Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 175 (2007) 59–80
www.elsevier.com/locate/entcs

Simulating Emergent Properties of Coordination in Maude: the Collective Sort Case

Matteo Casadei[1] Luca Gardelli[2] Mirko Viroli[3]

DEIS, Alma Mater Studiorum—*Università di Bologna, via Venezia 52, 47023 Cesena, Italy*

Abstract

Recent coordination languages and models are moving towards the application of techniques coming from the research context of complex systems: adaptivity and self-organization are exploited in order to tackle the openness, dynamism and unpredictability of today's distributed systems. In this area, systems are to be described using stochastic models, and simulation is a valuable tool both for analysis and design. Accordingly, in this work we focused on modelling and simulating emergent properties of coordination techniques.
We first develop a framework acting as a general-purpose engine for simulating stochastic transition systems, built as a library for the Maude term rewriting system. We then evaluate this tool to a coordination problem called *collective sort*, where autonomous agents move tuples across different tuple spaces according to local criteria, and resulting in the emergence of the complete clustering property.

Keywords: Stochastic transition system, self-organization, simulation, coordination, collective sort.

1 Introduction

Several works studying timing, probability and stochasticity issues in foundational calculi for interaction—e.g. [18,10,11]—have recently received increasing attention. The long-term goal of these researches is to set up solid foundations for analysing and modelling quantitative aspects of software systems. Not only this is useful to address performance issues [11], as typically considered in last years, but it becomes very also crucial when designing dynamic and open applications.

Systems that self-organise to unpredictable changes in their environment very often need to feature adaptivity as an emergent property. As this observation was

[1] Email: m.casadei@unibo.it
[2] Email: luca.gardelli@unibo.it
[3] Email: mirko.viroli@unibo.it

1571-0661
doi:10.1016/j.entcs.2007.05.022

first made in the context of natural systems, it was shortly recognised as an inspiring metaphor for artificial systems as well [3]. A main aspect of emergent properties, however, is that by their very definition they cannot be achieved through a systematic design: their dynamics and outcomes cannot be fully predicted. Providing some design support in this context is still possible. The whole system of interest, that is the application and the environment, 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 [6].

This scenario is particularly interesting for coordination models and languages. Some works like the TOTA middleware [12], SwarmLinda [13], and Stochastic KLAIM [14], 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 e.g. the idea that tuples in a tuple space eventually spread to other tuple spaces in a non-deterministic way, depending on timing and probability. Accordingly, our goal is to analyse the potential role that simulation tools can have in this context, towards the identification of some methodological approach to system design.

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 [16], SWARM [2] and REPAST [1]. Instead of relying on one of them, we sought for a general-purpose approach. We evaluate the applicability of the MAUDE specification tool as a general-purpose engine for running simulations [4]. 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. Hence, 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 to a *rate* (of occurrence). 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.

This framework is tested on an application to a tuple space scenario called *collective sort*, which is a generalization of the problem known in the swarm intelligence community as *brood sort* [3]. 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, *(i)* tuples of the same type are collected in the same tuple space, and *(ii)* tuples of different kinds tend to reside in different tuple spaces. We evaluate a solution to this problem based on a fully-distributed algorithm, where each agent moves tuples according to local criteria, and where sorting appears to emerge from initial chaotic configurations.

The remainder of this paper is as follows: Section 2 provides some background on coordination techniques for adaptivity and formal frameworks for stochastic mod-

elling, Section 3 presents our library for simulation of stochastic systems in MAUDE, Section 4 describes the collective sort case and its simulation results, and finally Section 5 concludes providing perspectives on future works.

2 Background

2.1 Complex Systems and Coordination

In the effort to improve the design process of software systems—i.e. to bridge the gap between design and 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. The field of coordination models and languages is 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 [12] 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 it 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 architecture is the SwarmLinda coordination model [13], which though similar to TOTA is more inspired by swarm intelligence and stigmergy [3,8,9]. In SwarmLinda, ant-like algorithms are used to retrieve tuples in the distributed system. The use of self- techniques in SwarmLinda derives from necessity of achieving adaptivity when dealing with openness and with the unpredictability of user interactions.

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 [3], 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—e.g. collecting items on the ground.

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.

2.2 Formal Tools for Specifying Stochastic Aspects

In particular, following the work on foundational calculi for interaction, we identify the stochastic dimension as a crucial one in system modelling [18]. A stochastic system is a system where the evolution in time is aleatory. On the one hand, this can be used to abstract away from implementation specific issues, by just stating that a process will take *some* time to execute. On the other hand, advancement in time is tracked, and its variability is accounted for, by considering the execution time as an aleatory variable distributed according to a specific distribution of probability.

A first example of work studying the issue of stochastic modelling is the study of stochastic π-Calculus by Priami in [18]. In that model, each communication channel is associated to a rate: the duration of an interaction through a channel is an aleatory variable distributed according to an exponential distribution defined by that rate. Accordingly, the semantics of non-deterministic choice changes, for the probability of an action is a function of the rates of the involved channels. For this language, the SPiM tool has been introduced to run simulations using the Gillespie algorithm [16,17,7]—the basic algorithm for running simulations of chemical reactions. This tool has been developed mainly to explore the dynamics of biochemical systems [17], though it can be applied to software systems as well [6].

