Towards a Formal Verification Methodology for Collective Robotic Systems[⋆]

Edmond Gjondrekaj[1], Michele Loreti[1], Rosario Pugliese[1], Francesco Tiezzi[2], Carlo Pinciroli[3], Manuele Brambilla[3], Mauro Birattari[3], and Marco Dorigo[3]

[1] Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, Italy
[2] IMT, Institute for Advanced Studies Lucca, Italy
[3] IRIDIA, CoDE, Université Libre de Bruxelles, Belgium

Abstract. We present a novel formal verification approach for *collective robotic systems* that is based on the use of the formal language KLAIM and related analysis tools. While existing approaches focus on either micro- or macroscopic views of a system, we model aspects of both the robot hardware and behaviour, as well as relevant aspects of the environment. We illustrate our approach through a robotics scenario, in which three robots cooperate in a decentralized fashion to transport an object to a goal area. We first model the scenario in KLAIM. Subsequently, we introduce random aspects to the model by stochastically specifying actions execution time. Unlike other approaches, the specification thus obtained enables quantitative analysis of crucial properties of the system. We validate our approach by comparing the results with those obtained through physics-based simulations.

1 Introduction

Collective robotic systems are systems in which a group of autonomous robots cooperates to tackle a task. By taking advantage of the absence of a centralized controller, the use of local communication and sensing, and the lack of global knowledge, these systems have the potential to display properties such as robustness and parallelism.

Collective robotic systems are difficult to design and analyse because the collective behaviour of the system is the result of the non-linear interaction of the individual robots with each other and with the environment. The realisation of these systems currently relies on the ingenuity and expertise of the designer due to the lack of sound engineering approaches and accountable engineering practices. The typical design approach involves several loops of development, testing and modification of the behaviour of each robot until the desired collective behaviour is obtained. This iterative process, often performed first using computer simulations and eventually on real robots, is in general expensive, time consuming, and cannot provide guarantees of system correctness. Indeed, *experimentation* with real robots is very costly and time consuming. Physics-based *simulation*, that attempts to realistically model the environment, the robots and their interactions, is faster and more reliable than experimentation, but requires an exhaustive scan of the design parameter space to reach any conclusion (see e.g. [18]). Besides, experimentation and simulation can only validate a small subset of the possible

⋆ This work has been partially sponsored by the EU project ASCENS, 257414.

system scenarios and are often impractical to exhaustively study a collective behaviour. In other words, these approaches cannot ensure a complete coverage of the critical aspects of the system nor the absence of residual anomalies.

A major open issue in the design and development of collective robotic systems is thus to guarantee the correctness of the collective behaviour of a system composed of autonomous components. Formal verification techniques, such as *model checking*, can complement traditional approaches by guaranteing that certain system properties hold.

In this paper, we introduce a formal verification approach for the design of collective robotic systems that lays down the basis of a principled development methodology for such systems. Our approach consists of two phases. In the first phase, we model the robot behaviour and the environment with the formal language Klaim [7]. Klaim is a tuple-space-based coordination language that allows one to define an accurate model of a distributed system using a small set of primitives. Unlike existing approaches that mainly focus on micro- or macroscopic aspects of the system, Klaim permits to capture both hardware aspects of the robots and their behaviour. In addition, Klaim can suitably model relevant environmental aspects. In the second phase, we enrich the model with stochastic aspects, using Klaim's stochastic extension StoKlaim [8], and formalise the desired properties using MoSL [8]. MoSL is a stochastic logic that, in addition to qualitative properties, permits specifying time-bounded probabilistic reachability properties and properties about resource distribution. The properties of interest are then verified against the StoKlaim specifications by exploiting the analysis tool Sam [8, 19].

To demonstrate the approach, we analyse a collective transport scenario [10], in which, while avoiding obstacles, a group of three robots must carry an object that is too heavy for a single robot to move. This behaviour is a good candidate to establish the validity of our approach since it has many of the features which characterize collective robotic systems. Indeed, the system is completely distributed (there is neither a centralized controller nor a leader), the robots do not have any global knowledge, such as a map of the environment, and no common frame of reference is used for coordination.

Modelling collective robotic systems is a challenging task. Indeed, for understanding their dynamics, it is necessary to model in detail both the spatial aspects (e.g. positions of robots, obstacles and carried objects) and the temporal aspects (e.g. robots' action execution time) of a system. These aspects are crucial since, e.g., without the position of robots and carried objects in time the model would be of limited use to verify the correctness of the collective transport behaviour. Differently from the relevant literature, in which spatial and temporal aspects are usually discarded or simplified (see, e.g., [18, 12]), our approach allows us to easily achieve the needed level of detail. The price to be paid is an increased complexity of the model that might limit the number of robots that can be considered in a given scenario.

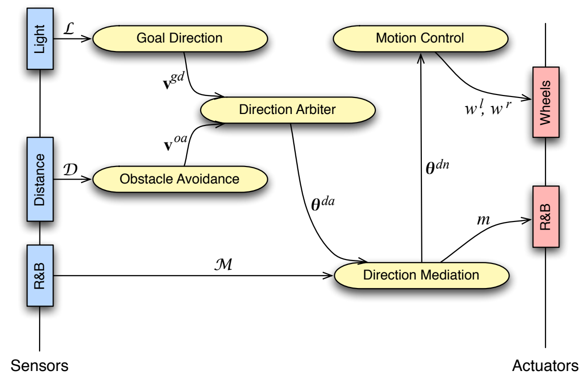
In addition to suit collective robotic systems, compositionality and high modularity, which are typical of Klaim and StoKlaim specifications, allow us to easily and flexibly experiment with different values of the behaviour and scenario parameters. This permits tuning them in order, for example, to optimise the performance of the system or prevent instabilities. Moreover, the possibility to change the parameters of the environment permits to easily check the collective behaviour of the robots under different environmental conditions while saving time and resources with respect to simulated or real robots. E.g., regarding our scenario, we have studied how robots' behaviour is influenced by the position of the light source indicating the goal area. We have verified that if the robots do not perceive the light they could not be able to reach the goal, and demonstrated that a simple change to their behaviour is enough to solve this problem.

State of the art and related work. Formal verification has been successfully applied to many different classes of distributed systems, such as embedded real-time systems and wireless sensor networks [6, 20]. These kinds of systems, even though distributed, do not present the challenges of robotic systems. In fact, their components do not move and do not interact with the environment as a robot does. Considering more specifically modelling and verification of collective robotic systems, most of the work focuses on systems that are not fully distributed as they have a centralized controller or a leader, or make use of global knowledge. Examples of such work can be found in [15, 11].

