Multi-Agent Systems with Virtual Stigmergy☆

Rocco De Nicola[a], Luca Di Stefano[b,*], Omar Inverso[b]

[a]*IMT School for Advanced Studies, Lucca, Italy*
[b]*Gran Sasso Science Institute (GSSI), L'Aquila, Italy*

Abstract

We introduce a simple language for multi-agent systems that lends itself to intuitive design of local specifications. Agents operate on (parts of) a decentralized data structure, the stigmergy, that contains their (partial) knowledge. Such knowledge is asynchronously propagated across local stigmergies. In this way, local changes may influence global behaviour. The main novelty is that our interaction mechanism combines stigmergic interaction with attribute-based communication. Specific conditions for interaction can be expressed in the form of predicates over exposed features of the agents. Additionally, agents may access a global environment. After presenting the language, we show its expressiveness by considering some illustrative case studies. We also include preliminary results towards automated verification via a mechanizable symbolic encoding that enables us to exploit verification tools developed for mainstream languages.

1. Introduction

Multi-agent systems are collections of autonomous agents that operate according to some local rules and a limited mutual awareness. They are a convenient formalism for representing several classes of complex systems and can support formal reasoning about them. An issue that arises when considering a multi-agent system is how to determine whether a global property of interest emerges from the combination of the local behaviours of the different individual agents. The availability of a formal description of a multi-agent system allows one to apply automated verification techniques and can be instrumental for obtaining strong guarantees about its global behaviour. Simulation-based approaches, on the other hand, may be more effective when dealing with larger multi-agent systems, due to the considerably large state spaces resulting from their distributed and asynchronous nature. Therefore, the two approaches should be considered complementary to each other.

In this paper, we introduce a language for describing multi-agent systems that lends itself to an intuitive design of local specifications and that can be used as the

☆Work partially funded by MIUR project PRIN 2017FTXR7S *IT MATTERS* (Methods and Tools for Trustworthy Smart Systems).

*Corresponding author.
Email address: luca.distefano@gssi.it (Luca Di Stefano)

basis for automated analysis. The language, which we call LAbS for Language with Attribute-based Stigmergies, is simple yet versatile enough to model several interesting classes of systems. It combines stigmergic interaction [1, 2] with attribute-based communication [3]. A key concept of the language is that of *virtual stigmergy*, a distributed data structure that can model global knowledge. Each agent operates only on his local copy of the stigmergy, that stores his own (partial) knowledge of the system. Individual knowledge is then asynchronously propagated across other local stigmergies. Thus, changes by an agent may indirectly affect the behaviour of another.

In the originally proposed version of the virtual stigmergy [4], agents are concrete entities, each at a specific position in space, that can communicate across the stigmergy only if they are within a given distance from each other. To increase expressiveness, we generalise stigmergic interaction to arbitrary *predicates* overexposed features, referred to as *attributes*, of the agents. In fact, our language has no explicit concept of position for agents, and thus of neighbourhood. An agent can have instead local attributes, and predicates over these attributes can express the conditions for two agents to be allowed to exchange knowledge. Movement is no longer seen as a specific action; an agent may update the attribute that encodes its position by performing a standard update action.

The generalisation of stigmergic interaction outlined above increases the flexibility of the language and allows to model a wider class of systems. However, it is still not sufficiently expressive to naturally model those classes of multi-agent systems where the global *environment* plays a crucial role [5, 6]. To address this shortcoming, we extend our language with tailored primitives to explicitly model actions on the environment.

This work extends our original presentation of the language [7] in several ways. The new syntax and semantics support *multiple stigmergies* and improve the specification of situated systems. Multiple stigmergies allow us to naturally describe further interesting classes of systems, where agents can communicate in different ways. For instance, they can be used to directly model multi-robot systems where robots have multiple sensors and communication devices, and decide the equipment to use depending on specific environmental conditions. As for situated systems, environment variables may now directly occur in expressions and guards. This makes it easier to describe systems where agents also interact via the environment.

New case studies have been added to those related to *birds flocking*, *robots foraging* and *opinion formation* considered in [7]. To vindicate the flexibility of our language and its ability of expressing different interesting classes of systems, in this paper we also model *Boids* [8], and *population protocols* [9, 10]. The former is a widely-used model of flocking behaviour as observed in different classes of natural systems. It extends the flocking case study by allowing additional interaction strategies, for getting closer and avoiding collisions, and not only for moving in the same direction. The latter are a type of *gossip protocols* that rely on a distributed communication paradigm inspired by the spreading of epidemics and by the gossip phenomenon observed in social networks. For both classes of systems, we do provide experimental results about their automated analysis. Our modelling of *Boids* systems shows the benefits of adding multiple stigmergies to the language and, to the best of our knowledge, our work reports the first results about formal verification of such systems; previous investigations only exploited simulation-based techniques.

The rest of the paper is organized as follows. In Section 2 we present a revised

version of the formal semantics of the core language, allowing us to define systems where agents interact indirectly through multiple stigmergies. In Section 3 we demonstrate the features of the language by modelling the *Boids* system, and include preliminary results about its automated analysis together with a discussion about the impact on verification of the different parameters of the specified system and of the used verification tools. In Section 4 we further enrich the language with environment-oriented primitives and show how LAbS can naturally model other classes of systems dealing with *robot foraging*, *opinion formation* and *gossiping*. In Section 5 we summarise our main achievements, compare our work with others, and suggest directions for future research.

2. The LAbS language

In this section we introduce LAbS (Language with Attribute-based Stigmergies), a language that has been designed to program multi-agent systems (MAS). The interaction mechanisms of LAbS are inspired by the specific form of stigmergic interaction originally proposed with the Buzz language [1] that we generalise by exploiting attribute-based communication as introduced in [3].

A key concept of LAbS is the *virtual stigmergy*, a distributed data structure that models the global knowledge of the system. Each agent maintains a local copy of (part of) this data structure, that contains his own (partial) knowledge of the system. We call these copies *local stigmergies*. An agent reads from and writes to his local stigmergy only. Knowledge is silently and asynchronously propagated across local stigmergies. This way, indirect agents interaction is achieved.

Formally, local stigmergies $L \in \mathcal{L}$ are partial functions that map keys to timestamped values: $\mathcal{L} = \mathcal{K}_L \hookrightarrow \mathcal{V} \times \mathbb{N}$, where $\mathcal{K}_L$ and $\mathcal{V}$ are the sets of allowed keys and values, respectively. We use natural numbers to represent timestamps. If $(x, v, t) \in L$, we say that v is the *value* of x and that t is its *timestamp* in the local stigmergy L. We refer to these as $value(L, x)$ and $time(L, x)$, respectively. We write $L(x) = \perp$ whenever $\forall v.\forall t.(x, v, t) \notin L$.

The operations on the stigmergy and the propagation mechanism are the following. When an agent *writes* a key-value pair into his local stigmergy, a timestamp is retrieved from a global clock and bound to the pair. If the local stigmergy contains an entry with the same key, it is replaced by the new one. The new data is then automatically (though asynchronously) propagated to its neighbours. For agents in the neighbourhood that already have a value bound to the same key but with a newer timestamp, the propagation has no effect; all the other agents update their local stigmergy, and in turn, propagate the new value. In the long run, this process allows information to be spread throughout the system. Conversely, each time an agent *reads* from its local stigmergy, a key confirmation request is sent to the neighbourhood to confirm whether the data just accessed is up-to-date. This will, in turn, trigger the propagation of more recent information from the local stigmergies nearby, the update of any older entries, and again their propagation.

Insertion of a value in a local stigmergy is a function $\oplus : \mathcal{L} \times (\mathcal{K}_L \times \mathcal{V} \times \mathbb{N}) \longrightarrow \mathcal{L}$ defined as the smallest relation that satisfies the rules in Table 1, where $L[x \mapsto (v, t)]$ denotes the partial function L' such that $L'(x) = (v, t)$ and $L'(x') = L(x')\ \forall x' \neq x$. Please notice that the actual definition of insertion of Table 1 implies that only new values are

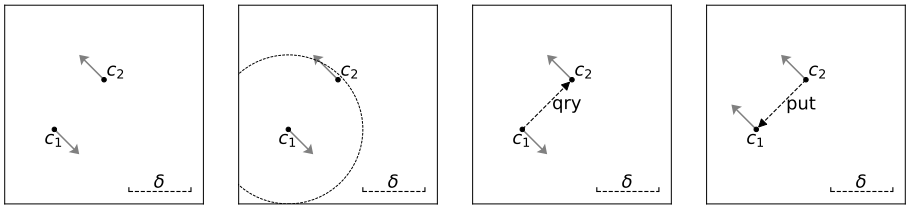
considered. A value is new if its key is missing from the local stigmergy or it has a more recent timestamp than the existing one.

$$\frac{L(x) = \bot}{L \oplus (x, v, t) = L[x \mapsto (v, t)]} \text{ (ADD)} \qquad \frac{t > time(L, x)}{L \oplus (x, v, t) = L[x \mapsto (v, t)]} \text{ (UPDATE)}$$

$$\frac{t \leq time(L, x)}{L \oplus (x, v, t) = L} \text{ (DISCARD)}$$

Table 1: Operations on the virtual stigmergy.

An example of stigmergic interaction is shown in Figure 1. Here, agents intend to move by following the direction stored in the virtual stigmergy. Initially (a), two



(a) (b) (c) (d)

Figure 1: A possible evolution of a system in the presence of stigmergic interaction.

agents, c_1 and c_2, are moving in opposite directions. When c_1 moves (b), it accesses the stigmergy to read its own direction: therefore, it asks its neighbours if a newer direction is available (c), and receives a more up-to-date value from c_2 (d). In the figure, the circle represents the communication range of c_1, which has radius δ (b), while the dashed arrows indicate stigmergic communication; labels *query* and *put* should be self-explanatory. We would like to stress that these protocols are transparent to the designer of the individual behaviour, who only needs to specify read and write operations on the agent's local copy of the data structure.

The above description slightly deviates from the Buzz language in a few points. First, Buzz stigmergies are based on Lamport timestamps [11] and rely on unique agent identifiers to break ties, which may occur when the same timestamp is used more than once; our language is currently more limited, as it relies on a global clock (see Sect. 2.5). Moreover, differently from Buzz, in our core language there is no explicit message passing between agents. They can only interact via the stigmergy or the environment. We introduced the above assumptions for the sake of simplicity and homogeneity. We might reconsider them in future revisions of our language, depending on the target domain. However, our calculus also generalises some of the concepts related to the virtual stigmergies of Buzz. Most importantly, in our language the ability to exchange information through the stigmergy is not directly constrained by spatial vicinity. In fact, there is no explicit concept of an agent's position at all. Rather, we rely upon local

properties of the agents to determine whether they are allowed to communicate.

The syntax of LAbS is described in Table 2. In **expressions**, we assume that $v \in \mathcal{V}$, $x \in \mathcal{K}_L \cup \mathcal{K}_I$ (where $\mathcal{K}_I$ is a set of keys disjoint from $\mathcal{K}_L$), and $\diamond$ stands for any binary operator over $\mathcal{V}$ (such as $+, -, \times \ldots$). In **guards**, $\bowtie$ denotes comparison relations over $\mathcal{V} \cup \{\bot\}$, namely $(=, <, >)$. We assume that K is taken from a set of named processes.

$S ::=$	$a \mid S \parallel S$	Systems
$a ::=$	$\langle I, L, P, Zc, Zp \rangle$	Agents
$P ::=$	$0 \mid \surd \mid \alpha \mid P; P \mid P + P \mid b \rightarrow P \mid P \mid P \mid K$	Processes
$b ::=$	$true \mid e \bowtie e \mid \neg b \mid b \wedge b \mid b \vee b$	Guards
$\alpha ::=$	$x \leftarrow e \mid x \leftsquigarrow e$	Elementary actions
$e ::=$	$v \mid x \mid e \diamond e$	Expressions

Table 2: LAbS syntax.

A **system** is the parallel composition of a number of agents. An **agent** is a 5-ple $\langle I, L, P, Zc, Zp \rangle$ where:

- $I \in \mathcal{I}$ is the *interface* of the agent;
- $L \in \mathcal{L}$ is the *local stigmergy* of the agent;
- P is a *process* describing the behaviour of the agent;
- Zc is the set of keys that the agent has to confirm (i.e. query);
- Zp is the set of keys that the agent must propagate.

Thus, each agent is equipped with a local stigmergy and an interface, which is a dynamic set of key-value pairs (*attributes*). Attributes can be specified in the initialization phase and modified at runtime; they represent either a variable in the agent's memory, or a physical property of the agent (for instance, its position). LAbS makes no distinction between these two kinds of information. For instance, an agent may move by updating the attribute that represents its position. Besides, attributes can be used to determine whether two agents are able to communicate: the user of the language can specify a custom, attribute-based predicate that, given two interfaces, determines whether the corresponding agents can communicate. This is an important source of flexibility, as different means of communication for an agent can be modelled. The ability of the agents to change their attributes at any time means that connections among agents can be dynamically established or removed.