In the context of coordination, KLAIM (Kernel Language for Agents Interaction and Mobility) [14] is a language introduced for modelling and programming distributed systems made of components asynchronously interacting via tuple spaces—thus extending Linda. KLAIM is similar in philosophy to the π-Calculus, the main exception is that processes communicate in an asynchronous manner, via the insertion and removal of tuples in tuple spaces. Particularly interesting is the stochastic extension of KLAIM, called STOcKLAIM, which basically follows the same approach of the Priami's extension of π-Calculus. The semantics of STOcKLAIM is given by a labelled transition system which is translated into a continuous-time Markov chain: this translation is performed to allow for quantitative analysis and model-checking [14]. A probabilistic extension to KLAIM exists as well, called pKLAIM [5], which replaces non-determinism with explicit probabilities, and where time is discrete.

As many other examples of stochastic process algebras exist, we are here interested in finding a general framework, one at the meta-level which does not promote a specific language but allows for a great deal of flexibility in the specification of syntactic and semantic aspects. The MAUDE meta-programming language appears quite promising to this end. MAUDE is a high-performance reflective language supporting both equational and rewriting logic specifications, for specifying a wide range of applications [4]. 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 considered MAUDE 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. This is therefore a natural starting point for addressing the simulation of stochastic aspects in coordination: other languages require the designer to model systems in terms of *off-the-shelf* abstractions—e.g. channels and processes in π-Calculus—which might not be suitable in the general case. Furthermore, MAUDE provides tools for performing the analysis of systems properties, including theorem proving and model checking—which opens interesting future works in this research.

3 A Stochastic Simulation Framework in Maude

In this section we describe a basic and general simulation framework for stochastic systems implemented as a MAUDE library. For the sake of brevity, we shall neglect a full description of MAUDE—the interested reader can refer to the official MAUDE documentation [4]—though some of its main aspects are presented throughout.

The idea of our library 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 [18], 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 generalization 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 occurs); *(iii)* accordingly changing the system state; and finally *(iv)* advancing the time counter following an exponential distribution, so that the average frequency is the sum of the action rates. This technique is again a generalization of the one adopted in SPiM [16].

The framework implementation is organized in five Maude modules: *(i)* `STOCHASTIC-SELECTION` contains the definition of the functions handling probabilities and randomness; *(ii)* `STANDARD-CARRIER` provides all the definitions a specific system has to implement in order to be simulated by this tool; *(iii)* `STOCHASTIC-TRACES-TYPES` contains the definition of the data structures of the *stochastic engine*; *(iv)* `STOCHASTIC-TRACES-FUNCTIONS` provides the definition

```
mod STOCHASTIC-SELECTION is
  pr COUNTER .
  pr RANDOM .
  pr CONVERSION .
  pr LIST{Float} .

  sort Event .
  op @     : [Nat] [Float] -> Event [ctor] .
  op next  : List{Float}   -> Event .

  *** Inner definitions and implementation
  ...
  op $rand : -> [Float] .
  eq $rand = float(random(counter)/ 4294967295) .
  ...
endm
```

Fig. 1. Definitions in the `STOCHASTIC-SELECTION` module.

of some essential functions for implementing the stochastic engine; and *(v)* `STOCHASTIC-TRACES-ENGINE` contains the actual definition of the stochastic engine. Each module is briefly described in turn.

3.1 STOCHASTIC-SELECTION module

As shown in Figure 1, the `STOCHASTIC-SELECTION` module starts with clauses to import definitions from other system modules, namely `COUNTER` for using an incremental counter, `RANDOM` for generating random numbers, `CONVERSION` to convert integers to floats, and finally `List{Float}` for handling lists of floats.

A sort (i.e. a "type") `Event` is defined, along with a "`@`" constructor operator for generating its values (`[ctor]`). The idea is that a term `@(N,F)` represents a simulation event, caused by the action expressed by natural number `N`, and where float number `F` represents the corresponding elapsed time. The `next` function is the most important function of the module, as it generates an event using a stochastic selection policy, starting from a list of rates. For instance, a term `next(2.0 3.0 5.0)` is evaluated when the system to simulate can perform one of three types of action, characterized by the rate 2, 3, and 5, orderly. It evaluates to an event `@(N,F)`, where `N` can be `0` with probability `20%`, `1` with probability `30%`, and `2` with probability 50%. `F` is computed from an exponential distribution, and is average value is 0.1—for the sum of rates is 10. A possible result obtained by the MAUDE command "`rewrite next(2.0 3.0 5.0).`" is e.g. the event `@(1, 7.330813624033139e-2)`. The selection of an action and of the elapsed time is of course random, and exploits the function `$rand` which yields a number in between 0 and 1—which itself uses the built-in function `random` as shown in the equation (`eq`) in the picture. Full details of the implementation of function `next` are not reported for brevity.

3.2 The STANDARD-CARRIER module

When a user provides a stochastic system specification, that specification must implement a number of definitions representing the different concepts exploited during simulation. The module `STANDARD-CARRIER` shown in Figure 2 provides that definitions and the necessary constraints on them—it roughly plays the role of an abstract class in OO languages, to be implemented with details of the system at

```
mod STANDARD-CARRIER is
  pr FLOAT .
  pr BOOL .

  sort State Action States .

  subsort State < States .
  op __  : States States -> States [ctor assoc comm] .

  sort Effect Effects .
  op _#_->[_] : Action Float States -> Effect [ctor] .

  subsort Effect < Effects .
  op nil :                -> Effects [ctor] .
  op _;_ : Effects Effects -> Effects [ctor assoc id: nil] .

  *** TO BE IMPLEMENTED
  sort Observation .
  op obs : Nat State Float -> Observation .

  op _==> : State -> Effects .
  op temp : State -> Bool .
  op quit : Nat State Float -> Bool .
endm
```

Fig. 2. Definitions in the `STANDARD-CARRIER` module.

hand.