Only a few studies deal explicitly with robotic systems which are fully distributed and do not use global knowledge. Winfield et al. [21] devised a microscopic modelling approach based on linear temporal logic (LTL) to model a swarm of robots whose goal is to navigate in the environment while keeping in communication range. The same approach was then studied and expanded in [9]. Konur et al. [17] proposed to use probabilistic computation tree logic (PCTL) to formally verify the properties of a swarm of robots that perform a foraging task using a macroscopic model. A similar property-driven approach is proposed in [3], where PRISM is used to verify PCTL formulae expressing properties of an aggregation scenario, described by a Discrete Time Markov Chain, where the robots have to cluster in an area of the environment. The design methodology proposed in [16] exploits a *post hoc* analysis to evaluate the expected performance of synthesized robot controllers. Such analysis does not permit verifying generic system properties, but just determining the probability of correct task execution to refine the controller synthesis. Moreover, robot systems are not specified through a linguistic approach, but in terms of states, functions over states, and state transitions. In [4], the use of Maude and related tools is put forward for analysing a self-assembling robots scenario, where robots physically connect to each other when the environment prevents them from reaching their goals individually. The work focusses on the adaptive aspects of the system, while abstracting from the spatial one, since the arena is modelled as a discrete grid and robots movements are discretized into four directions.

All the above approaches greatly simplify the spatial and/or temporal aspects of the system and are thus not suited for a collective transport behaviour. Moreover, in contrast to these approaches, we focus on analysing the properties of a robot's behaviour both from the point of view of the interaction between the running code and the robot's internal devices (i.e. sensors and actuators) and from the point of view of the interaction with the other robots. To the best of our knowledge, there are no published works that deal with modelling and formal verification of a collective transport behaviour.

Summary of the rest of the paper. In Section 2, we introduce the considered robotics scenario. In Section 3, we review the formal basis underlying the proposed verification approach, namely the specification language KLAIM, the stochastic extension STOKLAIM, the stochastic logic MoSL, and the analysis tool SAM. In Section 4, we describe the relevant aspects of the KLAIM specification of the scenario, while in Section 5, we present its stochastic analysis. Finally, in Section 6 we indicate directions for future work.



Symbol	Meaning
$\mathcal{L}$	Set of light readings
$\mathcal{D}$	Set of distances
$\mathcal{M}$	Set of received messages
m	Sent message
$\mathbf{v}^{gd}$	Vector toward to the goal area
$\mathbf{v}^{oa}$	Vector to avoid obstacles
θ^{da}	Individual desired direction
w^l/w^r	Left/right wheel speed
θ^{dn}	Mediated direction

Fig. 1. A diagrammatic description of the behaviour for collective transport of an object

2 A collective robotics scenario

In order to illustrate our approach, we consider a robotics scenario, borrowed from [10], whereby three identical robots must collectively transport an object to a goal area. The robots operate in an arena where a number of obstacles are present and a light source indicates the goal area. It is assumed that the three robots have already physically assembled to the object to transport and cannot disassemble until the goal area is reached.

Each robot is a marXbot [2] equipped with: (i) a *light sensor*, to perceive the direction to a light source; (ii) a *distance scanner*, to obtain relative distances from objects in the environment; (iii) a *range and bearing communication system*, to communicate with other robots; (iv) *wheels*, to move around the environment.

All robots execute the same code, i.e., the so-called *behaviour*. Each of them senses the environment and calculates the *desired direction*, that is, the direction the robot would follow if it were alone. Since each robot has a local perception of the environment, the desired directions of the robots could differ. In fact, at each control step, one robot could sense or not the position of the goal and/or the position of obstacles. According to the available information, in different moments, a robot can be *informed*, that is, it has a desired direction to follow, or *non-informed* otherwise. Informed robots communicate to the other robots their desired direction. Anyway, to actuate the wheels, any robot uses a *socially mediated direction* obtained by averaging the received directions, so all robots can follow the same direction even if they have a different perception of the environment.

A diagrammatic description of the behaviour, together with an explanation of the used notation, is reported in Fig. 1. The horizontal blobs are behavioural modules that take an input and produce an output. The output is usually a set of variables that can be input to other modules or set as actuator values, while the input can be the result of other modules and/or sensor readings. The behaviour is composed of five modules.

Goal Direction queries the light sensors to calculate the vector $\mathbf{v}^{gd}$ to the position of maximum light intensity sensed, which points toward to the goal area. 24 light sensors are located around the body of the robot in a ring at uniform fixed angles, and each of them returns the measured intensity expressed as a vector directed from the robot's center outwards. The vector $\mathbf{v}^{gd}$ is calculated as a normalised sum over these 24 vectors. *Obstacle Avoidance* detects the presence of obstacles and calculates the vector $\mathbf{v}^{oa}$ that points away from the obstacle. The distance scanner is a rotating sensor that can span the area around the robot and return 24 vectors whose length corresponds to the distance to a sensed object from the center of the robot (if no obstacle is perceived along a given direction, the length of the vector is obs_d_{MAX}). The length of vector $\mathbf{v}^{oa}$ corresponds to the distance to the closest object rescaled in [0, 1], while its angle corresponds to the angle of the sum of all the readings. Notably, the resulting angle points away from the closest obstacles, because the readings that correspond to obstacle-free areas have the highest value obs_d_{MAX}. *Direction Arbiter* takes as inputs $\mathbf{v}^{gd}$ and $\mathbf{v}^{oa}$ and calculates the direction θ^{da}, that is the desired direction of the robot before computing the mediated direction. Since the length of $\mathbf{v}^{oa}$ represents how urgent it is to avoid obstacles, we use it as a weight to combine the directions to the goal area and to avoid obstacles. *Direction Mediation* calculates the mediated direction θ^{dn} as the average of the directions received from other robots through the range and bearing communication system. This module sends a message m to nearby robots containing θ^{da}, if the robot is informed, and θ^{dn} otherwise. *Motion Control* converts the direction θ^{dn} into the wheel speeds w^r and w^l.

3 Formal foundations of the verification approach

In this section, we provide a brief overview of the formal methods exploited by the proposed approach for specifying and verifying collective robotic systems.

Specification. A distributed system is modelled in Klaim as a net of nodes, each one with a local data repository and a set of running processes. We informally present here a version of Klaim enriched with some standard control flow constructs (i.e., if-then-else, sequence, etc.) that are part of the input language of the analysis tool used in Section 5. These constructs simplify the specification task and can be easily rendered in the language originally presented in [7]. For simplicity's sake, we omit the linguistic constructs for dealing with name restriction and dynamic node creation, since they are not used in the considered robotics system specification. We refer to [7] for a formal presentation of the language and to [1] for a Java framework for programming in Klaim.

