Collective Sorting Tuple Spaces

Matteo Casadei, Luca Gardelli, Mirko Viroli
DEIS, Cesena
Alma Mater Studiorum – Università di Bologna
via Venezia 52, 47023 Cesena (FC), Italy
{m.casadei,luca.gardelli,mirko.viroli}@unibo.it

Abstract— **Coordination of multiagent systems is recently moving towards the application of techniques coming from the research context of complex systems: adaptivity and self-organisation are exploited in order to tackle openness, dynamism and unpredictability of typical multiagent systems applications. In this paper we focus on a coordination problem called *collective sorting*, where autonomous agents are assigned the task of moving tuples across different tuple spaces according to local criteria, resulting in the emergence of the complete clustering property. Using a library we developed for the MAUDE term rewriting system, we simulate the behaviour of this system and evaluate some solutions to this problem.**

I. INTRODUCTION

Systems that should self-organise to unpredictable changes in their environment very often need to feature adaptivity as an emergent property. As this observation was first made in the context of natural systems, it was shortly recognised as an inspiring metaphor for artificial systems as well [1]. However, a main problem with emergent properties is that, by their very definition, they cannot be achieved through a systematic design: their dynamics and outcomes cannot be fully predicted. Nonetheless, providing some design support in this context is still possible. The whole system of interest, that is the application to design and the environment it is immersed in, can be modelled as a stochastic system, namely, a system whose dynamics and duration aspects are probabilistic. In this scenario, simulations can be run and used as a fruitful tool to predict certain aspects of the system behaviour, and to support a correct design before actually implementing the application at hand [2].

This scenario is particularly interesting for agent coordination. Some works like the TOTA middleware [3], SwarmLinda [4], and stochastic KLAIM [5], though starting from different perspectives, all develop on the idea of extending standard coordination models with features related to adaptivity and self-organization. They share the idea that tuples in a tuple space eventually spread to other tuple spaces in a non-deterministic way, depending on certain timing and probability issues. Accordingly, in this paper we start analysing the potential role that simulation tools can have in this context, towards the identification of some methodological approach to system design.

As a reference example, we consider an application to a tuple space scenario of the so-called *collective sorting* problem for swarm intelligence [1]. This application features autonomous agents managing a set of distributed tuple spaces, with the goal of moving tuples from one space to the other until completely "sorting" them, that is, tuples of different types reside in different tuple spaces. We show a solution to this problem based on a fully-distributed algorithm, where each agent moves tuples according to fully-local criteria, and where complete sorting appear to emerge from initial chaotic tuple configurations. To provide evidence of correctness and appropriateness we rely on simulations.

Many simulation tools can be exploited to this end, though they all necessarily force the designer to exploit a given specification language, and therefore better apply to certain scenarios and not to others—examples are SPIM [6], SWARM [7] and REPAST [8]. Instead of relying on one of them, in this paper we seek for a general-purpose approach. We evaluate the applicability of the MAUDE specification tool as a general-purpose engine for running simulations [9]. It is very well known that MAUDE allows for modelling syntactic and dynamic aspects of a system in a quite flexible way, supporting e.g. process algebraic, automata, and net-like specifications—all of which can be seen as instantiations of MAUDE's term rewriting framework. We developed a library for allowing a system designer to specify in a custom way a system model in terms of a stochastic transition system—a labelled transition system where actions are associated with a *rate* (of occurrence) [10]. One such specification is then exploited by the tool to perform simulations of the system behaviour, thus making it possible to observe the emergence of certain (possibly unexpected) properties.

The remainder of this paper is as follows: Section 2 provides some background on coordination techniques featuring adaptivity, Section 3 describes the collective sorting problem, while Section 4 presents the MAUDE model of the Collective Sorting and its simulation results, and finally Section 5 concludes providing perspectives on future works.

II. BACKGROUND

In the effort to improve the design process of software systems—i.e. to bridge the gap between the design and the actual implementation—it has become very common practice to take into account not only functional and architectural requirements, but also quantitative aspects like temporal and probabilistic ones. When dealing with complex systems, it is often the case that aleatory in system dynamics may cause the emergence of interesting properties, that cannot therefore be abstracted away when designing the system. Coordination models and technologies for multiagent systems are witnessing the development of a number of works moving to this direction, most of which are inspired by natural phenomena.

A first example is the TOTA (Tuples On The Air) middleware [3] for pervasive computing applications, inspired by the concept of field in physics—like e.g. the gravitational or magnetic fields. This middleware supports the concept of "spatially distributed tuple": that is, a tuple can be cloned and spread to the tuple spaces in the neighborhood, creating a sort of computational field, which grows when initially pumped and then eventually fades. To this end, when injected in a tuple space, each tuple can be equipped by some application-dependent rules, defining how a tuple should spread across the network, how the content of the tuple should be accordingly affected, and so on. TOTA is mainly targeted to support multiagent systems whose environment is open, dynamic and unpredictable, like e.g. to let mobile agents meet each other in a dynamic network.