First of all, sorts for a system state (`State`), an action (`Action`), and a multiset of states (`States`) must be provided. A constructor operator `__` is introduced to let the juxtaposition of two states be of sort `States`. That operator is then declared to be commutative (`comm`) and associative (`assoc`): this is used to state that a `States` represents a (non-void) multiset of elements of sort `State`.

Then types `Effect` and `Effects` are defined. The operator `_#_->[_]` is used to construct an `Effect`. A term of the kind `A#F->[Ss]` means that in a certain system state, action `A` can be applied with rate `F`, which moves the system to any state in the multiset of states `Ss`. An operator `_;_` is then specified to state that sort `Effects` represents a list (`[assoc]`) of elements of sort `Effect`, separated by semi-colons, and with constant `nil` representing the empty list.

Sort `Observation` provides the user with the concept of *observability*: operator `obs` takes a system state and yields a partial view, namely, an element of sort `Observation`. The output of a simulation will be a trace of observations: function `obs` is then to be carefully designed whenever a user does not mean to trace the overall system dynamics but is just interested in few parameters—as is typically the case.

Most importantly, the user must provide an implementation of operator `==>`, which takes a system state and yields a list of effects, i.e. describes the transition system $S \xrightarrow{r:a} S'$.

Finally, the user must implement the predicates `temp` and `quit`. The `temp` predicate is defined over states so as to mark a given state as temporary, thus preventing the engine from adding it to the simulation trace. The `quit` predicate is instead used to check if/when a simulation has to be stopped, for the system seemingly reached a final state. These two predicates come with default implementations, both yielding `false`.

Concretely, as we will show in the example of Section 3.6, for a user to run a

```
mod STOCHASTIC-TRACES-TYPES{ X :: CARRIER } is
  pr STOCHASTIC-SELECTION .

  sort Step Observations Trace Steps Evt Evts .

  subsort Step < Steps .
  op [_:_@_] : Nat X$State Float -> Step [ctor format (ni d d d d d d d)] .
  op nil : -> Steps .
  op _+_ : Steps Steps -> Steps [ctor assoc id: nil ] .

  subsort X$Observation < Observations .
  op _,_ : Observations Observations -> Observations [ctor assoc id: empty] .
  op empty : -> Observations [ctor] .

  op _<_> : Step Observations -> Trace [ctor format (d d ni ni d)].
  op <_>_ : Observations Step -> Trace [ctor format (d ni ni d d)].

endm
```

Fig. 3. Definitions in the STOCHASTIC-TRACES-TYPES module.

system simulation he/she must define sorts `Action`, `State`, and `Observation`, along with implementations for operators `==>` and `obs`.

3.3 The STOCHASTIC-TRACES-TYPES module

As Figure 3 reports, the `STOCHASTIC-TRACES-TYPES` module contains the definition of the types that are necessary to implement the *stochastic engine*. `STOCHASTIC-TRACES-TYPES` is parametric in a module `X` that implements `STANDARD-CARRIER`, and that represents the actual system to simulate; accordingly, e.g. sorts `X$State` and `X$Observation` are used to denote the sorts of the system's states and observations.

Types and constructors for the concepts of *(i)* `Step`, *(ii)* `Steps`, and *(iii)* `Observations` are first introduced. A `Step` represents a simulation step, whose structure is `[N:S@F]`, where `N` is a countdown counter of the simulation, `S` is the current system state, and the float `F` is the elapsed time since the beginning. A `Steps` element is a list of steps separated by commas, while an `Observations` element is a list of observations separated by commas.

Then, sort `Trace` is defined. A `Trace` represents the outcome of a simulation `STn<OB1,OB2,...,OBn>`, where: `STn` is a `Step` that represents the current state of the simulated system, and `OB1,OB2,...,OBn` is a list of observations providing a view on the system evolution.

3.4 The STOCHASTIC-TRACES-FUNCTIONS module

The `STOCHASTIC-TRACES-FUNCTIONS` module contains the definition of the necessary functions for the stochastic engine. Figure 4 shows the definition of the most important functions in `STOCHASTIC-TRACES-FUNCTIONS`. `STOCHASTIC-TRACES-FUNCTIONS` is parametric in a module `X` that implements `STANDARD-CARRIER`, and that represents the system to simulate: hence, both type `X$State` and type `X$Effects` are specific for that system.

First of all, the `activities` function is defined. Given a list of `Effect` as input, the `activities` function yields a list of `Float` numbers that are the rates of the

```
mod STOCHASTIC-TRACES-FUNCTIONS{ X :: CARRIER } is
 pr STOCHASTIC-SELECTION .
 pr STOCHASTIC-TRACES-TYPES{X} .

 op activities : X$Effects -> List{Float} .
 eq activities( nil ) = nil .
 eq activities( ( A # F -> [ LS ] ) ; Es ) = F activities(Es) .

 op newState : Nat X$Effects -> X$State .
 eq newState( 0 , ( A # F -> [ LS ] ) ) = one( LS ) .
 eq newState( 0 , E ; Es ) = newState( 0, E ) .
 eq newState( s N , (E ; Es) ) = newState( N, Es) [owise].
endm
```

Fig. 4. Definitions in the STOCHASTIC-TRACES-FUNCTIONS module.

actions of each Effect.