Nets are finite plain collections of nodes composed by means of the parallel composition operator $\_ \parallel \_$. *Nodes* $s\ ::_\rho\ C$ have a unique *locality name* s (i.e., their network address) and an allocation environment ρ, and host a set of components C. An *allocation environment* provides a name resolution mechanism by mapping *locality variables* l (i.e., aliases for addresses), occurring in the processes hosted in the corresponding node, into localities s. The distinguished locality variable **self** is used by processes to refer to the address of their current hosting node. *Components* are finite plain collections of processes P and evaluated tuples $\langle t \rangle$, composed by means of the parallel operator $\_ \mid \_$.

Processes P are the Klaim active computational units and may be executed concurrently either at the same locality or at different localities. They are built up from basic actions (see below) and process calls $A(p_1, \ldots, p_m)$ by means of sequential composition $P_1; P_2$, parallel composition $P_1 \mid P_2$, conditional choice **if** (e) **then** $\{P\}$ **else** $\{Q\}$, iterative constructs **for** $i = n$ **to** m $\{ P \}$ and **while** (e) $\{P\}$, and (possibly recursive) process definitions $A(f_1, \ldots, f_n) \triangleq P$ with f_i pairwise distinct. Notably, A denotes a process identifier, while f_i and p_j denote formal and actual parameters, respectively. Moreover, e ranges over *expressions*, which contain basic values (booleans, integers, strings, floats, etc.) and value variables x, and are formed by using the standard operators on basic values, simple data structures (i.e., arrays and lists) and the non-blocking retrieval actions **inp** and **readp** (explained below). In the rest of this section, we will use the notation ℓ to range over locality names and locality variables.

During their execution, processes perform some *basic actions*. Actions $\textbf{in}(T)@\ell$ and $\textbf{read}(T)@\ell$ are retrieval actions and permit to withdraw/read data tuples from the tuple space hosted at the (possibly remote) locality ℓ: if a matching tuple is found, one is non-deterministically chosen, otherwise the process is blocked. They exploit templates as patterns to select tuples in shared tuple spaces. *Tuples t* are sequences of actual fields, i.e. locality names, locality variables, expressions and processes. Instead, *templates T* are sequences of actual and formal fields, where the latter are written $!\,x$, $!\,l$ or $!\,X$ and are used to bind variables to values, locality names or processes, respectively. For the sake of readability, we use "_" to denote a *don't care* formal field in a template; this corresponds to a formal field $!\,dc$ using the variable dc that does not occur elsewhere in the specification. Actions $\textbf{inp}(T)@\ell$ and $\textbf{readp}(T)@\ell$ are non-blocking versions of the retrieval actions: namely, during their execution processes are never blocked. Indeed, if a matching tuple is found, **inp** and **readp** act similarly to **in** and **read**, and additionally return the value *true*; otherwise they return the value *false* and the executing process does not block. $\textbf{inp}(T)@\ell$ and $\textbf{readp}(T)@\ell$ can be used where either a boolean expression or an action is expected (in the latter case, the returned value is simply ignored). Action $\textbf{out}(t)@\ell$ adds the tuple resulting from the evaluation of t to the tuple space of the target node identified by ℓ, while action $\textbf{eval}(P)@\ell$ sends the process P for execution to the (possibly remote) node identified by ℓ. Both **out** and **eval** are non-blocking actions. Action $\textbf{rpl}(T) \rightarrow (t)@\ell$ atomically replaces a non-deterministically chosen tuple in ℓ matching the template T by the tuple t; if no tuple in ℓ matches T, the action behaves as $\textbf{out}(t)@\ell$. Finally, action $x := e$ assigns the value of e to x and, differently from all the other actions, it is not indexed with an address because it always acts locally.

Verification. Quantitative analysis of a Klaim specification can be enabled by associating a rate to each action, thus obtaining a StoKlaim [8] specification. This rate is the parameter of an exponentially distributed random variable accounting for the action duration time. A real valued random variable X has a *negative exponential distribution* with *rate* $\lambda > 0$ if and only if the *probability* that $X \leq t$, with $t > 0$, is $1 - e^{-\lambda \cdot t}$. The expected value of X is λ^{-1}, while its variance is λ^{-2}. The operational semantics of StoKlaim permits associating to each specification a Continuous Time Markov Chain that can be used to perform quantitative analyses of the considered system.

The desired properties of a system under verification are formalised using the stochastic logic MoSL [8]. MoSL formulae use predicates on the tuples located in the considered Klaim net to express the reachability of the system goal, or more generally, of a certain system state, while passing or not through other specific intermediate states. Therefore, MoSL can be used to express quantitative properties of the overall system behaviour, such as, e.g., if the robots are able to reach the goal, or collisions between the robots and the obstacles ever happen in the system. The results of the evaluation of

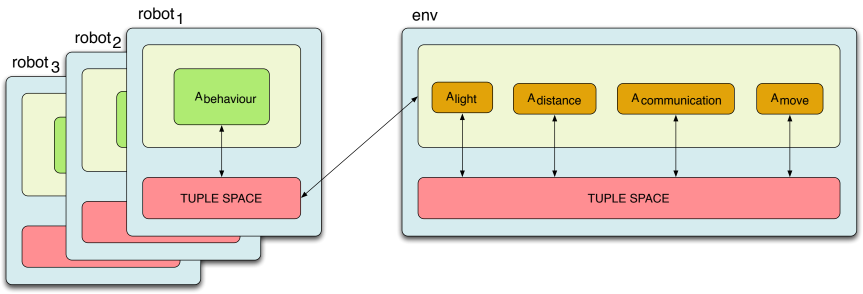


Fig. 2. Graphical representation of the Klaim specification

such properties do not have a rigid meaning, like *true* or *false*, but have a less absolute nature, e.g. *in 99.7% of the cases, the robots reach the goal within t time units*.

Verification of MoSL formulae over StoKlaim specifications is assisted by the analysis tool Sam [8, 19], which uses a statistical model checking algorithm [5] to estimate the probability of the property satisfaction. In this way, the probability associated to a path-formula is determined after a set of independent observations and the algorithm guarantees that the difference between the computed value and the exact one exceeds a given *tolerance* ε with a probability that is less than a given *error probability* p.

4 Specification of the robotics scenario

In this section, we present the Klaim specification of the robots' behaviour informally introduced in Section 2. Moreover, to formally analyse the behaviour, we specify the low-level details about the robots and the arena where the robots move, i.e., the obstacles, the goal, etc. We use Klaim to model also these aspects because, on the one hand, the language is expressive enough to suitably represent them and, on the other hand, this approach enables the use of existing tools for the analysis of Klaim specifications.