2.1. Processes and expressions

Processes are used to model behaviour of the agents. We present their syntax in Table 2 and their operational semantics in Table 3. There, P and Q denote processes

while $\surd$ denotes successful termination, α represents the actions used to update attributes ($x \leftarrow e$) or stigmergic variables ($x \leftsquigarrow e$), with the result of the evaluation of an expression, while λ is a placeholder for either $\surd$ or α.

$$\frac{}{\surd \overset{\surd}{\mapsto} 0}\ (\textsc{tick}) \qquad \frac{}{\alpha \overset{\alpha}{\mapsto} \surd}\ (\textsc{act}) \qquad \frac{P \overset{\lambda}{\mapsto} P'}{P + Q \overset{\lambda}{\mapsto} P'}\ (\textsc{choice-l}) \qquad \frac{Q \overset{\lambda}{\mapsto} Q'}{P + Q \overset{\lambda}{\mapsto} Q'}\ (\textsc{choice-r})$$

$$\frac{P \overset{\alpha}{\mapsto} P'}{P; Q \overset{\alpha}{\mapsto} P'; Q}\ (\textsc{seq}_1) \qquad \frac{P \overset{\surd}{\mapsto} P' \quad Q \overset{\lambda}{\mapsto} Q'}{P; Q \overset{\lambda}{\mapsto} Q'}\ (\textsc{seq}_2) \qquad \frac{P \overset{\lambda}{\mapsto} P' \quad K \triangleq P}{K \overset{\lambda}{\mapsto} P'}\ (\textsc{con})$$

$$\frac{P \overset{\alpha}{\mapsto} P'}{P \mid Q \overset{\alpha}{\mapsto} P' \mid Q}\ (\textsc{par}_1) \qquad \frac{P \overset{\surd}{\mapsto} P' \quad Q \overset{\lambda}{\mapsto} Q'}{P \mid Q \overset{\lambda}{\mapsto} Q'}\ (\textsc{par}_2) \qquad \frac{P_1 \mid P_2 \overset{\lambda}{\mapsto} P'}{P_2 \mid P_1 \overset{\lambda}{\mapsto} P'}\ (\textsc{par}_{\textsc{comm}})$$

Table 3: Semantics of processes.

Below, we briefly comment on the main semantic rules for each term. The term 0 represents the *idle* process and thus has no corresponding semantic rule. The term $\surd$ represents the elementary process that performs action $\surd$ and becomes idle (TICK). The term α represent the elementary process that performs an action and terminates (ACT).

The term $P + Q$ represents the nondeterministic process which can behave either as P (CHOICE-L) or Q (CHOICE-R). The sequential composition of two processes is denoted by the term $P; Q$, that represents the process that behaves as P until it terminates ($\textsc{seq}_1$), and if P does terminate and Q performs a transition λ to become Q', then $P; Q$ can perform the same λ transition and continue as Q' ($\textsc{seq}_2$).

Recursion is modelled through named process invocation. We assume that there exists a set of *process definitions* $K \triangleq P$, where P is a process term, named K, defined according to the syntax of Table 2 that may contain references to K itself and to other process constants; rule (CON) amounts to saying that K can perform exactly the same actions of the process term associated to it.

The parallel composition of processes is a process $P \mid Q$ where the executions of P and Q are interleaved ($\textsc{par}_1$), and upon termination of one of the parallel components the other continues in isolation ($\textsc{par}_2$). The parallel composition operator is commutative ($\textsc{par}_{\textsc{comm}}$).

Intuitively, the guarded process $b \rightarrow P$ can only continue as P if the guard b is satisfied. We will formalize this rule when we introduce the semantics of agents (see Table 7) since the evaluation of a guard depends on the state of the agent.

Expressions may contain constants, references to the value of local attributes, or stigmergic keys. A guard may either be the *true* predicate, which is always satisfied, or a comparison between two expressions. Guards can also be negated ($\neg b$) or composed through the conjunction and disjunction operators, $\wedge$ and $\vee$.

The semantics of expressions is formalized by a semantic function $\mathcal{E}[\![\cdot]\!]$ (Table 4), where $\mathcal{I}$ and $\mathcal{L}$ denote the set of all interfaces and stigmergies, respectively. We assume

$$\mathcal{E}[\![\cdot]\!] : Expr \longrightarrow \mathcal{I} \to \mathcal{L} \hookrightarrow \mathcal{V} \qquad \mathcal{K}[\![\cdot]\!] : Expr \longrightarrow 2^{\mathcal{K}_L}$$

$$\mathcal{E}[\![v]\!] = \lambda I.\lambda L.v \qquad \mathcal{K}[\![v]\!] = \emptyset$$

$$\mathcal{E}[\![x]\!] = \begin{cases} \lambda I.\lambda L.I(x) & \text{if } x \in \mathcal{K}_I \\ \lambda I.\lambda L.value(L,x) & \text{if } x \in \mathcal{K}_L \end{cases} \qquad \mathcal{K}[\![x]\!] = \begin{cases} \{x\} & \text{if } x \in \mathcal{K}_L \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathcal{E}[\![e_1 \diamond e_2]\!] = \lambda I.\lambda L.\mathcal{E}[\![e_1]\!](I,L) \diamond \mathcal{E}[\![e_2]\!](I,L) \qquad \mathcal{K}[\![e_1 \diamond e_2]\!] = \mathcal{K}[\![e_1]\!] \cup \mathcal{K}[\![e_2]\!]$$

$$\mathcal{E}[\![e \diamond \bot]\!] = \mathcal{E}[\![\bot \diamond e]\!] = \lambda I.\lambda L.\bot$$

Table 4: Semantics of expressions.

that $v \in \mathcal{V}$, $x \in \mathcal{K}$; $\diamond$ and $\bowtie$ are the same as in Table 2. We also assume that the equality $\bot = \bot$ holds, while all other relations $\bowtie$ involving $\bot$ never do. We denote with $\mathcal{K}[\![\cdot]\!]$ a function that computes the set of stigmergy keys needed to evaluate an expression. This function is instrumental to formalize the mechanisms of virtual stigmergies. We allow $\mathcal{E}[\![\cdot]\!]$ to return the undefined value $\bot$, for instance, when the expression refers to an undefined value or applies an operator to incompatible values (e.g. adding a number to a string). With a slight abuse of notation, we will use $\mathcal{K}[\![b]\!]$ to denote the union of $\mathcal{K}[\![e]\!]$ for all sub-expressions of a guard b.

Satisfaction of a guard b is formalized as a relation $I, L \models b$ (Table 5). We say that a guard b is *well-defined* with respect to interface I and stigmergy L if all the sub-expressions of b refer to defined attributes and stigmergy keys (this relation is denoted by $\vdash$ in Table 6). If b is not well-defined, then it may happen that neither b nor $\neg b$ hold. This means that the law of excluded middle is not generally valid, and this is why, although we have conjunction and negation, we have also introduced an operator for disjunction; $b_1 \vee b_2$ does not have the same meaning as $\neg(\neg b_1 \wedge \neg b_2)$.

Well-definedness is not a strict requirement for all types of guards: satisfaction of $b_1 \vee b_2$ only requires at least one of the two sub-guards to hold. By defining disjunction in this way, we allow agents to operate even though their knowledge is partial: in fact, $b_1 \vee b_2 \to P$ may enable P also when one of the sub-guards is not well-defined.

$$\begin{array}{lcll} I, L & \models & true & \\ I, L & \models & \neg b & \iff I, L \vdash \neg b \text{ and } I, L \not\models b \\ I, L & \models & e_1 \bowtie e_2 & \iff I, L \vdash e_1 \text{ and } I, L \vdash e_2 \text{ and } \mathcal{E}[\![e_1]\!](I,L) \bowtie \mathcal{E}[\![e_2]\!](I,L) \\ I, L & \models & b_1 \wedge b_2 & \iff I, L \models b_1 \text{ and } I, L \models b_2 \\ I, L & \models & b_1 \vee b_2 & \iff I, L \models b_1 \text{ or } I, L \models b_2 \end{array}$$

Table 5: Satisfaction of guards.

$$
\begin{array}{lcll}
I, L & \vdash & v & \\
I, L & \vdash & x & \iff (x \in \mathcal{K}_I \text{ and } I(x) \neq \bot) \text{ or } (x \in \mathcal{K}_L \text{ and } \mathit{value}(L, x) \neq \bot) \\
I, L & \vdash & e_1 \diamond e_2 & \iff I, L \vdash e_1 \text{ and } I, L \vdash e_2 \\
 & & & \\
I, L & \vdash & \mathit{true} & \\
I, L & \vdash & \neg b & \iff I, L \vdash b \\
I, L & \vdash & b_1 \wedge b_2 & \iff I, L \vdash b_1 \text{ and } I, L \vdash b_2 \\
I, L & \vdash & b_1 \vee b_2 & \iff I, L \vdash b_1 \text{ and } I, L \vdash b_2
\end{array}
$$

Table 6: Well-definedness of expressions and guards.

2.2. Link predicates

A *link predicate* is a predicate over the knowledge (i.e. interface and local stigmergy) of two agents, describing the conditions that allow them to communicate. We assume that each stigmergic variable x has an associated link predicate φ_x. When multiple variables occur within the same link predicate φ_s, we say that they belong to the same *virtual stigmergy* s. Two agents are *neighbours* with respect to stigmergy s if they satisfy φ_s. This abstraction is useful, for instance, in the case of multi-robot systems, where predicates allow to effectively model different sensors and capabilities for each robot. Link predicates have the following syntax:

$$
\begin{array}{lll}
\varphi ::= & \mathit{true} \mid \eta \bowtie \eta \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi & \text{predicate} \\
\eta ::= & v \mid x \mid \eta \diamond \eta \quad x \in \mathcal{K}_I \cup \mathcal{K}_L & \text{expression}
\end{array}
$$

We denote with $\mathcal{H}[\![\cdot]\!]$ the semantic function of expressions η. We omit a formal definition, as it is nearly identical to the function $\mathcal{E}[\![\cdot]\!]$ described in Table 3. The only difference is that $\mathcal{H}[\![\cdot]\!]$ evaluates a predicate against two interfaces and two local stigmergies. Identifiers are decorated with indexes (x_s, x_r) to clarify whether they refer to a variable in the knowledge of the sender or the potential receiver, respectively.

Similarly, the definitions of satisfaction and well-definedness closely follow the ones introduced for guards. The ability of combining link predicates offers an intuitive way to model different communication modes for agents. For instance, the predicate

$$
\|pos_s - pos_r\| \leq \delta \ \vee \ (\mathit{LongRange}_s = \text{“true”} \wedge \mathit{LongRange}_r = \text{“true”}),
$$

where $\|\cdot\|$ denotes the Euclidean norm, states that two agents can communicate if their positions are closer than a constant δ or if they both possess a long-range networking device.

2.3. Agents and systems

Agent-level transitions, triggered when an agent performs an action, are modelled in Table 7. We assume that $v = \mathit{value}(L, x)$ and $t = \mathit{time}(L, x)$. Rule (SKIP) states that an

agent can perform a transition when its behaviour allows a $\surd$-move. According to rule (ATTR) we have that, when an agent performs an attribute update $x \leftarrow e$, the result of expression e is bound to attribute x, and the stigmergy keys used to evaluate e are added to the set Zc of keys to be confirmed.

Stigmergy updates are defined by rule (LSTIG) and result in the insertion of a value in the local stigmergy of the agent. We use *tod*() to represent the timestamp (obtained from a global clock) for the new value. Since the newly inserted value must be propagated, its key is added to Zp; Zc may also be updated, like for the attribute update case. Rule (AWAIT) specifies that a guarded process $b \rightarrow P$ can only proceed if the guard b is satisfied. Notice that, if the guarded process can proceed, the stigmergy keys contained in the guard are added to the set Zc of the agent. The above transitions are labelled ε to denote they are internal to each agent, i.e. they are invisible from the point of view of the system. All agent-level rules are guarded by the condition $Zc = Zp = \emptyset$, meaning that an agent has to propagate or confirm all pending variables propagate or confirm all pending keys before continuing its execution.

$$\frac{P \overset{\surd}{\mapsto} P' \quad Zc = Zp = \emptyset}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{\varepsilon} \langle I, L, P', Zc, Zp\rangle} \text{(SKIP)}$$

$$\frac{P \overset{x \leftarrow e}{\longmapsto} P' \quad \mathcal{E}[\![e]\!](I, L) = v \quad I[x \mapsto v] = I' \quad Zc = Zp = \emptyset}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{\varepsilon} \langle I', L, P', Zc \cup \mathcal{K}[\![e]\!], Zp\rangle} \text{(ATTR)}$$

$$\frac{P \overset{x \leftsquigarrow e}{\longmapsto} P' \quad \mathcal{E}[\![e]\!](I, L) = v \quad t = tod() \quad Zc = Zp = \emptyset}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{\varepsilon} \langle I, L \oplus (x, v, t), P', Zc \cup \mathcal{K}[\![e]\!], Zp \cup \{x\}\rangle} \text{(LSTIG)}$$

$$\frac{I, L \models b \quad \langle I, L, P, Zc, Zp\rangle \xrightarrow{\varepsilon} \langle I', L', P', Zc', Zp'\rangle}{\langle I, L, b \rightarrow P, Zc, Zp\rangle \xrightarrow{\varepsilon} \langle I', L', P', Zc' \cup \mathcal{K}[\![b]\!], Zp'\rangle} \text{(AWAIT)}$$

Table 7: Semantics of agents.