Then the newState function is defined. Given a list of Effect and a Nat number N as input, this function yields a State representing the new system's state for the next step of a simulation. The new system's state is the State caused by the N^{th} Action, namely, the action belonging to the Effect in the N^{th} position of the input list.

The following Section 3.5 illustrates how the described functions are used in order to define the stochastic engine.

3.5 The STOCHASTIC-TRACES-ENGINE module

As illustrated in Figure 5, the STOCHASTIC-TRACES-ENGINE module provides the definition and the implementation of the *stochastic engine*. Likewise STOCHASTIC-TRACES-TYPES and STOCHASTIC-TRACES-FUNCTIONS, STOCHASTIC-TRACES-ENGINE is also parametric in a module X that has to implement STANDARD-CARRIER, and that represents the actual system to simulate; accordingly, types X$State and X$Action are e.g. used to denote the types of this system's states and actions.

The module starts defining a number of variables, e.g. F with type Float, and FF and FF' with type [Float]: this is called a *kind* in MAUDE, and represents a float expression that is possibly not fully evaluated yet.

Function move in the module implements the single-step behaviour of the simulation engine. It takes a Step and produces the next one, randomly, by properly using the functions defined both in STOCHASTIC-SELECTION and in STOCHASTIC-TRACES-FUNCTIONS. In particular, as the event @(NN,FF) is computed from the currently available rates, the simulation counter decreases (from (s N) to N in Peano notation), the elapsed time increases of FF, and finally the new state SS is obtained by applying the NN^{th} action (by means of the newState function). Note that the move function works if the simulation counter did not reached zero.

The trace function is exploited by users who want to obtain a complete trace of observations as outcome of their simulated systems.

The function is defined by mean of *three* equations: the *first* applies when the current state is temporary, in which case the new step is computed by function move without updating the countdown and without adding a new observation to

```
mod STOCHASTIC-TRACES{ X :: CARRIER } is
 protecting STOCHASTIC-SELECTION .

  *** INTERNALS
  var O  : X$Observation . var OO : [X$Observation] .
  var S S' S1 S2 : X$State . var P : Step .
  var SS SS1 : [X$State] .  var Es : [X$Effects] .
  vars N N1 N' : Nat . vars NN : [Nat] .
  vars F F1 F2 : Float . vars FF FF' FF1 : [Float] .
  vars L : Observations .

  op move : Step -> Step .

  ceq move( [ (s N) : S @ F] ) = [ N : SS @ FF ]
      if   Es := evalEffects(S ==>) /\
           @( NN , FF' ) := next(activities(Es)) /\
           NN =/= -1 /\
           FF := F + FF' /\
           SS := newState( NN , Es ) .

  eq move( [ (s N) : S @ F] ) = [ (s N) : S @ F ]  [owise] .

  op trace : Trace -> Trace .

  ceq trace( [N : S @ F]< L > ) = trace( [N : SS @ FF]< L > )
      if temp(S)
         /\ [ (N) : SS @ FF ] := move([ (s N) : S @ F ]) .

  ceq trace( [s N : S @ F]< L > ) = trace( [N : SS @ FF] < L , O > )
     if not temp(S)
       /\ not quit(N, S, F)
       /\ O := obs(s N, S, F)
       /\ [ (N) : SS @ FF] := move([ (s N) : S @ F ]) .

  ceq trace([s N : S @ F]< L >) = trace( [O : S @ F] < L , O > )
     if not temp(S)
       /\ quit(N, S, F)
       /\ O := obs(s N, S, F) .

  ceq trace([O : S @ F] < L >) = < L , O > [O : S @ F]
     if not temp(S)
       /\ O := obs(O,S,F) .

  op last : Trace -> Evt .

  ceq last( [N : S @ F]< L > ) = last( [N : SS @ FF]< L > )
    if temp(S)
      /\ [ (N) : SS @ FF ] := move([ (s N) : S @ F ]) .

  ceq last( [s N : S @ F]< L > ) = last( [N : SS @ FF] < O > )
     if not temp(S)
    /\ not quit(N, S, F)
    /\ O := obs(s N, S, F)
    /\ [ (N) : SS @ FF] := move([ (s N) : S @ F ]) .

  ceq last([s N : S @ F]< L >) = evt( N , O , F )
     if not temp(S)
       /\ quit(N, S, F)
       /\ O := obs(s N, S, F) .

  ceq last([O : S @ F] < L >) = evt(O , O , F )
     if not temp(S)
       /\ O := obs(O,S,F) .

endm
```

Fig. 5. Definitions in the STOCHASTIC-TRACES-ENGINE module.

the current trace of the simulation; the *second* defines the behaviour of the engine when the current state is not temporary, adding a new observation to the trace of the simulation and computing the new simulation step; finally the *third* applies when the current state is a final state, in which case a new observation is added to the current trace and the simulation is terminated.

To produce for instance 100 simulation steps starting from state S0, the

MAUDE command "`rewrite trace( [100 : S0 @ 0.0] < empty > ).`" is to be run, producing an output of the kind: `< Obs1 Obs2 ... Obs100 >`, where `Obs1,Obs2...Obs100` are observations showing the evolution of the simulated system.