Here, we focus only on the *qualitative* aspects of the scenario. In the next section, our specification will be enriched with *quantitative* aspects by simply associating a rate to each Klaim action, thus obtaining a StoKlaim specification.

The scenario model. The overall scenario is rendered in Klaim by the following net

$$\begin{array}{l}
robot_1 ::_{\{\mathbf{self}\mapsto robot_1\}}\ A_{behaviour} \mid C_{robotData\,1} \\
\parallel\ robot_2 ::_{\{\mathbf{self}\mapsto robot_2\}}\ A_{behaviour} \mid C_{robotData\,2} \\
\parallel\ robot_3 ::_{\{\mathbf{self}\mapsto robot_3\}}\ A_{behaviour} \mid C_{robotData\,3} \\
\parallel\ env ::_{\{\mathbf{self}\mapsto env, r_1\mapsto robot_1, r_2\mapsto robot_2, r_3\mapsto robot_3\}}\ A_{light} \mid A_{distance} \mid A_{communication} \mid A_{move} \mid C_{envData}
\end{array}$$

which is graphically depicted in Fig. 2. The three robots are modelled as three Klaim nodes whose locality names are $robot_1$, $robot_2$ and $robot_3$. Similarly, the environment around the robots is rendered as a node, with locality name *env*, as well. The allocation environment of each robot node contains only the binding for **self** (i.e., $\mathbf{self} \mapsto robot_i$), while the allocation environment of the *env* node contains the binding for **self** (i.e., $\mathbf{self} \mapsto env$) and the bindings for the robot nodes (i.e., $r_i \mapsto robot_i$, with $i \in \{1, 2, 3\}$).

The behaviour is rendered as a process identified by $A_{behaviour}$, which is exactly the same in all three robots. The items of *local knowledge* data $C_{robotData\,i}$ of each robot are stored in the tuple space of the corresponding node and consist of sensor readings and computed data; at the outset, such data are the sensor readings at the initial position.

The processes running on the *env* node provide environmental data to the robots' sensors and keep this information up-to-date as time goes by according to the actions performed by the robots' actuators. The process A_{light}, given the position of the light source and the current position of the robots, periodically computes the information about the light position for each robot and sends it to them. This data corresponds to the values obtained from light sensors and is stored in the tuple space of each robot. Similarly, the process $A_{distance}$ provides each robot with information about the obstacles around it. The process $A_{communication}$ models the communication infrastructure and, hence, takes care of delivering the messages sent by the robots by means of their range and bearing communication systems. Finally, the process A_{move} periodically updates the robots' positions according to their directions.

The data within the *env* node can be *static*, such as the information about the obstacles and the source of light, or *dynamic*, such as the robots' positions. The tuples $C_{envData}$ are stored in the tuple space of this node and their meaning is as follows (as usual, strings are enclosed within double quotes): $\langle$"*pos*"$, x_1, y_1, x_2, y_2, x_3, y_3\rangle$ represents the positions (x_1, y_1), (x_2, y_2) and (x_3, y_3) of the three robots; $\langle$"*light*"$, x_l, y_l, i\rangle$ represents a light source, with intensity i and origin in (x_l, y_l), indicating the goal position; $\langle$"*obstacles*"$, m\rangle$ indicates the total number of obstacles present in the environment (this permits simplifying the scanning of obstacles data in the Klaim specification); and $\langle$"*obs*"$, n, x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4\rangle$ represents the n-th rectangular-shaped obstacle, with vertices (x_1, y_1), (x_2, y_2), (x_3, y_3) and (x_4, y_4).

It is worth noticing that, while the Klaim process $A_{behaviour}$ is intended to model the actual robot's behaviour (e.g., it could be used as a skeleton to generate the code of the behaviour), the Klaim processes and data representing the robots' running environment (i.e., sensors, actuators, obstacles, goal, etc.) are just models of the environment and of physical devices, which are needed to enable the analysis.

The robot model. Each robot executes a behaviour that interacts with the robot's tuple space for reading and producing sensors and actuators data to cyclically perform the following activities: *sensing* data about the local environment, *elaborating* the retrieved knowledge data to make decisions, and *acting* according to the elaborated decisions (i.e., it transmits data to other robots and actuates the wheels to perform a movement).

Different choices can be made when developing the model of the robot [18]. We have chosen to model individually the behaviour of each robot and the corresponding sensors and actuators. We illustrate in this section the data associated to the robots' sensors and actuators, and the Klaim specification of the robots' behaviour (due to lack of space, the rest of the specification is relegated to a companion technical report [14]).

Robots' sensor and actuator data. The *light sensor* data is rendered in Klaim as a tuple of the form $\langle$"*light*"$, \boldsymbol{\ell}\rangle$, where "*light*" is a *tag* indicating the sensor originating the data while $\boldsymbol{\ell}$ is an array of 24 elements. For each $i \in [0, 23]$, $\boldsymbol{\ell}[i]$ represents the light intensity perceived by the sensor along the direction $2\pi\frac{i}{24}$.

The tuple containing the measures of the *distance scanner sensor* is similar. Indeed, it is of the form $\langle$"*obs*"$, \mathbf{d}\rangle$, where "*obs*" is the tag associated to *distance scanner sensor* data and $\mathbf{d}$ is an array of 24 elements. For each $i \in [0, 23]$, $\mathbf{d}[i]$ is the distance to the closest obstacle measured by the sensor along the direction $2\pi\frac{i}{24}$.

The *range and bearing communication system* acts as both a sensor and an actuator. Indeed, it allows a robot to send messages to other robots in its neighborhood and to receive messages sent by them. A process running in the environment node is used to read (and consume) the messages produced by each robot's behaviour and to *route* them to the other robots (through the environment node). This process models the communication medium and specifies the range and bearing communication system without considering explicitly the details of the underlying communication framework. Each robot stores received messages in a local tuple of the form $\langle$"*msgs*"$, [m_1, m_2, \ldots, m_n]\rangle$ representing a queue of length n containing messages m_1,m_2,...,m_n[4]. Instead, to send a message to other robots, a behaviour locally stores a tuple of the form $\langle$"*msg*"$, m\rangle$. The process running on the environment node is in charge of reading each message and propagating it to the other robots that are in the sender's communication range.

Finally, the *wheel actuators* are rendered as a process running in the environment node that reads the new directions to be followed by the robots (i.e., tuples of the form $\langle$"*move*"$, \theta\rangle$) and updates the robots' position (which is, in fact, an information stored in the tuple space of the environment node). This slightly differs from the original specification given in Section 2, where the *Motion Control* module converts the direction calculated by the *Direction Mediation* module into speeds for the two wheels, which are then passed to the wheels actuator. In fact, although our specification is quite detailed, it is still an abstract description of a real-world scenario. Thus, some details that do not affect the analysis, such as those involving the calculation of the robots' movements, are considered at an higher level of abstraction.

