge $\varsigma_0 \sqcup \varsigma_1 = \{\texttt{leader} \mapsto \{[0, N-1], [N, N]\}; \texttt{id} \mapsto [0, N-1]\}$.

5 Compositional verification workflow

In this section, we describe how we combined the contributions described so far into an automated workflow for the compositional verification of LAbS systems. The workflow is implemented as a module within the SLiVER analysis tool,[5] and it is depicted in Figure 4.

First, a frontend parses a LAbS file and extracts the temporal property to verify, as well as the system specification. The former is transformed into an equivalent MCL query [10]; the latter, instead, is fed to a code generator to construct a parallel emulation program as described in Section 3. The code

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

generator also runs the value analysis described in Section 4 and uses the results to constrain all input offers on gates `put`, `qry`. Then, we generate an SVL script that describes the compositional verification task, and submit it to CADP; when the task is completed, we interpret its verdict (e.g., if a counterexample is found, we translate it into the syntax of LABS) and show the result to the user.

Listing 5 shows the structure of a verification script generated by our workflow. Intuitively, we ask CADP to generate the state space of the parallel emulation program of Listing 2 by means of *root leaf* reduction, minimizing modulo divergence-preserving sharp bisimulation [35], and then to verify our MCL query against the resulting transition system using the *Evaluator4* model checker. Notice that the program is wrapped in a *hiding* and a *priority* operator.

Hiding (***hide*** G ***in*** P ***end hide***) replaces all offers over gate G that occur in P with internal actions, denoted by τ. When generating our script, we determine all labels that are relevant to our query, denoted as *gates*(`"query.mcl"`), and then hide all other gates. This reduces the state space (as sharp bisimulation compresses sequences of τ-transitions) and thus accelerates model-checking.

The priority operator (***prio*** Ω ***in*** P ***end prio***) allows to specify a partial order of labels so that, when the state space of P is generated, transitions with a low-priority label are cut from every state that also features at least one transition with a higher-priority label. We use priorities to prune some sections of our programs where agents are free to interleave their actions in any order (e.g., when they have to react to an incoming message). These sections are not part of the semantics of LAbS, where message exchanges are treated as atomic events, but are rather an artefact of the encoding into parallel LNT programs. Furthermore, in these sections, each agent only affects its internal state, so reordering their actions does not affect the satisfaction of properties we are interested in verifying. Thus, we can analyse all orderings by only considering a representative one. Specifically, we give decreasing priorities to offers over gates `refresh`, `request`, and `l` (which agents use to signal a new assignment to a stigmergic variable); additionally, when multiple agents are willing to perform an action over one of these gates, the agent with lowest *id* is prioritized. This prioritization is independent of the specification being analysed, as it concerns the LNT encoding of stigmergic messaging regardless of the actual data being exchanged.

Lastly, our choice of sharp bisimulation is motivated by our use of the priority operator. In fact, applying sharp minimization under an appropriate set of strong actions, as we do here, preserves priorities (like strong minimization does), but also results in smaller LTSs than the one obtained through strong minimization. Minimizing modulo divergence-preserving branching bisimulation [27] (also known as divbranching bisimulation, for short) or weaker equivalences could in principle lead to even smaller LTSs, but would not preserve the semantics of the system. In fact, divbranching bisimulation and weaker equivalences are not congruences for the priority operator [11]. To see this, it suffices to consider the process $\tau.a$ which is divbranching bisimilar to a (Eq. 5), and observe that we can easily find a context with priorities, e.g., $C[P] =$ ***prio*** $a > b$ ***in*** $P \,\|\, b$ ***end prio***, such that replacing P with either $\tau.a$ or a gives us non-bisimilar terms (Eq. 8).