On the other hand, system-level transitions formalize the handling of shared knowledge inside the virtual stigmergy and are shown in Table 8, where λ denotes a generic transition label. Rule (PAR) simply states that parallel subsystems interleave their internal actions. The symmetrical rule to (PAR) has been omitted. Rules (COMM) and (ASSOC) describe that parallel composition is commutative and associative. Rule (PROPAGATE) states that an agent can always remove a variable from Zp and propagate its value to neighbours. Rule (CONFIRM) specifies that the same can happen with Zc keys. The different nature of the messages is reflected by different transition labels (put for propagation; qry for confirmation). The (PUT) rule allows messages to spread to other agents. When a subsystem performs a $\mathsf{put}(I', L', x, v, t)$ transition, a neighbouring agent (that is, one that satisfies the predicate φ_x together with the sender) with an expired value will update its local stigmergy and add x to the keys to propagate. Notice that x is also removed from Zc, as it is assumed that the new value does not need to be confirmed anymore.

$$\frac{S \xrightarrow{\varepsilon} S'}{S \parallel T \xrightarrow{\varepsilon} S' \parallel T}\ (\textsc{par}) \qquad \frac{S_1 \parallel S_2 \xrightarrow{\lambda} S'}{S_2 \parallel S_1 \xrightarrow{\lambda} S'}\ (\textsc{comm}) \qquad \frac{(S_1 \parallel S_2) \parallel S_3 \xrightarrow{\lambda} S'}{S_1 \parallel (S_2 \parallel S_3) \xrightarrow{\lambda} S'}\ (\textsc{assoc})$$

$$\frac{x \in Zp \qquad L(x) = (v,t)}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{\mathsf{put}(I,L,x,v,t)} \langle I, L, P, Zc, Zp \setminus \{x\}\rangle}\ (\textsc{propagate})$$

$$\frac{x \in Zc \quad L(x) = (v,t)}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{\mathsf{qry}(I,L,x,v,t)} \langle I, L, P, Zc \setminus \{x\}, Zp\rangle}\ (\textsc{confirm})$$

$$\frac{S \xrightarrow{\mathsf{put}(I',L',x,v,t)} S' \quad I', L', I, L \models \varphi_x \quad L \oplus (x,v,t) \neq L}{S \parallel \langle I, L, P, Zc, Zp\rangle \xrightarrow{\mathsf{put}(I',L',x,v,t)} S' \parallel \langle I, L \oplus (x,v,t), P, Zc \setminus \{x\}, Zp \cup \{x\}\rangle}\ (\textsc{put})$$

$$\frac{S \xrightarrow{\mathsf{qry}(I',L'\,x,v,t)} S' \quad I', L', I, L \models \varphi_x \quad \mathit{time}(L,x) < t}{S \parallel \langle I, L, P, Zc, Zp\rangle \xrightarrow{\mathsf{qry}(I',L',x,v,t)} S' \parallel \langle I, L \oplus (x,v,t), P, Zc \setminus \{x\}, Zp \cup \{x\}\rangle}\ (\textsc{qry}_1)$$

$$\frac{S \xrightarrow{\mathsf{qry}(I',L',x,v,t)} S' \quad I', L', I, L \models \varphi_x \quad \mathit{time}(L,x) \geq t}{S \parallel \langle I, L, P, Zc, Zp\rangle \xrightarrow{\mathsf{qry}(I',L',x,v,t)} S' \parallel \langle I, L, P, Zc, Zp \cup \{x\}\rangle}\ (\textsc{qry}_2)$$

Table 8: Semantics of systems.

Notice that a composite system evolves by emitting the same transition label as its subsystem. This means that the rule is recursively applied until all neighbours perform their stigmergy update.

The rules for confirmation messages are quite similar, but the action of agents depend on the current state of their local stigmergy. Rule ($\textsc{qry}_1$) says that an agent with an older entry will react to a query transition $\mathsf{qry}(I', x, v, t)$ by updating its own stigmergy and propagating the value afterwards. On the other hand, an agent that has a more up-to-date value will just update Zp to propagate it, while discarding the received entry (rule $\textsc{qry}_2$).

2.4. Tuples and atomic assignments

In the rest of the paper, we will sometimes use compound assignments of the form $x_1, \ldots, x_n \leftarrow e_1, \ldots, e_n$. The agent performing such an action will execute a sequence of assignments $x_1 \leftarrow e_1; \ldots; x_n \leftarrow e_n$ with the guarantee that no other transition can happen between them. All expressions are evaluated over the initial state of the agent: therefore, their order is irrelevant. When performing a compound stigmergic assignment $x_1, \ldots, x_n \leftsquigarrow e_1, \ldots, e_n$, all values receive the same timestamp.

We further enrich our language with the notion of *stigmergy tuples*. Tuples are disjoint sets of stigmergic variables; all variables within the same tuple belong to the

same virtual stigmergy. A tuple acts as a single stigmergic variable: whenever any element of a tuple can be propagated or requested, the rest of the tuple will be as well, in an atomic fashion. Thus, these communication steps cannot be interleaved with any other action.

These design choices are driven by the need to restrict the interleaving of agents, by giving the user of the language a natural way to express operations that should always be performed together. Similarly, stigmergic tuples allow to store complex data in virtual stigmergies, with the guarantee that such data will be propagated atomically.

2.5. Clocks and verification

As stated in Section 2.3, the semantics of LAbS are based on the assumption that agents can always retrieve a (unique) timestamp from a global clock. Using a distributed clock would be a more realistic option for the actual implementation of multi-agent systems, and is the solution adopted by languages such as Buzz [1]. However, our language has an important difference in that its main focus is on formal verification, rather than execution on real or simulated platforms.

From a verification standpoint, the purpose of a clock is simply to enforce a total order on the transitions by assigning them a unique identifier (i.e., the timestamp). Using a distributed clock would unnecessarily inflate the state space with intermediate transitions for distributed time-stamping. Instead, we simply compute timestamps by relying on a global counter, which we increase each time a new transition takes place.

Please note that, due to interleaving, the above mechanism does not lose any feasible ordering of events with respect to a distributed schema. In fact, it captures all the total orders that would be possible with distributed clocks. In contrast, multiple (partial) orderings under the distributed clock can correspond to the same total order under the global clock. As a consequence, using a global clock helps to maintain a more compact state space, which is very desirable for verification.

Since we recognise that targeting real systems is a desirable goal for our language, we do plan to upgrade the SOS rules so that agents maintain a distributed clock. Even in that case, for encodings which are targeted to verification, due to the above mentioned benefits, it would still be more convenient to rely on a global clock.

3. Modeling flocking behaviour

In this section, we consider a well-known example of flocking behaviour, which can be naturally modelled by using the language features described above. The example extends the one we considered in [7] by allowing additional interaction strategies, which are modelled by multiple stigmergies. Previously, we were considering mechanisms for moving in the same direction; here we also consider moving closer and avoiding collisions. We also report some preliminary experimental results on automated verification of the described system.

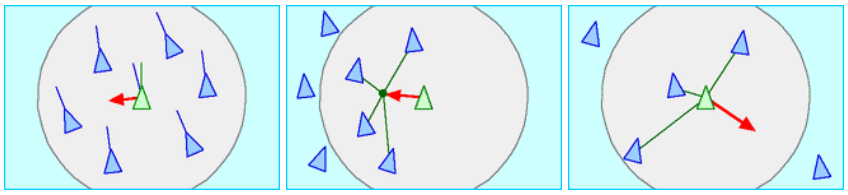
Boids (short for bird-oid objects) is the term used to refer to sets of artificial agents that aim at recreating the emergent behaviour commonly observed in flocks of birds, schools of fish, swarms of social insects, and other natural systems [12]. Their behaviour, known as *flocking*, can generally be described as «the collective coherent motion of

large numbers of self-propelled organisms» [13] and emerges from a set of simple local rules followed by the individual agents. Examples of such local rules are reported below (also see Fig. 2):

– *alignment*: head in the same direction as the other flockmates nearby

– *cohesion*: move closer to larger groups of flockmates

– *separation*: avoid collisions with flockmates nearby.



(a) Alignment (b) Cohesion (c) Separation

Figure 2: Individual rules for *Boids*. Images from Craig Reynolds' personal web page[1].

The three rules above may be applied simultaneously, subject to different ranges of action. For instance, a boid can only trigger the *separation* mechanism of another one only if the two are sufficiently close to each other: in the interaction shown in Figure 2 (c) the two boids at the corners of the arena do not play any role with respect to the boid actively performing separation. Similarly, the boid adjusting his position in Figure 2 (b) can only be influenced by the four flockmates nearby. In general, the ranges of action for the alignment, cohesion, and separation rules are represented by three different and possibly overlapping circular regions of space (Figure 3). This model of interaction was originally proposed as an efficient way to generate realistic computer animations of flocks [8] and has been very largely spread since, especially for simulation, with countless implementations available.

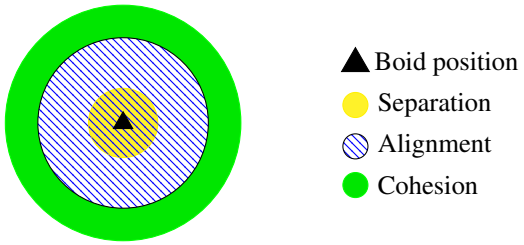


Figure 3: Regions of space where a flockmate can trigger a boid's reaction.

We now show that our language makes it possible to program the boid-like behaviour described above very naturally and concisely. The idea is to introduce appropriate

[1] `https://www.red3d.com/cwr/boids/`

stigmergic predicates that intuitively capture the three rules of Figure 2. In the rest of the section, we consider a swarm of N agents distributed on a square grid of size $G \times G$.

As a first step, we describe how to program only the *alignment* mechanism shown in Figure 2 and whose range of action corresponds to the intermediate ring of Figure 3. Such behaviour is described in Table 9 and is the same as in [7]. We store the position of an agent and its direction as a pair of attributes x, y and as a stigmergic tuple (dir_x, dir_y), respectively. We split the behavioural specifications into two processes. All the agents in the system initially execute the Init process, choosing a non-deterministic initial position and direction. Note the *generalized choice* construct to express $P[x_1/x] + P[x_2/x] + \ldots + P[x_n/x]$ as $\sum_{x \in \{x_1, x_2, \ldots, x_n\}} P(x)$.

After the initialization, every agent simply keeps moving on the arena one step at a time following his own direction and position. This continuous movement is captured by the Behavior process, which corresponds to repeatedly executing the Move process. Note that the arena wraps around: for instance, an agent at position $(x, G-1)$ that moves in direction $(0, 1)$ will reach $(x, 0)$. This explains the modulo operators in Move.

To specify the range of action of the alignment rule, we can now introduce the stigmergic predicate $\varphi_{alignment}$. The propagation of the stigmergic tuple (dir_x, dir_y) is enabled whenever the distance between two agents happens to be less or equal to a constant δ. In that case the two boids agree on the direction with the most recent timestamp (also see Figure 1).

Init $\triangleq$	$\sum_{(i,j) \in D} dir_x, dir_y \leftsquigarrow i, j; \sum_{i=0}^{G-1} \sum_{j=0}^{G-1} x, y \leftarrow i, j$
	where $D = \{(1,1), (1,-1), (-1,1), (-1,-1)\}$
Behavior $\triangleq$	Move; Behavior
Move $\triangleq$	$x, y \leftarrow x + dir_x \bmod G, y + dir_y \bmod G$
$\varphi_{alignment} \equiv$	$\|\|(x,y)_1 - (x,y)_2\|\| \leq \delta$

Table 9: A simple boid-like system implementing only the *alignment* mechanism

We can now extend our example by modelling the cohesion and separation dynamics through additional virtual stigmergies. The specifications are given in Table 10. To implement a cohesion mechanism, we need the boids to maintain some information about the *group* of agents they belong to. This information is stored within the *cohesion* stigmergy. Each group has a *leader* and zero or more *followers*. We also assume that each agent has an *id* attribute corresponding to a unique, immutable value. The leader registers its id and position in the variables *leader*, pos_x, and pos_y. The last variable in the tuple, *count*, is a counter that is repeatedly updated by the followers. In the initial state, each boid is the leader of its own group, and *count* is set to 1 (see Init function). Thanks to the link predicate $\varphi_{cohesion}$, a boid can decide to join a larger group than the one it is actually part of. When this happens, the new follower increases the group counter and moves towards its new leader, if they are far apart (Figure 4).