Another example is the SwarmLinda coordination model [4], which though similar to TOTA is more inspired by swarm intelligence and stigmergy [1], [11], [12]. In SwarmLinda tuples are moved from one tuple space to the other, and ant-like algorithms are used to retrieve them. The use of self-techniques in SwarmLinda derives from necessity of dealing with openness and with the unpredictability of a tuple space's users, against the need of achieving adaptivity.

Finally, the "swarm robotics" field applies strategies inspired by social insects in order to coordinate the activities of a multiplicity of robots systems. Typically, these systems are built on top of ad-hoc software middlewares [1], and solve problems with distributed-algorithms where, though each robot brings about very simple goals, the whole system can be used to solve quite complex problems—see e.g. the collective sorting problem in Section III-A.

These are all examples witnessing the fact that coordination in open, dynamic, and unpredictable systems have quantitative aspects playing a very important role. This calls for analysis and design tools that can support system development at various levels, from formal specification up to simulations.

III. Collective Sorting

A. General Scenario

We consider a case of Swarm-like intelligence known as *collective sorting* [1]. It features a multiagent system where the environment is structured and populated with items of different kinds: the goal of agents is to collect and move items across the environment so as to order them according to an arbitrary shared criterion. This problem basically amounts to clustering: homogeneous items should be grouped together and should be separated from others. Moving to a typical context of coordination models and languages, we consider the case of a fixed number of tuple spaces hosting tuples of a known set of tuple types. The goal of agents is to move tuples from one tuple space to the other until the tuples are clustered in different tuple spaces according to their tuple type.

In several scenarios, sorting tuples may increase the overall system efficiency. For instance, it can make it easier for an agent to find an information of interest based on its previous experience: the probability of finding an information where a previous and related one was found is high. Moreover, when tuple spaces contain tuples of one kind only, it is possible to apply aggregation techniques to improve their performance, and it is generally easier to manage and achieve load-balancing.

Increasing system order however comes at a computational price. Achieving ordering is a task that should be generally performed online and in background, i.e. while the system is running and without adding a significant overhead to the main system functionalities. Indeed, it might be interesting to look for suboptimum algorithms, which are able to guarantee a certain degree of ordering in time.

Nature is a rich source of simple but robust strategies: the behaviour we are looking for has already been explored in the domain of social insects. Ants perform similar tasks when organizing broods and larvae: this class of coordination strategies are generally referred to as *collective sorting* or *collective clustering* [1]. Although the actual behaviour of ants is still not fully understood, there are several models that are able to mimic the dynamics of the system. Ants wander randomly and their behaviour is modelled by two probabilities, respectively, the probability to pick up P_p and drop P_d an item

$$P_p = \left(\frac{k_1}{k_1 + f}\right)^2, \quad P_d = \left(\frac{f}{k_2 + f}\right)^2, \qquad (1)$$

where k_1 and k_2 are constant parameters and f is the number of items perceived by an ant in its neighborhood: f may be evaluated with respect to the recently encountered items. To evaluate the system dynamics, apart from visualising it, it can be useful to provide a measure of the system order. Such an estimation can be obtained by measuring the spatial entropy, as done e.g. in [11]. Basically, the environment is subdivided into nodes and P_i is the fraction of items within a node, hence the local entropy is $H_i = -P_i \log P_i$. The sum of H_i having $P_i > 0$ gives an estimation of the order of the entire system, which is supposed to decrease in time, hopefully reaching zero (complete clustering).

B. An Architecture for Implementing Collective Sorting

We conceive a multiagent system as a collection of agents interacting with/via tuple spaces: agents are allowed to read, insert and remove tuples in the tuple spaces. Additionally, and transparently to the agents, an infrastructure provides a sorting service in order to maintain a certain degree of order of tuples in tuple spaces. This service is realised by a class of agents that will be responsible for the sorting task. Hence, each tuple space is associated with a pool of agents, as shown in Figure 1, whose task is to compare the content of the local tuple space against the content of another tuple space in the environment, and possibly move some tuple. Since we want to perform this task online and in background, and with a fully-distributed, swarm-like algorithm, we cannot compute the probabilities in

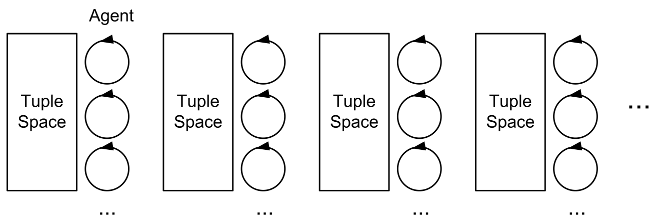

Fig. 1. The basic architecture consists in a set of sorter agents dedicated to a single tuple space.