Listing 5: Structure of an SVL script for our verification workflow

```
"system.bcg" = root leaf divsharp reduction
hold "refresh", "request", "l"
in (
  hide all but gates("query.mcl") in
    prio
    "refresh" > "request" > "l"
    "refresh i .*" > "refresh j .*", i < j
    "request i .*" > "request j .*", i < j
    "l i .*" > "l j .*", i < j
    in
    ... (* Parallel emulation program (Listing 2 or 3) *)
    end prio
  end hide);

property CHECK is
  verify "query.mcl" with evaluator4
  in "system.bcg"
  expected TRUE
end property;
```

$$\tau.a \sim_{db} a \tag{5}$$

$$C[\tau.a] = \textbf{\textit{prio}}\ a > b\ \textbf{\textit{in}}\ \tau.a \parallel b\ \textbf{\textit{end prio}} = (\tau.a.b + b.\tau.a) \tag{6}$$

$$C[a] = \textbf{\textit{prio}}\ a > b\ \textbf{\textit{in}}\ a \parallel b\ \textbf{\textit{end prio}} = a.b \tag{7}$$

$$(\tau.a.b + b.\tau.a) \not\sim_{db} a.b \tag{8}$$

Experimental evaluation. To demonstrate our approach, we carry out a collection of verification tasks [10] in two different ways: first, we use a baseline workflow that generates a sequential LNT program, constructs its state space, minimizes it modulo divergence-preserving branching bisimulation,[6] and finally model-checks the reduced state space; then, we apply the compositional procedure proposed above. We then measure and compare the time and memory requirements of the two approaches.

We now provide a short overview of each system along with the properties to verify. The reader may refer to [10] for a detailed description. Systems whose name ends in `-rr` were verified assuming round-robin scheduling of agents. All properties are checked under fairness assumptions that exclude unfair loops from the verification [44].

[6] We use this relation because it preserves all the properties that we are interested in checking.

Table 1: Experimental results for compositional verification. Values in bold are better. –[a]: theoretical, based on *Compositional* measurements.

	Baseline [10]		Compositional		Parallel[a]	
System	Time (s)	Memory (MiB)	Time (s)	Memory (MiB)	Time (s)	Memory (MiB)
`flock-rr`	**1875**	12000	4461	**11805**	4426	**11805**
`flock`	4787	30865	4071	**11113**	**4038**	**11113**
`formation-rr`	1670	**1657**	2511	1938	**1558**	5875
`leader5`	**10**	**41**	34	117	18	212
`leader6`	77	**147**	104	225	**65**	258
`leader7`	1901	2038	374	**404**	**326**	**404**
`twophase2`	**9**	**50**	67	93	34	210
`twophase3`	500	**209**	233	322	**131**	560

The `flock` and `flock-rr` systems describe a simplified flocking behaviour. The systems feature 3 agents in a 5×5 arena. Each agent is initially given a nondeterministic position and direction of movement, with the latter stored as a pair of stigmergic variables. The agents move by following this direction vector. When two agents are sufficiently close (5 spaces apart or fewer), one of them may imitate the other's direction by receiving a stigmergic message. We check that, eventually, all agents move in the same direction.

In the `formation-rr` system, 3 agents are placed on a line segment of length 10. They use stigmergic variables to signal their presence to nearby agents. If an agent detects that it is too close to another, it moves one step away from it, unless it is at either end of the segment. We check that, eventually, all agents are at least 2 spaces apart from each other.

The `leader<N>` systems are three instances of our running example (Listing 1), respectively with 5, 6, and 7 nodes. We verify that all nodes eventually choose the one with id 0 as the leader.

The `twophase<N>` systems describe a two-phase commit scenario [29] with N *workers* and one *coordinator*. The coordinator initiates a voting session where all workers must decide whether a transaction should be committed. If all workers agree, the coordinator commits the transaction and starts a new voting round. We implemented the workers so as they always agree to commit, and all communication happens through stigmergy variables. We check that the coordinator commits transactions infinitely often.

All the experiments were performed on the Grid'5000 testbed, specifically on a node of the *Dahu* cluster. The node is equipped with two Intel Xeon Gold 6130 CPUs and 192 GiB of physical memory, and runs Debian 11 with version 5.10.0 of the Linux kernel.[7] We used CADP version 2022-h, and set a timeout of 3 hours and a memory limit of 32 GiB for all experiments. We collected the raw experimental data into a persistent replication package [13], which also includes the input LAbS specifications as well as binaries and scripts to facilitate reproducing the experiments.