The `last` function is employed by users who are interested only to see the final observation of their simulated systems. Likewise `trace`, the `last` function is defined by means of *three* equations: the *first* applies when the current state is temporary, in which case the new step is computed by function `move` without updating the countdown and without adding a new observation to the current trace of the simulation; the *second* defines the behaviour of the engine when the current state is not temporary, computing the new simulation step and producing no observations; the *third* applies when the current state is a final state, in which case the simulation is terminated and an observation—representing the user-defined view on the final state—is produced as outcome of the simulation. Hence, unlike `trace`, the `last` function provides users with an outcome containing only the observation related to the final state of a simulated system. To produce a 100-steps simulation using `last`, the MAUDE command is "`reduce last( [100 : S0 @ 0.0] < empty > )`" . The resulting output is of the kind: `< Obs100 >`, representing the observation on the state associated with the 100^{th} (last) simulation step.

3.6 An example: the $Na - Cl$ specification

We consider now the standard example of the $Na - Cl$ chemical reaction dynamics, provided e.g. in SPiM documentation [4], in order to briefly explain the process of creating a system specification to simulate.

The module to realize this specification is reported in Figure 6. This system is characterized by a state of the kind `<Na,Na+,Cl,Cl->`, where `Na` is the of sodium atoms, `Na+` the number of sodium ions, `Cl` is the number of chlorine atoms, `Cl-` the number of chlorine ions. Two kinds of constant actions are then defined: `ionize` stands for ionization and `deionize` for deionization.

Then, the transition system is expressed by a single equation, associating to any state two possible effects: one in which ionization decrements `Na` and `Cl` (by prefix predecessor function `p`) and increments `Na+` and `Cl-` (by prefix successor function `s`), and the other that behaves in the opposite way. Note that, according e.g. to the Gillespie selection algorithm in [7], the rate of ionization and deionization is here proportional to the product of the two reactants, multiplied by a constant value: we here e.g. enforce deionization factor as being twice that of ionization.

Finally, an `Observation` is expressed by the user-defined operator `<_,_>@_`. The following equation on the `obs` predicate defines the actual meaning of the observation `<_,_>@_`. In particular, the definition expresses the interest to observe: *(i)* the number of `Na` atoms, *(ii)* the number of `Cl` atoms, *(iii)* the number of simulation steps.

The MAUDE command: "`rew trace([300:<100,0,100,0>@0.0]< empty >)`"

[4] `http://www.doc.ic.ac.uk/~anp/spim/Chemical.pdf`

```
mod NA-CL is
 pr FLOAT .
 pr INT .
 pr CONVERSION .
 pr STANDARD-CARRIER .

 sort NaClState .
 subsort NaClState < State .

 op <_,_,_,_> : Nat Nat Nat Nat -> State .

 ops ionization deionization : -> Action .

 vars Na Na+ Cl Cl- : Nat .

 eq  < Na,Na+,Cl,Cl- > ==>  =
   ( ionization   # (float(Na * Cl) * 1.0)   ->  [< p Na,s Na+,p Cl,s Cl- >]  );
   ( deionization # (float(Na+ * Cl-) * 2.0) ->  [< s Na,p Na+,s Cl,p Cl- >]  ) .

 op <_,_>@_ : Nat Nat Nat -> Observation .
 eq obs( Count:Nat, < Na,Na+,Cl,Cl- >, F:Float ) = < Na,Cl >@ Count:Nat .

endm

<
(< 100,100 >@ 300),
(< 99,99 >@ 299),
(< 98,98 >@ 298),
(< 97,97 >@ 297),
...
(< 61,61 >@ 7),
(< 60,60 >@ 6),
(< 59,59 >@ 5),
(< 58,58 >@ 4),
(< 57,57 >@ 3),
(< 58,58 >@ 2),
(< 59,59 >@ 1),
(< 60,60 >@ 0)
>
```

Fig. 6. Definition of the $Na - Cl$ system based on our stochastic library.

produces the trace reported in Figure 6, showing that the system reaches a stable state around `<60,60>`.

4 Collective Sort

To evaluate the applicability of our library as a simulation engine for coordination mechanisms, we consider a generalized case of the Swarm intelligence *brood sorting* problem [3], properly moved to a tuple spaces context.

4.1 General Scenario and Applications

We considered 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 a shared criterion. This problem basically amounts to clustering: homogeneous items should be grouped together and should be separated from different ones. 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 that 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 [3]: this class of coordination strategies are generally referred to as *collective sort* or *collective clustering*. 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

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

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 [8]. 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.

4.2 An Architecture for Implementing Collective Sort

We conceived 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 realized 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 7, 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 Equation 1 to decide wether to move or not a tuple: the approach would not be

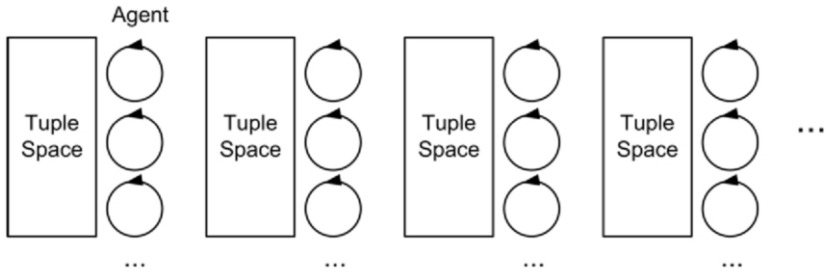

Fig. 7. The basic architecture for implementing Collective Sort.

scalable since it requires to count all the tuples for each tuple space, which might not be practical.