Equation 1 to decide whether to move or not a tuple: the approach would not be scalable since it requires to count all the tuples for each tuple space, which might not be practical.

Hence, we devise a strategy based on tuple sampling, and suppose that tuple spaces provide for a reading primitive we call `urd`, *uniform read*. This is a variant of the standard `rd` primitive that takes a tuple template and yields any tuple matching the template: primitive `urd` instead chooses the tuple in a probabilistic way among all the tuples that could be returned. For instance, if a tuple space has 10 copies of tuple $t(1)$ and 20 copies of tuple $t(2)$ then the probability that operation $urd(t(X))$ returns $t(2)$ is twice as much as $t(1)$'s. As standard Linda-like tuple spaces typically do not implement this variant, it can e.g. be supported by some more expressive model like ReSpecT tuple centres [13]. When deciding to move a tuple, an agent working on the tuple space TS_S follows this agenda:

1) it draws a destination tuple space TS_D different from the source one TS_S;
2) it draws a kind k of tuple;
3) it (uniformly) reads a tuple T_1 from TS_S;
4) it (uniformly) reads a tuple T_2 from TS_D;
5) if the kind of T_2 is k and it differs from the kind of T_1, then it moves a tuple of the kind k from TS_S to TS_D.

The point of last task is that if those conditions hold, then the number of tuples k in TS_D is more likely higher than in TS_S, therefore a tuple could/should be moved. It is important that all choices are performed according to a uniform probability distribution: while in the steps 1 and 2 it guarantees fairness, in steps 3 and 4 it guarantees that the obtained ordering is appropriate.

It is worth noting that the success of this distributed algorithm is an emergent property, affected by both probability and timing aspects. Will complete ordering be reached starting from a completely chaotic situation? Will complete ordering be reached starting from the case where all tuples occur in just one tuple space? And if ordering is reached, how many moving attempts are globally necessary? These are the sort of questions that could be addressed at the early stages of design, thanks to a simulation tool.

IV. THE COLLECTIVE SORTING IN MAUDE

In this section we briefly describe a MAUDE specification of our solution to the collective sorting problem, and show simulation results. Our model sticks to the case where 4 tuple spaces exist (labelled with identifiers 0, 1, 2 and 3), and four tuple kinds are subject to ordering (`'a`, `'b`, `'c`, and `'d`).

A. A MAUDE library for simulation

MAUDE is a high-performance reflective language supporting both equational and rewriting logic specifications, for specifying a wide range of applications [9]. The basic brick of a MAUDE program is the *module*, which is essentially a set of definitions determining an algebra: the modules can be either of the *functional* or *system* kind. Functional modules contain both (syntax-customed) type and operation declarations, along with *equations* which are actually *equational rewriting* rules defining abstract data types—this is hence useful to declare algorithmic aspects of computing systems. System modules can instead have *rewriting laws* as well—i.e. transition rules—that are typically used to implement a concurrent *rewriting semantics*, and are then able to deal with aspects related to interaction and system evolution. In the course of finding a general simulation tool for stochastic systems, we find MAUDE as a particularly appealing framework, for it allows to directly model a system in terms of transition rules, or to prototype a new domain-dependent language to have more expressiveness and compact specifications.

Using MAUDE, we realized a general simulation framework for stochastic systems: the idea of this tool is to model a stochastic system by a labelled transition system where transitions are of the kind $S \xrightarrow{r:a} S'$, meaning that the system in state S can move to state S' by action a, where r is the *(global) rate* of action a in state S. The rate of an action in a given state can be understood as the number of times action a could occur in a time-unit (if the system would rest in state S), namely, its occurrence frequency. This idea is inspired by the activity mechanism of stochastic π-Calculus [10], where each channel is given a fixed local rate, and the global rate of an interaction is computed as the channel rate multiplied by the number of processes willing to send a message and the number of processes willing to receive a message. Our model is hence a generalisation of this approach, for the way the global rate is computed is custom, and ultimately depends on the application at hand—e.g. the global rate can be fixed, or can depend on the number of system sub-processes willing to execute an action. Given a transition system of this kind and an initial state, a simulation is simply executed by: *(i)* checking each time the available actions and their rate; *(ii)* picking one of them probabilistically (the higher the rate, the more likely the action should occur); *(iii)* accordingly changing the system state; and finally *(iv)* advancing the time counter according to an exponential distribution, so that the average frequency is the sum of the action rates. This technique is again a generalisation of the one adopted in the SPIM simulation engine for stochastic π-Calculus [6]. For a detailed description of the simulation framework, refer to [14].