[7] `https://www.grid5000.fr/w/Grenoble:Hardware#dahu`

We summarize the experimental results in Table 1. Columns from left to right report the name of the system and the time and memory required to verify it by the baseline and compositional approaches, respectively. In the last column, called *Parallel*, we show the time and memory it would take to perform the compositional workflow if we generated the individual state spaces simultaneously, e.g, on separate machines, or on separate cores of a multi-core machine. These are hypothetical measurements, derived from the *Compositional* ones. Specifically, each compositional verification experiment is made of several *tasks*, namely: k tasks $\mathcal{T}_1, \ldots, \mathcal{T}_k$ that construct the individual state spaces of the n agents, plus those of the processes `Timestamps` and (for round-robin systems) `Sched`; a task $\mathcal{T}_{\mathbb{P}}$ that assembles these state spaces into the one of the whole emulation program; and lastly, a model-checking task $\mathcal{T}_{\models}$. Let us denote the time and memory required to execute a task $\mathcal{T}$ by $\mathit{time}(\mathcal{T})$ and $\mathit{mem}(\mathcal{T})$, respectively. We can gather these measurements by executing an experiment with the *Compositional* workflow. Under this workflow, tasks are carried out sequentially: thus, the time required by the experiment is the sum of $\mathit{time}(\mathcal{T})$ for each task $\mathcal{T}$. For the same reason, the memory footprint is just the maximum of the memory requirements of every task. However, if the tasks $\mathcal{T}_i$ are carried out in parallel, then we would only have to wait for the task with the maximum $\mathit{time}(\mathcal{T}_i)$ before we are able to begin $T_{\mathbb{P}}$. At the same time, we would need to satisfy the memory requirements of all individual tasks at the same time, so we have to take into account the *sum* of all $\mathit{mem}(\mathcal{T}_i)$. We summarize these simple computations in Table 2. Notice that, on smaller systems, the memory requirements of SLiVER itself (around 400 MiB) would dominate that of the actual memory used for model checking. To better focus on comparing the performance of the two verification workflows, the table omits this overhead; we reserve the implementation of a more memory-efficient SLiVER for future work.

We can see that the baseline method is more time-efficient than the compositional one on some specific cases, e.g., when the overall system is rather small (`leader5`, `leader6`, `twophase2`) or round-robin scheduling has to be enforced (`flock-rr`, `formation-rr`). Full interleaving has an opposite effect: with the baseline procedure, verifying `flock` takes longer than `flock-rr`, whereas the compositional one can verify it faster. This may sound counterintuitive, since the former system only considers a subset of the latter's traces. Our explanation is that, for the compositional procedure, it is much easier to just freely compose agents rather than having to take the scheduling constraints into account. In other words, the scheduler acts as a sort of bottleneck to the compositional task, even though the resulting state space is smaller.

On small systems, namely `leader5`, `leader6`, and `twophase2`, the performance of the compositional procedure is likely affected by the overhead brought about by the component-wise state space generation. Furthermore, we are aware that CADP currently invokes the LNT compiler multiple times, i.e., for each component process, compounding this overhead. In conclusion, under specific conditions, the baseline approach may still produce a verdict faster than the compositional one. At the same time, the compositional approach appears to

Table 2: Time and memory requirements for the *Compositional* and *Parallel* workflows.

	Compositional	Parallel
Time	$\sum_{Tasks} time(\mathcal{T})$	$\max_i \{time(\mathcal{T}_i)\} + time(\mathcal{T}_{\mathbb{P}}) + time(\mathcal{T}_{\models})$
Memory	$\max_{Tasks} mem(\mathcal{T})$	$\max \left\{ \sum_i mem(\mathcal{T}_i), mem(\mathcal{T}_{\mathbb{P}}), mem(\mathcal{T}_{\models}) \right\}$

scale better than the baseline as the size of the systems grows. This is most evident in the `leader` systems, where every additional agent severely impacts the time and memory required by the baseline workflow; instead, the compositional approach shows a much less explosive, though still super-linear, progression.