Even with the additional dynamics, the specifications are still very concise when

$$
\begin{array}{rl}
\textsc{Init} \triangleq & \sum_{(i,j)\in D} (dir_x, dir_y) \leftsquigarrow (i,j); \\
& \sum_{i=0}^{G-1} \sum_{j=0}^{G-1} x, y \leftarrow i, j; \\
& (leader, count, pos_x, pos_y) \leftsquigarrow id, 1, -1, -1 \\
& \text{where } D = \{(1,1),(1,-1),(-1,1),(-1,-1)\} \\
\textsc{Behavior} \triangleq & \textsc{Move} + \textsc{Separate}; \textsc{Attract}; \textsc{Behavior} \\
\textsc{Separate} \triangleq & x + dir_x \bmod G = sep_x \wedge y + dir_y \bmod G = sep_y \rightarrow \surd \\
\textsc{Move} \triangleq & x + dir_x \bmod G \neq sep_x \vee y + dir_y \bmod G \neq sep_y \rightarrow (\\
& \quad x, y \leftarrow x + dir_x \bmod G, y + dir_y \bmod G; \\
& \quad sep_x, sep_y \leftsquigarrow x, y \\
&) \\
\textsc{Attract} \triangleq & leader = id \rightarrow pos_x, pos_y \leftsquigarrow x, y \\
& + \\
& leader \neq id \rightarrow \quad count \leftsquigarrow count + 1; (\\
& \qquad\qquad |x - pos_x| \leq \delta \rightarrow \surd \\
& \qquad\qquad + \\
& \qquad\qquad |x - pos_x| > \delta \rightarrow dir_x \leftsquigarrow \frac{pos_x - x}{|pos_x - x|} \\
& \qquad); (\\
& \qquad\qquad |y - pos_y| \leq \delta \rightarrow \surd \\
& \qquad\qquad + \\
& \qquad\qquad \left|y - pos_y\right| > \delta \rightarrow dir_y \leftsquigarrow \frac{pos_y - y}{|pos_y - y|} \\
& \qquad)
\end{array}
$$

$$
\begin{array}{rll}
\varphi_{alignment} & \equiv \|(x,y)_1 - (x,y)_2\| \leq \delta & (dir_x, dir_y) \\
\varphi_{separation} & \equiv |x_1 - x_2| \leq 1 \wedge |y_1 - y_2| \leq 1 & (sep_x, sep_y) \\
\varphi_{cohesion} & \equiv count_1 \geq count_2 & (leader, count, pos_x, pos_y)
\end{array}
$$

Table 10: Specifications for a boid implementing all three mechanisms.

compared to several implementations in other general-purpose or agent-oriented languages. A comparison is reported in Table 11. Lines of code have been separated into initialization code, which sets the initial state of the system, and behavioural code, describing its evolution. The competing implementations need elaborate sequences of operations to update the agents' state, and this partially explains their higher number of lines of code. At the same time, it is worth to notice that LAbS is the only language to provide an asynchronous and distributed model of the system. In the other implementations, the state of each agent is updated at each simulation step to take into account the synchronous evolution of the states of its neighbours. This requirement about synchronization negatively affects the length and the readability of the code. In LAbS the dynamics are naturally expressed and easily identified, while in the other implementations the code related to agent-level dynamics is intermingled

Language	*Lines of code*		
	Initialization	*Behaviour*	*Total*
LAbS	2	16	18
AgentScript[2]	20	37	57
NetLogo[3] [14]	13	56	69
JavaScript[4]	10	98	108

Table 11: Comparison of several *boids* implementations.

with simulation-related computations. Furthermore, a distributed implementation of a synchronous model requires additional guarantees on the communication capabilities of agents, which are not needed under LAbS.

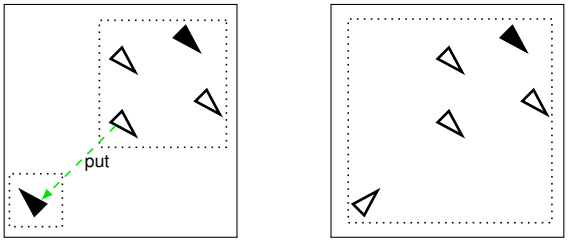


Figure 4: Our cohesion rule in action. Each dotted box contains a group of agents. Filled shapes denote leaders, while the followers are outlined. One of the followers of the larger group propagates the *cohesion* tuple $(leader, count, pos_x, pos_y)$ to the lonely leader in the bottom left, which then becomes a follower and points toward its new leader.

The purpose of the separation mechanism is to try to prevent two agents from getting too close to each other. We achieve this by letting boids communicate their position over a third virtual stigmergy containing the tuple (sep_x, sep_y). At the same time, we add guards to the Move process so that movements can only be performed if the next position of the boid is different from (sep_x, sep_y). The link predicate $\varphi_{separation}$ describes the Moore neighbourhood [15] of a boid: thus, only agents that are in adjacent locations are enabled to interact on this stigmergy (Figure 5).

Preliminary study on automated verification. In the rest of the section, we report our ongoing work towards automated verification of safety properties of LAbS systems. As a preliminary study, we focus in re-using mature existing techniques and tools for the verification of mainstream imperative languages. More specifically, we rely on a mechanized encoding of LAbS specifications as a C program that enables us to reduce safety checking of a given LAbS system to reachability analysis of a nondeterministic sequential imperative program. The advantage of this approach is that we can immediately benefit from new or improved techniques for this kind of analysis as soon as they

[2] `http://agentscript.org/models/flock.html`
[3] `http://ccl.northwestern.edu/netlogo/models/Flocking`
[4] `https://lexicalgap.com.au/playground/boids/`

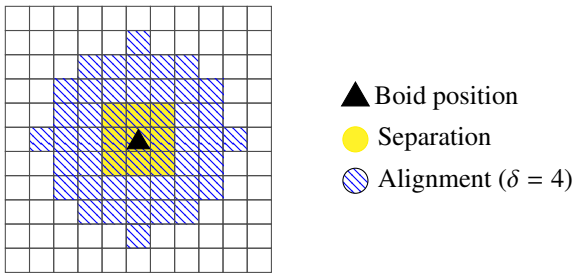


Figure 5: A visualization of the link predicates corresponding to regions in Figure 3. The *Cohesion* stigmergy is not depicted, as its predicate does not depend on distance.

are put forward.

Motivated by the apparent abundance of mature verification tools for C programs, we developed a mechanized translation procedure from LAbS specifications to C.

The idea of our translation is to encode the possible transitions of the LAbS system as separate functions in the target program, and repeatedly non-deterministically select and execute one of them to model the evolution of the system.

More specifically, in the target program, we introduce a separate function for each action occurring in the formal specifications. For example, the target program obtained from the system of Table 9 contains only one agent-level function that encodes the only action occurring within the Move sub-process. In addition, we add to the target program separate functions for system-level transitions. Such functions precisely implement the rules of Tables 7 and 8. Finally, we add the main function, which works as a *scheduler*, modelling the interleaving of the actions of the different agents and of the system, by non-deterministically invoking either a system or an agent function at a time.

During the generation of the target program, transitions (and thus their corresponding functions in the program) are associated with unique identifiers, that are used for providing the scheduler with the means for nondeterministically selecting the *enabled* action at any given step (note that there may be multiple actions enabled at any given step). A program counter keeps track of the action under execution and updated after the actual execution. This warrants the preservation of the structure of the specified processes, and disallows the execution of actions interleavings that are banned by the semantics of the language. To model these non-deterministic choices, we introduce *symbolic* variables in the target program and make suitable assumptions to restrict their values, whenever needed. Similarly, we introduce additional symbolic variables to model the non-deterministic interleaving of agent-level transitions with system-level ones, and the asynchronous operations triggered by the stigmergic interaction. External non-determinism (e.g., the position of a bird on the grid for the system of Table 9) is also modelled with non-deterministic variables. For a concrete example, we refer the reader to Appendix A where a simplified version of the encoding of Table 9 is presented.

The scheduler is the main function of the program. All the other functions encode either the behaviour for the actions of single agents actions or system-level events as mentioned above. Explicitly including the scheduler in the encoding is particularly convenient in our case. First, it allows to sequentialize the parallel composition of the

behaviour of the agents and the system transitions, thus eliminating concurrency and making it possible to use the tools for analysis of sequential programs. Second, it allows to model different scheduling variations by controlling the non-determinism in the interleaving of the individual transitions. For instance, we can impose fair interleaving of agents by considering only the executions where the scheduler lets agents perform their transitions in a round-robin fashion, as in the semi-synchronous model (SSYNC) commonly used to enforce fairness guarantees on infinite traces [16].

We intend to verify the following consensus properties for the *Boids* system of Table 10:

- *dir*: the variable (dir_x, dir_y) has the same value in all local stigmergies
- *leader*: all agents have the same leader.

We checked the above two properties for several variants of the system under consideration, experimenting with different kinds of techniques for the analysis of the corresponding C program. We tried SAT and SMT-based bounded model checking for under-approximate analysis using the already mentioned CBMC, 2LS [17], and the SMT-based tool ESBMC [18]. We also tried plain abstract interpretation with IKOS [19] using interval or gauge domains [20]. We tried SMACK [21] with its Corral backend [22] to experiment with abstraction refinement, and CPAchecker [23, 24] for explicit-value analysis based on counterexample-guided abstraction refinement as well as predicate abstraction. We also tried out 2LS to experiment with function-modular interprocedural analysis.

With the current encoding, we found under-approximation (i.e. bounding the number of iterations in the scheduler) to be the only way to obtain informative results from the analysis of any of the considered systems. In particular, the bounded model checkers CBMC, 2LS, and ESBMC provide us with correct results, with CBMC and 2LS achieving similar performances and ESBMC being slower. As for the other tools, we obtained inconclusive analyses with IKOS, that detects a potential assertion violations both on safe and unsafe instances; false positives with SMACK; non-terminating analysis with 2LS in function-modular interprocedural mode and with CPAchecker in both the considered configurations.

All tests were executed on an otherwise idle 64-bit GNU/Linux workstation with kernel 4.9.95, equipped with 128GB of physical memory and a dual 3.10GHz Xeon E5-2687W 8-core processor. Our experimental tool that automatically translates LAbS specifications into C programs is called SLiVER (Symbolic LAbS Verifier) is available at `https://github.com/labs-lang/sliver/releases/latest`.

The program variants, the exploration parameters, and verification results are reported in Table 12. The first column indicates the analyzed variant of the system of Table 10. The letters A, S, C are used to indicate the employed interaction mechanism, respectively Alignment, Separation, Cohesion. The letter *W* is used to indicate that the arena wraps around. Wrap-around movements can be avoided by replacing the Move process with the following one:

$$\begin{aligned}\textsc{Move}' \triangleq\ & (x + dir_x, y + dir_y) \neq (x + dir_x \bmod G, y + dir_y \bmod G) \rightarrow \surd \\ & + \\ & (x + dir_x, y + dir_y) = (x + dir_x \bmod G, y + dir_y \bmod G) \rightarrow \textsc{Move}\end{aligned}$$

The additional guard clauses prevent the agents from moving when $(x, y) + (dir_x, dir_y)$ does not correspond to a valid grid position. The other columns of the table, left to right, indicate the overall number of agents in the system, the number B of iterations of the scheduler, the threshold δ used in the link predicates, the property, the verification outcome, and the verification time in seconds. We report the best analysis performance we could obtain using using any of the considered symbolic model checkers. We set the size G of the arena to 10 for all instances. Please note that, within the reported bounds, the verification is *exhaustive*: the property under consideration is checked for every possible initial position and direction of the boids on the grid, and for every possible interleaving of their actions.

Each system parameter affects the complexity of the verification task in a different way. Enabling all interaction mechanisms increases the complexity of individual behaviour and the frequency of interaction over the stigmergies, resulting in a bigger state space. The size of the arena G and the number of agents influence the number of feasible initial states. A small value of δ makes stigmergic interaction less likely to happen: therefore, agents might not easily reach a consensus on the direction. Finally, without wrap-around movement a boid can get stuck on the border of the arena. This can in turn affect the number of transitions required for consensus.

When verifying *AW* instances, we found that for a 3-agent system and a number of steps $B = 12$ the smallest value of δ for which the system satisfies property *dir* is 13. However, by adding only one more agent the property would no longer hold within the same parameters: to find a safe 4-agent system we have to increase B to 19. With yet one more agent, the smallest value of B for which *dir* holds grows to 27. This difference is due to the fact that larger systems contain traces where stigmergic interaction among all agents does not happen within a smaller number of steps. We have experienced that the performance of the verifier becomes noticeably worse as the value of B gets closer to the one for which the property under consideration holds. With 5 agents, verifying a safe system takes over 15 hours.

The complexity of the individual behaviour also affects the performance of our analysis, especially in safe systems. In fact, we checked the *leader* property against two *AC* instances containing 3 agents, with parameters $G = 10$, $\delta = 10$, where B was set respectively to 23 and 24 for the unsafe and safe instance. Verification of the safe instance required about 22 hours; by comparison, the analysis of the unsafe system terminated in less than 30 minutes. If we allow agents to perform wrap-around movements, then the smallest value of B such that the property *leader* holds is 27, confirming our previous claim about the effect that this variation has on the collective behaviour of the system. However, despite the lower bound, the time needed to verify the safe *AC* instance is significantly longer than that required for the safe *ACW* one, since the Move′ process is more complex than Move. Finally, we analyzed additional instances with a 16×16 arena (Table 13). Even though the increased size of the arena

Variant	*Agents*	*B*	δ	*Property*	*Result*	*Time (s)*
A W	3	12	12	*dir*	Fail	14
A W	3	12	13	*dir*	Pass	153
A W	4	18	13	*dir*	Fail	637
A W	4	19	13	*dir*	Pass	1720
A W	5	26	13	*dir*	Fail	18315
A W	5	27	13	*dir*	Pass	55653
AC	3	23	10	*leader*	Fail	1640
AC	3	24	10	*leader*	Pass	79866
AC W	3	24	10	*leader*	Fail	531
AC W	3	26	10	*leader*	Fail	19337
AC W	3	27	10	*leader*	Pass	62646

Table 12: Automated analysis of *Boids* on a 10×10 arena.