B. The Collective Sorting model

The MAUDE specification of the Collective Sorting system is divided in three modules, respectively defining

```
mod CS is
 pr CS .  pr STANDARD-CARRIER .

 op source : Nat -> Action .              *** SYNTAX OF ACTIONS AND STATES
 op chooseTarget : -> Action .
 op chooseTupleType : -> Action .
 op readSource : -> Action .
 op readTarget : -> Action .
 op move : -> Action .

 subsort DataSpace < State .
                                          ***  A REFERNCE INITIAL STATE
 op SS : -> State .
 eq SS = ( init | < 0 @ ('a[100])|('b[100])|('c[10])|('d[10]) > |
                  < 1 @ ('a[  0])|('b[100])|('c[10])|('d[10]) > |
                  < 2 @ ('a[ 10])|('b[ 50])|('c[50])|('d[10]) > |
                  < 3 @ ('a[ 50])|('b[ 10])|('c[10])|('d[50]) > |
                  ('a , 'b , 'c , 'd ) ) .

 *** IDENTIFYING SOURCE                   ***  TRANSITION SYSTEM SEMANTICS
 eq  (init | DS)==>  =
   ( source(0) # 0.25 -> [ [0] | DS ] );
   ( source(1) # 0.25 -> [ [1] | DS ] );
   ( source(2) # 0.25 -> [ [2] | DS ] );
   ( source(3) # 0.25 -> [ [3] | DS ] ) .

 *** CHOOSING TARGET
 eq ([Ns]      | DS)   ==>  = (chooseTarget # now -> [ [Ns];[range(3)]| DS ]) .
 eq ([Ns];[Ns] | DS)   ==>  = (chooseTarget # now -> [ [Ns];[3]       | DS ]) .

 *** CHOOSING TUPLE TYPE QQ
 ceq ([Ns];[Nt]      | < Ns @ MT > | DS ) ==> = ( chooseTupleType # now -> [
     ([Ns];[Nt];[QQ] | < Ns @ MT > | DS ) ] )
        if QQ := choose(occurringTuples(MT)) .

 *** READING FROM SOURCE
 ceq ([Ns];[Nt];[Q]      | < Ns @ MT > | QL | DS ) ==> = ( readSource # now -> [
     ([Ns];[Nt];[Q];[QQ] | < Ns @ MT > | QL | DS ) ] )
        if QQ := get( QL , sample(quantities(QL, MT))) .

 *** READING FROM TARGET
 ceq ([Ns];[Nt];[Q];[Q1]      | < Nt @ MT > | QL | DS ) ==> = ( readTarget # now -> [
     ([Ns];[Nt];[Q];[Q1];[QQ] | < Nt @ MT > | QL | DS ) ] )
        if QQ := get( QL , sample (quantities(QL, MT))) .

  *** MOVING OR DISCARDING
 ceq ( [Ns];[Nt];[Q];[Q1];[Q] |
       < Ns @ (Q[s N ]) | MT > |
       < Nt @ (Q[ N' ]) | MT1 > | DS ) ==> = ( move # now -> [
     ( init  |
       < Ns @ (Q[  N ]) | MT >  |
       < Nt @ (Q[s N']) | MT1 > | DS) ] )
         if Q1 =/= Q .

 eq  ( [Ns];[Nt];[Q];[Q1];[Q2] | DS ) ==> =  ( move # now -> [
     ( init                    | DS ) ] ) [owise] .

 eq temp( init | DS ) = false .            *** TEMPORANEOUS STATES
 eq temp( DS ) = true [owise] .
endm
```

Fig. 2.   The transition system semantics in module CS.

the structure of a system state (CS-TYPES), some utility functions (CS-FUNCTIONS), and finally the stochastic transition system operator ==> (CS). Module CS-TYPES and module CS-FUNCTIONS are not reported for brevity. Module CS-TYPES specifies the necessary types to define the structure of a system state. In particular, sort Tuple is used to model the occurrence of a tuple in a tuple space: for instance, 'a[10] means 10 tuples of tuple type 'a occur. Sort Space is used to represent a tuple space: <0 @ ('a[10])|('b[10])|('c[10])|('d[10])> means the tuple space with identifier 0 has 10 copies of each tuple type. Module CS-FUNCTIONS defines three functions: choose takes a list of tuple type identifiers and returns one non-deterministically chosen; occurringTuples takes the content of a tuple space and returns the list of tuple types occurring in it; quantities takes the content of a tuple space and a list of tuple types and returns the cardinality of each of them.