We devised 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 [15].

When deciding to move a tuple, an agent working on the tuple space TS_S follows this agenda:

(i) it draws a destination tuple space TS_D different from the source one TS_S;

(ii) it draws a kind k of tuple;

(iii) it (uniformly) reads a tuple T_1 from TS_S;

(iv) it (uniformly) reads a tuple T_2 from TS_D;

(v) 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? 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.

```
mod CS-TYPES is
  pr QID .

  sort Tuple TupleMSet Space QList Task DataSpace .

  *** TUPLES
  op _[_] : Qid Nat -> Tuple [ctor] .

  subsort Tuple < TupleMSet .
  op empty : -> TupleMSet [ctor] .
  op _|_ : TupleMSet TupleMSet -> TupleMSet [ctor assoc comm id: empty] .

  *** TUPLE SPACE
  op <_@_> : Nat TupleMSet -> Space [ctor] .

  *** AGENT-TASK
  op init : -> Task [ctor] .
  op [_] : Nat -> Task [ctor] .
  op [_] : Qid -> Task [ctor] .
  op _;_ : Task Task -> Task [ctor assoc] .

  *** DATASPACE
  subsort Task Space QList < DataSpace .
  op empty : -> DataSpace [ctor] .
  op _|_ : DataSpace DataSpace -> DataSpace [ctor assoc comm] .

  ...
endm
```

Fig. 8. Module CS-TYPES.

4.3 Modelling and Simulating Collective Sort in Maude

In this section we briefly describe a Maude specification of our solution to the collective sort problem, and show simulation results. Our model sticks to the case where 4 tuple spaces exist, labelled with natural identifiers 0, 1, 2 and 3. Tuples are expressed as Maude quoted identifiers and can be any, though the simulations we consider here feature the four tuple types `'a`, `'b`, `'c`, and `'d`. Moreover, we suppose tuple spaces are accessed by agents at the same rate—more fine grained load-balancing issues could be taken into account, which is not considered in this paper for simplicity.

4.3.1 The Collective Sort model in Maude

The Maude specification of the Collective Sort system described in 4.2 is divided in three modules, respectively defining the structure of a system's state (`CS-TYPES`), some utility functions (`CS-FUNCTIONS`), and finally the stochastic transition system operator `==>` (`CS`).

Figure 8 shows the definitions in the first module. 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` occurs. 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. A `Task` is a sequence of terms holding the state of the agent currently in charge of evaluating a tuple move. The sequence grows incrementally as an agent takes decisions: at the end of the protocol it is of the kind `[N1];[N2];[Q];[Q1];[Q2]`, where `N1` is the source tuple space identifier, `N2` the target tuple space identifier, `Q` the type of tuple to be possibly moved, `Q1` the tuple read from the source, and `Q2` the tuple read from the target. A `QList` is a list of quoted identifiers, representing the tuple types to be sorted. Fi-

nally, a `DataSpace` is a (multiset-like) composition of `Space`'s, a `Task` representing current agent's work, and a `QList`.

Module `CS-FUNCTIONS` is not reported for brevity. It basically 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 9, can be viewed as the core of the Collective Sort 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. Note first that 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:

`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.

`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.

`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.

`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.

`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.

`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 realized 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

```
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. 9. The transition system semantics in module CS.

task `init`, which will then cause the simulation counter not to update.

From the previous description, it might become clear why we have chosen MAUDE among other languages for stochastic simulations: MAUDE really allows to define syntax and semantics in quite custom and flexible way. Instead, for example, when working with π-Calculus one is forced to model the system in terms of processes and channels: while these abstractions might be useful in certain domains, it may not be suitable to map tuple either to a process or a channel.

```
<
  [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],
  [4999 : 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
        @ 5.2282294679077934e-1],
  ...
  [4989 : init | < 0 @ ('a[100]) | ('b[100]) | ('c[10]) | ('d[10]) > |
                 < 1 @ ('a[0]) | ('b[101]) | ('c[10]) | ('d[10]) > |
                 < 2 @ ('a[10]) | ('b[50]) | ('c[50]) | ('d[10]) > |
                 < 3 @ ('a[50]) | ('b[9]) | ('c[10]) | ('d[50]) >  | 'a,'b,'c,'d
        @ 8.6379503776170434],
  ...
  [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],
  ...
  [3000 : init | < 0 @ ('a[112]) | ('b[69]) | ('c[0]) | ('d[0]) > |
                 < 1 @ ('a[0]) | ('b[191]) | ('c[0]) | ('d[0]) > |
                 < 2 @ ('a[0]) | ('b[0]) | ('c[80]) | ('d[0]) > |
                 < 3 @ ('a[48]) | ('b[0]) | ('c[0]) | ('d[80]) > | 'a,'b,'c,'d
        @ 2.0243203450809999e+3],
  ...
  [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. 10. Simulation result for the Collective Sort simulation

4.3.2 Simulating the Collective Sort in Maude

As described in previous sections, 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`. Figure 10 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 a complete sorting, as we expected from our distributed algorithm. Chart in Figure 11 reports the dynamics of

the winning tuple in each tuple space, showing e.g. that complete sorting is reached at different moments in time in each case. The chart in Figure 12 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`. Although it is 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—computed as described in Section 4.2—as a way to estimate the degree of order in the system through a single value: since the strategy we simulated is trying to increase the inner order of the system we expected the entropy to decrease, as actually shown in Figure 13.