Variant	*Agents*	*B*	δ	*Property*	*Result*	*Time (s)*
A W	3	26	21	*dir*	Fail	1350
AC	3	26	21	*leader*	Fail	1649
AC	3	27	21	*leader*	Pass	108935

Table 13: Automated analysis of *Boids* on a 16×16 arena.

leads to a much larger set of initial states, in the case of unsafe instances the analysis performance was not severely affected.

It is worth to notice that to generate the C programs our technique deliberately avoids using features that are well-known sources of complexity for program analysis, such as dynamic memory allocation, pointers, concurrency, etc. Nevertheless, the state space is remarkably large due to the symbolic variables introduced to simulate the interleavings, the internal and external non-determinism, and finally the asynchronous operation of the system-level transitions. Thus, under-approximation approaches have to explore a large set of states even for small bounds (which can make the analysis resource-intensive). On the other hand, over-approximation turns out to be either inconclusive or hardly terminating, due to the very large number of spurious traces introduced by abstraction. Automated analysis of programs of this kind seems, in fact, still a challenge.

4. Modeling the environment

Agents are mobile entities that are *situated* and operate in a physical environment. This agent-environment interaction is a fundamental feature of many real-world scenarios [25, 5]. Foraging, i.e. the task of collecting items that are scattered through an arena, is one such scenario. This kind of interaction enjoys some specific properties that are difficult to express with the constructs introduced in Section 2. In this section we extend the language to support a shared-memory abstraction of the environment; we then show how the extended language can be used to easily model some additional examples, by using the environment as a medium of interaction between agents or to model external objects.

In the initial version of our language [7], expressions and guards could only refer to attributes or stigmergic variables. Therefore, to access an environment variable the user of the language had to introduce a *read* operation to copy its value into an attribute. By contrast, we now allow environment variables to occur in expressions and guards, and we update the formal semantics of situated systems so that agents can atomically perform read and write operations on the environment. These changes lead to a more intuitive syntax and make it easier to reason about the interaction between the agents and the environment.

4.1. Semantics of situated systems

Assuming that there is a set $\mathcal{K}_E$ disjoint from $\mathcal{K}_I$ and $\mathcal{K}_L$, we define an *environment* to be a partial function from $\mathcal{K}_E$ to the set of values $\mathcal{V}$. A *situated system* is a pair (E, S) where E is an environment and S is a LAbS system. We now allow expressions to also contain identifiers from $\mathcal{K}_E$. This requires introducing a new semantic function, which extends the one in Table 4 as follows:

$$
\begin{aligned}
\mathcal{E}_2[\![\cdot]\!] &: \mathit{Expr} \longrightarrow \mathcal{E}nv \to \mathcal{I} \to \mathcal{L} \hookrightarrow \mathcal{V} \\
\mathcal{E}_2[\![v]\!] &= \lambda E.\lambda I.\lambda L.v \\
\mathcal{E}_2[\![x]\!] &= \begin{cases} \lambda E.\mathcal{E}_1[\![x]\!] & \text{if } x \in \mathcal{K}_E \\ \lambda E.\mathcal{E}[\![x]\!] & \text{otherwise} \end{cases} \\
&\qquad \text{where } \mathcal{E}_1[\![x]\!] = \lambda I.\lambda L.E(x) \\
\mathcal{E}_2[\![e_1 \diamond e_2]\!] &= \lambda E.\lambda I.\lambda L.\mathcal{E}_2[\![e_1]\!](E,I,L) \diamond \mathcal{E}_2[\![e_2]\!](E,I,L)
\end{aligned}
$$

In the definition above we have used $\mathcal{E}nv$ to denote the set of all environments. Agents are now able to perform an additional basic action to store the result of an expression into an environment variable. We denote this action by $x \Leftarrow e$ (Table 14). Since any expression may now potentially refer to environmental variables, their evaluation can no longer be done at the individual level. We therefore revise the semantics of agents and add a transition label $e \triangleright x$ denoting the willingness of an agent to assign the value of expression e to variable x (Table 15). Note that we omit Rule (SKIP) as it is the same as in Table 7.

To define the semantics of situated systems, in Table 16 we introduce an unlabeled transition relation ($\rightarrowtail$). As mentioned above, the evaluation of expressions and the assignment to the relevant store of variables is described by rules ($\text{EVAL}_{\text{I,L,E}}$). Rule (AWAIT) has also been removed from agent-level rules, as guards may now also refer to environmental variables.

Rule (MSG) simply states that the actions related to stigmergic communications only affect the system and leave the environment unchanged. Finally, rule (PAR_{E}), which is commutative, states that the parallel composition of two systems affects the environment in an interleaved fashion. The rule symmetrical to (PAR_{E}) is omitted

4.2. Case studies

In this section, we present a selection of situated systems in LAbS. We exploit the new operational semantics of the language to improve our specifications for *foraging*

$$\alpha ::= \quad x \leftarrow e \mid x \leftsquigarrow e \mid x \Leftarrow e \mid \surd$$

Table 14: Basic processes in situated systems.

$$\frac{P \overset{x \leftarrow e}{\longmapsto} P' \quad Zc = Zp = \emptyset}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{e \triangleright x} \langle I, L, P', Zc \cup \mathcal{K}[\![e]\!], Zp\rangle} \text{ (ATTR)}$$

$$\frac{P \overset{x \leftsquigarrow e}{\longmapsto} P' \quad Zc = Zp = \emptyset}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{e \triangleright x} \langle I, L, P', Zc \cup \mathcal{K}[\![e]\!], Zp\rangle} \text{ (LSTIG)}$$

$$\frac{P \overset{x \Leftarrow e}{\longmapsto} P' \quad Zc = Zp = \emptyset}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{e \triangleright x} \langle I, L, P', Zc \cup \mathcal{K}[\![e]\!], Zp\rangle} \text{ (ENV)}$$

Table 15: Semantics of agents in a situated system.

(Sect. 4.2.1), a very popular case study in the multi-robots literature. We also present improved versions of *opinion formation* protocols (Sect. 4.2.2), namely a basic *voter model* [26], and then a more general one based on *k-unanimity* [27]. Finally, we present the LAbS specifications of two *population protocols* [9] to demonstrate the use of the environment (Sect. 4.2.3).

4.2.1. Foraging

Foraging is a popular case study in distributed robotics, as it can model many other scenarios, such as *waste retrieval* and *search and rescue* [28]. In this scenario, a swarm of robots explores the arena with the goal of finding and collecting items. We can store these items in the environment and interact with them through the primitives introduced in Section 4. Specifications are shown in Table 17. The robots perform a random walk to explore the arena. Like in the flocking case study, we assume the arena is a torus and that D is the set of directions a robot can take. The process Step models a single step in a random walk.

Now suppose that there are m items and that the environment variables x_i, y_i contain the position of the i-th item. If the item has been collected, these variables are instead set to -1. Then, a robot can check if it has found the i-th item, and possibly collect it, by executing the process Check_i.

We denote by $\prod_{x_i \in X} P(x)$ the parallel composition of all processes in the form $P[x_i/x]$: the behaviour that checks for the presence of a generic item is a parallel composition of one Check_i process for each item. Thus, the foraging behaviour is just a recursive sequence of steps and checks.

4.2.2. Opinion formation

Opinion formation is a class of protocols where each agent starts with a certain *opinion* taken from a set of *options* $O = \{1, 2, \ldots, m\}$, and may dynamically change,

$$\frac{a \xrightarrow{e \triangleright x} \langle I, L, P, Zc, Zp\rangle \quad \mathcal{E}_2[\![e]\!](E, I, L) = v \neq \bot \quad x \in \mathcal{K}_I}{(E, a) \rightarrowtail (E, \langle I[x \mapsto v], L, P, Zc, Zp\rangle)} \ (\text{EVAL}_\text{I})$$

$$\frac{a \xrightarrow{e \triangleright x} \langle I, L, P, Zc, Zp\rangle \quad \mathcal{E}_2[\![e]\!](E, I, L) = v \neq \bot \quad x \in \mathcal{K}_L}{(E, a) \rightarrowtail (E, \langle I, L \oplus (x, v, tod()), P, Zc, Zp \cup \{x\}\rangle)} \ (\text{EVAL}_\text{L})$$

$$\frac{a \xrightarrow{e \triangleright x} \langle I, L, P, Zc, Zp\rangle \quad \mathcal{E}_2[\![e]\!](E, I, L) = v \neq \bot \quad x \in \mathcal{K}_E}{(E, a) \rightarrowtail (E[x \mapsto v], \langle I, L, P, Zc, Zp\rangle)} \ (\text{EVAL}_\text{E})$$

$$\frac{(E, \langle I, L, P, Zc, Zp\rangle) \rightarrowtail (E', \langle I', L', P', Zc', Zp'\rangle) \quad E, I, L \models b}{(E, \langle I, L, b \rightarrow P, Zc, Zp\rangle) \rightarrowtail (E', \langle I', L', P', Zc' \cup \mathcal{K}[\![b]\!], Zp'\rangle)} \ (\text{AWAIT})$$

$$\frac{S \xrightarrow{\mu(I,x,v,t)} S'}{(E, S) \rightarrowtail (E, S')} \ (\text{MSG}) \qquad \frac{(E, S) \rightarrowtail (E', S')}{(E, S \| T) \rightarrowtail (E', S' \| T)} \ (\text{PAR}_\text{E})$$

Table 16: Semantics of situated systems. a denotes a generic LAbS agent.

subject to certain conditions. The voter model is an elementary such protocol, where agents can inspect the opinion of a random neighbour and imitate it [26]. Stigmergies are not suitable in this scenario, as opinions would only propagate according to their attached timestamps. Opinion formation protocols, on the other hand, might have additional requirements, such as encouraging the spread of the option initially held by a majority of agents.

Table 18 shows a LAbS encoding of a voter model supported by the environment primitives introduced in Section 4. An agent can either *talk*, by writing the value of its attribute *opinion* to the environmental variable *env*, or *listen*, by doing the reverse. We could similarly model more complex protocols, such as the k-unanimity rule. In this case, an agent only changes its opinion to some option if it perceives that k other agents agree on that option [27]. We can use k environment variables to store opinions. We use a generalized choice to describe that, whenever an agent talks, it writes its own opinion into one of such variables.

In these examples, every agent can talk and listen to any other agent. This is not the case in most research on opinion formation, where agents are placed on a graph and can only communicate with their neighbours. However, we can model these scenarios by introducing more environment variables and letting agents interact through different subsets of the environment.

4.2.3. Population protocols

Population protocols are a model of distributed computing based on anonymous, mobile agents with uniform behaviour that may change their state through pairwise interactions [9]. The allowed interactions are encoded as a relation over pairs of states.

$$
\begin{aligned}
\textsc{Init} &\triangleq \sum_{i=0}^{G-1}\sum_{j=0}^{G-1} x, y \leftarrow i, j \\
\textsc{Behavior} &\triangleq \textsc{Step}; \prod_{i=1}^{m} \textsc{Check}_i; \textsc{Behavior} \\
\textsc{Step} &\triangleq \sum_{(i,j)\in D} x, y \leftarrow x + i, y + j \\
\textsc{Check}_i &\triangleq (x, y) = (x_i, y_i) \rightarrow x_i, y_i \Leftarrow -1, -1 \\
&\quad + \\
&\quad (x, y) \neq (x_i, y_i) \rightarrow \surd
\end{aligned}
$$

Table 17: Specifications for the *Foraging* example.

$$
\begin{aligned}
\textsc{Init} &\triangleq \sum_{i\in O} opinion \leftarrow i \\
\textsc{Behavior} &\triangleq \textsc{Talk} \mid \textsc{Listen} \\
\textsc{Listen} &\triangleq opinion \neq env \rightarrow opinion \leftarrow env; \textsc{Listen} \\
\textsc{Talk} &\triangleq env \Leftarrow opinion; \textsc{Talk}
\end{aligned}
$$

Table 18: Specifications for the *voter model* example.

An element $((p, q), (p', q'))$ of the relation, typically denoted as $(p, q) \mapsto (p', q')$, is called a *transition* and means that a pair of agents with states p and q can interact and evolve, respectively, to states p' and q'. The first element of the pair is called the *initiator* of the interaction, while the other is called the *responder*. Transition relations may be *asymmetric*, i.e. a pair of agents may evolve in different ways depending on which one initiates the interaction.

Given a set Q of individual states, a *configuration* is a multiset over Q, which expresses how many agents are in a specific state. Systems defined by population protocols never terminate: rather, they are said to *stabilize* when they reach a configuration that cannot be changed by further transitions. With the fairness assumption that any configuration which is reachable infinitely often is eventually reached, some population protocols always stabilize to a configuration which satisfies a given predicate over its initial configuration. In this case, we say that the protocol *computes* the predicate.

In this section, we focus on a subset of population protocols, known as *majority protocols*. Their initial states correspond to opinions of agents, and the goal of the protocol is to determine the most popular opinion. These protocols are considered correct if they stabilize to a configuration in agreement with the initial majority opinion.