The CS module, as depicted in Figure 2, can be viewed as the core of the Collective Sorting model. First of all, six kinds of action are defined: the former is of the kind source(0),...,source(3) and is used to start an agent working on a certain tuple space; the others are constants corresponding to the five steps of the agent agenda. The constant SS is assigned to the initial state of the system we want to simulate, where tuples are spread in different quantities in the various tuple spaces.

The stochastic transition system semantics is divided in six groups according to the actions to be executed. Initially, four actions of the first kind are allowed, each with rate 0.25. The rate of other actions is the constant now, which is assigned to a large float, meaning that these actions should happen immediately. By this modelling choice, we will simulate a system where one agent evaluates for moving a tuple at each time unit, and such an evalution is immediate. The behaviour of transitions is briefly described as follows.

a) source(i): When task init occurs in the space it is time to spawn a new agent task: any of the tuple spaces can be chosen as source, with same probability. Task [i] correspondingly replaces init, where i is the source chosen. Note that DS is a variable over DataSpace, which here matches with the rest of the system.

b) chooseTarget: To choose a target, any tuple space in 0,1,2 is tried. If the result is equal to the current source, tuple space 3 is actually taken as target. This guarantees the source and target tuple spaces to be distinct. The task moves then to state [Ns];[Nt]—source and target identifier, respectively.

c) chooseTupleType: A tuple type is chosen randomly out of those currently occurring in Ns. This is computed with functions choose and occurringTuple, and is used to avoid picking a tuple which is currently absent in the source tuple space. The task moves then to [Ns];[Nt];[QQ]—where QQ is the tuple type chosen.

d) readSource: In this step a tuple type is drawn from the source tuple space using uniform read. Expression get(QL,sample(quantities(QL, MT))) is used to sample a tuple giving higher probability to those that occur more.

e) readTarget: Similar sampling is done on the target tuple space. The task moves now to [Ns];[Nt];[Q];[Q1];[Q2], where Q1 and Q2 are the tuple types read.

f) move: If the task matches [Ns];[Nt];[Q];[Q1];[Q] and Q1 is different from Q, then a tuple of kind Q is to be moved from Ns to Nt, which is realised by properly updating the tuple counters. Otherwise ([owise]), the tuple spaces state is left unchanged. In both cases, the task gets back to init.

Finally, the temp function defines as temporary states those that do not have task init, which will then cause the simulation counter not to update.

C. Simulating the Collective Sorting

The simulation can be run by giving the MAUDE interpreter a command like

```
rewrite < [ 5000 : ( SS ) @ 0.0 ] > .
```

which executes precisely 5000 agent executions starting from state SS. Such a state is defined as a constant in the code of Figure 2, and represents a possible initial (disordered) configuration of tuples. Figure 3 shows a piece of the output produced by the execution of the simulation—where each step includes simulation countdown counter, system state, and elapsed time. After some steps, some tuple starts moving from one space to the others. After 2024 time units, for instance, tuple kind 'c is already completely collected in tuple space 2. After 4600 time units, the system converged to complete sorting, as we expected from our distributed algorithm. Chart in Figure 4 reports the dynamics of the winning tuple in each tuple space, showing e.g. that complete sorting is reached at different times in each case. The chart in Figure 5 displays instead the evolution of the tuple space 0: notice that only the tuple kind 'a aggregates here despite its initial concentration was the same of tuple kind 'b.

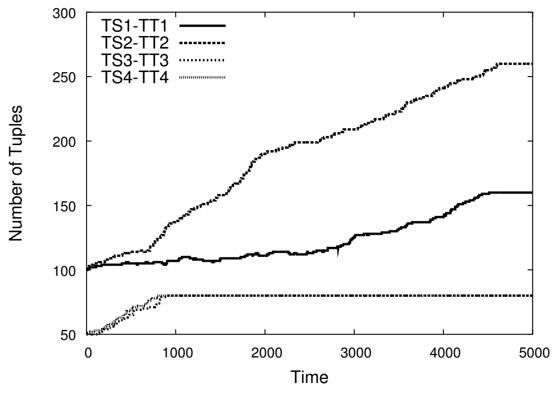

Fig. 4. Dynamics of the winning tuple in each tuple space: notice that each tuple aggregates in a different tuples space.

Although it would be possible to make some prediction, we do not know in general which tuple space will host a specific tuple kind at the end of sorting: this is an emergent property of the system and is the very result of the *interaction* of the tuple spaces through the agents! Indeed, the final result is not completely random and the concentration of tuples will evolve in the same direction *most* of the times. It is interesting to analyse the trend of the entropy of each tuple space as a way to estimate the degree of order in the system through a single value: since the strategy we simulate is trying to increase the inner order of the system we expect the entropy to decrease, as actually shown in Figure 6.