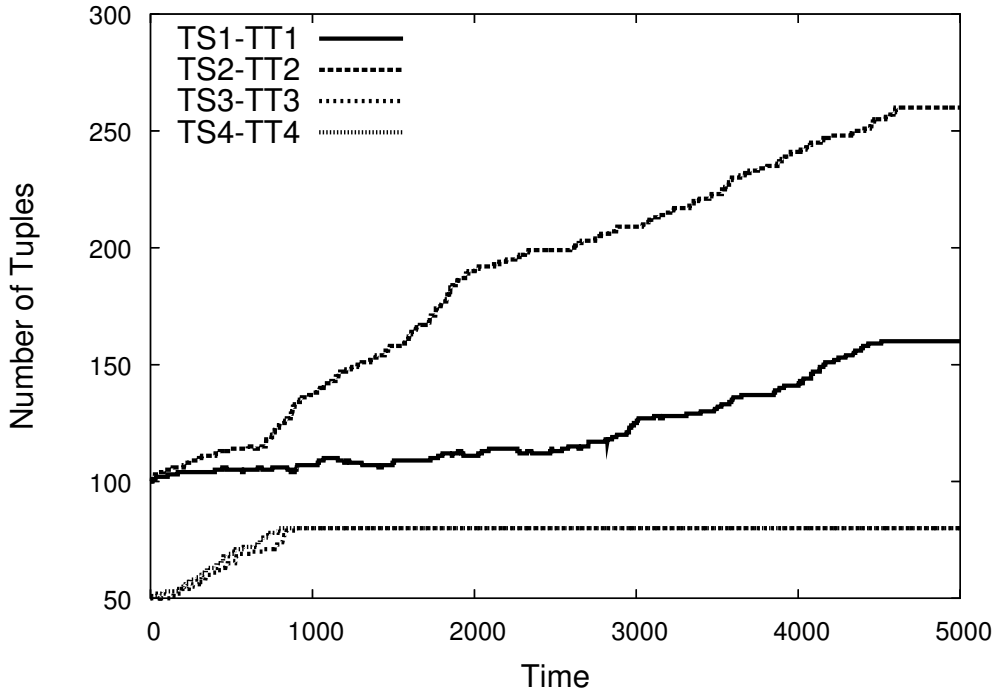

Fig. 11. Dynamics of the winning tuple in each tuple space: notice that each tuple aggregates in a different tuples space.

5 Conclusion

In this article we argued about the necessity of considering stochastic aspects when designing self-organization-like coordination mechanisms: this issue is both emerging in few proposals of new coordination models and in related research contexts. The MAUDE library we developed allows for easily prototyping simulations of coor-

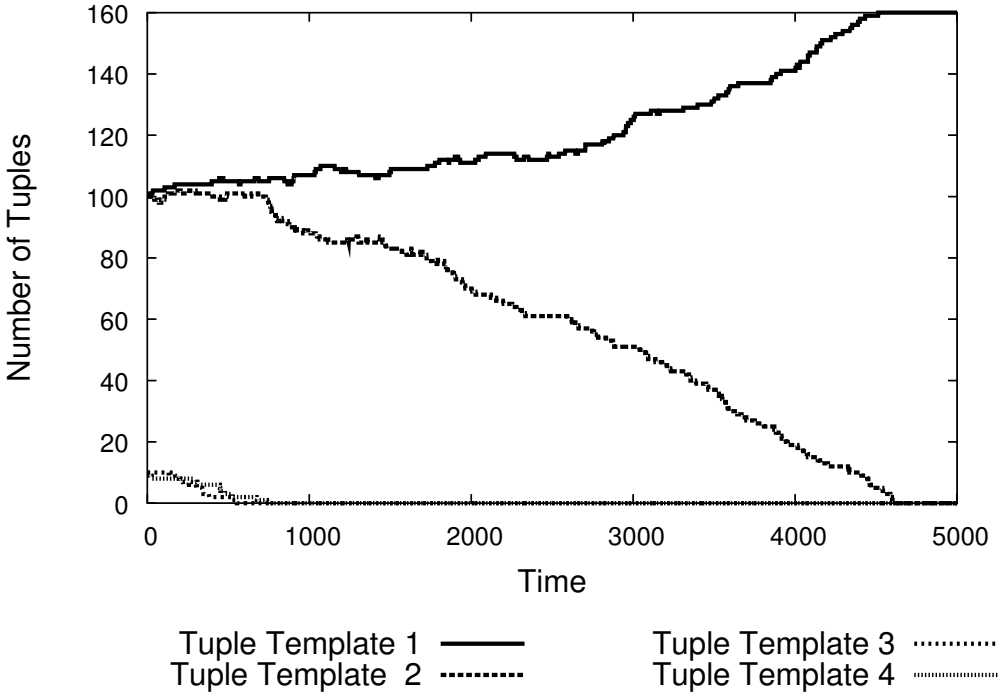

Fig. 12. Dynamic of tuple space `0`: notice that only one kind of tuple aggregates here.