The (theoretical) parallel procedure is, by definition, always faster than the compositional one. This speedup is most noticeable when the system involves many agents (`leader7`), or complex behavioural rules (`formation-rr`, `twophase3`). In some experiments, parallelization also incurs an increased memory usage, a rather obvious consequence of generating all individual state spaces at once. At the same time, it typically allows enjoying greater memory capacities (especially when done across multiple machines), so we do not expect this drawback to be significant. In others, however, both workflows have the same memory footprint, as the memory required to generate all state spaces simultaneously does not exceed the amount used by the other tasks ($\mathcal{T}_{\mathbb{P}}$ or $\mathcal{T}_{\models}$). Thus, in these cases the speedup from parallelization actually comes for free, i.e., it does not impact the overall memory usage.

6 Related work

Compositional verification has been successfully applied in several domains, ranging from hardware systems to communication protocols and service choreographies [18,20]. From a recent, extensive experimental evaluation, it appears to be effective under diverse network topologies, and its benefits generally become more evident as the size of the system under verification grows [8].

In this work, we exploit compositionality of state space generation. A somewhat related approach to fight state space explosion is modular (or compositional) reasoning [26], whereby a program is analysed by splitting it into components, for instance according to rely-guarantee conditions [34]. This form of compositionality has proved effective in several use cases, such as multi-robot and multi-agent systems [4, 33], railway networks [15], smart contracts [48], and authentication protocols [50]. All these applications, like our own work, exploit fully automated verification procedures; other frameworks, such as IVy [38], combine rely-guarantee reasoning with semi-automated procedures. LNT does provide constructs to express pre- and post- conditions on procedures (respectively denoted by `require` and `ensure`), but CADP does not use them in a compositional fashion yet.

Some classes of collective adaptive systems may be expressed in the form of population protocols [1], for which efficient parameterized verification procedures are known [2]. These may prove that a protocol satisfies a given property regardless of its size, but the properties of interest typically concern its eventual convergence to certain configurations, as the focus is to verify whether the protocol is able to carry out a desired computation. Our workflow checks systems of fixed size, but may support arbitrary branching-time temporal properties.

Preprocessing techniques to speed up program analysis by excluding invalid or infeasible values have also been proposed in the context of symbolic model checking. For instance, bounded model checking of programs featuring dynamic data structures may get more efficient by precomputing tight field bounds based on the structures' type invariants [42].

7 Conclusion and future work

In this work, we have argued that collective adaptive systems, being collections of autonomous and mutually interacting components, are naturally amenable to compositional techniques that can palliate state space explosion and thus aid in their verification. To support our claim, we have presented an encoding from high-level specifications into networks of LNT processes, introduced a simple value analysis to over-approximate the set of feasible offers between these processes, and demonstrated an automated workflow that exploits these ingredients to compositionally verify a collection of representative systems. Our experimental results do indicate that this procedure brings significant advantages over plain model checking. Besides evident gains in terms of absolute time and memory requirements, the proposed workflow appears to scale better in the number of agents, and can deal with freely-interleaved systems without particular effort compared to round-robin ones.

As future work, we intend to pursue several lines of research. For instance, the value analysis presented in this work is just a prototype and may be improved in several ways. Its approximation may be tightened by preserving some of the original behavioural structure and adding sensitivity to LAbS control constructs, such as guards. In general, powerset domains have well-known scalability issues that we could overcome by switching to more advanced abstract domains, such as Boxes [31] or donut domains [25], which may also track relations between variables. Our analysis only exploits data restriction; interfaces [28] could complement that with *behavioural* constraints, allowing to prune the state space of agents by cutting sequences of actions that are impossible under a given context. Thus, synthesizing such interfaces could enhance our compositional approach.