D. Adding a Load-Balancing Case

The basic strategy based on constant rates (see Section IV-B) is not very efficient, since agents are assigned to a certain tuple space also if the tuple space is already ordered! We may exploit this otherwise wasted computation by assigning idle agents to disordered tuple spaces, or rather to change the working rates of agents. This alternative therefore looks suited to realize a strategy to quicker reach the complete order of tuple spaces.

```
 <
   [5000 : init | < 0 @ ('a[100]) | ('b[100]) | ('c[10]) | ('d[10]) > |
                  < 1 @ ('a[0]) | ('b[100]) | ('c[10]) | ('d[10]) > |
                  < 2 @ ('a[10]) | ('b[50]) | ('c[50]) | ('d[10]) > |
                  < 3 @ ('a[50]) | ('b[10]) | ('c[10]) | ('d[50]) > | 'a,'b,'c,'d
         @ 0.0],
      ...
   [4000 : init | < 0 @ ('a[107]) | ('b[89]) | ('c[0]) | ('d[0]) > |
                  < 1 @ ('a[0]) | ('b[136]) | ('c[0]) | ('d[0]) > |
                  < 2 @ ('a[0]) | ('b[35]) | ('c[80]) | ('d[0]) > |
                  < 3 @ ('a[53]) | ('b[0]) | ('c[0]) | ('d[80]) > | 'a,'b,'c,'d
         @ 9.7664497212663287e+2],
   ...
   [2000 : init | < 0 @ ('a[127]) | ('b[50]) | ('c[0]) | ('d[0]) > |
                  < 1 @ ('a[0]) | ('b[210]) | ('c[0]) | ('d[0]) > |
                  < 2 @ ('a[0]) | ('b[0]) | ('c[80]) | ('d[0]) > |
                  < 3 @ ('a[33]) | ('b[0]) | ('c[0]) | ('d[80]) > | 'a,'b,'c,'d
         @ 3.0679938546387184e+3],
   ...
   [1000 : init | < 0 @ ('a[142]) | ('b[18]) | ('c[0]) | ('d[0]) > |
                  < 1 @ ('a[0]) | ('b[242]) | ('c[0]) | ('d[0]) > |
                  < 2 @ ('a[0]) | ('b[0]) | ('c[80]) | ('d[0]) > |
                  < 3 @ ('a[18]) | ('b[0]) | ('c[0]) | ('d[80]) > | 'a,'b,'c,'d
         @ 4.0271359303450395e+3],
   ...
   [438 : init  | < 0 @ ('a[160]) | ('b[0]) | ('c[0]) | ('d[0]) > |
                  < 1 @ ('a[0]) | ('b[260]) | ('c[0]) | ('d[0]) > |
                  < 2 @ ('a[0]) | ('b[0]) | ('c[80]) | ('d[0]) > |
                  < 3 @ ('a[0]) | ('b[0]) | ('c[0]) | ('d[80]) > | 'a,'b,'c,'d
         @ 4.6001450653146167e+3],
   ...
   [0 : init    | < 0 @ ('a[160]) | ('b[0]) | ('c[0]) | ('d[0]) > |
                  < 1 @ ('a[0]) | ('b[260]) | ('c[0]) | ('d[0]) > |
                  < 2 @ ('a[0]) | ('b[0]) | ('c[80]) | ('d[0]) > |
                  < 3 @ ('a[0]) | ('b[0]) | ('c[0]) | ('d[80]) > | 'a,'b,'c,'d
         @ 5.0313233386068514e+3]
>
```

Fig. 3. Result for the Collective Sorting simulation

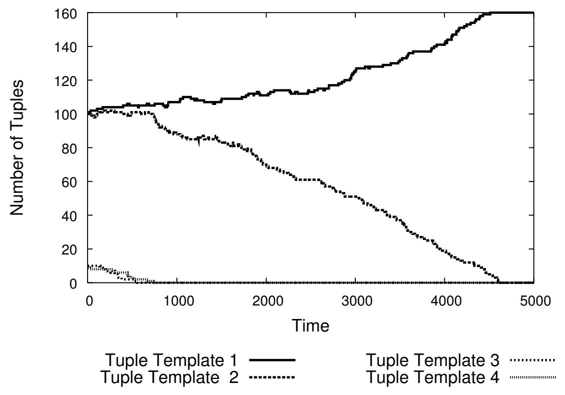

Fig. 5. Dynamic of tuple space `0`: notice that only one kind of tuple aggregates here.

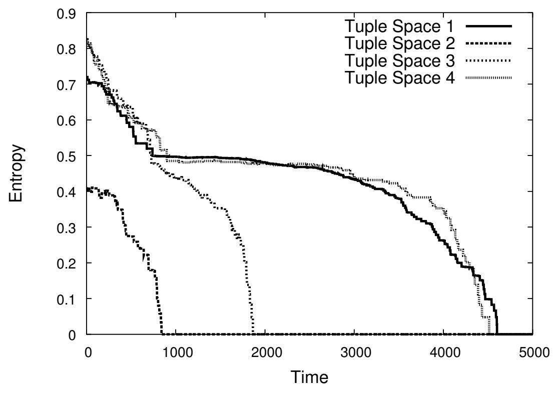

Fig. 6. Entropy of tuple spaces: they all eventually reach 0, that is, complete order.