Approximate majority protocol. Let us first examine a 3-state protocol [29] where all the agents are initially either in state Y or N and evolve according to the following rules:

$$YN \mapsto Yb \quad (1) \qquad Yb \mapsto YY \quad (2) \qquad NY \mapsto Nb \quad (3) \qquad Nb \mapsto NN \quad (4).$$

$$
\begin{aligned}
\textsc{Init} &\triangleq \sum_{i \in O} opinion \leftarrow i \\
\textsc{Behavior} &\triangleq \textsc{Talk}_k \mid \textsc{Listen}_k \\
\textsc{Listen}_k &\triangleq (unanimity \rightarrow opinion \leftarrow env_1 \\
&\quad + \\
&\quad \neg unanimity \rightarrow \surd); \textsc{Listen}_k \\
\textsc{Talk}_k &\triangleq \sum_{i=1}^{k} env_i \Leftarrow opinion; \textsc{Talk}_k \\
\\
\text{where } unanimity &\equiv \bigwedge_{i=2}^{k} env_1 = env_i
\end{aligned}
$$

Table 19: Specifications for the *k-unanimity rule* example.

Rules (1) and (3) state that an agent can reach a *blank* state, b, after meeting another agent with a different opinion. Rules (2) and (4) say that Y- and N-agents can convert blank agents to their opinion. Notice that this protocol is *one-way*, i.e. all transitions only affect the state of the responder. To model this protocol, we exploit the environment as a medium to model pairwise communication. Namely, an agent may *initiate* an interaction by writing its own id and state to the environment, or it can wait until another agent does that, and then *respond* by updating the state of its interaction partners accordingly (Table 20).

$$
\begin{aligned}
\textsc{Init} &\triangleq state \leftarrow \text{“}Y\text{”} + state \leftarrow \text{“}N\text{”} \\
\textsc{Init}_E &\triangleq agent, message \Leftarrow -1, -1 \\
\textsc{Behavior} &\triangleq \textsc{Initiate} + \textsc{Respond}; \textsc{Behavior} \\
\textsc{Initiate} &\triangleq state \neq \text{“}b\text{”} \rightarrow agent, message \Leftarrow id, state \\
\textsc{Respond} &\triangleq agent \neq id \rightarrow \quad message = \text{“}Y\text{”} \wedge state = \text{“}N\text{”} \rightarrow state \leftarrow \text{“}b\text{”} \\
&\qquad\qquad + \quad message = \text{“}Y\text{”} \wedge state = \text{“}b\text{”} \rightarrow state \leftarrow \text{“}Y\text{”} \\
&\qquad\qquad + \quad message = \text{“}N\text{”} \wedge state = \text{“}Y\text{”} \rightarrow state \leftarrow \text{“}b\text{”} \\
&\qquad\qquad + \quad message = \text{“}N\text{”} \wedge state = \text{“}b\text{”} \rightarrow state \leftarrow \text{“}N\text{”}
\end{aligned}
$$

Table 20: Specifications for the *approximate majority* protocol.

This protocol stabilizes with a high probability to a configuration where all agents are in the state that had the majority in the initial configuration. However, it is known that in some cases it will not compute the majority correctly: the system can reach the consensus on the opinion that was initially held by a minority of agents. Existing tools, such as Peregrine [30], can automatically prove that the protocol is incorrect; in Section 4.3 we use SLiVER to confirm these findings.

Majority protocol. This protocol [10] is a 4-state protocol with $Q = \{Y, N, y, n\}$ representing the full set of individual states and such that all agents are initially either in state Y or in state N. The transition relation is:

$$YN \mapsto yn \quad (1) \qquad Yn \mapsto Yy \quad (2) \qquad Ny \mapsto Nn \quad (3) \qquad yn \mapsto yy \quad (4).$$

It can be proved that this protocol successfully computes the majority. To be more precise, if the initial number of Y-agents is greater than or equal to the number of N-agents, the system stabilizes to a configuration that only contains Y- and y-agents. Otherwise, it stabilizes to a configuration that contains only N's and n's. To understand why we explain the transition relation in detail.

Rule (1) gradually removes pairs of Y- and N-agents from the system. Therefore, if the Y's have the initial majority then no N will eventually remain, and vice versa. Notice that, if the initial configuration contains the same number of Y's and N's, both states will disappear from the system. Rule (2) says that a Y-agent can turn an n into a y, while rule (3) says that N-agents can do the opposite. Finally, the fourth rule states that y's can also convert n's to the y state. Notice that n-agents do not have the opposite capability. This difference acts as a tie-breaker: a system that is initially tied will eventually reach a consensus on y.

The specification for this protocol is slightly more complex than the previous one, as the first rule ($YN \mapsto yn$) affects the state of both the initiator and the responder. We encode this by letting the responder store its state within an environment variable: the initiator will check this variable and update its own state accordingly. To preserve the semantics of population protocols and avoid unwanted interactions, we introduce a *lock* variable that every agent has to check before performing an action. Notice our language does not require specific primitives for locks, thanks to the possibility of performing multiple atomic assignments and the atomicity of guarded processes.

The full LAbS specification of the majority protocol is reported in Table 21. When *lock* is set to 0, agents can only initiate a transition. On the other hand, *lock* = 1 means that some agent has already initiated a transition and other agents are free to respond. In three cases, agents only need to update their state and reset the *agent* and *lock* variables to allow a new transition to take place. In the case of a YN-transition, however, the responder also stores N in the environment and signals the initiator that they must update their state by setting *lock* to 2. When this happens, the initiator is the only agent allowed to perform an action, namely by executing the Acknowledge process and changing its own state to y before increasing *lock* to 3. This signals the responder that the transition has been performed and it can thus safely reset the environment variables to enable new transitions. Notice that the overall structure presented in Table 21 can be adapted to other population protocols, potentially with a higher number of states and transitions.

4.3. Verification of population protocols

In this section we use SLiVER to verify that the *approximate majority* protocol of Table 20 is not correct. In fact, for some system evolutions, the agents reach a consensus on the minority opinion. To prove that, we create an instance where Y-agents are initially a minority and we consider the following safety property:

– *noYconsensus*: There is at least one agent whose state is not Y.

$$
\begin{array}{rl}
\textsc{Init} \triangleq & state \leftarrow \text{“}Y\text{”} + state \leftarrow \text{“}N\text{”} \\
\textsc{Init}_E \triangleq & agent, message, responder, lock \Leftarrow -1, -1, -1, 0 \\
\textsc{Behavior} \triangleq & (\ \ agent = id \rightarrow \textsc{Acknowledge} \\
& \ + agent \neq id \rightarrow (\textsc{Initiate} + \textsc{Respond})); \\
& \textsc{Behavior} \\
\textsc{Initiate} \triangleq & lock = 0 \wedge state \neq \text{“}n\text{”} \rightarrow agent, message, lock \Leftarrow id, state, 1 \\
\textsc{Respond} \triangleq & lock = 1 \rightarrow \\
& \quad (message = \text{“}Y\text{”} \wedge state = \text{“}N\text{”} \rightarrow \\
& \quad\ \ lock, responder \Leftarrow 2, \text{“}N\text{”}; \\
& \quad\ \ state \leftarrow \text{“}n\text{”}; \\
& \quad\ \ lock = 3 \rightarrow agent, lock, responder \Leftarrow -1, 0, -1) \\
& \quad + (\ (\ \ \ message = \text{“}Y\text{”} \wedge state = \text{“}n\text{”} \rightarrow state \leftarrow \text{“}y\text{”} \\
& \qquad\ + \ \ message = \text{“}N\text{”} \wedge state = \text{“}y\text{”} \rightarrow state \leftarrow \text{“}n\text{”} \\
& \qquad\ + \ \ message = \text{“}N\text{”} \wedge state = \text{“}y\text{”} \rightarrow state \leftarrow \text{“}n\text{”}); \\
& \qquad\ agent, lock \Leftarrow -1, 0) \\
\textsc{Acknowledge} \triangleq & lock = 2 \wedge state = \text{“}Y\text{”} \wedge responder = \text{“}N\text{”} \rightarrow \\
& \quad state \leftarrow \text{“}y\text{”}; lock \Leftarrow 3
\end{array}
$$

Table 21: Specifications for the *majority* protocol.

This is equivalent to checking that the consensus on the Y opinion is unreachable. We can check the above property on all states that are reachable after performing a given number B of system evolutions. If we find a counterexample, we can use it as proof that the protocol does not compute a majority in the general case.

Furthermore, we can also check whether the majority protocol of Table 21, instantiated with similar parameters, computes the wrong majority. Since this protocol relies on a different set of individual states, we slightly refine the property to verify as follows:

– *noYconsensus′*: There is at least one agent whose state is either N or n.

Table 22a reports our verification results for the *approximate majority* protocol, while Table 22b contains results for the *majority* protocol. In both tables we report the number of $Y-$ and N-agents in the initial configuration, the bound B on the number of transitions, and the verification result with the corresponding evaluation time. The experiments were performed on the same machine described in Section 3. Notice that we did not enforce any constraint on the scheduler: the agents freely interleave their executions. The considered instances of the *approximate majority* protocol fail to satisfy the property, meaning that there is at least one trace where the agents reach a consensus on Y: this is enough for us to conclude that the protocol does not always compute the majority correctly. On the other hand, all the considered instances of the *majority* protocol where Y does not have the initial majority satisfy the *noYconsensus′* property. These results are not sufficient to prove the correctness of the protocol, but they do show that the minority opinion does not spread among all the agents. Notice that we also verified two instances of the *majority* protocol with a majority of Y-agents. As we

Agents				
Y	*N*	*B*	*Result*	*Time (s)*
2	3	20	Fail	5
2	3	30	Fail	9
4	6	20	Fail	9
4	6	30	Fail	15

(a) Verification of property *noYconsensus* on instances of the *approximate majority* protocol.

Agents				
Y	*N*	*B*	*Result*	*Time (s)*
2	3	20	Pass	45
2	3	30	Pass	8498
4	6	20	Pass	72
4	6	30	Pass	12718
3	2	20	Fail	13
6	4	30	Fail	629

(b) Verification of property *noYconsensus*$'$ on instances of the *majority* protocol.

Table 22: Verification results for the population protocols of Section 4.2.3.

expected, SLiVER found that both instances failed to satisfy *noYconsensus*$'$, meaning that there exists at least one trace where the majority is computed correctly. We also noticed that the performance of the bounded model checker on safe instances is strongly affected by the number of considered agents and by the verification bound.

5. Conclusion, and Related and Future Work

We have introduced LAbS, a core language for multi-agent systems. The key feature of our language is a distributed, decentralized data structure to model inter-agent communication, or knowledge propagation. This data structure is based on the concept of virtual stigmergy. LAbS also offers the possibility of modelling the external environment, i.e. a shared data repository accessible by the different agents.

In LAbS, agents can directly access only their local copy of the stigmergy; however, local changes are transparently propagated through the system, thus enabling asynchronous and indirect agent interaction. Rather than restricting the message exchange to the spatial neighborhood, the language provides a flexible communication mechanism based on the attributes of the agents. This makes our language appropriate to describe and reason about different kinds of distributed systems. Also, thanks to the possibility of introducing multiple stigmergies, our language can model different kinds of communication that can be selected according to specific states of the system or to given environmental conditions. In the case of multi-robot systems, robots equipped with multiple sensors can be naturally modelled.

To assess the quality of the main features of the language and its expressive power, we have used LAbS to model some of the classical case studies considered by the multi-agents research community, such as *Boids* and gossiping protocols. We have automatically analyzed emerging properties of such case studies by relying on a mechanisable symbolic translation of LAbS code into imperative programs that allows to reuse general-purpose verification techniques for mainstream languages.

The actual design choices were driven by the analysis of different languages available in the multi-robot and multi-agent literature [6]. In fact, many other research groups have worked on the problems we have been considering. Below we briefly describe some of the most relevant approaches.

Most work in swarm robotics and multi-agent systems research follow a bottom-up approach, also known as behaviour-based design [28, 31]: designers iteratively alter the behaviour of individual agents and check whether the desired properties emerge on a global level. The ability to obtain quick feedback is essential at design time. In fact, many design methodologies for multi-agent systems heavily rely on simulation [32]. This preference is also reflected by the large amount of simulation platforms in the literature [33, 34, 35, 36, 37]. Since the design of each simulator results in different tradeoffs, there is no single best tool for all classes of systems [38]. A drawback of the simulation-based approaches to system validation is the limited coverage of the state space. On the other hand, the available agent-oriented formal analysis techniques make different restrictions on the used languages, e.g. do not support value-passing [39], or are limited to very specific classes of systems [30]. In contrast, our end-to-end technique supports systematic analysis without limiting the expressivity of the language.

A great deal of research provides ad-hoc solutions to specific tasks. In the case of swarm robotics, these can be grouped in a small number of broad categories such as robot aggregation, flocking, object foraging, construction, and swarm deployment (e.g. surveillance, distributed sensing, or signal relaying) [40]. Higher-level languages could help research by providing adequate primitives that can be combined to solve many of the tasks described above. Higher-level formalisms can also ease the design process by expressing individual behaviour in an intuitive way while avoiding ambiguity [41]. Buzz [1] is an example of a swarm-oriented language which follows object-oriented principles and is based upon communication between neighbours, team management, and consensus achievement. Another example is Proto/Protoswarm [42, 43], a functional language where individual agents are seen as part of a virtual *spatial computer*. Formalisations of the semantics of spatial computers have been proposed, to make it easier to analyse and predict the behaviour of this class of systems [44]. However, to the best of our knowledge, no formal verification tool for such a paradigm has been developed yet.