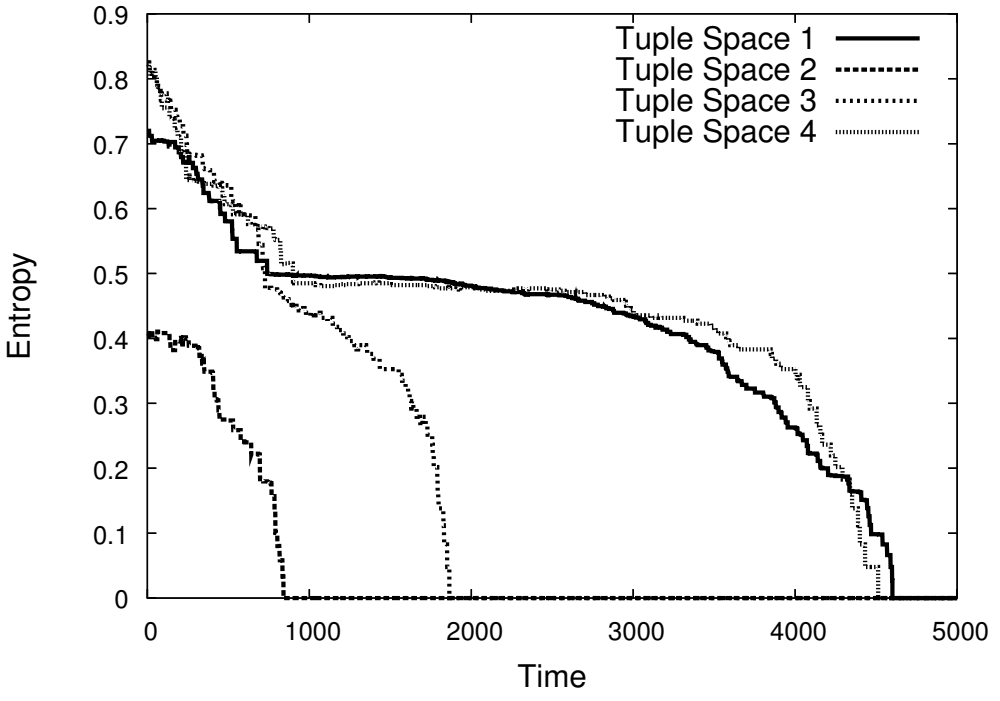

Fig. 13. Entropy of tuple spaces: they all eventually reach 0, that is, complete order.

dination techniques, and studying their emergent properties. We tested the module by specifying a typical scenario of swarm-like coordination, the collective sort 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 sort, we plan to evaluate techniques applying load-balancing approaches, optimising the convergence to complete order, and working with different combinations of the number of tuple spaces and tuple kinds. Indeed, though in our cases the system appears to stabilize to the desired state, when working with self-organising systems it is possible that the system evolves to a stable state which is not the final one. In other words, we have to provide more guarantees about the behaviour of the strategy.
- 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.
- Finally, it would be interesting to analyse the existing results on probabilistic model-checking, and see whether global emergent properties can be automatically inferred from a system specification.

References

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

[2] *Swarm* (2006), available online at http://www.swarm.org/.

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

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

[5] Di Pierro, A., C. Hankin and H. Wiklicky, *Continuous-time probabilistic* KLAIM, Electronic Notes in Theoretical Computer Science **128** (2005), pp. 27–38.

[6] Gardelli, L., 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: S. A. Brueckner, G. Di Marzo Serugendo, D. Hales and F. Zambonelli, editors, *Engineering Self-Organising Systems*, LNAI **3910**, Springer, 2006 pp. 153–168, 3rd International Workshop (ESOA 2005), Utrecht, The Netherlands, 26 July 2005. Revised Selected Papers.

[7] Gillespie, D. T., *Exact stochastic simulation of coupled chemical reactions*, The Journal of Physical Chemistry **81** (1977), pp. 2340–2361.

[8] Gutowitz, H., *Complexity-seeking ants*, in: Deneubourg and Goss, editors, *Proceedings of the Third European Conference on Artificial Life*, 1993.

[9] Hadeli, K., 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: G. D. M. Serugendo, A. Karageorgos, O. F. Rana and F. Zambonelli, editors, *Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering*, LNAI **2977**, Springer Berlin / Heidelberg, 2004 pp. 105–123.

[10] Hennessy, M. and T. Regan, *A Process Algebra for Timed Systems*, Information and Computation **117** (1995), pp. 221–239.

[11] Hermanns, H., U. Herzog and J. Katoen, *Process algebra for performance evaluation*, Theoretical Computer Science **274** (2002), pp. 43–87.

[12] Mamei, M. 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* (2004), pp. 263– 273.

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

[14] Nicola, R. D., 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* (2005), pp. 428–435.

[15] Omicini, A. and E. Denti, *From tuple spaces to tuple centres*, Science of Computer Programming **41** (2001), pp. 277–294.

[16] Phillips, A., *The Stochastic Pi Machine (SPiM)* (2006), version 0.042 available online at http://www.doc.ic.ac.uk/~anp/spim/.
URL http://www.doc.ic.ac.uk/~anp/spim/

[17] Phillips, A. and L. Cardelli, *A correct abstract machine for the stochastic pi-calculus*, Transactions on Computational Systems Biology (2005), to appear.

[18] Priami, C., *Stochastic pi-calculus*, The Computer Journal **38** (1995), pp. 578–589.