In order to adapt the agents rate we need a measure of order: as already stated in Section III, spatial entropy may be an effective measure for system order. If we denote with q_{ij} the amount of tuples of the kind i within the tuple space j, n_j the total number of tuples within the tuple space j, and k the number of tuple kinds, then, the entropy associated with the tuple kind i within the tuple space j is

$$H_{ij} = \frac{q_{ij}}{n_j} \log_2 \frac{n_j}{q_{ij}} \tag{2}$$

and it is easy to notice that $0 \leq H_{ij} \leq \frac{1}{k} \log_2 k$. We want to express now the entropy associated with a single tuple space

$$H_j = \frac{\sum_{i=1}^{k} H_{ij}}{\log_2 k} \tag{3}$$

where the division by $\log_2 k$ is introduced in order to obtain $0 \leq H_j \leq 1$. If we have t tuple spaces then the entropy of the system is

$$H = \frac{1}{t}\sum_{j=1}^{t} H_j \tag{4}$$

where the division by t is used to normalize H, so that $0 \leq H \leq 1$. Being t the number of tuple spaces then it also represents the number of agents: let each agent work at rate $H_j r$, and tr be the maximum rate allocated to the sorting task. If we want to adapt the working rates of agents we have to scale their rate by the total system entropy, since

$$\gamma \sum_{j=1}^{t} rH_j = tr \Rightarrow \gamma = \frac{t}{\sum_{j=1}^{t} H_j} = \frac{1}{H} \tag{5}$$

hence each agent will work at rate $\frac{rH_j}{H}$ where H_j and H are computed periodically.

In order to modify the Collective Sorting model of Figure 2, we replaced the constant agents rate of the first four action (refer to Section IV-B) with the Equation 3 : hence, the activity rate of the tuple space j becomes H_j instead of `0.25`. Using *load balancing* we introduced *dynamism* in our model: indeed in each simulation step the activity rate associated with a tuple space—i.e. the probability at a given step that an agent of the tuple space is working—is no longer fixed, but it depends on the entropy of the tuple space itself. Hence, as explained above, agents belonging to completely ordered tuple spaces can consider their goal as being achieved, and hence they no longer execute tasks. Moreover, this strategy guarantees a better efficiency in the load balancing of agents work: agents working on tuple spaces with higher entropy, have a greater activity rate than the others on more ordered tuple spaces.

Using the Collective Sorting specification with variable rates, we ran the same simulation of the Section IV-C: the chart of Figure 7 shows the trend of the entropy of each tuple space. Comparing the chart with the one in Figure 6, we can observe that the entropies reach `0` faster than the case with constant rates: indeed since step `3000` every entropy within the chart in Figure 7 is `0`, while with constant rates the same result is reached only after `4600` steps. The chart in Figure 8 compares the tendency of the global entropy (see Equation 5) in the case of constant and variable rates: the trend of the two entropies represents a further proof that variable rates guarantee a faster stabilization of the system, i.e. its complete order.

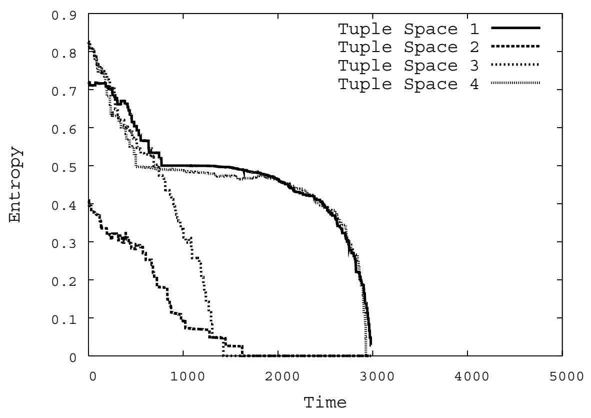

Fig. 7. Entropy of tuple spaces in the variable rate case: the system reaches the complete order since step `3000`.