For simplicity's sake, we do not consider noise and failures of sensors and actuators.

Robot's behaviour. The process $A_{behaviour}$, modelling the robot's behaviour graphically depicted in Fig. 1, is defined as follows:

$$A_{behaviour} \triangleq A_{goalDirection} \mid A_{obstacleAvoidance} \mid A_{directionArbiter} \mid A_{directionMediation} \mid A_{motionControl}$$

Each behavioural module (i.e., a yellow blob in Fig. 1) corresponds to one of the above Klaim processes, whose definitions are provided below. The specification code is made self-explanatory through the use of comments (i.e. strings starting with //).

The *Goal Direction* module takes as input the last light sensors readings (here rendered as a tuple of the form $\langle$"*light*"$, \boldsymbol{\ell}\rangle$) and returns the vector $\mathbf{v}^{gd}$ (actually, here only the direction of $\mathbf{v}^{gd}$ is returned, because its length is always 1 and does not play any role on the computation of the new direction to be followed). This behavioural module is modelled by the recursive process $A_{goalDirection}$ defined as follows:

```
$A_{goalDirection} \triangleq$
$x_{sum}$ , $y_{sum}$ := 0;
**read**("light", $!\ell$)@**self**;  // read the tuple containing the light sensor readings
**for** $i = 0$ **to** 23{
    $x_{sum}$ := $x_{sum} + \ell[i] \cdot \cos(2\pi i/24)$;  // calculate the coordinates of the final point of the...
    $y_{sum}$ := $y_{sum} + \ell[i] \cdot \sin(2\pi i/24)$;  // ... vector (with the origin as initial point) resulting...
};                                                  // ... from the vectorial sum of the reading vectors
```

[4] $[v_1, \ldots, v_n]$ denotes a list of n elements, [] the empty list, and :: the concatenation operator.

```
if (($x_{sum}$ ! = 0) $\wedge$ ($y_{sum}$ ! = 0)) then {   // check if the light is perceived
    $\angle \mathbf{v}^{gd}$ := $Angle(0, 0, x_{sum}, y_{sum})$;  // calculate $\angle \mathbf{v}^{gd}$, i.e., the direction of vector $\mathbf{v}^{gd}$
    rpl("vgd", _) → ("vgd", $\angle \mathbf{v}^{gd}$)@self;  // update the vector $\mathbf{v}^{gd}$ data
} else {
    inp("vgd", _)@self  // if the light is not preceived, remove the previous vector $\mathbf{v}^{gd}$ data
}; $A_{goalDirection}$
```

The sensor readings are always present in the tuple space, because they are present at the outset and the processes modelling behavioural modules do not consume sensor data while reading or updating them. Therefore, the **read** action before the **for** loop above never blocks the execution of process $A_{goalDirection}$. In principle, by pooling the tuple space in this way, the same sensor data could be read more than once; this faithfully reflects the actual interaction model between the robots code and the sensors. Anyway, it does not lead to divergent behaviours during the analysis, because such interactions are regulated by the action rates specified in the StoKlaim model (see Section 5).

The function $Angle(x_0, y_0, x_1, y_1)$, used above and in subsequent parts of the specification, returns the direction (i.e., the angle) of the vector from (x_0, y_0) to (x_1, y_1). We refer the interested reader to [13] for its definition.

The *Obstacle Avoidance* module takes as input the last distance sensors readings (here rendered as a tuple of the form $\langle$"*obs*", $\mathbf{d}\rangle$) and returns the vector $\mathbf{v}^{oa}$. This behavioural module is modelled by the process $A_{obstacleAvoidance}$ defined as follows:

```
$A_{obstacleAvoidance} \triangleq$
$x_{sum}$ , $y_{sum}$ := 0;  $min$ := $obs\_d_{MAX}$;
read("obs", !d)@self;  // read the tuple representing the distance sensor readings
for $i = 0$ to 23{
    $x_{sum}$ := $x_{sum} + d[i] \cdot \cos(2\pi i/24)$; // calculate the coordinates of the final point of the...
    $y_{sum}$ := $y_{sum} + d[i] \cdot \sin(2\pi i/24)$;  // ... vectorial sum of the reading vectors
    if ($d[i] < min$) then  $min$ := $d[i]$  // calculate the minimum length of the vectors
};
$\| \mathbf{v}^{oa} \|$ := $min/obs\_d_{MAX}$;  // calculate $\| \mathbf{v}^{oa} \|$, i.e., the length of vector $\mathbf{v}^{oa}$ rescaled in [0, 1]
$\angle \mathbf{v}^{oa}$ := $Angle(0, 0, x_{sum}, y_{sum})$;  // calculate $\angle \mathbf{v}^{oa}$, i.e., the direction of vector $\mathbf{v}^{oa}$
rpl("voa", _, _) → ("voa", $\| \mathbf{v}^{oa} \|$, $\angle \mathbf{v}^{oa}$)@self;  // update the vector $\mathbf{v}^{oa}$ data
$A_{obstacleAvoidance}$
```

where obs_d_{MAX} is the maximum range of the distance sensor (in [10], it is set to 1.5 m).

The process $A_{directionArbiter}$ modelling the *Direction Arbiter* module, which takes $\mathbf{v}^{gd}$ and $\mathbf{v}^{oa}$ as input and returns the direction θ^{da}, is defined as follows:

```
$A_{directionArbiter} \triangleq$
in("voa", !voa_l, !$\theta^{oa}$)@self; // read and consume the tuple containing $\mathbf{v}^{oa}$ (it's always present)
if (inp("vgd", !$\theta^{gd}$)@self) then {  // read and consume the tuple containing $\mathbf{v}^{gd}$ (if available)
    $v^{da}_x$ := $(1 - voa\_l) \cdot cos(\theta^{oa}) + voa\_l \cdot cos(\theta^{gd})$;  // calculate the coordinates of the...
    $v^{da}_y$ := $(1 - voa\_l) \cdot sin(\theta^{oa}) + voa\_l \cdot sin(\theta^{gd})$;   // ... vector to the desired direction
    $\theta^{da}$ := $Angle(0, 0, v^{da}_x, v^{da}_y)$;  // compute the angle $\theta^{da}$
    rpl("da", _) → ("da", $\theta^{da}$)@self;  // update the angle $\theta^{da}$ data
} else {
    if ($voa\_l < 1$) then {  // check if any obstacle has been detected
        rpl("da", _) → ("da", $\theta^{oa}$)@self  // use the obstacle avoidance direction as $\theta^{da}$
    }
}; $A_{directionArbiter}$
```

Notably, differently from sensor readings, data produced by other modules (e.g. $\mathbf{v}^{gd}$ and $\mathbf{v}^{oa}$) are removed from the tuple space when read.