We also plan to actually implement the *Parallel* workflow theorized in Section 5, so that the generation of individual state spaces is distributed across multiple machines. This could be integrated with existing procedures for distributed state space generation [22], to further exploit parallelism; it would also allow us to measure how other factors, e.g., networked storage latency and transfer times, may affect the theoretical measurements presented in this work. An implementation of lighter-weight formal techniques, such as runtime verification [36] or statistical model-checking [46], could also provide some degree of assurance about the behaviour of very large collective systems.

Acknowledgements

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 (see `https://www.grid5000.fr`). Preliminary experiments were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at Umeå University partially funded by the Swedish Research Council through grant agreement no. 2018-05973. The authors wish to thank Wendelin Serwe and Hubert Garavel for their assistance in installing and running CADP on these machines.

References

1. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. Distributed Computing **20**(4), 279–304 (2007). https://doi.org/10.1007/s00446-007-0040-2
2. Blondin, M., Esparza, J., Jaax, S.: Peregrine: A tool for the analysis of population protocols. In: 30th International Conference on Computer Aided Verification (CAV). LNCS, vol. 10981, pp. 604–611. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_34
3. Bonabeau, E.: Agent-based modeling: Methods and techniques for simulating human systems. Proceedings of the National Academy of Sciences **99**(suppl 3), 7280–7287 (2002). https://doi.org/10.1073/pnas.082080899
4. Cardoso, R.C., Dennis, L.A., Farrell, M., Fisher, M., Luckcuck, M.: Towards compositional verification for modular robotic systems. In: 2nd Workshop on Formal Methods for Autonomous Systems (FMAS). EPTCS, vol. 329, pp. 15–22 (2020). https://doi.org/10.4204/EPTCS.329.2
5. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: 2nd International Symposium on Programming. pp. 106–130. Dunod (1976)
6. Crowston, K., Rezgui, A.: Effects of stigmergic and explicit coordination on Wikipedia article quality. In: 53rd Hawaii International Conference on System Sciences (HICSS). pp. 1–10. ScholarSpace (2020)
7. 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
8. de Putter, S., Wijs, A.: To compose, or not to compose, that is the question: An analysis of compositional state space generation. In: 22nd International Symposium on Formal Methods (FM). vol. 10951, pp. 485–504. Springer (2018). https://doi.org/10.1007/978-3-319-95582-7_29