Translations from high-level to lower-level languages have also been proposed elsewhere. AgentSpeak [45] automatically generates Promela and Java programs that can be analyzed with Spin and Java Pathfinder [46]. An AgentSpeak-to-C translation is also available, which makes it possible to simulate the agent's behaviour and execute it on real hardware [47]. As agent interaction in AgentSpeak relies on a limited number of point-to-point primitives, implementing and verifying higher-level mechanisms such as attribute-based communication or virtual stigmergies would require a substantial effort.

Platforms and languages that provide an explicit notion of an environment include ISPL [39], the PALPS process calculus [48], and the JaCaMo framework [49], which is based on a variant of the AgentSpeak language called Jason [50]. Alternative approaches to support the system environment introduce distinct agents to collect and distribute data that model the behaviour of the environment the collective set of agents operate in. This is the case, for instance, of the PARS process algebra [51] and of the SCEL language [52].

Multi-agent systems can be seen as part of the larger fields of agent-based complex systems (ACS) [53] and collective adaptive systems (CAS) [54], which are related to areas as diverse as ecology and economics [55]. Languages and process algebras that have been used to model and analyse ACS and CAS include NetLogo [14], PEPA [56],

and CARMA [57]. Among these, CARMA is the only process algebra supporting attribute-based communication; however, an agent can only receive an (attribute-based) message via an explicit, blocking input primitive. By contrast, in LAbS the reception of a stigmergic message is only constrained by link predicates.

Verification of multi-robot systems has reached general results for a simple model (look-compute-move) where robots have identical behaviour and repeatedly perform three actions: store the position of other robots in their local memory; decide if and where it should move; and finally apply the decision. Depending on the synchronicity of these steps and on the shape of the arena, possibility or impossibility results have been proved for tasks related to pattern formation [58] or consensus achievement in the presence of Byzantine robots [59]. However, the actual requirements on the behaviour of robots make it difficult to extend these results to heterogeneous systems.

The Peregrine tool [30] can verify and simulate population protocols. This model of computation is well-suited to analyze chemical reaction networks as well as opinion formation protocols, but it cannot easily describe interaction patterns other than point-to-point communication. Peregrine also supports parametric verification, and can therefore check the correctness of a protocol (such as the example of Section 4.2.3) for an unbounded number of agents. However, Peregrine can only verify whether a protocol can compute a given predicate, while encoding the same protocol in LAbS enables us to verify arbitrary safety properties.

We would now like to conclude by mentioning some possible directions for future work. In the near future we plan to investigate the need of additional primitives to describe more complex individual behaviour, such as alternative models of communication between agents. Adding other constructs to the language, such as a macro system or the capability to include code from other files, will enable reuse and modularity of the specifications. This would also allow us to develop standard libraries of behaviours and patterns (e.g. for link predicates) commonly observed in the multi-agent literature, and in general simplify the specification of complex systems. We believe that most of these new functionalities may be added without revising the formal semantics of the language, and we plan to implement them in the next revision of our verification framework.

We acknowledge that the semantics of the language can be improved by removing the global clock and instead relying on distributed solutions such as Lamport timestamps [11]. Other directions of research might include the study of logical formalisms to conveniently describe key properties of LAbS systems, and the implementation of mechanised translations towards other languages to automatically verifying such properties.

It is worth observing that the experimental evaluation presented in this paper was not aiming at showing the efficiency of the specific verification technique but rather the feasibility of our end-to-end approach. In particular, the experiment with *Boids* has shown that our language can naturally model different kinds of complex collective behaviour, that are hard to express with other formalisms, and that automated analysis, relying on standard software verification tools after a mechanised translation, is indeed feasible, even though still with relatively small parameters.

One of the sources of complexity which makes analysis particularly demanding is the amount of non-determinism induced by the scheduling choices (also see Appendix A). In the target program, the scheduler models an execution step of the system under

analysis in correspondence of each executable action, by invoking the function that encodes that action. In our translation, for the sake of simplicity, this is accomplished by non-deterministically selecting any action regardless of the process structure, and then disallowing (via appropriate assumption statements) the alternatives leading to traces that are not permitted by the semantics of the language. The initial eagerness affects performance negatively, but can be limited by modifying the scheduler to take into account the syntactic structure of the encoded process. However, this would require a significantly more involved encoding, which we leave for future work.

Since reusing off-the-shelf verification tools for imperative programs has so far turned out to be useful only with relatively simple safety properties and under-approximation (i.e., bounded model checking), it would be interesting to consider techniques that would allow proving more complex properties, such as those supported by LTL or similar temporal logics, that require reasoning about an unbounded number of system evolutions. In the direction of richer formalisms, we plan to experiment with other verification frameworks such as LTSmin [60], which features both explicit-state and BDD-based exploration. With respect to unbounded verification, we plan to consider additional state-of-the-art techniques for symbolic analysis such as IC3 [61, 62], implemented for instance in nuXmv [63]. Integrating our analysis tool with simulation-based techniques, such as statistical model checking [64, 65], to complement formal verification, would also make our platform more suitable to deal with very large systems.

References

[1] C. Pinciroli, G. Beltrame, Buzz: An extensible programming language for heterogeneous swarm robotics, in: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2016, pp. 3794–3800. arXiv:1507.05946, doi:10.1109/IROS.2016.7759558.

[2] A. Ricci, A. Omicini, M. Viroli, L. Gardelli, E. Oliva, Cognitive Stigmergy: Towards a Framework Based on Agents and Artifacts, in: Environments for Multi-Agent Systems (E4MAS), Vol. 4389 of LNCS, Springer, 2006, pp. 124–140. doi:10.1007/978-3-540-71103-2_7.

[3] Y. Abd Alrahman, R. De Nicola, M. Loreti, On the Power of Attribute-Based Communication, in: E. Albert, I. Lanese (Eds.), 36th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE), Vol. 9688 of LNCS, Springer, Heraklion, Greece, 2016, pp. 1–18. doi:10.1007/978-3-319-39570-8_1.

[4] C. Pinciroli, A. Lee-Brown, G. Beltrame, A Tuple Space for Data Sharing in Robot Swarms, in: 9th EAI International Conference on Bio-Inspired Information and Communications Technologies (BICT), ICST/ACM, 2015, pp. 287–294.

[5] D. Weyns, M. Schumacher, A. Ricci, M. Viroli, T. Holvoet, Environments in multiagent systems, The Knowledge Engineering Review 20 (02) (2006) 127. doi:10.1017/S0269888905000457.

[6] R. De Nicola, L. Di Stefano, O. Inverso, Toward Formal Models and Languages for Verifiable Multi-Robot Systems, Frontiers in Robotics and AI 5. doi:10.3389/frobt.2018.00094.

[7] R. De Nicola, L. Di Stefano, O. Inverso, Multi-agent Systems with Virtual Stigmergy, in: M. Mazzara, I. Ober, G. Salaün (Eds.), Software Technologies: Applications and Foundations (STAF) - Collocated Workshops, Revised Selected Papers, Vol. 11176 of LNCS, Springer, Toulouse, France, 2018, pp. 351–366. doi:10.1007/978-3-030-04771-9_26.

[8] C. W. Reynolds, Flocks, herds and schools: A distributed behavioral model, in: 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), Vol. 21, ACM, 1987, pp. 25–34. doi:10.1145/37402.37406.

[9] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, R. Peralta, Computation in networks of passively mobile finite-state sensors, in: S. Chaudhuri, S. Kutten (Eds.), 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM, St. John's, Newfoundland, Canada, 2004, pp. 290–299. doi:10.1145/1011767.1011810.

[10] J. Aspnes, E. Ruppert, An Introduction to Population Protocols, in: B. Garbinato, H. Miranda, L. E. T. Rodrigues (Eds.), Middleware for Network Eccentric and Mobile Applications, Springer, 2009, pp. 97–120. doi:10.1007/978-3-540-89707-1_5.

[11] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM 21 (7) (1978) 558–565. doi:10.1145/359545.359563.

[12] A. Okubo, Dynamical aspects of animal grouping: Swarms, schools, flocks, and herds, Advances in Biophysics 22 (1986) 1–94. doi:10.1016/0065-227X(86)90003-1.

[13] J. Toner, Y. Tu, Flocks, herds, and schools: A quantitative theory of flocking, Physical Review E 58 (4) (1998) 4828–4858. doi:10.1103/PhysRevE.58.4828.

[14] P. Blikstein, W. Rand, U. Wilensky, Participatory, embodied, multi-agent simulation, in: H. Nakashima, M. P. Wellman, G. Weiss, P. Stone (Eds.), 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), ACM, Hakodate, Japan, 2006, pp. 1457–1458. doi:10.1145/1160633.1160913.

[15] L. Gray, A Mathematician Looks at Wolfram's New Kind of Science, Notices of the American Mathematical Society 50 (2) (2003) 200–211.

[16] A. Efrima, D. Peleg, Distributed algorithms for partitioning a swarm of autonomous mobile robots, Theoretical Computer Science 410 (14) (2009) 1355–1368. doi:10.1016/j.tcs.2008.04.042.

[17] H.-Y. Chen, C. David, D. Kroening, P. Schrammel, B. Wachter, Synthesising Interprocedural Bit-Precise Termination Proofs (T), in: M. B. Cohen, L. Grunske, M. Whalen (Eds.), 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, Lincoln, NE, USA, 2015, pp. 53–64. doi:10.1109/ASE.2015.10.

[18] M. Y. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, D. A. Nicole, ESBMC 5.0: An industrial-strength C model checker, in: M. Huchard, C. Kästner, G. Fraser (Eds.), ACM/IEEE International Conference on Automated Software Engineering (ASE), ACM, Montpellier, France, 2018, pp. 888–891. doi:10.1145/3238147.3240481.

[19] G. Brat, J. A. Navas, N. Shi, A. Venet, IKOS: A Framework for Static Analysis Based on Abstract Interpretation, in: D. Giannakopoulou, G. Salaün (Eds.), 12th International Conference on Software Engineering and Formal Methods (SEFM), Vol. 8702 of LNCS, Springer, Grenoble, France, 2014, pp. 271–277. doi:10.1007/978-3-319-10431-7_20.

[20] A. Venet, The Gauge Domain: Scalable Analysis of Linear Inequality Invariants, in: P. Madhusudan, S. A. Seshia (Eds.), 24th International Conference on Computer Aided Verification (CAV), Vol. 7358 of LNCS, Springer, Berkeley, CA, USA, 2012, pp. 139–154. doi:10.1007/978-3-642-31424-7_15.

[21] Z. Rakamaric, M. Emmi, SMACK: Decoupling Source Language Details from Verifier Implementations, in: A. Biere, R. Bloem (Eds.), 26th International Conference on Computer Aided Verification (CAV), Vol. 8559 of LNCS, Springer, Vienna, Austria, 2014, pp. 106–113. doi:10.1007/978-3-319-08867-9_7.

[22] A. Lal, S. Qadeer, S. K. Lahiri, A Solver for Reachability Modulo Theories, in: P. Madhusudan, S. A. Seshia (Eds.), 24th International Conference on Computer Aided Verification (CAV), Vol. 7358 of LNCS, Springer, Berkeley, CA, USA, 2012, pp. 427–443. doi:10.1007/978-3-642-31424-7_32.

[23] D. Beyer, M. E. Keremoglu, CPAchecker: A Tool for Configurable Software Verification, in: G. Gopalakrishnan, S. Qadeer (Eds.), 23rd International Conference on Computer Aided Verification (CAV), Vol. 6806 of LNCS, Springer, Snowbird, UT, USA, 2011, pp. 184–190. doi:10.1007/978-3-642-22110-1_16.

[24] S. Löwe, CPAchecker with Explicit-Value Analysis Based on CEGAR and Interpolation - (Competition Contribution), in: N. Piterman, S. A. Smolka (Eds.), 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Vol. 7795 of LNCS, Springer, Rome, Italy, 2013, pp. 610–612. doi:10.1007/978-3-642-36742-7_44.

[25] D. Weyns, T. Holvoet, A Formal Model for Situated Multi-Agent Systems, Fundamenta Informaticae 63 (2-3) (2004) 125–158.

[26] T. M. Liggett, Interacting Particle Systems, Classics in Mathematics, Springer, 2005. doi:10.1007/b138374.

[27] A. Scheidler, A. Brutschy, E. Ferrante, M. Dorigo, The k-Unanimity Rule for Self-Organized Decision-Making in Swarms of Robots, IEEE Transactions on Cybernetics 46 (5) (2016) 1175–1188. doi:10.1109/TCYB.2015.2429118.

[28] M. Brambilla, E. Ferrante, M. Birattari, M. Dorigo, Swarm robotics: A review from the swarm engineering perspective, Swarm Intelligence 7 (1) (2013) 1–41. doi:10.1007/s11721-012-0075-2.

[29] D. Angluin, J. Aspnes, D. Eisenstat, A simple population protocol for fast robust approximate majority, Distributed Computing 21 (2) (2008) 87–102. doi:10.1007/s00446-008-0059-z.