The *Direction Mediation* module takes as input the direction θ^{da} computed by the *Direction Arbiter* and the last received messages (here rendered as a tuple of the form $\langle$"*msgs*", $[m_1, m_2, \ldots, m_n]\rangle$) and returns the direction θ^{dn}, to be used by the *Motion Control* module, and a message m, to be sent to the other robots via the range and bearing system. The *Direction Mediation* module is modelled by the following process:

```
$A_{directionMediation} \triangleq$
$c\,,\ sum_x\,,\ sum_y\ :=\ 0;$
**rpl**("msgs", !l) → ("msgs", [])@**self**;  // read and reset the list of received messages
**while** ($l == \theta :: tail$) {  // scan the list
   $l\ :=\ tail;$
   $sum_x\ :=\ sum_x + cos(\theta);$  // calculate the sum of the received...
   $sum_y\ :=\ sum_y + sin(\theta);$  // ...directions
   $c\ :=\ c + 1$  // increase the counter of the received messages
};
**if** ($c == 0$) **then** {  // check if there are received messages
   **if** (**inp**("da", !$\theta^{da}$)@**self**) **then** {  // if there aren't, check if the robot is informed
      **rpl**("dir", _) → ("dir", $\theta^{da}$)@**self**;  // update the direction data for the motion control
      **rpl**("msg", _) → ("msg", $\theta^{da}$)@**self** // update the message to be sent to the other robots
   }
} **else** {                                   // if there are received messages,...
   $\theta^{dn}\ :=\ Angle(\ 0,\ 0,\ sum_x,\ sum_y\ );$  // ...calculate the average direction and proceed
   (  **rpl**("dir", _) → ("dir", $\theta^{dn}$)@**self**    // update the data for the motion control
      |
      ( **if** (**inp**("da", !$\theta^{da}$)@**self**) **then** {  // check if the robot is informed
            $m\ :=\ \theta^{da}$  // if it is, the produced message contains $\theta^{da}$
        } **else** {
            $m\ :=\ \theta^{dn}$  // if the robot is not informed, the produced message contains $\theta^{dn}$
        };
        **rpl**("msg", _) → ("msg", m)@**self** // update the message to be sent to the other robots
      )
   )
}; $A_{directionMediation}$
```

Notice that the tuple containing the direction θ^{da} is consumed when read. Thus, to avoid blocking the execution of the process, to read such tuple an action **inp** (within the condition of an **if** construct) is exploited.

The *Motion Control* module takes as input the direction computed by the *Direction Mediation* module and transmits it to the wheels actuator. The *Motion Control* module is modelled by the following process:

```
$A_{motionControl} \triangleq$
**in**("dir", !$\theta^{dn}$)@**self**;  // wait (and consume) a direction of movement
**rpl**("move", _) → ("move", $\theta^{dn}$)@**self**;  // transmit the direction to the wheels actuator
$A_{motionControl}$
```

As previously explained, we do not need to model the conversion of the direction calculated by the *Direction Mediation* module into speeds for the wheels.

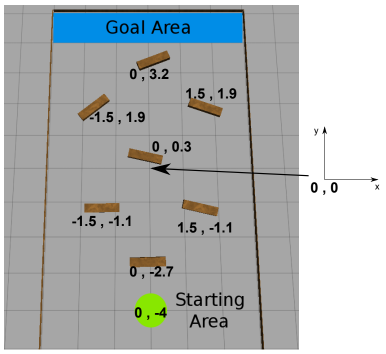


Fig. 3. Arena and initial configuration

5 Stochastic specification and analysis

In this section, we demonstrate how the Klaim specification presented n the previous section can support the analysis. The proposed methodology can be used to verify the system success, obtaining accurate estimations of the system performance expressed in terms of the probability of reaching the goal without entering unwanted intermediate states. This permits making comparisons of the algorithm performance in different scenarios, which may use different features of the obstacles, transported objects, terrain, and also different goals and requirements. It also permits to analyse different details of the system behaviour, which can provide helpful information for understanding the reasons why an unwanted behaviour is observed and can allow the system designer to tune the system in order to improve its performance under different conditions. Our approach relies on formal tools, like stochastic modal logics and model checking, that permits expressing and evaluating performance measures in terms of logical formulae. In this way, we obtain a framework for the analysis of collective robotic systems which is more abstract and expressive than existing simulation frameworks, where the analysis is typically performed by relying on an *a posteriori* data analysis. Moreover, for the sake of efficiency, simulators are usually deterministic, e.g. all robots act synchronously. This means that some possible behaviours of a real system are not taken into account in the simulation. Instead, stochastic modelling tools permit considering the typical *uncertainty* of real systems, e.g. by abstracting from the precise scheduling of robot movements. In this way, developers are guaranteed that their analyses cover more critical situations of the considered scenario, according to a given margin of error.

We now enrich the Klaim specification introduced in the previous section with stochastic aspects and consider the scenario configuration presented in [10] and depicted in Fig. 3. Seven rectangular objects are scattered in the arena, while the light source is positioned high above the goal area and is always visible to the robots. We assume that robots, on average, are able to perform 10 sensor readings per second and that they have an average speed of $2cm/sec$, and let the part of the specification modeling the environment be able to perform a mean of 100 operations per second. Starting from these parameters we have derived specific rates for defining the StoKlaim specification. As an excerpt of the StoKlaim specification, we report below the stochastic definition of process $A_{obstacleAvoidance}$:

$A_{obstacleAvoidance} \triangleq$
$x_{sum}, y_{sum} := 0;\ min := obs\_d_{MAX};$
$\mathbf{read}(\text{"}obs\text{"}, !d)@\mathbf{self} : \lambda_1$;
$\mathbf{for}\ i = 0\ \mathbf{to}\ 23\{\dots\};\ \|\mathbf{v}^{oa}\| := min/obs\_d_{MAX};\ \angle\mathbf{v}^{oa} := Angle(0, 0, x_{sum}, y_{sum});$
$\mathbf{rpl}(\text{"}voa\text{"}, \_, \_) \rightarrow (\text{"}voa\text{"}, \|\mathbf{v}^{oa}\|, \angle\mathbf{v}^{oa})@\mathbf{self} : \lambda_2$;
$A_{obstacleAvoidance}$

The actions highlighted by a gray background are those annotated with rates λ, where $\lambda_1 = 24.0$ and $\lambda_2 = 90.0$. These rates guarantee that obstacle avoidance data are updated every $\frac{1}{24} + \frac{1}{90}$ time units on average, i.e. about 20 times per second. We refer the interested reader to [13] for the rest of the stochastic specification.