9. Di Stefano, L., De Nicola, R., Inverso, O.: Verification of distributed systems via sequential emulation. ACM Transactions on Software Engineering and Methodology **31**(3) (2022). https://doi.org/10.1145/3490387
10. Di Stefano, L., Lang, F.: Verifying temporal properties of stigmergic collective systems using CADP. In: 10th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA). LNCS, vol. 13036, pp. 473–489. Springer (2021). https://doi.org/10.1007/978-3-030-89159-6_29
11. Di Stefano, L., Lang, F.: Compositional Verification of Priority Systems using Sharp Bisimulation. Research report, INRIA (2022), `https://hal.inria.fr/hal-03640683`
12. Di Stefano, L., Lang, F., Serwe, W.: Combining SLiVER with CADP to analyze multi-agent systems. In: 22nd International Conference on Coordination Models and Languages (COORDINATION). LNCS, vol. 12134, pp. 370–385. Springer (2020). https://doi.org/10.1007/978-3-030-50029-0_23
13. Di Stefano, L., Lang, F.: Replication Package for the paper: Compositional Verification of Stigmergic Collective Systems (2022). https://doi.org/10.5281/zenodo.7043353
14. El-Sayed, A.M., Scarborough, P., Seemann, L., Galea, S.: Social network analysis and agent-based modeling in social epidemiology. Epidemiologic Perspectives & Innovations **9** (2012). https://doi.org/10.1186/1742-5573-9-1
15. Fantechi, A., Haxthausen, A.E., Macedo, H.D.: Compositional verification of interlocking systems for large stations. In: 15th International Conference on Software Engineering and Formal Methods (SEFM). LNCS, vol. 10469, pp. 236–252. Springer (2017). https://doi.org/10.1007/978-3-319-66197-1_15
16. Filé, G., Ranzato, F.: The powerset operator on abstract interpretations. Theoretical Computer Science **222**(1-2), 77–111 (1999). https://doi.org/10.1016/S0304-3975(98)00007-3
17. Garavel, H., Lang, F.: SVL: A Scripting Language for Compositional Verification. In: 21st International Conference on Formal Techniques for Networked and Distributed Systems (FORTE). IFIPAICT, vol. 69, pp. 377–394. Springer (2001). https://doi.org/10.1007/0-306-47003-9_24
18. Garavel, H., Lang, F., Mateescu, R.: Compositional verification of asynchronous concurrent systems using CADP. Acta Informatica **52** (2015). https://doi.org/10.1007/s00236-015-0226-1
19. 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** (2013). https://doi.org/10.1007/s10009-012-0244-z
20. Garavel, H., Lang, F., Mounier, L.: Compositional verification in action. In: 23rd International Conference on Formal Methods for Industrial Critical Systems (FMICS). LNCS, vol. 11119, pp. 189–210. Springer (2018). https://doi.org/10.1007/978-3-030-00244-2_13
21. Garavel, H., Lang, F., Serwe, W.: From LOTOS to LNT. In: ModelEd, TestEd, TrustEd - Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday. LNCS, vol. 10500, pp. 3–26. Springer (2017). https://doi.org/10.1007/978-3-319-68270-9_1
22. Garavel, H., Mateescu, R., Bergamini, D., Curic, A., Descoubes, N., Joubert, C., Smarandache-Sturm, I., Stragier, G.: DISTRIBUTOR and BCG_MERGE: Tools for distributed explicit state space generation. In: 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3920, pp. 445–449. Springer (2006). https://doi.org/10.1007/11691372_30

23. Garavel, H., Sighireanu, M.: A graphical parallel composition operator for process algebras. In: Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE) and Protocol Specification, Testing and Verification (PSTV). IFIPAICT, vol. 156, pp. 185–202. Kluwer (1999)
24. Garcia-Molina, H.: Elections in a distributed computing system. IEEE Transactions on Computers **31**(1), 48–59 (1982). https://doi.org/10.1109/TC.1982.1675885
25. Ghorbal, K., Ivancic, F., Balakrishnan, G., Maeda, N., Gupta, A.: Donut domains: Efficient non-convex domains for abstract interpretation. In: 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 7148, pp. 235–250. Springer (2012). https://doi.org/10.1007/978-3-642-27940-9_16
26. Giannakopoulou, D., Namjoshi, K.S., Pasareanu, C.S.: Compositional reasoning. In: Handbook of Model Checking. Springer (2018)
27. van Glabbeek, R.J., Weijland, W.P.: Branching Time and Abstraction in Bisimulation Semantics. Journal of the ACM **43** (1996)
28. Graf, S., Steffen, B.: Compositional minimization of finite state systems. In: 2nd International Workshop on Computer Aided Verification (CAV). LNCS, vol. 531, pp. 186–196. Springer (1990). https://doi.org/10.1007/BFb0023732
29. Gray, J.: Notes on data base operating systems. In: Operating Systems, An Advanced Course. LNCS, vol. 60, pp. 393–481. Springer (1978). https://doi.org/10.1007/3-540-08755-9_9
30. Grimm, V., Railsback, S.F.: Agent-based models in ecology: Patterns and alternative theories of adaptive behaviour. In: Agent-Based Computational Modelling: Applications in Demography, Social, Economic and Environmental Sciences, pp. 139–152. Physica-Verlag (2006). https://doi.org/10.1007/3-7908-1721-X_7
31. Gurfinkel, A., Chaki, S.: Boxes: A symbolic abstract domain of boxes. In: 17th International Symposium on Static Analysis (SAS). LNCS, vol. 6337, pp. 287–303. Springer (2010). https://doi.org/10.1007/978-3-642-15769-1_18
32. Hillston, J.: Challenges for quantitative analysis of collective adaptive systems. In: 8th International Symposium on Trustworthy Global Computing (TGC), Revised Selected Papers. LNCS, vol. 8358, pp. 14–21. Springer (2013). https://doi.org/10.1007/978-3-319-05119-2_2
33. Jones, A.V.: Model Checking and Compositional Reasoning for Multi-Agent Systems. Ph.D. thesis, Imperial College London, UK (2014). https://doi.org/10.25560/32695
34. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Transactions on Programming Languages and Systems **5** (1983). https://doi.org/10.1145/69575.69577
35. Lang, F., Mateescu, R., Mazzanti, F.: Sharp congruences adequate with temporal logics combining weak and strong modalities. In: 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 12079, pp. 57–76. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_4
36. Leucker, M., Schallhart, C.: A brief account of runtime verification. Journal of Logic and Algebraic Programming **78** (2009). https://doi.org/10.1016/j.jlap.2008.08.004
37. Mateescu, R., Thivolle, D.: A model checking language for concurrent value-passing systems. In: 15th International Symposium on Formal Methods (FM). vol. 5014, pp. 148–164. Springer (2008). https://doi.org/10.1007/978-3-540-68237-0_12