[30] M. Blondin, J. Esparza, S. Jaax, Peregrine: A Tool for the Analysis of Population Protocols, in: H. Chockler, G. Weissenbacher (Eds.), 30th International Conference on Computer Aided Verification (CAV), Vol. 10981 of LNCS, Springer, Oxford, UK, 2018, pp. 604–611. doi:10.1007/978-3-319-96145-3_34.

[31] M. Dastani, A Survey of Multi-agent Programming Languages and Frameworks, in: O. Shehory, A. Sturm (Eds.), Agent-Oriented Software Engineering - Reflections on Architectures, Methodologies, Languages, and Frameworks, Springer, 2014, pp. 213–233. doi:10.1007/978-3-642-54432-3_11.

[32] C.-E. Hrabia, M. Lützenberger, S. Albayrak, Towards adaptive multi-robot systems: Self-organization and self-adaptation, The Knowledge Engineering Review 33. doi:10.1017/S0269888918000176.

[33] N. Koenig, A. Howard, Design and use paradigms for Gazebo, an open-source multi-robot simulator, in: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Vol. 3, IEEE, Sendai, Japan, 2004, pp. 2149–2154. doi:10.1109/IROS.2004.1389727.

[34] E. Rohmer, S. P. N. Singh, M. Freese, V-REP: A versatile and scalable robot simulation framework, in: IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, Tokyo, Japan, 2013, pp. 1321–1326. doi:10.1109/IROS.2013.6696520.

[35] J. Lächele, A. Franchi, H. H. Bülthoff, P. Robuffo Giordano, SwarmSimX: Real-Time Simulation Environment for Multi-robot Systems, in: D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, I. Noda, N. Ando, D. Brugali, J. J. Kuffner (Eds.), 3rd International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR), Vol. 7628 of LNCS, Springer, Tsukuba, Japan, 2012, pp. 375–387. doi:10.1007/978-3-642-34327-8_34.

[36] G. Echeverria, S. Lemaignan, A. Degroote, S. Lacroix, M. Karg, P. Koch, C. Lesire, S. Stinckwich, Simulating Complex Robotic Scenarios with MORSE, in: D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos,

D. Tygar, M. Y. Vardi, G. Weikum, I. Noda, N. Ando, D. Brugali, J. J. Kuffner (Eds.), 3rd International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR), Vol. 7628 of LNCS, Springer, Tsukuba, Japan, 2012, pp. 197–208. doi:10.1007/978-3-642-34327-8_20.

[37] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, M. Birattari, L. M. Gambardella, M. Dorigo, ARGoS: A modular, parallel, multi-engine simulator for multi-robot systems, Swarm Intelligence 6 (4) (2012) 271–295. doi:10.1007/S11721-012-0072-5.

[38] L. Pitonakova, M. Giuliani, A. G. Pipe, A. F. T. Winfield, Feature and Performance Comparison of the V-REP, Gazebo and ARGoS Robot Simulators, in: M. Giuliani, T. Assaf, M. E. Giannaccini (Eds.), 19th Annual Conference Towards Autonomous Robotic Systems (TAROS), Vol. 10965 of LNCS, Springer, Bristol, UK, 2018, pp. 357–368. doi:10.1007/978-3-319-96728-8_30.

[39] A. Lomuscio, H. Qu, F. Raimondi, MCMAS: An open-source model checker for the verification of multi-agent systems, International Journal on Software Tools for Technology Transfer 19 (1) (2017) 9–30. doi:10.1007/s10009-015-0378-x.

[40] L. Bayındır, A review of swarm robotics tasks, Neurocomputing 172 (442) (2016) 292–321. doi:10.1016/j.neucom.2015.05.116.

[41] L. Pitonakova, R. Crowder, S. Bullock, Behaviour-data relations modelling language for multi-robot control algorithms, in: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2017, pp. 727–732. doi:10.1109/IROS.2017.8202231.

[42] J. Bachrach, J. McLurkin, A. Grue, Protoswarm: A language for programming multi-robot systems using the amorphous medium abstraction, in: 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), Vol. 3, IFAAMAS, 2008, pp. 1175–1178.

[43] J. Bachrach, J. Beal, J. McLurkin, Composable continuous-space programs for robotic swarms, Neural Computing and Applications 19 (6) (2010) 825–847. doi:10.1007/s00521-010-0382-8.

[44] F. Damiani, M. Viroli, J. Beal, A type-sound calculus of computational fields, Science of Computer Programming 117 (2016) 17–44. doi:10.1016/j.scico.2015.11.005.

[45] D. Weerasooriya, A. S. Rao, K. Ramamohanarao, Design of a Concurrent Agent-Oriented Language, in: M. J. Wooldridge, N. R. Jennings (Eds.), International Workshop on Agent Theories, Architectures, and Languages, Vol. 890 of LNCS, Springer, Amsterdam, The Netherlands, 1994, pp. 386–401. doi:10.1007/3-540-58855-8_25.

[46] R. H. Bordini, M. Fisher, W. Visser, M. Wooldridge, Verifying Multi-agent Programs by Model Checking, Autonomous Agents and Multi-Agent Systems 12 (2) (2006) 239–256. doi:10.1007/s10458-006-5955-7.

[47] S. Bucheli, D. Kroening, R. Martins, A. Natraj, From AgentSpeak to C for Safety Considerations in Unmanned Aerial Vehicles, in: C. Dixon, K. Tuyls (Eds.), 16th Annual Conference Towards Autonomous Robotic Systems (TAROS), Vol. 9287 of LNCS, Springer, Liverpool, UK, 2015, pp. 69–81. doi:10.1007/978-3-319-22416-9_9.

[48] A. Philippou, M. Toro, M. Antonaki, Simulation and Verification in a Process Calculus for Spatially-Explicit Ecological Models, Scientific Annals of Computer Science 23 (1) (2013) 119–167. doi:10.7561/SACS.2013.1.119.

[49] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, A. Santi, Multi-agent oriented programming with JaCaMo, Science of Computer Programming 78 (6) (2013) 747–761. doi:10.1016/j.scico.2011.10.004.

[50] R. H. Bordini, J. F. Hbner, M. Wooldridge, Programming Multi-Agent Systems in AgentSpeak Using Jason, Wiley Series in Agent Technology, John Wiley and Sons, Ltd, 2007. doi:10.1002/9780470061848.

[51] M. O'Brien, R. C. Arkin, D. Harrington, D. Lyons, S. Jiang, Automatic Verification of Autonomous Robot Missions, in: D. Brugali, J. F. Broenink, T. Kroeger, B. A. MacDonald (Eds.), 4th International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR), Vol. 8810 of LNCS, Springer, Bergamo, Italy, 2014, pp. 462–473. doi:10.1007/978-3-319-11900-7_39.

[52] R. De Nicola, D. Latella, A. L. Lafuente, M. Loreti, A. Margheri, M. Massink, A. Morichetta, R. Pugliese, F. Tiezzi, A. Vandin, The SCEL Language: Design, Implementation, Verification, in: Software Engineering for Collective Autonomic Systems, Vol. 8998 of LNCS, Springer, 2015, pp. 3–71. doi:10.1007/978-3-319-16310-9_1.

[53] V. Grimm, E. Revilla, U. Berger, F. Jeltsch, W. M. Mooij, S. F. Railsback, H.-H. Thulke, J. Weiner, T. Wiegand, D. L. DeAngelis, Pattern-Oriented Modeling of Agent-Based Complex Systems: Lessons from Ecology, Science 310 (5750) (2005) 987–991. doi:10.1126/science.1116681.

[54] J. Hillston, Challenges for Quantitative Analysis of Collective Adaptive Systems, in: M. Abadi, A. Lluch-Lafuente (Eds.), 8th International Symposium on Trustworthy Global Computing (TGC), Revised Selected Papers, Vol. 8358 of LNCS, Springer, Buenos Aires, Argentina, 2013, pp. 14–21. doi:10.1007/978-3-319-05119-2_2.

[55] L. Tesfatsion, Agent-Based Computational Economics: Growing Economies From the Bottom Up, Artificial Life 8 (1) (2002) 55–82. doi:10.1162/106454602753694765.

[56] J. Hillston, A compositional approach to performance modelling, Ph.D. thesis, University of Edinburgh, UK (1994).

[57] M. Loreti, J. Hillston, Modelling and Analysis of Collective Adaptive Systems with CARMA and its Tools, in: M. Bernardo, R. De Nicola, J. Hillston (Eds.), Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems - 16th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM), Advanced Lectures, Vol. 9700 of LNCS, Springer, 2016, pp. 83–119. doi:10.1007/978-3-319-34096-8_4.

[58] I. Suzuki, M. Yamashita, Distributed Anonymous Mobile Robots: Formation of Geometric Patterns, SIAM Journal on Computing 28 (4) (1999) 1347–1363. doi:10.1137/S009753979628292X.

[59] C. Auger, Z. Bouzid, P. Courtieu, S. Tixeuil, X. Urbain, Certified Impossibility Results for Byzantine-Tolerant Mobile Robots, in: T. Higashino, Y. Katayama, T. Masuzawa, M. Potop-Butucaru, M. Yamashita (Eds.), 15th International Symposium Stabilization, on Safety, and Security of Distributed Systems (SSS), Vol. 8255 of LNCS, Springer, Osaka, Japan, 2013, pp. 178–190. doi:10.1007/978-3-319-03089-0_13.

[60] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, T. van Dijk, LTSmin: High-Performance Language-Independent Model Checking, in: C. Baier, C. Tinelli (Eds.), 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Vol. 9035 of LNCS, Springer, London, UK, 2015, pp. 692–707. doi:10.1007/978-3-662-46681-0_61.

[61] A. R. Bradley, SAT-Based Model Checking without Unrolling, in: R. Jhala, D. A. Schmidt (Eds.), 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), Vol. 6538 of LNCS, Springer, Austin, TX, USA, 2011, pp. 70–87. doi:10.1007/978-3-642-18275-4_7.

[62] A. Cimatti, A. Griggio, Software Model Checking via IC3, in: P. Madhusudan, S. A. Seshia (Eds.), 24th International Conference on Computer Aided Verification (CAV), Vol. 7358 of LNCS, Springer, Berkeley, CA, USA, 2012, pp. 277–293. doi:10.1007/978-3-642-31424-7_23.

[63] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, The nuXmv Symbolic Model Checker, in: A. Biere, R. Bloem (Eds.), 26th International Conference on Computer Aided Verification (CAV), Vol. 8559 of LNCS, Springer, Vienna, Austria, 2014, pp. 334–342. doi:10.1007/978-3-319-08867-9_22.

[64] A. Legay, B. Delahaye, S. Bensalem, Statistical Model Checking: An Overview, in: International Conference on Runtime Verification (RV), Vol. 6418 of LNCS, Springer, 2010, pp. 122–135. doi:10.1007/978-3-642-16612-9_11.

[65] B. Herd, S. Miles, P. McBurney, M. Luck, Quantitative Analysis of Multiagent Systems Through Statistical Model Checking, in: M. Baldoni, L. Baresi, M. Dastani

(Eds.), 3rd International Workshop on Engineering Multi-Agent Systems (EMAS). Revised, Selected, and Invited Papers, Vol. 9318 of LNCS, Springer, Istanbul, Turkey, 2015, pp. 109–130. doi:10.1007/978-3-319-26184-3_7.

Appendix A. C encoding of a LAbS system

Listing 1: C encoding of the *flocking* example (Table 9).

```
N = ...   // Number of agents
K_I = ... // Number of attributes
K_L = ... // Number of stigmergy keys
B = ...   // Number of transitions

int v[N];                                  // v[i] --> program counter for agent i
int I[N][K_I], Lvalue[N][K_L], Ltstamp[N][K_L];    // integer values for all keys
bool Zc[N][K_L], Zp[N][K_L]    // Zc[i][j]=1 --> key j is in Zc of agent i (or Zp)

/* System-level actions */
void attr(int id, int key, int value) { ... }   // encodes "key <- value"
void lstig(int id, int key, int value) { ... }  // encodes "key <~ value"
bool link(int a, int b, int key) { ... } // link predicate true iff.  I_a, L_a, I_b, L_b ⊨ φ_key
void confirm(void) { ... }
void propagate(void) { ... }

/* Agent-level actions */
void stmt1(int tid) {  // Encoding of "x, y <- (x + dir_x) % G, (y + dir_y) % G"
  assume((v[tid] == 1));

  int val0 = (I[tid][0]) + (Lvalue[tid][0]) % G;
  int val1 = (I[tid][1]) + (Lvalue[tid][1]) % G;
  attr(tid, 0, val0);
  attr(tid, 1, val1);
  Zc[tid][0] = 1;
  Zc[tid][1] = 1;
  setHin(tid, 1);

  v[tid] = 1;
}

/* Initialisation and properties to verify */
void init() { ... }
void monitor(void) { ... }
void check(void) {
  assert(Lvalue[0][0] == Lvalue[1][0] && ... && Lvalue[N-2][1] == Lvalue[N-1][1]);
}

/* Scheduler */
int main(void) {
  init();
  int agent = *;
  assume(agent < N);

  for (round=0;round<B;round++) {   // execution loop
    if ( * ) {
      int choice = * ;
      assume(choice >= 0 && choice < 1);
      switch (choice) {
        case 0: stmt1(agent);
      }
    }
    else {
      if ( * ) propagate();
      else confirm();
    }

    monitor();
    agent = agent + 1 % N;          // round-robin scheduling
  }

  check();
}
```