The result of a simulation run of the StoKlaim specification, performed by using Sam, is reported in Fig. 4 (a). The trajectories followed by the three robots in this run are plotted in the figure with three different colors; they show that the robots reach the goal without collisions. On an Apple iMac computer (2.33 GHz Intel Core 2 Duo and 2 GB of memory) simulation of a single run needs an average time of 123 seconds.

We have analysed the probability to reach the goal without colliding with any obstacles. The property "*robots have reached the goal area*" is formalized in MoSL, for the specific system under analysis, by the formula ϕ_{goal} defined below:

$$\phi_{goal} = \langle\text{"}pos\text{"}, !x_1, !y_1, !x_2, !y_2, !x_3, !y_3\rangle@env \rightarrow y_1 \geq 4.0 \wedge y_2 \geq 4.0 \wedge y_3 \geq 4.0$$

This formula relies on *consumption* operator, $\langle T\rangle@l \rightarrow \phi$, that is satisfied whenever a tuple matching template T is located at l and the remaining part of the system satisfies ϕ. Hence, formula ϕ_{goal} is satisfied if and only if tuple $\langle\text{"}pos\text{"}, x_1, y_1, x_2, y_2, x_3, y_3\rangle$, where each y_i is greater than 4.0, is in the tuple space located at *env* (all robots are in the goal area). Similarly, the property "*a robot collided an obstacle*" is formalized by:

$$\phi_{col} = \langle\text{"}collision\text{"}\rangle@env \rightarrow \mathsf{true}$$

where tuple $\langle\text{"}collision\text{"}\rangle$ is located at *env* whenever a robot collided an obstacle.

The considered analyses have been then performed by estimating the total probability of the set of runs satisfying $\neg\phi_{col}U^{\leq t}\phi_{goal}$ where the formula $\phi_1 U^{\leq t}\phi_2$ is satisfied by all the runs that reach within t time units a state satisfying ϕ_2 while only traversing states that satisfy ϕ_1. In the analysis, a time-out of 500*sec* has been considered.

Under the configuration of Fig. 3, i.e. when the robots are always able to perceive the light, we get that the goal can be reached without collisions with probability 0.916, while robots do not reach the goal or collide with obstacles with probability 0.084 (these values have been estimated with parameters $p = 0.1$ and $\varepsilon = 0.1$, 1198 runs). Such results are in accordance with those reported in [10], where the estimated probability to reach the goal is 0.94. The slight variation is mainly due to a different way of rendering robots movement, which is computed via a *physical simulator* in [10], while in our case it is approximated as the vectorial sum of the movement of each single robot.

We have then modified the original scenario by locating the light source on the same plane of the arena and we noticed that the overall system performances are deeply influenced. Indeed, since objects cast shadows, they can prevent robots from sensing

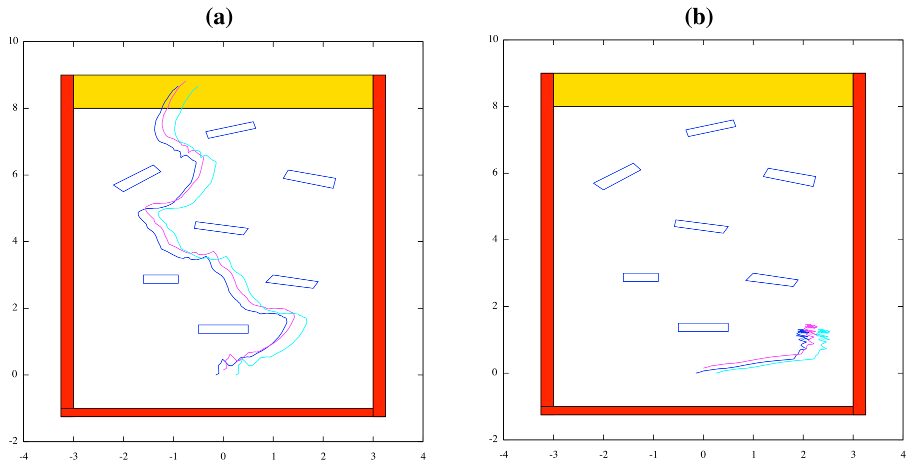


Fig. 4. Some simulation results obtained for the robotics scenario from [10]

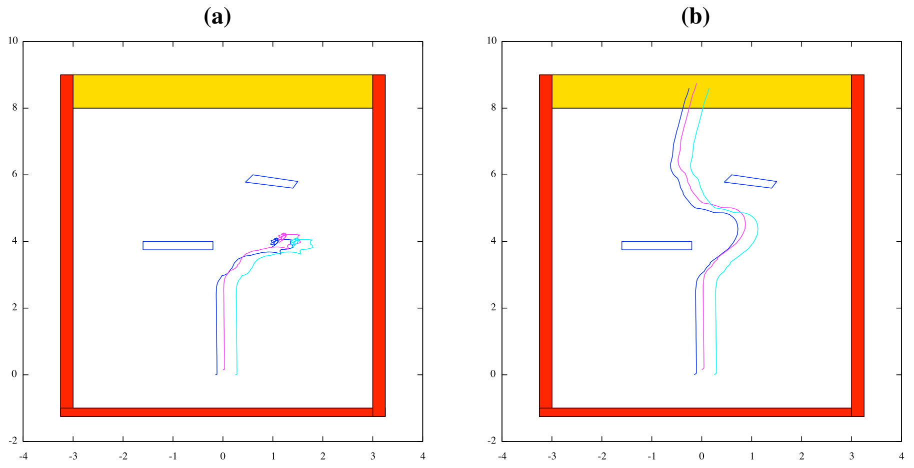


Fig. 5. Some simulation results obtained for a simple robotics scenario

the light. Under this configuration, the robots are not able to reach the goal area (see Fig. 4 (b) for a simulation trace representing a sort of counterexample for the given property) and the probability to reach the goal without collisions plummets to 0.0.

In order to validate our model and to verify the robots' behaviour, we have also considered other scenarios. In Fig. 5 (a) we show a simpler scenario where just two obstacles are placed at the center of the arena. At the beginning, the obstacles do not hide the light to the robots. However, when the first obstacle enters in the range of the robots' distance sensors, the robots turn to right, enter in the shadow cast by the second object and then never reach the goal area. This problem can be avoided by modifying the robot behaviour so that, when the light is not perceived, the last known goal direction is used. The adoption of this simple policy increases the probability to reach the goal area without collisions, from 0.234 to 1.0 (see the simulation run in Fig. 5 (b)).