38. McMillan, K.L., Padon, O.: Ivy: A multi-modal verification tool for distributed algorithms. In: 32nd International Conference on Computer Aided Verification (CAV). LNCS, vol. 12225, pp. 190–202. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_12
39. Milner, R.: A Calculus of Communicating Systems. Springer (1980). https://doi.org/10.1007/3-540-10235-3
40. Olner, D., Evans, A.J., Heppenstall, A.J.: An agent model of urban economics: Digging into emergence. Comput. Environ. Urban Syst. **54** (2015). https://doi.org/10/f77hp8
41. Pinciroli, C., Beltrame, G.: Buzz: An extensible programming language for heterogeneous swarm robotics. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 3794–3800. IEEE (2016). https://doi.org/10/gdnbz4
42. Ponzio, P., Godio, A., Rosner, N., Arroyo, M., Aguirre, N., Frias, M.F.: Efficient bounded model checking of heap-manipulating programs using tight field bounds. In: 24th International Conference on Fundamental Approaches to Software Engineering (FASE). LNCS, vol. 12649, pp. 218–239. Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_11
43. Qadeer, S., Wu, D.: KISS: Keep it simple and sequential. In: Conference on Programming Language Design and Implementation (PLDI). pp. 14–24. ACM (2004). https://doi.org/10.1145/996841.996845
44. 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
45. Robles, G., Merelo, J.J., Gonzales-Barahona, J.M.: Self-organized development in libre software: A model based on the stigmergy concept. In: 6th International Workshop on Software Process Simulation and Modeling (ProSim). Fraunhofer (2005)
46. Sen, K., Viswanathan, M., Agha, G.: Statistical model checking of black-box probabilistic systems. In: 16th International Conference on Computer Aided Verification (CAV). LNCS, vol. 3114, pp. 202–215. Springer (2004). https://doi.org/10.1007/978-3-540-27813-9_16
47. Theraulaz, G., Bonabeau, E.: A brief history of stigmergy. Artificial Life **5** (1999). https://doi.org/10.1162/106454699568700
48. Wesley, S., Christakis, M., Navas, J.A., Trefler, R.J., Wüstholz, V., Gurfinkel, A.: Compositional verification of smart contracts through communication abstraction. In: 28th International Symposium on Static Analysis (SAS). LNCS, vol. 12913, pp. 429–452. Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_21
49. Yeh, W.J., Young, M.: Compositional reachability analysis using process algebra. In: Symposium on Testing, Analysis, and Verification (TAV). pp. 49–59. ACM (1991). https://doi.org/10.1145/120807.120812
50. Zhang, Z., de Amorim, A.A., Jia, L., Pasareanu, C.S.: Automating compositional analysis of authentication protocols. In: 20th Conference on Formal Methods in Computer Aided Design (FMCAD). pp. 113–118. IEEE (2020). https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_18