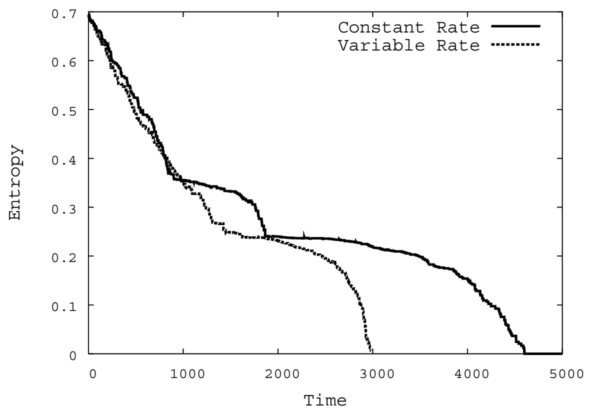

Fig. 8. Comparison of global entropy in the case of constant and variable rate: the latter reaches the complete order quicker.

V. Conclusion and Future Works

In this article we argued about the necessity of considering stochastic aspects when designing emergent coordination mechanisms: this issue is both emerging in few proposals of new coordination models and in related research contexts. We evaluated these ideas by using the MAUDE library we developed, considering and simulating a typical scenario of swarm-like coordination, the collective sorting problem, which we believe is a very paradigmatic application of emergent coordination because of its basic formulation.

Several interesting future works can be pursued:

- In the context of collective sorting, we plan to evaluate other load-balancing approaches, optimising the convergence to complete order, and working with different combinations of the number of tuple spaces and tuple kinds.
- The library itself is currently a very simple prototype, but we believe it could be improved in several ways and become a very practical simulation tool.
- Another interesting idea would be to apply our library to some existing coordination models like SwarmLinda, and provide the necessary tests for the proposed algorithms.

References

[1] E. Bonabeau, M. Dorigo, and G. Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems*, ser. Santa Fe Institute Studies in the Sciences of Complexity. Oxford University Press, Inc., 1999.

[2] L. Gardelli, M. Viroli, and A. Omicini, “On the role of simulations in engineering self-organising MAS: The case of an intrusion detection system in TuCSoN,” in *Engineering Self-Organising Systems*, ser. LNAI, S. A. Brueckner, G. Di Marzo Serugendo, D. Hales, and F. Zambonelli, Eds. Springer, 2006, vol. 3910, pp. 153–168, 3rd International Workshop (ESOA 2005), Utrecht, The Netherlands, 26 July 2005. Revised Selected Papers.

[3] M. Mamei and F. Zambonelli, “Programming pervasive and mobile computing applications with the tota middleware,” in *Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on*. IEEE, March 2004, pp. 263– 273.

[4] R. Menezes and R. Tolksdorf, “Adaptiveness in linda-based coordination models,” in *Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering*, ser. LNAI, G. D. M. Serugendo, A. Karageorgos, O. F. Rana, and F. Zambonelli, Eds. Springer Berlin / Heidelberg, January 2004, vol. 2977, pp. 212–232.

[5] R. D. Nicola, D. Latella, and M. Massink, “Formal modeling and quantitative analysis of KLAIM-based mobile systems,” in *SAC ’05: Proceedings of the 2005 ACM symposium on Applied computing*. New York, NY, USA: ACM Press, 2005, pp. 428–435.

[6] A. Phillips, “The Stochastic Pi Machine (SPiM),” 2006, version 0.042 available online at http://www.doc.ic.ac.uk/~anp/spim/. [Online]. Available: http://www.doc.ic.ac.uk/ anp/spim/

[7] “Swarm,” 2006, available online at http://www.swarm.org/.

[8] “Recursive porous agent simulation toolkit (repast),” 2006, available online at http://repast.sourceforge.net/.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *Maude Manual*, 2nd ed., Department of Computer Science University of Illinois at Urbana-Champaign, December 2005, version 2.2 is available online at http://maude.cs.uiuc.edu.

[10] C. Priami, “Stochastic pi-calculus,” *The Computer Journal*, vol. 38, no. 7, pp. 578–589, 1995.

[11] H. Gutowitz, “Complexity-seeking ants,” in *Proceedings of the Third European Conference on Artificial Life*, Deneubourg and Goss, Eds., 1993.

[12] K. Hadeli, P. Valckenaers, C. B. Zamfirescu, H. V. Brussel, B. S. Germain, T. Holvoet, and E. Steegmans, “Self-organising in multi-agent coordination and control using stigmergy,” in *Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering*, ser. LNAI, G. D. M. Serugendo, A. Karageorgos, O. F. Rana, and F. Zambonelli, Eds. Springer Berlin / Heidelberg, January 2004, vol. 2977, pp. 105–123.

[13] A. Omicini and E. Denti, “From tuple spaces to tuple centres,” *Science of Computer Programming*, vol. 41, no. 3, pp. 277–294, Nov. 2001.

[14] M. Casadei, L. Gardelli, and M. Viroli, “Simulating emergent properties of coordination in maude: the collective sorting case,” in *5th International Workshop on the Foundations of Coordination Languages and Software Architectures, Bonn, Germany*, August 2006, to appear into.