6 Concluding remarks

We have presented a novel approach to formal verification of collective robotic systems and validated it against a traditional approach consisting in physics-based simulations. We have shown that the obtained results are in accordance with those resulting from physics-based simulations and reported in [10], which have been in fact exploited for tuning the quantitative aspects of our analysis (e.g. the robots' actions execution time).

Our approach paves the way for the definition of a 5-step engineering methodology based on formal foundations. In the first step, the designer models the system formally with KLAIM. In the second step, he adds stochastic aspects to enable its analysis and analyses the system properties to discover flaws in the formal model. These two steps can be iterated, allowing the designer to discover and fix flaws of the system even before the actual code is written. In the third step, the specification is converted into (the skeleton of) the robots behaviour code. In the fourth step, the code is tested with physics-based simulations, to reveal further model-worthy aspects that were neglected in the first two steps. Finally, in the fifth step, robots behaviour is tested on real robots. The focus of this paper is on the definition of the first two steps.

We believe that the development methodology we envisage has many advantages when compared with ad-hoc design and validation through physics-based simulation and experimentation with real robots. Indeed, it permits to formally specify the requirements of the system and to minimize the risk of developing a system that does not satisfy the required properties, as these properties are checked at each step of the development phase. It also permits detecting potential design flaws early in the development process thus reducing the cost of later validation efforts.

Depending on the complexity of the system to develop, implementing the model for enabling physics-based simulation might not be straightforward and could require the ingenuity and expertise of the developer. Therefore, we intend to define and implement an automatic translation from KLAIM specifications of robot behaviours to actual code that can be taken as input by the physics-based simulator. This would allows us to complete the 5-step development process mentioned above. We also plan to apply our approach to other challenging robotic scenarios, by studying the performance of different robot behaviours while changing environmental conditions and system requirements.

Moreover, we intend to consider more abstract system specifications to conveniently deal with swarm robotics scenarios. In fact, to enable the verification of the class of properties we deemed interesting for the collective robotics domain, we have defined a very detailed model of the system under analysis. Indeed, the model we propose permits taking into account, during the analysis process, the exact position of each robot, as well as any other information about its internal state, at each instant of time. The model fits well with collective transport scenarios, where usually a limited number of robots are involved; however, it may become not tractable using available tools when the number of robots significantly grows. To deal with such kind of scenarios, like e.g. the swarm robotics one, the abstraction level of the model has to be gradually raised up in accordance with an increasing number of robots, by focussing on those aspects of the system that become most relevant. This approach would be reasonable in case of swarms, because the properties of interest are no longer related to the exact position of each single robots, but concern the global (abstract) behaviour of the overall system.

References

1. L. Bettini, R. De Nicola, and R. Pugliese. Klava: a Java Package for Distributed and Mobile Applications. *Software - Practice and Experience*, 32(14):1365–1394, 2002.
2. M. Bonani et al. The marxbot, a miniature mobile robot opening new perspectives for the collective-robotic research. In *IROS*, pp. 4187–4193. IEEE, 2010.
3. M. Brambilla, C. Pinciroli, M. Birattari, and M. Dorigo. Property-driven design for swarm robotics. In *AAMAS*. IFAAMAS, 2012. To appear.
4. R. Bruni et al. Modelling and analyzing adaptive self-assembling strategies with maude. In *WRLA*, LNCS. Springer, 2012. To appear.
5. F. Calzolai, and M. Loreti. Simulation and Analysis of Distributed Systems in Klaim. In *COORDINATION*, *LNCS* 6116, pp. 122–136. Springer, 2010.
6. E. M. Clarke, and J. M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28:626–643, December 1996.
7. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *Transactions on Software Engineering*, 24(5):315–330, 1998.
8. R. De Nicola, J. Katoen, D. Latella, M. Loreti, and M. Massink. Model checking mobile stochastic logic. *Theor. Comput. Sci.*, 382(1):42–70, 2007.
9. C. Dixon, A. Winfield, and M. Fisher. Towards Temporal Verification of Emergent Behaviours in Swarm Robotic Systems. In *TAROS*, *LNCS* 6856, pp. 336–347. Springer, 2011.
10. E. Ferrante, M. Brambilla, M. Birattari, and M. Dorigo. Socially-Mediated Negotiation for Obstacle Avoidance in Collective Transport. In *DARS*, STAR. Springer, 2010. To appear.
11. M. Fisher, and M. Wooldridge. On the formal specification and verification of multi-agent systems. *Int. Journal of Cooperative Information Systems*, 6(1):37–66, 1997.
12. A. Galstyan, T. Hogg, and K. Lerman. Modeling and Mathematical Analysis of Swarms of Microscopic Robots. In *SIS*, pp. 201–208. IEEE, 2005.
13. E. Gjondrekaj, M. Loreti, R. Pugliese, and F. Tiezzi. Specification and Analysis of a Collective Robotics Scenario in SAM, 2011. SAM source file available at `http://rap.dsi.unifi.it/SAM/`.
14. E. Gjondrekaj et al. Towards a formal verification methodology for collective robotic systems. Technical report, Univ. Firenze, 2011. `http://rap.dsi.unifi.it/~loreti/papers/collective_transport_verification.pdf`.
15. S. Jeyaraman et al. Formal techniques for the modelling and validation of a co-operating UAV team that uses Dubins set for path planning. In *ACC* 7, pp. 4690–4695. IEEE, 2005.
16. C. Jones, and M.J. Mataric. Synthesis and analysis of non-reactive controllers for multi-robot sequential task domains. In *ISER*, *STAR* 21, pp. 417–426. Springer, 2004.
17. S. Konur, and C. Dixon. Formal verification of probabilistic swarm behaviours. In *ANTS*, number 6234 in LNCS, pp. 572–573. Springer, 2010.
18. K. Lerman, A. Martinoli, and A. Galstyan. A Review of Probabilistic Macroscopic Models for Swarm Robotic Systems. In *SAB*, *LNCS* 3342, pp. 143–152. Springer, 2005.
19. M. Loreti. SAM: Stochastic Analyser for Mobility. `http://rap.dsi.unifi.it/SAM/`.
20. J.A. Stankovic. Strategic directions in real-time and embedded systems. *ACM Comput. Surv.*, 28:751–763, December 1996.
21. A. Winfield et al. On Formal Specification of Emergent Behaviours in Swarm Robotic Systems. *Int. Journal of Advanced Robotic Systems*, 2(4):363–370, 2005.